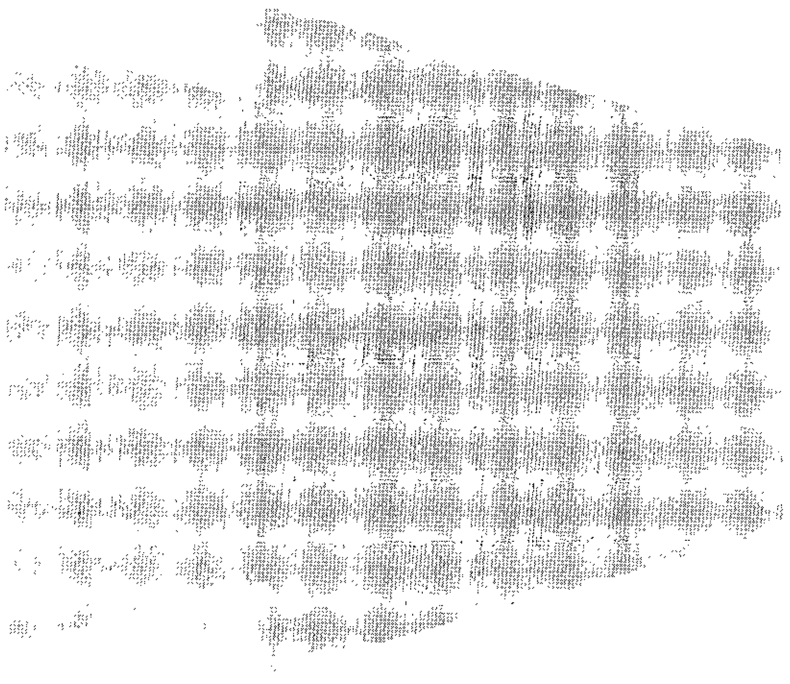
Oxford University

Automated Marking of Exam Papers

Using Semantic Parsing

Jamie Frost



08

Contents

1. Introduction

Motivation

'Automated Essay Scoring' has been a large area of research since the 1960s. In such a process, a variety of 'features' are extracted from essays, such as word and sentence length and the structure of sentences, before the data is collaborated to provide a final classification [SHERMIS 03].

'Automated Exam Scoring' is a more objective mode of classification, in which answers are analysed for the presence of concrete facts or statements instead of using any continuous measure. This has been the subject of much research in the Computational Linguistics department at Oxford University, using an online study as the source of data, in which students completed a GCSE Biology paper [PULMAN 05]. The techniques employed are varied, but falls into two main categories. One simulating a human style marker defines the marking scheme via patterns inputted by an administrator, allowing for as many variants of an answer as possible. The clear disadvantage of this method is the hours of work required to painstakingly write these patterns, but this method yields high accuracy. Average accuracy in excess of 95% was obtained.

The latter method adopts a machine learning approach using a set of pre-marked answers for the training process. [PULMAN 06] experimented with a system in which the answers are treated as a 'bag of words' with no semantic structure incorporated. A technique known as 'k nearest neighbour (KNN)' was used, which initially removes stop words from the answer sets (words of common usage such as 'and') and assigns each keyword a TDF-IDF measure ("Term Frequency" and "Document Frequency", concepts in the Information Retrieval domain combined to indicate the 'significance' of a term). Each answer in the training set is then converted to an incidence vector form given its terms (weighting each element with the corresponding term's TDF-IDF measure), as well as the answer to classify. Using a distance measure, the closest *k* training answers are obtained, and the modal score of these used for classification.

This naive method is subject to a number of problems, as highlighted by Professor Stephen Pulman in this BBC News article:

> *"Professor Srihari asked human examiners to grade 300 answer booklets. Half of the graded scripts were then fed into the computer to "teach" it the grading process. The software identified key words and phrases that were repeatedly associated with high grades. If few of these features are present in an exam script, it generally receives a low grade...*
>
> *Professor Stephen Pulman at the University of Oxford has identified another potential pitfall in Professor Srihari's approach. "You can't just look for keywords, because the student might have the right keywords in the wrong configuration, or they might use keywords equivalents," Professor Pulman says."*

Thus if the answer requirement is a statement such as 'the cat chased the mouse', then an answer of 'the mouse chased the cat' would be accepted despite the clear semantic inequality, due to the identical set of words.

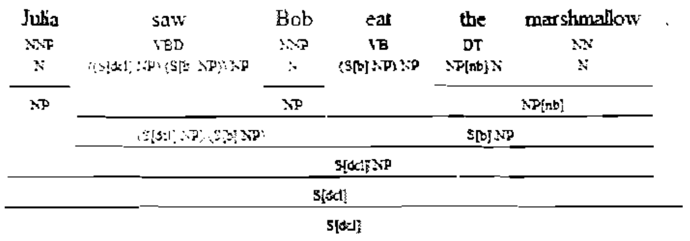
This project aims to extend this method by incorporating the semantic structure of sentences, so that for the above example 'the mouse chased the cat' would be marked as incorrect, whereas 'the mouse was chased by the cat' would be marked as correct.

CAndC Parser & Boxer

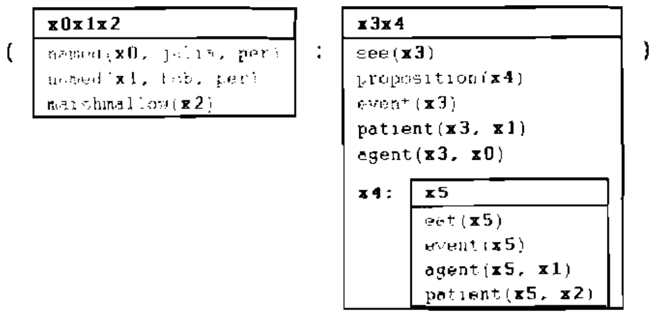
The CAndC (Clark and Curran) parser uses statistical methods and 'supertagging' to convert English sentences into a tree representing their structure, as detailed in [CLARK 07]. It was developed by Stephen Clark (a lecturer at Oxford University and member of the Oxford Computational Linguistics Group) and James Curran. The output of the parser is a CCG (Combinatory Categorial Grammar) file, representing this sentence structure.

Alone, this representation is insufficient for machine learning use, given that the semantic interpretation of the sentences is our concern. We therefore use a tool called *Boxer*, which uses Prolog to convert the CCG into a form called DRS (Discourse Representation Structure). This is compatible with first-order logic, and thus can be used to make reasoned logical deductions (with its application extending to other systems such as Question Answering).

Consider the sentence *"Julia saw Bob eat the marshmallow."* Its CCG form is:



The graphical form of its corresponding DRS is:



Verbs and nouns are represented as objects (here x0 to x5) with predicates defining their properties and relations connecting them.

The Data

All data is from students' answers to GCSE Biology papers (both foundation and higher tiers) from 2001 and 2003. The anonymised data is made available under a confidentiality agreement courtesy of the OCR Exam Board. Maximum marks for answers vary between 1 and 4 marks.

Weka

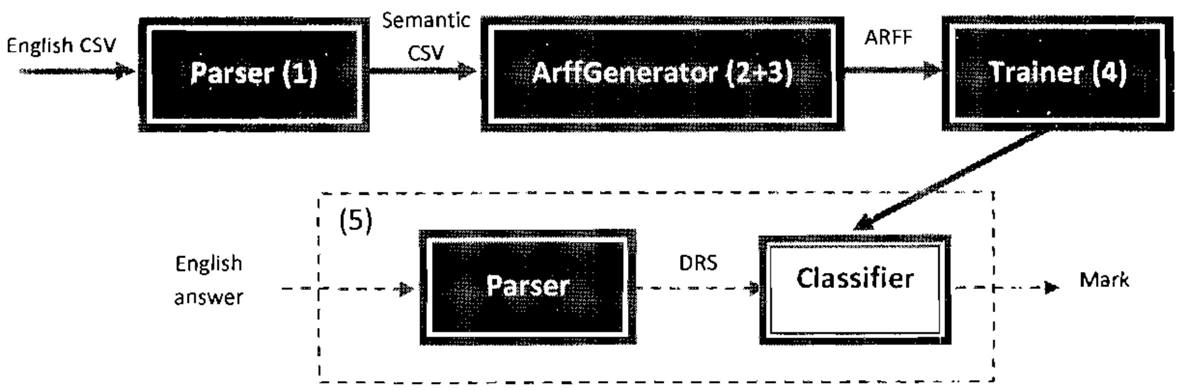
Weka is a suite of machine learning tools for Java, developed by the University of Waikato in New Zealand. It provides a variety of classifiers (including Naive Bayesian, Neural Networks and Decision Trees), clustering algorithms and other utilities. Downloads and documentation can be found at http://www.cs.waikato.ac.nz/~ml/weka/.

2. System Overview

In the scope of this project, we investigate 3 different methods of exam mark classification: a naive keyword approach in which the answers are treated as a 'bag of words' as described above (but using a more standard classifier than the 'k nearest neighbour' technique), an approach which incorporates the sentence semantics, and a potential optimisation of the latter which uses a new type of classifier. All 3 methods use a 'supervised learning' approach, that is, we train a classifier with pre-marked data, and use this to mark subsequent answers.

Such a classification problem requires a number of stages:

1. *Semantic Conversion:* This takes the answers in English (a CSV file containing the answers and marks), and converts them to a semantic representation, using the *CAndC* parser to identify the sentence structure. This is then fed this into *Boxer*, producing a semantic representation (i.e. a DRS). Clearly this step is skipped in the keyword approach. The conversion process is described in Section 3.
2. *Featurisation:* A 'feature' is a singular property encompassing some fact about an answer. A 'binary feature' takes the value 0, denoting the feature is not present, or 1 if it is. The aim of this stage is to provide some function which maps the answer (either in its raw form for the keyword approach, or the DRS for the semantic approach) to a set of such 'features' which are satisfied. For the keyword approach, the mapping simply produces a set of words the answer contains. For the semantic approach, the DRS is a non-linear structure and thus the mapping is more ambiguous. The latter is dealt with in Section 4.
3. *Data Preparation:* Such features from all answers must be collated to provide a suitable input for a classifier. This data is embodied as a file known as an 'ARFF'. See Section 5.
4. *Training:* A suitable machine learning classifier is chosen, and is 'trained' with this data.
5. *Classification:* Subsequent new answers are parsed (if necessary) and featurised as before. This is fed into the trained classifier to provide an estimated classification (i.e. a mark) for the answer.



3. Parsing

Our initial input data is a CSV file containing a list of English sentences, each with a classification. The aim of the parsing process is to produce a new CSV file, instead with DRS expressions representing each of the sentences. There are 2 methods of parsing.

One method uses the online parser found on the CAndC website.[1] However, a more practical solution would incorporate a server which can administer requests from clients and send the parsed expressions back. The CAndC tools include a *SOAP* server and client, which communicate with each other (potentially externally). The advantage of these is that the high overhead cost associated with starting the parser need only occur once in the server's lifetime, relative to once per parse request. Both of these operate on the same host, and we disregard their remote capabilities of these in order to produce a Java 'wrapper' which gives this functionality. The *SOAP* client should not be confused with the remote Java client, the latter of which generates the parse requests.

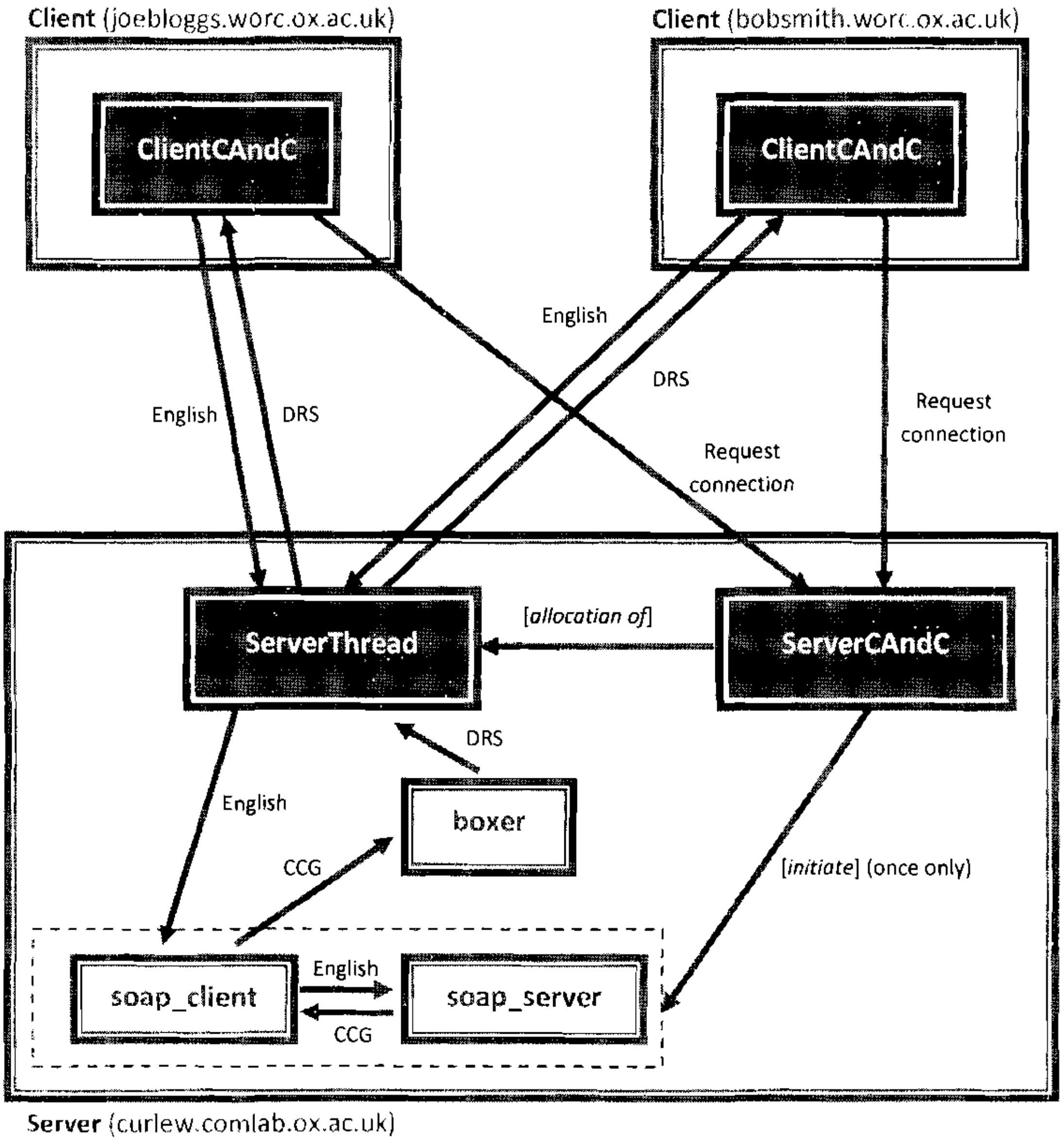
What is required is a server which forwards requests to the *SOAP* client (which in turn communicates with the *SOAP* server). It may need to manage the requests of multiple remote clients, and thus we create a separate manager for each connection, running concurrently. It performs the following:

1. The *SOAP* server is started.
2. We indefinitely wait on a specified port for external clients to make a connection. When an incoming request is received, a socket and a manager representing the connection are created.
3. In this manager, we construct an input and output stream from the socket, and continually read from it until the connection closes. We read an English sentence from the stream, write it to a file (given that the parser reads from a file) and instruct the *SOAP* server to parse the sentence. Finally we use Boxer to convert this to a DRS expression, which is delivered to the output stream.

A client end of the communication is also required. It sends the sentence to the server and delivers the parsed and boxed DRS expression returned to the required resource. As before, we establish a socket using the host name and port of the external machine running the server, and construct an input and output stream to read and write from.

The structure of the entire parsing system can be summarised as follows:

[1] This can be found at http://svn.ask.it.usyd.edu.au/trac/candc/wiki/Demo


Client (joebloggs.worc.ox.ac.uk)
Client (bobsmith.worc.ox.ac.uk)
ClientCAndC
ClientCAndC
English
DRS
English
DRS
Request connection
Request connection
ServerThread
[allocation of]
ServerCAndC
DRS
English
boxer
CCG
[initiate] (once only)
soap_client
English
CCG
soap_server
Server (curlew.comlab.ox.ac.uk)

4. Representing Semantic Atomic Facts

As previously discussed, previous attempts at implementing a machine learning system to classify answers have relied on treating them as a 'bag of words', such that marking was based on common keywords. However, this has a major flaw. The ordering and form of words have no bearing, leading to false positives. For example "Jack collects stamps with Jill" and "Jill stamps on Jack as he collects Jill" are likely to have similar classifications, despite their very different meanings and different uses of the word 'stamps'. In other words, there is no capacity for semantic representation. In the latter sentence, although Jill is performing the 'stamping', it could equally be Jack.

The aim therefore is to derive a way of semantically representing sentences that is complete, sound and most importantly, linear. That is, we wish to deconstruct a sentence into a collection of atomic predicates (here described as 'semantic atomic facts') such that these properties are satisfied. This is an example of 'featurisation', that is, we identify a number of possible properties of the input data (in this case, the presence of each of the generated atoms) for the use of machine learning application at a later stage. To establish these concepts of soundness, completeness and linearity, take for example a sentence represented by α and the 'semantic atom facts' we wish to generate from it by a set $\{\beta_1, \dots, \beta_n\}$:

- For soundness, $\forall i \,.\, \alpha \vDash \beta_i$. In other words, all such facts are implied by the original sentence.
- For completeness, $(\wedge|\vee|\neg)\ \beta_i \vDash \alpha$. That is, some disjunctive/conjunctive/negative combination of the atoms implies the original sentence. Allowing the disjunctive connective to yield α may at first seem odd, but the necessity becomes clear when considering sentences with the word 'or'; if for example "The kite is red or green" was split into "Red kite" and "Green kite".
- Linearity simply implies the lack of some recursive or complex data structure, instead keeping the atomic facts 'flat'. Here, atomicity implies the absence of the $\wedge$, $\vee$ and $\neg$ connectives.

We define a suitable function φ which produces every possible inference that can be made from a DRS expression α, that is, the β_i as defined above. Take for example the sentence "When jolly Carol laughs, there is an earthquake". We can make a number of inferences from the sentence, representing looser forms:

1. When Carol laughs, some event occurs.
2. When someone laughs, some event occurs.
3. When Carol performs some action, some event occurs.
4. When jolly Carol laughs, some event occurs.
5. Carol is involved in the sentence.
6. Carol is jolly.
7. An earthquake occurred.
8. When someone laughs, an earthquake occurs.

And so on. To formally define φ, we consider all the mappings for all possible DRS sub-expressions to sets of 'atoms'. The DRS consists of a number of objects, representing nouns or verbs, named via a specially typed predicate. The atom produced is simply the name of the predicate (note that the term after the colon indicates the type):

$$\begin{aligned}
\varphi(\text{p:noun}(x)) &= \{\ \text{p:String}\ \} \\
\varphi(\text{p:verb}(x)) &= \{\ \text{p:String}\ \}
\end{aligned}$$

For example, $\varphi(\text{cat}(x)) = \{\ \text{“cat”}\ \}$. In addition, other unary predicates representing adjuncts can modify such an entity; this is an adjective in the case of a noun, and an adverb in the case of a verb. Its corresponding atoms consist of the name of the modifier and the modifier combined with the name of the entity. The *nm(x)* function simply retrieves the assigned ‘name’ of the entity *x*.

$$\varphi(\text{p:adj}(x)) = \{\ \text{p:String},\ \text{p:String}[\text{nm}(x)]\}$$

For convenience, we define $\varphi(x)$ to be *all* atoms associated with the entity *x*. We similarly take φ of the whole DRS expression α as the union of sets produced by applying the function to all expressions contained within it. That is:

$$\begin{aligned}
\varphi(x) = \varphi(p_1(x)\ ;\ \dots\ ;\ p_n(x)) &= \varphi(p_1(x)) \cup \dots \cup \varphi(p_n(x)) \\
\varphi(\alpha) = \varphi(\alpha_1\ ;\ \dots\ ;\ \alpha_n) &= \varphi(\alpha_1(x)) \cup \dots \cup \varphi(\alpha_n(x))
\end{aligned}$$

Binary predicates (i.e. relations) connect two entities, and typically represent a verb or a prepositional modifier. In mind of providing all inferences, the resulting atoms must incorporate every possible pair of atoms from the atom sets of the constituent entities:

$$\varphi(p(x_1, x_2)) = \{\ \text{p:String}[e_1,e_2]\ \mid\ e_1 \in \varphi(x_1) \wedge e_2 \in \varphi(x_2)\ \}$$

Larger DRS expressions can be combined together using a variety of conjunctives. These must be dealt with appropriately:

$$\begin{aligned}
\varphi(\ \alpha \vee \beta\ ) &= \varphi(\alpha) \cup \varphi\ (\beta) \\
\varphi(\ \alpha \wedge \beta\ ) &= \varphi(\alpha) \cup \varphi\ (\beta) \\
\varphi(\alpha\ {=}/\text{is}\ \beta) &= \{\ \text{“}e_1 = e_2\text{”},\ \text{“}e_2 = e_1\text{”}\ \mid\ e_1 \in \varphi(\alpha) \wedge e_2 \in \varphi(\beta)\ \} \\
\varphi(\alpha \rightarrow \beta) &= \{\ \text{“}e_1 \rightarrow e_2\text{”}\ \mid\ e_1 \in \varphi(\alpha) \wedge e_2 \in \varphi(\beta)\ \} \\
\varphi(\ \neg\alpha\ ) &= \varphi(\alpha)
\end{aligned}$$

Such a function may initially seem disastrously hideous to a logician: negation is ignored, and conjunction and disjunction are treated in a uniform way. The motivation is to have as high coverage as possible of all inferences made by the sentence, with the only requirement being that the original sentence can be reconstructed from some conjunctive/disjunctive/negative combination of a subset of $\varphi(\alpha)$. Note that the equality relation is symmetric, therefore we must supply both orderings in the string representation. We do not treat $\alpha \rightarrow \beta$ as $\neg\alpha \vee \beta$ given we are only interested if the inference was made, not if it was true vacuously when α is false.

To demonstrate this definition, take the sentence “The big dog jumped over the lazy man.” This could be represented in DRS by:

$$\text{dog}(x_1)\ ;\ \text{big}(x_1)\ ;\ \text{man}(x_2)\ ;\ \text{lazy}(x_2)\ ;\ \text{jump}(x_3)\ ;\ \text{agent}(x_3,x_1)\ ;\ \text{over}(x_3,x_2)$$

In this example, the jumping is being executed by the dog (i.e. the dog is the agent of the jumping). Using the laws above, the following atoms are produced:

dog
big
big[dog]
man
lazy
lazy[man]
jump
agent[jump,big[dog]]
agent[jump,dog]
agent[jump,big]
over[jump,lazy[man]]
over[jump,lazy]
over[jump,man]

Relevance to Exam Marking

The above system is based on assumptions relating to a generic marking system. We assume that some number of marks are awarded for the conjunctive/disjunctive combination of these atomic facts. For example, a mark may be awarded for mentioning that the man is lazy ("lazy[man]") and another for identifying a jump occurred over him ("over[jump,man]").

