A Generalizable Framework for Automated Cloud Configuration Selection

Supervisors: Adam Barker & Yuhui Lin

Jack Briggs - 140011358

MSc Data-Intensive Analysis

2019-06-06

Abstract

Outline of the project using at most 250 words

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is NN,NNN* words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

List of Figures

1 Introduction

Describe the problem you set out to solve and the extent of your success in solving it. You should include the aims and objectives of the project in order of importance and try to outline key aspects of your project for the reader to look for in the rest of your report.

1.1 Background

1.2 Aims and Objectives

Algorithm Multiple cloud providers Latency/response tests from a separate machine Concurrent jobs to reduce search time

1.3 Contributions

1.4 Dissertation Overview

2 Literature Survey

Surveying the context, the background literature and any recent work with similar aims. The context survey describes the work already done in this area, either as described in textbooks, research papers, or in publicly available software. You may also describe potentially useful tools and technologies here but do not go into project-specific decisions.

3 Requirements Specification

Capturing the properties the software solution must have in the form of requirements specification. You may wish to specify different types of requirements and given them priorities if applicable.

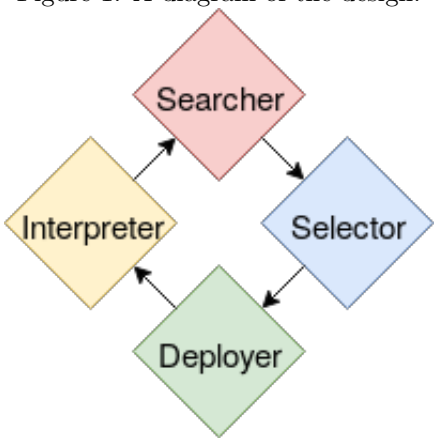
Previous solutions to the problem of automated cloud configuration selection have focused on specific use-cases or application types, and have not provided a functional implementation. Our design should be generalisable to any form of application deployed on the cloud, and should be able to recreate previous solutions. This paper should also come with an associated implementation, which can at least perform a Bayesian Optimization-based search for a given docker container containing a batch job or benchmark.

3.1 Use-case

4 Design

Indicating the structure of the system, with particular focus on main ideas of the design, unusual design features, etc.

Figure 1: A diagram of the design.



Optimisation is the process of minimising or maximising the value of some objective function by adjusting its input variables or parameters. Any optimisation algorithm begin with an initial guess of these variables and iterate through improved estimates until they terminate, hopefully providing an estimated solution. [1] In the generalized case, the optimisation method and objective function are both unknown, but we know that our objective function will always involve selecting some cloud configuration based on the inputs, deploying some application onto this configuration, and interpreting its performance to give some objective measure.

This process can be broken down into four components: A Searcher, which runs the optimisation algorithm, testing out various inputs in an attempt to maximise or minimise the objective function; a Selector, which interprets the inputs to determine what cloud configuration is being tested; a Deployer, which deploys the application onto this cloud configuration and returns any required logs from it; and an Interpreter, which takes these logs to calculate the objective measure which is returned to the Searcher as the returned value for the objective function. A diagram of this breakdown is shown in figure 1.

It is assumed that in the vast majority of cases, the user would provide their own Interpreter and Selector, that in some cases they must provide their own Deployer, and that only in rare cases would the user be required to provide their own Searcher. This is because optimization algorithms can be applied to

any deployments, with only small modifications necessary in rare cases for specific cases. Deployments can often be contained within Docker containers, and aside from occasional setup, for example in multi-node clusters, a Deployer which provisions a given configuration from a given provider, and then deploys and attaches to a user-provided docker image to collect its logs will be sufficient. Interpreters and Selectors, on the other hand, are very dependent on the form these logs will take, and the form the search space will take, and are extremely hard to generalise. For this reason, the modular design of our solution should make it simple for any component to be supplied or replaced by the user.

For each component

4.1 Searcher

The Searcher component performs an optimisation algorithm, such as Bayesian optimization, coordinate descent, random search, or exhaustive search, and drives the optimization process by iterating through potential input variables. For each set of these inputs, it take a sample from a single 'job,' and run through a single loop of the other three components. The constraints for these input variables must be specified, by describing their type (integer, float, categorical) and limits. A description of how to model cloud configurations into a set of variables is done in the Selector section.

