Welsh Proofing Tools-Making a Little NLP Go a Long Way

W J Hicks
e-Welsh Unit, Canolfan Bedwyr
University of Wales, Bangor, UK
w.hicks@bangor.ac.uk

Abstract

This paper describes the issues raised and the approach taken in the design and implementation of CySill, a Welsh spelling and grammar checker. This is a commercial product which has been on sale for a number of years. It has been designed to be used with a variety of word processors and applications running under Microsoft Windows. The history of the program is briefly described before discussing the improvements made in the latest version. It describes the tools used to create the program, the grammar rules it uses and how they are written, parsed and integrated into the program. It will also describe how the system has been integrated with word processors and other applications. We show how limited resources can take advantage of a simple rule based approach to develop the product of great benefit to users without necessarily using more complex NLP techniques. This can be of enormous value to minority languages.

Introduction

Welsh is a minority language, spoken in Wales by about 500,000 people. Over the past 10 year its use has been increasing rapidly as a written medium of communication in both the private and public sectors, creating a demand for the development of electronic proofing tools. Limited resources - both linguistic and financial - have meant that squeezing the most out of any NLP developments has been a necessity.

In 1991 the Welsh Language Board funded development of the first Welsh spell checker. Up until this time no proofing tools or indeed any electronic resources were available for Welsh. The first step in developing a Welsh spellchecker was to create a database of headwords with parts of speech by scanning in as many word lists and dictionaries as possible. Welsh is a moderately inflected language and verbs can be inflected into 43 different forms. Therefore in order to generate a fully expanded word list from this list of headwords, each verb needed to be assigned into a verb family for which the full conjugation was defined. This information did not previously exist in Welsh and the classification of verbs into families was a major part of the undertaking to build the first Welsh spelling checker.

A peculiarity of Welsh grammar, which it holds in common with other Celtic languages, is that the initial consonants of words may change, following specific grammar rules. Thus a word which in its radical form begins with the letter ‘c’, may under certain conditions begin with either ‘g’, ‘ch’ or ‘ngh’. To create a complete list of word forms from the headword list, the mutated forms of each word must therefore be generated. The original word list had 39000 base entries and when expanded to include verb forms, plurals and their mutated forms, this increased to approximately half a million words. A decision was taken early on to develop the spell checker using this original headword list combined with the morphological information instead of the expanded word list. This decision - made primarily to save space in holding the word list - proved invaluable in the further development of Welsh NLP tools. The spell checker therefore worked algorithmically, demutating words and stripping off endings until it found the headword form of the word. It would then check the morphology of this base form to ensure that the inflected form was valid. From the start then, the spell checker could correctly recognize the inflected forms of verbs and mutated forms of words: it had the ability to recognise ‘ellir’ as the mutated impersonal present tense of the verb ‘gallu’. It could therefore be used as a lemmatizer.

The original idea had been to produce a simple Welsh spell checker. Welsh is a very phonetic language and apart from problems with accents and occasional consonant doubling, the main spelling errors in Welsh are typographic. Mutation errors, however, are a very large problem and a major cause of insecurity with people writing in Welsh. Therefore a simple spell checker, while better than nothing, was of limited use and would fail to address the major source of errors in written Welsh. It was therefore decided from early on that a mutation corrector was as important as a spell checker and the original spell checker was enhanced to check for mutation errors. The fact that our spell checker worked algorithmically, and therefore knew how each word had been mutated, allowed us to do this relatively easily. Most mutations are caused by the preceding one or two words and a simple rule engine would be able to find most of these. Other mutation rules depend on the identification of noun

phrases and the parsing of sentences into subjects and objects and these would be beyond the capabilities of a simple rule matching approach. Nevertheless, most of the mutation errors could be identified without the need for a full parsing of the sentence. This mutation rule engine was then extended to cope with other types of common errors that could easily be caught by a slight extension to our system: the agreement of adjectives in number and gender with nouns, the agreement of cardinals with noun gender, the misuse of singular and plurals, homophones that are commonly confused with each other (especially words with accents, for example, ‘a’ with ‘â’, ‘mor’ with ‘môr’).

And so Cysill was born. Due to time constraints the grammar rules were hard-coded into the first version of the program. This was surprisingly effective and worked well but meant that the rules could only be written and changed by a programmer and full documentation and testing of the rule-base was impossible. The program itself proved extremely popular with users of Welsh and sold in excess of 5000 individual copies as well as site licences for over 8000 machines. It is used today by individuals, schools and colleges, the Welsh Assembly and local authorities throughout Wales. The code base for this program was successfully adapted to serve as the basis for the Microsoft Office spellchecker which first became available for Welsh with Office XP.

