MACHINE LEARNING FOR ANCIENT GREEK LINGUISTICS

A TENTATIVE METHODOLOGY AND APPLICATION TO THE CORPUS OF DOCUMENTARY PAPYRI

Master's thesis submitted in partial fulfilment of the requirements for the degree of

MASTER OF ARTS IN DE TAAL- EN LETTERKUNDE

Submitted by: Xavier Goas Aguililla
Supervisor: dr. T. Van Hal
KU Leuven
Faculty of Arts
Department of Greek Studies

Leuven, 2013

Typeset with pdfTeX 3.1415926-2.4-1.40.13 using a modified version of Lorenzo Pantieri's ArsClassica package.

E-MAIL:
xavier.goas@student.kuleuven.be

SAMENVATTING

Ondanks de bewezen diensten van corpusgebaseerde methoden in het taalkundig onderzoek is er een groot gebrek aan geannoteerde digitale corpora voor het Oudgrieks. Grote hoeveelheden tekst zijn ondertussen digitaal beschikbaar, maar door gebrek aan mankracht is het niet mogelijk deze handmatig te analyseren. Een betrouwbare methode om dit probleem automatisch aan te pakken, al zou deze niet perfect zijn, zou een stap in de goede richting zijn. Een belangrijk pijnpunt is dat dit gebrek aan geannoteerde corpora het ook zeer moeilijk maakt om dit soort methode te ontwikkelen.

In dit werk trachten we dit pijnpunt te omzeilen door een dergelijke methode te ontwikkelen met behulp van spitstechnieken uit de artificiële intelligentie en bij wijze van *case study* toe te passen op een corpus gedigitaliseerde documentaire papyri. Deze methode berust op een implementatie van een zgn. artificieel neuraal netwerk, d.i. een wiskundig model dat in staat is om patronen te herkennen en te leren. We passen de methode beschreven in Collobert, Weston e.a., 2011 toe op het Oudgrieks.

Het leerproces van dit netwerk wordt opgedeeld in twee fasen. In een eerste wordt gebruik gemaakt van een groot ongeannoteerd corpus om via een eenvoudig criterium een wiskundig model op te bouwen dat woorden kan kaderen binnen het taalgebruik in het geheel. Dit model kent kansen toe aan reeksen woorden al naargelang die al dan niet 'correct' zijn door te schatten of gelijkaardige reeksen kunnen voorkomen in het corpus.

Een tweede fase schakelt over op een kleiner, geannoteerd corpus. Na het definiëren van een wiskundige voorstelling voor morfologische en syntactische categorieën worden meerdere netwerken geïnitialiseerd die gebruik maken van de voorheen opgebouwde wiskundige representatie. De modellen worden afgestemd op de woordannotatieparen en beïnvloeden elkaar onderling om zo goed mogelijk gebruik te maken van patronen die van potentieel nut zijn voor de taak van elk netwerk.

We waren van plan om het zo bekomen model toe te passen op een digitaal beschikbaar corpus papyri, en dit na afloop in een verdeelbaar formaat om te zetten en te opensourcen voor verdere verwerking. De finale resultaten waren echter teleurstellend: het systeem is te eenvoudig om correcte morfologische en syntactische inferenties te maken en lijdt nog steeds onder de kleine hoeveelheden geannoteerde tekst die beschikbaar zijn voor Oudgrieks. De evolutie van het model dat ontwikkeld werd in de eerste fase van het leerproces is echter positief en stemt hoopvol. Met langere rekentijd kan dit type model zeker van nut zijn in toekomstig onderzoek. We stellen ook een aantal pistes voor om ook de tweede fase van het leerproces te verbeteren.

Deze masterproef bevat **77069** tekens.

PREFACE

Originally, this work was to be a treatise on select topics in the linguistic study of the papyri; the development of natural language processing tools for ancient Greek was only an accessory to the main goal; but as I embarked upon studies in computer science, I accordingly decided to apply newfound knowledge to the problem. The tools became the main goal, and thus this work stands as it is.

I chose a recent approach to natural language processing which is considerably complex in its implementation and set high hopes, which have, in a sensen, been dashed ever so slightly. It was hard work, but rewarding in its own right, and even though the experiment has not been entirely successful, I hope to have demonstrated that there are possibilities for further research in this grey zone between the classics and computer science, which are sorely hampered by the lack of resources, knowhow and manpower. The best I can do is lead by example, even though I am but one man stretching to have one foot in each camp, a decidedly unstable position if there ever was one.

I leave it up to others to decide whether my chosen pursuit is worthwhile: as for myself, I know it was, and still will be. *Perferam et obdurabo.*

ACKNOWLEDGEMENTS

I owe profound thanks to several people for the following work.

Firstly, I thank my supervisor, dr. Toon Van Hal, who from the get-go demonstrated a very open-minded attitude in the face of unorthodox subject matter for a classical philology thesis. He has demonstrated exemplary patience with the stop-start rhythm of development of this work. His corrections, suggestions and occasional nudges (usually delivered electronically and with utmost tact) were an invaluable help and encouragement. It is evident that without him, this work would not have been possible.

Next, I am grateful to dr. Francis Maes, until recently a postgraduate student at the Department of Computer Science at the KU Leuven Faculty of Science, who during a conversation on this thesis directed me to a set of state-of-the-art papers on machine learning for natural language processing. His suggestions form the methodological bedrock of this work in its current form.

I would also like to extend my thanks to my godmother María Aguililla, who immediately accepted to proofread this work and did so diligently, as well as Erwin De Koster for temporarily providing me with computational power in the form of an iMac.

I offer my gratitude to my high school classics teacher, Guy van den Heule, whose passion for Greco-Roman language and culture combined with a lively and engaging teaching style pushed me to pursue the study of the classical languages at a higher level.

I would also like to thank all my friends who offered me support and lended me their proverbial ear in some way when the going got rough or showed more than a cursory interest in this work, most of whom definitely know who they are; they are too many to mention and I fear forgetting but one.

I have not mentioned the most important persons of all: my parents, who have been unflagging in their encouragement, interest and support. They have enabled me to continue work on this thesis and to pursue my study of computer science, which has granted me the technical knowledge necessary to complete this work. To them I owe the greatest thanks of all.

Finally, I invoke the memory of my good friend Lorena Lage Piñeiro, who would have been mentioned in the previous paragraphs, if not for the intervention of fate. Her sudden passing a week before this writing deeply saddened me, and it is to her that I dedicate this work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

1 THESIS

1.1 STATEMENT

The following work deals with the application of techniques from the field of machine learning and natural language processing to ancient Greek. We intend to show that it is possible to use these techniques to generate linguistic annotation in an accurate and efficient manner.

In order to achieve this, we have implemented a state-of-the-art machine learning architecture for natural language processing, proposed in Collobert and Weston, 2008; Collobert, Weston *et al.*, 2011. The chosen approach makes use of raw Greek text to counterbalance the relative scarcity of annotated ancient Greek corpora, which are essential in most natural language processing systems.

We evaluate our method by measuring the absolute performance of the model against a validation corpus and executing a case study on a freely available digital corpus of documentary papyri.

1.2 MOTIVATION

We wish to demonstrate the potential of a methodology based on computational techniques for the study of the classical languages. The field has seen a move towards digitalisation in the past half century, but a lot of potential is left untapped. Steps in the right direction are currently being made, but we can drive progress much further. The classicist breed does not number many specimens; and though the true classics, the Homers, the Platos, the Virgils, have been subjected to thorough analysis for millennia, the amount of texts in need of scholarly attention remains large. Demonstrating the potential of computational methods for Greek linguistics will hopefully serve as further proof of their potential for other branches of classics, such as stylometry, authorship verification, textual criticism, and more.

Specifically, we want to apply such techniques to the problem of linguistic annotation in order to open new perspectives on the structure and evolution of the Greek language. Massive amounts of source material are now digitally available: resources such as the *Thesaurus Linguae Graecae* and the Perseus project provide us with dozens of millions of words. What has been lacking is the systematic development of large annotated corpora. While systems have been put in place which offer morphological and lemmatic analysis in a rudimentary form, this cannot serve as a full linguistic corpus; of these, only two of note exist, which together number about half a million words and thus are relatively small compared to the vast textual repository of the TLG. We need to expand this type of resource to many more texts. The problem is that using manual methods to tag the large Greek corpora is rather prohibitive due to their size. By making use of computational methods, we can make an attempt at offering a relatively complete linguistic corpus for ancient Greek.

The chosen case study, the annotation of a corpus of documentary papyri, is meant to evoke the possibilities afforded by these methods by directly applying them to a corpus which has not been subjected to a modern linguistic study. The question of the language of the papyri has in the past thirty years seen little evolution until the recent appearance of Evans and Obbink, 2010, which has placed the subject in the spotlight again. Twentieth-century scholarship on the topic, though still useful for those interested in the study of the papyri for historical purposes, is either antiquated, limited in scope or incomplete; for more on this, see section 2.1.1 on page 7.

Despite this, the papyri are useful source material for the history and evolution of the Greek language: the corpus consists of more than 4.5 million words, spanning more than a thousand years and many different discourse registers. Annotating this corpus would be a boon to scholars interested in the Greek of the papyri and Greek historical linguistics in general, as it would facilitate the creation of linguistically sound grammars and lexica.

1.3 OUTLINE

We begin by giving an overview of the background for this thesis. First the historical and linguistic background of the question is handled in chapter 2 on page 7. We give a historical overview of previous efforts to study the grammar of the papyri, as well as of the applications of the techniques of corpus linguistics to the Greek language in general.

Secondly, in section 2.2 on page 13, we illustrate critical concepts and techniques for the task at hand which are necessary to understand the underpinnings of the applied methodology. Formalism is kept to a minimum; for that, we refer to appendix **??** on page ??.

We proceed with an overview of directly related work in section 2.3 on page 14. A problem statement and proposed solution is offered in chapter 3 on page 19 with specific reference to the scholarly context; jointly, we extend a brief overview of the possibilities the proposed method offers.

The theory behind the natural language processing architecture, as well as a method to represent Greek morphological annotations mathematically are illustrated in chapter 4 on page 29. The implementation is detailed in chapter 5 on page 43: we provide details on the choice of programming language, the availability of the source code, the technical requirements for running our code, etc. The source code is found in appendix A on page A-1.

We detail our results in chapter 6 on page 55 with a critical assessment in chapter 7 on page 63, where we state our contribution. Section 7.3 on page 64 of this chapter is specifically dedicated to an overview of future avenues for research on this type of method. Finally, we summarize our findings in chapter 7.2 on page 64.

2 BACKGROUND

2.1 HISTORICAL BACKGROUND

2.1.1 The language of the papyri

The papyri began to be studied linguistically not by papyrologists and historians, but rather by Biblical scholars and grammarians interested in their relevance in the development of koinê Greek, particularly that of the New Testament. G. N. Hatzidakis, W. Crönert, K. Dieterich, A. Deissmann, and A. Thumb pioneered the field in the late nineteenth and early twentieth century, spurring a resurgence of scholarship on the topic;[1] an excellent overview of pre-1970s research may be found in Mandilaras, 1973 and Gignac, 1976, 1981.

During this period, E. Mayser began work on the earliest compendious grammar of the papyri; it limits itself to the Ptolemaic era but explores it at length and in great detail. The work [Mayser, 1938] consists of a part on phonology and morphology, made up of three slimmer volumes, and a part on syntax, encompassing three larger volumes. Its composition seems to have been exhausting: it took Mayser thirty-six years to finish volumes I.2 through II.3, with I.1 only completed in 1970 by Hans Schmoll, at which point the entire series was given a second edition.

When casually browsing through some of its chapters (though casual is hardly the word one would associate with the *Grammatik*) it is remarkable to see that Mayser brings an abundance of material to the table for each grammatical observation he makes, however small it may be. For instance, the section on diminutives essentially consists of pages upon pages of examples categorised by their endings.

This is its great strength as a reference work - whenever one is faced with an unusual grammatical phenomenon in any papyrus, consulting Mayser is bound to clarify the matter; or rather, it was, for the work is now inevitably dated. The volumes published during Mayser's lifetime only include papyri up to their date of publication; only the first tome by Schmoll includes papyri up to 1968. It is still a largely useful resource, but it is in urgent need of refreshment.

1 Vide Crönert, 1903; Deissmann, 1895, 1897, 1929; Dieterich, 1898; Thumb, 1901, 1906.

After Mayser set the standard for the Ptolemaic papyri, a grammar of the post-Ptolemaic papyri was the new *desideratum* in papyrology. The work had been embarked on by Salonius, Ljungvik, Kapsomenos, and Palmer, only to be interrupted or thwarted by circumstance or lack of resources. Salonius, 1927, for instance, only managed to write an introduction on the sources, though he offered valuable comments on the matter of deciding how close to spoken language a piece of writing is. Ljungvik, 1932 contains select studies on some points of syntax.

It is in the 1930's that we see attempts to create a grammar of the papyri that would be the equivalent of Mayser for the post-Ptolemaic period. S. Kapsomenos published a series of critical notes [Kapsomenos, 1938, 1957] on the subject; though he attempted at a work on the scale of the *Grammatik*, he found the resources sorely lacking, as the existing editions of papyrus texts could not form the basis for a systematic grammatical study. The other was L. Palmer, who had embarked on a similar project and had already set out a methodology [Palmer, 1934]; the war interrupted his efforts, and he published what he had already completed, a treatise on the suffixes in word formation [Palmer, 1945].

A new work of some magnitude presents itself two decades later with B. G. Mandilaras' *The verb in the Greek non-literary papyri* [Mandilaras, 1973]. Though it does not aim to be a grammar of the papyri, it does offer a thorough and satisfactory treatment of the verbal system as manifest in the papyri. Further efforts essentially do not appear until the publication of Gignac's grammar. It is essentially treading in the footsteps of Mayser, only with further methodological refinement and a more limited, though still sufficiently exhaustive, array of examples. The author, for reasons unknown to me, only managed to complete two of the three projected volumes, on phonology and on morphology. The volume on syntax is thus absent, a gap only partly filled by Mandilaras' *The verb in the Greek non-literary papyri*.

Finally, there is the aforementioned *The Language of the Papyri* [Evans and Obbink, 2010], which does not aim to be a work on the same scale as previous works in the field. It is a collection of articles on various topics, the whole of which is meant to illuminate new avenues for future research. A particularly relevant chapter for this thesis is the last one by Porter and O'Donnell [Porter and O'Donnell, 2010], who set out to create a linguistic corpus for a selection of papyri; their tagging approach, however, is manual, and their target corpus limited. The authors also are the creators of http://www.opentext.org/, a project aiming for the development of annotated Greek corpora and tools to analyse them; sadly, no progress seems to have been made since 2005.

2.1.2 Corpus linguistics[2]

A corpus or text corpus is a large, structured collection of texts designed for the statistical testing of linguistic hypotheses. The core methodological concepts of this mode of analysis may be found in the concordance, a tool first created by biblical scholars in the Middle Ages as an aid in exegesis. Among literary scholars, the concordance also enjoyed use, although to a lesser degree; the eighteenth century saw the creation of a concordance to Shakespeare.

The development of the concordance into the modern corpus was not primarily driven by the methods of biblical and literary scholars; rather, lexicography and pre-Chomskyan structural linguistics played a crucial role.

Samuel Johnson created his famous comprehensive dictionary of English by means of a manually composed corpus consisting of countless slips of paper detailing contemporary usage. A similar method was used in the 1880s for the Oxford English Dictionary project - a staggering three million slips formed the basis from which the dictionary was compiled.

1950s American structuralist linguistics was the other prong of progress; its heralding of linguistic data as a central given in the study of language supported by the ancient method of searching and indexing ensures its proponents may be called the forerunners of corpus linguistics.

Computer-generated concordances make their appearance in the late 1950s, initially relying on the clunky tools of the day - punch cards. A notable example is the Index Thomisticus, a concordance to the works of Thomas of Aquino created by the late Roberto Busa S.J. which only saw completion after thirty years of hard work; the printed version spans 56 volumes and is a testament to the diligence and industry of its author. The 1970s brought strides forward in technology, with the creation of computerised systems to replace catalogue indexing cards, a change that greatly benefited bibliography and archivistics.

It is only in the 1980s and 1990s that are marked the arrival of fully developed corpora in the modern sense of the word; for though the basic concepts of corpus linguistics were already widely used, they could not be applied on a large scale without the adequate tools. The rise of

2 The following section is based *passim* on McCarthy and O'Keeffe [2010].

the desktop computer and the Internet as well as the seemingly ever-rising pace of technological development ensured the accessibility of digital tools. The old tools - punch cards, mainframes, tape recorders and the like - were gladly cast aside in favour of the new data carriers.

The perpetual increase of computing power equally demonstrated the limits of large-scale corpora; while lexicographical projects that had as their purpose to document the greatest number of possible usages could keep increasing the size of their corpora, the size of others went down as they whittled the data down to a specific set of uses of language.

The possible applications of the techniques of corpus linguistics are diverse and numerous; for they allow for a radical enlargement in scope while remaining empirical, and remove arduous manual labour from the equation. Corpus linguistics can be an end to itself; it can, however, assert an important role in broader research. McCarthy and O'Keeffe, 2010, p. 7 mention areas such as language teaching and learning, discourse analysis, literary stylistics, forensic linguistics, pragmatics, speech technology, sociolinguistics and health communication, among others.

The term 'corpus' has a slightly different usage in classical philology: it designates a structured collection of texts, which is not forcibly intended for the testing of linguistic hypotheses. Instead, we have, for instance, the ancient corpus Tibullianum, or modern-day collection, for instance the Corpus Papyrorum Judaicarum, etc. We are primarily interested in the digital techniques used to create linguistic corpora; so let us first take a look at the progress of the digital classics.

2.1.3 The digital classics

Classical philology, despite its status as one of the oldest and most conservative scientific disciplines still in existence today, has in the last fifty years found itself at the front lines of the digital humanities movement. Incipient efforts in the fifties and sixties, mainly stylometric and lexical studies and the development of concordances, demonstrated the relevance of informatics in the classics, an evolution that was at first met with some skepticism, but later fully embraced.

The efforts began with the aforementioned *Index Thomisticus*, the first computer-based corpus in a classical language; but the first true impetus was the foundation of the *Thesaurus Linguae Graecae* project in 1972, a monumental project with as its goal the stocking of all Greek texts from the Homeric epics to the fall of Constantinople. Over the years, many functions have been added to this ever more powerful tool; and even in the beginning stages of its development, the TLG garnered praise.

The usefulness of the tool in its current form cannot be overstated: not only does it contain a well-formatted and easily accessible gigantic collection of text editions whose scope and dimensions exceed those of nearly any university library; it also offers all of these texts in a format that allows for lexical, morphological and proximity searches, as well as including a full version of the Liddell & Scott and Lewis & Short dictionaries. The TLG has become a staple of the digital classics.

Despite this, the TLG is becoming more and more dated as technology progresses. While recent years have seen the rise of Unicode as the standard for encoding ancient Greek, the TLG still uses beta code, a transliteration system designed to only use the ASCII character set, and the texts are stored using an obsolete text-streaming format from 1974, which divides the text in blocks of eight kilobytes and marks the division between segments.

A digitised version of the Liddell-Scott-Jones lexicon has been added to the TLG's web interface, but the texts themselves have not undergone extensive tagging, only lemmatisation. Searching through the database can be done by searching for specific forms of a lemma, or by searching for all forms of a lemma, but this is essentially the limit of the search tool's power; it is not possible to perform a query for all possible lemmata associated with a particular form, i.e. we cannot find all forms which are, for example, an active perfect indicative.

In the wake of the TLG, several notable projects have emerged: Brepols' Library of Latin Texts is trying hard to be for Latin texts what the TLG is for Greek texts; the Packard Humanities Institute has released CD's containing a selection of classical Latin works. In more recent times, the Perseus Project has enjoyed great popularity because of the attractive combination of an excellent selection of classical texts with translations, good accessibility and a set of interesting textual tools, the entire package carrying a very interesting price tag for the average user — it is free to use, and for the greatest part, open source as well.

