Assignment 2: TopicMapping

version 1.1

Mark Dras

June 5, 2016

(This is a release version, but a changelog will still list changes in Section 5.)

1 Introduction

When searching for a document on the web, you typically type in some terms (e.g. "heapsort complexity" or "cats dancing") and your search engine will return documents (or videos with captions or ...) that match those terms. To get documents on the same theme or topic, but with different terms (e.g. "quicksort analysis" or "cats falling off tables"), you typically have to generate a number of query terms yourself and try them all.

Something that researchers at Google and elsewhere are interested in is being able to characterise documents by themes or topics, where these themes or topics aren't listed anywhere, but can be inferred from analysing vast numbers of documents. One particular document might be a mix of a *data analysis* topic (seen through use of words such as "computer" or "prediction"), an *evolutionary biology* topic (via words such as "life" and "organism") and a *genetics* topic ("sequenced" and "genes"). A nice overview of this idea, and one particular approach to inferring the topics, has been written by David Blei.[1] An illustration from that overview is in Figure 1. A document can be thought of a statistical distribution over such topics: it's e.g. 50% about data analysis, 30% about evolutionary biology and 20% about genetics.

A recent approach to inferring these sorts of topics is called TopicMapping.[2] The idea here is to represent the relationships between words by a graph, and then use existing network community structure detection algorithms[3] to find relationships. Network community structure detection algorithms find vertices in a network graph that are strongly related to each other.[4] For this problem, the TopicMapping developers use a community structure detection algorithm — specifically, InfoMap[5] — to find out which words naturally group together (like "computer" and "prediction", or "life" and "organism") and infer topics from this.

[1] `https://www.cs.princeton.edu/~blei/papers/Blei2012.pdf` You don't have to read this for the assignment, but it is neat.

[2] `https://journals.aps.org/prx/pdf/10.1103/PhysRevX.5.011007`. You don't have to read this for the assignment.

[3] `https://en.wikipedia.org/wiki/Community_structure`

[4] In a social network graph, followers of the band Chvrches might constitute such a community, or IT workers in Sydney another such community.

[5] `http://www.mapequation.org/`. You don't have to look at this for the assignment, but it does have some nice illustrations of what network community algorithms can be used for.

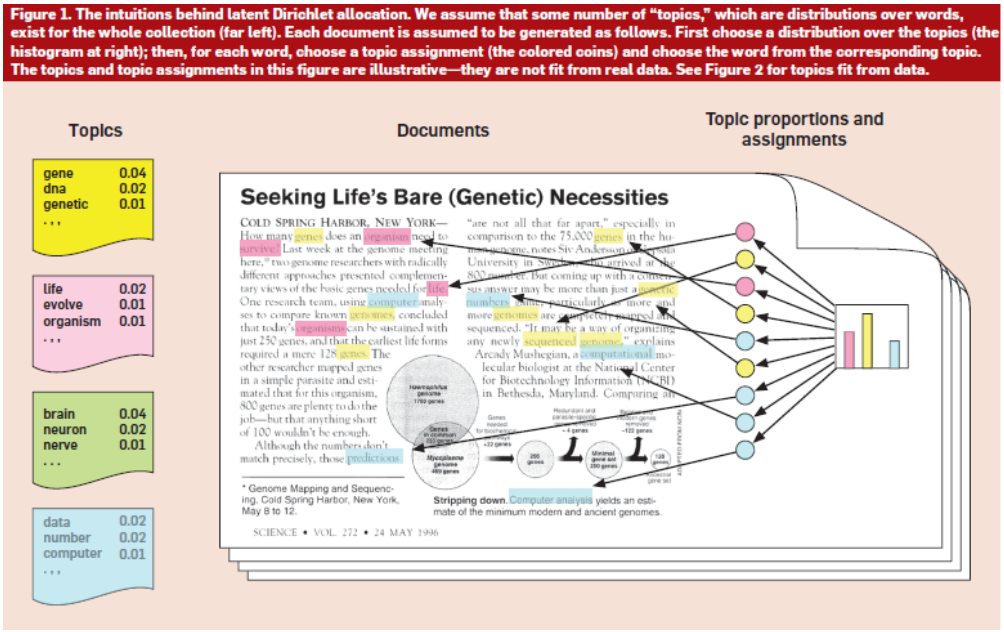

Figure 1: Figure from Blei overview

So the first step in this kind of process is representing the co-occurrence of words in a set of documents as a graph, which can then be passed to some algorithm for detecting community structure. You'll be doing a simplified version of this first step, constructing the graph and making some calculations over it, for this assignment.

