Design Manual

Frankland, Matthew Hafiz, Farhan

Hughes, George Malcolm Jędrzejczyk, Sabina Moir, Ross

Mukhtar, Ridwan Schmieg, Mark

Sparagano, Nicolas Ewan Giovanni Welsh, Cailean

November 9, 2020

Contents

List of Figures

1 Introduction

The purpose of this document is to provide a clear explanation of the structure and design of the software which was created for D2 of F21DG (referred to in this document as the D2-F21DG software). The document contains class diagrams, which give visual overviews of the different parts of the system, the relationships between classes and their dependencies. A more detailed explanation about the individual classes is provided, describing processes, input, and output.

This document is split into four sections: Topic Model Generation, Model Serialisation, computation of Metrics and the Visual Report.

- Topic Model Generation goes over the design and structure of the creation of topic models program, which is a wrapper for MALLET; it generates topics models given a set of parameters and using a lemmatised corpus.
- Model Serialisation covers the structure of a serialised model. Metrics explains the design of the metric program, listing which metrics were implemented, what parameters they can take, what they return and how they return it.
- Visual Report will explain the architecture of the interactive report, discussing how the topic models are displayed and the relationships between them are illustrated.

For detailed instructions on how to set up and run the D2-F21DG software, please consult the User Manual.

2 Topic Model Generation

Module 1 is a wrapper for MALLET which allows for the generation of multiple topic models given an array of parameters. The models are then serialised for handling by the Metrics program. Module 1 can be interacted with through a command line interface, as arguments and flags provided to the executable .jar or as arguments provided to the main method of an instance of a *TopicGenerator* object.

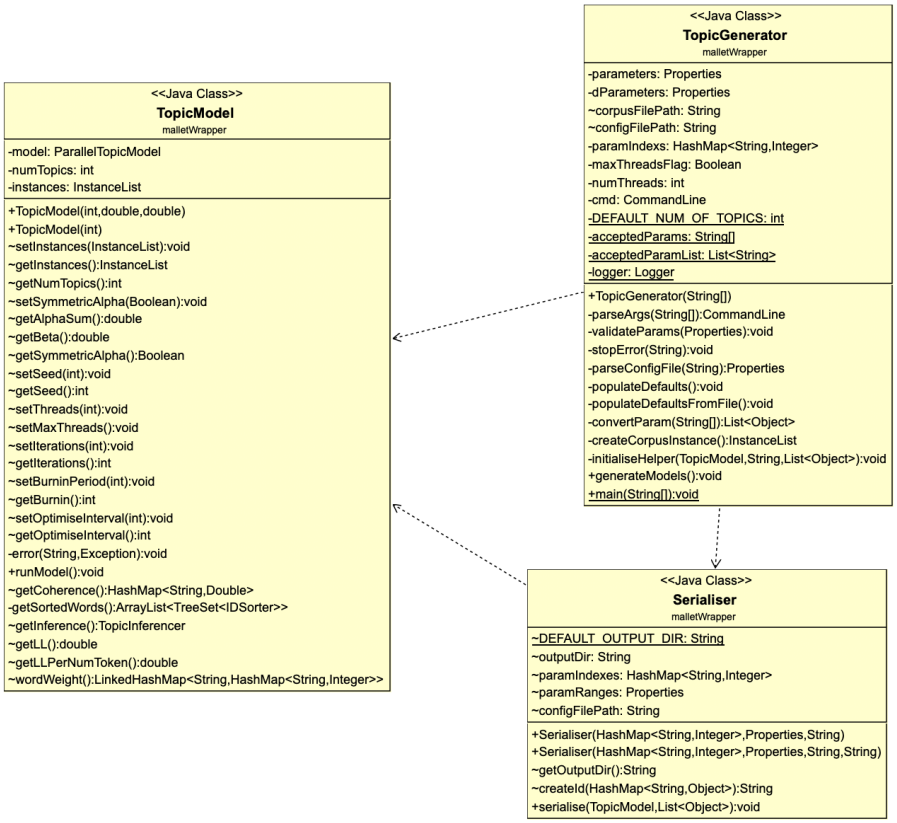

Figure 1: Class Diagram for Topic Model Generator.

The design principle was that all methods are assumed private - with invocation only possible through that class, or default - callable only within the same package. The only public methods are the constructors and singular methods required to run the topic model, generate multiple models or serialise a model. This increases abstraction and hides the implementation, even of getters and setters, for increased security and integrity. This does increase implied coupling between classes because it somewhat forces users to use Module 1 as a collection of classes, not separately, however this is preferred for the scope of this project. It would be a matter of changing modifiers to allow any of the 3 classes to be used independently of each other in future projects.

2.1 Topic Model Class

A Topic Model is the direct interface with MALLET classes, particularly of an instance of a Parallel Topic Model as defined by MALLET. It primarily consists of getters and setters which reference an instance of a Parallel Topic Model, although the getters are used almost exclusively for testing and are not essential.

