

Conditional Random Fields for Noisy Text Normalisation

by

Dirko Coetsee

*Thesis presented in partial fulfilment of the requirements for
the degree of Master of Science in Electrical and Electronic
Engineering in the Faculty of Engineering at Stellenbosch
University*



Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. J.A. du Preez

December 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2014/11/18

Copyright © 2014 Stellenbosch University
All rights reserved.

Abstract

Conditional Random Fields for Noisy Text Normalisation

Dirko Coetsee

*Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.*

Thesis: MScEng (E & E)

December 2014

The increasing popularity of microblogging services such as Twitter means that more and more unstructured data is available for analysis. The informal language usage in these media presents a problem for traditional text mining and natural language processing tools. We develop a pre-processor to normalise this noisy text so that useful information can be extracted with standard tools.

A system consisting of a tokeniser, out-of-vocabulary token identifier, correct candidate generator, and N-gram language model is proposed. We compare the performance of generative and discriminative probabilistic models for these different modules. The effect of normalising the training and testing data on the performance of a tweet sentiment classifier is investigated.

A linear-chain conditional random field, which is a discriminative model, is found to work better than its generative counterpart for the tokenisation module, achieving a 0.76% character error rate compared to 1.41% for the finite state automaton. For the candidate generation module, however, the generative weighted finite state transducer works better, getting the correct clean version of a word right 36% of the time on the first guess, while the discriminatively trained hidden alignment conditional random field only achieves 6%. The use of a normaliser as a pre-processing step does not significantly affect the performance of the sentiment classifier.

Uittreksel

Voorwaardelike Toevalsvelde vir die Normalisering van Teks met Ruis

(“Conditional Random Fields for Noisy Text Normalisation”)

Dirko Coetsee

*Departement Elektriese en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MScIng (E & E)

Desember 2014

Mikro-webjoernale soos Twitter word al hoe meer gewild, en die hoeveelheid ongestruktureerde data wat beskikbaar is vir analise groei daarom soos nooit tevore nie. Die informele taalgebruik in hierdie media maak dit egter moeilik om tradisionele tegnieke en bestaande dataverwerkingsgereedskap toe te pas. ’n Stelsel wat hierdie ruiserige teks normaliseer word ontwikkel sodat bestaande pakkette gebruik kan word om die teks verder te verwerk.

Die stelsel bestaan uit ’n module wat die teks in woordeenhede opdeel, ’n module wat woorde identifiseer wat gekorrigeer moet word, ’n module wat dan kandidaat korreksies voorstel, en ’n module wat ’n taalmodel toepas om die mees waarskynlike skoon teks te vind. Die verrigting van diskriminatie en generatiewe modelle vir ’n paar van hierdie modules word vergelyk en die invloed wat so ’n normaliseerder op die akkuraatheid van ’n sentiment-klassifiseerder het word ondersoek.

Ons bevind dat ’n lineêre-ketting voorwaardelike toevalsveld — ’n diskriminatie model — beter werk as sy generatiewe eweknie vir tekssegmentering. Die voorwaardelike toevalsveld-model behaal ’n karakterfoutkoers van 0.76%, terwyl die toestandsmasjien-model 1.41% behaal. Die toestantsmasjien-model

werk weer beter om kandidaat woorde te genereer as die verskuilde belyningsmodel wat ons geïmplementeer het. Die toestandsmasjien kry 36% van die tyd die regte weergawe van 'n woord met die eerste raaiskoot, terwyl die diskriminatie model dit slegs 6% van die tyd kan doen. Laastens het ons bevind dat die vooraf normalisering van Twitter boodskappe nie 'n beduidende effek op die akkuraatheid van 'n sentiment klassifiseerder het nie.

Acknowledgements

I would like to thank the following people and organisations:

- Prof. du Preez, for interesting discussions, for editing what I wrote, for all the good advice that I ignored, and for always being interested.
- All the friends I made in the medialab. Thank you for the countless hours of fussball.
- Gertjan, Herman, and McElory for all the organisation that goes into the lab.
- MIH and the medialab for financial support; the last two years in the lab has really been a fantastic journey.
- All the friends that found themselves being used as sounding boards, and all the others who made the last two years the most fun since the two years before that.
- My parents, siblings, and grandparents for the kuiers, support, and love, and especially my mother for editing parts of the report.
- My heavenly parent for bringing all of the above across my path, and for always being close by.

Contents

Declaration	i
Abstract	ii
Uittreksel	iii
Acknowledgements	v
Contents	vi
List of Figures	ix
List of Tables	xii
Nomenclature	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Background	2
1.3 Objectives	3
1.4 Contributions	4
1.5 Overview of the thesis	5
2 Probabilistic graphical models	13
2.1 Introduction	13
2.2 Notation	15
2.3 Representation	16
2.4 Inference	20
2.5 Learning	27
2.6 Conclusion	31

3	Weighted finite state machines	32
3.1	Weighted finite state acceptors	32
3.2	Weighted finite state transducers	33
3.3	Notation	35
3.4	Connection with linear chain models	35
3.5	Conclusion	36
4	Conditional random fields	37
4.1	Introduction	37
4.2	Learning	42
4.3	Logistic regression	46
4.4	Linear-chain CRFs	48
4.5	Hidden CRFs	51
4.6	Hidden alignment CRFs	54
4.7	Conclusion	61
5	Noisy text normalisation literature	62
5.1	Three metaphors	62
5.2	Spelling correction	63
5.3	Automatic speech recognition	68
5.4	Statistical machine translation	68
5.5	Hybrid FST system	69
5.6	CRF normalisation	70
5.7	Unsupervised normalisation	70
5.8	Evaluation	71
5.9	Conclusion	73
6	Text normalisation system	76
6.1	Architecture	76
6.2	Tokenisation	78
6.3	Out of Vocabulary Words	82
6.4	Candidate enumeration	83
6.5	Language model	94
6.6	System integration	95
6.7	Conclusion	96
7	Results	98

<i>CONTENTS</i>	viii
7.1 Module performance	98
7.2 Datasets	108
7.3 System performance	109
7.4 Effect on sentiment task	111
7.5 Conclusion	114
8 Conclusion	115
8.1 Summary	115
8.2 Conclusions	116
8.3 Recommendations	118
8.4 Future Work	119
Appendices	122
A Results	123
A.1 Tokeniser	123
A.2 Generator	126
A.3 Sentiment experiment	138
List of References	139

List of Figures

1.1	PGM example: $p(A, B, C, D, E) = \frac{1}{Z}\psi_1(A, B)\psi_2(B, C)\psi_3(C, D, E)$.	6
1.2	WFST “tonight” example.	6
1.3	CRF example: $p(F, G A, B, C, D)$.	7
1.4	HACRF graph.	8
2.1	Example MRF: $p(A, B, C, D, E) = \frac{1}{Z}\Psi_1(A, B)\Psi_2(B, E)\Psi_3(C, D, E)$.	17
2.2	A graph with observed (shaded) nodes.	19
2.3	Example of an MRF.	20
2.4	Elimination algorithm example.	21
2.5	Example of a clique tree.	22
2.6	Example clique tree showing the clique potentials Ψ and separator potentials Φ .	23
2.7	Belief update algorithm messages.	23
2.8	Message passing algorithm message.	24
2.9	Supervised learning MRF.	30
3.1	A Markov chain.	32
3.2	An example of a WFSA that represents a few URLs.	33
3.3	WFST “tonight” example.	34
3.4	WFST notation example.	35
4.1	A model where we always observe A , B , and C . We are interested only in querying D .	39
4.2	The same model as Figure 4.1 where only dependencies that are necessary if D is queried are modelled.	39
4.3	Graphical structure of the logistic regression CRF.	47
4.4	Graphical representation of linear chain CRF.	49

4.5	A linear chain CRF where every output variable y_t is only dependent on the corresponding \mathbf{x}_t	49
4.6	A junction tree representation of the linear chain CRF model.	50
4.7	A hidden state CRF where the dependencies between the hidden variables \mathbf{z} take the form of a linear chain.	52
4.8	An HCRF where every hidden variable z_t only depends on the corresponding input variable \mathbf{x}_t	53
4.9	A junction tree representation of the HCRF model.	53
4.10	HACRF graph.	55
4.11	HACRF state machine.	56
4.12	HACRF lattice example.	57
5.1	WFST representing Levenshtein distance.	64
5.2	WFST trigram language model example.	67
6.1	A high level graphical model of the text normalisation system.	77
6.2	<i>FSA1</i> . A fully connected WFSA to tokenise tweets.	79
6.3	<i>FSA2</i> . A WFSA tokeniser with memory.	80
6.4	<i>WFST1</i> . A basic edit distance transducer.	87
6.5	<i>WFST2 and WFST3</i> . WFSTs based on Levenshtein transducer.	87
6.6	<i>WFST4</i> . WFST with states for insertions, deletions, substitutions, and matches.	88
6.7	<i>WFST5</i> . WFST with states for insertions, deletions, and substitutions.	89
6.8	<i>WFST6</i> . WFST with a state for every character.	89
6.9	HACRF1 performance per training iteration for a small dataset.	94
6.10	HACRF1 performance per training iteration for a large dataset.	95
6.11	System output message structure.	97
7.1	WFST candidate generation F1-scores.	100
7.2	HCRF state machine.	112
A.1	Plots of the error rates on the training and validation sets to find the regularisation rate for FSA1 and FSA2 for the tokenisation task.	124
A.2	Plots of the error rates on the training and validation sets to find the regularisation rate for CRF1, CRF2 and CRF3 for the tokenisation task.	124

A.3	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP1.	127
A.4	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP1.	127
A.5	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP2.	128
A.6	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP2.	129
A.7	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP3.	130
A.8	Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP3.	130
A.9	Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP1. . . .	135
A.10	Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP2. . . .	136
A.11	Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP3. . . .	137

List of Tables

1.1	FSA and CRF tokeniser character error rates.	10
1.2	HACRF and WFST classification and candidate generation performance.	10
1.3	System WER on TWEETSUNALIGNED and TWEETSALIGNED. . .	11
4.1	Values for the input vectors for the HACRF example.	59
4.2	Example of a matching HACRF forward pass	61
4.3	Example of a mis-matching HACRF forward pass	61
5.1	N-best performance in the literature.	74
5.2	Normalisation system performance in the literature.	74
6.1	The features used in the different CRF tokenisers.	81
6.2	Examples of random matching word pairs.	84
6.3	The sizes of the three word pair datasets.	85
6.4	Parameter values learned by the WFST models for <i>deletions</i> of different symbols on MISSP3.	90
6.5	The different features that are implemented for the four different HACRF models.	92
7.1	FSA tokeniser character error rates.	99
7.2	Unregularised WFST candidate generator F1-scores.	101
7.3	N-best accuracy of the WFSTs.	101
7.4	HACRF candidate generator F1-scores.	102
7.5	HACRF candidate generator N-best scores.	103
7.6	Examples of generated candidates.	104
7.7	HACRF N-best scores with alternative distance measure.	105
7.8	HACRF N-best scores with revised training data.	106

7.9	The F1-scores of the best scoring WFST and HACRF for each dataset.	106
7.10	The N-best accuracy of the different models on the different test sets.	107
7.11	Language model perplexity scores.	107
7.12	System performance on TWEETSUNALIGNED.	109
7.13	System performance on TWEETSALIGNED.	110
7.14	Sentiment classification results.	113
A.1	Error rates on the training and validation sets to find the regularisation rate for FSA1 and FSA2 for the tokenisation task.	123
A.2	Error rates for different regularisation values for CRF1, CRF2, and CRF3 on the tokenisation task	125
A.3	F1-scores of the different WFSTs on the training and validation sets for MISSP1.	126
A.4	F1-scores of the different WFSTs on the training and validation sets for MISSP2.	128
A.5	F1-scores of the different WFSTs on the training and validation sets for MISSP3.	129
A.6	Parameter values learned by the WFST models for <i>deletions</i> of different symbols on MISSP3.	131
A.7	Parameter values learned by the WFST models for <i>insertions</i> of different symbols on MISSP3.	132
A.8	Parameter values learned by the WFST models for <i>matches</i> of different symbols on MISSP3.	133
A.9	F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP1.	134
A.10	F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP2.	135
A.11	F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP3.	136
A.12	The McNemar significant test results for the sentiment classification task.	138

Nomenclature

Abbreviations

PGM	Probabilistic graphical model
PDF	Probability density function
MRF	Markov random field
CRF	Conditional random field
i.i.d.	Independent and identically distributed
MPE	Most probable explanation
LM-BFGS	Limited memory Broyden-Fletcher-Goldfarb-Shanno (optimisation)
FSM	Finite state machine
WFST	Weighted finite state transducer
FST	Finite state transducer
WFSA	Weighted finite state acceptor
URL	Uniform resource locator
NLP	Natural language processing
POS	Part of speech
HCRF	Hidden conditional random field
HACRF	Hidden alignment conditional random field
SMS	Short message service
ASR	Automatic speech recognition
OCR	Optical character recognition
OOV	Out of vocabulary
SMT	Statistical machine translation
BLEU	Bilingual evaluation understudy

IEEE	Institute of Electrical and Electronics Engineers
WER	Word error rate
LM	Language model

Variables

σ	Standard deviation of Gaussian density function
μ	Mean of Gaussian density function
D	Dimensionality of data or model
\mathbf{x}	Input vector
\mathbf{y}	Output vector
\mathbf{z}	Hidden vector
\mathcal{C}	Set of cliques
Z	Partition function - Normalising constant of PDF
Ψ	Clique potential function
Φ	Separator potential function
$\boldsymbol{\lambda}$	Vector of parameters
λ_k	k th element in $\boldsymbol{\lambda}$
\mathbf{f}	Vector of indicator functions
f_k	k th element in \mathbf{f}
$\mu_{A,B}$	Message from A to B
D	Set of training examples
H	Hypothesis or model
N	Number of training examples
T	Length of the current training example
q	A state in a state machine

Chapter 1

Introduction

The internet has opened up amazing linguistic research possibilities [16]. Computer-mediated communications (CMCs) such as weblogs, chat, email, SMS, and recently microblogs such as Twitter (www.twitter.com) are a rich source of data. Language change has happened in a short period in this medium. Crystal describes the surface language changes produced by CMC as less important and pronounced than the other characteristics such as the hypertextuality, dynamisms, and simultaneous nature of CMC in a seminal paper on internet linguistics [16].

The surface changes have nevertheless attracted interest [61, 62, 18, 11]. The emerging field of *noisy text analytics* investigates these surface changes.

1.1 Motivation

Denoising, or normalisation, is the recovery of the standard surface form of a text given a noisy version of the text. For example, a denoising system receives the following (actual) tweet (Twitter messages are usually called “tweets”):

i shudnt of eaten that sushi so fats, i feel sick.

A reasonable normalised version is:

I shouldn't have eaten that sushi so fast, I feel sick.

Changing “shudnt” to “shouldn't” is called lexical normalisation, or spelling correction. We will concentrate on this type of normalisation. It is of course possible to expand “shouldn't” further to “should not”. It is an open question

whether this is better. Capitalisation is a separate task that we do not consider. Changing “of” to “have” is grammar correction, which is related to the problem of normalising “fats”. Both these examples start with words that are already correctly spelled dictionary words. Correcting these in-vocabulary tokens is a difficult task that we also do not concentrate on.

Noise in text presents problems for humans and computers. A denoising system is found to help human subjects understand tweets [12].

We concentrate on the benefits that such a denoising system can bring to the automatic processing of noisy text. Such a system forms part of a larger text processing system as a preprocessing module.

One possible application is the normalisation of SMSs before they are given to a text-to-speech module for visually impaired persons [11]. There is also interest in mining microtext such as tweets for sentiment information that would be valuable to market researchers [50].

Information retrieval can also benefit from a normalisations system. In a study on the effect of noise on information retrieval and text classification, Agarwal et al. find that although text classifiers are surprisingly robust against noise, the performance degradation is sensitive to the number of features in the classifier [1]. This means that small and fast classifiers should benefit the most from such a normalisation system. They also find that classifiers that are trained and tested on noisy text fare worse than those that are trained on clean text and then tested on noisy text. This suggests that normalisation should also be useful during the training of the other components of text processing systems.

We concentrate on the normalisation of tweets because of recent interest in microblogging and the availability of data.

Spelling correction and text normalisation are traditionally tackled with generative models. The use of discriminative models has not been explored fully and therefore the current study tries to contribute in this direction.

1.2 Background

In a survey on the types of noise in text and ways to handle them, Subramaniam et al. identify noisy text by the high incidence of misspellings and out-of-vocabulary (OOV) tokens [59]. According to the survey, noise occurs

in informal text such as SMSs and tweets, transcripts produced by automatic speech recognition (ASR) or optical character recognition (OCR) systems, or in the output of statistical machine translation (SMT) systems. The text that we are interested in, namely informal microblogging text, differ from ASR, OCR, and SMT noise in that most of the noise is intentional [28].

Gouws et al. investigate different types of these intentional lexical variants in tweets. The following transformations (with examples) account for more than 90% of noise in their tweet dataset [25]:

1. Truncation of words to a single letter (“and” becomes “n”).
2. Truncation to only the suffix (“of” becomes “f”).
3. Dropping of vowels (“tomorrow” becomes “tmrrw”).
4. Truncation to prefix (“tomorrow” becomes “tom”).
5. “You” changing to “u”.
6. Dropping of the last character (“making” becomes “makin”).
7. Repetition of letters (“so” becomes “sooooooooo”).
8. Contractions (“you will” becomes “you’ll”).
9. “th” changing to “d” (“the” becomes “de”).

It is difficult to decide what the “standard” surface form of an utterance or text is, and therefore we take the position that standard English is defined by the corpora that is used to train natural language processing tools. These corpora include the Brown corpus and Penn tree bank [45, 22].

1.3 Objectives

We have the following broad objectives with this study:

- To study probabilistic graphical models with the emphasis on conditional random fields (CRFs) with the goal of applying these models to text normalisation.
- To implement a discriminative probabilistic text normalisation system.

- To compare the discriminative model with generative probabilistic models for text normalisation.
- To investigate the effect of a text normalisation preprocessing module on the performance of another text-analysis task.

1.4 Contributions

The contributions of the thesis towards the above-mentioned goals are:

- A linear chain CRF is trained as a tokeniser for tweets. To our knowledge this is the first application of the model to this problem. A training dataset of 1488 tweets is annotated for this task. The CRF achieves a test-set error of 0.76%, which is half the error rate of the generative finite state acceptor we tested. See Sections 6.2 and 7.1.1.
- For the hidden alignment CRF (HACRF), dynamic programming equations for inference is derived from the general graphical model equations in Section 4.6.2. See Section 6.4.3 where the implementation of a HACRF is discussed.
- The direct optimisation of the HACRF model is shown to be effective. The model has previously only been trained with the EM algorithm. See Section 6.4.3.3.
- The HACRF model is applied as an edit distance for spelling correction. The model has previously been applied to database normalisation but not as a distance measure for spelling correction. See Section 7.1.2.2.
- Different weighted finite state transducer models are implemented and their performance as edit distances is compared to that of the HACRF. See Section 6.4.2 for the models and Section 7.1.2.3 for the comparison.
- The HACRF achieves a first-best guess rate of 6.33% and gets test words right 39.24% of the time with 20 tries. This is much worse than the finite state transducer baseline we used which achieved 36.46% and 71.65% for one and 20 tries respectively. See Section 7.1.2.3 for the final results.
- To evaluate the different systems, a parallel dataset of 2482 tweets is produced and corrected by hand. See Section 7.2.

- The system achieves a word error rate of 0.0770, compared to the baseline dictionary-based normaliser that gets a word error rate of 0.0660. See Section 7.3.
- The performance of a sentiment classification hidden CRF (HCRF) is evaluated on noisy and cleaned text, and the HCRF as we used it is found to have no advantage over a logistic regression classifier. The normalisation of the text before training and testing does not have a significant effect on classification performance. See Section 7.4.

1.5 Overview of the thesis

The thesis is organised into four theoretical chapters followed by three chapters that describe the contributions and results of the study.

Chapter 2 gives an introduction to probabilistic modelling with graphical models. *Probabilistic graphical models* form the framework in which the other models are cast. We are interested in probabilistic models because they present a principled way to take uncertainty into account. Probabilistic graphical models represent the factorisation of probability distributions graphically. For example, the distribution that factorises as

$$p(A, B, C, D, E) = \frac{1}{Z} \psi_1(A, B) \psi_2(B, C) \psi_3(C, D, E), \quad (1.5.1)$$

can be represented by the graph in Figure 1.1. A to E are random variables, ψ_1 to ψ_3 are the factors in the distribution, and Z is the normalising constant necessary to make the right-hand side sum to one. There are connections between the nodes that represent random variables that appear together in a factor.

To find marginals, dynamic programming algorithms can be defined. So

$$p(A) = \sum_{B, C, D, E} \frac{1}{Z} \psi_1(A, B) \psi_2(B, C) \psi_3(C, D, E), \quad (1.5.2)$$

becomes

$$p(A) = \frac{1}{Z} \sum_B \psi_1(A, B) \sum_C \psi_2(B, C) \sum_{D, E} \psi_3(C, D, E), \quad (1.5.3)$$

which can be calculated more efficiently than the summation which does not make use of the factorisation. The graph representation of the factorisation is

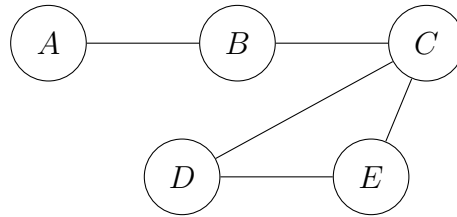


Figure 1.1: A probabilistic graphical model that represents the probability density function that factorises as $p(A, B, C, D, E) = \frac{1}{2}\psi_1(A, B)\psi_2(B, C)\psi_3(C, D, E)$.



Figure 1.2: An example of a WFST that transforms “tonight” into “tnight”, “tonight”, “tnite”, “tonite”, “tnyt”, “tonyt”, “2night”, “2nite”, or “2nyt” with different probabilities.

useful for finding and applying similarly efficient algorithms. When designing a probability model, it is also useful to visualise the probability distribution because of a fundamental correspondence between the factorisation of a distribution and its conditional independence properties.

In this framework, *Weighted finite state transducers (WFSTs)*, which we introduce in Chapter 3, are represented with chain graphs. Many of the state-of-the-art text-normalisation systems are implemented with weighted finite state transducers. WFSTs graphically represent the allowable transitions and allowable outputs of state machines that output two symbols on each transition from one state to another. An example of a WFST that “transduces” alternative spellings of “tonight” into the correct spelling is shown in Figure 1.2.

Each node represents a state. At discrete time steps, the state machine can move from one state to another if there is an arc between the nodes. The probability of following a certain edge is shown after the “/” symbol above the edge. With every transition, two output symbols are also produced. Above each edge is the source symbols, separated from the target symbols by a “:”. The special symbol ϵ denotes an edge which does not emit a character.

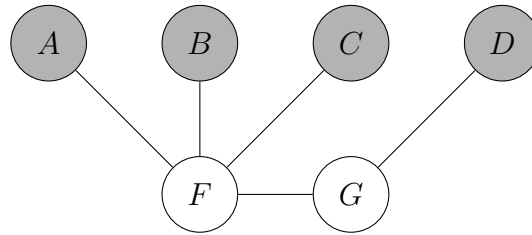


Figure 1.3: An example of a CRF. The shaded nodes A , B , C , and D represent random variables that are always observed. This CRF represents the conditional distribution that factorises as $p(F, G|A, B, C, D) = \frac{1}{Z} \psi_1(F|A) \psi_2(F|B) \psi_3(F|C) \psi_4(F, G) \psi_5(G|D)$.

The main focus of this investigation, namely *conditional random fields (CRFs)*, is introduced in Chapter 4. These models can be represented by probabilistic graphical models. One of the most well-known special case, the linear-chain CRF, is also a type of weighted finite state transducer. CRFs can be represented by conditional graphical models, which means that instead of having a graph that represents a joint distribution $p(X)$, we directly model the distribution $p(Y|X)$. Here, Y is the set of unknown random variables that we would like our model to predict, while X is the set of variables whose values we know because we can observe them directly. In Figure 1.3, an example of a CRF is shown.

Before a model is trained, the values that the factors ψ_i give for different configurations of the random variables are unknown. Training adjusts parameters that influence these values so that the model gives higher probabilities to the training examples.

There are a few specific CRF models that we are interested in. *Logistic regression* (See Section 4.3) is the simplest and consists of one unobserved node and any number of observed nodes. It is useful in classification problems where the unobserved node represents a random variable that can take on one of a number of classes.

Linear-chain CRFs (See Section 4.4) have many unknown variables that form a chain. Attached to each unobserved node is any number of observed nodes. These models are used to label a sequence of input variables. Each input variable is labelled as belonging to one of a discrete number of classes, and the probability of the label of the previous and next element in the sequence influences the probability of the label of the current element.

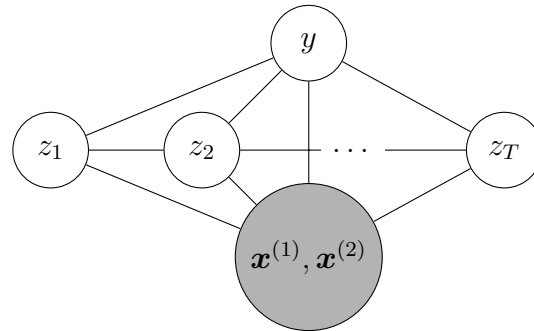


Figure 1.4: The hidden alignment CRF. The hidden variables $z_1 \dots z_T$ represent edit operations on the two input sequences $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$.

Hidden CRFs (HCRFs) (See Section 4.5) are used to classify a whole sequence as belonging to one of a number of classes. They add some latent structure so that the order of observed variables makes a difference to the final classification.

Hidden alignment CRFs (HACRFs) (See Section 4.6) classify two input sequences into one of a number of classes. We use it to classify input sequence-pairs as either matching or non-matching. So we would want it to classify the pair (“wrk”, “work”) as a match, and the pair (“wrk”, “cheese”) as a mismatch. A latent sequence of edit operations is used to align the two sequences, and this sequence of edit operations describes the way that the probability density function factorises as is illustrated in Figure 1.4. For the input strings “wrct” and “work”, one possible sequence of edit operations to change the first string into the second string would be: match “w”s, insert “o”, match “r”s, substitute “c” with “k”, delete “t”. y is the output label and for our purposes can be either “match” or “mismatch”. The probability of a match or mismatch given all possible such alignments is then calculated.

In Chapter 5, the literature on noisy text normalisation is introduced in the light of the models that are described in the previous chapters. Many systems can be described in terms of the *noisy channel model*. In the context of text normalisation, this model supposes that there is some clean intended stream of text \mathbf{y} that is sent over an imperfect channel where the message is corrupted. The noisy output \mathbf{x} is then what we can observe. Bayes’ rule, along with a model of the intended text $p(\mathbf{y})$ and a *channel model* $P(\mathbf{x}|\mathbf{y})$ is used to try

and reconstruct the “original” message given the observed message. So,

$$\arg \max_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}) = \arg \max_{\mathbf{y}} p(\mathbf{x}|\mathbf{y})p(\mathbf{y}). \quad (1.5.4)$$

One of the noisy channel model’s advantages is that it breaks the normalisation problem into a token-level module $p(\mathbf{x}|\mathbf{y})$ and a message level module $p(\mathbf{y})$. Many of the state-of-the-art normalising systems use WFSTs to implement a noisy channel model that is trained on data.

With consideration to the literature, a design for a text normalisation system that incorporates discriminative modules is presented in Chapter 6. The modules are arranged in a pipeline, so the output of one module is the input of the next. This pipeline can also be described as a graphical model. Graphical models require that the messages between the modules must be of the form of (possibly unnormalised) distributions. The modules that are implemented are listed below:

1. A tokeniser is necessary to break the input text into tokens that correspond to punctuation, words, and so forth. A weighted finite state acceptor and linear-chain CRF are trained for this task.
2. A module is necessary to classify each input token as either needing correction or as already correct. We use logistic regression as the *OOV classifier*.
3. For the tokens that must be corrected, candidate corrections are generated. We use a distance measure between strings to find words in the lexicon that are “near” the incorrect word. Two such distance measures are implemented:
 - a) The probability that a WFST gives for one of the strings to be transduced to the other string.
 - b) The probability that the two strings form a matching pair according to the HACRF model.
4. An N-gram language model is used to model word context so that ambiguous tokens can be corrected.

The module and end-to-end results of the system are presented in Chapter 7. The different modules are first tested individually. It is found that the

	Total labels	FSA1	FSA2	CRF1	CRF2
Test errors	8270	117	147	72	63
Test error %		1.415	1.778	0.8706	0.7618

Table 1.1: The character error rates of two FSA tokenisers and CRF tokenisers with different feature sets. FSA1 is a simpler finite state machine than FSA2. CRF1 is comparable to FSA2 in that only the current character is used as a feature. CRF2 uses different features of the current character. The CRF with additional features has the lowest error rate.

	Classification (F1-score)		Generation			
	Train	Test	1-best	3-best	20-best	100-best
WFST	0.440	0.426	0.3646	0.5089	0.7165	0.8051
HACRF	0.814	0.713	0.0633	0.1519	0.3924	0.6709

Table 1.2: The classification and candidate generation performance of the two models used to generate candidate corrections. The HACRF is a better classifier but the WFST produce more usable candidate lists.

CRF tokeniser works better than a comparable WFSAs tokeniser. The CRF, however, can use features that the WFSAs cannot use and the performance is then even better. Table 1.1 summarises the results of the tokenisation experiment.

The simple logistic regression OOV-classifier we used achieves an accuracy of 73.97% on the test data.

The candidate generation module is evaluated. For the task of classifying word-pairs as matches or mismatches, HACRFs do much better than the WFSTs. For scoring and ranking token-candidate pairs, however, the WFSTs give better results as can be seen in Table 1.2. The WFST produces the correct version of a given noisy token 36% of the time on the first guess, while the HACRF only manages 6%.

The different modules are arranged into a pipeline and the end-to-end performance is evaluated. Two datasets are used to evaluate the system’s performance. TWEETSUNALIGNED is a parallel collection of about 2500 tweets, while TWEETSALIGNED is a word-aligned parallel collection of about 550 tweets.

It is found that the OOV-classification module is critical. Even the best candidate generation models we tried, namely the WFSTs, introduce more errors than they correct. With a better OOV classifier, WER would go down

	TWEETSUNALIGNED	TWEETSALIGNED
	WER	WER
Original	0.0760	0.1121
Dictionary	N/A	0.0344
Dictionary+LM	0.0660	0.0335
WFST	N/A	0.0894
WFST+LM	0.1022	0.0601
WFST+LM+OOV	0.0770	N/A

Table 1.3: The word error rates of the system for two different datasets. TWEETSUNALIGNED is evaluated without the oracle OOV-classifier while TWEETSALIGNED is evaluated with an oracle.

as is shown on the TWEETSALIGNED test data in Table 1.3. The addition of a language model is shown to have a positive effect on performance.

The dictionary-based normaliser we use as a baseline improves the word error rate (WER) from 0.054 to 0.0528. The WFST models introduce more errors than they correct. This situation is improved somewhat by the addition of the OOV classification module. For the TWEETSALIGNED dataset, we evaluate what would happen with a perfect OOV classifier. With the oracle OOV classifier, the dictionary normaliser still improves the WER the most.

Lastly, the normalisation system is used as a preprocessing step for a sentiment classifier, but it has negligible effect (See Section 7.4).

Conclusions and recommendations are presented in Chapter 8. Some of the important conclusions are:

- having a pipeline architecture with individually trainable models is a practical way to modularise a normalisation system,
- it is difficult to identify the tokens that should be corrected in tweets, and the accuracy with which it can be done has a large influence on the system's performance,
- tokenisers can be trained from data, and the CRFs we tried worked better than the more traditional WFSAs,
- it is feasible to train the HACRF model by directly optimising its likelihood,

- the HACRF models are better than WFSTs at classifying word-pairs as matching or non-matching, but without engineering the training data the HACRFs are much worse at producing N-best lists,
- The use of an HCRF does not provide an advantage over logistic regression for the sentiment classification task when word-identities are used as features, and
- the lexical normalisation of training and testing data before sentiment classification is done does not improve the accuracy of an HCRF or logistic regression classifier.

In the future we can look at the following:

- The system's performance will be improved the most by working on the OOV classifier, by using a much larger lexicon, or by using a better open-vocabulary language model.
- A hybrid system that uses a dictionary for common misspellings and a distance measure for the rest could capture the advantages of both approaches.
- The HACRF model's generation performance can possibly be improved by experimenting with different training sets or by holding the matching or non-matching potentials fast.
- The performance of the HCRF model can possibly be improved by investigating different parameter initialisation strategies.
- It would be interesting to compare the difference between the direct optimisation of the HACRF model and training it with the EM algorithm.

Chapter 2

Probabilistic graphical models

Probabilistic graphical models (PGMs) provide a way of compactly representing probability distributions [35]. Many of the modelling assumptions in probabilistic models are in the form of conditional independence assumptions. These independence assumptions allow the distribution to be concisely and intuitively represented with a graph. Joint probabilities can be represented with directed, undirected, or a mixture of directed and undirected graphs. These representations can express different but largely overlapping sets of models. We consider only undirected graphs.

In this chapter we introduce probabilistic modeling with graphical models. We look at the representation of PGMs, how dynamic programming algorithms lead to efficient querying of the models, and how the model parameters can be learned from data. In later chapters the general theory given here is applied to specific models.

2.1 Introduction

Probabilistic modeling has become popular in machine learning because probability theory is a useful formalism when dealing with uncertainty. It quantifies uncertainty and defines rules with which one can reason under uncertainty.

A probabilistic model is therefore a way of encoding useful information about some object or system along with the uncertainties associated with the information. This “database” can later be queried when a specific piece of information is required [35]. Typical queries one could make are for marginal probabilities of some unknown variable given evidence, or for the probability of

the model given data. Automatic classification is a typical machine learning problem that can be tackled with a probabilistic model. To classify a data point with a probabilistic model, one could query the model as to the marginal probability over the classes given that data point, and then use decision theory to select the most useful class [5]. Querying the model for the configuration of variables with the highest probability is another way of doing classification with probability models.

We will start off by only using discrete distributions as examples, as they are more applicable to the content of the rest of the thesis. Almost all the theory, however, is also applicable to continuous distributions if the relevant changes are made such as replacing summations with integrals.

A question that arises when using probabilistic models on a computer is what type of *data structure* to use to represent the model in memory [35]. When dealing with continuous models such as a Gaussian probability density function (PDF) or some other simple density function one could store the parameter values μ and σ in memory. When a query for a marginal or conditional probability is received by the computer, the stored parameter values along with some pre-programmed analytical formulas are used to find the required marginal.

For discrete models, one could represent the model by having the joint density of the random variables that one is interested in in memory. A table lists the probability of every combination of values that the random variables in the model can take. For a PDF with D discrete variables with a cardinality of 2 each, one would require a table of size 2^D . The marginal of a certain variable can then be found by doing a summation over all the other variables' entries in the table to find a new table representing the marginal distribution.

The conditional distribution given some piece of evidence is found as follows: The entries in the probability table where the evidence-variables takes on the evidence-values are written to a new table that represents the conditional distribution, and is then normalised.

For large dimension D the model can no longer fit into memory. Probabilistic graphical models solve this problem by using conditional independence information to store the distribution compactly.

PGMs are a useful marriage between probability theory and graph theory. They provide a data structure for storing joint density functions, and also some efficient algorithms for doing computations such as finding marginals,

conditioning on evidence, or finding the configuration of variables with the highest probability. Although PGMs provide a way to solve the problem of representing a joint distribution efficiently in memory, they can be motivated from a few other perspectives. For example, because they encode the conditional independence assumptions of a distribution, they also give an intuitive way to represent and design probabilistic models.

Many texts divide the introduction of PGMs into three topics: representation, inference, and learning [5, 35]. We follow the same structure. In the rest of the chapter we provide an informal summary of some of the theory that is covered in more detail in these texts.

2.2 Notation

Before continuing let us define the notation that we will use in the rest of the report.

We represent column vectors with lower case bold letters, for example \mathbf{x} , and row vectors as the transpose of such column vectors. The transpose is written with a superscript T , so that $\mathbf{x} = [x_1, x_2, \dots, x_D]^{\text{T}}$ for a D dimensional column vector.

Matrices are upper case bold letter, for example $\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$.

Random variables and random vectors are denoted by upper case letters, for example A , B , and X . Some constants, like the dimension D of a vector, the cardinality W of a hidden variable, or the number N of training examples are also upper case letters. Sets are represented with upper case roman letters, so $A = \{2, 5, 7\}$.

