IMPROVING APPLICATION INFRASTRUCTURE PROVISIONING USING RESOURCE USAGE PREDICTIONS FROM CLOUD METRIC DATA ANALYSIS

by

MAHESH HARIHARASUBRAMANIAN

A thesis submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Maria Striki

and approved by

New Brunswick, New Jersey

October, 2018

ABSTRACT OF THE THESIS

IMPROVING APPLICATION INFRASTRUCTURE PROVISIONING USING RESOURCE USAGE PREDICTIONS FROM CLOUD METRIC DATA ANALYSIS

by MAHESH HARIHARASUBRAMANIAN

Thesis Director: Maria Striki

There has been a huge interest by companies to utilize the cloud for their day-to-day operations. Cloud providers like AWS, Microsoft Azure, Google have been quite successful in serving its ever-increasing customer base. It is interesting to study how these companies use the cloud metrics to efficiently schedule their customers' jobs and thereby utilize the shared infrastructure effectively. A lot of research has been done with the Google cloud cluster data released publicly in 2011 to analyze the task and job failure rates and predict failures thereby optimizing the resource utilization by smart scheduling techniques. 6 years from then, Microsoft Azure has also released their VM CPU utilization data publicly in October 2017 along with the SOSP 2017 paper called "Resource Central". We will be one of the first to analyze this data set. In this work, we analyze this data and try to answer the following questions:

1. What are the VM CPU usage patterns by Azure subscribers?

2. Can we predict the future usage if yes, how and who all can benefit from this data?

3. Which techniques among statistical machine learning and deep learning are most suited to the Microsoft Azure data?
4. Can the learning models so formed be generalized for other similar data sets and problems like anomaly detection using log analysis at the application level?
5. How can these models augment the performance of existing VM scheduling algorithms?

Acknowledgements

I would first like to thank my thesis advisor Prof. Maria Striki, faculty in the Electrical and Computer Engineering (ECE) Department at the Rutgers, The State University of New Jersey, for her immense support throughout this journey. Be it for the courses that I have taken under her or research guidance for this thesis, I have always received encouragement, motivation and advice that will help me not only in my professional but also my personal life.

I would also like to thank Dr. Zoran Gajic, professor and graduate director of the ECE department at Rutgers, The State University of New Jersey for his guidance which helped me plan my coursework and graduation time-line. The ECE department faculty also deserves credit since my graduate courses were also helpful in gaining new insights to my research problem. I thank the rest of my thesis committee: Dr. Athina Petropulu and Dr. Yingying Chen to have taken time out of their busy schedules to review, critique and share their feedback. It also feels great to have ever helpful ECE administrative staff like Christy Lafferty, Arletta Hoscilowicz and John Scaffidi assisting students like me with our doubts and clarifications.

It was also a great learning experience during my internship with New York Property Insurance Underwriting Association [1] where I was exposed to real world software application workloads, enterprise software architecture, infrastructure provisioning and setting up jobs using a workload automation tool called BMC Control-M. [2]

A big thank you to all my friends, roommates, colleagues, on-campus job supervisors to have eased my stay at Rutgers as an international student by helping out when in need.

Finally, I do not want to forget to express my profound gratitude to my mother Padmavathi Hariharasubramanian, my father Hariharasubramanian Kuppusamy and my

elder sister Rama Hariharasubramanian who have always backed me and also without whose support I could not have afforded to pursue a Master's degree.

Table of Contents

List of Tables

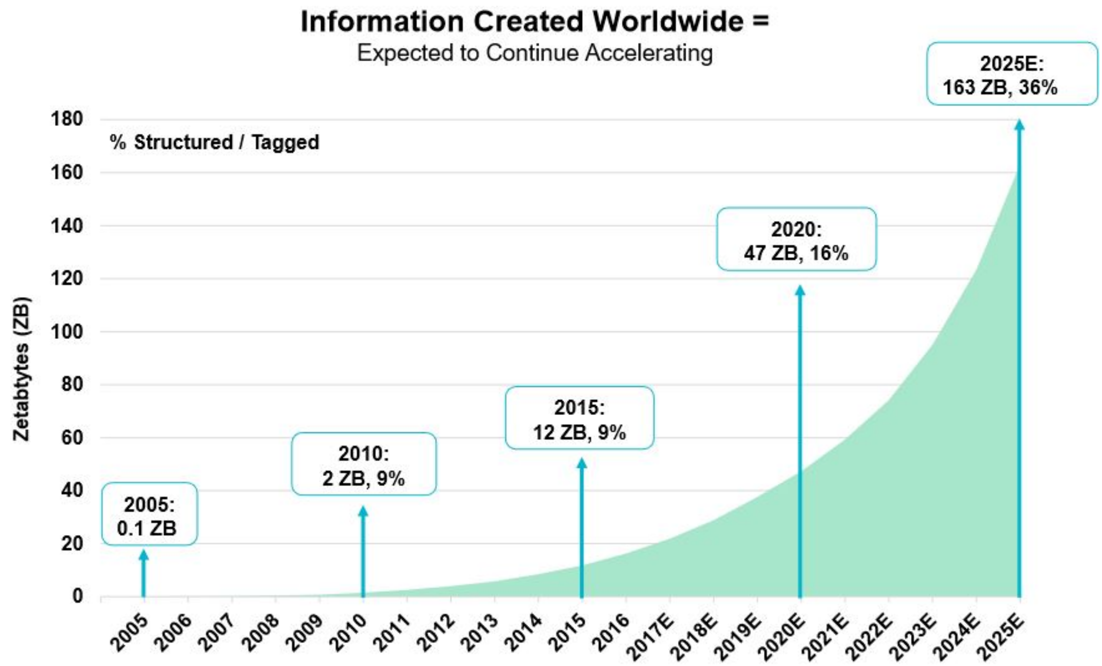
List of Figures

Chapter 1

Introduction

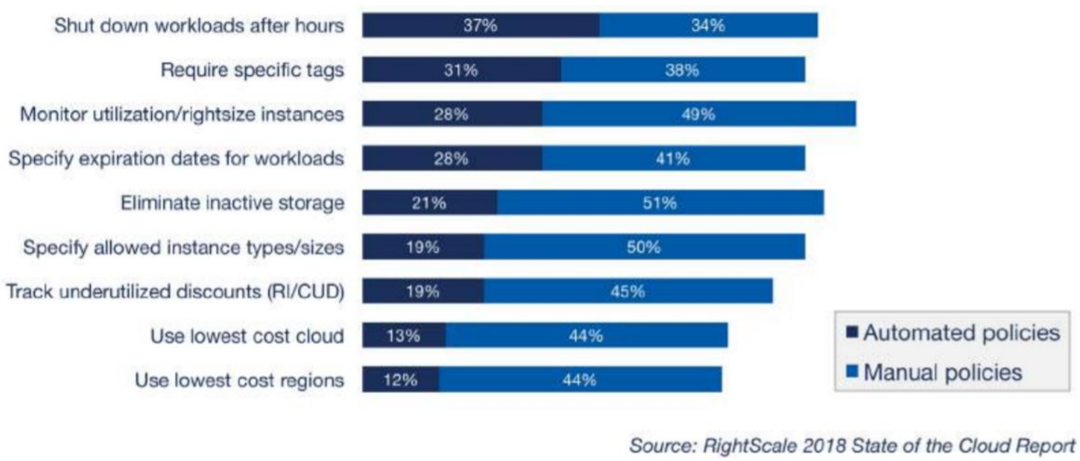
Computers have revolutionized the way we do things today. Starting from the dot com boom [5] in the 1990s to the cloud computing and machine learning/ artificial intelligence boom that we see today, technology has never been this dynamic and interesting. There has been a lot of investment of time and efforts in increasing the efficiency of data centers which store the ever increasing data. Even though a data center is efficient, 42% power is said to be used by the air conditioning systems according to [6]. While this paper talks about solar powered data centers being the future of the infrastructure fabric to save energy, our work is about how we can gain valuable insights out of cpu usage data. Our research is valuable because CPU utilization is one of the key contributors to heat generated in a datacenter as indicated by the energy consumption modeling equations in this paper [7].

The main driver of data analysis is data. In order for the machine learning learning tools to show useful results, it needs lot of training data. This training data is something that we extract and massage from a much large source data set. 90% of the data has been created over the last 20 years and it is expected to grow in the near future as shown in Figure 1.1. This requires sophisticated CPU processing, distributed computing, storage equipments programming methodologies and tools. We have evolved a lot to be able to manufacture efficient storage devices to store such excessive amounts of data. At the same time GPU processing power has also been improving at an exponential rate (to be confirmed). All these technologies put together has set the stage just right for gaining value out of massive data analysis.



Data Volume Growth graph from Internet Trends 2017- Code Conference [3]

According to the RightScale 2018 State of the Cloud Report [4], the biggest challenge for beginner cloud adopters is security, but as they deal with it over time, the top challenge becomes optimizing for costs. These mature cloud users are known to waste 35%, approximately $10 billion [8] of the cloud bills due to their inefficiencies and very few companies are making efforts to employ automated policies to monitor the usage and optimize cloud costs. Figure 1.2 shows the currently employed methodologies to save cloud usage costs. Our thesis work aims to help such companies monitor their cloud resource utilization, gain insights and take corrective actions accordingly.



How Companies are Optimizing Cloud Cost [4]

Before the era of cloud computing, it was virtualization that helped companies efficiently utilize the datacenters. Virtualization is still a key aspect of Cloud computing. According to a technical report by VMWare [9], private cloud can be cheaper than public cloud. Private cloud is a recommended option if you are looking for speed, agility and efficiency along with maintaining security of sensitive workloads and data governance. [10]. But the major cloud providers have been working towards these challenges of security and data governance with commendable success. It is evident that we are going to make these third party cloud providers richer in the years to come, but we can control the expenditure if we can see for ourselves how much resources we will require in the future and provision VMs accordingly from the cloud service providers. The usual practice of always over provisioning to be safe can be needless if we can accurately estimate our requirement during the initial few runs of the workload. So, be it public cloud, your own private cloud or hybrid cloud, collecting metrics and using it for better resource usage is a rewarding choice.

1.1 Motivation

Due to rapid research in the machine learning and artificial intelligence field, it is faster and easier to apply machine learning models to the data collected. Libraries in Python and frameworks like TensorFlow and Keras provide higher level abstractions making it easier for all kinds of users, developers as well as business analysts. Companies can start collecting metrics from their own infrastructure or applications and analyze them using appropriate machine learning algorithms and greatly improve performance by later moving the workload to any of the trusted third party cloud service providers.

Our end goal is to apply analysis, similar to ours, to other type of datasets that can be collected and stored by the information technology departments of enterprises. Examples of these include and are not limited to logs [11] [12] from application servers, web servers, transactional data and so on. The results of such analysis can be used to augment existing scheduling algorithms [13].

1.2 Contribution

We first analyze the overall dataset of Microsoft Azure, then further analyze a particular virtual machine's CPU usage details to predict the future usage patterns. This is done using the RNNs with LSTMs.

We further analyze all the VMs data irrespective of which customer it belonged to. We wanted to see if there exists any characteristic difference in the usage metrics for those classified as delay insensitive as opposed to delay sensitive VMs. We use the correlation values for the features chosen to share our findings.

1.3 Outline

Chapter 2 includes the literature suryvey that describes the previous work done in this field. Chapter 3 discusses the Azure Public Data data released by Microsoft with the help of a Jupyter Notebook. Chapter 4 describes our work analyzing this data and making prediction models for CPU utilization using regression and LSTM RNN. We discuss the results and future work in Chapter 5. We finally summarize and conclude in chapter 6.

When we say machine learning we may use it to collectively imply both statistical techniques as well as deep learning techniques leveraging neural networks.

Chapter 2

Related Work/ Literature survey

In this section, we go through some of the extensive work done by fellow researchers in the field of cloud trace analysis, CPU usage prediction, Virtual Machine scheduling and Key Performance Indicators (KPIs) in cloud computing. Having done a thorough review of such work, it gives us scope and foundation to build on it and explore new avenues along with the new developments in the field of cloud computing.

2.1 Cluster Trace Analysis

The Google cluster trace contains data [14] collected from 12500 machines running for about a month in May 2011. Many researchers have explored the machine, jobs and tasks information to come up with job failure prediction or resource usage prediction models. One such work is Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study [15] where they predict application failures using the resource usage data from the Google cluster traces. They leverage the Recurrent Neural Networks $RNNs$ for the analysis and quote 6% to 10% savings on resource.

Analysis and Lessons from a Publicly Available Google Cluster Trace [16], a technical report published by the University of California, Berkeley shares the statistical profile of the Google trace along with many key metrics like job arrival patterns, CPU and memory usage and task durations to name a few. This analysis was intended to aid system designers with capacity planning and system tuning.

Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations [17] analyses the Google cluster trace, CMU OpenCloud [18], LANL HPC Cluster [19] datasets. They first study the data to detect patterns of unsuccessful jobs with respect to their resource consumptions and job configuration. Further, they propose and demonstrate a machine learning framework to predict job and task terminations which can be useful in terminating tasks that are going to fail much earlier thus saving on cloud resource consumption.

Another work based on the Google cluster trace data is Predicting Scheduling Failures in the Cloud [20], where they use statistical models to predict task failures, so that they can be rescheduled earlier thus saving on the cluster resources.

2.2 Virtual Machine (VM) consolidation

LiRCUP Linear regression based CPU usage prediction algorithm for live migration of virtual machines. [21] focuses on the reducing SLA (Service Level Agreement) violation and cutting down power costs from the data center. To achieve this, a CPU usage prediction using linear regression technique is proposed. This approach when applied in the VM live migration process to identify under-loaded and over-loaded hosts is said to reduce the energy consumption and SLA violation significantly.

Some papers have considered the VM consolidation as a bin or vector packing NP-hard optimization problem. [22] [23] [24]. The bins are server hosts (or data centers) with different capacities whereas the different size objects that need to be fit are the virtual machines (or servers). The most optimal solution is the least number of bins that can fit all the various objects which implies more efficient energy consumption and reduced running costs.

One of a research paper coming out of IBM Watson Research center [25], talks about how dynamic server consolidation technique using forecasting based on time series analysis of historical data helps reduce SLA violations.

Virtual Machine Consolidation with Usage Prediction for Energy-Efficient Cloud Data Centers [26] talks about VM consolidation but with help of multiple resource prediction rather than just a single metric like CPU utilization. They use the current usage data as well as the historical data together to characterize overloaded and underloaded servers which can then be used for reducing the load and power consumption after the

VM consolidation process. They also compare their VM consolidation with multiple usage prediction (VMCUP-M) with BG(black box gray box) algorithm described in [22] and claim that VMCUP-M is better in terms of number of server switches and SLA compliance.

Psychas and Ghaderi's paper [13] takes the problem of multi-resource jobs scheduling further by taking the queuing theory approach. Using the Google cloud cluster trace, they prove that their proposed randomized scheduling algorithm for placing jobs in servers achieves better throughput and low computational complexity as opposed to Bin Packing and Max Weight alternate solutions.

Some of the recent work make use of neural networks too. Once such paper is [27]. It makes use of Long Short-Term Memory Recurrent Neural Network (LSTM RNN) to predict CPU usage values and avoid slash-dot effects. This is useful in auto scaling virtual resources thus reducing the slash-dot effects in cloud.

2.3 KPIs (Key Performance Indicators)

Key Performance Indicators for Cloud Computing SLAs [28] proposes some of the KPIs that needs to be considered while trying to adhere to the SLA agreed upon between the customer and the cloud provider. It also explains the general SLA life cycle and provide five KPI categories namely the General Service, Network Service, Cloud Storage, Backup and Restore and Infrastructure as a Services (IaaS).

