Decision Management Community Challenge May 2022

Medical Claim Processing

(Bob Moore, JETset Business Consulting, 31st August 2022)

1 Problem Statement (from the web site)

Claim processing is one of the most popular area where rule engines demonstrate their power. In this challenge we ask you to build a decision service that deals with for a typical claim validation issue related to the International Statistical Classification of Diseases and Related Health Problems known as ICD-10.

This CSV file "ICD10Codes.csv" consists of two columns that represent different diagnoses:

Column 1,Column 2
A48.5,A05.1
K75.0,A06.4
K75.0,K83.09
K75.0,K75.1
G07,A06.6
G07,B43.1
…

The actual file contains around 70,000 code combinations. Your claim processing decision service receives a set of claim diagnoses such as {M43.6, F45.8} as its input. By comparing these codes with those found in Column 1 and Column 2, it should validate the claim by producing the errors "**Diagnose Code [XXX] cannot be reported together with [YYY]**" in the following situations:

1. Diagnose Code is found in Column 1 while another Diagnosis Code found in Column 2
2. Diagnose Code is found in Column 2 while another Diagnosis Code found in Column 1.

Keep in mind that some diagnoses can be found in both columns, and your service should not produce duplicate errors.

Here are several test cases for your service to run:

1. D47.02,C94.32
2. E71.313,E72.3
3. R29.891, M43.6, F45.8
4. D75.81,C94.42

Please submit your solutions using your favorite BR/DM tools. Your solutions should include the following information:

- Actually produced error messages for all test cases
- How your rules are represented
- How your data is represented
- Performance results.

2 Approach

Seven or so years ago, a junior colleague of mine phoned up with a problem. He had been asked to incorporate a post code risk factor into the insurance underwriting application he was working on. How he wondered was he going to build a decision table to map the post code to a risk factor (i.e., a small integer). There are after all around 1.8 million UK post codes. Most tools – particularly the one he was working with – struggle to work with tables of a few thousand rows, never mind one orders of magnitude bigger. Very big tables are more of a problem at development time than runtime, but if the development system can't cope, getting as far a runtime can be difficult.

I told him to forget about decision tables. A hash table would solve the problem. It would only a few hours to implement, have very good performance, and would almost certainly have a significantly smaller memory footprint than the equivalent decision table. There were a couple of minor wrinkles to add, but basically, he followed my advice and had a solution up and running in a fraction of the time the customer had expected.

A similar approach seems the 'correct' way to address this challenge. It seemed to me there was a way to solve the problem which would be quick to implement, economic on memory and probably provide performance at least as good, if not better than any alternative. So, I started writing up some Java code to solve the problem and had a fully working solution in about an hour or so. The write up has taken a lot longer than the actual solution,

Before we start though, let's analyse the problem. We have a lot of business data[1], but the logic we are applying is extremely simple. We have a list of diagnostic codes as our case data, and we need to work out if any two of the codes in the list appear as a pair in the file of ICD-10 codes provided. So basically, what we have to do is for every possible pair of codes in the case data we have to check if that pairing is on the listing in the file.

Sounds easy, but there are a couple of problems. In the first case there are a lot of pairs (just under 70,000 pairs), which is a challenge to even load into several tools[2]. Secondly neither the pairings in the file of code, or the codes in the test cases are ordered. So, for example a case where the codes are:

---

[1] My upcoming DecisionCamp 2022 presentation discusses what I mean by the term 'business data'

[2] The old .xls format of Excel can't cope with numbers this big for example. During the early stages of the pandemic, using this file format caused COVID cases in the UK to be considerably undercounted.

```
D47.02, C94.32
```

and one with the code are:

```
C94.32,R29.891, M43.6, F45.8,D47.02
```

Should both give rise to an error of the form:

```
Diagnosis Code  D47.02 cannot be reported together with  C94.32
```

even though the incompatible codes appear in different orders, and in the second case there are many intervening codes[3].

An early idea I had was that one might aggregate codes. For example, the code Z86 conflicts with 253 other codes. By having one line for each aggregate one could significantly reduce the number of input lines. But at the end of the day the information content is not reduced by this step. You still have to compare each pair and in either order. You can 'pretend' you are not doing this by using something like a 'contains' call, but under the hood, the code of your 'contains' function still performs the equivalent of pairwise comparisons.