When writing probabilities, we use the common shorthand $p(X = \mathbf{x}) = p(\mathbf{x})$ for when the random vector X takes on the value \mathbf{x} .

The expected value of a function $f(x)$ over a distribution $p(x)$ is denoted $\mathbb{E}_{p(x)}[f(x)]$.

If $\mathbf{X} = \{X_1, X_2, \dots, X_D\}$ is a set of random variables, and C is a set of natural numbers $\{a, b, \dots, c\}$, then we use X_C as the set of random variables that is indexed by the elements in C . $p(X_{\{1,4,6,7\}})$ is therefore a short way of writing $p(X_1, X_4, X_6, X_7)$.

If $\Lambda = \{\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \dots, \boldsymbol{\lambda}_K\}$ is a set of vectors, then we will refer to the j th element in the i th vector as $\lambda_{i,j}$.

We denote the difference between sets with “\”. So $A \setminus B$ is the set of all elements in A that is not in B .

2.3 Representation

2.3.1 Undirected graphs

Definition 2.1. An *undirected graph* \mathcal{G} is a pair $\mathcal{G} = (V, E)$, consisting of *vertices* $V = \{V_1, \dots, V_D\}$, also called *nodes*, and *edges* E . Edges are connections between pairs of nodes $E = \{V_a - V_b, \dots, V_c - V_d\}$ [35]. A *clique* in a graph is a set of nodes that are all connected to each other. A *maximal clique* is a set of nodes that forms a clique so that no other node in the graph can be added to this clique without the set of nodes losing its clique status. The set of all maximal cliques in a graph we denote as \mathcal{C} to distinguish it from its elements.

We denote the set of neighbours of a node A as $N_G(A)$.

2.3.2 Markov Random Fields

Definition 2.2. An undirected PGM, also called a Markov network or *Markov random field (MRF)*, is a probability distribution $p(\mathbf{X})$ that is defined on a graph \mathcal{G} so that each random variable $X_j \in \mathbf{X}$ is assigned to a node $(V_1, X_1) \dots (V_D, X_D)$. Note that sometimes we will refer to the nodes by the names of the variables to which they are tied. Furthermore, the probability distribution factorises according to the maximal cliques $\mathcal{C} = \{C_1, \dots, C_{|\mathcal{C}|}\}$. So,

$$p(\mathbf{X}) = \frac{1}{Z} \prod_{C_i \in \mathcal{C}} \Psi_i(\mathbf{X}_{C_i}). \quad (2.3.1)$$

Here Z is the normalisation constant needed to make the right hand side of the equation sum to one. It is also called the *partition function*. The factors Ψ_i are called the clique potentials, or *potential functions*, and can take on any non-negative values.

For example, the distribution that factorises as

$$p(A, B, C, D, E) = \frac{1}{Z} \Psi_1(A, B) \Psi_2(B, E) \Psi_3(C, D, E) \quad (2.3.2)$$

can be represented with the graph in Figure 2.1. Here each random variable can take on either 0 or 1, and each potential is represented by a probability

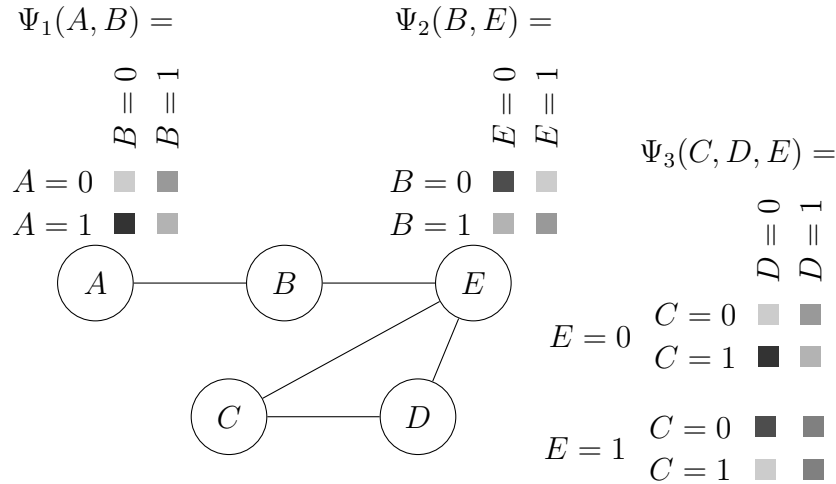


Figure 2.1: An MRF that represents the factorisation $p(A, B, C, D, E) = \frac{1}{Z} \Psi_1(A, B) \Psi_2(B, E) \Psi_3(C, D, E)$. The tables represent the potential functions Ψ_i . Since each variable can only take on 0 or 1, the value a potential function evaluates to can be found by a lookup in the table. White in the lookup table represents 0 and black $+\infty$.

table, giving two 2-dimensional tables and a 3-dimensional table. To find the probability of a certain assignment for the random variables, say $p(1, 0, 0, 1, 1)$, we can write

$$p(1, 0, 0, 1, 1) = \frac{1}{Z} \Psi_1(1, 0) \Psi_2(0, 1) \Psi_3(0, 1, 1). \quad (2.3.3)$$

The three potentials can be read from the tables directly. The partition function is more difficult to calculate and we will return to the problem later.

2.3.3 Conditional independence and factorisation equivalence

A fundamental result for graphical models is that the factorisation of a probability distribution is tied to the distribution’s conditional independence properties and is proved in [27]. An important consequence of this result is that a graph that encodes a certain factorisation of a distribution also encodes its conditional independence properties (for strictly positive distributions), and conversely that a graph that represents a certain set of conditional independence properties also gives the factorisation of the distribution.

Let us first define conditional independence for probability distributions, and the concept of separation for undirected graphs before showing the connection.

Definition 2.3. A set of random variables A is said to be *conditionally independent* of another set of random variables B given a third set of random variables E , written as

$$A \perp B | E, \quad (2.3.4)$$

if gaining information about B does not change your information about A .

$$A \perp B | E \iff P(A, B | E) = P(A | E)P(B | E) \quad (2.3.5)$$

$$\Rightarrow P(A | B, E) = P(A | E) \quad (2.3.6)$$

Nodes in an MRF that represent the “given” or observed variables E are called *observed nodes*. In diagrams these nodes are usually grayed or darkened to show that they are observed.

A path in an MRF is a series of nodes that are pairwise connected by edges. An *active path* is a path containing no observed nodes.

Definition 2.4. A set of nodes in a graph is *separated* from another set of nodes if there are no active paths between any of the nodes in the one set and any of the nodes in the other set.

Separation can be visualised by imagining that the observed nodes are removed. If there is no way to move along edges from one set of nodes to the other set of nodes, then they are separated by the observed nodes.

Proposition 2.1. *Hammersley-Clifford:* If $A \perp B | E$, then in the MRF graph the observed nodes E separate the nodes representing A and B . For the proof see [27].

For example, in the MRF represented by the graph in Figure 2.2, B and A are independent of C and D given E , or $\{A, B\} \perp \{C, D\} | \{E\}$. The fact that E is observed “breaks” the graph into two disjointed graphs.

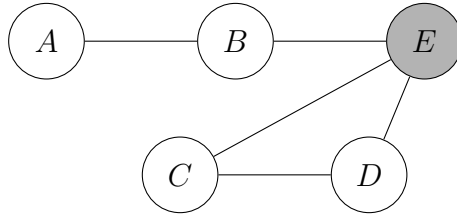


Figure 2.2: A graph with observed (shaded) nodes.

2.3.4 Log linear models

Up until now we have assumed that the potential functions are given as tables, where the value of the function for a certain input can be read off directly. Another useful way of parametrising the potential functions is to take the exponent of a linear function of the random variables A and the parameters, so that

$$\Psi(A) = \exp\{\boldsymbol{\lambda}^T \mathbf{f}(A)\}. \quad (2.3.7)$$

Here $\boldsymbol{\lambda}$ is a parameter vector of real numbers and $\mathbf{f}(\cdot)$ is a vector of functions. If the entire model is parameterised in this way it is called a *log linear model*. We restrict ourselves to the case where the functions are indicator functions. An indicator function f_k is defined by

$$f_k(A) = \begin{cases} 1 & A_B = E \\ 0 & \text{otherwise,} \end{cases} \quad (2.3.8)$$

for some assignment E of a subset B of the input variables. The model now becomes

$$p(\mathbf{X}) = \frac{1}{Z} \prod_{C_i \in \mathcal{C}} \exp\{\boldsymbol{\lambda}_i^T \mathbf{f}_i(\mathbf{X}_{C_i})\} \quad (2.3.9)$$

$$= \frac{1}{Z} \exp\left\{\sum_{C_i \in \mathcal{C}} \boldsymbol{\lambda}_i^T \mathbf{f}_i(\mathbf{X}_{C_i})\right\}, \quad (2.3.10)$$

where each clique C_i has its own vector of parameters $\boldsymbol{\lambda}_i$ and indicator functions \mathbf{f}_i .

Instead of writing the inside of the exponent as a vector dot product it is sometimes more convenient to write it as the sum of K scalar products

$$p(\mathbf{X}) = \frac{1}{Z} \exp\left\{\sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_{i,k} f_{i,k}(\mathbf{X}_{C_i})\right\} \quad (2.3.11)$$

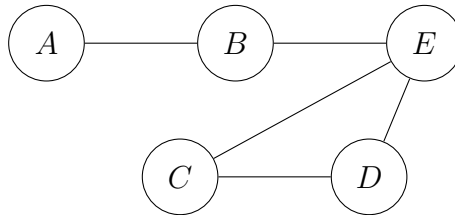


Figure 2.3: Example of an MRF.

where K is the number of indicator functions.

Log-linear models provide a finer grained parameterisation than graphical models with potential tables. Parameters can be shared arbitrarily and potentials can be specified with fewer parameters than cells, allowing an even more compact representation of probability distributions. In text applications, where the cardinality of variables are often large, this is useful [35, p. 125].

It has also been noted that parameter estimation is sensitive to the parameterisation that is used, and that with a log linear parameterisation the most probable estimates for the parameters are equal to their means [43]. When a discrete distribution is approximated with the Laplace approximation, the approximation is better with a log linear parameterisation because the parameters can take on any value [43].

2.4 Inference

The process of finding marginals, computing probabilities, or finding the maximal configuration of variables in the model given evidence is called *inference*. There exists efficient dynamic programming algorithms to do inference on graphical models. These algorithms often take the form of *message passing algorithms*.

2.4.1 Variable elimination

To motivate and get an intuition of how these algorithms work let us first look at an example of inference by variable elimination. Say we are interested in the marginal probability of E of our example MRF repeated in Figure 2.3. We have all the clique potentials but none of the marginals stored in the tables of

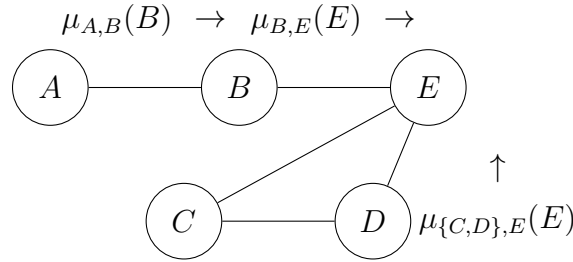


Figure 2.4: Example of the elimination algorithm yielding the messages $\mu_{A,B}(B)$, $\mu_{B,E}(E)$, and $\mu_{\{C,D\},E}(E)$.

potentials. We marginalise out all the variables we are not interested in, so

$$p(C) = \sum_{A,B,C,D} p(A, B, C, D, E) \quad (2.4.1)$$

$$= \sum_{A,B,C,D} \frac{1}{Z} \Psi_1(A, B) \Psi_2(B, E) \Psi_3(C, D, E). \quad (2.4.2)$$

Now we rearrange the order of the summations to something more convenient.

$$p(C) = \frac{1}{Z} \sum_{C,D} \Psi_3(C, D, E) \sum_B \Psi_2(B, E) \sum_A \Psi_1(A, B). \quad (2.4.3)$$

The problem has now broken up into three much smaller summations. First A is summed out of the potential $\Psi_1(A, B)$, leaving a one-dimensional table $\mu_{A,B}(B)$. That table is then multiplied with $\Psi_2(B, E)$ and B is summed out, leaving $\mu_{B,E}(E)$. From the variables on the other side of the graph we sum C and D out of $\Psi_3(C, D, E)$, leaving the one-dimensional potential table over E namely $\mu_{\{C,D\},E}(E)$. This table is then multiplied with the other table that is also only a function of E , namely $\mu_{B,E}(E)$, and normalised. What remains is the marginal probability of E . We thus have

$$p(E) = \frac{1}{Z} \mu_{\{C,D\},E}(E) \mu_{B,E}(E), \quad (2.4.4)$$

$$Z = \sum_E \mu_{\{C,D\},E}(E) \mu_{B,E}(E). \quad (2.4.5)$$

We therefore get both the probability of E and the normalisation constant Z from this process. This process can be visualised as the passing of messages between groups of nodes in the graph as shown in Figure 2.4. Although we have left out a number of details, such as how to decide the ordering of elimination, this algorithm carries the essence of a number of *sum product* algorithms.



Figure 2.5: Example of a clique tree.

2.4.2 Clique trees

One type of sum product algorithm is called the *clique tree*, or *junction tree* algorithm. We will start by defining a new data structure called a *cluster graph*. A cluster graph encodes the same information as an MRF but provides a more convenient data structure for inference.

A cluster graph is an undirected graph where each node represents a set of random variables. We are interested in cluster graphs that are trees, called *clique trees*. Message passing algorithms are possible for non-tree cluster graphs but then they are not exact.

MRFs can be converted to clique trees by creating a node for each set of nodes that forms a clique in the original graph.

Definition 2.5. A *junction tree* is a clique tree that has the running intersection property. Having this property constrains the tree so that all the nodes containing a certain random variable form a connected subtree.

MRFs can have more than one possible valid junction tree and it is an NP-hard problem to find the best one. We will assume, however, that a good enough junction tree can be found by inspection.

Our running example can be converted into the junction tree in Figure 2.5. It is also useful to include nodes representing the *separator sets* of the cliques. These are the sets of variables that two adjacent cliques share. The cliques and their separators each have a potential associated with it. The potentials associated with the separators are also the marginals of the variables in those clusters. The example with the potentials added is repeated in Figure 2.6. Here B and E are the separator variables, and Φ_B and Φ_E are their potentials.

2.4.3 Belief update algorithm

The junction tree message update algorithm sprouts from the following observation:

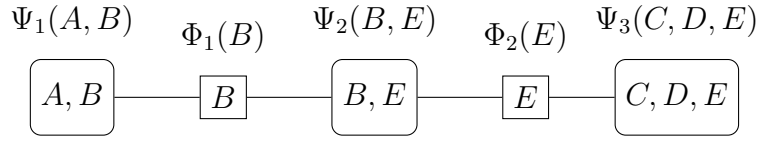


Figure 2.6: Example clique tree showing the clique potentials Ψ and separator potentials Φ .

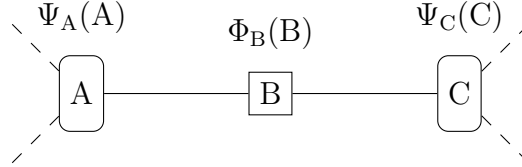


Figure 2.7: Part of a clique tree. The belief update algorithm passes a message from A to C.

Proposition 2.2. If neighbouring clusters have consistent marginals for the variables that they share, and also if the running intersection property holds, then the whole graph is consistent. So by doing local updates to make the potentials locally consistent, all the marginals can be computed. The algorithm and the proof are proposed in [39].

For a part of a general graph as shown in Figure 2.7, where A, B, and C are sets of random variables, the update rules when making A consistent with C are defined as

$$\Phi_B^*(B) = \sum_{A \setminus B} \Psi_A(A), \quad (2.4.6)$$

$$\Psi_C^*(C) = \frac{\Phi_B^*(B)}{\Phi_B(B)} \Psi_C(C). \quad (2.4.7)$$

When passing the update message from C to A, we have

$$\Phi_B^{**}(B) = \sum_{C \setminus B} \Psi_C^*(C), \quad (2.4.8)$$

$$\Psi_A^{**}(A) = \frac{\Phi_B^{**}(B)}{\Phi_B^*(B)} \Psi_A^*(A). \quad (2.4.9)$$

Global consistency can be guaranteed if the order in which messages are passed follows the message passing protocol, namely that a message can only be passed

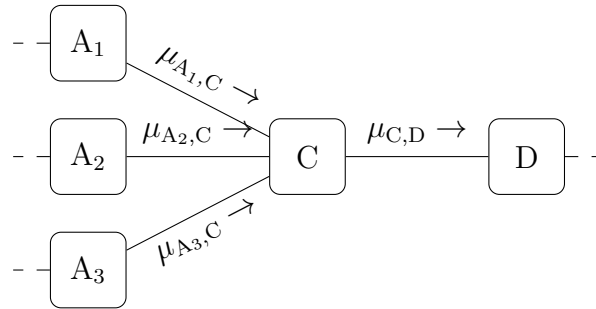


Figure 2.8: Part of a clique tree. The message passing algorithm passes a message from C to D once C received message from all its other neighbours (A_1 , A_2 , and A_3).

from a node if a message has been received from all the other neighbouring nodes.

2.4.4 Message passing algorithm

It is also possible to do the updates without storing the separator potentials. In this message passing algorithm, the message $\mu_{C,D}$ from any potential C to any neighbouring potential D is defined as

$$\mu_{C,D}(D) = \sum_{C \setminus D} \prod_{A \in N_G(C) \setminus D} \mu_{A,C}(C) \Psi_C(C). \quad (2.4.10)$$

This means that the message that C sends to D is proportional to the product of all the messages C received from its neighbours $N_G(C)$, except for the message that D is still to send back to C. This product is multiplied with C's potential and all the variables not in D are marginalised out. Figure 2.8 shows a part of a clique tree where C passes a message to one of its four neighbours.

This algorithm, which has its roots in [52] and is formulated in [57], differs from the belief update algorithm in that potentials are never updated because the messages carries all the information. We therefore have to add a separate data structure for the messages.

We use the same message passing protocol for this algorithm as for the belief update algorithm, namely that a cluster can only send a message to another cluster if it has received messages from all the other neighbouring clusters.

Once there are messages in both directions along all the edges, we can compute the probability of the variables represented by any node C by multiplying the cluster's potential function with all the cluster's incoming messages. So,

$$p(C) = \frac{1}{Z} \prod_{A \in N_G(C)} \mu_{A,C}(C) \Psi_C(C). \quad (2.4.11)$$

Z can be found at any node by multiplying all the incoming messages and summing out the remaining variables,

$$\sum_C p(C) = \sum_C \frac{1}{Z} \prod_{A \in N_G(C)} \mu_{A,C}(C) \Psi_C(C) \quad (2.4.12)$$

$$Z = \sum_C \prod_{A \in N_G(C)} \mu_{A,C}(C) \Psi_C(C). \quad (2.4.13)$$

This procedure is familiar because it is exactly the computation of the elimination algorithm, except that now we can find all the marginals by running the equivalent of the elimination algorithm twice, once towards some root node and once away from it.

Although we will not make use of the fact, it is interesting to note that the local message passing algorithm can be run even when the graph is not a tree. Messages then propagate in cycles, and in practice often converge, although not always. When they do converge, this *loopy belief propagation* scheme solves an approximation to the marginals that is known to statistical physicists as the Bethe approximation [53]. More recently, tree reweighted algorithms have been developed that will always converge [65].

2.4.5 Maximum probability configuration

We now turn to the problem of finding the single set of random variable values that gives the highest probability for a given distribution. This is in general different from the problem of finding the value of each variable that maximises the marginal of that variable.

Definition 2.6. The *most probable explanation (MPE)* or *maximal configuration* of $p(X)$ is

$$\arg \max_X p(X). \quad (2.4.14)$$

For the running example MRF, we want to find

$$\max_{A,B,C,D,E} p(A, B, C, D, E) = \max_{A,B,C,D,E} \frac{1}{Z} \Psi_1(A, B) \Psi_2(B, C) \Psi_3(C, D, E). \quad (2.4.15)$$

As with the variable elimination algorithm, we now reorder the maximisations to

$$\max_{A,B,C,D,E} p(A, B, C, D, E) = \frac{1}{Z} \max_{C,D,E} \Psi_3(C, D, E) \max_B \Psi_2(B, C) \max_A \Psi_1(A, B). \quad (2.4.16)$$

In fact, the same arguments apply to MPE as to variable elimination, and the algorithms can be adapted by changing the sums to maximums [53]. The sum product message now becomes the max product message

$$\mu_{C,D}(D) = \max_{C \setminus D} \prod_{A \in N_G(C) \setminus D} \mu_{A,C}(C) \Psi_D(D). \quad (2.4.17)$$

When the forward pass is finished, we have found the single maximal configuration of the root cluster but not of the others. The backwards messages now become the propagation of this maximum backwards by substituting in the previous maximal configurations.

If the cardinality of the variables is high, as is often the case in language applications where a variable representing a word might take on one of more than 10000 values, an approximate version of the max-product algorithm can be used to speed up inference. A message $\mu_{C,D}(D)$ from C to D, is approximated by $\mu'_{C,D}(D)$ where

$$\mu'_{C,D}(D) = \begin{cases} \mu_{C,D}(D) & \text{if } \mu_{C,D}(D) > \beta \\ 0 & \text{otherwise.} \end{cases} \quad (2.4.18)$$

for some threshold β . This is called *beam search* and it has the consequence that fewer multiplications have to be done between the potential and messages. Alternatively, instead of a threshold, the b values of $\mu_{C,D}(D)$ with the highest potentials can be used for some number b . Information is lost with this method. The true maximal configuration can be lost in one message if the corresponding value is too low. The number b is thus set to be as large as possible while allowing the computation to proceed in a reasonable time.

2.4.6 Semirings

The sum product and max product message passing algorithms are efficient because they rely on a recursion that is allowed by the associative and distributive properties of multiplication, summation, and maximisation. The abstraction of a semiring captures these properties and allows a more general way of looking at the message passing algorithms [2].

Definition 2.7. A *semiring* $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ consists of a set \mathbb{K} , a commutative operation \oplus with identity $\bar{0}$, and an associative operation \otimes with identity $\bar{1}$.

So far we looked at the *probability semiring* $(\mathbb{R}, +, \cdot, 0, 1)$ on the set of real numbers with the familiar additive and multiplying operators. The max product algorithm is defined on the semiring $(\mathbb{R}, \max, \cdot, 0, 1)$, where max replaces summation.

Algorithms can be shared between sum product and max product because summation and maximisation have similar properties.

For practical inference systems, numerical underflow is often a problem because many small numbers are multiplied. We therefore want to do the computations in the $-\log$ domain. The *log semiring*, $(\mathbb{R}_+, \oplus_{\log}, +, \infty, 0)$, can then be used. $\oplus_{\log}(A) = \log \sum_{a \in A} \exp(a)$ and can be calculated so as to avoid underflow by taking

$$\oplus_{\log}(A) = z + \log\left(\sum_{a \in A} \exp(a - z)\right)$$

$$\text{with } z = \max_{a \in A}(a).$$

2.5 Learning

So far we have assumed that the model and its parameters are known. In this section we look at how the parameters can be learned from data.

2.5.1 Bayesian statistics

Learning can be seen as inference if we take the Bayesian statistical view of parameters. In Bayesian statistics the parameters are themselves random variables, each with its own distribution given the data.

We are interested in the machine learning problem of classifying a new data point $\mathbf{x} = [x_1, x_2, \dots, x_D]^T$ into one of several classes $y \in \{1, \dots, M\}$ given a set of training examples $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$. In general, there can be more than one output variable for every example, so for the n th example $\mathbf{y}_n = [y_{n,1}, y_{n,2}, \dots, y_{n,T}]^T$ and $D = \{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_N, \mathbf{y}_N)\}$. We assume that the data points are *independent and identically distributed (i.i.d.)*.

To do the classification, we first want to find the probability over classes $p(\mathbf{y}|\mathbf{x}, D, \mathcal{Q})$, where \mathcal{Q} encodes any previous knowledge about the world we would want to use. We assume that the new data points are independent of all the previous points and our previous knowledge \mathcal{Q} if we are given a description, or model H_i from the set of all hypotheses we consider H . So,

$$p(\mathbf{y}|\mathbf{x}, D, \mathcal{Q}) = \sum_{H_i \in H} p(\mathbf{y}|\mathbf{x}, H_i) p(H_i|D, \mathcal{Q}). \quad (2.5.1)$$

Although in general one could consider many models and then sum over them to find the distribution over \mathbf{y} , we will often only evaluate with one model to save computation. Working with only one model is the same as assuming that the distribution over all models is very peaked at our chosen model H , so $p(H|D, \mathcal{Q})$ is approximately equal to one [44].

2.5.2 Parameter learning

We furthermore assume that the model H is parameterised by $\boldsymbol{\lambda}$ which describes how the potentials are filled. The problem of inferring \mathbf{y} now becomes that of finding

$$p(\mathbf{y}|\mathbf{x}, D, \mathcal{Q}) = \sum_{H_i \in H} p(\mathbf{y}|\mathbf{x}, H_i) p(H_i|D, \mathcal{Q}) \quad (2.5.2)$$

$$= \sum_{H_i \in H} \int_{\boldsymbol{\lambda}} p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) p(\boldsymbol{\lambda}|D, H_i) p(H_i|D, \mathcal{Q}) d\boldsymbol{\lambda} \quad (2.5.3)$$

$$\approx \int_{\boldsymbol{\lambda}} p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) p(\boldsymbol{\lambda}|D, H) d\boldsymbol{\lambda}. \quad (2.5.4)$$

Unfortunately, the integral over $\boldsymbol{\lambda}$ is often impossible to calculate analytically. There are approximate ways of solving the marginalisation problem, for instance various sampling based methods, but we will again assume that the parameter distribution is very peaked at its maximum. If we approximate the posterior distribution of $\boldsymbol{\lambda}$ with a Dirac delta function, then

$$p(\mathbf{y}|\mathbf{x}, D, \mathcal{Q}) \approx p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}_{ML}), \quad (2.5.5)$$

where $\boldsymbol{\lambda}_{ML}$ is the vector of most likely parameter values.

The problem now turn from a marginalisation problem to a maximisation problem. We want to find

$$\boldsymbol{\lambda}_{ML} = \arg \max_{\boldsymbol{\lambda}} p(\boldsymbol{\lambda} | D, H) \quad (2.5.6)$$

$$= \arg \max_{\boldsymbol{\lambda}} p(D | \boldsymbol{\lambda}, H) p(\boldsymbol{\lambda} | H) \quad (2.5.7)$$

$$= \arg \max_{\boldsymbol{\lambda}} \prod_{n=1}^N p(\mathbf{x}_n, \mathbf{y}_n | \boldsymbol{\lambda}, H) p(\boldsymbol{\lambda} | H), \quad (2.5.8)$$

where Bayes' rule is first used and then the fact that the training data are i.i.d.

Equation 2.5.4 is illustrated graphically in Figure 2.9. Here the common case is shown where there is a hidden variable \mathbf{z} present. Its counterparts in the training examples, $\{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_N\}$, are never observed. If we apply the message passing algorithm where we consider the node with unknown \mathbf{y} as the root of the tree, the marginal of \mathbf{y} is

$$p(\mathbf{y} | \mathbf{x}, D, H) = \frac{1}{Z} \int_{\boldsymbol{\lambda}} \sum_{\mathbf{z}} p(\mathbf{y}, \mathbf{z} | \mathbf{x}, \boldsymbol{\lambda}) \prod_{n=1}^N \sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\lambda}) p(\boldsymbol{\lambda} | H), \quad (2.5.9)$$

where the normalisation constant Z is again used. When the distribution of $\boldsymbol{\lambda}$ is approximated by Dirac deltas, we again want to find the most likely parameters as

$$\boldsymbol{\lambda}_{ML} = \arg \max_{\boldsymbol{\lambda}} \prod_{n=1}^N \sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\lambda}, H) p(\boldsymbol{\lambda} | H). \quad (2.5.10)$$

2.5.3 Optimisation

The marginalisation problem has been transformed to an optimisation problem where the objective function is the probability of the parameters given the training data and model. For the class of models that we are interested in, Newton-Raphson optimisation techniques are found to be useful [56]. These techniques require the first and second derivatives of the objective function, and they iteratively find points closer to where the derivative of the objective function is zero.

In practice, since

$$\boldsymbol{\lambda}_{ML} = \arg \max_{\boldsymbol{\lambda}} p(D | \boldsymbol{\lambda}, H) p(\boldsymbol{\lambda} | H) \quad (2.5.11)$$

$$= \arg \max_{\boldsymbol{\lambda}} \log p(D | \boldsymbol{\lambda}, H) p(\boldsymbol{\lambda} | H), \quad (2.5.12)$$

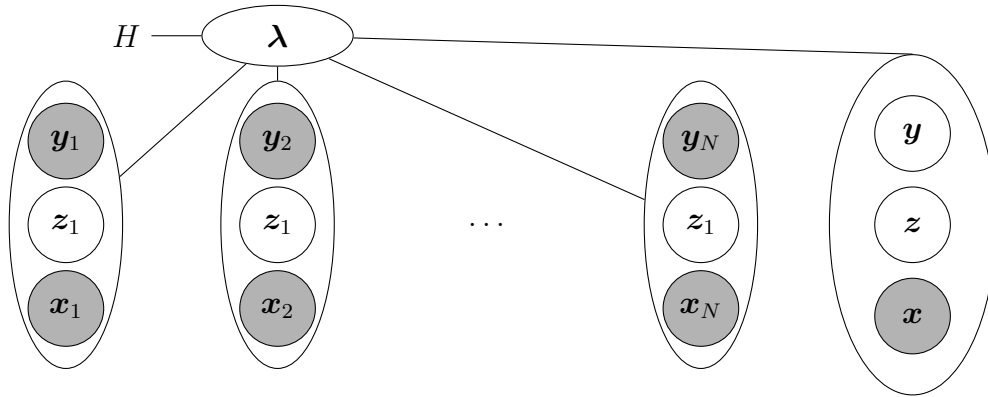


Figure 2.9: An MRF that represents the typical supervised learning situation. Each oval represents an MRF with dependencies between the label y_i , hidden variables z_i , and input features x_i of a training example. The training examples are independent of the new point x with unknown label y given the parameters.

and since the product over the training data now becomes the more easily differentiable sum over the training data, the objective function is taken to be the *log posterior*, also called the *regularised log likelihood* \mathcal{L}

$$\mathcal{L} = \log p(\mathbf{D}|\boldsymbol{\lambda}, H)p(\boldsymbol{\lambda}|H) \quad (2.5.13)$$

$$= \sum_{i=1}^N \log p(\mathbf{D}_i|\boldsymbol{\lambda}, H) + \log p(\boldsymbol{\lambda}|H). \quad (2.5.14)$$

To find the maximum, Newton-Raphson optimisation uses the iterative update

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t - \alpha \mathbf{H}_t^{-1} \nabla \mathcal{L}_t, \quad (2.5.15)$$

where \mathbf{H} is the Hessian matrix of second derivatives, $\nabla \mathcal{L}$ is the vector of first derivatives, and α is the learning rate.

Practically the Newton-Raphson update as described above is not used since the Hessian matrix becomes infeasibly large (its size is the number of parameters squared), and since it is only guaranteed to find the optimum if the search space is convex (See 4.2.3). For many problems of practical interest the search space is non-convex and special checks have to be added to ensure that the algorithm converges to a local optimum. An algorithm that iteratively builds up an approximation to the Hessian, without ever holding it in memory, and with the necessary checks that have become popular is the

LM-BFGS algorithm [48]. We will not describe how it works here, but there are implementations available and we assume that the algorithm can be used once we have a way of calculating the log likelihood and its derivative.

2.6 Conclusion

In this chapter we introduced undirected PGMs. They present a way of visualising dependencies in probability density functions, and also provide a data structure to store these PDFs and efficient algorithms to find marginals and maximal configurations of the random variables. PGMs provide a framework in which we can develop probabilistic models to do classification.

Chapter 3

Weighted finite state machines

Where PGMs graphically represent arbitrary dependencies between random variables in a probability density function, weighted finite state acceptors (WFSAs) and weighted finite state transducers (WFSTs) are more fine-grained graphical representations of associations between variables when the *first order Markov assumption* is made. These types of “chain” dependencies, as seen in Figure 3.1, are often used in language applications because language consists of sequences of symbols [47].

Weighted finite state transducers and weighted finite state acceptors, or weighted finite state machines (WFSMs) as we collectively call them, are defined in the general case on a semiring, but we look only at the probability semiring case.

3.1 Weighted finite state acceptors

Definition 3.1. According to [47], a WFSA is a tuple $(\Sigma, Q, I, F, E, \lambda, \rho)$, where: Σ is the input alphabet; Q is a finite set of states; $I \subseteq Q$ is the set of initial states; $F \subseteq Q$ is the set of final states; $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{R} \times Q$ is the set of transitions, where ϵ is a special symbol to denote a non-emitting edge.

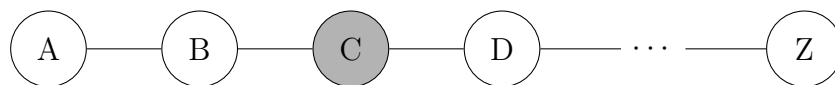


Figure 3.1: A Markov chain. All variables to the right of an observed variable are independent of all the variables to the left of that variable given that variable. So $A, B \perp D, \dots, Z | C$.

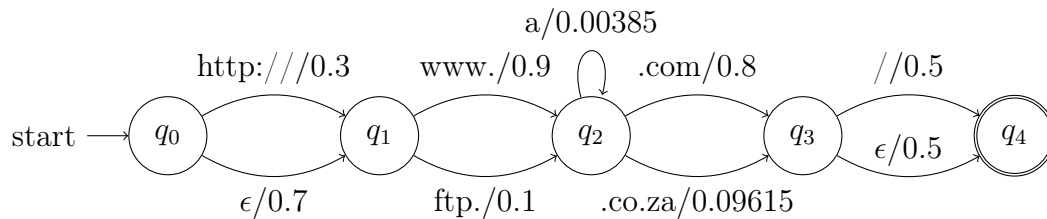


Figure 3.2: An example of a WFSA that represents a few URLs. The nodes represent states and allowable transitions are represented with edges. Each edge emits a symbol S with a probability P shown as S/P above the edge. The starting state is labelled “start” and the ending states are drawn double.

λ is the initial weight and ρ is the final weight function. We take λ and ρ to be always equal to one.

The other weights are probabilities so that all the weights on the edges coming out of any state sums to one.

A path through a WFSA is a sequence of states that are connected by edges. Every path emits a string which can be found by concatenating the symbols on the edges. Every path is also associated with a probability that can be found by multiplying all the weights on the edges. A WFSA therefore can be used to assign a probability to a string by summing the probability of all the paths through the WFSA that emits that string.

WFSA's are represented by directed graphs, where a node represents each state, with directed edges representing transitions. The example WFSA shown in Figure 3.2 represents different web URLs. The probability of the string “www.a.com” is found by following the path $q_0q_1q_2q_3$ and multiplying the associated weights to find $0.7 \cdot 0.9 \cdot 0.00385 \cdot 0.8 \cdot 0.5 \approx 0.001$.

We use WFSA's to tokenize text. For the tokenisation problem, words, URLs, white space and so forth are each represented with a state. See Section 6.2 for details.

3.2 Weighted finite state transducers

Weighted finite state transducers are defined similarly to WFSA's. In addition to the input alphabet they also have an output alphabet.

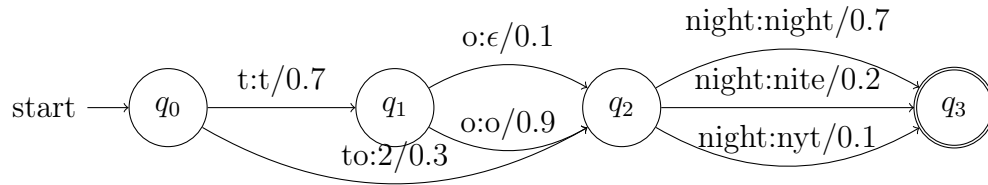


Figure 3.3: An example of a WFST that transforms “tonight” into “tnight”, “tonight”, “tnite”, “tonite”, “tnyt”, “tonyt”, “2night”, “2nite”, or “2nyt” with different probabilities. Above each edge is the source symbols, separated from the target symbols by a “:”, followed by a “/” and the probability of that transition. The special symbol ϵ denotes that that edge does not emit a character.

Definition 3.2. A WFST is an eight tuple $(\Sigma, \Omega, Q, I, F, E, \lambda, \rho)$, where: Σ is the input alphabet; Ω is the output alphabet; Q is a finite set of states; $I \subseteq Q$ is the set of initial states; $F \subseteq Q$ is the set of final states; $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times \mathbb{R} \times Q$ is the set of transitions. λ is the initial weight and ρ is the final weight function. We take λ and ρ to be also always equal to one.