4.1.1 Example - Grid/Exhaustive search

In an exhaustive search, every possible combination of the inputs is sampled, giving a complete analysis of the entire search space. This obviously takes many samples, $n * \prod_{i=1}^{j} x_i$ where x_i is the number of options for the ith of J variables, and n is the number of samples taken from each configuration. This results in a large or even infinite search cost and time, but is almost certain to return the optimal result, depending on the amount of randomness involved in sampling. A grid search is similar, but rather than sampling at every possible option within the search space, samples at regular intervals within it to reduce the number of samples taken. A coarser grid search results in a quicker, cheaper search, but a less confident prediction.

4.1.2 Example - Bayesian Optimization

4.2 Selector

The Selector interprets the variables provided by the Searcher component into the form of a cloud configuration that can be deployed. Cloud configurations have a number of variables that can describe them, such as vCPU number, Memory amount, number of instances, instance category, machine type, and cloud

provider. The selector must use whatever combination of these is provided and find either the exact or most similar cloud configuration available, passing this information on to the Deployer.

4.2.1 Example - Exact Match

4.2.2 Example - Closest Match

4.3 Deployer

The Deployer deploys the user-provided application, batch job, or benchmark onto the selected cloud configuration, and collect any necessary analysis from it. Typically this will involve provisioning the necessary machines from the given provider, followed by deploying the given application onto these machines, and either collecting logs from them or from a networked instance or cluster.

4.3.1 Example - VM Provisioner

The deployer should be capable of handling any response from the selector, and must work with multiple providers. This can be simplified using Infrastructure as code (IaC) tools such as Terraform or Chef.

4.3.2 Example - Docker Deployer

vBench

Cloudsuite3

Sysbench

4.3.3 Example - Ping server

4.3.4 Example - Fake Deploy/Old Deploy

4.4 Interpreter

The Interpreter must interpret whatever information is returned by the Deployer, along with the cost of the cloud configuration, in order to give the searcher an objective measure for the sampled cloud configuration. It is this returned value which will be minimized or maximized. By its nature, the interpreter will most likely be unique for a given application.

5 Implementation

How the implementation was done and tested, with particular focus on important / novel algorithms and/or data structures, unusual implementation decisions, novel user interface features, etc.

5.1 Searcher

For the sake of generalizability, the available implementation of spearmint was found to be outdated and incompatible with the latest versions of various python modules planned to be used in later steps. Because of this, spearmint was first updated to be compatible with Python 3 and newer versions of its dependencies such as Google Protocol Buffers. This implementation of spearmint has been made available.[1]

5.2 Selector

5.3 Interpreter

6 Evaluation

You should evaluate your own work with respect to your original objectives. You should also critically evaluate your work with respect to related work done by others. You should compare and contrast the project to similar work in the public domain, for example as written about in published papers, or as distributed in software available to you.

To demonstrate the effectiveness of the implementation of our framework, we wanted to show that it could replicate the methods set out in the Cherrypick paper [**?**] by performing Bayesian Optimization to attempt to find an optimal configuration for a given deployment. We then wanted to use the same implementation to perform multiple exhaustive searches so that the results from Bayesian Optimization could be properly compared to the real average results.

The deployment to be used was originally planned to be Cloudsuite3's media streaming benchmark, however this was found to be extremely variable and dependent entirely on network bandwidth, and so instead the vBench video transcoding benchmark was used. Our objective function measured an instance type's relative rate of transcoding of a single 5 second 1920x1080 video file, returning a score of 0 if the quality was below a given threshold, divided by the hourly price of that instance. Effectively, we maximised the rate of transcoding a unit of video length at a sufficient quality per hourly cost.

[1] https://github.com/briggsby/spearmint3

	Provider	
Machine Category	Amazon EC2	Google Compute Engine
General	n1-standard	m5
Memory	n1-highmem	r5
CPU	n1-highcpu	c5

Table 1: Machine types corresponding to different instance categories for the two providers

For choosing the boundaries of the search space, we decided to reduce costs by using only machines ranging from 2 vCPUs to 8 vCPUs. This was also convenient as to use more powerful machines would have required requesting additional raised quotas from the providers used, and so would be less easily replicable. Using exact match, and as we wanted to show that the implementation worked with multiple providers, we could not rely on Google cloud platform's custom machine types and had to find some way to categorize possible variables. Rather than including memory as a variable, CPU instead was coded as a categorical variable (2, 4, or 8), along with machine category (General, Memory, CPU), each with a different amount of memory. Table 1 show which machine types corresponded to each option.