Further Developments

The original program has now been updated and this has given us the opportunity to completely redesign the program architecture. The biggest difference from the previous version has been the addition of a part of speech tagger and the moving of the rules into an external rule base.

Program Architecture

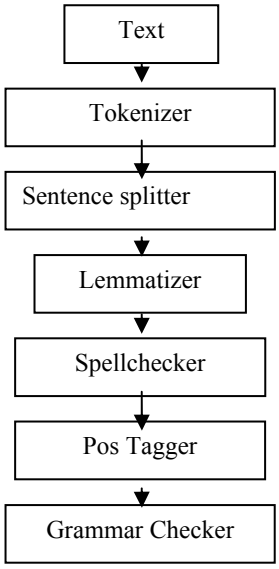

Figure 1 – Program Architecture

Figure 1 shows the overall architecture of the new version. The text is first tokenized into a stream of tokens These tokens are then assembled into sentences by a sentence splitter. The tokens in the sentence are then lemmatized into words with associated parts of speech. The spell checker then makes a pass over the sentence looking for spelling errors. Once these have been corrected by the user, the ambiguous parts of speech are partially resolved with a part of speech tagger. Finally, the sentence is checked for mutation and grammar errors, offering suggestions to the user for any mistakes found.

The Word List

Any spelling or grammar checker first of all needs a word list. The word list is stored in a relational database (Microsoft Access) with fields for plurals, parts of speech, morphological information (such as verb-families and plural formation) and mutation exceptions (some words don’t mutate as expected).

This information is exported to a text file to create our proprietary dictionary file format. This is used by the program to look up and lemmatize words. This dictionary file is then merged with a thesaurus database which holds the entries for our thesaurus.

Program Design

The program is written in C++ using a library of Welsh NLP classes. At the heart of the program lies the class Gair (Welsh for 'word'). It is this class that contains all the morphological information for a given word: its mutation, its list of possible parts of speech, and if it's a verb then its infinitive, tense and person. In addition, this class can also generate all possible morphological forms of a word, such as the mutated form, the plural form of nouns and any conjugation of a verb. This ability to generate morphology is used by our grammar checker when providing corrections and also by the thesaurus when replacing a word with its synonym.

This class has access to our word list dictionary (in fact, this is the only class that accesses our dictionary) and uses the dictionary to look up a token, performing algorithmic demutations and prefix and ending stripping. The entry for a given word form in the dictionary contains all the related grammatical information for that word form, including a parts of speech list.

The Spell Checker

The spell checker simply goes through each sentence looking for words not found in our dictionary and offering suggestions to the user for unrecognised words. The fact that we don't have a complete word list puts us at a disadvantage when generating spelling suggestions: every time a potential suggestion is generated it needs to be looked up in the dictionary and if this suggestion is not a headword (which is usually the case since most attempted suggestions turn out not be words), the program has to try its demutation and ending stripping algorithm, which is much more time-consuming than looking up a word in a list. Another problem is that the full word list does not exist per se and so iterating through the word list to find the closest phonetic match is also straightforward. However the program has the ability to generate the expanded word list dynamically and the spell checker can now search for the closest phonetic match to a misspelled word.

Since Welsh is often used in a bilingual environment, some parts of the document may contain sections of English. Cysill now has the ability to skip over such sections. The program contains an English dictionary as well as the Welsh dictionary enabling it to identify English sections of text and ignore them. An attempt has been made to avoid ignoring Welsh spelling mistakes which happen to look like English words. Whenever an isolated English word appears in the middle of a Welsh sentence it is checked to see if it is similar to a Welsh word (by comparing it to suggestions generated by the spelling suggester). If it is close to a misspelled Welsh word, it is flagged as a possible Welsh spelling mistake. If not, then it is marked as English. This works because Welsh words and English words are lexically far apart - more closely related languages would find it more difficult to distinguish between the two languages.

The Part of Speech Tagger.

