DOCTOR OF PHILOSOPHY

Adaptive models of Arabic text

Alhawiti, Khaled

Award date:
2014

Awarding institution:
Bangor University

Adaptive Models of Arabic Text

Submitted by Khaled M. Alhawiti

for the degree of

Doctor of Philosophy

Bangor University

March 21, 2014

Declaration and Consent

Details of the Work

I hereby agree to deposit the following item in the digital repository maintained by Bangor University and/or in any other repository authorized for use by Bangor University.

Author Name: .....

Title:.....

Supervisor/Department: .....

Funding body (if any): .....

Qualification/Degree obtained: .....

This item is a product of my own research endeavours and is covered by the agreement below in which the item is referred to as "the Work". It is identical in content to that deposited in the Library, subject to point 4 below.

Non-exclusive Rights

Rights granted to the digital repository through this agreement are entirely non-exclusive. I am free to publish the Work in its present version or future versions elsewhere.

I agree that Bangor University may electronically store, copy or translate the Work to any approved medium or format for the purpose of future preservation and accessibility. Bangor University is not under any obligation to reproduce or display the Work in the same formats or resolutions in which it was originally deposited.

Bangor University Digital Repository

I understand that work deposited in the digital repository will be accessible to a wide variety of people and institutions, including automated agents and search engines via the World Wide Web.

I understand that once the Work is deposited, the item and its metadata may be incorporated into public access catalogues or services, national databases of electronic theses and dissertations such as the British Library's EThOS or any service provided by the National Library of Wales.

I understand that the Work may be made available via the National Library of Wales Online Electronic Theses Service under the declared terms and conditions of use (`http://www.llgc.org.uk/index.php?id=4676`). I agree that as part of this service the National Library of Wales may electronically store, copy or convert the Work to any approved medium or format for the purpose of future preservation and accessibility. The National Library of Wales is not under any obligation to reproduce or display the Work in the same formats or resolutions in which it was originally deposited.

Statement 1:

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree unless as agreed by the University for approved dual awards.

Signed.....(candidate)

Date .....

Statement 2:

This thesis is the result of my own investigations, except where otherwise stated. Where correction services have been used, the extent and nature of the correction is clearly marked in a footnote(s).

All other sources are acknowledged by footnotes and/or a bibliography.

Signed ..... (candidate)

Date .....

Statement 3:

I hereby give consent for my thesis, if accepted, to be available for photocopying, for inter-library loan and for electronic storage (subject to any constraints as defined in statement 4), and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

NB: Candidates on whose behalf a bar on access has been approved by the Academic Registry should use the following version of Statement 3:

Statement 3 (bar):

I hereby give consent for my thesis, if accepted, to be available for photocopying, for inter-library loans and for electronic storage (subject to any constraints as defined in statement 4), after expiry of a bar on access.

Signed ..... (candidate)

Date .....

Statement 4:

Choose one of the following options

a) I agree to deposit an electronic copy of my thesis (the Work) in the Bangor University (BU) Institutional Digital Repository, the British Library ETHOS system, and/or in any other repository authorized for use by Bangor University and where necessary have gained the required permissions for the use of third party material.	
b) I agree to deposit an electronic copy of my thesis (the Work) in the Bangor University (BU) Institutional Digital Repository, the British Library ETHOS system, and/or in any other repository authorized for use by Bangor University when the approved bar on access has been lifted.	
c) I agree to submit my thesis (the Work) electronically via Bangor University's e-submission system, however I opt-out of the electronic deposit to the Bangor University (BU) Institutional Digital Repository, the British Library ETHOS system, and/or in any other repository authorized for use by Bangor University, due to lack of permissions for use of third party material.	

Options B should only be used if a bar on access has been approved by the University.

In addition to the above I also agree to the following:

1. That I am the author or have the authority of the author(s) to make this agreement and do hereby give Bangor University the right to make available the Work in the way described above.

2. That the electronic copy of the Work deposited in the digital repository and covered by this agreement, is identical in content to the paper copy of the Work deposited in the Bangor University Library, subject to point 4 below.

3. That I have exercised reasonable care to ensure that the Work is original and, to the best of my knowledge, does not breach any laws – including those relating to defamation, libel and copyright.

4. That I have, in instances where the intellectual property of other authors or copyright holders is included in the Work, and where appropriate, gained explicit permission for the inclusion of that material in the Work, and in the electronic form of the Work as accessed through the open access digital repository, or that I have identified and removed that material for which adequate and appropriate permission has not been obtained and which will be inaccessible via the digital repository.

5. That Bangor University does not hold any obligation to take legal action on behalf of the Depositor, or other rights holders, in the event of a breach of intellectual property rights, or any other right, in the material deposited.

6. That I will indemnify and keep indemnified Bangor University and the National Library of Wales from and against any loss, liability, claim or damage, including without limitation any related legal fees and court costs (on a full indemnity bases), related to any breach by myself of any term of this agreement.

   Signature: ..... Date : .....

To my Parents and Lama

Abstract

The main aim of this thesis is to build adaptive language models of Arabic text that can achieve the best compression performance over existing models.

Prediction by partial matching (PPM) language models has been the best performing over the other adaptive language models through the past three decades in term of compression performance. In order to get such performance for Arabic text, the rich morphological nature of Arabic language should be taken into consideration.

In this thesis, two new resources of Arabic language have been introduced for understanding the nature of Arabic language and standardizing the experiments on Arabic text. The first is a new corpus, the Bangor Arabic Compression Corpus (BACC), for standardizing compression experiments and creating a benchmark corpus for future compression experiments on Arabic text. The second is a new corpus, Bangor Balanced Corpus of Contemporary Arabic (BBCCA), The purpose of this corpus is to mirror similar balanced corpora that are available for the English language (Brown and LOB) but instead comprises the Arabic language.

Two new adaptive models, BS-PPM and CS-PPM, based on the Prediction by Partial Matching (PPM) compression scheme are then introduced to improve the compression performance of standard PPM model by using preprocessing techniques. The first model works by replacing the most

frequent bigraphs with unique characters and the second model works by separating the encoding of the processing text into two streams, named the vocabulary stream and symbols stream. Both models achieve excellent compression results with significant improvements over standard PPM.

A further novel model adapted especially for the characteristics of Arabic text, lossless dotted and lossy non-dotted variants of PPM, are then introduced to also improve the compression performance over standard PPM by using the historical feature of Arabic language being non dotted. This method also achieves excellent compression results.

We also have investigated some applications of PPM models to the problems of authorship attribution, word segmentation and correcting of OCR output for Arabic text that demonstrate excellent results using PPM.

Acknowledgments

I would like to express my deepest gratitude to Dr. William J. Teahan, for his outstanding supervision and enthusiasm during my research. I was fortunate to work with him.

I would like to pay special thanks to my parents, for their love, support and taking care of my daughter, Lama, during the years when I was pursuing my PhD degree.

My deepest emotions are for my daughter Lama; I know you missed me, as I do always. I have now finished my PhD degree, and I will never ever be away again.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Background and Motivation

The Arabic language "العربية" is one of the most widely spoken languages in the world as it is the primary language for 255 million people in Asia and North Africa (Lewis et al., 2009). Nowadays, the information being used, stored and transferred by computer users is rapidly growing. In Arabic countries, Internet use grew 5,836.9% between 2000 and 2009 (Marketing, 2013). Figure1.1 shows the statistics of the growth of Internet use in Arabic counties between 2000 and 2009.

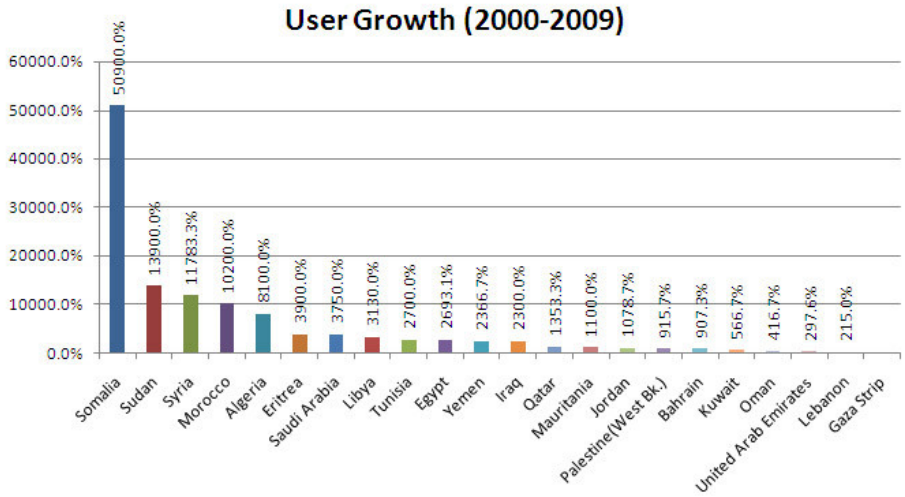

Figure 1.1: The statistics of the growth of Internet use in Arabic countries between 2000 and 2009 (Marketing,2013).

It is obvious there is an urgent need for electronic computer processing in order to understand and generate the information expressed by natural human languages. Natural Language Processing (NLP) is a science that investigates the interactions between computers and natural human languages. NLP uses these interactions to investigate state-of-the-art solutions for efficient processing of the information carried by these natural languages. Text compression, word segmentation and machine translation are examples of NLP research applications.

Text compression is the process of reducing the space needed to store files on computers, or it involves reducing the time needed to transmit information over a given bandwidth without losing any information from the

original text, known as being lossless compression (Bell et al., 1989). The main goal of text compression is to save expensive media and resources.

There are two main adaptive approaches to building text compression models of natural languages, which are dictionary and statistically based (Bell et al., 1990). Each of these approaches has advantages and disadvantages in specific terms.

In terms of execution speed, experiments show a dictionary-based approach is faster than a statistically-based approach, but, in terms of compression rate, a dictionary-based approach usually shows a worse rate than a statistically-based approach. Therefore, this thesis is only concerned with the statistical algorithms in the Arabic language to find the models that achieve the best performance in terms of compression rate, and can capture the properties of the language being modeled. One of these techniques is prediction by partial matching (PPM) (Cleary and Witten, 1984). PPM is an adaptive statistical technique, which builds and updates text compression models dynamically depending on the previous input stream of the characters being processed.

The Arabic language has its own characteristics that distinguish it from other languages (such as the English language). So far, there is a lack of research concerned with PPM-based modeling of Arabic text for NLP application in comparison with the English (Teahan, 1998) and Chinese (Wu, 2007) languages, which motivated us to conduct this research for the Arabic language.

1.2 Thesis Hypothesis

The different nature of languages is an important perspective that should be taken into consideration when processing these languages for NLP applications such as text compression. Arabic language has distinguishing characteristics which can be employed to customize PPM models for Arabic text and text in other languages that use Arabic script, such as Per-

sian, Kurdish and Urdu. This employment can significantly improve the text compression performance over standard PPM in terms of compression rate.

1.3 Thesis Aim and Research Questions

The broad aim of this thesis is to investigate how to set up adaptive computer models of Arabic text that are effective in term of compression performance. Adaptive statistical models of PPM will be examined to overcome the drawbacks of applying standard PPM when processing Arabic text due to the linguistic differences between the Arabic and English languages.

The research questions of this thesis are as follows.

- What are the disadvantages of the current models when they are applied to Arabic text?
- What is the best computer model for compressing Arabic text?
- How well do these models perform in several natural language processing applications?
- Can new language models be devised that lead to significant improvements in Arabic text compression?

1.4 Thesis Contribution

As this thesis is concerned with adaptive models of the Arabic language, the main contribution of this thesis is the proposal of three new language models designed especially for the Arabic language. These models are Bigraph Substitution for PPM (BS-PPM), Character Substitution for PPM (CS-PPM) and Lossless dotted and lossy non-dotted PPM.

The research questions listed in the previous section have been achieved by producing BS-PPM, CS-PPM and lossless dotted and lossy non-dotted PPM models designed for Arabic text (see Chapter 4). Also, BS-PPM and CS-PPM have shown positive results when applied to other languages that

use Arabic script in their writing system, such as Persian and Kurdish but also works well for non-Arabic script languages such as English, Armenian, Chinese and Russian. Many Arabic-scripted NLP applications can use these models to enhance their performance. The significant results concerning this thesis are listed as below.

1. We introduced a universal text preprocessing technique for PPM alphabet adjustment that adapts PPM for Arabic and other languages such as Persian, Armenian, Russian, Welsh and English by introducing BS-PPM and CS-PPM. These achieved significant improvement in the compression rate over standard PPM. This work is discussed in Chapter 4 and has been published in the Data Compression Conference proceedings (DCC 2014) (Alhawiti and Teahan, 2014).

2. We have set up a corpus called the Bangor Arabic Compression Corpus (BACC) comprised of 31 million words, available at `http://pages.bangor.ac.uk/~eepe04`. It can be used to examine the impact of PPM over different genres and sizes of Arabic text, which is the first Arabic corpus introduced in this manner. This work is described in Chapter 3 and has been published as a technical report available at `pages.bangor.ac.uk` (Teahan and Alhawiti, 2013).

3. We introduced a lossless dotted and a lossy non-dotted PPM compression scheme that makes use of non-dotted characters in Arabic for encoding instead of dotted characters, which achieves excellent experimental results. This work is discussed in Chapter 4 and has been published in the Data Compression Conference proceedings (DCC 2014) (Alhawiti and Teahan, 2014).

4. We introduced word- and tag-based models for Arabic text. This work is discussed in Chapter 5.

5. We investigated several natural language processing applications for

Arabic, such as correcting OCR text, language segmentation and authorship attribution. We discuss this in Chapter 6 and its in preparation to be submitted to Computational Linguistics Journal (CL2014).

6. We introduced the Bangor Balanced Corpus of Contemporary Arabic (BBCCA) that is comprised of one million words, which is, to our knowledge, the first balanced corpus using the same structural and sampling format used in the Brown Corpus (Francis and Kucera, 1979) which is a standard for corpus compilation and design. This work is discussed in Chapter 3.

7. We introduced a compression-based method for ranking n-gram differences between texts. The method can be used as the basis for producing tag clouds and is effective at revealing different topics between two or more texts. Also, this method can be used as a tool for comparing corpora. This work is introduced in Chapter 3 and has been published at the 7th Saudi Student Conference (Teahan and Alhawiti, 2014).

Conclusions, a summary and future work are discussed in the final chapter.

Chapter 2

Arabic Language Overview

Contents

2.1 Intoduction

"Language is a purely human and non-instinctive method of communicating ideas, emotions and desires by means of voluntarily produced symbols" (Sapir, 1921).

The purpose of this chapter is to review some of the structural characteristics of printed Arabic text. The chapter is organized as follows. First, we provide an overview of the Arabic language and its written characters. Then, we review several encoding methods designated for Arabic script. After that, some theoretical characteristics of Arabic characters and words are examined. The conclusion and summary are provided in the last section of this chapter.

2.2 Arabic: One of the Most Widely Spoken Languages in the World

Arabic "اللغة العربية" is one of the most widely spoken languages in the world, according to Lewis et al. (2009). It is the primary language for 255 million people in the Middle East and Africa and a second language for many other millions. The Arabic language affects directly and indirectly other languages in eastern Asia like Malay, Kurdish, Urdu and Persian. It also affects Romance languages like Spanish, Portuguese and Sicilian, and it borrows words from these languages (Weekley, 2012). The following table shows the most widely spoken languages in the world based on the number of speakers.

Table 2.1: The most widely spoken languages by number of speakers (Lewis et al. 2009).

Position	Language	Script Used	Speakers (millions)	Major Region
1	Mandarin	Chinese	1051	China ,Taiwan
2	English	Latin	508	USA, Canada, UK, Australia, New Zealand
3	Hindi	Devanagari	497	India
4	Spanish	Latin	392	The Americas, Spain
5	Russian	Cyrillic	277	Russia, Central Asia
6	Arabic	Arabic	255	Middle East, North and South Africa
7	Bengali	Bengali	211	Bangladesh, Eastern India
8	Portuguese	Latin	191	Brazil, Portugal
9	Malay	Latin	159	Indonesia, Malaysia
10	French	Latin	129	France, Canada

Arabic is a part of the Afro-Asiatic language family. The Arabic language belongs to the central Semitic languages, which were the first languages to apply an alphabetic script to the writing system, even before the Latin and Greek languages (Kenneth, 2002). Arabic is the original language of the Holy Qur'an, the Islamic holy book, which serves to maintain the vitality of the Arabic language and its importance to Muslims, even those who are not Arabic speakers. In addition, the names of some very old churches and the text of some Christian holy books and prayers were written and spoken in Arabic, such as at the al-Muallaqah church in Egypt. Therefore, the Arabic language is very important to Christians too. As well, during the Middle Ages, important Jewish holy books were written in Arabic.

The Arabic language has unique characteristics that distinguish it from other languages, like masculine and feminine forms and a distinction between singular, dual and plural forms in speech. These are represented by "انت" for single, "انتما" for dual and "انتم" for plural. Furthermore, the Arabic language is read from right to left.

2.3 Arabic Character: Written Language for Arabic

As Arabic was one of the earliest Semitic languages to establish a writing system, the characters have developed over time. Arabic has 28 basic

letters which are "ا ب ت ث ج ح خ د ذ ر ز س ش ص ض ط ظ ع غ ف ق ك ل م ن ه و ي" sorted in alphabetical order and reading form right to left and another four letters derived from the letter "ا" , the first letter in the Arabic alphabet, which are "ى ء ؤ ئ" . These derived letters in addition to the letter "ا ي و" are the vowels in Arabic while the rest of the letters are the consonants.

An Arabic character can be vowelised or not vowelised. In its natural form, Arabic words are separated by spaces. In recent centuries, the written Arabic system has been changed and developed from the "Musnad" and "Thamodi" writing system styles to the "Nabati" writing system style (Versteegh and Versteegh, 1997). Even the Arabic numbering style has changed. It used to appear as traditional numerals, like 0, 1, 2 and 3, but now appear as the current style, which is the Indian number system, represented as "..... ٣ ،٢ ،١ ،٠". Figure 2.1 shows the former Arabic writing style which is a letter by prophet Muhammad peace be upon him (UmAlqura, 2014).

Figure 2.1: The former Arabic writing style (UmAlqura, 2014).

Arabic is a vowelised language. Vowelised signs — special symbols written above or below a character — are formed to help readers to read words in the correct way. This is demonstrated with the examples like "كِتاب" and "كُتّاب", which mean book and writers, respectively; the alphabetic script is the same but the meaning is different. In the last example, the vowelised

form marks a difference between two words that occur only in the Arabic language. The two main types of hand-writing styles in Arabic script are "Naskh" and "Ruqa'a", as shown below, respectively.

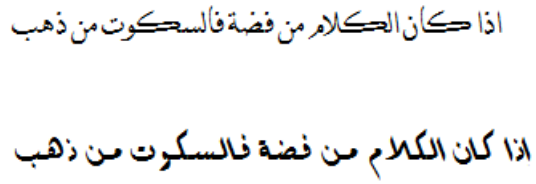

Figure 2.2: New writing style.

Also, Arabic is a morphologically rich language (Ng et al., 2009). The same word can be found in one text in many different forms. Words take on four different forms and can be comprised of one or more of these forms. The first form is a word with no prefixes and/or suffixes added to it, such as "ولد", which means "boy". The second form is a word plus a prefix, such as "الولد" with the prefix "ال" added, which means "the boy".

The third form is a word plus a suffix, such as "ولدان" with the suffix "ان" added, which means "two boys". The last form is a word plus a prefix and a suffix attached, such as "الولدان" with the prefix "ال" and the suffix "ان" added which means "the two boys". It should be noted that Arabic words may be written with more than one prefix and/or suffix attached, such as "والولد" with the prefixes "و" and "ال" attached, which means "and the boy". We will talk about these prefixes and suffixes in more detail in the next chapter in 3.3.1.

2.4 Arabic Encoding Method

The Arabic script employs Arabic characters "ا ب ت العربية", as it is a written language. Each character represents a unit of the language. Text can consist of one meaningful character, like "ص" or more than one character. Arabic has 28 characters and has no upper or lower case. Unlike English, Arabic words consisting of more than one character should be written cur-

sively, with the exception of characters, such as Alef "ا" at the beginning of the word and dal "د", thal "ذ", ra "ر" and zeen "ز" at any position in the word. In this section, various types of Arabic encoding schemes will be discussed.

2.4.1 ISO (8859-6) Arabic Coding Standard

ISO 8859-6 Arabic coding is an 8-bit character scheme, which was designed by the European Computer Manufacturers Association. This encoding method was designed especially for use with the Arabic language but not Persian or Kurdish. This is because Arabic script is used in the writing system of these languages but also ISO standard does not representing Arabic supplement characters as well. The ISO standard consists of 16 parts, each representing a different language plus Latin characters. The range of Arabic characters is located in part 6, which is "Arabic/Latin", as shown in Figure 2.3.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80																
90																
A0	NBSP 00A0				¤ 00A4								، 060C	00AD		
B0												؛ 061B				؟ 061F
C0		ء 0621	آ 0622	أ 0623	ؤ 0624	إ 0625	ئ 0626	ا 0627	ب 0628	ة 0629	ت 062A	ث 062B	ج 062C	ح 062D	خ 062E	د 062F
D0	ذ 0630	ر 0631	ز 0632	س 0633	ش 0634	ص 0635	ض 0636	ط 0637	ظ 0638	ع 0639	غ 063A					
E0	ـ 0640	ف 0641	ق 0642	ك 0643	ل 0644	م 0645	ن 0646	ه 0647	و 0648	ى 0649	ي 064A	ً 064B	ٌ 064C	ٍ 064D	َ 064E	ُ 064F
F0	ِ 0650	ّ 0651	ْ 0652													

Figure 2.3: ISO 8859-6 Arabic encoding scheme

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	€ 20AC	پ 067E	‚ 201A	ƒ 0192	„ 201E	… 2026	† 2020	‡ 2021	ˆ 02C6	‰ 2030	ٹ 0679	‹ 2039	Œ 0152	چ 0686	ژ 0698	ڈ 0688
90	گ 06AF	‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014	ک 06A9	™ 2122	ڑ 0691	› 203A	œ 0153	ZWNJ 200C	ZWJ 200D	ں 06BA
A0	NBSP 00A0	، 060C	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ھ 06BE	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 00AF
B0	° 00B0	± 00B1	² 00B2	³ 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	¹ 00B9	؛ 061B	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	؟ 061F
C0	ہ 06C1	ء 0621	آ 0622	أ 0623	ؤ 0624	إ 0625	ئ 0626	ا 0627	ب 0628	ة 0629	ت 062A	ث 062B	ج 062C	ح 062D	خ 062E	د 062F
D0	ذ 0630	ر 0631	ز 0632	س 0633	ش 0634	ص 0635	ض 0636	× 00D7	ط 0637	ظ 0638	ع 0639	غ 063A	ـ 0640	ف 0641	ق 0642	ك 0643
E0	à 00E0	ل 0644	â 00E2	م 0645	ن 0646	ه 0647	و 0648	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ى 0649	ي 064A	î 00EE	ï 00EF
F0	ً 064B	ٌ 064C	ٍ 064D	َ 064E	õ 00F4	ُ 064F	ِ 0650	÷ 00F7	ّ 0651	ù 00F9	ْ 0652	û 00FB	ü 00FC	LTR 200E	RTL 200F	ے 06D2

Figure 2.4: A Windows-1256 encoding scheme.

2.4.2 Windows-1256

Windows-1256 is an 8-bit character scheme used with Microsoft Windows to write Arabic and other languages that use Arabic script, such as Kurdish and Persian. This results in Windows-1256 being used more widely than the ISO scheme. In addition, Windows-1256 is not compatible with the ISO scheme. It encodes more forms of Arabic characters but, as in the ISO encoding scheme, it does not represent supplementary characters in Arabic, as shown in Figure 2.4.

2.4.3 UTF-8 encoding

The UTF-8 encoding scheme has become the most popular character encoding method on the web and in applications such as Google, Twitter, Facebook, Yahoo and YouTube (BuiltWith, 2012). Figure 2.5 shows the percentage of websites using various character encoding schemes, which clearly indicates that UTF-8 is the predominant method of encoding (W3Techs, 2013).

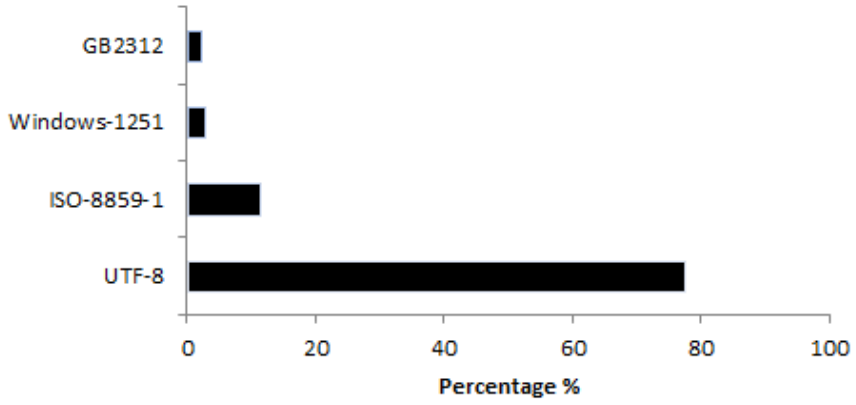

Figure 2.5: A UTF-8 scheme versus various encoding schemes (W3Techs, 2010).

