Algorithmics/Java project: morphysynctactic tagger

Antoine Lafouasse

May 2013

Introduction

This project's objective is simple: make an application capable of building a statistic model for morphosyntactic tagging. In other terms the program has to read a first corpus in order to learn its content (i.e. retain every word in memory, and associate to each of them a list of different parts of speech with which that word has appeared in the corpus – doing so, it knows for each word which part of speech has appeared the most times in the corpus and is therefore, theoretically, the most likely to appear in an unknown context). Its following task is then to test that model by confronting it to another corpus, which leads it to calculate its accuracy and generate a new corpus with the words from the testing corpus and the parts of speech guessed by the model.

If that objective is fairly simple to understand, it is difficult to deny that it involves storing and organising complex structures of data, which obviously makes performance non-trivial to achieve in such an application. Our choices in data structures therefore need to be well chosen and implemented, and any action needs to be thoughtfully planned so as not to destroy the program's execution times ; and as such, our key objective will be performance.

In that regard, this report will fall into three parts. We will first have an insight into the application's class structure. We will then have a look at the entire process's unfolding and try to assess its complexity and performance, before exploring in a third and final part possible improvements.

1 Class structure

This application could appear to be rather complex if we only look at the sheer number of classes. But truth is, dividing it into a few blocks is simple and makes understanding the whole structure fairly easy.

1.1 Abstract structures

These classes act as "templates" insofar as they have no fields to store any form of data whatsoever but provide accessors and references so as to build easy-to-use, reusable structures. These classes – except `Queue` and `NodeAlreadyExistsException` – are all abstract, which means they cannot be instantiated.

1.1.1 `AVLTree<T extends AVLNode<T>>`

This is, very simply, a self-balancing Binary Search Tree implementing the model by G. M. Adelson-Velskii and E. M. Landis, which allows it to consistently search and insert in $O(n \log n)$, both in average and worst case. What's more, its space requirement is only in $O(n)$, which stands for a rather anecdotic memory over-cost. Its two public methods, `find()` and `push()`, are quite straightforward and I will not dwell on them, but what is to note is that `AVLTree` is a generic type, with `T` being the type of nodes stored in the tree – hence `T` obviously being a specialisation of `AVLNode`.

1.1.2 `AVLNode<T extends AVLNode<T>> implements Comparable<T>`

As previously stated, `AVLNode` is the type of nodes typically stored in a Tree. It is also a generic type, but for a different reason. Being an abstract class it is fairly obvious that one will have to write a specialisation of `AVLNode` before using it. `T` in this particular context is the specialisation of `AVLNode` itself, which means that the `left()` and `right()` methods will return an object of type `T`. As a result, it is not possible for a given tree to store a mix of different specialisations of `AVLNode` together – its nodes will all have to be of the same type. As for the methods, they are quite straightforward as well: both the `left` and `right` attributes have two mutators, an accessor and an hasser to check for their existence. In conformity with their being nodes from a binary search tree, these methods also include `equals()`, `compareTo()` and `toString()` that are required to be properly overridden. The reason is that a binary search tree obviously relies on ordering, which implies comparison. `toString()` was also given as a required method for the `AVLNode` to be able to be compared to other objects – which proves handy, for example, with `AVLTree.lookup()` which takes a String as an argument.

1.1.3 `NodeAlreadyExistsException extends Exception`

This exception is thrown by `AVLTree` if the user tries to insert a node who already is in the tree – i.e. if there is already a node in the tree, whose value would return true if tested with `equals()` from the node we try to insert. It is a particular kind of exception insofar as it stores the node that was found by the tree, which can then be accessed through the `getNode()` method. This allows the user to directly pick the involved node without having to use `find()` as fallback and therefore do an unnecessary second lookup.

1.1.4 `ListNode<T extends ListNode<T>> implements Iterator<ListNode<T>>`

As its name reads, `ListNode` is a node from a single-linked list. As for its being a generic type, it is for the very same reason as with `AVLNode`, which is to prevent different specialisations of `ListNode` from being linked with one another. As for its methods, they are straightforward as well as they include two mutators, one accessor and one hasser for the `next` attribute, in conformity with `Comparable<T>`'s requirements. The only exception is that `remove()`, which is not needed in our application, was not implemented and will always raise a `UnsupportedOperationException`.

