CS 6375 Machine Learning: Project Report

Thennannamalai Malligarjunan – txm230003

Job Failure Prediction In Cloud Environments

Abstract:

This project focuses on proactive job failure prediction in cloud computing environments to enhance resource efficiency and minimize operational costs. By leveraging the Google Cluster Trace 2011 dataset, we analyze job execution patterns and apply classification algorithms such as Logistic Regression, Random Forest, Gradient Boosting, Support Vector Machines, K-Nearest Neighbors, Naïve Bayes, and Neural Networks. The goal is to identify jobs likely to fail during execution, enabling timely rescheduling or termination to optimize resource utilization and reduce turnaround times.

The report outlines the experimental setup, including the application of hyperparameter tuning using GridSearchCV to identify the best-performing models. Key evaluation metrics such as training accuracy, test accuracy, F1 score, and confusion matrices are presented to assess model effectiveness across multiple trials. This project provides insights into how machine learning can proactively address job failures, contributing to more reliable and cost-effective cloud computing services.

1. Introduction:

Cloud computing is at the forefront of efficient and reliable delivery of services to businesses and the general consumers. It reduces cost on the end user due to its pay-as-you-go model that eliminates the initial infrastructure expenditure head. It also removes the burden of operations and hardware management on businesses as they can simply rely on the cloud company that they bought the services from. This ensures that businesses can focus entirely on the business logic side of things compared to spending resources such as time and money on configuring, scaling and deploying enough compute for daily operations.

Cloud service vendors these days have improved utilization of their compute resources by proper load balancing, dynamic scaling, etc... But running a cloud service incurs heavy operating costs due to power consumption, maintenance, security and audits. This is why cloud vendors are striving to further maximize the utilization of their compute resources thus improving their performance-per-dollar (PPD) ratio. In this project, we are looking to achieve this improvement by predicting job failures proactively so that it can be considered for rescheduling or termination based on user configurations.

Most cloud architectures follow the principle of running applications and services in the form of multiple jobs and tasks at the lowest level. A job failure can cause multiple issues such as resulting in other jobs queueing up, failure of dependent jobs and high turnaround times. If a job is suspended but marked as failed after some time, then it holds up the resource and causes other important jobs to fall behind leading to pile up. Cloud services have algorithms such as load balancing to minimize queue sizes and architecture designs such as master-slave, etc.. to handle job failures at the expense of redundancy. But redundancy adds to the cost of operations by increasing resource utilization. The algorithms and architecture designs handle job failures and its effects using a passive approach.

Our goal is to handle job failures proactively, which leads us to the option of creating a model of job failures such that a job while running over time can be classified as a job likely to fail in the future and can be rescheduled as soon as possible. This saves time by not wasting compute on a suspended process and multiple reschedules of the same job. In this project, we look at traditional machine learning algorithms (especially classification algorithms) such as Logistic Regression, Random Forest, Gradient Boosting, Support Vector Machines, K-Nearest Neighbors and Naïve Bayes. We also look at a deep learning approach utilizing a Neural Network.

2. Background:

In this section, the dataset used in this project is introduced, followed by some of its limitations with respect to this project.

2.1 Google Cluster Traces:

Up to date, Google has released three public trace datasets for research and academic use. The first one was released in 2007 and contains 7-hour workload details. The dataset contains only basic information, including time, job id, task id, job category, and number of cores and memory. Both the cores and the memory count have been normalized.

The second dataset contains traces of 29 days' workload from about 12,500 machines in the month of May 2011. The data consist of 672,074 jobs and around 26 million tasks that have been submitted by the user. Unlike the previous dataset, this one includes more information about each task's resource utilization, as well as information about the task itself, such as scheduling class and task priority.

The third dataset is the most current, documenting the use of cloud resources from eight separate clusters, with one cluster containing roughly 12,000 computers in May 2019, and was released in early 2020. Compared to the 2011 dataset, this dataset focuses on resource requests and utilization and contains no information about end users. The 2019 dataset has three additions: CPU utilization information histograms for each 5 minute period, information regarding shared resources reservation by a job, and job-parent information for master/worker relationships such as MapReduce's jobs.

2.2 2011 Dataset:

For this project we will be working with the Google Cloud Traces 2011 v2 dataset. This dataset represents 29 day’s worth of Borg cell information from May 2011, on a cluster of 12.5 k machines. Borg is a Large-scale cluster management system at Google that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

The dataset is split over six tables in .csv file format amounting to a total compressed dataset size of 40 GB. Among the six tables, we will consider the following four which still amount to approximately 35 GB of uncompressed data.

1. job_events
2. task_events
3. task_usage
4. machine_events.

Unique Identifiers:

Every job and every machine is assigned a unique 64-bit identifier. These IDs are never reused; however, machine IDs may stay the same when a machine is removed from the cluster and returned. Very rarely, job IDs may stay the same when the same job is stopped, reconfigured and restarted.

User and job names are hashed and provided as opaque base64-encoded strings that can be tested for equality.