2 What's Involved

There are two stages in the graph construction process.[6]

2.1 The Bipartite Graph

The first stage is to construct an undirected bipartite graph.[7] Of the two sets of vertices making up the graph, one will represent documents and the other will represent words. There will be an edge between vertices if and only if the word represented by the vertex at one end of the edge is contained in the document represented by the vertex at the other end of the edge. Consider a very small example with three documents `d1`, `d2`, `d3`.

```
d1: the purple and green cat was angry cat
d2: i heard that purple and green cat is angry and green
d3: all cat are my best friend
```

[6] More detail is contained in Section S2 of the supplementary material to the paper that defined TopicMapping, `http://journals.aps.org/prx/supplemental/10.1103/PhysRevX.5.011007/SI.pdf`. The only people who might possibly want to look at this are people attempting the Distinction part. Even then, don't look at it too soon.

[7] `https://en.wikipedia.org/wiki/Bipartite_graph`

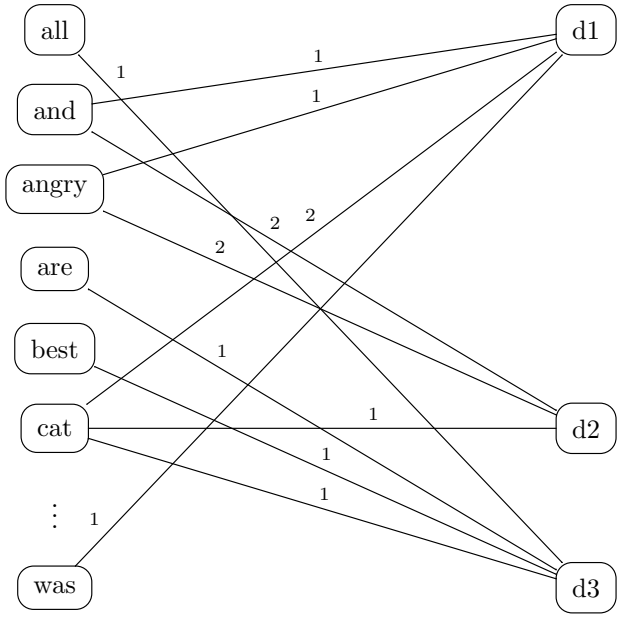

Figure 2: Bipartite graph representing the three example documents and the words in them.

The bipartite graph that would represent this is as in Figure 2. The weight on the edge between document d and word a, ω_a^d, is the number of times a occurs in d.

2.2 The Unipartite Graph

A unipartite graph is then derived from the bipartite graph. The vertices are words, and the edges will represent a measure of similarity between words, based on how often words co-occur in documents. So two words that are very closely related will have a high weight on the edge between them, and this is what determines the community structure in the process that follows (which isn't part of this assignment).

The unipartite graph that would represent this (omitting most of its edges, to maintain readability) is as in Figure 3.

For the assignment we're going to define a number of different unipartite graphs, with the weights varying. These will be defined in the Your Tasks section below.

3 Your Tasks

For your tasks, you'll be adding attributes and methods to existing classes given in the code bundle accompanying these specs. Where it's given, **you should use exactly the method stub provided** for implementing your tasks. Don't change the names or the parameters. You can add more functions if you like.

The input data will come in the form of directories consisting of a number of documents in the form of text files. We'll refer to all the documents in a directory as a *corpus*.