It should be noted that the Parallel Topic Model object has two points which introduces stochastic elements: setting instances (processing the corpus) and estimating the model. Developers should always set the seed prior to calling the 'setInstances' and 'runModel' methods if repeatable models are desired. This isn't enforced by the Topic Model class as there may be a use case for maintaining pseudo-randomness and users may not always provide seeds.

Topic Model is inherently highly coupled with Parallel Topic Model and an argument could be made that it is unnecessary, with it being possible to create a Parallel Topic Model object within the Topic Generator class. It was chosen to include this wrapper so that the Serialiser class implementation could be abstracted away from MALLET, similarly for Topic Generator. This class also allows the refactoring of data coming from the Parallel Topic Model so it is in the structure that the rest of the module desires.

The Topic Model can be extended to return numerous other data points from the model that were not required by this project. It should be noted that it was desirable to return only the word weights and not the word probabilities, as is possible, so that the serialised models could be reduced in size. This additionally allowed for the implementation of certain metrics in the future (e.g. variations of coherence) which required non-normalised word weights. Word probabilities can be later derived when required by the process of topic normalisation.

2.2 Topic Generator

The Topic Generator class is designed to process a parameter file and generate each of the models described by this parameter file, passing them to a Serialiser object for serialisation and storage.

An apache commons-cli library was used to allow efficient parsing of arguments and flags, similar to that you would expect from any argument parser. The User Manual describes these options and use cases in more depth. There is some validation for the arguments provided but this could definitely be extended to ensure appropriate arguments are provided, such as valid paths for files.

The actual parsing of a parameter configuration file is done by an input stream and a properties object. It follows the conventions set out by Properties.load() which are further described in the User Manual but also described in the Java Documentation. It was decided to follow the conventions for parsing the configuration files both for ease of implementation and to reduce risk of user error by adhering to common standards such as comments beginning with the '#' symbol. The parameters are then validated, primarily by regular expressions, to ensure that the provided parameters match the types required by a Parallel

Topic Model.

Initially, a dynamic approach to validation was used, where the program would continue running as far as possible if malformed parameters were provided, or, for example, only the beta parameter was used. During testing it was discovered that it was almost always desirable for the program to terminate if malformed parameters were provided and so the stopError method is used to write to a log4j2 logger object and then terminate the program.

A logging feature was started and but not finished as it was not on any of the requirements of the project, even as an extensible piece of work. The start of an implementation was developed so that it may be picked up by future students if desired. Instead of writing to System.out/err it is desirable to use a logger which writes primarily to a log file in a standardised fashion including timestamps and has the option of writing to the standard out or console. Log4j 2 is a library created and maintained by Apache Commons and allows easily implementation of this feature. A logger has been created and used for the purpose of error writing however this is stream is written only to the console, not to disk.

Defaults are read, at the first instance, from constants defined at the top of the program and are required to ensure enough parameters have been used to generate a Parallel Topic Model. The only default that must be provided to the Parallel Topic Model constructor is the number of topics, with an optional inclusion of both the beta and alpha. These defaults may be overwritten by including an additional configuration file with the flag '-d' and missing values are then used to populate the parameters Properties object.

The most complex part of the topic generator is taking the parameter values, which may take the form of a single value, a range of values, or a list of values, and converting these into appropriate integers, doubles, arrays of these types, or arrays of the specified values of like types. Converting from strings to lists of primitive data types, only if it is possible to convert to these data types, is cumbersome in Java and it should be noted that there may be a more efficient way of doing this. As it stands, this conversion does not take more than 1 second to execute and so is suitable at this time.

Once all an object containing all the values of all parameters desired to be run, the product of these parameters must be generated so every combination of parameters can be used to generate unique topic models; this is a Cartesian list. The Google Guava library is used to generate this Cartesian list from a set of the parameters.

Once this Cartesian list has been produced, the topic generator loops through each combination and creates an instance of a Topic Model, runs it and passes this object to the serialiser. It would have been possible to take the same Topic Model instance, and thus same Parallel Topic Model object, and change the parameters each time but this was undesirable for two reason. Firstly, if a range of seeds are used as parameters, the corpus would need to be instantiated again, in addition to estimating the model.

Secondly, and more importantly, paralellisation is possible with separate objects for each topic model, but a single topic model which is changed for each model combination is harder to execute simultaneously. By keeping each topic model a separate instance it is possible to extend our implementation to

allow models to be run simultaneously by different threads without read/write exceptions, deadlocking issues etc.

3 Model Serialisation

Once a topic model has been estimated, it can be serialised for offloading to Task 2 and further metrics calculation. This has the benefit of reducing the size of the topic model object by removing unnecessary Parallel Topic Model object data. The model is serialised to a JSON format using the external GSON library, firstly by Task 1 and secondly by Task 2 once the metrics have been calculated.

The following section describes the composition of the serialised JSON model and how the object is filled with parameters, metric, topics and an ID.