SLA Violation Prediction In Cloud Computing: A Machine Learning Perspective: There is some work done to predict the SLA violation before it actually occurs. One of them is [29]. It uses Naive Bayes and Random Forest Classifiers to predict SLA violations from the re-sampled Google cluster dataset.

Performance Challenges in Cloud Computing [30] gives an overview of the obstacles and opportunities with respect to Cloud Computing. They put the SLAs into five service-level categories: Availability, Performance, Capacity, Reliability, Scalability and explain the importance of performance engineering and capacity management of cloud environments.

We see a more systematic and mathematical approach in [31] regarding optimizing performance of Cloud Computing centers and coming up with the KPIs. It also uses a queuing theory to model the load balancer and data center to do performance analysis. Finally, it gives a balanced scorecard for KPIs across four domains: financial, customer, process, innovation and learning.

Chapter 3

Azure Public Dataset

For any kind of machine learning techniques to be applied, we need to have data. This is not enough alone. This data needs to be in a particular format and needs to be filtered so that it has only the features that matter to our end goal. Having this readily available expedites the research to a great extent. The challenge with obtaining cloud workload trace data is its relevance and concealing of private information about the customers. As we have seen in Chapter 2, most of the research has been done on the Google Cluster workload traces [14]. This contains a month long period trace of around 12500 machines and is a valuable data to be worked with for machine learning purposes. Since this data contains information for machines during May 2011, we were looking for sources for more recent trace data which will also count for the recent developments in the cloud services domain. Microsoft's Azure VM Trace Data [32] provides us just that, the trace data being released last year, i.e. October 2017. So, we decided to work with this dataset and explore the possibilities to use some of the modern machine learning and deep learning techniques to predict the future resource usage values. This data was released as part a paper titled "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms" [33].

3.1 Main Characteristics

It should be noted that the Azure Public Dataset is sanitized representative subset of the actual first-party VM workload of Microsoft Azure in a particular geographical region. Even then this public data is said to exhibit same overall trends as the full original dataset used for the paper [33].

3.2 First glance at the Azure public data

The Azure public dataset has information of about 2013767 VMs.The details of each of the VM is provided in the vmtable.csv file as shown in 3.2.3

```
In [1]: import numpy as np
import pandas as pd
from IPython.display import display
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
```

```
/root/anaconda3/envs/my_env/lib/python3.6/site-packages/h5py/__init__.py:36:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
from ._conv import register_converters as _register_converters
```

3.2.1 subscriptions.csv

```
In [2]: data_path = 'data/subscriptions.csv'
headers=['subscriptionid','timestamp first vm created','count vms created']
subscriptions_df = pd.read_csv(data_path, header=None,
index_col=False,names=headers,delimiter=',')
subscriptions_df.head()
```

```
Out[2]:                                       subscriptionid  \
0  ++0v0Ny4Fe3c5KwLPfu0Z9o0oUwYS9s0oRfnPDP4EFfkhI...
1  +/7swedsYYdH5dEQHZkMkq/z6yFfYCU7RmWNUI0WvgV3q7...
2  +0QrL7710PkahCGSRRh9aJ5m+04stqyoSy8VZ2gDPGXBnY...
3  +1/71ic//xJ3Q8kuoCIJeSaRBKsxEq0Fa0r7RLAyG9M7CI...
4  +1/Tx/1utSDCXsZsojwxMiw4iDwne0iSwceIdym1oihZST...

   timestamp first vm created  count vms created
0                           0                 11
1                           0                220
```

2	0	5
3	0	62
4	0	17

```
In [3]: subscriptions_df.describe()
```

Out[3]:

	timestamp first vm created	count vms created
count	5.958000e+03	5958.000000
mean	1.033952e+05	337.993790
std	3.857817e+05	3562.628063
min	0.000000e+00	1.000000
25%	0.000000e+00	2.000000
50%	0.000000e+00	10.000000
75%	0.000000e+00	49.000000
max	2.587200e+06	128047.000000

3.2.2 deployment.csv

```
In [4]: data_path = 'data/deployment.csv'
headers=['deployment id','deployment size']
deployment_df = pd.read_csv(data_path, header=None,
index_col=False,names=headers,delimiter=',')
deployment_df.head()
```

Out[4]:

	deployment id	deployment size
0	++mNOLJv64R5/YiUoKgebbCmqww1BIDN2kptt14qjuTOpW...	4
1	+/vGhVS4Q5V4gdBh6Z7eZimqTcgIn5i13AG8dHyxV1brIy...	12
2	+0d60C0i0UG5ZMhf5fpjW5p7x/kuY9JndgnDh3AjDWmlt9...	1
3	+0xehJM4d+6XRcUXYJdXghnRZaQdiAO2cFyEbWwge953Ok...	3
4	+1KRd7Z8ixfESvcHj0omEBWt1HFg82wqEyWq04OmiOusnk...	1

```
In [5]: deployment_df.describe()
```

Out[5]:

	deployment size
count	35941.000000

```
mean          18.063243
std           65.145059
min            1.000000
25%            2.000000
50%            4.000000
75%           14.000000
max         1814.000000
```

3.2.2.1 deployment query for VM0

```
In [6]: dep4VM0 = deployment_df.loc[deployment_df['deployment id'] == 'Pc2VLB8aDxK2DCC96
itq4vW/zVDp4wioAUiB3HoGSFYQ0o6/ZCegTpb9vEH4LeMTEWVObHTPRYEY81TYivZCMQ==']
dep4VM0
```

```
Out[6]:   deployment id  deployment size
15576  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...                2
```

3.2.3 vmtable.csv

Total: 2013767 VMs

```
In [7]: data_path = 'data/vmtable.csv'
headers=['vmid','subscriptionid','deploymentid','vmcreated', 'vmdeleted',
'maxcpu', 'avgcpu', 'p95maxcpu', 'vmcategory', 'vmcorecount', 'vmmemory']
vmtable_df = pd.read_csv(data_path, header=None,
index_col=False,names=headers,delimiter=',')
vmtable_df.head()
```

```
Out[7]:                                                vmid  \
0  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
1  H5CxmMoVcZSpjgGbohnVA3R+7uCTe/hM2ht2uIYi3t7KwX...
2  wR/G1YUjpMP4zUbxGM/XJNhYS8cAK3SGKM2tqhF7VdeTUY...
3  1XiU+KpvIa3T1XP8kk3ZY71Of03+ogFL5Pag9Mc2jBuh0Y...
4  z5i2HiSaz6ZdLR6PXdnDjGva3jIlkMPXx23VtfXx9q3dXF...

subscriptionid  \
```

```
0   VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
1   BSXOcywx8pUU0DueDo6UMol1YzR6tn47KLEKaoXp0a1bf2...
2   VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
3   8u+M3WcFp8pq183WoMB79PhK7xUzbaviOBv0qWN6Xn4mbu...
4   VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...

deploymentid  vmcreated  vmdeleted  \
0   Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...        0    2591700
1   3J17LcV4gXjFat62qhVFRfoiWArHnY763HVqqI6orJCfV8...        0    1539300
2   Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...  2188800    2591700
3   DHbeI+pYTYFjH8JAF8SewM0z/4SqQctvxcBRGIRglBmeLW...        0    2591700
4   Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...        0    2188500

maxcpu     avgcpu  p95maxcpu         vmcategory  vmcorecount  vmmemory
0    99.369869   3.424094  10.194309  Delay-insensitive            1      1.75
1   100.000000   6.181784  33.981360        Interactive            1      0.75
2    99.569027   3.573635   7.924250  Delay-insensitive            1      1.75
3    99.405085  16.287611  95.697890  Delay-insensitive            8     56.00
4    98.967961   3.036038   9.445484  Delay-insensitive            1      1.75
```

```
In [8]: vmtable_df.describe()
```

```
Out[8]:    vmcreated     vmdeleted        maxcpu        avgcpu     p95maxcpu  \
count  2.013767e+06  2.013767e+06  2.013767e+06  2.013767e+06  2.013767e+06
mean   1.318097e+06  1.504673e+06  7.233878e+01  1.513443e+01  5.901889e+01
std    7.806214e+05  7.535789e+05  3.385232e+01  1.534389e+01  3.561021e+01
min    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
25%    6.573000e+05  8.880000e+05  4.918151e+01  3.438740e+00  2.509601e+01
50%    1.411200e+06  1.571100e+06  9.087399e+01  1.045594e+01  6.931068e+01
75%    1.990800e+06  2.155350e+06  9.856823e+01  2.136583e+01  9.437023e+01
max    2.591700e+06  2.591700e+06  1.000000e+02  1.000000e+02  1.000000e+02
```

	vmcorecount	vmmemory
count	2.013767e+06	2.013767e+06
mean	2.563461e+00	6.069098e+00
std	2.380495e+00	1.062382e+01
min	1.000000e+00	7.500000e-01
25%	1.000000e+00	1.750000e+00
50%	2.000000e+00	3.500000e+00
75%	4.000000e+00	7.000000e+00
max	1.600000e+01	1.120000e+02

3.2.3.1 Some simple vmtable queries

3.2.3.2 Given deployment id find all VM data

```
In [9]: deps4VM0 = vmtable_df.loc[vmtable_df['deploymentid'] == 'Pc2VLB8aDxK2DCC96itq4vW
/zVDp4wioAUiB3HoGSFYQ0o6/ZCegTpb9vEH4LeMTEWVObHTPRYEY81TYivZCMQ==']
deps4VM0
```

```
Out[9]:                                                     vmid  \
0  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
4  z5i2HiSaz6ZdLR6PXdnDjGva3jIlkMPXx23VtfXx9q3dXF...

subscriptionid  \
0  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
4  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...

deploymentid  vmcreated  vmdeleted  \
0  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...          0    2591700
4  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...          0    2188500

maxcpu    avgcpu  p95maxcpu          vmcategory  vmcorecount  vmmemory
0  99.369869  3.424094  10.194309  Delay-insensitive            1      1.75
4  98.967961  3.036038   9.445484  Delay-insensitive            1      1.75
```

3.2.3.3 Given subscription id find all VM data

```
In [10]: subs4VM0 = vmtable_df.loc[vmtable_df['subscriptionid'] ==
'VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKR0uxPYysA2XYii9Y5ZkaYaq']
subs4VM0
```

```
Out[10]:                                                 vmid  \
0  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
2  wR/G1YUjpMP4zUbxGM/XJNhYS8cAK3SGKM2tqhF7VdeTUY...
4  z5i2HiSaz6ZdLR6PXdnDjGva3jIlkMPXx23VtfXx9q3dXF...

subscriptionid  \
0  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKR0uxP...
2  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKR0uxP...
4  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKR0uxP...

deploymentid  vmcreated  vmdeleted  \
0  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...          0    2591700
2  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...    2188800    2591700
4  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...          0    2188500

maxcpu    avgcpu  p95maxcpu         vmcategory  vmcorecount  vmmemory
0  99.369869  3.424094  10.194309  Delay-insensitive            1      1.75
2  99.569027  3.573635   7.924250  Delay-insensitive            1      1.75
4  98.967961  3.036038   9.445484  Delay-insensitive            1      1.75
```

3.2.4 vm_cpu_readings-file-1-of-125.csv

```
In [11]: data_path1 = 'data/vm_cpu_readings-file-1-of-125.csv'
headers=['timestamp','vm id','min cpu','max cpu', 'avg cpu']
cpu_readings1_df = pd.read_csv(data_path1, header=None,
index_col=False,names=headers,delimiter=',')
cpu_readings1_df.head()
```

```
Out[11]:    timestamp   vm id   min cpu  \
```

```
0          0  +ZcrOp5/c/fJ6mVgP5qMZlOAGDwyjaaDNMOWoWOt2IDb47...  2.052803
1          0  2zrgeOqUDy+lOGVi5NXudU+3sqZH+nLowfcz+D/JsCymTX...  1.646950
2          0  /34Wh1Kq/qkNkWOtQrMiQ1eZ8hg9hHopydCzsXriefhgrn...  2.440088
3          0  2lzdXk1Rqn1ibH2kZhGamYTMvVcRP6+x8b5zGiD/8t++5B...  0.302992
4          0  OGrUQuLhCER5bWWcoJAgblPJWkaU4v3nf+NUrZnFTlXWEK...  1.515922

max cpu   avg cpu
0  3.911587  2.869790
1  8.794403  3.254472
2  6.941048  4.336240
3  2.046712  0.970692
4  4.471657  2.438805
```

```
In [12]: cpu_readings1_df.tail()
```

```
Out[12]:     timestamp   vm id  \
9999995      21000  fVp1qu/0+CVls8myZE8x4cnUY6KqEU1UWVlm4vGhB2eVXw...
9999996      21000  gNEa2Sfj/+AFkdqegJKBYEiNlWuuxeFuDA7a0ncu8o2pQN...
9999997      21000  focrLM5imiwU46iidLwZFnB+3fJOO3uhC4zxtLlf7pprCX...
9999998      21000  e5B7mXjS3G5I+/p06MN3HygpmbXwH4azc80u+80ZUhWnqY...
9999999      21000  hipzzpSIL409YN+4/YKAfHnPrFXl+yVPi78BGN26nx890+...

min cpu    max cpu   avg cpu
9999995  0.013784   3.082615  0.640116
9999996  1.229370   4.126575  2.469720
9999997  3.399143   5.843358  4.583291
9999998  5.915925  14.521810  8.124820
9999999  0.477585  16.732999  4.632007
```

3.2.5 vm_cpu_readings-file-2-of-125.csv

```
In [13]: data_path2 = 'data/vm_cpu_readings-file-2-of-125.csv'
headers=['timestamp','vm id','min cpu','max cpu', 'avg cpu']
```

```
cpu_readings2_df = pd.read_csv(data_path2, header=None,
index_col=False,names=headers,delimiter=',')
cpu_readings2_df.head()
```

```
Out[13]:    timestamp   vm id   min cpu  \
0      21000  i602h+dF+Y8o9SXlfrlY50nbfF6umaHHWcOG3ay28+xFZA...  0.585760
1      21000  jX+l+6KyENIjUls2xo1hfHgswnIb+odiW8O8cnUCEDgjMn...  2.998676
2      21000  hUgkAZ1yIFuiry7fA9KHjY4uRPVETFNGTcq1TeZeSD91DT...  2.900732
3      21000  jhdxmeolVH9yBwbBFDFxBTMBgnu3cgmP2USpFA2X0NHa2a...  2.949930
4      21000  efWn5J2FyxlU+uTRtOC+ZpN88xl21Te3DpdebOgUNX5lpD...  1.847758

max cpu    avg cpu
0   2.691011   1.387735
1  34.716643   7.888491
2  48.581511  16.226454
3   6.818615   4.039858
4   7.923823   3.337690
```

```
In [14]: cpu_readings2_df.tail()
```

```
Out[14]:    timestamp   vm id  \
9999995      42300  V8f/xbizJrnsaGpkdXp3Llz7PoZCM3h1CPIq83Uyc8ez+2...
9999996      42300  SwjP42rY7ZaLDIcXeB/iUdzuzl93zTWc6NfwxfKKkukzOf...
9999997      42300  XCG6q9qc+9xeNRyWdxo+TKTEGAr+o3VechOegPo4LvCrd9...
9999998      42300  RkWiD2oOfhpaq2UaSmX6x4fa404FQ/erR70cReBIABb1CQ...
9999999      42300  YGp4BwhcSW06neLGPgucAJAQGbBra6bBAdjdzAwB0Fa7u6...

min cpu    max cpu    avg cpu
9999995  0.569823   2.471833   1.123275
9999996  0.381573   0.461752   0.421770
9999997  0.460615   2.847367   1.325941
9999998  0.622075   3.071922   1.056286
9999999  0.449021  58.039647  12.905890
```

3.2.6 CPU Readings every 5 minutes

3.2.6.1 CPU readings for VM0 from file 1

```
In [15]: vm0_cpu_readings1 = cpu_readings1_df.loc[cpu_readings1_df['vm id'] ==
'x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mqbBOe3wxBlQo0+Qx+uf']
vm0_cpu_readings1.head()
```

```
Out[15]:    timestamp   vm id  \
137885          0   x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
282440        300   x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
424613        600   x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
565644        900   x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
714365       1200   x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...

min cpu   max cpu   avg cpu
137885  2.755203  4.391175  3.420025
282440  2.786546  4.339331  3.267826
424613  2.878095  6.680684  3.635158
565644  2.327555  4.282121  3.367871
714365  2.925309  6.182837  3.601547
```

```
In [16]: vm0_cpu_readings1.count()
```

```
Out[16]: timestamp    70
vm id        70
min cpu      70
max cpu      70
avg cpu      70
dtype: int64
```

3.2.6.2 CPU readings for VM0 from file 2

```
In [17]: vm0_cpu_readings2 = cpu_readings2_df.loc[cpu_readings2_df['vm id'] ==
'x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mqbBOe3wxBlQo0+Qx+uf']
vm0_cpu_readings2.head()
```

```
Out[17]:     timestamp   vm id  \
33526        21000  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
178273       21300  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
312333       21600  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
455262       21900  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
595134       22200  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...

min cpu   max cpu   avg cpu
33526   2.666365  4.209862  3.330772
178273  2.759352  4.081712  3.198532
312333  2.831082  4.460893  3.427262
455262  2.560338  4.127717  3.249396
595134  2.909225  4.583207  3.392063
```

```
In [18]: vm0_cpu_readings2.count()
```

```
Out[18]: timestamp    71
vm id        71
min cpu      71
max cpu      71
avg cpu      71
dtype: int64
```

Chapter 4

Analysis of the Data

This chapter describes the process of how we analyzed and got the results in a step by step manner.

