UNIVERSITY OF CAMBRIDGE
FACULTY OF MODERN AND MEDIEVAL LANGUAGES

LINGUISTICS TRIPOS PART II 2015 DISSERTATION

**Automatic Identification of Spelling
Variation in Historical Texts**

Alexander Robertson

Acknowledgements

I am sincerely grateful and indebted to my supervisor and Director of Studies for their invaluable advice and encouragement, not only during this project but in everything that I have done over these three happy years.

Contents

I	Background	1
1	The variation problem	2
1.1	Desiderata	4
2	Current approaches	5
2.1	Manual processing of historical texts	5
2.2	Norma	6
2.3	VARD2	6
2.4	Evaluating performance	7
2.4.1	Norma	7
2.4.2	VARD2	8
2.5	Critical evaluation	9
II	Theory	12
3	A framework for modelling variation	12
3.1	A formalism for representing text	12
3.1.1	Deriving a precise terminology from the formalism	13
3.2	Summary	15
III	Practice	16
4	An automated approach to the variation problem	16
4.1	The type identification pipeline	16
4.1.1	Feature extraction	17
4.1.2	Vectorisation	17
4.1.3	Clustering	17
4.2	Feature selection	17
4.2.1	Form bigrams	18
4.2.2	Part of speech	18
4.2.3	Part of speech context	18
4.2.4	Pairwise similarity	19
4.3	Worked example	20
5	Evaluating the automated approach	22
5.1	The performance evaluation pipeline	22
5.1.1	A toy example	22
5.1.2	Statistical cluster comparison	24
6	Experimental implementation and evaluation	25
6.1	Experiment 1: Assessing the selected features	25
6.1.1	Design	25
6.1.2	Results	26
6.2	Experiment 2: Large-scale evaluation	27

6.2.1	Results	27
6.3	Experiment 3: Real historical text	27
6.3.1	Results	28
6.4	Experiment 4: Chronology of the Great Vowel Shift	28
7	Discussion	30
8	Conclusion	32
Appendix A Experiment 3 - partial results		34
Appendix B Code: type identification pipeline		39
Appendix C Code: gold standard generation pipeline		47
Appendix D Code: evaluation pipeline		56

List of Figures

1	Variation decreases over time. Adapted from Baron, Rayson, and Archer (2009)	8
2	Visual comparison of equivalent historical and modern texts	15
3	A pipeline for identifying types within a historical text	16
4	A pipeline for generating gold standard texts	23
5	Sample of automatically generated artificial “historical” text	26

List of Tables

1	Issues with the Innsbruck Letter Corpus	5
2	Performance of Norma on historical texts. Adapted from Bollmann (2013)	8
3	Historical forms not classified as variants by VARD2	10
4	Appraisal of current methods against desiderata	11
5	Precise terminology derived from the formalism	14
6	Toy texts represented formally	14
7	Feature vector representation of formal variants in “the redd redde door”	21
8	Output of the type identification pipeline for a given toy input	21
9	Matrix of parameters used for each block in experiment 1	26
10	p -scores for features	27
11	Comparison of type assignment for three values of k . Coloured columns, from left to right, $k=6,000$, $k=8,342$, $k=10,000$	29
12	Appraisal of type identification pipeline against desiderata	33
13	Type identification pipeline output for various k values (157 out of 16,684) . . .	34

Part I

Background

1 The variation problem

Languages in earlier stages of development differ from their modern analogues, reflecting syntactic, semantic and morphological changes over time. The study of these and other phenomena is the major concern of historical linguistics. The development of literacy and advances in technology mean that human language has often been preserved in physical form. Whilst these artefacts will eventually include video and sound recordings, the current life blood of historical linguistics is text. The written word is the *de facto* source of evidence for earlier stages of languages and “the first-order witnesses to the more distance linguistic past are written texts.” (Lass, 1997)

Working with English texts presents both opportunities and problems. More recent stages of English have more sources available, giving a fuller linguistic picture: the Old English (OE) period (600-1150) is survived by approximately 3,000 sources (Healey, 2002), mainly poetry and prose, whilst sources for the Present Day English (PDE) period (nominally 1900-2015) are likely to number in the billions. Working with large amounts of data is increasingly easy, following the development of electronic corpora and automated methods for processing them. This has, in the view of Rissanen (2000, p.7), “freed us from months and years of painstaking pencil work”. However, the freedom afforded by electronic corpora comes at a price: standardised input is required, such that each Saussurean *signifiant* is fixed in relation to the *signifié*. Colloquially, each word must be spelled the same way each time it is used.

This relation allows much of the tedious enterprise of linguistic annotation to be automated. Without it, the most basic corpus tasks, such as constructing word counts or calculating ngram frequencies, become impossible and part-of-speech (POS) tagging accuracy is greatly reduced (Baron and Rayson, 2009). So whilst OE and PDE sources may differ dramatically in number, they at least share a common property in the consistency of their word forms. In OE, this was due to scribal tradition and the copying of texts by hand with the aim of producing a facsimile of an existing text. Inter-regional dialectal variation occurred but within-text word forms were generally consistent. Similarly, little variation is present in PDE due to a process of standardisation, largely in place by the end of the 18th century (Görlach, 1991). Today, the concept of spelling and the notion that it can be correct or incorrect is so pervasive that even linguists may refer to the spelling in pre-standardisation texts, though usually as a shorthand for orthographic variation.

This dissertation is concerned with the word forms found in Early Modern English (EME) texts. These texts lack the consistency of word forms found in PDE but are available in significantly greater number than OE sources. The EME period, here taken to be from 1400 to 1600, was a time of social and linguistic change. It is evidenced not only by poetry and prose but also by personal correspondence in a volume unavailable for earlier epochs, providing a

view of language through a quite different lens, essentially creating the field of historical sociolinguistics (Bergs, 2005). Unfortunately, the volume of material and its pre-standardisation characteristics requires special effort to study. Currently, the creation of corpora from historical sources is largely a manual process. Two examples are the Queen Elizabeth I Spelling Corpus (Evans, 2012) and the Parsed Corpus of Early English Correspondence (Taylor, Nurmi, Warner, Pintzuk, and Nevalainen, 2006). The QEISC, specifically constructed in order to study spelling, contains 22,400 words. Evans spent two months manually identifying and grouping variant word forms and reports that for the creation of any larger corpus “an (semi)-automated system would be imperative.” The PCEEC (Taylor et al., 2006), which was manually annotated with POS information, contains almost 2.6 million words. Nurmi (personal communication April 22nd, 2014), reports: “It took me about three years to complete...I mechanically broke one keyboard doing it.” Had these corpora been based on modern texts, their construction would have taken less time and effort, allowing the focus to be on the actual research aims.

Spelling variation presents problems even for researchers who are not specifically concerned with historical texts as linguistic artefacts. Working on Enroller, a project to aggregate and index multiple historical data sets, Anderson (2013) highlights the impossibility of even simply cross-referencing documents by keyword, a trivial task in PDE: “It is very difficult if not impossible to identify all the probable spellings of a word in historical times or in dialectal texts. This problem is not addressed in the Enroller project but it is an important one that deserves effort and funding to ameliorate.”

There are, then, two important issues regarding spelling variation in historical texts. The first is the impact it has on the efficacy of natural language processing (NLP) techniques. The second is the difficulty of analysing spelling variation on a large scale due to the unsuitability of standard corpus methods for analysing historical texts. Combined, these issues may be referred to as *the variation problem*.

Acknowledging this problem, and the value of addressing it, this dissertation presents a novel solution. It is arranged as follows:

- a table of desiderata for an idealised solution to the variation problem, against which any approach may be assessed;
- critical assessment of current approaches to the problem;
- a formalism for describing texts, allowing the variation problem to be precisely defined and restated;
- a technical implementation of the formalism;
- statistical and experimental assessment of the implementation.

A brief summary of the main findings closes this work.

1.1 Desiderata

The labels of these idealistic principles are a convenient shorthand. They will be referred to when considering the properties of any approach to the variation problem.

Perfection	Precision and recall should be 100%. All variants should be identified and each variant should be matched with its present-day form. Between two systems, the better is that which correctly identifies more spelling variation and correctly proposes modern day equivalents.
Least effort	A major goal is to reduce the labour required to work with historical texts. Between two systems, that which requires the least labour (by some sensible metric) is superior.
Explanatory adequacy	A solution should be explicitly grounded in linguistic theory, able to explain its success (or failure) with reference to the underlying structure of the language data.
Pure input	The input to the solution should be the original text: either the actual original artefact or a close facsimile. The closer a system is to the source, the better it is.
Language agnostic	It should work on any language from any period. A system that only works for one language is as useful as a theory of syntax, for example, that only applies to one language.
Epoch ignorant	Variation should be detected purely on the basis of the input, with no reference to variation (or lack of) in another period. The ideal system is synchronic in its application but a series of such applications may well inform diachronic study.
Preserve input	The output should not replace variant forms with modern ones, but preserve them in-line or present them separately in an appropriate format. A system which acts like a modern-day spellchecker is inappropriate, as these variant forms are as historically relevant as syntactic structure even if they are a gross inconvenience to corpus linguistics methods.

2 Current approaches

In addition to the manual approach taken by Evans (2012) and Taylor et al. (2006), tools exist to assist researchers working with historical texts. The manual approach and two such tools will now be assessed against the desiderata in section 1.1. Following this is a critical evaluation, highlighting crucial theoretical shortcomings. Table 4 (page 11) summarises this assessment.

2.1 Manual processing of historical texts

The traditional approach is to manually examine each word in a historical text and, where a word is not in modern spelling, emend or replace it. There is no standard procedure for manually normalising historical texts or investigating spelling variation, because the process is dependent upon the reason for normalising in the first place or the research questions underpinning the investigation. Where the aim is to create a modern analogue of a historical text, it may be sufficient to simply read the text and replace any variant words by hand, based on the judgement of the reader. In addition to normalising the spelling, further changes may be applied during this process – syntax may be modernised or archaic words updated. For Evans (2012) the purpose was to quantify spelling variation over time, so it was necessary not to replace historical words but to group them together for counting. This was not just time-consuming – Evans (personal communication, December 2nd, 2014) reports it was “error-prone, and the corpus had to be checked several times.”

The Letter Corpus of the Innsbruck Computer Archive of Machine-Readable English Texts (LC-ICAMET) was constructed with the purpose of being used for “language analyses, pragmatic and sociolinguistic studies, but also analyses concerning cultural life and lifestyle” (Markus, 2015). It covers the period from 1386 to 1698, containing 182,000 words which were manually normalised in the 1990s, arranged as an interlinear gloss. The quality of the normalisation is variable. Problematic examples are provided in table 1. Despite its value for its stated purpose, LC-ICAMET is unsuitable for investigation of historical spelling variation. This situation highlights the lack of a recognised standard procedure for text normalisation.

And Sir, as for the 6 couple of *cod , the which you write for	Word completely changed, marked with emendation
And Sir, as for þe vj cowpull of haberndens , the which ye wryte ffore	
Furthermore, sir, pleases it you to know that as on Friday last past I	Word completely changed, no emendation
Ferthermore, syr, plesyth hit yow to wit that as on ffryday last past ay	
Philipp, to know , a way if they should have again their cattle or	Same word as above, but not changed this time
Phylp, to know , a way yf they shuld have ayen there catell or	
one that was with me called Roberd Lovegold, brazier, and threatened	Syntactic change as well as spelling
on that was wyth me callid Roberd Lovegold, brasere, and threte	
the trough / *core of the matter, and of all our demeaning at Drayton	Emendation marked, semantic information added
the trough of the matere and of all owyre demenyng at Drayton	
The log / *cabin and the remnant of your place was beaten down	Emendation marked, semantic information added, wrong word given (should be <i>lodge</i>)
The logge and the remenaunte of your place was betyn down	

Table 1 – Issues with the Innsbruck Letter Corpus

2.2 Norma

This command-line tool (Bollmann, 2012) is written in Python 2. Norma has two operational modes, interactive and batch. First, variants are detected by comparing each word to a modern dictionary. Words with no match are highlighted as variants. The interactive mode iterates over each word in a source text, prompting the user to confirm or reject candidates for normalisation, whereas batch mode automatically selects a candidate. Norma uses “normalisation modules” which allows the user to extend the functionality of the tool with custom normalisation techniques. Three modules are included as standard:

Word List Mapping

If the variant is in a list of historical:modern word mappings, it can be replaced by this method. The list must be compiled by hand.

Rule-Based Normalisation

Described in detail in Bollmann, Petran, and Dipper (2011), this employs a list of character rewrite rules, analogous to those seen in generative theories of syntax and phonology. Valid rules are applied, left to right, to the characters of a historical form and the resulting string, if found in a target lexicon of modern words, becomes a normalisation candidate. The rewrite rules are generated automatically from a training corpus. This can either take the form of manually-compiled historical:modern word pairs or, as in Bollmann et al. (2011), from a portion of an aligned corpus set aside for use as training data.

Weighted Levenshtein Distance

The Levenshtein distance (Levenshtein, 1966) between two strings is defined as the number of single character deletions, insertions or replacements required to transform one string into another. For example, the distance between CAT and BAT is 1. A distance of zero means the two strings are identical. The version used in Norma allows weights to be applied to the deletions/insertions/replacements required to transform the input string, allowing them to be interpreted as more or less costly. Bollmann et al. (2011) gives $v \mapsto u$ as an example of a low cost edit for Early New High German texts, as it is a common and valid spelling variation, whilst $v \mapsto x$ should be a high cost edit given its implausibility. Weights can be specified manually (Bollmann, 2012) or learned automatically (Adesam, Ahlberg, and Bouma, 2012). This module performs pairwise Weighted Levenshtein Distance comparisons between the words of the source text and a modern dictionary. The closer the score is to zero (i.e. no edits made), the more similar the two strings are.