3.1 Topic Model Generation

Parameters are read directly from the List object containing the parameters used to generate the individual topic model and serialised, for example:

```
"Parameters": {
        "symmetric_alpha": false,
        "burn_in_interval": 10.0,
        "number_of_topics": 21.0,
        "alpha_sum": 1.0,
        "optimise_interval": 12.0,
        "beta": 2.0,
        "iterations": 103.0,
        "seed_index": 10.0
}
```

The follow is a reduced version of the word weights which are associated with a topic model.

```
"Topics": {
        "topic0": {
            "word0": 1,
                "word1": 3,
                ...
                "word34": 6
        },
        "topic1": {
                "word0": 16,
                "word1": 1,
                ...
                "word26": 2
        },
        ...
        "topic20": {
                "word0": 1,
                "word1": 1,
                ...
                "word39": 12
        }
}
```

As noted previously, this was changed from initially storing word probabilities to storing word weights and deriving probabilities later in the program. This was decided as significant space was saved because word weights are integers and probabilities are overwhelmingly small value, large size, floats. In part possible due to this reduced topic model size, it was decided to store every word weight, no matter how potentially irrelevant it's effect on the topic. This allows for precise metrics to be calculated with access to all data associated with the topic model, and not just taking the top N words per topic.

The ID is derived, both for use as the file name, and for storage in the serialised model. It was initially desired for this to be a string of the parameters used to generate the model, however this generates too large a string if scaled past 3 parameters. Instead the focus was creating a unique ID that can by used to differentiate this model from almost every other model. This is achieved by concatenating every parameter used to generate the model, hashing the string using Java's .hashCode() function, and converting this to a hex string. Whilst the hashCode function does not guarantee uniqueness it uses a well established function using the prime number 31 to create a semi-reliable uniqueness. It is highly unlikely that enough topic models will be generated that the same ID will be created, as this would almost always need to be in the range of millions of topic models generated in a single run.

Example:

```
"ID": "13c8d5df"
```

3.2 Metrics

The metrics program updates the topic model created by the Topic Model Generation section. To do so, it adds all the metrics values generated to the topic model JSON. For quality metrics in particular (and pairwise-difference - which compares two topics at a time), these values represent properties of the individual topic model, making it reasonable to store these values inside each model, so to maintain the structure above, a "Metrics" dictionary object was appended.

Each individual metric has its own unique key (i.e. "log_likelihood) which references either a number or a dictionary object - this dictionary itself mapping from either topics (i.e. "topic0") to a number, models (as in model ID) to a number, or from a statistic key (i.e. "mean") to a number.

The updated topic model JSON now contains the following object after running the metrics program.

```
"Metrics": {
        "log_likelihood": -8.278284484479723,
        "model_log_likelihood": -1.9952296429639574E7,
        "coherence": {
                "topic1": -285.6363150281889,
                "topic2": -417.05805103311695,
                ...
                "topic20": -349.2246965857024
        },
        "coherence_stats": {
                "min": -417.05805103311695,
                "median": -310.36909344459434,
                "max": -257.9647574513934,
                "variance": 2256.0241018744355,
                "mean": -322.09288156846446,
                "range": 159.09329358172357,
                "standard_deviation": 47.49762206547224
        }
}
```

Statistics The serialised model contains statistics dictionary objects within the metrics section, listing the results of a small collection of statistics functions for each metric that contains a list/map of values, allowing for non-scalar metrics to be represented by a scalar value. The statistics object is named by taking the metric key (i.e. "coherence") and appending "_stats". The names of individual statistics are consistent between metrics, for example assuming pairwise_difference is a list-like metric, "standard_deviation" will refer to a number in both pairwise_difference_stats and coherence_stats.

4 Metrics

This section will discuss the metrics program, it will first give an overview of the metrics part. Thereafter, **Metric Loading** will be discussed, which contains a sub part that explains the different metrics implemented. This is then followed by the **Topic Model** section, that goes over appending metric data to a topic model, as well as the de-serialisation and re-serialisation of a topic model.

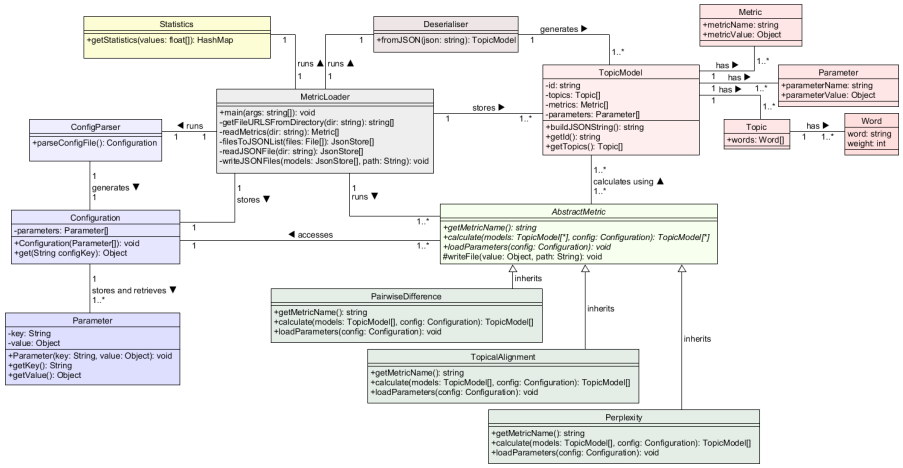

Figure 2: Full class diagram for metrics program.