The databases we have mentioned are quite general in scope; but within the domain of classical philology, other specialised projects exist. Within the field of papyrology the digital revolution has taken a firm foothold. Starting with several separate databases, the field has experienced a tendency towards convergence and integration of the available resources, as exemplarised by the `papyri.info` website, maintained by Columbia University, that integrates the main papyrological databases into a single database.

A great feature of this database is the shell in which all data is wrapped; they are compliant with the EpiDoc standard, a subset of XML based on the TEI standard and developed specifically for epigraphical and papyrological texts. One may access the database's resources through the Papyrological Navigator and suggest corrections and readings through the Papyrological Editor. What's more, all data is freely accessible under the Creative Commons License, crowd-sourced, regularly updated, and can be downloaded for easier searching and tweaking.

In other words, `papyri.info` has brought the open-source mentality from the computer world into the classics. For our purposes, this open setup is desirable, as the database is not fit for them as it is, but can with some effort be molded into a useful tool.

2.1.4 Natural language processing

Natural language processing (henceforth NLP) is a subdiscipline in computer science concerned with the interaction between natural human language and computers. Its history well and truly starts in the fifties, with a basic concept which has played a great role in natural language processing, and computer science in general, the Turing test. This test, put forth by Alan Turing in his seminal paper *Computing Machinery and Intelligence* [Turing, 1950], evaluates whether a machine is intelligent or not by placing a human in conversation with another human and a machine; if the first human cannot tell the other human and the machine apart, the machine passes the test.

Machine translation systems entered development, though progress soon stalled because of technical limitations and because of methodological obstacles: such systems were dependent on complex rulesets written by programmers that allowed for very little flexibility. Because of the slow return on investments made, funding for artificial intelligence in general and machine translation specifically was drastically reduced throughout the late sixties and the seventies.

A resurgence followed: in the eighties, advances in computational power permitted new statistical approaches which gradually displaced the rule-based systems used till then. The concept of generative linguistics as espoused by Chomsky, while still possessing as firm a foothold as ever in traditional linguistics departments, was pruned in favour of data-driven methodology.[3]

Modern natural language processing is situated on the crossroads between various fields: artificial intelligence, computer science, statistics, and corpus and computational linguistics. It looks to be an exciting field for the coming years as its techniques are under constant improvement and ever more present in our daily lives.

Most NLP software is designed explicitly with living languages in mind; English, being a world language and the international *lingua franca*, has enjoyed most of the attention, but other major languages have enjoyed some attention, too. Ancient languages, however, are neglected, presumably due to their often high complexity and the extensive study and analysis to which they have been submitted by skilled scholars. Yet most texts have not been integrated in annotated corpora; and though databases such as the Perseus project contain large swathes of morphologically and sometimes syntactically annotated text, the process has been driven largely by manual labour; to give an exhaustive list is not appropriate here, but another such example which is relevant is the PROIEL project [*PROIEL: Pragmatic Resources in Old Indo-European Languages*], which is also a treebank, i.e. a database of syntactically annotated sentences. It contains data for Herodotus and the New Testament.

2.2 CONCEPTS AND TECHNIQUES

While there is, of course, no room in this thesis for an extended course in mathematics or computer science, it is necessary to have some background in order to understand the techniques used for the design and implementation of the architecture. While most of this background is basic first-year university mathematics, the average classicist will not be fully grounded in it. That does not make this chapter the place for it.

We therefore refer the curious reader to the following works which are compendious and nature and may serve as a glossary:

- Russell and P. Norvig, 2010 (especially recommended);
- Wasserman, 2003;

3 This is an interesting controversy with the two camps being represented by Peter Norvig, a prominent AI and machine learning researcher, and Noam Chomsky himself, respectively. Though this is not the place to treat it extensively, we refer to Katz, 2012, an interview on artificial intelligence with Noam Chomsky, and P. Norvig, 2011, in which Peter Norvig provides an extensive rebuttal.

- Haykin, 1999.

2.3 RELATED WORK

Computational approaches to classical philology have been the object of increasing interest for the last few years. While none have chosen to focus on the language of the Greek papyri specifically, related areas have received attention and are relevant to the task at hand. Annotated corpora have been created, efforts to automatically tag Greek have been made, and some have even taken a stab at using natural language processing techniques for textual criticism.

2.3.1 Morphological analysis

Packard, 1973 was the first attempt to create a system for the automated morphological analysis of Greek, which he dubbed Morph. The author did not aim at creating a theoretically well-founded tool; instead, his aim was to assist him in the creation of a textbook and curriculum for American university students that would enable them to start reading the classics in the original language earlier by providing automatically generated analyses and fitting the curriculum to the most frequent forms found in the works selected for reading. Despite this, it is clear that the author realises the potential of the concept.

Most of the paper is dedicated to giving examples of how the program would analyse a given word. The method for generating parses is rule-based. The system is equipped with a set of morphological roots. The algorithm, when given a word, removes suffixes until it finds a match in this set of roots. If no matching roots are found, the same procedure is repeated with the algorithm now stripping prefixes from each word. The system proposed possesses several weaknesses: accents are completely ignored (most likely due to relatively limited technical resources at the time [4]), and the rule-based system relies heavily on the author's knowledge of Greek.

A critical assessment of the methodology and results of this paper is not possible: no concrete measure of the system's accuracy is given. The source code is also nowhere to be found; even if it was freely available, testing it would be a complicated affair, as it is written in assembly language specific to the IBM 360 mainframe and not compat-

4 A rather quaint detail is that computing time had to be rented; the author mentions having to use one dollar's worth of computing time to analyse Plato's *Apology of Socrates*

ible with modern computer architectures. We mention it for historical reasons, as Packard's method is central in the development of later morphological analysis tools.

Notably, Gregory Crane's *Morpheus* was developed in C and Lisp on top of Packard's implementation by augmenting the analyser with a generative component. Work on the system, that would later become the backend for the Perseus morphological analyser, began in 1984 and was aided by two graduate students at Berkeley, Neel Smith and Joshua Kosman.

The generative component of the system creates an extensive table of possible word forms using stems and suffixes such as provided by Morph. The exact system is detailed in Crane, 1991.

Morphological parsing is essentially done by looking up forms in this word table. This comes at the cost of a lot of space; on the other hand, parsing a word which is in the database is a very fast operation, since the word itself doesn't have to be manipulated, and the tables have been maintained and improved for years using user input.

An important issue is that of parse ambiguity. The morphological complexity of Greek ensures that over 50% of words present in the parse table have multiple parses [Dik and Whaling, 2008]. Disambiguation is therefore key for the next development stage of morphological analysis tools for ancient Greek.

Despite this, Morpheus is without question the best tool for Greek morphological analysis available. For years, it has been a core component of the success of the Perseus project. Recently, there has been a drive to create an open source version of the original program,[5] which is now available on GitHub at `https://github.com/PerseusDL/morpheus`.

Lee, 2008 offers a new approach to the analysis of Greek morphology by using machine learning methods melded with the approach used by Morpheus. The method proposed relies on large amounts of data and makes use of nearest-neighbour analysis techniques. This stands in contrast to the traditional rule-driven approach used by earlier parsers such as Packard, 1973 or Crane, 1991.

Affix transformations similar to the ones described in Packard, 1973 are applied to word forms in order to analyse their relation to other forms. A nearest-neighbour metric is established on the basis of the number of these transformations needed to generate another extant word form. As training data, an annotated corpus is fed architecture,

5 *Pace* Blackwell and Crane, 2009.

supported by a large amount of unlabelled data to facilitate the prediction of verbal stems.

An accuracy of 98.2% is achieved for words present in the training set, with the remaining forms necessitating contextual disambiguation. For words not present in the training set, an average accuracy of 85.7% is achieved with most of the loss in accuracy with most errors due to what the author terms 'novel roots', which are stems which are not directly derivable from a word form and are analysed with a practical 50% accuracy (though this was improved to 65% by loosening the standards by which accuracy was measured).

The use of machine learning techniques in this paper is laudable and certainly not badly executed, but the general methodology and in particular the choice of training corpora is problematic. An annotated version of the first five books Septuagint is used to establish these metrics. This corpus consists of 470K words and can be reduced to a set of about 37K unique words. It is unnecessary to restrict training to a corpus of sentences if one does not look at n-grams, but only at stand-alone word forms.

The question, then, is: why not use the output of Morpheus, the system designed by Gregory Crane and honed over the years, as training material? It is freely available as an SQL database and contains ~1M unique parses which are generated using very large corpora for verification. The nearest-neighbor metric may be applied to these word forms and arguably form a better picture, as well as allowing for better extrapolation to data not present in the training corpus.

A last notable result [Dik and Whaling, 2008, 2009] in morphological annotation we will only mention in passing in this section; it is treated in more detail *infra* in 2.3.3, as it is relevant not only to the problem of morphological analysis, but also to that of corpus annotation, our second major goal.

2.3.2 Syntactic parsing

Efforts to develop a model for syntactic parsing of ancient Greek are in an embryonic stage. Mambrini and Passarotti, 2012 contains a few experiments with syntactically parsing ancient Greek. Training data was taken exclusively from the Perseus ancient Greek dependency treebank and partitioned into different sets to observe the influence of differences in author and genre on the results. These were split up into two datasets for training and validation, respectively. They were then used to train MaltParser [Nivre *et al.*, 2006]. Initial performance on the Homeric texts is disappointing due to the scarcity of resources for training the parser. 44.1% tokens were given a fully correct labeling, including relations and their head word, 60.3% were assigned a head word correctly, and 49.2% were assigned a label correctly.

Adjusting the hyperparameters and features of the model manually yielded a considerable boost in performance, with accuracy rates increasing to 71.72%, 78.26% and 81.62% respective to the aforementioned accuracy criteria on the Homeric texts. The improved model is then tested on Hesiod, Sophocles and Plato. Unsurprisingly, performance on the Hesiodic poems is far better than on the Sophocles and Plato, which demonstrates that the barrier between textual genres is a serious obstacle for this type of parsing. The authors are currently working on expanding their training set and testing different systems.

Another interesting approach to the problem of parsing ancient Greek is found in Lee, 2010. The author develops and trains a parsing system for the Septuagint which relies on two resources: a treebank of the New Testament, made available by PROIEL, and a parallel text of the Septuagint in the original Hebrew. The original is annotated with cantillation marks, which serve as prosodic markers to be observed during public chanting of the religious texts. These marks at times facilitate disambiguation of sentence parses for the Hebrew text, which the author exploits to improve the Greek parses. Remarkable results are achieved: 79.4% of words are attached to their correct head word. Among these, 88.5% also receive correct labels, leading to general accuracy rate of 70.7%, results which are comparable to those attained by Mambrini and Passarotti, 2012.

2.3.3 Corpus annotation

In two papers based off their workshops on the topic [Dik and Whaling, 2008, 2009], H. Dik and R. Whaling (a classics professor and computer scientist turned classicist, respectively, both at the University of Chicago) demonstrate a relatively simple methodology for morphological tagging of a corpus of ancient Greek in the context of the Perseus under PhiloLogic project under Helma Dik. They trained Helmut Schmid's TreeTagger (found at `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/` and extensively described in Schmid, 1994, 1995) using a corpus of Homeric and New Testament Greek and applied it to run over their three-million word corpus. Initial results achieved about 80% accuracy and after adjustments rose up to 88%; since the corpus was designed for academic use, large swathes of text were then manually disambiguated to the best of the ability of the authors and a team of volunteers.

Their effort is remarkable in the sense that it is the only instance of an automatically annotated corpus of ancient Greek we have managed to find. While Perseus offers a morphological analysis tool, this tool is designed to assist linear reading and generates parses on the fly from a database, offering several options if several parses are possible. Recently, this system has been improved using user votes, frequency tables and a simple system of bigrams, in order to return the most likely parse. Dik and Whaling's corpus, though, has been annotated with the explicit intention of storing all parses and their context in a relational database. This makes it possible to perform morphological searches. For linguistic purposes, this is a very interesting tool.

3 ANALYSIS

In this chapter, the main objectives of this thesis are outlined more precisely, placed in their scholarly context and given motivation. We consider a **dual goal**: creating a statistical language model of ancient Greek using machine learning techniques, and applying this model to a corpus of documentary papyri.

We propose that both problems have not been tackled in an adequate way up to now.

3.1 PROBLEM STATEMENT

Firstly, we aim to develop a **language model for ancient Greek**. We understand this to be a statistical model designed to assign probability scores to word sequences; however, it is key that the model is also able to apply the knowledge of these probabilities to specific problems in the analysis of language. We extend this model to cover the tasks of **morphological analysis** and **partial syntactic parsing**.

3.1.1 Morphological analysis

Definition

The first problem largely corresponds with what is called **part-of-speech tagging** in the natural language processing jargon. For any token in the sentence, given its context, we want the model to produce a morphological analysis, which produces not only the part-of-speech *stricto sensu*, but all concomitant information as well: voice, tense, mood, case, gender, number, person, . . .

Benchmark

The Perseus word study tool is currently capable of analysing any Greek word form morphologically and offers a limited degree of disambiguation; nonetheless, no exact quantification exists of the correctness of these disambiguations and we cannot measure our results against the performance of this tool. The TreeTagger model developed in Dik and Whaling, 2008, 2009, on the other hand, has been evaluated for accuracy and reaches 91% accuracy by using an annotated training set consisting of 150K words from the Greek New Testament and 2K words culled from Lysias.

We aim to supersede this performance by leveraging raw textual data from the Perseus project and the *Thesaurus Linguae Graecae* using unsupervised machine learning techniques, as detailed *infra*. In our view, given much larger corpora than used by Dik and Whaling, we can achieve a 95% accuracy rate with minimal manual finetuning and integration with other morphological tools.

3.1.2 Partial syntactic parsing

Definition

The second problem corresponds with what is called **partial** or **shallow parsing**, or **chunking**. Given a sequence of words, we want to identify the main grammatical components of this sequence. This type of parsing stands in contrast with **deep parsing**, which aims to produce full syntactic analyses of entire sentences, including **parse trees**, which give a graphical representation of their syntactic structure.

Benchmark

Our closest competitor for this task is Mambrini and Passarotti, 2012, as the method employed by Lee, 2010 to attain high parsing accuracy is unfeasible. We stipulate that Mambrini and Passarotti, 2012 goes further than we plan to by engaging in deep parsing. Chunking as we will perform it is more restricted and does not attempt to map head-dependency structures. Instead, we simply look at text windows and attempt to assign a syntactic label to the central word of each window. For the deep parsing accuracy criterium most related to this, label accuracy, Mambrini and Passarotti achieve 62.9% accuracy on their only prose test set. We adopt this as our minimal benchmark.

3.1.3 Application to a corpus of papyri

Definition

Once the model is constructed, we aim to apply it to a corpus of papyri provided digitally by `papyri.info`. The object corpus contains about 4.5 million words. Given the nature of the provided material, the state of these texts varies from extremely corrupted to nearly pristine.

The texts are dated from 300BC to 800AD, spanning more than a millennium; many different discourse registers are represented, although literary texts are not included.

Linguistically speaking, this is a less than desirable state of affairs. By virtue of its diverse and fragmentary nature, the corpus will contain thousands of unorthodox or corrupted word forms. This heightens the complexity of the task at hand, and we cannot but lower our standards for accuracy if we wish to proceed in an automated way. Nevertheless, we aim to provide limited inferences for this type of token.

Benchmark

We can perform only very limited testing for this task, since there is no extant annotation for the object corpus. Instead, we evaluate the accuracy of our model when applied in an unsupervised fashion to a test set selected from the papyri. To supplement this, we plan to distribute this corpus in order to allow for manual verification by the philological community. Our end goal is to create an annotated version of the corpus in XML format, encoded in Unicode and following the TEI standard for XML documents.

3.2 RESEARCH CONTEXT

3.2.1 State of the question

While we can see from 2.3 that natural language processing for ancient Greek seems to have garnered some attention, we can make several observations of note on the current state of research.

The problem of Greek morphological analysis can essentially be considered solved for individual words, as demonstrated by the Perseus word study tool; contextual disambiguation, on the other hand, has only been attempted once with some success in Dik and Whaling, 2008, 2009. The idea of designing a system to automatically process ancient Greek as envisioned in this work was originally inspired by this approach.

Extending this methodology to syntactic analysis is a *desideratum*. There are only two projects concerned with treebanks or databases of semantically annotated Greek. The Perseus project has developed a dependency treebank for Latin and ancient Greek [decribed in Bamman and Crane, 2011, available at `http://nlp.perseus.tufts.edu/syntax/treebank/`]. It is an admirable effort, but limited in scope and containing mainly poetry. The project seems to be lacking manpower and has lost steam since its inception, as the last update dates from 2012, more than a year ago at the time of this writing.

Another interesting treebank is that hosted by the PROIEL [*PROIEL: Pragmatic Resources in Old Indo-European Languages*] project, which aims to offer morphologically and syntactically annotated multilingual corpora for comparative purposes. The project, contrary to the Perseus treebank, seems to be alive and well at the time of this writing. This corpus contains data which can be of much help: large swathes of Herodotus, the New Testament, and the writings of the Byzantine historian George Sphrantzes are fully annotated, both morphologically and syntactically.

An early prototype of this thesis attempted to use similar supervised methods to annotate the corpus of the papyri. Despite high expectations, experience showed that the lack of extensive annotated corpora is a severe hindrance, as the main way to improve the accuracy of any NLP system is to offer it more training data. Feeding 400.000 words as training data to the Stanford POS Tagger resulted in a measly 60% accuracy on a validation set held out from the training corpus.

It is key to observe that we have touched a sore point with this result: there is a deadlock between the development of extensive annotated corpora and the development of natural language processing tools for ancient Greek. The absence of sufficiently large and diverse annotated corpora is a severe impediment to the development of such tools; at the same time, these tools, due to lack of philological manpower, are necessary for the extension and improvement of such corpora! Breaking this deadlock is quintessential for further progress.

3.2.2 Proposed solution

We propose to do so by adopting alternative methodologies. Recent literature in the field of machine learning methods for English natural language processing revealed that state-of-the-art results can be attained using a combination of unsupervised and supervised learning techniques, dubbed semi-supervised approaches. Unsupervised approaches can make use of unannotated data as a preparation for supervised training, and work by trying to embed words in a vector space relative to other words according to their syntactic properties.

Notably, Collobert and Weston developed a versatile architecture which achieved high accuracy on several NLP tasks and required a relatively low amount of optimisation [Collobert and Weston, 2008;

Collobert, Weston *et al.*, 2011]. The architecture was originally applied to a diverse array of NLP tasks for English. Accuracy rates for POS tagging reached up to 97.20%, while for chunking, scores of up to 93.63% were achieved; state-of-the-art results were also achieved for named entity recognition and semantic role labeling, which we will not consider here. This is an impressive performance: most of the architecture is actually shared among all tasks and the majority of the parameters of the system are inferred through unsupervised methods.

Given that far larger amounts of raw textual material are available for ancient Greek, it seems that this kind of technique is suited to the problem at hand. The 400.000 word training corpus used in the experiment with the Stanford POS Tagger is much smaller and limited than corpora like that offered by the Perseus project (about 7M words) and the TLG (about 109M words at last count, though these are not freely available). Making use of this untapped resource is desirable.

Chapter 4 is dedicated to an overview of the architecture; the approach followed in Collobert, Weston *et al.* [2011] and Turian *et al.* [2010] is respected with amendments and simplifications where needed in order to accommodate for some characteristics of Greek (in particular the very high complexity of its morphology requires a subtler approach). The exact implementation of the system is left for chapter 5.

3.3 POSSIBILITIES

3.3.1 Corpus-based grammars and lexica

Expanding our method to other texts might bring the benefit of comprehensive corpus-based grammars and lexica, which can integrate available data on the fly and create a self-updating and reliable web of grammatical knowledge. Instead of focusing mainly upon a few choice authors or laboriously trudging through the huge wealth of ancient Greek literature to linearly create lexica and grammars, all of it could be harnessed at once in a quantitatively precise and easily visualisable way. Though this is not the place for an extended discourse on the methodology behind setting up such a system, we refer to Bamman and Crane, 2008, 2009 for an overview.