UTF-8 is a multi-byte encoding scheme. It uses ASCII code (0-127) to represent a Latin character in only one byte, and then it uses 1 to 4 bytes for other languages. UTF-8 encodes each letter form in the Arabic language plus the supplements characters, which, as an encoding scheme, gives it precedence over other encoding methods and widens its use in many software applications and operating systems like the Apple Mac, and Linux and on websites. The importance of Unicode derives from its compatibility with ASCII, as it encodes English letters with single-byte characters, along with the rest of the 7 bit ASCII code. Unicode also derives importance from its compactness and efficiency in most scripts that require more than one byte to encode, such as Arabic, Japanese and Chinese.

2.5 Theoretical Models for Arabic Text

In this section, we will review some theoretical models of Arabic text. We will present statistics regarding the frequency of characters and words in order to discover the characteristics of Arabic text. Also, statistics regarding n-graphs (character sequences of length n) and word length are examined for the same purpose. Type-token relations and the growth of vocabulary in Arabic text are reviewed. Finally, we will show how Zipf's law (Zipf, 1949) can be used to predict the number of word occurrences for a given size of Arabic text.

2.5.1 Some Characteristics of Arabic Characters

Arabic has 28 basic letters and another six forms are derived from some of those letters. Regarding character frequency, this study aims to examine how many times a letter occurs in a given text. For example, if we look to the following sentence, "بسم الله الرحمن الرحيم", it consists of four words and a total of 19 characters (without spaces).

Each character has occurred once or more; for example, "ب" occurs once, "س" has the same frequency of occurrence and "ا" occurs three times. If we wish to calculate the character percentage of "ا" and "ب", we can do it by using the following formula.

$$character\ \% = \frac{number\ of\ times\ character\ occurs}{total\ number\ of\ all\ characters} \times 100. \qquad (2.1)$$

Using the given formula to calculate the frequency of "ا" in the sentence above will result in (3/19)*100 = 15.78%. For the "ب" character, the percentage will be 5.26%. Table 2.2 examines the character percentage of three Arabic texts: the Holy Qur'an, the Essex Arabic Summaries Corpus (EASC) (El-Haj et al., 2010) and the Corpus of Contemporary Arabic (CCA) (Al-Sulaiti and Atwell, 2006).

Table 2.2: Character-frequency distribution for three Arabic texts

Holy Qur'an				EASC Corpus				CCA Corpus			
Rank	Character	frequency	%	Rank	Character	Frequency	%	Rank	Character	Frequency	%
1	space	83974	18.89	1	space	58049	16.79085	1	space	600048	17.65097
2	ا	44021	9.90	2	ا	42510	12.29615	2	ا	408047	12.00308
3	ل	38754	8.72	3	ل	32794	9.485766	3	ل	316184	9.300845
4	ن	27420	6.17	4	ي	20867	6.035844	4	ي	206116	6.063093
5	م	27109	6.10	5	م	17837	5.159407	5	م	169647	4.990324
6	و	24945	5.61	6	و	15072	4.359623	6	ن	144308	4.244953
7	ي	22104	4.97	7	ن	14045	4.06256	7	و	134891	3.967944
8	ه	14967	3.37	8	ت	12944	3.744092	8	ت	128283	3.773563
9	ر	12782	2.88	9	ر	12554	3.631283	9	ر	122193	3.59442
10	ب	11616	2.61	10	ب	9905	2.865052	10	ب	94217	2.77148
11	ك	10504	2.36	11	ة	9261	2.678773	11	ع	94010	2.765391
12	ت	10485	2.36	12	ع	9173	2.653319	12	ة	86091	2.532447
13	ع	9422	2.12	13	د	8148	2.356834	13	د	81853	2.407782
14	أ	9123	2.05	14	ف	6815	1.97126	14	ه	80867	2.378778
15	ف	8769	1.97	15	س	6670	1.929318	15	ف	67179	1.976133
16	ق	7055	1.59	16	ه	6659	1.926136	16	أ	63316	1.862499
17	س	6253	1.41	17	ك	5985	1.73118	17	س	61919	1.821405
18	د	6005	1.35	18	ق	5623	1.62647	18	ق	57420	1.689062
19	إ	5082	1.14	19	أ	5352	1.548083	19	ك	52643	1.548543
20	ذ	4933	1.11	20	ح	5184	1.499488	20	ح	51800	1.523745
21	ح	4382	0.99	21	ج	3692	1.067922	21	ج	39149	1.151604
22	ج	3329	0.75	22	ط	2702	0.781562	22	ص	26105	0.767903
23	ى	2591	0.58	23	ى	2659	0.769124	23	ى	25606	0.753224
24	ة	2527	0.57	24	ص	2615	0.756397	24	ش	25007	0.735604
25	خ	2500	0.56	25	ش	2498	0.722554	25	ط	24954	0.734045
26	ش	2132	0.48	26	خ	2227	0.644167	26	إ	24345	0.716131
27	ص	2079	0.47	27	إ	2079	0.601357	27	خ	23376	0.687627
28	ض	1687	0.38	28	ذ	1854	0.536275	28	ذ	22665	0.666712
29	ز	1607	0.36	29	ض	1829	0.529044	29	ث	18278	0.537664
30	ء	1583	0.36	30	ث	1794	0.51892	30	ض	17451	0.513337
31	آ	1510	0.34	31	ز	1685	0.487391	31	ز	15257	0.448799
32	ث	1418	0.32	32	ئ	1155	0.334087	32	غ	11791	0.346843
33	ط	1280	0.29	33	غ	1072	0.310079	33	ئ	10249	0.301484
34	غ	1224	0.28	34	ء	1009	0.291856	34	ء	9921	0.291835
35	ئ	1215	0.27	35	ظ	665	0.192353	35	ظ	6360	0.187085
36	ظ	853	0.19	36	آ	283	0.081859	36	آ	3782	0.111251
37	ؤ	676	0.15	37	ؤ	262	0.075784	37	ؤ	3035	0.089277

Characters such as "space", "ا", "ل" and "ن", which are A, L and N in Latin script, have the highest percentage in all texts, while the characters "ط", "ظ" and "غ", which are TTA, THA and GHA in Latin script, have a significantly lower percentage.

The following figure plots the character percentage in three Arabic texts.

The Holy Qur'an, EASC and CCA corpora and their frequency percentages are ranked in alphabetical order where [ ] represents the space character.

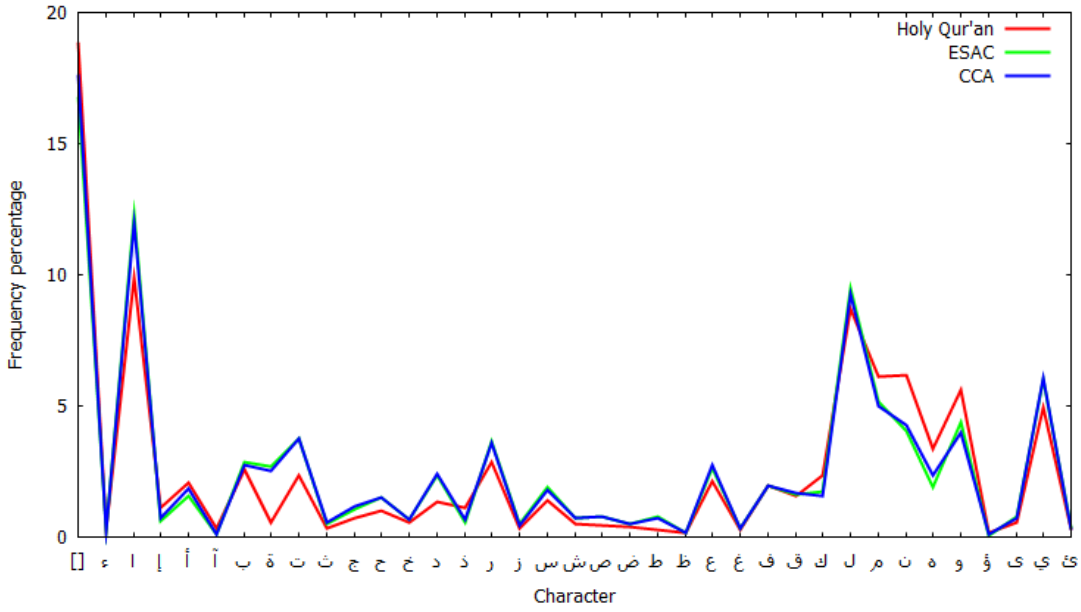

Figure 2.6: Character-frequency distribution from the Holy Qur'an, EASC and CCA corpora.

Figure 2.6 shows that space is the most frequently used in all three corpora. Also, there is almost a match between the CCA and EASC corpora in the character percentage, despite the fact that there is a big difference in the number of words between these corpora, where the EASC has 58973 words and the CCA has 583945 words (without punctuation marks). At the same time, it shows a large mismatch in the character percentages between both the CCA, EASC and Holy Qur'an, which can indicate a change between modern Arabic texts and classic Arabic text, as the CCA and the EASC represent modern text and the Holy Qur'an represents a classic text.

For example, characters such as "ة ت and د" occur less frequently in the Holy Qur'an than in the CCA and EASC corpora. Also, characters such as "و ن م" occur more frequently in the Holy Qur'an more than in the CCA and EASC corpora. These differences are due to the growth of vocabulary size being used in modern Arabic compared to the vocabulary used in the Holy Qur'an.

The top 20 most frequent n-graphs statistics found in the CCA are listed

in the following table in order to explore the frequency of these graphs and its representation in the text.

Table 2.3: Character statistics from the CCA corpus.

Rank	Bigraphs	Frequency	%	Trigraphs	Frequency	%	4-graphs	Frequency	%
1	ال	149501	5.480	الم	24144	0.885	التي	4118	0.151
2	من	21918	0.803	وال	11854	0.434	الذي	3195	0.117
3	في	21668	0.794	الت	11636	0.426	والعا	2629	0.096
4	وا	18746	0.687	الأ	11289	0.414	العر	2600	0.095
5	عل	13687	0.502	الع	11237	0.412	المس	2210	0.081
6	ما	11429	0.419	على	7938	0.291	والم	1811	0.066
7	أن	11320	0.415	إلى	6597	0.242	المت	1798	0.066
8	لا	11176	0.410	الس	6460	0.237	المر	1789	0.066
9	ية	10991	0.403	الح	6097	0.223	الأم	1766	0.065
10	ات	10962	0.402	الب	5399	0.198	المع	1679	0.062
11	با	10062	0.369	بال	5302	0.194	السي	1600	0.059
12	لم	9433	0.346	الق	4780	0.175	المن	1567	0.057
13	ها	9307	0.341	الن	4667	0.171	العل	1542	0.057
14	إل	8558	0.314	الإ	4455	0.163	والت	1394	0.051
15	ار	8493	0.311	الج	4251	0.156	المو	1317	0.048
16	بي	8435	0.309	الش	4185	0.153	الأس	1201	0.044
17	ان	8323	0.305	الد	4045	0.148	عليه	1200	0.044
18	ين	8196	0.300	الذ	3923	0.144	الإن	1175	0.043
19	عا	7693	0.282	الا	3854	0.141	الإس	1171	0.043
20	ري	7518	0.276	كان	3717	0.136	الأو	1160	0.043
total		367416	16.667		145830	5.345		36922	1.353

The top 20 bigraphs represent about 17% of the CCA corpus characters, trigraphs represent 5.3% and 4-graphs represent 1.4%, which indicates that Arabic words consist of many repeated bigraphs. Many bigraphs occur more frequently than some unigraph characters, such as "ظ" and "ث". There is a remarkable change in the percentage between the first bigraph "ال", which means "the", and the next bigraph, "من", which means "from". The difference is about 5% which indicates some bigraphs occur much more frequently than others.

2.5.2 Some Characteristics of Arabic Words

Unlike for English, there has been no extensive study of words and their frequencies for The Arabic language so far. The importance of these studies is to provide a proper understanding of the language structure. Table 2.4 shows the top 20 most frequent words and their frequencies from three Arabic texts, the CCA, the EASC, and the Holy Qur'an texts, respectively. In this analysis we excluded punctuation marks and full text was examined. As Arabic is naturally segmented language by spaces, words are easily identified by these spaces.

Table 2.4: The top 20 most frequent words from three corpora.

CCA Corpus				EASC Corpus				Holy Qur'an			
Rank	Word	Frequency	%	Rank	Word	Frequency	%	Rank	Word	Frequency	%
1	في	17454	2.997	1	في	1899	3.216	1	من	2766	3.520
2	من	15678	2.692	2	من	1579	2.705	2	الله	2267	2.885
3	على	7873	1.352	3	على	843	1.428	3	في	1187	1.510
4	ان	7388	1.268	4	و	713	1.208	4	ما	1011	1.286
5	إلى	6594	1.132	5	إلى	587	0.994	5	ان	966	1.229
6	التي	4025	0.961	6	ان	432	0.732	6	لا	813	1.034
7	عن	3300	0.566	7	التي	383	0.649	7	الذين	812	1.033
8	ما	3132	0.537	8	أو	296	0.501	8	على	670	0.852
9	لا	3125	0.536	9	عن	293	0.496	9	لا	665	0.846
10	هذا	2967	0.509	10	هذه	218	0.369	10	ولا	660	0.840
11	هذه	2779	0.477	11	الذي	190	0.322	11	وما	646	0.822
12	الذي	2580	0.443	12	ما	178	0.301	12	ان	640	0.814
13	أو	2564	0.440	13	مع	170	0.288	13	قال	417	0.530
14	و	2224	0.381	14	بين	167	0.283	14	إلى	405	0.515
15	كان	2084	0.357	15	هذا	164	0.278	15	لهم	374	0.476
16	مع	1728	0.296	16	كان	161	0.273	16	يا	349	0.444
17	لم	1665	0.285	17	عام	159	0.269	17	ومن	342	0.435
18	كل	1654	0.284	18	كما	145	0.246	18	ثم	341	0.434
19	ذلك	1644	0.282	19	حيث	141	0.239	19	لكم	337	0.428
20	بين	1540	0.264	20	هي	137	0.232	20	به	328	0.417
Overall			16.059				15.029				20.35

The relationship between word length and frequency has been explored earlier as a noticeable correspondence (Miller et al., 1958). For instance, the average word length of the top ten frequent words in the CCA corpus, which makes up to 14% of the total number of words in CCA, is 2.5 letters. The longest words in the CCA are 16 characters, with the average word length being 4.6 characters.

The following figure presents the word lengths and the percentages of these words in the CCA, the EASC as modern Arabic texts and the Holy Qur'an as a classic Arabic text to examine the variety of word length among these texts.

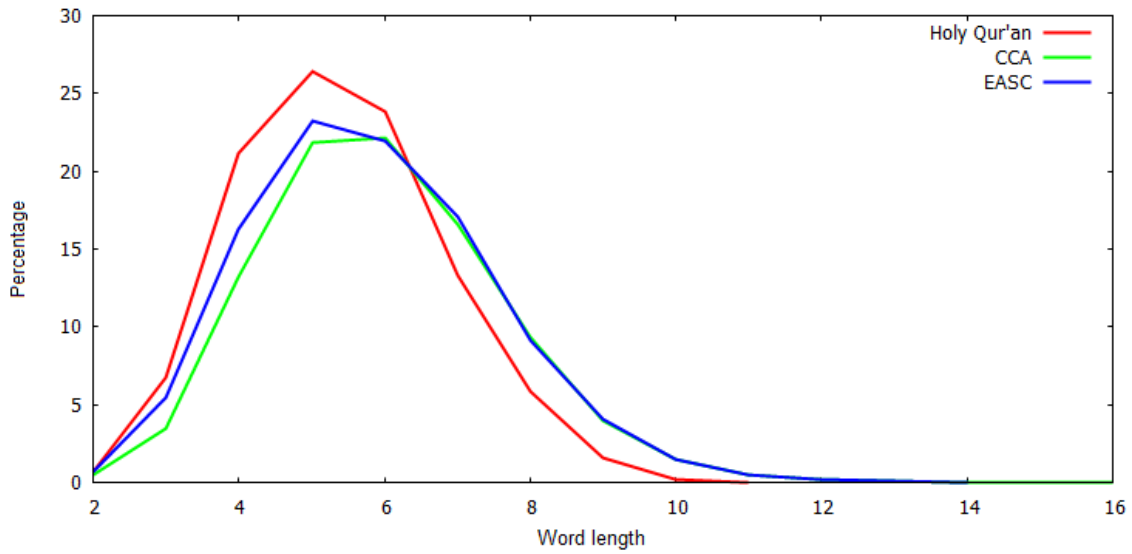

Figure 2.7: Word length of 3 Arabic text.

The longest word in the Holy Qur'an is 11 characters in length, while the longest word in the CCA is 16 characters and 14 characters in the EASC. Also, the Holy Qur'an has between 1% and 8% more vocabulary using word lengths of two through six characters, than the CCA and the EASC. The CCA and the EASC have between 1% and 4% more vocabulary with word lengths between seven and eleven characters than the Holy Qur'an. This indicates that modern Arabic uses longer word lengths than classical Arabic texts. For word lengths of six or more, there are matching vocabulary sizes used in the CCA and the EASC.

2.5.3 The Type-Token Relationship

Another way to explore theoretical models of Arabic text is the type-token relation for the text. Table 2.5 represents the vocabulary distribution of words found in the CCA corpus, a modern Arabic text. The words are listed by most frequently occurring word where types represent the number of unique words in the CCA and ordered by the frequency of these types over the whole text. The fraction of types is calculated by dividing the number of types in each row over the total number of types in the CCA, while the fraction of tokens is calculated by dividing the tokens over the total number of words of the whole text.

Table 2.5: Vocabulary distribution for the CCA corpus.

Types	No. of Types	Fraction of Types %	No. of Tokens	Fraction of Tokens %
في	1	0.001	17447	2.991
في من	2	0.002	33073	5.67
في من على	3	0.003	40944	7.019
في من على أن إلى	5	0.005	54881	9.408
في من على أن إلى التي عن ما لا هذا	10	0.011	71414	12.242
في من على أن إلى التي عن ما لا هذا هذه الذي أو و كان مع لم ذلك كل بين	20	0.021	92130	15.794
في من على أن إلى التي عن ما لا هذا هذه الذي أو و كان مع لم ذ لك كل بين هو بعد كما قد إن حتى وقد العربية كانت	50	0.053	121739	20.87
في من على أن إلى التي عن ما لا هذا هذه الذي أو و كان مع لم ذلك كل بين هو بعد كما قد إن حتى وقد العربية	100	0.106	148425	25.444
من على أن إلى التي عن ما لا هذا هذه الذي أو و كان مع لم ذلك كل بين هو بعد كما قد إن حتى وقد العربية كانت هي عام العالم وفي وهو ثم بعض حيث له	93972	100	583329	100

For example, the list in the third row contains the three most frequently occurring words "في من على" with the number of types and tokens found in the text and their percentage as a fraction of the whole text. This table reveals two obvious tendencies. First, a small number of words occur more frequently. Second, large quantities of words occur rarely. For example,

the top 100 most frequent words represent about 30% of the text, but it represents only 0.11% of the vocabulary. The following figure shows the type-token relationship, which clearly graphs the tendencies.

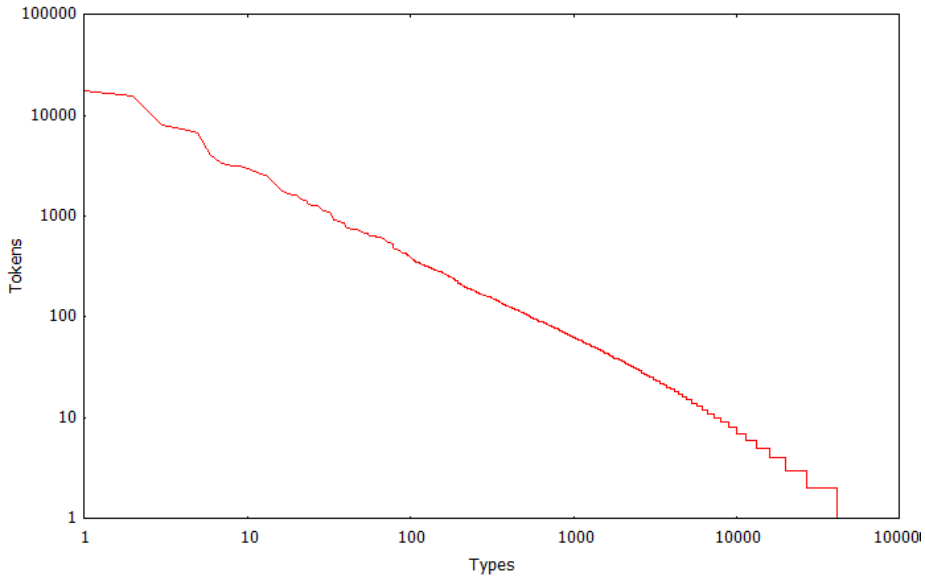

Figure 2.8: A type-token plot for the CCA corpus.

Table 2.5 can be rearranged into type-token data, as in the following table where the words with the same number of tokens are placed together with its token count (first column).

Table 2.6: Type-token data for the CCA corpus.

Tokens	Word	Types
1	ويظل وفي الخامس الإنسان ..... ءامنوا	52549
2	يهان ينسب ينسى يلقاها يعطى ..... آب	14596
10	ينهض ينشأ ينتبه يلزم يلزم يقطع ..... أؤمن	753
100	لبنان قطاع صناعية زمن رجال ..... الانسانية	8
441	العمل	1
1728	مع	1
2964	هذا	1
7343	أن	1
17447	في	1

Referring to the table, for example, one word occurs 1728 times, "مع" which means "with"; "هذا" which means "this", is the only word that occurs 2964 times; "أن", which means "that", is the only word that occurs 7343 times; "في", which means "in", is the only word that occurs 17447 times;

and there are 52549 singleton words, words that occur one time only (the words listed in the first row).

2.5.4 Vocabulary Growth

For Arabic, there is a noticeable growth of the size in the vocabulary as the text expands. For example, in the EASC corpus of 58973 words, there are 20514 word types. In contrast, the larger corpus CCA has 583945 words, and there are 93972 word types. This initially indicates two things. First, there is a relationship between text size and vocabulary growth; second, vocabulary growth diminishes as text expands. According to Heaps' law (Heaps, 1978), this relationship is as follows.

$$V = K \times n^{\beta}. \tag{2.2}$$

where V is vocabulary size (number of word types), n is the number of words in the text, K and β are the parameters that vary for each text. Typical values given are $10 \leq K \leq 100$ and $\beta \approx 0.5$. For Arabic text we found that typical values are between 10 and 20 for K and between 0.6 and 0.7 for β where K =20 and β =0.7 for the Holy Qur'an as classic text and K =10 and β =0.6 for both CCA and EASC corpora as modern text. Experiments with these texts are shown in Figure 2.9 which shows that Heaps' observation generally holds true, as shown by the dashed lines.

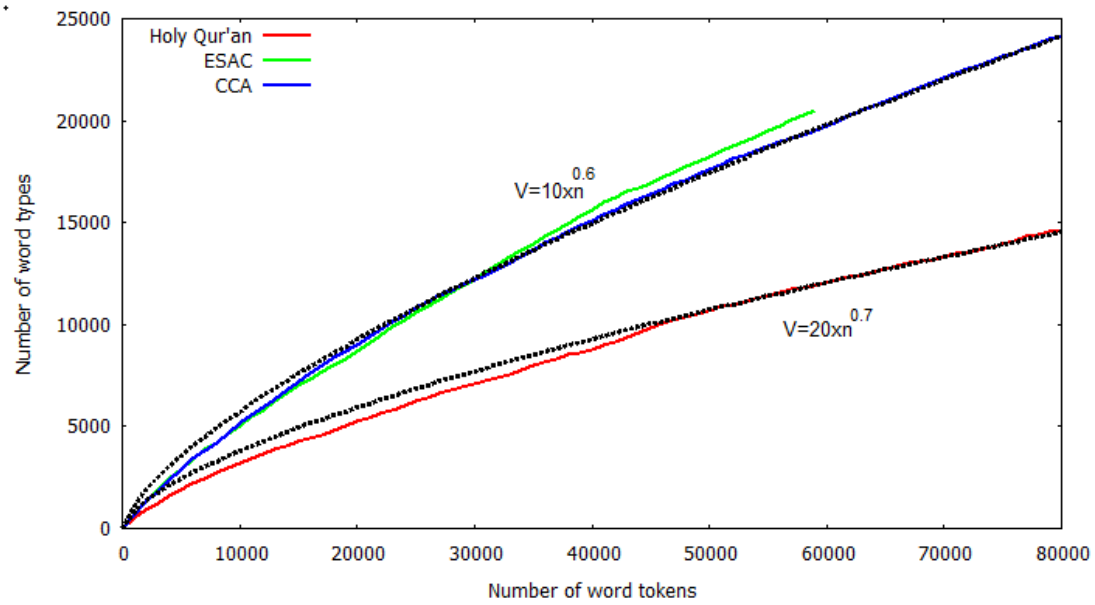

Figure 2.9: Vocabulary growth of several texts.