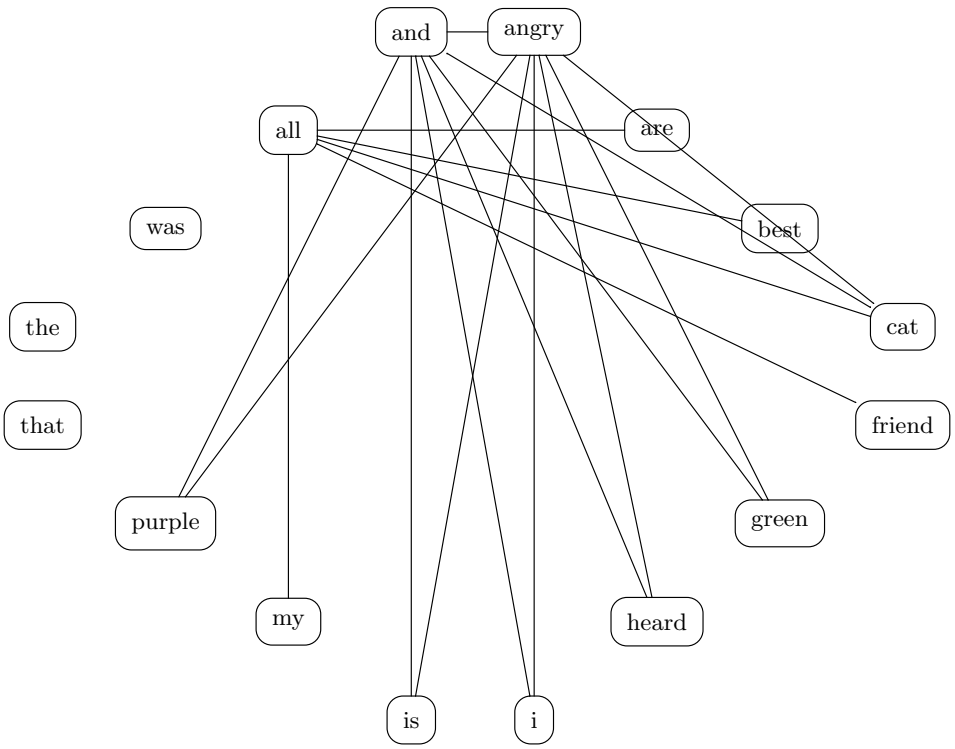

Figure 3: Partial (unweighted) unipartite graph of all words in the three example documents. All vertices are present, but I've only included a few of the edges (otherwise it's unreadable).

3.1 Pass Level

To achieve at least a Pass (50–64%) for the assignment, you should do all of the following. You'll be defining the bipartite graph, and functions associated with that. To do this, you will modify the class defined in `BipartiteGraph.java`.

For this section, you can test your code on the data contained in the zipped directory `in1.zip`, which contains the graph from the example above.

T1 You will choose the representation for your graph. You may or may not choose to use other classes I've supplied (`Vertex`, `VertexIDList`). Material from weeks 9–11 of lectures will be particularly relevant in helping you decide.

You'll need to write a constructor based on your chosen representation that instantiates an empty graph.

```
public BipartiteGraph() {
// constructor

// TODO
}
```

T2 I've supplied a function `readFromDirectory(String dirInName)` that will read in the content of all the files from a directory `dirInName` and store this in a `Vector` of `Edge`s. For this graph the edges are pairs of words and documents; for the running example, you would have

the following edges:

```
the         d1
purple      d1
and         d1
...         ...
i           d2
heard       d2
...         ...
all         d3
cat         d3
...         ...
```

You need to write a function to set the vertices and edges of the graph based on your chosen graph representation.

```
public void setGraph(Vector<Edge> eList) {
// PRE: -
// POST: instantiates a bipartite graph based on a list of edges;
//       edges represent word-document pairs

// TODO
}
```

T3 You now need to write the following functions on your graph. Values for the graph from the sample input are in the JUnit test file `AllTest.java`. The graph has 16 word vertices and 3 document vertices (for 19 in total), and there are 22 edges. As an example of an edge weight, $\omega_{\text{cat}}^{\text{d1}}$, the number of times the word "cat" appears in document d_1, is 2; this is the value returned by `numTimesWordOccursInDoc('cat','d1')`.

```
public Integer numVertices() {
// PRE: -
// POST: returns number of vertices (both word and doc) in the graph

// TODO
}
```

```
public Integer numEdges() {
// PRE: -
// POST: returns number of edges in the graph

// TODO
}
```