A path in a WFST produces two strings and a probability. The weights are normalised so that the WFST gives the probability of the second string given the first string and the transducer.

For example, given H_w , the WFST in Figure 3.3, the probability $p(2\text{nyt}|\text{tonight}, H_w) = 0.3 \cdot 0.1 = 0.03$.

A number of operations are possible with transducers. An important operation that we will see again when we look at the noisy channel model (See Section 5.2.6), is transducer *composition*, which we describe briefly and informally: When two transducers w and s are composed, written $g = w \circ s$, a new transducer g is formed. For a certain input string, this transducer outputs the string that would be output if the input string is first transduced by w , and its output then given to s as input. The probability of the output string that is produced when the string has moved through the whole “pipeline” is the probability that w would have given for the first transduction multiplied by the probability that s would give for the second stage. We will, however, use WFSTs only to assign probabilities to string pairs.

The weights of WFSTs can be trained with *expectation maximization* type algorithms [19] when input and output string pairs are give as training examples. When training a WFST, the likelihood of the training pairs are iteratively

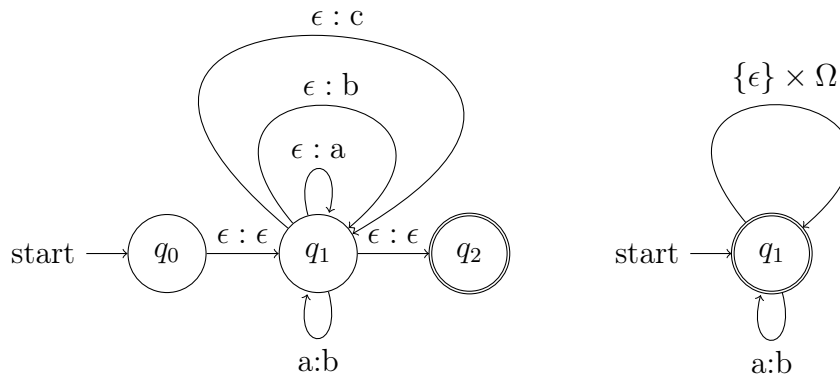


Figure 3.4: An example of the notation. On the left hand side is a complete FST, and on the right is a simplified representation of the same FST. In the shortened version on the right, $\Omega = \{a, b, c\}$ and the arc that “emits” the set $\{\epsilon\} \times \Omega = \{(\epsilon, a), (\epsilon, b), (\epsilon, c)\}$ is expanded so that there is an arc for each pair in the set, where a pair represents the input and output symbols.

maximised by updating the weights on the edges of the model. We will not go into details, but optimised software packages are available [26].

3.3 Notation

Some of the finite state models we discuss have thousands of arcs, so to represent them in this thesis we use a shorthand where similar arcs are grouped together and the set of their emission symbols is represented with a single character as can be seen in Figure 3.4. Each of these arcs also has a different emission weight unless specified otherwise.

3.4 Connection with linear chain models

When weighted finite state automata and transducers are interpreted as probabilistic graphical models, they form *Markov chains* as seen in Figure 3.1. Each element of a path through a WFSA or WFST is a random variable that is only dependent on the previous state and the current position in the input and output strings.

We can adapt the message passing algorithm to do inference in WFSA and WFSTs, and this variant of the algorithm is referred to as the *forward-backward*

algorithm. The max product algorithm is also known as the *Viterbi algorithm* in this setting.

Although we have shown the case where the probabilities are locally normalised (all outgoing edges' weights sum to one), we can also have globally normalised state machines [19]. We discuss them further in Chapter 4.

3.5 Conclusion

WFSA's and WFST's provide graphical representations of probability distributions for which the Markov assumption is made. This assumption is useful in many language applications. Although these models are traditionally only locally normalised, they can also be globally normalised. We look at this case in Chapter 4.

Chapter 4

Conditional random fields

In Chapter 2, PGMs and MRFs are introduced. Here we look at a specific type of undirected graphical model. Conditional random fields (CRFs) are used for classification and directly model the conditional probability of the label given observations. They have been successfully applied to natural language problems, such as tagging words in a sentence with their parts of speech [37].

After motivating the use of conditional models, we see that CRFs can be trained by using the gradient of the conditional log-likelihood. We then look at a few specific models that will be useful when building a text normalisation system.

As in [60], the different CRF models are discussed as both generalisations of logistic regression, and also the discriminative counterparts to well-known generative models such as hidden Markov models and naive Bayes classifiers.

4.1 Introduction

CRFs are usually used in a supervised classification setting. In this setting, the data points D consist of pairs of input vectors \mathbf{x}_i , also known as input features, and output vectors \mathbf{y}_i that are also called labels, $\{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_N, \mathbf{y}_N)\}$. The task is to train a model to give label predictions for features with unknown labels.

4.1.1 Generative and discriminative models

As we saw in Section 2.5.2, we are interested in the probability distribution of the unknown label \mathbf{y} given the input features,

$$p(\mathbf{y}|\mathbf{x}, D, \mathcal{Q}) \approx p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}_{ML}), \quad (4.1.1)$$

One way of tackling the problem of finding $p(\mathbf{y})$ is first to model the probability of \mathbf{x} given \mathbf{y} and then to use Bayes' rule to find \mathbf{y} given \mathbf{x} ,

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) = \frac{p(\mathbf{x}|\mathbf{y}, \boldsymbol{\lambda})p(\mathbf{y}|\boldsymbol{\lambda})}{p(\mathbf{x}|\boldsymbol{\lambda})}. \quad (4.1.2)$$

This type of model is called a *generative* model, because synthetic examples can be 'generated' by sampling from the joint distribution. These types of models are described naturally by directed graphical models. Generative models often make the assumption that observations are independent given the labels, and it is difficult to build models where this strict conditional independence assumption can be relaxed without running into computational problems [6]. Inappropriate independence assumptions are indeed often made for computational tractability [37].

Opposed to these types of models are *discriminative* models. We look specifically at conditional models. They model the conditional distribution $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda})$ directly. One disadvantage is that it is more difficult to interpret conditional models because they do not necessarily give insight into how the data are generated. Discriminative models are also known to give better performance than similar generative models if given enough training data, but may overfit with limited or partially labeled data [6].

For example, consider the MRF described in Figure 4.1. We might know that A , B , and C will be always observed and that we will be only interested in querying the model for D . Then it is unnecessary to model any of the variables that are conditionally independent of our variable of interest D . For the sake of finding D it is therefore sufficient to model the dependencies in Figure 4.2.

4.1.2 Feature engineering

With a conditional model, the model stays simple while allowing complicated dependencies between the observed variables. This allows good predictions to be made without explicitly having to model a system in the finest detail [37].

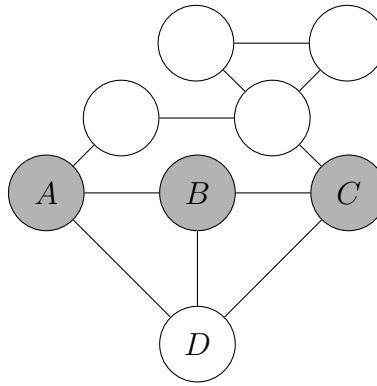


Figure 4.1: A model where we always observe A , B , and C . We are interested only in querying D .

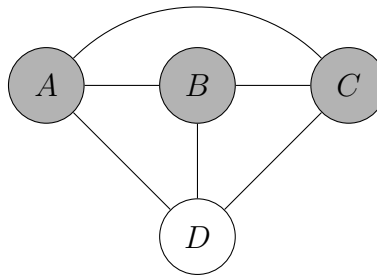


Figure 4.2: The same model as Figure 4.1 where only dependencies that are necessary if D is queried are modelled.

It also allows complicated and *overlapping features* to be used, which would violate the independence assumptions in comparable generative models [55]. This means that we can take arbitrary functions and combinations of the variables that we observe directly and use them as features instead. CRFs therefore lend themselves to *feature engineering*. Experts design the input features according to what is known to be discriminative features for a certain problem.

For example, the naive Bayes model and logistic regression model (See Section 4.3) share a similar graphical model structure [30]. Naive Bayes is a generative model and logistic regression is discriminative. The naive Bayes

model assumes independence of the features \mathbf{x} given the label y ,

$$p(y|\mathbf{x}, \boldsymbol{\lambda}) \propto p(\mathbf{x}|y, \boldsymbol{\lambda})p(y|\boldsymbol{\lambda}) \quad (4.1.3)$$

$$= p(y|\boldsymbol{\lambda}) \prod_{i=1}^D p(x_i|y, \boldsymbol{\lambda}). \quad (4.1.4)$$

If some of the x_i s are dependent given y , the independence assumption is violated and model performance deteriorates [37]. Say we want to classify the language usage in a sentence as formal or informal. We can let \mathbf{x} be a vector the length of our vocabulary with ones for the words that are present in the sentence and zeros otherwise. This bag-of-words model already violates the independence assumption, as words in a sentence are correlated even when we know whether the sentence is formal or informal.

Logistic regression, on the other hand, models $p(y|\mathbf{x})$ without assuming independence between features. We can therefore even add correlated features explicitly without violating the independence assumption. For instance, we can add a feature that is one if the sentence starts with a capital letter and zero otherwise, thereby improving classification performance if the feature proves to be discriminative.

The advantage of feature engineering is that previous knowledge about a problem or domain can be incorporated easily in the model. The disadvantage is that it takes previous knowledge about a problem to come up with the features. Although there are ways of doing automatic feature selection, feature engineering can be a slow and expensive process [38].

4.1.3 Parameterisation

For NLP problems, CRFs often use a log linear parameterisation (See Section 2.3.4). So,

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}}} \exp\left\{ \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_{i,k} f_{i,k}(\mathbf{y}_{C_i}, \mathbf{x}_{C_i}) \right\}, \quad (4.1.5)$$

with

$$Z_{\mathbf{x}} = \sum_{\mathbf{y}} \exp\left\{ \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_{i,k} f_{i,k}(\mathbf{y}_{C_i}, \mathbf{x}_{C_i}) \right\}. \quad (4.1.6)$$

Here we partition the input vectors \mathbf{x} and \mathbf{y} so that the partitions can be used as the realisations of the variables in each clique. So, \mathbf{x} is the vector $[\mathbf{x}_{C_1}^T, \mathbf{x}_{C_2}^T, \dots, \mathbf{x}_{C_{|C|}}^T]^T$, and $\mathbf{y} = [\mathbf{y}_{C_1}^T, \mathbf{y}_{C_2}^T, \dots, \mathbf{y}_{C_{|C|}}^T]^T$.

For problems where the input variables \mathbf{x} are text, it is useful to let the feature functions $f_{i,j}(\mathbf{y}_{C_i}, \mathbf{x}_{C_i})$ be binary indicator functions.

An input vector of dimension D is thus transformed into a binary vector of dimension K by the feature functions. K is therefore also the dimension of the parameter vector $\boldsymbol{\lambda}$, and can be greater or smaller than D .

For example, in a larger CRF, the potential function

$$\Psi_1(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}, \boldsymbol{\lambda}_1) = \exp\left\{\sum_{k=1}^K \lambda_{1,k} f_{1,k}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1})\right\} \quad (4.1.7)$$

is defined on the first clique C_1 and tells us more about an animal we are observing. $\mathbf{y}_{C_1} = [y_{C_1,1}, y_{C_1,2}]$, with $y_{C_1,1} \in \{\text{“bat”}, \text{“bird”}\}$ and $y_{C_1,2} \in \{\text{“mammal”}, \text{“non-mammal”}\}$. So we want to know simultaneously to what class of animals it belongs and also specifically whether it is a bat or a bird. $x_{C_1,1}$ indicates different actions that the animal is doing, and we disregard the other components of \mathbf{x}_{C_1} .

Each indicator function is defined to be 1 for a different assignment of the input and output variables, so

$$\begin{aligned} f_{1,1}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}) &= \begin{cases} 1 & \text{if } y_{C_1,1} = \text{“bat” and } x_{C_1,1} = \text{“flying”} \\ 0 & \text{otherwise} \end{cases} \\ f_{1,2}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}) &= \begin{cases} 1 & \text{if } y_{C_1,1} = \text{“bat” and } x_{C_1,1} = \text{“hanging upside down”} \\ 0 & \text{otherwise} \end{cases} \\ f_{1,3}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}) &= \begin{cases} 1 & \text{if } y_{C_1,1} = \text{“bat” and } x_{C_1,1} = \text{“laying eggs”} \\ 0 & \text{otherwise} \end{cases} \\ f_{1,4}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}) &= \begin{cases} 1 & \text{if } y_{C_1,1} = \text{“bird” and } x_{C_1,1} = \text{“flying”} \\ 0 & \text{otherwise} \end{cases} \\ &\vdots \\ f_{1,10}(\mathbf{x}_{C_1}, \mathbf{y}_{C_1}) &= \begin{cases} 1 & \text{if } y_{C_1,1} = \text{“bat” and } y_{C_1,2} = \text{“mammal”} \\ 0 & \text{otherwise} \end{cases} . \end{aligned}$$

Each feature function is associated with a parameter $\lambda_{i,j}$. The parameters take on any real values. The parameters therefore can be interpreted as

either “supporting” the configuration of variables that makes the associated feature function return 1 if the parameter value is positive, or “opposing” the configuration if the parameter value is negative.

4.1.4 Parameter tying

The models we discuss in the rest of this thesis make use of *parameter tying*, which means that parameters and feature functions are associated with more than one clique potential.

In our case all the cliques share the same parameters and feature functions, so $f_{i,k} = f_k$ and $\lambda_{i,k} = \lambda_k$ for $i = 1$ to $|\mathcal{C}|$.

4.2 Learning

The training objective function for CRFs is from Equation 2.5.13

$$\mathcal{L} = \log p(\mathbf{D}, \boldsymbol{\lambda}|H) \quad (4.2.1)$$

Maximising this quantity is equivalent to maximising

$$\arg \max_{\boldsymbol{\lambda}} \mathcal{L} = \arg \max_{\boldsymbol{\lambda}} \log p(\mathbf{x}, \mathbf{y}, \boldsymbol{\lambda}|H) \quad (4.2.2)$$

$$= \arg \max_{\boldsymbol{\lambda}} \log p(\mathbf{y}, \boldsymbol{\lambda}|\mathbf{x}, H)p(\mathbf{x}|H) \quad (4.2.3)$$

$$= \arg \max_{\boldsymbol{\lambda}} \log p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda})p(\boldsymbol{\lambda}|\mathbf{x}, H) \quad (4.2.4)$$

$$= \arg \max_{\boldsymbol{\lambda}} \log p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda})p(\boldsymbol{\lambda}|H), \quad (4.2.5)$$

where we assume that $\boldsymbol{\lambda}$ is independent of \mathbf{x} given H (or we can equivalently assume that \mathbf{x} is parameterised by a different set of parameters that is independent of $\boldsymbol{\lambda}$ given H [6].)

The regularised *conditional log likelihood* is now defined as

$$\mathcal{L}_{\mathcal{C}} = \sum_{n=1}^N \log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\lambda}) + \log p(\boldsymbol{\lambda}|H). \quad (4.2.6)$$

This is the sum of a regularisation term $\log p(\boldsymbol{\lambda}|H)$ and the unregularised conditional log likelihood $\mathcal{L}'_{\mathcal{C}} = \log p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda})$.

4.2.1 Derivative

To train a CRF with LM-BFGS we need the derivative of the objective with respect to the parameters

$$\nabla_{\lambda} \mathcal{L}_{\mathcal{C}} = \left[\frac{\partial \mathcal{L}_{\mathcal{C}}}{\partial \lambda_1}, \frac{\partial \mathcal{L}_{\mathcal{C}}}{\partial \lambda_2}, \dots, \frac{\partial \mathcal{L}_{\mathcal{C}}}{\partial \lambda_K} \right]^T \quad (4.2.7)$$

where, for the case where the model is fully observed during training

$$\mathcal{L}'_{\mathcal{C}} = \sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \lambda) \quad (4.2.8)$$

$$= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \log \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \lambda) - \log Z_{\mathbf{x}_n}(\mathbf{x}_n, \lambda) \quad (4.2.9)$$

$$= \sum_{n=1}^N \left\{ \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_k f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) \right\} - \log \sum_{\mathbf{y}'_n} \exp \left\{ \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_k f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \right\}. \quad (4.2.10)$$

The derivative is found by substituting in Equations 4.1.5 and 4.1.6, taking the derivative, and factorising out the probabilities. So,

$$\frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} = \frac{\partial}{\partial \lambda_k} \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \log \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \lambda) - \frac{\partial}{\partial \lambda_k} \log \sum_{\mathbf{y}'_n} \prod_{C_i \in \mathcal{C}} \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \lambda) \quad (4.2.11)$$

$$= \frac{\partial}{\partial \lambda_k} \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_k f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \frac{\sum_{\mathbf{y}'_n} \frac{\partial}{\partial \lambda_k} \prod_{C_i \in \mathcal{C}} \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \lambda)}{\sum_{\mathbf{y}'_n} \prod_{C_i \in \mathcal{C}} \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \lambda)} \quad (4.2.12)$$

$$= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \sum_{\mathbf{y}'_n} \frac{1}{Z_{\mathbf{x}_n}} \exp \left\{ \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_k f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \right\} \times \frac{\partial}{\partial \lambda_k} \sum_{C_i \in \mathcal{C}} \sum_{k=1}^K \lambda_k f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \quad (4.2.13)$$

$$= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \sum_{\mathbf{y}'_n} p(\mathbf{y}'_n | \mathbf{x}_n, \lambda) \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \quad (4.2.14)$$

$$= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \mathbb{E}_{p(\mathbf{y}'_n | \mathbf{x}_n, \lambda)} \left[\sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \right], \quad (4.2.15)$$

which is the empirical feature count minus the expected feature count over the model distribution [66]. Then, using the properties of expected values, we find the derivative can be calculated when the marginals are known,

$$\frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} = \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{y}'_{n,C_i}} p(\mathbf{y}'_{n,C_i} | \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}). \quad (4.2.16)$$

For the more general case where there are hidden or latent variables \mathbf{z} that are never observed, the unregularised log likelihood is

$$\mathcal{L}'_{\mathcal{C}} = \sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\lambda}) \quad (4.2.17)$$

$$= \sum_{n=1}^N \log \sum_{\mathbf{z}_n} \frac{1}{Z_{\mathbf{x}_n}} \prod_{C_i \in \mathcal{C}} \Psi_i(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \mathbf{z}_{n,C_i}, \boldsymbol{\lambda}) \quad (4.2.18)$$

and its derivative similarly found to be

$$\begin{aligned} \frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} &= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{z}_{n,C_i}} p(\mathbf{z}_{n,C_i} | \mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \mathbf{z}_{n,C_i}) \\ &\quad - \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}} p(\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i} | \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}) \end{aligned} \quad (4.2.19)$$

$$\begin{aligned} &= \sum_{n=1}^N \mathbb{E}_{p(\mathbf{z}_n | \mathbf{x}_n, \mathbf{y}_n, \boldsymbol{\lambda})} \left[\sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \mathbf{z}_{n,C_i}) \right] \\ &\quad - \mathbb{E}_{p(\mathbf{y}'_n, \mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\lambda})} \left[\sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}) \right]. \end{aligned} \quad (4.2.20)$$

For both cases, the derivative is thus the difference between the expected value of the sum of the k th feature functions over the empirical distribution, and the expected value of the sum of the k th feature functions over the model distribution. These expected values can be calculated once all the marginals are known for all the example inputs. To calculate the derivative it is thus necessary to first run inference with the current parameters on all the training examples.

We expect the derivative to be zero at the maximum of the unregularised objective. This has the intuitive consequence that the empirical expected value must equal the model expected value at the optimum.

4.2.2 Regularisation

To avoid overfitting, however, the model is *regularised*. This means that large parameter values are penalised, which in practice leads to models that generalise better.

From the Bayesian perspective, regularisation is a consequence of setting a prior distribution for the parameters. A mathematically convenient prior is the Gaussian PDF with zero mean and a diagonal covariance matrix. So, with the covariance matrix $\Sigma = \sigma^2 \mathbf{I}$, we have

$$p(\boldsymbol{\lambda}|H) = \frac{1}{(2\pi)^{K/2} |\Sigma|^{1/2}} \exp \left[-\frac{1}{2} \boldsymbol{\lambda}^T \Sigma^{-1} \boldsymbol{\lambda} \right] \quad (4.2.21)$$

$$= \frac{1}{(2\pi)^{K/2} \sigma^K} \exp - \sum_{k=1}^K \frac{\lambda_k^2}{2\sigma^2}. \quad (4.2.22)$$

The derivative of the log of this prior is

$$\frac{\partial \log p(\boldsymbol{\lambda}|H)}{\partial \lambda_k} = -\frac{\lambda_k}{\sigma^2}. \quad (4.2.23)$$

The regularised log likelihood is now

$$\mathcal{L}_C = \mathcal{L}_C' - \sum_{k=1}^K \frac{\lambda_k^2}{2\sigma^2} - \frac{K}{2} \log 2\pi - K \log \sigma, \quad (4.2.24)$$

and its derivative

$$\frac{\partial \mathcal{L}_C}{\partial \lambda_k} = \frac{\partial \mathcal{L}_C'}{\partial \lambda_k} - \frac{\lambda_k}{\sigma^2}. \quad (4.2.25)$$

A new parameter, the regularisation parameter σ , has been introduced. Typically it is set by training the model with different values of σ and evaluating on a held out validation set. The value that gives the best classification performance on this data set is then used to train the final model with the validation data added to the training data.

4.2.3 Concavity of objective function

Convergence to a global optimum can be guaranteed if the objective function is convex or concave. The properties of convex and concave functions [7] can be used to show that the fully observed log-linear CRF's log-likelihood is concave [37]:

- Linear functions are concave and convex,
- sums of concave functions are concave,
- the composition of a convex function and a linear mapping is convex, and
- the $\log \sum \exp$ function is convex.

The first term of the log-likelihood (See Equation 4.2.10) is a linear function of the parameters and is thus concave, while the second term is the negative of the log-sum-exp of a linear function and is therefore also concave. The regularisation term is a parabola which is also concave, so the sum of these terms are concave. The LM-BFGS optimiser we use can only minimise, so we give it the negative of the log-likelihood. This final objective function is therefore convex.

For CRFs with hidden variables, the same arguments cannot be used and these models' log-likelihoods are in general non-concave. For example, we are interested in models where the hidden variables are discrete. For parameter values that maximise the log likelihood, an equivalent solution can be found by relabelling the hidden variable values. The log likelihood is therefore multi-modal and non-concave [67]. Please see the section on Hidden CRFs (Section 4.5) and HACRFs (Section 4.6) for more on the models with hidden variables and Section 6.4.3.3 for an example of a training run.

4.3 Logistic regression

Logistic regression is an example of a simple type of log linear CRF. It is used for two class classification. The generalisation to more classes is known as *softmax* classification. This model can differentiate between points that are separable by a linear decision boundary. The name sprouts from the fact that it uses the *logistic function* $\frac{1}{1+\exp\{-ax\}}$ to 'soften' the decision boundary and thereby give a probability to a point according to how near the point is to the boundary.

According to Ng and Jordan's terminology [30], logistic regression is the discriminative counterpart to the naive Bayes classifier.

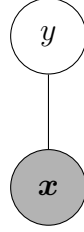


Figure 4.3: Graphical structure of the logistic regression CRF.

4.3.1 Model

Linear regression is defined as a model where the log of the ratio of the probability of a point being in a certain class to the probability of the point being in the other class is a linear combination of the features and the parameters,

$$\log \frac{p(y = 0|\mathbf{x})}{p(y = 1|\mathbf{x})} = \boldsymbol{\lambda}^T \mathbf{x}. \quad (4.3.1)$$

This is equivalent to using the one clique log linear model depicted in Figure 4.3, where we can write the conditional probability directly as

$$p(y|\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}}} \exp\left\{\sum_{k=1}^K \lambda_k f_k(\mathbf{x}, y)\right\}. \quad (4.3.2)$$

In NLP applications, \mathbf{f} is often taken to be a vector of indicator functions so that

$$f_k(\mathbf{x}, y) = \begin{cases} 1 & \text{if } y = 0 \text{ and } x_k = 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.3.3)$$

$$(4.3.4)$$

for $k = 0, 1, 2, \dots, D - 1$, where D is the number of binary features in \mathbf{x} , and

$$f_k(\mathbf{x}, y) = \begin{cases} 1 & \text{if } y = 1 \text{ and } x_{k-D} = 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.3.5)$$

for $k = D, D + 1, \dots, 2D - 1$.

The partition function $Z_{\mathbf{x}}$, which is a function of \mathbf{x} , can be calculated as

$$Z_{\mathbf{x}} = \sum_y \exp\left\{\sum_{k=1}^K \lambda_k f_k(\mathbf{x}, y)\right\} \quad (4.3.6)$$

$$= \exp\left\{\sum_{k=1}^K \lambda_k f_k(\mathbf{x}, y = 0)\right\} + \exp\left\{\sum_{k=1}^K \lambda_k f_k(\mathbf{x}, y = 1)\right\}. \quad (4.3.7)$$

The extension to more than two classes can be done by adding more indicator functions $f_k(\mathbf{x}, y)$ that select higher values of y .

4.3.2 Learning

The derivative of the log likelihood can be found by using Equation 4.2.16 for general CRFs that are fully observed during training. Here there is only one clique, and \mathbf{y} has one element y that is a discrete variable that can take on a finite number of values $y \in \{1 \dots M\}$, so

$$\frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} = \sum_{n=1}^N f_k(\mathbf{x}_n, y_n) - \sum_{y'_n=1}^M p(y'_n | \mathbf{x}_n, \boldsymbol{\lambda}) f_k(\mathbf{x}_n, y'_n). \quad (4.3.8)$$

4.4 Linear-chain CRFs

Linear-chain CRFs are introduced by Lafferty, McCallum, and Pereira as a way of doing discriminative sequence labelling [37]. These models are the discriminative counterparts to hidden Markov models and WFSAs.

4.4.1 Model

When we have many output variables, we might be interested in modelling the dependencies between them. Linear-chain CRFs are a simple way of modelling dependencies in sequences of output variables such as time dependent models for language applications. For this model we make the first order Markov assumption, namely that if we have a sequence of random variables Y_1, Y_2, \dots, Y_T , then $Y_1, \dots, Y_{t-1} \perp Y_{t+1}, \dots, Y_T | Y_t$. This assumption is graphically depicted in Figure 3.1.

This model can be seen as a way of extending the softmax model so that it takes time dependencies into account. It lends itself to tagging sequences like words with tags that are dependent on context. One successful application is the NLP task of part of speech tagging (POS tagging). Each word must be tagged with ‘noun’, ‘verb’, ‘preposition’, and so forth. A softmax classifier can be trained for this problem, but one would typically find that nouns are very likely after articles, so neighbouring output variables give useful information in deciding a tag.

From the graphical model in Figure 4.4, we have the conditional probability

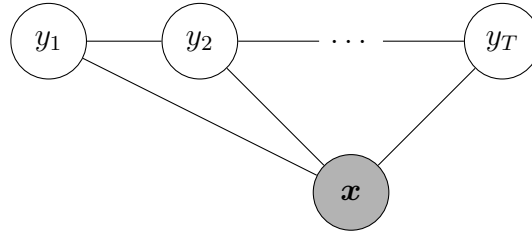
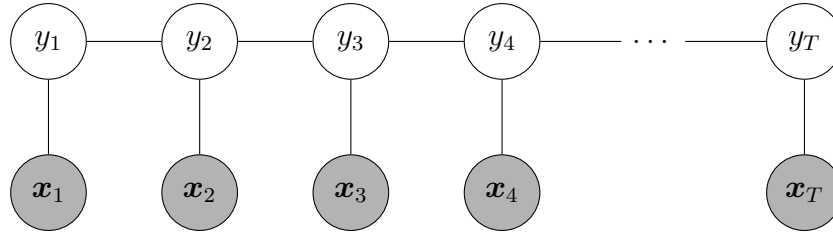


Figure 4.4: The graphical representation of the linear chain CRF.

Figure 4.5: A linear chain CRF where every output variable y_t is only dependent on the corresponding \mathbf{x}_t .

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}, y_t, y_{t+1}, \boldsymbol{\lambda}), \quad (4.4.1)$$

where $\mathbf{y} = [y_1, y_2, \dots, y_T]^T$, and each y_t can take on the value of a label or tag. For example, $y_t \in \{\text{noun, preposition, } \dots, \text{verb}\}$.

In a typical text application each potential Ψ_t takes only a window of \mathbf{x} around the current t instead of the whole \mathbf{x} . To simplify the implementation of a practical system, the input vectors that are “triggered” in time step t are duplicated so that there is a single vector for every time step as can be seen in Figure 4.5. So \mathbf{x} is partitioned so that $\mathbf{x} = [\mathbf{x}_1^T, \mathbf{x}_2^T, \dots, \mathbf{x}_T^T]^T$ and

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}_t, y_t, y_{t+1}, \boldsymbol{\lambda}). \quad (4.4.2)$$

4.4.2 Feature Functions

For the POS tagging problem, the input sequence “The cat” can be encoded $\mathbf{x}_1 = [-, \text{The}, \text{cat}, 1, 0]$ and $\mathbf{x}_2 = [\text{The}, \text{cat}, -, 0, 0]$. The first element in the vector \mathbf{x}_t is the previous token in the sequence and $-$ if there is no previous token; the second element is the current token; the third element is the next

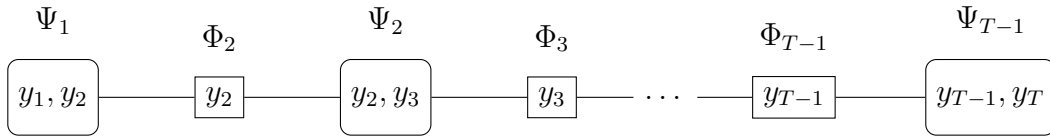


Figure 4.6: A junction tree representation of the linear chain CRF model.

token; the fourth element is 1 if the current token is capitalised and 0 otherwise; and the last element is 1 if the current token ends with “ing” and 0 otherwise.

Typical feature functions are

$$f_{t,1}(\mathbf{x}_t, y_t, y_{t+1}) = \begin{cases} 1 & \text{if } y_{t+1} = \text{'noun'} \text{ and } y_t = \text{'determiner'} \\ 0 & \text{otherwise} \end{cases}, \quad (4.4.3)$$

$$f_{t,2}(\mathbf{x}_t, y_t, y_{t+1}) = \begin{cases} 1 & \text{if } y_{t+1} = \text{'verb'} \text{ and } y_t = \text{'determiner'} \\ 0 & \text{otherwise} \end{cases}, \quad (4.4.4)$$

$$f_{t,3}(\mathbf{x}_t, y_t, y_{t+1}) = \begin{cases} 1 & \text{if } y_t = \text{'determiner'} \text{ and } x_{t,2} = \text{'the'} \\ 0 & \text{otherwise} \end{cases}, \quad (4.4.5)$$

$$f_{t,4}(\mathbf{x}_t, y_t, y_{t+1}) = \begin{cases} 1 & \text{if } y_t = \text{'past participle'} \text{ and } x_{t,5} = 1 \text{ (ends on 'ing')} \\ 0 & \text{otherwise} \end{cases}, \quad (4.4.6)$$

and so forth.

After training on examples, we might expect that the weight λ_1 associated with $f_{t,1}$ will be positive but λ_2 negative. $f_{t,3}$'s weight should be very strong and $f_{t,4}$'s possibly less so.

4.4.3 Inference

We apply the belief update algorithm to do inference. First we construct the junction tree shown in Figure 4.6. We leave out the \mathbf{x} 's in the figure because they are always known. Now the update equations (See Section 2.4.3) become

$$\Phi_t^*(y_t) = \sum_{y_{t-1}=1}^M \Psi_{t-1}(\mathbf{x}_t, y_{t-1}, y_t), \quad (4.4.7)$$

$$\Psi_t^*(\mathbf{x}_t, y_t, y_{t+1}) = \frac{\Phi_t^*(y_t)}{1} \Psi_t(\mathbf{x}_t, y_t, y_{t+1}), \quad (4.4.8)$$

for $t = 1, 2, \dots, T - 1$, and

$$\Phi_t^{**}(y_t) = \sum_{y_{t+1}=1}^M \Psi_t(\mathbf{x}_t, y_t, y_{t+1}), \quad (4.4.9)$$

$$\Psi_{t-1}^{**}(\mathbf{x}_t, y_{t-1}, y_t) = \frac{\Phi_t^{**}(y_t)}{\Phi_t^*(y_t)} \Psi_{t-1}(\mathbf{x}_t, y_{t-1}, y_t), \quad (4.4.10)$$

for $t = T, T - 1, \dots, 2$.

The inference updates thus consist of a forward pass of updates followed by a backwards pass, after which all the marginals are consistent.

4.4.4 Learning

Once again we take the general form of the derivative for CRFs with no latent variables, Equation 4.2.16,

$$\frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} = \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}) - \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{y}'_{n,C_i}} p(\mathbf{y}'_{n,C_i} | \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}) \quad (4.4.11)$$

and see that there is a potential for every time step t , and that there is one output variable y for every potential,

$$\begin{aligned} \frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} &= \sum_{n=1}^N \sum_{t=1}^{T-1} f_k(\mathbf{x}_{n,t}, y_{n,t}, y_{n,t+1}) \\ &\quad - \sum_{t=1}^{T-1} \sum_{y_t=1}^M \sum_{y_{t+1}=1}^M p(\mathbf{x}_{n,t}, y_t, y_{t+1}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,t}, y_t, y_{t+1}). \end{aligned} \quad (4.4.12)$$

4.5 Hidden CRFs

Where linear chain CRFs are a generalisation of softmax for the case where there are many output variables, HCRFs generalise softmax in another direction. Here we are still interested in one output variable, but there are hidden variables that model some latent structure in the input vectors.

HCRFs do not directly have a well-known generative counterpart, but because they are used to classify whole sequences, they can be thought of as discriminative WFSAs, where the WFSAs are trained so that there is one for each class. A sequence is classified by giving it the label of the WFSAs that gives it the highest probability.

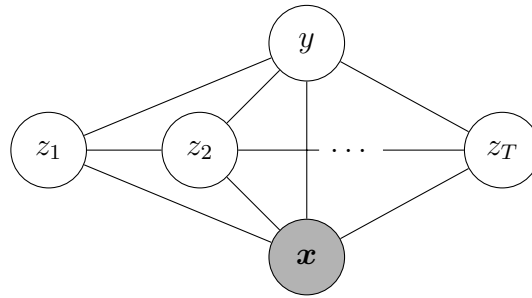


Figure 4.7: A hidden state CRF where the dependencies between the hidden variables \mathbf{z} take the form of a linear chain.

HCRFs are introduced in [67] for the problem of recognising visual gestures. For example, say we want to distinguish between a gesture beckoning someone closer and one telling someone to go further away. Each of these gestures are composed of similar poses and it would be difficult to distinguish between the two if individual frames are seen out of order. There are, however, a number of sub-gestures that can be recognised. A beckoning gesture might consist of the hidden sequence (start, away, away, closer), while go away might be (start, closer, closer, away). The latent variables thus segment the input sequence into these sub-gestures and the final class probability is found by summing over all possible sequences of these sub-gestures. For the type of HCRF discussed here, these hidden variables are of course not seen during training, so the hidden units might learn to represent something different from what we had in mind.

4.5.1 Model

In the simple case we look at, this dependence takes the form of a linear chain as can be seen in Figure 4.7. This allows us to use the fact that the same features can have different meanings depending on the order in which they occur. The conditional probability of the output variable y is

$$p(y|\mathbf{x}, \boldsymbol{\lambda}) = \sum_{\mathbf{z}} \frac{1}{Z_{\mathbf{x}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}, y, z_t, z_{t+1}, \boldsymbol{\lambda}). \quad (4.5.1)$$

The sum over all possible sequences \mathbf{z} is at first daunting but because of the sparse connections between the latent variables, the belief update algorithm calculates it efficiently.

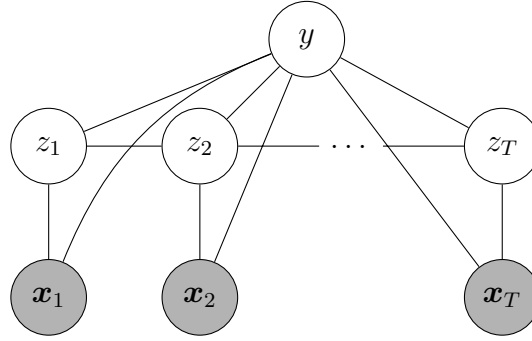


Figure 4.8: An HCRF where every hidden variable z_t only depends on the corresponding input variable \mathbf{x}_t .