2.5.5 Zipf's Law

Zipf's law was introduced by George Zipf (Zipf, 1949). It is an empirical law based on mathematical statistics from different samples. It is simply a record of the observation of frequency that occurs in some events. The power of this law is that it can be applied to many observational data. Here, we are talking about languages and the frequencies of words and characters. We can use Zipf's law to examine the frequency distribution for characters in a given text. In other words, we can estimate, in reference to Zipf's law, how often words or characters occur. The formula for Zipf's law is as follows:

$$f(r) \times r = C. \tag{2.3}$$

where f is the frequency of the character or the word in a given text, r is the rank of the word according to the frequency f of its occurrence and C is a constant number. For example, the following figure shows using logarithmic scales the rank on the X-axis and the frequency on the Y-axis for the three corpora. This graph shows two things: high ranking accounts for a large percentage of the text, and there is a long tail of words that occur seldomly.

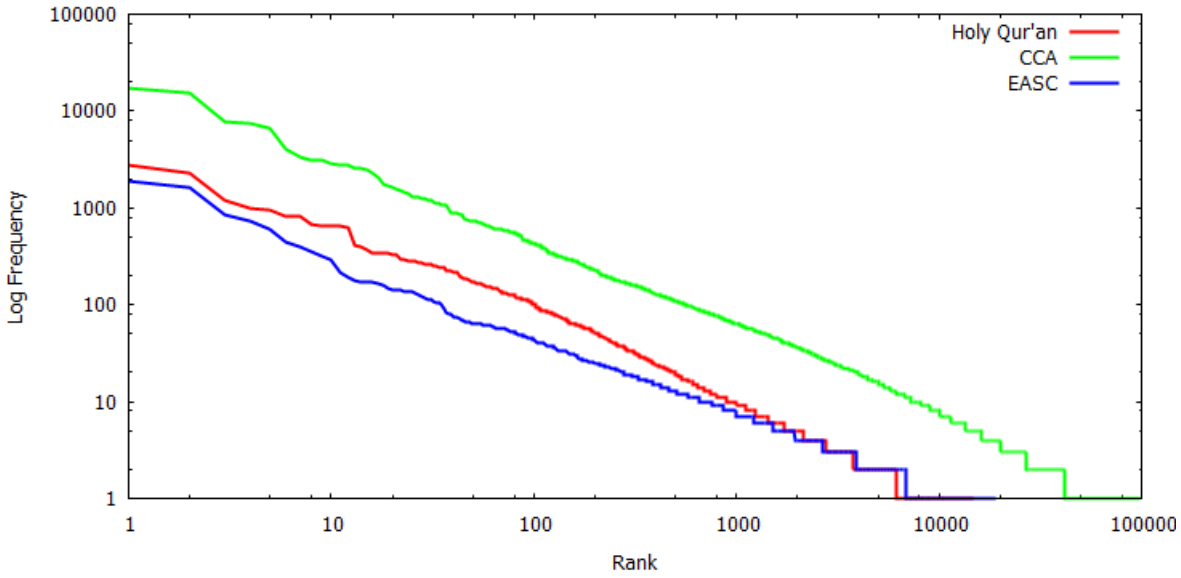

Figure 2.10: Zipf's Law applied to words for three Arabic corpora.

These findings indicate that Arabic text, at least initially, follow Zipf's law.

2.6 Conclusion

We have reviewed several basic fundamental characteristics of the Arabic language and its encoding schemes. There are some similarities between English and Arabic in terms of the size of the alphabets, and there are some differences, such as the direction of writing and in the morphological nature of the languages. Also, we have reviewed some characteristics of Arabic characters and words. We found that the Arabic language has a noticeable growth in the size of the vocabulary as the text expands and also it follows Zipf's law, as shown in Section 2.5.5.

Chapter 3

New Arabic Language Corpora

Contents

3.1 Introduction

Corpora are sets of structured text stored and processed for purposes of statistical analysis that are usually stored in a machine-readable form (Teahan, 1998). It may contain more than one language. Corpora are used in research that relates to linguistic areas such as part of speech tagging, machine translation, word segmentation and text compression, and it also could be a useful resource for teaching purposes (Alsuliti, 2004). In our research, we used corpora in order to standardize the results of the experiments for text compression.

In this chapter, we will review some available Arabic language corpora and then we will introduce the Bangor Arabic Compression Corpus (BACC) (Teahan and Alhawiti, 2013), designed for experimental usage and standardizing the result of compression algorithms. Also, we will introduce the Bangor Balanced Corpus of Contemporary Arabic (BBCCA) of one million words, designed for natural language processing research. Finally, we will describe a new method to evaluate corpora by ranking n-gram differences between texts. We propose this as a novel corpora assessment tool and also as a means to reveal different topics between two or more texts (Teahan and Alhawiti, 2014). We use this new method to evaluate the quality of the BACC corpus.

3.2 Existing Arabic Language Corpora

Unlike English, Arabic has a comparatively smaller number of available reference corpora, such as the Brown corpus (Francis and Kucera, 1979) and the LOB corpus (Johansson, 1980) for English. This results in NLP research for the Arabic language being a harder task for researchers. The next section reviews some available Arabic corpora with their structure and statistics.

3.2.1 Corpus of Contemporary Arabic (CCA)

This corpus was collected by Latifa Alsuliti and Eric Atwell at the University of Leeds (Al-Sulaiti and Atwell, 2006) and is available at `www.comp.leeds.ac.uk/eric/latifa/CCA_raw_utf8.txt`. For the written portion, it is derived mainly from websites, while the spoken part was collected from Radio Qatar in the form of five small files selected from a set of 415 text files. CCA has 598718 words overall, in the following categories: autobiography, short stories, children's stories, economics, education, health, medicine, interviews, politics, recipes, religion, sociology, science, sports, and travel.

The overall file size of the CCA is 6289509 bytes encoded using the UTF-8 scheme. In terms of file size, CCA is a small corpus created for teaching purposes and a larger corpus is required for compression purposes. What is required in order to investigate and emphasise the impact of the compression methods is to categorise experiments into a group of files based on size (small, medium, large and very large). Therefore, the corpora used must contain different file sizes. However, CCA is excellent as a training corpus for small text experiments, as it contains variant genres as well.

3.2.2 Essex Arabic Summaries Corpus (EASC)

The EASC corpus was created by El-Haj et al. (2010) at the University of Essex. It contains 153 Arabic articles and 765 human-generated extractive summaries of those articles. It is a written text-based corpus that consists of the following categories: art and music, education, environment, finance, health, politics, religion, science, sport and tourisms.

The overall file size in the EASC is 630321 bytes encoded using a UTF-8 scheme. The EASC corpus is also small in terms of file size. It has been created for the purposes of text summarization; therefore, we cannot consider it as a good corpus for compression purposes.

3.2.3 Arabic Treebank Corpus (ATC)

The ATC is one of the Linguistic Data Consortium (LDC) corpora (Maamouri et al., 2005). It was released in early 2005. The ATC is also a small corpus of 145386 words for the purpose of content extraction, information retrieval and part of speech tagging. The data of the corpus was extracted from the Agence France Presse (AFP) newswire between the period of July and November 2000. For the same reasons as stated above, we cannot consider the ATC as a good training corpus for compression purposes based on the need for both size and variation in the text.

3.2.4 King Saud University Corpus of Classical Arabic (KSUCCA)

The KSUCCA is a 50-million-word corpus (Alrabiah et al., 2013) with the main purpose of studying the lexical semantics of the Holy Qu'ran. The KSUCCA is a selection of classical Arabic texts from the period of the seventh until the early eleventh century. The categories in the KSUCCA are religion, linguistics, literature, science, sociology and biography. Since the KSUCCA is designed mainly for lexical studies of classical Arabic text during the pre-Islamic era, it cannot be considered a good training corpus for compression experiments, as it does not represent the modern text of the Arabic language.

3.3 New Arabic Language Corpora

In order to standardize our compression experiments and other natural language processing applications, we have set up two new corpora, which are the Bangor Arabic Compression Corpus (BACC) (Teahan and Alhawiti, 2013) and the Bangor Balanced Corpus of Contemporary Arabic (BBCCA), to overcome the shortage of Arabic language resources. These corpora have been made freely available for research purposes. These two corpora will now be described in more detail.

3.3.1 Bangor Arabic Compression corpus (BACC)

A 31-millon-word corpus called the Bangor Arabic Compression Corpus (BACC) was created for this study (Teahan and Alhawiti, 2013). The corpus contains text files of different genres, such as sports, culture, economics and so forth, which were collected from many sources including websites, magazines and books to ensure a variety of texts. Also, the size of the chosen files was considered as an important target in designing the BACC. This will allow for a variety of compression experiments on different file sizes and genres in terms of compression performance and execution speed. The purpose of this new corpus was to create a benchmark corpus for compression experiments on Arabic text.

The BACC consists of four sub-corpora in terms of the text file size (small, medium, large and very large), and comprises 16 text files of different genres encoded using the UTF-8 encoding schemes, as described in table 3.1.

Table 3.1: BACC corpus.

File name	Categorize	Size in Byte	Description
economic	Small	15924	Economics & finance
education	Small	27086	Education
sports	Small	31706	Sport
culture	Small	34760	Culture issues
artandmusic	Small	42648	Art and music
political	Small	47556	Political
articles	Medium	103839	Articles
press	Medium	549063	Press
novel1	Medium	860680	Qarytona
novel2	Medium	912604	Wayzhr Alqondol
novel3	Medium	1023987	Afrah Lilat Alqdr
shortstories	Medium	1041952	Adventure and horror
literature	Large	19187425	Literature and heritage
history	Large	30714551	History
bookcollection1	Large	56633170	Popular lore
bookcollection2	Very large	201693734	Collection of books (Religion)

The small corpus consists of files that are under 100 KB, while the medium corpus consists of files that are larger than 100 KB but smaller

than 18000 KB. The large corpus consists of files that are larger than 18000 KB but smaller than 190000 KB and the very large corpus consists of files that are larger than 190000 KB.

In the BACC, we take into consideration the representation of modern Arabic as well as classical Arabic, which can be found in the religious and literature material and should be included in compression experiments. The BACC was collected with permission from the owners of the material used except material that referred to historical times, such as religious and popular lore texts. To our knowledge, the BACC is the first Arabic corpus designed for compression purposes.

As an additional component to the BACC corpus, we have also produced a Stem-Segmented Small Bangor Arabic Corpus (SSSBAC) comprising 50,000 words that has been manually segmented by morphology to ensure high quality. The purpose of this corpus is to segment an Arabic word that could be found in any text into the form *prefix(es)-stem-suffix(es)* which can be very useful to many NLP applications (see section 2.3). For example, the rich morphological nature of the Arabic language presents many challenges to the machine translation applications of Arabic (Soudi et al., 2012). Samples of frequent prefixes and suffixes are shown in Table 3.2.

Table 3.2: Samples of prefixes and suffixes found in Arabic text.

Prefix	Stem	Suffix
و	بيت	ه
ب	بيت	هم
ب + ال	مسلم	ين
و + ب + ال	مسلم	ون

As we can see from the table, more than one prefix and/or suffix could be attached to an Arabic word, which can make the segmentation process not such a trivial task. A good solution is training a statistical language model, such as PPM models, on SSSBAC to learn about the probability of

these prefixes and suffixes (Teahan et al., 1998).

3.3.2 Bangor Balanced Corpus of Contemporary Arabic (BBCCA)

The Bangor Balanced Corpus of Contemporary Arabic (BBCCA) is a compilation of one million words of contemporary Arabic text. The BBCCA is the first balanced corpus using the same format (structure and size) as the Brown Corpus (Francis and Kucera, 1979). The Brown Corpus has been a template that has been copied by many other English based corpora, such as the LOB (Johansson, 1980) and the Macquarie and Wellington corpora. The purpose of this corpus is to mirror similar balanced corpora that are available for the English language (Brown and LOB) but instead include Arabic language. The BBCCA is divided into 500 samples of more than 2000 words for each. The sampling criteria in the BBCCA corpus is described as follows.

(A) PRESS: REPORTAGE (44 texts)

(B) PRESS: EDITORIAL (27 texts)

(C) PRESS: REVIEWS (17 texts)

(D) RELIGION (40 texts)

(E) SKILL AND HOBBIES (36 texts)

(F) POPULAR LORE (48 texts)

(G) BELLES-LETTRES (52 texts)

(H) MISCELLANEOUS: GOVERNMENT & HOUSE ORGANS (30 texts)

(J) LEARNED (80 texts)

(K) FICTION: GENERAL (29 texts)

(L) FICTION: MYSTERY (24 texts)

(M) FICTION: SCIENCE (6 texts)

(N) FICTION: ADVENTURE (29 texts)

(P) FICTION: ROMANCE (29 texts)

(R) HUMOR (9 texts)

BBCCA is a corpus of standard, modern, printed Arabic text written in the last few years. Some of the samples, such as religion and popular lore texts, were written earlier due to limitations in these categories in Arabic. Books, journals, magazines and websites were the sources of the collected materials. Most of the samples of the corpus were chosen from different authors to ensure a wide range of vocabulary being used in the selected sample. The selection of the sample had two phases. First, the material was selected from several alternatives and then at least 2,000 words were selected randomly.

For the 2000+ selected words sample, we started with the beginning of a sentence in a paragraph or a new line and did not break up lines at the end of the selection to maintain the meaning of the selected text. BBCCA will be a good resource for POS tagging and text correction as it is represents modern Arabic use in printed text.

For copyright issues, these materials were collected with permissions from the copyright holders of these samples.

3.4 A Codelength Method for Ranking N-gram Diferences Between Texts

In this section, we describe a new method that we have developed in order to evaluate the quality of corpora. The method is based on ranking n-gram (unigrams, bigrams or trigrams) differences between texts. The method can also be used as the basis for producing tag clouds and is effective at revealing different topics between two or more texts.

Various methods have been devised for intelligently processing text documents. Fundamental to most of these methods is a requirement to rank a text document according to a metric. Traditionally, the metric is calculated from the whole document according to relevant criteria. For example, for vector-space-based information retrieval, a cosine metric is used to find the documents that are nearest to the user query, where documents and

queries are represented as points in an n-dimensional space (Manning et al., 2008).

An alternative to using ranking metrics based on processing the entire document is to use a feature-based approach, where features within the document (such as n-grams) are considered individually and ranking statistics are obtained separately for each feature. These are often either processed across the entire collection of documents or used to compare two documents (Teahan and Alhawiti, 2014). This method can be used as the basis for producing tag clouds and is effective in revealing different topics between two or more texts. The method has also been found to be effective in revealing trends and emerging topics.

This method uses a simple estimate for the probability of each n-gram, based on its frequency of use in each text:

$$P_T(g) = C_T(g)/N_T \tag{3.1}$$

where $P_T(g)$ is the probability of the n-gram g in the text T; $C_T(g)$ is the frequency of the n-gram, and N_T is the total number of n-grams of the same length (i.e. unigrams, bigrams or trigrams) in the text T. The relative entropy-based distance metric used for ranking the 'unusualness' of each n-gram g that appears in both texts, $T1$ and $T2$, is calculated as follows.

$$D_{T_1,T_2}(g) = |H_{T_1}(g) - H_{T_2}(g)| = |\log_2 P_{T_1}(g) - \log_2 P_{T_2}(g)| \tag{3.2}$$

where the entropy of the n-gram g is calculated as

$$H(g) = -log_2 P(g). \tag{3.3}$$

From a compression perspective, this measurement which is called 'codelength difference' is simply the absolute difference between compression codelengths and the cost of encoding the n-gram using two different

models, one trained on the text T_1 and the other trained on the text T_2. The codelength is a way of measuring the 'information' for an n-gram compared to the other n-grams.

For example, we can calculate the codelength for encoding the unigram word "Britain" in the Brown Corpus as follows.

$H_{Brown}(\text{“}Britain\text{”}) = -\log_2 P_{Brown}(\text{“}Britain\text{”}) = -\log_2(61/1014416) = 14.021.$

The word "Britain" occurs 61 times in 1,014,416 words. In contrast, the word "Britain" occurs 290 times in 1,010,401 words in the Lancaster-Oslo-Bergen (LOB) (Johansson, 1980).

$H_{LOB}(\text{“}Britain\text{”}) = -\log_2 P_{LOB}(\text{“}Britain\text{”}) = -\log_2(290/1010401) = 11.767.$

We can use the absolute difference between the two codelength values as a means to measure how unusual the difference in probability is for the two corpora:

$D_{Brown,LOB}(\text{“}Britain\text{”}) = |H_{Brown}(\text{“}Britain\text{”}) - H_{LOB}(\text{“}Britain\text{”})| = |14.021 - 11.767| = 2.245.$

Table 3.3 lists the top 20 codelength difference values for words, bigrams and trigrams that appear in political files from the SBACC and CCA.

For these results, proper nouns, such as "المساجين المتظاهرين الدستور" (which mean prisoner, constitution, and demonstrators) appear more frequently in the SBACC than the CCA, as the CCA is a corpus created in 2006 before the Arab Spring started in Egypt and Tunisia. However, the SBACC was created in late 2011 after the revolutions in Tunisia, Egypt and elsewhere.

The differences between the SBACC and the CCA are more revealing when bigrams are used in the analysis. Here proper name bigram sequences are ranked highly, such as "ميدان التحرير" , "الحزب الوطني", "النيابة العامة" and "مجلس الشعب", which mean "Tahrir Square" in Cairo, "Alwatani party", "solicitor general" and "Parliament". Figure 3.1 shows the unigram tag chain produced using the codelength difference measurement of the SBACC and the CCA corpus. The top ranked unigrams according to the value $H_{SBACC} - H_{CCA}$ are shown with the red circles on the left, and, according

Table 3.3: The top 20 words, bigrams and trigrams that appear in both the SBACC and the CCA, ranked in descending order according to the differences in codelength.

D	Unigram	*D*	Bigram	*D*	Trigram
8.093	المساجين	8.366	ميدان التحرير	5.508	قرار وزير الداخلية
8.031	الانفلات	7.508	الحزب الوطن	5.508	، وهو ما
7.678	المسلحة	6.967	اللجنة الوطنية	5.315	و من ثم
7.637	الى	6.83	القانون رقم	5.093	ولا يجوز أن
7.552	المتظاهرين	6.83	الأمن القومي	5.093	و هو ما
7.552	أمس	6.678	غفر الله	5.093	لا تزيد عن
6.83	بميدان	6.678	النيابة العامة	5.093	في قطاع غزة
6.83	المستشار	6.508	من المتظاهرين	5.093	في عدة دول
6.83	السلمي	6.508	لسنة ٢٠٠٣	5.093	فلا بد أن
6.83	الانترنت	6.508	قوات الشرطة	5.093	تحت خط الفقر
6.83	الامني	6.508	أبو عبدالله	5.093	الشرق الأوسط وشمال
6.678	غفر	6.315	جلالة الملك	5.093	الحرمين الشريفين
6.678	دستور	6.245	وزير الداخلية	5.093	أقل من شهر
6.678	المصري	6.093	يوم ٢٥	5.093	؟ إذا كانت
6.678	الدستورية	6.093	وزير الخارجية	5.093	. و من
6.595	للقوات	6.093	من القانون	4.83	، إضافة إلى
6.595	انتخابات	6.093	على موقع	4.508	يمكن له أن
6.508	سلمية	6.093	الطرق الصوفية	4.508	يمكن إلغاؤها أو
6.508	بلال	6.093	التيار الكهربائي	4.508	يعيشون تحت خط
6.508	التظاهر	5.967	مجلس الشعب	4.508	وهو ما يخالف

to the value $H_{CCA} - H_{SBACC}$ are shown with the blue circles on the right, with the minimum codelength difference set at 3.0.

Figure 3.1 clearly shows the different bigram phrases commonly used for proper nouns found in the SBACC and the CCA. As a result, the different topics of interest that appear in the two respective texts are revealed effectively. Phrases with proper names, such as "المساجين المتظاهرين، بميدان" and "المستشار", which means consultant, demonstrator, square and prisoners, appear and clearly indicate the importance of the January revolution in Egypt and relates to the Arab spring. Alternatively, the phrases "العلماء، الشعر ، الحضارة" and "الإسلامية", which mean scientist, poetry, civilization and Islamic, are clearly related to civilization in Islamic culture.

Figure 3.2 shows the unigram tag cloud produced using codelength differences for the SBACC and the CCA. The tag cloud in this case was produced using the Processing language that automatically allocated the positions of the tags either horizontally or vertically in the visualisation. The figure provides an effective method for visualizing the data provided in

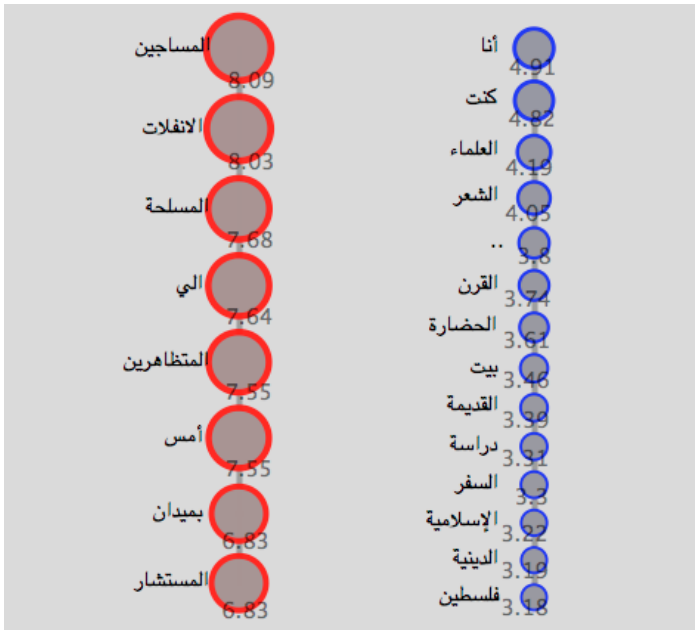

Figure 3.1: A unigram tag chain is produced using the codelength difference measurement of the SBACC and the CCA. The top ranked trigrams according to the value $H_{SBACC} - H_{CCA}$ are shown with the red circles on the left, and, according to the value $H_{CCA} - H_{SBACC}$, the unigram are shown with the blue circles on the right.

Table 3.3 and Figure 3.1. It clearly reveals the importance of such phrases as "بميدان ،المستشار" and "المتظاهرين".

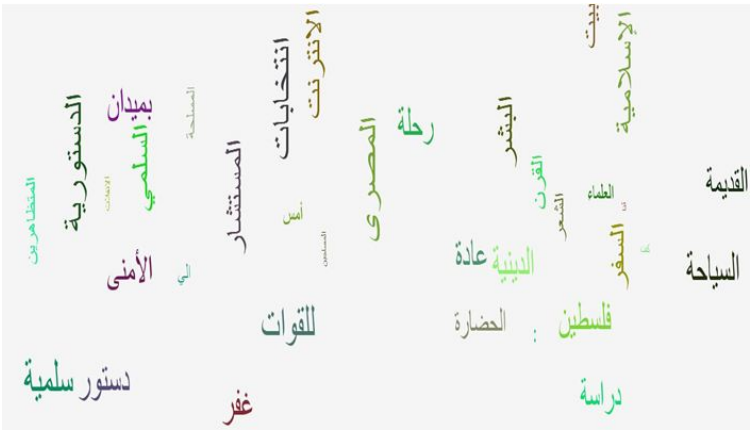

Figure 3.2: Unigram tag clouds are produced using the codelength difference measurement from the SBACC and the CCA. The minimum codelength difference threshold has been set at 3.0.

As another example, the yearly speeches of King Abdullah Bin Husain in Jordan (Court, 2013) were analyzed using the same method. The results are shown in Figure 3.3. From the tag cloud, we can see that in the early period of King Abdullah's rule (1999-2005), the king considered important issues, such as "الفساد ،الدستورية" and "المتطرفون" which mean corruption,

constitution and terrorism. However, for the period of 2006 through 2012, words such as "الاحتلال" and "التنمية", which mean occupation and development, become important. This is understandably due to the Gaza war in 2006 and demonstrations in Jordan because of rising unemployment among young Jordanian citizens and the influence of the Arab Spring, which makes development an important issue.

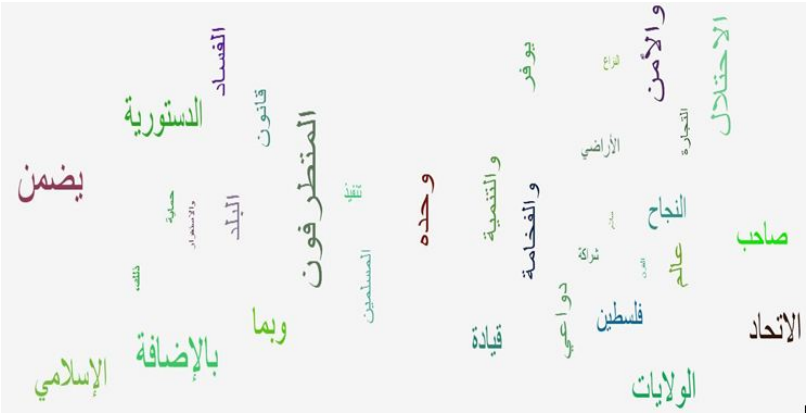

Figure 3.3: A unigram tag cloud produced using the codelength difference measurement in the last 14 years (1999-2012) of the speeches of King Abdullah Bin Husain in Jordan. The minimum codelength difference threshold has been set at 3.0.