Resource units:

Most resource utilization measurements and requests have been normalized, including:

- CPU (core count or core-seconds/second)
- memory (bytes)
- disk space (bytes)
- disk time fraction (I/O seconds/second)

For each of the foregoing, separate normalizations were computed. The normalization is a scaling relative to the largest capacity of the resource on any machine in the trace (which is 1.0). This is true even for disk space and disk time fraction.

Event_type:

The job_events and task_events tables both have a field named event_type. This field represents the multiple states of each job and (job,task) pair. The states are shown in the figure below and are described in detail following it.

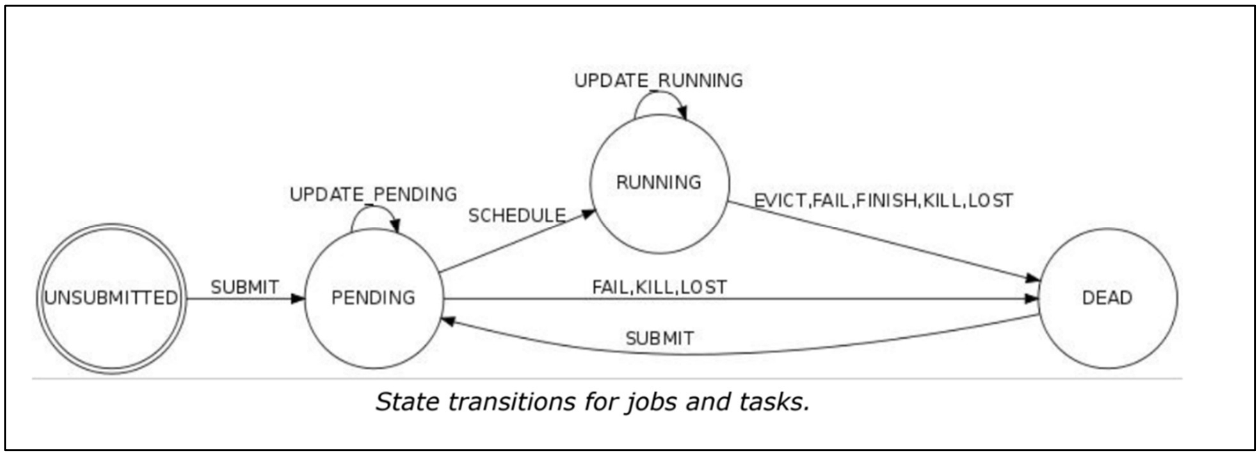

Figure 1: Job and task life cycles and event types

- SUBMIT (0): A task or job became eligible for scheduling.
- SCHEDULE (1): A job or task was scheduled on a machine. (It may not start running immediately due to code-shipping time, etc.) For jobs, this occurs the first time any task of the job is scheduled on a machine.
- EVICT(2): A task or job was descheduled because of a higher priority task or job, because the scheduler overcommitted and the actual demand exceeded the machine capacity, because the machine on which it was running became unusable (e.g. taken offline for repairs), or because a disk holding the task's data was lost.
- FAIL(3): A task or job was descheduled (or, in rare cases, ceased to be eligible for scheduling while it was pending) due to a task failure.
- FINISH(4): A task or job completed normally.
- KILL(5): A task or job was cancelled by the user or a driver program or because another job or task on which this job was dependent died.
- LOST(6): A task or job was presumably terminated, but a record indicating its termination was missing from our source data.
- UPDATE_PENDING(7): A task or job's scheduling class, resource requirements, or constraints were updated while it was waiting to be scheduled.
- UPDATE_RUNNING(8): A task or job's scheduling class, resource requirements, or constraints were updated while it was scheduled.

The task_usage table consists of fields such as CPU utilization rate, memory utilization rate, cache usage, disk I/O time, local disk space usage, etc… which are good indicators about a job's health which can in turn be used for predicting failure rate of a job.

The machine_events table consists of fields such as machine ID, platform ID, num of CPUs, memory availability which may or may not prove to be useful.

2.3 Limitations in this project:

Our computing resource is a single machine core-i7 quad core CPU with hyperthreading enabled, 16 Gigs of RAM and no GPU compute available. With this setup, we are unable to fully load the raw dataset into memory. So, we utilized approximately 10% of this data for our prediction models. After the cleaning and preprocessing steps, the dataset shrank even further.

Since we only utilized part of the complete trace, we are left with incomplete log traces for few jobs. So, we assume that the most general pattern from the data analysis step holds for all jobs in our subset of the data.

There are fields missing in multiple tables due to changes in Google's monitoring system, deliberate omissions and data loss.

Some fields in the task_usage table are inaccurate and only extend to approximately 40% of the trace.

3. Method:

3.1 Data Acquisition:

The data is stored in Google Storage for Developers. The data can be downloaded using the GSUtil utilities. GSUtiil needs to be installed and must be activated by signing into a google account in order to use this.

The dataset is downloaded using the command:

```
gsutil cp -R gs://clusterdata-2011-2/  destination-directory
```

Due to size constraints, the dataset is fetched as multiple partitions and each partition consists of a .csv file and compressed resulting in a .csv.gz format. There are a total of 500 partitions each for task_events, job_events and task_usage tables. This means that we need to combine all 500 partitions for each tables.