Figure 2 shows the full architecture of the metrics program. The goal of this design was to make the program as modular as possible, enabling anyone to add new metrics to the program and allowing users to change the configuration of the metrics and the program with ease.

4.1 Metric Loading

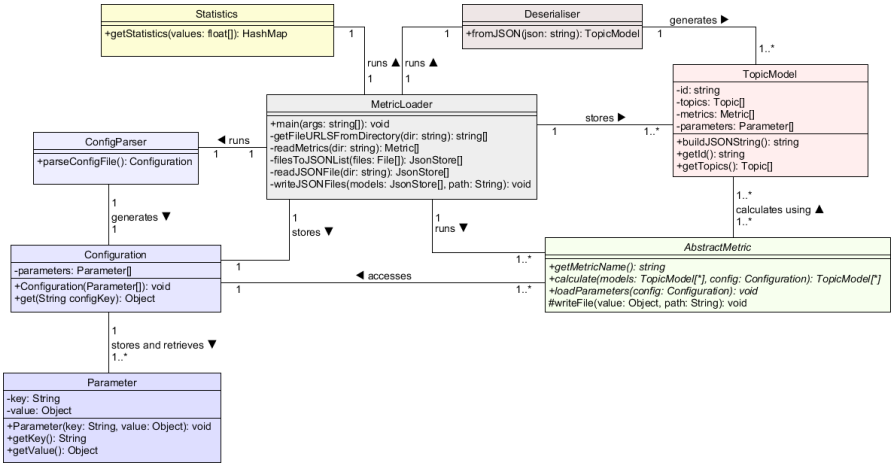

Figure 3: Main Metrics Class Structure for loading metrics and running on topic models.

MetricLoader is the main class of the metric program. It starts by loading the configurations set by the user from the configuration file, using the *ConfigParser* class, which returns an instance of type Configuration.

Then all the metrics are loaded in from the directory containing all the metrics saved as .class files. The metric directory is also specified in the configuration file.

Following that, each topic model coming into the program is converted using GSON (an open source java library created by Google, which de-serialises and serialises Java Objects into JSON) into an object of the *TopicModel* class. The directory containing the models is specified in the configuration file.

It then proceeds to calculate all the metrics for all the topic models by calling the metrics' "calculate" methods, which takes in a list TopicModel objects and an object of type Configuration. Each metric when called, will calculate their metric and update the list of TopicModel objects and returns it back to the *MetricLoader*.

Then the program calculates statistics (mean, variance, median, ...), by calling the "calculateStatistics" method from the *Statistics* class. "calculateStatistics" takes a list of Doubles and returns a Map<String, Double>. The method to calculate statistics is called on the metrics in topic models that appear as a list and updates the topic models.

At the end the *MetricLoader* re-serialises all the updated TopicModel objects into JSON and writes them into an output directory, which is specified in the configuration file.

4.1.1 Abstract Metric

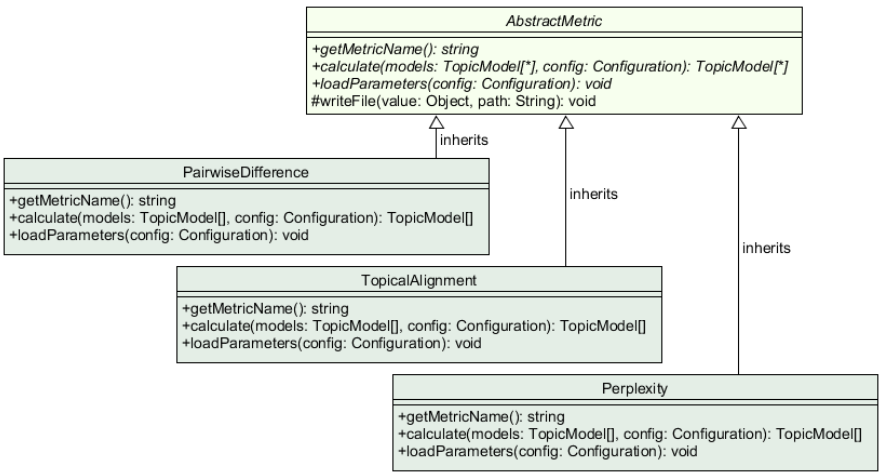

Figure 4: Metrics Class Structure for *AbstractMetric* class and the classes which inherit from it.