3.4.1 Defining an N-gram Feature-Based Approach as a Metric for Corpora Content Evaluation

Evaluation of corpora is a non-trivial task. If we have a new corpus, how we can assess it for its quality? Against what criteria? The token frequency analysis (word and character) might be a way to do so. Alternatively, we could compare the new corpus with an existing one that has desirable properties or qualities that we are trying to reproduce. So we can examine similarities or differences between the two corpora being compared since token frequency analysis can only show count information (Rayson, 2009).

The following n-gram, feature-based approach is a new metric that has been devised as a tool for corpora comparison. The new metric is based on the relative entropy metric defined in the previous section. The corpus

evaluation metric is calculated using the following formula:

$$M_{C_1,C_2} = \frac{\sum_{i=1}^{n} D_{C_1,C_2}(g_i)}{n} \tag{3.4}$$

where M is the average codelength difference of the two corpora C_1 and C_2. D is the codelength difference of the n-gram, and n is the number of n-grams of the same length that occur in the two corpora C_1 and C_2.

If the value of $M_{C1,C2}$ is equal to zero, then the two corpora would be exactly the same. If the $M_{C1.C2}$ score was low (< 1), then the corpus being compared has much in common with the reference corpus. If the M score is high (> 1), it would indicate that the corpus being compared is quite different to the reference corpus. As another means for evaluating corpora, the tag clouds can also be used to visualize the top 100 unigram codelength differences between compared corpora.

We have found that the proposed techniques are useful in comparisons with well-known corpora, such as the CCA and the Brown corpora. In addition, the metrics of average codelength differences of the top 1, 5, 10 and 100 n-grams can also be used to illustrate the changes in codelength differences between compared corpora. Low codelength values indicate that the two corpora being compared have similar characteristics and high changes of these values indicate more differences. Also as a future technique, the percentage of novel words in both compared corpora can also be examined such as the sample of top 10 unseen words in the compared corpora.

In order to illustrate the techniques, the following table shows the comparison of the Brown corpus as the reference corpus vs. the LOB and the King James Bible corpora, using the proposed methods.

Table 3.4: Comparing corpora: Brown, LOB and the King James Bible.

Metric	Brown vs. LOB	Brown vs. King James Bible
Average codelength (M_{C_1,C_2})	0.84	1.84
Top1 unigram	7.65	10.52
Top5 unigram	6.94	10.05
Top10 unigram	6.38	9.72
Top100 unigram	4.88	7.54
Top1 bigram	8.52	11.25
Top5 bigram	6.68	10.23
Top10 bigram	5.94	9.38
Top100 bigram	4.17	7.19
Top1 trigram	5.4	9.56
Top5 trigram	4.77	8.68
Top10 trigram	4.6	8.17
Top100 trigram	3.58	6.3
% of words in 1st corpus not in 2nd corpus	6.05	8.73
% of words in 2nd corpus not in 1st corpus	5.54	2.94
Top 10 unseen words in 1st corpus	Mr. Mrs. Dr. Mr SIC J. St. Colour Fig. R.	LORD Shalt and was are hast Is Were be
Top 10 unseen words in 2nd corpus	Mr& **f Mrs& **h Program Dr& <The **f. Center St&	Has Mr& American Mrs& Around during United Does don't didn't

Since the Brown and LOB are balanced modern corpora of English texts, the M value = 0.84 indicates a strong similarity between these corpora, unlike the King James Bible, the M value is larger at 1.84, which indicates that there are differences between the Brown and King James Bible corpora. Also, the average codelength of n-grams is lower for Brown vs. LOB than Brown vs. King James Bible, which also indicates similarities between the Brown and LOB corpora and draws attention to the fact that there are important differences between the Brown and King James Bible corpora.

In addition, the percentage of novel words in the Brown corpus not seen in the LOB corpus was 6.05% and the percentage of novel words in the LOB corpus not seen in the Brown corpus was 5.54%, which is typical for English text that are similar. In contrast, the percentage of novel words in the Brown corpus not seen in the King James Bible is 8.73% versus

2.94 % of novel words in the King James Bible not seen in the Brown corpus. This is more than double, indicating major differences between these corpora. A sample of the top 10 unseen words in both corpora is listed in the last two rows in Table 3.4. A tag cloud of unigram codelength differences is shown in Figure 3.4 for Brown vs. LOB and for Brown vs. King James Bible in Figure 3.5.

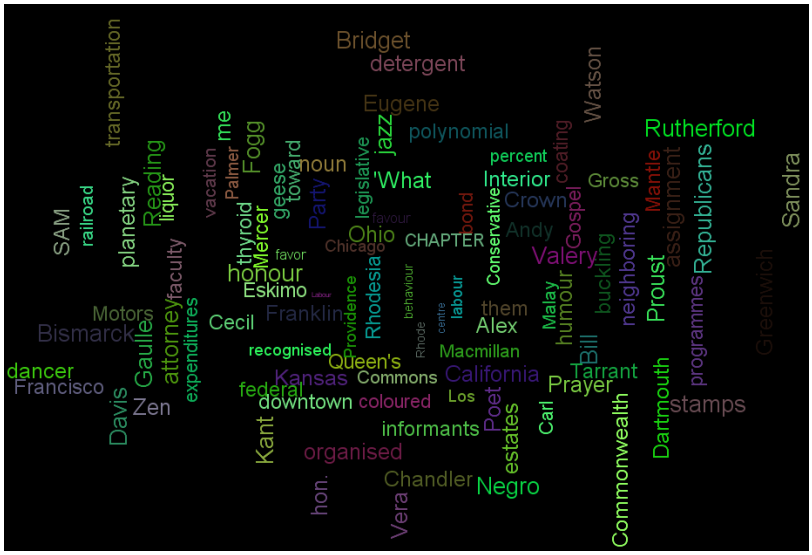

Figure 3.4: A tag cloud of unigram codelength differences of Brown vs. LOB corpora.

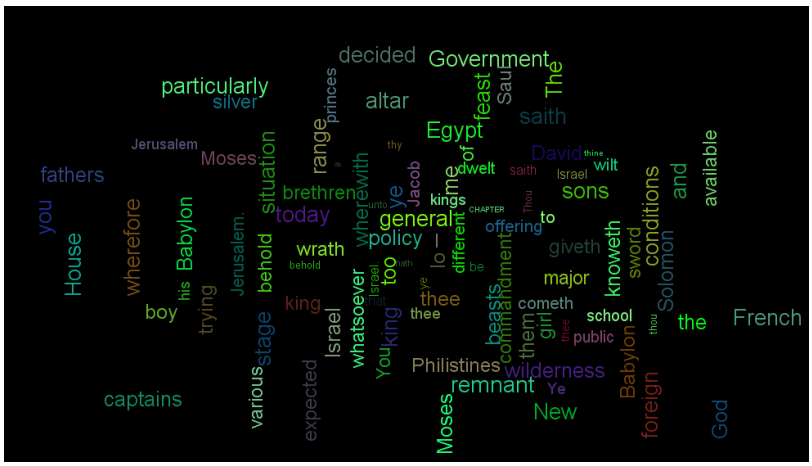

Figure 3.5: A tag cloud of unigram codelength differences of Brown vs. King James Bible.

We will now use the metric to evaluate our BACC corpus against the Arabic Bible corpus as an example of a classic Arabic text, and we will use the CCA corpus as an example of a modern Arabic text. Table 3.5 shows the comparison of SBACC as reference corpus to the CCA and Arabic Bible

corpora using the proposed metrics.

Table 3.5: Comparing corpora: SBACC, CCA and Arabic Bible corpora.

Metric	SBACC vs.CCA	SBACC vs. Arabic Bible
Average codelength (M_{C_1,C_2})	0.9	1.5
Top1 unigram	6.8	10.5
Top5 unigram	6.4	9.6
Top10 unigram	6.24	9.12
Top100 unigram	5.21	7.09
Top1 bigram	6.52	8.59
Top5 bigram	6.36	8.17
Top10 bigram	5.99	7.7
Top100 bigram	4.53	5.81
Top1 trigram	4.82	6.48
Top5 trigram	4.33	5.75
Top10 trigram	4.05	5.16
Top100 trigram	3.06	3.18
% of words in 1st corpus not in 2nd corpus	16.5	19.57
% of words in 2nd corpus not in 1st corpus	15.33	8.37
Top 10 unseen words in 1st corpus	؟هل الكحلاوي أيضًا النخلة البامبو الأوربية القوصي العراقية مالكوم جديدة	الرب اورشليم الأصحاح بنو شاول لاني هرون لانهم اسرائيل الارض
Top 10 unseen words in 2nd corpus	شاهة صوفياجوزفين الانكليز بوارو دياب تافيرنر هيستنغز عمها ليونايدز	أنه وهي أنا ولا إن إلا أن إل أو وهو

As the SBACC and the CCA are modern corpora of Arabic text, the M value is 0.90, which indicates similarities between these corpora. Unlike the Arabic Bible, the M value is larger at 1.50, which indicates noticeable differences between the SBACC and Arabic Bible corpora. Also, the average codelength of an n-gram is lower for SBACC vs. CCA than SBACC vs. Arabic Bible, which also indicate similarities between the SBACC and the CCA and more differences between the SBACC and Arabic Bible corpora.

In addition, the percentage of novel words in the SBACC not seen in the the CCA was 16.05%, and the percentage of novel words in the CCA not seen in the SBACC was 15.33%, which is typical for Arabic text (higher than for English texts). In contrast, the percentage of novel words in the SBACC and not seen in the Arabic Bible is 19.57% versus 8.37 % of novel words in Arabic Bible not seen in the SBACC, which is again more than

double, indicating major differences between these corpora. A sample of the top 10 unseen words in both corpora is listed in the last two rows in Table 3.5. A tag cloud of unigram codelength differences is shown in Figure 3.6 for SBACC vs. CCA and SBACC vs. Arabic Bible in Figure 3.7.

Figure 3.6: A tag cloud of unigram codelength differences of SBACC vs. CCA.

Figure 3.7: A tag cloud of unigram codelength differences of SBACC vs. Arabic Bible.

3.5 Summary

In this chapter, we have reviewed several existing corpora of Arabic language and produced new resources in the Arabic language for compression and other NLP applications. A new compression-based method for ranking n-gram differences between texts has been proposed. The method can readily be applied to producing n-gram tag clouds and lists, and these have been found to be effective at highlighting differences in topics. By using

the codelength difference method to compare how a text stream changes over time, the method can be used for corpora comparison and to reveal trends or emerging topics of interest.

Chapter 4

PPM Character-Based compression of Arabic Text

Contents

4.1 Introduction

Text compression is the process of reducing the amount of data needed to encode text that can be reproduced exactly as the original text after being decoded without losing any information. PPM has set the performance standard in terms of lossless compression of text throughout the past three decades (Teahan, 1998). In this chapter, we adapt PPM for Arabic text compression that will lead to significant improvements in Arabic text and other languages that use Arabic script in their writing system. Also we have found that several of our new techniques work very well when applied to other languages, such as English, Chinese, Arminian, Russian and Welsh.

We will first review the PPM model and apply it to Arabic text. Then, we will discuss our new techniques. The last section shows the experimental results of compressing different language corpora with a discussion and summarization.

4.2 Prediction by Partial Matching

PPM was introduced by Cleary and Witten (Cleary and Witten, 1984). The PPM compression algorithm applies a statistical model; it uses a number of previous symbols that determines the maximum order of the model to predict the next symbol. For example, if the maximum order of the PPM model is 3, the prediction, or probability, of the upcoming symbol will be estimated based on the three previous symbols. There are several variations of PPM, such as PPMA and PPMB (Cleary and Witten, 1984), PPMC (Moffat, 1990), PPMD (Howard, 1993), PPM* (Cleary and Teahan, 1997) and PPMO (Wu and Teahan, 2008). To demonstrate the operation of PPM, Figure 4.1 shows the state of various conditioning classes where k= 2, 1, 0 and -1 after the input string "المسلم" has been processed.

Usually, each symbol will be encoded arithmetically with the probability estimated by the model (Witten et al., 1987). The PPM encoding

Order k=2			**Order** k=1			**Order** k=0			**Order** k=-1		
Prediction	c	p	Prediction	c	p	Prediction	c	p	Prediction	c	p
ال→م	1	$\frac{1}{2}$	ا→ل	1	$\frac{1}{2}$	→ا	1	$\frac{1}{10}$	$\rightarrow A$	1	$\frac{1}{\lvert A\rvert}$
→Esc	1	$\frac{1}{2}$	→Esc	1	$\frac{1}{2}$	→ل	2	$\frac{2}{10}$			
						→م	2	$\frac{2}{10}$			
لم→س	1	$\frac{1}{2}$	م→ل	2	$\frac{2}{3}$	→س	1	$\frac{1}{10}$			
→Esc	1	$\frac{1}{2}$	→Esc	1	$\frac{1}{3}$	→Esc	4	$\frac{4}{10}$			
مس→ل	1	$\frac{1}{2}$	س→م	1	$\frac{1}{2}$						
→Esc	1	$\frac{1}{2}$	→Esc	1	$\frac{1}{2}$						
سل→م	1	$\frac{1}{2}$	ل→س	1	$\frac{1}{2}$						
→Esc	1	$\frac{1}{2}$	→Esc	1	$\frac{1}{2}$						

Figure 4.1: A PPMC model after processing the string "المسلم" with maximum order of 2.

scheme starts from the highest context order (in this case k=2). Where the highest order context predicts the upcoming symbol, the probability distribution associated with this symbol will be used to encode it. Otherwise, the escape probability estimated by the PPM model will be applied to let the encoder know to move to the next highest context order (k=1), and so on. This process will be continued until it reaches the lowest order in the model (k=-1), where all symbols would be encoded based on the probability of $\frac{1}{|A|}$, where A is the alphabet size. For English texts, the experiment shows that increasing context length above five does not generally improve compression (Cleary and Witten, 1984; Cleary and Teahan, 1997; Teahan, 1998).

For example, if "س" followed the string "المسلم", (reading from right to left) the probability of $\frac{1}{2}$ will be used as a successful prediction can be made in the order 2 model with the clause "لم→س ". This requires only 1 bit to encode (-$\log\frac{1}{2}$=1) . Suppose that the character "ا" follows the string "المسلم", the escape probability of $\frac{1}{2}$ will be encoded for the order k=2 model, and

the encoding process will downgrade from the order 2 model to the order 1 model. Again, a further escape probability $\frac{1}{2}$ will be encoded as order 1 does not predict "ا", and the encoding process will downgrade from order 1 to order 0, where "أ" will be selected with the probability of $\frac{1}{10}$. The total probability to encode the character "ا" is $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{10} = \frac{1}{40}$, which is 5.3 bits.

If the next symbol has not been encountered before, like "ض", the escape probability would be encoded down through the models to order -1, where all symbols have equal denoted by $\frac{1}{|A|}$ where A is the alphabet size. Assuming that the alphabet size is 256 for a standard byte-based (8bit) encoding, the probability will be $\frac{1}{256}$. So, "ض" will be encoded using the probability which requires $\frac{1}{2} \times \frac{1}{2} \times \frac{4}{10} \times \frac{1}{256}$ which is 11.5 bits. Actually, a more accurate estimation of the probability could be done by using an exclusion mechanism that will exclude symbols appearing in higher order. So, the new probability for "ض" will be $\frac{1}{2} \times \frac{1}{2} \times \frac{4}{10} \times \frac{1}{252}$, which is 11.2 bits.

4.3 Adapting PPM Character-Based Compression for Arabic

Surprisingly, considering UTF-8's popularity as an encoding scheme, there have been very few publications that have investigated the issues with finding the most effective compression of UTF-8 text. Fenwick and Brierly (Fenwick and Brierley, 1998) concluded that, for UTF-8 text, "accepted 'good' compressors such as finite-context PPM do not necessarily work well."

In this section, we will introduce several new universal preprocessing techniques to improve PPM compression of the UTF-8 encoding scheme. These methods essentially adjust the alphabet in some manner prior to the compression algorithm and is then applied to the amended text. The impact of the text preprocessing algorithms are examined using different file sizes and text genres from the BACC and other text corpora, such as the Hamshahri corpus of Persian texts (AleAhmad et al., 2009), Ar-

menian (Christensen, 3013) and Russian (Christensen, 3013) texts. One method described that has been found to be effective for English texts (Abel and Teahan, 2005; Teahan, 1998) is substituting bigraphs with a single, unique symbol. This bigraph substitution method, described in more detail in the next section, was only applied previously to English ASCII text and its effectiveness for other languages and encoding schemes such as UTF-8 has not been explored previously.

4.3.1 Bigraph Substitution for PPM (BS-PPM)

Bigraphs and Languages

By its nature, languages contain words that have many repeated bigraph characters, with two characters often appearing together in the same order such as "th" and "ea" in English text, "ال" and "من" in Arabic text. Usually, each language has common bigraphs that represent a significant percentage of the text (as was demonstrated in Chapter 2). For example, examining the most frequent 20 bigraphs over CCA, Brown and LOB corpora using 500,000 words produces some interesting results, as shown in Table 4.1.

The top 20 bigraphs take up almost 10% of the English texts. For Arabic texts, on other hand, the top ranked bigraphs take up significantly more at over 16%. Clearly, dealing with bigraphs for Arabic texts is an important consideration.

Bigraphs can be employed to enhance the compression performance over standard PPM by using preprocessing and postprocessing techniques before and after the compression and decompression, in a technique we call Bigraph Substitution for PPM (BS-PPM).

Table 4.1: The most frequent bigraphs found in text from three corpora.

Bigraph	**Arabic** (CCA)	**English-American** (Brown)	**English-British** (LOB)
1	ال	th	th
2	ـة	he	he
3	ية	in	in
4	لا	er	er
5	من	an	an
6	في	re	re
7	وا	on	on
8	ما	en	en
9	ات	at	nd
10	ان	or	at
11	ها	nd	es
12	لت	es	is
13	ين	is	or
14	عل	ti	ar
15	لى	te	ed
16	را	ed	to
17	ار	ar	of
18	تا	st	ng
19	ا	it	te
20	لأ	of	it
Percentage	16.61	9.69	9.56

Preprocessing and Postprocessing

The preprocessing technique involves sequentially processing the text and replacing the most frequent bigraphs in the order that they appear with unique single characters for each. From experiments, for most natural-language texts, we have found that replacing the top 100 bigraphs works best with the alphabet expanded by 100 more characters. The postprocessing operation simply performs the reverse mapping by replacing the new, unique characters with the original equivalent two-byte bigraphs, as follows.

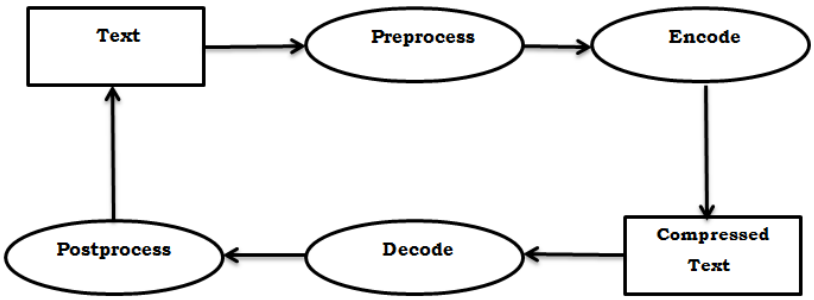

Figure 4.2: The use of preprocessing and postprocessing for compression.

Experimental Results for BS-PPM

In order to examine our new method, BS-PPM has been implemented using C programming language (see appendix A) to examine its efficiency over different natural languages text encoded using UTF-8 scheme. These include Arabic, Armenian, Chinese, English, Persian, Russian and Welsh. The result of the proposed technique is compared (see Table 4.2) against various well-known compression methods. These include the dictionary-based approach (Gzip) and the context-based approached (Bzip2 and ABC 2.4) in addition to standard PPM using the PPMD variant. These results are given in the standard measure bits per character (bpc) with the best result shown in bold font.

Table 4.2 shows the experimental results using order 4 BS-PPM compared with other well-known compression schemes under Windows and Unix including standard PPM. These include ABC 2.4 (Advanced Block-sorting Compressor) which is available to download at `http://www.data.info/ABC` and Bzip2 (Seward, 1998) available to download at `http://www.bzip.org`. Both ABC 2.4 and Bzip2 use The Burrows-Wheeler compression algorithm (Burrows and Wheeler, 1994). Gzip uses The Ziv-Lempel compression model (Ziv and Lempel, 1977) and is available to download at `http://www.gzip.org`. The files being compressed (specified in the second column of the table) are UTF-8 encoded files in the BACC for Arabic text, the HC-Russian corpus for Russian text (Christensen, 3013),

the HC-Armenian text for Armenian (Christensen, 3013), the Hamshahri corpus for Persian text (AleAhmad et al., 2009), the LCMC (McEnery and Xiao, 2004) for Chinese text, the CEG corpus (Ellis et al., 2001) for Welsh text, and the Brown and LOB corpora for American and British English texts respectively.

Table 4.2: BS-PPM vs. other methods applied to various language texts.

Language	**Corpus Text File**	**Size** (bytes)	**Bzip2** (bpc)	**ABC2.4** (bpc)	**Gzip** (bpc)	**PPM Order4** (bpc)	**BS-PPM** Order4 (bpc)
Arabic	BACC	56633170	1.45	1.43	2.14	1.79	**1.34**
Armenian	HC-Armenian	36700160	1.56	1.37	2.39	1.69	**1.17**
Chinese	LCMC	4555457	2.65	2.57	3.47	2.49	**2.46**
English	Brown	5998528	2.46	2.29	3.16	2.23	**2.10**
English	LOB	5877271	2.43	2.27	3.14	2.21	**2.08**
Persian	Hamshahri	41567603	1.53	1.38	2.22	1.75	**1.26**
Russian	HC-Russian	52428800	1.52	1.31	2.45	1.73	**1.12**
Welsh	CEG	6169422	2.55	2.34	3.19	2.30	**2.14**
Average			2.02	1.86	2.77	2.02	**1.70**

Clearly, BS-PPM works very well on UTF-8-encoded texts in many languages, such as Arabic, Persian, Armenian, Russian, Welsh, Chinese, and English since it records significant improvements over other methods in terms of compression rate (bpc). In all cases of these languages, BS-PPM significantly outperforms the other compression methods, as it has the best results shown in all cases and also significantly outperforms standard PPM itself. For example, for Arabic text, BS-PPM shows a 25.14% improvement over standard PPM. For Armenian text, BS-PPM shows a 14.6% improvement over ABC2.4, 25% over Bzip2 and 30.77% over standard PPM. For Persian text, BS-PPM showed a 8.70% improvement over ABC2.4, 17.7% over Bzip2 and 28% over standard PPM. For Russian text, BS-PPM showed a 14.5% improvement over ABC2.4, 26.32% over Bzip2 and 35.26% over standard PPM. Also, there was a significant improvement in compression rates for both American and British English with BS-PPM recording a 14.6% improvement over Bzip2 for the Brown corpus and 14.4% for the LOB corpus. This represents an 8.3% improvement over ABC2.4 for the Brown corpus and 8.4% for the LOB corpus, 33.5% over

gzip for Brown and 33.8% for LOB and 5.8% over standard PPM for Brown and 5.9% for LOB.

In order to examine the impact of bigraph coding using different order PPM models, the following table shows the results of both PPM and BS-PPM for orders 1 through 7. The results for order 4 from the previous table are repeated here for clarity. The best result for each row is shown in bold font.

Table 4.3: Different order of PPM and BS-PPM over seven corpora.

File	**Language**	**PPM** order 1	**BS-PPM** order 1	**PPM** order 2	**BS-PPM** order 2	**PPM** order 3	**BS-PPM** order 3	**PPM** order 4	**BS-PPM** order 4	**PPM** order 5	**BS-PPM** order 5	**PPM** order 6	**BS-PPM** order 6	**PPM** order 7	**BS-PPM** order 7
BACC	Arabic	2.42	2.03	2.17	1.68	2.04	1.45	1.79	1.34	1.66	1.30	1.51	1.28	1.44	**1.27**
HC	Armenian	2.43	2.06	2.02	1.49	1.90	1.26	1.69	1.17	1.61	**1.15**	1.42	1.16	1.36	1.18
LCMC	Chinese	4.03	3.72	3.01	2.86	2.66	2.58	2.49	2.46	2.46	**2.44**	2.47	2.45	2.49	2.46
English	Brown	3.67	3.13	3.02	2.34	2.51	2.12	2.23	**2.10**	2.16	2.13	2.17	2.17	2.22	2.21
English	LOB	3.66	3.11	2.99	2.32	2.48	2.1	2.21	**2.08**	2.14	2.11	2.15	2.15	2.19	2.18
Hamshahri	Persian	2.29	1.92	2.09	1.53	1.96	1.3	1.75	1.26	1.60	**1.17**	1.42	1.17	1.33	1.18
HC	Russian	2.47	1.95	2.08	1.48	1.97	1.23	1.73	1.12	1.63	1.09	1.41	**1.08**	1.33	1.09
CEG	Welsh	3.66	3.2	3.07	2.44	2.60	2.18	2.30	**2.14**	2.20	2.17	2.21	2.21	2.25	2.24

For Arabic text, the best compression rate was using order 7; for Armenian, Persian and Chinese text, the best compression rate was using order 5; for Russian text the best compression rate was using order 6; and for Welsh and English text, order 4 is the best compression rate for both American and British texts and welsh text as well.