For realistic situations, the search space could follow this same pattern, with the CPU variable extended, possibly instead as an integer between 1 and 6 which 2 is raised to the power of to determine vCPU number, as this covers many of the available CPU options.

The 18 machine types decided upon (3 vCPU numbers for 3 machine categories for the 2 providers) were stored in a single dataset for use with the 'Exact match' instance selector. The vBench deployer was used which utilized Terraform to provision a single instance of the given machine type, on which the remote docker api was used to deploy and collect logs from an docker image containing vBench. The vBench interpreter then isolated the vBench score from these logs, and divided it by the hourly cost of that machine type to return the final value.

Where not otherwise specified, statistical results are quoting the p-values returned from a Tukey's Honest Significant differences test to correct for multiple comparisons.

The results of the exhaustive search are shown in figures 2 and 3. The raw scores (time taken to transcode relative to the reference) show clear overlap between many possible configurations, and in general show a significant difference between 2 vCPUs and either 4 ($P < .001$) or 8 ($P < .001$) vCPUs, with a larger number of vCPUs increasing the score, but show less clear differences between 4 and 8 ($P = .066$), dependent on the provider and machine category, suggesting either diminishing returns or a limit of the benchmark to utilize all vCPU cores.

However, once the scores are instead given relative to the machine's hourly costs, there is much less overlap. A clear optimal configuration can be seen in the c5.large machine type, if one is purely interested