3.3.2 Historical and variational linguistics

The language of the papyri has an important role to play in the historical linguistics of Greek; once a full annotation has been achieved, it could be possible to implement the same methods used for synchronic language processing to map language changes in a statistical way; it could be possible to estimate the transition probabilities for diachronic grammatical evolutions, which has the potential to create a picture of the evolution of Greek that would be both comprehensive and precise. It even has potential on a comparative level; given the long history and meandering evolutionary trajectory of the Greek language, one could observe from the data catalysts for language evolution in one direction or the other and apply that comparatively.

One might also win valuable insight into language diversity in Egypt; using the paraliterary data already available from the Trismegistos, linguistic phenomena and evolutions could be visualised on a map and give insight into the diatopic, diastratic and diaphasic variation of Egyptian koinê, much in the way of modern dialect survey maps but directly linked to the original texts.

3.3.3 Textual criticism

Textual criticism, too, could benefit from improved access to linguistic data; dubious *passus* could be disambiguated by comparing them to similar instances in papyri from the same period and adapting constructions and words from them. This technique is harnessed by Mimno and Wallach, 2009, who use the techniques of statistical NLP solely for these specific critical problems. Though textual criticism will for the foreseeable future still necessitate trained papyrologists, the need for a very in-depth knowledge of the corpus of papyri can be greatly reduced by calling upon data from other parts of the corpus to present a series of statistically possible solutions for textual issues.

3.4 NAMED ENTITY RECOGNITION

Named entity recognition is a subdiscipline in natural language processing which is concerned with the automatic extraction and localisation of all kinds of names from texts. It has been used extensively in literary texts with a view to discern the importance of certain characters throughout the text. The KU Leuven's long-standing Prosopographia

Ptolemaica project, which aims to be a repository of all inhabitants of Egypt between 300 and 30 B.C., could easily benefit from these techniques.

The abundant manual labour that has gone into the project could be fed as training data to and then supplemented by a named entity recognition engine that could also categorise personal names by any criteria and establish contextual relations between them. To take a very rudimentary example, the name 'Alexander' could be retrieved in all texts and a cluster of related names generated, so that related individuals may be placed in a web of relations; or one could ask, by combining the already present linguistic annotation, to display all adjectives which accompany the name 'Alexander'.

It could even go further than this and also include other particular names, such as places, distances, monetary units, weights, and so on. Historians could create a comprehensive overview of, for instance, the inflation of Egyptian currency, or map out trade connections using a search for all mentions of currency, weight and places which are in proximity to each other.

Part II

Methodology

4 DESIGN

4.1 OVERVIEW

In general, a language model is created by choosing a modeling criterium and applying it to a large set of observations. These observations may be raw word sequences or annotated word sequences, depending on the chosen criterium. Each observation is coupled with an adjustment of the language model to maximise the score for that particular observation type. The larger the set, the better, as each observation makes the model a better reflection of linguistic reality.

We choose a two-phase training method and exclusively follow the natural language processing architecture based on deep neural networks described in Collobert, Weston *et al.*, 2011; all theories and techniques described are their work and ideas, except for the specific adaptations made to accommodate the system to ancient Greek. We give a summary overview of the architecture as described in the paper; subsequent sections provide more detail and specific modifications necessary to process ancient Greek.

The first phase uses a large unlabelled corpus to create a simple probabilistic model using as little linguistic knowledge as possible: the goal is to assign scores to word sequences that are proportional to the probability of these sequences being 'correct', i. e. likely to appear in real linguistic data. Maximising these scores is achieved by corrupting data from the observation corpus; the model is then adjusted to give the real observation a higher score than the corrupted observation.

This method creates a structured internal representation for each word in vector form, which defines how the word is embedded within an n-dimensional vector space; hence, these vectors are termed **embeddings**. (*Ne sit confusio*: note that the components of this feature vector do not necessarily show a one-on-one correspondence with linguistic features.)

The second phase is implemented on top of the first. The embeddings created in the first phase are used to initialise new networks for each of the tasks we want to train. We then use a new observation corpus, this time equipped with annotation. We convert these annotations to a vector form, with each vector component representing a linguistic feature. The parameters of the network are then further adjusted to fit the behavior of the observation corpus. The scores returned by the model are now in vector form, each component being a score similar to the one developed in the first phase.

Performance improvements are expected due to the fact that at this stage, most of the general learning is actually already done and we are applying classification to certain clusters in the vector space, which allows the model to make more accurate inferences when classifying rare words or phrases.

The model can then be used to generate annotation for unlabelled sentences. Given a sequence centered around a word, a vector containing scores for each possible tag for this central word is returned. Prediction is handled by concatenating this type of output for entire sentences and applying the Viterbi algorithm [described formally in **??**] to the resulting matrix.

4.2 NETWORK STRUCTURE

Deep networks contain multiple layers which are sequentially trained; the input of a layer is weighted and passed on to the following layer, which may be either an output layer or a hidden layer. Several layers are stacked in this manner.

4.2.1 Hyperparameters

Before constructing a network, we need to decide on a set of hyperparameters that will influence the parameters of the neural network and its training process. These are:

- the **embedding dimensions**, written d_{wrd}: the number of components in a word feature vector;
- the **dictionary size**, written D: how many words we want to consider when training;
- the **window size**, written wsz: the number of words we want to consider per example;
- the **learning rate**, written λ: when using gradient descent, how much we want to adjust the network parameters at each gradient step;

- the **embedding learning rate**, written λ_e: the same as λ, but for the purpose of learning embeddings - usually smaller to limit the influence of individual observations on the training process;
- the **input, output and hidden size**, written n^l_{in}, n^l_{out}, and n^l_{hu}, respectively: the number of neurons contained in a layer l.

4.2.2 Lookup table layer

The lookup table itself is a matrix with d_{wrd} rows and D columns. Its initial construction is as follows: given a frequency table, the D most common words are chosen and placed in order. Each word is now assigned an index according to its ranking in the frequency table. The most common word gets index 1, the second most common index 2 etc., up to the word in the D^{th} place in the table, which is assigned index D. For each index, a new embedding of size d_{wrd} with small random values is created and assigned to that index. The lookup table matrix itself is constructed by concatenating these D vectors as columns. In this way, a lookup operation for an index i is actually nothing more than the selection of the i^{th} column from this matrix.

The lookup table layer itself consists of wsz input neurons; given a text window, each word is converted to its corresponding index, which is then fed to the neuron corresponding to its position in the window, which retrieves the embedding of that word from the lookup table. The output of all neurons is then concatenated into a single matrix, whose columns are now the embeddings for the input window.

Formally, given a word index vector $w \in N^{wsz}$ and a lookup table $L \in R^{d_{wrd} \times D}$, we can express this as a function LT:

$$LT(L, w) = \begin{pmatrix} L_{1,w_1} & L_{1,w_2} & \dots & L_{1,w_{wsz}} \\ L_{2,w_1} & L_{2,w_2} & \dots & L_{2,w_{wsz}} \\ \dots & \dots & \dots & \dots \\ L_{d_{wrd},w_1} & L_{d_{wrd},w_2} & \dots & L_{d_{wrd},w_{wsz}} \end{pmatrix} \tag{1}$$

An interesting feature of this type of representation is that it is in fact a highly performant abstraction for the classic n-gram. In most NLP architectures n is given a relatively low value in order to limit computational expenses; the Google n-gram project, which is the largest of its kind, limits itself to five-grams. N-grams are also used differently: the first $n-1$ words serve as the context for the n^{th} word in the n-gram. Here, we can take advantage of the purely numerical form of

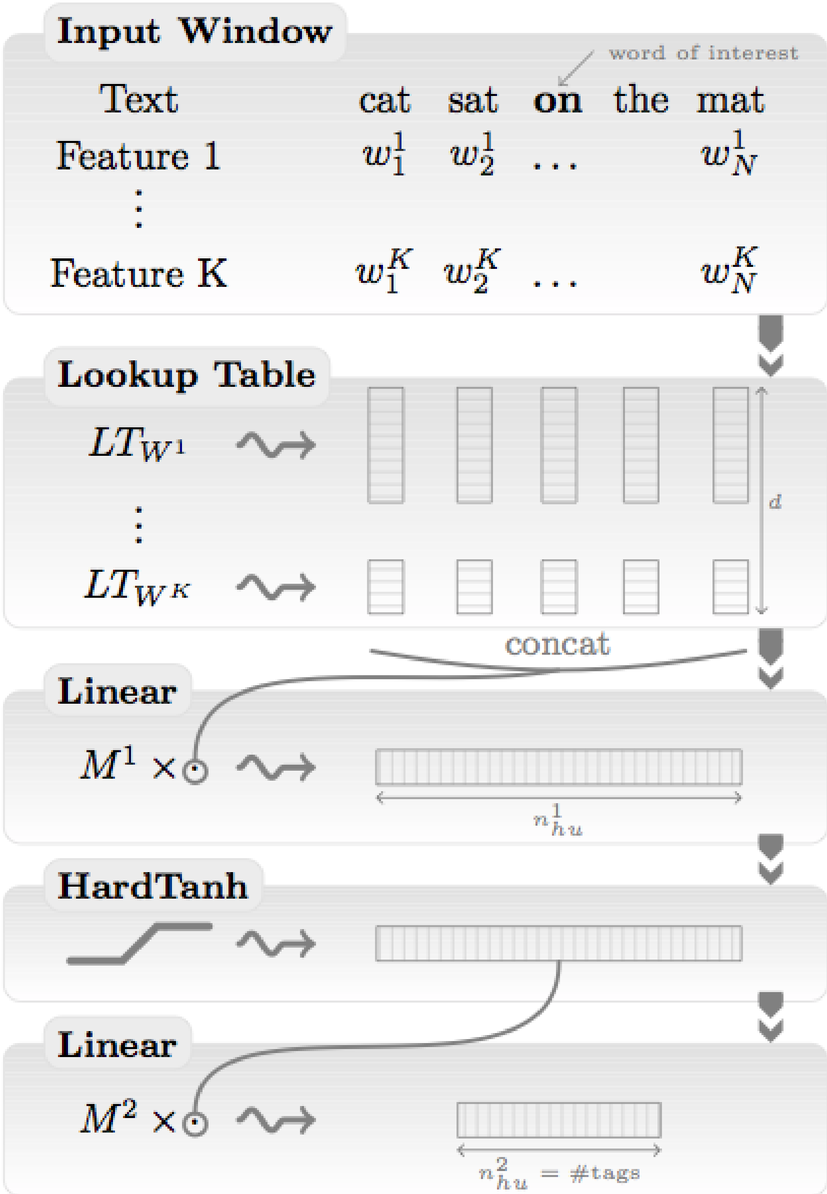


Figure 1: The basic network structure, using a window approach. Figure from Collobert, Weston *et al.*, 2011, p. 2499.

text window embeddings by choosing a somewhat larger text window size and considering the central column in such windows in its full context, both prior and posterior.

4.2.3 Linear layer

A linear layer takes a fixed size vector and performs a linear operation on it: the dot product of this vector with a set of parameters W is computed and a bias added. Formally, given the output vector f_θ^{l-1} of layer $l-1$, the following computation is performed in layer l:

$$f_\theta^l = W^l \cdot f_\theta^{l-1} + b^l \tag{2}$$

Where θ indicates the existing parameters of the network and $W^l \in R^{n_{hu}^l \times n_{hu}^{l-1}}$ and $b^l \in R^{n_{hu}^l}$ are the parameters of the layer to be trained, with n_{hu}^l representing the amount of hidden units in layer l. Linear layers transform their input and several such layers can be stacked, similar to how linear functions can be composed.

4.2.4 Hard hyperbolic tangent layer

If we intend for our network to be able to model a highly nonlinear system such as language, we need to introduce nonlinearity somewhere. A good function for this is the hyperbolic tangent, which is differentiable everywhere and approximates a linear threshold function very nicely. A layer l using the hyperbolic tangent as an activation function contains n_{hu}^l neurons taking n_{hu}^{l-1} inputs. In this case, the activation function $g(x)$ for a scalar x is:

$$g(x) = tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3}$$

For an input vector generated by a layer $l-1$, the function g represented by a hyperbolic tangent layer can be defined as:

$$f_\theta^l = g(f_\theta^{l-1}) = \begin{bmatrix} g(f_{\theta 1}^l) \\ g(f_{\theta 2}^l) \\ \cdots \\ g(f_{\theta n_{hu}^{l-1}}^l) \end{bmatrix} \tag{4}$$

We approximate this function using the hard hyperbolic tangent, defined for a scalar as:

$$\mathrm{hardtanh}(x) == \begin{cases} 1, & \text{if } x > 1 \\ -1, & \text{if } x < -1 \\ x & \text{otherwise.} \end{cases} \tag{5}$$

We can define this function for vector inputs in the same manner as for the hyperbolic tangent function.

4.2.5 Output layer

The final linear layer. This layer is designed to output a vector containing as many elements as there are possible tags for the task at hand. Each output element is a score which reflects a network score for the corresponding tag for the central word in the input window.

4.2.6 Softmax layer

An optional layer, used for supervised training and tagging (see *infra*). This layer applies the softmax operation [Bridle, 1990] to the output vector of the previous layer to obtain an output which converts the scores such that they sum to one, just as the probabilities in a sample space do.

4.3 PHASES OF LEARNING

4.3.1 Unsupervised learning

The first phase of learning is unsupervised; large amounts of raw language data are fed to the network. Instead of training using a classical squared-loss function, a pairwise ranking function is introduced. The network is constructed as described in the previous section; we want a single score $f_\theta(x)$ to be output for a given window of text x. The window is first corrupted using a word w from the dictionary by replacing the central word in x by w. We express this corrupted window as $x^{(w)}$. The pairwise ranking of any two such pairs x and w is defined as $r(\theta, x, w) = \max\left\{0, 1 - f_\theta(x) + f_\theta(x^{(w)})\right\}$. In effect, we want the non-corrupted window to achieve a higher score than the corrupted window. We can achieve this by adjusting the parameters θ such that the pairwise ranking of x and w is minimal, since this implies that $f_\theta(x)$ must yield a higher score than $f_\theta(x^{(w)})$.

Summing this operation over all possible pairs (x, w) and defining a mapping from the parameters θ to this sum, we obtain a general cost function:

$$\theta \mapsto \sum_{x \in X} \sum_{w \in D} r(\theta, x, w) \tag{6}$$

Where X is the set of possible windows of size *wsz* and D the chosen dictionary. Minimizing this function with respect to θ will ensure the relevant parameters (the embeddings and the first two layers) are tuned so that our ranking function f_θ yields accurate scores.

Using this simple criterion, we have a method for crafting a set of parameters that contains a consistent structured internal representation of the training data. Despite the relative simplicity of the criterion, the large amount of parameters results in a very taxing and lengthy computation. Furthermore, there is no guarantee that the cost function has a single minimum with respect to θ; a full grid search would be necessary, which necessitates vast amounts of computing time.

Instead, the process is sped up using **curriculum learning**; the basic idea of this technique is analogous to the learning process children are put through in school: instead of starting their education with university-level quantities of difficult material immediately, a restricted set of elementary concepts is introduced on which they concentrate. Successive phases of learning are performed by gradually expanding the set of concepts which is to be learned, making use of earlier concepts to facilitate the understanding of more complex concepts.

The same method, for reasons not yet fully understood, can be applied to unsupervised learning.[1] First, the training material is restricted to the most frequent observations of the process we want to model. Training over this restricted set creates a simplified model which, due to the abundance of examples, should be accurate. Subsequently, new iterations of the learning algorithm are run over successively larger sets; at each iteration, the model becomes more detailed and describes more classes of observations more accurately.

This is applied to the problem at hand by choosing successively larger dictionary sizes. During the calculation of the minimum of the cost function, windows which are not centered around a word which is in the dictionary are ignored. This initially entails a significant reduction of our sets X and D, which makes the process a bit less computa-

1 Among the scholarship of note on this subject we find Bengio *et al.* [2009] and Erhan *et al.* [2010].

tionally demanding. Subsequent iterations are computationally more expensive, but are initialised with the parameters found by previous iterations; observations previously used in learning will already return excellent scores and will only necessitate minor adjustments to the parameters, while new observations can be fit into the general picture more easily.

We note that we do not generate all possible corrupt windows for parameter adjustment each time we treat a text window. Instead, at each window, we pick a random word with which to corrupt the window; we improve the model by running the learning algorithm for several epoch; this type of training minimises the time to process a training corpus once but can be run indefinitely. We can set a number of epochs to run successively, and define a validation criterium for transitioning to the next epoch.

4.3.2 Supervised learning

The supervised training phase involves the creation of task-specific networks which are initialised with the embeddings created during the unsupervised phase; these form a shared first part of all networks.

The output of a network of this type is a vector; each specific feature is encoded in on component of this vector by setting this component to one and all others to zero. This is called a one-hot vector (*cfr. infra*). We convert each output component to a probability by passing it through the softmax layer, which normalises all output scores such that they sum to 1.

As our cost function, we choose the sum of log-likelihoods for the entire training corpus [Collobert, Weston *et al.*, 2011, pp. 2505–6]:

$$\theta \mapsto \sum_{(x,y)\in T} \log p(y|x,\theta) \tag{7}$$

where x is a window, y is the tag for the central word, and T is our training corpus. This is known as the *cross-entropy* criterion [Rubinstein and Kroese, 2004]. Where we tried to minimise the cost function with respect to its parameters in the unsupervised phase, we will now try maximise the new cost function such that the probability output for windows similar as the ones provided during supervised training is as high as possible.

All tasks are jointly trained, that is to say, the networks share parameters in the form of embeddings and the hidden layer, are trained simultaneously by alternating training between them and are allowed to modify these shared parameters. The individual (non-shared) network parameters, those belonging to the output layers, are modified only during training for a specific task. This technique allows us to generalise the training benefit from each example.

4.4 ADAPTING THE ARCHITECTURE FOR ANCIENT GREEK

Composing such a one-hot vector is not simple for Greek morphology: any given word will belong to multiple morphological categories. For instance, a verbal form has a tense, a mood, a number, a person, and sometimes even a case and gender. Gathering all possible features into a single one-hot vector is therefore not feasible.

We could approach this problem in different ways. An instinctive test is to simply feed the system raw parses and assign one component of the output vector to each possible parse. This is needlessly complicated; if we look at the list of morphological parses from the Perseus database, we find more than 2000 distinct morphological analyses! An output vector of this size is simply too unwieldy.

A more dynamic approach would be to create one-hot vectors for each of the following categories, with the number of options assigned to each category corresponding to the number of components in the corresponding output vector:

- major part of speech: verb, noun, adjective, pronoun, particle, adverb, numeral, preposition, conjunction, interjection;
- minor part of speech: article / determinative, personal, demonstrative, indefinite, interrogative, relative, possessive, reflexive, reciprocal, proper;
- person: first, second, third;
- number: singular, plural, dual;
- tense: present, imperfect, aorist, perfect, pluperfect, future, future perfect;
- mood: indicative, subjunctive, optative, imperative, infinitive, participle, gerundive, gerund, supine;
- voice: active, middle, passive, middle-passive;

- gender: masculine, feminine, neuter, common;
- case: nominative, genitive, dative, accusative, ablative, vocative;
- degree: comparative, superlative.

These can be encoded in various ways. In tables 1 and 2, we give a description of the morphological annotation system used in the annotated corpora made by the PROIEL project, which we use in our supervised phase.

The one-hot vector approach has its downside: we now have to train distinct networks for each network. The upside, though, is that each of these networks is much, much smaller than a single network mapping all possible parses and will be easier to train; an example of the divide-and-conquer technique. We could possibly be confronted with impossible parses, such as an 'imperfect optative', but this is highly unlikely due to the total absence of examples for this form.

The process of syntactic annotation only requires one network, but is a bit more complex due to an increased amount of tags. We see in table 3 that twenty-four different features are possible. We use a similar method to convert annotations.