These results show that an impressive improvement in compression performance is possible for UTF-8 encoded natural language texts using the bigraph substitution method. The next section describes a new method that also uses preprocessing techniques and also yields impressive improvements in compression performance.

4.3.2 Character Substitution for PPM (CS-PPM)

UTF-8 is a variable-length, byte-based encoding and, therefore, a bigraph-based substitution method as described in the previous section may not be best suited for languages where two-byte encoding is not the norm. This is

illustrated by the results for Chinese texts, which in UTF-8 encoding often requires 3 or 4 bytes to encode each character. This suggests a character, rather than a bigraph, substitution method might also yield impressive results.

Preprocessing and Postprocessing

Unlike the bigraph substitution method just described, which replaces the top 100 bigraphs in the text during the preprocessing stage, our character substitution method (called CS-PPM) substitutes all the UTF-8 multi-byte or single-byte character sequences in the original text. Two output files are produced as a result of the preprocessing; one contains a stream of UTF-8 characters (called the 'vocabulary' stream) and the other contains a list of symbol numbers (called the 'symbols' stream).

Whenever a new character is encountered in the original text, this character is assigned a symbol number equal to the current symbols count, which is then incremented. The symbol count is initialised to 1 for the first character. When that same character is encountered later on in the text, the symbol number assigned to it is written out to the symbols output file. At the same time, the UTF-8 character-byte sequence for new characters is written out to a separate vocabulary output file.

Both the vocabulary output file and the symbols output file need to be compressed during the encoding stage. Therefore, two files also need to be decoded separately, with the vocabulary output file requiring decoding first in order to define the reverse mapping between symbols and UTF-8 characters during the postprocessing stage.

We have found the following method works well at encoding the two files. For encoding the vocabulary output file, standard order 1 byte-based PPM is quite effective. For the symbols output file, where the symbol numbers can get quite large for some languages, a similar technique to word-based PPM (Teahan, 1998) works well with the alphabet size being

unbounded. We have found that an order 4 model works best overall for the languages we have experimented with.

Experimental Results for CS-PPM

In order to examine our new method, CS-PPM has been implemented using the C programming language (see appendix B) to examine its efficiency over texts of different file sizes (small, medium, large and very large) from the BACC corpus of Arabic language. The results of the proposed technique is compared (see Table 4.4) against standard PPM using the PPMD variant. The percentage improvement is calculated to explore the impact of the proposed technique over these files. As discussed above, CS-PPM will processed files into two streams, the vocabulary stream and symbols stream, therefore, the percentage of encoding the vocabulary and symbols stream for CS-PPM is shown as both of them need to be encoded separately. These results are given in the standard measure bits per character (bpc) with the best result shown in bold font.

Table 4.4 shows that, in all cases, there is a significant improvement in performance for CS-PPM over standard PPM, as shown in column 4 of Table 4.4. The improvement is noted most for the largest files in the corpus with almost 25% improvement for the file $bookcollection1$. Column 5 provides the percentage cost of encoding the symbols output file and the last column provides the percentage cost of encoding the vocabulary output file. The symbols output file consistently takes up 75 to 80% of the overall encoding cost.

Table 4.5 lists results for PPM and CS-PPM on various language texts, with results from $bookcollection1$ from the BACC corpus repeated on the first row for comparison. The languages for which CS-PPM is most effective are Arabic (23.3% improvement), Armenian (30.4%), Persian (31.5%) and Russian (35.2%). With Table 4.4, the percentage cost for encoding the symbols and vocabulary are also shown in the last column.

Table 4.4: PPM vs. CS-PPM of BACC.

File	**Size** (bytes)	**PPM** Order4 (bpc)	**CS-PPM** (bpc)	**Improv.** %	**Percentage** of encoding symbols %	**Percentage** of encoding vocabulary %
economic	15924	2.03	**1.90**	6.4	76.86	23.14
education	27086	2.06	**1.96**	4.85	75.99	24.01
sports	31706	1.95	**1.77**	9.23	78.17	21.83
culture	34760	2.03	**1.88**	7.39	76.79	23.21
artandmusic	42648	2.08	**1.93**	7.21	76.06	23.94
political	47556	1.97	**1.74**	11.68	78.41	21.59
articles	103839	1.94	**1.76**	9.28	78.18	21.82
press	549063	1.83	**1.57**	13.74	80.41	19.59
Novel1	860680	1.87	**1.63**	12.83	79.62	20.38
Novel2	912604	1.86	**1.62**	12.9	79.72	20.28
Novel3	1023987	1.86	**1.61**	13.44	79.95	20.05
bookcollection1	56633170	1.79	**1.38**	24.59	82.7	17.3
Bookcollection2	201693734	1.76	**1.35**	23.3	83.17	16.83
Average		1.94	**1.73**	11.37	78.57	21.44

Table 4.5: Results for PPM and CS-PPM on various language texts.

Language	Corpus text file	Size (bytes)	PPM (bpc)	CS-PPM (bpc)	Improve. %	Percentage of encoding symbols %	Percentage of encoding Vocabulary %
Arabic	bookcollection1	56633170	1.79	**1.38**	23.3	82.7	17.3
Armenian	HC	36700160	1.69	**1.18**	30.4	85.3	14.7
Chinese	LCMC	4555457	2.49	**2.37**	4.86	70.62	29.38
English	Brown	5998528	2.23	**2.15**	3.47	73.1	26.9
English	LOB	5877271	2.21	**2.13**	3.56	73.36	26.64
Persian	Hamshahri	41567603	1.75	**1.20**	31.53	85.02	14.98
Russian	HC	52428800	1.73	**1.12**	35.15	85.98	14.02
Welsh	CEG	6169422	2.3	**2.20**	4.52	72.55	27.45
Average			2.03	**1.72**	17.26	78.58	21.42

4.3.3 Character Substitution of Arabic for PPM (CSA-PPM)

In this section, we describe a third method that is tailored specifically for Arabic text. We call the method Character Substitution of Arabic for PPM (CSA-PPM).

Unlike CS-PPM which produces two output files (the symbol and vocabulary stream), one output file is produced as a result of the preprocessing method of CSA-PPM. Since we can assume in advance that we will be encoding Arabic text, we can directly substitute the characters with the equivalent number of the UTF-8 scheme to eliminate the need for a vocabulary output file altogether, which will decrease the size of the output compressed file. This technique is also implemented using the C programming language (see Appendix C). The compressed output file size in bytes of both CSA-PPM and CS-PPM for comparison for different file sizes from the BACC corpus are shown in Table 4.6. The difference in the number of compressed output file size in bytes between these methods are also shown in the last column of the table.

Table 4.6: CS-PPM vs. CSA-PPM.

File name	Size (Bytes)	CSA-PPM Compressed Output (Bytes)	CS-PPM Compressed output (Bytes)	Different Bytes
economic	15924	**3629**	3681	-52
education	27086	**6427**	6516	-89
sport	31706	**6856**	6934	-78
culture	34760	**8009**	8097	-88
artandmusic	42648	**10148**	10246	-98
political	47556	**10144**	10238	-94
articles	103839	**22683**	22813	-130
press	549063	**108083**	108339	-256
novel1	860680	**176215**	176444	-229
novel2	912604	**186302**	186505	-203
novel3	1023987	**206690**	206902	-212
shortstories	1041952	**201613**	201859	-246
literature	19187425	**3312444**	3312864	-420
history	30714551	**4403343**	4403730	-387
bookcollection1	56633170	**9370563**	9371244	-681
Bookcollection2	201693734	**32249015**	32249566	-551

The savings from not having to encode the characters that make up the alphabet leads to significant improvements in compression rate (bpc) for

the small sized files as shown in Table 4.7 with the best result shown in bold font. However, for large files, the improvements are minimal when compared to the overall compressed file size.

Table 4.7: CS-PPM vs. CSA-PPM

File name	Size (bytes)	CSA-PPM (bpc)	CS-PPM (bpc)	Improvements %
economic	15924	**1.83**	1.90	3.68
education	27086	**1.91**	1.96	3.54
sport	31706	**1.74**	1.77	2.25
culture	34760	**1.85**	1.88	2.12
artandmusic	42648	**1.91**	1.93	2.05
political	47556	**1.71**	1.74	1.72
articles	103839	**1.75**	1.76	1.13
press	549063	**1.57**	1.57	0
novel1	860680	**1.63**	1.63	0
novel2	912604	**1.64**	1.62	0
novel3	1023987	**1.62**	1.61	0
shortstories	1041952	**1.55**	1.55	0
literature	19187425	**1.37**	1.37	0
history	30714551	**1.14**	1.14	0
bookcollection1	56633170	**1.38**	1.38	0
Bookcollection2	201693734	**1.35**	1.35	0

4.3.4 Dotted (Lossless) and Non-Dotted (Lossy) Compression of Arabic Text

Data compression is divided into two main categories, lossless and lossy compression. On the other hand, text compression and, more specifically, the problem of natural language text compression, is usually considered to be lossless compression since changing the text in natural language will usually alter the meaning. Despite this, there have been a few papers that have considered the problem, from the satirical paper by Witten et al. (1994) to the more recent paper by Çelikel Çankaya et al. (2011).

The Arabic language contains 28 letters, fifteen of them that are dotted and thirteen of them non-dotted, as shown in table below.

Table 4.8: Arabic letters.

Dotted	Non-Dotted
ي ن ق ف غ ظ ض ش ز ذ خ ج ث ت ب	و ه م ل ك ع ط ص س ر د ح ا

Dots above and below the letter give it a different pronunciation, such

as one dot below "ب" which is equivalent to a B in English and two dots above "ت" which is equivalent to the T in English, and so on. In old Arabic texts, all letters were not originally dotted but, despite some ambiguity as a result, this could still be easily identified by native Arabic readers familiar with the Arabic language. Figure 4.3 shows a sample ancient Arabic script that uses the non-dotted form.

Figure 4.3: Sample of Arabic script prior to 820 A.D. (UmAlqura, 2014).

Note that, there are difference between dots and diacritics in Arabic text, where the dots, either one, two or three are used to identify the character itself, while the diacritics is used to pronounce the character in the right way. For instance, character "ا" can be pronounced in many different ways using the diacritics symbols such as "اُ اَ اِ اً اٍ اٌ".

Due to the ambiguity of identifying the correct letter by non-native Arabic language learners, since 820 A.D, these letters have become dotted (Baik, 1992). We exploited this historical feature of the language to improve the compression rate of Arabic in some cases by preprocessing the Arabic text prior to compression with PPM and recovering it during the postprocessing stage after decompression, as explained in more detail below.

Preprocessing Arabic Text for Lossy Non-Dotted Compression

During the first stage, letters such as "ب ت ث ن ي" are normalized by removing the dots to generate the lossy (non-dotted) version of the original text before it is compressed using PPM. For example, "ي" becomes "ى" and letters such as "ج خ" are normalized to be "ح", and letters such as "ذ ز ش ض ظ غ ف ق" are normalized to be "د ر س ص ط ع ف".

Encoding and Decoding

During the encoding stage, the preprocessed (lossy) text is compressed using PPM. During the decoding stage, the compressed text is decoded using PPM to reproduce the lossy version of the original text.

Recovering the Lossless Version of the Text

In this stage, the lossy text is automatically corrected using the Viterbi algorithm (Viterbi, 1967) in order to try to recover the original text. This is done by finding the most probable sequence when a match occurs in the text from the list shown in Table 4.9. The most probable sequence is selected which is the sequence that has the best encoding according to a PPM character-based language model trained on text from the BACC corpus as it is a large and representative corpus of contemporary Arabic language.

Table 4.9: List of correction.

List of correction		
ب → ب ب	ت→ ب	ث→ب
ن→ب	ي→ب	ج→ح
خ→ح	ح→ح	ذ→د
د→د	ز→ر	ر→ر
ش→س	س→س	ض→ص
ص→ص	ظ→ط	ط→ط
غ→ع	ع→ع	ف→و
ق→و	ة→ه	ه→ه

The Viterbi-based correction method will make mistakes, so these have

to be encoded separately to inform the decoder which of the dotted forms (zero, one or two) each character should take. In our implementation, we directly encode the file position of the character that has been incorrectly dotted, which requires $\lceil \log_2 N \rceil$ bits, where N is the number of characters in the file. One further bit is needed to encode the number of dots, either one or two dots, with the default form not requiring correction being zero dots.

Experimental Results

To examine our new method, we conducted experiments using the CCA and the EASC corpora while training the PPM-based corrector on the BACC corpus. The results are shown in Table 4.10 with the best results shown in bold font. Both the CCA and ESAC corpora are preprocessed to generate the lossy version (non-dotted) which in then compressed, then the Viterbi-based correction method is used after decompression in order to recover the lossless version of the text (dotted) when a match occurs between confusion list and text (see Appendix D). The fifth column shows the percentage of improvement of non-dotted PPM over dotted PPM, while the sixth and seventh shows the number of errors (characters that have been dotted incorrectly) and its percentage respectively. The last column shows the compression rate after the cost of correcting errors have been added.

Table 4.10: Dotted PPM vs. Non-Dotted PPM.

File	Size (Bytes)	PPM-Dotted (bpc)	PPM Non-Dotted (bpc)	Improve. %	Number of error	Error %	PPM Non-Dotted with dots corrected (bpc)
EASC	630321	1.86	**1.72**	7.53	2301	0.66	1.78
CCA	6265790	1.83	**1.73**	5.46	15155	0.43	1.74

As anticipated, lossy compression showed significant improvement over dotted compression by over 7% for EASC and over 5% for CCA, respectively. When the lossy form of the text was corrected using the Viterbi-

based correction method in order to recover the original non-dotted text, a small number of errors were discovered. These errors made up 0.66% of the EASC and 0.43% for the CCA. When the cost of encoding these errors was added in (see the last column in Table 4.8), the lossless scheme still resulted in a significant improvement in compression (1.78 bpc compared to 1.86 bpc for the EASC and 1.74 bpc compared to 1.83 bpc for the CCA).

In order to investigate this result further, 10-fold cross-validation was performed on the BACC with the results shown in Table 4.11. The corpus was split into 10 equal parts (splits S1 through S10 as in the first column of the table). Training of the Viterbi-based corrector was then done on 9/10 of the corpus and testing on the other 1/10, and this was repeated 10 times using each split for testing.

Table 4.11: Ten-fold cross-validation.

BACC Split	Size (bytes)	PPM -Dotted (bpc)	PPM Non-Dotted (bpc)	Improv. %	Number of Errors	Error %	PPM Non-Dotted With Dots Corrected (bpc)
S1	31188787	1.76	1.69	3.98	129662	0.74	1.8
S2	31351880	1.70	1.65	2.94	101282	0.58	1.74
S3	31394336	1.71	1.65	3.51	107259	0.61	1.74
S4	31249486	1.82	1.74	4.4	229401	1.31	1.94
S5	31369255	1.69	1.64	2.96	101334	0.58	1.73
S6	31253465	1.71	1.65	3.51	95563	0.54	1.73
S7	30931867	1.77	1.71	3.39	171525	0.98	1.86
S8	31045940	1.64	1.60	2.43	103952	0.59	1.70
S9	31394039	1.75	1.67	4.57	138527	0.79	1.79
S10	31434684	1.75	1.67	4.57	155972	0.89	1.81

The non-dotted compression shows an improvement of 2 to 5% over the dotted texts. However, a significant number of errors were made by the correction software as shown in column six of the table and this resulted in a lossless compression result (as shown in the last column) worse than the dotted compression result, shown in column three. The worse result is possibly due to the significant number of dotted characters which represent half of the Arabic alphabet. Another possible reason is the nature of the BACC corpus with files being on a variety of subjects such as economics, education, sport and culture for example.

4.4 Conclusion

The PPM compression performance of Arabic text can be significantly improved by applying preprocessing and postprocessing techniques that rely on adjusting the alphabet, including expanding or reducing the number of symbols. The BS-PPM, CS-PPM and Non-Dotted PPM techniques described in this chapter are all examples of these adjustments with the first two working by expanding the alphabet and the last one working by reducing the alphabet.

Chapter 5

Word and Tag Based Models for Arabic Text

Contents

5.1 Introduction

This chapter looks into the problem of encoding Arabic words and parts of speech tags as units for encoding. Many natural language applications for machine translation and speech recognition feature models that are constructed on the basis of words or tags (Brown et al., 1992).

Similar to character-based models, word-based models predict an upcoming word on the basis of a previous number of words in the context specified as the order of the model. Word-based compression is usually faster than character-based compression because considerably fewer symbols need to be encoded. A number of word-based compression methods for natural language text have been proposed (Horspool and Cormack, 1992; Moffat, 1989; Teahan, 1998).

Another way of building a language model is through the use of part of speech tag components. Teahan (1998) states that

> *"The idea is that knowing the tag of the word helps in predicting it. The advantage of using the tag is that it may have occurred many times previously and so a good representative sample of what is likely to follow it has been built up. By contrast, an individual word may have occurred only a small number of times."*

Two important issues are considered in using a tag-based model. First, each word in a text must have a tag attached to it in some manner. Second, the tags are encoded along with the text and therefore has the potential to increase the size of the text being compressed.

In this chapter, the adaptive word- and tag-based models first proposed by Teahan (1998) are explored, particularly in relation to the Arabic language. First, we review previous work and then apply word- and tag-based PPM models. The last section of this chapter discusses the experiments.

5.2 Previous Work

As character-based PPM models, word- and tag-based models predict an upcoming symbol, beginning from the highest context; when the upcoming symbol does not appear in this context, an escape symbol is encoded to command the decoder to downgrade to the next highest context (Moffat, 1989; Teahan, 1998). Several escape algorithms have been described in (Witten and Bell, 1991) and in (Teahan, 1998) for word-based models.

Experiments show that the X1 method is the best for most English text cases (Teahan, 1998). This method is represented as follows:

$$e = \frac{t_1 + 1}{T_d + t_1 + 1} \tag{5.1}$$

where t_1 is the number of symbols appearing exactly once in context C_d and T_d denotes the frequency with which context C_d occurs. This method predicts the escape probability proportional to the number of words that have occurred once in the context, or what are defined as singletons (see section 2.4.2). For the English language, experiments show that the word- and tag-based models shown in Table 5.1 exhibit the best performance amongst all other models (Teahan, 1998).

Table 5.1: Some models for predicting characters, tags and words (Teahan, 1998).

C|C^5 Model	**W|W Model**	**W|TW Model**	**T|TWT Model**
$p(c_i\|c_{i-1}c_{i-2}c_{i-3}c_{i-4}c_{i-5})$ $\hookrightarrow p(c_i\|c_{i-1}c_{i-2}c_{i-3}c_{i-4})$ $\hookrightarrow p(c_i\|c_{i-1}c_{i-2}c_{i-3})$ $\hookrightarrow p(c_i\|c_{i-1}c_{i-2})$ $\hookrightarrow p(c_i\|c_{i-1})$ $\hookrightarrow p(c_i\|)$ $\hookrightarrow p_{eq}(c_i\|)$	$p(w_i\|w_{i-1})$ $\hookrightarrow p(w_i\|)$ $\hookrightarrow$ *Character model*	$p(w_i\|t_iw_{i-1})$ $\hookrightarrow p(w_i\|t_i)$ $\hookrightarrow$ *Character model*	$p(t_i\|t_{i-1}w_{i-1}t_{i-2})$ $\hookrightarrow p(t_i\|t_{i-1}w_{i-1})$ $\hookrightarrow p(t_i\|t_{i-1})$ $\hookrightarrow p(t_i\|)$ $\hookrightarrow p_{eq}(t_i\|)$

The first model, $C|C^5$, is an order-five PPM model that estimates the probability of symbols based on characters. In this model, the previous five characters constitute the context that predicts the probability of an

upcoming symbol, as shown in the following formula:

$$p(\alpha) = \prod_{i=1}^{m} (c_i | c_{i-1} c_{i-2} c_{i-3} c_{i-4} c_{i-5}). \tag{5.2}$$

This estimation varies depending on the escape method used (represented as $\hookrightarrow$ in Table 5.1). If the highest context (order 5) fails to predict the upcoming symbol, an escape probability is encoded to downgrade the decoder to the next highest context as discussed earlier in Chapter 4.

Model $W|W$ is an order-one PPM model that estimates the probability of symbols based on words. In this model, the previous word is the context that predicts the probability of an upcoming word, as expressed in:

$$p(S) = \prod_{i=1}^{m} (w_i | w_{i-1}). \tag{5.3}$$

where S is the sequence of the text that is being predicted. If the model fails to successfully predict a word, an escape probability is encoded to downgrade the coder to the order 0 model to verify if the word has previously occurred. If not, the escape probability is encoded to downgrade the model to a character-based PPM model.

The $W|TW$ model uses both word- and tag-based streams (such as noun and verb tags that are assigned to the word-based stream on the tag set scheme) to estimate the probability of the upcoming word. This probability is expressed thus:

$$p(S) = \prod_{i=1}^{m} (w_i | t_i w_{i-1}). \tag{5.4}$$

In this model, the prediction based on an upcoming word tag and a previous word as the context are estimated to predict the upcoming word. If the model fails to predict the word, an escape probability is encoded and the model is downgraded to the next order, where the prediction is based on the tag of the upcoming word only. In this stage, failure to predict an

upcoming word results in the encoding of an escape probability and the transformation of the model into a character-based model.

As in the previous model, $T|TWT$ uses both word- and tag-based streams to estimate the probability of an upcoming tag, as shown in the following formula:

$$p(S) = \prod_{i=1}^{m} (t_i | t_{i-1} w_{i-1} t_{i-2}). \tag{5.5}$$

The probability of the current tag is estimated using the previous tag, word and tag prior to the previous word as the context. Under failed prediction, an escape probability is encoded to move the coder to the next order model, in which case the previous word and tag are the contexts for estimating the prediction of the upcoming tag.

In case the context fails to predict the upcoming tag, an escape probability is again encoded to command the coder to downgrade to the next order model. At this stage, the estimation of the next tag is based on only the receding tag as the context. If necessary, the coder encodes an escape and then moves on to the next sequence, where the model verifies if the tag has previously occurred. This step assigns a probability to the escape on the basis of the frequency with which the tag has occurred. If escape encoding is necessary, the model moves on to the final order model, wherein the prediction of all tags have equal probability. Teahan (1998) found that $T|TWT$ is the best model for predicting tags in English text.

5.3 Experimental results

This section discusses the experimental results on the word- and tag-based models for Arabic text. Given the limitation of available resources on tagged corpora for the Arabic language (as discussed in Chapter 3), we report experiments in which only the Arabic Treebank Corpus (ATC) (Maamouri et al., 2005) is used to train the tag-based models.

In this experiment, the ATC corpus is divided into 10 equal parts as shown in Table 5.2 with lists the results for various PPM variants that include order 4 CSS-PPM, order 4 BS-PPM, order 4 standard PPM, the order 1 word-based model and the tag-based model, which are used to examine and compare the effect of applying these models on Arabic text compression.

Table 5.2: Results for PPM variants on the ATC.

File	Size (Bytes)	PPM order 4 (bpc)	CS-PPM order 4 (bpc)	BS-PPM order 4 (bpc)	W|W (bpc)	W|TW+ T|TWT (bpc)
ATC1	138733	1.82	1.55	1.56	1.82	2.81
ATC2	140173	1.84	1.59	1.59	1.86	2.91
ATC3	142991	1.78	1.47	1.46	1.78	2.59
ATC4	140607	1.78	1.50	1.50	1.78	2.67
ATC5	141613	1.77	1.47	1.46	1.76	2.59
ATC6	140843	1.69	1.38	1.38	1.69	2.43
ATC7	137489	1.78	1.48	1.48	1.79	2.61
ATC8	141552	1.76	1.49	1.49	1.76	2.67
ATC9	142327	1.75	1.45	1.45	1.74	2.59
ATC10	139148	1.78	1.47	1.47	1.77	2.61

The character-based PPM models CS-PPM and BS-PPM outperform the other methods, achieving a 15% to 18% improvement in compression rate over that generated with the standard character-based PPM and PPM word-based $W|W$. It also achieves 40% to 24% improvement in compression rate over that produced by the tag-based $W|TW + T|TWT$.

A near-match in improvement occurs between the character-based PPM model and the word-based $W|W$ model. The $W|W$ model frequently escapes to the character-based model to predict novel words that are unsuccessfully predicted by the word-based model. The poor performance of the tag-based model is attributed partially to the encoding of the words and the novel words for which the character-based model and the tags are used. These results also indicate that the ATC tagset may not be the best suited for language modelling purposes when we consider how well the English tag-based models perform using the Brown corpus tagset in comparison. Table 5.3 lists the number of bytes needed for encoding as

well as the percentage cost of each component of $W|TW$ and $T|TWT$ as well the cost of encoding the novel words.

Table 5.3: Percentage cost of encoding tag-based model.

File	W|TW (Bytes)	%	T|TWT (Bytes)	%	Novel words order 4 (Bytes)	%
ATC1	7245	26.1	6102	21.9	14412	51.9
ATC2	7348	25.3	6214	21.4	15423	53.2
ATC3	7083	26.9	5901	22.4	13313	50.6
ATC4	7267	27.2	7267	22.1	13580	50.8
ATC5	6961	26.8	5867	22.6	13177	50.7
ATC6	6861	28.2	5813	23.9	11636	47.9
ATC7	7112	27.8	5857	22.9	12580	49.2
ATC8	7318	27.2	6065	22.6	13474	50.2
ATC9	7001	26.8	6012	22.9	13138	50.2
ATC10	7015	22.4	5802	22.4	13095	50.5

The cost of word encoding accounts for about 23% to 30% of the total cost of encoding, whereas the tag encoding accounts for approximately 22% to 24% of the total. These percentages amount to about 50%, whereas the percentages for the encoding of novel words represent more than 50% of the total. This percentage is notably higher than for comparable English experiments which indicate that Arabic has a much larger vocabulary due to a higher use of inflexions, as a result of the rich morphological nature of Arabic text.