The *AbstractMetric* class was created, so that all the created metrics will have the same three main methods:

- "getMetricName", which simply returns the name of the metric.
- "calculate", which takes a list of type TopicModel and an Object of type Configuration. The metric calculates its value and returns an updated list of type TopicModel, where each topic model within the list contains the calculated metric. This is called in *MetricLoader* after the initialisation of a metric class.
- "loadParameters", which takes an Object of type Configuration and loads all the configurations for the current metric from the configuration file.
- "loadParameterString", which takes an instance of *Configuration*, an ID and a default string, returning the value found for the ID in the configuration if there exists one, otherwise returning the default value.
- "loadParameterDouble", which takes an instance of *Configuration*, an ID and a default double value, returning the value found for the ID in the configuration if there exists one, otherwise returning the default value.
- "loadParameterInteger", which takes an instance of *Configuration*, an ID and a default integer, returning the value found for the ID in the configuration if there exists one, otherwise returning the default value.
- "loadParameterBoolean", which takes an instance of *Configuration*, an ID and a default Boolean, returning the value found for the ID in the configuration if there exists one, otherwise returning the default value.

The *AbstractMetric* class also has two protected methods which are not mandatory for the metrics inheriting from this class.

- "buildFullConfID", takes the name of a configuration parameter needed for the current metric, and creates a string as follows: currentMetricName-configurationParameterName. This string helps the metric find the configuration variable it need within the configuration file.
- "writeFile", takes in a Map and an Object of type Configuration. It then writes the content of the Map to an output directory specified in the configuration file. This method is only used for metrics, which are not added to a topic model. Currently only Topical Alignment uses this method, since it uses all topic models at once and requires to be outputted separately.

4.1.2 Perplexity

Perplexity uses topics of words and their respective word counts to calculate the statistical perplexity of the topic when applied to a given corpus of words. This metric determines a perplexity for each topic and returns this as a list of topic IDs and their respective perplexity. Perplexity has four parameters:

- "enable" - An integer which determines if the metric runs or skips. A value of 1 corresponds to running and a value of 0 corresponds to skipping.
- "corpus path" - A string representing the file path of the corpus to use for the perplexity calculation.
- "log base" - A numerical value representing the logarithm base value to use within the perplexity calculation.
- "weight smoothing" - A numerical value that is used as the probability of words encountered in the corpus that are not predicted by the topic.

4.1.3 Pairwise-Difference

Pairwise-differences looks at every pair of models in the list received from "calculate", taking the largest weights from each topic and comparing them across the vocabulary vector space, building a distance matrix between topics and using the Hungarian assignment algorithm to find a single distance value. This calculated distance is placed in each model's metrics dictionary, with the key referencing the other model's ID.

Pairwise-difference has three parameters:

- "enable" - An integer which determines if the metric runs or skips. A value of 1 corresponds to running and a value of 0 corresponds to skipping.
- "top_M" - An integer determining how many of the largest weights should be used during calculation of the distance between two topics.
- "distance_type" - A string used to determine which of the four implemented vector distance functions to use. The options are "manhattan", "euclidean", "chebyshev", and "cosine".

4.1.4 Topical-Alignment

Topical Alignment takes all topics from all topic models and clusters the most similar topics together. This metric generates a tree of clusters, which is outputted as its own file since it's the only metric that uses all topic models at once. Topical Alignment has two parameters:

- "enable" - An integer which determines if the metric runs or skips. A value of 1 corresponds to running and a value of 0 corresponds to skipping.
- "top_M" - An integer determining how many of the largest weights should be used during the calculation of the distance matrices that are used in the agglomerative part of Topical Alignment to generate the topic clusters.

4.2 Topic Model

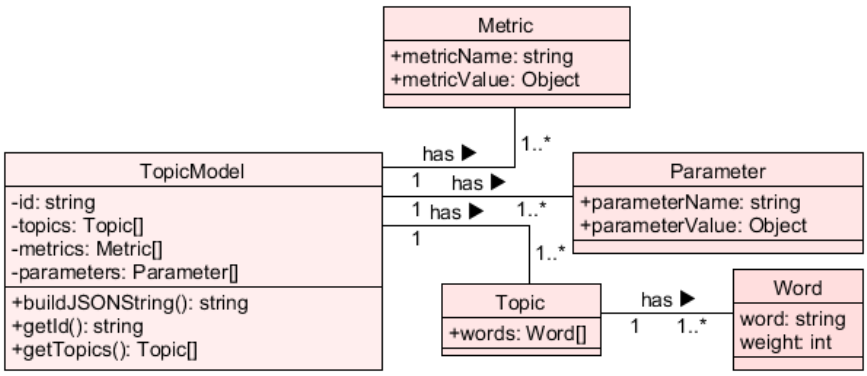

Figure 5: Metrics Class Structure for dealing with Topic Models.