We assign numbers corresponding to vector components to all possible features and store them in different hash tables. These are then used to convert back from the normal notation to one-hot vector notation. Section 5.3.2 contains a JSON dump of these hash tables.

field	category	possible values
first field	person	1 - first person 2 - second person 3 - third person
second field	number	d - dual p - plural s - singular
third field	tense	a - aorist f - future i - imperfect l - pluperfect p - present r - perfect t - future perfect
fourth field	mood	i - indicative m - imperative n - infinitive o - optative s - subjunctive
fifth field	diathesis	a - active e - energetic m - medial p - passive
sixth field	gender	f - feminine m - masculine n - neuter
seventh field	case	a - accusative d - dative g - genitive n - nominative v - vocative
eighth field	degree of comparison	c - comparative s - superlative
ninth field	placeholder column	-
tenth field	inflectibility	i - inflected n - not inflected

Table 1: The PROIEL decaliteral morphological abbreviation system.

field	value
A-	adjective
C-	paratactic conjunctions
Df	adverbs
Dq	adverbial response particles (where, how, etc.)
Du	adverbial question particles (where, how, etc.)
F-	Hebrew loan words
G-	hypotactic conjunctions
I-	illocutive particles
Ma	cardinal numerals
Mo	ordinal numerals
Nb	nouns (in general)
Ne	nouns (proper names)
Pc	pronouns (reciprocative)
Pd	pronouns (demonstrative)
Pi	pronouns (interrogative)
Pk	pronouns (reflexive)
Pp	pronouns (personal)
Pr	pronouns (relative)
Ps	pronouns (possessive)
Px	pronouns (quantitative, i.e. some, all, none, same, other)
R-	prepositions
S-	article
V-	verb

Table 2: The PROIEL biliteral lemmatic abbreviation system.

tag	**value**
adnom	adnominal
adv	adverbial
ag	agens
apos	apposition
arg	argument (object or oblique)
atr	attribute
aux	auxiliary
comp	complement
expl	expletive
narg	adnominal argument
nonsub	non-subject (object, oblique or adverbial)
obj	object
obl	oblique
parpred	parenthetical predication
part	partitive
per	peripheral (oblique or adverbial)
pid	Predicate identity
pred	predicate
rel	apposition or attribute
sub	subject
voc	vocative
xadv	open adverbial complement
xobj	open objective complement
xsub	external subject

Table 3: The PROIEL treebank annotation system.

5 IMPLEMENTATION

5.1 LANGUAGE AND SOURCE CODE

5.1.1 Choice of language

The language modeler is programmed in Python; as programming languages go, it possesses the clearest syntax and is reasonably concise. Python runs in an interpreter and is slower than compiled languages such as C or Java, but this is remedied by the large amount of available libraries designed to circumvent this issue. For computationally demanding numerical problems such as are frequently found in machine learning, we can use libraries such as NumPy, SciPy, Theano, ... These offer implementations of frequently used numerical algorithms written in C, which are compiled during the execution of the program and cached.

In the past few years, Python has been switching from version 2 to version 3, which brought a lot of changes in syntax and generated a great deal of cross-compatibility problems. Despite this, many libraries are now available for Python 3, including the ones we are interested in using. Therefore, we chose to use Python 3.3 instead of Python 2.7.x; it offers superior Unicode and string processing capabilities to preceding versions.

5.1.2 Source code availability

The source code is available at `https://github.com/sinopeus/thrax`. The core structure and numerical algorithms, encapsulated in Theano graphs, are largely based on Joseph Turian's implementation of [Collobert and Weston, 2008] in Python 2. I rebuilt the entire program surrounding this numerical component to fit my needs: the result is a documented, cleaned up and overall improved program. I also added functionality for the supervised phase to complete the picture. The program's configuration system was enhanced and is now easily editable by hand as well as programmatically. With basic knowledge of programming, it is possible to train a network for any given NLP task by modifying the configuration.

5.1.3 Requirements

The program was written for Python 3.3.2 using development versions of the following libraries:

- NumPy: 1.8.0, https://github.com/numpy/numpy;
- SciPy: 0.13.0, https://github.com/scipy/scipy;
- Theano: 0.6rc3, https://github.com/Theano/Theano;
- h5py: 2.2.0b1, https://github.com/h5py/h5py/tree/master/h5py.

Each of these libraries has its own dependencies, which, evidently, the user desirous to replicate our setup should install.

5.2 IMPLEMENTING THE NETWORK EFFICIENTLY

In this section, we highlight a few optimisations applied during the implementation of the neural network; using these, we reduce the computational requirements of training such a network significantly. These techniques are commonplace in machine learning and programming in general and should not be considered original ideas by the author.

5.2.1 Lookup table

The lookup table is initialised from a dictionary file sorted by descending order of frequency (see *infra*). A hash table is created which associates each word in the dictionary with its frequency rank. Separately, a matrix of dimensions $D \times d^{wrd}$ is set up which is filled with random floating point numbers between 0 and 10^{-2}. A lookup table operations is now a two-step process on two discrete data structures. This allows us to store the word sequences more compactly for processing. If necessary, we can meld both structures into one (if we wish to redistribute the embeddings and the dictionary together in a serialised format, for example). During tagging, we reverse this dictionary to allow us to lookup the word corresponding to a given lookup table index.

5.2.2 Batch processing

Examples are not processed individually by the system, but in batches. First, a batch size is chosen. Then, a number of text windows corresponding to this batch size is read from the training corpus. Each individual window is passed through the lookup table, which returns matrices such as the one in 1. We linearise these matrices into vectors by concatenating their columns.

Batch processing is advantageous when we want to train according to a cost function which contains a large set of parameters, such we do; by using batches, the gradient of the cost function is taken with respect to several examples to ensure that parameter adjustments necessary for each the training examples balance each other out.

5.2.3 Matrix-vector representation

The representation given in the previous chapter is advantageous in the sense that it is organised clearly, with each layer assuming exactly one task. When actually programming such a network, it is best to choose a data structure which reduces the amount of space needed and simplifies the computation. We switch to matrix-vector representation, where a network contains the following objects:

- an embedding matrix as defined in 5.2.1;
- a hidden weight matrix of dimensions $n_{in} \times n_{hu}$;
- a hidden bias vector of dimension n_{hu};
- an output weight matrix of dimension $n_{hu} \times n_{out}$;
- a output bias vector of dimension n_{out}.

We generate an output by performing a simple sequence of operations. Given the output of a lookup table operation over a text window in matrix form, we multiply this with the hidden weight matrix and then add the hidden bias vector to each column. The resulting matrix is passed through the chosen nonlinear function, in our case the hard hyperbolic tangent, which is applied to each element in the matrix. The resulting matrix is multiplied once more, this times with the output weight matrix, and the output bias vector is added to each column. In this manner, we obtain a correctly sized output vector with minimal computational overhead. An additional advantage is, due to the properties of matrix multiplication, we can easily apply the exact same operation to a complete batch such as described in 5.2.2 by repeating this operation over each row of the batch tensor.

5.2.4 Optimising computation with Theano

Theano, proposed in Bergstra *et al.*, 2010 and available at http://deeplearning.net/software/theano/, is a numerical library for Python which allows for efficient computations on large arrays using symbolic expressions. We mention a few key optimisations of the implementation made using Theano.

Graphs and symbolic expressions

A Theano graph is a structured representation of a computation. The nodes of such a graph belong to one of three types: variables, operations and applications, termed variable, op and apply nodes. The structure and sequence of the computation is stored by connecting these nodes; a connection is called an edge. Such graphs can be constructed symbolically by defining a set of variables of the types provided by Theano and then applying a sequence of operations to them. The library registers the operations and incrementally constructs a graph. Once the graph is constructed, we can store it as a function which we can apply to any valid input which is of the form of the initial variables of the graph.

For instance, we might define an input matrix as a Theano variable, apply the necessary neural network layers to it, and then return a function which we can apply to any variable of the same type as the original graph input. Furthermore, Theano applies optimisations to these functions to improve their efficiency, accuracy and execution speed.

Automatic differentiation

This approach yields the benefit of automatic differentiation. Every operation defined by Theano is also paired with a its derivative function, which is precomputed. If we model a computation which consists of the application of several operations (i.e. we create a composite function), it is possible to apply the chain rule to the respective derivatives of the operations of which the computation consists; by doing this, we can find the derivative of the entire computation. In this way, Theano can compute the derivative of symbolic expressions of arbitrary length at minimal cost. Given the essential role of derivative functions in machine learning in general and in neural networks in particular, this is truly a boon. We can forgo the arduous manual computation of the neural network layer gradients and automate the process, and simultaneously benefit from increased precision and efficiency.

Compiling to C

Another important optimisation performed by Theano is situated 'closer to the metal', so to speak. Python is a high-level programming language, that is to say, it is to a great degree an abstraction of the internal workings of a computer, hiding details such as memory management and register operations from the user in favor of a more intuitive way of writing programs. This reduces the amount of manual optimisation that can be applied to a program: lower-level languages such as C allow direct access to basic machine functions. A Python program will typically require a running time ten to one-hundred times longer than an equivalent C program.

For computationally demanding numerical routines which are applied over large datasets, this is wholly undesirable. Theano can avoid being tied down to Python-level performance by partially or fully compiling functions constructed from graphs to C code, which is then compiled and stored for the rest of the execution of the program. This offers a tremendous speed increase for certain types of computation.

5.3 THE FULL PROCESS

5.3.1 Preparing the training corpora

We begin by preprocessing the training corpora. Since we need a corpus for each phase of learning, but most preprocessing is analogous, we give one set of instructions and add on a few for the supervised phase. The `prep_scripts` directory in the repository and the bundled CD provides Python scripts for performing all necessary steps efficiently; the `README` file shows how to use these.

5.3.2 Encoding tags

To encode the morphological annotation system, we use two data structures: a task dictionary, which assigns integer indexes to each task, and a list of dictionaries, each of which assign numerical indexes to each specific tag, corresponding to the component of the one-hot vector we want to activate.

```
[
  { "POS": 0, "PERS": 1, "NUMB": 2, "GEND": 3, "CASE": 4, "TENS": 5,
    "MOOD": 6, "VOIC": 7, "DEGR": 8, "INFL": 9, "SYNT": 10 },

  [
    { "A-": 0, "C-": 1, "Df": 2, "Dq": 3, "Du": 4, "F-": 5, "G-": 6,
      "I-": 7, "Ma": 8, "Mo": 9, "Nb": 10, "Ne": 11, "Pc": 12,
      "Pd": 13, "Pi": 14, "Pk": 15, "Pp": 16, "Pr": 17, "Ps": 18,
      "Px": 19, "R-": 20, "S-": 21, "V-": 22 },

    { "-": 0, "1": 1, "2": 2, "3": 3 },

    { "-": 0, "s": 1, "d": 2, "p": 3 },

    { "-": 0, "m": 1, "f": 2, "n": 3, "o": 4, "p": 5, "q": 6 },

    { "-": 0, "n": 1, "g": 2, "d": 3, "a": 4, "v": 5 },

    { "-": 0, "a": 1, "f": 2, "i": 3, "l": 4, "p": 5, "r": 6, "t": 7 },

    { "-": 0, "g": 1, "p": 2, "i": 3, "m": 4, "n": 5, "o": 6, "s": 7 },

    { "-": 0, "a": 1, "e": 2, "m": 3, "p": 4 },

    { "-": 0, "c": 1, "p": 2, "s": 3 },

    { "i": 0, "n": 1 },

    { "adnom": 0, "adv": 1, "ag": 2, "apos": 3, "arg": 4, "atr": 5, "aux": 6,
      "comp": 7, "expl": 8, "narg": 9, "nonsub": 10, "obj": 11, "obl": 12,
      "parpred": 13, "part": 14, "per": 15, "pid": 16, "pred": 17, "rel": 18,
      "sub": 19, "voc": 20, "xadv": 21, "xobj": 22, "xsub": 23 }
]
]
```


Figure 2: JSON encoding for PROIEL annotation.

General method

Firstly, we convert all corpora to plain text. The TLG is made available in its own format; Perseus provides all texts in XML; and the PROIEL project offer different formats, the handiest of which will be the CoNLL format. We strip all critical, linguistic and discourse annotation.

We then convert the plain text corpora to Unicode characters; both the TLG and Perseus encode all their Greek texts using ASCII by proxy of Beta Code, but our object corpus is encoded in Unicode. We consider the spacing and punctuation provided by the corpora as sufficient tokenisation for our purposes. We then realign the corpora by inserting and deleting line breaks such that each line maps to exactly one sentence; since the approach depends on text windows centered on one word, we add padding to both sides of the sentence by prepending and appending the word PADDING $wsz/2$ times before and after each line.

After having converted all texts to this format, we merge them into one large text file. For assessment purposes, we split the file into nine parts training corpus and one part validation corpus. We realign everything once more; we now obtain a file where every line maps to exactly one word or punctuation symbol. Sentences are now delimited using empty lines. From this file, we also create a frequency table which we use to create a comprehensive dictionary which we use for the lookup table layer in the network.

Supervised corpora

The creation of our supervised corpus is performed analogously, with the difference that all annotations are exported to separate files. The annotation systems are then normalised to one unified system; all layout changes are performed jointly on both the text and the annotation file to prevent misalignment. Once this is done, all annotations are split into individual characters and distributed over several text files for further processing. We want to create a one-hot vector for each specific character; a script is provided to automate this. The final vectors are not stored in plain text but as a serialised matrix to reduce space usage and facilitate the initialisation of the program.

5.3.3 Training the model

The algorithm is iterated over increasing dictionary sizes: we started with 4.000 words and subsequently doubled the dictionary size at each iteration. At each iteration, we validate our model; we stop at the point of diminishing returns to avoid computational overhead, since the full dictionary contains more than 700K word forms, and we do not want to be calculating the pairwise ranking criterion for this many words over our corpus, given the fact that it is already large.

The supervised phase is initialised with the embeddings created by the unsupervised algorithm, as well as the first linear layer. The same hyperparameters are used, except that we modify our output size according to the task at hand by giving it the same dimensions as the necessary output vector.

After the architecture (model and networks) is built, it is serialised; that is to say, the internal state of the architecture during training time is stored to disk. Serialisation allows us to immediately load the model into memory during the execution of our tagging program. When tagging, we use the architecture in a read-only manner, i.e. we only predict and do not adjust parameters any more.

Tagging is essentially a process of probabilistic prediction; text windows are passed through each of the networks, which return a prediction of the expected features of the central words in these windows in the form of an output vector. For tagging a sentence with n words, we create n text windows and use these as input. Each window generates an output vector; we pick the component with the maximum score and attribute the corresponding tag to the central word in the original text window. This process is iterated over every sentence in the target text.

5.3.4 Preparing the object corpus

The corpus is provided by `papyri.info` in EpiDoc XML format and freely accessible at https://github.com/papyri/idp.data. Each papyrus is stored in one XML file; the format itself allows for extensive annotation, facilitating textual searches to a great degree. However, the techniques we have used to annotate text require it to be provided in plain text format. We provide a script to strip all XML markup.

Once we have everything in plain text, we can process the text as we did in the previous phase: extraneous characters are removed, one word is placed on each line, and sentences are delimited using spaces. In this way, we can easily align annotations with minimal headaches. We also create an extra accompanying file equipped with line numbers for the corresponding words in the text file in order to track each word back to its original line.

Once the corpus is preprocessed, we can proceed with tagging. The method described in the previous chapter is used. We tag one papyrus at a time and then write the tagging output to disk following the directory structure of the `papyri.info` repository. We repeat the process for every papyrus.

Part III

Results & discussion

6 RESULTS

6.1 EXPERIMENTAL SETUP

6.1.1 Training material

For the unsupervised learning phase, we need a maximally large corpus. We chose the TLG CD-ROM E, which contains about 9.3M words, and the Perseus texts, which contain about 7.7M words. Since both corpora share material, duplicate sentences were scrapped. The final corpus contains about 16.9M words.

This corpus was split sentence-wise into a training corpus, from which representations are learned, and a validation corpus, to check the accuracy of the generated representations. The file is split 90-10.

The supervised learning phase makes use of the PROIEL annotated texts of Herodotus and the New Testament for both morphology and syntax. This contains approx. 195K words. Again, a validation set is withheld, in a slightly lower proportion than in the unsupervised phase due to the restricted size of the corpus.

6.1.2 Execution

For the training hyperparameters, we chose to set embedding sizes at 50. The text window size was set at 11 due to the prevalence of long-range dependencies in ancient Greek. The learning rates for the neural network parameters and the embeddings were set at $1.1 \cdot 10^{-8}$ and $3.4 \cdot 10^{-11}$, respectively, for the unsupervised phase. For the supervised phase, we raised the parameter learning rate to 0.01. The input layer size was set equal to the window size; the output layer size was set to 1; the hidden layer size was set to 100.

Corpus preprocessing was done on the author's own computer. We then conducted training on an Amazon EC2 compute-optimized instance. The unsupervised algorithm was left to generate embeddings for three days. This was equivalent to about thirteen training epochs. The supervised algorithm was left to iterate over the annotated corpus, finishing in about fifteen minutes. Logs of the training process were

σωφρόνως 6823	θαλαττίου 52759	περιβάλλοντες 67631	μέμψαιτο 32325
τ;	ατθυε	ἀποδείξειν	ἐκπορίζειν
δρουγγαρίου :β ϛο	κάμψαι	ἰέναι	τοῖαι
κτώμενοι	κεῖνται	ἀφῃρημένον	νημερτέα
ποιεῖσθε	Σαυνίτας	Εἵλωτας	κτίσει
διανέμεσθαι	ἀγανακτοῦντος	ἀπήνεγκαν	καθεστῶτα
δημιουργοῦ	πράξεως	μυίας	χάριτες
δέρρεις	πόλεως	τοπικὰ	κατηγορουμένων
πλανηθέντες	πληθυντικῶς	πελαγίαν	Κάλλιστον
παρελάμβανεν	ταράσσειν	Μαγδαληνὴ	συμπλεκομένους
προαιρουμένου 96784	**ἔρχεσθαι 9084**	**ψεύστας 99698**	**῾Περὶ 6334**
θυιππε	ἐρωτᾶται	ἀφελόμενος	κεῖσε
ἀγνοεῖν	ἱππικὰ	κἄλλως	ἀναλύειν
Τηλεβόας	Ἀθηναῖοσὦ	ὁὒχ	σύγκρασιν
Ἅλυς	ἑωυτῇ	Ἀχελῷος	μύραιναν
εὐδοκήσας	θαυμαστός	δαῖμον	Παῦλος
ταλαιπωρίαις	καρτερὰ	ἐπανιόντα	παλαιστής
ὡρισμένος	ἧττοσι	ἐγγυᾶται	μετάληψιν
καθεστῶτι	κρᾶτ΄	ποκ΄	φαύλῳ
καθέδρας	Λήδας	διασαφῶν	ἀπετάξατο

Table 4: Eight randomly chosen words and their nearest neighbors. We only picked words from the first 100.000 embeddings, as the cost of computing the nearest neighbors of a point increases exponentially when the point set becomes larger.

generated, which we converted into graphs which are displayed on the following pages. We did not tag the corpus for reasons detailed in 7. The resulting language model was then serialised for immediate reuse; a dump of the model parameters was also created in HDF5 format.

6.2 PERFORMANCE

6.2.1 Unsupervised model

To quickly get a general picture of the quality of the embeddings, we pick ten words throughout the lookup table and take their ten nearest neighbors in the vector space according to the Euclidean metric. The results are visible in 4.

The training evolution is visible from figure 3 onward. The red vertical lines on each graph indicate the start of new training epochs.