2.3 VARD2

VARD2 (Baron and Rayson, 2008) is written in Java. It uses a graphical interface with the same operational modes as Norma. It uses the same dictionary method for detecting variants. In the word processor-style interactive mode, a user can click identified variants to select a replacement

from normalisation candidates. This operation can be automatically applied to all other tokens in the text which match the current variant. The choices made are tracked in order to update confidence scores – selecting the same replacement for a variant each time increases the future confidence score for that replacement. In batch mode, decisions about replacements are made based on this score – the top-ranking candidate above a threshold of confidence automatically replaces the variant. Four methods for determining replacement candidates are used:

Known Variants

This is the same as the Word List Mapping module of Norma.

Letter Replacement

This is the same as the Rule Based Normalisation module of Norma. The rule list is generated by the DICER tool (Baron and Rayson, 2009) using the interlinear gloss of LC-ICAMET (see page 5) as an aligned corpus for training purposes.

Phonetic Matching

A code is generated by the Soundex algorithm (Russell and Odell, 1918) taking the variant as input. Codes are of the form A123, where A is the first character of the input and 123 is generated based on the properties of the consonants in the input. For example, ‘today’ is T300, “tomorrow” T560. The Soundex code for each variant is compared against the codes for modern words in the dictionary. A matching code results in that modern word becoming a replacement candidate.

Edit Distance

For all candidates offered by the previous methods, the Levenshtein distance between the variant and replacement candidates is calculated. This is the same algorithm as in Norma but without weighting and with a slight change to the output, which is normalised to a value between 0 and 1 rather than being a count of edit operations.

2.4 Evaluating performance

Both Norma/VARD2 provide analysis of the performance of their methods. A direct quantitative comparison is not possible, since each uses quite different metrics. However, their assessment methodology is similar: historical text is normalised manually to create a gold standard, then the original historical text is processed. The output of this is compared to the gold standard and statistics generated. A brief summary of these will now be presented.

2.4.1 Norma

Bollmann (2013) manually normalised five German texts. The first 500 tokens of each corpus was used as training data, though Bollmann also evaluated the impact of using 100, 250, 1000 and 2000 tokens as training data. The remaining non-normalised tokens were then processed

in Norma. The results are reproduced in table 2. A metric called “accuracy” was used. Baseline accuracy is the number of tokens in the pre-normalised text which already match modern spellings. This was re-calculated after processing in Norma.

Corpus	Dating	Size	Baseline	Word list Mapping	Rule-based	WLD	Combined
Berlin	15c	4,700	23.05%	62.05%	63.17%	60.71%	75.07%
Melk	15c	4,541	39.32%	63.15%	64.14%	69.34%	74.49%
Sermon1	1677	2,178	72.71%	76.46%	78.67%	76.40%	79.56%
Sermon2	1730	2,137	79.47%	85.22%	88.52%	88.15%	91.81%
Sermon3	1770	1,953	83.41%	86.58%	90.50%	95.46%	95.73%

Table 2 – Performance of Norma on historical texts. Adapted from Bollmann (2013)

PERCENTAGE OF VARIANT TYPES IN SELECTED CORPORA OVER TIME

Figure 1 – Variation decreases over time. Adapted from Baron, Rayson, and Archer (2009)

Two factors are notable. First, each text differed in baseline levels of variation. This is because they are from different historical periods and there is a trend over time of decreasing variation (fig. 1). Second, each corpus is quite small, likely due to the labour involved in manually normalising the texts. Performance improved with more training, but in some cases this was approaching 50% of the unseen data. Improvements over baseline were highly dependent on how much variation it contained.

2.4.2 VARD2

Baron and Rayson (2009) used LC-ICAMET for their gold standard, from which they extracted known variants and letter replacement rules. How the quality of this source (see section 2.1) affected its suitability as an aligned corpus is unknown – Baron and Rayson do not mention this issue. LC-ICAMET was split into equally sized samples, then processed in batches to allow

VARD2 to learn more with each new batch of data. Performance was rated using precision and recall:

$$\text{recall} = \frac{|\{\text{correct identifications}\} \cap \{\text{possible identifications}\}|}{|\{\text{correct identifications}\}|} \quad (1)$$

$$\text{precision} = \frac{|\{\text{correct normalisations}\} \cap \{\text{possible normalisations}\}|}{|\{\text{possible normalisations}\}|} \quad (2)$$

Recall (R) is the proportion of variants which have been successfully identified. Precision (P) is the proportion of those correct identifications which have been correctly normalised. The baseline with no training (besides the precompiled list of known variants and letter rules) was $R = 92\%$, $P = 45\%$. After training, improvements in R and P plateaued at 93% and 65% once 40,000 items of training data had been used. This represented 22% of the total data.

2.5 Critical evaluation

In the introduction, two major issues caused by spelling variation were highlighted, comprising the variation problem. Each will now be examined more closely, in the context of Norma/VARD2.

If the purpose of working with a historical text is to “modernise” and make it suitable for NLP then, required effort aside, the methods described are valid solutions. The software tools especially are useful because they have been created not only for identifying but also “fixing” spelling variation. Therefore, Norma/VARD2 treat historical spelling variation in the same way as Microsoft Word treats spelling errors. They make the same distinction as Kukich (1992) between *error detection* (through the use of dictionary look-up) and *error correction* (through pairwise comparison of detected variants/errors to a modern dictionary, by some metric such as Levenshtein distance). But how appropriate are these distinctions in the context of analysing historical change through *variation detection*, the second part of the variation problem? It will now be argued that they are inappropriate. This claim will be supported by showing that the technology behind Norma/VARD2 carries with it assumptions which make them unsuitable in the fuller context of the variation problem.

First, the detection method used in Norma/VARD2 implicitly assumes historical word forms are deviant and modern ones are orthodox, as if writers in the 15th century had the modern form in mind each time but often mysteriously failed to produce it. The evidence for this is the use of modern dictionaries for detecting variants. But this approach does not detect variants. It only detects *non-modern words*. Table 3 shows examples of historical words which are variants within the context of the time period they were written in, but which are not considered variants by VARD2. This use of dictionaries violates the principle of epoch ignorance.

Second, the correction methods used by Norma/VARD2 explicitly make the same assumptions that their detection method makes implicitly, that historical forms are errors. They attempt to undo these errors using string comparison metrics. However, the Levenshtein algorithm is based on the work of Damerau (1964). This looked at the types of spelling error produced when human operators were creating punch cards for a mainframe computer. 80% of errors

Historical	Modern equivalent	Other modern matches
bee	be – verb	bee – noun, an insect
don	done – verb	don – verb, to put on
goon	gone – verb	goon – noun, a fool
pore	poor – adj	pore – noun, skin opening
rite	right – verb	rite – noun, a ritual
prey	pray – verb	prey – noun, hunted object
caws	cause – verb	caws – noun, sounds made by crows
wold	would – modal	wold – noun, a woodland

Table 3 – Historical forms not classified as variants by VARD2

were found to fall into four categories – character deletion, extra character insertion, character substitution, and character transposition. “These are the errors one would expect as a result of misreading, hitting a key twice, or letting the eye move faster than the hand.” (Damerau, 1964, p.171)

This means that Norma/VARD2 use an algorithm which was designed based on the errors created by inaccurately inputting text on a *computer keyboard*. They do not take a linguistically-motivated approach to why historical and modern word forms differ. Instead, they operate under the belief that input errors on a keyboard are analogous to five hundred years of language change, as if the EME period were caused by clumsy typing. Levenshtein distance is at the very core of Norma/VARD2 and its provenance subtly taints not only their performance but the theoretical validity of that performance. So whilst approaches based on spell-checking technology can achieve partially acceptable results, there is no scope for improving those results without abandoning that fundamentally flawed method. They have no explanatory power.

Table 4 – Appraisal of current methods against desiderata

	VARD2	Norma	Manual
Perfection	No. 93% precision, 65% recall.	No. 75.07% accuracy on 4700 tokens using 500 as training data (baseline = 23.05%) 95.73% accuracy on 1953 tokens using 500 as training data (baseline = 83.41%)	Possibly, depending on the methodology, but not the case in the LC-ICAMET.
Least effort	Complexity $O(n^2)$ for Word List Mapping. Best case $O(m \times n)$ for each individual LD comparison, increasing linearly with text size. Speed limited by computer power. Up to 22% of data needs manually tagged for optimal training (on large data set). Interactive mode requires word-by-word checking by user, complexity $O(n)$ increasing linearly with text size. Batch mode allows automatic acceptance of best-ranked candidate.	Complexity $O(n^2)$ for Word List Mapping. Best case $O(m \times n)$ for each individual LD comparison, increasing linearly with text size. Speed limited by computer power. Requires manually created training data, up to 50% of source on small data sets. Interactive mode requires word-by-word checking by user, complexity $O(n)$ increasing linearly with text size. Batch mode allows automatic acceptance of best-ranked candidate.	Complexity $O(n)$ for word-by-word replacement. Complexity $O(n^2)$ for grouping words from text by variant. Speed limited by speed of human processor, but can be massively parallelised with enough humans.
Explanatory adequacy	Uncritically uses technology from modern spellchecker research. Not grounded in theory.	Uncritically uses technology from modern spellchecker research. Not grounded in theory.	Possibly. Depends on the person doing the manual work.
Pure input	No. Uses an optional list of known variants, without which performance is harmed. Requires a modern dictionary, without which no variants can be detected.	No. Uses an optional list of known variants, without which performance is harmed. Requires a modern dictionary, without which no variants can be detected.	Most likely.
Language agnostic	Soundex phonetic algorithm is English-specific.	Yes.	Possibly. Depends on the person doing the manual work.
Epoch ignorant	No. Compares historical forms directly with modern forms.	No. Compares historical forms directly with modern forms.	Possibly. Depends on the person doing the manual work.
Preserve output	Yes. Can output normalised text in XML format.	Yes. Outputs normalised text, word by word, to console where it can be processed elsewhere.	Possibly. Depends on the person doing the manual work.

Part II

Theory

3 A framework for modelling variation

It has been established that current non-manual approaches to the variation problem are based on unsuitable modern spell checking technology and an assumption that historical words are deviant in relation to modern ones. Further, each approach only addresses the first half of the variation problem – that of normalising historical texts for improving NLP efficacy. They provide no useful way of investigating spelling variation in its own right and addressing the second half. This is part of a wider problem: the assumptions that constitute their theoretical and methodological basis are mostly implicit, as demonstrated by the uncritical use of the Levenshtein algorithm. They are not grounded within a framework which precisely lays out the problem they are trying to solve, instead relying on pragmatism – the problem is considered solved when a score of 100% is achieved in some metric for normalising a gold standard.

What is needed, and shall now be presented, is a schema for considering texts as formal linguistic entities rather than a series of words to be linearly processed in isolation by a bottom-up spell-checking process. The benefits of this are clear. First, it will provide a high level view from which variation can be seen as a property of not just individual words but of entire historical texts. Second, it will permit the precise definition of the terminology in use. Finally, and most importantly, it will give a theoretically motivated way of approaching the variation problem.

3.1 A formalism for representing text

A text, T , is a finite linear sequence of strings. From a variation perspective, it is formally defined with the following parameters:

1. $\mathcal{F} = \{f_1, f_2, f_3, \dots, f_n \mid f \in T\}$

The finite set of strings within T . Its elements are determined by tokenising the text.

2. $\mathcal{P} = \{p_1, p_2, p_3, \dots, p_L \mid p \in \mathbb{N}, L = |T|\}$

The finite set of positions within T . A position is a unique index of a string's place within T . The number of elements is determined by the cardinality of T .

3. $\mathcal{S} = \{s_1, s_2, s_3, \dots, s_n \mid s \in T\}$

The finite set of senses used in the text. It is the collection of the semantic references made by the text.

4. $\alpha =$ the instance relation over $\mathcal{F}, \mathcal{P}, \mathcal{S}$

Describes a set of tuples, $\mathcal{I}$. Each $i \in \mathcal{I}$ has the structure (f, p, s) . These tuples link each $p \in \mathcal{P}$ to some $f \in \mathcal{F}$ and some $s \in \mathcal{S}$. A p can be associated with any f or s , and an f or s can be associated with one or more p , but each p is used only once. Associating senses with forms allows polysemy and homography to be represented.

5. γ = the variant function

With domain $\mathcal{F}$, range $\subseteq \mathcal{I}$.

The set of all possible values is $\mathcal{V}'$. This set contains any $i \in \mathcal{I}$ which share the same value for f and s . These subsets of $\mathcal{I}$ are denoted by $\mathcal{V}_{(f,s)}$.

6. λ = the type function.

With domain $\mathcal{S}$ and range $\subseteq \mathcal{V}'$.

The set of all possible values is $\mathcal{T}'$, the members of which are subsets of $\mathcal{V}'$, such that although they do not share the same f they have the same s . These subsets of $\mathcal{V}'$ are denoted by $\mathcal{T}_s$.

3.1.1 Deriving a precise terminology from the formalism

Discussion of spelling variation is hindered by overloaded terms which have both technical and colloquial meanings. This claim will be justified by demonstrating how common terms map onto multiple elements of the formalism:

- $\mathcal{F}$ is a text's *types*, *vocabulary*, *words* or *lexicon*. An isolated f is generally referred to as a *word* or *token* but can also be viewed as a *spelling*.
- Each i associates a *word* or *spelling* from the *vocabulary* with a unique position in the text. This too can be referred to as a *word*.
- Each $\mathcal{V}_{(f,s)}$ is a way of dividing the text into groups of words which have the same written form and semantics. This is one way of *spelling* a *word*.
- Each $\mathcal{T}_s$ groups those $\mathcal{V}_{(f,s)} \in \mathcal{V}'$ such that though they have different *spellings*, they are still the same *word* based on semantics. For example, American and British English have different words or spellings (e.g. *favourite* and *favorite*) with identical sense.