There may initially appear to be a problem with answers either being too specific or not specific enough. Say for example that "over[jump,man]" was required for a mark. If the user was to answer with "jumped over the large man", we obtain "over[jump,large[man]]" and by deconstruction, "over[jump,man]" also. Is a mark awarded for "jumped over the *large* man" if the answer required was "jumped over the man"? This is somewhat of an ambiguous question. In some cases, the extra detail may be unnecessary but not detrimental to the answer; we may consider that our example merits the mark. However, there are some cases where clearly extra detail changes the answer. Take for example the answer "The act of putting magnesium in water causes a reaction." Then "The act of putting magnesium in water causes a *fatal* reaction." is clearly wrong, given that it grossly misrepresents the magnitude of the reaction.

Fortunately, a suitable machine learning method can usually cater for this. Let an answer be α (e.g. red dog) and its looser implication β (e.g. dog) such that $\alpha \rightarrow \beta$ (i.e. a red dog is a dog):

- For an actual answer α where β is not specific enough (e.g. where *red dog* would be awarded a mark but not *dog*): $\alpha \wedge \neg\beta$ holds.
- For an actual answer β where α is an acceptable extension (e.g. if *dog* was accepted, then *red dog* would also be accepted): $\alpha \vee \beta$ holds.
- For an actual answer β where α is not a valid extension (such as with the chemistry example above): $\neg\alpha \wedge \beta$ holds.

These are all simple Boolean expressions which can be 'learned' during the training process. Classifiers' ability to learn any such Boolean expression justifies our function φ used for linearisation.

There is however one particular flaw with the definition of φ, in the way that negation is handled. It is difficult to distinguish between "The dog was red" and "The dog was *not* red", since the negation is ignored. The underlying problem is that the binary value in the vector representation of the answer represents the presence or absence of a clause, not the Boolean interpretation of 'true' or 'false'. We could propagate the $\neg$ through subclauses (resulting in sets like { dog, red, red[dog] } and { dog, $\neg$red, $\neg$red[dog] }), or use 3-valued logic (i.e. 0 for the absence of a feature, 1 if present but negative, and 2 if present and positive), but such attempts generally lowered the accuracy of the system (by up to 10% for some data sets). One might suggest that this is due to students generally avoiding specifying a correct answer before negating it. For example, the question "By which process to plants produce oxygen and food?" is unlikely to be answered with "Not by photosynthesis."

5. Data Preparation and Training

Parsing and atomising from the previous stage will produce sets of atoms for each of the answers. In order to produce an input file suitable for a classifier, these must be collated.

In Weka, a training file for a classifier is known as an ARFF (Attribute-Relation File Format) file. The file has two segments: the first is an ordered list of attributes (each with a data range defined), and the second a data table representing the instances (i.e. answers) as vectors, where each element represents a value for the corresponding attribute (the last value being the classification for the instance).

Producing the set of attributes is trivial; we simply take the union of all the sets of atoms. To produce the data table, for each instance in the training set we use the value '1' for an element if the attribute appears in its set of atoms, and '0' otherwise. This is best illustrated by an example; consider these 3 sets of atoms for the 3 answers in the training set:

A_1 = { cat, dog, mouse }, A_2 = { dog, horse, rabbit }, A_3 = { horse, monkey }

Suppose that A_1 received 1 mark, A_2 received 0 and A_3 received 2. Then the appropriate ARFF file would look like such:

```
@relation animal_example
@attribute "cat" { 0, 1 }
@attribute "dog" { 0, 1 }
@attribute "mouse" { 0, 1 }
@attribute "horse" { 0, 1 }
@attribute "rabbit" { 0, 1 }
@attribute "monkey" { 0, 1 }
@attribute "classification" { 0, 1, 2 }

@data
1, 1, 1, 0, 0, 0,     1
0, 1, 0, 1, 1, 0,     0
0, 0, 0, 1, 0, 1,     2
```

Notice that we must determine the maximum mark in the training set in order to define the range for the "classification" attribute. Upon collating the attributes, it is beneficial to maintain a count of all attributes. Any attribute occurring only once is removed, as it is deemed as having insufficient impact on the classifier. This has the benefit of massively reducing the number of attributes, thus making training and classification considerably faster.

Training a classifier in Weka is near trivial using an ARFF file. We construct an untrained classifier, and an object representing the data in the ARFF file. After providing a reference to this data in the classifier, the training process is finally invoked[2].

[2] The trained classifier is required for future use, so we use Java's *FileOutputStream* class to write the classifier to a file. File representations of classifiers are (arbitrarily) given 'model' as a file extension.

There are a number of established classifiers that one may wish to use. Here is a brief description of the options available:

- *Neural Network:* This consists of a number of connected units known as 'neurones', analogous to the functioning of the brain. Each neurone takes the weighted sum of its inputs, and applies a function (usually either a threshold or sigmoid function) to produce its output. The network uses the training set and error minimisation via a process of 'back propagation' to iteratively adjust these weights.
- *Decision Tree:* This repeatedly partitions the training set on attributes, ordered by the 'information gain' (i.e. the extent of diversity in the proposed partitioning) each attribute provides. This results in a tree, with each node splitting into two given the presence of a certain feature, and the 'leaves' designating the resulting classification.
- *Naive Bayesian Classifier:* This uses simple probabilistic theory based on Bayes' Theorem. It calculates the probability of a classification (given the features) by using the probability of each feature given the classification and the prior probability of each classification, both of which are easily calculated. The classification which maximises this probability is selected. It assumes that each of the features are independent.
- *Maximum entropy modelling:* A framework for integrating information from many sources for classification. The idea is that it finds a model that satisfies the given constraints but does not go 'beyond the data', that is, it avoids a model not justified by the empirical evidence (i.e. the training set). Its advantage over the Naive Bayesian classifier is that it does not enforce independence assumptions with regard to the features.

Each of these classifiers has its own advantages and disadvantages. For the purposes of the evaluation process, the decision tree and naive Bayesian classifiers will be used, given their support by the Weka libraries.

Given our 4 stages of semantic parsing, featurisation, data preparation and training, we now have a complete system suitable for testing. However, there are some possible refinements we can make given the specific nature of the classification problem at hand. Two issues raised and explored are that of spelling correction, and a new form of classification that deals with the cumulative nature of exam marking.

6. Spelling Correction

Exam answers, typically those of a younger contiguate, must be expected to be filled with both grammatical and spelling errors. While the former are incredibly difficult to correct by autonomous means, it is possible to develop a system which corrects spelling errors with a good degree of accuracy.

One must initially question the importance of this correction in marking. Errors in inconsequential words which form the structure of the sentence will be of little concern to the examiner. However correcting such words would seem vital in order for the parser and boxer to establish a more accurate interpretation of the sentence, and to reduce the number of attributes in training. On the other hand, misspellings of keywords directly encapsulated by the mark scheme may heavily influence the examiner's marking. While the 'examiner's generosity' is possible to approximately model, it raises the issue that 'desirable spelling correction' is a fairly ambiguous and nondeterministic problem. This would suggest that a semi-autonomous approach should be adopted rather than a fully autonomous one. The system should produce a modifiable mapping defining how spelling corrections should be made, before mapping this (whenever the administrator chooses to do so) on the data.

An important observation to make is that a universal set of correct spellings is difficult to obtain. Many words are subject specific and thus will not be found in a standard dictionary. Therefore we must use a combination of the words in the data and words in a dictionary. A simplified version of the proposed correction algorithm can be defined as thus:

```
correction(word) = if correct(word) then word else
    arg max ( { score(x,word) | x ∈ suggestedCorrections(word) } ∪ { score`(x,word) | x ∈ corpus } )
```

where score(x,y) and score`(x,y) are two different scoring functions for dictionary and corpus comparison. We distinguish between the 2 functions since we wish to take into account the frequency of words in the corpus.

Dictionary Comparison

Upon obtaining a list of suggestions for an incorrectly spelled word, we obtain a similarity measure between the word *w* and the suggested correction *s*:

$$\begin{aligned}\text{score}(w,s) = {} & \alpha.\text{FirstLetterBonus}(w,s) - \beta.\text{LevenshteinDistance}(w,s) \\ & - \text{LetterDifference}(w,s) - \text{SizeDifference}(w,s)\end{aligned}$$

It was found $\alpha = 3$ and $\beta = 2$ produced the best results. A number of functions, of the type *(String,String)* $\mapsto \mathbb{N}$, provide various string metrics:

Name	Explanation	Motivation	Implementation
Levenshtein Distance	The number of insertion, deletion, or substitution character operations required to change from *w* to *s*. For example, the distance between "kitten" and "sitting" is 3.	This provides an initial well-established comparison measure.	A dynamic programming algorithm using a (n + 1) × (m + 1) matrix, where n and m are the lengths of the two strings.

First Letter Bonus	Returns 1 if the first letter of both words is the same, and 0 otherwise.	Regardless of how badly a word is spelled, typically the first letter is correct. Thus from "recieve", it is more likely that "receive" was intended than "believe", despite both having the same Levenshtein distance.	
Letter Difference	Taking each word as a set of letters, it counts the number of letters which are in one word but not the other.	We favour corrections in which the letters used is more consistent. This for example favours a correction from "doen" to "done" rather than the less probable "den".	$\#\vert \text{letterset}(s) \cap \text{letterset}(w)' \vert + \#\vert \text{letterset}(s)' \cap \text{letterset}(w) \vert$
Size Difference	The difference in size between *w* and *s*.	This gives more preference to a 'letter pair swap' over an addition/deletion, and eliminates candidate corrections too different in size.	$\vert \# w - \# s \vert$

We keep the correction with the highest score, but delay adding this to the mapping given that this may be overwritten by a better match in the corpus.

Corpus Comparison

When comparing a misspelled word against the corpus (i.e. the exam answers), the assumption above that the words are correctly spelled can no longer be made; therefore the frequency of each term must be taken into account. We can augment the previous metric to provide a new one:

$$\text{score}'(w,s) = \mu \,.\, \log \text{frequency}(s) + \text{score}(w,s) - \lambda$$

In essence, we provide extra weighting if the word is commonly occurring, that is, we can be more confident of the word's 'correctness'. The constant λ ensures that it is not necessarily the case that score'(w,s) > score(w,s) (which would essentially discard dictionary comparison). We make a number of checks to prevent incorrect mappings:

1. Most obviously, we ignore the pair *(w,s)* if w is equal to s (preventing the identity mapping).
2. We ignore *(w,s)* if the Levenshtein Distance between them is greater than 5. This prevents a mapping to a very dissimilar word if it is the best that exists (i.e. it may be the case there is no mapping available at all).
3. We ignore *(w,s)* if the frequency of *w* is greater than the frequency of *s* in the corpus. This prevents mapping cycles, which would cause an infinite loop when attempting to map the corrections on the data.

Other Considerations

There are a few forms of errors that cannot be easily detected by the above methods. We can correct these accordingly:

1. *Plurality errors:* It is common for the plural form of a word to be misspelled. For example, "babies" is commonly written incorrectly as "babys", and "halves" as "halfs". In the above techniques, it is likely that an alternative word will be suggested, therefore we must compensate accordingly. We hypothetically replace common incorrect suffices with their correct equivalent, such as "ys" for "ies" and "fs" for "ves". If the new word is in the dictionary, we add this to the mapping and ignore any other correction methods.
2. *Words augmented with apostrophes:* Nouns can have "'s" added to the end. The new string is not in the dictionary, despite this extension being perfectly valid. We check if the substring before the apostrophe is in the dictionary, and if so, regard the word as correct.
3. *Word splitting:* It is common for certain pairs of words to be incorrectly merged, for example "aswell", "alot" and "thankyou". To split these, we hypothesise a space insertion at each point in the string; if this forms two words both found in the dictionary, we add this to the mapping. This method can be problematic if not augmented, given that unlikely splits such as "aswell" into "a swell" can be made. This problem can be partially alleviated by favouring splits in which the bigram occurs most frequently in the corpus (i.e. "as well" is likely to occur more frequently than "a swell").

Mapping the Corrections

The mappings of incorrect words to their (hopefully) correct forms are saved in a file. This allows the user to inspect the suggested mappings and either add their own or modify/correct the ones already present. The mapping is applied simply by searching for occurrences of each member of the mapping key in the data, and replacing it with its corresponding value.

A Demonstration

Here is a sample of corrections produced when dealing with a question on 'vasoconstriction':

alot	a lot
ammount	amount
aswell	as well
bodys	bodies
capilaries	capillaries
capilary	capillary
capileries	capillaries
capillarites	capillaries
capillars	capillaries
capillarys	capillaries
capilleries	capillaries
capilliaries	capillaries
capilliery	capillary
cappillaries	capillaries
cappillary	capillary
cappilleries	capillaries
colser	colder
dont	don't
errect	erect
everytime	every time

hypothlamus	hypothalamus
insulater	insulator
inturn	in turn
mantain	maintain
normaly	normally
prents	parents
presonne	person
stoped	stopped
suplie	supply
transfered	transferred
traveling	travelling
vascoconstrict	vasoconstrict
vascondriction	vasoconstriction
vasconstriction	vasoconstriction
vascontriction	vasoconstriction
vasilaconstriction	vasoconstriction
vasoconstruction	vasoconstriction
vesseles	vessels
vessles	vessels

7. Cumulative Classification

The approaches used thus far produced promising results despite the small training sets available. However, there is a significant conceptual flaw in the classification approach used, stemming from the fact that classifiers have no abstract notion of 'magnitude' in their outputted values. Marks assigned may as well be replaced with arbitrary letters or symbols, since the classifiers cannot distinguish that a mark of 2 is greater than a mark of 1. This leads to some unfortunate consequences:

- As the maximum mark becomes large, standard classifiers rapidly deteriorate in performance. Consider for example all answers which are awarded 5 out of 10 marks. These marks could have been awarded in ${}^{10}C_5 = 252$ ways. Clearly, even with a large training set, it would be impossible for classifiers to fully encapsulate 5 mark answers, when one considers that each of these 252 ways are based on multiple features, further increasing the uncertainty.
- We neglect dependency assumptions. If an answer is classified as 2 and satisfies some property $A \wedge B$, then clearly if two answers both classified as 1 have the properties A and B respectively satisfied, then they are strongly correlated. By ignoring these dependencies, we surrender a large quantity of inference that can potentially greatly aid in the training process.

A possible solution we propose is to introduce a notion of 'cumulative classification', with just one fundamental property:

If two sets of properties X and Y are satisfied resulting in classifications x and y respectively, then the overall resulting classification for $X \cup Y$ is $x + y$ (given that X is not a subset of Y, and vice versa).

This is an intuitive principle, and encompasses the way in which an examiner would mark a paper. If an answer gets a mark for mentioning one point and a second for another, then one would expect this to contribute a total of 2 marks to their final mark. Standard machine learning classifiers cannot represent this concept.

The method proposed can be divided into 3 distinct stages:

1. *Segmentation:* We divide the features into sets of features relating to each mark in the mark scheme. These sets may not necessarily be disjoint, neither must they span the whole set of features. We maintain a classifier for each of the marks, with the corresponding sets denoting each one's attributes.
2. *Training:* For each answer, we determine which of the marks are most likely to be satisfied and which aren't, and train each classifier accordingly.
3. *Classification:* We combine the outputs of these classifiers to provide an overall classification for the answer.

In summary, we are delegating the problem to each of the individual marks, combining their results to solve the overall classification task. The following sections deal with the motivation and implementation for each of the three stages.

Segmentation

The cumulative classifier is a composite of simpler classifiers managing the classification of each individual mark. Each of these classifiers requires a list of attributes, and intuitively one would expect this to incorporate features relating to this particular mark. There are two measures which we must consider to determine the accuracy of this algorithm. The first is precision; the proportion of features which appear in the 'optimum group'. The second is recall; the proportion of features from the 'optimum group' that appear.

Precision is important, when we consider that taking the entire set of features for all groups, which produces 100% recall but low precision, would result in poor classification, given that the system has no way to distinguish between marks. Recall is also important, given that neglecting important features produces a classifier that is not representative of the mark.

Some consideration should be given to the relationship between the sets of features. Should for example the sets be disjoint? This depends greatly on the mark scheme in question. Marks tend to be based on independent points to avoid ambiguity in marking, but it is very possible that two different marks will mention the same item or action. The sets need not provide full coverage of the entire feature set, when we consider that many points contained within some answer may be entirely irrelevant to the mark scheme, and thus not associated with any particular mark.

We construct 3 different methods that attempt to perform this segmentation. The first 2 of these use a 'co-occurrence' matrix, that is, a symmetric *n* x *n* matrix (*n* is the number of features) where each element $M_{ij} = M_{ji}$ is the number of times feature i occurs with feature j. This can be interpreted as a measure of the connection between 2 features. To compute this, we take a binary matrix *A* representing the training data, such that each row is an incidence vector representing one answer[3]. M is easily calculated:

$$M = A^T . A$$

This follows from the fact M_{ij} takes the dot product of columns *i* and *j* (where such a vector is the incidence vector of a particular feature across all answers), equating to the number of times both features occur across all answers (i.e. if both features are present in a particular answer, their product is 1, contributing to the total count, otherwise their product is 0). We could feasibly use M^k for some k > 1 to improve the connectivity of features. If k = 2 for example, then M^2_{ij} conceptually indicates the extent to which feature *i* is connected to feature *j* via some intermediate feature; if for example "cat" occurs with "fur" in an answer and "fur" with "dog", but not "cat" with "dog", then M^2 allows us to make such a connection between "cat" and "dog". In practice this has an adverse effect for the type of data in question. We finally map the log function onto all elements of the

[3] An initial assumption was that filtering the matrix to rows that received '1' as a classification may provide an optimisation. This is because each row would have present features mostly corresponding to one mark, thus increasing the probability that two features have a high co-occurrence due to them relating to the same mark. However, this reduced the training set to such an extent that this was not a viable approach.

matrix before normalising the columns[4]. This ensures that particular high co-occurrence values do not dominate and force other values towards 0 after normalisation.

Method 1: Standard K-means Clustering

K-means clustering is a commonly used method that partitions elements into disjoint sets (which span the entire set) using a standard algorithm. The resulting clusters have been partitioned in such a way that the mean distance from each vector to the centroid[5] of its respective cluster is minimised. The process consists of simply clustering on the columns of M into k sets, where k is the maximum mark. Weka provides clustering facilities, thus the implementation involves producing a 'wrapper' which uses the matrix M to construct an input suitable for the clusterer, before returning a list of sets, where each set contains the indices of the features in the cluster.

The algorithm is fast and efficient, and provides moderate accuracy for most questions. However the disjoint set assumption is likely to cause problems for questions where high feature overlap occurs between marks.

Method 2: Iterative Clique Merging

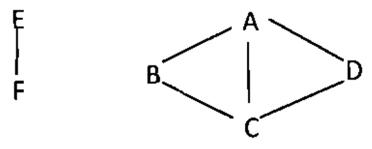
An alternative approach proposed here generates a number of seeds, where each seed is an island of closely connected features, before merging them into the required number of clusters.

Instead of normalising the columns in M as before, we scale all elements down so that the maximum element is 1. We define a threshold α ($0 < \alpha \leq 1$) such that if $M_{ij} \geq \alpha$, the bond between features i and j is considered to be 'strong', and 'weak' otherwise. This allows us to construct an undirected graph G, where we define the nodes to be the features, and the edges to be the 'strong bonds'.

To demonstrate this, take an example such that the co-occurrence matrix M is:

$$\begin{array}{c|cccccc|}
 & A & B & C & D & E & F \\
\hline
A & 1 & 1 & 1 & 1 & 0 & 0 \\
B & 1 & 1 & 1 & 0 & 0 & 0 \\
C & 1 & 1 & 1 & 1 & 0 & 0 \\
D & 1 & 0 & 1 & 1 & 0 & 0 \\
E & 0 & 0 & 0 & 0 & 1 & 1 \\
F & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
\end{array}$$

with features A to F. The value of α is arbitrary here since all non-zero values are 1. This produces the graph:



[4] More specifically, the function is *f(x) = log(x+1)*. This ensures that 0 maps to 0, rather than produce an error.

[5] The centroid of a cluster is the mean of its constituent vectors.

We start by identifying all cliques contained within the graph. A clique is a set of nodes such that there is an edge between every pair of nodes within it. From the graph, we can see that there are 3 cliques: {E, F}, {A, B, C} and {A, C, D}. To identify the cliques, we devise the following algorithm:

```
Let cliques = Ø
For (each row i of M)
    If (cliques is empty) add new clique ( i ) to cliques
    Else (
        For ( each clique q in cliques)
            If all the bonds from i to each of the elements in q are strong,
                add i to q.
            Else if all the bonds from i to each of the elements in q are weak
            (and additionally, i was not added to any other clique in cliques)
                create a new clique ( i ) and add it to cliques.
            Else (i.e. we have a mixture of weak and strong bonds) create
                a new clique:
                $q' = ( i ) \cup ( j \mid strong(i, j), j \in q)$
                and add it to cliques.
```

Presuming that *cliques* is not empty, we observe for each of the accumulated cliques which nodes the node *i* has edges to. Clearly if there are edges to all the nodes, the clique is preserved so we add the node *i* to the set. Similarly, if *i* has no edges to any element in the set of cliques (such as row E), we create a new clique (delayed until the end of iteration to ensure this condition is met). Lastly, if there are edges to some but not all the nodes in a clique (such as node D with {A,B,C}), we create a new clique containing the node itself and those nodes which it has an edge to. One might initially think this is a polynomial time algorithm[6] given the nested for loops. However, the inner loop becomes larger as *cliques* expands, leading to an exponential algorithm.

If we follow this algorithm for the above example, on each iteration we obtain: [{A}], [{A,B}], [{A,B,C}], [{A,B,C}, {A,C,D}] (where the split occurs), [{A,B,C}, {A,C,D}, {E}], [{A,B,C}, {A,C,D}, {E,F}].

We desire only *k* groups however, so the sets above must be merged. Merging is performed by constructing an appropriate algorithm:

```
Let groups = cliques; Discard all g $\in$ groups of size 1
While( |groups| > k)
    Let bestGroup1 = Ø
    Let bestGroup2 = Ø
    For( each group g1 in groups)
        For(each group g2 in groups)
            Take the cross product of g1 and g2 (g1 x g2) and
            find the average bond strength over all pairs. If this value
            exceeds the best so far, set bestGroup1 = g1, bestGroup2 = g2.
    Add bestGroup1 $\cup$ bestGroup2 to groups.
    Remove bestGroup1 and bestGroup2 from groups.
```

[6] If this were the case, then *P = NP* given that a similar problem, the 'maximum clique' problem, appears as one of Richard Karp's original 21 problems shown NP-complete in his seminal 1972 paper "Reducibility Among Combinatorial Problems".

Suppose k = 2. Then just after one iteration of the algorithm, {A,B,C} and {A,C,D} will be merged to {A,B,C,D}, leaving this and {E,F}. Observing the original diagram, such group identification for this particular example appears sensible.

The fundamental flaw in this algorithm is that its unavoidable exponential complexity renders segmentation on hundreds of features (as would usually be expected) infeasible. Even if divide-and-conquer techniques could be used on the merging to produce an O(n.logn) algorithm, there exists tens of thousands of cliques produced from the previous process. Using singleton sets with each of the features as the seeds for the merging algorithm simply does not work as the resulting sets will be disjoint; in which case the standard clustering algorithm would be a superior alternative.

One should also note that a slight alteration in α can drastically alter the output of the algorithm. Clearly tweaking such a parameter dependent on the data set to improve results is undesirable.

Method 3: Manual specification

A non-autonomous alternative is to manually specify the groups each feature appears in. Clearly manually assigning features to groups removes the problems with the methods listed above, and any loss in accuracy is due only to human error. Given that a data set of 150 answers typically produces 800 features, the task is an arduous one and becomes impractical as the data set increases. A command line interface developed can alleviate this problem if the mark scheme is based on simple keywords (see *Implementation Notes*).