Whether the tag-based PPM models for Arabic text will always perform poorly compared with other methods is unclear because of the limited resources on POS-tagged corpora for the Arabic language. This limitation prevents evaluation with different text sizes, genres and tag sets; the poor performance of the models may be due to the tagging applied to this text.

To further explore word-based models for Arabic text, we conducted experiments on various order word-based PPM models for the BACC corpus files. Table 5.4 shows that high-order (e.g. 2, 3 and 4) word-based PPM models do not improve the compression rate for Arabic text. This result is understandable given the rich morphological nature of Arabic text which expands the size of the vocabulary used. For example, "رجل" which means

Table 5.4: PPM word-based models vs. character-based models for BACC corpus.

File	Size (bytes)	PPM Word Based Order 1 (bpc)	PPM Word Based Order 2 (bpc)	PPM Word Based Order 3 (bpc)	PPM Word Based Order 4 (bpc)	PPM Character Based Order 4 (bpc)	BS-PPM Character Based Order 4 (bpc)
economic	15924	2.02	2.02	2.02	2.02	2.03	**1.83**
education	27086	2.04	2.04	2.04	2.04	2.06	**1.91**
sports	31706	1.94	1.94	1.94	1.94	1.95	**1.73**
culture	34760	2.01	2.01	2.01	2.01	2.03	**1.85**
artandmusic	42648	2.07	2.07	2.07	2.07	2.08	**1.91**
political	47556	1.93	1.93	1.93	1.93	1.97	**1.72**
articles	103839	1.94	1.94	1.94	1.94	1.94	**1.75**
press	549063	1.83	1.83	1.83	1.83	1.83	**1.58**
novel1	860680	1.86	1.86	1.86	1.86	1.87	**1.63**
novel2	912604	1.86	1.86	1.86	1.86	1.86	**1.62**
novel3	1023987	1.85	1.85	1.85	1.85	1.86	**1.61**
shortstories	1041952	1.85	1.85	1.85	1.85	1.85	**1.55**
literature	19187425	1.76	1.76	1.76	1.76	1.76	**1.41**
history	30714551	1.60	1.60	1.60	1.60	1.60	**1.20**
bookcollection1	56633170	1.81	1.81	1.81	1.81	1.79	**1.34**
bookcollection2	201693734	1.77	1.77	1.77	1.77	1.76	**1.39**

"man", has been presented in many forms such as "الرجل الرجلان الرجلين" "الرجال رجلان رجلين رجال". In most cases, the word-based PPM models improve the compression rate by 1% to 3% over that produced by the standard PPM character-based models. This improvement applies primarily to small and medium BACC corpus files. The character-based BS-PPM model generates the best compression rate amongst the other methods, with a significant improvement of 8 to 28% (relative to file size) (Figure 5.1).

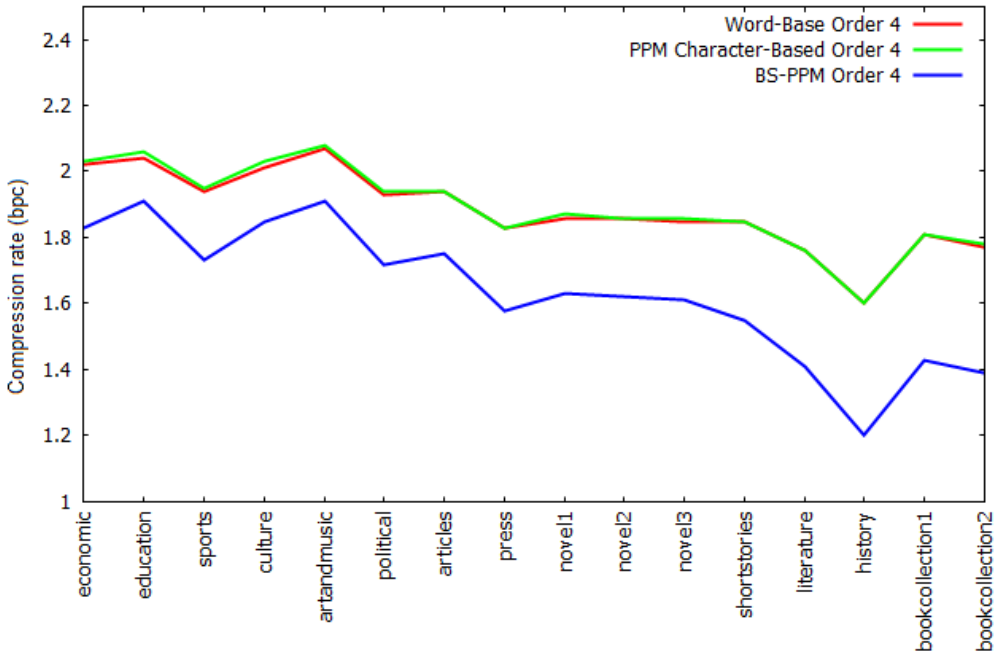

Figure 5.1: Compraring between various methods of PPM over BACC.

5.4 Conclusion

In this chapter, word- and tag-based PPM models have been compared with character-based PPM models. The new character-based BS-PPM model described in Chapter 4 outperforms the other methods. The next highest performing models are the word-based PPM models, followed by the standard character-based PPM models for Arabic text. The tag-based PPM models exhibit the poorest performance. Whether tag-based PPM models will always perform poorly is unclear given the limited resources available of tagged Arabic text of different tagsets, text sizes and genres.

Chapter 6

Some Applications of PPM Models

Contents

6.1 Introduction

"Before the invention of the digital computer, language was seen as the exclusive province of human beings. The signaling systems of other animals were too rigid and simple to deserve the designation 'language', and all that mechanical and electrical devices could do was to store and transmit sequences of code to be interpreted by people. With the computer, new possibilities appeared. The essential quality of the digital computer is its ability to manipulate symbols – not just numbers, but symbols of any kind. It was recognised from the earliest days of computing that, in addition to their obvious applications in scientific calculation and bookkeeping, computers could work with language" (Winograd, 1983).

In this chapter, we apply character-based PPM compression models described in Chapter 4 to produce language models that can be used to solve problems such as authorship attribution, word segmentation and OCR text correction for Arabic text.

6.2 Authorship Attribution

Authorship attribution is the process of identifying the exact author for random anonymous text among several alternative authors by the means of characteristics of the text being identified (Juola et al., 2006). In other words, authorship attribution is a text classification task where each author is considered a class (Koppel et al., 2009).

There are numerous methods that have been proposed to address the authorship attribution problem based on analyzing semantic, syntactic and morphological features used by the authors (Koppel et al., 2009). Authorship attribution has a very limited number of studies that are concerned specifically with Arabic language compared to other languages such as English and French (Shaker and Corne, 2010).

In this section, we will apply PPM character-based compression models to the problem of authorship attribution for Arabic language using the

same approach by Teahan (1998). First, we selected five famous authors in Arabic countries – Mohammed Alsalibi, Ibrahim Khalil, Bassem Ibrahim, Mary Rashow and Taleb Omran. Then we selected two novels for each author to train the PPM character-based models on these novels, and then each tested text that was not a part from the trained data was compressed using these trained models.

Ideally, the best encoding for the tested text being encoded should be using the model that has been trained on text written by the same author. In other words, the tested text will achieve the minimum cross-entropy when it is compressed using the model that has been trained on the author of this text. The following table presents the result of the experiment as shown in Table 6.1.

Table 6.1: Identifying the authorship for several texts.

Tested Text	Training Text (tested text is not included)				
	Bassim Ibrahim	Ibrahim Khalil	Marry Rashow	Mohammed Alsalibi	Taleb Omran
Bassem Ibrahim	**2.998**	3.489	3.661	3.577	3.689
Ibrahim Khalil	3.193	**2.803**	3.349	3.410	3.522
Marry Rashow	3.047	3.192	**2.585**	3.278	3.275
Mohammed Alsalibi	3.222	3.298	3.351	**2.567**	3.394
Taleb Omran	3.381	3.472	3.296	3.368	**2.851**

From this experiment, the accuracy rate is 100% since the correct author has been successfully identified for each tested text as the best cross-entropy shown in bold in the table is associated with the correct author of each text. In contrast, experiments with the same samples used above achieved 93.82% as the accuracy rate by analyzing the pattern of functional words usage by the authors (Shaker and Corne, 2010), which indicates the high performance of PPM character-based models. However, much larger datasets need to be investigated to more accurately determine the performance of the PPM approach, but these initial results are very promising.

6.3 Word Segmentation

The Oxford dictionaries define a 'word' as "a single distinct meaningful element of speech or writing, used with others (or sometimes alone) to form a sentence and typically shown with a space on either side when written or printed" (Oxford, 2014).

Similar to English words, Arabic words are naturally delimited by space — a feature that enables easier word identification than that allowed in languages that do not use space or other delimiters between words (e.g. Chinese and Japanese). Word segmentation is the process of determining the smallest unit (word) in a meaningful context. This process is a critical component of some NLP tasks, such as speech recognition.

For Arabic, however, word segmentation is a problematic issue (Zeki, 2005). The rich morphological nature of the Arabic language can represent problems in identifying correctly segmented words amongst several equally correct possibilities; incorrectly segmented words also add to the complexity of Arabic word segmentation. Table 6.2 shows examples of correct segmentation for the same word to demonstrate the problems that originate from the morphological qualities of the Arabic language when recognizing these words in handwritten text and speech recognition applications.

Table 6.2: Example of Arabic word identification.

Word	Correct Segmentations	English Translation
في ما	فيما	with
	في ما	in what
ورده	ورده	rose
	و رده	and his reply

As can be seen from the table, the words "في ما" and "ورده" ('which' and 'rose', respectively) can be segmented into more than one correct form; that is, "في ما" and "فيما" , and "ورده" and "و رده", respectively. These forms

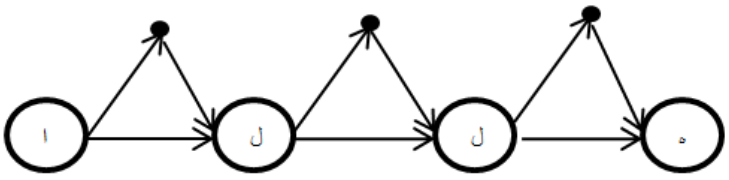

Figure 6.1: Space insertion of the segmentation model for the word "الله" .

are all correct in themselves but denote different contexts and meanings.

Teahan (1998) used the character-based PPM model for the word segmentation of English text; the model achieves 99.52 accuracy with the use of the Viterbi Algorithm (Viterbi, 1967). Using the model and the algorithm, word segmentation is implemented by first generating alternative segmentations of the text. These alternatives are produced through the insertion of spaces after each letter and then the best probable segmentation sequences are searched, as shown in Figure 6.1.

The most probable segmentation sequence that exhibits the best encoding performance, as determined by the PPM language model which is trained on the Brown corpus, is selected as the correct segmentation. Two segmentations are possible for each character: the character itself and the character followed by a space, denoted by "•" in Figure 6.1.

The possible sequences being searched are 2^n, where n is the length of the sequence. However, the Viterbi algorithm eliminates many of the search results because of the substantial pruning of poorly performing paths. PPM is a finite-context model and only one path for each similar context should remain after each symbol is processed. For example, the substring "الي" that forms the word "اليهم" would have eight search possibilities, and the best segmentations that present the best encoding sequence, as identified by the PPM model, are 22.5, 23.2 and 24.4. The rest of the alternative segmentations are discarded (Figure 6.2).

We use the character-based PPM technique, as shown in Figure 6.2, to solve the problem presented by Arabic word segmentation. The PPM

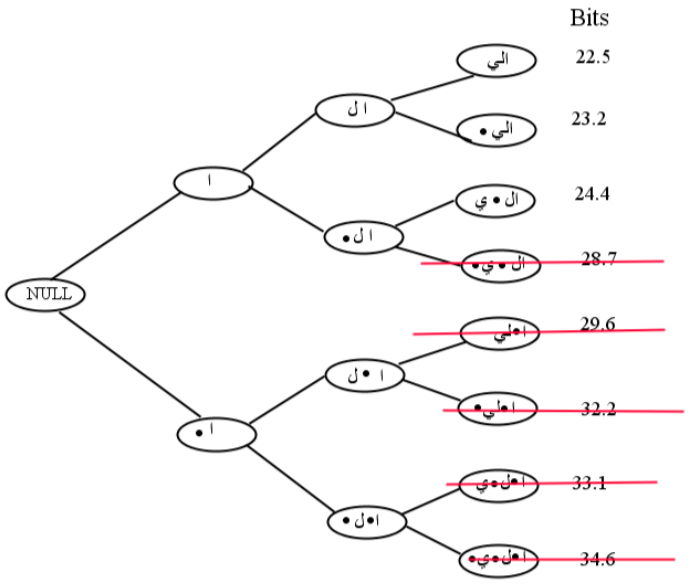

Figure 6.2: Segmentations process.

model is trained on the BACC corpus (containing spaces) to segment the CCA and ESAC corpora (with spaces removed). Table 6.2 shows the results of our analysis of segmentation for the aforementioned corpora. The table presents the calculated *recall rate, error rate and precision rate*, which enable us to measure the accuracy of the PPM-based model in segmenting Arabic text.

Recall rate is calculated by dividing the number of words that is successfully segmented by the PPM model (C) over the total number of words in the original text (N). *Error rate* is calculated by dividing the number of words that is unsuccessfully segmented by the PPM model (E) over the total number of words in (N). *Precision rate* is calculated by dividing the number of words that is successfully segmented (C) over the total number of words that are correctly and incorrectly segmented (L). The calculations are written as follows:

$$(\%)Recall\ rate = C/N \times 100, \tag{6.1}$$

$$(\%)Error\ rate = E/N \times 100, \tag{6.2}$$

$$(\%)Precision\ rate = C/L \times 100. \tag{6.3}$$

Table 6.3: Add caption

Test File	Size (bytes)	Recall Rate %	Error Rate %	Precision rate %
CCA	6289509	93.77	6.23	95.49
EASC	630321	94.65	5.34	96.24

The character-based PPM model achieves recall rates of 94% and 95% for CCA and EASC, respectively, indicating high accuracy of Arabic text segmentation. The 7% and 6% error rates are attributed to the rich morphology of Arabic text; the errors generated are those on proper names of people and places (e.g. "مكنيل" and "بافاروتي" which stand for 'Macneel' and 'Pavarotti' in CCA and EASC, respectively).

6.4 Correcting OCR Text

We apply the PPM character-based models to the output of the Tesseract OCR software, which is the most accurate freely available OCR engine, as stated by Google (Google, 2014). Tesseract was developed originally by HP Labs in 1985 to 1995 and was subsequently improved by Google.

No OCR engine is dedicated to Arabic text so far because this language is typically included as a second option and not the main language for recognition in some OCR programmes that offer OCR for Arabic text. An exception is that referred to by Sakher Inc., who claim a 96.8% accuracy for normal-quality scanned text (Sakher, 2014). We were not able to independently verify this claim, however, as we were not able to obtain the software.

For our experiments to evaluate how effective PPM is at correcting Arabic OCR output, first 2 pages from Taha Hussain's famous novel 'Alayam', printed in 1996 (Hussain, 1996) has been scanned. These pages, scanned at 250 dpi, contain 608 words that are used as test text. A sample of the scanned text is shown in Figure 6.3.

The order-5 character-based PPM model is trained on the BACC corpus to build the language models that will be used to correct the errors generated by the Tesseract OCR. The training is executed in the same manner

وأكبر ظنه أن هذا الوقت كان يقع من ذلك اليوم في فجره أو في عشائه. يرجح ذلك لأنه يذكر أن وجهه تلقى في ذلك الوقت هواءً فيه شيء من البرْد الخفيف الذي لم تذهب به حرارة الشمس. ويرجح ذلك لأنه على جهله حقيقة النور والظلمة, يكاد يذكر أنه تلقى حين خرج من البيت نورًا هادئًا خفيفًا لطيفًا كأن الظلمة تغشى بعض حواشيه.

Figure 6.3: Sample of scanned text (Hussain, 1996).

as the previous method used in the word segmentation experiments.

Teahan (1998) defines the error generated by the OCR engine as a 'confusion' that requires correction using a 'observed → corrected' rule, which denotes the transformation from the observed state to the correct state. A sample of confusions generated by the Tesseract OCR in the recognised test text is shown in Table 6.4.

Table 6.4: Sample of confusions generated from the Tesseract output for 'Alayam'.

confusions		
ا→م	م→ن	ذ → ن
ه→,	ش→ث	ت→ث
ي→ه	ج→ق	ء→ة
ف→ل	م→ه	ر→ن
و→,	م→ع	ه→م
م→,	م→ص	ح→ع
ت→ق	ي→م	ب→ن
ف→ق	ب→ي	ح→ع
ة→ة	ث→ث	ن→ن
ع→ع	ه→ه	ف→ف

As an example, "ن → ذ" denotes the letter "ن" corrected to the letter "ذ". When a match occur to the observation strings in the text, the alternative corrected form is generated. We search for the most probable sequence amongst all the different possible correction using the Viterbi algorithm. These defined confusions are used to limit the search possibilities required. The number of errors generated by the Tesseract OCR engine increases as the number of scanned texts increases. A sample of the output generated by the Tesseract OCR engine for the same sample shown above is presented in Figure 6.4.

وأكبر ظنه أن من الوقت كان يقع من ذلك اليوم في فجره أو في عشهاته0 يرجع ذلك لأنه يذكر أن وجهه تلقى فهو
ذلك الوقت هواة فيه شيء من البرد الخفيف الذي لم تذهب به حرارة الشمس ويرجع ذلك لانه على جهله حقيقه
.النور والظلمة. يكاد يذكر أنه تلقى حين خرق من البيت نونا هادئا خليلا لطيفا كان الظلمة تغشى بل حواشيه

Figure 6.4: Sample of output generated by the Tesseract OCR engine.

The accuracy of the Tesseract OCR in recognising the scanned test text is 62.3%, despite the high quality of the scanned text (250 dpi). The Tesseract OCR engine produces 64 errors out of 229. In 64 cases, an entire word is completely changed to a different word – an error cannot be corrected by the PPM correction method to recover the original words in the scanned text. An example of such an error is "بل", which should be written as "بعض" to denote 'some'. Out of the 229 errors, 165 are the target for correction as they are errors produced by the Tesseract OCR; in these errors, one or more letters in the original words need to be changed.

As shown in Figure 6.4, a number of errors occur in the text recognised by the Tesseract OCR. These errors include "خرق" and "يرجع", which should be written as "خرج" and "يرجح" to denote 'went out' and 'probably', respectively. The Tesseract OCR generates errors in the letters "ق" and "ع", which should be written as "ج" and "ح". These are denoted by "ق" → "ج" and "ع" → ح in Table 6.4 because the confusion is observed in the output produced by the OCR engine.

Numerous multiple errors (errors in more than one letter) occur in the same word, where the context of a word is completely altered as a result of the poor performance of the Tesseract OCR engine. For example, "عشهاته0" should be represented as ".عشائه" in Figure 6.4. It is instead recognised "ش", "ئ" and "." as "ثه", "ت" and "0".

Figure 6.5 shows the corrected Tesseract OCR output after correction by the PPM model for the sample shown in Figure 6.4. The PPM model successfully corrects 133 words out of the 165 target words that are incorrectly recognised by the Tesseract OCR engine.

As we can see, multiple errors occur on the same word. For example,

وأكبر ظنه أن من الوقت كان يقع من ذلك اليوم في فجره أو في عثماته. يرجح ذلك لأنه يذكر أن وجهه تلقى فهو ذلك الوقت هواء فيه شيء من البرد الخفيف الذي لم تذهب به حرارة الشمس ويرجح ذلك لانه على جهله حقيقة النور والظلمة. يكاد يذكر أنه تلقى حين خرج من البيت نوراً هادئاً خفيفاً لطيفاً كان الظلمة تغشى بل حواشيه

Figure 6.5: Corrected Tesseract OCR output after correction by the PPM model.

"عثماته." in the text corrected by PPM is denoted as "عشائه." in the original scanned text and "عثهاته0" in the text recognised by the Tesseract OCR. This error is uncorrected because of the poor performance of Tesseract in recognising such words. PPM successfully corrects words such as "يرجح", "حقيقة" and "خرج", as well as punctuation such as ".", which is written as "0" in the text recognised by the Tesseract OCR engine.

Overall, the character-based PPM model corrects 133 out of the 165 errors generated by the Tesseract OCR engine. This result is despite the poor recognition (62.3% accuracy) of Arabic text by this OCR engine. Nevertheless, after the application of the PPM models, the Tesseract accuracy increases to 84.21% — a significant improvement.

6.5 Conclusion

Character-based PPM models have been successfully used to solve several problems presented by Arabic text; these problems include word segmentation, OCR text and authorship attribution. The model achieves approximately 94% accuracy in segmenting words and 80.60% accuracy in improving the output of OCR texts. The low percentage for OCR text correction is due primarily to the poor performance of Tesseract OCR engine for recognising Arabic text, despite the high quality of the scanned document. This percentage improves by 22%, a significant improvement. The Tesseract OCR engine can be substantially improved if PPM models are embedded into the Tesseract OCR system. Also, PPM Character-based models successfully employed to identify the correct author of anonymous Arabic text amongst several alternative authors as shown in section 6.1.

Chapter 7

Summary and Future Work

Contents

7.1 Summary and Conclusions

Adaptive models of Arabic text are the main concern of this thesis. We propose several improvements for standard PPM models that significantly enhance compression results. These improvements are CS-PPM, BS-PPM and dotted lossless and non-dotted lossy compression for PPM. The experimental results for these new methods are discussed in the following sections.

7.1.1 BS-PPM: Bigraph Substitutions for PPM Models

By its nature, languages contain words that have many repeated bigraph characters. For Arabic texts, the top ranked bigraphs take up a significant percentage of any text. These bigraphs can be employed to enhance the compression performance over standard PPM. The BS-PPM technique involves sequentially processing text and replacing the most frequent bigraphs in the order that they appear, with unique single characters assigned to each bigraph. For most natural-language texts, we found that replacing the top 100 bigraphs works best with the alphabet being expanded with 100 more characters as a result. In the postprocessing operation, the reverse mapping is performed by replacing the new unique characters with the original equivalent two-byte bigraphs.

BS-PPM achieves the best compression results for Arabic text, as well as for languages that use Arabic script (e.g. Persian and Kurdish). Moreover, it presents excellent results for other languages, such as English, Armenian, Russian and Welsh.

7.1.2 CS-PPM: Character Substitutions for PPM Models

The CS-PPM technique substitutes all the UTF-8 multi-byte or single-byte character sequences in the original text. Preprocessing produces two output files: one that contains a stream of UTF-8 characters, called the vocabulary stream, and another that contains a list of symbol numbers,

called the symbol stream. Both the output stream files require compression during the encoding stage. Therefore, two files also need to be separately decoded, with the vocabulary stream file requiring decoding first to enable the definition of the reverse mapping between symbols and UTF-8 characters during the post-processing stage.

CS-PPM achieves the best compression results for Arabic text, as well as for languages that use Arabic script (e.g. Persian and Kurdish). It also generates excellent results for other languages as well such as English, Armenian, Russian, Welsh and Chinese.

7.1.3 CSA-PPM: Character Substitution of Arabic for PPM

Unlike CS-PPM which produces two outputs (symbol and vocabulary streams), CSA-PPM involves the production of one output file via preprocessing. This procedure creates a symbol stream whilst pre-defining the range of Arabic characters in the UTF-8 scheme, thereby enabling the direct substitution of characters with the equivalent number of characters used in the UTF-8 scheme. Direct substitution eliminates the need for a vocabulary stream, which decreases the size of the output compressed file. CSA-PPM presents excellent results in compressing Arabic text.

7.1.4 Dotted Lossless and Non-dotted Lossy Compression of Arabic Text

In this method, we exploit the historical feature of the Arabic language–that is, its non-dotted nature until late 800 A.D.—to improve the compression rate of PPM. The improvement is achieved by normalising a dotted letter to a non-dotted one, in which dots are removed to generate the lossy version of the original text. We then compress the result by PPM and then automatically recover the original dotted text using the Viterbi algorithm. This method also achieves excellent results for Arabic scripted text.

7.1.5 Arabic Word- and Tag-based Models

Unlike the experiments on English text for which word-and tag-based models achieve excellent compression, those on Arabic text using word- and tag-based models present contrasting results. That is, the highest performing models are the character-based BS-PPM and CS-PPM types, followed by the word- and tag-based models. As previously stated, an unclear issue is whether tag-based models of Arabic text will consistently perform poorly compared with other methods because of the limited availability of tagged corpora for Arabic text.

7.1.6 Authorship Attribution

Authorship attribution is the process of identifying the exact author for text of unknown authorship. There are a very limited number of studies that are concerned specifically with Arabic language compared to other languages such as English. PPM character-based compression models have been employed to solve the problem of authorship attribution which achieves excellent results for Arabic text.