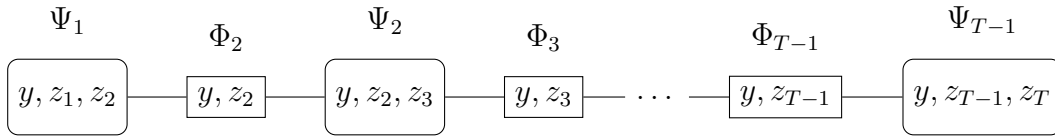


Figure 4.9: A junction tree representation of the HCRF model.

Here it is also natural in language applications to have one latent variable for each input token, and to take a window of features around the current token. In Figure 4.8 we show the case where all the features that are activated at time-step t is collected into \mathbf{x}_t .

4.5.2 Inference

To do inference, we first convert the graph to the junction tree in Figure 4.9. Then the message updates become

$$\Phi_t^*(y, z_t) = \sum_{z_{t-1}=1}^W \Psi_{t-1}(\mathbf{x}, y, z_{t-1}, z_t) \quad (4.5.2)$$

$$\Psi_t^*(\mathbf{x}, y, z_t, z_{t+1}) = \frac{\Phi_t^*(y, z_t)}{1} \Psi_t(\mathbf{x}, y, z_t, z_{t+1}) \quad (4.5.3)$$

for $t = 1, 2, \dots, T - 1$, and

$$\Phi_t^{**}(y, z_t) = \sum_{z_{t+1}=1}^W \Psi_t(\mathbf{x}, y, z_t, z_{t+1}) \quad (4.5.4)$$

$$\Psi_{t-1}^{**}(\mathbf{x}, y, z_{t-1}, z_t) = \frac{\Phi_t^{**}(y, z_t)}{\Phi_t^*(y, z_t)} \Psi_{t-1}(\mathbf{x}, y, z_{t-1}, z_t) \quad (4.5.5)$$

for $t = T, T - 1, \dots, 2$. These updates are the same as the updates for the linear chain CRF, except that the higher level variable y is propagated along, first forwards and then backwards. When the algorithm has finished the two sweeps, y can be found by marginalising the hidden variable z_j out of any of the separator potentials Φ_j , or by marginalising z_j and z_{j+1} out of any one of the clique potentials Ψ_j .

4.5.3 Learning

The general form of the derivative for models with latent variables (See Equation 4.2.19) is

$$\begin{aligned} \frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} &= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{z}_{n,C_i}} p(\mathbf{z}_{n,C_i} | \mathbf{y}_{n,C_i}, \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \mathbf{z}_{n,C_i}) \\ &\quad - \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}} p(\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i} | \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}) \end{aligned} \quad (4.5.6)$$

which becomes

$$\begin{aligned} \frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} &= \sum_{n=1}^N \sum_{t=1}^{T-1} \sum_{z_t=1}^W \sum_{z_{t+1}=1}^W p(z_t, z_{t+1} | \mathbf{x}_{n,t}, y_n, \boldsymbol{\lambda}) f_{i,k}(\mathbf{x}_{n,t}, y_n, z_t, z_{t+1}) \\ &\quad - \sum_{t=1}^{T-1} \sum_{z_t=1}^W \sum_{z_{t+1}=1}^W \sum_{y=1}^M p(y, z_t, z_{t+1} | \mathbf{x}_{n,t}, \boldsymbol{\lambda}) f_{i,k}(\mathbf{x}_{n,t}, y, z_t, z_{t+1}), \end{aligned} \quad (4.5.7)$$

when the summation over the cliques is seen to be just the summation over t — as was the case with linear-chain CRFs — and the summation over the latent variables becomes the summation over all values of z_t for every time step.

4.6 Hidden alignment CRFs

Where linear-chain CRFs and HCRFs are different ways of extending WF-SAs, *hidden alignment CRFs (HACRFs)* are the discriminative counterpart to WFSTs. The model takes two sequences that are not necessarily of the same length and classifies them into one of a number of classes.

This model was first introduced for the problem of database normalisation [46]. In a database, multiple entries may refer to the same thing. To

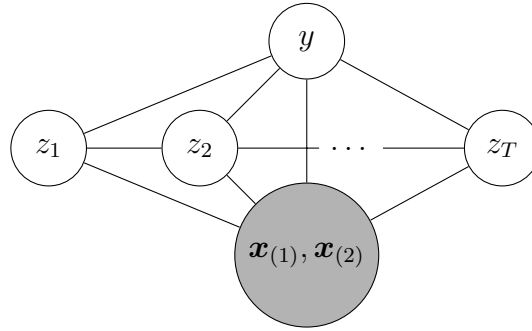


Figure 4.10: The hidden alignment CRF. The hidden variables \mathbf{z} represent edit operations on the two input sequences $\mathbf{x}_{(1)}$ and $\mathbf{x}_{(2)}$.

remove duplicates, the normaliser classifies pairs of entries as duplicates or non-duplicates. We use this model for the related problem of finding differently spelled variants of the same word.

4.6.1 The Model

An HACRF is a product of cliques, each representing an edit operation from the one input sequence $\mathbf{x}_{(1)}$ to the other sequence $\mathbf{x}_{(2)}$. For a specific sequence of states $\mathbf{z} = (z_1, \dots, z_T)$, the joint probability is

$$p(y, \mathbf{z} | \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}_{(1)}, \mathbf{x}_{(2)}, y, z_t, z_{t+1}, \boldsymbol{\lambda}). \quad (4.6.1)$$

The probability for the label of an input pair is found by summing over all possible edit sequences.

$$p(y | \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \boldsymbol{\lambda}) = \sum_{\mathbf{z}} \frac{1}{Z_{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}_{(1)}, \mathbf{x}_{(2)}, y, z_t, z_{t+1}, \boldsymbol{\lambda}), \quad (4.6.2)$$

which is graphically depicted in Figure 4.10.

Edit operations are, as for WFSTs, represented by the transitions between the states z_t . These transitions consume either a part of the input sequence $\mathbf{x}_{(1)}$, or a part of the input sequence $\mathbf{x}_{(2)}$, or both. We use the variables z to keep track of how much of each sequence has been consumed by letting $z_t = (q_t, i_t, j_t)$. Here q_t is the state in the state machine, i_t is the position in the first input sequence $\mathbf{x}_{(1)}$, and j_t is the position in the second input sequence $\mathbf{x}_{(2)}$.

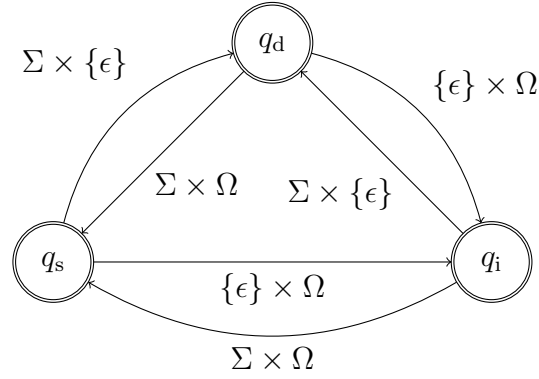


Figure 4.11: A transducer for an HACRF where Σ is the input alphabet and Ω is the set of output characters. When in state q_s , a substitution has just been completed. When in state q_d , a deletion, and in state q_i an insertion. The start state, which is not shown to simplify the drawing, has arcs to each of the other states. An emission from the start state to one of these states is similar to the other emissions coming into that state.

Although it is possible to implement a HACRF with any underlying WFST topology, we will concentrate on the WFST in Figure 4.11. This topology allows only certain transitions between z_t and z_{t+1} . For a deletion, a character a is emitted for the first sequence while no character is emitted for the second sequence, increasing i by one and leaving j as it was. So, for example, if the state machine was in $z_4 = (q_i, 2, 6)$ and a deletion occurs, it will move to $z_5 = (q_d, 3, 6)$. Similarly, if an insertion occurs between t and $t + 1$, the machine ends up in $z_{t+1} = (q_i, i_t, j_t + 1)$, while a substitution will take the machine to $z_{t+1} = (q_s, i_t + 1, j_t + 1)$. Although it is possible to add a distinct state for a character match, this topology handles a match in the same way as a substitution. By combining these two states computation can be saved, and we assume that using the same state for substitutions and matches will have less of an effect on the performance of the model than combining any of the other states. This can be investigated further in the future.

Allowable transitions are encoded by letting the potential $\Psi(\mathbf{x}, y, z_t, z_{t+1}, \boldsymbol{\lambda})$ be positive, while for invalid transitions, such as $z_{t+1} = (q_s, i_t - 2, j_t + 26)$, $\Psi(\mathbf{x}, y, z_t, z_{t+1}, \boldsymbol{\lambda}) = 0$.

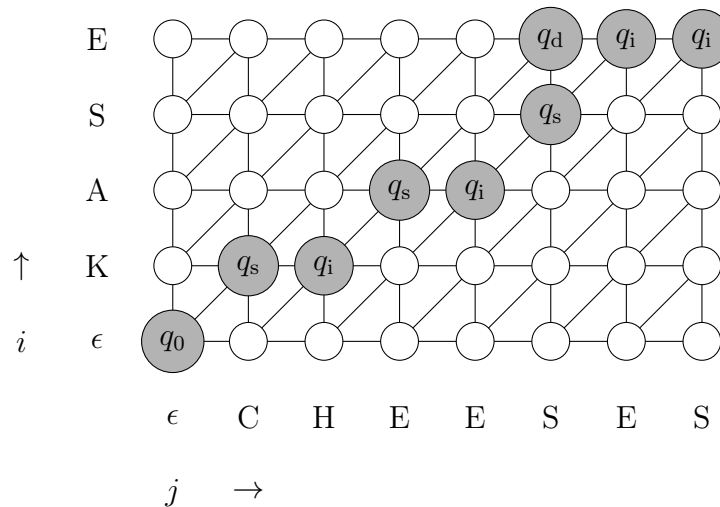


Figure 4.12: A plot of the edit-sequence (s,i,i,s,s,d,i,i) on a two-dimensional lattice. The darkened nodes represent the visited states. From any node, an upwards transition represents a deletion, a diagonal transition a substitution, and a rightwards transition an insertion.

4.6.2 Inference

This type of HCRF is more difficult to represent as a PGM than the HCRF in the previous section, because states without output symbols are possible. If we represent each edit operation with a node it is unclear even how many nodes to add. We can, however, represent it on a two-dimensional lattice.

Each edit operation changes i and j in a deterministic way. In Figure 4.12 we plot the sequence of edit operations (s,i,i,s,s,d,i,i) .

Since there is a one to one mapping between t and (i, j) , we can now change variables and work with $z_{i,j}$ instead of z_t . We denote the variable previously given by z_{t-1} as z_{i^-,j^-} and z_{t+1} as z_{i^+,j^+} .

Similarly to the CRFs discussed earlier, we partition the input vectors so that there is an input vector for each potential (See Sections 4.4.1 and 4.5.1). Here the features that are triggered at each point (i, j) in the lattice are collected into $\mathbf{x}_{(i,j)}$, and $\mathbf{x} = [\mathbf{x}_{(1,1)}^T, \mathbf{x}_{(1,2)}^T, \dots, \mathbf{x}_{(I,J)}^T]^T$ where I and J are the lengths of $\mathbf{x}_{(1)}$ and $\mathbf{x}_{(2)}$.

The message passing equation, Equation 2.4.10, can now be written in

terms of these variables as

$$\mu_{C,D}(D) = \sum_{C \setminus D} \prod_{A \in N_G(C) \setminus D} \mu_{A,C}(C) \Psi_C(C) \quad (4.6.3)$$

$$\mu_{(i,j),(i^+,j^+)}(y, z_{i^+,j^+}) = \sum_{i^-,j^-} \sum_{z_{i,j}} \mu_{(i^-,j^-),(i,j)}(z_{i,j}) \psi_{i,j}(\mathbf{x}_{(i,j)}, y, z_{i,j}, z_{i^+,j^+}) \quad (4.6.4)$$

where i^- and j^- can take on

$$(i^-, j^-) \in \{(i, j), (i-1, j), (i, j-1)\}, \quad (4.6.5)$$

which represents substitution, insertion, and deletion respectively. The backwards messages are similarly found to be

$$\mu_{(i,j),(i^-,j^-)}(y, z_{i,j}) = \sum_{i^+,j^+} \sum_{z_{i^+,j^+}} \mu_{(i^+,j^+),(i,j)}(z_{i^+,j^+}) \psi_{i,j}(\mathbf{x}_{(i,j)}, y, z_{i,j}, z_{i^+,j^+}) \quad (4.6.6)$$

where i^+ and j^+ can take on

$$(i^+, j^+) \in \{(i, j), (i+1, j), (i, j+1)\}. \quad (4.6.7)$$

The marginal of the variables in $\Psi_{i,j}$ is found by multiplying all the incoming messages and normalising:

$$p(y, z_{i,j}, z_{i^+,j^+}) = \frac{\mu_{(i^-,j^-),(i,j)}(z_{i,j}) \Psi_{i,j}(y, z_{i,j}, z_{i^+,j^+}, \mathbf{x}) \mu_{(i^+,j^+),(i,j)}(z_{i^+,j^+})}{Z_{\mathbf{x}}}. \quad (4.6.8)$$

4.6.3 Example

To illustrate the working of the model, we discuss a simple example. Say we have an HACRF that classifies character sequence pairs as “match” or “mis-match”. For example, we want “cat” and “kat” to be classified as a match, and “cat” and “eat” to be a mis-match.

A simple underlying WFST is chosen with only one state. After any edit operation, the state machine returns to this single state.

The input features $\mathbf{x}_{i,j} = [x_{(i,j),1}, x_{(i,j),2}]^T$ are two dimensional vectors based on the characters of the two input strings at position i, j . $x_{(i,j),1} = 1$ if the character in the first string at position i is the same as the character in the

second string at position j , and $x_{(i,j),1} = 0$ otherwise. $x_{(i,j),2} = 1$ if the character in the first string at position i and the character in the second string at position j are both either consonants or vowels, and zero if the one is a consonant and the other a vowel.

The feature functions are chosen to connect these vectors to the label. The input vectors for our example strings are listed in Table 4.1.

$$f_{(i,j),1}(\mathbf{x}_{i,j}, y, z_t, z_{t+1}) = \begin{cases} 1 & \text{if } y = \text{'match'} \text{ and } x_{(i,j),1} = 1 \\ 0 & \text{otherwise} \end{cases}, \quad (4.6.9)$$

$$f_{(i,j),2}(\mathbf{x}_{i,j}, y, z_t, z_{t+1}) = \begin{cases} 1 & \text{if } y = \text{'mis-match'} \text{ and } x_{(i,j),1} = 0 \\ 0 & \text{otherwise} \end{cases}, \quad (4.6.10)$$

$$f_{(i,j),3}(\mathbf{x}_{i,j}, y, z_t, z_{t+1}) = \begin{cases} 1 & \text{if } y = \text{'match'} \text{ and } x_{(i,j),2} = 1 \\ 0 & \text{otherwise} \end{cases}, \quad (4.6.11)$$

$$f_{(i,j),4}(\mathbf{x}_{i,j}, y, z_t, z_{t+1}) = \begin{cases} 1 & \text{if } y = \text{'match'} \text{ and } x_{(i,j),2} = 0 \\ 0 & \text{otherwise} \end{cases}, \quad (4.6.12)$$

i	j	cat/kat example	$\mathbf{x}_{1,(i,j)}^T$	cat/eat example	$\mathbf{x}_{1,(i,j)}^T$
1	1	ck	[0, 1]	ce	[0, 0]
1	2	ca	[0, 0]	ca	[0, 0]
1	3	ct	[0, 1]	ct	[0, 1]
2	1	ak	[0, 0]	ae	[0, 1]
2	2	aa	[1, 1]	aa	[1, 1]
2	3	at	[0, 0]	at	[0, 0]
3	1	tk	[0, 1]	te	[0, 0]
3	2	ta	[0, 0]	ta	[0, 0]
3	3	tt	[1, 1]	tt	[1, 1]

Table 4.1: Values for the input vectors for the HACRF example.

The parameter vector is chosen as $\boldsymbol{\lambda} = [2, 1, 1, 0]^T$. If many characters of the two input sequences match, the label “match” will be favoured by this choice of parameters, while many character mis-matches will give evidence of an overall mis-match. A sequence with vowels and consonants in the same order will also support the “match” label with these parameters, but more weight is attached to exact character matches.

The alignment $\begin{array}{c} \text{c a t} \\ \text{k a t} \end{array}$ corresponds to the edit sequence $\mathbf{z} = ((s, 1, 1), (s, 2, 2), (i, 2, 3), (d, 3, 3))$. We can find the probability of a match by substituting into Equation 4.6.1. So,

$$p(y = \text{match}, \mathbf{z} | \mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z_{\mathbf{x}}} \prod_{i,j} \exp\left\{\sum_{k=1}^K \lambda_k \cdot f_{(i,j),k}(\mathbf{x}_{i,j}, y = \text{match}, z_t, z_{t+1})\right\} \quad (4.6.13)$$

$$= \frac{1}{Z_{\mathbf{x}}} \exp\{2 \cdot 0 + 1 \cdot 1\} \exp\{2 \cdot 1 + 1 \cdot 1\} \exp\{2 \cdot 0 + 1 \cdot 0\} \exp\{2 \cdot 1 + 1 \cdot 1\} \quad (4.6.14)$$

$$= \frac{\exp 7}{Z_{\mathbf{x}}} \quad (4.6.15)$$

Similarly,

$$p(y = \text{mis-match}, \mathbf{z} | \mathbf{x}, \boldsymbol{\lambda}) = \frac{\exp\{2\}}{Z_{\mathbf{x}}}, \quad (4.6.16)$$

and we find that $Z_{\mathbf{x}} = \exp\{7\} + \exp\{2\}$.

We use the second pair of example strings to show how the sum over all paths is calculated. In Tables 4.2 and 4.3 the forward messages are calculated. The value in cell i, j is the message $\mu_{(i,j),(i+,j+)}(y)$ that is to be passed on to its neighbours. The values in the top-right cells represent the potentials $\Psi_{T-1}(\mathbf{x}, y = \text{match}, z_{T-1}, z_T)$ and $\Psi_{T-1}(\mathbf{x}, y = \text{mis-match}, z_{T-1}, z_T)$. The distribution over y can thus be obtained at this point by normalising these two values, so

$$p(y | \mathbf{x}) = \begin{cases} \frac{5871}{5871+262} \approx 0.96 & \text{for } y = \text{'match'}, \\ \frac{262}{5871+262} \approx 0.04 & \text{for } y = \text{'mis-match'}. \end{cases} \quad (4.6.17)$$

Our parameter choice in this case therefore leads the model to classify “cat” and “eat” as a match. It is not obvious how to set the parameters by hand to ensure the results we wanted originally, and we would have to rely on learning the parameters from training examples. Please see Section 6.4.3 for more on the application of the HACRF, Section 6.4.3.2 for more on the implementation and Section 7.1.2.2 for the results obtained with the model.

4.6.4 Learning

To learn parameters, the derivative of the log-likelihood must be calculated. By substituting into the general expression for the derivative of CRFs with

t	$2.7 \cdot e^0 = 2.7$	$(2.7 + 2.7 + 94.4) \cdot e^0 = 99.8$	$(99.8 + 94.4 + 98.1) \cdot e^3 = 5871$
a	$1 \cdot e^1 = 2.7$	$(2.7 + 1 + 1) \cdot e^3 = 94.4$	$(94.4 + 1 + 2.7) \cdot e^0 = 98.1$
c	$e^0 = 1$	$1 \cdot e^0 = 1$	$1 \cdot e^1 = 2.7$
	e	a	t

Table 4.2: Example of the forward part of a dynamic programming run for the input sequences “cat” and “eat” for the case $y = \text{match}$. The algorithm starts in the lower left cell of the table which represents “c” substituted for “e”.

t	$7.4 \cdot e^1 = 20.1$	$(20.1 + 7.4 + 17.5) \cdot e^1 = 122$	$(122 + 17.5 + 122) \cdot e^0 = 262$
a	$2.7 \cdot e^1 = 7.4$	$(7.4 + 2.7 + 7.4) \cdot e^0 = 17.5$	$(17.5 + 7.4 + 20.1) \cdot e^1 = 122$
c	$e^1 = 2.7$	$2.7 \cdot e^1 = 7.4$	$7.4 \cdot e^1 = 20.1$
	e	a	t

Table 4.3: Example of dynamic programming run for the input sequences “cat” and “eat” for the case $y = \text{mis-match}$.

hidden variables, Equation 4.2.19, the derivative is found to be

$$\begin{aligned}
 \frac{\partial \mathcal{L}'_{\mathcal{C}}}{\partial \lambda_k} &= \sum_{n=1}^N \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{z}_{n,C_i}} p(\mathbf{z}_{n,C_i} | \mathbf{y}_{n,C_i}, \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}_{n,C_i}, \mathbf{z}_{n,C_i}) \\
 &\quad - \sum_{C_i \in \mathcal{C}} \sum_{\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}} p(\mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i} | \mathbf{x}_{n,C_i}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,C_i}, \mathbf{y}'_{n,C_i}, \mathbf{z}_{n,C_i}) \quad (4.6.18) \\
 &= \sum_{n=1}^N \sum_{(i,j)} \sum_{z_{i,j}} \sum_{z_{i^+,j^+}} p(z_{i,j}, z_{i^+,j^+} | y_n, \mathbf{x}_{n,(i,j)}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,(i,j)}, y, z_{i,j}, z_{i^+,j^+}) \\
 &\quad - \sum_y \sum_{(i,j)} \sum_{z_{i,j}} \sum_{z_{i^+,j^+}} p(y, z_{i,j}, z_{i^+,j^+} | \mathbf{x}_{n,(i,j)}, \boldsymbol{\lambda}) f_k(\mathbf{x}_{n,(i,j)}, y, z_{i,j}, z_{i^+,j^+}). \quad (4.6.19)
 \end{aligned}$$

4.7 Conclusion

CRFs are introduced for the problem of classifying and labelling sequences. They often have well-known generative counterparts such as locally normalised WFSA's and WFST's. Softmax is used when we want to classify a data point into one of many classes. Linear-chain CRFs are used when we want to label each token in an input sequence. HCRFs are useful when classifying a whole sequence and we want to take the order of the input tokens into account. Finally, HACRFs can be used to classify pairs of sequences.

Chapter 5

Noisy text normalisation literature

The classical spelling correction approach and other metaphors for text normalisation are introduced. Many of the techniques fall within the noisy channel framework and can be implemented with WFSTs. We also look at CRF and unsupervised solutions to the problem.

5.1 Three metaphors

Although much research in noisy text normalisation is focused on normalising SMSs because it is considered the most difficult normalisation problem [33], there is some interest in applying the same techniques to microblog entries such as tweets [41, 28, 25]. These two media share many properties because of similar restrictions. Texts are short, usually less than 160 characters, and are often typed on restrictive mobile phone keypads.

We look at some of the SMS and microblog normalisation techniques before concentrating on the use of CRFs for this problem.

Kobus et al. identify “three metaphors for noisy text normalisation” [33]:

1. *Spelling correction* is a similar problem.
2. The problem, however, also lends itself to *statistical machine translation (SMT)*, where the noisy messages are in the source “language”, and the clean message in the target language.
3. Finally, SMS messages are sometimes described as nearer to spoken language than written language. This suggests that the problem can be tackled with *automatic speech recognition (ASR)* techniques.

Below is an overview of the literature on the use of these three metaphors for noisy text normalisation.

5.2 Spelling correction

There are two classical approaches to spelling correction. The first one uses an edit distance between words to find candidate correct words that are near the misspelled word. This method is useful for typical errors in formal language documents, where errors are usually only one character away from the correct spelling [17].

The second approach adds a language model to the edit distance so that ambiguous tokens are corrected. This model is called the *noisy channel model*.

5.2.1 Edit distance

The Levenshtein distance between two strings is the minimum number of character insertions, substitutions, and deletions to transform the first string into the second string [40]. Usually the weight of each of these transformations is chosen to be equal to one. The total cost, or distance, is the sum of the weights of the edit operations.

For example, the Levenshtein distance between “misspelling” and “mspellinq” is 4. When the two strings are aligned like

m	i	s	s	p	e	l	l	i	n	g
m	s	p	e	l	l	l	i	n	q	
	D		D				I		S	,

there are two character deletions (the D’s), one insertion (I), and one substitution (S) necessary to transform the first string into the second string.

The Levenshtein distance is representable as a WFST. The edit distance between two strings is the lowest weight of any path through the WFST in Figure 5.1 that produces those two strings. The WFST is defined on the $-\log$ -semiring so that weights add along a path.

A spelling correction system in this setting first identifies misspelled words with a dictionary lookup. The misspelled word’s edit distance from each word in the lexicon is found and the word with the smallest distance is chosen as the correction candidate. Alternatively, all candidates within a distance d of the

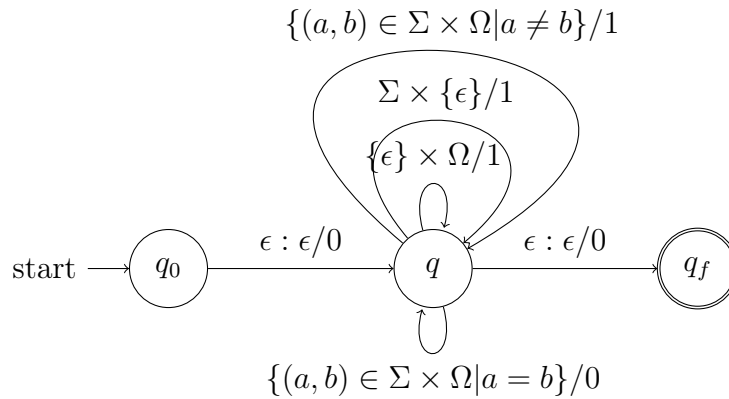


Figure 5.1: A transducer defined on the $-\log$ semiring. The Levenshtein distance between two strings is the shortest path through this transducer that produces those two strings. Here $\{(a, b) \in \Sigma \times \Omega | a \neq b\}$ denotes a substitution from any character in the first string to any other character in the second string. $\Sigma \times \{\epsilon\}$ denotes a transition where any character is produced in the first string but no character is produced in the second string. This is also called a character deletion. $\{\epsilon\} \times \Omega$ is similarly a character insertion. $\{(a, b) \in \Sigma \times \Omega | a = b\}$ denotes a matching character.

original can be created by applying all possible reverse edit transformations to the original string d times.

Edit distance is a special case of a WFST weight, which means that various generalisations are possible. Weights can be learned for the different edit operations. Different weights for different character substitutions are possible, so we might want to set the weight for exchanging “c” for “v” low because they are next to each other on a qwerty keyboard, while the weight for exchanging “z” for “p” is allowed to be large.

When the transducer is defined on the probability semiring, instead of the $-\log$ -semiring, a probabilistic interpretation is possible. Now it is not only possible to find the candidate word with the lowest cost, but rather the candidate word with the highest probability. The probability is the sum of all possible paths through the transducer, while the lowest cost is only the single lowest cost path through the transducer. A probabilistic transducer thus uses more information to give a measure of the distance between two strings than using just the edit distance.

5.2.2 Context sensitive spelling correction

Different approaches to context sensitive spelling correction have been proposed [23], but the noisy channel framework has emerged as a principled and well-understood way to tackle the problem [9]. The noisy channel model is also often applied to problems in automatic speech recognition and telecommunication.

The noisy channel model is a generative model with two components. The source model $p(\mathbf{y}|H_S)$ is a model of the source signal. In the spelling correction context this is the “intended” correctly spelled message. The channel model $p(\mathbf{x}|\mathbf{y}, H_C)$ models the corruption that occurs when the source signal \mathbf{y} is transmitted over the channel and becomes \mathbf{x} .

So if we want to find the most likely source sequence given the observed sequence, we use Bayes’ rule and drop the normalising factor to find

$$\arg \max_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}, H_C, H_S) = \arg \max_{\mathbf{y}} p(\mathbf{x}|\mathbf{y}, H_C)p(\mathbf{y}|H_S), \quad (5.2.1)$$

which is the sequence which maximises the channel model multiplied by the source model. For spelling correction, the source model is called the *language model*. N-gram language models, which are discussed in Section 5.2.4, are simple but fast and often effective enough.

If we make the assumption that errors occur independent of context, the correction problem becomes finding

$$\arg \max_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}, H_C, H_S) = \arg \max_{\mathbf{y}} \prod_{t=1}^T p(x_t|y_t, H_C)p(\mathbf{y}|H_S), \quad (5.2.2)$$

where t ranges over all the tokens in the text.

There is now thus two independent modules namely the spelling model $p(x_t|y_t, H_C)$ and the language model $p(\mathbf{y}|H_S)$.

5.2.3 Channel model

The WFST models from the previous section can now be used for the spelling model. Furthermore, different FST topologies can be used and the weights can be learned from data.

Various other channel models are proposed. Brill et al. propose a model that allows substring substitutions that are conditioned on where in the string the transformation takes place [9].

In [64], a pronunciation model is added because misspelled words are often phonetically related to the intended word.

5.2.4 Language model

A statistical language model tries to assign a probability to every sequence of words given a training corpus. With the first order Markov assumption, the probability of a word sequence $p(x_1, x_2, \dots, x_T | H_S)$ given the language model H_S can be factorised as $p(x_1 | H_S)p(x_2 | x_1, H_S)p(x_3 | x_2, H_S) \dots p(x_T | x_{T-1}, H_S)$. When the assumption is extended to higher orders, each factor in this distribution is conditioned on more of the previous tokens in the sequence. A *word N-gram* is a sequence of N consecutive words in the training corpus.

Although this type of model cannot account for the long range dependencies that are characteristic of natural language, there are a few advantages to making these Markov assumptions. One of them is that the training of the model becomes counting N-grams, which can be done quickly.

An N-gram language model can furthermore be implemented with a WFST. This means that efficient dynamic programming algorithms are available to give probabilities to sequences of words and to find the most probable sequences. Figure 5.2 shows an example of a small third-order language model.

5.2.5 Unknown words

The previous approaches rely on a fixed lexicon. In natural language, *out of vocabulary (OOV)* tokens occur. These tokens do not appear in the lexicon because they are either misspelled and should be corrected, or they are rare words — such as compound words — that are not in the dictionary. Neologisms, names, or foreign language tokens are also possible OOV tokens. To handle OOV words that are correct and thus should not be normalised, a model for these types of tokens can be constructed.

In [63], decision trees are used to classify OOV tokens as either misspellings or names. The frequency of the token in a training corpus, token length, edit distance to words in the dictionary, character N-grams, and the presence or absence of non-English characters are used as features.

In [28], an SVM classifier is trained to classify every OOV token as correctable or not based on edit distance to candidates and on the context of

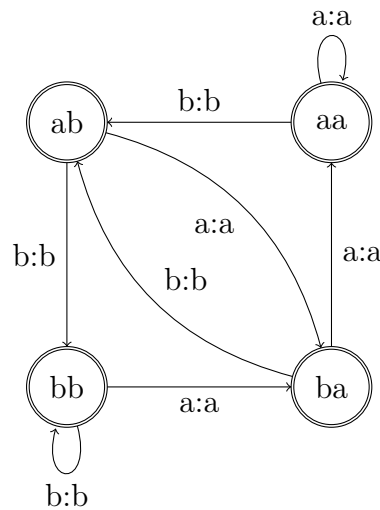


Figure 5.2: An example of a WFST that represents a trigram language model for a language with only two words namely “a” and “b”. The current state stores the two previously seen tokens. When a new token is observed, the transducer moves to the state that again represents the last two tokens. There can thus be different weights for every triplet that is observed. The WFST produces two identical strings of tokens. This assures that its composition with another WFST produces the same output but with differently scored transitions. Note that all the arcs into a certain state have the same emissions, including the arc from the start state which is not shown to simplify the drawing.

the token. If a token is classified as correctable, it is passed to a candidate generation module which generates candidate corrections.

5.2.6 Conclusion

The noisy channel model is a convenient way to break the spelling correction problem into two modules. Both modules can be modelled by WFSTs. If we denote the channel WFST as c and the language model WFST as s , the sum of the probabilities of all paths through the composition $w \circ s$ that outputs \mathbf{x} and \mathbf{y} is the probability $p(\mathbf{x}, \mathbf{y} | H_C, H_S) = p(\mathbf{x} | \mathbf{y}, H_C) p(\mathbf{y} | H_S)$.

WFSTs thus present a unifying framework in which all the modules of a spelling correction system can be implemented. To combine the modules, the composition of the transducers is taken. The highest probability path through the resulting transducer for a given input string gives the most likely output string and its probability.

5.3 Automatic speech recognition

Automatic speech recognition is another candidate problem that can be tackled with the noisy channel model implemented with WFSTs.

Where Brill et al. allowed arbitrary letter-sequence to letter-sequence conversions with a transducer c [9], Toutanova and Moore decompose the transducer further as $c = f \circ g \circ h$ [64]. f is a letter to phone transducer; g is a phone to phone transducer; and h is a phone to word transducer. This model resembles automatic speech recognition in the presence of the phone to word model.

A different use of the ASR metaphor is made by Choudhury in [11]. Here, hidden Markov models (HMMs) are trained for a few different common words. Each model has states representing a “graphemic” path and a “phonemic” path. The first accounts for symbol substitutions and deletions, while the latter accounts for homophone errors. The problem with this technique is that a different HMM must be trained for every word. Many lexical variants of a single word is thus necessary to have enough training examples.

5.4 Statistical machine translation

Statistical machine translation (SMT) techniques use data to learn text translation models. Human translated texts are used as training data. These *parallel corpora* can be supplemented with bilingual dictionaries and N-gram language models for the source and target languages. The *source language* is the language of the text that is to be translated. The *target language* is the language into which the text is to be translated.

This setting is applicable to text normalisation. The noisy input text can be thought of as being written in the source language, while in our case the target language is standard English.

Once again, the noisy channel model is used to break the problem into a language model $p(\mathbf{y}|H_S)$ and a translation model $p(\mathbf{x}|\mathbf{y}, H_C)$.

A simple word-based translation model can be implemented with a composition of WFSTs [32]. Phrase-based models, however, have proven more effective for this problem [34]. These models use a latent segmentation of the input sequence into phrases. Each segment is then translated into a phrase

in the target language. Finally, a reordering model is applied to reorder the phrases.

The reordering model is not important for lexical text normalisation, as word order is assumed to be the same for noisy and standard English. The fact that whole phrases are translated, however, allows many-to-many token transformations. This allows the correction of accidental spaces, run-on words, and incorrectly spaced words. With the usual formulation of the spelling correction problem, these types of errors are difficult to correct. ASR-type systems are also capable of correcting these types of errors.

SMT-type systems often use BLEU to measure how near the system's translations are to the human-translated gold standards. We do not go into much detail, but a BLEU score of 1.0 means the candidate translation agrees with the gold standard perfectly, and a score of 0.0 means that there is perfect disagreement.

Aw et al. use a phrase-based statistical machine translation system to translate noisy SMSs to clean SMSs [3]. They report a BLEU score of 0.58 for the raw input text, which is improved to 0.8 by their system.

A similar approach for the normalisation of tweets is proposed in [31], where a BLEU score of 0.67 for the original text is improved to 0.80.

5.5 Hybrid FST system

Various hybrid systems have been proposed that try to capture the advantages of the different approaches.

SMT systems are fast and allow arbitrary phrase substitutions but depend on a large parallel corpus and are unable to handle the creativity inherent in SMS communications [33]. A module based on WFSTs is therefore added to handle novel tokens. Kobus' hybrid system first normalises the text by using an SMT system, and then uses an ASR-like system to correct the remaining OOV tokens. For French SMSs, a BLEU score of 0.8 is obtained [33].

A hybrid SMT and WFST system is proposed in [4]. They follow much the same reasoning as Kobus et al. Here, however, two different WFST models are trained, one for in-vocabulary words, and one for OOV words.

5.6 CRF normalisation

So far, generative models for noisy text normalisation have been discussed. They often take the form of cascades of WFSTs. We now turn to discriminative approaches, which do not yet benefit from the same type of mature tools that are available for the generatively trained and locally normalised models.

In [49], a simultaneous candidate generator and scorer is proposed. A logistic regression classifier is used to classify candidate words as either matches or mismatches. It uses the transformations that turn the original word into the candidate as features. These transformations and their weights are used to generate candidates that will likely be classified as matches, without having to enumerate all the words in the lexicon.