The **TopicModel** class is used to store a topic model. This class was created to access the different parts of a topic model easier and to process and modify it better. The class is separated into two parts, the Getter methods and the JSON builder method. The Getter methods are the the following:

- "getID", returns the ID of the topic model
- "getMetrics", returns a Map<String, Object>, where String is the metric name and object is the value of that metric, this could be a map, or a number of type Double. The map returned by this method contains calculated by MALLET and also the ones calculated by the metrics within the program.
- "getParameters", returns a Map<String, Object>, where String is a parameter name and Object is Double or Integer value. The parameters returned by this method are the ones used for the generation of the topic models.

- "getTopics", returns a Map<String, Map<String, Integer», where the first String refers to the topic ID. The second nested map contains all the words of the given topic with their weight associated to them.
- "addMetrics" serves for a metric to add their own value to the topic model object. This method takes two parameters, a String, which should be the name of the current metric and an Object, which can be a string, number or map that was generated by the metric.

The JSON builder method is the following:

- "buildJSONString", returns a serialised JSON of the TopicModel object.

4.3 Configuration Parsing

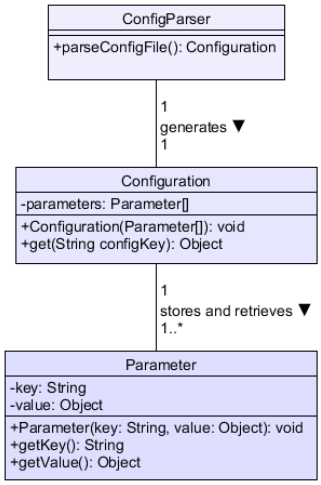

Figure 6: ConfigParser Class Structure for dealing with loading and reading configurations. The implementation of this design does not follow precisely, but the design is used to give intuition.

The **ConfigParser** is a class designed to read a text file using Java properties and generate a Configuration object from it. Each line in a configuration file is of the form "key: value"

- "parseConfigFile", takes a file path as a string and returns an instance of *Configuration*.

Configuration is a class designed to to store a map of parameters, with key being the parameter ID, from the left hand side of the line, and value being the value from the right hand side. Values can be of type String, Double, Integer, Boolean and can also have no value (null) if a key was specified but no value given.

- "get", takes an ID and returns the associated values, acting as a hash map. The main difference is that null is returned, with the error being handled; a message is printed to inform the user that a parameter in the configuration is missing.

5 Visual Report

This section provides an overview of the visual report implementation. The visual report provides visualisation and interaction with the serialised models produced in section 2, Topic Model Generation, examining and comparing the difference and quality measures calculated in section 4, Metrics.

5.1 Visual Design

The Visual Report is designed using D3 as per the requirements; it allows for directory selection, and has two scatter plots displayed on the page, with adjustable axis selection for each of the scatter plots. The data is displayed on the scatter plots in the form of circles which can be clicked on. The selection event loads in the data for the selected circle, which gets displayed below the scatter plots and the parameter file data.

The report was designed with a key focus of preserving modularity and simplicity as core development constraints. This philosophy was carried throughout the visual design as clean space and minimal buttons were embedded into the report to minimise complexity and visual noise. For example, as a key visual design choice the removal of any submission buttons for the drop-down selection menus allowed the visual report to expand its use of dynamic visual transitions to provide increased visual feedback to the user and improve overall experience.