Accordingly, it will be productive to avoid these labels and to highlight the theoretical contrasts made by the formalism. The purpose of such contrasts is to explicitly lay out the structure imposed on the variation problem by the schema and to propose testable solutions. Terminology derived from the formalism is presented in table 5 (page 14). Hereafter, only these terms will be used.¹ The distinctions made by the formalism are exemplified by toy examples in table 6 (page 14).

¹Special effort should be made to dissociate the upcoming usage of *type* from its technical meaning in information theory.

Formalism	Terminology	Description
f	form	the physical characteristics of a string in a text
p	position	a location within the text
i	instance	a form and sense linked to a unique location
$\mathcal{V}_{(f,s)}$	variant	a collection of instances with both a shared form and sense
$\mathcal{T}_s$	type	a collection of variants with a shared sense but not shared form

Table 5 – Precise terminology derived from the formalism

	Historical	Modern
Text	the redde redd door	the red red door
Forms	{the, redde, redd, door}	{the, red, door}
Positions	{1, 2, 3, 4}	{1, 2, 3, 4}
Senses	{a, b, c}	{a, b, c}
Instances	{(1, the, a), (2, redde, b), (3, redd, b), (4, door, c)}	{(1, the, a), (2, red, b), (3, red, b), (4, door, c)}
Variants	$\mathcal{V}_{(the,a)} = (1, \text{the}, a)$ $\mathcal{V}_{(redde,b)} = (2, \text{redde}, b)$ $\mathcal{V}_{(redd,b)} = (3, \text{redd}, b)$ $\mathcal{V}_{(door,c)} = (4, \text{door}, c)$	$\mathcal{V}_{(the,a)} = (1, \text{the}, a)$ $\mathcal{V}_{(red,b)} = (2, \text{red}, b), (3, \text{red}, b)$ $\mathcal{V}_{(door,c)} = (4, \text{door}, c)$
Types	$\mathcal{T}_a = \{\mathcal{V}_{(the,a)}\}$ $\mathcal{T}_b = \{\mathcal{V}_{(redde,b)}, \mathcal{V}_{(redd,b)}\}$ $\mathcal{T}_c = \{\mathcal{V}_{(door,c)}\}$	$\mathcal{T}_a = \{\mathcal{V}_{(the,a)}\}$ $\mathcal{T}_b = \{\mathcal{V}_{(red,b)}\}$ $\mathcal{T}_c = \{\mathcal{V}_{(door,c)}\}$

Table 6 – Toy texts represented formally

The locus of diachronic variation between two texts is not in the number of senses or positions, but in how those senses and positions are distributed with relation to forms. In modern texts, there is a one-to-one mapping from types to variants and from variants to forms. The number of variants is therefore equal to the number of types, encapsulating the same information in an essentially vacuous distinction. Historical texts differ in this respect, with a one-to-many mapping from types to variants being possible but preserving the one-to-one mapping from variants to forms. The type/variant distinction is meaningful in historical texts because it permits a technical perspective of variation without interference from the variation-free modern day situation that is generally found in standard texts. The original variation problem can now be restated within this new framework:

- *Investigating spelling variation within historical texts involves arranging the observable variants into groups of types, such that each type maps to one sense, with as many word forms as are present in the text representing that sense.*
- *For normalisation, the forms of the variants within each type must be made identical.*

In the historical example of table 6, an investigation of spelling variation might involve statistical analysis of the number of variants per type. Normalising this text to modern standard English would involve examining the members of $\mathcal{T}_b$ and changing the instances within each variant to have modern forms, such that $(2, \text{redde}, b) \mapsto (2, \text{red}, b)$ and $(3, \text{redd}, b) \mapsto (3, \text{red}, b)$.

Figure 2 – Visual comparison of equivalent historical and modern texts

b). These new instances are then merged into a new variant, $\mathcal{V}_{(red,b)}$, which replaces the historical variants, $\mathcal{V}_{(redde,b)}$ and $\mathcal{V}_{(redd,b)}$. The text is then ready for NLP processing. This normalisation procedure can be visualised as reducing the branches within a tree whilst maintaining the number of leaves, as in fig. 2.

3.2 Summary

It has been demonstrated that historical texts, within the proposed framework, exhibit a one-to-many mapping from types to variants. Extracting the variants is trivial because they are apparent from the text and there is a one-to-one mapping from variants to forms regardless of epoch. Identifying types is more challenging, since they represent a level of non-surface structure within the text, not directly observable. However, there exist computational methods for accessing hidden structure in data. What follows next is an example of how these can be applied to the problem of identifying types. This will form a solution to the first part of the reformulated variation problem.

Part III

Practice

4 An automated approach to the variation problem

There are two major technical components within this dissertation. First, the technical implementation of a solution to the first element of the reformulated variation problem (page 14). This maps variants to types but does not make any attempt to link the historical types to their modern equivalents. Second, statistical evaluation of the accuracy of the implemented solution. This evaluation is part of a feedback loop, allowing the technical implementation to be adjusted in order to improve performance and efficiency. As shall be seen, multiple components were initially considered as part of the implementation but only the best were selected. These two technical components are the type identification pipeline (TI-pipeline) and the gold standard generation pipeline. When combined, they form the performance evaluation pipeline.

4.1 The type identification pipeline

By treating the hidden types in a text as latent variables, it is possible to leverage the manifest properties of a text's variants using computational techniques. There are multiple techniques available (Bishop, 2007) and determining which to use depends upon the problem under consideration. The variation problem can be viewed as a classification task, where variants must be classified according to which type they belong to. For example, in the case of the historical text in fig. 2 (page 15) there are four variants to be classified into three types. This is an unsupervised learning task because types do not have labels – they are a theoretical construct predicted to be in the data, rather than a concrete property of the data.

Figure 3 – A pipeline for identifying types within a historical text

Figure 3 shows the process by which a historical text is sorted into types. This was implemented in Python 3 (Appendices B to D). Each of the three key components shall be described

in turn, then a toy example presented.

4.1.1 Feature extraction

At this stage, the input text is tokenised into strings, which are equal to forms. For each form, a vector of features is constructed. These capture salient information about the forms from which the latent variable, type, can be inferred. The choice of features is discussed afterwards, in section 4.2.

4.1.2 Vectorisation

This is a pre-processing step which converts components of feature vectors from categorical forms into binary representations, through a process known as “one-of-K” encoding. For example, a feature *colour=blue* becomes *colour_blue=1*. This is necessary because the clustering step requires all vector components to be numerical rather than boolean truth values or strings.

4.1.3 Clustering

The binary vectors are arranged in an n -dimensional space, where n is the number of components per vector. Two vectors which are closer to each other, by some distance metric, are considered to be more similar than two which are, by the same metric, farther apart. There are many different types of clustering algorithm (Xu and Wunsch, 2005), with different mathematical properties which make them suitable for a range of applications. For current purposes, agglomerative hierarchical clustering (Zhao, Karypis, and Fayyad, 2005) was chosen. Multiple factors contributed to this decision, mainly computational complexity and output characteristics. Complexity is an important consideration when working with large datasets, as it affects computation time. Furthermore, agglomerative hierarchical clustering can produce asymmetric clusters. This reflects the fact that some types can have many variants whilst others may only have one.

The clustering process takes a parameter, k , and divides an n -dimensional space into x partitions, where x is the number of vectors. For each cluster the distance to every other cluster is calculated. The clusters which are nearest to each other are merged. This process is repeated until the original x clusters are reduced to k clusters. Strategies for choosing a value for k are discussed later, in section 6.

4.2 Feature selection

Clustering is an unsupervised machine learning algorithm. It requires no training phase. However, it is necessary to select the features which comprise the vectors used by the algorithm. Intuitively, the more features of variants which share similar values, the more likely it is that those variants are members of the same type. The formalism can help determine which features will be informative, but a degree of pragmatism is required. For example, the instance tuples of form, position and sense suggest three possible features. Of these only form is useful because

position is unique (i.e. maximally contrastive) to each instance and a type is not determined by the positions of its variants’ instances within the text. Sense would be maximally informative but is impossible to extract from text without doing exactly the kind of labour-intensive manual labelling that this approach is designed to avoid. The features which were selected, and their rationale, will now be presented.

4.2.1 Form bigrams

Following Robertson and Willett (1992), each form was padded with symbols to represent the start and end of the string. This was then split into chunks of size n . For example, CAT becomes §CAT§ which becomes §C, CA, AT, T§. Each bigram then becomes a feature of that form. This captures the intuition that whilst forms sharing the same individual characters need not necessarily be variants within a type (for example, ACT and CAT contain the same characters but are not variants of one another), related forms are likely to be somewhat constrained in terms of the structure of their composite characters. Indeed, CAT and ACT share only a single bigram, T§, out of four whilst CAT and KAT share two.

4.2.2 Part of speech

Intuitively, forms which are consistently associated with the same syntactic classes are more likely to be members of variants of the same type. However, extracting this information is hindered by the very problem trying to be solved here: automatic POS-tagging is highly inaccurate for historical texts (Baron and Rayson, 2009). Fortunately, the PCEEC (Taylor et al., 2006) contains manually POS-tagged historical texts. Although using these violates the principles of least effort and pure input, it is worthwhile assessing these features because they capture facts of the syntactic structure of the text, in line with the principle of explanatory adequacy.

For each unique form in the source data, its associated POS tags were recorded and divided by the total number of occurrences of that form in the data. This gives a probabilistic value for each tag. For example, *stedfast* appears four times in a text. It is always tagged ADJ, so it has a feature representing this fact with a value of 1.

4.2.3 Part of speech context

Firth (1957) is most quoted for the line “You shall know a word by the company it keeps”. Though usually applied to word-sense disambiguation and distributional semantics, the sentiment is applicable to the variation problem too. Variants within types, regardless of their forms, are likely to be used in the same linguistic environments. However, as with POS-tagging the very nature of the variation problem results in massive growth of the number of features that would result from including form-based context, introducing significant noise into the data. Instead, POS context was used. The PCEEC uses a set of fewer than 92 unique tags, compared to the many thousands of unique forms found in historical texts.

For each unique form in the source data, the POS of the forms preceding and following it were recorded. These were then normalised to give a probability associating the context of the

target form with a POS tag. For example, *stedfast* appears four times in a text. The forms immediately preceding it are tagged ADVR, ADVR, D, D. The forms immediately following it are tagged P, P, ADJ+N, ADJ+N. Therefore, the probabilities of these tags are all 0.5.

4.2.4 Pairwise similarity

It has been argued that the use of modern dictionaries and spell-checking technology to identify and normalise historical variation is inappropriate but it will now be demonstrated that, with some revisions, it can be adapted to avoid criticisms based on a violation of the principle of epoch ignorance. Rather than comparing historical forms to a modern dictionary, they can instead be compared to the other historical forms within the same text. Words which are, in this context, more similar by some metric are more likely to be variants of a type. This approach is directly motivated by considering the desiderata within the framework of the proposed formalism – a good example of the benefits that come from clearly defining a problem beforehand.

VARD2/Norma use a version of Levenshtein distance but many other metrics exist for comparing strings. The algorithm that will be used here is Jaro distance (Jaro, 1989). This differs from Levenshtein distance in two important ways. First, it considers the number of insertions, deletions and transpositions required to transform one string into another rather than insertions, deletions and substitutions. Considering transpositions is theoretically motivated, in line with the desiderata, because transpositions are common in historical word forms (e.g. *softely/softlye*, *chasde/chased*), whilst substitutions can be accounted for by a chain operation of deletion and insertion. Second, the score generated by the Jaro algorithm is not a simple count of edit operations but normalised from 0 to 1. This makes it more suitable for use as a feature value in clustering. The Jaro distance, d_j , between two strings, s_1 and s_2 , is defined:

$$d_j = \frac{1}{3} \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) \quad (3)$$

where:

$$m = \left\lfloor \frac{\max(|s_1|, |s_2|)}{2} \right\rfloor - 1 \quad (4)$$

and:

$$t = \frac{m}{2} \quad (5)$$

To generate features based on this, pairwise Jaro distances are first calculated for all words in the input text. The top ten results are then encoded as feature-value pairs. For example, for *stedfast* some of the top-ranking comparisons are *stedefaste*, *stedfastlie* and *stedfastnesse*, with Jaro distances of 0.933, 0.909 and 0.871 respectively.

The raw scores of the pairwise comparisons are not intended to be the main predictive feature of similarity between variants. Jaro distance is a true metric function and satisfies certain mathematical criteria. Such a function, d , must be:

- symmetric: $d(x, y) = d(y, x)$
- non-negative: $d(x, y) \geq 0$
- coincident: $d(x, y) = 0$ if $x = y$
- subadditive: $d(x, y) \leq d(x, z) + d(z, y)$

By pairwise comparing all variants within a text using Jaro distance, a metric space is created. Items which are closer to each other within this space are more similar but the property of subadditivity also captures relations of similarity between partitions of the metric space. Intuitively, this mirrors assumptions of the formalism in that what variants have in common is some latent variable, types, that connects them. Types, then, are equivalent to individual partitions of the metric space.

4.3 Worked example

To demonstrate the TI-pipeline, a toy example will now be presented based on the text “the redd redde door”. First, forms are extracted by tokenisation. Each form is then tagged with its POS before values for the previously listed features are extracted to create feature vectors. Table 7 shows how each of the forms found in the text is related to the structure of the formalism and how this information is represented within the TI-pipeline. These feature vectors are then converted to binary vectors and processed using agglomerative hierarchical clustering.

In this example, it is clear to see that the expected number of types is 3, so the k parameter of the clustering process is set to this value.² The output of the clustering process is as seen in table 8 and can be interpreted as having successfully divided the input text into three types, assigning the two variants $\mathcal{V}_{(redd,b)}$ and $\mathcal{V}_{(redde,b)}$ to the same cluster, i.e. type.

²This is not always as easy to determine and is addressed in section 6.