Training

The segmentation from the previous stage produces *k* classifiers with assigned attributes ready for training. In order to train multiple networks, we must determine a method of taking one instance and distributing this over the *k* classifiers.

Suppose an answer in the dataset obtained a mark of *j* out of *k* ($0 \leq j \leq k$). Each classifier can only produce a classification of 0 or 1 (indicating whether the mark was obtained or not), so we expect to train *j* of these classifiers with 1 (where the instance is restricted to the features the feature set corresponding to that classifier contains), and the remaining *k-j* with 0.

To determine which of these *j* classifiers are trained with 1, we rank the *k* groups in order of *cosine similarity* between the cluster's vector (with 1s for attributes that occur in the cluster, and 0s for those that don't) and the instance vector, from highest to lowest. In practical terms, this gives a relative measure of the magnitude of the instance 'matching' each cluster. Once this ranking is achieved, we simply take the top *j* elements from the list as the clusters that 'received' the mark.

The cosine similarity is a measure of the similarity of two vectors by calculating the angle between them. For two vectors A and B, it is defined to be:

$$\cos\theta = \frac{A\,.\,B}{|A||B|}$$

We can ignore the Euclidean length of the vector corresponding to the answer to classify, since this will remain constant as we vary the cluster vector. Thus the measure is simply:

$$\frac{cluster \,.\, answer}{|cluster|}$$

Classification

For a new instance, we simply restrict the answer vector to each of the clusters, classify each instance, and sum over the classifications. This is analogous to a marker determining which marks in the mark scheme were awarded, before totaling their score.

Extending to alternative types of mark scheme requires little change. Take for example a scheme of the form "Award 1 mark for any of the k points, maximum n". The number of clusters remains the same, but in classification we take *min(mark, n)* so that the maximum mark is not exceeded.

A worked example

Suppose a mark scheme consisted of 2 marks, with the following criteria:

1) Mention of a 'dog'.
2) Mention of the act of 'walking' or 'exercise'.

Suppose also the following training answers were supplied, their classification denoted in brackets:

A. The big dog started barking. (1)
B. The dog buried his bone. (1)
C. Adam enjoys walking and exercise. (1)
D. The dog enjoyed exercising. (2)
E. The big lecturer scared the timid student. (0)

Each of the answers would be semantically analysed and deconstructed into its constituent features. For the sake of simplicity, we extract only 4: 'dog', 'big', 'walk' and 'exercise'. We could therefore generate the following incidence matrix A for the above answers:

	dog	big	walk	exercise
A	1	1	0	0
B	1	0	0	0
C	0	0	1	1
D	1	0	0	1
E	0	1	0	0

Computing M = $A^T . A$, taking the log of each value, and normalising the columns, we obtain:

	dog	big	walk	exercise
dog	0.5	0.387	0	0.279
big	1	0.613	0	0
walk	0	0	0.5	0.279
exercise	1	0	0.5	0.442

By inspection of the non-zero values, it is clear that the two groups obtained by clustering are G_1 = { dog, big } and G_2 = { walk, exercise }. This is consistent with the original mark scheme. We create a standard classifier for each of these groups (say C_1 and C_2), and now use the rows of the matrix A to train these. We define an ordered list L_R to be the groups in descending order of their similarity with the row R.

The incidence vectors for the 2 groups are (1, 1, 0, 0) and (0, 0, 1, 1) respectively. Since the Euclidean distance of these are the same, for this example we can ignore these from the cosine measure and simply use the dot product instead.

For the first row A, we have the vector (1, 1, 0, 0). We take its dot product with each of the group incidence vectors:

$$(1, 1, 0, 0) . (1, 1, 0, 0) = 2$$
$$(1, 1, 0, 0) . (0, 0, 1, 1) = 0$$

Since 2 > 0, this gives us $L_A = [G_1, G_2]$. For this first row, 1 out of 2 marks were awarded, so we train C_1 (corresponding to G_1, the first element of the list) with the row (restricted to the features it contains, i.e. (1, 1)) and the classification '1', and train C_2 with the row (restricted to (0, 0)) and the classification '0'.

Using this method for each row, we train each classifier 5 times.

				C_1		C_2	
Row	Row . G_1	Row . G_2	Marks	Training instance	Class	Training instance	Class
A	2	0	1	(1, 1)	1	(0, 0)	0
B	1	0	1	(1, 0)	1	(0, 0)	0
C	0	2	1	(0, 0)	0	(1, 1)	1
D	1	1	2	(1, 0)	1	(0, 1)	1
E	1	0	0	(0, 1)	0	(0, 0)	0

Now that training is complete, classifying a new answer is a simple process. For example, the answer "The big hamster had some exercise on his wheel" is converted to the vector (0, 1, 0, 1) using the 4 features. We take the vector (0, 1) corresponding to the group G_1 and put it through C_1. The classifier hopefully should return '0' given that a dog isn't mentioned and the 'big' adjective is irrelevant. Next we take the vector (0, 1) corresponding to the group G_2 and put it through C_2. The classifier should return '1' given that 'exercise' is mentioned. 0 + 1 = 1, therefore the final classification for this answer is 1.

Pre-Testing Analysis

Before this system is tested, it is beneficial to establish a hypothetical upper bound on accuracy compared with conventional classifiers. While cumulative classification has some merits, its cumulative nature leads unfortunately to cumulative error.

Consider the case where the maximum mark is 1. Since partitioning the feature set into 1 leaves it unaffected, the algorithm behaves the same as a normal classifier. Suppose that we assign a probability *p* that the classification for an instance is correct, and for simplification, presume this is the same probability as classifying any mark in a mark scheme correctly. If a mark of *k* out of *n* was obtained, we obtain the following probabilistic upper bound for a correct overall classification being received:

$$\mathrm{P(correct)} = \sum_{r=0}^{\min(k,n-k)} \binom{k}{r}\binom{n-k}{r} p^{n-2r}(1-p)^{2r}$$

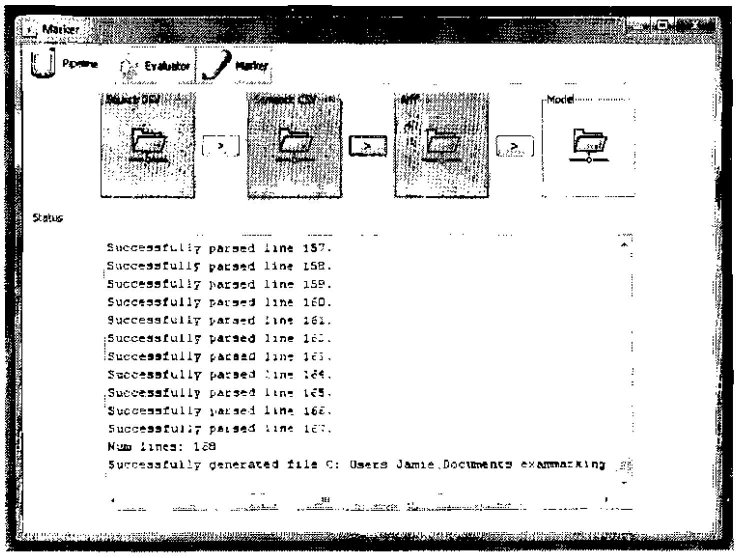
This upper bound could only theoretically be reached if the segmentation is optimal, and if in training we always identify the correct clusters for which a mark was awarded. This expression seems complex, but is a simple modification of a standard binomial distribution. To understand it, consider firstly the probability of the examiner classifying all marks correctly. This is simply p^n. Now consider that the examiner accidently assigns 2 marks incorrectly, such that for one mark it was wrongly identified as incorrect, and another wrongly identified as correct. The total classification would remain the same, since we are not changing the total number of marks. There are ${}^{k}C_1 = k$ ways of picking a 1 to switch to a 0, and ${}^{(n-k)}C_1 = (n-k)$ ways of picking a 0 to switch to a 1. (n-2) marks are assigned correctly with probability p (giving p^{n-2r}) and 2 marks are assigned incorrectly with probability (1-p). We generalize this to *2r* mistakes where *r* is the number of incorrect pairs, and sum over all the possible number of pairs of mistake. There is a maximum of *min(k,n-k)* mistakes, since we are limited by the number of 0s we can pair with 1s, or 1s we can pair with 0s.

To assess the practical impact of this, take p to be a sensible value of 0.85. Presuming a random distribution of marks out of 4, the average upper bound for the probability of correct classification is 0.55462. Given that errors in clustering and training are likely to lower the accuracy, this upper bound is bad news. However, accuracy is not the only method of evaluation, and a measure known as 'disposition' is introduced later to justify the usefulness of this classification technique.

8. Graphical User Interface

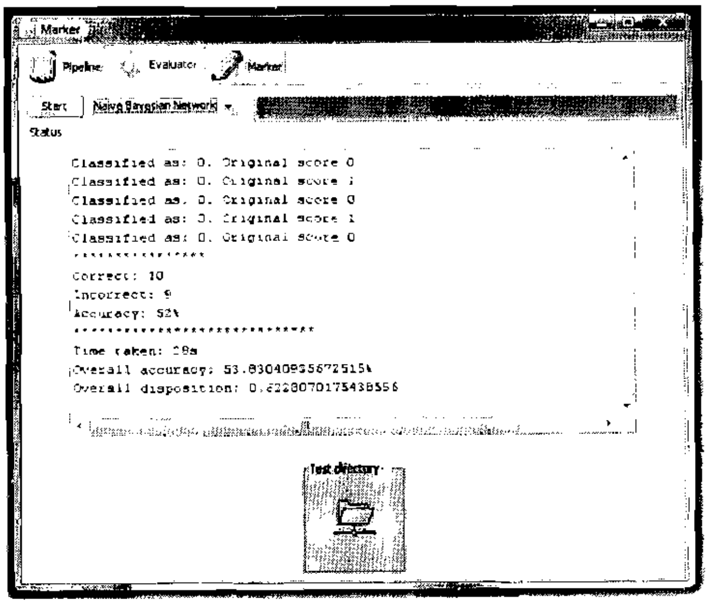
Currently all methods and operations discussed previously are only accessible via the command line. When dealing with numerous files across the conversion pipeline, this can be somewhat of a cumbersome process. Therefore, a simple GUI (Graphical User Interface) was developed and used to facilitate such operations.

The GUI consists of 3 tabbed panels. The first allows conversion between various file formats across the overall pipeline. The first button takes a CSV of English answers and parses them to produce a CSV of DRS expressions. The second button uses this and produces an ARFF, and finally the last button uses the ARFF to train a classifier (prompting the user to select between the different classification methods).



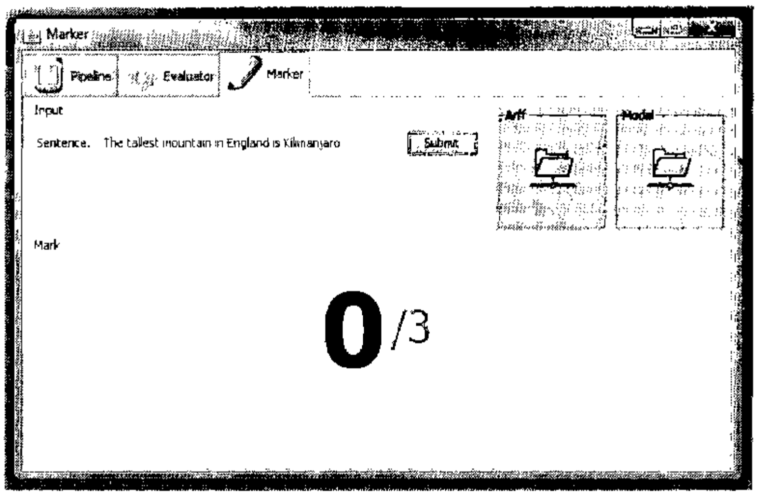
```
Successfully parsed line 157.
Successfully parsed line 158.
Successfully parsed line 159.
Successfully parsed line 160.
Successfully parsed line 161.
Successfully parsed line 162.
Successfully parsed line 163.
Successfully parsed line 164.
Successfully parsed line 165.
Successfully parsed line 166.
Successfully parsed line 167.
Num lines: 168
Successfully generated file C: Users Jamie Documents exammarking
```

A second panel allows the user to test the performance of the system for a given dataset. The user simply selects their desired training type and the folder containing the input data (which must contain a file named 'answers.csv'), and the results are printed.



```
Classified as: 0. Original score 0
Classified as: 0. Original score 1
Classified as: 0. Original score 0
Classified as: 0. Original score 1
Classified as: 0. Original score 0
*****************
Correct: 10
Incorrect: 9
Accuracy: 52%
*****************************
Time taken: 28s
Overall accuracy: 53.83040935672515%
Overall disposition: 0.6228070175438556
```

Finally, a third panel allows individual answers to be classified given a trained classifier.

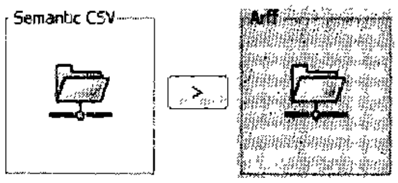


Sentence. The tallest mountain in England is Kilimanjaro

0/3

There are two features of the GUI worth drawing attention to:

- *Drag & Drop file selection box*: The class *FileDragPanel* is simply a component, and allows the user to specify a file or directory quickly. It has 'drag and drop' functionality, allowing the user to drag a file or directory into the box to make the selection. Alternatively (given that the file to specify may not yet exist if it is the output of an operation) the user can click the panel to open a standard browse dialog. The constructor takes a file type (enforced when browsing) and a name, which is displayed at the top of the panel.



- *Loading bar*: The evaluation process can be long, and thus it is desirable to give the user a rough indication of the time remaining. Progress bars are easy to create and update in the Swing framework. However, an issue arises of GUI components becoming 'frozen' until the entire evaluation process is complete, resulting in a jump from 0% to 100% as it finishes. The solution is to run the evaluation process 'in the background'. The abstract class *SwingWorker<Void, Void>* has a method *doInBackground()*; by implementing this, we ensure that updates to the GUI during an operation are administered. The progress is merely an approximation, with 0-20% progress for parsing, 20-40% for generating the ARFF files, 40-80% for training and 80-100% for testing.

The GUI used the Swing toolkit available to Java. The 'Look & Feel' (that is, the interface style) is set to match that of the operation system; as one may have observed, all screenshots were taken while using Windows Vista.

9. Evaluation

The form of evaluation used employs a technique known as "10-fold cross-validation", commonly used in Computation Linguistic systems. This splits the training set into 10 segments, trains a classifier on 9 of the segments and classifies the remaining 1 segment. The results are averaged over the 10 possible testing configurations.

Each answer in the test segment is classified, and is compared to the actual classification from the data. A measure of the system's performance is obtained, encompassing two properties:

- *Accuracy:* The percentage of answers that were classified correctly.
- *Disposition:* The average displacement of the estimated marks to the actual marks. This is potentially a more useful measure for higher mark questions, since for a mark of 4 out of 4, an estimation of 3 is superior to an estimation of 0.

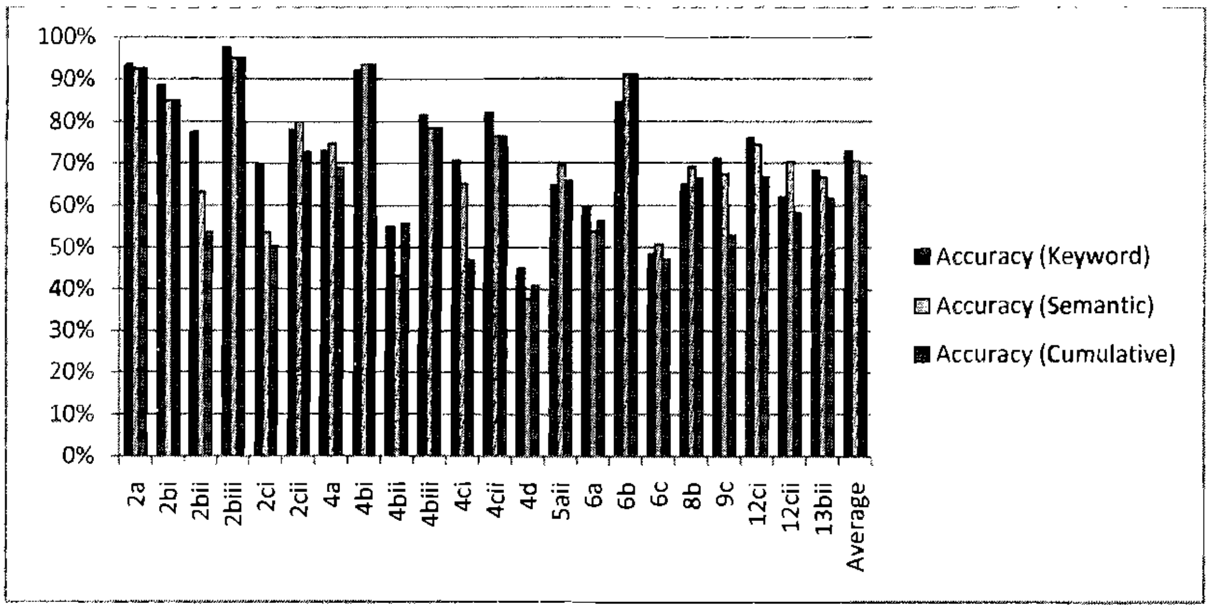
The Naive Bayesian Network was chosen as the classifier (and for the cumulative classifier as the sub-classifiers). The decision tree classifier gave on average approximately 5% worse performance, and thus was ignored. Results were obtained for the 3 different approaches: the keyword approach, the semantic approach using the parser, and the cumulative classifier described in Section 7.

Results

			Keyword Approach		Semantic Approach		Cumulative Classifier	
Q	Max mark	Num answers	Accuracy	Disposition	Acc.	Dis.	Acc.	Dis.
2a	1	200	94%	0.065	92.50%	0.075	92.50%	0.075
2bi	1	192	89%	0.115	84.87%	0.151	84.87%	0.151
2bii	2	190	78%	0.225	63.16%	0.374	53.68%	0.489
2biii	1	200	98%	0.025	95%	0.05	95%	0.05
2ci	2	189	70%	0.335	53.48%	0.518	50.32%	0.534
2cii	2	193	78%	0.220	79.92%	0.201	72.68%	0.278
4a	2	162	73%	0.306	74.74%	0.296	69.01%	0.322
4bi	1	195	92%	0.080	93.34%	0.067	93.34%	0.067
4bii	2	192	55%	0.465	43.29%	0.687	55.76%	0.504
4biii	1	68	81%	0.186	78.33%	0.217	78.33%	0.217
4ci	2	195	71%	0.305	65.13%	0.379	46.92%	0.612
4cii	1	193	82%	0.180	76.29%	0.237	76.29%	0.237
4d	3	163	45%	0.625	37.65%	0.708	40.88%	0.77
5aii	2	170	65%	0.370	69.41%	0.329	65.88%	0.37
6a	3	186	60%	0.490	53.83%	0.623	56.37%	0.506
6b	1	201	85%	0. 155	91.00%	0.09	91.00%	0.09
6c	4	139	48.5%	0.644	50.71%	0.707	47.14%	0.636
8b	2	186	65%	0.366	69.06%	0.332	66.26%	0.382
9c	2	182	71%	0.300	67.35%	0.332	52.75%	0.478
12ci	2	168	76%	0.240	74.41%	0.256	66.73%	0.339

12cii	2	158	62%	0.415	70.29%	0.316	58.17%	0.463
13bii	4	119	68%	0.522	66.67%	0.567	61.67%	0.558
Avg			73%	0.309	70.47%	0.340	67.07%	0.370

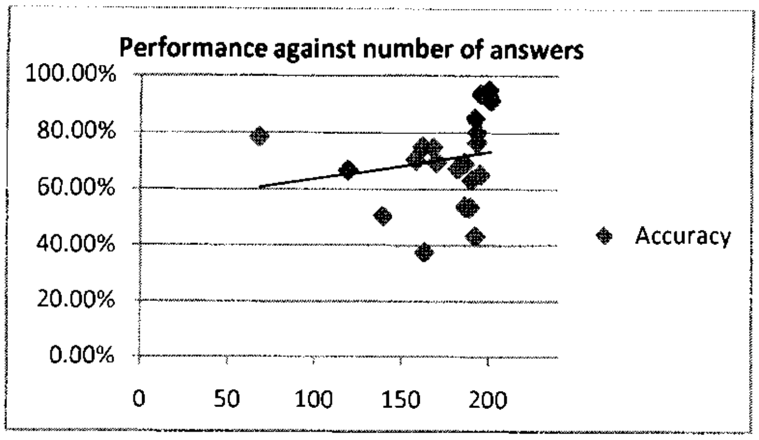
	Keyword Approach	Semantic Approach	Cumulative Classifier
Mark	Average Accuracy	Average Accuracy	Average Accuracy
1	88.71%	87.33%	87.33%
2	69.45%	66.39%	59.83%
3	52.50%	45.74%	48.63%
4	58.25%	58.69%	54.41%

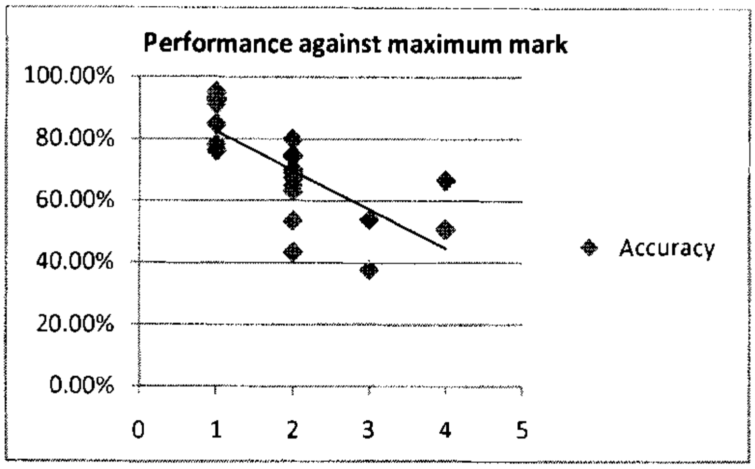


The first noticeable property of the above graph is that the semantic approach was marginally worse than the keyword approach. This is initially surprising, given that one would expect the set of features produced from the keyword approach to be a subset of that for the semantic approach, and thus the performance at least as good. There is a variety of explanations for this decrease in performance:

1. Some variation of performance is expected given that the input data used was different. While the keyword approach used all input data, some badly spelled/ungrammatical answers were rejected by the parser, and thus discarded from the training data.
2. The parser had the effect of transforming some of the words, such as changing plurality or removing tense, and other forms of 'noise'.
3. Similarly, the parser is not 100% accurate, and therefore some answers will have an incorrect semantic representation.

More generally, the results show impressive performance for 1 mark questions, but a considerable decline in accuracy as the mark increased. The cumulative classifier gave relatively poorer performance to the semantic approach for 2 mark questions, but better performance for a few higher mark questions. Accuracy had some correlation with the size of the data set, albeit small.





The disposition data did not yield any significant trends across the 2 classifiers, other than being approximately proportional to the accuracy. There was only one dataset in which the accuracy was lower for the cumulative classifier but had a higher disposition. This is likely to be because the modal mark for each dataset tended to be 0; thus if a 4 out of 4 mark answer is classified (incorrectly) by the cumulative technique as 3, and as 0 by a standard classifier, the low frequency of such high-mark answers renders such improvement (i.e. a classification closer to the actual mark) insignificant.

