Syntagma Vn 1.2 - Program notes

Summary
Syntagma is a program written in C# to take a corpus of text and discover word classes and structures of classes. This document describes the internal workings of the program.

Installation
The running program is available in a zip package on my website, colinday.co.uk/downloads. Download the file into a folder of your choosing and double click on it for the separate files to be unzipped. Run the program by double-clicking on Syntagma.exe.

The program source files are also contained in a zip package on the same website. When unzipped, they form a project to be run under Microsoft's Visual Studio.

Terminology
A distinction is to be made between word tokens (individual words appearing in the corpus) and a word type (the collection of word tokens with the same form). Here *word* is used, usually meaning word type, but hopefully the context makes it clear which is intended.

A *class* is a collection of word types which are linked by occurring in the same or similar environments. A *structure* is a pair of classes or structures which occur in sequence with adequate frequency.

A *component* is a word, a class or a structure. A *composite* is a class or a structure.

Overall design
The program is plentifully equipped with parameters which the user can alter at any time. It is also possible for the user to change the amount of information which the program puts in a log file. In these ways, it is hoped that the program will be a suitable experimental tool.

After initialisation, the program works in two phases. Phase 1 seeks to find initial classes on the basis of statistical collocation between words. Phase 2 looks for further classes and structures of classes on the basis of collocation with the classes and structures found so far. Phase 1 may only be run once. Phase 2 may be run any number of times.

Data structures
Each component is mapped into a unique integer, so the program is always handling integers and not strings. These integers are the indices within a list *wdCl* of *Item*. Zero is not used to identify a component, and 1 is used for a full stop.

An *Item* is a C# object, with the following structure:
- *itName* string, the name of the component;
- *itCount* integer, the number of occurrences in the corpus;
- *itMembers* List<int>, the members of this class;
- *itClass* List<int>, the classes for which this is a member;
- *itPre* Dictionary<int,int> the components found preceding this, with the frequencies of co-occurrence;
- *preSig* List<int> the components found significantly preceding this component;
- *itPost* Dictionary<int,int> the components found following this, with the frequencies of co-occurrence;
- *postSig* List<int> the components found significantly following this component.

Contexts are pairs of particular classes between which components may be listed. These are stored in *conText*:
Dictionary <long, Dictionary<int,int>>
Here the long variable is used to pack together the integers representing the two classes, and the dictionary stores the components found within this context, with the frequencies with which they are found in this context.

Input text
The input file is to be an ASCII text file. For the purposes of development the corpus used has been the book *Frontier Ways* (1959) by Edward Everett Dale, University of Texas Press. The University of Texas Press have kindly given me permission to distribute this text along with my program.

The way that the corpus is handled depends on the initial parameters.

Initial parameters
The default settings may be changed by clicking on *Set initial parameters*.

The input file may be specified, and the file to be used for logging information. The text will by default be capitalised, but this may be blocked.

The only punctuation which the program needs is the end of sentence. All end-of-sentence characters are treated alike. Punctuation at the start or end of words will be ignored. Lists of these types of punctuation may be defined.

Display information
Some information is displayed on the screen by means of message boxes. The level of this information may be changed before a phase is run.
- Level 1 (default) Information is given at end of every phase.
- Level 2 Extra information is displayed from time to time.

Logging levels
Information is written to the logging file. The amount of information may be changed before a phase is run.
- Level 0. No information is written. At the end of the phase, the level reverts to 1.
- Level 1 (default). Basic information is written.
- Level 2. Extra details on the program working.
- Level 3. More detailed information.
- Level 4. Extremely detailed information.

Running parameters
A number of parameters may be changed to alter the way the program is run. These are essentially thresholds which govern whether certain actions are taken or not. The defaults have been chosen to match the input text used during development. These parameters may be changed before any phase is run. Specific details are given below as the working of the program is explained.

All the parameters may be read from the file *Parameters.txt*. This my only be done once, before Phase 1 is run.

Documentation and help
This gives some description of the way the program runs and what the user needs to supply. It is generated from an ASCII file *Documentation.txt*, and is displayed by means of a specially written C# method which runs within Form 3.

Phase 1

Phase 1 may only be run once. Initial parameters are activated. The purpose of Phase 1 is to find 'seed classes', usually involving those few words which occur very frequently. These classes can then be used to make a way into discovering other classes in view of collocations with the seed classes.

Reading the corpus (Method *readCorpus*)
The input file is read line by line. Ends of lines are treated as spaces. Words are transformed into unique integers by means of a temporary dictionary, and an item created for each word type. The dictionary is not needed after the text has been read, as the string form of each word is stored in the corresponding *Item*.

The words are assembled into sentences, each sentence beginning and ending with a full stop (*Item* number 1). This means that each word has a preceding and a following neighbour.

Collocations (Method *countCollocations*)
The collocations (co-occurrences) of words are counted. This involves processing all the sentences, and recording for each word in the *itPre* and *itPost* parts of its *Item* the words found immediately before and after it, with the number of such occurrences. The number of times each word occurs is also counted, and stored in *itCount* within *Item*.

Significant collocations (Method *findSignificances*)
Each word is compared with every other word to see whether the collocations between them are significant.

Let the total number of word in the corpus be C and the number of occurrences of word A be N_a and that of word B be N_b. Then the probability of finding word A at a particular location in the corpus is N_a/C. Similarly the probability of finding word B at a particular location is N_b/C. The probability of finding A followed by B at a particular location is $(N_a/C) * (N_b/C)$. However, there are C locations at which this might happen. So we may say that the most likely number of times we may find this collocation is N_a*N_b/C. Let us call this value M.

