11-792 Project Report

Nicholas Gekakis, Boyue Li

May 8, 2018

1 Introduction

In this project, we built a distributed parallel configuration space exploration pipeline framework, which can exhaustively try all possible parameter combinations automatically. Users can easily configure multiple modules, create their own modules and run on multiple processes or even multiple machines.

2 Requirements

2.1 Easy to configure and deploy

The framework should be easy to configure and deploy.

2.2 Save and resume

The framework should be able to save intermediate results so that it can be interrupted and resume running at a later time.

2.3 Automatic parameter exploring

The framework should be able to automatically execute using all possible parameter combinations that have been configured by the pipeline developers.

2.4 Easy to develop users' modules

The framework should support a easy way for users to develop their own modules.

2.5 Load balancing

The framework should be able to handle load balancing since different modules require different excution time.

3 Design

3.1 Overview

As shown in Fig. 1, a pipeline is constructed from several independent modules. A module reads data from the data server, processes data according to all possible parameters, then save the results for each configuration to the data server.

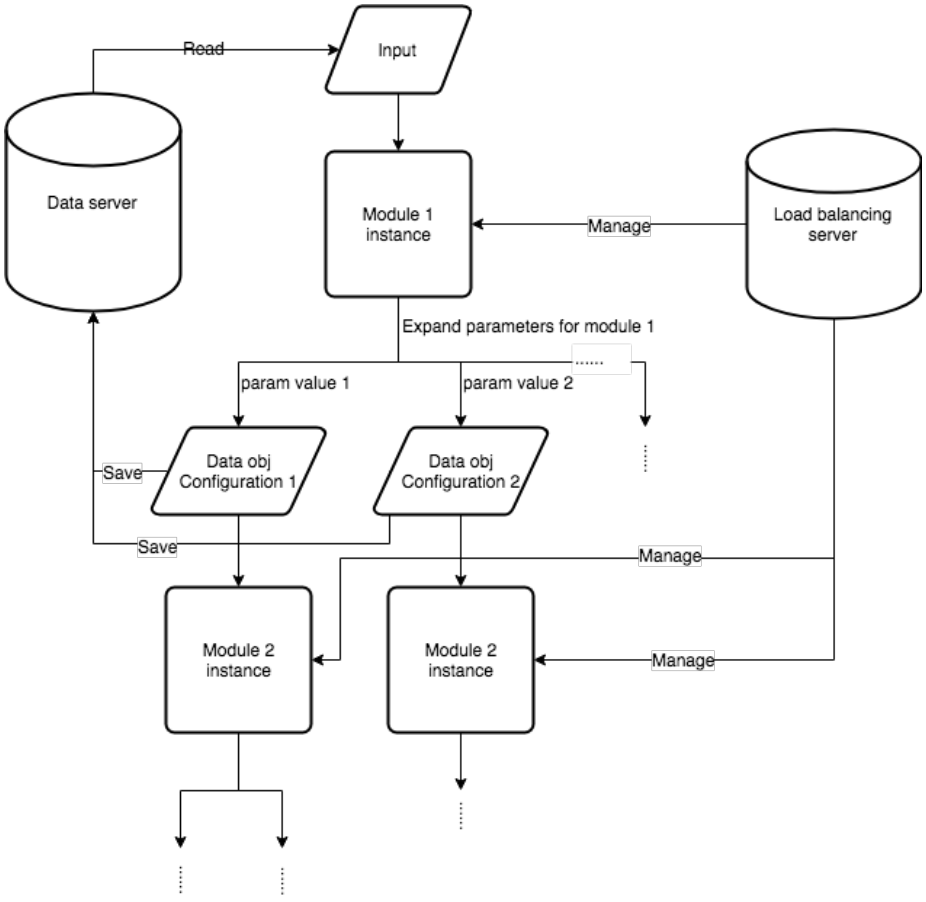


Figure 1: Control flowchart.