I also thought that ensuring the IDC-10 codes were sorted in the case data might help. But then this would require sorting all the case data. If the examples are valid in that only a few codes are ever involved the overhead of a sort routine seems excessive most of the time[4].

But a straightforward and simple algorithm is obvious. For any case, I look at the first IDC-10 code and ask if it conflicts with the second code, or the third code, or the fourth until I have checked against all the codes in the case. Then I take the second IDC-10 code and check against the third code the fourth and so on. So, if a case has n codes, n*(n-1)/2 comparisons are made. This obviously scales quadratically with the size of a case. But if most of the time, there are only two or three codes this is no big deal (and whatever approach one takes, this number of comparisons are needed anyway, as discussed above). Note that this algorithm inherently avoids duplicated error messages unless the same code is duplicated in a case which would make no sense (see also section 3.4).

So, all that is needed is a data structure which can tell if a pair of codes conflict. One simple way to do this is to build a set data structure whose elements are strings formed by concatenating the strings representing the conflicting IDC-10 codes. The only little wrinkle is we need two entries for each pair, one with them one way around, one the other.

---

[3] The data is not pristine. I've not done a detailed analysis, but it does contain three entries which are not valid IDC-10 codes. There is one duplicated line. In a few cases I have found that there are pairs of lines where the codes are the same but in different orders, which is redundant. The solution is not sensitive to any of these issues, but it is a reminder that one should always look at the data set 😊

[4] Sorting in principle halves the memory footprint of the data structure used, but memory is cheap nowadays, so it is worth using up a few megabytes to avoid the extra processing needed by the sorting.

3 Implementation

The implementation is in Java but is very simplistic, any other language would do. The whole solution could be implemented in a single class. I've chosen to do it in three, simply to separate the three distinct parts of the problem solution. I've written things to run as a standalone command line program, but the logic would be easy to move to a web service implementation.

3.1 The LoadData Class

This class has a single method `readCSVFile` which takes a single parameter which is the filename of the csv containing the forbidden combinations of IDC-10 codes. It uses this to build a `forbiddenCombos` set which is what the method returns. The logic is simply to read each line from the file, split it, and concatenate the two IDC-10 codes it contains in both directions, to add two entries to the set. Note using a set implicitly avoids duplicate entries, so is robust with regard to duplicate rows in the file, and ones which repeat a previous pair, but with the order of codes reversed.

While the logic is geared to getting the information from a file, it is very simple and could be easily adapted to load the forbidden combinations from a URL, from a database, or from some other kind of representation, in order that the service could be hosted as a web service.

3.2 The EvaluateCase Class

This class has a single method, `checkCase`, this take a case identifier, an array of strings which are the IDC-10 codes of the case, and the `forbiddenCombos` set generated by the LoadData class. It simply checks every possible pair of codes in the case, to see which if any conflict, by see if the concatenation of the codes matches an entry in the `forbiddenCombos` set. For each matched entry a string is added to an array of error messages. Once all pairs are tested the array of errors is returned. If there are no errors the array is empty.

The logic has no external dependencies and is thread safe, so could easily be incorporated into a web service

3.3 The Runner Class

This class basically just runs the decision service. Using the `readCSVFile` method of `LoadData` it generates the `forbiddenCombos` set. Then it reads a number of test cases from a file. Each test case is parsed to generate an array of strings containing the list of IDC-10 codes, and these are passed to the `checkCase` method of `EvaluateCase` along with the case number (to link the errors to the relevant case) and the `forbiddenCombos` set. For each case, if conflicts are found a message for each conflict is written to a result file. Optionally a message can be written for cases which are valid.

3.4 Minor Limitations

There is limited error checking on the inputs. In particular if a case has multiple occurrences of the same IDC-10 code error messages might be duplicated.

The ordering the IDC-10 codes in the error message is dependent on the order the codes appear in the case, so the case:

```
D47.02, C94.32
```

generates the message:

```
Diagnosis Code  D47.02 cannot be reported together with  C94.32
```

while the case:

```
C94.32,R29.891, M43.6, F45.8,D47.02
```

generates the message:

```
Diagnosis Code  C94.32 cannot be reported together with  D47.02
```

note that unless codes are repeated, only one variant can ever appear.