4.1 Techniques, Tools and Technologies

We used Jupyter Notebook with Python to run the entire analysis. Python supports a lot of useful frameworks like Pandas for data cleaning and machine learning libraries such as Keras and Tensorflow.

4.2 Digging into the Vmtable.csv dataset

Import python libraries

```
In [1]: import numpy as np
import pandas as pd
from IPython.display import display
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
```

```
/root/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:34: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floating` is
deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
from ._conv import register_converters as _register_converters
```

Developing feature columns for the data in vmtable.csv

We get the VM lifetime in hours by subtracting the 'vmcreated' value from 'vmdeleted' value and dividing by 3600. We then, populate a new column called "core-hour" by multiplying this lifetime with the vmcorecount value.

```
In [2]: data_path = 'data/vmtable.csv'
headers=['vmid','subscriptionid','deploymentid','vmcreated', 'vmdeleted',
'maxcpu', 'avgcpu', 'p95maxcpu', 'vmcategory', 'vmcorecount', 'vmmemory']
vmtable_df = pd.read_csv(data_path, header=None,
index_col=False,names=headers,delimiter=',')
vmtable_df['lifetime'] = np.maximum((vmtable_df['vmdeleted'] -
vmtable_df['vmcreated']),300)/ 3600
vmtable_df['corehour'] = vmtable_df['lifetime'] * vmtable_df['vmcorecount']
vmtable_df = vmtable_df.drop(['vmcreated','vmdeleted','vmcorecount','lifetime'],
axis=1, inplace=False)
vmtable_df.head()
```

```
Out[2]:                                               vmid  \
0  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
1  H5CxmMoVcZSpjgGbohnVA3R+7uCTe/hM2ht2uIYi3t7KwX...
2  wR/G1YUjpMP4zUbxGM/XJNhYS8cAK3SGKM2tqhF7VdeTUY...
3  1XiU+KpvIa3T1XP8kk3ZY71Of03+ogFL5Pag9Mc2jBuh0Y...
4  z5i2HiSaz6ZdLR6PXdnDjGva3jIlkMPXx23VtfXx9q3dXF...

                                      subscriptionid  \
0  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
1  BSXOcywx8pUU0DueDo6UMol1YzR6tn47KLEKaoXp0a1bf2...
2  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
3  8u+M3WcFp8pq183WoMB79PhK7xUzbaviOBv0qWN6Xn4mbu...
4  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...

                                        deploymentid      maxcpu     avgcpu  \
0  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   99.369869   3.424094
1  3J17LcV4gXjFat62qhVFRfoiWArHnY763HVqqI6orJCfV8...  100.000000   6.181784
2  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   99.569027   3.573635
3  DHbeI+pYTYFjH8JAF8SewM0z/4SqQctvxcBRGIRglBmeLW...   99.405085  16.287611
4  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   98.967961   3.036038
```

```
p95maxcpu         vmcategory  vmmemory     corehour
0  10.194309  Delay-insensitive      1.75   719.916667
1  33.981360        Interactive      0.75   427.583333
2   7.924250  Delay-insensitive      1.75   111.916667
3  95.697890  Delay-insensitive     56.00  5759.333333
4   9.445484  Delay-insensitive      1.75   607.916667
```

Clean up data

```
In [3]: vmtable_df.shape
```

```
Out[3]: (2013767, 9)
```

```
In [4]: vmtable_df.isnull().values.any()
```

```
Out[4]: False
```

4.2.1 Correlation of the whole vmtable.csv data

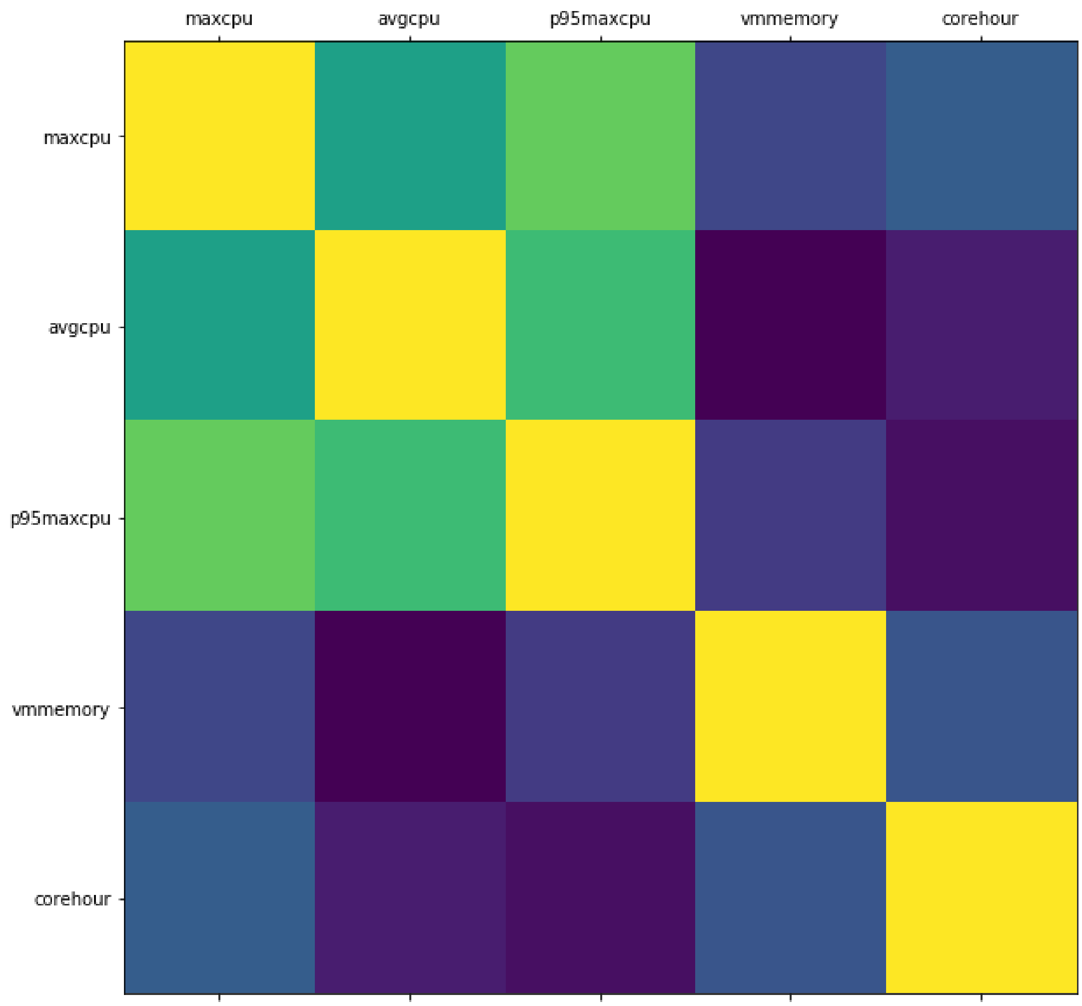
```
In [5]: def plot_corr(df,size=10):
'''Function plots a graphical correlation matrix for each pair of columns in
the dataframe.

    Input:
        df: pandas DataFrame
        size: vertical and horizontal size of the plot'''

corr = df.corr()
fig, ax = plt.subplots(figsize=(size, size))
ax.matshow(corr)
plt.xticks(range(len(corr.columns)), corr.columns);
plt.yticks(range(len(corr.columns)), corr.columns);
```

```
In [6]: vmtable_df.corr()
```

```
Out[6]:              maxcpu    avgcpu  p95maxcpu  vmmemory  corehour
maxcpu     1.000000  0.480313   0.718682  0.057079  0.151668
avgcpu     0.480313  1.000000   0.623686 -0.200087 -0.104058
```

```
p95maxcpu   0.718682  0.623686   1.000000  0.007796 -0.152137
vmmemory    0.057079 -0.200087   0.007796  1.000000  0.115972
corehour    0.151668 -0.104058  -0.152137  0.115972  1.000000
```

```
In [7]: plot_corr(vmtable_df)
```



Correlation plot for the entire vmtable.csv file data

```
In [8]: vmtable_df.vmcategory.unique()
```

```
Out[8]: array(['Delay-insensitive', 'Interactive', 'Unkown'], dtype=object)
```

We have three three VM categories. i.e. 'Delay-insensitive', 'Interactive', 'Unkown'. We map these strings to numbers for ease of further analysis.

```
In [9]: vm_category_map = {'Delay-insensitive':0,'Interactive':1,'Unkown':np.nan}

In [10]: vmtable_df['vmcategory'] = vmtable_df['vmcategory'].map(vm_category_map)

In [11]: vmtable_df.vmcategory.unique()

Out[11]: array([ 0.,  1., nan])

In [12]: vmtable_df.head(5)

Out[12]:                                                vmid  \
0  x/XsOfHO4ocsV99i4NluqKDuxctW2MMVmwqOPAlg4wp8mq...
1  H5CxmMoVcZSpjgGbohnVA3R+7uCTe/hM2ht2uIYi3t7KwX...
2  wR/G1YUjpMP4zUbxGM/XJNhYS8cAK3SGKM2tqhF7VdeTUY...
3  1XiU+KpvIa3T1XP8kk3ZY71Of03+ogFL5Pag9Mc2jBuh0Y...
4  z5i2HiSaz6ZdLR6PXdnDjGva3jIlkMPXx23VtfXx9q3dXF...

subscriptionid  \
0  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
1  BSXOcywx8pUU0DueDo6UMol1YzR6tn47KLEKaoXp0a1bf2...
2  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...
3  8u+M3WcFp8pq183WoMB79PhK7xUzbaviOBv0qWN6Xn4mbu...
4  VDU4C8cqdr+ORcqquwMRcsBA2l0SC6lCPys0wdghKROuxP...

deploymentid      maxcpu     avgcpu  \
0  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   99.369869   3.424094
1  3J17LcV4gXjFat62qhVFRfoiWArHnY763HVqqI6orJCfV8...  100.000000   6.181784
2  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   99.569027   3.573635
3  DHbeI+pYTYFjH8JAF8SewM0z/4SqQctvxcBRGIRglBmeLW...   99.405085  16.287611
4  Pc2VLB8aDxK2DCC96itq4vW/zVDp4wioAUiB3HoGSFYQ0o...   98.967961   3.036038

p95maxcpu  vmcategory  vmmemory    corehour
0  10.194309         0.0      1.75  719.916667
```

```
1  33.981360          1.0      0.75   427.583333
2   7.924250          0.0      1.75   111.916667
3  95.697890          0.0     56.00  5759.333333
4   9.445484          0.0      1.75   607.916667
```

```
In [13]: vmtable_df.isnull().values.any()
```

```
Out[13]: True
```

Drop rows whose VM category is unknown (null)

```
In [14]: vmtable_df = vmtable_df.dropna()
```

```
In [15]: vmtable_df.count()
```

```
Out[15]: vmid              841170
subscriptionid    841170
deploymentid      841170
maxcpu            841170
avgcpu            841170
p95maxcpu         841170
vmcategory        841170
vmmemory          841170
corehour          841170
dtype: int64
```

So we lost 2013767 - 841170 = 1172597 rows Further analysis is only on these 841170 VMs

```
In [16]: vmtable_df.dtypes
```

```
Out[16]: vmid               object
subscriptionid     object
deploymentid       object
maxcpu            float64
```

```
avgcpu            float64
p95maxcpu         float64
vmcategory        float64
vmmemory          float64
corehour          float64
dtype: object
```

Get the number of VMs for each subscriptionid,deploymentid combination sorted desc VM count

```
In [17]: sub_dep_sort_by_vm_cnt_desc_df = vmtable_df.groupby(['subscriptionid','deploymen
tid']).agg({'vmid':'count'}).sort_values('vmid',ascending=False)
```

```
In [18]: sub_dep_sort_by_vm_cnt_desc_df.head()
```

```
Out[18]:    vmid subscriptionid    deploymentid
IBRuELx83... GVubwtq7m...   13256
1pvP5oaK4... qNRw2mFob...   10604
             qNRw2mFob...   10597
             qNRw2mFob...   10586
IBRuELx83... GVubwtq7m...   10098
```

Get the number of VMs for each subscriptionid combination sorted desc VM count

```
In [19]: sub_sort_by_sub_id_desc_df = vmtable_df.groupby(['subscriptionid']).agg({'vmid':
'count'}).sort_values('vmid',ascending=False)
```

```
In [20]: sub_sort_by_sub_id_desc_df.head()
```

```
Out[20]:                                                      vmid
subscriptionid
1pvP5oaK47WSSY0IZRNEQYdTLEx79rf7Gj1isBYW1jDOFGZ...   75007
IBRuELx83WZHD8ZBmRnQ7nN53DxcMPAO7szqGt218k7STW7...   61847
BShs5OvpbfrccmXj7X4MwSxkSFVNdSOzhYaDEKCijpvxWWk...   59707
+9OPyI+/Eeu5PSXVMdkPw3cB99+uk+YiAwMRGJU1cDm2ESA...   38140
OvdSquMKtCTJfHp792xq9WDE7nsLRulmPdbdqDAR/F/SaEU...   31512
```

```
In [21]:  vmtable_df[(vmtable_df['subscriptionid'] ==
'IBRuELx83WZHD8ZBmRnQ7nN53DxcMPAO7szqGt218k7STW7rx0pjgjj5eLJOFLbn') &
(vmtable_df['deploymentid'] == 'GVubwtq7m5auirutGFAvy2EKO61gAq7T9miuDadJS4tSiLPO
Srfk11eegzfpDDq/0KtD6kE3PMyxga34uuCpDA==')].count()
```

```
Out[21]: vmid              13256
subscriptionid    13256
deploymentid      13256
maxcpu            13256
avgcpu            13256
p95maxcpu         13256
vmcategory        13256
vmmemory          13256
corehour          13256
dtype: int64
```

given sub id = kaj8MGQgQTpbMpV6FhQnK... (because it has interactive VMs), find all VM data

```
In [22]: sub_sort_by_sub_id_desc_has_interactive_df =
vmtable_df.loc[vmtable_df['subscriptionid'] ==
'kaj8MGQgQTpbMpV6FhQnKYduXCc06zGHGxQGODJu9Q7VLZLkVX5RA7Rg84RQl8nT']
sub_sort_by_sub_id_desc_has_interactive_df.count()
```