We'll be distinguishing between WORD TYPES (where you ignore duplicates — the example texts' word type count is 16) and WORD TOKENS (where you count duplicates — the example texts' word token count is 25).

```
public Integer numWordTypes() {
// PRE: -
// POST: returns number of word types in the graph (i.e. ignoring duplicate words)
```

```
// TODO
}

public Integer numWordTokens() {
// PRE: -
// POST: returns number of word tokens in the graph (i.e. including duplicate words)

// TODO
}

public Integer numTimesWordOccurs(String w) {
// PRE: -
// POST: returns the number of times word w occurs across all documents

// TODO
}

public Integer numWordTypesInDoc(String d) {
// PRE: d corresponds to a vertex in the graph
// POST: returns number of different words (i.e. word types) that occur in document d

// TODO
}

public Integer numWordTokensInDoc(String d) {
// PRE: d corresponds to a vertex in the graph
// POST: returns number of total words (i.e. including duplicates)
//         that occur in document d

// TODO
}

public Integer numDocsWordOccursIn(String w) {
// PRE: w corresponds to a vertex in the graph
// POST: returns number of documents word w occurs in

// TODO
}

public Integer numTimesWordOccursInDoc(String w, String d) {
// PRE: d corresponds to a vertex in the graph
// POST: returns number of times word w occurs in doc d

// TODO
}
```

```
public ArrayList<String> listOfWordsInSingleDocs() {
// PRE: -
// POST: returns the words that only occur in single documents
//       the returned list should be ordered alphabetically
//         and should contain no duplicates

// TODO
}

public Double propnOfWordsInSingleDocs() {
// PRE: -
// POST: returns the words (out of all word types) that only occur in single documents

// TODO
}

public String mostFreqWordToken() {
// PRE: -
// POST: returns the most frequent word
//       if there is more than one with maximum frequency,
//         return the first alphabetically

// TODO
}

public String wordInMostDocs() {
// PRE: -
// POST: returns the word that occurs in the largest number of documents
//       if there is more than one with maximum frequency,
//         return the first alphabetically

// TODO
}
```

3.2 Credit Level

To achieve at least a Credit (65–74%) for the assignment, you should do the following. You should also have completed all the Pass-level tasks.

Essentially, you will be building the basis of the unipartite graph class, and defining the key functions. As noted above, the vertices of the graph will correspond to the words of the corpus; there will be an edge between them if they co-occur in the same document. For the Credit level, the initial weight on the edge between (word) vertex a and (word) vertex b, $z_{a,b}$, is defined as follows:

$$z_{a,b} = \sum_{d} \omega_a^d \times \omega_b^d$$

So words that occur frequently together will have higher weights on the edges between them.

As an example, for the unipartite graph from the sample data, there will be 16 vertices, as the total number of words used in the corpus across all documents is 16. $\omega_{\text{cat}}^{\text{d1}} = 2$ and $\omega_{\text{the}}^{\text{d1}} = 1$; $z_{\text{cat,the}}$ is obtained by taking the product of these, and the equivalent product for the other two documents d_2 and d_3, and summing them all. (As "the" doesn't occur in the other two documents — so $\omega_{\text{the}}^{\text{d2}} = 0$ and $\omega_{\text{the}}^{\text{d3}} = 0$ — the weight $z_{\text{cat,the}} = 2$.)