Due to limited compute resources, we combine only 50 partitions for job_events, task_events and task_usage tables.

In the preliminary exploration process, it was observed that few fields are missing for some entries in the tables. So we need to filter out those entries with missing fields as part of the data cleaning process.

Also, few fields are of type object. On further exploration of the data tables, it was seen that data was entered in string/int/float format for the same field resulting in a mixed datatype column. Conversion to the respective datatypes is also done in the data cleaning process.

3.2 Data Cleaning:

For the task_events table, the column “missing_info” is dropped as it is not providing useful information. Next the entries with missing “machine_id”, "cpu_request", "memory_request", "disk_space_request" values are dropped. The first entry in this table is dropped because of some values being corrupted. Next we convert object types into int/str/float types corresponding to the expected datatype for each field.

For the job_events table, the column “missing_info” is dropped for similar reasons as above. The “user”, “job_name” and “logical_job_name” columns are converted to type string. There were no missing entries in this table.

For machine_events table, entries with missing data for the fields “cpu_capacity”, "memory_capacity" are dropped. The “platform_id” field is converted to string type.

For task_usage table, entries with missing values for the columns 'max_disk_io_time', 'cycles_per_instruction', 'memory_acesses_per_instruction' are dropped. For one entry, the maximum_memory_usage value was the string ‘0.6872.1’ which could not be converted to floating point. So, this entry was dropped, and the remaining maximum_memory_usage entries were converted to float type.

3.3 Preprocessing:

We are done with cleaning the raw data. Now we need to do some processing so that the data makes sense.

Job Events:

The job_events table contains multiple datapoints for each job and each entry represents a state in which the job is at a time t. Since we don’t observe the full dataset, an assumption is made that the final entry for a job ID will always be either job FINISH, FAIL, KILL or LOST. This assumption is based partly on logical deduction and data exploration.

We will consider jobs that have a terminal state as FAIL to be FAILED jobs and the jobs with terminal states other than FAIL will be SUCCESS jobs. The reasoning for FAIL is trivial but for KILL and LOST to be considered as SUCCESS we infer that on KILL the user decided termination and for LOST, it means that a job has been descheduled and lost. For both these cases, the job need not block the queue which according to our goal is defined as SUCCESS.

We can say that if a job gets rescheduled multiple times, then there is a good chance that the job could fail. So, we count the state changes for a process and subtract one since we need not consider the terminal state. From the event_type field, we can get the terminal state for a job by selecting the final event_type value. This is stored in a new column “final_event_type”. The average of the scheduling _class values is considered for the job id since a scheduling class might change during execution of a job. Finally, we can also consider the runtime of a job to be a good indicator of a job’s failure chances. So, we calculate the difference between the max timestamp and min timestamp for every job id. The above details are implemented in the groupby aggregation on the job_events table resulting in a new table named “job_events_aggregated”. A new column named "target" is added to the job_events_aggregated table by considering jobs with final_event_type = 3 as FAIL(1) and the jobs with other final_event_type values as SUCCESS(0).

Looking at the counts for job failures and job success, it can be observed that the data is highly imbalanced.

Count(FAILED) = 947

Count(SUCCESS) = 67615

Pr(SUCCESS) = 0.986

This means that a model that just outputs positive will have an accuracy of 98.6%. This leads to the conclusion that we will need to rebalance either by under-sampling the SUCCESS data or over-sampling the FAILED data. We will look at rebalancing at the end of compilation of the 3 tables.

Task Events:

This table consists of the details for all tasks belonging to a job id. A unique record can be identified with the (timestamp, job_id, task_index) tuple. Here the important fields are "priority", "cpu_request", "memory_request", "disk_space_request" and "different_machine_constraint".

Looking at a few examples of the data:

For the Job with id 5357863310, we can observe that this job was started at timestamp 0 but its status was updated to FAIL at timestamp 238753813745. This leads to the fact that runtime of a job is also a good feature for predicting whether it will fail. The main observation here is that event_type for a task is not the same as event_type for a job. Because the event_type for a job is updated as FAIL but not for each task. The number of tasks spawned is also a good feature to test with.

Sometimes, the job hasn't been updated as FAIL but some of its tasks might have failed as is the case with job_id 6252870873. It may be due to the fact that we are unable to access the entire dataset or that the task was rescheduled again and it succeeded later. Most of the time, both are marked as failed.

We can get the number of retries for each task by aggregation over job_id. Hence, we can assume that the number of tasks retries is a good feature for job failure prediction. We can also get the number of task failures which can also be a relevant feature for job failure prediction. For the other columns, the max values of the task entries are taken in the aggregation based on the "job_id" column. The new table is named "tasks_events_aggregated".

The number of jobs in the task_events_aggregated table is much less than number of jobs in the job_events table. This means that after the merging job_events and task_events, the amount of data will decrease since it will be equal to the minimum of the number of rows in all tables involved in the merge. This is a bottleneck for the training data.