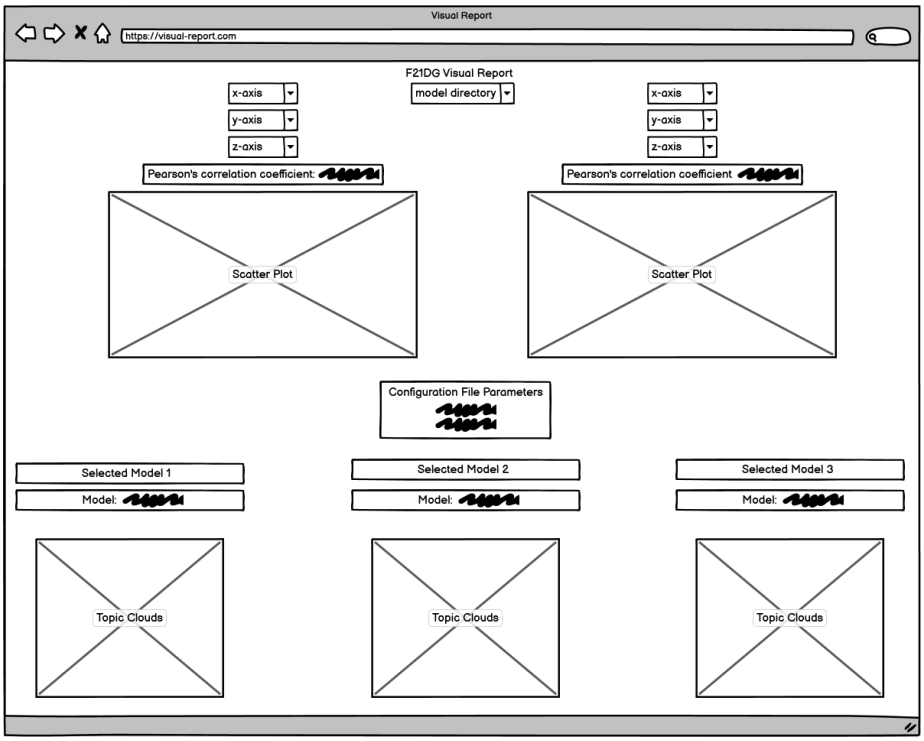

Figure 7: Wireframe design of the visual report

5.2 Main file structure

- *index.php (main file)*
  This is the main file, all other files require it to work.
- *loadData.php*
  This is depended on the data-files directory with the folders that contain the model data
- *general.css*
  No requirements other than index.php
- *metrics_ display.css*
  Dependent on display_model_data, display_clouds and init.js
- *scatterplot.css*
  Dependent on scatterplot.js and init.js
- *display_ model_ data.js* This is dependent on the loadData.php, and the selections from the scatter graphs
- *display_ topic_ clouds.js*
  This file is used to display the word bubbles on the Visual Report and is dependent on display_model_data.js, init.js, and the selections from the scatter graph.
- *init.js*
  Main entry point no dependencies other than the index.php, it is a requirement for scatterplot, and display data/clouds
- *load_ directory_ data.js*
  Dependent on loadData.php
- *scatterplot.js*
  Requires loadData.php and init.js, all scatterplots are linked but they dont require each other
- *data-files (default directory for the model data)*
  This is depended on the user and the running of Task 1 and 2.

5.3 Loading the Data

The data files are loaded in through a PHP script defined in the loadData.php file. The script loops over all the files present in the defined directory which is defaulted to data-files. The script also keeps track of sub-directories present in data-files and lists them on the Visual Report from which the user can select a sub-directory through a drop down list to get the models from instead of the default one. If there are no models found in the data-files directory, it will display a message asking to select a sub-directory to choose models from. If it detects a JSON file by checking the extension of the file, it reads all the data from it using the *file_ get_ contents()* method where the parameter given is the full path to the JSON file. After reading the file it appends the data into a PHP array through:

array_ push($dataArray, json_ decode($data));

If the file in question is not a JSON file then it will be skipped without taking any actions. In the end the data from all the JSON files is placed into a PHP array *$dataArray* which is then passed into a JavaScript variable data through:

var data = <?php echo json_ encode($dataArray) ?>;

On loading of the page the data shown on the graphs will be from all the JSON files present in the data-files directory, since that is the default directory to search files from unless changed by the user to a sub-directory present in the data-files directory through the drop down list provided.

After the data is loaded on visual report start up, the available metrics and parameters are retrieved from the first model in the data object and are loaded in as drop down menu options for the scatter plot x and y axes which are used for the displaying of models, and an z axis which is used to change the size of the circles on the scatter plot based on the selected metric. If a given metric has an array of other information e.g. coherence has an array of information about the mean, standard deviation, range etc, then the mean element from that metric is taken as an option for it. If the metrics and parameters cannot be found, an error message will be displayed.

5.4 Displaying Parameter Configuration File

Similarly to how the data is read for the models (using PHP), the extension of the file is checked to see if it is a .conf file, if it is then the data is read into a variable using the *file_ get_ contents()* method and passing in the full directory path and filename *$fulldirectory.$fileName*. Following on from that the data is broken down into an array using the *explode()* method in PHP, which uses a new line as the separator. This ensures that all of the contents of the conf file are read in.

The array is then passed into a JavaScript variable similarly to how the JSON files are passed in:

var confData = <?php echo json_ encode($confArray) ?>;

Once all of the data is read in it is then displayed on the visual report underneath the scatter plots, by reading the elements of the array, with a check to see if there are any empty elements in the array, and if any of the elements contain a "#" (which is a comment in the conf file and thus does not need to be displayed on the visual report).

5.5 Scatter Plot Graph Visualisations

The scatter plot graphs provide the primary source of information display and interaction between the user and the topic model data. Each scatter plot is defined in the index.php and linked in to the d3 JavaScript within the initialization script, init.js. Addition of new scatter plot graphs was designed to be completely modular and dynamic, standard scatter plot HTML div format is shown below:

```
<!-- A div that contains all the sections to render a new scatter plot-->
    <div class="div-scatterplot">
       <div>
    <p>Choose the x and y axes for the scatter plot</p>
    </div>
    <div class="axis-select">
    <form name="select-form" id="select-form">
    <label>x-axis:</label>
    <select name="x-axis" id="x-axis" onchange="update_axis()">
    </select>
    <br><br>
    <label>y-axis:</label>
    <select name="y-axis" id="y-axis" onchange="update_axis()">
    </select>
    <br><br>
    <label>z-axis:</label>
    <select name="z-axis" id="z-axis" onchange="update_axis()">
      <option value="fixed">fixed</option>
    </select>
    <br><br>
    </form>
    </div>
    <div id="correlation"></div>
    <!-- Create a div where the graph will take place -->
    <div id="scatter_plot_cont" class="scatter_container"></div>
    </div>
```