1.1.5 `Queue<T extends ListNode<T>>`

Note that contrary to the three classes described above, `Queue` is *not* an abstract class. As its name states fairly obviously, this is a FIFO (First In, First Out) structure which means

that the first element that was inserted in this structure will be the first to be removed every time. As such, its methods are canonical as well, and they are `push()` `peek()` and `pop()` whose roles are to add an element to the end of the queue, return the first element and return the first element while deleting it from the queue. One peculiar thing is that while `pop()` unlinks the node before returning it, `peek()` does not, which allows the user to use the result of that method as a regular linked list. While this possibility is handy when the user needs to loop through the contents of the queue without emptying it and simply appending an item at the end of it, modifying such a linked list without using `Queue`'s mutators is strongly deprecated.

1.2 Data storage: the corpora

In order to both make data processing easier and verifying that the corpora are correctly formatted, it appeared necessary to tokenise them into objects. We will therefore see how this particular structure is achieved.

1.2.1 `Entry extends ListNode<Entry>`

Very simply, this class contains two `String` attributes: one which stores the value of a word and the other which is the part of speech associated to it. It is therefore the direct object representation of the corpus's `Word_PartOfSpeech` format. As such, these attributes are set once and only once through the object's constructor and cannot be modified afterwards, but they can be read with the `value()` and `partOfSpeech()` methods.

1.2.2 `Sentence extends Queue<Entry>`

This object's prototype is quite straightforward: it is a `Queue` containing instances of `Entry` and it represents one sentence in a Corpus. It therefore has all of the methods defined in its mother class, plus one which allows it to directly add an element with two `String` arguments, namely a word and its associated part of speech.

1.2.3 `Corpus`

This is the object representation of one entire corpus. In formal terms, this could be seen as the *very* simplified implementation of a hash table, containing thus an integer key and a value which is a `Sentence` – the key actually is the order of the sentence in the actual text, so that the `keys()` method returns an array of all the keys that are used in the table, and that array can be looped through to browse the corpus in the order of the text. As for the other methods, they are rather trivial: one constructor creates a table using the size given in argument, while the other sets the size to the `defaultSize` static argument, and you can find `put()` to add or overwrite a `Sentence` to the table and `get()` to fetch one using its key as argument.

1.3 Data storage: the model

The other data structure we need to implement is, of course, the statistic model itself. It is most probably the most critical point in terms of performance, since its toll in terms of lookups is most certainly heavy. But as we will see here, its implementation is in fact rather simple.

1.3.1 `PartOfSpeech extends ListNode<PartOfSpeech> implements Comparable<PartOfSpeech>`

If we remember what we talked about in the introduction, it appears quite obvious that this class must have a `String` value and another integer attributes which is the number of times this particular part of speech has appeared for a given word in the learning corpus. These attributes can be accessed through the `toString()` and `occurrences()` methods, and the number of occurrences can be incremented with the `addOccurrence()` method. The value of the part of speech can only be set through the constructor and of course, it is not possible to modify it afterwards. And as `PartOfSpeech` objects need to be sorted, there is also a `compareTo()` method as specified by the `Comparable<T>` interface.

1.3.2 `Word extends AVLNode<Word>`

This is, quite obviously, the representation of a single Word as it is "remembered" in the model, so to speak. As a result, it contains a `String` value which is defined in the constructor, unmodifiable afterwards, and can be accessed through the `toString()` method. As it is a specialisation of `AVLNode` it has both `equals()` and `compareTo` implemented, which allows it to be compared both to other `Word` and `String` objects. It also contains a list of `PartOfSpeech` – namely, a `Queue` which needs to be sorted as soon as the model is complete.