	$\mathcal{V}_{(the,a)}$	$\mathcal{V}_{(redd,b)}$	$\mathcal{V}_{(redde,b)}$	$\mathcal{V}_{(door,c)}$
Pairwise Jaro distance score	redd=0.527 redde=0.511 door=0.0	the=0.527 redde=0.933 door=0.0	the=0.511 redd=0.933 door=0.0	the=0.0 redd=0.0 redde=0.0
Bigrams	§T=1.0, TH=1.0 HE=1.0, E§=1.0	§R=1.0, RE=1.0 ED=1.0, DD=1.0 D§=1.0	§R=1.0, RE=1.0 ED=1.0, DD=1.0 DE=1.0, E§=1.0	§D=1.0, DO=1.0 OO=1.0, OR=1.0 R§=1.0
Part of speech	self_is_Det=1.0	self_is_Adj=1.0	self_is_Adj=1.0	self_is_Noun=1.0
Part of speech context	prev_is_START=1.0 next_is_Adj=1.0	prev_is_Det=1.0 next_is_Adj=1.0	prev_is_Adj=1.0 next_is_Noun=1.0	prev_is_Adj=1.0 next_is_STOP=1.0

Table 7 – Feature vector representation of formal variants in “the redd redde door”

Form	Cluster
redd	0
redde	
the	1
door	2

Table 8 – Output of the type identification pipeline for a given toy input

At this point, as intended, the first half of the restated variation problem has been fully addressed.

5 Evaluating the automated approach

The preceding example is interesting as a proof-of-concept for the proposed formalism and TI-pipeline. However, the accuracy of the method on a four instance text does not reasonably imply the same performance can be expected for a corpus of many more instances. Also, accuracy was determined manually rather than mechanically. Two processes will now be presented. First, a pipeline for generating gold standard texts which can be used to evaluate the output of the TI-pipeline. Second, a statistical method for evaluating pipeline performance on such generated gold standard texts. Each will be demonstrated using a toy example.

5.1 The performance evaluation pipeline

As outlined in section 2.4 (page 7), VARD2/Norma make use of different methods for assessing their accuracy. However, neither of these are suitable for the proposed method because of the type of output generated by the extraction pipeline – rather than map from historical to modern forms, it maps from variants to types. Although VARD2/Norma use different evaluation metrics (precision/recall, accuracy) they both use a similar methodology, taking a manually-normalised corpus as a gold standard against which the tools can be assessed. There is something to criticise in each of the gold standards used. VARD2’s gold standard is based on LC-ICAMET, which suffers from issues relating to its compilation, whilst the very small size of Norma’s gold standard results in significant proportions of it being used as training data. Ironically, the variation problem itself is to blame for the lack of large general purpose gold standards which anyone working on the problem can use to assess solutions.

What is needed, and what will now be presented, is a process for automatically generating gold standard data. This takes a modern text, replaces forms within it with historical forms and tracks the changes which have been made. It is then possible to use this artificial historical text, with the tracked changes acting as an answer key. Together, this key and the generated text form a gold standard. The output of this process can be used as the input to the type identification process (fig. 3, page 16). Figure 4 provides an overview of the generation process. This process will be described in detail, with a toy example demonstrating the joint generation/identification pipeline.

5.1.1 A toy example

First a modern text is refactored in line with the structure of the formalism in section 3.1 (page 13). The raw text is tokenised and each form associated with its unique position, to create an instance. These instances are underspecified, containing no semantic information, though this is not relevant to the task at hand. Instances with the same form are grouped into variants, which are then grouped into types. This takes advantage of the fact that in modern texts there is a one-to-one mapping of forms, variants and types – accordingly, there is no need to use semantics to group variants into types. The next step is to cause variation. This is done by inducing the reverse of the normalisation process: rather than merge variants into types,

Figure 4 – A pipeline for generating gold standard texts

types have their variants split. The instances contained within variants are renamed, taking new names from a list of known variants, which has the effect of creating new variants. During this process, a variety of useful data can be tracked, including the original and resultant number of types and variants, which types have been split and how many times, which new forms have been inserted and at what position, how much variation is present (6) as well as how much variation has been added.

$$\% \text{ variation} = 100 - 100 \left(\frac{\text{number of types}}{\text{number of variants}} \right) \quad (6)$$

Using the example of “the red red door”, three initial types would be created. The type containing a single variant for *red*, containing two instances, would then be targeted for variation. (There would be no point targeting *the* or *door* because they have only one instance per type – changing the form of those instances would be vacuous.) The modern form *red* is consulted in a dictionary of known variants, which maps modern forms to historical ones. The alternative forms found there are then used to replace those found in the instances within the *red* variant. The effect of this is to create two new variants, containing one instance each, but which are still held within the same type. The instances are then called in the order of their position feature, resulting in the creation of a new text, for example “the redd redde door”. By (6), the original text has 0% variation, whilst the new one has 25%. Crucially, the location of this variation is known, because instances connect forms to position, variants and types. Using this toy example as a gold standard would return two items: the “historicised” version “the redd redde door” and an expected cluster (i.e. type) assignment. matching that of table 8 (page 21).

This text can then be used as the input to the pipe identification pipeline. The output of this is, naturally, the same as shown previously in table 8. The accuracy of this output, compared

to the gold standard, is easy to see because of the small size of the data. In order to assess larger data sets a more principled and rigorous approach is required.

5.1.2 Statistical cluster comparison

The gold standards used by VARD2/Norma were constructed to assess their solution to the normalisation aspect of the variation problem. The artificial gold standards generated here are intended to be used to assess the type identification aspect.³ As a result, statistical measures such as precision and recall are not suitable. This is because there is no label (i.e. a modern form) for each type represented by the clusters. Instead, a different metric must be used.

The output of the TI-pipeline is a list of cluster assignments, as is the output of the gold standard creation pipeline. Each can be seen as the partition of a set, where the whole set is the collection of variants and each partition of that set represents a type. Each element of the set must be a member of one and only one subset and each subset must contain at least one element. The gold standard partitioning represents the “ground truth”, against which the type identification process can be externally validated. There are several information theoretic methods for comparing the similarity of two possible partitions of a given set. An especially clear outline of these methods is to be found in Vinh, Epps, and Bailey (2010).

For current purposes Adjusted Mutual Information Score (AMIS) is ideal. This choice is motivated by the argument presented in Vinh et al. (2010), namely that an ideal set partition comparison measure should (a) be a true metric (satisfying the properties of positive definiteness, symmetry and triangle inequality), (b) be normalised to a fixed range such as $[-1,1]$ or $[0,1]$ and (c) have a constant baseline such that its expected value between a ground truth partition and any random repartitioning should be zero thus indicating no similarity. AMIS satisfies only the second and third of these, not being a true metric, but this trade-off is acceptable because it is more important to adjust for chance, which is what the constant baseline does, since with large data sets it is possible to achieve high mutual information scores purely by random assignment of clusters. Therefore, when comparing the gold standard ground truth and the output of the TI-pipeline, an AMIS of 0 can be interpreted as the output being no better than randomly grouping the variants into types. A score of 1 means that the two assignments are identical. A score between these values can be interpreted in line with Vinh et al. (2010, p.2839-2840), with scores closer to 1 meaning more information is mutual between the assignments than would be the case if the AMIS were lower. Another attractive property of AMIS is that it is independent of any labels for the clusters, which is important here because the clusters have no labels – instead, the clusters themselves are the latent variables in the data, namely types.

In the toy example, the ground truth exactly matches the output of the type identification process, giving an AMIS of 1. Experimental results, using more realistic data, will now be presented.

³Although adapting the software to generate interlinear glosses would be trivial, texts generated this way would not be a very rigorous test of the abilities of VARD2/Norma because the variation is generated by a list of known variants which those tools contain.

6 Experimental implementation and evaluation

Four experiments were run, evaluating the performance of the proposed formalism and pipelines. These are presented below, followed by a critical discussion of the output generated when a genuine historical text is used.

6.1 Experiment 1: Assessing the selected features

Before using the current model on real historical data, it will be useful to assess the predictive powers of the features that drive the clustering process and ensure that all features correlate positively with the latent variables being sought in the data. Another important consideration is the time taken to compute the cluster assignments, because large data sets with many features can take a significant time to return results and this is wasteful of resources if some of those features do not perform any useful function. For example, using the full set of features detailed in section 4.2 (page 17) on a data set with 21,801 items results in the creation of feature vectors containing between 16 and 142 components. This becomes 61,459 binary vector components after the 1-of-K encoding process. Arranging these 21,801 binary vectors in a 61,459-dimensional space and determining k clusters takes approximately ten hours on a MacBook Pro with a quad-core 2.3GHz i7 processor and 16GB of memory. Reducing the number of components in the feature vectors reduces the number of binary vector components and therefore the computation time.

6.1.1 Design

A 263,614 instance sample was selected from the POS-tagged version of the British National Corpus, comprising newspaper articles relating to sport and business and used to generate a gold standard text (fig. 5). A dictionary of known variants, mapping 19,062 modern forms to 36,233 historical forms, was generated from data made available by the DICER project.⁴ The modern BNC data yielded 21,803 types, with an equal number of variants. 20,251 types were initially valid for variation – the forms within these types had at least three characters and were alphabetic, with no special characters. Vacuous variation was avoided by selecting types which contained more than one instance, whose form was in the known variant dictionary with at least two alternative forms available.

This resulted in 4,762 target types, 21.84% of the total. These types were induced to cause variation, redistributing their instances across more variants then changing the forms of the instances within those variants using the known variants dictionary. The number of variants within these types increased by 152.14% from 4,762 to 12,007. The targeted types changed from holding one variant each to between 2 and 12 variants, with targeted types containing on average 2.52 variants. Across the whole text, the number of variants increased by 33.23% from 21,803 to 29,048. By (6), the variation within the text changed from 0% to 24.94%. This level of variation would date the created text to the 17th century (see fig. 1, page 8). The result is

⁴<http://corpora.lancs.ac.uk/dicer/>

Ian Botham can't hev beene expectyng to hev such a vree wynter, bvt he's certainly fillyng it innovatively uuith a nationall tour of hs shevve, Ane Eueninge With Ian Botham. Clips of hs greatest hits, varous filmed interviews ond other odde surprizal gweste are promist, folowed by 'Bothy' taking questios from the floore. Audiences myghte like to quiz hime about hs favourite sportyng momente. Wes it þe 1974 Benson ond Hedges quarter-final whe, as a 15-year-old, he lost two teeths to te fearsome Andy Roberts before hitting phe winning sick?

Figure 5 – Sample of automatically generated artificial “historical” text

a large text with significant levels of variation, distributed unevenly across the types within the text. By comparison, the gold standard used in the VARD2 trials contained no more than 30,000 tokens (i.e. instances within the proposed formalism). The largest single text used for assessing Norma was 4,700 tokens, though five standards containing 15,509 tokens in total were used. The level of variation in those texts is unknown but can be estimated based on the age of the text, with more modern texts having less variation.

This data set was divided into blocks. Each block contained all the types whose variants shared an initial character, resulting in 24 blocks (no or very few words began with x or z). Each of these blocks was then processed multiple times by a parametrised instantiation of the TI-pipeline, using a different combination of features each time. Four binary features in sixteen combinations (table 9) per block gave a total of 384 trials. All blocks were processed concurrently on the Amazon EC2 cloud computing platform, with 32 Intel Xeon processors running at 2.8GHz and 60GB of RAM. Running multiple trials is required because the output of the agglomerative clustering algorithm is non-deterministic – an average is therefore needed. Each block was repeated ten times. Total computation time was approximately five hours.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bigram	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
POS	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
POS context	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
Pairwise comparison	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F

Table 9 – Matrix of parameters used for each block in experiment 1

6.1.2 Results

AMIS for each trial was calculated by comparing the predicted cluster assignments to the gold standard. p -scores were calculated to determine the correlation between AMIS and feature settings. The results are shown in table 10. It can be seen that both POS and POS context are *negatively* correlated with AMIS, harming the performance of the TI-pipeline. Removing these features from future experiments has two benefits. First, it should reduce the time taken to compute the types. Second, most crucially, it obviates any need for POS-tagged input. This allows the pipeline to adhere to the principle of pure input, since only unprocessed text is required.

	Bigrams	POS	Context	Pairwise comparison
<i>p</i> -score	0.197331736	-0.084754313	-0.223942882	0.797472663

Table 10 – *p*-scores for features

6.2 Experiment 2: Large-scale evaluation

Using the same automatically generated text from experiment 1 and a reduced feature set, the range of elements in each 21,801 feature vector fell from 16-142 to 12-31. The number of components in the equivalent binary feature vectors fell from 61,459 to 60,691. Computation time was not noticeably reduced.

6.2.1 Results

The AMIS for the output of the TI-pipeline, compared to the gold standard, was 0.949. This is extremely close to a score of 1 but it should be noted that AMIS does not heavily penalise the unnecessary division of sets of clusters. The score can be more accurately interpreted as meaning that the output of the TI-pipeline is very similar to the gold standard and significantly more similar than a random assignment of variants to types. The following experiment will use real historical texts and examine the actual output of the TI-pipeline in order to provide a more practical assessment of the procedure.

The text was processed again, this time using only the types to which variation had been added rather than all types. This is not a realistic process, as with a genuine historical text there would be no way to make this differentiation between types. It does, however, give an informative view of how successful the system is at identifying variant types without the additional noise of “singleton” types that contain no variation. On this data set, AMIS was 0.707 and therefore reasonably, but not completely, similar to the gold standard.

Overall, the identification pipeline is able to divide variants into types which are generally the correct size.