7.1.7 Arabic Word Segmentation

Word segmentation is problematic for Arabic text because it is a rich morphological language. Character-based PPM models have been successfully used to segment Arabic words – a critical component of NLP applications, such as speech recognition. The character-based PPM models exhibit an accuracy of 93.77% and 94.65% for the CCA and EASC corpora, respectively.

7.1.8 Correcting OCR Text for Arabic

OCR applications that accurately recognise Arabic text are lacking. One of the available OCR engines for Arabic text is the Tesseract OCR engine powered by Google. Numerous errors are generated by Tesseract OCR in recognising Arabic text, motivating the use of character-based PPM models

to correct the errors. The approach improves the accuracy of the output of the Tesseract OCR engine from 62.3% to 84.2%, despite the poor output produced by the OCR engine – a significant improvement in accuracy. As stated earlier, Tesseract can be substantially improved by embedding character-based PPM models into its system.

7.2 Review of Hypothesis and Research Questions

The characteristics of Arabic have been employed successfully to optimise PPM models which as a result achieve significant improvement in terms of compression rate over standard PPM for Arabic text and other languages that use Arabic script, such as Persian and Kurdish. This was done by producing the new BS-PPM and CS-PPM models and the lossless dotted and lossy non-dotted variants of PPM. We have shown that the different nature of languages is an important perspective that should be taken into consideration when processing these languages for NLP applications.

As mentioned in Chapter 1, the broad aim of this thesis is to investigate the manner by which we can establish adaptive computer models of Arabic language that exhibit effective compression performance. More specifically, we provide answers to the research questions listed in Section 1.3, discussed as follows.

What is the best computer model for compressing Arabic text?
As indicated in the experiments on the investigated models, the best computer models for compressing Arabic text are BS-PPM and CS-PPM, which are character-based models, followed by dotted lossless and non-dotted lossy compression models. These models significantly outperform standard PPM.

What are the disadvantages presented by the current models when applied to Arabic text?
The main drawback of standard PPM models when applied to Arabic text originates from the nature of the language. Arabic is considered a rich

morphological language and substantially effects the prediction made by standard PPM models. In order to capture the properties of the Arabic language, training on much larger texts via high-order models is required by standard PPM to optimise its models. Clearly, this is not always possible if smaller texts are being encoded and therefore alternative compression schemes may yield better results.

How well do these models perform in several natural language processing applications?

Character-based PPM models have been applied to authorship attribution, word segmentation and OCR text correction for Arabic language. The models exhibit significant improvements in NLP application despite the fact that Arabic is a rich morphological language.

Can new language models be devised that lead to significant improvements in Arabic text compression?

Three new language models have been designed especially for the Arabic language. These models are Bigraph Substitution for PPM (BS-PPM), Character Substitution for PPM (CS-PPM) and Lossless dotted and lossy non-dotted PPM which significantly outperform standard PPM in terms of compression performance.

7.3 Future Work

The idea behind the use of BS-PPM is driven by the fact that Arabic words consists of many repeated bigraphs. These features are the factors that compel us to substitute the bigraphs with one character, thereby resulting in significant improvements even under low-order PPM. Therefore, BS-PPM can be optimised by looking into syllable-based models instead of character- or word-based models as units of encoding for compression. This approach may potentially yield better results than BS-PPM because the probability estimated by this proposed method could captures the properties of the Arabic language better and possibly those of other lan-

guages.

Further investigations of tag-based models require the establishment of new tagged corpora that represent contemporary Arabic text in terms of size and genre with different tag sets.

In our future work, we intend to tag the BBCCA corpus as a balanced corpus of the Arabic language that has been presented in Chapter 3 as a part of project entitled “Natural Language Processing Resources for Arabic with focus on Saudi Arabian language usage”. This project is a joint research endeavour supported by Tabuk University and King Abdul Aziz City for Science and Technology.

Bibliography

Jürgen Abel and William Teahan. Universal text preprocessing for data compression. *Computers, IEEE Transactions on*, 54(5):497–507, 2005.

Latifa Al-Sulaiti and Eric Steven Atwell. The design of a corpus of contemporary Arabic. *International Journal of Corpus Linguistics*, 11(2):135–171, 2006.

Abolfazl AleAhmad, Hadi Amiri, Ehsan Darrudi, Masoud Rahgozar, and Farhad Oroumchian. Hamshahri: A standard Persian text collection. *Knowledge-Based Systems*, 22(5):382–387, 2009.

Khaled Alhawiti and William Teahan. Universal text preprocessing and postprocessing for PPM using alphabet adjustment. In *Data Compression Conference, 2014. Proceedings. DCC*, 2014.

Maha Alrabiah, Al-Salman AbdulMalik, and Eric Atwell. The design and construction of the 50 million words KSUCCA King Saud university corpus of classical Arabic. In *Second Workshop on Arabic Corpus Linguistics, 2013. Proceedings. WACL2 2013*, pages 5–8. Lancaster University, UK, 2013.

Latifa Alsuliti. Designing and developing a corpus of contemporary Arabic. Master's thesis, University of Leeds, 2004.

Majed Khair Baik. *History of Arabic Language*, volume 1. Dar Sa'ad Aldin, Damascus, 1992.

Timothy Bell, Ian H Witten, and John G Cleary. Modeling for text compression. *ACM Computing Surveys (CSUR)*, 21(4):557–591, 1989.

Timothy C Bell, John G Cleary, and Ian H Witten. *Text compression*. Prentice-Hall, Inc., 1990.

Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

BuiltWith. UTF-8 usage statistics. Online, 2012. URL `http://trends.builtwith.com/encoding/UTF-8`. [Accessed 21 Jun 2013].

Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Online, 1994. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.5254&rep=rep1&type=pdf`.

Ebru Çelikel Çankaya, Venka Palaniappan, and Shahram Latifi. Exploiting redundancy to achieve lossy text compression. *Pamukkale University Journal of Engineering Sciences*, 16(3), 2011.

Hans Christensen. HC Corpora. Online, 3013. URL `http://www.corpora.heliohost.org/download.html`. [Accessed 14 Feb 2013].

John G Cleary and William J Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2 and 3):67–75, 1997.

John G Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32 (4):396–402, 1984.

Royal Hashemite Court. Speech of King Abdullah II. Online, 2013. URL `http://kingabdullah.jo`. [Accessed 19 Feb 2013].

Mahmoud El-Haj, Udo Kruschwitz, and Chris Fox. Using Mechanical Turk to create a corpus of Arabic summaries. In *Proceedings of the Seventh conference on International Language Resources and Evaluation*, 2010.

NC Ellis, C O'Dochartaigh, W Hicks, M Morgan, and N Laporte. Cronfa electroneg o gymraeg (ceg): A 1 million word lexical database and frequency count for Welsh, 2001.

Peter M Fenwick and Simon Brierley. Compression of unicode files. In *Data Compression Conference*, page 547, 1998.

W Nelson Francis and Henry Kucera. Brown corpus manual. *Brown University Department of Linguistics*, 1979.

Google. Tesseract OCR engine. Online, 2014. URL `https://code.google.com/p/tesseract-ocr`. [Acceseed 10 Feb 2014].

Harold Stanley Heaps. *Information retrieval: Computational and theoretical aspects*. Academic Press, Inc., 1978.

Nigel Horspool and Gordon Cormack. Constructing word-based text compression algorithms. In *Data Compression Conference*, pages 62–71, 1992.

Paul Glor Howard. The design and analysis of efficient lossless data compression systems. Technical report, Citeseer, 1993.

Taha Hussain. *Alayam*, volume 3. Cairo, Dar Alshoroq, 1996.

Stig Johansson. The LOB corpus of British English texts: presentation and comments. *ALLC journal*, 1(1):25–36, 1980.

Patrick Juola, John Sofko, and Patrick Brennan. A prototype for authorship attribution studies. *Literary and Linguistic Computing*, 21(2):169–178, 2006.

Katzner Kenneth. *The Languages of the World (2002)*. London and New york, third edition, 2002.

Moshe Koppel, Jonathan Schler, and Shlomo Argamon. Computational methods in authorship attribution. *Journal of the American Society for information Science and Technology*, 60(1):9–26, 2009.

Paul Lewis, Simons Gary, and Fennig Charles. *Ethnologies Languages of the World*. Dallas, Texas: SIL International, sixteenth edition edition, 2009.

Mohamed Maamouri, Bies Ann, Buckwalter Tim, and Jin Hubert. Arabic Treebank: Part 1 version 3.0 (pos with full vocalization + syntactic analysis). Linguistic Data Consortium, 2005.

Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

Miniwatts Marketing. Internet world stats. Online, 2013. URL `http://www.internetworldstats.com/stats.htm`. [Accessed 19 July 2013].

A M McEnery and Zhonghua Xiao. The Lancaster corpus of Mandarin Chinese: A corpus for monolingual and contrastive language study. *Religion*, 17:3–4, 2004.

George A Miller, Edwin B Newman, and Elizabeth A Friedman. Length-frequency statistics for written English. *Information and control*, 1(4): 370–389, 1958.

Alistair Moffat. Word-based text compression. *Software: Practice and Experience*, 19(2):185–198, 1989.

Alistair Moffat. Implementing the PPM data compression scheme. *Communications, IEEE Transactions on*, 38(11):1917–1921, 1990.

Tim Ng, Kham Nguyen, Rabih Zbib, and Long Nguyen. Improved morphological decomposition for Arabic broadcast news transcription. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 4309–4312. IEEE, 2009.

Oxford. English dictionary. Online, 2014. URL `http://www.oed.com`. [Accessed 10 Jan 2014].

P Rayson. Wmatrix corpus analysis and comparison tool. Online, 2009. URL `http://ucrel.lancs.ac.uk/wmatrix`.

Sakher. Arabic language technology. Online, 2014. URL `http://www.sakhr.com/index.php/en`. [Accessed 12 Jan 2014].

Edward Sapir. *Language: an introduction to the study of speech*. Courier Dover Publications, 1921.

Julian Seward. Bzip2. Online, 1998. URL `http://sources.redhat.com/bzip2`.

Kareem Shaker and David Corne. Authorship attribution in Arabic using a hybrid of evolutionary search and linear discriminant analysis. In *Computational Intelligence (UKCI), 2010 UK Workshop on*, pages 1–6. IEEE, 2010.

Abdelhadi Soudi, Ali Farghaly, Günter Neumann, and Rabih Zibib. *Challenges for Arabic Machine Translation*, volume 9. John Benjamins, 2012.

William Teahan and Khaled Alhawiti. Design compilation and preliminary statistics of compression corpus of written Arabic. Technical report, Bangor University, 2013. URL `http://pages.bangor.ac.uk/~eepe04/index.html`.

William Teahan and Khaled Alhawiti. A compression-based method for ranking n-gram differences between texts and corpora evaluation. In *7th Saudi students scientific conference 2014. Proceedings. SSSC 2014*, Feb 2014.

William John Teahan. *Modelling English text*. PhD thesis, University of Waikato, New Zealand, 1998.

William John Teahan, Stuart Inglis, John G Cleary, and Geoffrey Holmes. Correcting English text using PPM models. In *Data Compression Conference, 1998. DCC'98. Proceedings*, pages 289–298. IEEE, 1998.

WJ Teahan and David J Harper. Combining ppm models using a text mining approach. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 153–162. IEEE, 2001.

UmAlqura. Letter by Prophet Muhammad peace be upon him. Online, 2014. URL `http://uqu.edu.sa/page/ar/39589`. [Accessed 15 Jan 2014].

Kees Versteegh and CHM Versteegh. *The Arabic Language*. Columbia University Press, 1997.

Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, 1967.

W3Techs. Usage of character encoding for websites. Online, 2013. URL `http://w3techs.com/technologies/overview/character_encoding`. [Accessed 17 Jun 2013].

Ernest Weekley. *An etymological dictionary of modern English*, volume 2. Courier Dover Publications, 2012.

Terry Winograd. Language as a cognitive process. vol. 1: Syntax. *Reading, MA: Addison-Wesley, 1983*, 1, 1983.

Ian H Witten and Timothy C Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085–1094, 1991.

Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

Ian H Witten, Timothy C. Bell, Alistair Moffat, Craig G. Nevill-Manning, Tony C. Smith, and Harold Thimbleby. Semantic and generative models for lossy text compression. *The Computer Journal*, 37(2):83–87, 1994.

Peiliang Wu. *Adaptive Models of Chinese Text*. PhD thesis, University of Wales, Bangor, 2007.

Peiliang Wu and William John Teahan. A new PPM variant for Chinese text compression. *Natural Language Engineering*, 14(3):417–430, 2008.

Ahmed M Zeki. The segmentation problem in Arabic character recognition the state of the art. In *Information and Communication Technologies, 2005. ICICT 2005. First International Conference on*, pages 11–26. IEEE, 2005.

George Kingsley Zipf. Human behavior and the principle of least effort. *Reading, MA7 Addison Wesley*, 1949.

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Appendix A

Appendix A – BS-PPM

Included in this appendix is the code that can be used to perform the preprocessing and postprocessing stages of BS-PPM. The code comprises the following files: preprocess.c and postprocess.c. These are described in more detail below. preprocess.c does the preprocessing, taking an input file into a stream of numbers and replaces those found in the input stream. postprocessing.c does the reverse mapping for the postprocessing stage. The output file processed can then be used as input to the encode program.

A.1 preprocessing.c

```
/* Preprocesses the input file into a stream of numbers. It also
optionally loads a file of bigraphs and their replacement number,
one per line, and replaces those found in the input stream. */
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include <unistd.h>
#define MAX_BIGRAPHS 101         /* Maximum numbr of bigraphs */
char Bigraphs_filename [128];    /* Name of Bigraphs file */
int Bigraphs [MAX_BIGRAPHS][3];  /* The bigraphs;
/* The bigraphs consist of three numbers [0] is the replacement
number; [1] and [2] are the bigraph character numbers. */
int Bigraphs_size = 0;  /* The number of bigraphs in the array */
int Bigraphs_max = 0;   /* Maximum number of bigraphs. */
void
usage (void)
{
    fprintf (stderr,
     "Usage: preprocess [options] <input-text\n"
     "\n"
     "options:\n"
     "  -b fn\tfilename of file containing a list of bigraphs=fn\n"
     "  -m n\tmaximum number of bigraphs to replace=n\n"
    );
    exit (2);
}
void init_arguments (int argc, char *argv[])
{
    int opt;
    extern char *optarg;
    extern int optind;
```

```
    /* get the argument options */
    Bigraphs_filename [0] = '\0';
    while ((opt = getopt (argc, argv, "b:m:")) != -1)
      {
        switch (opt)
        {
          case 'b':
        strcpy (Bigraphs_filename, optarg);
        break;
          case 'm':
        Bigraphs_max = atoi (optarg);
        break;
        default:
        usage ();
        break;
      }
      }
    for (; optind < argc; optind++)
  usage ();
}
void load_bigraphs ()
/* Load the bigraphs from the file named
 Bigraphs_filename if specified. */
{
    FILE *fp;
    int rp, bg1, bg2;
    if (Bigraphs_filename [0] != '\0')
      {
        fprintf (stderr,
"Loading bigraphs from file %s\n",
Bigraphs_filename);
        if ((fp = fopen (Bigraphs_filename, "r")) == NULL)
          {
   fprintf (stderr,
 "preprocess: can't open bigraphs file %s\n",
 Bigraphs_filename);
             exit (1);
}
        while ((fscanf (fp, "%d %d %d", &rp, &bg1, &bg2) != EOF))
        { /* found the next bigraph */
          if (Bigraphs_size == MAX_BIGRAPHS)
            {
              fprintf (stderr, "preprocess: too many bigraphs %s\n",
                Bigraphs_filename);
              exit (1);
}
            Bigraphs [Bigraphs_size][0] = rp;
            Bigraphs [Bigraphs_size][1] = bg1;
            Bigraphs [Bigraphs_size][2] = bg2;
            Bigraphs_size++;
            if (Bigraphs_max && (Bigraphs_size >= Bigraphs_max))
            break;
      }
      }
    fprintf (stderr, "Bigraphs loaded\n");
}
int find_bigraph (int pc, int cc)
/* Returns 0 if no bigraph is found, or the replacement
number if the bigraph defined by the previous character
```

```
(pc) and the current character (cc). pc could also be a
 replaced bigraph.*/
{
    int p;
     if (Bigraphs_size == 0)
      return 0; /* no bigraph found */
    p = 0;
    for (;;)
      { /* See if the next bigraph matches */
      if (p >= Bigraphs_size)
      return 0; /* no bigraph found */
      if ((pc == Bigraphs [p][1]) && (cc == Bigraphs [p][2]))
      return Bigraphs [p][0];
        p++;
      }
    /* We shouldn't get here */
    return 0; /* no bigraph found */
}
void preprocess_file (FILE *fp)
/* Preprocess the input file fp by first replacing
matching bigraphs as found in the Bigraphs array,
and then outputting a number which represents the
resultant character. */
{
  int cc, pc, p, first;
    pc = 0;
    first = 1;
    for (;;)
      { /* read through the file, replacing bigraphs */
     cc = getc (fp);
     p = find_bigraph (pc, cc);
     if (p)
       {
       pc = p;
       if (cc != EOF)
       cc = getc (fp);
       }
    if (!first)
     printf ("%d\n", pc);
    if (cc == EOF)
     break;
     first = 0;
     pc = cc;
      }
}
int main(int argc, char *argv [])
{
    init_arguments (argc, argv);
    load_bigraphs ();
    preprocess_file (stdin);
    exit (0);
}
```

A.2 postprocess.c

```
/* Postprocesses the input file consisting of a stream
of numbers and writes it out as an ASCII text file.
It also optionally loads a file of bigraphs, one
per line, and expands those that are found in the
input stream. The input file can be output from
the decode program.*/
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include <unistd.h>
#define MAX_BIGRAPHS 101 /* Maximum number of bigraphs */
char Bigraphs_filename [128]; /* Name of Bigraphs file */
int Bigraphs [MAX_BIGRAPHS][3];  /* The bigraphs*/
/* The bigraphs consist of three numbers [0] is the
replacement number; [1] and [2] are the bigraph character
numbers. Usually, these will be their ASCII numbers, but
they can also be other bigraph numbers specified elswhere
in the array. */
int Bigraphs_size = 0; /* The number of bigraphs */
int Bigraphs_max = 0;  /* Maximum number of bigraphs*/
void usage (void)
{
       fprintf (stderr,
     "Usage: preprocess [options] <input-text\n"
     "\n"
     "options:\n"
     "  -b fn\tfilename of file contain a list of bigraphs=fn\n"
     "  -m n\tmaximum number of bigraphs to replace=n\n"
             );
       exit (2);
}
void init_arguments (int argc, char *argv[])
{
    int opt;
    extern char *optarg;
    extern int optind;
    /* get the argument options */
    Bigraphs_filename [0] = '\0';
    while ((opt = getopt (argc, argv, "b:m:")) != -1)
      {
      switch (opt)
        {
          case 'b':
          strcpy (Bigraphs_filename, optarg);
          break;
            case 'm':
          Bigraphs_max = atoi (optarg);
          break;
          default:
          usage ();
          break;
        }
      }
    for (; optind < argc; optind++)
  usage ();
}
void load_bigraphs ()
/* Load the bigraphs from the file named Bigraphs_filename */
```

```
{
    FILE *fp;
    int rp, bg1, bg2;
    if (Bigraphs_filename [0] != '\0')
      {
      fprintf (stderr, "Loading bigraphs from file %s\n",
      Bigraphs_filename);
      if ((fp = fopen (Bigraphs_filename, "r")) == NULL)
       {
         fprintf (stderr, "preprocess: can't open bigraphs
     file %s\n", Bigraphs_filename);
         exit (1);
       }
     while ((fscanf (fp, "%d %d %d", &rp, &bg1, &bg2) != EOF))
       { /* found the next bigraph */
        if (Bigraphs_size == MAX_BIGRAPHS)
         {
        fprintf (stderr, "preprocess: too many bigraphs %s\n",
        Bigraphs_filename);
        exit (1);
         }
        Bigraphs [Bigraphs_size][0] = rp;
        Bigraphs [Bigraphs_size][1] = bg1;
        Bigraphs [Bigraphs_size][2] = bg2;
          Bigraphs_size++;
        if (Bigraphs_max && (Bigraphs_size >= Bigraphs_max))
         break;
       }
      }
    fprintf (stderr, "Bigraphs loaded\n");
}
int find_bigraph_replacement (int nn)
/* Returns -1 if nn is not found as a replacement
number in the Birgraphs array. If it is found, it
returns its index in the Bigraphs array. */
{
    int p;
    if (Bigraphs_size == 0)
      return -1; /* no bigraph replacement found */
    p = 0;
    for (;;)
      { /* See if the next bigraph replacement matches */
      if (p >= Bigraphs_size)
        return -1; /* no bigraph replacement found */
      if (nn == Bigraphs [p][0])
        return p;
      p++;
      }
    /* We shouldn't get here */
    return -1; /* no bigraph replacement found */
}
void postprocess_file (FILE *fp)
/* Postprocess the input file fp by first expanding
numbers that represent bigraphs as found in the
Bigraphs array, and then outputting the ASCII
character or bigraph. */
{
    int nn, p;
    while ((fscanf (fp, "%d", &nn) != EOF))
```

```
      {
      p = find_bigraph_replacement (nn);
       if (p < 0) /* Not a bigraph replacement number */
          putchar (nn);
       else
        {
          putchar (Bigraphs [p][1]);
          putchar (Bigraphs [p][2]);
        }
      }
}
int main(int argc, char *argv [])
{
    init_arguments (argc, argv);
    load_bigraphs ();
    postprocess_file (stdin);
    exit (0);
}
```

Appendix B

Appendix B – CS-PPM

Included in this appendix is the code that can be used to perform the preprocessing and postprocessing stages of CS-PPM. The code comprises the following files: utf8-encode.c, utf8-decode.c and keys.c. These are described in more detail below. utf8-encode.c does the preprocessing, taking a single file as input and produces two streams, the vocabulary stream (all possible UTF-8 character existing in the text) and the symbols stream (sequence of numbers equivalent to the character). utf8-decode.c does the reverse mapping for the postprocessing stage. keys.c is used by the preprocessing and postprocessing code and defines code for storing keys (strings) and their frequencies of occurrence in a dictionary.