For a given dataset, the wildly variable accuracies for each segment in the 10-fold cross-validation are also indicative of inadequate training data. If a large dataset was used, one would expect these accuracies to be roughly consistent. It should be noted that the parser was not always successful in producing a parse, particularly for long answers. This is unfortunately unavoidable, given the poor quality of grammar and spelling in many of the answers.

Cumulative Classifier

The poorer than expected results for the cumulative classifier can partially be explained by the often poor datasets. As identified above, the modal mark was often 0, and for higher mark questions, full-mark or near full-mark answers were very infrequent (in some cases appearing only once or twice). This, coupled with the low amount of training data, means that many of the individual mark sub-classifiers predominantly receive a 0 as a classification in the training data.

It is curious that questions *4d* and *6a* had very contrasting performance despite both having the same maximum mark. Inspecting the mark scheme, this is likely to be because there was less overlap of features associated with marks in *6a* (a question on plants, in which the marks were fairly independent concepts) than there was with *4d* (a question on vasoconstriction, in which 2 of the 3 marks made reference to blood). This strongly suggests that the cumulative classifier performs better when the clustering is less ambiguous.

Spelling Correction

Question	Accuracy	Change
4a	73.30%	-1.44%
5aii	68.82%	-0.59%
6a	52.78%	-1.05%
6b	88.05%	-2.95%
6c	50.90%	0.19%
8b	68.54%	-0.52%
9c	66.76%	-0.59%
12ci	72.46%	-1.95%
12cii	68.13%	-2.16%
13bii	67.69%	1.02%
Average:		-1.00%

The results show that spelling correction yields no improvement on accuracy. There are a number of reasons why there may have been a slight reduction in performance:

- The correction of words integral to the grammatical structure of the sentence led to less answers being rejected by the parser. This combined with the other alterations to the dataset leads to a minor change in accuracy, either positive or negative.
- The spelling corrections may be too generous – that is, an incoherent answer with key terms misspelled may be penalised by the examiner. This results in some classifications in the training set being too high.

10. Conclusion

The results are a mixture of encouragements and discouragements. While the spell correction and cumulative classifier enhancements failed to provide an increase in performance, accuracy for many of the data sets was high. However, there was a disappointing lack of improvement in performance over the 'bag of words' approach, even if this performance was significantly superior to that of the 'k nearest neighbour' approach in [PULMAN 06]. Unless there were fundamental inadequacies in the method of featurisation used, it is clear that semantic information from the sentences bears little or no influence on classification on the type of data used. This is perhaps of little surprise; while theoretically semantics are of paramount value, in the context of factual exam questions its use is limited. Typically one would not expect examinees to write incoherent answers containing the desired 'keywords' in an attempt to fool the system – instead the inclusion of such keywords in an answer is a strong reflection that the student has understood the topic at hand.

Despite this, the very high accuracies in some test segments suggest that accuracies could be significantly improved by datasets of a larger size, and thus one could deem the results 'inconclusive' given the inadequacy of the data.

Ultimately, the problem with a machine learning approach to automated exam marking is the existence of a no-win situation. A small training set leads to poor results, but a large training set is counterproductive for the classification problem at hand; each classifier can only be used for a specific question, and thus unless the total number of exam papers is very large, manually marking a large subset of these is not economical.

The conclusion is that the approach of manually specifying the mark scheme required (via the patterns previously explained in the introduction) is superior to attempting to learn such patterns. The cumulative classifier may aid in identifying possible groups of features for each mark, but given 100% accuracy in the segmentation algorithm is unrealistic, this leads to features occurring in incorrect groups, and thus renders such information limited in use. The cumulative classifier does however provide some 'feedback' where traditional classifiers would not, in that the classifier indicates which of marks in the classification were and were not awarded.

There are number of extensions that may be made to this research. A more accurate segmentation algorithm that doesn't rely on the co-occurrence matrix could be investigated. Relying solely on the co-occurrence matrix is possibly naive, as one cannot assume that two features occurring frequently together necessarily relate to the same mark (and as discussed, limiting the dataset to 1-mark answers is too restrictive). One might also propose that the system may effectively be used in another domain where one large dataset is used to classify instead of multiple smaller ones. Other optimisations could also be investigated, such as 'stemming', which merges classes of words containing variants of the same word (e.g. "fertilise", "fertilised" and "fertilisation") into a single representation.

11. Implementation notes

All coding was written in Java (1.6).

a) Parsing

The *WebsiteParser* class used for 'online parsing' simply inserts the English sentence into the URL (acting as a parameter), obtains the resulting page (using Java's *InputStreamReader* on the URL as a stream), and extracts the DRS expression from it.[7]

ServerThread is a class used to manage a particular connection. As an extension of Java's *Thread* class, such managers can be run concurrently to service multiple connections simultaneously.

Instructions to the SOAP server and client can be made via the command-line. We can execute such instructions from Java by constructing a *Process* object. This can be obtained by *Runtime.getRuntime().exec(command)*. The *Process* class usefully has a method *waitFor()*, which causes the method to halt until the scheduler signals completion of the process. One might wonder why inelegant file writing and reading were used in *ServerThread* instead of piping. This is because the command-line functionality is very limited in Java, and thus can only execute basic commands.

The following command-line prompts are used in implementing the server:

- soap_server --server localhost:9000 --candc ../candc/candc-1.00/models/boxer
  is used to start the server on the local machine (using an arbitrary port), specifying the Boxer data as the parsing model to use. Since there is no way of determining whether the parser has fully loaded, we wait a fixed amount of time (20 seconds) before continuing.
- soap_client --input inputFile --output parsedFile --url http://localhost:9000
  This command instructs the SOAP client to communicate with the SOAP server. The parsed output (in CCG form) is outputted to a file. The port must be the same as that specified in initialising the SOAP server.
- boxer --input parsedFile --output finalOutputFile --box true –flat
  This takes the parsed file and uses Boxer to produce the DRS expression. We extract this from the output file and write this to the output stream of the server.

b) Implementing φ

Implementing the φ function involves 2 stages. The first is to convert the DRS string obtained from parsing into some object oriented form. The second takes this construction to generate the desired atoms. The *DRSreader* class in the 'parser' package is used to execute this first stage, producing a *SemanticModel* object. A *SemanticModel* represents all the variables, predicates and relations (including equality, etc.) within the DRS.

To do this conversion, we establish an abstract class *Expr*. This has 2 unimplemented methods:

[7] Such a technique of "web-scraping" is generally discouraged for large amounts of data, and therefore *WebsiteParser* can only be used to obtain parses for individual sentences (i.e. for classification rather than training purposes).

- *int parse(List tokens, Integer index)*: Takes a list of tokens (where each token is a word or character of the DRS string) and the point to which we have currently scanned. This parses the tokens in a way relevant to the part of the DRS tree we are currently in, and builds up the internal structure of this object. The integer returned is the new token position after parsing.
- *void process(SemanticModel model)*: Used for the second phase. It takes the structure of the object accumulated from parsing, and updates the *SemanticModel.*

We use an abstract class as opposed to an interface so that we can define a method *simpleParse*, which parses a single token (containing no information, such as parenthesis) and produces an error if the token is not as expected.

Clearly we must first convert the DRS string to tokenised form. A method *List<Object> tokenize(String str)* in *DRSreader* performs this, using Java's *StringTokenizer* class. To then initialise the parsing process, we just then construct a new *Drs* object (representing the root of the new tree being constructed), and then execute its parse method with these tokens. One might wonder why a *List* of tokens is used and not a more intuitive data structure such as an *Iterator*, where once a token is consumed, it is not seen again. The reason is that sometimes we wish to 'peek' at the current token without consuming it, in scenarios where there are multiple types of children that a node could have.

Now that a tree structure of the DRS has been obtained, we use it to construct *SemanticModel*. This object contains the following items:

- A map of variable names to *SemanticVar*, where a *SemanticVar* represents a variable in the DRS. Each has a reference to identify it (e.g. x1), a name of the object/event it is representing (e.g. dog), and a list of attributes (e.g. [hairy, large]). It has methods to set its name/add an attribute[8], set its cardinality, and to obtain a list of its semantic atoms (as defined earlier).
- A list of *SemanticRel*, representing relations in the DRS. All relations are binary, therefore we have a reference to the left and right variables (we use their name rather than a pointer to the variable itself), and the predicate being applied. It similarly has a method to generate a list of semantic atoms as defined in Section 4.

SemanticModel maintains all other relations, that is, equalities and implications, and has methods to generate their atoms appropriately. It is possible to have nested *SemanticModels*, in the case when the DRS expression uses the *'Prop'* (i.e. proposition) construction. When processing this, we construct a new child *SemanticModel*, set its parent to the current model, and redirect the processing of the sub-expression to the child model. Therefore when looking up a *SemanticVar* for a given name (a process required by *SemanticRel*), if it is not contained within the local context, we recursively search its ancestors.

The *KeywordIdentifier* class can be used to display the list of atoms for a DRS expression. While not used in the overall pipeline, it is useful for analytical purposes.

[8] We can determine an entity's type by its *'posType'* value. If we have *hairy(x1)* and hairy has the type 'a' (indicating an attribute) we add it to the list. If we have *dog(x1)* and dog has the type 'n' (for noun), we set the variable's name.

c) Dictionary Methods

The class *SpellUtils* provides access to methods associated with misspellings, such as determining whether a word is incorrect misspelled, and if so, producing a list of suggestions. An external library *"Swabunga Spell Engine"* was used, which provides such methods.

d) Cumulative Classification

- *Manual Mark Group Specification*

  A *marklist* file simply contains rows of the form *"feature -> [list of groups]"*. If the directory the classifier model is being outputted to contains a file named *'main.marklist'*, then this will override any other segmentation method used. A class *MarkListModifier* provides a command-line interface to reduce the hugely time-consuming burden of manually assigning groups, with commands such as *"CONTAINS word group"* to add any feature containing 'word' to the specified group.

- *Training*

  A class *ClusterUtils* provides methods to perform this ranking and converting vectors to their appropriate form:

  List<Integer> getClustersBySimilarity(double[] row, List<Set<Integer>> clusters) takes the row and clusters and produces a ranked list of cluster indices using the cosine measure.
  double[] getClusterIncidenceVector(Set<Integer> clusters, int numFeatures) puts the cluster in a vector form for use in the above method.
  Lastly, *double[] getRowRestrictedToCluster(Set<Integer> cluster, double[] row)* removes values from the vector that correspond to features not in the cluster. This is the 'instance' used in training each classifier.

12. Bibliography

[PULMAN 05] S. G. Pulman and J. Z. Sukkarieh (2005): Automatic Short Answer Marking.

[PULMAN 06] Stephen G. Pulman, Nicholas J. Raikes and Jana Z. Sukkarieh (not yet published): Customising an automated marking system.

[SHERMIS 03] Mark D. Shermis, Jill Burstein, (2003): Automated Essay Scoring: A Cross-disciplinary Perspective.

[CLARK 07] James R. Curran, Stephen Clark, and Johan Bos (2007): Linguistically Motivated Large-Scale NLP with C&C and Boxer. Proceedings of the ACL 2007 Demonstrations Session (ACL-07 demo), pp.29-32.

Artificial examiners put to the test: http://news.bbc.co.uk/1/hi/technology/6961088.stm

Data Mining: Practical Machine Learning Tools and Techniques (Second Edition). Ian H. Witten, Eibe Frank (*documentotion for the Weko tools*).

13. Acknowledgements

I would like to thank my project supervisor Professor Stephen Pulman for his helpful comments and pointing me towards useful sources and tools. I would also like to thank the OCR Exam Board for their provision of training data.

```java
package schemagenerator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

import weka.core.Matrix;

/**
 * This class provides access to the information contained
 * within an Arff file, such as the number of features, each
 * of their counts, and a matrix representing the data.
 * @author Jamie
 *
 */
public class ArffReader implements Serializable
{
   Matrix matrix;
   // The classifications as defined by the Arff file.
   List<Integer> classifiedAs = new LinkedList<Integer>();
   // The total number of times each of the features (represented
   // by its index) occurs, that is, the summation of the column.
   Map<Integer,Integer> featureCounts = new TreeMap<Integer,Integer>();

   String fileName;
   int maxMark = 0;
   List<String> attributes = new LinkedList<String>();
   public ArffReader(String fileName)
   {
      this.fileName = fileName;
      read();
   }


   /**
    * Reads the Arff file. Executed by the constructor.
    */
   public void read()
   {
      try
      {
         // Get attributes and read up to "@data"
         BufferedReader br = new BufferedReader(new FileReader(fileName));
         boolean reading = true;
         int numCols=0;
         while(br.ready()&&reading)
         {
            String line = br.readLine();
            if(attributes.size()!=0&&line.indexOf("@attribute")<0)
               reading = false;
            else if(line.indexOf("@attribute")==0){
               String[] parts = line.split(" ");
               if(!parts[1].equals("score")){
                     attributes.add(parts[1].replaceAll("\"", ""));
                     numCols++;
               }
            }
         }
         br.readLine(); // @data

         // Now read the data.

         int numRows = 0;
         List<List<Double>> vals = new LinkedList<List<Double>>();

         while(br.ready())
         {
            String[] lineParts = br.readLine().split(", ");
            List<Double> row = new LinkedList<Double>();
            numRows++;
            for(int i=0; i<(lineParts.length-1); i++)
            {
               Double val = new Double(lineParts[i]);
               // If necessary, we increment the count for that feature.
               if(val==1.0)
               {     if(featureCounts.containsKey(i))
                        featureCounts.put(i, featureCounts.get(i)+1);
                     else featureCounts.put(i, 1);
               }
               row.add(val);
            }
            vals.add(row);

            int mark = new Integer(lineParts[lineParts.length-1]);
            classifiedAs.add(mark);
            if(mark>maxMark)maxMark = mark;
         }

         Matrix mx = new Matrix(numRows,numCols);

         for(int i=0; i<numRows; i++)
         {     for(int j=0; j<numCols; j++)
               {
                  mx.setElement(i,j,vals.get(i).get(j));
               }
         }
         br.close();

         matrix = mx;

      } catch(Exception e){
         throw new Error(e);
      }
   }



   /**
    * Gets the prescribed mark for an answer with index i.
    */
   public Integer getActualMark(Integer i)
   {     return classifiedAs.get(i);     }

   public Integer getFeatureCount(Integer i)
   {     return featureCounts.get(i);     }

   public Matrix getMatrix()
   {     return matrix;                   }

   /**
```

```java
     * @deprecated
     * The method returns a matrix consisting only of rows with the given mark
     * classification. Initially in training, only rows with a classification
     * of 1 were taken. The logic was that such answers would contain features
     * only relating to predominantly one mark, and thus identifying groups
     * would be more accurate. However in practice this did seem to be the
     * case, and had the drawback of discarding the majority of the data set.
     */
    public Matrix getMatrix(int mark)
    {

        int cols = matrix.numColumns();
        int numRowsWithMark = 0;
        for(int i=0; i<classifiedAs.size(); i++)
            if(classifiedAs.get(i)==mark)numRowsWithMark++;

        Matrix matrixForMark = new Matrix(numRowsWithMark,cols);

        int currentRow = 0;
        for(int i=0; i<matrix.numRows(); i++)
        {
            if(classifiedAs.get(i).intValue()==mark)
            {
                for(int j=0; j<cols; j++)
                    matrixForMark.addElement(currentRow, j,
                            matrix.getElement(i, j));
                currentRow++;
            }
        }
        return matrixForMark;
    }


    public int getMaxMark()
    {
        return maxMark;
    }

    public List<String> getAttributes()
    {
        return attributes;
    }


    /**
     * @deprecated
     * This method is no longer used.
     * Given a list of groups (i.e. sets of feature indices) and the original
     * matrix, it would produce a new matrix, with #groups columns, that
     * indicated whether each of the rows in the original matrix 'matched'
     * the group. For example, the nth row in the matrix of [1, 0, 1] would
     * indicate that the nth row of the original matrix 'matched' groups 1 and
     * 3, but not 2. This method was used when group identification was done
     * in 2 stages.
     */
    public Matrix getGroupMatrix(List<Set<Integer>> groups, Matrix mx)
    {
        Matrix gMx = new Matrix(mx.numRows(), groups.size());
        for(int i=0; i<mx.numRows(); i++)
        {
            int j=0;
            for(Set<Integer> group : groups)
            {
                if(matchesGroup(group,mx.getRow(i)))
                    gMx.setElement(i, j, 1.0);
                else gMx.setElement(i, j, 0.0);
                j++;
            }

        }
        return gMx;
    }

    /**
     * @deprecated No longer used. This method (referenced above) indicates whether a row
     * matches a particular group. This was done by taking the elements of the
     * row occurring in the group, counting the number of 1s, and seeing if
     * this was greater the root of the size of the group.
     */
    public static boolean matchesGroup(Set<Integer> s, double[] row)
    {
        int matching = 0;
        for(int i=0; i<row.length; i++)
        {
            if(row[i]==1.0&&s.contains(i))matching++;
        }

        if(new Double(matching)>=Math.floor(Math.sqrt(new Double(s.size()))))
            return true;
        else return false;
    }

}
```