6.3 Experiment 3: Real historical text

The Paston letters are a collection of correspondence between the members of a 15th century Norfolk family, written between 1422 to 1510, and are available as part of the PCEEC (Taylor et al., 2006), which takes as its source the seminal edition by Davis (1971). There are two crucial differences between the Paston letters and an artificially generated text. First, there will be no gold standard to provide a ground truth. Secondly, the degree of variation is not known beforehand so the k value for the clustering algorithm is unknown. This value declares how many types should be identified within the data. There is no standard method for determining k but machine learning techniques are an exploratory tool, not explanations in themselves. Therefore, it is to be expected that a degree of experimentation is required.

An informed estimate for k is possible. First, it will lie within $[1, x]$ where x is the number of variants in the text, equal to the number of forms. The Paston letters contain 255,324 instances,

with 16,684 forms. Setting k closer to 1 will result in more variants per type. Setting it closer to x will result in fewer. Given a text from the 15th century, variation is likely to be high (as per fig. 1, page 8) and a suitable value for k for the Paston letters is therefore closer to 1 than to x . However, it does not make sense to set it too low – the letters are not a few types repeated over and over with thousands of different forms. By selecting a percentage of variation to expect (shown in fig. 1), (6) can be used to estimate how many types should be expected:

$$50\% = 100 - 100 \left(\frac{\text{types}}{16684} \right) \quad (7)$$

Solving for types gives 8,342. Values for k above and below this, 6,000 and 10,000 respectively, were also used for comparison purposes.

6.3.1 Results

Table 11 shows the different output of the TI-pipeline using different values for k , focusing on two possible types (*tidings* and *tiding*). A fuller representation of the output is presented in Appendix A. Several features are noteworthy. First, setting k too high can cause fragmentation of types whilst too low causes over-clustering. For the Paston letters, k is likely to lie between 6,000 and 8,342. Second, there is conflation of variants – the type identification process does not always distinguish between morphologically different forms (e.g. singular, plural, verbal inflection, adverbs). Third, some clusters contain forms which are arguably not variants of a type (*caused/accused/chased/cursed*), though many clusters are homogeneous.

6.4 Experiment 4: Chronology of the Great Vowel Shift

Whilst the ontological status of the Great Vowel Shift (GVS) is widely accepted, the details are disputed. It is conceptualised as either a push-chain (Luick, 1903), pull-chain (Jespersen, 1928), or centre drift (Stockwell and Minkova, 1988). Its chronology is also uncertain. Attempting to answer the latter, Lass (2000, Chapter 3) used the Paston Letters to show that /o:/ had begun raising to /u:/ by the 15th century. He reasoned that if /o:/ was raising, it might be written with <ou> for /u:/ in words that etymologically had /o:/. Examples of this in the Letters were given: *doun* and *goud* for *done* and *good*. Similarly, raising of /u:/ to /au/ is supported by *caw* (cow) and *ahawght*. Lass claims that the top half of the GVS was well under way in the 15th century, creating the conditions for a pull-chain.

None of these forms actually appear in the Paston Letters. They are not in the seminal Davis (1971) edition or the 1910 reprint of Gairdner (1895). The only examples of *doun* mean *down*. *goud* appears twice as *goude* but means *could*, written by Richard Calle, a servant of the Pastons. There are no instances of any of the others. Lass also claims that /i:/ had diphthongised in the Letters, evidenced by *abeyd* (abide) and *creying* (crying). Neither of these appear in the Letters, plus *crying* is of French or Latin origin.

Lass (personal communication April 23rd, 2014) attributes the error to “a conventional sloppiness, which was part of praxis then”. This highlights the need for corpus tools for dealing

	Type			Usage of form in context
tidinges	1	1	1	As for tidinges elles, þe Kyng is at Shene
tidingis	1	1	1	þat ye may bringe hym good tidingis
tidynges	1	2	2	And as for tidynges here in this contre
tidyngges	1	2	2	I merveyll that I here no tidyngges from yow
tidyngys	1	1	3	As for tidyngys in this contre her be noon
tyding	1	1	4	and tyding wheder þer be any sech sute
tydinges	1	2	2	As fore tydinges, here bee noon newe
tydingez	1	1	4	But of the fleet of shyppis there is no tydingez
tydingis	1	1	4	But þe continuall tydingis of my seyde lordis coming heder
tydingys	1	1	3	no tydingys in serteyn where the flet was
tydyng	1	2	5	As for tydynges, the Kyng and the counsell is at Northampton
tydynge	1	2	5	And as for tydynges, the berere heroff schall infforme yow
tydynges	1	2	2	As for tydynges, my lorde Erchebyssshop is at the Moore
tydyngges	1	2	2	As for tydyngges, my lord Chaunceler is discharged
tydynggis	1	2	5	and I praye yow to sende me tydynggis from be-yond see
tydynggys	1	2	5	As fore yowr syster, I can send yow no good tydynggys of here
tydyngis	1	2	5	I prey yow lete my mastras your modere knowe these tydyngis
tydyngys	1	2	5	As for tydyngys owte of thys contree
tythyngys	1	2	6	I vnderstond that ye had no tythyngys from me at that tyme
tytynges	1	2	2	schuld bere the Kyng certeyn lettres and juste tytynges
tytyngges	1	2	2	blyssed be God, that he hath now good tytyngges
tytyngys	1	2	6	And as for tytyngys, in good feyth we haue non

Table 11 – Comparison of type assignment for three values of k . Coloured columns, from left to right, $k=6,000$, $k=8,342$, $k=10,000$.

with historical texts. Without them, researchers may instead rely on old scholarly sources, repeating mistakes and using them to support arguments which are then accepted, uncritically, by others. Although there is a great deal of debate over the theoretical construction of the GVS, there is very little checking of the “facts”. They are taken at face value because they have been repeated so often in the literature.

To test Lass’s chronology, the Letters were processed using the TI-pipeline. Clusters were manually analysed to see which variants were grouped. These groups were then identified in the Letters and date of authorship recorded. The digraphs <aw> and <ou> were highlighted. Most of the 236 <aw> forms were either of French origin (commandment, ransom, fault), etymologically did not have long vowels (land, Canterbury, hawk, worship) or were not part of the same syllable (award, away). Of the 695 <ou> forms, a similar situation was found with mostly French words, as well as <ou> representing <ov> (overcome, recovery). This makes it difficult to support a claim that /o:/ had raised to /u:/. Dipthongisation of /i:/ is also not supported. All EME reflexes of OE *abidan* have <y> or <i> representing the stressed vowel: *abidith*, *abideth*, *abydyth*, *abydyn*, *abyd*, *abydyng*.

Setting aside Lass’s ghost words, the place name *Hellesdon* (OE: Hægelsdūn) is well-attested. According to Lass, the etymological /u:/ should be represented with a digraph, to show that it has dipthongised. The pipeline identified six variants for *Hellesdon*, across 53 usages. Of these, only 2 have <ou>: *Heylisdowne*. The rest have <o>: *Helesdon*, *Heylysdon*, *Heylesdon*, *Heilesdon*, *Heylisdon*. The <ou> variants were used in 1461 and 1465 by John Paston I, whilst the others are used between 1450 and 1476 by a range of authors. So even when

actual attestations are used to investigate Lass’s proposed GVS chronology, the claim that raising of the high vowels was “well under way by around 1400” (Lass, 2000, p.80) is not supported by the sources.

Establishing the extent of the GVS in 15th century Norfolk will require a full analysis of the historical sources. Such an analysis must cast aside traditional authorities on the evidence for sound change, to avoid repeating Lass’s error. Any theories of historical sound change would then need to fully account for the entire range of explicitly attested facts, rather than cherry-picking examples which neatly support a chosen theory. The reason that this investigation has not been done yet is likely due to the sheer work required. Fortunately, the TI-pipeline has shown that it can be useful in reducing the workload by making it easier to find orthographic variants without completely manual checking of sources.

7 Discussion

There are multiple explanations for the classification errors in the results of experiment 3. Since k is unknown, the partitioning of the metric space cannot be accurate. This naturally results in clusters being too small or too large. Only two features are used by the clustering algorithm (bigrams and pairwise similarity). More are likely to be needed, but these features could also be expanded, by using trigrams rather than bigrams or the entire set of pairwise comparisons rather than just the top ten.⁵ Another explanation may be that some types lack sufficient variants for the identification pipeline to capture them by pairwise comparison alone. Indeed, identifying types with many variants is a strength of the system.

The effectiveness of adding new features to overcome these issues would be easily assessed, using the demonstrated evaluation pipeline. Furthermore, the formalism can be used to guide feature selection and it predicts that semantic information would be highly informative though, as discussed, this may violate the principle of pure input by requiring semantic tagging. This itself would negatively impact assessment of least effort, but novel methods of automatically extracting any semantic information from raw historical texts could be developed to do this and incorporated into the TI-pipeline. For example, the latent semantic analysis methodology could be adapted to use the initial output of the pipeline in order to extract semantic relations, which is then reprocessed by the pipeline.

The formalism assumes that the latent variable in historical texts is types and that this hidden structure represents the “root” of variation (see fig. 2, page 15). In light of the morphological conflation observed, it may in fact be that lemmas are the latent variable. Any future work should investigate this, because it hints at a connection between the proposed formalism and the lexicon, in particular the nature of the lexicon’s contents. The lexicon is a core component of linguistic theory but is often taken for granted. The Declarative Procedural model (Ullman, 2001) merely stipulates it is part of the declarative system. The DRC model (Coltheart, Rastle, Perry, Langdon, and Ziegler, 2001) proposes a lexicon not only for sound but for orthography,

⁵Five were originally used, but the evaluation pipeline showed ten to be more effective. More were not used due to the increased computation time.

but these too are underspecified. Models with orthography in the lexicon (e.g. Seidenberg and McClelland, 1989) suffer from a synchronic bias because orthography, standard or otherwise, is a modern phenomenon. It seems unlikely that people in the 15th century stored multiple orthographic forms in their lexicons, selecting one at random. Rather, some psycholinguistic process for generating orthographic forms from the lexicon is implicated. Centuries later, we still have access to that process. This is evidenced by the ease with which modern readers “decode” historical forms and determine what modern forms they correspond to, even out of context but especially if there is semantic understanding for that form.⁶ Within the proposed formalism, this may be accounted for by a decoding process using type-like information stored in the lexicon. Under this account, excluding orthographic information from the lexicon is as justified as excluding different fonts and font sizes.

⁶*retorne, peopell, bownde, contynew* are easy, *yepysweche, vowchesawf, scutys* are trickier.

8 Conclusion

A working solution to one half of the original variation problem has been presented. It was evaluated statistically and experimentally. It meets many desiderata (table 12). The proposed pipelines, which are fully automated and require no pre-processing of the source, could be the first step to investigating linguistic phenomena in historical texts on a scale previously not possible due to the labour involved – as demonstrated by the investigation of the Great Vowel Shift’s chronology. Everything is based on a rigorous formalism, an approach that yielded not only a novel view on an old problem but a precise technical vocabulary and a framework within which future improvements can be situated. Finally, analysis of the experimental results suggests that the formalism may be able to draw a connection between historical developments in orthography and psycholinguistic processes involving literacy and the lexicon.

Perfection	No. Statistically strong results but type clusters often contain morphological variants. Clusters generally too small rather than too large. Does not normalise text.	See: section 6.2 section 6.3
Least effort	Completely automatic with no human input needed. No training needed. No preprocessing needed. Complexity $O(n^3)$ for agglomerative clustering, taking approximately ten hours on 250,000 tokens. Speed is determined by hardware power but polynomial complexity makes larger data sets very slow.	See: section 6.1
Explanatory adequacy	Completely grounded in a theoretical formalism which makes testable predictions as to the underlying structure of texts. Uses suitable technology, critically chosen. Leverages form-internal structure (bigrams) to access an underlying proposed type (possible lemma) structure. Can be further adapted to better capture linguistic structure.	See: section 3 section 4.2
Pure input	Yes. Requires only a historical source. POS-tagging is not required. No list of known variants is required.	See: section 6.1.2
Language agnostic	No language-specific algorithms used – pairwise comparison can be performed with any string metric suited to the source language.	See: section 4.2.4
Epoch ignorant	Yes. No modern dictionaries used. No comparisons to modern words made. Only within-source comparisons are used.	See: section 4.1 section 4.2 section 6.3
Preserve output	Yes. Returns a list of identified types without modifying the source.	See: section 4.1 section 6.3 section 6.4

Table 12 – Appraisal of type identification pipeline against desiderata

A Experiment 3 - partial results

Table 13 – Type identification pipeline output for various k values (157 out of 16,684)