The overall graph will look as follows. (The format is `word vertex (degree) : adjacency list(weight)`.)

```
Number of word nodes is 16
Number of edges is 62
Word graph:
all (5) : are(1.0) best(1.0) cat(1.0) friend(1.0) my(1.0)
and (10) : angry(3.0) cat(4.0) green(5.0) heard(2.0) i(2.0) is(2.0) purple(3.0) that(2.0) the(1.0) was(1.0)
angry (10) : and(3.0) cat(3.0) green(3.0) heard(1.0) i(1.0) is(1.0) purple(2.0) that(1.0) the(1.0) was(1.0)
are (5) : all(1.0) best(1.0) cat(1.0) friend(1.0) my(1.0)
best (5) : all(1.0) are(1.0) cat(1.0) friend(1.0) my(1.0)
cat (15) : all(1.0) and(4.0) angry(3.0) are(1.0) best(1.0) friend(1.0) green(4.0) heard(1.0) i(1.0) is(1.0)
       my(1.0) purple(3.0) that(1.0) the(2.0) was(2.0)
friend (5) : all(1.0) are(1.0) best(1.0) cat(1.0) my(1.0)
green (10) : and(5.0) angry(3.0) cat(4.0) heard(2.0) i(2.0) is(2.0) purple(3.0) that(2.0) the(1.0) was(1.0)
heard (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) i(1.0) is(1.0) purple(1.0) that(1.0)
i (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) is(1.0) purple(1.0) that(1.0)
is (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) i(1.0) purple(1.0) that(1.0)
my (5) : all(1.0) are(1.0) best(1.0) cat(1.0) friend(1.0)
purple (10) : and(3.0) angry(2.0) cat(3.0) green(3.0) heard(1.0) i(1.0) is(1.0) that(1.0) the(1.0) was(1.0)
that (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) i(1.0) is(1.0) purple(1.0)
the (6) : and(1.0) angry(1.0) cat(2.0) green(1.0) purple(1.0) was(1.0)
was (6) : and(1.0) angry(1.0) cat(2.0) green(1.0) purple(1.0) the(1.0)
```

T4 You will again choose the representation for your graph, and will need to write a constructor based on your chosen representation that instantiates an empty graph.

```
public UnipartiteGraph() {
// constructor

// TODO
}
```

T5 You'll need to set a graph from a bipartite graph as defined in the Pass level. Edge weights are the initial weights $z_{a,b}$.

```
public void setGraph(BipartiteGraph bpg) {
// PRE: -
// POST: instantiates the unipartite graph based on a bipartite graph of words and docs;
//       creates vertices and edges, with edge weights the word similarity z_ab

// TODO
}
```

T6 You'll need to define the following functions, most of which correspond to similar functions for the bipartite graph.

```
public Integer numVertices() {
// PRE: -
// POST: returns number of vertices in the graph

// TODO
}

public Integer numEdges() {
// PRE: -
// POST: returns number of edges in the graph

// TODO
}

public Integer degreeWord(String w) {
// PRE: w corresponds to a vertex in the graph
// POST: returns the degree of the vertex corresponding to word w

// TODO
}

public Integer numVerticesDegreeK(Integer k) {
// PRE: k >= 0
// POST: returns the number of vertices that have degree k

// TODO
}

public Integer numVerticesConnectedAll() {
// PRE: -
// POST: returns the number of vertices that are connected to all other vertices

// TODO
}

public Double avgDegreeOfGraph() {
// PRE: -
// POST: returns the average degree of all vertices in the graph;
//       returns zero for the null graph

// TODO
}

public Double propnPossibleEdges() {
// PRE: -
// POST: returns the proportion of actual edges to maximum possible edges;
//       returns zero for a null graph or single-vertex graph

// TODO
}
```

```
public Double getEdgeWeight(String a, String b) {
// PRE: a, b correspond to vertices in the graph
// POST: returns the weight of the edge between vertices corresponding to words a and b

// TODO
}

public void setEdgeWeight(String a, String b, Double val) {
// PRE: a, b correspond to vertices in the graph
// POST: sets the weight of the edge between vertices
// corresponding to words a and b to be val

// TODO
}
```

3.3 (High) Distinction Level

To achieve at least a Distinction (75–100%) for the assignment, you should do the following. You should also have completed all the Credit-level tasks.

The unipartite graph defined in the Credit level could in principle be used for finding topics or themes, being passed straight to the network community structure detection algorithm. However, there are a few issues with it as it stands.

Problem 1 Relatively uninteresting generic words like "to" or "of", which have high frequencies, will be strongly connected to more specific words. It's really only these specific words (like "organism" or "genes") that we want to detect relations between.