schemagenerator/ClustererWrapper.java

```java
package schemagenerator;

import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

import weka.clusterers.Clusterer;
import weka.clusterers.SimpleKMeans;
import weka.core.Attribute;
import weka.core.FastVector;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Matrix;

/**
 * Provides a less cumbersome way of accessing Weka's clustering
 * facilities, using the incidence matrix as input data.
 * @author Jamie
 *
 */
public class ClustererWrapper {

   Matrix incidence;
   int numClusters;
   Clusterer clusterer = new SimpleKMeans();

   public ClustererWrapper(Matrix incidence, int numClusters)
   {
        // We normalise the columns to improve the accuracy
        // of the clustering.
        this.incidence = MatrixUtils.normaliseColumns(incidence);
        this.numClusters = numClusters;
        try
        {
           ((SimpleKMeans) clusterer).setNumClusters(numClusters);
        } catch(Exception e){
           e.printStackTrace();
        }
   }

   /**
    * Converts the Matrix to an 'Instances' object, as required
    * by the clustering.
    */
   public void buildClusterer(List<String> attrs)
   {
        FastVector fv = new FastVector(attrs.size());
        for(String s : attrs)
           fv.addElement(new Attribute(s));

        // Need matrix in Instances form.
        // Each column of the matrix is an instance.
        Instances ins = new Instances("somename", fv, attrs.size());
        for(int i=0; i<incidence.numColumns(); i++)
           ins.add(new Instance(1.0, incidence.getColumn(i)));

        try
        {
           clusterer.buildClusterer(ins);
        } catch(Exception e){
           e.printStackTrace();
        }
   }

   /**
    * Identifies the cluster for each column, and produces a list of sets,
    * each set representing a group of columns (in index form).
    */
   public List<Set<Integer>> getClusters()
   {
        List<Set<Integer>> clusters = new LinkedList<Set<Integer>>();
        for(int i=0; i<numClusters; i++)
           clusters.add(new HashSet<Integer>());

        for(int i=0; i<incidence.numColumns(); i++)
        {
           try
           {
                int cNum = clusterer.clusterInstance(
                   new Instance(1.0, incidence.getColumn(i)));
                clusters.get(cNum).add(i);
           } catch(Exception e){
                e.printStackTrace();
           }

        }

        return clusters;
   }

}
```

```java
package schemagenerator;

import java.util.LinkedList;
import java.util.List;
import java.util.Set;

/**
 * Provides utilities with regards to clusters (where
 * each cluster represents a group of related features
 * intending to represent one mark of the mark scheme).
 * @author Jamie
 *
 */
public class ClusterUtils {

   // Return an ordered list of cluster indexes,
   // such that the first cluster of the list matches
   // the row to the greatest extent.
   public static List<Integer> getClustersBySimilarity(
         double[] row, List<Set<Integer>> clusters)
   {

         List<Double> cosSims = new LinkedList<Double>();

         for (int i = 0; i < clusters.size(); i++) {
            // First turn cluster into incidence array
            double[] clusterArray = getClusterIncidenceVector(clusters.get(i),
row.length);
         }

         // Now get vector (cosine) similarity
         // Ignore length of row, since constant 'given the row'.
         double cosSim = 0.0;
         for (int k = 0; k < clusterArray.length; k++)
            cosSim += clusterArray[k] * row[k];
            cosSim = cosSim / euclideanDistance(clusterArray);
            cosSims.add(cosSim);
         }

         // Generate list clustersBySim

         List<Integer> clustersBySim = new LinkedList<Integer>();

         double lastMax = 10000;
         while (clustersBySim.size() < cosSims.size()) {
            int currentMaxCluster = -1;
            double currentMax = -1.0;
            for (int i = 0; i < cosSims.size(); i++) {
                  if (cosSims.get(i) >= currentMax && cosSims.get(i) <  lastMax
                     && !clustersBySim.contains(i))
                  {
                     currentMax = cosSims.get(i);
                     currentMaxCluster = i;
                  }
            }
            clustersBySim.add(currentMaxCluster);
            lastMax = currentMax;
         }

         // System.out.println("Cos sims: "+cosSims);

         return clustersBySim;
   }

   // Returns a vector representing which features occur with n
   // the cluster. e.g. A cluster of
   // { 0, 1, 3, 5 } -> {1, 1, 0, 1, 0, 1}
   public static double[] getClusterIncidenceVector(Set<Integer> cluster, int
numFeatures) {
         double[] clusterArray = new double[numFeatures];
         for (int i = 0; i < numFeatures; i++)
            clusterArray[i] = 0.0;

         for (Integer i : cluster)
            clusterArray[i] = 1.0;

         return clusterArray;
   }


         private static double euclideanDistance(double[] vArray) {
                  double temp = 0.0;
                  for (int i = 0; i < vArray.length; i++)
                           temp += Math.pow(vArray[i], 2);
                  temp = Math.sqrt(temp);
                  return temp;
         }

   // This gives a vector containing only the feature values
   // corresponding to features contained by the cluster.
   // For example, if the cluster is { 0, 1, 4 } and the row
   // [1, 0, 0, 0, 1], then we obtain the vector [1, 0, 1]
   public static double[] getRowRestrictedToCluster(Set<Integer> cluster,
         double[] row)
   {
         double[] clusterArray = getClusterIncidenceVector(cluster,row.length);
         double[] toReturn = new double[getClusterVectorSize(clusterArray)];
         int count = 0;
         for (int i = 0; i < clusterArray.length; i++)
            if (clusterArray[i] == 1.0) {
                  toReturn[count] = row[i];
                  count++;
            }
         return toReturn;
   }

   public static int getClusterVectorSize(double[] clusterArray) {
         int oneCount = 0;
         for (int i = 0; i < clusterArray.length; i++)
            if (clusterArray[i] == 1.0)oneCount++;
         return oneCount;
   }
}
```

```java
package schemagenerator;

import java.io.PrintWriter;
import java.util.List;
import java.util.Set;

import weka.classifiers.Classifier;
import weka.classifiers.bayes.NaiveBayes;
import weka.core.Attribute;
import weka.core.FastVector;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Matrix;

/**
 * A classifier which takes into account the cumulative nature of a
 * classification. That is, the classifications x and y if 2 independent
 * properties X and Y are satisfied correspondingly should result in a
 * classification of x+y if both X and Y are satisfied.
 *
 * @author Jamie
 *
 */
public class CumulativeClassifier extends Classifier {
    // The classifiers for each mark.
    Classifier[] classifiers;
    // The data set relating to each mark.
    Instances[] instancesArray;
    // The features relating to each mark in the mark scheme.
    List<Set<Integer>> clusters;
    int totalClusters = 0;
    // Note that the maximum mark is not necessarily
    // the same as the number of clusters (i.e. if the mark
    // is limited to a maximum below the total feasible marks).
    int maxMark = 0;
Outputs the mark scheme to a .markscheme file.
    String outputFile = null;
    // Optionally inputs a manually constructed .marklist
    // file, which overrides the clustering and specifies
    // which mark each feature corresponds to.
    String inputFile = null;

    public CumulativeClassifier() {
    }

    public CumulativeClassifier(String inputFile, String outputFile) {
        this.inputFile = inputFile;
        this.outputFile = outputFile;
    }

    public CumulativeClassifier(String inputFile, String outputFile,
        int totalClusters)
    {
        this(inputFile, outputFile);
        this.totalClusters = totalClusters;
    }

    public void buildClassifier(Instances instances) throws Exception {

            // The feature strength matrix is an nxn matrix (where there are n
          // features) such that M(i,j) indicates the number of times features
            // i and j appear together in the dataset.
        Matrix featureStrengthMx = MatrixUtils
                .getIncidenceMatrix(InstancesReader
                        .getMatrixFromInstances(instances));

        maxMark = InstancesReader.getMaxMark(instances);
        // If we haven't got the total number of clusters,
        if (totalClusters == 0)
                totalClusters = maxMark;

        List<String> attributes = InstancesReader.getAttributes(instances);

        if (inputFile == null)
                clusters = GroupIdentifier.identifyGroupsUsingClustering(
                                featureStrengthMx, attributes, maxMark);
        // else clusters = SchemaUtils.readMarkSchemeFile(inputFile,
        // attributes);
        else
                clusters = SchemaUtils.readMarkListFile(inputFile, attributes,
                        totalClusters);

        classifiers = new Classifier[totalClusters];
        instancesArray = new Instances[totalClusters];

        // STEP 0:
        // Output mark scheme if necessary.
        if (outputFile != null) {
                System.out.println("Outputting mark scheme to: "
                        + outputFile);
                PrintWriter pw = new PrintWriter(outputFile);
                pw.println(attributes);
                for (Set<Integer> cluster : clusters)
                        pw.println(cluster);
                pw.close();
        }

        // STEP 1:
        // Initiate the classifiers, and start a fresh instances object
        // for each of the classifiers, each with the correct attribute
        // information.

        for (int i = 0; i < totalClusters; i++) {
                classifiers[i] = new NaiveBayes();
                // classifiers[i] = new DecisionTable();
        }

        for (int i = 0; i < maxMark; i++)
        {
            int sizeOfCluster = ClusterUtils.getClusterVectorSize(ClusterUtils
                    .getClusterIncidenceVector(clusters.get(i), attributes
                            .size()));
            FastVector fv = new FastVector(sizeOfCluster + 1);
            for (int j = 0; j < sizeOfCluster; j++)
                    fv.addElement(new Attribute("attr" + j));

            // The classification is a NOMINAL. Either 0 or 1,
            // therefore need to make group.
            FastVector binary = new FastVector(2);
            binary.addElement("0");
            binary.addElement("1");
            fv.addElement(new Attribute("score", binary));
            instancesArray[i] = new Instances("iSet" + i, fv, 0.0);
            instancesArray[i]
```

```
                        .setClassIndex(instancesArray[i].numAttributes() - 1);
            }

        // STEP 2:
        // Now populate the instances object for each of the classifiers.

        Matrix oMx = InstancesReader.getMatrixFromInstances(instances);

        for (int i = 0; i < oMx.numRows(); i++) {

            double[] row = oMx.getRow(i);

            // Need to find clusters it's most similar to
            List<Integer> clustersOrdered = ClusterUtils
                    .getClustersBySimilarity(row, clusters);
            int classification = InstancesReader
                    .getClassification(instances, i);

            // We take the first 'classification' clusters from ordered
            // clusters, and train the networks for those clusters.
            for (int j = 0; j < clusters.size(); j++) {
                // Train cluster

                // But first, need to construct instance!
                // To do this, we simply map incidence vector for cluster
                // on the row.
                double[] clusterArray =
                    ClusterUtils.getRowRestrictedToCluster(
                            clusters.get(clustersOrdered.get(j)), row);
                Instance instance = new Instance(clusterArray.length + 1);
                instance.setDataset(instancesArray[clustersOrdered.get(j)]);
                for (int k = 0; k < clusterArray.length; k++)
                    instance.setValue(k, clusterArray[k]);
                if (j < classification) {
                    instance.setValue(clusterArray.length, "1");
                } else {
                    instance.setValue(clusterArray.length, "0");
                }

                instancesArray[clustersOrdered.get(j)].add(instance);

            }

        }

        // STEP 3:
        // Finally, build the classifiers using the instances objects.

        try {
            for (int i = 0; i < totalClusters; i++)
                    classifiers[i].buildClassifier(instancesArray[i]);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // For the user to see, print the mark scheme
        SchemaUtils.print(clusters, attributes);
    }

    public double classifyInstance(Instance instance)
                throws java.lang.Exception {
        // For each classifier, we need to:
        // 1. Restrict the instance only to the attributes it contains.
        // 2a. Do some jiggery pokery to make the instance suitable for
        //   classification.
        // 2b. Classify this restricted instance.
        // 3a. Add together all classifications.
        // 3b. Cap the mark at maxMark.

        double total = 0.0;

        for (int i = 0; i < classifiers.length; i++) {
                double[] restrictedInstance = ClusterUtils
                    .getRowRestrictedToCluster(clusters.get(i), instance
                            .toDoubleArray());
                double[] withArbitraryClassification = appendClassification(
                    restrictedInstance, 2.0);
                Instance instanceB = new Instance(1.0,
                        withArbitraryClassification);
                instanceB.setDataset(instancesArray[i]);
                try {
                    double d = classifiers[i].classifyInstance(instanceB);
                    total += d;
                } catch (Exception e) {
                    throw new Error(e);
                }
        }

        if (total > maxMark)total = maxMark;

        return total;
    }

    private double[] appendClassification(double[] cArray, double toAppend) {
        double[] cArrayNew = new double[cArray.length + 1];
        for (int i = 0; i < cArray.length; i++)
                cArrayNew[i] = cArray[i];
        cArrayNew[cArray.length] = toAppend;
        return cArrayNew;
    }

}
```

```java
package schemagenerator;

import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

import weka.core.Matrix;

/**
 * This class supplies different methods to identify
 * groups of features based on the features' co-occurence.
 * The intended output of such methods is a list of sets,
 * where each set represents a group of features (in index
 * form).
 * @author Jamie
 */

public class GroupIdentifier
{
    /**
     * This method uses standard (simple K-means) clustering.
     */
    public static List<Set<Integer>> identifyGroupsUsingClustering(Matrix featureStrengthMx, List<String> features, int numClusters)
    {
        featureStrengthMx = MatrixUtils.normaliseToMax(featureStrengthMx);
        ClustererWrapper cw = new ClustererWrapper(featureStrengthMx, numClusters);
        cw.buildClusterer(features);
        return cw.getClusters();
    }

    // Put in mind matrix is normalised.
    // Anything under value is considered a 'weak bond'.
    // Anything above is consider 'strong'.
    final static double MINIMUM_BOND = 0.4;

    /**
     * @deprecated
     * Produces the groups by identifying cliques (where feature
     * nodes have an edge between them if their connection is
     * sufficiently strong) before merging them into the desired
     * number of groups.
     */
    public static Set<Set<Integer>> identifyGroupsUsingCliqueMerging(Matrix featureStrengthMx)
    {
        featureStrengthMx = MatrixUtils.normaliseToMax(featureStrengthMx);

        Set<Set<Integer>> groups = new HashSet<Set<Integer>>();

        for(int i=0; i<featureStrengthMx.numRows(); i++)
        {
           System.out.println("Processing row "+i);
           if(groups.isEmpty())
           {
               // Simple start case
               Set<Integer> s = new HashSet<Integer>();
               s.add(i);
               groups.add(s);
           } else {

               Set<Set<Integer>> newGroups = new HashSet<Set<Integer>>();
               boolean didntAppend = true;

               for(Set<Integer> group : groups)
               {
                  Set<Integer> strongBonds = new HashSet<Integer>();
                  Set<Integer> weakBonds = new HashSet<Integer>();

                  for(Integer j : group)
                  {     if(j!=i)
                            if(featureStrengthMx.getElement(i,j)>MINIMUM_BOND) strongBonds.add(j);

                                else weakBonds.add(j);
                  }

                  // System.out.println("Strong bonds: "+strongBonds);
                  // System.out.println("Weak bonds: "+weakBonds);

                  // If weakBonds is empty, then the element becomes part of that set.

                  if(weakBonds.isEmpty())
                  {
                       group.add(i);
                       didntAppend = false;
                  } else if(strongBonds.isEmpty())
                  {
                       // If strongBonds is empty, then we need to create a new set

                       // containing just that feature. HOWEVER, this only occurs

                       // if the feature was not added to any set.

                  } else {
                       // Both strongBonds and weakBonds is non-empty.
                       // We simply make a new group with the strongBonds
                       // and the element to add. The original group is
                       // unaffected.
                       strongBonds.add(i);
                       newGroups.add(strongBonds);
                       didntAppend = false;
                  }
               }

               if(didntAppend)
               {
                  Set<Integer> s = new HashSet<Integer>();
                  s.add(i);
                  newGroups.add(s);
               }
               groups.addAll(newGroups);
           }
        }

        return groups;
    }


    public static void main(String[] args)
    {
            // Test the above group identification
        List<String> features = new LinkedList<String>();
        features.add("A");
        features.add("B");
        features.add("C");
        features.add("D");
        features.add("E");
        features.add("F");
        features.add("G");
        features.add("H");

        Matrix mx = new Matrix(4,4);
        mx.setElement(0,0, 1.0);
        mx.setElement(0,1, 0.0);
        mx.setElement(0,2, 1.0);
        mx.setElement(0,3, 1.0);
        mx.setElement(1,1, 1.0);
        mx.setElement(1,2, 1.0);
        mx.setElement(1,3, 0.0);
        mx.setElement(2,2, 1.0);
        mx.setElement(2,3, 1.0);
        mx.setElement(3,3, 1.0);
        mirrorMx(mx);

        Set<Set<Integer>> groups = GroupIdentifier.identifyGroupsUsingCliqueMerging(mx);
        SchemaUtils.print(groups, features);

        System.out.println("");

        Matrix mx2 = new Matrix(5,5);
        mx2.setElement(0,0, 1.0);
        mx2.setElement(0,1, 1.0);
        mx2.setElement(0,2, 1.0);
        mx2.setElement(0,3, 0.0);
        mx2.setElement(0,4, 0.0);
        mx2.setElement(1,1, 1.0);
        mx2.setElement(1,2, 1.0);
        mx2.setElement(1,3, 0.0);
        mx2.setElement(1,4, 0.0);
        mx2.setElement(2,2, 1.0);
        mx2.setElement(2,3, 1.0);
        mx2.setElement(2,4, 1.0);
        mx2.setElement(3,3, 1.0);
        mx2.setElement(3,4, 1.0);
        mx2.setElement(4,4, 1.0);
        mirrorMx(mx2);

        groups = GroupIdentifier.identifyGroupsUsingCliqueMerging(mx2);
        SchemaUtils.print(groups, features);
        System.out.println("");

        Matrix mx3 = new Matrix(6,6);
        mx3.setElement(0,0, 1.0);
        mx3.setElement(0,1, 1.0);
        mx3.setElement(0,2, 1.0);
        mx3.setElement(0,3, 1.0);
        mx3.setElement(0,4, 0.0);
        mx3.setElement(0,5, 0.0);
        mx3.setElement(1,1, 1.0);
        mx3.setElement(1,2, 1.0);
        mx3.setElement(1,3, [illegible]);
        mx3.setElement(1,4, 0.0);
        mx3.setElement(1,5, [illegible]);
        mx3.setElement(2,2, 1.0);
        mx3.setElement(2,3, 1.0);
        mx3.setElement(2,4, 0.0);
        mx3.setElement(2,5, 0.0);
        mx3.setElement(3,3, 1.0);
        mx3.setElement(3,4, 0.0);
        mx3.setElement(3,5, 1.0);
        mx3.setElement(4,4, 1.0);
        mx3.setElement(4,5, 1.0);
        mx3.setElement(5,5, 1.0);
        mirrorMx(mx3);

        groups = GroupIdentifier.identifyGroupsUsingCliqueMerging(mx3);
        groups = Merger.mergeGroups(groups, mx3, 2);
        SchemaUtils.print(groups, features);

        System.out.println();

        ArffReader ar = new ArffReader("./files/13bii-0.arff");
        Matrix mx4 = MatrixUtils.getIncidenceMatrix(ar.getMatrix());
        mx4 = MatrixUtils.normaliseToMax(mx4);

//      groups = GroupIdentifier.identifyGroups(mx4);
//      SchemaPrinter.print(groups, ar.getAttributes());
//      System.out.println("Merging");
//      groups = Merger.mergeGroups(groups, mx4, ar.getMaxMark());
//      SchemaPrinter.print(groups, ar.getAttributes());

        List<Set<Integer>> clusters = identifyGroupsUsingClustering(mx4, ar.getAttributes(), ar.getMaxMark());
        SchemaUtils.print(clusters, ar.getAttributes());
    }

    /**
     * Using by the demo to copy a triangular matrix into
     * its diagonal reflection so that the resulting matrix
     * is symmetric.
     */
    private static void mirrorMx(Matrix mx)
    {
        for(int r=0; r<mx.numColumns(); r++)
            for(int c=0; c<r; c++)
                mx.setElement(r,c,mx.getElement(c,r));
    }

}
```

```java
package schemagenerator;

import java.util.LinkedList;
import java.util.List;

import weka.core.Attribute;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Matrix;

/**
 * Whereas ArffReader obtains the representative matrix and other
 * data from an Arff file, this provides similar extraction from
 * an Instances object.
 * @author Jamie
 *
 */
public class InstancesReader {
   public static Matrix getMatrixFromInstances(Instances instances)
   {
        Matrix mx = new Matrix(instances.numInstances(),
                instances.numAttributes()-1);

        for(int row = 0; row < instances.numInstances(); row++)
        {  Instance ins = instances.instance(row);
           double[] insArray = ins.toDoubleArray();
           for(int col = 0; col < instances.numAttributes()-1; col++)
           {
                mx.setElement(row, col, ins.value(col));
           }
        }
        return mx;

   }

   public static int getMaxMark(Instances instances)
   {
        double maxMark = 0.0;
        for(int row = 0; row < instances.numInstances(); row++)
        {
           Instance ins = instances.instance(row);
           double[] insArray = ins.toDoubleArray();
           double classification = insArray[instances.numAttributes()-1];
           maxMark = (classification>maxMark) ? classification : maxMark;
        }

        return (new Double(maxMark)).intValue();
   }

   public static List<String> getAttributes(Instances instances)
   {
        List<String> attributes = new LinkedList<String>();
        // Don't want the last attribute (which is 'score')
        for(int i=0; i<instances.numAttributes()-1; i++)
        {
           Attribute att = instances.attribute(i);
           attributes.add(att.name());
        }
        return attributes;
   }

   public static int getClassification(Instances instances, int index)
   {
        Instance ins = instances.instance(index);
        return (new Double(ins.toDoubleArray()[ins.numAttributes()-1]))
                .intValue();
   }
}
```

```java
package schemagenerator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * Used to convert a markscheme file into a marklist file.
 * A marklist file is a more convenient form for manual
 * editing, and has lines of the form:
 * "feature [list of groups-it-appears-in]"
 * If a file main.marklist is placed in a test directory,
 * this will override the mark groups.
 * @author Jamie
 *
 */
public class MarkSchemeLister {

   static Map<String,Set<Integer>> groupMappings =
        new TreeMap<String,Set<Integer>>();

   public static void main(String[] args) throws Exception {
        if(args.length !=0)
        {
          System.out.println("USAGE:");
          System.out.println("\tMarkSchemeLister in.markscheme out.marklist");
          System.out.println("\t\t or");
          System.out.println("\tMarkSchemeLister in.marklist out.marklist");
          System.out.println.exit(0);
        }

        String inputFile = args[0];
        String outputFile = args[1];

         // Read in
        if (inputFile.contains("markscheme"))
        {
           BufferedReader br = new BufferedReader(new FileReader(inputFile));

           String attrs = br.readLine();
           int maxMark = 0;
           while(br.ready()){
                maxMark++;
                br.readLine();
           }
           br.close();
           System.out.println("NOTE: There are "+maxMark+" groups.");

           // Now put attributes in list.
           attrs = attrs.substring(1,attrs.length()-1);
           List<String> attributes = new LinkedList<String>();
           String[] attrs2 = attrs.split(", ");
           for(int i=0; i<attrs2.length; i++)attributes.add(attrs2[i]);

           [illegible]

           // Put in map, with all initialised to empty lists (i.e. no group)
           for(String attr : attributes)
                groupMappings.put(attr, new TreeSet<Integer>());
        } else {
           // Read marklist file
           groupMappings = SchemaUtils.getGroupMappings(inputFile);

        }

         // Provide a command line interface to modify features' groups.
        System.out.println("Commands:");
        System.out.println("\tASSIGN attr groups");
        System.out.println("\tCONTAINS word groups");
        System.out.println("\tQUIT");
        System.out.println("********************************");

        BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));
        boolean quit = false;

        while(!quit)
        {
           System.out.print("> ");
           String line = br.readLine();

           String[] lineParts = line.split(" ");
           if(lineParts[0].equals("ASSIGN"))
           {
                String[] groups = lineParts[2].split(",");
                for(int j=0; j<groups.length; j++)
                   groupMappings.get(lineParts[1]).add(
                        new Integer(groups[j]));
                output(outputFile);
           }
           if(lineParts[0].equals("CONTAINS"))
           {
                String[] groups = lineParts[2].split(",");
                for(String attr : groupMappings.keySet())
                   if(attr.contains(lineParts[1])){
                        for(int j=0; j<groups.length; j++)
                          groupMappings.get(attr).add(new Integer(groups[j]));
                output(outputFile);
           }
           if(lineParts[0].equals("QUIT"))quit = true;
        }
        br.close();


        public static void output(String fileName) throws Exception
        {
                PrintWriter pw = new PrintWriter(fileName);
                for(String attr : groupMappings.keySet())
                {
                        pw.print(attr+"\t");
                        pw.print(groupMappings.get(attr));
                        pw.println();
                }
                pw.close();
        }
}
```

```java
package schemagenerator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

/**
 * @deprecated
 * @author Jamie
 *
 */
public class MarkSchemeModifier {
        public static void main(String[] args) throws Exception {
                if(args.length!= 0)
                {
                        System.out.println("USAGE:");
                        System.out.println("\tMarkSchemeModifier in.markscheme out.markscheme");
                        System.exit(0);
                }

                String inputFile = args[0];
                String outputFile = args[1];

                BufferedReader br = new BufferedReader(new FileReader(inputFile));

                String attrs = br.readLine();
                int maxMark = 0;
                while(br.ready()){
                        maxMark++;
                        br.readLine();
                }
                br.close();
                System.out.println("NOTE: There are "+maxMark+" groups.");

                // Now put attributes in list.
                attrs = attrs.substring(1,attrs.length()-1);
                List<String> attributes = new LinkedList<String>();
                String[] attrs2 = attrs.split(", ");
                for(int i=0; i<attrs2.length; i++)attributes.add(attrs2[i]);

                System.out.println("For each attribute enter a group from 1 to "+maxMark+" to place it in.");
                System.out.println("For multiple groups, separate by spaces.");
                System.out.println("---------------------------------------");


                // Now get new groups
                Set<Integer>[] newGroups = new Set[maxMark];
                for(int i=0; i<maxMark; i++)newGroups[i] = new HashSet<Integer>();


                br = new BufferedReader(new InputStreamReader(System.in));
                for(int i=0; i<attributes.size(); i++)
                {
                        System.out.print(attributes.get(i));
                        System.out.print("> ");
                        String groupSelection = br.readLine();
                        if(!groupSelection.equals(""))
                        {
                                String[] groups = groupSelection.split(" ");
                                for(int j=0; j<groups.length; j++)newGroups[new Integer(groups[j])-1].add(i);
                        }

                }

                System.out.println("---------------------------------------");

                System.out.println("Done, writing to file.");

                PrintWriter pw = new PrintWriter(outputFile);
                pw.println(attributes);
                System.out.println(attributes);
                for(int i = 0; i<maxMark; i++){ pw.println(newGroups[i]); System.out.println(newGroups[i]); }
                pw.close();
        }
}
```