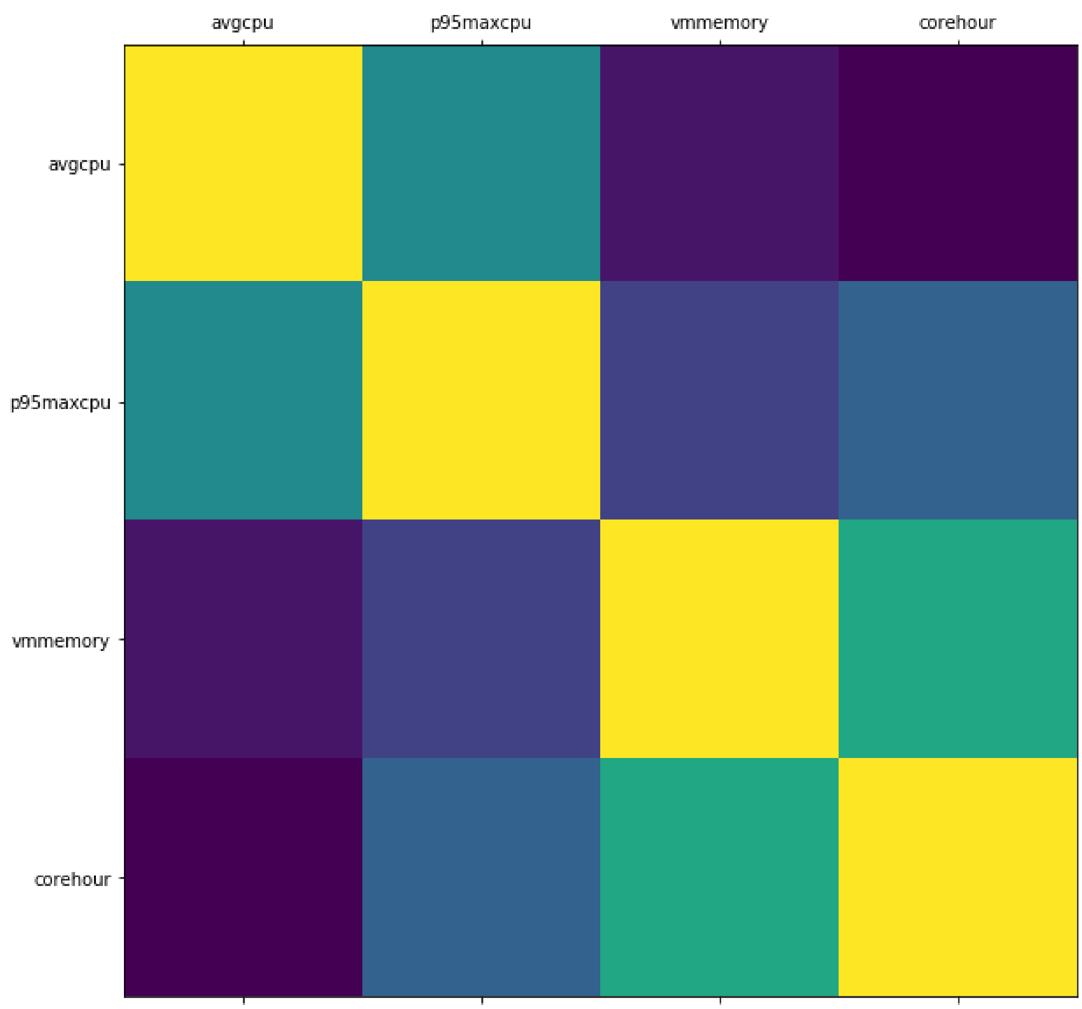
```
Out[22]: vmid              21637
subscriptionid    21637
deploymentid      21637
maxcpu            21637
avgcpu            21637
p95maxcpu         21637
vmcategory        21637
vmmemory          21637
corehour          21637
dtype: int64
```

for this group only interactive vmcategory

```
In [23]: sub_sort_by_sub_id_desc_only_interactive_df =
vmtable_df.loc[(vmtable_df['subscriptionid'] ==
'kaj8MGQgQTpbMpV6FhQnKYduXCc06zGHGxQGODJu9Q7VLZLkVX5RA7Rg84RQl8nT') &
(vmtable_df['vmcategory'] > 0) ]
sub_sort_by_sub_id_desc_only_interactive_df =
sub_sort_by_sub_id_desc_only_interactive_df.drop(['vmcategory','maxcpu'],
axis=1, inplace=False)
```

4.2.2 Correlation for interactive VMs for a chosen subscription id

```
In [24]: plot_corr(sub_sort_by_sub_id_desc_only_interactive_df)
```



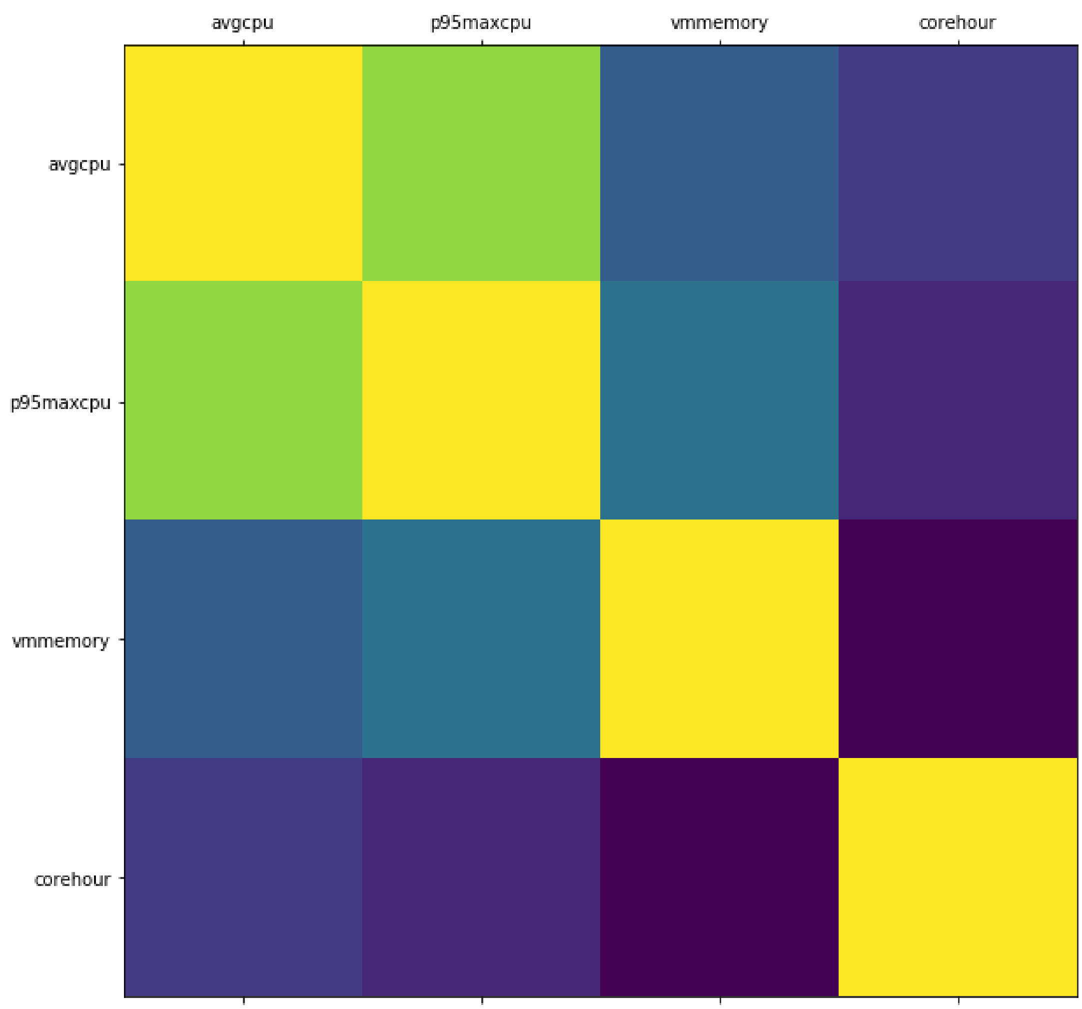
Correlation plot for Interactive VMs for a chosen subscription id

for this group only Delay-insensitive vmcategory

```
In [25]: sub_sort_by_sub_id_desc_only_batch_df =
vmtable_df.loc[(vmtable_df['subscriptionid'] ==
'kaj8MGQgQTpbMpV6FhQnKYduXCc06zGHGxQGODJu9Q7VLZLkVX5RA7Rg84RQl8nT') &
(vmtable_df['vmcategory'] < 1) ]
sub_sort_by_sub_id_desc_only_batch_df =
sub_sort_by_sub_id_desc_only_batch_df.drop(['vmcategory','maxcpu'], axis=1,
inplace=False)
```

4.2.3 Correlation for Delay-insensitive VMs for the chosen subscription id

```
In [26]: plot_corr(sub_sort_by_sub_id_desc_only_batch_df)
```



Correlation plot for Delay-insensitive VMs for the chosen subscription id

As we can see, for the above sub id bacth vms have greater correlation for corehour, lifetime, avgcpu, p95cpu features and compared to the interactive VMs. Here the interactive VMs were only (716/21637) i.e 3.31% of the total VMs provisioned for this subscription id. So, we have to keep this bias in mind.

4.2.4 Linear Regression

```
In [27]: sub_sort_by_sub_id_desc_only_interactive_df.describe()
```

Out[27]:

	lifetime	avgcpu	p95maxcpu	vmmemory	corehour
count	716.000000	716.000000	716.000000	716.000000	
mean	16.242731	63.001301	4.394553	997.655726	
std	13.912855	32.951065	1.833560	738.619372	
min	0.651969	4.759536	3.500000	142.500000	
25%	3.834555	26.366134	3.500000	509.791667	
50%	15.057395	79.985456	3.500000	808.166667	
75%	17.728612	90.368223	3.500000	1439.833333	
max	64.070261	99.171042	14.000000	5759.333333	

```
In [28]: sub_sort_by_sub_id_desc_only_batch_df.describe()
```

Out[28]:

	avgcpu	p95maxcpu	vmmemory	corehour
count	20921.000000	20921.000000	20921.000000	20921.000000
mean	7.840270	28.824360	4.331629	37.083760
std	9.435212	29.069097	2.215359	211.001410
min	0.020781	0.049428	1.750000	0.083333
25%	0.580033	2.161299	3.500000	0.166667
50%	4.843454	21.907838	3.500000	0.333333
75%	12.095304	44.764788	3.500000	0.666667
max	83.171846	100.000000	14.000000	5759.333333

Create training and test data using train_test_split

interactive

```
In [29]: from sklearn.model_selection import train_test_split
```

```
X = sub_sort_by_sub_id_desc_only_interactive_df.drop(['p95maxcpu','vmid','subscr
iptionid','deploymentid'], axis=1)

# Taking the labels (avg_cpu)
Y = sub_sort_by_sub_id_desc_only_interactive_df['p95maxcpu']

# Spliting into 80% for training set and 20% for testing set so we can see our
accuracy
X_train, x_test, Y_train, y_test = train_test_split(X, Y, test_size=0.2,
random_state=0)
```

Delay-insensitive

```
In [30]: X_subid_batch = sub_sort_by_sub_id_desc_only_batch_df.drop(['p95maxcpu','vmid','
subscriptionid','deploymentid'], axis=1)

# Taking the labels (avg_cpu)
Y_subid_batch = sub_sort_by_sub_id_desc_only_batch_df['p95maxcpu']

# Spliting into 80% for training set and 20% for testing set so we can see our
accuracy
X_train_subid_batch, x_test_subid_batch, Y_train_subid_batch, y_test_subid_batch
= train_test_split(X_subid_batch, Y_subid_batch, test_size=0.2, random_state=0)
```

Create a LinearRegression model with our training data

Interactive

```
In [31]: from sklearn.linear_model import LinearRegression

linear_model = LinearRegression()
linear_model.fit(X_train, Y_train)
```

```
Out[31]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

Delay-insensitive

```
In [32]: linear_model_subid_batch = LinearRegression()
linear_model_subid_batch.fit(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[32]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

Check R-square on training data

Interactive

```
In [33]: linear_model.score(X_train, Y_train)
```

```
Out[33]: 0.33079005962722097
```

Delay-insensitive

```
In [34]: linear_model_subid_batch.score(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[34]: 0.7302633566441574
```

View coefficients for each feature

Interactive

```
In [35]: linear_model.coef_
```

```
Out[35]: array([ 1.15866841, -0.54319514,  0.01428175])
```

Delay-insensitive

```
In [36]: linear_model_subid_batch.coef_
```

```
Out[36]: array([ 2.45957045,  1.8644932 , -0.00385822])
```

A better view of the coefficients List of features and their coefficients, ordered by coefficient value

Interactive

```
In [37]: predictors = X_train.columns
coef = pd.Series(linear_model.coef_,predictors).sort_values()

print(coef)
```

```
vmmemory   -0.543195
corehour    0.014282
avgcpu      1.158668
dtype: float64
```

Delay-insensitive

```
In [38]: predictors_subid_batch = X_train_subid_batch.columns
coef_subid_batch =
pd.Series(linear_model_subid_batch.coef_,predictors_subid_batch).sort_values()

print(coef_subid_batch)
```

```
corehour   -0.003858
vmmemory    1.864493
avgcpu      2.459570
dtype: float64
```

Make predictions on test data

Interactive

```
In [39]: y_predict = linear_model.predict(x_test)
```

Delay-insensitive

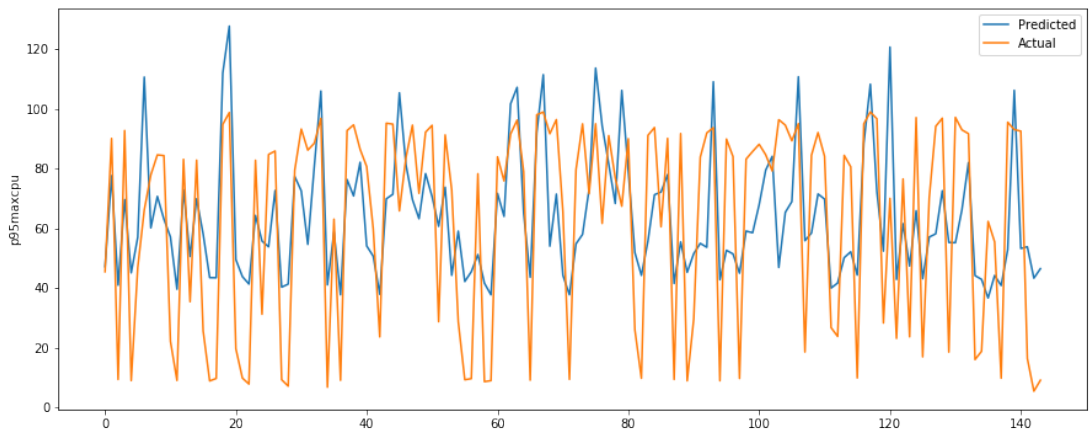
```
In [40]: y_predict_subid_batch = linear_model_subid_batch.predict(x_test_subid_batch)
```

Compare predicted and actual values of p95maxcpu

Interactive

```
In [41]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict, label='Predicted')
plt.plot(y_test.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()
```

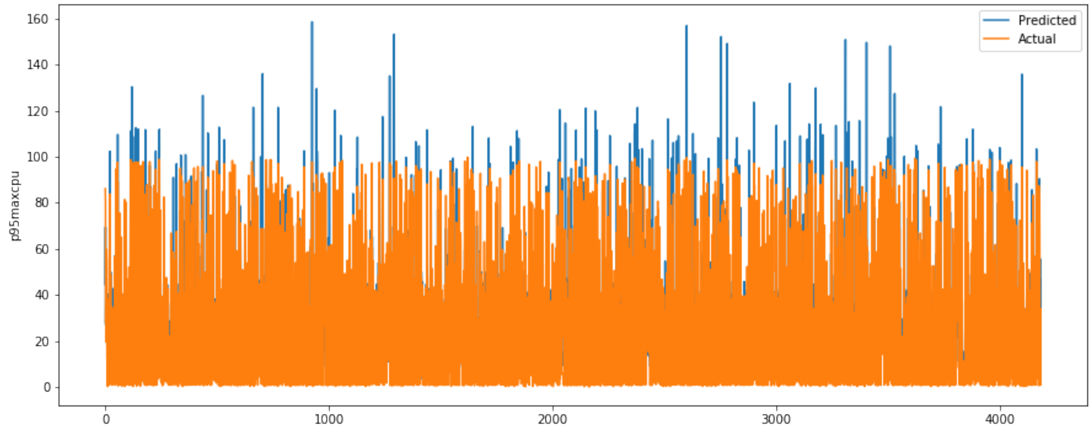
```
Populating the interactive namespace from numpy and matplotlib
```



p95maxcpu prediction for Interactive VMs using Linear Regression

Delay-insensitive

```
In [42]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict_subid_batch, label='Predicted')
plt.plot(y_test_subid_batch.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()

Populating the Delay-insensitive namespace from numpy and matplotlib
```



p95maxcpu prediction for Delay-insensitive VMs using Linear Regression

R-square score For our model, how well do the features describe the p95maxcpu?