Problem 2 It ignores that some associations would happen just by chance.

Problem 3 For realistically sized applications, the graph is too dense: community detection structure algorithms typically won't work well on such a graph.

In this level you'll be changing the weights on the graph to solve these problems.

T7 A simple way to solve Problem 1, and partially Problem 2, is to define a STOPLIST of words that you've decided in advance are uninteresting, and then leave these out when constructing the graph.

In our running example, if the stoplist just contained the words "the" and "and", the graph would have 14 vertices.

To do this, define the following function.

```
public void setGraphMinusStopwords(BipartiteGraph bpg, String stopFileName) {
// PRE: stopFileName is the name of a file containing stopwords
// POST: as the regular setGraph(), but stopwords do not become vertices

// TODO
}
```

T8 An alternative, more sophisticated approach is to consider what edge weights you might get just by chance association of words in documents, and then evaluate which edges have weights greater than the values you'd expect by chance.

The first step in this is to define what the creators of TopicMapping call a NULL MODEL. We'll use the following definitions for the null model:

- L_d, the total number of words in document d;
- $s_a = \sum_d \omega_a^d$, the number of occurrences of word a in the whole corpus; and
- $L_C = \sum_d L_d$, the total number of words in the corpus.

In our running example, $L_{\mathrm{d1}} = 8$, $s_{\mathrm{cat}} = 4$ and $L_C = 25$.

Under some assumptions about statistical distributions, the number of times we'd expect word a to occur randomly in document d (that is, the mean of ω_a^d under the null model) is

$$\langle \omega_a^d \rangle = \frac{L_d \times s_a}{L_C}$$

You can see this just allocates the number of times a occurs in the whole corpus proportionately depending on how big document d is. Making a few assumptions, we can estimate how many times we'd expect word a to co-occur with word b just by chance:

$$\langle z_{a,b} \rangle = \sum_d \langle \omega_a^d \rangle \times \langle \omega_b^d \rangle = \frac{s_a s_b}{L_C^2} \sum_d L_d^2$$

From our example, recall that $s_{\mathrm{cat}} = 4$ and $L_C = 25$; also, $s_{\mathrm{the}} = 1$ and $\sum_d L_d^2 = L_{\mathrm{d1}}^2 + L_{\mathrm{d2}}^2 + L_{\mathrm{d3}}^2 = 221$. Then $\langle z_{\mathrm{cat,the}} \rangle = 1.414$.

Now define the following function.

```
public void setNullGraph(BipartiteGraph bpg) {
// PRE: -
// POST: as the regular setGraph(),
//         but sets the edge weights to be the values under the null model
//         <z_ab> = s_a * s_b / L_C^2 * \sum_d L_d^2

// TODO
}
```

T9 Now it turns out that, as a result of the model developed by the TopicMapping guys, we can work out how likely particular values are. Is an edge weight of, say, 5.2 likely between "cat" and "the", and specifically, what's its actual probability?

Under the null model this is given by a Poisson distribution.[8] The Poisson distribution takes one parameter λ,[9] and you can use it to determine the probability p of a value of at least x. For us λ will just be the expected number of co-occurrences of words a and b (i.e. $\langle z_{a,b} \rangle$).

[8] `https://en.wikipedia.org/wiki/Poisson_distribution`

[9] The Gaussian or normal distribution, which you'd be more familiar with, is defined by two parameters, its mean μ and its standard deviation σ. The Poisson just has this one.

Before you read on: To understand in more detail how the Poisson distribution is used to model events and calculate their probability, look at the example in the Appendix.

The value you want for this part of the assignment is the inverse of the Poisson: given a Poisson with parameter $\lambda = \langle z_{a,b} \rangle$ and a probability p, what's the corresponding value of x? Note that since the Poisson is a distribution over integers (so we can't get a real number that will give us exactly p), what we want is the value of x that gets as close as possible to putting us within the top p without actually going over. We'll refer to this value as $Z_p(s_a, s_b)$; more precisely

$$Z_p(s_a, s_b) = \max_x \{x \text{ such that: } \sum_{z=x}^{\infty} \text{Pois}_{\langle z_{a,b} \rangle}(z) > p\}.$$

Fortunately, there's an inverse Poisson probability function in the Apache Commons Maths library[10] that gives you just this value. You'll want to download this and include it as an external JAR file.