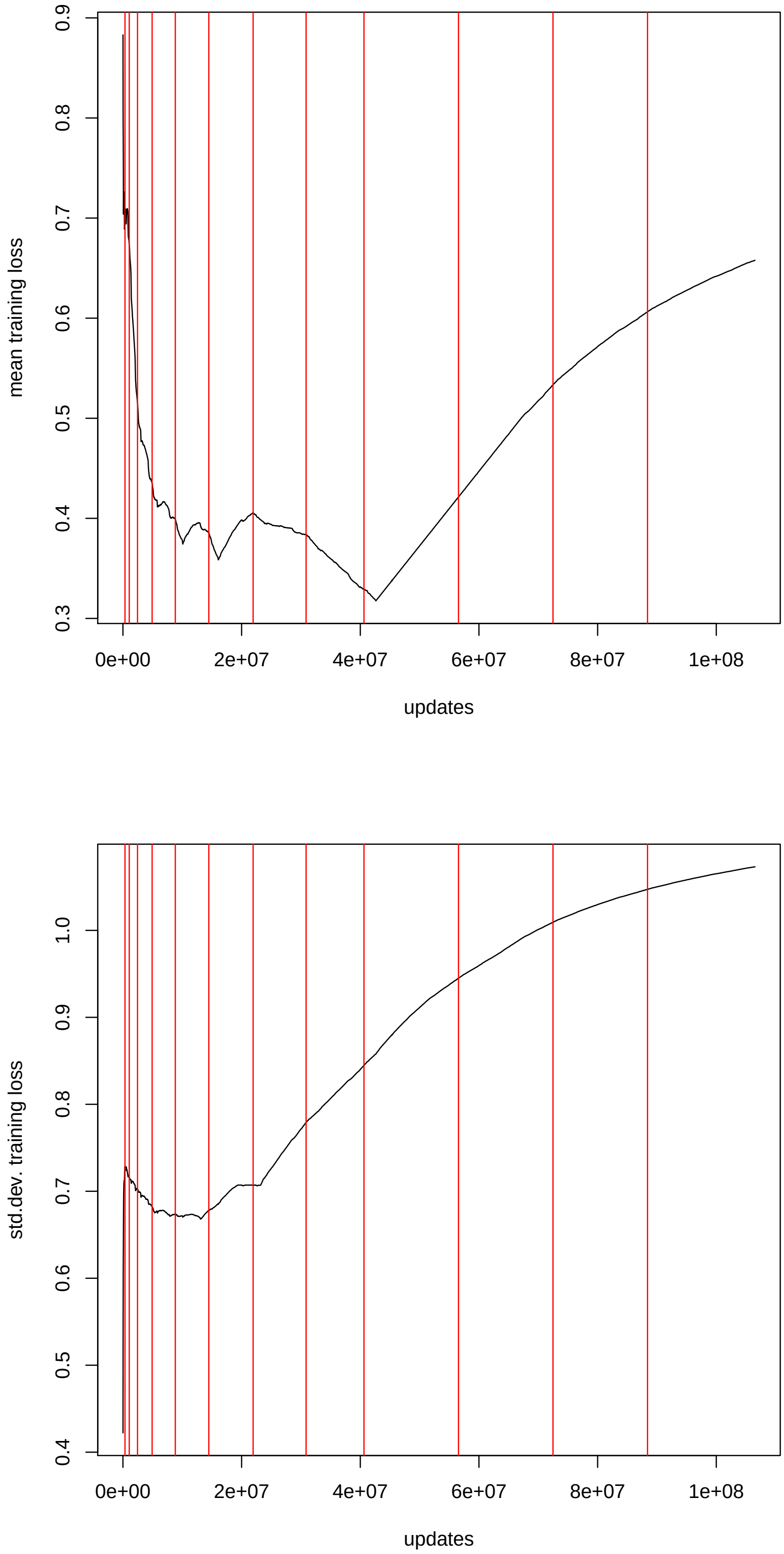
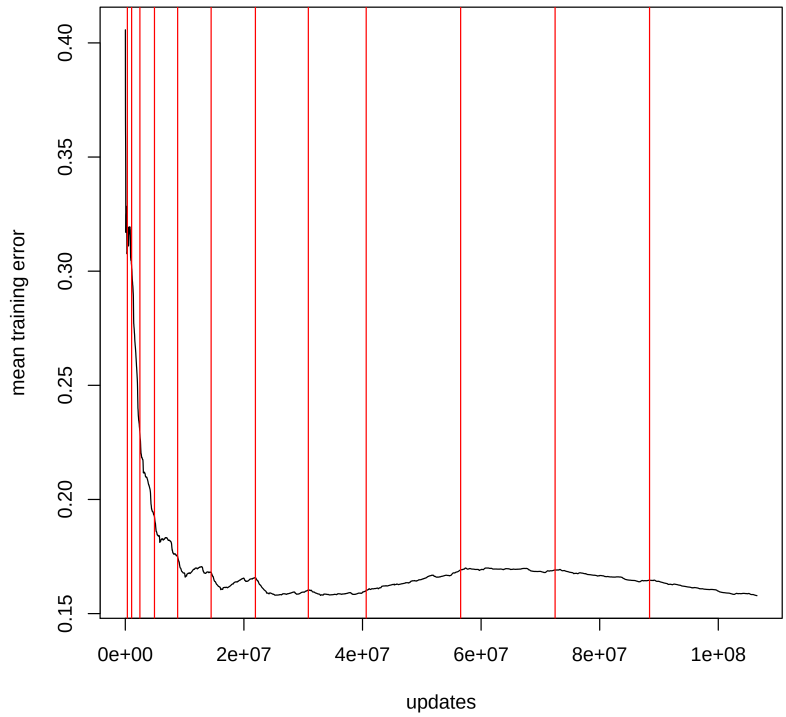


Figure 3: Mean and standard deviation for values of the pairwise ranking cost function.



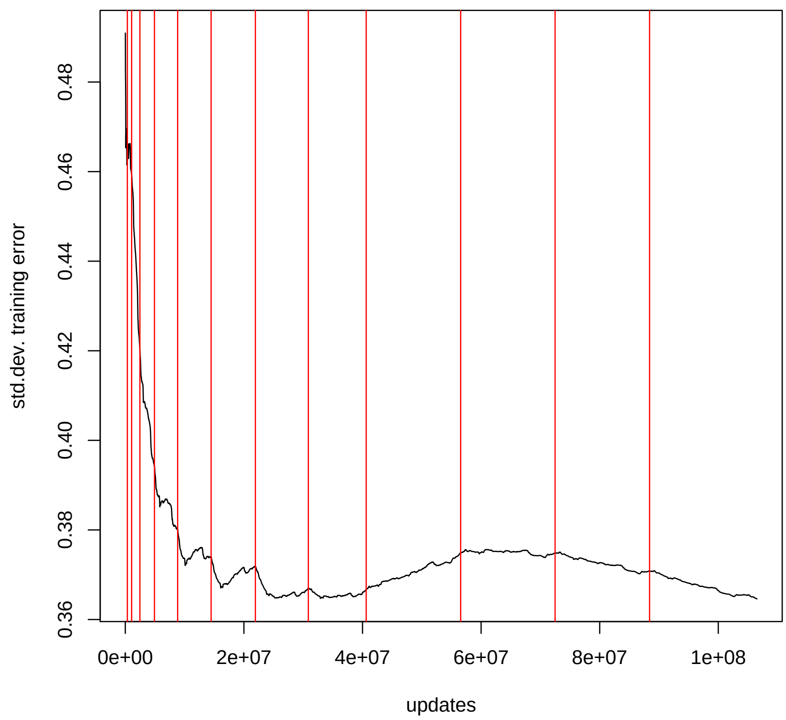


Figure 4: Mean and standard deviation of the number of erroneous scores (i.e. amount of times correct windows were rated lower than corrupted windows).

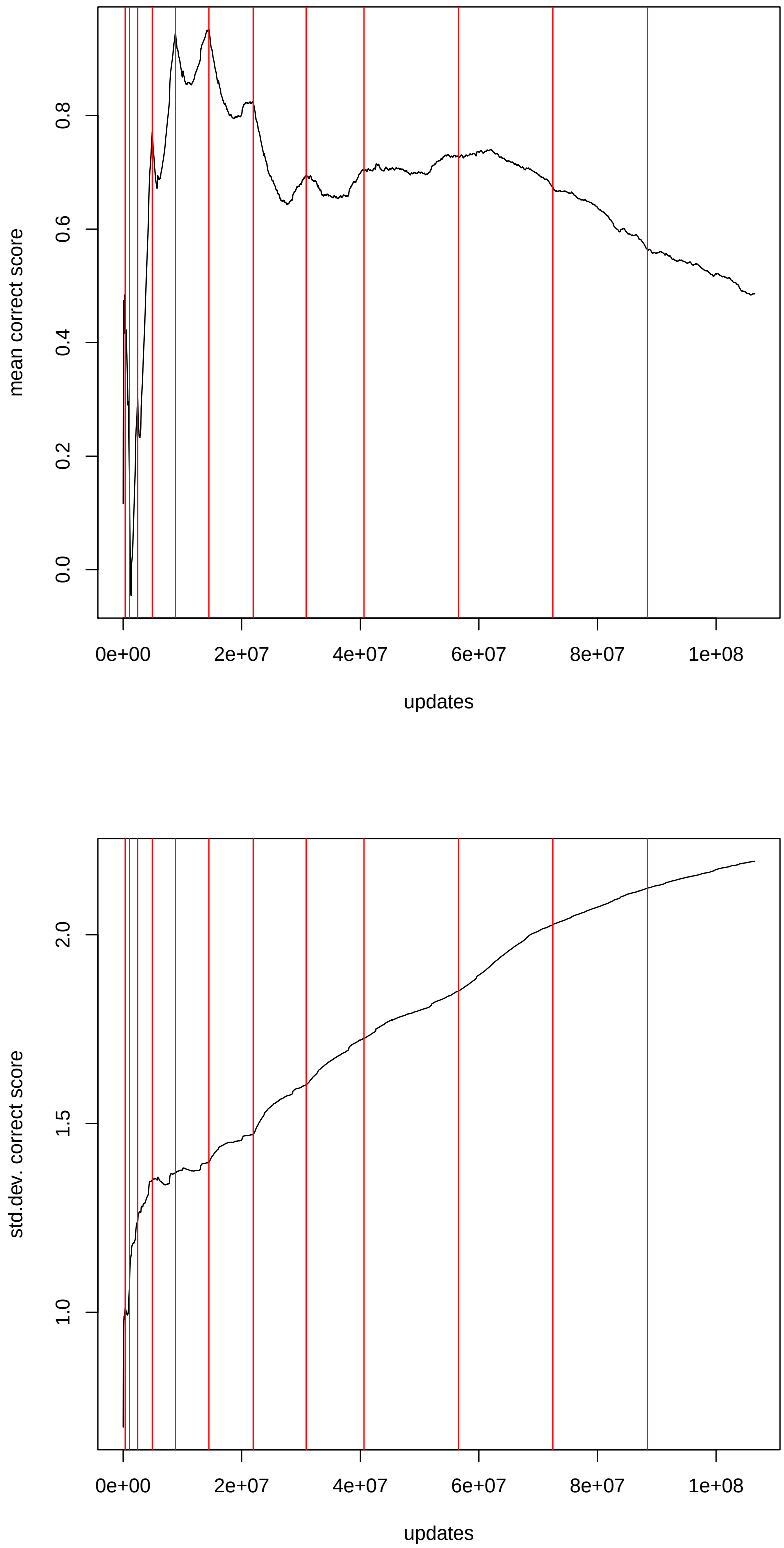


Figure 5: Mean and standard deviation for scores of correct windows.

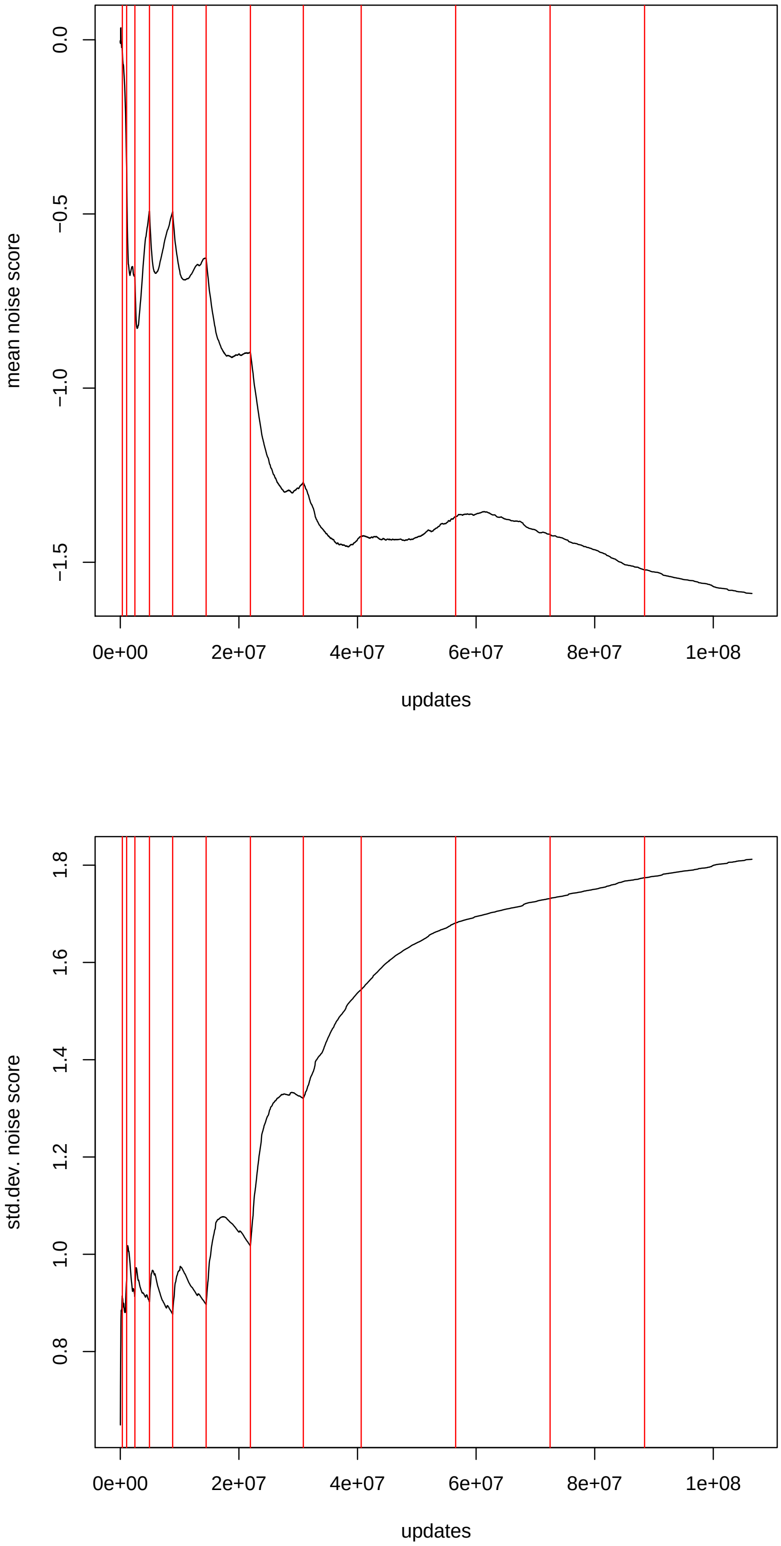


Figure 6: Mean and standard deviation for scores of corrupted windows.

6.2.2 Supervised model

See 7.

7 ASSESSMENT & CONCLUSION

7.1 EVALUATION & HYPOTHESIS

7.1.1 Unsupervised model

7.1.2 Unsupervised model[1]

For unsupervised training, we attained reasonable results as shown in the figures in the previous section. For the first few curricula, scoring accuracy during training steadily increased, but then decreased during the last curriculum, using the full vocabulary. This is due to the fact that this last curriculum contained 300K more words than the penultimate one, all of which needed to be trained. We also integrated sentence padding during this last curriculum, which increases the amount of words to be processed by a large margin.

The progress of training might seem discouraging, but it is good to keep in mind that the average score of a correct window is a good deal higher than that of a corrupted window, and that in general the model evolves positively until the addition of the last curriculum, which is of relatively little importance at this stage because of the rarity of the words it adds.

We only computed select word rankings and were confronted with the fact that most of them were not good. We propose that the trainer needs much more time.

7.1.3 Supervised model

The supervised model is most certainly the weakest part of our architecture. For the sake of simplicity, we did not implement any safeguards for our morphological analyses, and now pay for it: the system assigns tags without too much regard for the rules of Greek grammar. For instance, for the first sentence of our test corpus, not a single tag is correct. Nouns are assigned tenses, moods, etc ... It is evident that

1 We did not have time to compute word rankings. We did pick a few words to test the rankings, but they were not good (generally in the upper part of the spectrum for rankings).

we have not even approached the accuracy rates of Mambrini and Passarotti, 2012 and Dik and Whaling, 2008, 2009. We believe this is in part also due to the relatively small training corpus for the supervised phase.

7.2 CONCLUDING REMARKS

We have given a brief but complete *status quaestionis* of the scholarship on NLP for ancient Greek, something which we did not find anywhere else. We consider that we have a made a modest methodological contribution by attempting to use techniques which as of yet have not been applied to ancient Greek, even though the results are disappointing. Furthermore, the unsupervised technique shows promise if given more time to train. We offer some perspectives on how our work should be improved and continued.

7.3 FURTHER WORK

7.3.1 Improving the language model

Larger training sets

Improving the unsupervised model can be done in two ways: by running the unsupervised modeling phase for more epochs, or by increasing the amount of training data. The first option is certainly feasible at this point, but our training set exhausts most of the available material.

A first option is the corpus of papyri; however, we chose to exclude it from the training set to avoid skewing the results. But the most obvious option is the integral *Thesaurus Linguae Graecae*, which contains more than 109M words but is still not freely available. This is an order of magnitude larger than our current training corpus! This is a huge amount of resources which we can exploit to improve our model.

The main reason for these texts not being published in a downloadable format is that the TLG requires funds to run its servers. However, we still find it hard to justify that such a rich resource for classical scholarship should not be distributed more widely to facilitate further study. While this is no place for an extended philippic in favour of open-sourcing the TLG, we still think a strong case can be presented for this.

The supervised component of our model can also be improved by running more epochs (which we did not have time for), but especially by increasing the size of the training corpus. We opted to only use the PROIEL treebank and not the Perseus treebank to minimise headaches related to the conversion of one annotation standard to another. Especially the syntactic annotation standards used by both, while sharing some common concepts, are very different in implementation.

This could be fixed by developing an effective and accurate system for this type of conversion; this has been attempted in Lee and D. Haug, 2010, but the accuracy rate of that experiment (in the low 80%s for both Latin and Greek) is too low to allow extended use. In any case, it is our opinion that developing a more unified system for ancient Greek treebanks is a *desideratum* before the scale of current treebanks is expanded, lest we end up with two competing standards with none of both to lead us into the fray.

Linguistic foreknowledge

We chose our training method because of its simplicity; but it is a good question whether it is possible to use more linguistic foreknowledge strategically. Collobert, Weston *et al.*, 2011 equip their model with a selective set of simple linguistic features which are tuned to improve performance for certain tasks. For instance, they improve performance for named entity recognition by adding a feature for capitalisation, which is handy given the use of capital letters in English to indicate proper nouns. They also equip their system with a gazetteer, which is a large list of proper nouns.

Similarly, we could recruit supplemental resources to improve performance at our chosen tasks as well as others 7.3.1. A large database of morphological parses exists in the form of the SQL dump of the Perseus word study tool. We could implement a supplemental network which reduces the number of possible parses for each word which is in this database and is trained to pick the correct one; if it is not, we use the unmodified neural architecture to estimate a likely parse.

A similar technique we could use is cascading. Using this technique, we train taggers for diverse tasks which are interconnected; we apply these taggers sequentially and use the outputs of previous taggers for each task. In this way, we can also heavily limit possibilities. For example, by first finding the correct part-of-speech for a word, we can immediately eliminate certain morphological categories. A noun, for instance, has no voice, tense or mood, and developing such sequential tagging method would prevent us from spending valuable computation time on these 'no-brainers'.

Probabilistic parameters

Our model as it stands is actually very simple, a card which we have unsuccessfully tried to play out against the complexity of Greek. We therefore propose to add extra parameters to the model. More specifically, we would like to add supplemental parameters for probabilistic inference on two levels: the level of morphological parsing and the level of sentence structure. As it stands, we compute probabilities of tags being correct for individual word windows, but the main weakness in this approach is that we do not check either the consistency of morphological parses or the likelihood of a given tag sequence.

This could be remedied by developing transition tables for each distinct problem and applying these during tagging. During training, counts of tag transitions would be kept inside these tables and involved in training the parameters. This is an approach also proposed by Collobert, Weston *et al.*, 2011 which we have not implemented due to a lack of time. In this way, we could ensure that our tags are not as nonsensical as they have in our experiment.