schemagenerator/MatrixUtils.java

```java
package schemagenerator;

import weka.core.Instance;
import weka.core.Instances;
import weka.core.Matrix;

/**
 * Utilities relating to matrix manipulation.
 * @author Jamie
 */
public class MatrixUtils {
   public static Matrix normaliseToMax(Matrix mx)
   {
        // We take the log of all the values.
        // This ensures very high values do not scale
        // everything else down.
        // Then we scale everything so that the maximum is 1.

        Matrix copy = (Matrix) mx.clone();

        // Find max
        double currentMax = 0.0;
        for(int i=0; i<mx.numRows(); i++)
        {   for(int j=0; j<mx.numRows(); j++)
            {
                // Set element to the log of it.
                // We plus 1 since log(0) is undefined.
                // Therefore 0 -> 0 as wanted.
                copy.setElement(i,j,Math.log(copy.getElement(i,j)+1.0));
                if(copy.getElement(i,j)>currentMax)
                   currentMax = copy.getElement(i,j);
            }
        }

        // Divide everything by max
        // Diagonals are set to 1.0, such that a feature will always match
        // with itself.
        for(int i=0; i<mx.numRows(); i++)
           for(int j=0; j<mx.numRows(); j++)
                if(i!=j)copy.setElement(i, j,
                   copy.getElement(i,j) / currentMax);
                else copy.setElement(i,j, 1.0);

        return copy;
   }

   // Obtains (A^T).A
   public static Matrix getIncidenceMatrix(Matrix mx) {
        return mx.transpose().multiply(mx);
   }

   public static Matrix normaliseColumns(Matrix mx)
   {
        Matrix mx2 = new Matrix(mx.numRows(), mx.numColumns());
        // Normalises columns
        for(int i=0; i<mx.numColumns(); i++)
        {
           double squaredSum = 0.0;
           for(int j=0; j<mx.numRows(); j++)
                squaredSum+= Math.pow(mx.getElement(j, i), 2.0);
           squaredSum = Math.sqrt(squaredSum);
           for(int j=0; j<mx.numRows(); j++)
                mx2.setElement(j, i, mx.getElement(j, i) / squaredSum);
        }

        return mx2;
   }

}
```

```
package schemagenerator;

import java.util.HashSet;
import java.util.Set;

import weka.core.Matrix;

/**
 * A class to merge groups (connected components) into n groups
 * by using the feature strengths betwen their nodes.
 * @deprecated
 * @author Jamie
 *
 */
public class Merger
{
   public static Set<Set<Integer>> mergeGroups(Set<Set<Integer>> groups,
         Matrix featureStrengthMx, int n)
   {
         // We only want n groups, therefore we need
         // to merge groups in the current list until
         // there is a sufficiently small number.

         // To pick which pair of groups to merge, we
         // find the maximum inter-group bond strength,
         // by taking the average of all the element pairs
         // in the cross product.

         featureStrengthMx = MatrixUtils.normaliseToMax(featureStrengthMx);

         // First, we remove any groups that only have one element. i.e.
         // it has no connection with anything.
         Set<Set<Integer>> groupsCopy =
            (Set<Set<Integer>>) ((HashSet<Set<Integer>>) groups).clone();
         for(Set<Integer> group : groups)
            if(group.size()==1)groupsCopy.remove(group);
         groups = groupsCopy;

         if(featureStrengthMx.numRows()!=featureStrengthMx.numColumns())
            throw new Error("Must have same number of rows as columns.");

         System.out.println("Merging into "+n+" groups.");

         while(groups.size()>n)
         {
            System.out.println("Current group size: "+groups.size());
            Set<Integer> bestGroup1 = null;
            Set<Integer> bestGroup2 = null;
            double bestPairScore = 0.0;

            for(Set<Integer> group1 : groups)
                  for(Set<Integer> group2 : groups)
                  {  if(!group1.equals(group2))
                     {
                        int linkCount = 0;
                        double currentPairScore = 0.0;
                        for(Integer i : group1)
                        {  for(Integer j : group2)
                           {
                              currentPairScore+=
                                    featureStrengthMx.getElement(i, j);
                              linkCount++;
                           }
                        }

                        double avgScore = currentPairScore /
                           new Double(linkCount);
                        if(avgScore>bestPairScore)
                        {
                           bestPairScore = avgScore;
                           bestGroup1 = group1;
                           bestGroup2 = group2;
                        }
                     }
                  }
            }

            // System.out.println("Best pair score: "+bestPairScore);
            // System.out.println("Best group 1: "+bestGroup1);
            // System.out.println("Best group 2: "+bestGroup2);

            // Merge 2 groups:
            // 1. Add element of second to first.

            for(Integer i : bestGroup2)bestGroup1.add(i);
            // 2. Delete second from list.
            // System.out.println("Removing "+bestGroup2+" from "+groups);
            // boolean removed = groups.remove(bestGroup1);
            // System.out.println(removed);
            Set<Set<Integer>> newGroups = new HashSet<Set<Integer>>();
            for(Set<Integer> group : groups)
               if(!group.equals(bestGroup2))newGroups.add(group);
            groups = newGroups;

         }

         return groups;

   }
}
```

```java
package schemagenerator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * Provides utilities with regards to reading mark scheme/list
 * files.
 * @author Jamie
 *
 */
public class SchemaUtils {
    public static void print(Set<Set<Integer>> groups,
            List<String> featureNames)
    {
            for(Set<Integer> group : groups)
            {
                System.out.print(("{ "));
                // Print each group
                int c = 0;
                for(Integer i : group)
                {
                        if(c>0)System.out.print(", ");
                        System.out.print(featureNames.get(i));
                        c++;
                }
                System.out.println(" }");
            }
    }

    public static void print(List<Set<Integer>> groups,
            List<String> featureNames)
    {
            for(Set<Integer> group : groups)
            {
                System.out.print(("{ "));
                // Print each group
                int c = 0;
                for(Integer i : group)
                {
                        if(c>0)System.out.print(", ");
                        System.out.print(featureNames.get(i));
                        c++;
                }
                System.out.println(" }");
            }
    }
public static List<Set<Integer>> readMarkSchemeFile(String inputFile,
            List<String> attributesToCheck)
    {
            List<Set<Integer>> clusters = new LinkedList<Set<Integer>>();

            try
            {
                BufferedReader br = new BufferedReader(new FileReader(inputFile));
                int i=0;
                while(br.ready())
                {
                        String line = br.readLine();
                        if(i==0){
                            List<String> attributes = new LinkedList<String>();
                            // Reading attributes
                            line = line.substring(1,line.length()-1);
                            String[] lineParts = line.split(", ");
                            for(int j=0; j<lineParts.length; j++)
                                attributes.add(lineParts[j]);
                            if(!attributes.equals(attributesToCheck))
                                throw new Error("This mark scheme file is not " +
                                    "compatible with the training model. " +
                                    "Ensure that their attributes are the same.");

                        } else {
                            // Remove [ ]
                            line = line.substring(1,line.length()-1);
                            String[] lineParts = line.split(", ");
                            Set<Integer> cluster = new HashSet<Integer>();
                            for(int j=0; j<lineParts.length; j++)
                                cluster.add(new Integer(lineParts[j]));
                            clusters.add(cluster);
                        }

                        i++;
                }

                return clusters;
            } catch(Exception e){
                throw new Error("Problem reading mark scheme file");
            }

    }

    public static List<Set<Integer>> readMarkListFile(String inputFile,
            List<String> attributes, int numClusters)
            List<Set<Integer>> clusters = new LinkedList<Set<Integer>>();
            for(int i=0; i<numClusters; i++)clusters.add(new HashSet<Integer>());
            Map<String,Set<Integer>> groupMappings =
                    SchemaUtils.getGroupMappings(inputFile);

            // We have to combine 'attributes' and 'groupMappings' to obtain
            // 'clusters'. In the marklist file, groups numbers go from 1 to n,
            // whereas in lists we reference 0 to n-1, so we deduct 1
            // accordingly.
            for(int i=0; i<attributes.size(); i++)
            {
                Set<Integer> groups = groupMappings.get(attributes.get(i));
                for(Integer group : groups)clusters.get(group-1).add(i);
            }

            return clusters;
    }

    /**
     * Produces a map of feature names to a list of groups
     * each corresponds to. It takes a marklist file as
     * its input.
     */
    public static Map<String,Set<Integer>> getGroupMappings(String inputFile)
```

```
        Map<String,Set<Integer>> groupMappings =
                new TreeMap<String,Set<Integer>>();
        try
        {
            BufferedReader br = new BufferedReader(new FileReader(inputFile));
            while(br.ready())
            {
                String[] line = br.readLine().split("\t");
                String attribute = line[0];
                String[] groups =
                    line[1].substring(1,line[1].length()-1).split(", ");
                groupMappings.put(attribute, new TreeSet<Integer>());
                if(!line[1].equals("[]"))
                    for(int i = 0; i<groups.length; i++)
                        groupMappings.get(attribute).add(
                            new Integer(groups[i]));

            }
        } catch(Exception e){
            e.printStackTrace();
        }
        return groupMappings;
    }
}
```

```java
package schemagenerator;

import java.util.HashSet;
import java.util.Set;

/**
 * @deprecated
 * @author Jamie
 *
 */
public class SetUtils {
    public static Set<Set<Integer>> removeSubsets(Set<Set<Integer>> groups)
    {
            // Clean up: merge groups when one is a subset of the other.

            Set<Set<Integer>> groupsTemp = new HashSet<Set<Integer>>();
            for (Set<Integer> s : groups)groupsTemp.add(s); // Make copy of groups

            for (Set<Integer> s : groups) {
               for (Set<Integer> t : groups) {
                    if (isSubset(s, t)&&s!=t)
                       groupsTemp.remove(s);
               }
            }
            return groupsTemp;

    }

    public static boolean isSubset(Set<Integer> s, Set<Integer> t) {
            for (Integer i : s)
               if (!t.contains(i))
                    return false;
            return true;
    }

```

```java
package semantic;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

/**
 * This class is a representation of the semantic DRS
 * expression, in object oriented form.
 * Its main method linearises the DRS into a set of
 * semantic atoms.
 * @author Jamie
 *
 */
public class SemanticModel {
        // A list of variables in the DRS.
        Map<String,SemanticVar> svars = new HashMap<String,SemanticVar>();
        // A list of relations in the DRS.
        List<SemanticRel> srels = new LinkedList<SemanticRel>();
        // The LHS of equalities within the DRS.
        // The ith element corresponds to the ith element of eqRefR.
        List<String> eqRefL = new LinkedList<String>();
        List<String> eqRefR = new LinkedList<String>();
        // Whether each of the above are equalities or inequalities.
        List<Boolean> eqSense = new LinkedList<Boolean>();

        // The current sense while navigating the DRS tree.
        Boolean sense = true;
        // The current parent while navigating the DRS tree.
        // null if at the root.
        SemanticModel parent;

        // The set of generated semantic atoms from the model.
        Set<String> atoms = new TreeSet<String>();

        // Methods to add atoms to the growing set.
        public void addAtom(String atom){       atoms.add(atom);        }
        public void addAtoms(Set<String> atomsToAdd){   atoms.addAll(atomsToAd
d);     }

        // Gets a variable given its name.
        public SemanticVar getVar(String ref)
        {
                SemanticVar sv = svars.get(ref);
                // The var might be located in the parent model if not local.
                if(sv!=null) return sv;
                else return getParent().getVar(ref);
        }

        // Declares a variable.
        public void declareVar(String ref)
        {
                svars.put(ref, new SemanticVar(ref));
        }

        // Adds a predicate, where ref is the name of the variable,
        // symbol is the unary relation being applied to it, and
        // posType is the type of relation.
        public void addPred(String ref, String symbol,String posType)
        {
                if(svars.containsKey(ref))
                {
                        svars.get(ref).addPred(symbol,posType);
                } else if(parent!=null){
                        if(parent.getVar(ref)!=null)parent.getVar(ref).addPred
(symbol,posType);
                        else declareVar(ref);
                }
                else declareVar(ref);
        }

        // Adds a binary relation, of the form "symbol(ref1,ref2)".
        public void addRel(String ref1, String ref2, String symbol)
        {
                srels.add(new SemanticRel(ref1,ref2,symbol));
        }

        // Adds an equality, of the form ref1=ref2 or ref1!=ref2.
        public void addEq(String ref1, String ref2,Boolean sense){
                eqRefL.add(ref1);
                eqRefR.add(ref2);
                eqSense.add(sense);
        }

        // Sets the cardinality of a variable.
        public void setCard(String ref, int card, String cardType)
        {       svars.get(ref).setCard(card);   }

        public void changeSense(){      sense = !sense; }
        public boolean getSense(){      return sense;   }

        public void setName(String ref, String symbol, String neType)
        {
                svars.get(ref).setName(symbol);
        }

        // Used for debugging purposes.
        // Prints the variables and relations from
        // the tree.
        public void print()
        {
                for(SemanticVar svar : svars.values())
                {
                        svar.print();
                }

                System.out.println("*** Relations ****");
                for(SemanticRel srel : srels)
                        System.out.println(srel.toString());
        }

        // Using the information accumulated from the tree,
        // a list of semantic atoms is generated.
        public Set<String> getAtoms()
        {
                for(SemanticVar svar : svars.values())
                {
                        svar.getAtoms(this);
                }
                for(SemanticRel srel : srels)
                {
```

```
                                srel.getAtoms(this);
                }

                   // Deal with equalities.
                for(int i=0; i<eqRefL.size(); i++)
                {
                        SemanticVar l = getVar(eqRefL.get(i));
                        SemanticVar r = getVar(eqRefR.get(i));
                        String senseStringStart = "";
                        String senseStringEnd = "";
                        if(!eqSense.get(i))
                        {
                                senseStringStart = "![";
                                senseStringEnd = "]";
                        }
                        for(String atomL : l.getAtoms(this))
                        {       for(String atomR : r.getAtoms(this))
                                {       // - is commutative.
                                        atoms.add(senseStringStart + atomL+" =
"+atomR + senseStringEnd);
                                        atoms.add(senseStringStart + atomR+" =
"+atomL + senseStringEnd);
                                }
                        }
                        i++;
                }

                return atoms;
        }

        public SemanticModel getParent() {
                return parent;
        }

        public void setParent(SemanticModel parent) {
                this.parent = parent;
        }
}
```

semantic/SemanticRel.java

```
package semantic;

import java.util.HashSet;
import java.util.Set;

/**
 * Represents a relation in the semantic model.
 * @author Jamie
 *
 */
public class SemanticRel {

        // Represents:
        // name:type(ref1,ref2)
        String ref1;
        String ref2;
        String name;
        String type = "";

        // Relations which we ignore.
        String[] rejectableKeywords = { "topic" };

        // We may need to produce the list of atoms
        // numerous times, so they are cached for
        // subsequent use.
        Set<String> cachedAtoms;

        public SemanticRel(String ref1, String ref2, String symbol)
        {
                this.ref1 = ref1;
                this.ref2 = ref2;
                this.name = symbol;
        }

        public String toString()
        {
                return name+"["+ref1+","+ref2+"]";
        }

        // The rule for getting atoms is as such:
        // * We get the sets of atoms A and B for the
        //   left and right variables in the relations.
        // * We take the set:
        //    { name[a,b] | a<-A, b<-B }
        public Set<String> getAtoms(SemanticModel model)
        {
                if(cachedAtoms!=null)return cachedAtoms;
                Set<String> set = new HashSet<String>();

                String senseString = "";
                if(!model.getSense())senseString = "!";

                if(!shouldReject())
                {
                        Set<String> setL = model.getVar(ref1).getAtoms(model);
                        Set<String> setR = model.getVar(ref2).getAtoms(model);

                        for(String atomL : setL)
                                for(String atomR : setR)
                                        set.add(senseString+name+"["+atomL+","
+atomR+"]");

                        model.addAtoms(set);


                cachedAtoms = set;
                return set;
        }

        private boolean shouldReject()
        {
                for(int i = 0; i<rejectableKeywords.length; i++)
                        if(rejectableKeywords[i].equals(name))return true;

                return false;
        }

}
```

```java
package semantic;

import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

public class SemanticVar {
	// Of the form:
	// {attr1 ^ ... ^ attrN}[name]
	String ref;
	String name;
	List<String> attrs = new LinkedList<String>();

	// Generic variable names which are ignored.
	String[] rejectableKeywords = { "event", "proposition" };

	String type = "";
	int card = 1; // need to change to lower and upper bound

	SemanticModel prop;
	public void setProp(SemanticModel model){	prop = model;	}

	// We may need to produce the list of atoms
	// numerous times, so they are cached for
	// subsequent use.
	Set<String> cachedAtoms;

	public SemanticVar(String ref){ this.ref = ref; }

	// We the DRS contains propositions such as
	// John(x1) and large(x1), we need to distinguish
	// between the name of the variable, and attributes
	// relating to it. We can determine this by the
	// posType.
	public void addPred(String symbol,String posType)
	{
		if(!shouldReject(symbol))
		{
			if(posType.equals("n")||posType.equals("v"))
			{
				name = symbol;
			} else {
				attrs.add(symbol);
			}
		} else {
			type = symbol;
		}
	}

	// Sets the cardinality of the variable.
	public void setCard(int card){	this.card = card;	}

	// For debug purposes.
	public void print()
	{
		System.out.println("**** "+ref+" *****");
		System.out.println(name);
		for(String attr : attrs)
			System.out.println(attr+"["+name+"]");
	}


	// Gets a list of atoms represented by
	// the variable.
	public Set<String> getAtoms(SemanticModel model)
	{
		if(cachedAtoms!=null) return cachedAtoms;

		String senseString = "";
		if(!model.getSense()) senseString = "!";

		Set<String> set = new HashSet<String>();
		if(shouldReject(name))return set;

		if(name==null&&prop==null)
		{
			return set;
		}

		if(prop==null)
		{
			// Get variable name itself
			set.add(senseString+name);
			// Get cardinality if present
			if(card!=1)set.add(senseString+"|"+name+"|="+card);
			// Get var with attributes
			for(String attr : attrs)
			{
				set.add(senseString+attr+"["+name+"]"); // add attr with subject
				set.add(senseString+attr);	// add attr
			}
		} else {
			// We have a proposition
			Set<String> childAtoms = prop.getAtoms();
			// Add original child atoms to this level
			set.addAll(childAtoms);
			// Add attributed child atoms if any attributes
			for(String attr : attrs)
				for(String atom : childAtoms)
					if(!shouldReject(attr))
					{
						set.add(senseString+attr+"["+atom+"]");
						set.add(senseString+attr);
					}
		}

		model.addAtoms(set);
		cachedAtoms = set;
		return set;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getType() {
```

```
                return type;
        }

        public void setType(String type) {
                this.type = type;
        }

        public int getCard() {
                return card;
        }


        private boolean shouldReject(String symbol)
        {
                for(int i=0; i<rejectableKeywords.length; i++)
                        if(rejectableKeywords[i].equals(symbol))return true;

                return false;
        }

}
```