Obviously fixing both problems is easy but gilding the lily.

4 Results

4.1 Initial Test Cases

After creating a text file comprising the given set of four test cases and calling the Runner class the following results were obtained:

```
Case Record     1: Diagnosis Code  D47.02 cannot be reported together with  C94.32
Case Record     2: Diagnosis Code E71.313 cannot be reported together with   E72.3
Case Record     3: Diagnosis Code R29.891 cannot be reported together with   M43.6
Case Record     3: Diagnosis Code R29.891 cannot be reported together with   F45.8
Case Record     3: Diagnosis Code   M43.6 cannot be reported together with   F45.8
Case Record     4: Diagnosis Code  D75.81 cannot be reported together with  C94.42
```

The first time of running, the timing output were as follows:

```
Time to process 4 cases is 16 ms (4040.40 µs/case).
Average case size is 2.25, number of cases with clashes is 4 (100.0%)
Overall Execution is 156 ms
```

The overall execution time includes loading the CSV file of conflicting diagnoses. Superficially this is disappointing. 4 milliseconds seems a long time to process a case. The numbers here are misleading though. We are running the code from a ‘cold start’, so the JIT optimiser has had no chance to tune the code. There is also some overhead associated with the file processing, in terms of opening and closing the files. A few tweaks and things get a lot better.

4.2 Extended Testing

To get a more realistic idea of performance, I built a simple program to generate large numbers of test cases. This generated a list of all the IDC-10 codes and then randomly selected a random number of these to create a case. I found in practise that these random lists rarely actually generated errors, so tweaked the code a little to (randomly) insert an invalid combination of codes into a case. With a test case file of 10,000 cases, each with a size of between 1 and 5 codes gives the following timing output:

```
Time to process 10000 cases is 66 ms (6.65 µs/case).
Average case size is 2.99, number of cases with clashes is 412 (4.1%)
Overall Execution is 211 ms
```

This is rather more what was expected, only a few microseconds to evaluate a case. Notice the execution time for processing the cases has only gone up by a factor of about 4, even though we are now considering 2500 times as many cases (and the average case size is larger). But the JIT optimiser is still not getting much of a chance

to do anything. So, I put in a warmup phase. This processes 1,000,000 cases (each of up to 10 diagnoses/case) before doing the timings. The improvement is considerable, re-running the original test cases we now get:

```
Time to process 1000000 cases is 2815 ms (2.82 µs/case).
Average case size is 5.49, number of cases with clashes is 45412 (4.5%)
Warm up and load is 2956 ms
Time to process 4 cases is 1 ms (371.65 µs/case).
Average case size is 2.25, number of cases with clashes is 4 (100.0%)
Overall Execution is 3 ms
```

Notice that now the data load and warm up are separated out, so the overall execution time excludes the load. Instead of taking ~ 6 milliseconds per case, when warmed up, the system only takes 0.4 milliseconds per case. A substantial amount of the time is still down to I/O setup (rather than I/O per case) as can be seen by the fact that the performance per case during warm up is ~ 140 times faster than the test case processing, even though the cases in the warmup are twice the size. Even after warming up the performance improves:

```
Time to process 1000000 cases is 2618 ms (2.62 µs/case).
Average case size is 5.49, number of cases with clashes is 45412 (4.5%)
Warm up and load is 2755 ms
Time to process 1000000 cases is 2351 ms (2.35 µs/case).
Average case size is 5.49, number of cases with clashes is 45412 (4.5%)
Overall Execution is 2350 ms
```

There is another 11% improvement in the warmup case a second time around.

5 Discussion

As with most of the DMC challenges, there is a degree of artificiality about what we are doing here. We consider a very small part of a complex process. However, it is a part that presents implementation challenges because of the vast amount of associated business data which is needed to make this particular sub-decision of an overall medical claim processing decision. The solution presents a simple, highly performant way of solving to the problem, which can be easily plugged in as a component part of an overall solution.