If an additional scatter plot is to be required for the visual report the developer can simply copy this standardised format into the parent div, mainscatterplot-container, and rename the ten name/id identifiers found in standardised div. Next as shown below, the developer is simply required to append the new id tag names to the data arrays found in the initialisation script init.js.

```
var scatterplots = [
    scatterplot("scatter_plot_cont"),
    scatterplot("scatter_plot_cont2")
]
var xAxisSelections = [
    document.getElementById("x-axis"),
    document.getElementById("x-axis2")
]
var yAxisSelections = [
    document.getElementById("y-axis"),
    document.getElementById("y-axis2")
]
var zAxisSelections = [
    document.getElementById("z-axis"),
    document.getElementById("z-axis2")
]
var correlations = [
    document.getElementById("correlation"),
    document.getElementById("correlation2")
]
```

Modularity and simplicity enabling the quick addition or removal of components was a key philosophy during development and can be seen throughout the report, for an additional example of component swapping see section 5.5 Model Selection.

Extending beyond the core components of the report, this philosophy can also be seen in the dynamic scaling of axes. Utilising the Enter, Update, Exit design of D3, scalars are employed to dynamically scale axis lengths to a fixed size regardless of the inputted metrics range.

These scalars allow the report to visualise any metric or parameter inputted into the serialised JSON of topic models and dynamically update the axes when a new option is selected from the input drop-down lists without the requirement of additional programming or development time.

In parallel to the dynamic updating of axis values, the visual report also repositions and resizes all visualised models to represent the newly selected metric or parameter using the same scalar functionality.

5.6 Model Selection

In addition to the high level visual comparisons provided by the scatter graphs, this report also features the ability to select any particular model to focus on the specific metrics, topics, and parameters that build the model. Once selected, a model will be displayed as a collapsible container. The collapsible model is recursively built within the *display_ model_ data.js* script allowing dynamic creation without the requirement for specific development for any metric or parameter.

This is achieved by recursively building child collapsible containers from the topic model JSON and displaying the values for every metric, topic, and parameter found. By circumventing the requirement for additional programming with the addition of any new metric, the visual report can dynamically display an infinite number of metrics for any given model without the additional development time that would otherwise be required to implement specific source code to parse a metric or parameter.

At present the selection of three concurrent models is built directly into the visual report. However, similar to the addition of new scatter graphs described in section 5.4 Scatter Plot Graph Visualisations, a modular approach has been deployed to streamline the increase of additional model selections.

Standardised model selection html div

```
<div id="model_selection_1" class="model_div empty selectionclass1">
    <h1>Selected Model 1</h1>
</div>
```

CSS for a selection class

```
.selectionclass1{
    color: #1B2BEA;
    fill: #1B2BEA;
}
.selectionclass2{
    color: #B20D30;
    fill: #B20D30;
}
```

To increase the number of possible models that can be selected, the standardised selection div above can be simply copied below the currently implemented selection divs and the *model_ selection* values appropriately incremented. This addition will allow the report to loop over the available selection divs and append a selected model to any empty location that is found increasing the overall number of possible selected models. Finally, the CSS of the new model selection class can be appended to *metrics_ display.css* as shown above, allowing unique identification of the newly added selection possibility.

5.7 Topic Clouds Visualisation

The word bubbles are displayed using the D3-cloud plugin. The data for the word bubbles is gathered through the *getDataForClouds()* function where the parameters given are the model ID and the display container for the word bubbles. The function is called in the *render_ selected_ model_ data()* which can be found in *init.js*.

After the function is called, to gather data, it goes through each topic one by one of the selected model and adds the words and the weight of all the words for that topic into a *wordsList* array. After all the words of one topic are placed into an array along with their weights, the array is sorted into descending order based on the weights of the words. From that, the top 30 words with highest weights are selected using the slice function and are placed into another array called *top30Words* which is then used to display the word bubble. Before moving onto the next topic in the model, the word bubble for that model is displayed by calling the *displayTopicClouds()* function. The parameters of the function are the array which contains the top 30 words and the container where the word bubbles are to be displayed.

In the *displayTopicClouds()* function the format of the words to be displayed is defined including the font, padding and size along with the overall width and height of the graphic. The weight is used to define the font of the word to be displayed. The weight value for each word is scaled utilising the D3 scaling functions to make the words that are enormous in size to be reduced to fit in the defined container without causing formatting issues.

The bigger the size of the word displayed in the word bubble, the higher its importance in that topic. After the top 30 words of one topic have been displayed, it moves onto the next topic and will repeat until there are no more topics left in the selected model.