spelling/MapCorrections.java

```java
package spelling;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

/**
 * Takes a CSV containing the English sentences and
 * "spell correction mapping file" (where each line
 * contains a spelling correction), and produces a
 * new CSV with the corrections made.
 * @author Jamie
 *
 */
public class MapCorrections {
        final static String inputFile = "./files/13bii-answers.csv";
        final static String outputFile = "./files/13bii-answers2.csv";
        final static String mappingFile = "./files/13bii-spelling-corrections.txt";
        Map<String,String> corrections = new HashMap<String,String>();

        public static void main(String[] args) {

                MapCorrections mc = new MapCorrections();
                if(args.length==0)mc.map(mappingFile,inputFile,outputFile);
                else mc.map(args[0],args[1],args[2]);
        }

        public void map(String mappingFile, String inputFile, String outputFile)
        {
                try
                {
                        // Read in spelling corrections
                        BufferedReader br = new BufferedReader(new FileReader(mappingFile));
                        while(br.ready())
                        {
                                String[] parts = br.readLine().split("\t");
                                if(parts.length!=2)System.out.println("Problem with line "+parts[0]);
                                corrections.put(parts[0], parts[1]);
                        }
                        br.close();

                        // Read in input file, and for each line, make the corrections
                        // and write to output file.
                        br = new BufferedReader(new FileReader(inputFile));
                        PrintWriter pw = new PrintWriter(outputFile);

                        while(br.ready())pw.println(transformLine(br.readLine()));


                        br.close();
                        pw.close();

                } catch(Exception e){
                        e.printStackTrace();

        }

        public String transformLine(String line)
        {
                for(String key : corrections.keySet())
                {       if(line.toLowerCase().indexOf(key)>=0)
                        {
                                int pos = line.toLowerCase().indexOf(key);

                                // Need to check it's not part of some other word
                                if(pos>0&&pos<line.length()-key.length())
                                {
                                        if(!Character.isLetter(line.charAt(pos-1))&&!Character.isLetter(line.charAt(pos+key.length())))
                                        {       // Not part of a word, so replace!
                                                line = line.substring(0,pos)+corrections.get(key)+line.substring(pos+key.length());
                                        }
                                } else if(pos==0){
                                        if(!Character.isLetter(line.charAt(pos+key.length())))
                                        {
                                                line = corrections.get(key)+line.substring(pos+key.length());
                                        }
                                } else {
                                        if(!Character.isLetter(line.charAt(pos-1)))
                                        {
                                                // Not part of a word, so replace!
                                                line = line.substring(0,pos)+corrections.get(key);
                                        }
                                }
                        }
                }

                return line;
        }
}
```

```java
package spelling;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Map;
import java.util.TreeMap;

import com.swabunga.spell.engine.Word;
import com.swabunga.spell.event.StringWordTokenizer;

/**
 * Takes a dictionary as well as the original English
 * answers, and produces a "spell correction mapping
 * file" which indicates what words should be replaced
 * and what to replace them with.
 * Outputting this correction file rather than modifying
 * the CSV immediately allows manual alterations to be
 * made.
 * @author Jamie
 *
 */
public class SpellCorrectionGenerator
{
        // The dictionary.
        final static String dicFile = "C:/Users/Jamie/Desktop/Programming/exammarking/dictionary/eng_com.dic";
        // The CSV containing potentially mispelled English answers.
        final static String fileToCheck = "./files/13bii-answers.csv";
        // The file to output.
        final static String mappingFile = "./files/13bii-spelling-corrections.txt";


        public static void main(String[] args) throws Exception {

                SpellCorrectionGenerator scg = new SpellCorrectionGenerator();
                if(args.length[illegible])scg.generate(dicFile,fileToCheck,mappingFile);

                else scg.generate(dicFile,args[0],args[1]);
        }

        public void generate(String dicFile, String fileToCheck, String mappingFile) throws Exception
        {
                // A utility class to check spellings.
                SpellUtils spellUtils = new SpellUtils(dicFile);
                BufferedReader br = new BufferedReader(new FileReader(fileToCheck));

                // We keep a count of each of the words in the data-set.
                // This allows us to detect correctly spelled words which
                // are not in the dictionary.
                Map<String, Integer> wordCounts = compareWords(fileToCheck);

                // Remove common words
                wordCounts.remove("the");
                wordCounts.remove("has");

                // Want to use wordCounts to produce a map for misspelled words.
                Map<String, String> spellingCorrections = new TreeMap<String, String>();

                for (String val : wordCounts.keySet

                        // Correct plurality errors such as "babys -> babies".
                        boolean correctedPlurality = spellUtils.correctPluralityErrors(
                                        spellingCorrections, val);

                        if (!correctedPlurality) {
                                // Is the word in the dictionary?
                                boolean correct = spellUtils.isCorrect(val);
                                // Is the word before the 's (if applicable) correct?
                                boolean correctA = spellUtils.isApostropheAndCorrect(val);

                                if (correct || correctA || val.length()<=[illegible]
                                        // Leave short or correctly spelled words alone.

                                        System.out.println("** " + val + " -> LEAVE");

                                } else {
                                        // We have a word not in the dictionary.


                                        // If it's commonly occurring in the data-set,

                                        // leave it alone.
                                        if (wordCounts.get(val) > 5) {
                                                // Do nothing.
                                        } else {

                                                // SpellUtils.split(val) is no longer used,

                                                // and simply returns null.
                                                String splitWord = spellUtils.split(val);

                                                if(splitWord!=null)

                                                        spellingCorrections.put(val, splitWord);

                                                } else {
                                                        double bestScore = Double.NEGATIVE_INFINITY;

                                                        String bestWord = "";

                                                        // Find the word in the data-set which

                                                        // has the strongest relationship with

                                                        // the misspelled word.

                                                        for (String val2 : wordCounts.keySet()) {

                                                                int editDistance = StringUtils
                                                                .getLevenshteinDistance(val, val2);

                                                                int letterDiff = StringUtils
                                                                .getLetterSetDifference(val, val2);

                                                                int [illegible]rBonus = StringUtils
                        .getFirstLetterBonus(val, val2);
                int sizeDiff = StringUtils.getSizeDifference(val,
                        val2);

                double score = (Math.log(wordCounts.get(val2)) / Math
                        .log(2))
                        - (3 * firstLetterBonus)
                        - 2
                        * editDistance
                        - letterDiff - sizeDiff;
                if (score > bestScore
                        && !val2.equals(val)
                        && editDistance <= 3 // We limit edit
                                             // distance
                        && wordCounts.get(val2) > wordCounts
                                .get(val) // Can only map to a
                                          // more popular word.
                                          // Avoids cycles.
                ) {
                    bestScore = score;
                    bestWord = val2;
                }
            }

            // Additionally find a word in the dictionary
            // which has the closest relationship with the
            // word. This may override a selection from
            // the data-set.
            for (Object val3 : spellUtils.getSuggestions(val)) {

                String val2 = ((Word) val3).toString();
                int editDistance = StringUtils
                        .getLevenshteinDistance(val, val2);
                int letterDiff = StringUtils
                        .getLetterSetDifference(val, val2);
                int firstLetterBonus = StringUtils
                        .getFirstLetterBonus(val, val2);
                int sizeDiff = StringUtils.getSizeDifference(val,
                        val2);

                double score = 1 - (2 * editDistance)
                        - (3 * firstLetterBonus) - letterDiff
                        - sizeDiff;
                if (score > bestScore) {
                    bestScore = score;
                    bestWord = val2;
                }
            }

            spellingCorrections.put(val, bestWord);
                    }
                }
            }
        }
    }

    // Adds common spelling mistakes.
    spellUtils.addCommonMistakes(spellingCorrections);

    // Write correction mapping to a file.
    PrintWriter pw = new PrintWriter(mappingFile);

    for (String val : spellingCorrections.keySet()) {
        System.out.print("** " + val);
        String lastVal = null;

        // A word may map to a word which maps to another.
        String temp = spellingCorrections.get(val);
        while (temp != null) {
            lastVal = temp;
            System.out.print(" -> " + temp);
            // Need to ignore identity mappings
            if(!temp.equals(spellingCorrections.get(temp))) temp = spellingCorrections.get(temp);
            else temp = null;
        }

        if(lastVal!=null)pw.println(val+"\t"+lastVal);
        System.out.println();
    }
    pw.close();


  /**
   * [illegible] for all the words in the data set.
```

```
         */
        public static Map<String, Integer> collateWords(String fileName) {
                Map<String, Integer> wordCounts = new TreeMap<String, Integer>
();

                try {
                        BufferedReader br = new BufferedReader(new FileReader(
fileToCheck));
                        while (br.ready()) {

                                String line = br.readLine();
                                if(line.indexOf("\t")>0)
                                {
                                        String sentence = line.split("\t")[1];
                                        StringWordTokenizer swt = new StringWo
rdTokenizer(sentence);
                                        while (swt.hasMoreWords()) {
                                                String word = swt.nextWord().t
oLowerCase();

                                                if (wordCounts.get(word) == nu
ll)
                                                        wordCounts.put(word, 0
);
                                                wordCounts.put(word, wordCount
s.get(word) + 1);

                                        }
                                }
                        }

                } catch (Exception e) {
                        e.printStackTrace();
                }

                System.out.println("*** Words ***");
                for(String s : wordCounts.keySet())
                        System.out.println(s+" ("+wordCounts.get(s)+")");
                System.out.println("*************");

                return wordCounts;
        }

}
```

```java
package spelling;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Map;

import com.swabunga.spell.engine.SpellDictionary;
import com.swabunga.spell.engine.SpellDictionaryHashMap;
import com.swabunga.spell.event.SpellCheckEvent;
import com.swabunga.spell.event.SpellCheckListener;
import com.swabunga.spell.event.SpellChecker;
import com.swabunga.spell.event.StringWordTokenizer;

/**
 * A wrapper to make the methods of a dictionary
 * library used more accessible. The necessity is
 * mainly because the library has no direct method
 * of determining a word's correctness, and alerts
 * a SpellCheckListener when a misspelling is
 * detected.
 * @author Jamie
 *
 */
public class SpellUtils {

        boolean flag;
        List suggestions;
        SpellDictionary dictionary;
        SpellChecker spellChecker;

        public SpellUtils(String fileName) throws IOException
        {
                this(new SpellDictionaryHashMap(new File(fileName)));
        }

        public SpellUtils(SpellDictionary dictionary)
        {
                this.dictionary = dictionary;
                spellChecker = new SpellChecker(dictionary);
            spellChecker.addSpellCheckListener(new SuggestionListener());
        }

        public class SuggestionListener implements SpellCheckListener {

                public void spellingError(SpellCheckEvent event) {
                        flag = false;
                        suggestions = event.getSuggestions();
                }

        }

        public boolean isCorrect(String word)
        {
                flag=true;
                spellChecker.checkSpelling(new StringWordTokenizer(word));
                return flag;
        }

        public List getSuggestions(String word)
        {
                spellChecker.checkSpelling(new StringWordTokenizer(word));
                return suggestions;
        }

        /**
         * Makes corrections with regards to incorrect suffixes
         * for plurals. For example "babys" and "halfs".
         */
        public boolean correctPluralityErrors(Map<String,String> spellCorrections, String val)
        {
                String transformedPlural = "";
                if(val.endsWith("ys"))transformedPlural = val.substring(0,val.length()-2) + "ies";
                if(val.endsWith("lfs"))transformedPlural = val.substring(0,val.length()-3) + "lves";

                if(!transformedPlural.equals(""))
                {
                        if(isCorrect(transformedPlural))
                        {
                                spellCorrections.put(val, transformedPlural);
                                return true;
                        } else {
                                return false;
                        }
                } else {
                        return false;
                }
        }

        public void addCommonMistakes(Map<String,String> spellCorrections)
        {
                spellCorrections.put("aswell","as well");
                spellCorrections.put("dont", "don’t");
                spellCorrections.put("everytime","every time");
                spellCorrections.put("colour","colour");        // colour not in dictionary!
                spellCorrections.put("color", "colour");
                spellCorrections.put("childs", "children");
                spellCorrections.put("i", "I");
        }

        public boolean isApostrophedWordCorrect(String word)
        {
                // Note, if isCorrect for the whole word returns true,
                // then the output of this method does not matter.

                int aIndex = word.indexOf("’");
                // If it doesn’t contain apostrophe, test not applicable.
                if(aIndex==-1) return false;

                if(word.endsWith("’s"))
                {
                        String subword = word.substring(0,aIndex);
                        if(isCorrect(subword)) return true;
                        else return false;
                } else {
                        return false;
                }
        }

        /**
         * @deprecated
         * Used to detect if two words had been merged
         * together. split("aswell") would return "as well"
         * and split "alot") return "a lot". It was found
         * the method led to poor results as it split words
         * often when it shouldn't have.
         *
        public String split(String word)
        {
//              for(int i=2; i<(word.length()-1); i++)
//              {
//                      String wordL = word.substring(0,i);
//                      String wordR   word.substring(i,word.length());
//                      if(isCorrect(wordL)&&isCorrect(wordR))return wordL+" "+wordR;
//              }
                return null;
        }

}
```

```java
package spelling;
import java.util.HashSet;
import java.util.Set;

/**
 * Provides comparison methods for strings.
 * @author Jamie
 *
 */
public class StringUtils {

        public static int getSizeDifference(String s, String t)
        {
                return Math.abs(s.length() - t.length());
        }

        /**
         * Treats the words as a 'bag of letters', and
         * for 2 sets of letters A and B, computes:
         * #((A^¬B)v(B^¬A))
         */
        public static int getLetterSetDifference(String s, String t)
        {
                Set<Character> sLetters = new HashSet<Character>();
                Set<Character> tLetters = new HashSet<Character>();
                for(int i=0; i<s.length(); i++)sLetters.add(s.charAt(i));
                for(int i=0; i<t.length(); i++)tLetters.add(t.charAt(i));

                int diff = 0;
                for(Character c : sLetters)if(!tLetters.contains(c))diff++;
                for(Character c : tLetters)if(!sLetters.contains(c))diff++;

                return diff;
        }

        /**
         * As a heuristic, we class two words as particularly
         * similar if their first letter is the same.
         * Therefore "receive" and "recieve" would be more
         * similar than "believe" and "relieve".
         */
        public static int getFirstLetterBonus(String s, String t)
        {
                if(s.charAt(0)==t.charAt(0))return 1;
                else return 0;
        }

        /**
         * String edit distance.
         * Unknown author.
         */
        public static int getLevenshteinDistance (String s, String t) {
                    if (s == null || t == null) {
                      throw new IllegalArgumentException("Strings must not be null");
                    }

                    int n = s.length(); // length of s
                    int m = t.length(); // length of t

                    if (n == 0) {
                      return m;
                    } else if (m == 0) {
                      return n;
                    }

                    int p[] = new int[n+1]; //'previous' cost array, horizontally
                    int d[] = new int[n+1]; // cost array, horizontally
                    int _d[]; //placeholder to assist in swapping p and d

                    // indexes into strings s and t
                    int i; // iterates through s
                    int j; // iterates through t

                    char t_j; // jth character of t

                    int cost; // cost

                    for (i = 0; i<=n; i++) {
                      p[i] = i;
                    }

                    for (j = 1; j<=m; j++) {
                      t_j = t.charAt(j-1);
                      d[0] = j;

                      for (i=1; i<=n; i++) {
                         cost = s.charAt(i-1)==t_j ? 0 : 1;
                         // minimum of cell to the left+1, to the top+1, diagonally left and up +cost
                         d[i] = Math.min(Math.min(d[i-1]+1, p[i]+1), p[i-1]+cost);
                      }

                      // copy current distance counts to 'previous row' distance counts
                      _d = p;
                      p = d;
                      d = _d;
                    }

                    // our last action in the above loop was to switch d and p, so p now
                    // actually has the most recent cost counts
                    return p[n];
                  }
}
```

server/ClientCAndC.java

```
package server;
import java.io.*;
import java.net.*;
import java.util.Set;
import java.util.TreeSet;

import parser.DRSreader;

public class ClientCAndC {
        final static int port = 4915; // port to which to connect
        final static String computer = "curlew.comlab.ox.ac.uk";
        final static String endString = ".";

        PrintWriter toServer;
        BufferedReader fromServer;

        public static void main(String[] args) throws IOException {

                BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

                // Initialize socket
                System.out.println("** Attempting to connect. **");
                Socket t = new Socket(computer, port);
                System.out.println("** Client connected.");
                System.out.println("** Please input a sentence to parse and box.");

                // Associate writer with socket
                PrintWriter pw = new PrintWriter(t.getOutputStream(), true);

                // Also associate reader with socket for receiving back response.
                BufferedReader br = new BufferedReader(new InputStreamReader(t
                                .getInputStream()));

                // Repeatedly input string from terminal, and write to socket, until
                // endString is input
                String st;
                System.out.print("> ");
                st = in.readLine();
                while (!st.equals(endString)) {
                        pw.println(st);

                        String response = br.readLine();
                        // System.out.println(response);

                        Set<String> set = DRSreader.atomize(response);
                        for(String s : set)System.out.println(s);

                        System.out.print("> ");
                        st = in.readLine();
                }
                ;

                System.out.println("Connection closed");
        }

        public void connect() throws IOException
        {
                // Initialize socket
                Socket t = new Socket(computer, port);

                // Associate writer with socket
                toServer = new PrintWriter(t.getOutputStream(), true);

                // Also associate reader with socket for receiving back response.
                fromServer = new BufferedReader(new InputStreamReader(t
                                .getInputStream()));

                System.out.println("*-- Connected to server");
        }

        public String getDRS(String sentence) throws IOException
        {
                toServer.println(sentence);
                String response = fromServer.readLine();
                System.out.println("*-- Got response from client.");
                return response;
        }

        public void close() throws IOException
        {
                // Giving the sentence '.' tells the server thread to shut down.
                toServer.println(".");
                // But we still need to close the connection.
                toServer.close();
                fromServer.close();

                System.out.println("*-- Closed connection to server");
        }

}
```

```java
package server;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Listens to client requests to parse English sentences,
 * and sends back the parsed expression in DRS form.
 * Can operate over a remote connection.
 * @author Jamie
 *
 */
public class ServerCAndC {

        final static int port = 49157; // port to which to listen
        final static String INPUT_FILE = "input.txt";
        final static String INTERMEDIATE_FILE = "output.txt";
        final static String OUTPUT_FILE = "outputBOXED.txt";

        // Starts the soap server, which receives parse commands.
        final static String soapServerCommand = "../candc/candc-1.00/bin/soap_server --server localhost:9000 --candc ../candc/candc-1.00/models/boxer";
        // Starts the soap client, which sends parse commands.
        final static String soapClientCommand = "../candc/candc-1.00/bin/soap_client --input "+INPUT_FILE+" --output "+INTERMEDIATE_FILE+" --url http://localhost:9000";
        // A command to 'box' the CCG to produce a DRS.
        final static String boxerCommand = "../candc/candc-1.00/bin/boxer --input "+INTERMEDIATE_FILE+" --output "+OUTPUT_FILE+" --box true --flat";

        public static void main(String[] args) {
                Runtime r = Runtime.getRuntime();
                try
                {

                        ServerCAndC cle = new ServerCAndC();
                        execute(soapServerCommand,false,20000); // Start the SOAP server

                        startServerListener();

                } catch(Exception e){
                        e.printStackTrace();
                }
        }

        public static void execute(String command,boolean wait,long delay)
        {
                try
                {
                        // Execute the command via command line prompt.
                        Process p = Runtime.getRuntime().exec(command);
                        // Some commands we can detect when they're finished.
                        // For starting the soap server, we don't know when it's
                        // ready, so instead we wait a suitable amount of time.
                        if(wait)
                        {
                                p.waitFor();
                        } else {
                                System.out.println("Sleeping for "+delay+"ms to allow server to start.");
                                Thread.sleep(delay);
                        }
                } catch(Exception e){
                        e.printStackTrace();
                }
        }

        /**
         * Reads the file produced from boxing, and extracts
         * the DRS string from it.
         */
        public static String extractDRS()
        {
                try
                {
                        BufferedReader br = new BufferedReader(new FileReader("./outputBOXED.txt"));
                        int i=0;
                        String line = "";
                        while(i<=13&&br.ready())
                        {
                                line = br.readLine();
                                i++;
                        }

                        if(i==14)
                        {
                                int lastBracketPos = line.lastIndexOf(')');
                                line = line.substring(0,lastBracketPos).trim();
                                return line;
                        } else {
                                // Case where the C&C parser failed
                                return "";
                        }

                } catch(Exception e){
                        throw new Error(e);
                }
        }

        /**
         * Listens to incoming connection requests.
         */
        public static void startServerListener()
        {
                try
                {
                        System.out.println("*** Listening for incoming connections. ***");

                        // Initialise server socket
                         ServerSocket listener = new ServerSocket(port);

                        // We want the program to always accept connections, so we loop continuously.
                        while(true){
```

```
                        // Wait for, and then accept, connections
                        Socket s = listener.accept();
                                        // Create a new thread using this socket.
                        ServerThread st = new ServerThread(s);
                        st.start();        // Start the thread.

                        }

                } catch(Exception e){
                        e.printStackTrace();
                }

        }

        /**
         * The CAndC Parser reads the input string from a file.
         * The method puts this string in a file to produce
         * the suitable input for the parser.
         */
        public static void writeInputToFile(String str)
        {
                try
                {
                        PrintWriter pw = new PrintWriter(INPUT_FILE);
                        pw.println(str);
                        pw.close();
                } catch(Exception e){
                        e.printStackTrace();
                }

        }
}
```

```java
package server;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * A ServerThread in instantiated by ServerCAndC, and
 * represents a service for a particular client.
 * Several ServerThreads may be running if multiple
 * clients are being concurrently served
 * @author Jamie
 *
 */
public class ServerThread extends Thread {

        Socket s;

        public ServerThread(Socket s){  this.s = s;     }

        public void run()
        {
            try
            {
                System.out.println("*** Accepted incoming connection. ***");


                // Associate reader with socket
                    BufferedReader br =
                              new BufferedReader(new InputStreamReader(s.getInputStream()));

                    PrintWriter pw = new PrintWriter(s.getOutputStream(),true);


                        // Read just 1 line
                        String st = br.readLine();

                        // Now read until we terminate.
                        while(!st.equals("."))
                        {
                                System.out.println("Parsing: "+st);

                                ServerCAndC.writeInputToFile(st);
                                ServerCAndC.execute(ServerCAndC.soapClientCommand,true,0);     // parse the input using CandC
                                ServerCAndC.execute(ServerCAndC.boxerCommand,true,0);          // box the output CCG
                                String drs = ServerCAndC.extractDRS();

                                System.out.println("DRS: "+drs);

                                // Prints the DRS string to the socket.
                                pw.println(drs);

                                System.out.println("*** Request successfully processed. ***");


                                st = br.readLine();
                                if(st==null)break;
                        }


                        br.close();
                        pw.close();

                        System.out.println("*** Connection terminated ***");

            } catch(Exception e){
                e.printStackTrace();


        }
}
```

pipeline/AnswerConverter.java

```java
package pipeline;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;

/**
 * Converts the pl file into a neater CSV.
 * Rows of the CV have the form:
 * "mark \t answer"
 * @author Jamie
 *
 */
public class AnswerConverter {

    static final String inputFile = "./files/answers/2/4(d).pl";
    static final String outputFile = "./files/answers/2/4d-answers.csv";

    public static void main(String[] args) {
        try
        {
            BufferedReader br = new BufferedReader(new FileReader(inputFile));
            PrintWriter pw = new PrintWriter(outputFile);

            while(br.ready())
            {
                String line = br.readLine();

                if(line.indexOf("%")==0||line.indexOf("noanswer")> 0||line.indexOf("answer")!=0)
                {
                        // Reject
                } else {
                        int openBrac = line.indexOf("(");
                        int nextComma = line.indexOf(",",openBrac);
                        char score = getScore(line,nextComma+1);
                        int openQuote = line.indexOf("\"",nextComma);
                        int closeQuote = line.indexOf("\"",openQuote+1);
                        String answer;
                        if(closeQuote>=0)answer = line.substring(openQuote+1,closeQuote);
                        else answer = line.substring(openQuote+1);

                        answer = answer.replaceAll("\t","");
                        answer = answer.replaceAll(","," ,");
                                // make sure there's space before comma
                        answer = answer.replaceAll("\\."," \\.");
                                // make sure there's space before full stop
                        answer = answer.replaceAll("  "," ");
                                // remove duplicate spaces
                        answer = answer.trim();

                        System.out.println(score+"\t"+answer);
                        pw.println(score+"\t"+answer);
                }
            }

            br.close();
            pw.close();

          } catch(Exception e) {
            e.printStackTrace();
        }
    }



    private static char getScore(String line, int startPos)
    {
        if(Character.isDigit(new Character(line.charAt(startPos))))
                return line.charAt(startPos);
        else return getScore(line,startPos+1);
    }
}
```

```java
package pipeline;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

import parser.DRSreader;


/**
 * Generates an arff file.
 * The input file is a CSV with tab separated fields "score     DRS string".
 * It extracts all the semantic atoms from each DRS, collates them,
 * and then produces a binary table with the atoms as the columns.
 * @author Jamie
 *
 */
public class ArffGenerator {

   static String inputFile = "./files/8b-answers-processed.csv";
   static String outputFile = "./files/8b.arff";

   // If this is set to true, atoms which occur only 1 time
   // in the corpus are removed.
   final static boolean CONDENSE = true;
   public static void main(String[] args)
   {
        System.out.println("Usage: ArffGenerator INPUTFILE.csv "+
                "OUTPUTFILE.arff");
        generateFile(args[0],args[1],1,0,false,System.out);
   }

   public static void generateFile(String inputFile,String outputFile,
        int numsegments, int segnumber, boolean exclude,
        PrintStream ps)
   {

        ps.println("EXECUTING ARFF GENERATOR");
        ps.println("Input file: "+inputFile);
        ps.println("Output file: "+outputFile);

        List<Integer> scores = new LinkedList<Integer>();
        List<Set<String>> atomSets = new LinkedList<Set<String>>();
        Map<String,Integer> allAtoms = new HashMap<String,Integer>();
        Set<String> finalAtomList = new TreeSet<String>();

        try
        {
           BufferedReader br = new BufferedReader(
                new FileReader(inputFile));

           int maxScore = 0;
           int numLines=1;

           while(br.ready())
           {
                String line = br.readLine();
                if(line.indexOf("%")!=0)
                {
                        String[] lineParts = line.split("\t");

                        int theScore = new Integer(lineParts[0]);
                        if(theScore>maxScore)maxScore = theScore;
                        scores.add(theScore);
                        Set<String> set = DRSreader.atomize(lineParts[1]);
                        atomSets.add(set);

                        ps.println("Successfully parsed line "+numLines+".");
                }
                numLines++;
           }

           br.close();

           ps.println("Num lines: "+numLines);

           // Work out segments
           int startPos = 0;
           int segmentLength = 0;
           int k=0;

           while(k<=segnumber)
           {
                startPos += segmentLength;
                segmentLength = numLines / numsegments; // takes divisor;
                numsegments--;
                numLines -= segmentLength;
                k++;
           }

           if(!exclude){
                startPos = -1;
                segmentLength = 0;
           }

           // Want a set with all the atoms (but we want
           // to ignore the testing segment)
           for(int i=0; i<atomSets.size(); i++)
           {
                if(!(i>=startPos&&i<startPos+segmentLength))
                {
                   Set<String> atoms = atomSets.get(i);
                   for(String atom : atoms)
                   {
                        if(allAtoms.get(atom)==null)allAtoms.put(atom, 1);
                        else allAtoms.put(atom, allAtoms.get(atom)+1);
                   }
                }
           }

           // If an atom has a count of 1, we only include it if
           // we're not condensing the list.
           for(String atom : allAtoms.keySet())
                if(allAtoms.get(atom)>1)finalAtomList.add(atom);
                        else if(!CONDENSE)finalAtomList.add(atom);


           PrintWriter pw = new PrintWriter(outputFile);
```

```

            if [illegible]
            {
                pw.println("% This is part of a test suite.");
                pw.println("% Took all lines except "+startPos+" to "+
                        (startPos+segmentLength-1)+".");
                pw.println();
            }

            pw.println("@relation myexample");

            for(String atom : finalAtomList)
                pw.println("@attribute \""+atom+"\" { 0, 1 }");

            String scoreString = "@attribute score { 0";
            for(int i=1; i<=maxScore; i++)scoreString+= ", "+i;
            scoreString+= " }";
            pw.println(scoreString);

            pw.println();
            pw.println("@data");

            int i = 0;
            for(Set<String> atomSet : atomSets)
            {
                if(!(i>=startPos&&i<startPos+segmentLength))
                {
                    for(String atom : finalAtomList)
                    {
                        // Check to see if the atomSet has this atom.
                        if(atomSet.contains(atom))pw.print("1");
                        else pw.print("0");
                        pw.print(", ");
                    }

                    pw.println(scores.get(i));
                }
                i++;
            }

            pw.close();
            ps.println("Successfully generated file "+outputFile);

        } catch(Exception e){
                e.printStackTrace();
                throw new Error("Stopped");
        }
    }