Liu et al. use a similar approach to the one above [42]. Instead of using a logistic regression classifier, a linear chain CRF is used. Each character in the original word is tagged with either a null tag — signifying a character deletion — or another character — signifying a letter substitution. A label consisting of two or more characters — signifying character insertions — is also possible. When a noisy word is used as input, each possible sequence of tags thus gives a different candidate clean word. An N-best list of candidates can thus be generated and compared to a lexicon to find and score candidate corrections.

5.7 Unsupervised normalisation

It is expensive to put together parallel corpora of noisy and clean text, so various unsupervised and semi-supervised approaches have been proposed. Although unsupervised discriminative model training is possible, depending on how it is defined, generative models fit into the unsupervised setting much more naturally.

Contractor et al. use a noisy channel SMT system to normalise SMSs in an unsupervised way [13]. The system consists of two parts. The first is a string similarity measure that is based on the longest common subsequence between strings. This similarity measure generates candidate translations for each noisy token. These candidates are added to a list of orthographic variants. This dictionary is then used together with an N-gram language model that is trained on clean text in a standard SMT system implemented with Moses.

Gouws et al. extend this idea but concentrate on building the normalisation dictionary [24]. They first extract words that share similar context from a large corpus of text with both noisy and clean text. Pairs of tokens extracted in this way are then re-ranked according to string similarity. Using an empirically set threshold, the most highly scored pairs are added to the dictionary.

5.8 Evaluation

ASR-type approaches use word error rate (WER) to evaluate system performance, while tasks that only consider single token normalisation use precision and recall. We also consider specialised versions of precision and recall that are used in a grammatical error correction task.

5.8.1 Precision, recall, and F-measure

When the normalisation task does not change the number of tokens, precision and recall can be used. Precision in this setting is defined as the fraction of normalised words that is correctly normalised.

$$\text{Precision} = \frac{\#\text{correctly normalised}}{\#\text{normalised}} \quad (5.8.1)$$

Recall is the fraction of words that should be normalised that were correctly normalised.

$$\text{Recall} = \frac{\#\text{correctly normalised}}{\#\text{should be normalised}} \quad (5.8.2)$$

The F1-score is the normalised harmonic mean of precision and recall.

$$\text{F1} = \frac{2(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})} \quad (5.8.3)$$

Precision, recall, and the F1-score give a better impression of a classifier's performance than just the accuracy or error rate if the classes are unbalanced. For example, the accuracy for the normalisation task may be 91%, but if only 10% of the tokens should be normalised, the recall may be only 0.1.

We are also interested in a more traditional use of precision and recall. To evaluate a two-class classifier where one class represents a “positive” outcome, and the other class a “negative” outcome, precision can be defined as the fraction of positive predictions that the system gave that was actually positive.

Recall is then the fraction of all cases that should be labelled as “positive” that the classifier label correctly. In our case the outcome that a certain token-pair represents a matching pair is the positive result and the outcome that a token-pair is non-matching is negative.

5.8.2 N-best lists

N-best lists are produced by generating a list of candidates for each token in the set of test tokens and sorting the candidates from most likely to least likely. The percentage of words for which the correct candidate is in the N first positions is reported for different values of N . This evaluation can also be seen as giving the recall of the system for different precisions, but instead of plotting a precision/recall curve, the recall scores are reported in tabular form against $\frac{1}{\text{precision}}$.

5.8.3 Word error rate

When the number of tokens in the normalised and unnormalised texts are not the same, an alignment must first be done. To calculate the word error rate (WER), a word alignment that is based on the Levenshtein distance is computed between the gold standard and the candidate text. The number of token insertions, deletions, and substitutions is found. The WER is the sum of the number of these three types of errors divided by the total number of tokens.

$$\text{WER} = \frac{\#\text{insertions} + \#\text{deletions} + \#\text{substitutions}}{\#\text{tokens}} \quad (5.8.4)$$

5.8.4 Recognition and correction rates

The HOO-2012 shared task at the Building Educational Applications Workshop at NAACL 2012 uses a three way alignment between the original, gold, and candidate texts.¹ It thus combines the alignment idea of WER with precision and recall.

¹The HOO-2012@BEA evaluation definition can be found at <http://clt.mq.edu.au/research/projects/hoo/hoo2012/eval.html>

Inspired by this evaluation, we define the following measures:

$$\text{Recognition precision} = \frac{\#\text{changed at the correct place}}{\#\text{changed}}, \quad (5.8.5)$$

$$\text{Recognition recall} = \frac{\#\text{changed at the correct place}}{\#\text{should be changed}}, \quad (5.8.6)$$

$$\text{Correction precision} = \frac{\#\text{changed to correct token}}{\#\text{changed}}, \quad (5.8.7)$$

$$\text{Correction recall} = \frac{\#\text{changed to correct token}}{\#\text{should be changed}}. \quad (5.8.8)$$

The correction precision and recall give credit when the proposed corrections are correct. The recognition precision and recall, on the other hand, give credit when a correction is made where a correction should be made, even though the proposed token is not correct. These measures are thus ways of differentiating between the system's recognition of misspellings and its correction of misspellings.

5.9 Conclusion

The problem of normalising noisy web text has mostly been tackled with systems that can be informally described as being ASR-like, SMT-like, or similar to spelling correction systems.

There are, however, systems that employ discriminative models such as SVMs, logistic regression classifiers, and linear chain CRFs.

There is some research on using unsupervised systems to normalise noisy text and to extract lexical variants from noisy text. Discriminative models unfortunately do not seem to fit into this framework.

The noisy channel model is useful for breaking the spelling correction problem into a word generation module and a sentence decoding module. The word level module's performance puts an upper bound on the performance of the whole system, and research often concentrates on this module as a consequence. In Table 5.1, the N-best performance reported by different authors for this module is summarised. For this module we want to see how HACRFs compare to more traditional WFST models for Tweets. It would also be interesting to see how these proposed approaches compare to the methods listed here.

	Domain	Method	1-best	3-best	10-best	20-best
Choudhury et al. [11]	SMS	HMMs	59.9	n/a	84.3	88.7
Cook et al. [14]	SMS	WFST	59.4	n/a	83.8	87.8
Liu et al. [42]	Twitter	CRF	68.88	78.27	80.93	81.17
Pennell et al. [54]	SMS	CRF	60.4	74.6	75.6	75.6
Liu et al. [41]	Twitter	Hybrid	69.81	82.51	92.24	93.79

Table 5.1: The N-best performance of a few different approaches to the generation of candidate correct words in the literature. The state of the art hybrid system achieves a first-best rate of almost 70%.

	Method	Precision	Recall	F1
Han and Baldwin [28]	SVM	75.30	75.30	75.30
Liu et al. [41]	Hybrid+N-gram LM	84.14	78.38	81.15
Xue et al. [68]	Hybrid+N-gram LM	50.0	79.0	61.0

Table 5.2: The performance of different normalisation systems on a parallel word-aligned corpus of tweets. Note that the superior performance of Liu et al.’s candidate generator (see Table 5.1) translate into better overall system performance.

The final normalisation performance of a system is the important thing to gauge. Table 5.2 summarises some of the results in the literature specifically for tweets. They are all evaluated on a dataset of normalised tweets produced by Han and Baldwin [28]. Together with the dataset, Han and Baldwin also introduce a module that looks promising in the tweet normalisation setting. Tweets contain many tokens that are not in traditional lexicons like the Aspell word-list but which are correct. These include proper names, neologisms and common initialisms (for example, “IDK” for “I don’t know”). In this new module, tokens are classified as needing correction or not needing correction. Only the tokens to be corrected are passed to the next module.

There are thus three modules that are important in the literature, and one that looks promising.

- A tokeniser.
- A candidate word generator and scorer that produces N-best lists of candidate corrections for each token.
- A language model that takes the N-best lists, builds a lattice, rescores the lattice and decodes it to get the most probable sequence of words.

- An OOV classifier.

In the next chapter we look each of these modules in more detail and discuss how they can be integrated.

Chapter 6

Text normalisation system

A simple design for a text normalisation system is proposed in this chapter. It consists of interchangeable modules so that the performance of discriminative and generative models can be compared.

6.1 Architecture

We want to investigate discriminative models for noisy text normalisation. Although the noisy channel model is generative, it allows the problem to be decomposed into a word level module and a sentence level module. We thus want our system to also split into modules that can be developed and improved in isolation.

It would be useful to have a pipeline architecture for the system. This means that the output of one module becomes the input of the next module. One way to think of pipeline systems is by considering layers in a graphical model. There should be layers to represent the character, word, and sentence levels. Such a graphical model is presented in Figure 6.1.

Since there are loops in the model, inference can be done by either running loopy belief propagation, or converting the graph to a junction tree. We opt for the junction tree representation, because it converts the graph into a chain structure. This is the graphical model equivalent to the pipeline architecture. When inference is run, messages flow up and down the chain from one factor to another.

The first layer receives the raw input text \mathbf{x} and outputs a distribution over all tokenisations of that text. Tokens that should be corrected are then

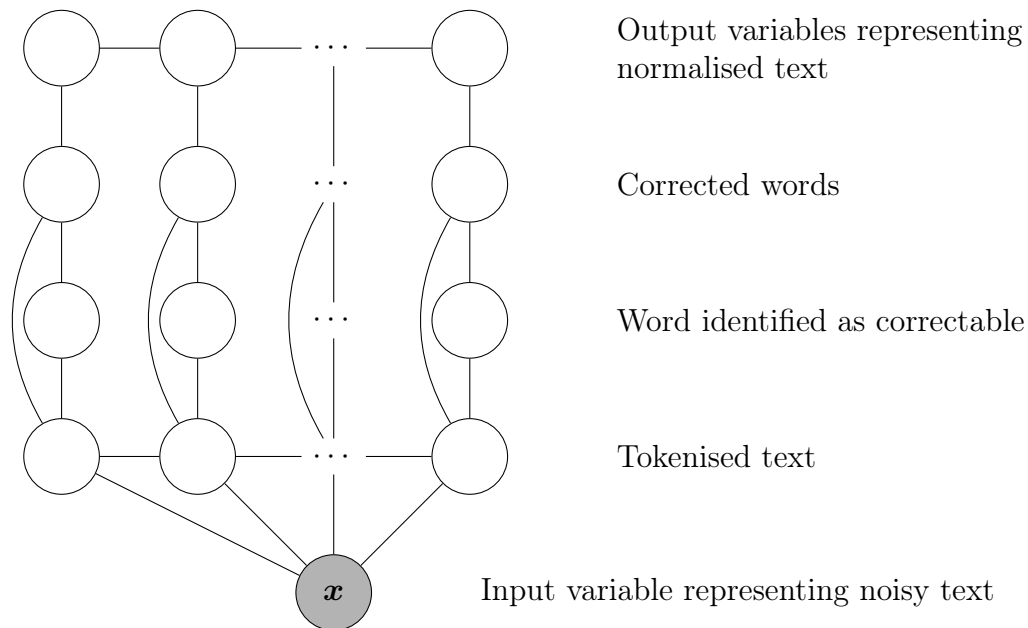


Figure 6.1: A high level graphical model of the text normalisation system. Each variable in the tokenisation layer represents a token. Every variable in the identification layer can take {should be corrected, should not be corrected}. Every variable on the correction layer represents a corrected word. Lastly, the variables on the output layer are connected to represent the dependencies when a language model is added.

identified in the second layer. The word layer then takes this message as input, along with the tokens, and outputs a distribution over candidate correct words. This last set of messages is received by the language model layer where a distribution over sentences \mathbf{y} is produced. The maximal configuration of the output variables is the most probable clean version of the text. This completes the forward pass of the inference algorithm. Inference can be stopped at this point, before the backward pass, because we are only interested in the output distribution.

Finite state machines provide a compact representation of the distribution over sequences of different length. Although FSTs would theoretically be a good way to represent the messages that flow up and down the model, the implementation of a CRF state machine library was too ambitious for the current project. Therefore we opted to use N-best lists as far as possible, in effect doing beam-search inference.

In this project we decided to implement and learn the parameters of each of

these layers separately, although joint training should be possible and should present some interesting problems. When the factors are broken up and handled separately, they become smaller graphical models. Message passing becomes doing inference on each separate model and passing its output to the next model. Below we discuss the implementation of each of these modules before describing their integration.

6.2 Tokenisation

Tokenisation is the process of dividing an input string into objects of interest. In text, these are usually distinct words and punctuation.

For tweets, we identify six different types of tokens:

1. words (for example: “cheese”),
2. punctuation (for example: “. , ! : ?”),
3. URLs (for example: “<http://cheese.com/?what>”),
4. emoticons (for example: “:D 8) :/”),
5. hashtags (for example: “#cheese”), and
6. mentions (for example: “@cheesemaker”).

We abbreviate these classes as W(ords), P(unctuation), U(RLs), E(moticons), H(ashtags), and M(entions). The class S is added to denote white space.

The task of splitting tweets along these different types of tokens is more difficult than the tokenisation of standard text, because symbols are ambiguous. For example, we want to tokenise

We’re about to win!!!:D#winning

as

We’re | about | to | win | !!! | :D | #winning

where the exclamation marks are counted as a single unit of punctuation, and the “:D” is an emoticon that also counts as a single unit. The token “#winning” is a hashtag. Hashtags are informally used in tweets to label messages so they can be grouped and searched for. The “!!!:D#w” part of the string is especially

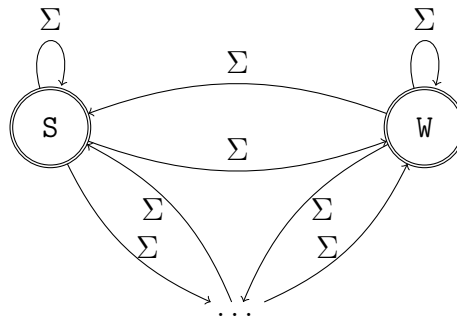


Figure 6.2: *FSA1*. A fully connected WFSA to tokenise tweets. Only the **S** and **W** states are shown. The dots represent the other states (start, **P**, **U**, **E**, **H**, and **M**), each of which is similarly connected to every other state. (The start state, of course, has only outgoing arcs.)

tricky for conventional tokenisers because of the mix of alphabetical and non alphabetical characters.

Since only words must be corrected in the system, we pass both the tokenisation and its class along to the next module. The task is therefore to tokenise the input text and simultaneously classify each token as **W**, **P**, **U**, **E**, **H**, **M**, or **S**.

For this task a dataset with 744 tweets is annotated. Each character of every message is tagged with a token type. The example above becomes:

```

W e ' r e a b o u t t o w i n ! ! ! : D # w i n n i n g
W w w w S W w w w S W w S W w w P p p E e H h h h h h h

```

The uppercase class labels signify the beginning of a new token and the lowercase class labels signify the within-token characters.

6.2.1 WFSA

A WFSA (See Section 3.1) is constructed to do the tokenisation. We train the fully connected WFSA in Figure 6.2 on 500 messages comprising 41142 characters. This is the simplest WFSA without any restriction on the order that states can occur because states are fully connected. This model cannot recognise the boundary between two tokens of the same type.

To recognise the start of a new token, the WFSA can be extended to have some state memory. Each character is now assigned either to **w**, which is the beginning of a word token, or **w**, which represents a character within a word token. Similar states for the other classes are added. Again, this is

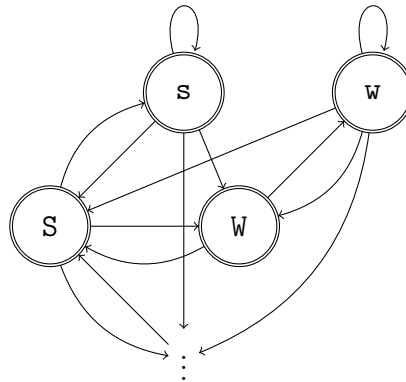


Figure 6.3: *FSA2*. A WFSA tokeniser with memory. Only the white-space and word states are shown. The dots represent the other states which is set up similarly to S and W. Every arc accepts all the characters in the input alphabet Σ , each with a different weight. The start state (not shown) is connected to each of the upper-case states with arcs similar to the other arcs coming into that state.

the simplest setup where any start-of-token state can follow any state, while within-token states can only follow corresponding start-of-token states or loop within themselves. This arrangement is shown in Figure 6.3.

For the implementation of the WFSA, Carmel, a finite state transducer package is used [26]. The training is done with its built-in EM training algorithm.

6.2.2 CRF

A linear chain CRF (See Section 4.4) is trained for the same task. Each character is assigned a label.

For comparison with the WFSA, the features for each time step t is set to the identity of the character at t . This is done by having feature functions for

every combination of symbol and label:

$$f_{t,0}(y, \mathbf{x}) = \begin{cases} 1 & \text{if } y = \text{W and } x_t = \text{'a' } \\ 0 & \text{otherwise.} \end{cases} \quad (6.2.1)$$

$$f_{t,1}(y, \mathbf{x}) = \begin{cases} 1 & \text{if } y = \text{w and } x_t = \text{'a' } \\ 0 & \text{otherwise.} \end{cases} \quad (6.2.2)$$

$$f_{t,2}(y, \mathbf{x}) = \begin{cases} 1 & \text{if } y = \text{P and } x_t = \text{'a' } \\ 0 & \text{otherwise.} \end{cases} \quad (6.2.3)$$

$$\vdots \quad (6.2.4)$$

$$f_{t,492}(y, \mathbf{x}) = \begin{cases} 1 & \text{if } y = \text{S and } x_t = \text{any other symbol} \\ 0 & \text{otherwise.} \end{cases} \quad (6.2.5)$$

$$(6.2.6)$$

With CRFs we can use features that are difficult or impossible with WFSAs. Instead of only adding feature functions for the symbols at the current time step t , we also add the characters in a window around the current character, namely $x_{t-w}, \dots, x_t, \dots, x_{t+w}$. Furthermore, the N-grams $\mathbf{x}_{[t-w:t]}, \dots, \mathbf{x}_{[t-1:t]}, \dots, \mathbf{x}_{[t:t+w]}$ are added to capture informative symbol combinations.

Three CRF tokenisers are trained with different feature sets, as is summarised in Table 6.1.

	CRF1	CRF2	CRF3
Character identity	X	X	X
Is alphanumeric		X	X
Is alphabetic		X	X
Is digit		X	X
Is uppercase		X	X
Window	0	0	2

Table 6.1: The features used in the different CRF tokenisers.

The linear chain CRFs are implemented with CRF++, an open source toolkit for training and applying linear chain CRFs [36]. A separate feature extraction module is written in Python. It takes an input string and outputs the list of features that is activated at each time step in CRF++'s input format.

6.3 Out of Vocabulary Words

A simple logistic regression classifier is trained to classify tokens as tokens that should be normalised or tokens that should be left as they are. In the ideal case, every input token's candidate words are reweighted according to how likely it is that that input token should be normalised. A better place for this module is thus later in the pipeline, after candidate words have been generated. It is, however, computationally much cheaper to prune the number of words that must be corrected sooner rather than later. For the current system we therefore decided to keep this module early in the pipeline.

To limit the number of candidate words that must be considered further, we also opted to only classify tokens that are not in a lexicon. The vocabulary is defined by a slightly modified Aspell word-list.¹ This immediately excludes the correction of in-vocabulary tokens based on context. For example, “wit” is a common misspelling of “with”, but since “wit” is also in the Aspell word-list, it will not be normalised.

The word-list is modified to exclude the word ‘u’, since it is often used as an abbreviation for ‘you’. The most common correct neologisms in tweets, including words like ‘lol’, are added to the list.

It is found that many words that are not in the word list should not be normalised. The lexicon does not include many names and almost no commonly used brand names. Neologisms and common abbreviations are also excluded. About 53% of “OOV” tokens according to the lexicon are in fact correctly spelled and should not be normalised.

The lexicon is thus incomplete. It is difficult or impossible to compile a “complete” lexicon because language changes continually and because of the large number of possible candidates. A better lexicon will definitely not harm the current system. Here, however, we start to explore a machine learning solution to the problem. A logistic regression classifier is trained on 3981 OOV tokens that are hand-annotated as either a variant of an in-vocabulary word or as a legitimate word that is not in the lexicon. Character uni-grams and bi-grams are used as features. The training examples are weighted according to their frequency in the tweet corpus. The logistic regression training tool `liblinear` is used to implement and train the model [21].

The model obtains an accuracy of 73.97% with 10-fold cross-validation.

¹Available at <ftp://ftp.gnu.org/gnu/aspell/dict/en/>

6.4 Candidate enumeration

The tokenisation layer outputs a distribution over all tokenisations. The candidate enumeration layer takes the tokenisation distribution as input and outputs a distribution over corrected words. This task of finding correctly spelled variants of the input tokens can be seen as a classification task. Each noisy token is classified as one of the words in a lexicon. The output labels are assumed to be independent of each other in this layer. The dependencies are only added in the language model layer.

For the candidate enumeration task we compare the performance of a generatively trained WFST and the discriminative HACRF.

6.4.1 Datasets

To train the WFST and HACRF, matching word pairs are needed (See Sections 3.2 and 4.6). Three datasets that contain examples of matching string pairs are used.

The first is a collection of common misspellings collected by Wikipedia users.² It contains 4438 pairs. This dataset is denoted MISSP1.

The second dataset was automatically collected from tweets by Han et al. as described in [29].³ It consists of 41181 pairs. The dataset is denoted as MISSP2.

Thirdly, Liu et al. collected 3974 noisy tokens from twitter and automatically found their clean counterparts by using search engine suggestions [42].⁴ We denote the dataset as MISSP3.

Table 6.2 shows a few example word pairs from the three datasets.

MISSP1 is thus an accurate dataset but one which we expect does not represent the typical types of spelling errors in social media such as twitter. MISSP2 on the other hand has very applicable data, but the quality is lower because of the way in which it is collected. This dataset consists of word pairs that occur in similar contexts and have a low Levenshtein distance. There are instances where a word and its plural form are present as lexical variants in the

²Available at http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines

³Available at <http://www.csse.unimelb.edu.au/~tim/etc/emnlp2012-lexnorm.tgz>

⁴Available at http://www.hlt.utdallas.edu/~yangl/data/Text_Norm_Data_Release_Fei_Liu/

MISSP1		MISSP2	
prepartion	preparation	bacame	became
villification	vilification	jusst	just
extentions	extensions	maintenan	maintenance
infiltrate	infiltrate	vallium	valium
beneficary	beneficiary	groundd	ground
resembes	resembles	whaatever	whatever
copywrite	copyright	comapny	company
ocasion	occasion	followu	follow
relinqushment	relinquishment	pplease	please
achievment	achievement	mintel	intel
MISSP3			
helloooo	hello		
freshh	fresh		
ffb	facebook		
h0me	home		
wonderin	wondering		
relaionship	relationship		
hubby	husband		
krazii	crazy		
mite	might		
tropic	topic		

Table 6.2: Examples of random matching word pairs from the three datasets. MISSP1 is the wikipedia list of common misspellings. MISSP2 is Han et al.’s automatically generated list of lexical variants, while MISSP3 is Liu et al.’s list of lexical variants.

dataset, which is not what we want. Even though MISSP3 is also automatically collected, it has both higher quality and applicability, and it is expected that it will give the best results.

An HACRF model needs positive and negative examples to train. We already have three datasets with matching word pairs, so non-matching examples must be generated somehow. There are usually only one or two lexicon words that form a matching word pair with a noisy token, but all the other words in the lexicon form mismatching word pairs with that noisy token. This means that we have a potentially huge number of examples of mismatching pairs, in the order of the lexicon size squared. Training would take very long if all of these examples are used, so we prune the negative examples by trying to add only the most informative mismatching pairs.

A scheme similar to that of McCallum et al. is used to choose negative

	MISSP1	MISSP2	MISSP3
Positive examples	4438	41181	3974
Negative examples	44380	41181	39740
Total	48818	82362	43714

Table 6.3: The sizes of the three word pair datasets. MISSP1 and MISSP3 are of similar size and MISSP2 is roughly twice as large.

examples [46]. A word in the lexicon is hypothesised to be an informative negative example if it has a low edit distance to a given noisy token, but it is known not to be the clean version of that token.

For the smaller datasets, MISSP1 and MISSP3, we generate ten negative examples for each positive example by choosing ten lexicon words with low edit distance to each of the noisy tokens in that dataset. For MISSP2, we generate only one negative example for every positive example to reduce training time. As the summary in Table 6.3 shows, the three datasets' sizes are comparable.

6.4.2 Weighted finite state transducer

Six WFSTs are trained. The topologies are shown in Figures 6.4 to 6.8. They are all extensions of the edit distance transducer (See Figure 5.1) in one way or another, with the first one being an edit distance transducer with different weights for insertions, deletions, and character matches. Substitutions are handled by doing a deletion and an insertion.

As mentioned in Section 3.2, parameters are set so that the likelihood of the training pairs is maximised. For example, when the training pair (kat, cat) is seen, we expect the parameter for a substitution from “k” to “c” to increase. The parameter values will not increase indefinitely since the parameters associated with the arcs going out of a node must sum to one and since there might be many other examples in the training data where “k” does not substitute to “c”.

Instead of using the full unicode character set as the input and output alphabets, we decided to lower the number of parameters to avoid overfitting and to limit the computation required for learning by using only lower-case symbols that are commonly used to write words. These are the alphabetical symbols “a” to “z”, and the numerical symbols “0” to “9”. Upper-case letters are converted to lower-case and all other symbols are replaced by “<OTHER>”.

6.4.2.1 Models

We aim to test some of the simplest transducer models. Firstly there are three models with only one state but with different numbers of arcs. Then there are two models where the type of the previous edit operation can influence the weight of the current operation. Lastly we set up a WFST with many states and arcs, although this model was computationally expensive to train and use.

Where WFST1 does not have a parameter for character substitutions, WFST2 and WFST3 add a substitution arc. In WFST2 the parameters are tied so that there are only four parameters. One of them is the probability of doing an insertion, one for doing a substitution, one for doing a deletion, and one for matching characters. We therefore expect that WFST2 might perform slightly better than using the Levenshtein distance because deletions is more common than insertions and should therefore have a higher probability.

WFST3, on the other hand, have different parameters for each edit operation on each symbol. The substitutions in particular can be imagined as a transition matrix where there is a different weight for every symbol turning into every other symbol. This will allow the probability of substituting “kat” for “cat” to be higher than “eat” for “cat”. Certain letters are also much more likely to be dropped or added, for example we would expect that it is more likely that the clean version of “runnin” is “running” and not “runninp”.

WFST4 adds some memory of the previous edit operation. Here there are four states. Each state represents the last edit operation type. If the WFST is currently in state q_i , it means that the last operation was an insertion. If it is in q_m , a character match has just been observed. Similarly, q_d represents character deletion and q_s represents character substitution. There are different parameters for every edit operation given that the previous operation was a match or a deletion and so forth. So the probability of deleting the letter “r” might be higher after a substitution. For example when “ever” is transduced to “eva”, the state machine is in q_s after substituting “a” for “e”. The state machine then moves to q_d and the probability of substituting “r” to nothing can be higher on that arc than “r” to nothing on the other arcs.

WFST5 is similar to WFST4, but there are now only three states. There is not a distinct state for a character match. The state q_m is subsumed under the character substitution state q_s , where a match is now a substitution to the same character.

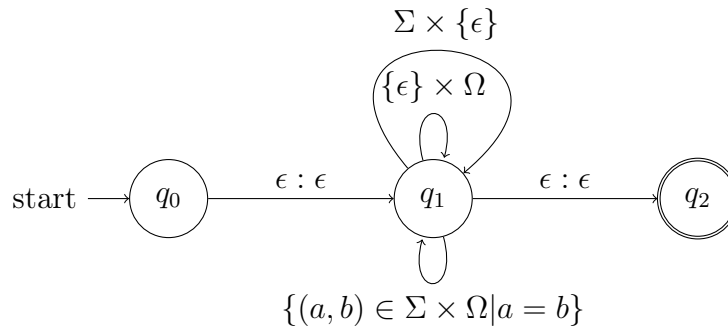


Figure 6.4: *WFST1*. A basic edit distance transducer. $\Sigma \times \{\epsilon\}$ are deletions, $\{\epsilon\} \times \Sigma$ insertions, and $\{(a, b) \in \Sigma \times \Sigma \mid a = b\}$ represent character matches.

Where *WFST4* and *WFST5* add memory of the previous edit operation's type, *WFST6* adds memory of both its type and the specific symbols involved. A state for every input character is added. When an edit operation is performed, the *WFST* ends up in the state that represents the character that is consumed in the first string. When another edit operation occurs, the *WFST* again moves to the start state. This means that there are different probabilities for any pair (or less) of input characters being transduced to any other pair (or less) of output characters.

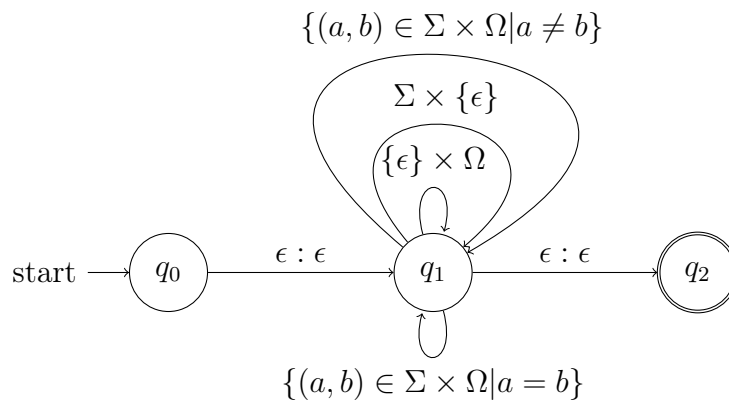


Figure 6.5: *WFST2* and *WFST3*. These two *WFSTs* are the same as *WFST1*, except that an arc, $\{(a, b) \in \Sigma \times \Omega \mid a \neq b\}$, is added to represent character substitutions. In *WFST2*, the parameters of each of these four edit operations are tied. There is therefore only four parameters. In *WFST3* on the other hand, each arc is allowed its own weight.

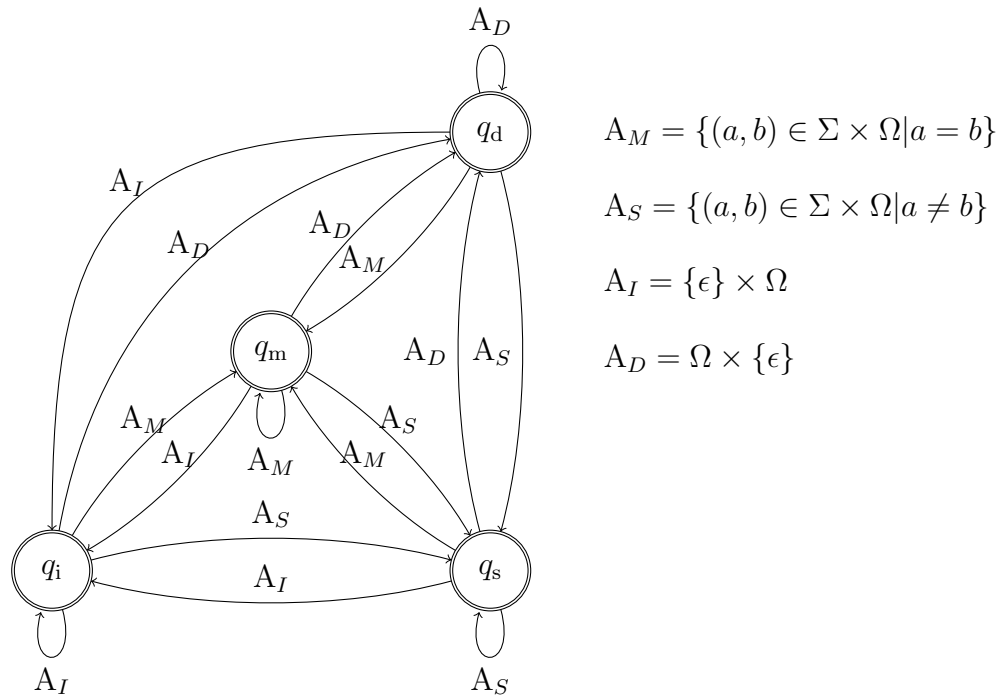


Figure 6.6: *WFST₄*. For each of the four edit operations (insertions, deletions, substitutions, and matches), a distinct state is added. This adds some memory of the previous edit operation. Every arc ending at node q_i is an insertion arc, and every arc ending at node q_m represents a character match. Arcs ending at q_d are deletions and arcs ending at q_s are substitutions. The start state (not shown), is connected similarly to each of these states.

Table 6.4 shows some typical parameters that were learned on MISSP3 to illustrate the differences between the models. *WFST₆* is omitted here because it contains too many parameters. The weights for all deletions are the same for *WFST₁* and *WFST₂*, while for the other models different weight for the deletion of different symbols are allowed. For example, the weight for the deletion of “E” in *WFST₃* is more than the weight of the deletion of “D”. This is expected, as vowels are often dropped to shorten messages [25].

The previous edit operation was an insertion if the current arc comes from q_i , a deletion if the previous state was q_d , and so forth for the other edit operations. For *WFST₄* and *WFST₅* we can therefore see that the weights for the deletion of “E” and “I” are higher if the previous edit operation was an insertion or a deletion. An example in MISSP3 where this might happen is for “h8”, a variant of “hate”. One possible alignment for this token-pair is

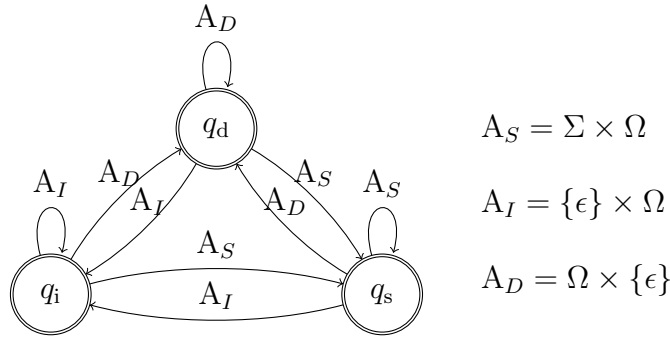


Figure 6.7: *WFST5*. This WFST is the same as *WFST4*, except that the state q_n which represents character matches is subsumed under the character substitution state q_s . This model is thus simpler than *WFST4*.

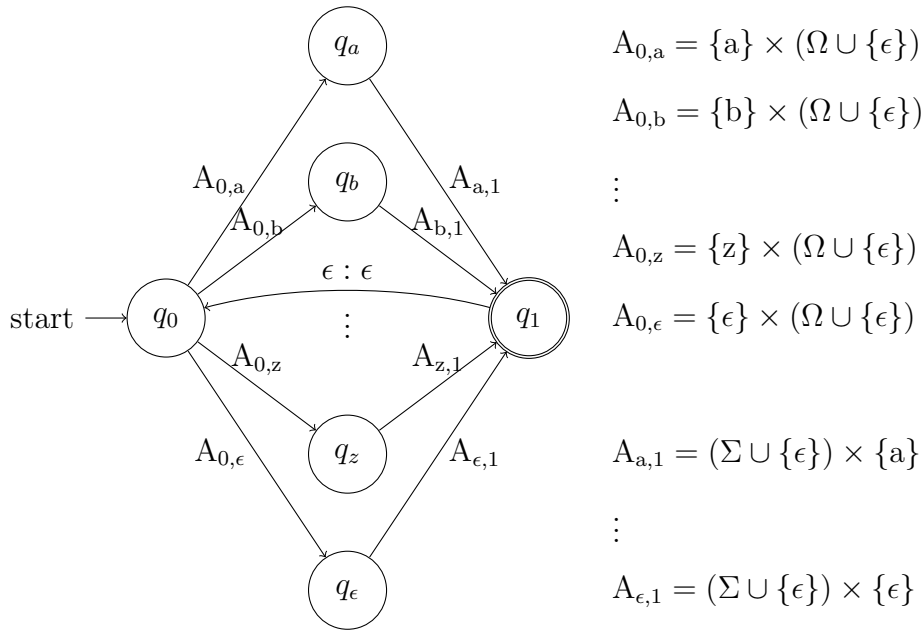


Figure 6.8: *WFST6*. *WFST6* adds memory of the symbols in the input string. When an “a” is consumed in the input string, the state machine ends up in q_a . From there an insertion, a deletion, or a substitution can be performed to go back to the start state. There are thus separate parameters for each edit operation, depending on what symbol the previous operation consumed in the input string.