Interactive

```
In [43]: r_square = linear_model.score(x_test, y_test)
r_square
```

```
Out[43]: 0.40748150843068387
```

Delay-insensitive

```
In [44]: r_square_subid_batch = linear_model_subid_batch.score(x_test_subid_batch,
y_test_subid_batch)
r_square_subid_batch
```

```
Out[44]: 0.7500118594887903
```

Calculate Mean Square Error

Interactive

```
In [45]: from sklearn.metrics import mean_squared_error

linear_model_mse = mean_squared_error(y_predict, y_test)
linear_model_mse
```

Out[45]: 695.2385193698618

Delay-insensitive

```
In [46]: from sklearn.metrics import mean_squared_error

linear_model_mse_subid_batch = mean_squared_error(y_predict_subid_batch,
y_test_subid_batch)
linear_model_mse_subid_batch
```

Out[46]: 206.6706436996513

Root of Mean Square Error

Interactive

```
In [47]: import math

math.sqrt(linear_model_mse)
```

Out[47]: 26.367376042561798

Delay-insensitive

```
In [48]: import math

math.sqrt(linear_model_mse_subid_batch)
```

Out[48]: 14.376044090766113

4.2.5 Lasso Regression

Cost Function: RSS + α*(sum of absolute values of coefficients)

RSS = Residual Sum of Squares

Larger values of α should result in smaller coefficients as the cost function needs to be minimized

Interactive

```
In [49]: from sklearn.linear_model import Lasso

lasso_model = Lasso(alpha=0.7, normalize=False)
lasso_model.fit(X_train, Y_train)
```

```
Out[49]: Lasso(alpha=0.7, copy_X=True, fit_intercept=True, max_iter=1000,
normalize=False, positive=False, precompute=False, random_state=None,
selection='cyclic', tol=0.0001, warm_start=False)
```

Delay-insensitive

```
In [50]: lasso_model_subid_batch = Lasso(alpha=0.5, normalize=False)
lasso_model_subid_batch.fit(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[50]: Lasso(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=1000,
normalize=False, positive=False, precompute=False, random_state=None,
selection='cyclic', tol=0.0001, warm_start=False)
```

Check R-square on training data

Interactive

```
In [51]: lasso_model.score(X_train, Y_train)
```

```
Out[51]: 0.3305669233028511
```

Delay-insensitive

```
In [52]: lasso_model_subid_batch.score(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[52]: 0.7302021565572885
```

Coefficients when using Lasso

Interactive

```
In [53]: coef = pd.Series(lasso_model.coef_,predictors).sort_values()
print(coef)
```

```
vmmemory   -0.212193
corehour    0.013789
avgcpu      1.151806
dtype: float64
```

Delay-insensitive

```
In [54]: coef_subid_batch =
pd.Series(lasso_model_subid_batch.coef_,predictors_subid_batch).sort_values()
print(coef_subid_batch)
```

```
corehour   -0.003814
vmmemory    1.758096
avgcpu      2.462035
dtype: float64
```

Make predictions on test data

Interactive

```
In [55]: y_predict = lasso_model.predict(x_test)
```

Delay-insensitive

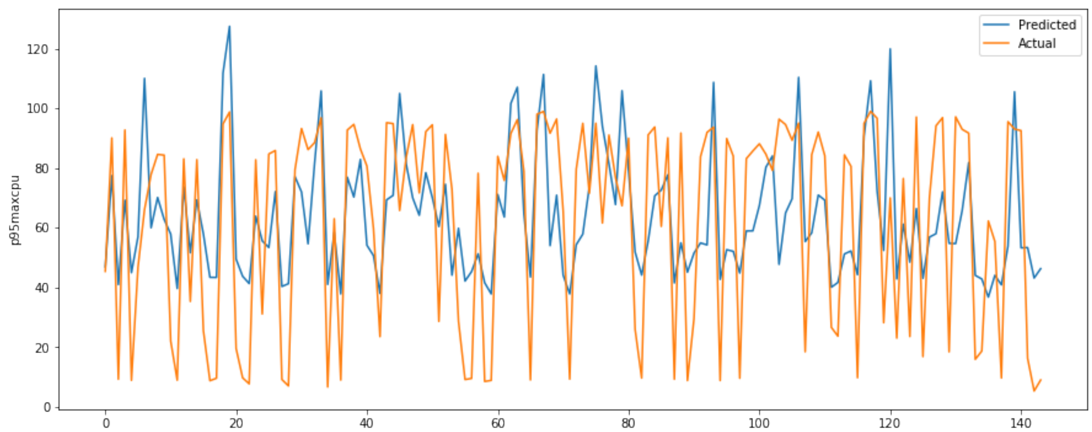
```
In [56]: y_predict_subid_batch = lasso_model_subid_batch.predict(x_test_subid_batch)
```

Compare predicted and actual values of p95maxcpu

Interactive

```
In [57]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict, label='Predicted')
plt.plot(y_test.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()
```

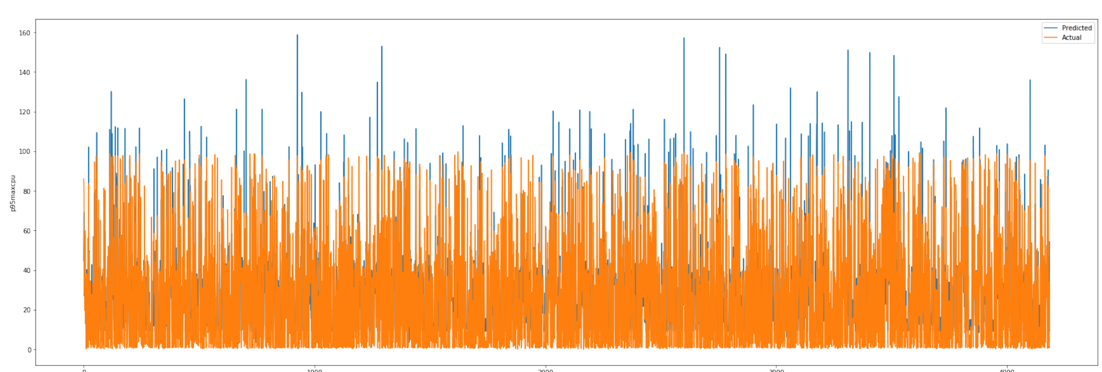
```
Populating the interactive namespace from numpy and matplotlib
```



p95maxcpu prediction for Interactive VMs using Lasso Regression

Delay-insensitive

```
In [58]: %pylab inline
pylab.rcParams['figure.figsize'] = (30, 10)

plt.plot(y_predict_subid_batch, label='Predicted')
plt.plot(y_test_subid_batch.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()

Populating the Delay-insensitive namespace from numpy and matplotlib
```



p95maxcpu prediction for Delay-insensitive VMs using Lasso Regression

Check R-square value on test data

Interactive

```
In [59]: r_square = lasso_model.score(x_test, y_test)
r_square
```

```
Out[59]: 0.40841470247604866
```

Delay-insensitive

```
In [60]: r_square_subid_batch = lasso_model_subid_batch.score(x_test_subid_batch,
y_test_subid_batch)
r_square_subid_batch
```

```
Out[60]: 0.7496365151711825
```

Is the root mean square error any better?

Interactive

```
In [61]: lasso_model_mse = mean_squared_error(y_predict, y_test)
math.sqrt(lasso_model_mse)
```

```
Out[61]: 26.346604054318284
```

Delay-insensitive

```
In [62]: lasso_model_mse_subid_batch = mean_squared_error(y_predict_subid_batch,
y_test_subid_batch)
math.sqrt(lasso_model_mse_subid_batch)
```

```
Out[62]: 14.38683248761928
```

4.2.6 Ridge Regression

Cost Function: RSS + α*(sum of squares of coefficients)

RSS = Residual Sum of Squares

Larger values of α should result in smaller coefficients as the cost function needs to be minimized

Ridge Regression penalizes large coefficients even more than Lasso as coefficients are squared in cost function

Interactive

```
In [63]: from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=0.1, normalize=True)
ridge_model.fit(X_train, Y_train)
```

```
Out[63]: Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
normalize=True, random_state=None, solver='auto', tol=0.001)
```

Delay-insensitive

```
In [64]: #from sklearn.linear_model import Ridge

ridge_model_subid_batch = Ridge(alpha=0.1, normalize=True)
ridge_model_subid_batch.fit(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[64]: Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
normalize=True, random_state=None, solver='auto', tol=0.001)
```

Check R-square on training data

Interactive

```
In [65]: ridge_model.score(X_train, Y_train)
```

```
Out[65]: 0.32753252981125314
```

Delay-insensitive

```
In [66]: ridge_model_subid_batch.score(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[66]: 0.7246368614895887
```

Coefficients when using Ridge

Interactive

```
In [67]: coef = pd.Series(ridge_model.coef_,predictors).sort_values()
print(coef)
```

```
vmmemory   -0.004598
corehour    0.012278
avgcpu      1.049424
dtype: float64
```

Delay-insensitive

```
In [68]: coef_subid_batch =
pd.Series(ridge_model_subid_batch.coef_,predictors_subid_batch).sort_values()
print(coef_subid_batch)
```

```
corehour   -0.001351
vmmemory    2.001922
avgcpu      2.215234
dtype: float64
```

Make predictions on test data

Interactive

```
In [69]: y_predict = ridge_model.predict(x_test)
```

Delay-insensitive

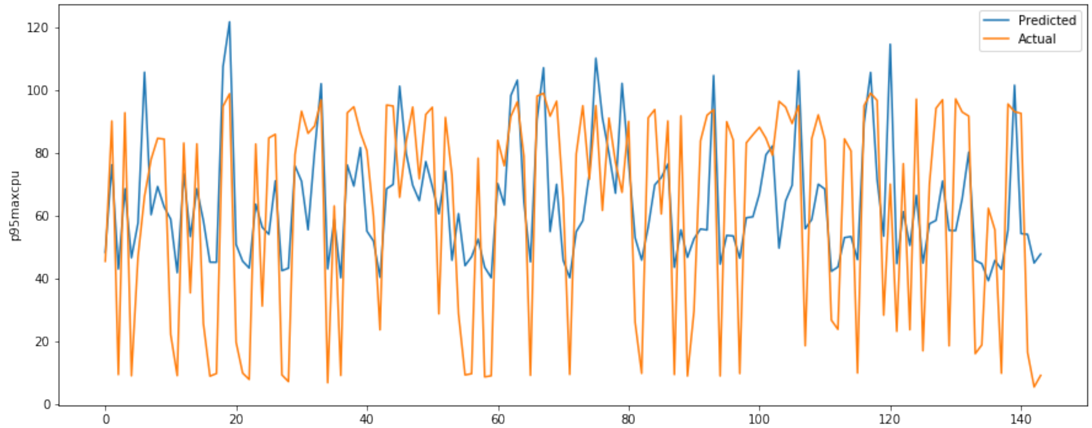
```
In [70]: y_predict_subid_batch = ridge_model_subid_batch.predict(x_test_subid_batch)
```

Compare predicted and actual values of p95maxcpu

Interactive

```
In [71]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict, label='Predicted')
plt.plot(y_test.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```

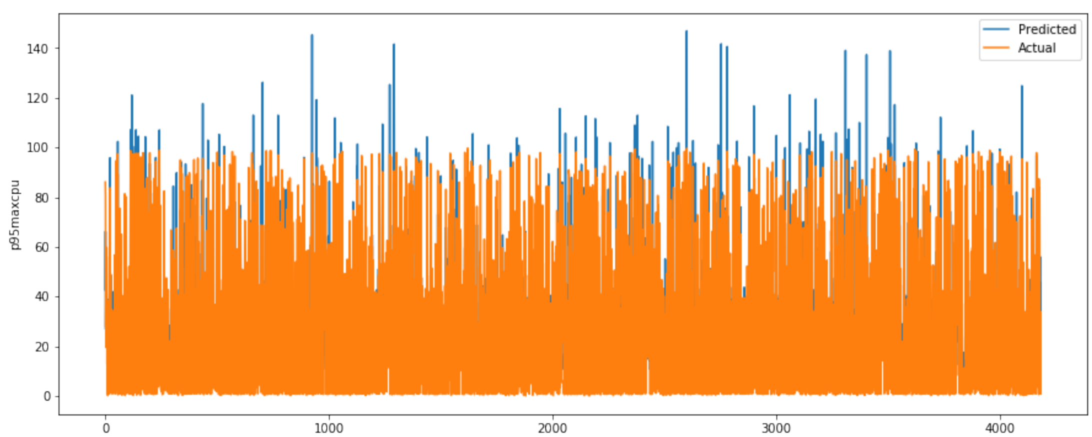


p95maxcpu prediction for Interactive VMs using Ridge Regression

Delay-insensitive

```
In [72]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)


plt.plot(y_predict_subid_batch, label='Predicted')
plt.plot(y_test_subid_batch.values, label='Actual')
plt.ylabel('p95maxcpu')


plt.legend()
plt.show()
```

```
Populating the Delay-insensitive namespace from numpy and matplotlib
```



p95maxcpu prediction for Delay-insensitive VMs using Ridge Regression

Get R-square value for test data

Interactive

```
In [73]: r_square = ridge_model.score(x_test, y_test)
r_square
```

```
Out[73]: 0.4020324715291811
```

Delay-insensitive

```
In [74]: r_square_subid_batch = ridge_model_subid_batch.score(x_test_subid_batch,
y_test_subid_batch)
r_square_subid_batch
```