Continuing the running example, recall that $s_{\text{cat}} = 4$, $L_C = 25$ and $\sum_d L_d^2 = 221$; also, $s_{\text{and}} = 3$. Then $\langle z_{\text{cat,and}} \rangle = 4.2432$. If we choose $p = 0.05$, what we're interested in is the integer value $Z_{0.05}(\text{cat}, \text{and})$ such that it puts us as close as possible to the top 5% of values of the Poisson distribution with $\lambda = 4.2432$; this is $Z_{0.05}(\text{cat}, \text{and}) = 8$.[11] If we choose $p = 0.95$, $Z_{0.95}(\text{cat}, \text{and}) = 1$.

We'll filter the graph by assigning the weight $z_{a,b} - Z_p(s_a, s_b)$ to an edge if positive, or zero (deleting the edge) if not.

Now define the following function:

```
public void filterNoise(UnipartiteGraph nullGraph, Double p) {
// PRE: 0 <= p <= 1
// POST: reduce values on edges taking into account the null model graph;
//         calculates Z_p(s_a, s_b) as per specs: the value which represents the
//         (1-p)-quantile of the Poisson distribution Pois_<z_ab>(z)

// TODO
}
```

Applying this function to the graph in the running example with probability $p = 0.95$, you should get the following graph. (The format is
`word vertex (degree) : adjacency list(weight).`)

[10] `https://commons.apache.org/proper/commons-math/`

[11] Using the online calculator suggested in the Appendix, `https://www.easycalculation.com/statistics/poisson-distribution.php`, you can verify that for $\lambda = 4.2432$ and Poisson Random Variable value 7, you get cumulative Poisson ($\sum_{z=0}^{7} \text{Pois}_\lambda(z)$) of 0.933 (so $\sum_{z=8}^{\infty} \text{Pois}_\lambda(z) = 1 - \sum_{z=0}^{7} \text{Pois}_\lambda(z) = 0.067$), which doesn't quite get into the top 5%; for Poisson Random Variable value 8, you get 0.97, which has tipped you over into the top 5%.

```
Number of word nodes is 16
Number of edges is 62
Word graph:
all (5) : are(1.0) best(1.0) cat(1.0) friend(1.0) my(1.0)
and (10) : angry(3.0) cat(3.0) green(4.0) heard(2.0) i(2.0) is(2.0) purple(3.0) that(2.0) the(1.0) was(1.0)
angry (10) : and(3.0) cat(3.0) green(3.0) heard(1.0) i(1.0) is(1.0) purple(2.0) that(1.0) the(1.0) was(1.0)
are (5) : all(1.0) best(1.0) cat(1.0) friend(1.0) my(1.0)
best (5) : all(1.0) are(1.0) cat(1.0) friend(1.0) my(1.0)
cat (15) : all(1.0) and(3.0) angry(3.0) are(1.0) best(1.0) friend(1.0) green(3.0) heard(1.0) i(1.0) is(1.0) my(1.0)
           purple(3.0) that(1.0) the(2.0) was(2.0)
friend (5) : all(1.0) are(1.0) best(1.0) cat(1.0) my(1.0)
green (10) : and(4.0) angry(3.0) cat(3.0) heard(2.0) i(2.0) is(2.0) purple(3.0) that(2.0) the(1.0) was(1.0)
heard (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) i(1.0) is(1.0) purple(1.0) that(1.0)
i (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) is(1.0) purple(1.0) that(1.0)
is (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) i(1.0) purple(1.0) that(1.0)
my (5) : all(1.0) are(1.0) best(1.0) cat(1.0) friend(1.0)
purple (10) : and(3.0) angry(2.0) cat(3.0) green(3.0) heard(1.0) i(1.0) is(1.0) that(1.0) the(1.0) was(1.0)
that (8) : and(2.0) angry(1.0) cat(1.0) green(2.0) heard(1.0) i(1.0) is(1.0) purple(1.0)
the (6) : and(1.0) angry(1.0) cat(2.0) green(1.0) purple(1.0) was(1.0)
was (6) : and(1.0) angry(1.0) cat(2.0) green(1.0) purple(1.0) the(1.0)
```