It levers off the fact that if you have a large chunk of business data as characterised by the IDC-10.csv file, then any business rules associated with it, are almost certainly going to both very simple and very few in number. No one person can really get their head around a list of 70,000 items, they can only hope to understand parts of it. If one added complex rules into the mix, it becomes unlikely anyone could make any sense at all of what was going on. For the challenge there is essentially only one rule (no conflicting pairs), though the mechanics of actually determining if the rule ‘fires’ are not entirely simple.

The challenge treats the IDC-10.csv file as a single piece of knowledge. But in practise the responsibility for defining and maintaining this list inevitably must be shared between a number of experts. And particularly in medical diagnosis, things never fall into neat silos. For example, it would be unwise to assume diagnoses normally related to oncologists don’t overlap/interact with those of cardiologists or neurologists. So, whoever makes up the team responsible for creating and maintaining the data will need

facilities to filter out and focus on the subsets of the data of immediate interest, and there is no simple *a priori* means of deciding what kind of filtering will be needed. Tools like relational databases and Excel are better geared to this kind of task than most decision system tools. This suggests that the 'source of truth' for the business data for this decision is mostly likely better held outside rather than inside the decision system tool in a more dedicated data management tool.

As a side note there are a couple of performance considerations which need to be considered. Firstly, the performance depends on the number of diagnoses in a case (and this is the case for any solution, not just this one). However, it is hard to imagine a claim running to a large number of distinct diagnoses, so this is unlikely to be a big issue. Secondly, the challenge does not indicate how the cases are processed. If the cases were processed one at a time, via a web service, high performance is simply not important. If the messaging overhead is tens of milliseconds. It matters little if it takes a millisecond or a microsecond for the internal processing of a case. On the other hand, if the cases are processed in an overnight batch, performance per case may become more important. Again, if the processing for the challenge is only one step in a much more complicated decision-making process, regardless of whether case processing is one at a time, or in batches, the performance of this task is becomes important only if it takes a significant amount of time relative to the other components of the decision.

To my mind the merits of this solution are less the run-time performance (however nice that might be) but the following:

- The approach can directly and efficiently consume the 'source of truth' (even if the IDC-10.csv file is derived from some other data set, or a database, it would be easy to hook up such a thing), so there is no dependency on any 'conversion' process to get the business data into the solution.

- All modern decision system tools should be able to 'plug-in' a solution like this, without the tool's development system being bogged down with what would be in effect a huge decision table – and one which ideally would be maintained by specialised tools – rather than the internal editors of the decision system tool.

6 Source Code

Below is the source code of the classes mentioned in section 3, and the test case generator

6.1 LoadData Class

```
package dmc202205;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;

/**
 * The logic use to load the list of conflicting diagnoses pairs and to convert
 * it into a set suitable for checking if a case contains any of them. See also
 * EvalutateCase for more details on how the data structure is utilised
 * structure used
 *
 * @author bob
 *
 */
public class LoadData {

	/**
	 * Method loads the conflicting diagnoses pairs from a CSV file and generates a
	 * set where each element is a string formed by concatenating the pairs
	 * together. Separate elements are created for both orderings of the pair, so
	 * that the order of diagnoses in a case is unimportant
	 *
	 * @param filename filename of the CSV file containing the conflicting diagnoses
	 *                 pairs
	 * @return a set whose elements represent the conflicting pairs
	 * @throws Exception
	 */
	public Set<String> readCSVFile(String filename) throws Exception {
		HashSet<String> forbiddenCombos = new HashSet<String>();
```

```
		Path path = Paths.get(filename);
		try (BufferedReader reader = Files.newBufferedReader(path)) {
			String line = reader.readLine();
			while (line != null) {
				String[] codes = line.replaceAll("\\s+", "").split(",");
				String first = codes[0];
				String second = codes[1];
				forbiddenCombos.add(first + "|" + second);
				forbiddenCombos.add(second + "|" + first);
				line = reader.readLine();
			}
			reader.close();
		} catch (FileNotFoundException e) {
			System.err.println("Forbidden combinations file " + filename + " could not be found");
		}
		return forbiddenCombos;
	}
}
```