```
Out[74]: 0.7408046772480763
```

Interactive

```
In [75]: ridge_model_mse = mean_squared_error(y_predict, y_test)
math.sqrt(ridge_model_mse)
```

```
Out[75]: 26.488341033523948
```

Delay-insensitive

```
In [76]: ridge_model_mse_subid_batch = mean_squared_error(y_predict_subid_batch,
y_test_subid_batch)
math.sqrt(ridge_model_mse_subid_batch)
```

```
Out[76]: 14.638388637696366
```

4.2.7 Support Vector Regression (SVR)

Interactive

```
In [77]: from sklearn.svm import SVR

regression_model = SVR(kernel='linear', C=2)
regression_model.fit(X_train, Y_train)
```

```
Out[77]: SVR(C=2, cache_size=200, coef0=0.0, degree=3, epsilon=0.1,
gamma='auto', kernel='linear', max_iter=-1, shrinking=True,
tol=0.001, verbose=False)
```

R-square on training data

```
In [78]: regression_model.score(X_train, Y_train)
```

```
Out[78]: 0.2732295573490885
```

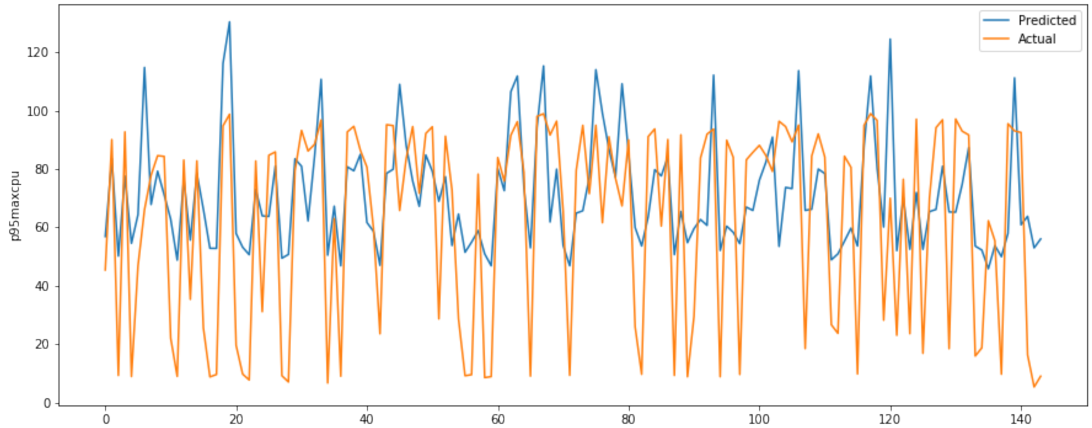
```
In [79]: coef = pd.Series(regression_model.coef_[0], predictors).sort_values()
print(coef)
```

```
vmmemory   -1.491601
corehour    0.015023
avgcpu      1.047717
dtype: float64
```

```
In [80]: y_predict = regression_model.predict(x_test)
```

```
In [81]: %pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict, label='Predicted')
plt.plot(y_test.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```



p95maxcpu prediction for Interactive VMs using SVR

R-square on test data

```
In [82]: r_square = regression_model.score(x_test, y_test)
r_square
```

```
Out[82]: 0.33512860915120257
```

```
In [83]: regression_model_mse = mean_squared_error(y_predict, y_test)
math.sqrt(regression_model_mse)
```

```
Out[83]: 27.93089046339419
```

Delay-insensitive

```
In [84]: #from sklearn.svm import SVR

regression_model_subid_batch = SVR(kernel='linear', C=2)
regression_model_subid_batch.fit(X_train_subid_batch, Y_train_subid_batch)
```

```
Out[84]: SVR(C=2, cache_size=200, coef0=0.0, degree=3, epsilon=0.1,
gamma='auto', kernel='linear', max_iter=-1, shrinking=True,
tol=0.001, verbose=False)
```

R-square on training data

```
In [85]: regression_model_subid_batch.score(X_train_subid_batch, Y_train_subid_batch)
```

Out[85]: 0.67538479854174

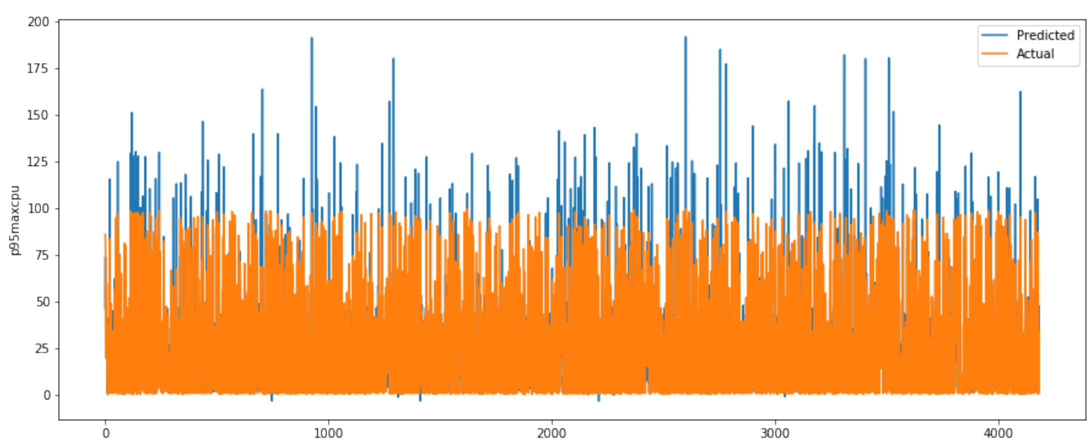
In [86]:
```
coef_subid_batch = pd.Series(regression_model_subid_batch.coef_[0],
predictors_subid_batch).sort_values()
print(coef_subid_batch)
```

```
corehour   -0.003186
vmmemory    1.141142
avgcpu      3.128491
dtype: float64
```

In [87]:
```
y_predict_subid_batch = regression_model_subid_batch.predict(x_test_subid_batch)
```

In [88]:
```
%pylab inline
pylab.rcParams['figure.figsize'] = (15, 6)

plt.plot(y_predict_subid_batch, label='Predicted')
plt.plot(y_test_subid_batch.values, label='Actual')
plt.ylabel('p95maxcpu')

plt.legend()
plt.show()
```

```
Populating the Delay-insensitive namespace from numpy and matplotlib
```



p95maxcpu prediction for Delay-insensitive VMs using SVR

In [89]:
```
r_square_subid_batch = regression_model_subid_batch.score(x_test_subid_batch,
y_test_subid_batch)
r_square_subid_batch
```

```
Out[89]: 0.7079164741245016
```

```
In [90]: regression_model_mse_subid_batch = mean_squared_error(y_predict_subid_batch,
y_test_subid_batch)
math.sqrt(regression_model_mse_subid_batch)
```

```
Out[90]: 15.53936344115162
```

4.2.8 Interactive VMs v/s Delay Insensitive VMs

df with only interactive VMs not considering the subscription id

```
In [91]: only_interactive_df = vmtable_df.loc[vmtable_df['vmcategory'] > 0]
```

```
In [92]: only_interactive_df.describe()
```

```
Out[92]:     maxcpu        avgcpu      p95maxcpu  vmcategory      vmmemory  \
count  60682.000000  60682.000000  60682.000000     60682.0  60682.000000
mean      97.420276     10.248779     38.870767         1.0      4.303088
std        5.824761     11.410434     28.542252         0.0      6.448530
min       12.511224      0.192261      0.312124         1.0      0.750000
25%       98.095370      4.175261     18.214594         1.0      1.750000
50%       99.114400      6.314372     26.766012         1.0      1.750000
75%       99.722199     11.324602     56.290819         1.0      3.500000
max      100.000000     97.800798    100.000000         1.0    112.000000


           corehour
count  60682.000000
mean    1183.554909
std     1139.674975
min       70.083333
25%      719.916667
50%      719.916667
75%     1439.833333
max    11518.666667
```

df with only delay-insensitive VMs not considering the subscription id

```
In [93]: only_batch_df = vmtable_df.loc[vmtable_df['vmcategory'] < 1]
```

```
In [94]: only_batch_df.describe()
```

```
Out[94]:    maxcpu         avgcpu      p95maxcpu  vmcategory      vmmemory  \
count  780488.000000  780488.000000  780488.000000    780488.0  780488.000000
mean       59.170758      13.003309      45.626637         0.0       5.157735
std        40.886243      17.298005      39.129215         0.0       9.324064
min         0.000000       0.000000       0.000000         0.0       0.750000
25%         8.768236       0.883998       5.099918         0.0       1.750000
50%        78.915568       5.898452      35.982504         0.0       1.750000
75%        98.182540      18.216650      89.361248         0.0       3.500000
max       100.000000     100.000000     100.000000         0.0     112.000000

           corehour
count  780488.000000
mean      188.298322
std       657.856882
min         0.083333
25%         0.250000
50%         0.750000
75%        10.833333
max     11518.666667
```

We can see that there are 780488 delay-insensitive VMs 60682 interactive VMs

Delay-insensitive

```
In [95]: only_batch_df.groupby(['vmid']).agg({'vmid':'count'}).sort_values('vmid',ascendi
ng=False).head()
```

```
Out[95]:                                                              vmid
vmid
```

```
+++2nVlUB8GxQa5mPbeaQx7T+O1PwQiIl4Q/9OI6UHYd9kS...      1
ejJYoOF28Gp3NfoBktj1qM5daWxt6/M+yUaEQR84Mi4JKUO...      1
ejGLzuHGCmjEiRP7G3eVqomZJtjXqsk414oTix83h0gAEa6...      1
ejHnNOLgYaHeZRzTS9+AGoHOLot7ZDC5Dm8MWkfNf12bwxb...      1
ejI64Y4hPy3qxUz4UHn9gJpCspiyJP2FHYT8pN4xK7ZjbVT...      1
```

```
In [96]: only_interactive_df.groupby(['vmid']).agg({'vmid':'count'}).sort_values('vmid',a
scending=False).head()
```

```
Out[96]:                                                    vmid
vmid
++6pzlrow/E95W9M9nzu+HfTft8NNdKv5pcM78HhFqsRiPX...      1
eZZuDOmrNh2gfJhN6cayKLrFLqHWtIEmixeM4i5eBO9ot9b...      1
eYJvCglokI+yCUYa2oBawQ1NAjt32E49HzpkIxyl47hpe9v...      1
eYKyxtd5oFpezs+FYC959DNwdFM1EJKN0gYmxu9fZQhsGRS...      1
eYUYEJyuEEWyPzfJgu9Q1jnZHqrMQ/JpcRye8eI6svndqH3...      1
```

4.2.8.1 Correlation for the Delay Insensitive and Interactive VMs

Drop vmcreated, vmdeleted, vmcorecount columns to have only the features

```
In [97]: only_batch_features_df = only_batch_df.drop(['vmid','subscriptionid','deployment
id','maxcpu','vmcategory'], axis=1, inplace=False)
```

```
In [98]: only_interactive_features_df = only_interactive_df.drop(['vmid','subscriptionid'
,'deploymentid','maxcpu','vmcategory'], axis=1, inplace=False)
```

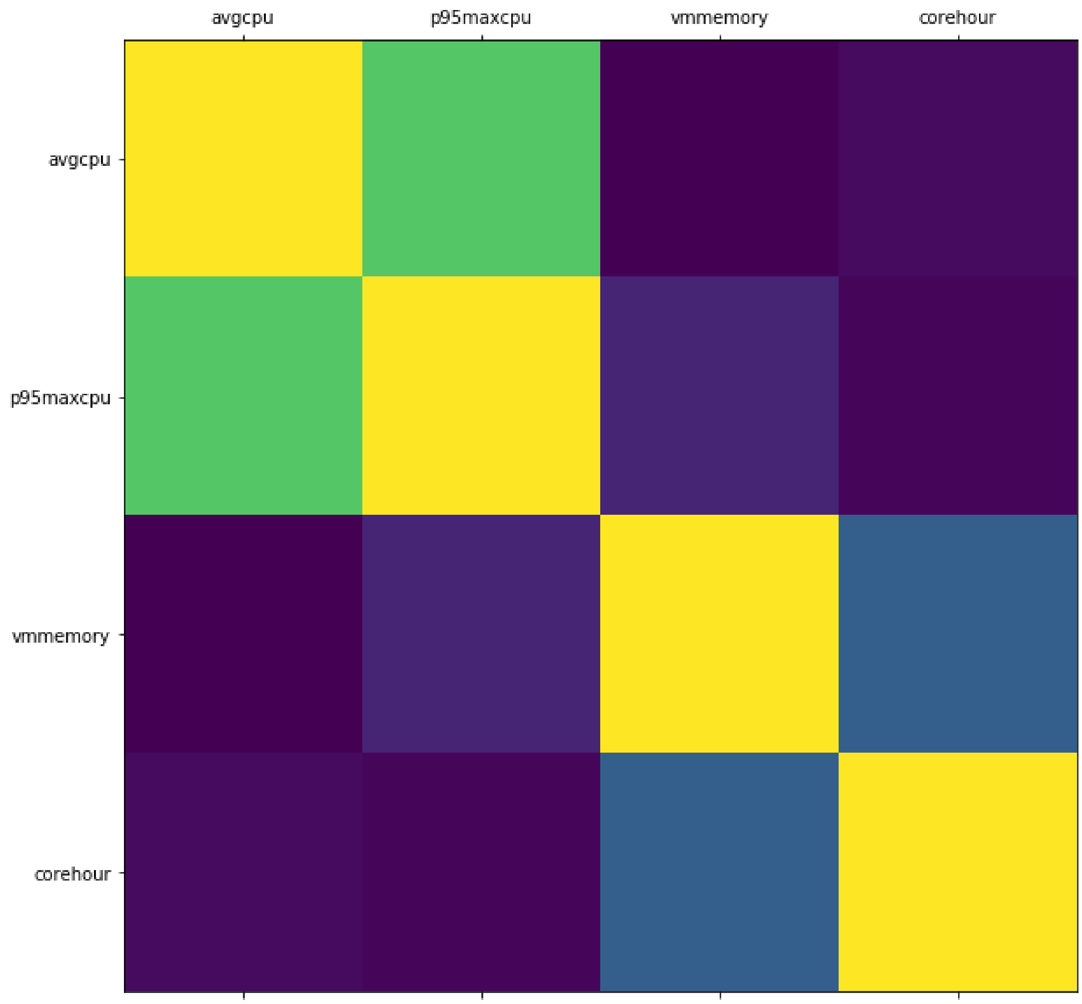
4.2.8.2 Plot Correlation for Delay Insensitive VMs

```
In [99]: only_batch_features_df.corr()
```

Out[99]:

	avgcpu	p95maxcpu	vmmemory	corehour
avgcpu	1.000000	0.701381	-0.136359	-0.104452
p95maxcpu	0.701381	1.000000	-0.017144	-0.114765
vmmemory	-0.136359	-0.017144	1.000000	0.206682
corehour	-0.104452	-0.114765	0.206682	1.000000

```
In [100]: plot_corr(only_batch_features_df)
```