Expansion and integration

A key point for the further development of a large-scale infrastructure for ancient Greek annotated corpora is the integration of diverse resources. We find an example of this in the recent merger of several papyrological resources on the web into `papyri.info`. The recently announced Open Philology Project [Crane, 2013] is another good example of this kind of enterprise.

We propose the development of a full NLP stack for corpus annotation. We envision this as a library similar to the Python Natural Language Toolkit [Natural Language Toolkit, 2012], which will offer a diverse range of tools tailored for ancient Greek: tokenizers, POS taggers, chunkers, parsers, training tools, concordance creators ... This tool could then be integrated with other digital classics resources.

Expanding the range of tasks

Another interesting prospect is the expansion of the architecture to a larger array of tasks. The possibilities, are certainly there: Collobert, Weston *et al.*, 2011 explores named entity recognition (*cfr.supra*) and semantic role labeling with noteworthy success. Deep parsing using the same type of embeddings with recurrent neural networks has shown excellent results, as shown in Collobert, 2011, which attains benchmark performance.

All of this is done using large-scale unsupervised learning with a small kernel of supervised training data, the same technique we have applied. The same method would also fit for ancient Greek; but again progress is blocked by a lack of annotated language data.

BIBLIOGRAPHY

Bamman, David and Gregory Crane

2008 "Building a dynamic lexicon from a digital library", in *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, ACM, pp. 11–20. (Cited on p. 23.)

2009 "Computational Linguistics and Classical Lexicography", *Digital Humanities Quarterly*, 3, 1, `http://www.digitalhumanities.org/dhq/vol/3/1/000033.html`. (Cited on p. 23.)

2011 "The Ancient Greek and Latin Dependency Treebanks", in *Language Technology for Cultural Heritage*, ed. by Caroline Sporleder, Antal van den Bosch and Kalliopi Zervanou, Theory and Applications of Natural Language Processing, Springer Berlin Heidelberg, pp. 79–98, ISBN: 978-3-642-20227-8. (Cited on p. 21.)

Bengio, Yoshua, Jérôme Louradour, Ronan Collobert and Jason Weston

2009 "Curriculum learning", in *Proceedings of the 26th annual international conference on machine learning*, ACM, pp. 41–48. (Cited on p. 35.)

Bergstra, James, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley and Yoshua Bengio

2010 "Theano: a CPU and GPU Math Expression Compiler", in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Oral Presentation, Austin, TX. (Cited on p. 46.)

Blackwell, Christopher and Gregory Crane

2009 "Conclusion: Cyberinfrastructure, the Scaife Digital Library and classics in a digital age", *Digital Humanities Quarterly*, 3, 1. (Cited on p. 15.)

Bridle, John S.

1990 "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition", in *Neurocomputing*, Springer, pp. 227–236. (Cited on p. 34.)

Collobert, Ronan

2011 "Deep learning for efficient discriminative parsing", in *International Conference on Artificial Intelligence and Statistics*, pp. 224–232. (Cited on p. 67.)

Collobert, Ronan and Jason Weston

2008 "A unified architecture for natural language processing: deep neural networks with multitask learning", in *Proceedings of the 25th international conference on Machine learning*, ICML '08, ACM, Helsinki, Finland, pp. 160–167, ISBN: 978-1-60558-205-4, DOI: 10.1145/1390156.1390177, http://doi.acm.org/10.1145/1390156.1390177. (Cited on pp. 3, 22, 43.)

Collobert, Ronan, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu and Pavel Kuksa

2011 "Natural Language Processing (Almost) from Scratch", *Journal of Machine Learning Research*, 12 [Aug. 2011], pp. 2493–2537, http://leon.bottou.org/papers/collobert-2011. (Cited on pp. iii, 3, 22, 23, 29, 32, 36, 65, 66.)

Crane, Gregory

1991 "Generating and parsing classical Greek", *Literary and Linguistic Computing*, 6, 4, pp. 243–245. (Cited on p. 15.)

2013 *The Open Philology Project and Humboldt Chair of Digital Humanities at Leipzig*, http://sites.tufts.edu/perseusupdates/2013/04/04/the-open-philology-project-and-humboldt-chair-of-digital-humanities-at-leipzig/. (Cited on p. 66.)

Crönert, Wilhelm

1903 *Memoria Graeca Herculanensis*, Teubner: Leipzig. (Cited on p. 7.)

Deissmann, Adolf

1895 *Bibelstudien*, Marburg: N. G. Elwert. (Cited on p. 7.)

1897 *Neue Bibelstudien*, Marburg: N. G. Elwert. (Cited on p. 7.)

1929 *The N. T. in the Light of Modern Research*, London: Hodder & Stoughton. (Cited on p. 7.)

Dieterich, K.

1898 *Untersuchungen zur Geschichte der griechischen Sprache von der hellenistischen Zeit bis zum 10. Jh. n. Chr.* Leipzig: Teubner. (Cited on p. 7.)

Dik, Helma and Richard Whaling

2008 "Bootstrapping Classical Greek Morphology", in *Digital Humanities*. (Cited on pp. 15–17, 20, 21, 64.)

2009 "Implementing Greek Morphology", in *Digital Humanities*. (Cited on pp. 16, 17, 20, 21, 64.)

Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent and Samy Bengio

2010 "Why Does Unsupervised Pre-training Help Deep Learning?", *J. Mach. Learn. Res.*, 11 [Mar. 2010], pp. 625–660, ISSN: 1532-4435, http://dl.acm.org/citation.cfm?id=1756006.1756025. (Cited on p. 35.)

Evans, Trevor V. and Dirk D. Obbink

2010 *The Language of the Papyri* (eds.), Oxford University Press. (Cited on pp. 4, 8.)

Gignac, Francis Thomas

1976 *A Grammar of the Greek Papyri of the Roman and Byzantine Periods. I. Phonology.* Milano: Goliardica. (Cited on p. 7.)

1981 *A Grammar of the Greek Papyri of the Roman and Byzantine Periods. II. Morphology.* Milano: Goliardica. (Cited on p. 7.)

Haug, Dag Trygve Truslew *et al.*

PROIEL: Pragmatic Resources in Old Indo-European Languages, http://foni.uio.no:3000/, info at http://www.hf.uio.no/ifikk/proiel/. (Cited on pp. 13, 22.)

Haykin, Simon S.

1999 *Neural networks: a comprehensive foundation*, 2nd ed., Prentice Hall, ISBN: 9780132733502, http://books.google.be/books?id=bX4pAQAAMAAJ. (Cited on p. 14.)

Kapsomenos, Stylianos G.

1938 *Voruntersuchungen zu einer Grammatik der Papyri der nachchristlichen Zeit*, Münchener Beiträge zur Papyrusforschung und antiken Rechtsgeschichte, München: C. H. Beck. (Cited on p. 8.)

1957 "Ἔρευναι εἰς τὴν γλῶσσαν τῶν ἑλληνικῶν παπύρων. Σείρα Πρώτη", *EEThess*, vii, pp. 225–372. (Cited on p. 8.)

Katz, Yarden

2012 *Noam Chomsky on Where Artificial Intelligence Went Wrong*, http://www.theatlantic.com/technology/archive/2012/11/noam-chomsky-on-where-artificial-intelligence-went-wrong/261637/. (Cited on p. 13.)

Lee, John

2008 "A nearest-neighbor approach to the automatic analysis of ancient Greek morphology", in *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, Association for Computational Linguistics, pp. 127–134. (Cited on p. 15.)

2010 "Dependency parsing using prosody markers from a parallel text", in *Ninth International Workshop on Treebanks and Linguistic Theories*, p. 127. (Cited on pp. 17, 20.)

Lee, John and Dag Haug

2010 "Porting an Ancient Greek and Latin Treebank.", in 7th *International Conference on Language Resources and Evaluation*, ed. by Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner and Daniel Tapias, European Language Resources Association, ISBN: 2-9517408-6-7, http://dblp.uni-trier.de/db/conf/lrec/lrec2010.html#LeeH10. (Cited on p. 65.)

Ljungvik, Herman

1932 *Beiträge zur Syntax der spätgriechischen Volkssprache.* Vol. 27, Skrifter utgivna av K. Humanistiska Vetenskaps-Samfundet i Uppsala, 3, Uppsala & Leipzig: Humanistiska Vetenskaps-Samfundet. (Cited on p. 8.)

Mambrini, Francesco and Marco Passarotti

2012 "Will a Parser Overtake Achilles? First experiments on parsing the Ancient Greek Dependency Treebank", in *Eleventh International Workshop on Treebanks and Linguistic Theories*, pp. 133–144. (Cited on pp. 16, 17, 20, 64.)

Mandilaras, Basileios G.

1973 *The verb in the Greek non-literary papyri*, Athens: Hellenic Ministry of Culture and Sciences. (Cited on pp. 7, 8.)

Mayser, Edwin

1938 *Grammatik der griechischen Papyri aus der Ptolemäerzeit, mit Einschluss der gleichzeitigen Ostraka und der in Ägypten verfassten Inschriften.* Berlin, Leipzig: De Gruyter. (Cited on p. 7.)

McCarthy, Michael and Anne O'Keeffe

2010 "What are corpora and how have they evolved?", in *The Routledge Handbook of Corpus Linguistics*, ed. by Anne O'Keeffe and Michael McCarthy, London and New York: Routledge. (Cited on pp. 9, 10.)

Mimno, David and Hanna Wallach

2009 "Computational Papyrology (presentation)", in *Media in Transition 6: Stone and Papyrus, Storage and Transmission*, Cambridge, MA. (Cited on p. 24.)

Natural Language Toolkit

2012 *Natural Language Toolkit*, http://www.nltk.org/. (Cited on p. 66.)

Nivre, Joakim, Johan Hall and Jens Nilsson

2006 "Maltparser: A data-driven parser-generator for dependency parsing", in *Proceedings of the LREC*, vol. 6, pp. 2216–2219. (Cited on p. 16.)

Norvig, Peter

2011 *On Chomsky and the two cultures of statistical learning*, http://norvig.com/chomsky.html. (Cited on p. 13.)

Packard, David W

1973 "Computer-assisted morphological analysis of ancient Greek", in *Proceedings of the 5th conference on Computational linguistics-Volume 2*, Association for Computational Linguistics, pp. 343–355. (Cited on pp. 14, 15.)

Palmer, Leonard Robert

1934 "Prolegomena to a Grammar of the post-Ptolemaic Papyri", *Journal of Theological Studies*, xxxv, pp. 170–5. (Cited on p. 8.)

1945 *A Grammar of the post-Ptolemaic Papyri: Accidence and Word-Formation*, London: Oxford University Press. (Cited on p. 8.)

Porter, S. E. and M. B. O'Donnell

2010 "Building and Examining Linguistic Phenomena in a Corpus of Representative Papyri", in *The Language of the Papyri*, Oxford University Press, pp. 287–311. (Cited on p. 8.)

Rubinstein, Reuven Y and Dirk P Kroese

2004 *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*, Springer. (Cited on p. 36.)

Russell, S.J. and P. Norvig

2010 *Artificial Intelligence: A Modern Approach*, 3rd ed., Prentice Hall Series in Artificial Intelligence, Pearson Education/Prentice Hall, ISBN: 9780136042594, http://books.google.be/books?id=8jZBksh-bUMC. (Cited on p. 13.)

Salonius, A. H.

1927 *Zur Sprache der griechischen Papyrusbriefe*, vol. i, Die Quellen, Akademische Buchhandlung. (Cited on p. 8.)

Schmid, Helmut

1994 "Probabilistic Part-of-Speech Tagging Using Decision Trees", in *Proceedings of International Conference on New Methods in Language Processing*. (Cited on p. 17.)

1995 "Improvements in Part-of-Speech Tagging with an Application to German", in *Proceedings of the ACL SIGDAT-Workshop*. (Cited on p. 17.)

Thumb, Albert

1901 *Die griechische Sprache im Zeitalter des Hellenismus: Beiträge zur Geschichte und Beurteilung der KOINH.* Strassburg: Trübner. (Cited on p. 7.)

1906 "Principienfragen der Κοινή -Forschung.", *Neue Jahrbücher für das klassische Altertum*, 17, pp. 246–63. (Cited on p. 7.)

Turian, Joseph, Lev Ratinov and Yoshua Bengio

2010 "Word representations: a simple and general method for semi-supervised learning", in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, pp. 384–394. (Cited on p. 23.)

Turing, Alan Mathison

1950 "Computing Machinery and Intelligence", *Mind*, 59, 236 [Oct. 1950], pp. 433–460. (Cited on p. 12.)

Wasserman, Larry

2003 *All of Statistics: A Concise Course in Statistical Inference*, Springer, ISBN: 0387402721, http://www.worldcat.org/isbn/0387402721. (Cited on p. 13.)

Part IV

Appendix

A SOURCE CODE

For the sake of completeness, we have provided the Python source code used to create the language model in this appendix. A plain-text copy is bundled with this document in compact disc format for archival purposes. Running the model creator without correctly setting up the directory structure and configuration files will simply yield all sorts of errors, so please conserve the directory structure found on the CD if you wish to run it yourself. The source may equally be found on GitHub at https://github.com/sinopeus/thrax.

A.1 UNSUPERVISED TRAINING

A.1.1 Trainer

```
#!/usr/bin/env python

import logging, pickle
from hyperparameters import UnsupervisedHyperparameters as
    Hyperparameters
from lexicon import UnsupervisedCorpus as Corpus
from state import UnsupervisedTrainingState as TrainingState
from lexicon import Dictionary


if __name__ == "__main__":
    hyperparameters = Hyperparameters("language_model.cfg")

    import os.path, os
    # Setting up a log file. This is handy to follow progress
         during
    # the program's execution without resorting to printing
        to stdout.
    logfile = os.path.join(hyperparameters.run_dir,
        hyperparameters.logfile)
```

```
verboselogfile = os.path.join(hyperparameters.run_dir,
    hyperparameters.verboselogfile)
logging.basicConfig(filename=logfile, filemode="w", level
    =logging.DEBUG)
print("Logging to %s, and creating link %s" % (logfile,
    verboselogfile))

try:
    logging.info("Trying to read training state from %s
        ..." % hyperparameters.run_dir)
    filename = os.path.join(hyperparameters.run_dir,
        hyperparameters.statefile)
    with open(filename, 'rb') as f:
        saved_state = pickle.load(f)

    corpus_state, dictionary_state, hyperparameters =
        saved_state[:3]

    training_corpus = Corpus(corpus_state[0],corpus_state
        [1], hyperparameters)
    dictionary = Dictionary(*dictionary_state)

    trainstate = TrainingState(training_corpus=
        training_corpus, dictionary=dictionary,
        hyperparameters=hyperparameters)
    trainstate.__setstate__(saved_state)
    logging.info("Successfully read training state from %
        s. Training may begin." % hyperparameters.run_dir
        )
except IOError:
    logging.info("Failure reading training state from %s.
         Initialising a new state." % hyperparameters.
        run_dir)

    logging.info("Processing training corpus ...")
    training_corpus = Corpus(os.path.join(hyperparameters
        .data_dir, hyperparameters.training_sentences),
        0, hyperparameters)
    logging.info("Training corpus processed, initialising
         dictionary ...")
```

```
        dictionary = Dictionary(os.path.join(hyperparameters.
            data_dir, hyperparameters.dictionary),
            hyperparameters.curriculum_sizes[0])
        logging.info("Dictionary initialised, proceeding with
             training.")

        trainstate = TrainingState(training_corpus=
            training_corpus, dictionary=dictionary,
            hyperparameters=hyperparameters)
        logging.info("State initialised.")


    trainstate.run()
```

A.1.2 Unsupervised language model

```
from unsupervised.parameters import Parameters
from unsupervised.graph import Graph
import math, logging, numpy

class Model:
    def __init__(self, hyperparameters):
        self.hyperparameters = hyperparameters
        self.parameters = Parameters(self.hyperparameters)
        self.trainer = Trainer()
        self.graph = Graph(self.hyperparameters, self.
            parameters)

    def __getstate__(self):
        return (self.hyperparameters, self.parameters, self.
            trainer)

    def __setstate__(self, state):
        (self.hyperparameters, self.parameters, self.trainer)
             = state
        self.graph = Graph(self.hyperparameters, self.
            parameters)

    def corrupt_example(self, e):
        import copy, random
```

```
        e = copy.deepcopy(e)
        pos = - self.hyperparameters.window_size // 2
        mid = e[pos]
        while e[pos] == mid: e[pos] = random.randint(0, self.
            hyperparameters.curriculum_size - 1)
        pr = 1. / self.hyperparameters.curriculum_size
        weight = 1. / pr
        return e, numpy.float32(weight)

    def corrupt_examples(self, correct_sequences):
        return zip(*[self.corrupt_example(e) for e in
            correct_sequences])

    def train(self, correct_sequences):
        noise_sequences, weights = self.corrupt_examples(
            correct_sequences)
        for w in weights: assert w == weights[0]
        learning_rate = self.hyperparameters.learning_rate

        r = self.graph.train(self.parameters.embeds(
            correct_sequences), self.parameters.embeds(
            noise_sequences), numpy.float32(learning_rate *
            weights[0]))
        correct_inputs_gradient, noise_inputs_gradient,
            losses, correct_scores, noise_scores = r

        to_normalize = set()
        for example in range(len(correct_sequences)):
            correct_sequence = correct_sequences[example]
            noise_sequence = noise_sequences[example]
            loss, correct_score, noise_score = losses[example
                ], correct_scores[example], noise_scores[
                example]

            correct_input_gradient = numpy.reshape(
                correct_inputs_gradient[example], (self.
                hyperparameters.window_size, self.
                hyperparameters.embedding_size))
            noise_input_gradient = numpy.reshape(
                noise_inputs_gradient[example], (self.
```

```
            hyperparameters.window_size, self.
            hyperparameters.embedding_size))

        self.trainer.update(loss, correct_score,
            noise_score)

        for w in weights: assert w == weights[0]
        embedding_learning_rate = self.hyperparameters.
            embedding_learning_rate * weights[0]
        if loss == 0:
            for di in correct_input_gradient +
                noise_input_gradient:
                assert (di == 0).all()
        else:
            for (i, di) in zip(correct_sequence,
                correct_input_gradient):
                self.parameters.embeddings[i] -= 1.0 *
                    embedding_learning_rate * di
                to_normalize.add(i)
            for (i, di) in zip(noise_sequence,
                noise_input_gradient):
                self.parameters.embeddings[i] -= 1.0 *
                    embedding_learning_rate * di
                to_normalize.add(i)

        self.parameters.normalize(list(to_normalize))

    def predict(self, sequence):
        (score) = self.graph.predict(self.parameters.embed(
            sequence))
        return score

    def verbose_predict(self, sequence):
        (score, prehidden) = self.graph.verbose_predict(self.
            parameters.embed(sequence))
        return score, prehidden

    def rank(self, sequence, correct_score):
        import copy
        corrupt_sequence = copy.copy(sequence)
        rank = 1
```

```
        mid = self.hyperparameters.window_size // 2

        for i in range(self.hyperparameters.curriculum_size -
             1):
            if i == sequence[mid]: continue
            corrupt_sequence[mid] = i
            corrupt_score = self.predict(corrupt_sequence)
            rank += (correct_score <= corrupt_score)

    def validate(self, sequences):
        correct_scores = self.graph.predict(self.parameters.
            embeds(sequences))
        rank = self.rank
        ranks = [rank(sequence, score) for sequence, score in
             zip(sequences, correct_scores)]
        return ranks

class Trainer:
    """
    We use a trainer to keep track of progress. This is, in
        effect, a
    wrapper object for all kinds of data related to training:
         average
    loss, average error, and a whole host of other variables.
    """
    def __init__(self):
        self.loss = MovingAverage()
        self.err = MovingAverage()
        self.lossnonzero = MovingAverage()
        self.squashloss = MovingAverage()
        self.correct_score = MovingAverage()
        self.noise_score = MovingAverage()
        self.cnt = 0

    def update(self, loss, correct_score, noise_score):
        self.loss.add(loss)
        self.err.add(int(correct_score <= noise_score))
        self.lossnonzero.add(int(loss > 0))
        squashloss = 1. / (1. + math.exp(-loss))
        self.squashloss.add(squashloss)
        self.correct_score.add(correct_score)
```

```
        self.noise_score.add(noise_score)
        self.cnt += 1

        if self.cnt % 10000 == 0: self.update_log()

    def update_log(self):
        logging.info(("After %d updates, pre-update train
            loss %s" % (self.cnt, self.loss.verbose_string())
            ))
        logging.info(("After %d updates, pre-update train
            error %s" % (self.cnt, self.err.verbose_string())
            ))
        logging.info(("After %d updates, pre-update train Pr(
            loss != 0) %s" % (self.cnt, self.lossnonzero.
            verbose_string())))
        logging.info(("After %d updates, pre-update train
            squash(loss) %s" % (self.cnt, self.squashloss.
            verbose_string())))
        logging.info(("After %d updates, pre-update train
            correct score %s" % (self.cnt, self.correct_score
            .verbose_string())))
        logging.info(("After %d updates, pre-update train
            noise score %s" % (self.cnt, self.noise_score.
            verbose_string())))


class MovingAverage:
    def __init__(self, percent=False):
        self.mean = 0.
        self.variance = 0
        self.cnt = 0
        self.percent = percent

    def add(self, v):
        """
        Add value v to the moving average.
        """
        self.cnt += 1
        self.mean = self.mean - (2. / self.cnt) * (self.mean
            - v)
        this_variance = (v - self.mean) * (v - self.mean)
```

```
        self.variance = self.variance - (2. / self.cnt) * (
            self.variance - this_variance)

    def __str__(self):
        if self.percent:
            return "(moving average): mean=%.3f%% stddev=%.3f
                " % (self.mean, math.sqrt(self.variance))
        else:
            return "(moving average): mean=%.3f stddev=%.3f"
                % (self.mean, math.sqrt(self.variance))

    def verbose_string(self):
        if self.percent:
            return "(moving average): mean=%g%% stddev=%g" %
                (self.mean, math.sqrt(self.variance))
        else:
            return "(moving average): mean=%g stddev=%g" % (
                self.mean, math.sqrt(self.variance))
```