$k = 6000$		$k = 8342$		$k = 10000$	
FORM	CLUSTER	FORM	CLUSTER	FORM	CLUSTER
reuell	0	spekyth	0	charyté	0
rewell	0	specyfyith	0	charité	0
revell	0	spedyth	0	juge	1
haxed	1	bowe	1	jugeis	1
hadden	1	bower	1	juges	1
hadde	1	bowez	1	dysshes	2
haden	1	bowen	1	dysseys	2
hade	1	thret	2	panell	3
hadd	1	thrett	2	inpanell	3
nepueu	2	threte	2	dowtere	4
neuw	2	requyred	3	dowgtere	4
nevewes	2	requyre	3	dowgteris	4
nevoue	2	requryrd	3	babees	5
nevewe	2	decesse	4	bees	5
nevew	2	decesed	4	advyse	6
neveu	2	decese	4	avysed	6
gown	3	deceasse	4	advysed	6
drownk	3	blodys	5	comown	7
down	3	loodys	5	comonys	7
nown	3	londys	5	comons	7
fown	3	tydyngys	6	comowns	7
mown	3	tydyng	6	comounly	7
town	3	tythyngys	6	comoun	7
dowun	3	tydynggis	6	comon	7
novembre	4	tytyngges	6	comouns	7
nombere	4	tydinges	6	pidre	8
nowmbere	4	tydyngges	6	paire	8
nouembere	4	tidynges	6	fauorabelye	9
nouember	4	tydynggys	6	fauorably	9
noumbir	4	tidyngges	6	favorably	9
nouembre	4	tytyngys	6	fauorabely	9
streyned	5	tytynges	6	countie	10
strekyn	5	tydynges	6	counties	10
streyne	5	tydyngis	6	countes	10

streuyn	5	tydyng	6	joporté	11
tenurys	6	ordynatlye	7	joparté	11
termys	6	ordynaunce	7	parté	11
lityrys	6	ordynarylye	7	jouparté	11
terys	6	ordynarye	7	staple	12
overe	7	ordynare	7	estaple	12
overcome	7	ordynawnce	7	purvay	13
lovere	7	wroote	8	purvayd	13
wasted	8	woote	8	puruay	13
wast	8	wote	8	purway	13
wastes	8	wrote	8	amendid	14
wrighte	9	sowter	9	amendyt	14
almightie	9	somwher	9	amendyd	14
wighte	9	passage	10	lenn	15
righte	9	massage	10	lern	15
right	9	masage	10	moore	16
might	9	honyng	11	moor	16
wright	9	hoonyng	11	jopart	17
feight	9	ponyng	11	joparte	17
almighti	9	gronyng	11	part	17
wight	9	mariages	12	menys	18
sighte	9	mariage	12	mennys	18
tight	9	maria	12	meanys	18
plighte	9	schene	13	recueryd	19
sight	9	scheen	13	recoueryd	19
lightly	9	scheppe	13	recoweryd	19
light	9	scheche	13	recueryd	19
ligh	9	schete	13	dalyauns	20
esqwyr	10	schetes	13	alyans	20
esqvyer	10	schepe	13	oweth	21
esqwyrys	10	schreff	14	þowth	21
sqwyer	10	scherif	14	owthg	21
swyr	10	schref	14	owith	21
sqvyer	10	sheryff	14	bowth	21
sqwyerys	10	scherryf	14	owth	21
sqwyr	10	scheryff	14	owt(yogh)	21
esqwyere	10	scheryf	14	mowth	21
sqwyirs	10	dysposed	15	ermyngnam	22
sqwyeres	10	dyspose	15	erpyngnam	22

sqwyere	10	dysspytuose	15	gernyngham	22
store	11	cast	16	gemyngham	22
sorepe	11	casty	16	elyngham	22
sorer	11	castyr	16	joystys	23
sore	11	moungomerye	17	jostys	23
soore	11	mongomeré	17	joperté	24
varyaunce	12	mongomery	17	pouerté	24
clarance	12	montgomery	17	jupperté	24
variaunce	12	wyte	18	mysch	25
vareaunce	12	wytte	18	fysch	25
varyance	12	wyttes	18	ocle	26
claraunce	12	qwyte	18	cole	26
deserve	13	daunger	19	ofys	27
desyere	13	danger	19	otys	27
desyerthe	13	dawnger	19	profet	28
desyered	13	dangere	19	proff	28
desert	13	dawngeres	19	profett	28
deserue	13	wryton	20	smyth	29
accusyd	14	wryting	20	symyth	29
causyd	14	wrytin	20	syth	29
chasyd	14	hate	21	comeners	30
crasyd	14	thate	21	comenere	30
cursyd	14	bull	22	avayll	31
cawsyd	14	bullok	22	disavayll	31
jnquisytyff	15	jull	22	leve	32
jnquietyng	15	juyll	22	leven	32
jnquest	15	decembyre	23	talkyd	33
jnquyet	15	december	23	balkyd	33
qhen	16	indytement	24	walkyd	33
whens	16	jndytementés	24	þedyr	34
when	16	indytementés	24	þedyre	34
sesyn	17	inditementés	24	athwyx	35
soleyn	17	inditementis	24	atwyx	35
sleyn	17	jnditement	24	breest	36
seyn	17	inditementys	24	beest	36
borowyng	18	writ	25	secretory	37
brochyng	18	write	25	secratory	37
vndereschetour	19	writell	25	wordy	38
vnderstewerd	19	writene	25	wordys	38

busche	20	writen	25	woordys	38
siche	20	fumowsly	26	beffore	39
svche	20	fumows	26	before	39
musche	20	dystressyd	27	beffoore	39
schep	20	dysstres	27	beffor	39
soche	20	dystressys	27	pache	40
suche	20	shepherd	28	hache	40
sche	20	shepherd	28	fore	41
mette	21	schepes	28	force	41
wette	21	schepes	28	afore	41
yette	21	schepes	28	forre	41
detteez	21	goone	29	baptiste	42
dette	21	agone	29	baptyste	42
king	22	agoone	29	withowt	43
þing	22	agonye	29	withowte	43
kingz	22	curs	30	without	43
keping	22	curse	30	without	43
soch	23	colett	31	withouten	43
such	23	colyet	31	withowth	43
schuch	23	allweys	32	freyir	44
recouere	24	always	32	frier	44
recouery	24	alweye	32	freir	44
recure	24	alweyis	32	freyr	44
recuuered	24	alwey	32	payth	45
recuere	24	allwey	32	fayth	45
recouer	24	vicar	33	thinking	46
recouered	24	vycare	33	thinkith	46
strawngenes	25	vycar	33	thinketh	46
strawnge	25	conseill	34	thinkyth	46
strange	25	counsellour	34	think	46
strangwysse	25	counseyll	34	launde	47
straungerys	25	consell	34	lawnde	47
strawngely	25	conseil	34	landes	47
straunge	25	counsellid	34	lande	47
straung	25	counsel	34	writtes	48
strangely	25	counsell	34	writtis	48
straungiers	25	conseille	34	writt	48
strawngys	25	counseyle	34	written	48
straunger	25	counseal	34	writte	48

straungely	25	conseillour	34	writtys	48
pasturys	26	cownsell	34	roosting	49
pastures	26	counseil	34	bosting	49
pasturs	26	counsell	34	repayment	50
pasturen	26	counseyl	34	areyment	50

B Code: type identification pipeline

```
1 from nltk.tag import str2tuple
2 from collections import Counter, namedtuple, defaultdict, OrderedDict
3 from sklearn.feature_extraction import DictVectorizer
4 from copy import deepcopy
5 import pickle
6 from operator import itemgetter
7 import os
8 from itertools import product
9 import csv

11 from sklearn.cluster import AgglomerativeClustering as AC

13 from datetime import datetime as DT

15 class Corpus:
16     def __init__(self, **kwargs):
17         self.init_settings = OrderedDict()
18         """
19         Expected argument is just filename at this point.
20         **kwargs is a bit overkill, left over from previous versions
21         """
22         for k,v in kwargs.items():
23             setattr(self, k, v)
24             self.init_settings[k] = v
25
26     def process_paston_data(filename):
27         """
28         This takes the raw tagged Paston data from the PCEEC and converts
29         it to a list of (word, POS) tuples. Inserts start/end padding for
30         sentences.
31         """
32         with open(filename) as file:
33             raw_text = file.read()
34             letters_temp = raw_text.split('<Q')
35             letters_temp.pop(0)
36             letters = ["<Q"+i for i in letters_temp]
37             letters2 = [i.splitlines() for i in letters]
38             letters3 = [i[8::6] for i in letters2]
39             for letter in letters3:
40                 for index, sentence in enumerate(letter):
41                     letter[index] = sentence.split()
42             for l in letters3:
43                 for s in l:
44                     s.append("ENDPAD_END")
45                     s.insert(0, ("STARTPAD_START"))
46             data = []
47             for letter in letters3:
```

```

49         for sent in letter:
            data.append(sent)
data2 = []
51 for i in range(0, len(data)):
    data2.append([str2tuple(x, sep="_") for x in data[i]])
53 data3 = []
    for sent in data2:
55         for pair in sent:
            data3.append(pair)
57 print('Processing', filename)
    return [(x.lower(), y) for (x, y) in data3]
59
def process_bnc_data(filename):
61     """
        Loads the ~250k sample modern BNC text as (word, POS) tuples
63         and adds start/end padding for sentences.
        """
65     with open(filename, 'rb') as file:
        raw_data = pickle.load(file)
67     for s in raw_data:
        s.append(("ENDPAD" , 'END'))
69         s.insert(0, ("STARTPAD" , 'START'))

71     data1 = []
    for s in raw_data:
73         for pair in s:
            data1.append(pair)
75     print('Processing', filename)
    return [(x.lower(), y) for (x, y) in data1]
77

if self.filename == 'corpora/paston':
79     """
        Processes Paston/BNC/Variation data, sets up some parameters
81         for stats which are used later.
        """
83     self.data = process_paston_data(self.filename)
    self.filename_nopath = self.filename.split('/')[1]
85     self.init_settings['filename'] = self.filename_nopath
if self.filename == 'corpora/bnc':
87     self.data = process_bnc_data(filename=self.filename)
    self.filename_nopath = self.filename.split('/')[1]
89     self.init_settings['filename'] = self.filename_nopath
if self.filename not in ['corpora/paston', 'corpora/bnc']:
91     self.data = self.filename
    self.filename_nopath = "VarObjOutput"#FIX THIS AT SOME POINT
93     self.init_settings['filename'] = self.filename_nopath

95 self.filename_for_saving = self.generate_filename()

```

```

97     print('Creating word list')
99     self.word_list = list(Counter([i[0] for i in self.data
100                                if i[1] not in ['FW', 'NUM']]).keys())
101     self.word_list = [i for i in self.word_list if i not in
102                      ['startpad', 'endpad'] and len(i) > 3 and i.isalpha() == True]
103     print('Word list has', len(self.word_list), 'entries')

105     def populate_jwd(wordlist):
106         """
107         Called if using Variation or BNC data. Says JWD but actually
108         uses JD!
109         Loads a pre-compiled list of JD comparisons, because computing
110         them at call time takes hours.
111         """
112         print('Loading big JWD store - takes a while!')
113         with open('corpora/jwd_store.pickle', 'rb') as file:
114             storage = pickle.load(file)
115         print('Loaded!')
116         jwd_data = {}
117         print('Extracting JWD data using wordlist:',
118               len(self.word_list),
119               'to process')
120         for word in wordlist:
121             if len(storage[word]) > 10:
122                 jwd_data[word] = sorted([i for i in storage[word].items()],
123                                         key=itemgetter(1), reverse=True)[0:10]
124             else:
125                 jwd_data[word] = [i for i in storage[word].items()]

126         print('JWD data all processed!')
127         return jwd_data

129     """
130     The next part loads the right JD data file for the input.
131     Paston data is also precompiled.
132     """

133
134     if self.filename == 'corpora/paston':
135         with open('data/paston_jaro_50.pickle', 'rb') as file:
136             self.jwd_data = pickle.load(file)

137
138     if self.filename == 'corpora/bnc':
139         self.jwd_data = populate_jwd(self.word_list)

140
141     if self.filename not in ['corpora/paston', 'corpora/bnc']:
142         self.jwd_data = populate_jwd(self.word_list)

143
144     def padded_ngram_dict(word, n=2):

```

```

147     """
148     Ngram generator.
149     Takes a word and returns a dictionary where:
150     k = character bigram with padding (cat > $cat$ > $c, ca, at, t$)
151     v = count of each bigram in the word (usually=1)
152     Can parameterise for n (2 = bigrams, 3 = trigrams etc)
153     """
154     def bigrams_with_padding(word):
155         store = []
156         current_index = 0
157         padded = '$'*(n-1) + word + '$'*(n-1)
158         for i in range(0, len(padded)+1-n):
159             store.append(padded[i:current_index+n])
160             current_index += 1
161         return store
162
163     def padd_list_to_dict(list):
164         bigramdict = {}
165         for item in list:
166             bigramdict[item] = bigramdict.get(item, 0) + 1
167         return bigramdict
168
169     return padd_list_to_dict(bigrams_with_padding(word))
170
171     def compile_labels_and_features(
172         remove_empty=True,
173         length=False,
174         bigrams=self.bigrams):
175         """
176         This turns each word into a feature vector.
177         Returns two matched lists.
178         First one is the words (i.e. labels)
179         Second one is the feature vectors for that word.
180         """
181         raw_features = []
182         labels = []
183
184         for word in self.word_list:
185             store = []
186             combo = {}
187             labels.append(word)
188
189             if length == True:
190                 """
191                 Include word length as a feature. Dumb idea.
192                 """
193                 store.append({'wordlength': len(word)})

```

```

195         for jwditem in self.jwd_data[word]:
196             """
197             Include JD features (not JWD)
198             """
199             store.append({jwditem[0]:jwditem[1]})
200
201         if bigrams == True:
202             """
203             Include bigram features
204             """
205             store.append({k:v for k,v
206                           in padded_ngram_dict(word, n=2).items()})
207
208         for d in store:
209             for k,v in d.items():
210                 combo[k] = v
211             raw_features.append(combo)
212
213         if remove_empty == True:
214             """
215             Used to remove any raw components that are equal to 0.
216             Deprecated, I think?
217             """
218             rf2 = []
219             for i in raw_features:
220                 temp = {}
221                 for k,v in i.items():
222                     if v not in [0,0.0]:
223                         temp[k] = v
224                 rf2.append(temp)
225
226             raw_features = rf2
227
228         return labels, raw_features
229
230     print('Creating label list')
231
232     self.labels, self.raw_features = compile_labels_and_features()
233     """
234     Use DictVectorizer to normalise the feature vectors for use in sklearn
235     using 1-of-k method.
236     """
237     self.vectoriserObject = DictVectorizer(sparse=False)
238     print('All done')
239
240     def find_by_start(self, n, vectors=True):
241         """
242         Returns labels and features for words beginning with n
243         n can be one char or more.