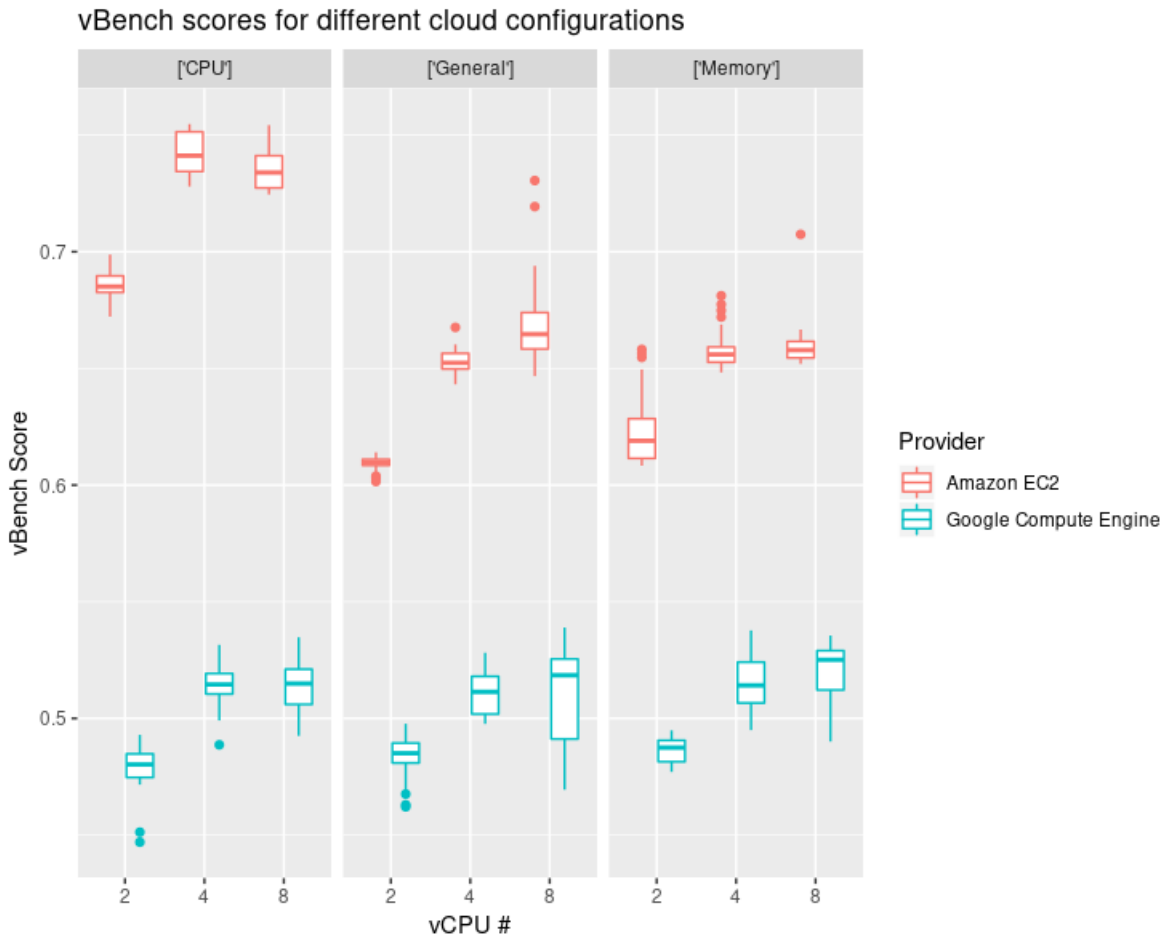


Figure 2: Distribution of vBench scores

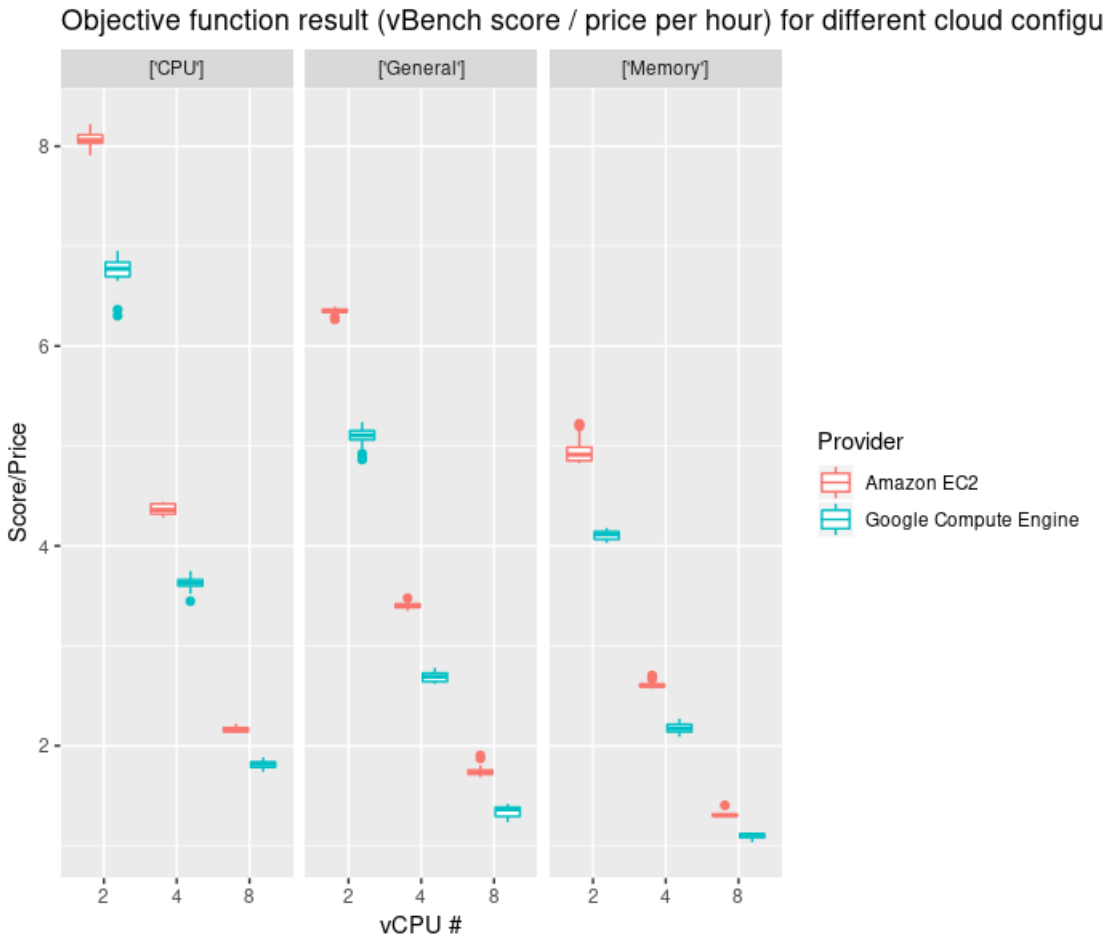