A.1.3 Network parameters

```
import numpy, math, theano.configparser
from theano.compile.sharedvalue import shared

theano.config.floatX = 'float32'
floatX = theano.config.floatX

class Parameters:
    def __init__(self, hyperparameters):
        self.hyperparameters = hyperparameters
        numpy.random.seed()

        self.embeddings = numpy.asarray((numpy.random.rand(
            self.hyperparameters.vocab_size, self.
            hyperparameters.embedding_size) - 0.5)* 2 * 0.01,
             dtype=floatX)
        self.hidden_weights = shared(numpy.asarray(
            random_weights(self.hyperparameters.window_size *
             self.hyperparameters.embedding_size, self.
```

```
            hyperparameters.hidden_size, scale_by=1), dtype=
            floatX))
        self.output_weights = shared(numpy.asarray(
            random_weights(self.hyperparameters.hidden_size,
            self.hyperparameters.output_size, scale_by=1),
            dtype=floatX))
        self.hidden_biases = shared(numpy.asarray(numpy.zeros
            ((self.hyperparameters.hidden_size,)), dtype=
            floatX))
        self.output_biases = shared(numpy.asarray(numpy.zeros
            ((self.hyperparameters.output_size,)), dtype=
            floatX))

    def __iter__(self):
        for param in (self.hidden_weights, self.
            output_weights, self.hidden_biases, self.
            output_biases): yield param

    def embed(self, window):
        seq = [self.embeddings[word] for word in window]
        return numpy.hstack([numpy.resize(s, (1, s.size)) for
             s in seq])

    def embeds(self, sequences):
        return numpy.vstack([self.embed(seq) for seq in
            sequences])

    def normalize(self, indices):
        l2norm = numpy.square(self.embeddings[indices]).sum(
            axis=1)
        l2norm = numpy.sqrt(l2norm.reshape((len(indices),1)))
        self.embeddings[indices] /= l2norm
        import math
        self.embeddings[indices] *= math.sqrt(self.embeddings
            .shape[1])

"""
This function was taken straight from the Pylearn library to
    avoid an
extra dependency; the library as a whole is also deprecated.
"""
```

```
sqrt3 = math.sqrt(3.0)
def random_weights(nin, nout, scale_by=1./sqrt3, power=0.5):
    return (numpy.random.rand(nin, nout) * 2.0 - 1) *
        scale_by * sqrt3 / math.pow(nin,power)
```

A.1.4 Network graph

```
"""
Theano graph of Collobert & Weston language model.
Originally written by Joseph Turian, adapted for Python 3 by
    Xavier Go\'as Aguililla.
"""

import theano, logging, numpy
import theano.tensor.basic as t
from theano.gradient import grad

theano.config.floatX = 'float32'
floatX = theano.config.floatX
COMPILE_MODE = "FAST_RUN"

class Graph:
    def __init__(self, hyperparameters, parameters):
        self.hyperparameters = hyperparameters
        self.parameters = parameters
        self.cache = {}

    def score(self, window):
        prehidden = t.dot(window, self.parameters.
            hidden_weights) + self.parameters.hidden_biases
        hidden = t.clip(prehidden, -1, 1)
        score = t.dot(hidden, self.parameters.output_weights)
             + self.parameters.output_biases
        return score, prehidden

    def predict(self, correct_sequence):
        f = self.functions(sequence_length=len(
            correct_sequence))[0]
        return f(correct_sequence)
```

```
def train(self, correct_sequence, noise_sequence,
    learning_rate):
    f = self.functions(sequence_length=len(
        correct_sequence))[1]
    return f(correct_sequence, noise_sequence,
        learning_rate)

def verbose_predict(self, correct_sequence):
    f = self.functions(sequence_length=len(
        correct_sequence))[2]
    return f(correct_sequence)

def functions(self, sequence_length):
    key = (sequence_length)

    if key not in self.cache:
        logging.info("Constructing graph for batches of
            size %s ..." % (sequence_length))

        # creating network input variable nodes
        correct_inputs = t.fmatrix("correct input")
        noise_inputs = t.fmatrix("noise input")
        learning_rate = t.fscalar("learning rate")

        # creating op nodes for firing the network
        correct_score, correct_prehidden = self.score(
            correct_inputs)
        noise_score, noise_prehidden = self.score(
            noise_inputs)

        # creating op nodes for the pairwise ranking cost
             function
        loss = t.clip(1 - correct_score + noise_score, 0,
             1e999)
        total_loss = t.sum(loss)

        # the necessary cost function gradients
        parameters_gradient = grad(total_loss, list(self.
            parameters))
        correct_inputs_gradient = grad(total_loss,
            correct_inputs)
```

```
noise_inputs_gradient = grad(total_loss,
    noise_inputs)

# setting network inputs
predict_inputs = [correct_inputs]
train_inputs = [correct_inputs, noise_inputs,
    learning_rate]
verbose_predict_inputs = predict_inputs

# setting network outputs
predict_outputs = [correct_score]
train_outputs = [correct_inputs_gradient,
    noise_inputs_gradient, loss, correct_score,
    noise_score]
verbose_predict_outputs = [correct_score,
    correct_prehidden]

nnodes = len(theano.gof.graph.ops(predict_inputs,
     predict_outputs))
logging.info("About to compile prediction
    function over %d ops [nodes]..." % nnodes)
predict = theano.function(predict_inputs,
    predict_outputs, mode=COMPILE_MODE)
logging.info("...done constructing graph for
    sequence_length=%d" % (sequence_length))

nnodes = len(theano.gof.graph.ops(
    verbose_predict_inputs,
    verbose_predict_outputs))
logging.info("About to compile verbose prediction
     function over %d ops [nodes]..." % nnodes)
verbose_predict = theano.function(
    verbose_predict_inputs,
    verbose_predict_outputs, mode=COMPILE_MODE)
logging.info("...done constructing graph for
    sequence_length=%d" % (sequence_length))

nnodes = len(theano.gof.graph.ops(train_inputs,
    train_outputs))
logging.info("About to compile training function
    over %d ops [nodes]..." % nnodes)
```

```
            train = theano.function(train_inputs,
                train_outputs, mode=COMPILE_MODE, updates=[(p
                , p - learning_rate * gp) for p, gp in zip(
                list(self.parameters), parameters_gradient)])
            logging.info("...done constructing graph for
                sequence_length=%d" % (sequence_length))

            self.cache[key] = (predict, train,
                verbose_predict)

        return self.cache[key]
```

A.2 SUPERVISED TRAINING

A.2.1 Trainer

```
#!/usr/bin/env python

import logging, pickle
from hyperparameters import SupervisedHyperparameters as
    Hyperparameters
from lexicon import SupervisedCorpus as Corpus
from state import SupervisedTrainingState as TrainingState
from lexicon import Dictionary

if __name__ == "__main__":
    hyperparameters = Hyperparameters("supervised-model.cfg")

    import os.path, os
    logfile = os.path.join(hyperparameters.run_dir,
        hyperparameters.logfile)
    logging.basicConfig(filename=logfile, filemode="w", level
        =logging.DEBUG)
    print("Logging to %s." % logfile)
    from supervised.parameters import SupervisedParameters as
         Parameters
    parameters = Parameters(hyperparameters)

    try:
```

```
        logging.info("Trying to read model information from %
            s..." % hyperparameters.run_dir)
        filename =hyperparameters.paramfile
        import h5py
        hdf = h5py.File(filename,'r+')
        unsup = hdf['unsupervised']
        parameters.embeddings = unsup['embeddings'].value
        from theano.compile.sharedvalue import shared
        parameters.hidden_weights = shared(unsup['
            hidden_weights'].value)
        parameters.hidden_biases = shared(unsup['
            hidden_biases'].value)
    except IOError:
        logging.info("Failed to read initial parameters from
            %s. Supervised training will be less accurate as
            a result." % hyperparameters.run_dir)

    logging.info("Processing training corpus ...")
    training_corpus = Corpus(os.path.join(hyperparameters.
        data_dir, hyperparameters.training_sentences),
        hyperparameters)
    logging.info("Training corpus processed, initialising
        dictionary ...")
    dictionary = Dictionary(os.path.join(hyperparameters.
        data_dir, hyperparameters.dictionary),
        hyperparameters.vocab_size)
    logging.info("Dictionary initialised, proceeding with
        training.")

    trainstate = TrainingState(training_corpus=
        training_corpus, dictionary=dictionary,
        hyperparameters=hyperparameters, parameters=
        parameters)
    logging.info("State initialised.")
    trainstate.run()

    filename = hyperparameters.newparamfile
    hdf = h5py.File(filename,'r+')
    unsup = hdf['unsupervised']
    sup = hdf['supervised']
```

```
    unsup['embeddings'] = trainstate.network.parameters.
        embeddings
    unsup['hidden_weights'] = trainstate.network.parameters.
        hidden_weights.get_value()
    unsup['hidden_biases'] = trainstate.network.parameters.
        hidden_biases.get_value()
    sup.create_group('output_weights')
    sup.create_group('output_biases)
    for task in hyperparameters.task_hash.keys():
        sup['output_weights'][task] = trainstate.network.
            parameters.output_weights[hyperparameters.
            task_hash[task]].get_value()
        sup['output_biases'][task] = trainstate.network.
            parameters.output_biases[hyperparameters.
            task_hash[task]].get_value()
    hdf.close()
```

A.2.2 Supervised language model

```
from supervised.parameters import SupervisedParameters as
    Parameters
from supervised.graph import Graph
import math, logging, numpy

class Network:
    def __init__(self, hyperparameters, parameters):
        self.hyperparameters = hyperparameters
        self.parameters = parameters
        self.graph = Graph(self.hyperparameters, self.
            parameters)

    def train(self, sequences):
        tuples = [list(x) for x in [zip(*seq) for seq in
            sequences]]
        seqs = [example[0]for example in tuples]
        correct_outputs = [example[1] for example in tuples]
        correct_outputs = [tag[self.hyperparameters.
            window_size // 2] for tag in correct_outputs]
        correct_outputs = [self.vectorise(x) for x in
            correct_outputs]
```

```
        correct_outputs = [numpy.vstack([correct_outputs[
            example][task] for example in range(len(
            correct_outputs))]) for task in range(len(self.
            hyperparameters.tasks))]

        learning_rate = self.hyperparameters.learning_rate
        embedding_learning_rate = self.hyperparameters.
            embedding_learning_rate
        embeds = self.parameters.embeds(seqs)

        for task in self.hyperparameters.task_hash.keys():
            r = self.graph.train(embeds, correct_outputs[self
                .hyperparameters.task_hash[task]], task,
                learning_rate)

            embeddings_gradients = r[0]

            to_normalize = set()
            for example in range(len(sequences)):
                sequence = seqs[example]
                embeddings_gradient = numpy.reshape(
                    embeddings_gradients[example], (self.
                    hyperparameters.window_size, self.
                    hyperparameters.embedding_size))

                for (i, di) in zip(sequence,
                    embeddings_gradient):
                    self.parameters.embeddings[i] += 1.0 *
                        embedding_learning_rate * di
                    to_normalize.add(i)

                self.parameters.normalize(list(to_normalize))

    def vectorise(self, outputs):
        vectors = [numpy.zeros(len(self.hyperparameters.tasks
            [i]), dtype='float32') for i in range(len(self.
            hyperparameters.tasks))]
        for i, component in enumerate(outputs): vectors[i][
            component] = 1
        return [numpy.resize(v, (1,v.size)) for v in vectors]
```

```
    def predict(self, sequence, task):
        (score) = self.graph.predict(self.parameters.embed(
            sequence),task)
        return score

    def verbose_predict(self, sequence):
        (score, prehidden) = self.graph.verbose_predict(self.
            parameters.embed(sequence),task)
        return score, prehidden
```

A.2.3 Network parameters

```
import numpy, math, theano
from theano.compile.sharedvalue import shared

theano.config.floatX = 'float32'
floatX = theano.config.floatX

class SupervisedParameters:
    def __init__(self, hyperparameters):
        self.hyperparameters = hyperparameters

        numpy.random.seed()

        self.embeddings = numpy.asarray((numpy.random.rand(
            self.hyperparameters.vocab_size, self.
            hyperparameters.embedding_size) - 0.5)* 2 * 0.01,
             dtype=floatX)
        self.hidden_weights = shared(numpy.asarray(
            random_weights(self.hyperparameters.window_size *
             self.hyperparameters.embedding_size, self.
            hyperparameters.hidden_size, scale_by=1), dtype=
            floatX))
        self.hidden_biases = shared(numpy.asarray(numpy.zeros
            ((self.hyperparameters.hidden_size,)), dtype=
            floatX))

        self.output_weights = []
        self.output_biases = []
```

```
        tasks = self.hyperparameters.tasks
        for i in range(len(tasks)):
            tagset_size =  len(tasks[i])
            self.output_weights.append(shared(numpy.asarray(
                random_weights(self.hyperparameters.
                hidden_size, tagset_size, scale_by=1), dtype=
                floatX)))
            self.output_biases.append(shared(numpy.asarray(
                numpy.zeros((tagset_size,)), dtype=floatX)))

    def get(self, task):
        return (self.hidden_weights, self.output_weights[self
            .hyperparameters.task_hash[task]], self.
            hidden_biases, self.output_biases[self.
            hyperparameters.task_hash[task]])

    def embed(self, window):
        seq = [self.embeddings[word] for word in window]
        return numpy.hstack([numpy.resize(s, (1, s.size)) for
             s in seq])

    def embeds(self, sequences):
        return numpy.vstack(map(self.embed, sequences))

    def normalize(self, indices):
        l2norm = numpy.square(self.embeddings[indices]).sum(
            axis=1)
        l2norm = numpy.sqrt(l2norm.reshape((len(indices),1)))
        self.embeddings[indices] /= l2norm
        import math
        self.embeddings[indices] *= math.sqrt(self.embeddings
            .shape[1])

"""
This function was taken straight from the Pylearn library to
    avoid an
extra dependency; the library as a whole is also deprecated.
"""
sqrt3 = math.sqrt(3.0)
def random_weights(nin, nout, scale_by=1./sqrt3, power=0.5):
```

```
    return (numpy.random.rand(nin, nout) * 2.0 - 1) *
        scale_by * sqrt3 / math.pow(nin,power)
```

A.2.4 Network graph

```
"""
Theano graph of Collobert & Weston language model.
Originally written by Joseph Turian, adapted for Python 3 by
    Xavier Go\'as Aguililla.
"""

import theano, logging, numpy
import theano.tensor.basic as t
from theano.gradient import grad

theano.config.floatX = 'float32'
floatX = theano.config.floatX
COMPILE_MODE = "FAST_RUN"

class Graph:
    def __init__(self, hyperparameters, parameters):
        self.hyperparameters = hyperparameters
        self.task_hash, self.tasks = self.hyperparameters.
            task_hash, self.hyperparameters.tasks
        self.parameters = parameters
        self.cache = {}

    def score(self, window, task):
        prehidden = t.dot(window, self.parameters.
            hidden_weights) + self.parameters.hidden_biases
        hidden = t.clip(prehidden, -1, 1)
        score = t.dot(hidden, self.parameters.output_weights[
            self.task_hash[task]]) + self.parameters.
            output_biases[self.task_hash[task]]
        return score, prehidden

    def predict(self, sequence,task):
        f = self.functions(sequence_length=len(sequence),task
            =task)[0]
        return f(sequence, task)
```

```
def train(self, sequence, correct_outputs, task,
    learning_rate):
    f = self.functions(sequence_length=len(sequence),task
        =task)[1]
    return f(sequence, correct_outputs, learning_rate)

def verbose_predict(self, sequence,task):
    f = self.functions(sequence_length=len(sequence),task
        =task)[2]
    return f(sequence)

def functions(self, sequence_length, task):
    key = (sequence_length, task)

    if key not in self.cache:
        logging.info("Constructing graph for batches of
            size %s ... for task %s" % (sequence_length,
            task))

        # creating network input variable nodes
        inputs = t.fmatrix("input")
        correct_outputs = t.fmatrix("correct output")
        learning_rate = t.fscalar("learning rate")

        # creating op nodes for firing the network
        outputs, prehidden = self.score(inputs, task)

        correct_tags = t.argmax(correct_outputs, axis=0)
        softmax_outputs = theano.tensor.nnet.softmax(
            outputs)
        argmaxes = outputs[correct_tags]

        total_log_likelihood = t.sum(t.log(argmaxes))

        # the necessary cost function gradients
        parameters_gradient = grad(total_log_likelihood,
            self.parameters.get(task))
        embeddings_gradient = grad(total_log_likelihood,
            inputs)
```

```
# setting network inputs
predict_inputs = [inputs]
train_inputs = [inputs, correct_outputs,
    learning_rate]
verbose_predict_inputs = predict_inputs

# setting network outputs
predict_outputs = [softmax_outputs]
train_outputs = [embeddings_gradient]
verbose_predict_outputs = [outputs, prehidden]

nnodes = len(theano.gof.graph.ops(predict_inputs,
     predict_outputs))
logging.info("About to compile prediction
    function over %d ops [nodes]..." % nnodes)
predict = theano.function(predict_inputs,
    predict_outputs, mode=COMPILE_MODE)
logging.info("...done constructing graph for
    sequence_length=%d" % (sequence_length))

nnodes = len(theano.gof.graph.ops(
    verbose_predict_inputs,
    verbose_predict_outputs))
logging.info("About to compile verbose prediction
     function over %d ops [nodes]..." % nnodes)
verbose_predict = theano.function(
    verbose_predict_inputs,
    verbose_predict_outputs, mode=COMPILE_MODE)
logging.info("...done constructing graph for
    sequence_length=%d" % (sequence_length))

nnodes = len(theano.gof.graph.ops(train_inputs,
    train_outputs))
logging.info("About to compile training function
    over %d ops [nodes]..." % nnodes)
train = theano.function(train_inputs,
    train_outputs, mode=COMPILE_MODE, updates=[(p
    , p + learning_rate * gp) for p, gp in zip(
    self.parameters.get(task),
    parameters_gradient)])
```

```
            logging.info("...done constructing graph for
                sequence_length=%d" % (sequence_length))

            self.cache[key] = (predict, train,
                verbose_predict)

        return self.cache[key]
```