The distribution of words in the corpus is considered to exhibit the Poisson distribution. For such a distribution, the expected or mean value (M here) is the same as the variance, which is the square of the standard deviation. So if we want to know how many standard deviations x is from the mean, this is given by
(x - M) / sqrt(M).

A test is made on all the *itPre* and *itPost* lists of all words (method *itemSignifs*). Taking the *itPre* lists, if two words have the same word in the lists at least Parameter 2 times, and if there are at least Parameter 1 standard deviations of significance for these collocations, then the word is added to the appropriate *preSig* list. The corresponding action is carried our on the *itPost* lists also.

Class creation (Method *classifyWords*)
For each word, the following tests are carried out. First the word is compared with all classes found to date, and if a sufficient pairing is found, the word is added to the class. Then the word is compared with all other words, and if a sufficient pairing is found, two words are joined together into a new class.

Sufficient pairing is considered if both the pre and post lists are sufficiently paired. For this to be the case for the pre lists, at least Parameter 3 words should be in common to both the *preSig* lists of the two components (method *testPair*).

When a new class is created, the *itPre* and *itPost* lists of the two words are merged into the corresponding lists for the class. The *itCount* of the class is the sum of the two counts for the words. The significant collocations for the class are computed. Finally a test is made to see whether the new class is sufficiently paired to any already existing class, and if it is, the new class is combined with the previous one.

Class names
Normally, the program will use names such as 'Class1', 'Class2' and so on whenever a new class is created. However, in order to ease the task of assessing whether the classes are reasonable, the program allows the user to append a file *Classnames.txt* with the class names to be used. After the program is run once, the names which are desired can then be chosen. Each name occupies one line in the file. This file is not needed for the program to run successfully. If the file contains more lines than there are classes, the rest are ignored. If the file has too few lines for the classes, the program reverts to its normal naming for the remaining classes.

This file is read whenever a new class is created.

Phase 2

Phase 2 may be run repeatedly. Each time before it is invoked the running parameters and levels of information may be changed. Each time Phase 2 is run, just one extra class may be formed, though this may be subsumed within existing classes.

Parsing the text (Method *parseText*)
The counts for all composites are set to zero, as the frequencies will be found as the parsing takes place.

Each sentence is copied, and then members of classes are replaced by those classes. If a word or structure is a member of more than one class, it is not replaced, in view of the ambiguity as to which class is represented. Structures are recognised if adjacent classes are found which are stored in the dictionary of structures *structIndex*.

Now the sentence is scanned to find contexts, i.e. situations where components are bracketed between two composites. The pair of composites is known as a *context*. Information is stored in a dictionary *conText*, defined as <*long, Dictionary<int,int>>*. The pair of composites is stored in the *long*, and the value dictionary is used to store the component occurring within this context with its frequency of occurrence.

The sentence is then scanned to find occurrences of classes adjacent to each other. Such pairs are stored as incipient structures in the dictionary *structWait*.

Storing structures (Method *findStructs*)
The incipient structures deposited in *structWait* during the parsing of sentences are now examined. If the two composites spend at least Parameter 7 percent of their existence as part of the structure, that structure is now given its own *Item* and its details stored in *structIndex*. At the end, the dictionary *structWait* may now be emptied.

One class in a context (Method *gleanContexts*)
If a context brackets only one class, then it may be considered that all other items in that context belong as members of that class. The criteria are that the class spends at least Parameter 5 per cent of its existence within that context, that words to be added to the class spend at least Parameter 6 per cent of their existence within that context, and that words exhibit sufficient

pairing with the class (method *testPair* as described under **Class creation** for Phase 1 above).

Major context (Method *selectContext*)
A search is conducted to find the context which occurs with the greatest number of components bracketed within it. This context is then taken as a matrix for building another class.

Building another class (Method *assimilateClass*)
An attempt is now made to build a further class out of the components which occur within the most frequently occurring context.

A new class is started with a new *Item*. All of the components which occur in the context (apart from classes) are added to the new class, provided that they spend Parameter 8 per cent of their lives within this context.

The significant collocations of all members of this new class are recorded (method *itemSignifs*).

Testing the class (Method *checkClass*)
A test is carried out to see whether this last class to be built will merge with a previously existing class. This involves comparing the overlap of *preSig* and *postSig* lists between the two classes. If the overlap is less than Parameter 4 per cent of the frequencies of either classes, then merging is aborted.

Uniting classes (Method *uniteClass*)
This is only ever needed when the last class to be built, whose *Item* is the last in the list (class A, say), is to be united with a previously existing class (class B, say). This involves transferring members of Class A to be members of class B, merging the lists of collocations and significant collocations and adding together the frequencies of occurrence.

The problem now exists, what should happen to the *Item* which was assigned to Class A? Deleting it would involve too many problems, so the solution which has been chosen is to render it invalid (method *nullClass*). This removes all members from the class, sets its frequency of occurrence to zero and sets its name (*itName*) to ".Null".

Postscript

Syntagma is intended as an experimental vehicle. It has not been tested on other (non-inflected) languages, or on different volumes of text. Even with the supplied input text, things start to go wrong when Phase 2 has been run four or more times. Some improve- ment might be found if the running parameters were varied.

The elapsed times for running the phases cannot be accurate. The time taken depends on what other activities the computer was engaged in at the same time. Therefore, the times can only be taken as maxima. Even so, the program can read in 80,000 words of text and classify a quarter of its words in no more than half a second in total.

Colin Day
12th August 2018