Ambiguous parts of speech need to be disambiguated because the grammar rules use parts of speech in their pattern matching and these must therefore be accurate. The included part of speech tagger does not attempt to disambiguate all of the ambiguous parts of speech in a sentence: while some parts of speech need to be very accurately disambiguated whenever they appear as part of a rule, some parts of speech, where they do not affect any rules, do not need to be disambiguated. This need for a guaranteed high accuracy for certain parts of speech ruled out statistical probabilistic taggers. In addition, statistical pos taggers would incur a speed overhead and would need an accurate training corpus, which was not available. More importantly, mutations are a very good way of disambiguating parts of speech in Welsh and these cannot be used by existing statistical POS taggers. It was decided therefore to develop a rule-based tagger using a constraint grammar with hand-written rules, loosely based on the ENGCG formalism developed by a group of Finish linguists for tagging English text [2], [4] . This meant we could improve the disambiguation rules incrementally as needed. It was also very fast and was easy to implement using our C++ classes and a simple parser.

The tagger takes a sentence as a list of words (of the class *Gair*) with possible multiple parts-of-speech. A simple grammar that uses a pattern-action rule formalism is used to write disambiguation rules: each rule has a pattern-matching clause and an action clause. The pattern-matching clause can use literals,

parts of speech (of various kinds of certainty , such as 'could be', 'must be', 'is most probably'), lemmas and the full set of Boolean operators 'and', 'or' and 'not'. The tagger can also use mutations: the presence or absence of mutation in a word is an indicator of the part of speech of both the word undergoing mutation and of the word causing the mutation. For instance the word 'a' can either be the relative pronoun 'which' or the conjunction 'and'. 'A' as the relative pronoun 'which' causes a soft mutation in the following word (this would make a word starting with 'c' change to 'g') whereas 'a' as the conjunction 'and' causes an aspirate mutation in the following word (this would cause a word starting with 'c' to change to 'ch'). Thus the mutation on the word after 'a' can be used to disambiguate between the parts of speech of the word 'a' (and may be the only reliable way to do this). This mutation can also be used to disambiguate the part of speech of the word that follows 'a' as this could be ambiguous between a verbal form and a noun. The tagger makes a one-time pass over each word in the sentence and whenever it finds a pattern match it executes the action part of the rule. This action may assign definite parts of speech to or remove a part of speech from any of the words in the pattern.

A simple parser (in Ruby) performs regular expression substitutions on the tagging rules to generate the C++ code for the tagger. A few of the more complicated rules were implemented directly in C++ to save complicating the grammar. The following rule,

```
a + VERBAL_FORM? && TM?  ->  1 = RELATIVE_PRONOUN, 2 =VERBAL_FORM
```

states that if the word following 'a' could be a verbal form (VERBAL_FORM?) and has a soft mutation (TM?), then set the POS of 'a' to a relative pronoun and the POS of the following word to a verbal form.

This gets translated by the parse into the C++ below:

```
if (
     gs[index].getUnmutatedWord() ==  L"a"   &&
      gs.PosAt(index + 1).CouldBe(POS::VERBAL_FORM)  &&
      gs[index + 1].GetMutation().Is(Mutation::SOFT))
      {
         gs.GairAt(index).SetPos(POS::RELATIVE_PRONOUN);
         gs.GairAt(index + 1).SetPos(POS::VERBAL_FORM);
      };
```

where 'index' is the current word index and 'gs' is the current sentence

The Grammar Checker

The program checks for three main kinds of errors: mutation errors, incorrect syntax and incorrect word choice (mostly incorrect literal translations from English). Rules are written using a simple constraint grammar which is kept in a separate rule-base and developed and maintained by a linguist. A small number of errors proved too complicated to be expressed in this grammar without over complicating the grammar and parser, and so these were written directly into C++ (examples of these include rules to check for long sentences; missing question marks; and verb and pronoun agreements). Each rule is divided into two parts, similar to the POS tagger: the pattern matching, and, for non-mutation rules, the action to take to correct the mistake when the pattern is matched. The rules also include explanatory messages in both English and Welsh. The action part of the rule describes how to make the suggestion and this usually involves replacing, adding, rearranging or deleting words.

A simple grammar was written to parse the rules and JavaCC (a free Java compiler-writer, [3]) was used to generate the rule compiler from the grammar. The rules are first parsed by this rule compiler which converts the rules into an intermediate form, checking for syntax errors. This intermediate form is then converted by a script (written in Ruby) which uses a series of regular expression substitutions to convert the patterns and actions into C++, using our 'Gair' class, in a similar way to the part of speech tagger. The C++ code generated by this stage is then merged into a grammar checker class which contains the additional hand-written rules and this is then compiled and linked into the program.