B.1 utf8-encode.c

```
/* Outputs two files - a file containing the unique
characters as they appear in the input text, and
another file containing the sequence of symbol numbers
associated with each bigraph on separate lines in the
same order that they appear in the text, with symbol
number 1 being associated with the first bigraph that
appeared, symbol number 2 with the second and so on. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include "keys.h"
#define UTF8_MAX 4
FILE *Output_fp = NULL;
FILE *Vocab_fp = NULL;
void
usage (void)
{
 fprintf (stderr,
 "Usage: process [options] <input-text-file\n"
 "\n"
 "options:\n"
 "  -N \toutput file is list of numbers on separate lines\n"
 "  -o fn\tconverted bigraphs text output filename=fn (required)\n"
 "  -v fn\tbigraphs vocabulary output filename=fn (required)\n"
        );
  exit (2);
}
void
```

```
init_arguments (int argc, char *argv[])
{
  extern char *optarg;
  extern int optind;
  int opt, vocab_file_found, output_file_found;
  /* get the argument options */
  vocab_file_found = 0;
  output_file_found = 0;
  while ((opt = getopt (argc, argv, "No:v:")) != -1)
    switch (opt)
      {
      case 'o':
      if ((Output_fp = fopen (optarg, "w")) == NULL)
      {
        fprintf (stderr, "Encode: can't open output file %s\n",
         optarg);
        exit (1);
      }
      output_file_found = 1;
      break;
      case 'v':
      if ((Vocab_fp = fopen (optarg, "w")) == NULL)
      {
        fprintf (stderr, "Encode: can't open vocabulary file %s\n",
        optarg);
        exit (1);
      }
      vocab_file_found = 1;
      break;
      default:
      usage ();
      break;
      }
  if (!vocab_file_found)
    {
      fprintf (stderr, "\nFatal error: missing
vocabulary filename\n\n");
      usage ();
    }
  if (!output_file_found)
    {
      fprintf (stderr, "\nFatal error: missing
output filename\n\n");
      usage ();
    }
  for (; optind < argc; optind++)
    usage ();
}
int  UTF8(FILE *fp, char *position) {
    int First_byte[] = {192, 224, 240};
    int cc, dd;
    memset(position, 0, UTF8_MAX + 1);
    position[0] = getc(fp);
    if (position[0] == EOF) {
        return 0;
    }
    cc = 0;
    if ((position[0] & First_byte[0]) == First_byte[0]) cc++;
    if ((position[0] & First_byte[1]) == First_byte[1]) cc++;
```

```
    if ((position[0] & First_byte[2]) == First_byte[2]) cc++;
    dd = 0;
    while (dd < cc) {
        dd++;
        position[dd] = getc(fp);
    }
    return cc + 1;
}
int main(int argc, char **argv)
{
    char *position = (char*)calloc(UTF8_MAX + 1, sizeof(char));
    struct keys_type Keys;
    int new, symbol, id_count = 1;
    keys_init_keys( &Keys );
    init_arguments (argc, argv);
    while (UTF8(stdin, position))
      {
        new = keys_add_keys( &Keys, position, id_count );
        if (new)
      {
        fprintf (Vocab_fp, "%s", position);
        id_count++;
      }
      symbol = keys_find_id ( &Keys, position );
      fprintf (Output_fp, "%d\n", symbol);
        /*printf("%s\n", position);*/
      }
    fclose (Vocab_fp);
    fclose (Output_fp);
    return 0;
}
```

B.2 utf8-decode.c

```
/* Outputs the UTF-8 characters converted from a
sequence of symbol numbersfound in the input file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <assert.h>
#include "keys.h"
#define UTF8_MAX 4
#define SYMBOLS_ESCAPE 128   /* Used for coping with
characters that are not Arabic or ASCII */
int decoded_byte_value (int ch)
{
  if (ch >= 128)
    return - (ch - 128);
  else
    return (ch);
}
int
getNumber (FILE *fp, unsigned int *number)
/* Returns the next number from input stream fp. */
{
    unsigned int n;
    int result;
    n = 0;
    result = fscanf (fp, "%u", &n);
    switch (result)
      {
        case 1: /* one number read successfully */
        break;
        case EOF: /* eof found */
        break;
        case 0:
        fprintf (stderr, "Formatting error in file\n");
        exit (1);
        break;
        default:
        fprintf (stderr, "Unknown error (%i) reading file\n", result);
        exit (1);
      }
    *number = n;
    return (result);
}
int main(int argc, char **argv)
{
    printf("\xef\xbb\xbf");
    char *char_bytes = (char*) calloc(UTF8_MAX + 1, sizeof(char));
    char *char_bytes1;
    char char_bytes2 [3];
    unsigned int byte1, byte2;
    struct keys_type Keys;
    int new, n, symbol, id_count = 0;
    unsigned int number;

keys_init_keys( &Keys ); /* Initialise the characters
lookup table */
    /* Preload the Keys lookup table with ASCII
characters first (symbols 0 to 127, followed
```

```
by a special UTF8_ESCAPE character next
(symbol number 128) to be used to cope with
unrecognized UTF8 characters, followed by the UTF-8
    characters allocated with incrementing symbol numbers. */
    /* Preload ASCII characters first */
    for (symbol = 0; symbol < 128; symbol++)
      { /* Add ASCII characters into lookup table
  so that their associated symbol number
matches their ASCII number. */
      if (symbol == 0)
      { /* Special treatment needed for null as null
         signifies end of strong */
        char_bytes [0] = '\\';
        char_bytes [1] = '0';
        char_bytes [2] = '\0';
      }
      else
      {
        char_bytes [0] = symbol;
        char_bytes [1] = '\0';
      }
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */

    /* Then preload the Arabic characters next. */
  for (n = 160; n != 383; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
        new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 697; n != 880; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
        id_count++;
      }
    for (n = 1536; n != 1792; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
        char_bytes [1] = 0x80 + n % 0x40;
        char_bytes [2] = '\0';
        new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
   for (n = 5760; n != 5761; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
```

```
  for (n = 12288; n != 12289; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
       char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
        id_count++;
      }
  for (n = 8192; n != 8336; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */
    id_count++; /* Skip over SYMBOLS_ESCAPE symbol */
    /* Now read in the symbol numbers from STDIN one per line
       and write out their equivalent UTF8 characters. */
    /* repeat until EOF */
    for (;;)
      {
        if (getNumber (stdin, &number) == EOF)
          break;
        if (number != SYMBOLS_ESCAPE)
        {
          char_bytes1 = keys_find_key (&Keys, number);
          printf ("%s", char_bytes1);
        }
      else
        { /* Non ASCII or Arabic two byte sequence */
          if (getNumber (stdin, &byte1) == EOF)
              break;
          if (getNumber (stdin, &byte2) == EOF)
              break;
          char_bytes2 [0] = byte1;
          char_bytes2 [1] = byte2;
          char_bytes2 [2] = '\0';
          printf("%lc", (wchar_t) decoded_byte_value (byte1));
          printf("%lc", (wchar_t) decoded_byte_value (byte2));
          /*printf ("%s", char_bytes2);*/
        }
      }
    return 0;
}
```

B.3 keys.c

```
/* Routines for maintaining keys and their frequencies
in a dictionary. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <ctype.h>
#include "keys.h"
int Keys_malloc = 0; /* size of memory mallocated */
void keys_print_malloc( fp )
/* Dumps the memory allocated */
FILE *fp;
{
    fprintf( fp, "Freq malloc = %d\n", Keys_malloc );
}
char *keys_save_key( key )
char *key;
/* Save the key. */
{
    char *newkey;
    newkey = NULL;
    if (key != NULL) { /* copy the key */
        int slen;
        slen = strlen( key );
        if (slen != 0) {
          Keys_malloc += slen+1;
            newkey = (char *) malloc( slen+1 );
            strcpy( newkey, key );
        }
    }
    return( newkey );
}
void keys_init_keys( keys )
struct keys_type *keys;
/* Initializes the frequencies. */
{
    keys->trie = NIL;
}
struct keys_trie_type *keys_create_trie( node, pos, key, id )
/* Creates a new node (or reuses old node). Insert KEY into it. */
struct keys_trie_type *node;
int pos;
char *key;
int id;
{
    struct keys_trie_type *this;

    if (node != NIL)
        this = node;
    else {
        Keys_malloc += sizeof( struct keys_trie_type );
        this = (struct keys_trie_type *) malloc( sizeof
(struct keys_trie_type ));
    }
    this->cc = key [pos];
    this->key = keys_save_key( key );
    this->next = NIL;
    this->down = NIL;
```

```
    this->id = id;
    return( this );
}

struct keys_trie_type *keys_copy_trie( node )
/* Creates a new node by copying from an old one. */
struct keys_trie_type *node;
{
    struct keys_trie_type *this;

    assert( node != NIL );
    Keys_malloc += sizeof( struct keys_trie_type );
    this = (struct keys_trie_type *) malloc( sizeof
(struct keys_trie_type ));
    this->cc = node->cc;
    this->key = node->key;
    this->next = node->next;
    this->down = node->down;
    this->id = node->id;
    return( this );
}
struct keys_trie_type *keys_find_list( head, cc, found )
/* Find the link that contains the character CC and return
a pointer to it. Assumes the links are in ascending
lexicographical order. If the character is not found,
return a pointer to the previous link in the list. */
struct keys_trie_type *head;
int cc;
int *found;
{
    struct keys_trie_type *this, *that;

    *found = 0;
    if (head == NIL)
        return( NIL );
    this = head;
    that = NIL;
    while ((this != NIL) && (!*found)) {
        if (cc == this->cc)
    *found = 1;
  else if (cc < this->cc)
    break;
        else {
    that = this;
    this = this->next;
}
    }
    if (!*found) /* link already exists */
        return( that );
    else
        return( this );
}
struct keys_trie_type *keys_insert_list
( head, here, pos, key, freq )
/* Insert new link after here and return it.
Maintain the links in ascending
lexicographical order. */
struct keys_trie_type *head;
struct keys_trie_type *here;
```

```
int pos;
char *key;
int freq;
{
    struct keys_trie_type *there, *new;
    assert( head != NIL );
    if (here == NIL) { /* at the head of the list */
        /* maintain head at the same node position
by copying it */
        new = keys_copy_trie( head );
        keys_create_trie( head, pos, key, freq );
      head->next = new;
      return( head );
    }
    new = keys_create_trie( NIL, pos, key, freq );
    there = here->next;
    if (there == NIL) /* at the tail of the list */
      here->next = new;
    else { /* in the middle of the list */
      here->next = new;
      new->next = there;
    }
    return( new );
}
struct keys_trie_type *keys_find_node
(keys, node, pos, key )
/* Returns pointer to node if the key
is found in the trie. */
struct keys_type *keys;
struct keys_trie_type *node;
int pos;
char *key;
{
    int found;
    struct keys_trie_type *here;
    assert( key != NIL );
    if (node == NIL)
      return( NIL );
    here = keys_find_list( node, key[pos], &found );
    if (!found) /* Not in the list -insert the new key */
        return( NIL );
    /* Found in the list - is it the same key? */
    if (here->key != NIL)
  if (!strcmp( key, here->key)) /* key matches */
      return( here );
    if (here->down == NIL) /* move old key one
level down if needed */
        return( NIL );
    if (!key [pos+1]) /* end of the key */
        return( NIL );
    return( keys_find_node( keys, here->down, pos+1, key ));
}
struct keys_trie_type *keys_find_keys( keys, key )
/* Finds the key in the keys trie. */
struct keys_type *keys;
char *key;
{
    return( keys_find_node( keys, keys->trie, 0, key ));
}
```

```
struct keys_trie_type *keys_add_node
(keys, node, pos, key, id )
/* Add the KEY into the NODE of the
trie. If NODE is NIL,then creates and
returns it. Adds the frequency. */
struct keys_type *keys;
struct keys_trie_type *node;
int pos;
char *key;
int id;
{
    int found, cc;
    struct keys_trie_type *here, *pnode;
    assert( key != NIL );
    if (node == NIL) {
    node = keys_create_trie( NIL, pos, key, id );
    return( node );
    }
    here = keys_find_list( node, key[pos], &found );
    if (!found) { /* Not in the list - insert the new key */
      node = keys_insert_list( node, here, pos, key, id );
      return( node );
    }
    /* Found in the list - is it the same key? */
    if (here->key != NIL) {
    if (!strcmp( key, here->key)) { /* key matches - do nothing */
        return( here );
    }
    }
    if (here->down == NIL) { /* move old key one level down if needed */
        cc = here->key[pos+1];
        if (cc) { /* check if not at end of the key */
        node = keys_copy_trie( here );
        node->cc = cc;
        node->next = NIL;
        here->down = node;
        here->key = NIL;
        here->id = id;
      }
    }
    if (!key [pos+1]) { /* end of the key */
      here->key = keys_save_key( key );
      here->id = id;
      return( here );
    }
    pnode = here->down;
    node = keys_add_node( keys, pnode, pos+1, key, id );
    if (!pnode)
        here->down = node;
    return( node );
}
int keys_add_keys( keys, key, id )
/* Adds the key to the trie. Returns true if the key is new. */
struct keys_type *keys;
char *key;
int id;
{
    struct keys_trie_type *node, *pnode;
    pnode = keys->trie;
```

```
    node = keys_add_node( keys, keys->trie, 0, key, id );
    if (pnode == NIL)
        keys->trie = node;
    return( (node != NIL) && (node->id == id) );
}
void keys_dump_node( fp, node, level )
/* Dumps out the keys at the NODE in the trie. */
FILE *fp;
struct keys_trie_type *node;
int level;
{
    while (node != NIL) {
        fprintf( fp, "        %3d [%c] ", level, node->cc );
        if (node->key != NIL) {
          fprintf( fp, "%5d %s", node->id, node->key );
      }
      fprintf( fp, "\n" );
      keys_dump_node( fp, node->down, level+1 );
      node = node->next;
    }
}

void keys_dump_keys( keys, fp )
/* Dumps out the keys in the freq data structure. */
struct keys_type *keys;
FILE *fp;
{
    keys_dump_node( fp, keys->trie, 1 );
}
int keys_find_id (keys, key)
/* Returns the id associated with the key; 0 if it is not in the keys table. */
struct keys_type *keys;
char *key;
{
  struct keys_trie_type *node;
  node = keys_find_keys( keys, key );
  if (node == NULL)
    return (0);
  else
    return (node->id);
}
```

Appendix C

CSA-PPM

Included in this appendix is the code that can be used to perform the preprocessing and postprocessing stages of CSA-PPM. The code comprises the following files: arabic-encode.c, arabic-decode.c and keys.c. These are described in more detail below. arabic-encode.c does the preprocessing, taking a single file as input and produces only one stream of symbol numbers as output. arabic-decode.c does the reverse mapping for the postprocessing stage. keys.c is the same as that provided in Appendix B.

C.1 arabic-encode.c

```
/* Outputs a sequence of symbol numbers that represent the
 Arabic UTF-8 characters found in the input file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <assert.h>
#include "keys.h"
#define UTF8_MAX 4
#define SYMBOLS_ESCAPE 128   /* Used for coping with characters
that are not Arabic or ASCII */
int  UTF8 (FILE *fp, char *char_bytes)
/* Processes the input file FP one UTF8 character at a time. */
{
    int First_byte[] = {192, 224, 240};
    int cc, dd;
    memset(char_bytes, 0, UTF8_MAX + 1);
    char_bytes[0] = getc(fp);
    if (char_bytes [0] == EOF) {
        return 0;
    }
    cc = 0;
    if ((char_bytes [0] & First_byte [0]) == First_byte [0]) cc++;
    if ((char_bytes [0] & First_byte [1]) == First_byte [1]) cc++;
    if ((char_bytes [0] & First_byte [2]) == First_byte [2]) cc++;
    dd = 0;
    while (dd < cc) {
        dd++;
        char_bytes [dd] = getc(fp);
    }
    return cc + 1;
}
```

```
int encoded_byte_value (int ch)
{
  if (ch < 0)
    {
      assert (ch >= -128);
      return (128 - ch);
    }
  else
    {
      assert (ch <= 127);
      return (ch);
    }
}
int main(int argc, char **argv)
{
    char *char_bytes = (char*) calloc(UTF8_MAX + 1, sizeof(char));
    struct keys_type Keys;
    int new, n, symbol, id_count = 0;
    keys_init_keys( &Keys ); /* Initialise the characters
lookup table. Preload the Keys lookup table with ASCII
characters first (symbols 0 to 127, followed by a special
UTF8_ESCAPE character next (symbol number 128)to be used
to cope with unrecognized UTF8 characters, followed by the
Arabic characters allocated with incrementing symbol numbers. */
    /* Preload ASCII characters first */
    for (symbol = 0; symbol < 128; symbol++)
      { /* Add ASCII characters into lookup table so
       that their associated symbol number matches
     their ASCII number. */
      if (symbol == 0)
        { /* Special treatment needed for null as null
           signifies end of strong */
          char_bytes [0] = '\\';
          char_bytes [1] = '0';
          char_bytes [2] = '0';
        }
      else
        {
          char_bytes [0] = symbol;
          char_bytes [1] = '\0';
        }
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */

    /* Then preload the Arabic characters next. */
for (n = 160; n != 383; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 697; n != 880; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
```

```
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    for (n = 1536; n != 1792; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 5760; n != 5761; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 12288; n != 12289; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 8192; n != 8336; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */
    id_count++; /* Skip over SYMBOLS_ESCAPE symbol */
    /* Now read in the UTF8 characters from STDIN one at a time
       and output the symbol numbers associated with each symbol
       number one per line. */
    while (UTF8(stdin, char_bytes))
      {
        symbol = keys_find_id (&Keys, char_bytes);
      if (symbol >= 0)
          printf ("%d\n", symbol);
      else
        { /* Symbol not found - not an ASCII or Arabic character */
          /* new = keys_add_keys (&Keys, char_bytes, id_count++); */
          /* fprintf (stderr, "Error: Unrecognized character <%x><%x>\n",
          char_bytes [0], char_bytes [1]); */
          printf ("%d\n%d\n%d\n", SYMBOLS_ESCAPE, encoded_byte_value
       (char_bytes [0]),
           encoded_byte_value (char_bytes [1]));
        }
      }
    return 0;
}
```

C.2 arabic-decode.c

```
/* Outputs the Arabic UTF-8 characters converted from a
sequence of symbol numbersfound in the input file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <assert.h>
#include "keys.h"
#define UTF8_MAX 4
#define SYMBOLS_ESCAPE 128   /* Used for coping with
characters that are not Arabic or ASCII */
int decoded_byte_value (int ch)
{
  if (ch >= 128)
    return - (ch - 128);
  else
    return (ch);
}
int
getNumber (FILE *fp, unsigned int *number)
/* Returns the next number from input stream fp. */
{
    unsigned int n;
    int result;
    n = 0;
    result = fscanf (fp, "%u", &n);
    switch (result)
      {
        case 1: /* one number read successfully */
        break;
        case EOF: /* eof found */
        break;
        case 0:
        fprintf (stderr, "Formatting error in file\n");
        exit (1);
        break;
        default:
        fprintf (stderr, "Unknown error (%i) reading file\n", result);
        exit (1);
      }
    *number = n;
    return (result);
}
int main(int argc, char **argv)
{
    printf("\xef\xbb\xbf");
    char *char_bytes = (char*) calloc(UTF8_MAX + 1, sizeof(char));
    char *char_bytes1;
    char char_bytes2 [3];
    unsigned int byte1, byte2;
    struct keys_type Keys;
    int new, n, symbol, id_count = 0;
    unsigned int number;

keys_init_keys( &Keys ); /* Initialise the characters
lookup table */
    /* Preload the Keys lookup table with ASCII
characters first (symbols 0 to 127, followed
```

```
by a special UTF8_ESCAPE character next
(symbol number 128) to be used to cope with
unrecognized UTF8 characters, followed by the UTF-8
    characters allocated with incrementing symbol numbers. */
    /* Preload ASCII characters first */
    for (symbol = 0; symbol < 128; symbol++)
      { /* Add ASCII characters into lookup table
  so that their associated symbol number
matches their ASCII number. */
      if (symbol == 0)
        { /* Special treatment needed for null as null
           signifies end of strong */
          char_bytes [0] = '\\';
          char_bytes [1] = '0';
          char_bytes [2] = '\0';
        }
      else
      {
        char_bytes [0] = symbol;
        char_bytes [1] = '\0';
      }
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */

    /* Then preload the Arabic characters next. */
  for (n = 160; n != 383; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 697; n != 880; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    for (n = 1536; n != 1792; ++n)
      {
        char_bytes [0] = 0xC0 + n / 0x40;
      char_bytes [1] = 0x80 + n % 0x40;
      char_bytes [2] = '\0';
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
   for (n = 5760; n != 5761; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
```

```
  for (n = 12288; n != 12289; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
  for (n = 8192; n != 8336; ++n)
      {
        char_bytes [0] = 0xE0 + n/ 0x40/ 0x40 ;
      char_bytes [1] = 0x80 + n/ 0x40 % 0x40;
      char_bytes [2] = 0x80 + n %0x40;
      new = keys_add_keys (&Keys, char_bytes, id_count);
      id_count++;
      }
    /* printf ("id count = %d\n", id_count); */
    id_count++; /* Skip over SYMBOLS_ESCAPE symbol */
    /* Now read in the symbol numbers from STDIN one per line
       and write out their equivalent UTF8 characters. */
    /* repeat until EOF */
    for (;;)
      {
        if (getNumber (stdin, &number) == EOF)
          break;
      if (number != SYMBOLS_ESCAPE)
        {
          char_bytes1 = keys_find_key (&Keys, number);
          printf ("%s", char_bytes1);
        }
      else
        { /* Non ASCII or Arabic two byte sequence */
          if (getNumber (stdin, &byte1) == EOF)
              break;
          if (getNumber (stdin, &byte2) == EOF)
              break;
          char_bytes2 [0] = byte1;
          char_bytes2 [1] = byte2;
          char_bytes2 [2] = '\0';
          printf("%lc", (wchar_t) decoded_byte_value (byte1));
          printf("%lc", (wchar_t) decoded_byte_value (byte2));
          /*printf ("%s", char_bytes2);*/
      }
      }
    return 0;
}
```

Appendix D

Lossy non-dotted correction

Included in this appendix is the code that can be used to perform the correction process for non-dotted text (lossy version) using Vetirbi-based correction in order to recover the original text (lossless version) when a match occurs between the text and confusion list defined. The code first requires the installation of the Text Mining Toolkit (Teahan and Harper, 2001).

D.1 lossy.c

```
/* Correct non-dotted text given a trained model . */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#ifdef SYSTEM_LINUX
#include <getopt.h> /* for getopt on Linux systems */
#endif
#include "io.h"
#include "text.h"
#include "paths.h"
#include "model.h"
#include "markup.h"
#define MAX_FILENAME_SIZE 128
unsigned int Input_file;
char Input_filename [MAX_FILENAME_SIZE];
unsigned int Output_file;
char Output_filename [MAX_FILENAME_SIZE];
boolean Use_Numbers = FALSE;
unsigned int Language_model;
boolean Segment_alphanumeric = FALSE;
boolean Segment_before = FALSE;
boolean Segment_Viterbi = FALSE;
unsigned int Segment_stack_depth = 0;
void
debug_play ()
/* Dummy routine for debugging purposes. */
{
    fprintf (stderr, "Got here\n");
}
void
usage (void)
{
```

```
    fprintf (stderr,
   "Usage: play [options] training-model <input-text\n"
   "\n"
   "options:\n"
   "  -A\tsegment alphanumeric characters only\n"
   "  -B\tsegment before each character (rather than after)\n"
   "  -C \tdebug coding ranges\n"
   "  -d n\tdebug paths=n\n"
   "  -D n\tstack algorithm only: stack depth=n\n"
   "  -i fn\tinput filename=fn (required argument)\n"
   "  -l n\tdebug level=n\n"
   "  -m fn\tmodel filename=fn\n"
   "  -N\ttext stream is a sequence of unsigned numbers\n"
   "  -o fn\toutput filename=fn (required argument)\n"
   "  -p n\tdebug progress=n\n"
   "  -V\tsegment using Viterbi algorithm\n"
);
    exit (2);
}
void
init_arguments (int argc, char *argv[])
{
    extern char *optarg;
    extern int optind;
    int opt;
    boolean Input_found = FALSE, Output_found = FALSE;
    /* set defaults */
    Debug.level = 0;
    Debug.level1 = 0;
    /* get the argument options */
    while ((opt = getopt (argc, argv,
       "ABCd:D:i:l:m:No:p:V")) != -1)
    switch (opt)
    {
      case 'A':
          Segment_alphanumeric = TRUE;
          break;
      case 'B':
          Segment_before = TRUE;
          break;
      case 'C':
          Debug.coder = TRUE;
          break;
      case 'd':
          Debug.level1 = atoi (optarg);
          break;
      case 'D':
          Segment_stack_depth = atoi (optarg);
          break;
      case 'i':
          Input_found = TRUE;
          sprintf (Input_filename, "%s", optarg);
          break;
      case 'l':
          Debug.level = atoi (optarg);
          break;
      case 'm':
          Language_model =
              TLM_read_model (optarg, "Loading model from file",
```

```
    "Segment: can't open model file");
          /*TLM_dump_model (Stderr_File, Language_model, NULL);*/
          break;
      case 'N':
          Use_Numbers = TRUE;
          break;
      case 'o':
          Output_found = TRUE;
          sprintf (Output_filename, "%s", optarg);
          break;
      case 'p':
          Debug.progress = atoi (optarg);
          break;
      case 'V':
          Segment_Viterbi = TRUE;
          break;
      default:
          usage ();
          break;
      }
    if (!Input_found)
        fprintf (stderr, "\nFatal error:
missing input filename\n\n");
    if (!Output_found)
        fprintf (stderr, "\nFatal error:
missing output filename\n\n");
    if (!Input_found || !Output_found)
      {
        usage ();
      exit (1);
      }
    for (; optind < argc; optind++)
    usage ();
}
int
getSymbol (unsigned int file,
unsigned int *symbol)
/* Returns the next symbol from
the input file. */
{
    unsigned int sym;
    int result;
    sym = 0;
    if (Use_Numbers)
      {
        result = fscanf (Files [file], "%u", &sym);
      switch (result)
{
        case 1: /* one number read successfully */
          break;
        case EOF: /* eof found */
          break;
        case 0:
          fprintf (stderr, "Formatting error in file\n");
          break;
        default:
          fprintf (stderr,
    "Unknown error (%i) reading file\n", result);
          exit (1);
```

```
      }
    else
      {
        sym = getc (Files [file]);
      result = sym;
      }
    *symbol = sym;
    return (result);
}
void
dump_markup_symbol (unsigned int file,
unsigned int symbol)
/* Writes the ASCII symbol out in human
readable form (excluding white space). */
{
    char line [20];
    assert (TXT_valid_file (file));
    if (Use_Numbers)
       sprintf (line, "%d\n", symbol);
    else
       sprintf (line, "%c", symbol);
    TXT_write_file (file, line);
}
int
main (int argc, char *argv[])
{
    unsigned int input;   /* The input text to correct */
    unsigned int markup_text;
    unsigned int markup_model;

    init_arguments (argc, argv);

    Input_file = TXT_open_file
      (Input_filename, "r", "Encoding input file",
       "Encode_ppmo: can't open input file" );

    Output_file = TXT_open_file
      (Output_filename, "w", "Writing to output file",
       "Encode_ppmo: can't open output file" );

    if (TLM_numberof_models () < 1)
        usage();

    if (Segment_Viterbi)
        markup_model = TMM_create_markup
(TMM_Viterbi);
    else
        markup_model = TMM_create_markup (TMM_Stack,
TMM_Stack_type0, Segment_stack_depth, 0);
    if (Use_Numbers)
        input = TXT_load_numbers (Input_file);
    else
        input = TXT_load_text (Input_file);

    TMM_start_markup (markup_model,
TMM_markup_multi_context, input,
      Language_model);

    markup_text = TMM_perform_markup
```

```
(markup_model, input);
    /* Ignore the sentinel and model
symbols that are inserted at the
head of the marked up text. */
    TXT_dump_text1 (Output_file, markup_text, 2,
dump_markup_symbol);

    TXT_release_text (markup_text);
    exit (0);
}
```