Figure 3: Distribution of objective function values

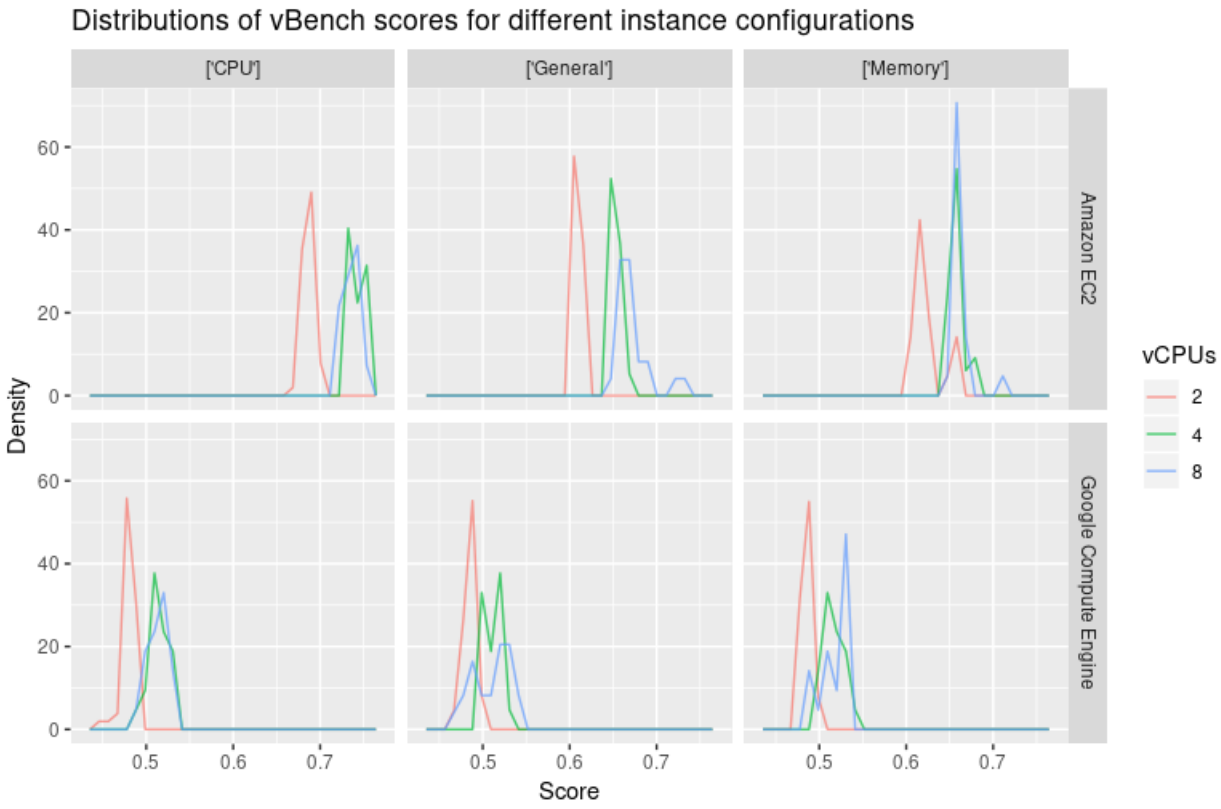
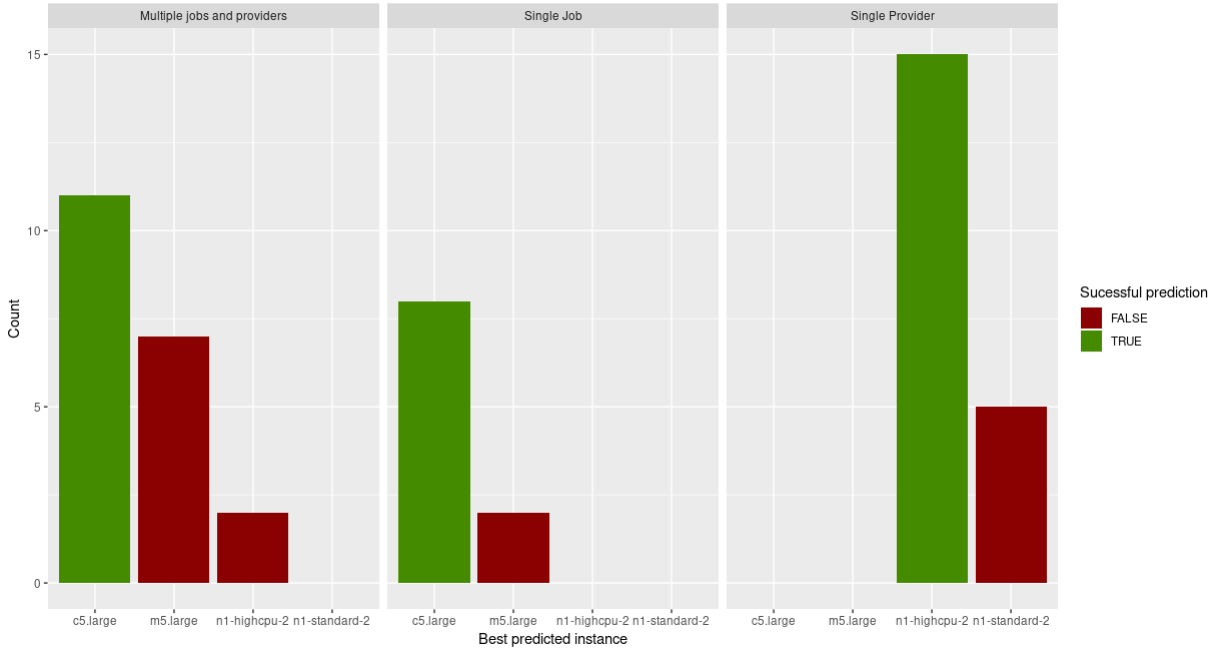