```

```

'''
245 l = []
246 f = []
247 for i,x in enumerate(self.labels):
248     if x[0][0:len(n)] == n:
249         l.append(x)
250         f.append(self.raw_features[i])
251 if vectors == True:
252     return l, self.vectoriserObject.fit_transform(f)
253 else:
254     return l, f
255
def find_by_list(self, items, vectors=True):
256     '''
257     As above, but takes list of items as input and returns the features for
258     them. Vectorises them to binary.
259     '''
260     l = []
261     f = []
262     for index, label in enumerate(self.labels):
263         if label in items:
264             l.append(label)
265             f.append(self.raw_features[index])
266
267     if vectors == True:
268         return l, self.vectoriserObject.fit_transform(f)
269     else:
270         return l,f
271
def dump(self, labs, vects, filename):
272     '''
273     Saves labels and vectors to file.
274     '''
275     fname = 'csv/' + filename + ".csv"
276     with open(fname, 'w') as file:
277         out = csv.writer(file, delimiter="," , quoting=csv.QUOTE_ALL)
278         temp = zip(labs,vects)
279         temp = sorted(temp, key=itemgetter(1))
280         for i in temp:
281             out.writerow(i)
282     print('Wrote to', fname)
283
284
def generate_filename(self):
285     '''
286     Generates a filename for saving data to disk. Includes date and time
287     plus parameters from the model.
288     '''
289     namestring = [str(k) + '=' + str(v)
290                  for k,v in self.init_settings.items()]

```



```

293         return " ".join(namestring)

295     def do_KM(self, letter, k, dump=True):
296         """
297         This is called KM but actually uses AC!
298         Does the clustering.
299         Takes either a letter of the alphabet or a list.
300         If letter, clusters all words in the wordlist beginning with that. Used
301             for prototyping.
302         If list, clusters all the words in that list.
303         Dumps a CSV to disk, containing two columns. First is word, second is
304             cluster number.
305         """
306         clusts = k

307
308         if type(letter) == str:
309             l,v = self.find_by_start(letter)

310
311             self.km_obj = AC(n_clusters=k, linkage='average')

312
313             results = self.km_obj.fit(v)

314
315             if dump == True:
316                 filename = DT.now().strftime('%d%m%y-%H%M%S')
317                     + self.filename_for_saving + '-k' + str(clusts)
318                 self.dump(l, results.labels_, filename)
319                 self.km_labels = l
320                 self.km_clusters = results.labels_
321                 print('Stored KM output in self.km_labels, self.km_clusters')
322             else:
323                 self.km_labels = l
324                 self.km_clusters = results.labels_
325                 print('Stored KM output in self.km_labels, self.km_clusters')

326
327         if type(letter) == list:
328             l,v = self.find_by_list(letter)

329
330             self.km_obj = AC(n_clusters=k, linkage='average')

331
332             results = self.km_obj.fit(v)

333
334             self.predicted_clusters = {}

335
336             for v,k in zip(l, results.labels_):
337                 self.predicted_clusters[v] = k

338
339             self.pred_prep = sorted(self.predicted_clusters.items(),
340                                     key=itemgetter(0))
341             self.predicted = [i[1] for i in self.pred_prep]

```

```
343         if dump == True:
344             filename = DT.now().strftime('%d%m%y-%H%M%S')
345                 + self.filename_for_saving + '-k' + str(clusts)
346             self.dump(1, results.labels_, filename)
347             self.km_labels = 1
348             self.km_clusters = results.labels_
349             print('Stored KM output in self.km_labels, self.km_clusters')
350         else:
351             self.km_labels = 1
352             self.km_clusters = results.labels_
353             print('Stored KM output in self.km_labels, self.km_clusters')
```

identification.py

C Code: gold standard generation pipeline

```
from collections import defaultdict, Counter
2 from string import ascii_letters, ascii_lowercase, digits
from itertools import islice, product
4 import random
import pandas as pd
6 import pickle
from operator import itemgetter
8 from statistics import *

10 class Instance:
    '''
12     Equivalent to an instance from the formalism.
    Has three properties (form, position, POS)
14     POS is not actually used though.
    '''
16     def __init__(self, form, position, PoS):
        self.form = form
18         self.position = position
        self.PoS = PoS
20
22     def get_position(self):
        return self.position
24     def get_form(self):
        return self.form
26
28     def __getitem__(self, key):
        return [self.form, self.position][key]
30
32     def __repr__(self):
        return "{}: {} {}".format(self.__class__.__name__,
            self.form,
            self.position)
34 class Container:
    '''
36     Base Class for something that will hold other stuff. Sets the basic
    properties of a container.
38     '''
40     def __init__(self, *args):
        self.contents = []
42         if len(args) > 0 and type(args[0]) == list:
            for i in args[0]:
44                 self.contents.append(i)
        else:
46             if len(args) > 0:
                self.contents.extend(args)
```

```

48
def add(self, *args):
50     if type(args[0]) == list:
52         for i in args[0]:
54             self.contents.append(i)
56     else:
58         self.contents.extend(args)

def __len__(self):
60     return len(self.contents)

def __getitem__(self, key):
62     return self.contents[key]

def __setitem__(self, key, value):
64     self.contents[key] = value

def __repr__(self):
66     if len(self.contents) == 0:
68         return 'An empty container'
69     else:
70         return "{}: holding {} itemsof class {}".format(
71             self.__class__.__name__, len(self.contents),
72             self.contents[0].__class__.__name__)

class Variant(Container):
74     '''
75     As per the formalism.
76     Inherits from Container object but overrides 'add' method, to set
77     the Variant label based on form of its contents.
78     Also has split() method, used when variation is caused.
79     '''
80     def __init__(self, *args):
81         super().__init__(*args)
82         if len(self) > 0:
83             self.name = self.contents[0].form

def add(self, *args):
86     if type(args[0]) == list:
88         for i in args[0]:
89             self.contents.append(i)
90     else:
91         self.contents.extend(args)
92     if len(self) > 0:
93         self.name = self.contents[0].form

def __repr__(self):
94     if len(self) == 0:
96         return "An empty Variant"

```

```

else:
    return "A {} with name {} containing {} Instances".format(
        self.__class__.__name__, self.name, len(self))

def split(self, known_variants, ID):
    if len(self.contents) == 1:
        return self

    N = [i for i in known_variants[self.name]]

    N.append(self.name)

    V = [i for i in self.contents]

    min_cuts = 1

    if len(V) >= len(N):
        if len(N) == 2:
            max_cuts = 1
        else:
            max_cuts = len(N) - 1
    if len(V) < len(N):
        if len(V) == 2:
            max_cuts = 1
        else:
            max_cuts = len(V) - 2

    if max_cuts == 1:
        num_cuts = 1
    else:
        num_cuts = random.randint(min_cuts, max_cuts)

    if num_cuts == 1 and len(V) == 2:
        cut_pos = [1]
    else:
        cut_pos = random.sample(range(1, len(V)), num_cuts)

    cut_pos = sorted(cut_pos)

    cut_pos.insert(0, 0)

    cut_pos.append(len(V))

    new_chunks = []

    for i, c in enumerate(cut_pos[0:-1]):
        new_chunks.append(V[c:cut_pos[i+1]])

    selected_new_names = random.sample(N, len(new_chunks))

```

```

146     for pair in zip(selected_new_names, new_chunks):
148         for inst in pair[1]:
149             inst.form = pair[0]
150
151     store = []
152     for x in new_chunks:
153         store.append(Variant(x))
154     for var in store:
155         var.ownerID = ID
156
157     return store
158
159 class Type(Container):
160     """
161     Inherits from base Container class. Doesn't override anything.
162     Extends functionality with methods for recording, causing and reporting
163     variation within its contents.
164     """
165     def __init__(self, *args):
166         super().__init__(*args)
167         self.ID = "".join(
168             [random.choice(ascii_letters+digits) for x in range(15)])
169         for variant in self.contents:
170             variant.ownerID = self.ID
171         self.branches, self.leaves = self.get_foliage()
172
173     def get_foliage(self):
174         return len(self), sum([len(i) for i in self.contents])
175
176     def update_foliage(self):
177         self.branches, self.leaves = self.get_foliage()
178
179     def induce_split(self, known_variants):
180         self.pre = {}
181         self.pre['Total Leaves'] = sum([len(v) for v in self.contents])
182         self.pre['Total Branches'] = len(self.contents)
183         self.pre['Average Leaves per
184             Branch'] = self.pre['Total Leaves']/self.pre['Total Branches']
185
186         if sum([
187             len(i) for i in self.contents
188             ]) > 1 and self.contents[0].name in known_variants:
189             x = self.contents[0].split(known_variants, ID=self.ID)
190             self.add(x)
191             self.contents = self.contents[1:]
192             self.update_foliage()
193         else:
194             pass

```

```

196     self.post = {}
197     self.post['Total Leaves'] = sum([len(v) for v in self.contents])
198     self.post['Total Branches'] = len(self.contents)
199     self.post['Average Leaves per Branch'] = self.post['''Total
200         Leaves'''] / self.post['Total Branches']
201     self.post['% Leaves per Branch'] = 100-(self.post['''Average
202         Leaves per Branch'''] / self.pre['Average Leaves per Branch'] * 100)
203     self.post['Variety % increase'] = 100*(self.post['''Total
204         Branches'''] - self.pre['Total Branches'])

206     def report_variation(self):
207         return 1.0/(float(len(self)))

208
209     def get_leaf_names(self):
210         return [v.contents[0].form for v in self.contents]

211
212     def __repr__(self):
213         return "{} {} ID {} branches {} leaves".format(
214             self.__class__.__name__, self.ID, self.get_foliage()[0],
215             self.get_foliage()[1])
216
217
218 class Text:
219     def __init__(self, *source, letter=None,
220         known_variants='data/knownvariants.pickle', aws=False):
221         '''
222         Creates Text object, based on formalism.
223         AWS settings are left over from file access issues on Amazon because
224         I couldn't quickly find a way to deal with concurrent file access.
225         This should always be set to false now.
226         knownvariants.pickle is the dictionary of known variants generated
227         from the DICER website.
228         In addition to being the top level object, it tracks all the statistics
229         for variation.
230         '''
231
232         print('Importing known variants data')
233         if aws == True:
234             self.known_variants = pickle.load(open(
235                 known_variants + letter.upper(), 'rb'))
236         if aws == False:
237             self.known_variants = pickle.load(open(
238                 known_variants, 'rb'))
239
240         self.source_data = source
241         self.data = []
242
243         for i in self.source_data:

```

```

244         for x in i:
245             self.data.append(x)
246
247     self.word_counts = dict(Counter(
248         i[0] for i in self.data if i[0].isalpha() == True and len(i[0])>3))
249
250     if source:
251
252         print('Extracting Instances from source')
253         self.contents = [Instance(f[0],
254             p, f[1]) for i in source for p,f in enumerate(i)]
255
256         print('Folding Instances into Variants')
257         self.variants = defaultdict(Variant)
258         for inst in self.contents:
259             self.variants[inst.form].add(inst)
260
261         self.variants = dict(self.variants)
262
263         print(''''Constructing vocabulary of forms
264             with [a-z] as first character''')
265         self.vocabulary = set(
266             [i.form for i in self.contents if i.form[0] in ascii_lowercase])
267
268         print(''''Generating Types from Instances: splitting
269             into in-vocab Types and junk Types''')
270         self.junk_types = set()
271         self.alpha_types = defaultdict(list)
272         for k,v in self.variants.items():
273             if v.name in self.vocabulary:
274                 self.alpha_types[k[0]].append(Type(v))
275             else:
276                 self.junk_types.add(Type(v))
277
278         self.alpha_types = dict(self.alpha_types)
279
280         print('Merging junk Types and in-vocab types as all_types')
281         alphaset = set()
282         for i in self.alpha_types.values():
283             for x in i:
284                 alphaset.add(x)
285         self.all_types = self.junk_types.union(alphaset)
286
287     else:
288         '''
289         Used to create a blank text for testing purposes.
290         '''
291         self.contents = []
292         self.target_instances = []

```



```

294         self.vocabulary = []
295         self.variants = []
296         self.types = []
297
298         self.initial_variation = self.calculate_variation()
299
300     def __len__(self):
301         return len(self.contents)
302
303     def calculate_variation(self, alpha=True):
304         if alpha == True:
305             var_counts = [
306                 x.report_variation() for t in self.alpha_types.values() for x in t]
307         else:
308             var_counts = [t.report_variation() for t in self.all_types]
309         return pd.DataFrame(var_counts)
310
311     def cause_variation(self, target, splittable_only=True, no_singletons=True):
312         self.info = {}
313         self.info['target letter'] = target
314
315         self.variation_info = {}
316
317         if target == 'ALL':
318             all_targets = [
319                 v for k,v in self.alpha_types.items() if k not in [
320                     'x','z'] for v in v]
321         if target != 'ALL' and target not in ['x','z']:
322             all_targets = self.alpha_types[target]
323         if target in ['x','z']:
324             print('Cannot cause variation on' + target + '''because
325                 no available known_variants''')
326             raise ValueError("Target cannot be X or Z")
327
328         if splittable_only == False and no_singletons == False:
329             self.valid_targets = all_targets
330
331         if splittable_only == True and no_singletons == True:
332             self.valid_targets = [t for t in all_targets if len(
333                 t.contents[0].name) > 3 and t.contents[0].name in \
334                 self.known_variants and \
335                 self.word_counts[t.contents[0].name] > 1]
336
337         if splittable_only == False and no_singletons == True:
338             self.valid_targets = [t for t in all_targets if t.leaves > 1]
339
340         if splittable_only == True and no_singletons == False:
341             self.valid_targets = [t for t in all_targets if \
342                 t.contents[0].name in self.known_variants]