1.3.3 `Dictionary extends AVLTree<Word>`

This is the object representation of the model itself. As its inheritance says it is an `AVLTree`, with all the methods previously enumerated. To those we must also count another `add()` methods which takes as arguments two `String` arguments, a word and a part of speech, which are in fact the content of an `Entry`. This particular method is responsible for finding and incrementing the correct part of speech, or creating either the part of speech or the entire word if needed. Another method, `sort()`, needs to be used at the end of the learning corpus to allow the `PartOfSpeech` that has occurred the most times in the corpus to rise to the top of its `Queue` – its role is to sort every `Queue` of `PartOfSpeech` in every `Word` by descending order of occurrences. Note that as the `Dictionary` must be one and only throughout the application's entire process, it is a Singleton in conformity with Bill Pugh's implementation of the Singleton pattern in Java. – hence the instance being stored in an encapsulated private `InstanceHolder` class and its only being able to be accessed through the static `getInstance()` method.

1.4 File handling classes

Of course, the two instances of `Corpus` representing the learning and testing corpora need to be generated at some point – and the result, formatted as a `Corpus` as well, needs to be saved to a file. We will therefore see which classes are responsible for this process.

1.4.1 `FileHandler`

This could be scene as the object representation of a file. First a `FileHandler` needs to be instantiated through the constructor, which takes a `String` path to the file as an argument. If the file does not exist, `FileHandler` creates an empty one. After that, the

user can call `isEmpty()` to check if the file is empty, `read()` to read the file and render a `Corpus` out of it or `save()` to save a `Corpus` to the file.

1.4.2 `Loader`

Its role is to load the program arguments and load the three `FileHandler`s responsible for the learning, testing and result corpus – checking if the files are empty and loading the default corpora as fallback if needed. All of that is done in the constructor, and the user can then call `getLearn()`, `getResult()` and `getTest()` to fetch the three `FileHandler`s. The fallback paths are specified in the `defaultLearn`, `defaultResult` and `defaultTest` static attributes.

1.5 Execution classes

They could be seen as the "main matter" so to speak, as they are the classes who do the actual job of building the statistic model and testing it. They are, as such, more utility classes than actual objects – except for `Result`.

1.5.1 `Learner`

Its role is straightforward and is the reason why its content only consists in one `getDictionary()` method: it takes a `Corpus` in input, loops through it and builds the dictionary using the data from the `Corpus`.

1.5.2 `Tester`

Its role is a little more subtle, since its output contains two elements: the accuracy of the model, and the resulting corpus which is, so to speak, a `Corpus`. All of its processing work is therefore made in the constructor, which takes the testing corpus as argument and confronts it to the `Dictionary`. All there is for the user to do is then to call `getAccuracy()` and `getCorpus()` to retrieve the results.

1.5.3 `Result extends ListNode<Result>`

This class is solely used by `Tester` and is a `ListNode` that only contains an integer value, which represents the number of words that have been correctly guessed by the `Dictionary`. The reason why this class is a specialisation of `ListNode` is that `Tester` actually loops through the entire `Queue` of `PartOfSpeech` and as such, it makes several loops, hence the need for several separate results.

1.5.4 `Timer`

This is, very simply put, a timer which is started using its constructor. The user can then use `lap()` to see the time elapsed since the last intermediate time (or lap), and the time elapsed since the start of the timer.

1.6 Class diagram

An UML class diagram can be found in SVG format, in the same folder as this report – it was not included directly in the document because of its size. I believe it to be a synthetic account of the descriptions of the classes given above, but there are a few differences:

- Only the inheritance and aggregation relations are depicted in the diagram. The association relations are far too numerous and eventually end up making the whole diagram difficult to read, which is why they were removed.
- Classes such as `AVLNode` and `ListNode` have been depicted as non-generic. This is also to simplify the diagram since a thorough analysis has already been given in this very document, and putting for example `AVLNode` instead of `T` means, I believe, *roughly* the same thing.

2 Unfolding and complexity

2.1 Loading the corpora

In a sequential application, there is not much we can do about this step. As the corpora need to be read from the beginning to the end, and then written word by word into the `Corpus`, I do not think there is much we can do to improve this process, which runs in $O(n)$.