The Rule Base

The pattern part of the rule consists of a series of tests on adjacent words. Tests can use literals,

lemmas, mutations, parts of speech, words that start with certain letters, words that start or end with vowels or consonants and words that start with capital letters. These can be combined using any of the Boolean operators ‘and’, ‘or’ and ‘not’.

A rule is either a mutation rule or a grammar rule. The mutation rules omit the action clause but have mutation flags on one or more of the words in the condition part. A required mutation (there are four different kinds) is denoted by placing one of the flags ‘/tm’, ‘/tl’ ‘/th’ or ‘/to’ after any of the words that require a mutation. Each rule is checked to see if all conditions in the pattern match and, if so, any mutation switches are checked against the mutation on the word. If any mutation does not match a mutation switch, a message is displayed to the user and the correct mutation is offered as a suggestion. The user can then decide whether to accept the suggestion or continue.

For example, the following rule states that an adjective after a masculine or plural noun does not take a mutation:

```
nm|npl + adj/t0
```

where ‘+’ is used to separate the tests on adjacent words. /t0 means ‘requires no mutation’ and ‘|’ is the Boolean operator ‘or’. This is parsed by the rule compiler to the intermediate form:

```
(ISPOS (nm) OR (ISPOS (nlp))) PLUS (ISPOS (adj))/RequiresMutation(t0)
```

This is then converted by the Ruby script into the C++ code:

```
if (
    (pGramSentence_->PosAt(index).GetPos() == POS::NOUN_MASC||
    (pGramSentence_->PosAt(index).GetPos() == POS::NOUN_PLURAL))&&
    (pNextWord_->IsAdjective())) {
       if (MissingMutation(index_ + 1, Treiglad::DIM)){
        // flag error and create suggestion
       }
```

Since it is possible that a sentence fragment can match more than one rule, the rules are ordered. Each rule is given a weight which increases the more specific the rule. This weighting is derived from the number of words in the rule and how specific each condition is, so that a literal condition has a higher weight than a part of speech condition. The rules are then sorted using this weighting function by the Ruby script before they are output as C++. This ensures that the more specific rules are checked first.

The non-mutation rules have two parts: the pattern and the action taken to correct the mistake. The pattern part of the rule is similar to the mutation rule (without the mutation flag). The action part is separated from the pattern by ‘>>’ and defines how to create the suggested correction. Actions can substitute or delete words or change a word’s morphology.

One of the simplest examples of a rule is a rewrite rule on two literals which finds the error 'os buaset' and offers 'pe bait' as the suggestion:

```
os + buaset >> {1} = pe, {2} = bait
```

(All literals match both the mutated and unmutated form of a word unless otherwise specified). This becomes in C++ code:

```
  if(
    (IsLiteral((*pCurrentWord_),  L"os")) &&
    (IsLiteral((*pNextWord_),  L"buaset"))  )
    {
      StartReplaceErrors(1,
          L"Use 'pe bait' instead of 'os buaset'.",
        L"Defnyddiwch 'pe bait' yn lle 'os buaset'.",
        L"97",
        L"os + buaset >> {1} = pe, {2} = bait"
      );
```

```
AddReplaceWord(0, L"pe");
AddReplaceWord(1, L"bait");
```

This also shows how suggestions are generated by the program. Rules can copy over morphology from one word to another:

```
@gwario + amser >> {1} = @treulio
```

This replaces the phrase 'gwario amser ' (a common erroneous literal translation of the English idiom 'spend time') with the more correct form 'treulio amser' (lit, 'pass time'). The '@' instructs the program to use the same morphological form for the verb 'treulio' as was used in the form of 'gwario' so that the form 'wariais amser' using the first person singular past tense of the verb 'gwario' with a soft mutation ('I spent') will be replaced with the corresponding first person singular past tense form of 'truelio' with a soft mutation, 'dreulias' to give 'dreulias amser' ('I passed') as the suggestion.

Currently we have over 200 mutation rules and over 300 non-mutation rules.

Integration with Word Processors

Writing a grammar checker is only of use to end-users if it can be integrated into their everyday writing environment and can be called upon easily from within various applications. Grammar checkers are mainly used from within word processors and at the moment this usually means Microsoft Word. A grammar checker must be able to take formatted text, perform all the required corrections and copy the corrected text back to the word processor while preserving any formatting. Cysill has a COM interface (a Microsoft technology that allows one program to control another) so that it can be called from Word via Word's scripting language, Visual Basic for Applications (VBA). A toolbar is supplied so that. users can click on a button to check either selected text or the entire document. It does this by iterating through each of the paragraphs selected, copying them in turn to the clipboard, checking the paragraph for errors and pasting the paragraph back if any corrections have been made.