Correlation plot for Delay-insensitive VMs

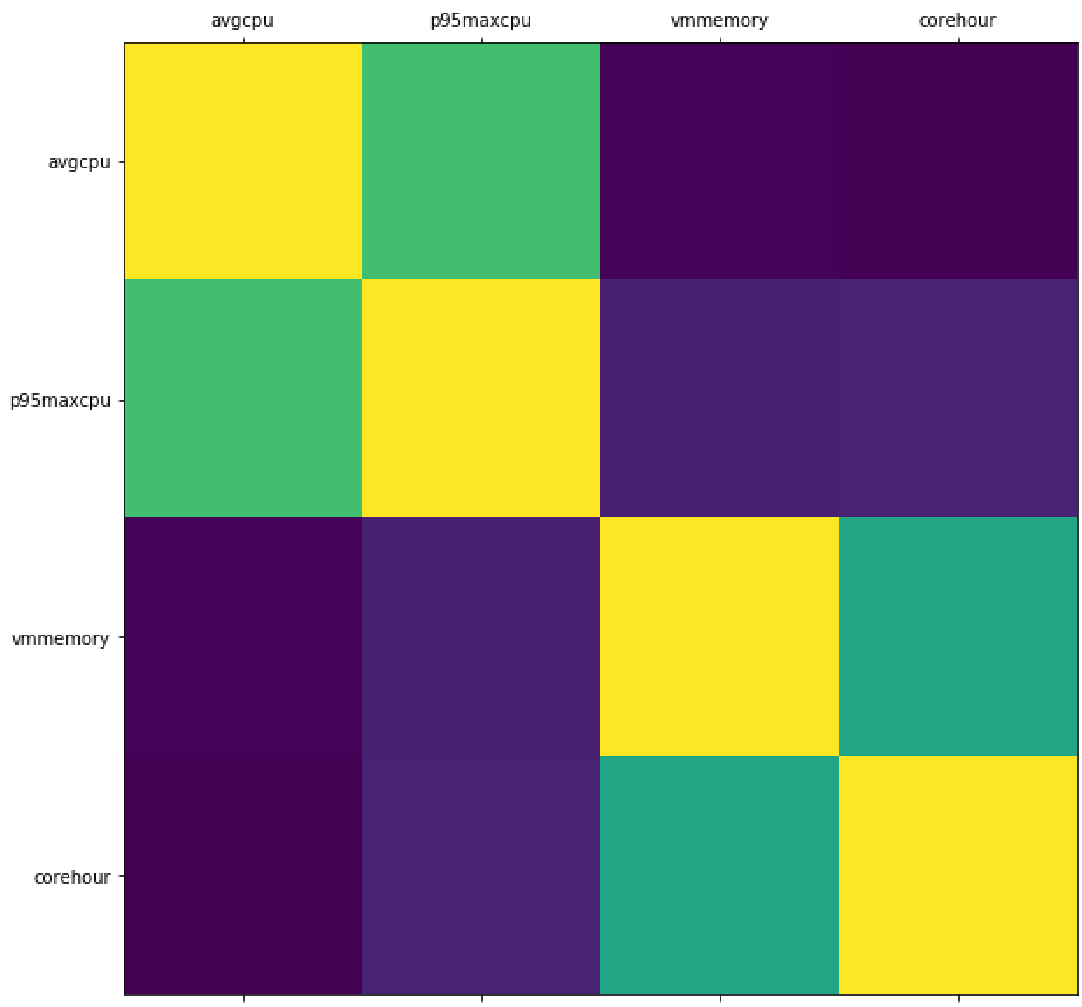
4.2.8.3 Plot Correlation for Interactive VMs

```
In [101]: only_interactive_features_df.corr()
```

Out[101]:

	avgcpu	p95maxcpu	vmmemory	corehour
avgcpu	1.000000	0.709775	0.044812	0.036921
p95maxcpu	0.709775	1.000000	0.122231	0.130363
vmmemory	0.044812	0.122231	1.000000	0.608515
corehour	0.036921	0.130363	0.608515	1.000000

```
In [102]: plot_corr(only_interactive_features_df)
```



Correlation plot for Interactive VMs

4.3 RNN with LSTMs

- Generating input file

1. Select a VM
2. from vmtable.csv, get the vmid, vmcreated, vmdeleted time.
3. Go through the vm_cpu_readings files and get the row record corresponding to this VM ID and write to a file which will be input file for this VM

Repeat the above three steps for generating input files for other VMs.

4.3.1 Import the Keras, scikit-learn and python libraries

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
from keras import metrics
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.callbacks import Callback
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras import optimizers
```

```
/root/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:34: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floating` is
deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
In [2]: headers=['timestamp','vm id','min cpu','max cpu', 'avg cpu']
```

4.3.2 Load the input dataset generated for the VM

```
In [3]: df = pd.read_csv("input/input_vm0_cpu_readings-file-1-to-125.csv", header=None,
index_col=False,names=headers,delimiter=',')
```

Since we require only the 'min cpu','max cpu','avg cpu' values, we takes these values and convert as numpy array

```
In [4]: df = df[['min cpu','max cpu','avg cpu']]
```

```
In [5]: dataset = df.values
dataset = dataset.astype('float32')
```

Neural networks are sensitive to input data, especiallly when we are using activation functions like sigmoid or tanh activation functions are used. So we rescale our data to the range of 0-to-1, using MinMaxScaler.

```
In [6]: scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

```
In [7]: train_size = int(len(dataset) * 0.8)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size, :], dataset[train_size:len(dataset), :]
print(len(train), len(test))
```

```
6905 1727
```

The create_training_dataset function below is used to convert an array of values into a dataset matrix. It takes two inputs: 1. dataset - numpy array to be converted into a dataset 2. look_back - number of previous time steps to use as input variables to predict the next time period

```
In [8]: def create_training_dataset(dataset, look_back=1):
dataX, dataY = [], []
for i in range(len(dataset)-look_back-1):
a = dataset[i:(i+look_back), :3]
dataX.append(a)
dataY.append(dataset[i + look_back, :])
return np.array(dataX), np.array(dataY)
```

```
In [9]: look_back = 40
trainX, trainY = create_training_dataset(train, look_back=look_back)
testX, testY = create_training_dataset(test, look_back=look_back)
```

4.3.3 Build our Model

```
In [10]: model = Sequential()
model.add(LSTM(4, input_shape=(trainX.shape[1], trainX.shape[2])))
model.add(Dense(3))
adamOpt = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
model.compile(loss='mean_squared_error', optimizer=adamOpt,
metrics=[metrics.mae])
history = model.fit(trainX, trainY, validation_split=0.25,epochs=40,
batch_size=64, verbose=2)
```

```
Train on 5148 samples, validate on 1716 samples
Epoch 1/40
```

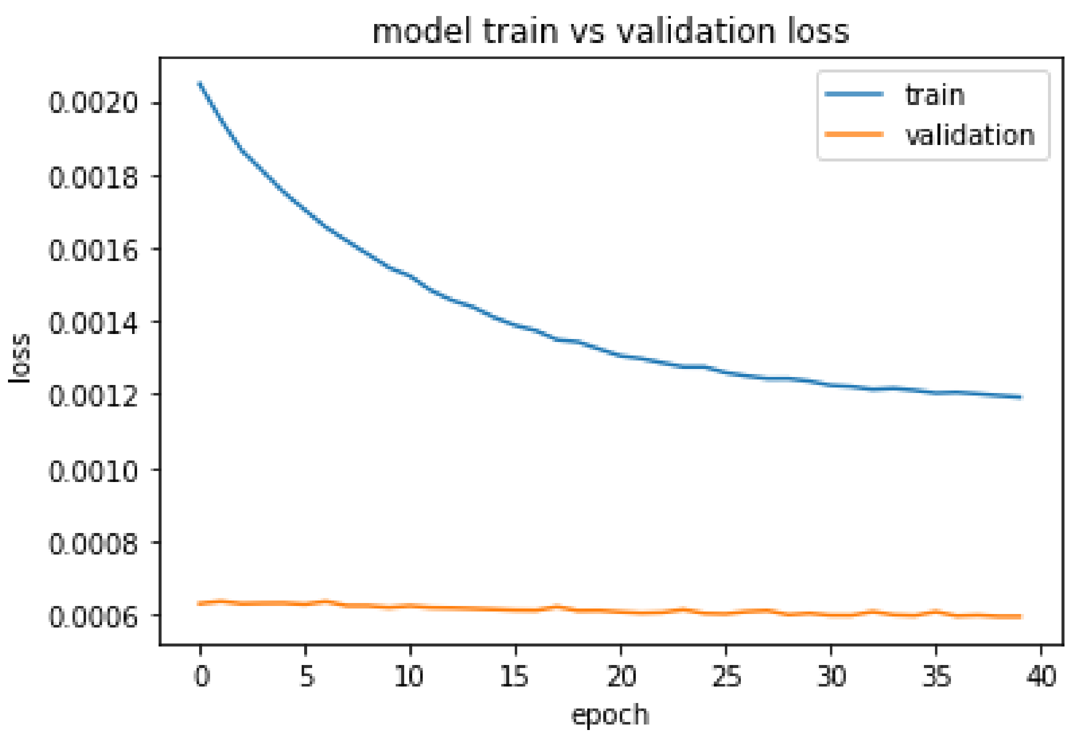
```
- 4s - loss: 0.0020 - mean_absolute_error: 0.0110 - val_loss: 6.2943e-04 -
val_mean_absolute_error: 0.0084
Epoch 2/40
- 3s - loss: 0.0019 - mean_absolute_error: 0.0109 - val_loss: 6.3563e-04 -
val_mean_absolute_error: 0.0093
Epoch 3/40
- 3s - loss: 0.0019 - mean_absolute_error: 0.0106 - val_loss: 6.2918e-04 -
val_mean_absolute_error: 0.0088
Epoch 4/40
- 3s - loss: 0.0018 - mean_absolute_error: 0.0108 - val_loss: 6.3036e-04 -
val_mean_absolute_error: 0.0092
Epoch 5/40
- 3s - loss: 0.0018 - mean_absolute_error: 0.0105 - val_loss: 6.3076e-04 -
val_mean_absolute_error: 0.0094
Epoch 6/40
- 3s - loss: 0.0017 - mean_absolute_error: 0.0102 - val_loss: 6.2740e-04 -
val_mean_absolute_error: 0.0087
Epoch 7/40
- 3s - loss: 0.0017 - mean_absolute_error: 0.0103 - val_loss: 6.3526e-04 -
val_mean_absolute_error: 0.0097
Epoch 8/40
- 3s - loss: 0.0016 - mean_absolute_error: 0.0102 - val_loss: 6.2349e-04 -
val_mean_absolute_error: 0.0088
Epoch 9/40
- 3s - loss: 0.0016 - mean_absolute_error: 0.0102 - val_loss: 6.2334e-04 -
val_mean_absolute_error: 0.0089
Epoch 10/40
- 3s - loss: 0.0015 - mean_absolute_error: 0.0100 - val_loss: 6.1886e-04 -
val_mean_absolute_error: 0.0085
.
.
.
.
Epoch 32/40
- 2s - loss: 0.0012 - mean_absolute_error: 0.0092 - val_loss: 5.9752e-04 -
val_mean_absolute_error: 0.0077
Epoch 33/40
```

```
- 3s - loss: 0.0012 - mean_absolute_error: 0.0089 - val_loss: 6.0768e-04 -
val_mean_absolute_error: 0.0095
Epoch 34/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0092 - val_loss: 5.9884e-04 -
val_mean_absolute_error: 0.0073
Epoch 35/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0091 - val_loss: 5.9684e-04 -
val_mean_absolute_error: 0.0075
Epoch 36/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0090 - val_loss: 6.0756e-04 -
val_mean_absolute_error: 0.0098
Epoch 37/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0094 - val_loss: 5.9584e-04 -
val_mean_absolute_error: 0.0076
Epoch 38/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0088 - val_loss: 5.9787e-04 -
val_mean_absolute_error: 0.0088
Epoch 39/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0090 - val_loss: 5.9451e-04 -
val_mean_absolute_error: 0.0081
Epoch 40/40
- 3s - loss: 0.0012 - mean_absolute_error: 0.0091 - val_loss: 5.9448e-04 -
val_mean_absolute_error: 0.0084
```

```
In [11]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 4)                 128
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 15
=================================================================
Total params: 143
Trainable params: 143
Non-trainable params: 0
_________________________________________________________________
```

```
In [12]: # plot train and validation loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model train vs validation loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```



Plot of LSTM model training v/s validation loss

```
In [13]: trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
```

We have to invert the predictions before calculating error to so that reports will be in same units as our original data

```
In [14]: trainY = scaler.inverse_transform(trainY)
trainPredict = scaler.inverse_transform(trainPredict)
testY = scaler.inverse_transform(testY)
testPredict = scaler.inverse_transform(testPredict)
```

```
In [15]: trainScore = math.sqrt(mean_squared_error(trainY[:], trainPredict[:]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[:], testPredict[:]))
print('Test Score: %.2f RMSE' % (testScore))
```

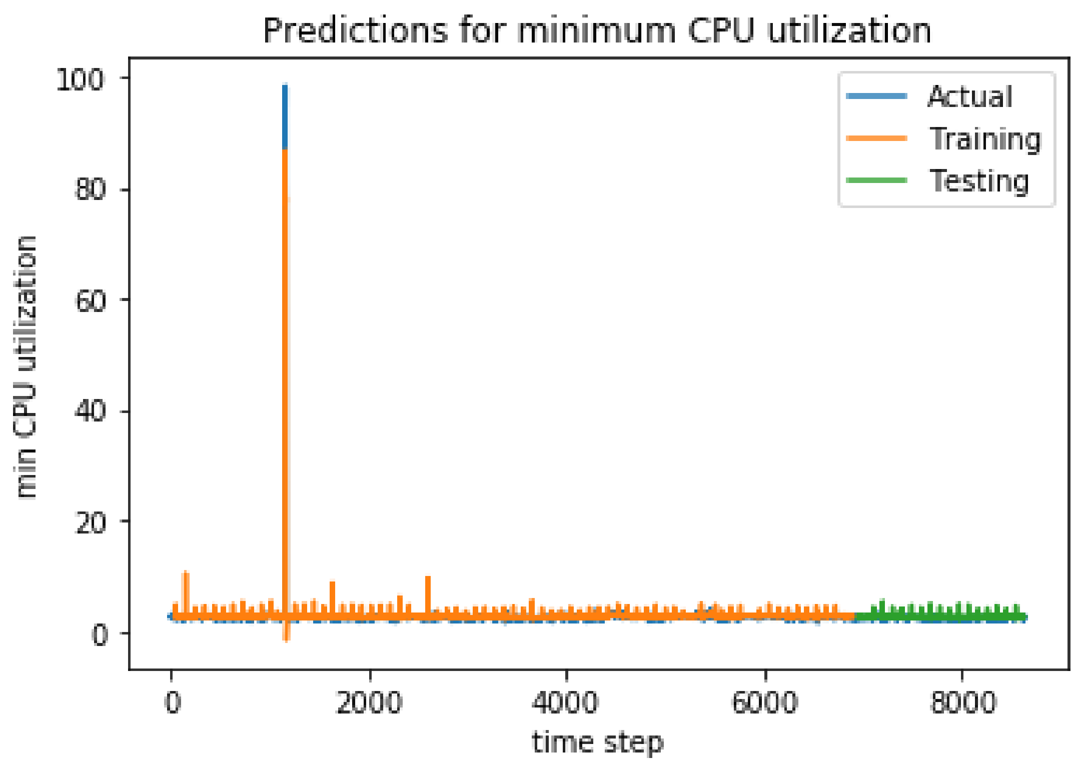
```
Train Score: 3.11 RMSE
Test Score: 2.73 RMSE
```

4.3.4 Plot for minimum CPU utilization

```
In [16]: # shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset[:,:1])
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict[:,:1]) + look_back, :1] =
trainPredict[:,:1]
```

```
In [17]:  # shift test predictions for plotting
testPredictPlot = np.empty_like(dataset[:,:1])
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict) + (look_back * 2) + 1:len(dataset) - 1, :] =
testPredict[:,:1]
```