```

```java
package pipeline;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintStream;
import java.io.PrintWriter;

import server.ClientCAndC;

/**
 * Takes a CSV file of English answers and converts
 * them into DRS semantic form using an external
 * parser.
 * @author Jamie
 *
 */
public class FileParser {

    static String inputFile = "./files/8b-answers.csv";
    static String outputFile = "./files/8b-answers-processed.csv";

    public static void main(String[] args) {
         processSentences(args[0], args[1], System.out);
    }

    /**
     * Processes a single sentence.
     */
    public static String processSentence(String sentence)
    {
         ClientCAndC client = new ClientCAndC();
         try
         {
            client.connect();
            String drsString = client.getDRS(sentence);
            return drsString;

         } catch(Exception e){
            throw new Error("Failed to connect to server.");
         }
    }

    public static void processSentences(String inputFile,
         String outputFile, PrintStream ps)
    {
         try {
            ps.println("*** Processing sentences. ***");

            ClientCAndC client = new ClientCAndC();
            client.connect();

            BufferedReader br = new BufferedReader(new FileReader(inputFile));
            PrintWriter pw = new PrintWriter(outputFile);

            while (br.ready()) {
                 ps.println("Reading line");
                 String line = br.readLine();

                 int textPos = line.indexOf("\t") + 1;
                 String score = line.substring(0, textPos - 1);

                 String sentence = line.substring(textPos);
                 if(!sentence.equals(""))
                 {
                         String drsString = client.getDRS(sentence);
                         if(!drsString.equals(""))
                         {
                            ps.println("!!! PARSED OK");
                            pw.println(score + "\t" +
                                 drsString.replace("/"," or "));
                         } else {
                            ps.println("XXX FAILED");
                         }
                 }

            }

            ps.println("*** Finished processing sentences. ***");

            br.close();
            pw.close();

            client.close();

         } catch (Exception e) {
            e.printStackTrace();
         }

    }

}
```

```java
package pipeline;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

import parser.DRSreader;


/**
 * This identifies the most popular atoms in the dataset.
 * Used for testing purposes only, and is not used in the
 * overall pipeline.
 * @author Jamie
 *
 */
public class KeywordIdentifier {

    static String inputFile = "./files/5aii-answers-processed2.csv";
    public static void main(String[] args)
    {
        if(args.length> 1)generateFile(args[0]);
        else generateFile(inputFile);
    }

    public static void generateFile(String inputFile)
    {
        Map<String,Integer> atomCounts = new HashMap<String,Integer>();

        try
        {
            BufferedReader br = new BufferedReader(new FileReader(inputFile));
            int numLines=0;
            while(br.ready())
            {
                String line = br.readLine();
                if(line.indexOf("%")!=0)
                {
                    String[] lineParts = line.split("\t");
                    Set<String> set = DRSreader.atomize(lineParts[1]);
                    for(String atom : set)
                    {
                        if(atomCounts.get(atom)==null) atomCounts.put(atom, 1);
                        else atomCounts.put(atom, atomCounts.get(atom)+1);
                    }
                }
                numLines++;
            }

            br.close();

            // Want sorted list by the atoms' counts.

            Set<Integer> counts = new TreeSet<Integer>(atomCounts.values());
                for(Integer i : counts)
                    for(String val : atomCounts.keySet())
                        if(atomCounts.get(val).equals(i))
                            System.out.println(i+": " + val);

        }catch(Exception e){
                e.printStackTrace();

        }
```

pipeline/Trainer.java

```java
package pipeline;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.ObjectOutputStream;
import java.io.PrintStream;

import schemagenerator.CumulativeClassifier;
import weka.classifiers.Classifier;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.rules.DecisionTable;
import weka.core.Instances;

/**
 * Takes an arff file to train a decision tree with, and an instance
 * to classify, initially in DRS form. Produces the number of marks
 * this instance has been awarded.
 * @author Jamie
 *
 */
public class Trainer
{
   public static void train(String trainingFile, String trainedModel,
        String learningType, PrintStream ps)
   {
        try
        {
           ps.println("** No trained model found. Creating new one.");

           FileReader reader = new FileReader(trainingFile);
           Instances instances = new Instances(reader);
           instances.setClassIndex(instances.numAttributes()-1);

           Classifier dt = new NaiveBayes(); // default
           if(learningType.charAt(0)=='d')dt = new DecisionTable();
           if(learningType.charAt(0)=='j'){
                // Learning type is either 'j' (a standard mark scheme)
                // or of the form 'j-k', where k is the total possible marks
                // obtainable before it is capped the maximum mark.
                // (e.g. "Any of the k following, maximum n")

                // We also need to see if there is a 'main.marklist' file
                // in the same folder as the trainingFile. If so, use it.
                String markListFile = getFolder(trainingFile)+
                        "/main.marklist";
                if(!(new File(markListFile)).exists())markListFile = null;

                if(markListFile!=null)
                   ps.println("Using the manually created mark scheme.");

                if(learningType.length()==1)dt = new CumulativeClassifier(
                   markListFile,
                   changeExtension(trainingFile,"markscheme"));
                else dt = new CumulativeClassifier(
                   markListFile,
                   changeExtension(trainingFile,"markscheme"),
                   new Integer(learningType.split("-")[1]));
           }
           if(learningType.charAt(0)=='n')dt = new NaiveBayes();

           dt.buildClassifier(instances);

           ps.println("#############################");
           ps.println("EXECUTING TRAINING");
           ps.println("Using learning type: "+learningType);
           ps.println("** Saving trained model to '"+trainedModel+"'.");

           // Write trained model to file
           ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream(trainedModel));
           oos.writeObject(dt);
           oos.flush();
           oos.close();

        } catch (Exception e) {
           e.printStackTrace();
        }
   }

   /**
    * Changes the extension of a filename.
    */
   private static String changeExtension(String fileName, String newExt)
   {
        int pos = fileName.lastIndexOf(".");
        return fileName.substring(0,pos) + "." + newExt;
   }

   private static String getFolder(String fileName)
   {
        int pos = fileName.lastIndexOf("\\");
        pos = Math.max(pos, fileName.indexOf("/"));
        return fileName.substring(0,pos);
   }

   static final String trainingFile = "./files/testing/5aii-0.arff";
   static final String trainedModel = "./files/trained/5aii-0.model";
   public static void main(String[] args) {
        train(trainingFile,trainedModel,"d",System.out);
   }
}
```

```java
package pipeline;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.URL;
import java.net.URLEncoder;

/**
 * An alternative way of parsing English sentences
 * which uses an online version of the CAndC parser
 * and boxer. This is acceptable for single parses,
 * but it would be considered antisocial behaviour
 * to parse and entire data set.
 * @author Jamie
 *
 */
public class WebsiteParser
{
    // URL to use.
    final static String url = "http://svn.ask.it.usyd.edu.au/"+
          "demo/demo3.cgi?sentence=";
    final static String url2 = "&printer=prolog&model=a&resolve=yes"+
          "&submit=Parse+and+Box%21";

    // Used in the main method for parsing an entire file.
    static String inputFile = "./files/8b-answers.csv";
    static String outputFile = "./files/8b-answers-processed.csv";
    final static int exampleLimit = 1000; // do all of them
    final static long delay = 20000; // 20seconds

    public static String getDrsFromWeb(String sentence)
    {
          try
          {
               String fullUrl = url+URLEncoder.encode(sentence)+url2;
               System.out.println("Reading from: "+fullUrl);
               URL webUrl = new URL(fullUrl);
               BufferedReader urlReader = new BufferedReader(
                    new InputStreamReader(webUrl.openStream()));

               int i = 0;
               String drsString = "";
               while(urlReader.ready()&&i<=13)
               {
                    String line = urlReader.readLine();
                    if(i== 13)
                    {
                       int lastBracketPos = line.lastIndexOf(')');
                       drsString = line.substring(0,lastBracketPos).trim();
                    }
                    i++;
               }

               urlReader.close();

               if(drsString.equals(""))throw new Error("Problem obtaining "+
                    "DRS string.");
               System.out.println("Obtained the DRS: "+drsString);

               return drsString;
          } catch(Exception e){
               e.printStackTrace();
               return null;
          }

    }


    /**
     * @deprecated
     * Use of the online parser for an entire data set
     * is strongly discouraged. Use FileParser instead.
     */
    public static void main(String[] args) {

          try
          {
               BufferedReader br = new BufferedReader(new FileReader(inputFile));
               PrintWriter pw = new PrintWriter(outputFile);
               int exampleCount = 0;

               while(br.ready()&&exampleCount<exampleLimit)
               {
                    String line = br.readLine();

                    int textPos = line.indexOf("\t")+1;
                    String score = line.substring(0,textPos-1);

                    String drsString = getDrsFromWeb(line.substring(textPos));
                    System.out.println("Obtained DRS: "+drsString);
                    pw.println(score+"\t"+drsString);

                    exampleCount++;

                    System.out.println("** Sleeping for 20 seconds");
                    Thread.sleep(delay);
               }

               br.close();
               pw.close();

          } catch(Exception e){
               e.printStackTrace();
          }

}
```

```java
package [illegible];

import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible].Act[illegible];
import [illegible];
import [illegible];
import [illegible]Map;
import [illegible]Map;

import [illegible];
import javax.sw[illegible];
import [illegible];
import javax.swing.[illegible];
import [illegible];
import javax.swing.[illegible];
import javax.swing.JTextArea;
import javax.swing.[illegible];
import javax.swing.border.Border[illegible];
import javax.swing.[illegible];

import [illegible];

/**
 * The panel on which the user [illegible]
 * of the system.
 * @author [illegible]
 *
 */
public class EvaluatorPanel extends [illegible] implements PropertyChangeListener
{
        final String INSTRUCTION = "Create a new folder, containing either " +
                "of the following:\n" +
                "* answers.csv - a file of the source answers in English.\n"
                "* answers-processed.csv - a file of the parsed sentences.\n"+
                "  This skips the parsing stage.";

        // Used to map learning type selection
        // to their representative letter.
        final Map<Integer,String> [illegible]Map = new HashMap<Integer,String>();
        final [illegible];
        // The choice of learning types.
        [illegible];
        [illegible] progressBar;
        [illegible];
        final [illegible] = new [illegible](this,
                "Test directory","DIRECTORY");

        public EvaluatorPanel()
        {
                super();
                [illegible] new [illegible];

                [illegible] new [illegible] new [illegible];
                [illegible] = new [illegible];
                // [illegible]TextArea instructions = new JTextArea(1,30);
                [illegible];
                startButton = new JButton("Start");
                [illegible] new [illegible];
                [illegible];
                [illegible]
                [illegible], "d");
                [illegible], "n");
                [illegible], "j");
                [illegible] "Decision Tree",
                        "Naive Bayesian Network", "Cumulative Classifier" };
                [illegible] = new [illegible];
                [illegible];
                [illegible];

                [illegible];

                [illegible] = new [illegible];
                [illegible](progressBar, [illegible]);


                [illegible] = new [illegible]("Status");
                JPanel statusPanel = new JPanel();
                statusPanel.[illegible](statusBorder);

                statusArea = new JTextArea(14,60);
                [illegible];
                JScrollPane scrollPane = new JScrollPane(statusArea);
                statusArea.setEditable(false);
                statusPanel.add(scrollPane);

                JPanel bottomPanel = new [illegible];
                centerPanel.add(testFolder);

                add(topPanel,BorderLayout.NORTH);
                add(statusPanel,BorderLayout.CENTER);
                add(bottomPanel, BorderLayout.SOUTH);
        }

    public void propertyChange(PropertyChangeEvent evt) {
        if ("progress" == [illegible]) {
            int progress = (Integer) evt.getNewValue();
            [illegible].setValue(progress);
        }
    }

        public class EvaluatorAction implements ActionListener
        {
                public void actionPerformed(ActionEvent e)
                {
                        // Flush textarea
                        statusArea.setText("");

                        // Get learning type
                        final [illegible] = [illegible]
                                learningList.getSelectedIndex());

                        final String folderName = testFolder.getFileNam[illegible];
                        if ([illegible] == null) statusArea.setText(
                                "ERROR: No folder selected");
                        else {
                        [illegible](false);
                        [illegible]
                                [illegible];
                        Task task = new [illegible](folderName, learningType);
                        [illegible].this);
                        [illegible] ;




        /**
         * [illegible]
         * [illegible]
         * [illegible]
         *
         */
public class [illegible] extends [illegible]

                [illegible];
                [illegible];
        public [illegible]
                this.[illegible] = [illegible];
                this.[illegible] [illegible];


        public [illegible]

                if [illegible] "j"
                {
                        String s
                                (String)JOptionPane.[illegible]
                                "OPTIONAL: If the mark scheme is of"+
                                " the form 'Any of the k following,"+
                                " maximum n', enter k.");
                        if(s!=""&&s!=null)[illegible] "j-"[illegible];
                }

                Tester.test(folderName, learningType,
                        new GUIOutput(statusArea), [illegible];
                return null;
        }

        public void done(){
                Toolkit.getDefaultToolkit().beep();
                startButton.setEnabled(true);
                setCursor(null); //turn off the wait [illegible]
        }
}
```

```
package [illegible];

import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];

import [illegible];
import javax.swing.ImageIcon;
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.JLabel;
import javax.swing.[illegible];
import javax.swing.border.Border;
import javax.swing.border.[illegible];
import javax.swing.border.[illegible]Border;
import javax.swing.filechooser.FileFilter;

/**
 * A square panel that represents a file or [illegible]
 * selection. The users can specify a file by one of
 * two methods:
 * [illegible] click on the panel and select the file [illegible]
 *    in the normal fashion.
 * [illegible] drag a file [illegible] into the [illegible]
 * [illegible]
 *
 */
public class [illegible] extends [illegible] implements [illegible] {

        static [illegible] = "./icons/document.gif";
        static String [illegible] = "./icons/document_greyed.gif";

        final static [illegible] = new [illegible];

        [illegible];
          JLabel [illegible];
          [illegible];
          String fileType;
          [illegible];
          [illegible];
          [illegible];
          [illegible];

          public [illegible]      return [illegible];

          public [illegible]

            super [illegible];
            this.[illegible];
            this.[illegible];
            [illegible] new [illegible] this,[illegible];

            [illegible] new [illegible] " ";
            [illegible];
            [illegible] new [illegible] "Tahoma", [illegible];
            [illegible];

            [illegible] = new [illegible] new [illegible];

            [illegible] new [illegible] new [illegible];
            [illegible] new [illegible];
            [illegible];
            [illegible];
            [illegible] new [illegible] new [illegible];
            [illegible];
            [illegible];

            [illegible] new [illegible];
            Font [illegible] = new Font "Tahoma",[illegible];
            [illegible];

              [illegible] text [illegible] drops...
              [illegible] class will handle drop events
            [illegible] = new DropTarget(this, this);
          }

          public void [illegible]
          public void dragExit([illegible] dte) { }
          public void dragOver(DropTargetDragEvent [illegible]
          public void [illegible]

          public void [illegible]
                   try
              // Get the dropped object and try to figure out what it is
              [illegible];
              [illegible];
              for (int [illegible]
                       Check for file [illegible] specifically
                   if [illegible]
                            [illegible] drop...
                         [illegible]
                           [illegible];

                         java.util.List [illegible]
                           [illegible];
                         [illegible]"");

                         // If we made it this far, everything worked.
                         [illegible] true ;
                         return;
                   }

                      [illegible]
                   else if [illegible]
                         [illegible];
                           [illegible];
                         // [illegible];
                           [illegible];
                         [illegible] true);
                         return;

                     [illegible] about an input stream.
            else if [illegible]
                    [illegible] ;
                    [illegible]
                    [illegible] true ;
                    return;

                [illegible]
        [illegible] ;
            catch [illegible]
                    [illegible] ;
                    [illegible] ;

    }

    public void [illegible]

            this.[illegible] ;
            [illegible] ;
            [illegible] ;

    private static [illegible]

            int [illegible] Math.max(fileName.lastIndexOf("\\" ,
                fileName.lastIndexOf("/"));
            String fileNameOnly = fileName.substring([illegible]);
            if(fileNameOnly.length()<=17) return fileNameOnly;
            // Otherwise we need to shorten it.
            // First find dot
            int posDot = fileNameOnly.lastIndexOf(".");
            // Remove middle.
            return [illegible]NameOnly.substring(0,[illegible])+"...".
                [illegible]NameOnly.substring(posDot-[illegible]);
    }

    public class [illegible] implements [illegible]

            [illegible] new [illegible] ;
            final [illegible] ;
            public [illegible] final [illegible]

                    this.[illegible] = type;
                    [illegible] = new My[illegible] ;
                    if [illegible]("DIRECTORY" [illegible]
                            [illegible]CHOOSE DIRECTORIES ONLY ;
                    [illegible] ;
            }

            public void [illegible] MouseEvent

                    // Handle open button action.
              int [illegible] ;

              if [illegible]
                    [illegible] ;
                    [illegible] ;
                else
                    [illegible]

            public void [illegible]
            public void [illegible]
            public void [illegible]
            public void [illegible]

    public class My[illegible] extends [illegible]

            String [illegible] ;
            public My[illegible]
                    this.[illegible]Type;

            public [illegible]
               return [illegible] "DIRECTORY" [illegible] "Directory" :
                    "."+[illegible] ;

            public boolean [illegible]

                            [illegible]
                            [illegible]
                    if [illegible] return true;

                    [illegible] ;
                    if [illegible] != null

                            if [illegible] return true;
                            else return false;
                    }
                    return false;
            }
            public String getExten[illegible] f) {
                        [illegible] null;
                    String [illegible] ;
                    int [illegible] = [illegible].lastIndexOf('.');

                    if [illegible]
                        [illegible] ;

                    return [illegible] ;
```

```
package [illegible];

import [illegible];

import [illegible];

[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
public class [illegible] extends [illegible]

        [illegible];
        public [illegible]TextArea [illegible]
                super [illegible]; this[illegible]

        public void [illegible]          [illegible].append([illegible] "\n");

        public void [illegible]          [illegible]

        public void [illegible] int [illegible]          [illegible]("");

        public void [illegible]          [illegible].append("\n");

[illegible]
```

```
package [illegible];

import [illegible];
import [illegible];
import [illegible];
import [illegible];

[illegible]

public class [illegible]
        public static void [illegible]
                [illegible] new [illegible] "Marker" [illegible];
                [illegible];
                [illegible] true [illegible];

            try
                UIManager.setLookAndFeel[illegible]
                            UIManager.getSystemLookAndFeelClassName[illegible];
              catch (Exception e[illegible]


            [illegible]TabbedPane tabs [illegible] new [illegible]TabbedPane();
            tabs.addTab("Pipeline",
                new ImageIcon("./icons/palmpilot.gif"),
                new PipelinePanel());
            tabs.addTab("Evaluator",
                new ImageIcon("./icons/cog_small.gif"),
                new EvaluatorPanel());
            tabs.addTab("Marker",
                new ImageIcon("./icons/construct.gif"),
                new MarkerPanel());

            [illegible]ContentPane[illegible].add(tabs);

            [illegible];
```

```java
package [illegible];

import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];

import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible]dBorder;

import parser.DRSReader;
import pipeline.WebsiteParser;
import weka.classifiers.[illegible];
import weka.[illegible];
import weka.[illegible];

/**
 * A panel to input sentences one by one and
 * receive individual classification.
 * @author [illegible]
 *
 */
public class MarkerPanel extends [illegible]
{
        [illegible] = new [illegible](this,"Arff","arff");
        [illegible] = new FileLoadPanel(this,"Model","model");
        [illegible];
        [illegible];

        public MarkerPanel()
        {
                [illegible](new [illegible]);

                [illegible] = new [illegible](new [illegible]);
                [illegible](new [illegible]);

                [illegible] = new [illegible](new [illegible]);
                [illegible](new JLabel("Sentence: "));
                [illegible] = new [illegible];
                [illegible];
                [illegible] = new [illegible]("Submit");
                [illegible](new [illegible]);
                [illegible];
                [illegible];
                [illegible] = new [illegible]("Input");
                [illegible];

                [illegible];

                [illegible] new [illegible] new [illegible];
                [illegible];
                  [illegible];

                [illegible];

                [illegible] = new [illegible] new [illegible];
                [illegible] new [illegible]" ";
                  [illegible] new [illegible]("Tahoma", [illegible]);
                [illegible];
                [illegible];
                  [illegible];
                [illegible] = new [illegible]("Mark");

                [illegible];
                [illegible];
        }

        /** [illegible] 'Submit' button.
        public class [illegible] implements [illegible]
        {
                public void actionPerformed([illegible])
                {
                        // arff and model files must be loaded.
                        if([illegible] == null) return;
                        if([illegible]==null) return;

                          [illegible](
                                  answer.getText());
                        [illegible] = WebsiteParser.[illegible](
                                  answer.getText());
                        Set<String> set = DRSReader.[illegible];
                        List<Double> [illegible] = new LinkedList<Double>();

                        try
                        {
                           [illegible] = new [illegible](
                                  new [illegible]);
                           [illegible]("** Reading ARFF.");

                           boolean [illegible] = true;
                           int [illegible];
                           while([illegible])
                           {
                                  [illegible];
                                  if([illegible]("@attribute")[illegible]
                                          [illegible] false;
                                  else if [illegible]("@attribute"[illegible]
                                     [illegible] " ";
                                     if([illegible]("score"[illegible]
                                             if [illegible] "\"","" 
                                                     [illegible];
                                             else [illegible];


                                  [illegible];

                           [illegible];

                           [illegible];
        int [illegible];

        double [illegible] = new double [illegible];
        for int [illegible];
                [illegible];

                new [illegible];

        [illegible] "** Found trained model '".
            [illegible] +"'." ;

        [illegible]
        new [illegible] new [illegible];
        [illegible];
        [illegible];

        [illegible] new [illegible];
        [illegible] new [illegible];
        [illegible] new [illegible];
            [illegible];
        [illegible];

        double [illegible];
        [illegible] "Classified as: " [illegible];

        [illegible].setText([illegible]new [illegible]+"") ;


    } catch([illegible] ex){
            ex.printStackTrace();
    }
```

```
package [illegible];

import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];
import [illegible];

import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];
import javax.swing.[illegible];

import [illegible];
import [illegible];
import [illegible];


/**
 * This panel allows the user to perform each
 * stage of the parsing/training pipeline, and
 * converts between the 4 file formats used:
 * 1. Source CSV - English sentences.
 * 2. Semantic CSV - parsed sentences.
 * 3. Arff - a classifier input file.
 * 4. Model - A trained classifier.
 * @author Jamie
 *
 */
public class PipelinePanel extends JPanel {

        FileDragPanel [illegible] = new FileDragPanel(this,"Source CSV","csv");
        FileDragPanel [illegible] = new FileDragPanel(this,"Semantic CSV","csv
");
        FileDragPanel [illegible] = new FileDragPanel(this,"Arff","arff");
        FileDragPanel [illegible] = new FileDragPanel(this,"Model","model");
        TextArea statusArea;

        public PipelinePanel()
        {
                setLayout(new [illegible]);

                Border [illegible] = new [illegible]("Status");
                JPanel [illegible] = new JPanel();
                [illegible];

                [illegible] = new [illegible];
                [illegible] = new [illegible];
                [illegible](false);
                [illegible];

                [illegible];

                [illegible]
                [illegible] = new JPanel();
                [illegible](new [illegible]);
                [illegible](new [illegible]);
                [illegible]
                [illegible] = new [illegible]">");
                [illegible] new [illegible]
;
                [illegible];
                [illegible];

                [illegible] = new [illegible]">");
                [illegible] new [illegible];
                [illegible];
                [illegible];

                [illegible] new [illegible]">");
                [illegible] new [illegible];
                [illegible];
                [illegible];
                [illegible];


        public class [illegible] implements [illegible]
        {
                public void [illegible]
                {
                        String [illegible];
                        String [illegible];
                        if([illegible]==null [illegible]==null)
                                [illegible]("ERROR: Files not specified
.");
                                return;
                        }
                        [illegible]
ame,new [illegible]);
                }
        }

        public class [illegible] implements ActionListener
        {
                public void actionPerformed(ActionEvent [illegible])
                {
                        String sourceFileName = [illegible];
                        String [illegible]FileName = [illegible];
                        if(sourceFileName== null [illegible]==null)
                                statusArea.setText("ERROR: Files not specified
.");
                                return;
                        }
                        [illegible] new [illegible];
                        [illegible]
[illegible], false, new [illegible]);
                }
        }

        public class [illegible] implements ActionListener
        {
                [illegible] = new [illegible];
                public [illegible]
                {
                        [illegible]("Decision Tree", "d");
                        [illegible]("Naive Bayesian Network", "n");
                        [illegible]("Cumulative Classifier", "j");
                }

                public void [illegible]
                [illegible]
                [illegible]
                if [illegible]==null [illegible]==null
                        [illegible] "ERROR: Files not specified
." ;

                        return;


                [illegible]
y[illegible];
                [illegible]
                                        [illegible].this,
                                        "Please choose a learning type",
                                        "Learning Type",
                                        [illegible],
                                        null,
                                        [illegible],
                                        "Naive Bayesian Network" ;

                if(s! null)[illegible]
[illegible] "", new [illegible];
```