Machine events:

After exploring the data in this table, it was observed that not much useful information can be extracted here. Thus, we drop this table from further analysis.

Task usage:

From this table we can extract the number of machines each task has run on which can be relevant for the job failure prediction task. The entries are aggregated based on the (“job_id”, “task_index”) pair and the remaining column entries except “machine_id” column are averaged. For the “machine_id” column, we want the count of the unique machines which is obtained with ‘nunique’ parameter. The new aggregated table is named “task_usage_aggregated”. Another aggregation is done on this table over the job_id column. The max value of the aggregated entries is taken for all the columns. Few fields are removed in this aggregation to make the models less complex. The final aggregated table is named “task_usage_final_aggregated”.

The number of data points for this table is much lower than that of the task_events_table. Using this table in the final merged data table will considerably reduce the size of the data available. So we look at two variants of the merged data table and certain sampling techniques later on. The two variants of the combined data table are derived from combining task_events + job_events + task_usage and task_events + job_events.

To resolve the data imbalance issue, oversampling of the FAIL data is done using the SMOTE (Synthetic Minority Oversampling Technique) method. After this is done, both classes have the same amount of data entries.

3.4 Experiment setup:

Since at the fundamental level the problem of job failure prediction is a classification task, we use classification algorithms such as Logistic Regression, Random Forest Classifier, Gradient Boosting, Support Vector Machines, K-Nearest Neighbors, Naïve Bayes and Neural Networks.

This experiment consists of multiple trials over these models but each trail varies in data size and data properties. As the trials progress, more model evaluation metrics are added. The trials starts off with the base evaluation metric – training accuracy, test accuracy. As the trials progress, more data such as the F1 score, confusion matrix and other metrics are added.

Over each trial, the training mechanism is the same. For each algorithm we train multiple models over a hyper parameter space given by the param_grids variable with 5 fold cross validation. Using the GridSearchCV function provided in scikit-learn, we obtain the best model for each algorithm and evaluate them against the test data.

The hyper parameter space for each algorithm is as follows:
Logistic Regression

- C	: [0.1, 1, 10]
- Penalty	: ['l2']
- Solver	: ['liblinear']

Random Forest

- Number of Estimators (n_estimators)	: [100, 200, 500]
- Maximum Depth (max_depth)	: [None, 5, 10, 20, 30]
- Minimum Samples Split (min_samples_split)	: [2, 5, 10]
- Minimum Samples Leaf (min_samples_leaf)	: [1, 2, 4]

- Bootstrap	: [True, False]

Gradient Boosting

- Number of Estimators (n_estimators)	: [100, 200]
- Learning Rate (learning_rate)	: [0.01, 0.1, 0.2]
- Maximum Depth (max_depth)	: [3, 5, 7]

Support Vector Machine (SVM)

- C	: [0.1, 1, 10]
- Kernel	: ['linear', 'rbf']
- Gamma	: ['scale', 'auto']

K-Nearest Neighbors (KNN)

- Number of Neighbors (n_neighbors)	: [3, 5, 7, 10]
- Weights	: ['uniform', 'distance']
- Metric	: ['euclidean', 'manhattan']

Naive Bayes

- Variance Smoothing (var_smoothing)	: [1e-9, 1e-8, 1e-7]

Neural Network

- Hidden Layer Sizes (hidden_layer_sizes)	: [(50,), (100,), (100, 100)]
- Activation	: ['relu', 'tanh']
- Solver	: ['adam']
- Alpha	: [0.0001, 0.001]
- Learning Rate (learning_rate)	: ['constant', 'adaptive']

4. Data Analysis:

Before analyzing the data, we first split it into three sets namely (X_train, y_train), (X_train_val, y_train_val) and (X_test, y_test). As the name implies, the first set will be used for training, the second set will be used for validation of hyperparameters and the third set will be used for evaluation of the best models.

The correlation matrix of the (X_train, y_train) set obtained from the combination of task_events, job_events and task_usage tables is given in the figure below. From the figure, it is observed that target is positively correlated with job_id, task_failure_count, state_changes, runtime, max_avg_cpu_usage and max_avg_disk_io_time. The latter two are only slightly correlated with the target. The rest of the parameters are slightly correlated with target in the negative sense.