A.3 TAGGING

A.3.1 Tagger

```
#!/usr/bin/env python

import logging, pickle,sys, os, numpy
from hyperparameters import SupervisedHyperparameters as
    Hyperparameters
from lexicon import ObjectCorpus as Corpus
from lexicon import Dictionary

if __name__ == "__main__":
    hyperparameters = Hyperparameters("supervised-model.cfg")

    from supervised.parameters import SupervisedParameters as
        Parameters
    parameters = Parameters(hyperparameters)

    try:
        logging.info("Trying to read model information from %
            s..." % hyperparameters.run_dir)
        filename = '/Users/xavier/projects/thrax/run/
            supervised/params.hdf5'
        import h5py, copy
        hdf = h5py.File(filename,'r')
        unsup = hdf['shared']
        parameters.embeddings = numpy.asarray(unsup['
            embeddings'].value)
        from theano.compile.sharedvalue import shared
```

```
        parameters.hidden_weights = shared(numpy.asarray(
            unsup['hidden_weights'].value))
        parameters.hidden_biases = shared(numpy.asarray(unsup
            ['hidden_biases'].value))
        sup = hdf['individual']
        for task in hyperparameters.task_hash.keys():
            parameters.output_weights[hyperparameters.
                task_hash[task]] = shared(numpy.asarray(sup['
                output_weights'][task]))
            parameters.output_biases[hyperparameters.
                task_hash[task]] = shared(numpy.asarray(sup['
                output_biases'][task]))
        hdf.close()
    except IOError:
        print("Failed to read parameters from %s. Exiting." %
             hyperparameters.run_dir)
        sys.exit()

    logging.info("Processing training corpus ...")
    corpus = Corpus(sys.argv[1], hyperparameters=
        hyperparameters)
    logging.info("Training corpus processed, initialising
        dictionary ...")
    dictionary = Dictionary(os.path.join(hyperparameters.
        data_dir, hyperparameters.dictionary),
        hyperparameters.vocab_size)
    logging.info("Dictionary initialised, proceeding with
        training.")

    from tagging.tagger import Tagger
    tagger = Tagger(corpus=corpus, dictionary=dictionary,
        hyperparameters=hyperparameters, parameters=
        parameters)
    logging.info("State initialised.")
    tagger.run()
    hdf.close()
```

```
class Tagger:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
        from tagging.nn import Network
```

```
        self.network = Network(self.hyperparameters, self.
            parameters)
        from examples import TaggingStream
        self.stream = TaggingStream(corpus=self.corpus,
            dictionary=self.dictionary, hyperparameters=self.
            hyperparameters)

    def run(self):
        from lexicon import ReverseDictionary
        inv = ReverseDictionary(self.dictionary)
        for sentence in iter(self.stream):
            for window in sentence:
                print(window[self.hyperparameters.window_size
                     // 2][0], inv.lookup(window[self.
                    hyperparameters.window_size // 2][1]) ,'|
                    '.join([self.network.predict(window, task
                    ) for task in self.hyperparameters.
                    task_hash.keys()]))
            print('\n')
```

A.3.2 Tagging network

```
from supervised.parameters import SupervisedParameters as
    Parameters
from tagging.graph import Graph
import math, logging, numpy

class Network:
    def __init__(self, hyperparameters, parameters):
        self.hyperparameters = hyperparameters
        self.parameters = parameters
        self.graph = Graph(self.hyperparameters, self.
            parameters)

    def predict(self, sequence, task):
        pos, words = zip(*sequence)
        score = self.graph.predict(self.parameters.embed(
            words),task)
```

```
        return task + self.hyperparameters.inv_tasks[self.
            hyperparameters.task_hash[task]][numpy.argmax(
            score)]

    def verbose_predict(self, sequence):
        (score, prehidden) = self.graph.verbose_predict(self.
            parameters.embed(sequence),task)
        return score, prehidden
```

A.3.3 Network graph

```
"""
Theano graph of Collobert & Weston language model.
Originally written by Joseph Turian, adapted for Python 3 by
    Xavier Go\'as Aguililla.
"""

import theano, logging, numpy
import theano.tensor.basic as t
from theano.gradient import grad

theano.config.floatX = 'float32'
floatX = theano.config.floatX
COMPILE_MODE = "FAST_RUN"

class Graph:
    def __init__(self, hyperparameters, parameters):
        self.hyperparameters = hyperparameters
        self.task_hash, self.tasks = self.hyperparameters.
            task_hash, self.hyperparameters.tasks
        self.parameters = parameters
        self.cache = {}

    def score(self, window, task):
        prehidden = t.dot(window, self.parameters.
            hidden_weights) + self.parameters.hidden_biases
        hidden = t.clip(prehidden, -1, 1)
        score = t.dot(hidden, self.parameters.output_weights[
            self.task_hash[task]]) + self.parameters.
            output_biases[self.task_hash[task]]
```

```
        return score, prehidden

    def predict(self, sequence,task):
        f = self.functions(sequence_length=len(sequence),task
            =task)[0]
        return f(sequence)

    def verbose_predict(self, sequence,task):
        f = self.functions(sequence_length=len(sequence),task
            =task)[1]
        return f(sequence)

    def functions(self, sequence_length, task):
        key = (sequence_length, task)

        if key not in self.cache:
            logging.info("Constructing graph for batches of
                size %s ... for task %s" % (sequence_length,
                task))

            # creating network input variable nodes
            inputs = t.fmatrix("input")

            # creating op nodes for firing the network
            outputs, prehidden = self.score(inputs, task)

            softmax_outputs = theano.tensor.nnet.softmax(
                outputs)

            # setting network inputs
            predict_inputs = [inputs]
            verbose_predict_inputs = predict_inputs

            # setting network outputs
            predict_outputs = [softmax_outputs]
            verbose_predict_outputs = [outputs, prehidden]

            nnodes = len(theano.gof.graph.ops(predict_inputs,
                 predict_outputs))
            logging.info("About to compile prediction
                function over %d ops [nodes]..." % nnodes)
```

```
            predict = theano.function(predict_inputs,
                predict_outputs, mode=COMPILE_MODE)
            logging.info("...done constructing graph for
                sequence_length=%d" % (sequence_length))

            nnodes = len(theano.gof.graph.ops(
                verbose_predict_inputs,
                verbose_predict_outputs))
            logging.info("About to compile verbose prediction
                 function over %d ops [nodes]..." % nnodes)
            verbose_predict = theano.function(
                verbose_predict_inputs,
                verbose_predict_outputs, mode=COMPILE_MODE)
            logging.info("...done constructing graph for
                sequence_length=%d" % (sequence_length))

            self.cache[key] = (predict, verbose_predict)

        return self.cache[key]
```

A.4 DATA STRUCTURES & STREAM GENERATORS

A.4.1 Training state keepers

```
import logging, os.path, pickle, validate, multiprocessing

class UnsupervisedTrainingState:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

        logging.info("Initializing training state.")

        from unsupervised.model import Model
        self.model = Model(self.hyperparameters)

        self.count, self.epoch = (0, 1)

        self.curriculum_phase = 0
```

```
        from lexicon import UnsupervisedCorpus as Corpus
        from validate import Validator

        logging.info("Processing validation corpus ...")
        self.validation_corpus = Corpus(os.path.join(self.
            hyperparameters.data_dir, self.hyperparameters.
            validation_sentences),0,self.hyperparameters)
        logging.info("Validation corpus processed.")

        logging.info("Initialising model validator...")
        self.validator = Validator(corpus=self.
            validation_corpus, dictionary=self.dictionary,
            hyperparameters=self.hyperparameters, model=self.
            model)
        logging.info("Model validator initialised.")

        from examples import ExampleStream, BatchStream

        logging.info("Initialising text window stream...")
        self.examples = ExampleStream(corpus=self.
            training_corpus, dictionary=self.dictionary,
            hyperparameters=self.hyperparameters)
        logging.info("Text window stream initialised.")

        logging.info("Initialising batch stream...")
        self.batches = BatchStream(self.examples)
        logging.info("Batch stream initialised.")

    def run(self):
        while True:
            if self.curriculum_phase < len(self.
                hyperparameters.curriculum_sizes):
                self.curriculum_size = self.hyperparameters.
                    curriculum_sizes[self.curriculum_phase]
                self.hyperparameters.curriculum_size = self.
                    curriculum_size
                logging.info("Resizing dictionary ... ")
                self.dictionary.enlarge(self.curriculum_size)
                logging.info("Resized dictionary to size %s."
                     % self.curriculum_size)
```

```
            logging.info("Initialising curriculum phase %
                i." % self.curriculum_phase)
            self.run_epoch()
            self.curriculum_phase += 1
        else:
            self.dictionary.enlarge(self.hyperparameters.
                vocab_size)
            logging.info("Initialising final curriculum
                phase %i. This phase will go on
                indefinitely." % self.curriculum_phase)
            self.run_epoch()

def run_epoch(self):
    logging.info("Starting epoch #%d." % self.epoch)

    for batch in self.batches:
        self.process(batch)

    self.epoch += 1

    logging.info("Finished epoch #%d. Rewinding training
        stream." % self.epoch)

    self.training_corpus.rewind()

    from examples import ExampleStream, BatchStream
    self.examples = ExampleStream(corpus=self.
        training_corpus, dictionary=self.dictionary,
        hyperparameters=self.hyperparameters)
    self.batches = BatchStream(self.examples)

def process(self, batch):
    self.count += len(batch)
    self.model.train(batch)

    if self.count % (int(1000. / self.hyperparameters.
        batch_size) * self.hyperparameters.batch_size) ==
         0:
        logging.info("Finished training step %d (epoch %d
            )" % (self.count, self.epoch))
```

```
        if self.count % (int( self.hyperparameters.save_every
             * 1./self.hyperparameters.batch_size ) * self.
            hyperparameters.batch_size) == 0:
            self.save()

#        if self.count % (int( self.hyperparameters.
    validate_every * 1./self.hyperparameters.batch_size ) *
    self.hyperparameters.batch_size) == 0:
#            self.validator.validate(self.count)

    def save(self):
        filename = os.path.join(self.hyperparameters.run_dir,
             self.hyperparameters.statefile)
        logging.info("Trying to save training state to %s..."
             % filename)
        with open(filename, 'wb') as f:
            pickle.dump(self.__getstate__(), f)

    def __getstate__(self):
        return (self.training_corpus.__getstate__(), self.
            dictionary.__getstate__(), self.hyperparameters,
            self.model.__getstate__(), self.count, self.epoch
            , self.examples.__getstate__(), self.
            curriculum_phase)

    def __setstate__(self, state):
        from unsupervised.model import Model
        self.model = Model(self.hyperparameters)
        self.model.__setstate__(state[4])
        self.count, self.epoch = state[-4:-2]
        self.examples.__setstate__(state[-2])
        from examples import BatchStream
        self.batches = BatchStream(self.examples)
        self.curriculum_phase = state[-1]

class SupervisedTrainingState:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

        self.count, self.epoch = 0, 1
```

```
        from supervised.nn import Network

        logging.info("Initialising network...")
        if self.parameters != None:
            self.network = Network(self.hyperparameters, self
                .parameters)
        else:
            from unsupervised.parameters import Parameters
            self.network = Network(self.hyperparameters,
                Parameters(self.hyperparameters))
        logging.info("Network initialised...")

        from examples import AnnotatedExampleStream as
            ExampleStream
        from examples import BatchStream

        logging.info("Initialising text window stream...")
        self.examples = ExampleStream(corpus=self.
            training_corpus, dictionary=self.dictionary,
            hyperparameters=self.hyperparameters)
        logging.info("Text window stream initialised.")

        logging.info("Initialising batch stream...")
        self.batches = BatchStream(self.examples)
        logging.info("Batch stream initialised.")

    def run(self):
        # while True:
        self.run_epoch()

    def run_epoch(self):
        for batch in self.batches: self.process(batch)
        # self.epoch +=1

        # logging.info("Finished epoch #%d. Rewinding
            training stream." % self.epoch)

        # self.training_corpus.rewind()

        # from examples import ExampleStream, BatchStream
```

```
        # self.examples = ExampleStream(corpus=self.
            training_corpus, dictionary=self.dictionary,
            hyperparameters=self.hyperparameters)
        # self.batches = BatchStream(self.examples)

    def process(self, batch):
        self.network.train(batch)

        self.count += len(batch)

        if self.count % (int(1000. / self.hyperparameters.
            batch_size) * self.hyperparameters.batch_size) ==
             0:
            logging.info("Analysed %d windows (epoch %e)" % (
                self.count, self.epoch))

        if self.count % (int( self.hyperparameters.save_every
             * 1./self.hyperparameters.batch_size ) * self.
            hyperparameters.batch_size) == 0:
            self.save()

    def save(self):
        filename = os.path.join(self.hyperparameters.run_dir,
             self.hyperparameters.supervisedmodel)
        logging.info("Trying to save parameters to %s..." %
            filename)
        with open(filename, 'wb') as f:
            pickle.dump(self.network.parameters, f)
```

A.4.2 Training example stream

```
import logging
from collections import Iterator

class ExampleStream(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
        self.count = 0

    def __iter__(self):
```

```
        self.count = 0

        for sentence in self.corpus:
            prevwords = []
            for word in sentence:
                if self.dictionary.contains(word):
                    prevwords.append(self.dictionary.lookup(
                        word))
                    if len(prevwords) >= self.hyperparameters
                        .window_size:
                        self.count += 1
                        yield prevwords[-self.hyperparameters
                            .window_size:]
                else:
                    prevwords = []

    def __getstate__(self):
        return self.count

    def __setstate__(self, count):
        logging.info("Fast-forwarding example stream to text
            window %s..." % count)
        iterator = self.__iter__()
        while count != self.count:
            next(iterator)
        logging.info("Back at text window %s." % count)

class AnnotatedExampleStream(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __iter__(self):
        for sentence in self.corpus:
            prevwords = []
            for word in sentence:
                if self.dictionary.contains(word[0]):
                    prevwords.append((self.dictionary.lookup(
                        word[0]), word[1]))
                    if len(prevwords) >= self.hyperparameters
                        .window_size:
```

```
                        yield prevwords[-self.hyperparameters
                            .window_size:]
                else:
                    prevwords = []

class TaggingStream(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __iter__(self):
        for sentence in iter(self.corpus):
            windows = []
            prevwords = []
            for word in sentence:
                if self.dictionary.contains(word[1]):
                    prevwords.append((int(word[0]),self.
                        dictionary.lookup(word[1])))
                    if len(prevwords) >= self.hyperparameters
                        .window_size:
                        windows.append(prevwords[-self.
                            hyperparameters.window_size:])
                else:
                    prevwords.append((int(word[0]),self.
                        dictionary.lookup('PADDING')))
            yield windows

class BatchStream(Iterator):
    def __init__(self, stream):
        self.stream = iter(stream)
        self.hyperparameters = stream.hyperparameters

    def __iter__(self):
        return self

    def __next__(self):
        batch = []
        while len(batch) < self.hyperparameters.batch_size:
            batch.append(next(self.stream))
        return batch
```

A.4.3 Model validator

```
import numpy, logging

from examples import BatchStream

class Validator:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
        self.stream = ValidationStream(**self.__dict__)
        self.batches = BatchStream(self.stream)
        import pdb
        pdb.set_trace()

    def validate(self, cnt):
        logging.info("Setting validation parameters.")
        self.corpus.rewind()
        self.stream = ValidationStream(**self.__dict__)
        logging.info("Beginning validation at training step %
            d. This will take a while." % cnt)

        # generator = (validate(batch) for batch in self.
            batches)
        # logranks = [append(lgr) for lgr in [log(score) for
            score in [validate(window) for window in self]]]
            # list comprehension magic
        ranks = []

        for batch in self.batches:
            ranks += self.model.validate(batch)
            logging.info("Computed %d rankings." % len(ranks)
                )

        num_ranks = numpy.array(ranks)
        log_ranks = numpy.log(ranks)
        logging.info("Validation at training step %d: mean(
            logrank) = %.2f, stddev(logrank) = %.2f, cnt = %d
            ", (cnt, numpy.mean(log_ranks), numpy.std(
            log_ranks), len(log_ranks)))

class ValidationStream:
```

```
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __iter__(self):
        for sentence in self.corpus:
            prevwords = []
            for word in sentence:
                if self.dictionary.contains(word):
                    prevwords.append(self.dictionary.lookup(
                        word))
                    if len(prevwords) >= self.hyperparameters
                        .window_size:
                        yield prevwords[-self.hyperparameters
                            .window_size:]
                else:
                    prevwords = []
```

A.4.4 Dictionary & corpus

```
import re
from collections import Iterator
class UnsupervisedCorpus:
  def __init__(self, corpus_file, count, hyperparameters):
    self.text = open(corpus_file)
    self.count = 0
    self.hyperparameters = hyperparameters
    if count != None:
        self.__setstate__(count)
    self.padding = ["PADDING"] * (self.hyperparameters.
        window_size // 2)

  def __iter__(self):
    self.count = 0
    sentence = []
    for word in self.text:
        if not word.strip():
            self.count += 1
            yield self.pad(sentence)
            sentence = []
        else:
```

```
          sentence.append(word.strip())

  def rewind(self):
    self.text.seek(0)

  def pad(self,sent):
      return self.padding + sent + self.padding

  def __getstate__(self):
    return (self.text.name, self.count)

  def __setstate__(self,count):
    iterator = self.__iter__()
    while self.count < count:
        next(iterator)

class SupervisedCorpus:
  def __init__(self, corpus_file, hyperparameters):
    self.text = open(corpus_file)
    self.hyperparameters = hyperparameters
    self.tasks = self.hyperparameters.tasks
    self.task_hash = self.hyperparameters.task_hash
    self.padding = [("PADDING",  [0] * len(self.tasks))] * (
        self.hyperparameters.window_size // 2)

  def __iter__(self):
    sentence = []
    for word in self.text:
        if not word.strip():
            yield self.pad(sentence)
            sentence = []
        else:
            sentence.append(self.parse_tags(word))

  def parse_tags(self, line):
      line = line.strip().split()
      word = line[1]
      parse = [0] * len(self.tasks)

      # arguably inelegant
```

```
            parse[self.task_hash['POS']] = self.tasks[self.
                task_hash['POS']][line[4]]
            parse[self.task_hash['SYNT']] = self.tasks[self.
                task_hash['SYNT']][line[7]]
            rawparse = line[5].split('|')
            for comp in rawparse:
                parse[self.task_hash[comp[:-1]]] = self.tasks[self.
                    task_hash[comp[:-1]]][comp[-1]]

            return (word, parse)

    def rewind(self):
        self.text.seek(0)

    def pad(self,sent):
            return self.padding + sent + self.padding

class ObjectCorpus:
    def __init__(self, corpus_file, hyperparameters):
        self.text = open(corpus_file)
        self.hyperparameters = hyperparameters
        self.padding = [(-1, "PADDING")] * (self.hyperparameters.
            window_size // 2)

    def __iter__(self):
        sentence = []
        for word in self.text:
            if not word.strip():
                yield self.pad(sentence)
                sentence = []
            else:
                line = word.split()[0]
                w = word.split()[1]
                sentence.append((line,w))

    def rewind(self):
        self.text.seek(0)

    def pad(self,sent):
        return self.padding + sent + self.padding
```

```
class Dictionary(Iterator):
  def __init__(self, dict_file, size):
    self.indices = {}
    self.size = size
    self.dict_file = open(dict_file)
    self.indices["PADDING"] = -1
    self.build()

  def __iter__(self):
    return self

  def __next__(self):
    return self.dict_file.readline().strip()

  def build(self):
    idx = 0
    while idx < self.size:
      self.indices[next(self)] = idx
      idx +=1

  def enlarge(self, size):
    while self.size < size:
      self.indices[next(self)] = self.size
      self.size +=1

  def lookup(self, word):
    return self.indices[word]

  def contains(self, word):
    return (word in self.indices)

  def __getstate__(self):
      return (self.dict_file.name, self.size)

class ReverseDictionary:
  def __init__(self, dictionary):
      self.indices = {}
      for k, v in dictionary.indices.items():
          self.indices[v] = k

  def lookup(self, idx):
```

```
    return self.indices[idx]
```