	WFST1	WFST2	WFST3	WFST4				WFST6		
				q_m	q_i	q_d	q_s	q_s	q_i	q_d
D	0.226	0.224	0.199	0.087	0.023	0.293	0.031	0.098	0.013	0.439
E	0.226	0.224	0.576	0.185	0.765	0.723	0.047	0.226	0.914	0.889
F	0.226	0.224	0.084	0.025	0.055	0.035	0.030	0.024	0.113	0.062
G	0.226	0.224	0.191	0.081	0.064	0.164	0.030	0.084	0.156	0.292
H	0.226	0.224	0.210	0.069	0.141	0.329	0.033	0.059	0.368	0.541
I	0.226	0.224	0.246	0.078	0.570	0.462	0.029	0.054	0.838	0.749

Table 6.4: Some of the parameter values learned by the WFST models for *deletions* of different symbols on MISSP3. For WFST4 and WFST5, the weights of the arcs coming from states q_i , q_d , q_s , and q_m to q_d are shown. So, for example, the value in the table for symbol “E” and WFST4 q_i is 0.765. This is the weight of a deletion of “E” if the previous edit operation was an insertion. For the full table, please see Table A.6.

h a t e
h 8 ε ε, which corresponds to the WFST4 state sequence (q_m, q_s, q_d, q_d) . The last deletion therefore follows on another deletion, and the last deletion will have the relatively large weight of 0.723.

6.4.2.2 Classification

For every type of WFST, two models are trained. One is trained for the dataset of matching word pairs and another one for the dataset of non-matching word pairs. We thus have $p(\mathbf{x}|\text{match})$ and $p(\mathbf{x}|\text{mismatch})$. To classify a new word pair as either matching or non-matching, we use Bayes’ rule to find where the probability of a match is higher than the probability of a mismatch: If

$$p(\text{match}|\mathbf{x}) > p(\text{mismatch}|\mathbf{x}) \quad (6.4.1)$$

$$\frac{p(\mathbf{x}|\text{match})p(\text{match})}{p(\mathbf{x})} > \frac{p(\mathbf{x}|\text{mismatch})p(\text{mismatch})}{p(\mathbf{x})} \quad (6.4.2)$$

$$p(\mathbf{x}|\text{match})p(\text{match}) > p(\mathbf{x}|\text{mismatch})p(\text{mismatch}), \quad (6.4.3)$$

then the word pair \mathbf{x} is classified as a match, otherwise it is classified as a mismatch. The priors, $p(\text{match})$ and $p(\text{mismatch})$ are set to the proportions of datapoints in each class.

6.4.2.3 Implementation

The WFSTs described here are implemented with Carmel, an open source finite state transducer package [26]. Carmel’s EM training command is used to learn

the parameters from the example word pairs. The models are regularised and the regularisation parameter is chosen with a validation set. See Appendix A.2.1 for the results of the validation experiments.

6.4.3 Hidden alignment CRF

An HACRF as described in Section 4.6 is trained on the three datasets. The topology of its state machine is the same as that of WFST5.

6.4.3.1 Features

Different features sets are experimented with. Recall from Section 4.6 that the HACRF model is a product of edit operations

$$p(y|\mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \boldsymbol{\lambda}) = \sum_{\mathbf{z}} \frac{1}{Z_{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}}} \prod_{t=1}^{T-1} \Psi_t(\mathbf{x}_{(1)}, \mathbf{x}_{(2)}, y, z_t, z_{t+1}, \boldsymbol{\lambda}), \quad (6.4.4)$$

where each hidden variable stores the current state and the alignment of the two input strings, $z_t = (q_t, i_t, j_t)$. Each potential function can thus be a function of the input strings, the current and next states, and the current positions in the two strings.

At time step t , the character in the first string at position i_t is referred to as the *current character* in the first string, and the character in the second input string at position j_t as the current character in the second string.

Binary feature functions are chosen that try to capture the appropriateness of a certain alignment. A features is added that activates if the current character in the first string matches the current character in the second string.

A feature function is added that activates when the character before the current character in one string equals the current character in the second string, while the same is true if the strings are exchanged. This feature function thus activates when a character transposition is present at the current position in the two strings.

There is a feature function that activates when both the current characters in the two strings are vowels and one that activates when they are both consonants. Feature functions for vowel to consonant substitutions and consonant to vowel substitutions are added.

Feature	HACRF1	HACRF2	HACRF3	HACRF4
Character match	X	X	X	X
Character transposition	X	X	X	X
Characters both vowels	X	X	X	X
Characters both not vowels	X	X	X	X
Vowel to consonant substitution	X	X	X	X
Consonant to vowel substitution	X	X	X	X
Character match lookahead		X	X	X
Single character substitution matrix			X	X
Two character substitution matrix				X

Table 6.5: The different features that are implemented for the four different HACRF models.

To take a wider context into account, feature functions are added that activate when the next character in one string matches the current character in the other string.

Lastly, character specific feature functions are added. One set of these functions activates when a specific pair of characters are present. For example, when “a” is the current character in the first string and “u” is the current character in the second string, the feature function $f_{(a,u)}$ activates. Since there are feature functions for every combination of character pairs, they can be thought of as being arranged in a transition matrix.

The other set of character specific feature functions represents character pair substitutions. For example, $f_{(au,ey)}$ activates when the current character in the first string is “u” and the previous character is “a”, while the current character in the second string is “y” and its previous character is “e”.

Four different feature sets are used. The combinations are described in Table 6.5.

6.4.3.2 Implementation

The HACRF model is implemented in C++.⁵ The implementation is divided into three modules:

HACRF class: The HACRF class contains the core inference algorithms. It contains methods to load features, do inference, and return marginals and derivatives.

⁵Available at github.com/dirko/hacrf_sparse_cpp

Learning: `Learning` loads training examples, instantiates `HACRF` objects for each training example, calls their inference routines, and uses the derivatives that they return to train a model file. `Learning` calls a Python implementation of the LM-BFGS optimisation algorithm to do the optimisation. The implementation is part of the SciPy package, and can be found under `SciPy.optimize.lbfgsb.min_lbfgsb`. The resulting parameter vector is written to a file.

Interface: This module provides a command line interface to the other modules. `HACRF` objects can be instantiated, inference run on them, and the results returned. The `Learning` module is also called from `Interface`.

As is described in Section 8.3, a more specialised implementation like this one runs faster than a general CRF library, which allows time for more experimentation.

6.4.3.3 Learning

The four models are trained with an LM-BFGS optimiser. The regularisation parameter for each model is set with a validation set of 10% of the training examples. The F1-score (See Section 5.8.1) of the model on the validation set is calculated for different values of the regularisation parameter and the parameter value that gives the highest score is chosen to train the final model on all the training data. The results of these experiments can be found in Appendix A.2.2.

Figure 6.9 shows a training run with a relatively small dataset comprising 1334 example pairs. The training set is small, so the resulting model does not generalise well and there is a large gap between the training and validation accuracies. Early stopping would not be a useful regularisation technique in this case because specialisation is not causing the validation score to drop as more optimisation steps are done. We can see that the optimiser has found an optimum and that training can be stopped because the magnitude of the derivative has gone to zero.

In contrast, Figure 6.10 shows how much better the same model generalises when more training data is used. 66714 training examples are used in this case. The objective function is non-convex and therefore does not necessarily find a smooth path to a local minimum and has to backtrack often when in non-

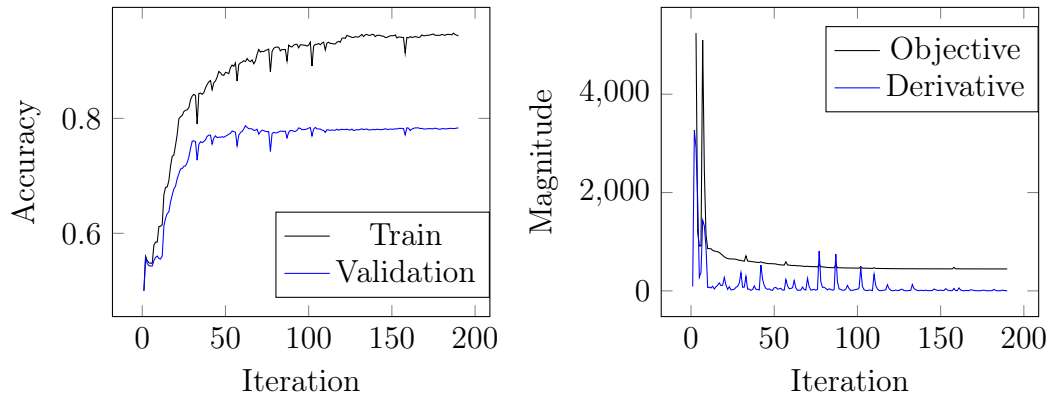


Figure 6.9: Plots of training and validation accuracies as HACRF1 is trained on MISSP2 with 1334 training examples. Note the poor generalisation shown by the large gap between the training and validation accuracies. On the right is the magnitude of the objective function and it's derivative. Note that the magnitude of the derivative goes to zero as a local optimum is approached.

convex regions (See Section 4.2.3). In this run optimisation is stopped before the derivative has gone to zero because training takes between three and fifteen hours to complete 200 iterations, depending on the number of features that are used. It seems that the point of diminishing returns on the validation set's accuracy has been reached and it is not worthwhile continuing this training run.

We use the smaller datasets to set the regularisation parameters because the smaller sets run much faster and we assume that a good regularisation value on the smaller training set will also be a good value on the full training set. To limit the training time for the training of the final model, training is stopped after 230 iterations.

6.5 Language model

The language modelling toolkit SRILM is used to implement the language model [58]. Three of its tools are used:

ngram-count Learns a language model from text. Counts N-grams in the training text and outputs a language model file.

ngram Sets up an *N-gram server*. It is a process that runs in the background and which provides fast access to the N-gram language model.

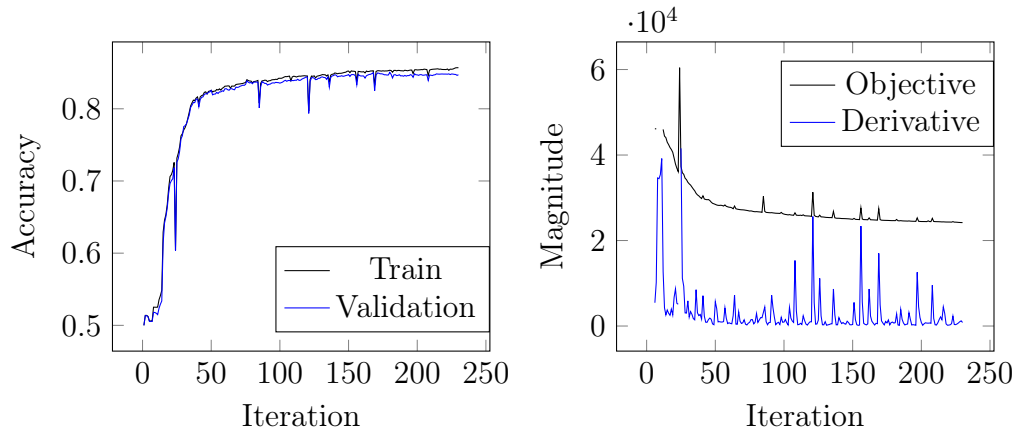


Figure 6.10: The training and validation accuracies as HACRF1 is trained on MISSP2 with 66714 training examples. To the right is the corresponding objective and its derivative's magnitudes.

lattice-tool Sets up a lattice by using lists of candidate words and the N-gram server as input. Decodes the lattice. It uses the Viterbi algorithm to find the most probable sequence of words.

Two language models are trained and then mixed to provide the final language model. The first model's training data is a collection of 68 million tweets comprising 244 million tokens. The Aspell dictionary is used as the vocabulary. All other tokens are mapped to the unknown token $\langle \text{UNK} \rangle$. To ensure that clean Twitter text is modelled, we only use messages where 80% of its tokens are lexicon words.

The second language model is of more formal language and it uses the Brown Corpus as training text [22].⁶

For both these language models, modified Kneser-Ney smoothing is used as is described by Chen and Goodman [10]. Their method of choosing the smoothing parameters is also used.

6.6 System integration

The system is built and tested in Ubuntu Linux. Some of the modules use third party software, so each module is wrapped in a Python script. These scripts provide a uniform interface to the different modules, and can be invoked by sending and receiving data either through the standard I/O, or through files.

⁶The manual is available at <http://icame.uib.no/brown/bcm.html>

The data sent and received by the different modules are the messages in the original graphical model. The graphical model takes the form of a pipeline system, so we want each of these messages to carry enough information so that the next module have enough information to work with if it only receives that message.

The messages must therefore be structured in some way. We consider two widely used open data exchange standards for structured data.

Extensible Markup Language (XML) A markup language that is suited to document data [8].

JavaScript Object Notation (JSON) A data interchange standard [15].

We decide to use JSON because it is lightweight and we find it more readable. XML is powerful, but the syntax is unnecessarily verbose for the current task.

In the tokenisation step, each input string is annotated with different tokenisations and their probabilities. In the generation step, candidate words are added. Finally, in the decoding step, the final clean text is added. Figure 6.11 shows the structure of the final JSON message of the input string “2nite!”.

6.7 Conclusion

A modular text normalisation system is presented in this chapter. It takes the form of a pipeline system, and uses the idea of a chain structured graphical model to integrate the different modules. The modules are: a tokeniser to break the input strings into parts, an OOV-classifier to identify correctable tokens, a candidate generator to find candidate corrections for each token, and a language model to find the most likely correct sequence of corrected tokens. For the tokeniser and the candidate generator a generative model based on WFSTs and a discriminative model based on CRFs are implemented. In the next chapter their performances are presented and compared.

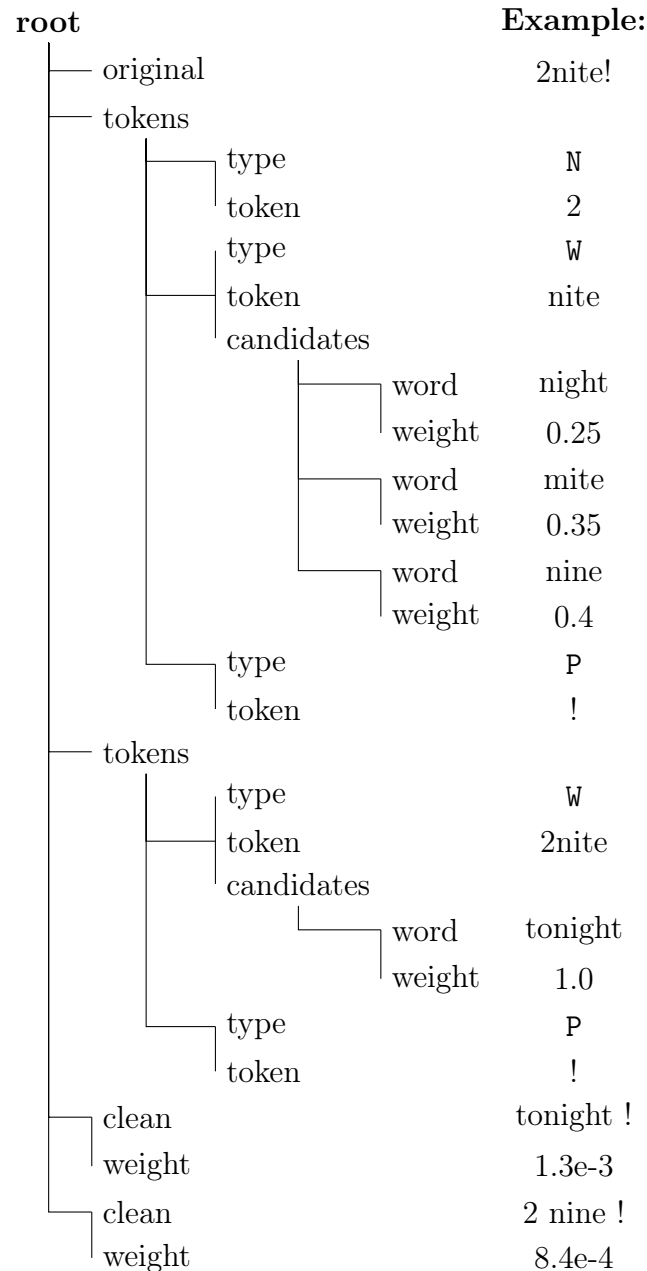


Figure 6.11: A tree showing the structure of the final output message. On the left hand side are the names of the different types of nodes in the message. On the right is an example of the contents of these nodes for the simple input string “2nite!”. The tokenisation step adds a list of tokens. Each token has the “type” and “token” attributes. In the candidate generation step, a list of candidates is added to the tokens of type W. Each candidate has a “word” and “weight” attribute. The weight is the value of the message potential for that word. Finally, during the decoding step, lists of clean versions are added, along with their potentials.

Chapter 7

Results

The system and its components' performance on different test data is presented in this chapter. We look at the performance of the individual modules to assess how well the machine learning algorithms perform. Then the datasets on which the system performance is evaluated are introduced. The end-to-end performance of the system is then looked at. Finally, the effect of text normalisation on sentiment classification is investigated.

7.1 Module performance

The different modules are trained and evaluated separately. Their individual performances are presented in this section.

7.1.1 Tokeniser

The WFSM and CRF tokenisers described in Sections 6.2.1 and 6.2.2 are tested on 101 messages with a total of 8270 characters. The performance of the classifiers are presented in Table 7.1.

It is interesting to note that the FSA with memory, FSA2 (See Figure 6.3), performs worse than the memoryless FSA1 (See Figure 6.2). FSA1 performs better on both the training and testing sets. This means that the difference between it and FSA2 is because FSA2's regularisation parameter is set to an inappropriate value. The regularisation parameters of the two models are set independently with a validation set of 143 tweets. So the regularisation parameters are set to avoid overfitting in FSA2, which brings its training set performance closer to that of FSA1. At this level of regularisation, however,

	Total labels	FSA1	FSA2	CRF1	CRF2	CRF3
Train errors	41142	451	456	168	163	12
Train error %		1.096	1.108	0.4083	0.3962	0.02917
Test errors	8270	117	147	72	63	87
Test error %		1.415	1.778	0.8706	0.7618	1.052

Table 7.1: The per-character error rates of the two FSA tokenisers and the CRF tokenisers with three different feature sets are presented here. FSA1 is a memoryless fully connected WFSA, while FSA2 differentiates between the start of a token and the within-token labels. CRF1 is comparable to FSA2, in that only the current character is used as a feature. CRF2 uses different features of the current character, while CRF3 uses different features of the current character and characters around it.

it still performs worse on the test set. The results of the experiments on the validation data to set the regularisation parameter can be found in Appendix A.1.1.

The CRF1 and FSA2 models are comparable. Both use only the current character as input feature. The discriminative CRF performs better than the generative FSA in this case. When more features of the current character are added, as in CRF2, the CRF performs even better. However, when the window is enlarged the performance drops again due to overfitting as shown by the large difference between the training and testing scores of CRF3.

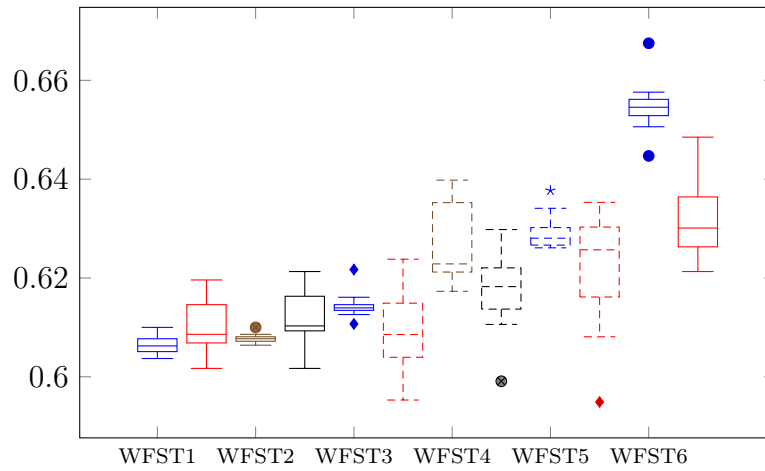
7.1.2 Generation

The candidate generation module (See Section 6.4) is trained as a classifier, but ultimately it is to be used as a scoring function. There are thus two different roles that the module must be evaluated for. Firstly we look at the classification performance by computing F1-scores (See Section 5.8.1). Secondly, the ability of each model to rank candidates is evaluated by using N-best lists (See Section 5.8.2).

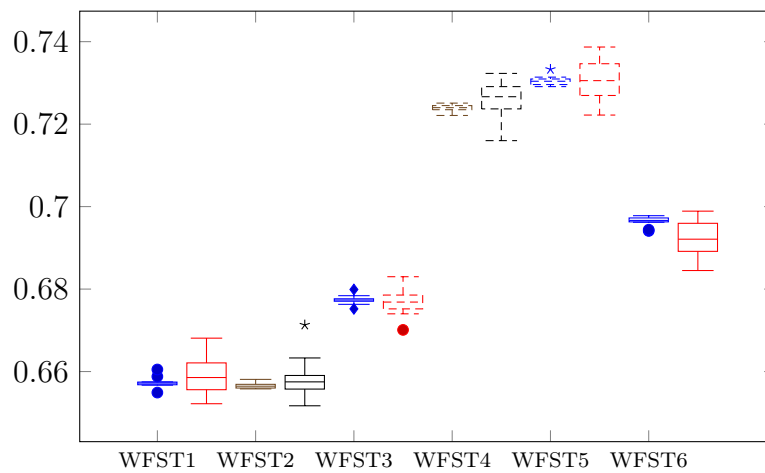
7.1.2.1 WFST

Figure 7.1 summarises the performance of the different WFSTs for the task of classifying word pairs as matches or mismatches.

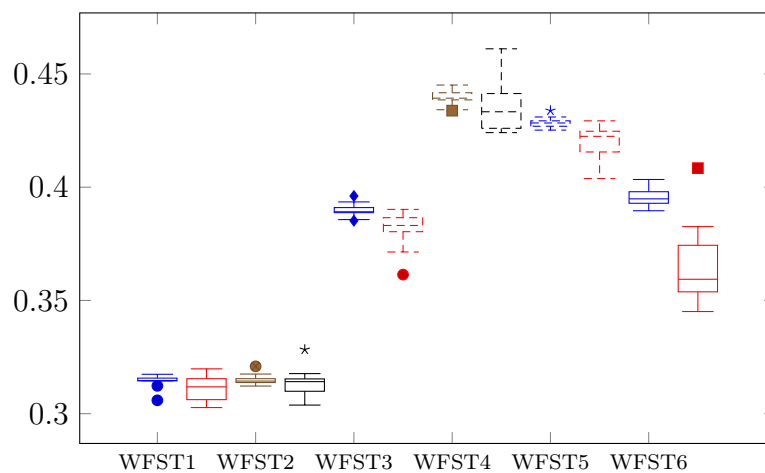
The scores on the unbalanced datasets MISSP1 and MISSP3 are lower than on MISSP2. MISSP3 does not consist mostly of pairs within one edit of each



(a) MISSP1



(b) MISSP2



(c) MISSP3

Figure 7.1: Box and whiskers plots of the F1-scores of different WFST models on the three datasets. Ten training runs on different folds of the data were done. For every model the F1-score on the training data is plotted first, followed by the test score.

other like the other two datasets, and is therefore a more difficult problem which explains the models' lower performance. The fact that the models perform better on MISSP2 than on MISSP1 might be partly because MISSP1 is unbalanced while MISSP2 is not. Further investigation is necessary to confirm this.

WFST4 and WFST5 are similar models and obtain similar scores. The simpler WFST5 even does marginally better on MISSP1 and MISSP2, although its scores vary more.

The testing scores are close to the training scores for all the models. This means that the models are not overfitting, and that the performance may be improved by using more complicated models or adding parameters.

WFST6, the transducer that models two-character edits, is more complicated but fails to improve the score. It is also not overfitting, which means that regularisation is limiting overfitting. The first row of Table A.3, reproduced here in Table 7.2, shows that the training F1-score does increase with model complexity. WFST6 does indeed overfit when there is no regularisation. When the regularisation is added, however, it performs worse than the other models perform with regularisation. The fact that it performs worse than the less powerful models means that an inappropriate regularisation parameter is chosen with the validation set because the validation set is too small, or that this type of model does not fit the data well.

Set	Reg	WFST1	WFST2	WFST3	WFST4	WFST5	WFST6
Train	0.0	0.6064	0.6069	0.6414	0.7119	0.7074	0.7637
Val	0.0	0.6248	0.6248	0.6141	0.6401	0.6446	0.6496

Table 7.2: F1-scores for the unregularised WFST models on the training and validation sets for MISSP1. Note that the models with more parameters have better training set performance.

	1-best	3-best	20-best	100-best
WFST5 on MISSP1	0.7652	0.8330	0.8691	0.8781
WFST5 on MISSP2	0.6250	0.8250	0.9425	0.9700
WFST4 on MISSP3	0.3646	0.5089	0.7165	0.8051

Table 7.3: N-best accuracy of the WFSTs.

	MISSP1	MISSP2	MISSP3
HACRF1 Train	0.950	0.861	0.815
HACRF1 Test	0.924	0.850	0.656
HACRF2 Train	0.966	0.850	0.741
HACRF2 Test	0.908	0.845	0.665
HACRF3 Train	0.953	0.867	0.814
HACRF3 Test	0.888	0.859	0.713
HACRF4 Train	0.968	0.872	0.904
HACRF4 Test	0.910	0.863	0.705

Table 7.4: The F1-scores of the different HACRF models on the training and testing data.

The N-best performance of the WFSTs that performed the best on the classification task is presented in Table 7.3. Here we omitted WFST6 in favour of WFST5 for MISSP1 because WFST6 takes impractically long to train and apply.

For MISSP3, the WFST gets only about half of the first-best rate of state of the art candidate generators (See Section 5.9). For higher values of N it compares with spell checkers for informal text that are based on the longest common subsequence.

These values are a higher bound on the recall possible in the final normalisation system. N-best lists are passed to the language model module where words are re-scored and where words that rank lower in the N-best lists might possibly be selected as the final candidate. If the correct word is not in the list, however, it is impossible to identify the correct message in the final step.

7.1.2.2 HACRF

Table 7.4 shows the classification results for the different HACRFs on the three datasets. For the simple dataset MISSP1, which contains mostly misspellings that are one edit operation from the correct word, the simplest HACRF does the best. The other HACRFs overfit on this dataset as can be seen by the large difference between the training and testing performance.

For MISSP2, the most powerful HACRF, namely HACRF4, performs the best. For MISSP3, however, it overfits and the simpler HACRF3 scores higher.

When the models are used to generate candidate words for the test tokens in each of the three datasets, the models that are trained on MISSP1 seem to

Dataset	Featureset	1-best	3-best	20-best	100-best
MISSP1	HACRF1	0.7652	0.8465	0.8691	0.8758
	HACRF2	0.6998	0.8014	0.8691	0.8736
	HACRF3	0.6885	0.8081	0.8668	0.8736
	HACRF4	0.7065	0.8375	0.8691	0.8736
MISSP2	HACRF1	0.0050	0.0275	0.1050	0.3025
	HACRF2	0.0050	0.0125	0.0725	0.2150
	HACRF3	0.0050	0.0075	0.0425	0.1175
	HACRF4	0.0025	0.0275	0.0850	0.2100
MISSP3	HACRF1	0.0000	0.0076	0.0430	0.1190
	HACRF2	0.0025	0.0101	0.0430	0.1114
	HACRF3	0.0253	0.0405	0.1367	0.2228
	HACRF4	0.0076	0.0177	0.0658	0.1519

Table 7.5: N-best performance of the HACRF models with the initial training data. Although the scores for MISSP2 and MISSP3 are so low that we might not want to draw too many conclusions, it seems as if HACRF1 does the best on Missp1 and Missp2, and that HACRF3 does the best on MISSP3.

give reasonable suggestions, while the HACRFs that are trained on the other two datasets give poor suggestions, as can be seen in Table 7.5. When the outputs of the two poorly performing HACRFs are inspected, we find that all suggestions are either two-letter words, or very long words as can be seen in Table 7.6. Even though the models seem to break on this task, the relative performance of the different feature-sets correlate with their performance on the classification task. HACRF1 performs best on MISSP1, and HACRF3 performs best on MISSP3. For MISSP2, HACRF4 did the best for the classification task but here it does slightly worse than HACRF1.

A possible reason for the failure of good performance on the classification task to carry over to the word ranking task is that the classifier is very confident about very long or very short words. It is thus possible that the magnitude of probability of a match is not a good measure of how good the match is. To test whether a distance measure is learned at all, we generated words and ranked them by looking only at the unnormalised potential of a match Z_{match} , thus disregarding the influence of the mismatch partition function. The result of this experiment can be found in Table 7.7. On MISSP2, this setup produced reasonable scores and the performance improved dramatically. The scores for MISSP1 and MISSP3, however, decreased slightly. We hypothesise that the potentials associated with the matching states do learn a reasonable distance

WFST5 on MISSP1					
meterologist	0.36	pertubation	0.4	opponant	0.51
meteorologist	0.031	permutation	1.2e-05	opponent	0.0025
neurologist	6.3e-07	probation	2.4e-06	opponents	0.00022
meteorology	2.5e-07	permutations	1e-06	opponent's	1.9e-05
meteorological	1.6e-07	operation	8.1e-07	poignant	6.3e-06
WFST5 on MISSP2					
mushroomy	0.21	cavaties	0.49	macdonalds	0.26
mushroom	0.005	cavities	0.043	macdonald	0.0062
mushroomed	0.0019	cavatina	0.01	mcdonald's	0.0021
mushrooms	0.0012	excavations	0.0017	mcdonald	0.0004
mushrooming	0.00034	caveat's	0.0017	macedonia	6.5e-06
WFST4 on MISSP3					
asz	0.057	yahh	0.24	somewhr	0.13
ask	0.0049	yah	0.007	somewhere	0.0022
asp	0.0043	yahoo	0.0033	somewhat	0.00071
ass	0.0042	yahweh	0.0019	somewhen	0.00068
ash	0.0032	yaw	0.0011	somewheres	0.00065
HACRF1 on MISSP1					
meterologist	0.52	pertubation	0.92	opponant	0.48
meteorologist	0.48	probation	0.056	opponent	0.47
urologist	0.0031	pertain	0.0096	opponents	0.033
neurologist	0.00059	permutation	0.0041	pont	0.0045
geologist	0.00037	petition	0.004	conant	0.0022
HACRF4 on MISSP2					
(mushroomy)		(cavities)		(macdonalds)	
qm	1.0	's	1.0	gm	1.0
by	1.0	ks	0.01	qm	1.0
mu	1.0	cz	0.01	rms	1.0
ms	1.0	bs	0.01	mds	1.0
mr	1.0	ms	0.01	mr	1.0
HACRF3 on MISSP3					
		(yahh)		(somewhr)	
az	0.048	ye	0.056	sr	0.01
assassinate	0.034	uv	0.04	cm	0.01
assassin	0.026	yr	0.04	sw	0.01
assassination	0.026	ia	0.035	cf	0.01
asz	0.025	uk	0.033	sd	0.01

Table 7.6: Examples of words generated by the WFST and HACRF models. Three noisy tokens from each dataset are shown. The top five candidates for each noisy token are shown along with their scores. The original token was also scored and is printed in bold. For HCRF4 and HCRF3 the original token scores lower than at least five other tokens.

		1-best	3-best	20-best	100-best
MISSP1	HACRF1	0.5530	0.6975	0.8104	0.8533
MISSP2	HACRF4	0.3800	0.5775	0.8300	0.9225
MISSP3	HACRF3	0.0025	0.0101	0.0304	0.0608

Table 7.7: N-best performance of the HACRF models with the original training data. Here word-pairs are scored by using Z_{match} in stead of $p(\text{match}) = \frac{Z_{\text{match}}}{Z_{\text{match}} + Z_{\text{mismatch}}}$. The performance for MISSP2 improves, but it is worse for the other two test sets.

measure in the case of MISSP2, but that the edit distance information is concentrated in the potentials associated with mismatching states for MISSP3. For MISSP1, both types of potentials contribute, with the matching potentials contributing more to the final probabilities. Since it is not possible to know beforehand whether useful distance information will be found in the matching or the mismatching potentials, it is not a good idea to use the partition values as distance scores. Something that can be investigated in the future is whether good distances can be found if either the matching or mismatching potentials are fixed during training. The mismatching partition, for example, can be fixed at the values that a probabilistic Levenshtein transducer would produce.

Another possible reason that good classification performance does not carry over to the ranking task is that the training data are not representative of the example pairs that occur during candidate generation. This is indeed the case, as matching training pairs are sampled according to how commonly misspelled words are distributed, while the mismatching training pairs are automatically generated. Initially, only informative negative examples are used to train the system to reduce training time. We hypothesised that training examples with low Levenshtein distance, but that are known to be mismatches, will be the most informative (See Section 6.4.1). It seems, however, that this hypothesis is false. Examples of very bad matches are never seen by the model, so it never learns to penalise word pairs with a large length difference. This will explain why only very long or very short candidates are generated.

A better policy for the generation of training examples would be to sample them from the distribution of their actual occurrence. Unfortunately, the set of positive word pairs is fixed and we do not have control over how they are sampled, and the associated distribution of negative examples is unknown.

	1-best	3-best	20-best	100-best
HACRF1 MISSP1	0.1580	0.2799	0.6366	0.8217
HACRF4 MISSP2	0.3000	0.4500	0.7475	0.9075
HACRF3 MISSP3	0.0633	0.1519	0.3924	0.6709

Table 7.8: N-best performance of the HACRF models on the revised training data. Training the models with the alternative dataset improves the scores on MISSP2 and Missp3, but the score on MISSP1 goes down.

	MISSP1	MISSP2	MISSP3
WFSTs 5, 5, and 4: Train	0.637	0.728	0.440
WFSTs 5, 5, and 4: Test	0.666	0.727	0.426
HACRFs 1, 4, and 3: Train	0.950	0.872	0.814
HACRFs 1, 4, and 3: Test	0.924	0.863	0.713

Table 7.9: The F1-scores of the best scoring WFST and HACRF for each dataset.

We therefore engineer a set of negative training examples so that it is more representative of word pairs that will be found during candidate generation. Negative examples are chosen so that 50% have low Levenshtein distance, as was the case previously. 30% are chosen uniformly from the lexicon. The lexicon consists mostly of long words, so we found it necessary to explicitly add shorter examples 20% of the time. There is thus examples of bad matches that are either too long or too short.

The result of the evaluation with this new training set is shown in Table 7.8. The performance on MISSP2 and MISSP3 improved and the performance on MISSP1 decreased. The performance is thus sensitive to which negative examples are used. Further improvements are possible by experimenting with different training sets, but a situation where this is necessary is of course not ideal.

7.1.2.3 Comparison

Table 7.9 summarises the classification performance of the classifiers. The discriminative HACRF outperform the WFSTs. The WFSTs seem to struggle with the unbalanced data especially, but even on the balanced dataset MISSP2 the HACRFs do better.

For word generation, however, the WFSTs give much more usable results

	1-best	3-best	20-best	100-best
WFST5 on MISSP1	0.7652	0.8330	0.8691	0.8781
WFST5 on MISSP2	0.6250	0.8250	0.9425	0.9700
WFST4 on MISSP3	0.3646	0.5089	0.7165	0.8051
HACRF1 on MISSP1	0.1580	0.2799	0.6366	0.8217
HACRF4 on MISSP2	0.3000	0.4500	0.7475	0.9075
HACRF3 on MISSP3	0.0633	0.1519	0.3924	0.6709

Table 7.10: The N-best accuracy of the different models on the different test sets.

as can be seen in Table 7.10.

We conclude that it is easy to get a discriminative model to work well on the classification task, and a generative model to work well for candidate generation, but not the other way around.

7.1.3 Language model

The language model as described in Section 6.5 is evaluated for different smoothing settings and different N-gram orders. KN1, KN2, and KN3 denote the first, second, and third non-interpolated Kneser-Ney (KN) smoothed N-gram language models. KNI1, KNI2, and KNI3 are their interpolated counterparts. As can be seen in Table 7.11, the difference between the training and testing perplexity increases with N-gram order. The third order model's score is, however, still the highest. Overfitting is therefore not a problem in this case. The interpolated version of KN smoothing works better than the non-interpolated version for 3-grams.

Data	KN1	KN2	KN3	KNI1	KNI2	KNI3
Train	298.409	127.108	118.513	298.203	123.708	107.699
Test	303.881	142.654	141.807	303.645	142.511	133.299

Table 7.11: Perplexity scores for different order N-gram language models for Kneser-Ney smoothing (KN), and interpolated Kneser-Ney smoothing (KNI). KNI does better than KN for higher order N-grams, because the training data is fit better without a large increase in the degree of overfitting. KNI3 has the lowest overall perplexity on the test data.

7.2 Datasets

Two datasets are available to evaluate the performance of the system. The first is a word aligned collection of 549 tweets set up by Han and Baldwin [28]. In this dataset, denoted TWEETSALIGNED, only single word substitutions are considered. We include it so that a comparison can be made with other systems.