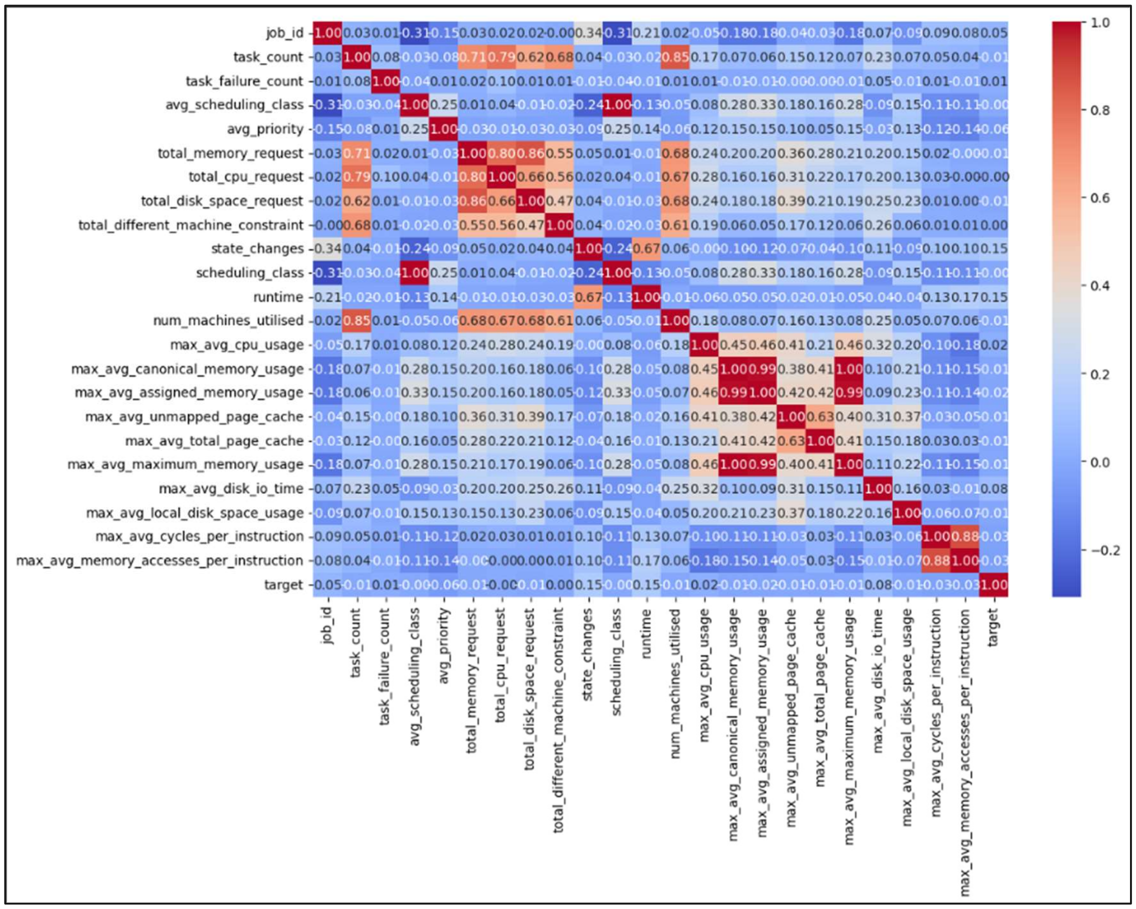

Figure 2: Correlation matrix of train data obtained by combination of task_event, job_event and task_usage.

From the above correlation matrix, it can be observed that parameters obtained from the task_usage table is not highly correlated with the targt. Thus, for the second variant we drop the table and combine only task_events and job_events.

The correlation matrix of the (X_train, y_train) set obtained from the combination of task_events and job_events tables is given in the figure below. From the figure, it is observed that target is positively correlated with job_id, task_failure_count, state_changes and runtime.

Figures 4 to 7 show box plots over numerical data to determine the presence and count of outliers. Outliers in numerical data can cause the model to be prone to overfitting. The model can become overly sensitive to these extreme values. This means the model may "learn" patterns based on the outliers, which do not represent the overall distribution of the data. This leads to a model that performs very well on the training data (because it fits the outliers too closely) but generalizes poorly with new, unseen data.

Further data analysis is done in the ML_Project.ipynb notebook.