Figure 4: Distribution of vBench scores in frequency polygons

Figure 5: Optimal configurations found after convergence for Bayesian Optimization



in getting the most video transcoding for a given cost. The c5.large machine was significantly better than the next best option, the n1-highcpu-2 ($P < .001$). Amazon EC2's machine types consistently outperformed Google Compute Engine's equivalents of the same category and vCPU number at both raw score (Anova, $F = 32111.456, P < .001$) and value for money (Anova, $F = 22710.7, P < .001$). However, despite this general trend, the provider seemed to be the least important factor in determining the optimal configuration. For example, the n1-highcpu-2 still gave significantly better values than the m5.large ($P < .001$), or the c5.xlarge ($P < .001$) making it the second most cost-efficient option.

With the exhaustive search complete, we then used a Spearmint based searcher to perform a Bayesian Optimization search, assuming low noise (-1 to 1) and using the same stopping conditions as used by default in the Cherrypick paper [**?**], namely when the Expected improvement (EI) is less than 10%, and at least 6 samples have been taken. We were able to successfully run this experiment with both multiple and single providers, as well as with both single jobs and multiple concurrent jobs. The results from these experiments are shown in figure 5, while examples of the job paths taken during them are shown in figure 6.

With only a single concurrent job running, we were able to replicate the previous results, with the correct optimal instance predicted in 8 out of 10 evaluations. It is interesting to note, however, that the incorrectly predicted instance in both failed cases was not the second but third best choice, resulting in a reduction in the score/cost value by 21.3%.