Secondly, we normalised a collection of 2500 tweets by hand. Care is taken to avoid using the same tweets for this dataset as were used for the tokenisation and word generation datasets. The tweets are sampled from a different period in the public timeline than those of the other datasets. We denote this dataset with TWEETSUNALIGNED.

For every unknown token, we use an online slang dictionary along with the context of the unknown token to manually find the standard form of the word.

The following guidelines are used to set up this dataset:

- Since the high level goal is to train a preprocessor for other NLP tools, we try to normalise the text to the level in which these tools are trained.
- Standard contractions like “can’t” and “we’re” are kept as they are.
- Acronyms and initialism are kept as they are. This is for both formal abbreviations like CRF (conditional random field), and informal abbreviations like IDK (I don’t know).
- Tweets that are longer than 140 characters are often truncated and a hyperlink is added to a page containing the rest of the message. Sometimes a message is truncated so that a word is cut in half. These cases are treated individually. If it is clear what the truncated word should be, it is normalised. Otherwise it is left as it is.
- Slang words such as “gotta” and “wanna” are kept. The two relatively new words “finna” (going to find a) and “tryna” (trying to) are also kept. The word “Imma” (I am going to) is normalised to “I’mma”. If necessary, these slang words can be expanded easily enough later on.

Surprisingly, no words are too ambiguous to normalise in the dataset. It is of course possible that not all the ambiguities are recognised.

	Recognition		Correction		WER
	Precision	Recall	Precision	Recall	
Dictionary	0.7554	0.5491	0.6734	0.4896	0.05658
WFST5 MISSP1	0.4871	0.6770	0.1624	0.2258	0.1086
WFST5 MISSP2	0.4914	0.6785	0.1787	0.2468	0.1054
WFST4 MISSP3	0.4915	0.6805	0.2180	0.3018	0.1022
WFST5 MISSP1+OOV	0.6419	0.4070	0.2432	0.1542	0.0807
WFST5 MISSP2+OOV	0.6428	0.4063	0.3500	0.2212	0.0763
WFST4 MISSP3+OOV	0.6428	0.4061	0.3307	0.2089	0.0770

Table 7.12: Results for TWEETSUNALIGNED. The WER of the unnormalised text is 0.0760. The dictionary lookup system is the only one that corrects more errors than it produces. The OOV recognition module helps to reduce the WER but not enough to make the system viable.

7.3 System performance

The normalisation system described in the previous chapter is used to normalise the two test datasets. We use the CRF tokeniser with the lowest error to tokenise all the messages. Candidate words are then generated with the WFSTs that are trained on the three different datasets. The HACRFs are not evaluated for this task as their performance on the candidate generation task already shows that it would not be worthwhile.

The N-best lists produced by the candidate generators are then rescored with the smoothed third-order language model and decoded with the language modelling tool to find the candidate normalisations of the messages.

As a baseline, we use the word pairs used to train the word generation models as a lookup dictionary. If a token is in MISSP1, MISSP2, or MISSP3, the correct version is added to the list of candidate corrections with a weight of one. This N-best list is then passed to the language modelling module where the weights are normalised to one. The resulting lattice is rescored and decoded as usual.

The results on TWEETSUNALIGNED is presented in Table 7.12. The dictionary lookup system reduces the WER from $\frac{2671 \text{ edit operations}}{35150 \text{ total operations}} = 0.0760$ to $\frac{2325 \text{ edit operations}}{35208 \text{ total operations}} = 0.0660$.

The other systems introduce more errors than they correct. The WFSTs without the OOV recognition module cannot improve the WER since 53% of the OOV tokens should not be normalised. When dictionary candidates are generate for all OOV tokens, 53% of these tokens will thus be changed to to

	No LM		KNI3	
	Accuracy	WER	Accuracy	WER
Dictionary	0.6931	0.03440	0.7007	0.03354
WFST5 MISSP1	0.1318	0.09732	0.3136	0.07695
WFST5 MISSP2	0.2383	0.08538	0.3499	0.07287
WFST4 MISSP3	0.2020	0.08946	0.4640	0.06008

Table 7.13: Results for the TWEETSALIGNED dataset with and without a language model. The WER of the original text is 0.1121. The accuracy is the fraction of oracle OOV tokens that is correctly normalised.

something incorrect even if the correct candidates is found for the 47% that should be corrected. When the WFSTs’ correction recall (See Section 5.8.4) is compared to that of the dictionary system, it is interesting to note that more of the errors are indeed corrected by the more flexible WFSTs, as one would expect. The dictionary’s correction precision is again higher as can be expected from a system that only changes well-known lexical variants.

When the OOV classifier is added, the recognition precision goes up because tokens that should be corrected are more accurately identified. The performance of the classifier, however, is still too weak to counter the low precision of the WFSTs and thereby improve the WER. Future work should concentrate on improving the OOV-classifier.

It is interesting to note that without the OOV-classifier, the WFST that is trained on MISSP3 performs the best, while with the OOV module the WFST that is trained on MISSP2 does better. This is because the OOV-classifier throws out the difficult tokens that the MISSP2-trained generator would have struggled with anyways. The remaining tokens are thus nearer to MISSP2 than MISSP3.

The performance of the system on TWEETSALIGNED can be found in Table 7.13. Here an oracle OOV-classifier is used to show the effect of the language model on the performance of the system and the best performance the system will achieve if a perfect OOV-classifier is implemented. The language model is beneficial, more than doubling the number of corrections for the best performing WFST.

Here the dictionary does better than on TWEETSUNALIGNED. This is probably not because the dictionary has been contaminated by the popularly used test dataset TWEETSALIGNED, but because the oracle OOV-classifier is

used.

In Table 7.13 it can be seen that the model trained on MISSP3 performs the best with the addition of the language model. Without the language model, however, MISSP2 does better. This means that for the tokens that must be corrected in this dataset, the MISSP2 model gives better first-best predictions while the MISSP3 model often has the correct candidate a bit lower in the candidate list. MISSP3 is closer to the incorrect tokens in TWEETSALIGNED, but is also more difficult to learn as can be seen by the lower N-best performance in Table 7.10. The applicability and the learnability of the data are thus in opposition here.

7.4 Effect on sentiment task

The usefulness of the normalisation system as a preprocessor for some other text-mining task is evaluated. There is some interest in automatically gauging the sentiment of tweets [50]. Companies can use sentiment analysis to do market research and can supplement the use of feedback forms with sentiment tools to evaluate their products.

A simple sentiment analyser is a two-class text classification system. Every input message is classified as showing either positive or negative sentiment. A company such as Microsoft can then search for tweets containing the word “Microsoft”, automatically classify each one, and either average to roughly gauge sentiment about the company, or find the most negative or most positive tweets to see where they should concentrate their effort.

The task is first described in [51] as a positive/negative classification problem. Training data is generated by taking film reviews from www.imdb.com, an online film review database, and classifying the reviews that give a film 2.5 or more stars out of 5 as positive and less than 2.5 stars as negative. An SVM classifier is trained on the bag of words of each review and an accuracy of 82.9% is obtained.

An annotated tweet sentiment dataset is available.¹ Tweets containing the words “Microsoft”, “Apple”, “Google”, and “Twitter” are classified by hand as positive, negative, neutral, or not applicable.

¹Available at <http://www.sananalytics.com/lab/twitter-sentiment/>

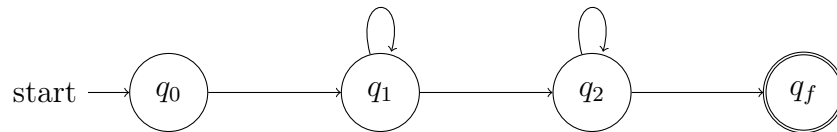


Figure 7.2: A state machine describing the allowed state transitions of the HCRF with hidden variables with a cardinality of two. The start state q_0 and end states q_f are not counted.

We hypothesise that a logistic regression (See Section 4.3) or HCRF (See Section 4.5) classifier’s error is lower if it is trained and tested on cleaned text than if it is trained and tested on the raw noisy text. This is because the data is less sparse and information can better be shared. The word “gud” might be present in the testing set but only the standard form “good” in the training data. If “gud” is normalised to “good” before classification, it should be recognised as giving evidence that the message has positive sentiment. We use the dictionary-based normaliser to automatically normalise the tweets.

An HCRF model as described in Section 4.5 is implemented with one feature per dictionary word.² When the hidden variable has a cardinality of 1, the HCRF is in effect a logistic regression classifier. When the hidden variable can take on more values, the model changes from a bag of words model to a model that can take the order of words into account. We constrain the hidden units so that the hidden variable at a certain time-step can only take on a value equal to or greater than the variable at the previous time-step, as seen in Figure 7.2.

The intuition for this choice of model is that some words convey ambiguous sentiment on their own and that their context provides the polarity of the sentiment. For example, the word “good” usually conveys positive sentiment, but when used after “not” or “never” it usually conveys negative sentiment. A model with two states would have the potential to identify regions in an input message where sentiment is negated by switching to the second state in those regions for example.

The results on the sentiment classification task is summarised in Table 7.14. The regularisation is set with a validation set of 10% of the total training examples. The HCRF with hidden units with a cardinality of one is equivalent to a logistic regression classifier, but increasing the cardinality does not have

²Available at github.com/dirko/hcrf_sparse_python.

	Hidden units	Training acc	Testing acc	#Correct (762)
Noisy tweets	0	97.07%	79.92%	609
Noisy tweets	1	96.97%	79.79%	608
Noisy tweets	2	97.22%	79.13%	603
Cleaned tweets	0	96.75%	80.45%	613
Cleaned tweets	1	96.84%	80.18%	611
Cleaned tweets	2	96.90%	80.45%	613

Table 7.14: Results of the sentiment classification task. Above, the HCRF is trained and tested on the raw input text. The number of hidden units does not have a significant effect on the performance. Below, the HCRF is trained and tested on the cleaned text. There is no significant improvement in performance. The test set consists of 762 tweets.

a significant effect on the performance. The initialisation of the HCRF parameters is probably where future work should concentrate. For the current experiment parameters are initialised uniformly randomly on $[-0.001, 0.001]$. Other intervals can be experimented with. Reasonable feature–state combinations could also be interesting. One possibility is to start the HCRF in state 0, and to initialise the parameters so that the HCRF changes to state 1 if the word “not” is observed. Sentences with negation can possibly be handled better in this way.

The performance on the cleaned tweets is slightly higher than on the original tweets. Significance is tested with McNemar’s test and the following p values are obtained: 0.265 for the HCRFs with hidden units with a cardinality of 1; 0.063 for the HCRFs with hidden units with a cardinality of 2; and 0.607 for the HCRFs with hidden units with a cardinality of 3. For more on the tests see Table A.12. The difference is thus not significant because none of the p values are lower than 0.05.

This experiment shows that the use of a dictionary based normaliser does not significantly affect the performance of our HCRF or logistic regression sentiment classifier. The dictionary normaliser has a low recall, so it is possible that a normaliser that corrects more of the tokens will affect the sentiment results. This has to be investigated further. Further experimentation is also necessary to gauge whether more sophisticated sentiment classifiers will be more sensitive to the presence of noise. Some sentiment classification systems, for example, first parse the input text before classification is done [20].

7.5 Conclusion

In this chapter we evaluated the normalisation system. First the different modules were tested individually. It is found that the CRF tokeniser works better than a comparable WFSAs tokeniser when they are trained on the same data. The CRF, however, can use features that the WFSAs cannot use and the performance is then even better.

Secondly, the candidate generation module is evaluated. For the task of classifying word-pairs as matches or mismatches, HACRFs do better than the WFSTs. For scoring and ranking token-candidate pairs, however, the WFSTs give better results.

When the different modules are arranged into a pipeline and the end-to-end performance is evaluated, it is found that the OOV-classification module is important. The WFSTs introduce more errors than they correct. With a better OOV-classifier, WER would go down as is shown on the TWEETSALIGNED test data. The addition of a language model is shown to have a significant positive effect on the classification accuracy.

Lastly, the normalisation system is used as a preprocessing step for a sentiment classifier but has negligible effect.

Chapter 8

Conclusion

In this chapter we start with a short summary of the work, followed by the main conclusions. Recommendations, especially those pertaining to the practical aspects of the work are then given. Finally, questions that arose during the project and that can be tackled in the future are discussed.

8.1 Summary

The models used in this project, namely WFSTs, logistic regression, linear-chain CRFs, HCRFs, and HACRFs are special cases of probabilistic graphical models. Inference is efficient on all these models because dynamic programming algorithms for inference exist for graphical models. The algorithms take advantage of the graph structure and the algebraic properties of semirings.

CRFs are discriminatively trained graphical models that model the conditional probability that is used for classification directly. Training is done with a Newton-Raphson type optimisation algorithm. It uses the gradient of the regularised log likelihood of the training examples with respect to the parameter vector. The gradient can be computed as the difference between the expected value of the parameters over the empirical distribution and the expected value of the parameters over the model distribution. These can be computed once inference is done and since inference is efficient, training is also efficient.

WFSMs and CRFs are implemented and experiments performed on them. The task is the lexical normalisation of Twitter messages. A pipeline architecture is used to break the task into four smaller tasks. These smaller modules

can be seen as subgraphs of a larger probabilistic graphical model, and the messages between them are the dynamic programming messages. The four modules include a tokeniser, a correct candidate generator for ill-formed tokens, an OOV-classifier to identify the tokens that must be corrected, and a language model to find the most likely candidates given the context.

Lastly, an experiment was performed to see whether an HCRF model improves the accuracy of a sentiment classifier on tweets, and whether using a lexical normaliser as a pre-processing step improves the performance of the classifier.

8.2 Conclusions

From the theoretical part of the investigation we conclude that it is worthwhile to study probabilistic graphical models and finite state machines (FSMs) together. In many cases, FSMs and PGMs describe the same models, and it is useful to have different perspective from which to look at a problem. Furthermore, for chain structured PGMs, FSMs provide a finer grained representation of the allowable transitions between the random variables in the chain as we saw with the HCRF in the sentiment experiment.

It is also interesting to note that there are often generative and discriminative versions of the same PGM structure, forming generative-discriminative pairs. It is useful to know both, since sometimes the discriminative version works better as we saw with the tokeniser, and sometimes the generative version is a more natural fit to the problem as we saw with the candidate generation.

From the tokenisation experiment, we conclude that it is possible to learn tokenisers from examples. Traditionally, rule based FSAs are used. While they are fast, it is difficult to make them flexible enough to give reasonable tokenisations in domains such as Twitter where many novel character combinations are found. It is also useful to simultaneously tokenise and classify the tokens. The type of the token can then be used further on in the pipeline. While the CRF is slower, its error rate is about half that of the WFSA.

From the word generation experiments, we conclude that HACRFs classify word pairs as either matching or non-matching better than WFSTs. This superiority does not translate into a better candidate generator, however. The

model is applied to a task for which it is not directly trained, which means that its training setup should be designed to be as near to the final task as possible. In the case of the HACRF, the word generation performance is sensitive to the negative examples that the models sees during training. Tweaking the training data to improve the performance on a different task is not ideal because it takes time and is expensive, but is unavoidable in this case.

We also showed that the direct optimisation of HACRFs is possible. Previously the model has been trained with EM. Unfortunately, time did not permit a proper comparison between these two techniques for the HACRF, but it can be investigated in the future.

For the normalisation system as a whole, the lexicon that defines the in-vocabulary and out-of-vocabulary tokens is found to be important. For the dataset we tested, more than half of the tokens that were not in the lexicon were well-formed. When the system tried to correct these, it changed them to something incorrect. More errors were thus introduced than could be corrected even with a perfect candidate generation module. To counter this, a better lexicon is thus necessary, or another way of identifying the tokens that must be corrected and the tokens that must be left alone. We started to experiment with a classifier that uses the character N-grams as features to solve this problem.

Except for the OOV module, the performance of the candidate generation module provides an upper bound on the performance of the system as a whole. Improvements here will therefore translate into better system performance as well.

From the fact that the systems that used the WFST trained on MISSP3 and MISSP2 fared better than those trained on MISSP1 we conclude that in-domain training data is better than other data, which is not surprising. MISSP3 did not always produce better results than MISSP2, although MISSP2 was gathered using a cruder mechanism. This is probably because the OOV-classifier only passed the “easier” tokens to the generation step, and the MISSP3 examples are more difficult to learn.

The language model is shown to re-score candidate tokens effectively. The accuracy of the best performing WFST doubled with the addition of the language model.

The addition of hidden units and the normalisation of the text did not significantly affect the output in the sentiment analysis experiment. We ex-

pect that the training accuracy at least will increase as the model complexity increases with the addition of the hidden states. The fact that this does not happen might be because of the way that the parameters are initialised. With better initialisation the training might be able to find a better local optimum. Further investigation is necessary here.

Although normalisation did not improve the classification rate, the scores also did not go down. Further investigation is necessary to see whether a statistically significant increase in accuracy can be obtained with a better normaliser or with more data.

8.3 Recommendations

We briefly look at some practical recommendation sprouting from the implementation of the system.

During the course of the project, an inference library for general discrete CRFs was implemented. It is, however, not used for any of the experiments because:

- the specialised logistic regression and linear-chain CRF packages that are already available are much more efficient and run orders of magnitude faster,
- any other more complicated model would run even slower because an approximate method such as loopy belief propagation would have to be used. The approach taken in this project was rather to put simpler models together than build one big model.
- The inference algorithms that were implemented are not general enough to implement an HACRF.

So the CRF inference library implementation is too general to be fast, but not general enough to be able to implement an HACRF. It was then decided to rather concentrate on building practical HACRF software.

The computational overhead involved in having a general CRF system is high at this stage. It might become worthwhile once computational power increases, or when a highly optimised package becomes available. For WFSTs, this has already happened and such packages are available and general operation are fast.

For a project such as this where different software packages are used to implement a larger system, it seems that a large part of the code will inevitably be scripting glue to process the input and output of the modules. These scripts are difficult to maintain and document. What we found to be helpful in this case is to implement the input and output data processing in Python, and the whole pipeline in a Bash script. The output of each intermediate step is stored in a different file so that an experiment can be resumed at any point. Every experiment can then live in its own directory and the Bash script provides some documentation as to what happened.

It is also helpful to have a uniform interface for as many of the scripts and other modules as possible. This reduces the need for documentation. The `liblinear` [21] (part of `libsvm`) package's interface, which we found easy to use and understand, is used as a model for our implementations of HACRFs and HCRFs.

For the candidate generation module, none of the models that we tested did as well as the state-of-the-art hybrid systems. We did find that a WFST-type model is a better starting point if a high performance system is to be implemented. From the literature it is clear that models that use longest common subsequence information, together with lexical and phonemic information will have the highest first-best accuracies (See Section 5.9). Practical text normalisation systems should start with one of these or a combination of them.

8.4 Future Work

There are numerous ways in which the system can be improved:

- For the tokeniser, and indeed for any of the CRF models, better features can always be investigated. For the tokenisation, it seems as if taking a bigger window and also higher order character N-grams leads to overfitting. Restricting it to just character bigrams would possibly be better. It would also be interesting to implement a feature to check for matching brackets and quotes.
- Together with this, feature selection with ℓ_1 regularisation can be investigated for all the CRF models.

- There are a few other typical errors that the tokeniser made that can be remedied with more training data.
- The OOV-classification module needs some work before it will work well. Again, better features can be experimented with. In [28], the word-context is also added. To incorporate context into the model here and also later on in the language model does not seem elegant. A better open-vocabulary language model that breaks OOV tokens into smaller parts is therefore a better option. Parts of speech and named entity recognition can then also be incorporated.
- The error measure used for the OOV-classifier can be changed. At the moment, the false accept and false reject rates on the training data are the same. The system would work better if tokens are only identified as correctable when the system's confidence is high, thus improving the recognition precision, lowering the recall, but improving the WER. A more Bayesian integration of the modules will be beneficial in this case.
- The generation of negative examples for the HACRF candidate generator must be further investigated. If a model of the sampling of matching word-pairs can be constructed, reasonable non-matching pairs can then be sampled from the list of all non-matching word-pairs.
- The dictionary normaliser is found to give good results, but it is not capable of correcting novel tokens like the edit-distance type normalisers. A hybrid system as is proposed in [24] may therefore give the best of both.

Other than improving the current system, other extensions are possible:

- The system can be set up to work in an unsupervised way. An EM-type iteration will take noisy text and correct it with a normaliser that is initialised with reasonable parameters. This might lead to further correction becoming likely according to the language model, so the word-level normaliser's parameters can be updated. This in effect builds a model of the channel when large amounts of noisy and clean data are available.
- There is a need for a CRF package that can handle deletions and insertions. With such a package one would be able to implement both

linear-chain CRFs and HACRFs, and maybe combine them as one can do with WFSA's and WFSTs. Software available at the moment can only use CRFs to do labelling, while CRFs can be used much like WFSTs.

- Since the whole system can be thought of a graphical model, joint training of all the modules are possible. Although this would be slow, it would eliminate many of the problems we had with training and testing conditions not being the same.

Finally, some question came up that we did not have enough time to explore:

- How can HCRFs be effectively used for sentiment classification? How can they be initialised or trained to give better results?
- Is it better to train HACRFs with EM or by directly using the gradient?
- It seems that sentiment classification with a bag-of-words type model is robust against noise. What types of NLP tasks are negatively influenced by noise and how tolerant are they to informal language usage?

The investigation of informal and noisy text is a dynamic and interesting field. PGMs and FSMs provide a framework into which many of the myriad existing techniques to handle noisy text fit, and they serve as a good starting point for future work.

Appendices

Appendix A

Results

A.1 Tokeniser

A.1.1 WFSA

Reg	FST1		FST2	
	Train	Val	Train	Val
0.0	0.0109	0.0178	0.0175	0.0360
0.001	0.0141	0.0252	0.0176	0.0295
0.00178	0.0141	0.0247	0.0176	0.0273
0.00316	0.0141	0.0240	0.0175	0.0273
0.00562	0.0137	0.0235	0.0171	0.0303
0.01	0.0132	0.0231	0.0148	0.0278
0.0178	0.0133	0.0230	0.0111	0.0258
0.0316	0.0128	0.0227	0.0102	0.0266
0.0562	0.0120	0.0228	0.0108	0.0269
0.1	0.0103	0.0163	0.0108	0.0282
0.178	0.0110	0.0162	0.0113	0.0304
0.316	0.0110	0.0180	0.0149	0.0316
0.562	0.0140	0.0201	0.0217	0.0346
1.0	0.0198	0.0274	0.0316	0.0451
1.78	0.0324	0.0358	0.0452	0.0585
3.16	0.0442	0.0540	0.0814	0.0898
5.62	0.0799	0.0830	0.1209	0.1160

Table A.1: Error rates on the training and validation sets to find the regularisation rate for FSA1 and FSA2 for the tokenisation task.

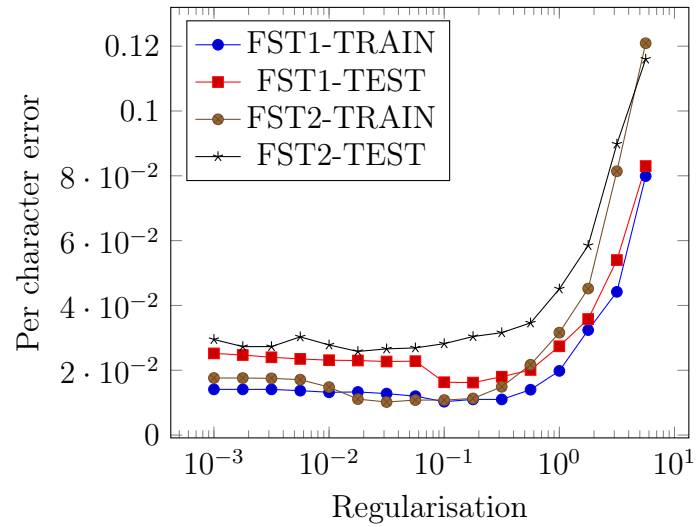


Figure A.1: Plots of the error rates on the training and validation sets to find the regularisation rate for FSA1 and FSA2 for the tokenisation task.

A.1.2 CRF

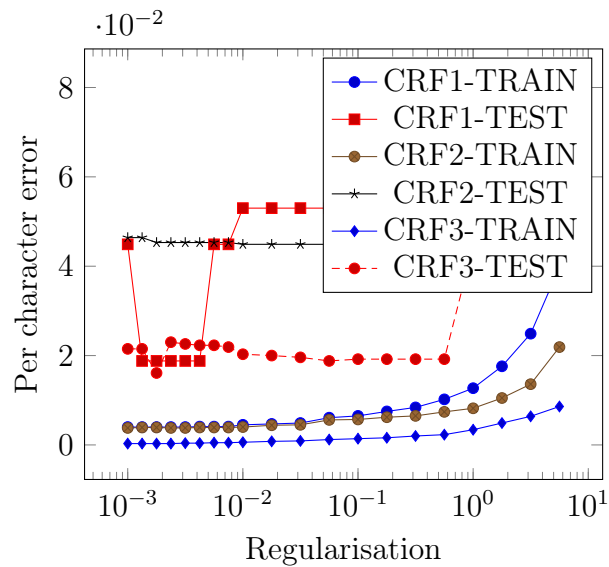


Figure A.2: Plots of the error rates on the training and validation sets to find the regularisation rate for CRF1, CRF2 and CRF3 for the tokenisation task.

Reg	CRF1		CRF2		CRF3	
	Train	Val	Train	Val	Train	Val
0.001	0.0040	0.0449	0.0038	0.0464	0.0003	0.0215
0.001333	0.0040	0.0188	0.0039	0.0464	0.0003	0.0215
0.00178	0.0040	0.0188	0.0039	0.0453	0.0003	0.0161
0.002371	0.0040	0.0188	0.0038	0.0453	0.0003	0.0230
0.00316	0.0040	0.0188	0.0038	0.0453	0.0004	0.0226
0.004217	0.0041	0.0188	0.0039	0.0453	0.0004	0.0223
0.00562	0.0041	0.0449	0.0039	0.0453	0.0005	0.0223
0.007499	0.0041	0.0449	0.0039	0.0453	0.0005	0.0219
0.01	0.0045	0.0530	0.0040	0.0449	0.0006	0.0203
0.0178	0.0047	0.0530	0.0044	0.0449	0.0008	0.0200
0.0316	0.0049	0.0530	0.0045	0.0449	0.0009	0.0196
0.0562	0.0061	0.0530	0.0056	0.0449	0.0012	0.0188
0.1	0.0065	0.0537	0.0057	0.0545	0.0014	0.0192
0.178	0.0075	0.0541	0.0062	0.0545	0.0016	0.0192
0.316	0.0084	0.0541	0.0065	0.0491	0.0020	0.0192
0.562	0.0102	0.0549	0.0074	0.0491	0.0023	0.0192
1.0	0.0127	0.0572	0.0082	0.0491	0.0034	0.0445
1.78	0.0176	0.0603	0.0105	0.0499	0.0049	0.0445
3.16	0.0249	0.0691	0.0136	0.0553	0.0064	0.0518
5.62	0.0398	0.0806	0.0219	0.0630	0.0086	0.0530

Table A.2: Error rates on the training and validation sets to find the regularisation rate for CRF1, CRF2, and CRF3 for the tokenisation task. When different regularisations scores the same, a regularisation value in the middle is chosen.

A.2 Generator

A.2.1 WFST

Set	Reg	FST1	FST2	FST3	FST4	FST5	FST6
Train	0.01	0.6064	0.6064	0.6414	0.7022	0.7021	0.7741
Val	0.01	0.6248	0.6248	0.6145	0.6310	0.6407	0.6518
Train	0.03	0.6064	0.6064	0.6402	0.7013	0.7001	0.7715
Val	0.03	0.6248	0.6248	0.6157	0.6271	0.6488	0.6462
Train	0.1	0.6064	0.6064	0.6383	0.6937	0.7001	0.7668
Val	0.1	0.6248	0.6242	0.6147	0.6207	0.6448	0.6509
Train	0.3	0.6064	0.5979	0.6340	0.6743	0.6836	0.7282
Val	0.3	0.6248	0.6153	0.6100	0.5948	0.6269	0.6232
Train	1.0	0.6064	0.5743	0.6105	0.6523	0.6512	0.6583
Val	1.0	0.6242	0.6007	0.5875	0.5908	0.5904	0.6005
Train	3.0	0.6034	0.4992	0.5975	0.6563	0.6316	0.5758
Val	3.0	0.6228	0.5277	0.5893	0.5942	0.5779	0.5396
Train	10.0	0.5792	0.3535	0.5243	0.6437	0.5717	0.3924
Val	10.0	0.5969	0.3608	0.5161	0.6056	0.5294	0.3713
Train	30.0	0.5508	0.2232	0.4050	0.3906	0.2906	0.2788
Val	30.0	0.5668	0.2246	0.3921	0.3710	0.2857	0.2803
Train	100.0	0.4731	0.1761	0.2420	0.2481	0.2027	0.1959
Val	100.0	0.4804	0.1793	0.2424	0.2458	0.2038	0.1958

Table A.3: F1-scores of the different WFSTs on the training and validation sets for MISSP1. When different regularisations get the same score, the higher middle value is chosen.

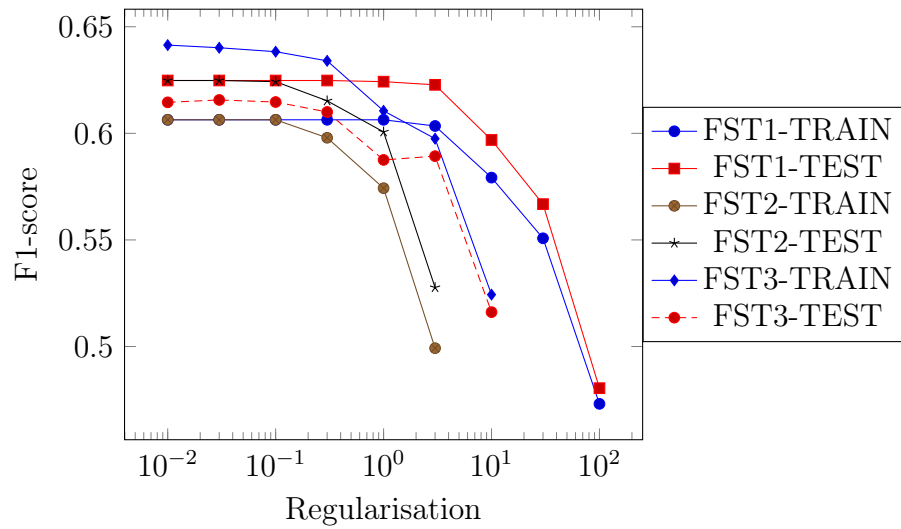


Figure A.3: Plots of the F1-scores of the different WFSs on the training and validation sets for MISSP1.

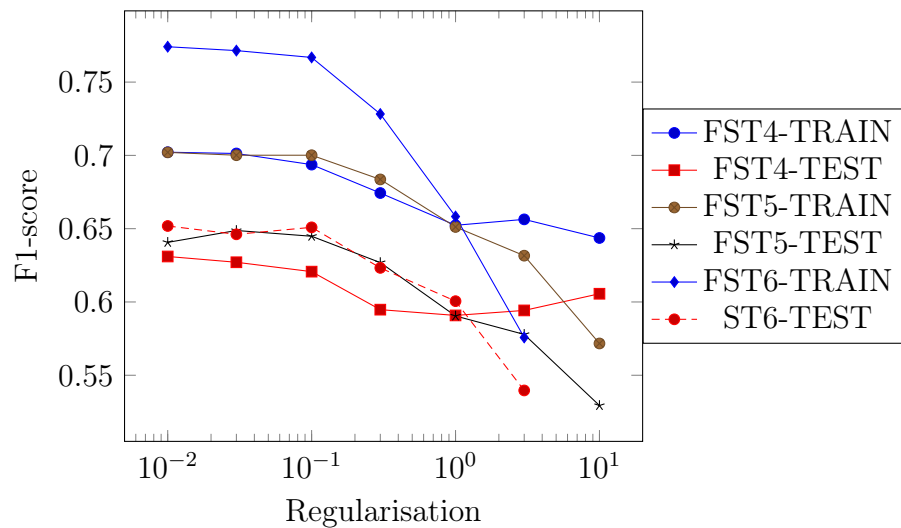


Figure A.4: Plots of the F1-scores of the different WFSs on the training and validation sets for MISSP1.

Set	Reg	FST1	FST2	FST3	FST4	FST5	FST6
Train	0.0	0.6264	0.6264	0.6720	0.7199	0.7182	0.7494
Val	0.0	0.6278	0.6278	0.6664	0.7047	0.6941	0.6901
Train	0.01	0.6264	0.6264	0.6721	0.7222	0.7188	0.7508
Val	0.01	0.6278	0.6278	0.6660	0.7049	0.6933	0.6913
Train	0.03	0.6264	0.6264	0.6729	0.7217	0.7183	0.7525
Val	0.03	0.6278	0.6278	0.6649	0.7009	0.6959	0.6935
Train	0.1	0.6264	0.6264	0.6724	0.7169	0.7183	0.7613
Val	0.1	0.6278	0.6278	0.6648	0.6939	0.6984	0.7014
Train	0.3	0.6264	0.6629	0.6726	0.7256	0.7209	0.7613
Val	0.3	0.6278	0.6630	0.6645	0.7063	0.6993	0.7046
Train	1.0	0.6264	0.6714	0.6739	0.7375	0.7308	0.7478
Val	1.0	0.6278	0.6694	0.6649	0.7206	0.7072	0.7019
Train	3.0	0.6264	0.6749	0.6775	0.7566	0.7635	0.7267
Val	3.0	0.6278	0.6812	0.6688	0.7457	0.7414	0.6991
Train	10.0	0.6517	0.6663	0.6884	0.8007	0.7590	0.7062
Val	10.0	0.6456	0.6659	0.6798	0.7936	0.7448	0.6921
Train	30.0	0.6634	0.6665	0.6958	0.7511	0.7071	0.6861
Val	30.0	0.6626	0.6667	0.6912	0.7495	0.6965	0.6819
Train	100.0	0.6743	0.6667	0.6720	0.7243	0.6846	0.6799
Val	100.0	0.6795	0.6667	0.6704	0.7288	0.6817	0.6706

Table A.4: F1-scores of the different WFSTs on the training and validation sets for MISSP2. When different regularisations get the same score, the higher middle value is chosen.

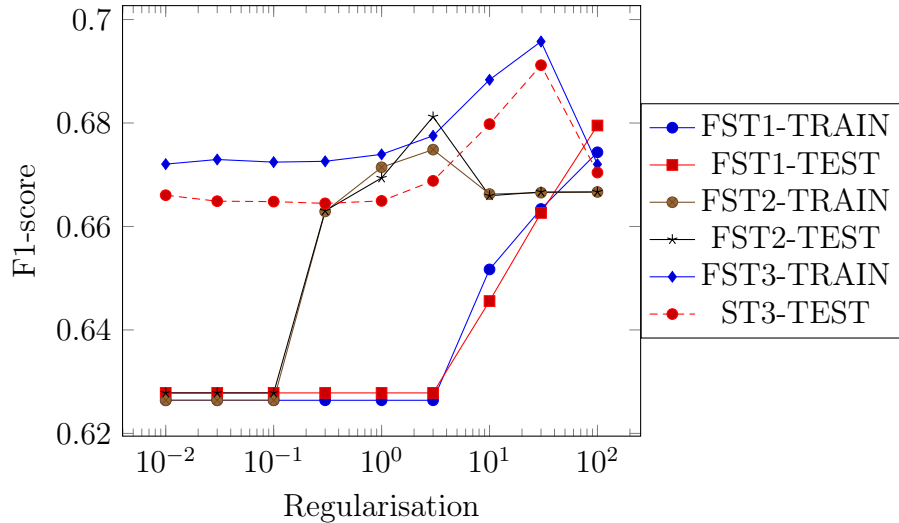


Figure A.5: Plots of the F1-scores of the different WFSTs on the training and validation sets for MISSP2.