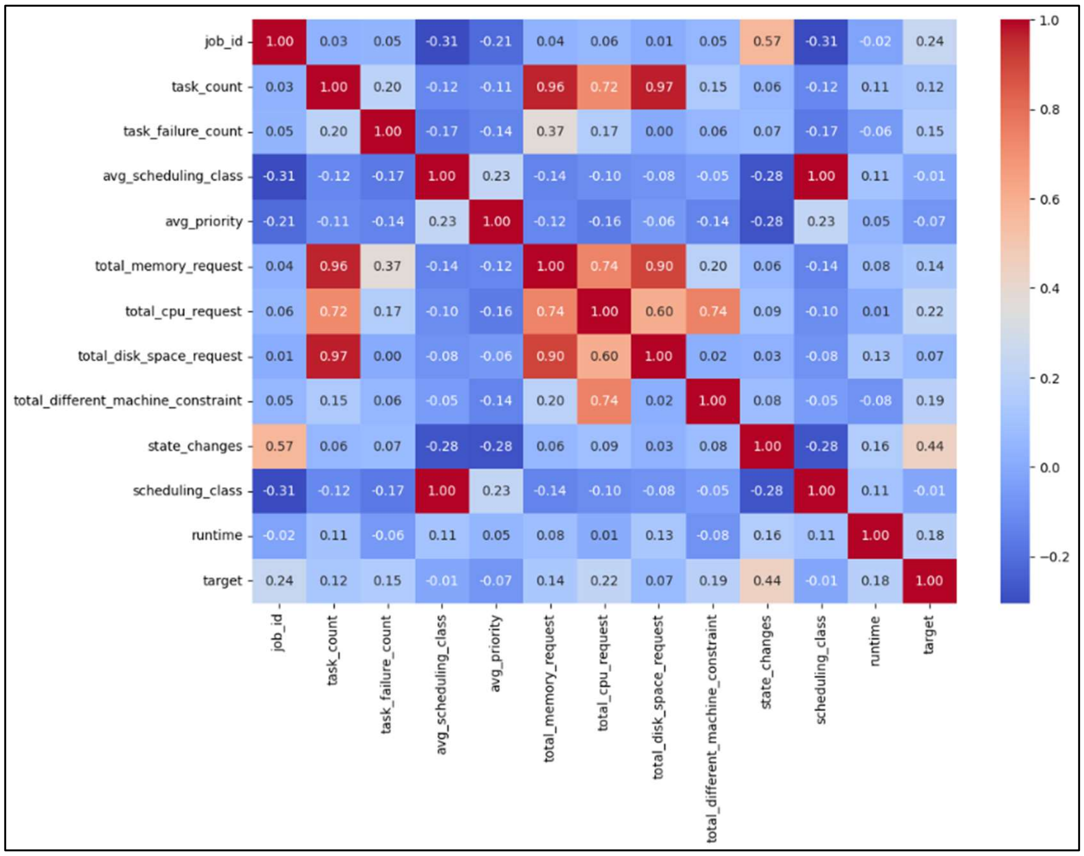

Figure 3:Correlation matrix of train data obtained from the combination of task_events and job_events table.

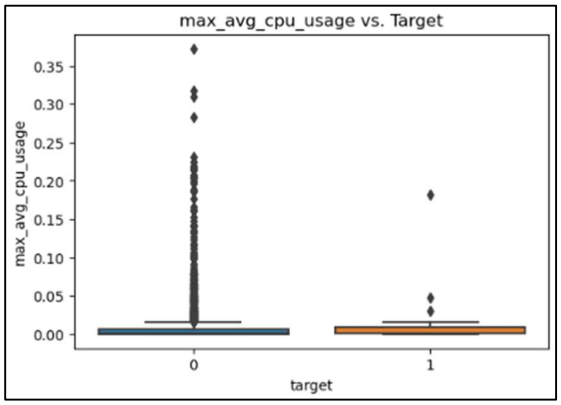

Figure 4: Box plot of max_avg_cpu_usage over target.

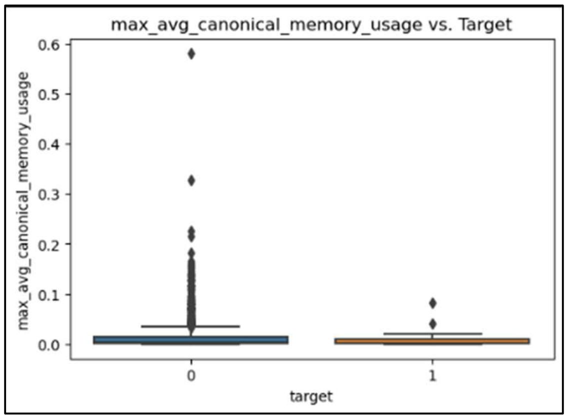

Figure 5: Box plot of max_avg_canonical_memory_usage over target

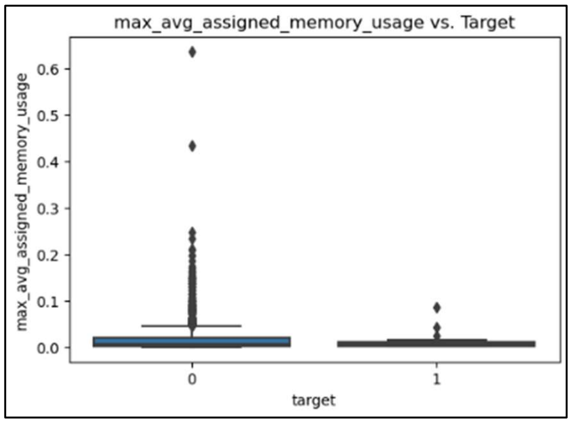

Figure 6: Box plot of max_avg_assigned_memory_usage over target

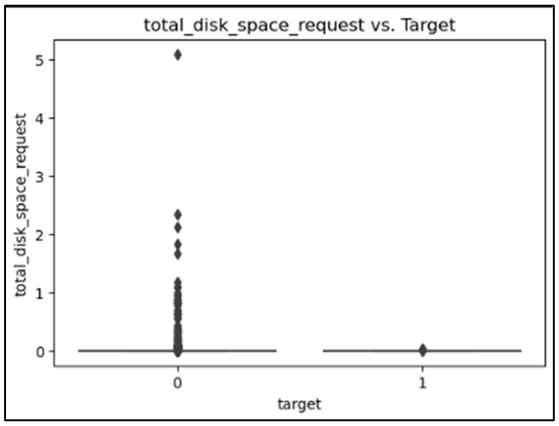

Figure 7: Box plot of total_disk_space_request over target.