It is vital to preserve the formatting while making the necessary corrections. The program must therefore parse either the native format of the word processor or a common standard such as Rich Text Format (RTF). In the first version of the program we parsed a number of native word processor formats with varying degrees of success. However, the maintenance and development effort became excessive and we now only support RTF. RTF is a Microsoft mark-up language for the interchange of formatted text such as word processing documents, supported (with varying degrees of success) by most major word processors and most applications that use formatted text. We originally parsed the RTF ourselves to extract the raw text but due to the effort involved in writing and maintaining an RTF parser we now use the Microsoft Rich Edit Control. This is a Microsoft control that comes as part of Windows that can parse, edit and display text in RTF. This allows the program to extract and replace text without worrying about the underlying formatting. Unfortunately this has not proved to be entirely without problems as the control only understands a subset of RTF used by Microsoft Word. It is supposed to preserve any formatting that it does not understand, but it does not always succeed. For instance, bullet points are not displayed or preserved properly and the font after a bullet point will sometimes change after having been pasted back. It also sometimes loses Unicode characters under Windows 98 (important for Welsh since the program works on Unicode text as there are no supported non-Unicode character sets). This has required a certain amount of effort to code around these in the program and supporting VBA macros but we have overcome most of these problems.

While our primary target has been Microsoft Word since this has the biggest market-share for word processors in Wales, we also support Star Office (from Sun Microsystems) and OpenOffice (the free, Open Source version) as a number of local authorities are looking at these as a cheaper alternative to Microsoft Office. We have been helped by Sun Microsystems in the writing of the Star Basic code to connect Cysill and OpenOffice, even to the point of them changing the Star Basic API to allow us to iterate more easily through the document paragraphs.

Cysill also has a hotkey interface that allows it to be called upon from any application that can cut and paste text.

The Verb Conjugator and Thesaurus

Using our class 'Gair' we can also generate word morphology, such as verb conjugations and

mutations. Thus Cysill has the ability to show the full conjugation of a given verb and copy any chosen conjugation back into the user's document. Cysill also contains a Welsh thesaurus, callable from inside the word processor, allowing the user to replace a word from a list of synonyms. Since the program has the ability both to analyse and generate morphology for words, we have used this ability in the thesaurus. When a word is replaced with a synonym, the morphology of the replaced word can be matched to the morphology of the original word. This means we can copy over any mutation and verb conjugation from the original word to the synonym. For example, Cysill will replace 'rhedais', the first person singular plural of the verb 'rhedeg' with 'drefnais', the first person singular plural of the verb 'trefnu'.

Future Developments

In the future, we hope to include a shallow parser for noun phrases, verb phrases and preposition phrase chunks. This will allow us to use more sophisticated rules containing noun phrases and verb phrases in order to catch the few remaining mutation errors that depend on a more complex parsing. The challenge will be to do this without noticeably slowing down the speed of the program.

Conclusion

This paper has shown one way to develop and implement a complete solution for a combined grammar and spell checker from limited resources. The resulting program provides real value to end-users without relying on an enormous amount of sophisticated NLP. Cysill has proved to be of major benefit to users of Welsh. Feedback from one local authority has stated that it has been responsible for shifting the working language of written communication from English to Welsh within the authority. Even simple NLP tools can be of enormous value to minority languages competing with major world languages such as English. Often, minority language speakers – particularly the older generation –have had limited education through the minority language and lack confidence in writing it. Practical NLP tools can help restore confidence, encouraging people to make more use of the minority language, increasing its use and therefore its chances of survival.

References

[1] Canolfan Bedwyr, More information on Canolfan Bedwyr's Welsh Language Proofing tools can be found at http://www.bangor.ac.uk/ar/cb/meddalwedd.php

[2] ENGCG , Information on the English Constraint Grammar ENGCG can be found at *http://www.ling.helsinki.fi/~avoutila/cg/index.html*.

[3] JavaCC, The home page of the JavaCC compiler complier http://javacc.dev.java.net/

[4] Voutilainen, A. 2003, Part-Of-Speech tagging, In (ed.) Mitkov, R *Oxford Handbook of Computational Linguistics*

[5] The Welsh Language Board, The website of the Welsh Language Board http://www.bwrdd-yr-iaith.org has information and background on the Welsh language.