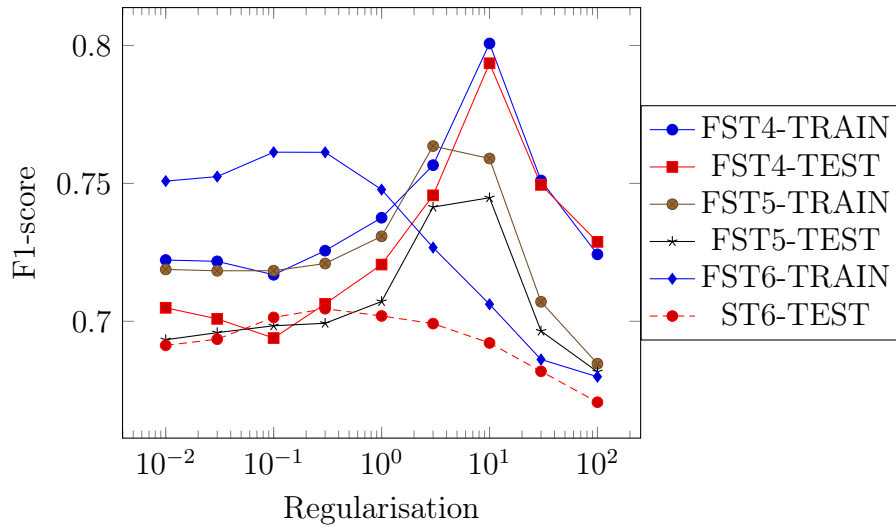


Figure A.6: Plots of the F1-scores of the different WFSs on the training and validation sets for MISSP2.

Set	Reg	FST1	FST2	FST3	FST4	FST5	FST6
Train	0.01	0.3159	0.3163	0.3834	0.4224	0.4281	0.4477
Val	0.01	0.3202	0.3202	0.4066	0.4182	0.4287	0.4353
Train	0.03	0.3159	0.3162	0.3834	0.4223	0.4280	0.4473
Val	0.03	0.3202	0.3202	0.4066	0.4192	0.4283	0.4363
Train	0.1	0.3159	0.3160	0.3835	0.4237	0.4278	0.4461
Val	0.1	0.3202	0.3202	0.4066	0.4183	0.4303	0.4416
Train	0.3	0.3159	0.3156	0.3837	0.4239	0.4281	0.4439
Val	0.3	0.3202	0.3202	0.4060	0.4189	0.4311	0.4406
Train	1.0	0.3159	0.3102	0.3837	0.4244	0.4266	0.4388
Val	1.0	0.3202	0.3158	0.4038	0.4199	0.4257	0.4418
Train	3.0	0.3157	0.2958	0.3810	0.4280	0.4274	0.4284
Val	3.0	0.3202	0.3008	0.3978	0.4257	0.4258	0.4339
Train	10.0	0.3157	0.2885	0.3698	0.4337	0.4276	0.4067
Val	10.0	0.3198	0.2948	0.3905	0.4208	0.4323	0.4216
Train	30.0	0.3095	0.2265	0.3410	0.4409	0.4043	0.3490
Val	30.0	0.3145	0.2326	0.3645	0.4301	0.4096	0.3663
Train	100.0	0.2943	0.1675	0.2773	0.3298	0.2654	0.2320
Val	100.0	0.2992	0.1696	0.2882	0.3359	0.2642	0.2363

Table A.5: F1-scores of the different WFSs on the training and validation sets for MISSP3. When different regularisations get the same score, the higher middle value is chosen.

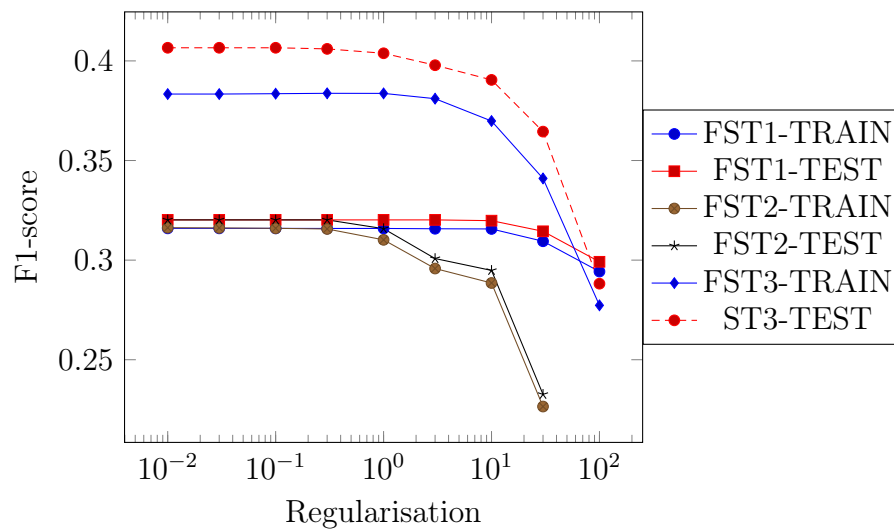


Figure A.7: Plots of the F1-scores of the different WFSs on the training and validation sets for MISSP3.

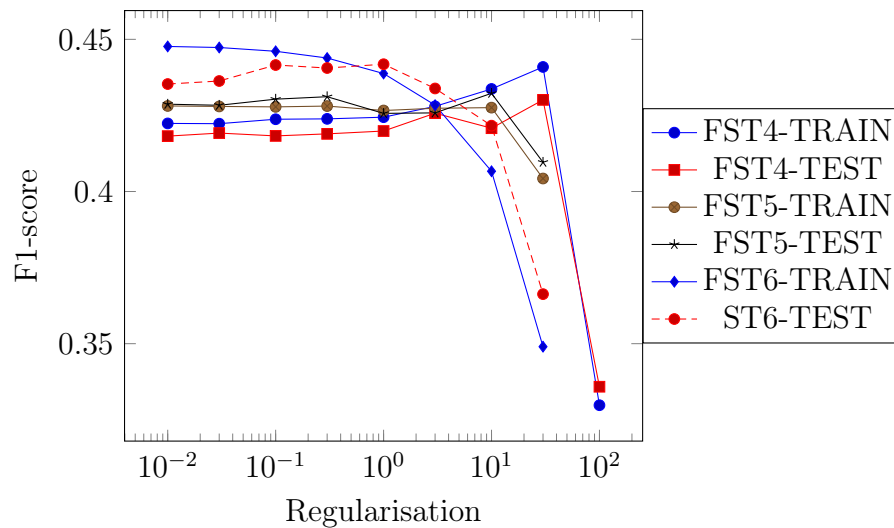


Figure A.8: Plots of the F1-scores of the different WFSs on the training and validation sets for MISSP3.

	WFST1	WFST2	WFST3	WFST4				WFST5		
				q_m	q_i	q_d	q_s	q_s	q_i	q_d
A	0.226	0.224	0.378	0.115	0.668	0.570	0.039	0.129	0.848	0.819
B	0.226	0.224	0.034	0.024	0.030	0.039	0.026	0.021	0.033	0.054
C	0.226	0.224	0.051	0.029	0.047	0.036	0.028	0.025	0.086	0.038
D	0.226	0.224	0.199	0.087	0.023	0.293	0.031	0.098	0.013	0.439
E	0.226	0.224	0.576	0.185	0.765	0.723	0.047	0.226	0.914	0.889
F	0.226	0.224	0.084	0.025	0.055	0.035	0.030	0.024	0.113	0.062
G	0.226	0.224	0.191	0.081	0.064	0.164	0.030	0.084	0.156	0.292
H	0.226	0.224	0.210	0.069	0.141	0.329	0.033	0.059	0.368	0.541
I	0.226	0.224	0.246	0.078	0.570	0.462	0.029	0.054	0.838	0.749
J	0.226	0.224	0.095	0.027	0.030	0.026	0.026	0.027	0.032	0.027
K	0.226	0.224	0.172	0.113	0.038	0.113	0.031	0.130	0.042	0.179
L	0.226	0.224	0.103	0.059	0.021	0.171	0.026	0.057	0.013	0.248
M	0.226	0.224	0.097	0.046	0.025	0.111	0.030	0.048	0.020	0.177
N	0.226	0.224	0.081	0.037	0.040	0.161	0.036	0.037	0.057	0.190
O	0.226	0.224	0.423	0.126	0.528	0.686	0.035	0.119	0.809	0.877
P	0.226	0.224	0.099	0.052	0.031	0.123	0.027	0.050	0.028	0.227
Q	0.226	0.224	0.087	0.019	0.036	0.025	0.065	0.056	0.033	0.023
R	0.226	0.224	0.121	0.045	0.031	0.258	0.028	0.034	0.041	0.383
S	0.226	0.224	0.165	0.110	0.028	0.270	0.031	0.120	0.026	0.356
T	0.226	0.224	0.144	0.072	0.028	0.292	0.031	0.077	0.029	0.411
U	0.226	0.224	0.303	0.152	0.322	0.293	0.040	0.114	0.537	0.521
V	0.226	0.224	0.071	0.019	0.035	0.040	0.029	0.014	0.058	0.060
W	0.226	0.224	0.173	0.058	0.045	0.205	0.030	0.048	0.081	0.324
X	0.226	0.224	0.316	0.061	0.033	0.094	0.044	0.127	0.030	0.185
Y	0.226	0.224	0.373	0.210	0.172	0.303	0.033	0.224	0.425	0.466
Z	0.226	0.224	0.268	0.085	0.060	0.072	0.077	0.164	0.109	0.125
1	0.226	0.224	0.359	0.042	0.027	0.025	0.025	0.074	0.027	0.026
2	0.226	0.224	0.001	0.023	0.027	0.026	0.025	0.019	0.027	0.027
3	0.226	0.224	0.002	0.037	0.036	0.025	0.025	0.027	0.035	0.025
4	0.226	0.224	0.001	0.024	0.025	0.025	0.025	0.022	0.025	0.025
5	0.226	0.224	0.002	0.025	0.025	0.025	0.025	0.023	0.025	0.025
6	0.226	0.224	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
7	0.226	0.224	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
8	0.226	0.224	0.001	0.031	0.032	0.025	0.025	0.029	0.028	0.025
9	0.226	0.224	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
0	0.226	0.224	0.101	0.061	0.037	0.034	0.025	0.051	0.028	0.054
'	0.226	0.224	0.270	0.073	0.059	0.030	0.026	0.068	0.053	0.037
*	0.226	0.224	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025

Table A.6: Parameter values learned by the WFST models for *deletions* of different symbols on MISSP3. For WFST4 and WFST5, the weights of the arcs going from states q_m , q_m , q_m , and q_m are shown. So, for example, the value in the table for symbol “E” and WFST4 q_i is 0.765. This is the weight of a deletion of “E” if the previous edit operation was an insertion. The symbol “*” here represents the <OTHER> character.

	WFST1	WFST2	WFST3	WFST4				WFST5		
				q_m	q_i	q_d	q_s	q_s	q_i	q_d
A	0.026	0.026	0.092	0.105	0.079	0.144	0.035	0.106	0.075	0.185
B	0.026	0.026	0.007	0.009	0.009	0.005	0.009	0.008	0.008	0.002
C	0.026	0.026	0.016	0.007	0.024	0.012	0.009	0.008	0.028	0.007
D	0.026	0.026	0.013	0.005	0.023	0.010	0.009	0.004	0.024	0.007
E	0.026	0.026	0.241	0.277	0.106	0.246	0.215	0.287	0.103	0.275
F	0.026	0.026	0.002	0.002	0.005	0.004	0.008		0.004	0.001
G	0.026	0.026	0.081	0.138	0.011	0.006	0.009	0.128	0.016	0.004
H	0.026	0.026	0.028	0.019	0.009	0.010	0.168	0.020	0.024	0.009
I	0.026	0.026	0.087	0.119	0.076	0.061	0.078	0.140	0.076	0.033
J	0.026	0.026		0.001	0.001	0.004	0.007			0.001
K	0.026	0.026	0.004	0.004	0.003	0.005	0.047	0.004	0.002	0.002
L	0.026	0.026	0.026	0.017	0.048	0.009	0.011	0.011	0.053	0.004
M	0.026	0.026	0.005	0.002	0.013	0.005	0.008	0.001	0.012	0.002
N	0.026	0.026	0.040	0.006	0.114	0.006	0.009	0.002	0.115	0.002
O	0.026	0.026	0.111	0.128	0.071	0.150	0.120	0.142	0.073	0.109
P	0.026	0.026	0.006	0.005	0.007	0.006	0.008	0.003	0.007	0.003
Q	0.026	0.026		0.001	0.002	0.003	0.007			0.001
R	0.026	0.026	0.056	0.011	0.146	0.020	0.009	0.007	0.151	0.016
S	0.026	0.026	0.029	0.008	0.047	0.056	0.039	0.004	0.047	0.075
T	0.026	0.026	0.057	0.019	0.089	0.057	0.025	0.023	0.090	0.101
U	0.026	0.026	0.037	0.038	0.041	0.038	0.022	0.038	0.037	0.041
V	0.026	0.026	0.001	0.001	0.004	0.004	0.007		0.002	0.001
W	0.026	0.026	0.009	0.005	0.008	0.008	0.009	0.004	0.007	0.012
X	0.026	0.026		0.001	0.003	0.003	0.007		0.001	0.001
Y	0.026	0.026	0.025	0.020	0.022	0.068	0.011	0.020	0.021	0.079
Z	0.026	0.026		0.001	0.003	0.004	0.007		0.001	0.001
1	0.026	0.026		0.001	0.001	0.003	0.007			0.001
2	0.026	0.026		0.001	0.001	0.003	0.007			0.001
3	0.026	0.026		0.001	0.001	0.003	0.007			0.001
4	0.026	0.026		0.001	0.001	0.003	0.007			0.001
5	0.026	0.026		0.001	0.001	0.003	0.007			0.001
6	0.026	0.026		0.001	0.001	0.003	0.007			0.001
7	0.026	0.026		0.001	0.001	0.003	0.007			0.001
8	0.026	0.026		0.001	0.001	0.003	0.007			0.001
9	0.026	0.026		0.001	0.001	0.003	0.007			0.001
0	0.026	0.026		0.001	0.001	0.003	0.007			0.001
'	0.026	0.026	0.015	0.027	0.003	0.005	0.011	0.024	0.002	0.002
*	0.026	0.026		0.001	0.001	0.003	0.007			0.001

Table A.7: Parameter values learned by the WFST models for *insertions* of different symbols on MISSP3. For WFST4 and WFST5, the weights of the arcs coming from states q_m , q_m , q_m , and q_m are shown. So, for example, the value in the table for symbol “E” and WFST4 q_i is 0.106. This is the weight of an insertion of “E” if the previous edit operation was an insertion. The symbol “*” here represents the <OTHER> character. Empty cells represent values of less than 0.001.

	WFST1	WFST2	WFST3	WFST4				WFST5		
				q_m	q_i	q_d	q_s	q_s	q_i	q_d
A	0.773	0.772	0.620	0.764	0.020	0.059	0.212	0.824	0.004	0.017
B	0.773	0.772	0.942	0.706	0.144	0.101	0.037	0.857	0.292	0.276
C	0.773	0.772	0.913	0.746	0.279	0.229	0.030	0.856	0.498	0.517
D	0.773	0.772	0.773	0.665	0.466	0.201	0.039	0.796	0.621	0.318
E	0.773	0.772	0.423	0.706	0.010	0.016	0.112	0.730	0.002	0.005
F	0.773	0.772	0.861	0.644	0.096	0.144	0.040	0.827	0.137	0.320
G	0.773	0.772	0.789	0.667	0.237	0.151	0.030	0.797	0.361	0.287
H	0.773	0.772	0.788	0.758	0.074	0.116	0.135	0.873	0.165	0.161
I	0.773	0.772	0.739	0.806	0.056	0.095	0.162	0.894	0.011	0.043
J	0.773	0.772	0.805	0.258	0.040	0.040	0.027	0.479	0.051	0.076
K	0.773	0.772	0.706	0.488	0.255	0.186	0.097	0.618	0.489	0.379
L	0.773	0.772	0.895	0.771	0.453	0.315	0.091	0.874	0.704	0.507
M	0.773	0.772	0.894	0.710	0.383	0.256	0.042	0.843	0.602	0.460
N	0.773	0.772	0.914	0.845	0.614	0.469	0.045	0.915	0.782	0.667
O	0.773	0.772	0.576	0.750	0.031	0.032	0.104	0.829	0.007	0.007
P	0.773	0.772	0.894	0.706	0.226	0.153	0.045	0.844	0.473	0.319
Q	0.773	0.772	0.227	0.091	0.037	0.041	0.024	0.127	0.062	0.084
R	0.773	0.772	0.877	0.796	0.523	0.288	0.083	0.901	0.737	0.412
S	0.773	0.772	0.830	0.752	0.570	0.364	0.068	0.819	0.780	0.488
T	0.773	0.772	0.854	0.799	0.578	0.328	0.076	0.871	0.757	0.423
U	0.773	0.772	0.649	0.606	0.209	0.095	0.091	0.723	0.226	0.126
V	0.773	0.772	0.905	0.529	0.211	0.162	0.028	0.745	0.392	0.395
W	0.773	0.772	0.814	0.663	0.176	0.078	0.040	0.827	0.354	0.135
X	0.773	0.772	0.477	0.133	0.121	0.063	0.025	0.222	0.249	0.135
Y	0.773	0.772	0.624	0.544	0.086	0.209	0.037	0.656	0.107	0.289
Z	0.773	0.772	0.125	0.092	0.045	0.028	0.024	0.123	0.069	0.025
1	0.773	0.772	0.001	0.025	0.024	0.025	0.025	0.024	0.022	0.025
2	0.773	0.772		0.022	0.025	0.025	0.025	0.018	0.024	0.025
3	0.773	0.772		0.024	0.024	0.025	0.025	0.021	0.023	0.025
4	0.773	0.772	0.001	0.024	0.025	0.025	0.025	0.022	0.024	0.025
5	0.773	0.772	0.002	0.024	0.025	0.025	0.025	0.023	0.025	0.025
6	0.773	0.772	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
7	0.773	0.772	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
8	0.773	0.772		0.025	0.024	0.025	0.025	0.024	0.021	0.025
9	0.773	0.772	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025
0	0.773	0.772		0.023	0.024	0.025	0.025	0.018	0.024	0.024
'	0.773	0.772	0.315	0.094	0.055	0.026	0.025	0.176	0.095	0.026
*	0.773	0.772	0.025	0.025	0.025	0.025	0.025	0.025	0.025	0.025

Table A.8: Parameter values learned by the WFST models for character *matches* of different symbols on MISSP3. For WFST4 and WFST5, the weights of the arcs going from states q_m , q_m , q_m , and q_m are shown. So, for example, the value in the table for symbol “E” and WFST4 q_i is 0.010. This is the weight of a matching “E”s if the previous edit operation was an insertion. The symbol “*” here represents the <OTHER> character. Empty cells represent values of less than 0.001.

A.2.2 HACRF

Set	Reg	HACRF1	HACRF2	HACRF3	HACRF4
Train	0.0	1.0000	1.0000	1.0000	1.0000
Val	0.0	0.7699	0.7181	0.7432	0.6955
Train	0.01	1.0000	1.0000	1.0000	1.0000
Val	0.01	0.7839	0.7989	0.8087	0.7688
Train	0.03	1.0000	1.0000	1.0000	1.0000
Val	0.03	0.7789	0.7911	0.8374	0.7804
Train	0.1	1.0000	1.0000	1.0000	1.0000
Val	0.1	0.8067	0.7853	0.7869	0.7764
Train	0.3	1.0000	1.0000	1.0000	1.0000
Val	0.3	0.8017	0.8048	0.8005	0.7552
Train	1.0	1.0000	1.0000	1.0000	1.0000
Val	1.0	0.8033	0.7819	0.7908	0.7461
Train	3.0	1.0000	1.0000	1.0000	1.0000
Val	3.0	0.7637	0.7713	0.7729	0.7461
Train	10.0	0.9859	0.9930	0.9930	1.0000
Val	10.0	0.8151	0.7725	0.7642	0.7319
Train	30.0	0.8682	0.9265	0.9412	0.9640
Val	30.0	0.7158	0.6991	0.6972	0.6583
Train	100.0	0.3820	0.5400	0.5743	0.6286
Val	100.0	0.1629	0.3447	0.3333	0.3313

Table A.9: F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP1.

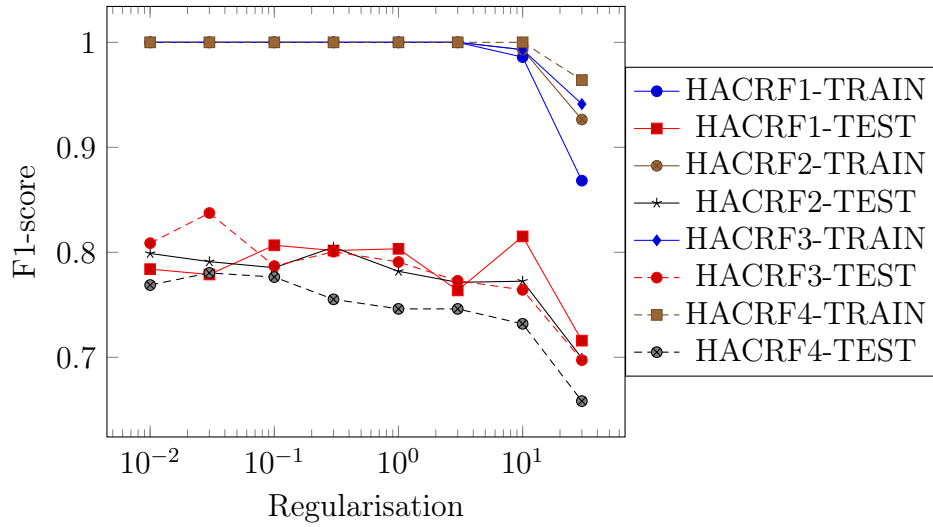


Figure A.9: Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP1.

Set	Reg	HACRF1	HACRF2	HACRF3	HACRF4
Train	0.01	1.0000	1.0000	1.0000	1.0000
Val	0.01	0.7586	0.7605	0.7408	0.7462
Train	0.03	1.0000	1.0000	1.0000	1.0000
Val	0.03	0.7624	0.7625	0.7434	0.7439
Train	0.1	1.0000	1.0000	1.0000	1.0000
Val	0.1	0.7659	0.7636	0.7487	0.7466
Train	0.3	1.0000	1.0000	1.0000	1.0000
Val	0.3	0.7771	0.7611	0.7518	0.7526
Train	1.0	1.0000	0.9985	1.0000	1.0000
Val	1.0	0.7748	0.7700	0.7557	0.7493
Train	3.0	0.9910	0.9709	0.9903	0.9985
Val	3.0	0.7781	0.7798	0.7713	0.7554
Train	10.0	0.9456	0.9190	0.9402	0.9768
Val	10.0	0.7854	0.7789	0.7855	0.7742
Train	30.0	0.9017	0.8825	0.8970	0.9289
Val	30.0	0.7770	0.7844	0.7925	0.7848
Train	100.0	0.8234	0.8448	0.8580	0.8808
Val	100.0	0.7342	0.7674	0.7838	0.7738

Table A.10: F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP2.

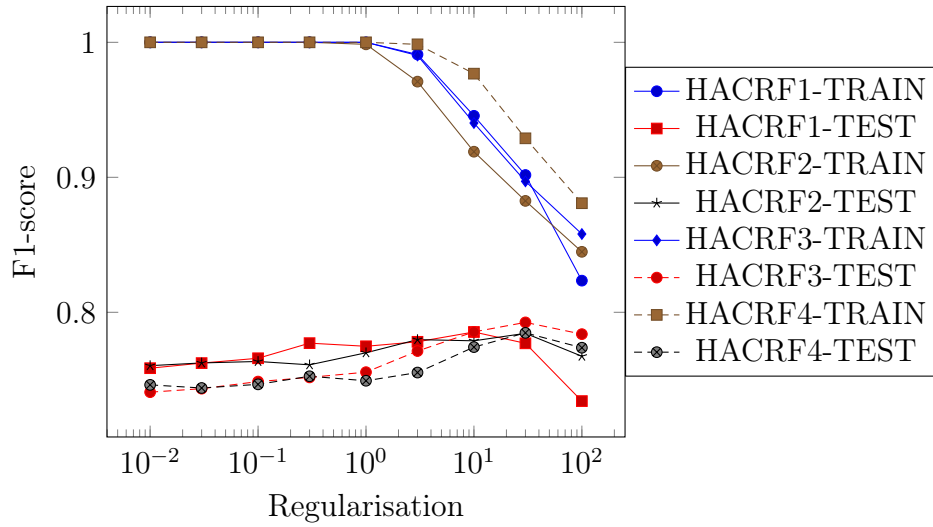


Figure A.10: Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP2.

Set	Reg	HACRF1	HACRF2	HACRF3	HACRF4
Train	0.01	1.0000	0.9921	1.0000	1.0000
val	0.01	0.4034	0.3969	0.3659	0.3426
Train	0.03	1.0000	0.9921	1.0000	1.0000
val	0.03	0.3922	0.3612	0.3705	0.3352
Train	0.1	1.0000	0.9921	1.0000	1.0000
val	0.1	0.4085	0.3605	0.3834	0.3106
Train	0.3	1.0000	0.9921	1.0000	1.0000
val	0.3	0.4180	0.3694	0.3764	0.3212
Train	1.0	1.0000	0.9921	1.0000	1.0000
val	1.0	0.4000	0.4181	0.4056	0.3089
Train	3.0	0.9760	0.9677	1.0000	1.0000
val	3.0	0.3836	0.4071	0.4390	0.3267
Train	10.0	0.8257	0.8257	0.8468	0.9508
val	10.0	0.3614	0.3901	0.4164	0.3113
Train	30.0	0.4578	0.4198	0.5287	0.6526
val	30.0	0.2259	0.1823	0.2290	0.2080
Train	100.0	0.1972	0.1972	0.1972	0.1972
val	100.0	0.0909	0.0909	0.0858	0.0701

Table A.11: F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP3.

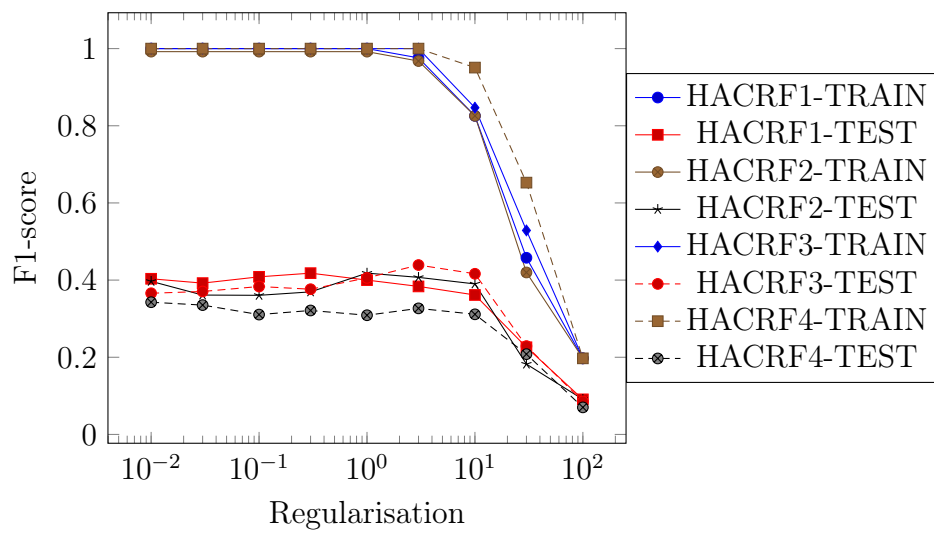


Figure A.11: Plots of the F1-scores on the training and validation sets for different regularisation values for the different HACRFs on MISSP3.

A.3 Sentiment experiment

1 hidden units		
	Positive	Negative
Positive	244	11
Negative	18	470
Chi squared	1.241	
Two-tailed P value	0.2652	

2 hidden units		
	Positive	Negative
Positive	241	9
Negative	20	471
Chi squared	3.448	
Two-tailed P value	0.0633	

3 hidden units		
	Positive	Negative
Positive	241	15
Negative	19	468
Chi squared	0.265	
Two-tailed P value	0.6069	

Table A.12: The McNemar significant test results for the sentiment classification task. The significance of the difference in performance between the classifier that is trained and tested on the original data and the classifier that is trained and tested on the cleaned data is calculated. None of the classifiers differ significantly if a significance threshold of 0.05 is used. McNemar's test with the continuity correction is used [69].

List of References

- [1] S. Agarwal, S. Godbole, D. Punjani, and S. Roy. How much noise is too much: A study in automatic text classification. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 3–12. IEEE, 2007.
- [2] S.M. Aji, G.B. Horn, and R.J. McEliece. Iterative decoding on graphs with a single cycle. In *Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on*, page 276. IEEE, 1998.
- [3] A.T. Aw, M. Zhang, J. Xiao, and J. Su. A phrase-based statistical model for sms text normalization. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 33–40. Association for Computational Linguistics, 2006.
- [4] R. Beaufort, S. Roekhaut, L.A. Cougnon, and C. Fairon. A hybrid rule/model-based finite-state framework for normalizing sms messages. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 770–779. Association for Computational Linguistics, 2010.
- [5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [6] Christopher M Bishop, Julia Lasserre, et al. Generative or discriminative? getting the best of both worlds. *Bayesian Statistics*, 8:3–23, 2007.
- [7] Stephen Poythress Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [8] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). <http://www.w3.org/TR/2008/REC-xml-20081126/>, 2008.
- [9] E. Brill and R.C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Com-*

- putational Linguistics*, pages 286–293. Association for Computational Linguistics, 2000.
- [10] S.F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.
- [11] M. Choudhury, R. Saraf, V. Jain, A. Mukherjee, S. Sarkar, and A. Basu. Investigation and modeling of the structure of texting language. *International Journal on Document Analysis and Recognition*, 10(3):157–174, 2007.
- [12] E. Clark and K. Araki. Text normalization in social media: Progress, problems and applications for a pre-processing system of casual english. *Procedia-Social and Behavioral Sciences*, 27:2–11, 2011.
- [13] D. Contractor, T.A. Faruque, and L.V. Subramaniam. Unsupervised cleansing of noisy text. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 189–196. Association for Computational Linguistics, 2010.
- [14] P. Cook and S. Stevenson. An unsupervised model for text message normalization. In *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity*, pages 71–78. Association for Computational Linguistics, 2009.
- [15] D. Crockford. The application/json media type for javascript object notation (json). <http://www.json.org/>, 2006.
- [16] D. Crystal. The scope of internet linguistics. In *Proceedings of American Association for the Advancement of Science Conference; American Association for the Advancement of Science Conference, Washington, DC, USA*, pages 17–21, 2005.
- [17] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.
- [18] J. Eisenstein, B. O’Connor, N.A. Smith, and E.P. Xing. A latent variable model for geographic lexical variation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1277–1287. Association for Computational Linguistics, 2010.

- [19] J. Eisner. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 1–8. Association for Computational Linguistics, 2002.
- [20] B. Eriksson. Sentiment classification of movie reviews using linguistic parsing. *Natural Language Processing. CS*, 838, 2006.
- [21] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.
- [22] W.N. Francis and H. Kucera. Brown corpus manual: Manual of information to accompany a standard corpus of present-day edited american english for use with digital computers. *Brown University, Providence, Rhode Island*, 1964.
- [23] A.R. Golding and D. Roth. A winnow-based approach to context-sensitive spelling correction. *Machine learning*, 34(1):107–130, 1999.
- [24] S. Gouws, D. Hovy, and D. Metzler. Unsupervised mining of lexical variants from noisy text. In *Proceedings of the First workshop on Unsupervised Learning in NLP*, pages 82–90. Association for Computational Linguistics, 2011.
- [25] S. Gouws, D. Metzler, C. Cai, and E. Hovy. Contextual bearing on linguistic variation in social media. *ACL HLT 2011*, page 20, 2011.
- [26] J. Graehl. Carmel finite-state transducer package. <http://www.isi.edu/licensed-sw/carmel/>, 1997.
- [27] J.M. Hammersley and P. Clifford. Markov fields on finite graphs and lattices. 1968.
- [28] B. Han and T. Baldwin. Lexical normalisation of short text messages: Makn sens a# twitter. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 368–378, 2011.
- [29] B. Han, P. Cook, and T Baldwin. Automatically constructing a normalisation dictionary for microblogs. In *In Proceedings of the Conference on Empirical Methods in Natural Language Processing and Natural Language Learning , Jeju, Korea*, pages 421–432, 2012.
- [30] A. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in neural information processing systems*, 14:841, 2002.

- [31] M. Kaufmann and J. Kalita. Syntactic normalization of twitter messages. In *International Conference on Natural Language Processing, Kharagpur, India*, 2010.
- [32] K. Knight and Y. Al-Onaizan. Translation with finite-state devices. *Machine translation and the information soup*, pages 421–437, 1998.
- [33] C. Kobus, F. Yvon, and G. Damnati. Normalizing sms: are two metaphors better than one? In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 441–448. Association for Computational Linguistics, 2008.
- [34] P. Koehn, F.J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics, 2003.
- [35] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
- [36] T. Kudo. Crf++: Yet another crf toolkit. <http://crfpp.googlecode.com/svn/trunk/doc/index.html>, 2005.
- [37] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *International Conference on Machine Learning*, pages 282–289, 2001.
- [38] Pat Langley et al. *Selection of relevant features in machine learning*. Defense Technical Information Center, 1994.
- [39] S.L. Lauritzen and D.J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 157–224, 1988.
- [40] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Forschungsbericht*, 8:707–710, 1966.
- [41] F. Liu, F. Weng, and X. Jiang. A broadcoverage normalization system for social media language. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2012), Jeju, Republic of Korea*, 2012.
- [42] F. Liu, F. Weng, B. Wang, and Y. Liu. Insertion, deletion, or substitution? normalizing text messages without pre-categorization nor supervision. *Proc. of ACL-HLT*, 2011.

- [43] David JC MacKay. Choice of basis for laplace approximation. *Machine learning*, 33(1):77–86, 1998.
- [44] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.
- [45] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.
- [46] A. McCallum, K. Bellare, and F. Pereira. A conditional random field for discriminatively-trained finite-state string edit distance. Technical report, DTIC Document, 2005.
- [47] M. Mohri, F. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.
- [48] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.
- [49] N. Okazaki, Y. Tsuruoka, S. Ananiadou, and J. Tsujii. A discriminative candidate generator for string transformations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 447–456. Association for Computational Linguistics, 2008.
- [50] A. Pak and P. Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *Proceedings of LREC*, volume 2010, 2010.
- [51] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics, 2002.
- [52] J. Pearl. *Reverend Bayes on inference engines: a distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.
- [53] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.
- [54] D. Pennell and Y. Liu. Toward text message normalization: Modeling abbreviation generation. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5364–5367. IEEE, 2011.

- [55] Yuan Qi, Martin Szummer, and Thomas P Minka. Bayesian conditional random fields. In *Proceedings of the 10th Tenth International Workshop on Artificial Intelligence and Statistics*, pages 269–276, 2005.
- [56] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 134–141. Association for Computational Linguistics, 2003.
- [57] P.P. Shenoy and G. Shafer. Propagating belief functions with local computations. *IEEE Expert*, 1(3):43–52, 1986.
- [58] A. Stolcke et al. Srilm-an extensible language modeling toolkit. In *Proceedings of the international conference on spoken language processing*, volume 2, pages 901–904, 2002.
- [59] L.V. Subramaniam, S. Roy, T.A. Faruquie, and S. Negi. A survey of types of text noise and techniques to handle noisy text. In *Proceedings of The Third Workshop on Analytics for Noisy Unstructured Text Data*, pages 115–122. ACM, 2009.
- [60] C. Sutton and A. McCallum. *An introduction to conditional random fields for relational learning*. Introduction to statistical relational learning. MIT Press, 2006.
- [61] C. Tagg. *A corpus linguistics study of SMS text messaging*. PhD thesis, University of Birmingham, 2009.
- [62] C. Thurlow and A. Brown. Generation txt? the sociolinguistics of young people’s text-messaging. *Discourse analysis online*, 1(1):1–27, 2003.
- [63] J. Toole. Categorizing unknown words: Using decision trees to identify names and misspellings. In *Proceedings of the sixth conference on Applied natural language processing*, pages 173–179. Association for Computational Linguistics, 2000.
- [64] K. Toutanova and R.C. Moore. Pronunciation modeling for improved spelling correction. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 144–151. Association for Computational Linguistics, 2002.

- [65] M.J. Wainwright, T.S. Jaakkola, and A.S. Willsky. Tree-reweighted belief propagation algorithms and approximate ml estimation by pseudo-moment matching. In *Workshop on Artificial Intelligence and Statistics*, volume 21, page 97. Society for Artificial Intelligence and Statistics Np, 2003.
- [66] H.M. Wallach. Conditional random fields: An introduction. *Technical Reports (CIS)*, page 22, 2004.
- [67] S.B. Wang, A. Quattoni, L.P. Morency, D. Demirdjian, and T. Darrell. Hidden conditional random fields for gesture recognition. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 1521–1527. Ieee, 2006.
- [68] Z. Xue, D. Yin, and B.D. Davison. Normalizing microtext. In *Proceedings of the AAAI Workshop on Analyzing Microtext*, pages 74–79, 2011.
- [69] F. Yates. Contingency tables involving small numbers and the χ^2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.