Kotlin Search Engine

COMP40009 - Computing Practical 1

27th November – 1st December 2023

Aims

- To make use of different types of collection to manipulate data.
- To integrate with a 3rd party library.
- To handle errors gracefully.

Introduction

When Google was first building its search engine, the engineers designing it (notably Jeff Dean and Sanjay Ghemawat) made use of principles from functional programming to help them build a large scale system that could index the whole of the web. This was the basis for the web search feature that most of us now use many times per day.

Jeff and Sanjay designed a large-scale data processing infrastructure called MapReduce. This allows engineers to run computations across huge sets of data, by breaking those computations down into functions that process the data, either by mapping across it to transform each element, or reducing (folding) it into an aggregate value. In a MapReduce computation we typically take a set of data, map one function over it, and then reduce (fold) the results with a different function. For example if we wanted to compute the sum of the squares of a list of numbers, we could do:

```
fun square(x: Int): Int = x * x
fun sum(x: Int, y: Int): Int = x + y

val squares = listOf(1, 2, 3, 4, 5).map(::square)
val sumOfSquares = squares.reduce(::sum)
```

Google's MapReduce infrastructure is described in the paper "MapReduce: Simplified Data Processing on Large Clusters"[1] by Dean and Ghemawat, which you may find interesting. One of the examples that they give in the paper is an elementary way of using MapReduce to compute the web index. We will take inspiration from this to build our own (small) search engine.

Computing the Web Index

It would be too slow to explore the web every time someone searches for something, so to return search results quickly, search engines make use of a pre-computed *index*. This is a data structure that the search engine can use to find which pages it should refer us to for a given query. The index is recomputed periodically so that it is kept up to date with changes on the web.

[1] `https://research.google/pubs/pub62/`

The basic algorithm for computing the index is:

1. Download a copy of the internet (we'll come back to this point later).
2. Store this data as a collection of pairs of document-id (typically the URL) and the document itself (the content of the web page).
3. Map: For each URL/document pair, process the content of the page, and for every word on the page, emit a pair of (word, URL) - i.e. word X appears on page Y.
4. Shuffle: group the information in these pairs so that for each word, we have a list of all the pages that mention that word.
5. Reduce: sort each list to rank the results, most relevant first.
6. Store the index in a *map*, with the keys being words (potential search terms), and the values being the associated lists of URLs.

Once we have computed this data structure, serving a query is as simple as looking up the search term in the map and returning the results.

Getting started

As per the previous exercises, you can get your skeleton repository using:

`git clone https://gitlab.doc.ic.ac.uk/lab2324_autumn/kotlinsearchengine_`*`username`*`.git`

Open the code in IntelliJ to work on it. In this exercise we'll make use of a third party library, but the project you download should include this automatically.

What to do: Part 1

Defining some Types

Although map and reduce are constructs from functional programming, we can combine elements of functional and object-oriented programming in our solution to this problem. We will design a set of types that model the problem domain, to help us organise our code, to make it easy to understand, and to help prevent errors in the code that the type-checker can catch.

We will also make use of a third-party library called `jsoup`[2] to help us to work with web pages and HTML. `jsoup` is actually a Java library, but because both Java and Kotlin compile to Java bytecode, we can use the Java library in our Kotlin code without problems.

Create a new file called `Web.kt` and in this file define a class `URL`, constructed by passing in a string (the URL). Add a suitable `toString()` to your `URL` class. Then create a class `WebPage` constructed by passing in a jsoup `Document` (use `import org.jsoup.nodes.Document`) representing the content of the web page.

Create a new file `SearchEngine.kt` and in this file define a class `SearchEngine` which should be constructed by passing in a `Map` from `URL` to `WebPage`. This map will be the corpus of downloaded webpages that we will pass in to the `SearchEngine` for it to index.

In the same file, define a class `SearchResult`, which will be what we finally return from a search query. A `SearchResult` should contain a URL and the number of times the page at that URL mentions the search term. We'll calculate this later.

[2] `https://jsoup.org/`

Define `index` as a private property of `SearchEngine`, which is a map of strings to lists of `SearchResult`s. To begin with this can be initialised as an empty map.

Then define a class `SearchResultsSummary` which will encapsulate the results of a particular search to be returned to the user. We'll come back and complete this class later on.

Add signatures for two methods on `SearchEngine`:

- `compileIndex()` which when it is called, will process the downloaded webpages, compute the index, and store the results in the `index` map. As this method will mutate the state of the index itself, the function does not need to return anything.
- `searchFor(...)` which takes a query `String`, and returns a `SearchResultsSummary`.

You might want to leave the bodies of these methods as `TODO()` for now.

Processing HTML

Web pages are typically written using HTML[3], which includes a lot of formatting tags. For our index we just want to extract the text. We could write our own code to process HTML, but to save some work, we can use a third-party library to do this for us. We will use `jsoup` to work with the HTML *document* model in a convenient way.

You can find some examples of how to use `jsoup` and `Document` in the file `JsoupExamples.kt`.

In your `WebPage` class, implement a method `extractWords()` returning a `List<String>` containing all the words on the page. You can use the `Document.text()` method to get a string including all the text on a page without the tags. Process this string to separate it into individual words.

We won't discriminate on case when looking up search queries, so convert all the text on the page to lower case. You should also strip out commas and full stops.

Building the Index

Now complete the body of the `compileIndex()` method in `SearchEngine`. Follow the algorithm described above in *Computing the Web Index*. Take it step by step.