Every module runs on an independent process and communicates through RabbitMQ using the job class (defined in section 3.5) which contains parameters, current excuetion status and the path to input file. Fig. 2 describes the information flow. The Load balancing Server distributes jobs to different modules' instances. Once the job is finished, the instance sends a confirmation to the Load Balancing Server.

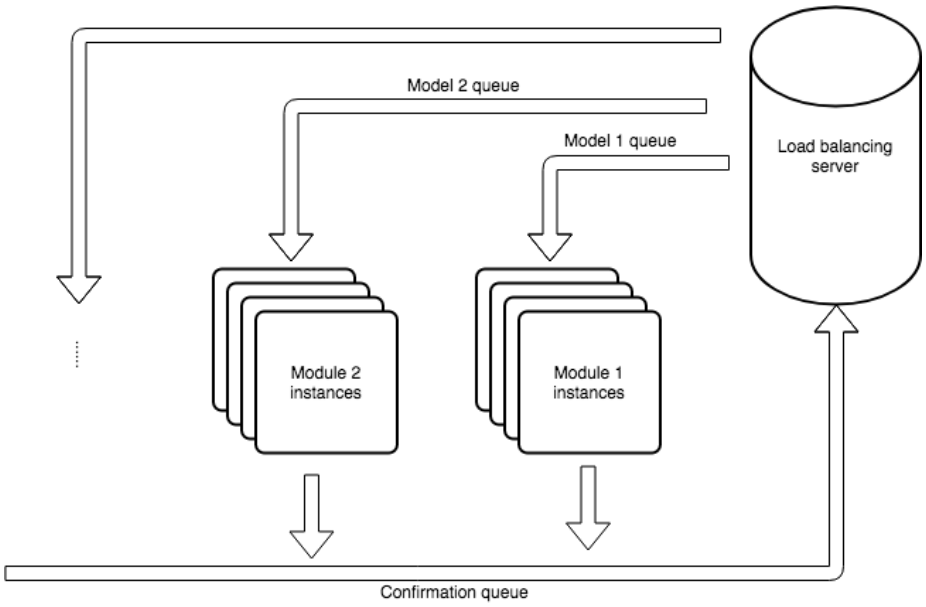


Figure 2: Information flowchart.

Users only need to specify the connections between modules and the parameters associated with each module, the framework will automatically handle the exploration and execution of the configuration space.

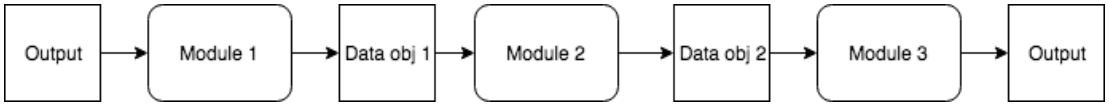


Figure 3: A sample pipeline

Figure 3 shows a sample pipeline which passes the input information object through some modules and produces the output information object.

We also provided a command line executable, to ease users' pain of coding.

3.2 Pipeline

The pipeline class manages modules and parameters. It reads the configuration file, creates the pipeline and is the class that implements the Load Balancing Server. The pipeline can't have branches or merges, and the order of the configuration file should be the same as the actual pipeline.

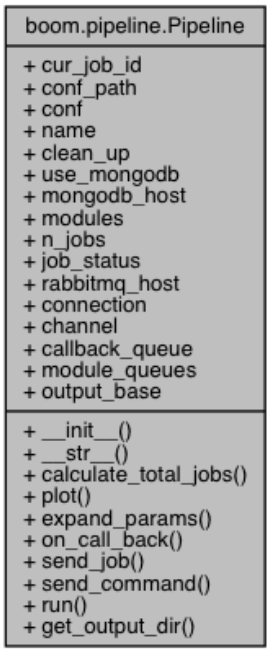


Figure 4: UML diagram for the pipeline class.