6.2 EvaluateCase Class

```
package dmc202205;

import java.util.*;

/**
 * The logic to check if a case contains any conflicting diagnoses. See also
 * LoadData for more details on how conflicting pairs are build into the data
 * structure used
 *
 * @author bob
 *
 */
public class EvaluateCase {

	/**
	 * Checks if a given case contains any conflicting diagnoses. Using a nested
	 * loop, the logic generates all possible pairings of diagnoses in the case and
	 * check to see if the pairing occurs in the forbidden combinations set the set
	 * contains entries for both ordering of the pair, so only a single look up is
```

```
         * needed.
         *
         * @param caseNum         identifies case
         * @param caseData        array of strings, each a diagnosis code
         * @param forbiddenCombos set representing the conflicting diagnoses
         * @return
         */
        public ArrayList<String> checkCase(int caseNum, String[] caseData, Set<String> forbiddenCombos) {
               ArrayList<String> clashes = new ArrayList<String>();
               int length = caseData.length;
               // note if length <= 1, the loop does not execute
               for (int first = 0; first < length - 1; first++) {
                      String firstStr = caseData[first];
                      for (int second = first + 1; second < length; second++) {
                             String secondStr = caseData[second];
                             String key = firstStr + "|" + secondStr;
                             if (forbiddenCombos.contains(key)) {
                                    clashes.add(String.format(
                                                  "Case Record %5d: Diagnosis Code %7s cannot be reported together with %7s",
                                                                caseNum, firstStr, secondStr));
                             }
                      }
               }
               return clashes;
        }
}
```

6.3 Runner Class

```
package dmc202205;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Set;
```

```
/**
 * Simple class to read in a list of cases (lists of IDC-10 diagnoses) and to
 * determine if any of the diagnoses conflict with one another<br>
 *
 * For performance testing a 'warm-up' phase is incorporated
 *
 * @author bob
 *
 */
public class Runner {

	/**
	 * Main routine. Input arguments are:<br>
	 * 0. CSV file containing list of incompatible pairs of diagnoses<br>
	 * 1. CSV file containing list of test cases to be used in warm up<br>
	 * 2. CSV file containing list of cases to be evaluated<br>
	 * 3. Output file in which details of incompatible pairs are reported<br>
	 * 4. Flag to indicate if warm up required (presence not value is important)<br>
	 * 5. Flag to indicate if line are written for case where there are no issues
	 * (presence not value is important)<br>
	 *
	 * @param args
	 * @throws Exception
	 */
	public static void main(String[] args) throws Exception {
		if (args.length > 3) {
			// create worker object and start timing
			long time = System.currentTimeMillis();
			Runner runner = new Runner();

			// load the business data
			LoadData loadData = new LoadData();
			Set<String> forbiddenCombos = loadData.readCSVFile(args[0]);

			// warm up processing this is triggered if there is a fifth input parameter
			if (args.length > 4) {
				runner.processCases(forbiddenCombos, args[1], args[3], false);
				time = System.currentTimeMillis() - time;
				System.out.printf("Warm up and load is %d ms\n", time);
```

```
                time = System.currentTimeMillis(); // re-init time if do warm up
            }
            // real processing
            runner.processCases(forbiddenCombos, args[2], args[3], args.length > 5 ? true : false);
            time = System.currentTimeMillis() - time;
            System.out.printf("Overall Execution is %d ms\n", time);
      } else {
            System.err.println("Please provide combo file and warmup file and case file");
      }
}

/**
 * The workhorse - it reads each case from the case file, an uses an
 * EvaluateCase instance to check for clashes writing the outcomes to a file
 *
 * @param forbiddenCombos set of forbidden pairings
 * @param caseFilename    file containing cases to be evaluated
 * @param resultFilename  file to write results to
 * @param verbose         flag indicating if cases with no errors are written
 *                        out
 * @throws Exception
 */
private void processCases(Set<String> forbiddenCombos, String caseFilename, String resultFilename, boolean verbose)
            throws Exception {
      // initialise statics and timer - use nano time as milliseconds too coarse
      long time = System.nanoTime();
      int caseNum = 0;
      int codeTotal = 0;
      int errorCases = 0;
      EvaluateCase evaluateCase = new EvaluateCase();

      // open input and output files
      try (BufferedReader reader = new BufferedReader(new FileReader(caseFilename))) {
            try (PrintWriter writer = new PrintWriter(resultFilename)) {
                  // Read each case from the input line, remove any white space and split it
                  // Then pass the data to the evaluateCase instance
                  String line = reader.readLine();
                  while (line != null) {
                        caseNum++;
```

```
                        String[] codes = line.replaceAll("\\s+", "").split(",");
                        codeTotal += codes.length;
                        ArrayList<String> clashes = evaluateCase.checkCase(caseNum, codes, forbiddenCombos);

                        // report any clashes, or report case validated
                        if (clashes.size() > 0) {
                              errorCases++;
                              for (String clash : clashes) {
                                    writer.println(clash);
                              }
                        } else if (verbose) {
                              writer.println(String.format("Case Record %5d: validated successfully", caseNum));
                        }
                        line = reader.readLine();
                  }
            } catch (Exception e) {
                  e.printStackTrace();
                  System.err.println("Error encountered when writing " + resultFilename);
            }
      } catch (FileNotFoundException e) {
            System.err.println("Case file " + caseFilename + " could not be found");
      }

      // print out statistics of processing
      time = System.nanoTime() - time;
      System.out.printf("Time to process %d cases is %d ms (%4.2f µs/case). \n", caseNum, time / 1000000,
                  time / 1000.0 / caseNum);
      System.out.printf("Average case size is %4.2f, number of cases with clashes is %d (%.1f%%)\n",
                  (double) codeTotal / caseNum, errorCases, (100.0 * errorCases / caseNum));
   }

}
```