This is a correction of the previous output below, which contained an error. (Since many of you were working on the basis of the below being the required output, I'll revise the JUnit tests so it only tests correct answers that were also in this earlier output.)

```
Number of word nodes is 16
Number of edges is 20
Word graph:
all (0) :
and (8) : angry(2.0) cat(3.0) green(4.0) heard(1.0) i(1.0) is(1.0) purple(2.0) that(1.0)
angry (4) : and(2.0) cat(2.0) green(2.0) purple(1.0)
are (0) :
best (0) :
cat (6) : and(3.0) angry(2.0) green(3.0) purple(2.0) the(1.0) was(1.0)
friend (0) :
green (8) : and(4.0) angry(2.0) cat(3.0) heard(1.0) i(1.0) is(1.0) purple(2.0) that(1.0)
heard (2) : and(1.0) green(1.0)
i (2) : and(1.0) green(1.0)
is (2) : and(1.0) green(1.0)
my (0) :
purple (4) : and(2.0) angry(1.0) cat(2.0) green(2.0)
that (2) : and(1.0) green(1.0)
the (1) : cat(1.0)
was (1) : cat(1.0)
```

4 What To Hand In

In the submission page on iLearn for this assignment you must include the following:

Submit a zip file consisting of all the Java classes in the package from the original assignment code bundle.

Instructions on how to create the zipfile are available in iLearn: you'll find them with all the assignment 2 material.

Your file must leave unchanged the specification of already implemented functions, and include your implementations of your selection of method stubs outlined above.

Do not change the names of the method stubs because the auto-tester assumes the names given. Do not change the package statement. You may however include additional auxiliary methods if you need them.

Please note that we are unable to check individual submissions and so it is very important to abide by the above submission instructions.

5 Changelog

- **6/5/16**: Assignment draft released.
- **23/5/16**: Added unipartite graph example; corrected filterGraph() output, and corrected and extended running example, in Task **T9**; suggested different online Poisson calculator.
- **25/5/16**: Corrected Fig 1.
- **27/5/16**: Corrected Poisson example, added note about submission file.
- **5/6/16**: Made further correction to Poisson example.

A Poisson Example

This comes from `http://www.real-statistics.com/binomial-and-related-distributions/poisson-distribution/`. The example uses Excel for the POISSON() function, but there are a lot of online calculators where you can try out this function as well.[12]

> If the average number of occurrences of a particular event in an hour (or some other unit of time) is λ and the arrival times are random without any tendency to bunch up (i.e. the assumptions for what is called a Poisson process) then the probability of x events occurring in an hour is given by
>
> $$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$
>
> **Example** A large department store sells on average 100 MP3 players a week. Assuming that purchases are as described in the above observation, what is the probability that the store will have to turn away potential buyers before the end if they stock 120 players? How many MP3 players should the store stock in order to make sure that it has a 99% probability of being able to supply a weeks demand?
>
> The probability that they will sell $\leq$ 120 MP3 players in a week is
>
> POISSON(120, 100, TRUE) = 0.977331
>
> Thus, the answer to the first problem is 1 - 0.977331 = 0.022669, or about 2.3%. We can answer the second question by using successive approximations until we arrive at the correct answer. E.g. we could try $x = 130$, which is higher than 120. The cumulative Poisson is 0.998293, which is too high. We then pick $x = 125$ (halfway between 120 and 130). This yields 0.993202, which is a little too high, and so we try 123. This yields 0.988756, which a little too low, and so we finally arrive at 124, which has cumulative Poisson distribution of 0.991226.

[12] E.g. `https://www.easycalculation.com/statistics/poisson-distribution.php`

What's been calculated in the answer to the second question is the inverse of the Poisson: given a probability p and a parameter value λ, what's the value of x that gives this?