```
In [18]: plt.plot(df[['min cpu']], label='Actual')
plt.plot(pd.DataFrame(trainPredictPlot, columns=['min cpu'], index=df.index),
label='Training')
plt.plot(pd.DataFrame(testPredictPlot, columns=['min cpu'], index=df.index),
label='Testing')
plt.legend(loc='best')
plt.title('Predictions for minimum CPU utilization')
plt.ylabel('min CPU utilization')
plt.xlabel('time step')
plt.show()
```



Graph showing minimum CPU utilization predictions

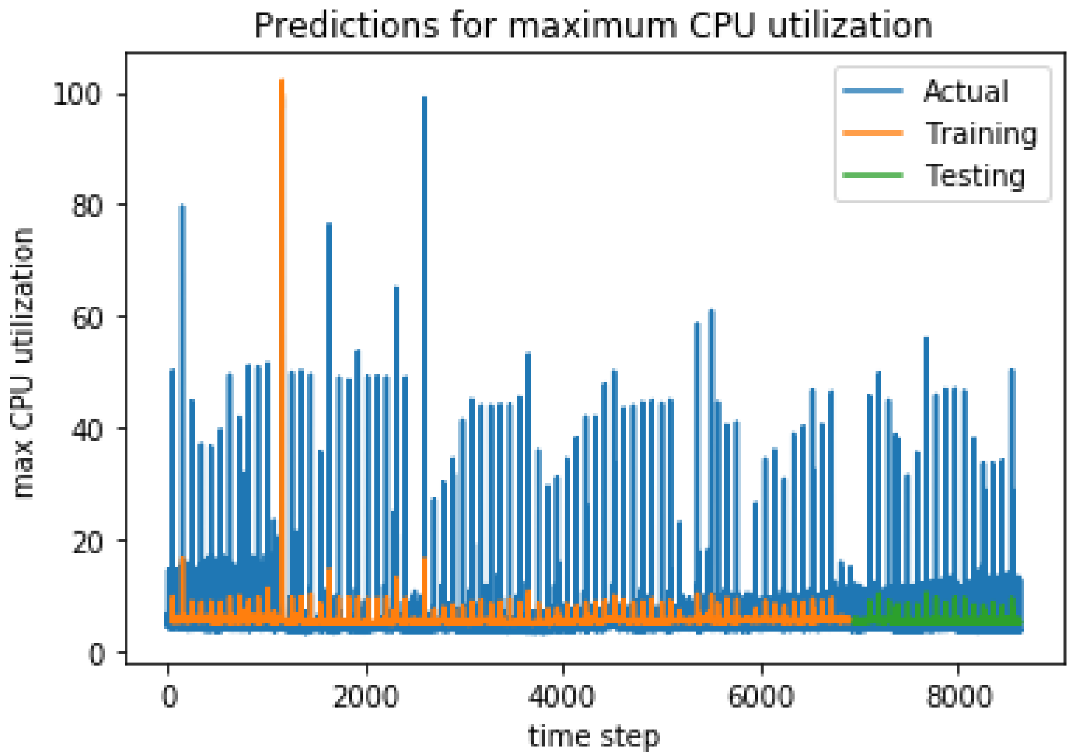
4.3.5 Plot for maximum CPU utilization

```
In [19]: # shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset[:,1:2])
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict[:,:1]) + look_back, :1] =
trainPredict[:,1:2]
```

```
In [20]:  # shift test predictions for plotting
testPredictPlot = np.empty_like(dataset[:,1:2])
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict) + (look_back * 2) + 1:len(dataset) - 1, :] =
testPredict[:,1:2]
```

```
In [21]: plt.plot(df[['max cpu']], label='Actual')
plt.plot(pd.DataFrame(trainPredictPlot, columns=['max cpu'], index=df.index),
label='Training')
plt.plot(pd.DataFrame(testPredictPlot, columns=['max cpu'], index=df.index),
label='Testing')
plt.legend(loc='best')
plt.title('Predictions for maximum CPU utilization')
```

```
plt.ylabel('max CPU utilization')
plt.xlabel('time step')
plt.show()
```



Graph showing maximum CPU utilization predictions

4.3.6 Plot for average CPU utilization

```
In [22]: # shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset[:,2:])
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict[:,:1]) + look_back, :1] =
trainPredict[:,2:]
```

```
In [23]: trainPredictPlot[100]
```

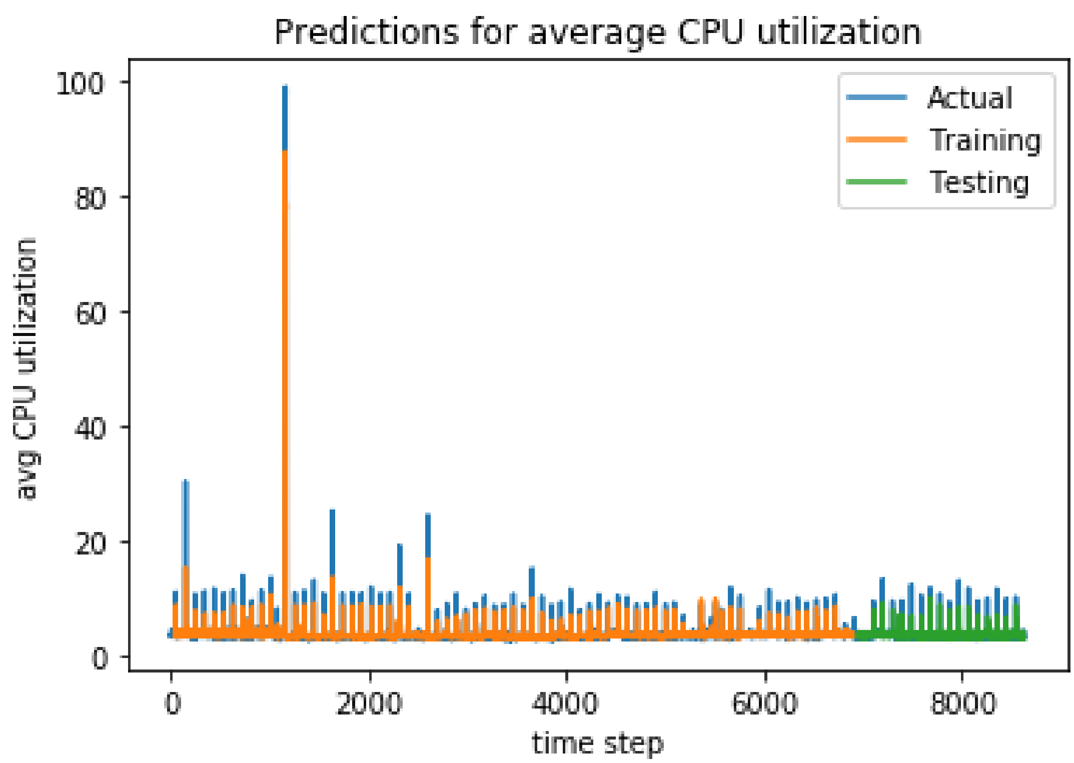
```
Out[23]: array([3.3520565], dtype=float32)
```

```
In [24]:  # shift test predictions for plotting
testPredictPlot = np.empty_like(dataset[:,2:])
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict) + (look_back * 2) + 1:len(dataset) - 1, :] =
testPredict[:,2:]
```

```
In [25]: testPredictPlot[7000:7005]
```

```
Out[25]: array([[3.2191951],
[3.0679529],
[3.152328 ],
[3.0347176],
[3.1627386]], dtype=float32)
```

```
In [26]: plt.plot(df[['avg cpu']], label='Actual')
plt.plot(pd.DataFrame(trainPredictPlot, columns=['avg cpu'], index=df.index),
label='Training')
plt.plot(pd.DataFrame(testPredictPlot, columns=['avg cpu'], index=df.index),
label='Testing')
plt.legend(loc='best')
plt.title('Predictions for average CPU utilization')
plt.ylabel('avg CPU utilization')
plt.xlabel('time step')
plt.show()
```



Graph showing average CPU utilization predictions

Chapter 5

Discussion and Future Work

5.0.0.1 Summarizing the Regression techniques

Model	R-squared on training data		R-squared on test data		RMS Error	
VM Type	Interactive	Delay Insensitive	Interactive	Delay Insensitive	Interactive	Delay Insensitive
Linear Regression	0.33	0.73	0.41	0.75	26.37	14.38
Lasso Regression	0.33	0.73	0.41	0.75	26.35	14.37
Ridge Regression	0.33	0.72	0.40	0.74	26.48	14.64
Support Vector Regression	0.27	0.67	0.33	0.71	27.93	15.54

Results of applying various regression techniques to predict 95 percentile CPU utilization

5.0.0.2 Summarizing the LSTM RNN prediction

We wanted to predict the 'min cpu','max cpu','avg cpu' values using the below LSTM RNN configuration: Lookback = 40 time steps

Trainable parameters: 143

4 LSTM units

3 output units

Optimizer used: Adam optimizer

Learning rate: 0.001

Minimizing mean square error loss

Train Score: 3.10 RMSE

Test Score: 2.73 RMSE

5.1 Discussion

The results observed come from analyzing a small subset of data released by Microsoft Azure. The models were trained on my personal computer with below configuration:

Processor: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s)

System Type: x64-based PC

Total Physical Memory: 15.9 GB

OS Name: Microsoft Windows 10 Home

We can expect to achieve a greater accuracy with a larger training dataset and more hyper parameter tuning on a better infrastructure like a powerful GPU cluster.

For the regression based prediction, we chose to predict the 95 percentile CPU usage data since it is more useful than predicting maximum cpu usage. Predicting the average CPU usage may also be a good choice in some use cases. The regression coefficients and correlation plots both indicate that Delay Insensitive VMs are more stable and predictable than Interactive VMs.

The focus was not much on getting the best model possible using hyper-parameter tuning but to create framework that is generic and that can be extended to other datasets. We believe that with resource metric data analysis, application development teams can better request infrastructure for their applications and avoid over-provisioning.

Future resource metric usage predictions can be to used in conjunction with VM scheduling algorithms as described in [13] for better results from optimizing algorithms.

5.2 Future Work

We could perform similar analysis on a different data set like Failure Trace Archive [34] or The Computer Failure Data Repository (CFDR) [35].

Newer algorithms such as reinforcement learning and attention model are evolving and it would be interesting to see how they perform with time series data.

Analyzing other features like memory usage or network I/O is another area that can be worked on.

We believe that making use of high power GPUs would help us analyze a larger dataset at once faster since modern machine learning frameworks like Tensorflow work well while training on a GPU. The prediction results can be used to simulate an actual datacenter on CloudSim. Ultimately, the goal should be testing the VM scheduling in a physical datacenter infrastructure using prediction results.

Chapter 6

Conclusion

We took a practical and data centric approach as opposed to developing mathematical models and optimization algorithms as seen in the related work section. As research continues in the field of Artificial Intelligence, Big Data Analysis, cloud and distributed computing, we can expect deriving greater value out of data center resource utilization prediction.

We recommend all organizations to start collecting data of their server infrastructure if they are not already doing so and to analyze for monitoring or prediction purposes. The accuracy of result is only going to improve as more data accumulates over the period of time.

References

[1] *NYPIUA*. http://www.nypiua.com/.

[2] *BMC Control-M*. http://www.bmc.com/it-solutions/control-m.html.

[3] Mary Meeker. Internet Trends 2017- Code Conference. *Internet Trends 2017- Code Conference*, page 355, 2017.

[4] Right Scale. State of the Cloud Report™ Data To Navigate Your Multi-Cloud Strategy. page 46, 2018.

[5] *Dot-com bubble*. https://en.wikipedia.org/wiki/Dot-com_bubble.

[6] Laura Hosman and Bruce Baikie. Solar-powered cloud computing datacenters. *IT Professional*, 15(2):15–21, 2013.

[7] M Dayarathna, Y Wen, and R Fan. Data Center Energy Consumption Modeling: A Survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2016.

[8] Right Scale. Optimizing Cloud Costs Through Continuous Collaboration The Challenge of Managing Cloud Costs.

[9] 451 Research. Can private cloud be cheaper than public cloud? Technical Report June, 2017.

[10] Intel. Virtualization and Cloud Computing Steps in the Evolution from Virtualization to Private Cloud Infrastructure as a Service. *Intel IT Center*, (August):24, 2013.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. *Ccs'17*, pages 1285–1298, 2017.

[12] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in IBM BlueGene/L event logs. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2007.

[13] Javad Ghaderi. Randomized algorithms for scheduling VMs in the cloud. *Proceedings - IEEE INFOCOM*, 2016-July:1–14, 2016.

[14] Google cluster data. Available: https://github.com/google/cluster-data.

[15] Xin Chen, Charng Da Lu, and Karthik Pattabiraman. Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2014.

[16] Yanpei Chen and Randy H Katz. Analysis and Lessons from a Publicly Available Google Cluster Trace. *System*, page 11, 2010.

[17] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *Proceedings - International Conference on Distributed Computing Systems*, 2017.

[18] OpenCloud Hadoop cluster trace: format and schema. Available: `http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html`.

[19] Operational Data to Support and Los Alamos National Laboratory. Enable Computer Science Research. Available: `http://institute.lanl.gov/data/fdata/`.

[20] Mbarka Soualhia, Foutse Khomh, and Andsofi Ene Tahar. Predicting Scheduling Failures in the Cloud. 2015.

[21] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. LiRCUP: Linear regression based CPU usage prediction algorithm for live migration of virtual machines in data centers. *Proceedings - 39th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2013*, pages 357–364, 2013.

[22] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[23] Yasuhiro Ajiro and Atsuhiro Tanaka. Improving packing algorithms for server consolidation. *Proceedings of the 33rd International Computer Measurement Group Conference (CMG),*, pages 299–406, 2007.

[24] Meng Wang, Xiaoqiao Meng, and Li Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. (L):71–75, 2011.

[25] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. *10th IFIP/IEEE International Symposium on Integrated Network Management 2007, IM '07*, 5:119–128, 2007.

[26] Nguyen Trung Hieu, Mario Di Francesco, and Antti Yla-Jaaski. Virtual Machine Consolidation with Usage Prediction for Energy-Efficient Cloud Data Centers. *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*, XX(X):750–757, 2015.

[27] Ashraf A Shahin. Automatic Cloud Resource Scaling Algorithm based on Long Short-Term Memory Recurrent Neural Network. *IJACSA) International Journal of Advanced Computer Science and Applications*, 7(12):279–285, 2016.

[28] Stefan Frey, Claudia Lüthje, and Christoph Reich. Key Performance Indicators for Cloud Computing SLAs.

[29] Reyhane Askari Hemmat and Abdelhakim Hafid. SLA Violation Prediction In Cloud Computing: A Machine Learning Perspective. 2016.

[30] Shailesh Paliwal. Performance Challenges in Cloud Computing. Available: https://www.cmg.org/wp-content/uploads/2014/03/1-Paliwal-Performance-Challenges-in-Cloud-Computing.pdf.

[31] A. Ragmani, A. E. Omri, N. Abghour, K. Moussaid, and M. Rida. A global performance analysis methodology: Case of cloud computing and logistics. In *2016 3rd International Conference on Logistics Operations Management (GOL)*, pages 1–8, May 2016.

[32] Azure public dataset. Available: https://github.com/Azure/AzurePublicDataset, 2017.

[33] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. *26th ACM Symposium on Operating Systems Principles*, pages 153–167, 2017.

[34] Failure Trace Archive. Available: http://fta.scem.uws.edu.au/index.php?n=Main.DataSets.

[35] The Computer Failure Data Repository (CFDR). Available: https://www.usenix.org/cfdr.