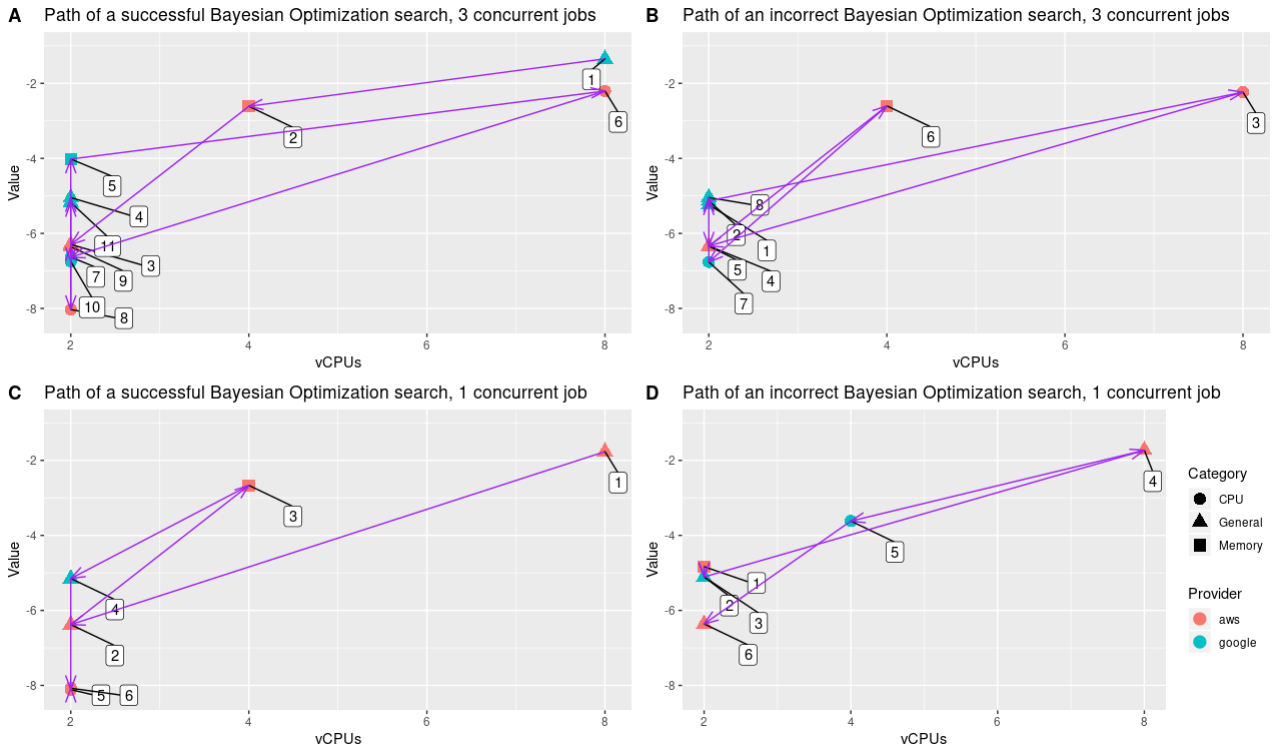


Figure 6: Paths of example Bayesian Optimization jobs

While running multiple concurrent jobs could dramatically decrease the search time, it did come at a cost to accuracy. With the same stopping conditions, a successful prediction was made only 12 out of 20 experiments (60%). This suggests that multiple concurrent jobs is only a good choice when reducing search time is more important than reducing search cost, as more stringent stopping conditions should be used. Unsurprisingly, reducing the search space to a single provider increased the likelihood of making correct predictions with multiple concurrent jobs, leading to 15 correct predictions out of 20 repeats (75%).

Having performed evaluation on our implementation for a deployment of a simple docker container, effectively corresponding to running a single batch job or benchmark and interpreting the results, we then wanted to evaluate the same technique applied to an web-based application, which may be better evaluated through its responses to a client. For this a single 5-node Kubernetes cluster was set up for the same of sending repeated requests to the evaluated deployment, as described for the 'Pingserver' deployer and interpreter. This experiment is much more intensive to perform exhaustive search for, as it would require separate pinging clusters to be set up for every sample. The mean response time for requests of a normally distributed load was divided by the hourly cost of the instance to give maximised value. From the samples taken during 10 repetitions of this experiment, it seemed that there were no significant

difference between the two optimal configurations of c5.large and m5.large ($\Delta\bar{x} = 0.0001, P \approx 1.000$), which the predictions correctly converged upon in all cases.

6.1 Related Work

7 Critical discussion

You should evaluate your own work with respect to your original objectives. You should also critically evaluate your work with respect to related work done by others. You should compare and contrast the project to similar work in the public domain, for example as written about in published papers, or as distributed in software available to you.

As mentioned, the evaluation above likely does not correspond to comparisons with which to base real deployment decisions on. In reality, a small increase in transcoding speed may lead to a far greater increase in customer uptake, rather than the effectively 1:1 ratio between price and transcoding speed assumed in the experiment. However, the evaluation shows that the methodology works very well with a given objective score measure, and it would be trivial for a new objective function to be implemented with a different relationship between the score, price, and 'value' of a given configuration.

7.1 Future extensions

8 Conclusions

You should summarise your project, emphasising your key achievements and significant drawbacks to your work, and discuss future directions your work could be taken in.

References

[1] J. Nocedal and S. Wright, *Numerical Optimization 2nd Ed.* No. 9781447122234 in Springer Series in Operations Research and Financial Engineering, Springer New York, 2006.

Appendices

Testing Summary

User Manual