6.4 GenTest Class

```
package dmc202205;

import java.io.BufferedReader;
```

```
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;

/**
 * Test class to generate larger numbers of test cases
 *
 * @author bob
 *
 */
public class GenTest {

	private HashMap<String, ArrayList<String>> clashes = new HashMap<>();
	private Random random = new Random();
	final private static double THRESHOLD = 0.1;
	final private static int CASE_SIZE = 10;

	/**
	 * Main routine. Input arguments are:<br>
	 * 0. CSV file containing list of incompatible pairs of diagnoses<br>
	 * 2. CSV file to which test cases are written<br>
	 * 3. Number of cases to generate<br>
	 * 4. Maximum test case size (on average test cases are half this size)<br>
	 * 5. Factor giving proportion of conflicting cases to be added (in range 0 to 1)<br>
	 *
	 * @param args
	 * @throws Exception
	 */
	public static void main(String[] args) throws Exception {
		if (args.length > 2) {
			String forbiddenComboFile = args[0];
			String testCaseFile = args[1];
			int numberCases = Integer.parseInt(args[2]);
			int caseSize = args.length > 3 ? Integer.parseInt(args[3]) : CASE_SIZE;
			double threshold = args.length > 4 ? Double.parseDouble(args[4]) : THRESHOLD;
			(new GenTest()).process(forbiddenComboFile, testCaseFile, numberCases, caseSize, threshold);
```

```
            System.out.println(
                        String.format("Created %d testcases of maximum size %d in file %s force error threshold %f",
                                    numberCases, caseSize, testCaseFile, threshold));
      } else {
            System.err.println("Please provide combo file, output file, number cases, case size");
      }
}

/**
 * Load IDC codes and use them to generate random test files
 *
 * @param forbiddenComboFile CSV file containing list of incompatible pairs of
 *                           diagnoses
 * @param testCaseFile       CSV file to which test cases are written
 * @param numberCases        Number of cases to generate
 * @param caseSize           Maximum test case size (on average test cases are
 *                           half this size)
 * @param threshold          Factor giving proportion of conflicting cases to be
 *                           added (in range 0 to 1)
 * @throws Exception
 */
private void process(String forbiddenComboFile, String testCaseFile, int numberCases, int caseSize,
            double threshold) throws Exception {
      ArrayList<String> allCodes = readCSVFile(forbiddenComboFile);
      genearateCases(allCodes, testCaseFile, numberCases, caseSize, threshold);
}

/**
 * Generate and write out cases
 *
 * @param allCodes    all the distinct IDC-10 codes in the list of incompatible
 *                    pairs of diagnoses
 * @param filename    testCaseFile CSV file to which test cases are written
 * @param numberCases Number of cases to generate
 * @param caseSize    Maximum test case size (on average test cases are half
 *                    this size)
 * @param threshold   Factor giving proportion of conflicting cases to be added
 *                    (in range 0 to 1)
 */
```

```
private void genearateCases(ArrayList<String> allCodes, String filename, int numberCases, int caseSize,
		double threshold) {
	try (FileWriter writer = new FileWriter(filename)) {
		for (int count = 0; count < numberCases; count++) {
			String line = genOneCase(allCodes, caseSize, threshold);
			writer.write(line);
		}
	} catch (Exception e) {