5. Experiment results:

5.1 Trials

Trial 1:

Variant: [job_events + task_events + task_usage]

Dataset is imbalanced according to target.

Number of parameters: 23

Model	Accuracy	Class 0 Precision	Class 0 Recall	Class 0 F1-Score	Class 1 Precision	Class 1 Recall	Class 1 F1-Score	Weighted Avg Precision	Weighted Avg Recall	Weighted Avg F1-Score
Logistic Regression	0.9911	0.99	1	1	0	0	0	0.98	0.99	0.99
Random Forest	0.9911	0.99	1	1	0.5	0.14	0.22	0.99	0.99	0.99
Gradient Boosting	0.9911	0.99	1	1	0.5	0.14	0.22	0.99	0.99	0.99
SVM	0.9924	0.99	1	1	1	0.14	0.25	0.99	0.99	0.99
KNN	0.9899	0.99	1	0.99	0	0	0	0.98	0.99	0.99
Naive Bayes	0.9202	0.99	0.92	0.96	0.05	0.43	0.09	0.99	0.92	0.95
Neural Network	0.9924	0.99	1	1	0.67	0.29	0.4	0.99	0.99	0.99

Model	Best Parameters
Logistic Regression	{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}

Random Forest	{'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
Gradient Boosting	{'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}
SVM	{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
KNN	{'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}
Naive Bayes	{'var_smoothing': 1e-07}
Neural Network	{'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}

Trial 2:

Variant: [job_events + task_events]

Data is imbalanced according to target.

Number of parameters: 13

Model	Accuracy	Class 0 Precision	Class 0 Recall	Class 0 F1-Score	Class 1 Precision	Class 1 Recall	Class 1 F1-Score	Weighted Avg Precision	Weighted Avg Recall	Weighted Avg F1-Score
Logistic Regression	0.9861	0.99	1	0.99	0	0	0	0.97	0.99	0.98
Random Forest	0.9991	1	1	1	0.99	0.95	0.97	1	1	1
Gradient Boosting	0.9988	1	1	1	0.97	0.94	0.95	1	1	1
SVM	0.9866	0.99	1	0.99	0.89	0.04	0.08	0.99	0.99	0.98
KNN	0.9908	0.99	1	1	0.92	0.37	0.53	0.99	0.99	0.99
Naive Bayes	0.9729	0.99	0.99	0.99	0.02	0.02	0.02	0.97	0.97	0.97
Neural Network	0.9868	0.99	1	0.99	0.86	0.06	0.12	0.99	0.99	0.98

Model	Best Parameters
Logistic Regression	{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
Random Forest	{'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}
Gradient Boosting	{'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}

SVM	{'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
KNN	{'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}
Naive Bayes	{'var_smoothing': 1e-07}
Neural Network	{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 100), 'learning_rate': 'constant', 'solver': 'adam'}

Trial 3:

Variant: [job_events + task_events]

Data balanced by using SMOTE resampling on the minority class.

Number of parameters: 13

Model	Accuracy	Class 0 Precision	Class 0 Recall	Class 0 F1-Score	Class 1 Precision	Class 1 Recall	Class 1 F1-Score	Weighted Avg Precision	Weighted Avg Recall	Weighted Avg F1-Score
Logistic Regression	0.9957	1	1	1	1	0.18	0.31	1	1	0.99
Random Forest	0.9976	1	1	1	0.88	0.64	0.74	1	1	1
Gradient Boosting	0.9962	1	1	1	0.64	0.64	0.64	1	1	1
SVM	0.9957	1	1	1	1	0.18	0.31	1	1	0.99
KNN	0.9966	1	1	1	0.67	0.73	0.7	1	1	1
Naive Bayes	0.3930	1	0.39	0.56	0.01	0.91	0.02	0.99	0.39	0.56
Neural Network	0.9962	1	1	1	0.64	0.64	0.64	1	1	1

Model	**Best Parameters**
Logistic Regression	{'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Random Forest	{'bootstrap': True, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}
Gradient Boosting	{'learning_rate': 0.2, 'max_depth': 5, 'n_estimators': 200}
SVM	{'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
KNN	{'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}
Naive Bayes	{'var_smoothing': 1e-07}
Neural Network	{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 100), 'learning_rate': 'constant', 'solver': 'adam'}

The differences in the hyperparameters learned during each trial is given in the table below.

MODEL	TRIAL 1	TRIAL 2	TRIAL 3
LOGISTIC REGRESSION	{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}	{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}	{'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
RANDOM FOREST	{'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}	{'bootstrap': False, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}	{'bootstrap': True, 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}
GRADIENT BOOSTING	{'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}	{'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}	{'learning_rate': 0.2, 'max_depth': 5, 'n_estimators': 200}
SVM	{'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}	{'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}	{'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
KNN	{'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'uniform'}	{'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}	{'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}
NAIVE BAYES	{'var_smoothing': 1e-07}	{'var_smoothing': 1e-07}	{'var_smoothing': 1e-07}
NEURAL NETWORK	{'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}	{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 100), 'learning_rate': 'constant', 'solver': 'adam'}	{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 100), 'learning_rate': 'constant', 'solver': 'adam'}

Trial 4:

Variant: [job_events + task_events]

Data balanced by using SMOTE resampling on the minority class.