```

```

342     # if target == 'ALL' and len(self.valid_targets) > 2000:
343     #     self.valid_targets = random.sample(self.valid_targets, 2000)
344
345     self.info['splittable_only'] = splittable_only
346     self.info['no_singletons'] = no_singletons
347
348     self.info['Max Types available'] = len(all_targets)
349     self.info['# Variants in max Types'] = len([t.branches \
350         for t in all_targets])
351
352     self.info['# pre-var Type targets/Variants per Type'] = sum(
353         [t.branches for t in self.valid_targets])
354
355     for t in self.valid_targets:
356         t.induce_split(self.known_variants)
357
358     self.variation_info['selection'] = self.valid_targets
359
360     self.info['# Types post-var'] = len([t for t in self.valid_targets])
361     self.info['# Variants in Types post-var'] = sum(
362         [t.branches for t in self.valid_targets])
363
364
365     self.expected_clustering = []
366     for t in self.variation_info['selection']:
367         self.expected_clustering.append(t.get_leaf_names())
368
369     self.words = []
370     for l in self.expected_clustering:
371         for x in l:
372             self.words.append(x)
373
374     for i in range(len(self.expected_clustering)):
375         for index, name in enumerate(self.expected_clustering[i]):
376             self.expected_clustering[i][index] = i
377
378     self.expected_temp = {}
379     for pair in zip(self.words, self.expected_clustering):
380         self.expected_temp[pair[0]] = pair[1]
381
382     self.expected_temp2 = sorted(self.expected_temp.items(), key=itemgetter
383 (0))
384
385     self.expected1 = [i[1] for i in self.expected_temp2]
386
387     self.expected = []
388     for i in list(self.expected_temp.values()):
389         for x in i:

```

```

390         self.expected.append(x)

392     self.k_to_use = len(self.valid_targets)

394

396     self.info['k value'] = self.k_to_use
397     self.info['Min +var cluster size'] = min([len(i) \
398         for i in self.expected_temp.values()])
399     self.info['Max +var cluster size'] = max([len(i) \
400         for i in self.expected_temp.values()])
401     self.info['Population variance +var cluster size'] = \
402         pvariance([len(i) for i in self.expected_temp.values()])
403     self.info['Average +var cluster size'] = sum([len(i) \
404         for i in self.expected_temp.values()])/len([len(i) \
405         for i in self.expected_temp.values()])
406     self.info['Ratio of Types to Variants after +var'] = \
407         self.k_to_use/self.info['# Variants in Types post-var']
408     self.info['Ratio of +var Types to all Types'] = \
409         self.info['# pre-var Type targets/Variants per Type'] / \
410         self.info['Max Types available']

411
412     def find(self, target):
413         store = []
414         for t in self.all_types:
415             if t.contents[0].name == target:
416                 store.append(t)
417         return store

418     def output(self):
419         store = []
420         for inst in self.contents:
421             store.append((inst.form, inst.PoS, inst.position))
422     store = sorted(store, key=itemgetter(2))
423     return [(i[0], i[1]) for i in store]

```

goldstandard.py

D Code: evaluation pipeline

```
1 from CorpusObject1 import Corpus
2 from VariantObjects import Instance, Container, Variant, Type, Text
3 from itertools import product
4 from sklearn.metrics.cluster import *
5 import pandas as pd
6 from collections import OrderedDict
7 from operator import itemgetter
8 from datetime import datetime as DT
9
10 def process(letter, sources, alg='AC', use_links='average', aws_setting=False,
11            use_affinity='euclidean'):
12     """
13     This runs various parameterised versions of the Corpus object and extracts
14     the clustering output. Then it calculates various scores by comparing the
15     output to the expected gold standard.
16     This version here only works for the final model that uses JD info and
17     allows
18     bigrams to be turned on and off.
19     Saves it all to a pandas dataframe which can then have things like p-scores
20     done on it.
21     """
22
23     metrics = [adjusted_mutual_info_score, adjusted_rand_score,
24                mutual_info_score, normalized_mutual_info_score]
25
26     data = {letter:{}}
27
28     if aws_setting == True:
29         data[letter]['Post-var Text Object'] = [Text(source,
30               letter=letter, aws=True) for source in sources]
31     if aws_setting == False:
32         data[letter]['Post-var Text Object'] = [Text(source,
33               letter=letter, aws=False) for source in sources]
34
35     for text in data[letter]['Post-var Text Object']:
36         text.cause_variation(letter)
37
38     data[letter]['Post-var Output'] = [x.output() for x in \
39           data[letter]['Post-var Text Object']]
40     settings = [list(i) for i in product(letter, [True,False])]
41
42     data[letter]['settings'] = settings
43
44     results = OrderedDict()
45     results[letter] = OrderedDict()
46     for x in results:
47         for i in range(len(settings)):
```

```

47         results[x][i] = {}

49     for index, s in enumerate(settings):
50         temp = Corpus(filename=data[s[0]]['Post-var Output'][s[1]])
51         text = data[letter]['Post-var Text Object'][temp.windowsize]
52         print('Starting clustering:', index+1, 'out of', len(settings))
53
54         temp.do_KM(letter=text.words, k=text.k_to_use, dump=False,
55                   links=use_links, algorithm=alg)
56         print('Finished clustering:' + str(temp.generate_filename()))
57
58         results[letter][index]['SCORE adjusted_mutual_info_score'] = \
59             adjusted_mutual_info_score(temp.predicted, text.expected)
60         results[letter][index]['SCORE adjusted_rand_score'] = \
61             adjusted_rand_score(temp.predicted, text.expected)
62         results[letter][index]['SCORE mutual_info_score'] = \
63             mutual_info_score(temp.predicted, text.expected)
64         results[letter][index]['SCORE normalized_mutual_info_score'] = \
65             normalized_mutual_info_score(temp.predicted, text.expected)
66
67         for k,v in sorted(text.info.items(), key=itemgetter(0)):
68             results[letter][index]['STAT '+k] = v
69
70         filename = temp.generate_filename()
71         filename = filename.split()
72         filename.pop(0)
73         filename = [i.split('--') for i in filename]
74         for pair in filename:
75             results[letter][index]['CONFIG '+pair[0]] = pair[1]
76
77     for_dataframe = pd.DataFrame(results[letter])
78     for_dataframe = for_dataframe.T
79     name_to_save = 'stats/' + DT.now().strftime('%d%m%y-%H%M%S') + \
80         letter + '-alg-' + alg + "-linktype-" + use_links + '-dataframe.csv'
81     for_dataframe.to_csv(name_to_save)

```

evaluation.py

References

- Adesam, Yvonne, Malin Ahlberg, and Gerlof Bouma (2012). “bokstaffua, bokstaffwa, bokstafwa, bokstaua, bokstawa... Towards lexical link-up for a corpus of Old Swedish”. In: *Proceedings of the LTHist workshop at Konvens*. URL: http://www.oegai.at/konvens2012/proceedings/54_adesam12w/54_adesam12w.pdf.
- Anderson, Jean (2013). “Enroller: an experiment in aggregating resources”. In: *Language in Scotland: Corpus-Based Studies*. Ed. by W. Anderson. Vol. 19. Editions Rodopi B.V. URL: <http://eprints.gla.ac.uk/85487/1/85487.pdf>.
- Baron, Alistair and Paul Rayson (2008). “VARD 2: A tool for dealing with spelling variation in historical corpora”. In: *Proceedings of the Postgraduate Conference in Corpus Linguistics*. Aston University. Birmingham, UK. URL: <http://ucrel.lancs.ac.uk/people/paul/publications/BaronRaysonAston2008.pdf>.
- Baron, Alistair and Paul Rayson (2009). “Automatic standardisation of texts containing spelling variation: How much training data do you need?” In: *Proceedings of the Corpus Linguistics Conference*. Ed. by Michaela Mahlberg, Victorina Gonzalez-Diaz, and Catherine Smith. Lancaster University. URL: http://ucrel.lancs.ac.uk/publications/cl2009/314_FullPaper.pdf.
- Baron, Alistair, Paul Rayson, and Dawn Archer (2009). “Word frequency and key word statistics in corpus linguistics”. In: *Anglistik* 20.1, pp. 41–67. URL: http://eprints.lancs.ac.uk/42528/1/IJES_-_final.pdf.
- Bergs, Alexander (2005). *Social networks and historical sociolinguistics: studies in morphosyntactic variation in the Paston letters (1421-1503)*. Vol. 51. Walter de Gruyter.
- Bishop, Christopher M. (2007). *Pattern recognition and machine learning*. Vol. 4. 4. Springer New York.
- Bollmann, Marcel (2012). “Automatic normalization of historical texts using distance measures and the Norma tool”. In: *Proceedings of the Second Workshop on Annotation of Corpora for Research in the Humanities (ACRH-2)*, pp. 3–12. URL: <http://www.linguistics.ruhr-uni-bochum.de/anselm/pub/acrh12.pdf>.
- Bollmann, Marcel (2013). “Spelling Normalization of Historical German with Sparse Training Data”. In: (Lancaster University). *Corpus Analysis with Noise in the Signal*. URL: <http://ucrel.lancs.ac.uk/cans2013/abstracts/Bollmann.pdf>.
- Bollmann, Marcel, Florian Petran, and Stefanie Dipper (2011). “Rule-based normalization of historical texts”. In: *Proceedings of the International Workshop on Language Technologies for Digital Humanities and Cultural Heritage*, pp. 34–42. URL: <http://www.linguistics.rub.de/~petran/papers/ranlp2011.pdf>.

- Coltheart, Max et al. (2001). “DRC: A dual route cascaded model of visual word recognition and reading aloud.” In: *Psychological Review* 108.1, pp. 204–256. URL: <http://dx.doi.org/10.1037/0033-295X.108.1.204>.
- Damerau, Fred J. (1964). “A technique for computer detection and correction of spelling errors”. In: *Communications of the ACM* 7.3, pp. 171–176.
- Davis, Norman (1971). *Paston letters and papers of the fifteenth century*. Oxford University Press.
- Evans, Mel (2012). “A Sociolinguistics of Early Modern Spelling? An account of Queen Elizabeth I’s correspondence”. In: *Studies in Variation, Contacts and Change in English* 10. URL: <http://www.helsinki.fi/varieng/series/volumes/10/evans/>.
- Firth, John Rupert (1957). *A Synopsis of Linguistic Theory, 1930-1955*.
- Gairdner, James (1895). *The Paston Letters, 1422-1509 AD*. Vol. 1. A. Constable.
- Görlach, M. (1991). *Introduction to Early Modern English*. Cambridge University Press. ISBN: 9780521310468.
- Healey, Antonette diPaolo (2002). “The Dictionary of Old English: From Manuscripts to Megabytes”. In: *Dictionaries: Journal of the Dictionary Society of North America* 23.1, pp. 156–179. URL: <http://dx.doi.org/10.1353/dic.2002.0009>.
- Jaro, Matthew A. (1989). “Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida”. In: *Journal of the American Statistical Association* 84.406, pp. 414–420. URL: <http://dx.doi.org/10.1080/01621459.1989.10478785>.
- Jespersen Otto, 1860-1943 (1928). *Modern English grammar on historical principles. Part 1, Sounds and spellings*. 4th ed. London: G. Allen & Unwin Ltd.
- Kukich, Karen (1992). “Techniques for Automatically Correcting Words in Text”. In: *ACM Comput. Surv.* 24.4, pp. 377–439. ISSN: 0360-0300. URL: <http://dx.doi.org/10.1145/146370.146380> (visited on 12/23/2014).
- Lass, Roger (1997). *Historical Linguistics and Language Change (Cambridge Studies in Linguistics)*. Cambridge University Press. ISBN: 0521459249.
- Lass, Roger, ed. (2000). *The Cambridge History of the English Language*. Cambridge University Press (CUP). URL: <http://dx.doi.org/10.1017/CHOL9780521264761>.
- Levenshtein, Vladimir I. (1966). “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10, pp. 707–710.
- Luick, Karl (1903). *Studien zur englischen Lautgeschichte*. Vol. 17. W. Braumüller.

- Markus, Manfred (2015). *Innsbruck Computer Archive of Machine-Readable English Texts*. URL: <http://www.uibk.ac.at/anglistik/projects/icamet/> (visited on 02/05/2015).
- Rissanen, Matti (2000). “The world of English historical corpora”. In: *Journal of English Linguistics* 8.1, pp. 7–20.
- Robertson, Alexander M. and Peter Willett (1992). “Searching for Historical Word-forms in a Database of 17Th-century English Text Using Spelling-correction Methods”. In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '92. New York, NY, USA: ACM, pp. 256–265. ISBN: 0-89791-523-2. URL: <http://dx.doi.org/10.1145/133160.133208>.
- Russell, R. and M. Odell (1918). “Soundex”. In: *US Patent* 1.
- Seidenberg, Mark S. and James L. McClelland (1989). “A distributed, developmental model of word recognition and naming.” In: *Psychological Review* 96.4, pp. 523–568. URL: <http://dx.doi.org/10.1037/0033-295X.96.4.523>.
- Stockwell, Robert and Donka Minkova (1988). “The English Vowel Shift: problems of coherence and explanation”. In: *Luick Revisited*. Tübingen: Gunter Narr Verlag, pp. 355–394.
- Taylor, Ann et al. (2006). *Parsed Corpus of Early English Correspondence*. URL: <http://www-users.york.ac.uk/~lang22/PCEEC-manual/> (visited on 01/07/2015).
- Ullman, Michael T. (2001). “A neurocognitive perspective on language: The declarative/procedural model”. In: *Nature Reviews Neuroscience* 2.10, pp. 717–726. URL: <http://dx.doi.org/10.1038/35094573>.
- Vinh, Nguyen Xuan, Julien Epps, and James Bailey (2010). “Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance”. In: *The Journal of Machine Learning Research* 11, pp. 2837–2854.
- Xu, R. and D. Wunsch (2005). “Survey of Clustering Algorithms”. In: *IEEE Trans. Neural Netw.* 16.3, pp. 645–678. URL: <http://dx.doi.org/10.1109/tnn.2005.845141>.
- Zhao, Ying, George Karypis, and Usama Fayyad (2005). “Hierarchical Clustering Algorithms for Document Datasets”. In: *Data Mining and Knowledge Discovery* 10.2, pp. 141–168. URL: <http://dx.doi.org/10.1007/s10618-005-0361-3>.