For each URL/WebPage pair in the corpus, extract the text from the web page, then for each word extracted, create a `Pair` showing that that word appears on the page at that URL.

Once you have a list of all the Word/URL reference pairs, to create the index map, first group the reference pairs by word, so you have a list of all the URLs of pages referencing a particular word.

Write a function `rank()` taking a list of URLs, and producing a list of `SearchResult`s, where each `SearchResult` includes a distinct URL, and the number of times that URL appears in the list.

Sort the results for each word with the page with the most mentions of the word first.

For example, applying `rank()` to the list `[ http://a.com/a, http://a.com/a, http://b.com/b, http://a.com/a ]` would produce a list of two `SearchResult`s, one for `http://a.com/a` with 3 references, and one for `http://b.com/b` with 1 reference.

Use your `rank()` function to process the URLs for each word, and assemble a map of `String` to `List<SearchResult>` which you can then assign to the `index` property.

Note that like with lists, maps provide various higher-order functions amongst their methods that you may find useful in transforming and organising the data. Have a look at the Kotlin documentation online[4] for more information on the functions available. You may particularly want to look at functions that allow data to be grouped.

[3] `https://developer.mozilla.org/en-US/docs/Web/HTML`

[4] `https://kotlinlang.org/docs/reference/collection-grouping.html`

Handling Queries

With the index in place, complete your `searchFor()` method. Take the query string, look it up in the map, and construct a `SearchResultsSummary`. Expand `SearchResultsSummary` so that it has properties for both the query and the list of results.

Think about what happens if there are no pages in the index matching your query. What's the best way to handle that?

Implement a suitable `toString()` for `SearchResultsSummary` so that the results are formatted nicely when they are printed. For example:

```
Results for "news":
  https://www.bbc.co.uk/news - 2 references
  https://www.imperial.ac.uk - 1 references
```

You should now be able to try some queries. There is some test data in `DownloadedPages.kt` that you can use to test your indexer. Write a `main()` function or a unit test to try out some queries.

Extension - Crawling

This part is optional, but do try it if you have time. Putting your `SearchEngine` together with this extension will give you a complete (if simple) system for indexing the web.

One important step in the original algorithm that we skipped over was "Download a copy of the internet". Obviously you won't be able to download the whole world wide web onto your computer, but we can download part of it. The way that this is usually done is using a *crawler*.

We give the crawler a URL to start from, it downloads that page, then it finds all the links on the page it has downloaded and follows them, repeating the process on every new page that it finds until either a) a preset number of pages has been downloaded, b) you run out of disk space, or c) you've downloaded the whole internet.

Building a Crawler

Write a simple crawler to download some web pages to feed into your search index.

Add a method `download()` to your `URL` class that returns a `WebPage`. You can see an example of how to download a document from the web using `jsoup` in `JsoupExamples.kt`.

Add a method `extractLinks()` to your `WebPage` class that returns a list of `URL`s that are linked to from that page. You can extract the *anchor tags* (`<a href="...">link text</a>`) from the HTML using `jsoup`. The URLs will be in the `href` attributes of the `<a>` tags. Again there are examples of doing this in `JsoupExamples.kt`. We can only download URLs that start with `http://` or `https://`. You may want to filter out any links that have different types of links (e.g. relative links to other pages on the same website), or find a different way to deal with these.

Create a class `WebCrawler` in a new file. `WebCrawler` should take a constructor argument of a `URL` from where the crawler should start exploring. You will also probably want to define a private property for the maximum number of pages to download (we'd suggest starting with 10 or 20).

Define a method `run()` in `WebCrawler` that runs the crawling process. Download the first URL, and save the `URL` and the `WebPage` in a map. Extract all the links from this page, and repeat the process for each of these URLs. You can skip pages that you have already downloaded to prevent processing the same page more than once (be careful of getting stuck in an infinite loop). Stop when you've downloaded enough pages according to the limit you set above.

Handling Download Errors

When we interact with other systems, and the internet, errors can occur that are beyond our control. Network connections may fail, servers may be down, or they may return us things that aren't well-formed HTML pages that we can parse.

In any of these cases, the third-party code we are using will likely throw an *exception*. If we don't deal with this, then our program will likely crash whenever this occurs.

Use a try/catch block to recover from exceptions being thrown. A useful thing to do in the crawler is to keep a record of URLs that caused errors when you tried to download them, and don't try them again – otherwise you might repeatedly try to download a page that doesn't work.

During development you might find it useful to log each URL as you try to download it, and whether it was successful or whether there was an error. You can do this just using `println()`. In general we try to remove this type of logging once the code is working as it can become verbose.

Putting It All Together

Add a method `dump()` to `WebCrawler` that returns a `Map<URL, WebPage>` containing all the data that you've downloaded. You can then use your crawler and your search engine together by doing something like:

```
fun main() {

    val crawler = WebCrawler(startFrom = URL("http://www.bbc.co.uk"))
    crawler.run()

    val searchEngine = SearchEngine(crawler.dump())
    searchEngine.compileIndex()

    println(searchEngine.searchFor("news"))
}
```

Try out your search engine by starting off your crawler in different places and seeing how well it works. Are there any further improvements you can make to either the crawling or indexing to improve the results?

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_autumn/kotlinsearchengine_username.* ***As always, you should use LabTS to test and submit your code.***

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```