		e.printStackTrace();
		System.err.println("Error encountered when writing " + filename);
	}
}

/**
 * Generate one test case. The logic is as follows:<br>
 * 1. Pick a random length between 1 and the maximum size<br>
 * 2. Loop picking random codes from the list of all codes, and adding them to
 * the test case if they are not already there<br>
 * 3. Additionally in the loop a conflicting case is added based on the
 * threshold<br>
 * 4. After enough cases are found, the cases are concatenated into a comma
 * separated string with a new line
 *
 * @param allCodes  all the distinct IDC-10 codes in the list of incompatible
 *                  pairs of diagnoses
 * @param caseSize  Maximum test case size (on average test cases are half this
 *                  size)
 * @param threshold Factor giving proportion of conflicting cases to be added
 *                  (in range 0 to 1)
 * @return test case as comma separated string with newline
 */
private String genOneCase(ArrayList<String> allCodes, int caseSize, double threshold) {
	int caseLen = random.nextInt(caseSize) + 1;
	int codesLen = allCodes.size();
	HashSet<String> caseCodes = new HashSet<String>();
	while (caseCodes.size() < caseLen) {
		String newCode = allCodes.get(random.nextInt(codesLen));
		caseCodes.add(newCode);
		ArrayList<String> clashesWith = clashes.get(newCode);
```

```
			if (caseCodes.size() < caseLen && random.nextDouble() < threshold && clashesWith != null) {
				caseCodes.add(clashes.get(newCode).get(0));
			}
		}
		String result = "";
		for (String code : caseCodes) {
			result += code + ",";
		}
		return result.substring(0, result.length()) + "\n";
	}

	/**
	 * This loads incompatible pairs, into a set, and a hash map. The first is then
	 * converted into a list of distinct IDC-10 codes. The HashMap maps between a
	 * code and it's incompatible pairs to allow a case to be seeded with
	 * incompatibilities. Note that for any code, there is always at least one
	 * incompatible pairing
	 *
	 * @param filename CSV file containing list of incompatible pairs of diagnoses
	 * @return
	 * @throws Exception
	 */
	private ArrayList<String> readCSVFile(String filename) throws Exception {
		HashSet<String> allCodesAsSet = new HashSet<String>();
		Path path = Paths.get(filename);
		try (BufferedReader reader = Files.newBufferedReader(path)) {
			String line = reader.readLine();
			while (line != null) {
				String[] codes = line.replaceAll("\\s+", "").split(",");
				String first = codes[0].trim();
				String second = codes[1].trim();

				// add to set so only get unique codes
				allCodesAsSet.add(first);
				allCodesAsSet.add(second);
				ArrayList<String> clashesWithFirst = clashes.getOrDefault(first, new ArrayList<String>());
				clashesWithFirst.add(second);
				clashes.put(first, clashesWithFirst);
				ArrayList<String> clashesWithSecond = clashes.getOrDefault(second, new ArrayList<String>());
```

```
                        clashesWithSecond.add(first);
                        clashes.put(second, clashesWithSecond);
                        line = reader.readLine();
                }
                reader.close();
        } catch (FileNotFoundException e) {
                System.err.println("Forbidden combinations file " + filename + " could not be found");
        }

        // convert set to string - the ordering is irrelevant since elements are picked
        // at random
        return new ArrayList<String>(allCodesAsSet);
    }
}
```