3.3 Module

A module is the basic computation unit which takes an input and produces an output. Every input and output is a job object defined in sec. 3.5.

A module needs to maintain the following fields:

- Name of the module.
- Number of instances.
- Configuration of the pipeline.

When running, the pipeline will send jobs to module instances.

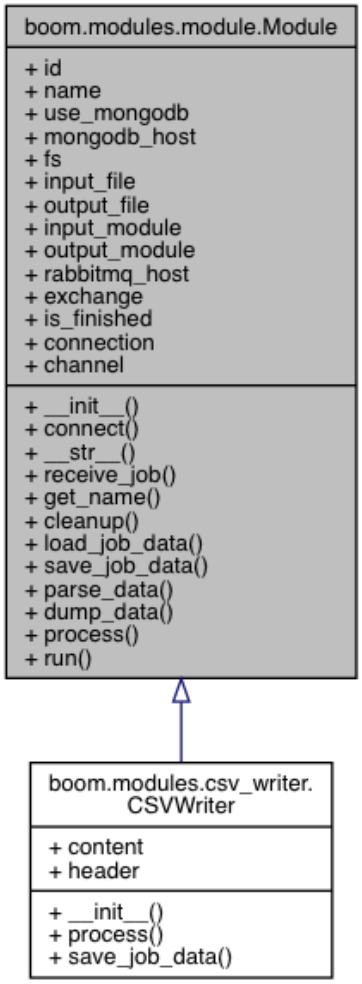


Figure 5: UML diagrams for the abstract module class and derived classes.

3.4 Parameter

The parameter class manages one parameter. It should handle all operations related the parameter, including updating the parameter value, set and reset the value. There are several types of params: integer, float and collection.

It also needs to save maintain the following fields:

- Name of the parameter.
- Type of the parameter: int, float or collection.
- Interval of possible values or possible values of a collection.

- Step size of the parameter (for numerical params).

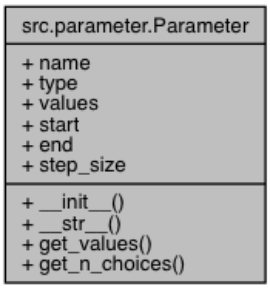


Figure 6: UML diagram for parameter class.

3.5 Job

The job class is the information object used between modules. It maintains the following fields:

- Job id: the unique id of the job.
- Producer: the module that produced this information object.
- Consumer: the module that this information object to be passed to.
- Input uri: the uri to the data file.
- Output path: the path to store the resulting data file.
- Params: the params that the module should use to process the data object.
- Timestamp: the timestamp when the data object was created.
- Processing time: the time taken to process the data object.

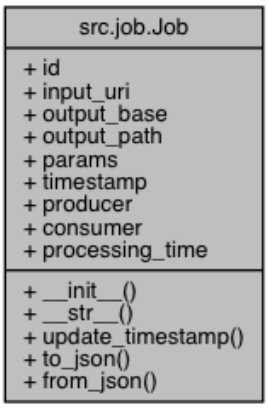


Figure 7: UML diagram for job class.

3.6 Configuration file

We use YAML files to configure the framework.

3.7 User defined modules

Users need to put their module classes in a file in the same folder with the configuration file. Then the framework will automatically load them.

3.8 Executable

We provide an executable that reads in the configuration file, and starts the whole pipeline. It reads the configuration file and extracts needed infomation. Then according to the configuration, it generates code for each module and execute the code using different processes.

4 Example pipeline

This pipeline consists of a sample module, which does nothing but adds its own configuration and parameters to the data to show this pipeline is working.