2.2 Building the dictionary

Looping through the corpus runs in $O(n)$ and adding to the `Dictionary` runs in $O(\log n)$ since it is an AVL Tree. After that, all the `PartOfSpeech`s are sorted using merge sort, which runs in $O(n \log n)$. All in all, this step runs in $O(n \log n)$.

2.3 Testing the dictionary

In this step we have to loop through the testing corpus, which runs in $O(n)$. Writing the result `Corpus` runs in $O(n)$ as well. A lookup in the Dictionary, in the same way as an insert, also runs in $O(\log n)$ and looping through the `PartOfSpeech` also runs in $O(n)$, though the maximum length of the `Queue` is heuristically thought to be 4, which makes this factor rather unimportant – which is why I will ignore it. As such, this step runs in $O(n \log n)$.

2.4 Saving the output

All there is to do is to loop through the `Corpus` and save its content to a `String`, and then a file. This step therefore runs in $O(n)$.

3 Room for improvement

3.1 AVL Tree or Hash table?

Our `Dictionary` class here is an AVL Tree, which consistently runs in $O(\log n)$. But of course, a Hash table could also have been considered, since both its put and get methods

run in constant time. But the problem is, it is difficult to make an efficient Hash table without knowing what we are going to put in it: a table that is too large results in waste of memory resources, and a table that is too small results in numerous collisions and eventually degradation into $O(n)$. That is why I personally chose the AVL Tree: a tree is more reliable in the sense that its access time is consistent whatever its size, and its space requirements of $O(n)$ are excellent.

3.2 Multithreading

Obviously, a corpus is made of sentences that have no bearing one on another, except their order, and as such, it is easy to consider processing them independently and as a result, at the same time. But truth is, I failed to achieve multithreading in this application, as I was progressively led to use more and more classes from Java's standard library and ended up giving up on that idea, since I chose to put priority on one of the rules which was to use only data structures I had implemented myself. But here is how I would have personally considered this project using parallel computing.

3.2.1 Loading the corpora

As stated before, a corpus can be described as a set of independent Sentences associated with their order in the original text – which is why `Corpus` was ever a hash table to begin with. This particular form, rather than an entire sequential structure such as `Queue` for an entire corpus, allowed for parallel processing of all sentences at the same time.

3.2.2 Building the dictionary

In the same manner, all sentences could be processed at the same time, and the `Dictionary` could be written at the same time, as long as `Dictionary.push()` was not called – hence its being synchronised. Arguably, a Hash table would have performed much better here, with the same shortcomings that have already been mentioned before.

3.2.3 Testing the dictionary

Here as well, every sentence can be processed at the same time, and since only `Dictionary.find()` is called, it means the `Dictionary` can be concurrently accessed without problems, which makes for a significant performance gain.

3.2.4 Saving the output

Alas, saving the result corpus can hardly be improved with multithreading – I do not believe this phase to possibly run any faster than $O(n)$.

Conclusion

Was my application a satisfactory answer to the problem posed by the project ? Whether I should believe it is, is rather uncertain. It has several shortcomings – to quote one, the very fact that the corpora are tokenised and stored in memory can have disastrous consequences such as memory overloads, not to mention that my failed attempt at multithreading this application makes it overall slow and resource-inefficient. But if we stay

in a sequential perspective, I believe I have done my best and there is not much I could have done to improve this application's performance.

All in all, this was a very instructive experience for me, as I tried as much as I could to overcome my limits and try to achieve something I had never done before – such as, for example, parallel computing. Of course, this is not the first full application I have written. But performance-wise, it may be the best I have ever written.

Contents

Sources

- Java concurrency (multi-threading) - tutorial: An all-around article about concurrency and Java, which gave me precious information about thread locks, concurrency, atomic actions, and much more.
- Java Singleton and Synchronization: A very informative thread on Stackoverflow which gave me a link to Bill Pugh's Singleton pattern.
- How Volatile in Java Works ? Example of volatile keyword in Java: An article from a blog named Java revisited which focused on the keyword `volatile` and its role in thread locks.