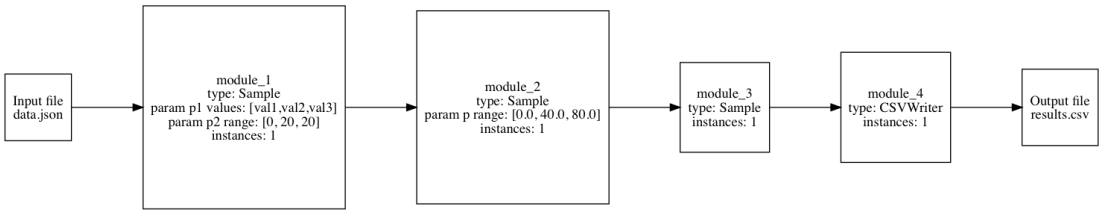


Figure 8: The structure for toy pipeline

5 BioASQ

5.1 Implemented Modules

Table 1 listed all modules and their parameters we implemented. We have also made modifications to the BioASQ pipeline in the process of adapting it to work with our framework. First, because the modules each need to run hundreds of times, we have made performance improvements where we could. This includes making a multi-process CoreMMR module, (the most computation heavy module) that is more than 4x faster than the original module (depending the configuration). We have also had to refactor the way the data gets passed along the pipeline because each module needs to handle all data in order to pass it along to the next module, rather than only being passed the data relevant to its own functionality.

Module	Parameters
CoreMMR	α
Orderer	k
Tiler	word_limit
Rouge	

Table 1: Table of implemented modules.

5.2 Pipeline

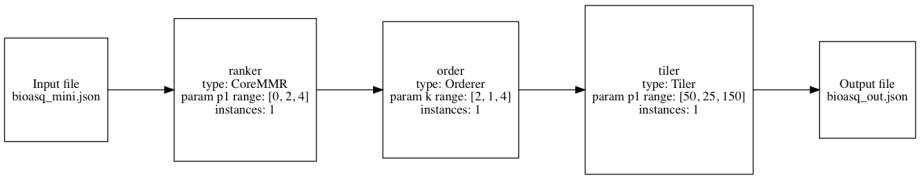


Figure 9: The structure for BioAsq pipeline

5.3 Experiments

5.3.1 Dataset

The results were run on a subset (1800 total questions) of the bioasq_train_formatted.json dataset.

5.3.2 Results

Our configuration tested 2688 unique parameter combinations. Fig. 10 shows the distribution of the 2688 Rouge scores that resulted from the configuration space exploration. The results clearly show that for BioASQ have two distinct clusters of scores, one for which the Rouge score is between 0.24 and 0.26, and the second with scores between 0.28 and 0.32.

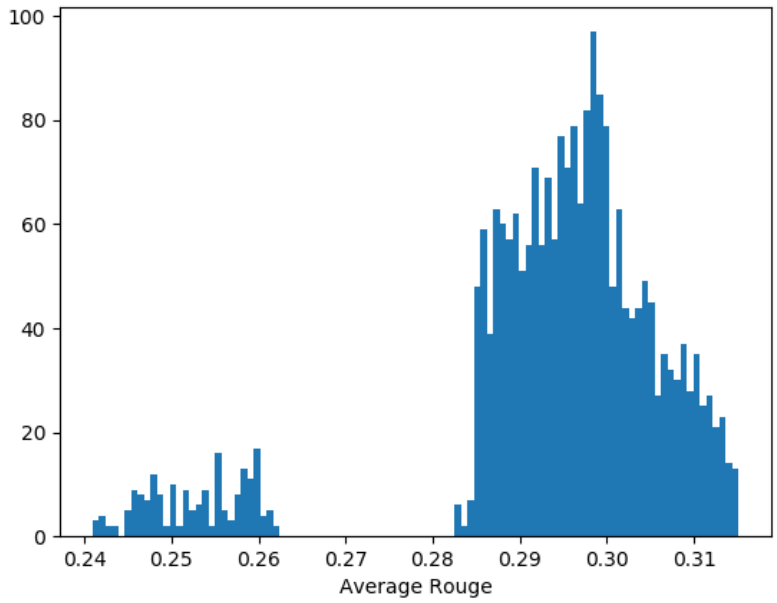
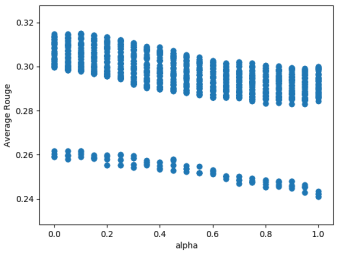


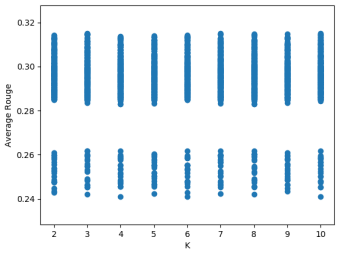
Figure 10: The distribution of scores across all 2688 parameter configurations

Looking at each parameter plotted individually, as in Fig. 11, we can see that the separate clusters of scores are entirely explained by the word limit parameter.

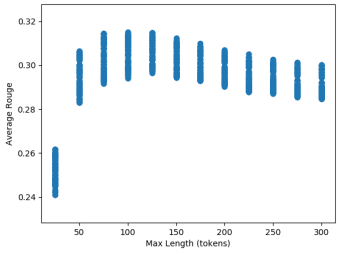
The results showed that he alpha parameter (Fig. 11a) showed noticeable penalties for values greater than 0.2, although as a penalty for repetition higher values may score better on a human-evaluated readability metric.



(a) Average Rouge score vs. alpha



(b) Average Rouge score vs. k (number of clusters)



(c) Average Rouge score vs. word limit

Figure 11: Distribution of Rouge scores against each individual parameter

The K parameter (Fig. 11b) used in the KMeans clustering to perform sentence ordering does not have a large impact on the results. This is intuitive since the purely reordering the sentences does not affect the Rouge score unless a sentence is moved to the end and clipped due to the maximum length requirement.

The maximum length parameter (Fig. 11c) has by far the biggest impact on the results with a huge penalty for allowing fewer than 75 tokens, but also a decrease at greater than 150.

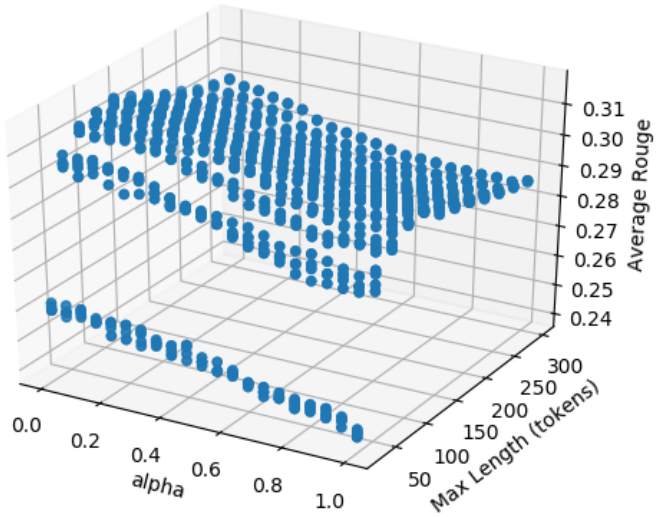


Figure 12: The distribution of against both word_limit and alpha

Fig. 12 shows Rouge score plotted against both the alpha parameter and the word_limit parameter, the two parameters with the most influence over the Rouge score. This graph shows the interaction of the two parameters and and encompasses the vast majority of the variation in Rouge score since the k parameter has a negligible impact. From this graph, we can see that the maximum Rouge score appears where the alpha is low (<0.2) and the word_limit parameter is in the range of 100-150 tokens.

Our experimental results show that the best configuration for the existing BioASQ pipeline uses the following parameters: alpha=0.10, k=3, word_limit=100.

6 Discussion

7 Future works

Acknowledgements

Thank you to Khyathi Chandu for her assistance.