Number of parameters: 13

Dataset size is larger than Trial 3.

Model	Training Time (s)	Train Accuracy	Train F1	Test Accuracy	Test F1	Train-Test Gap	Bias	Variance
Logistic Regression	5.03	0.5798	0.3654	0.5777	0.359	0.0021	0.4202	0.0021
Random Forest	1503.48	1	1	0.998	0.998	0.002	0	0.002
Gradient Boosting	175.36	1	1	0.9982	0.9982	0.0018	0	0.0018
SVM	14400*	-	-	-	-	-	-	-
KNN	12.25	1	1	0.8565	0.8644	0.1435	0	0.1435
Naive Bayes	0.18	0.5974	0.7102	0.5967	0.7103	0.0007	0.4026	0.0007
Neural Network	63.99	0.6984	0.7466	0.6972	0.7449	0.0012	0.3016	0.0012

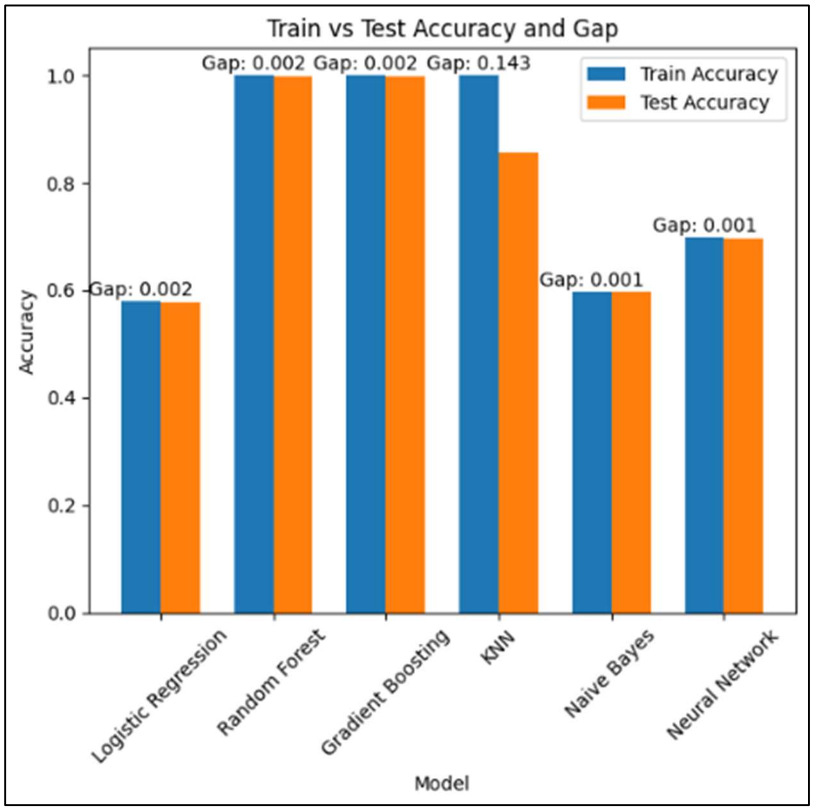

Figure 8: Train vs Test accuracy and respective gap.

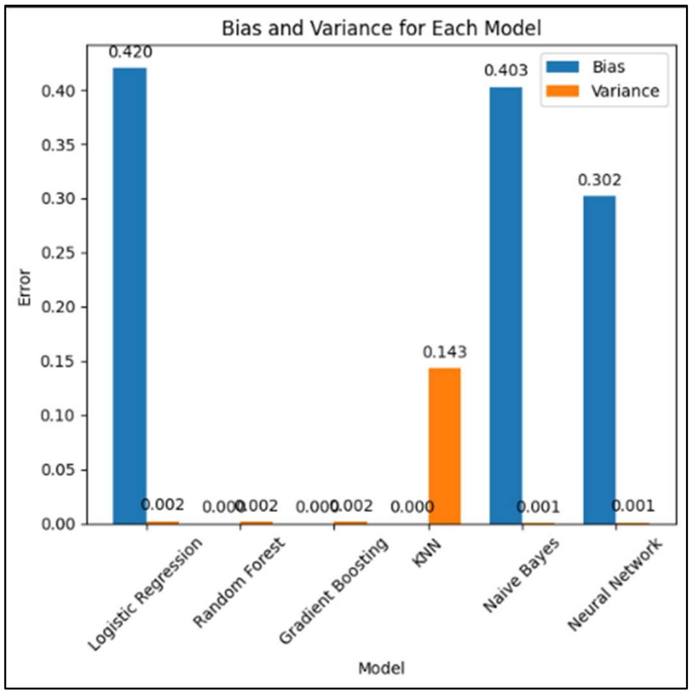

Figure 9: Bias and variance for each model

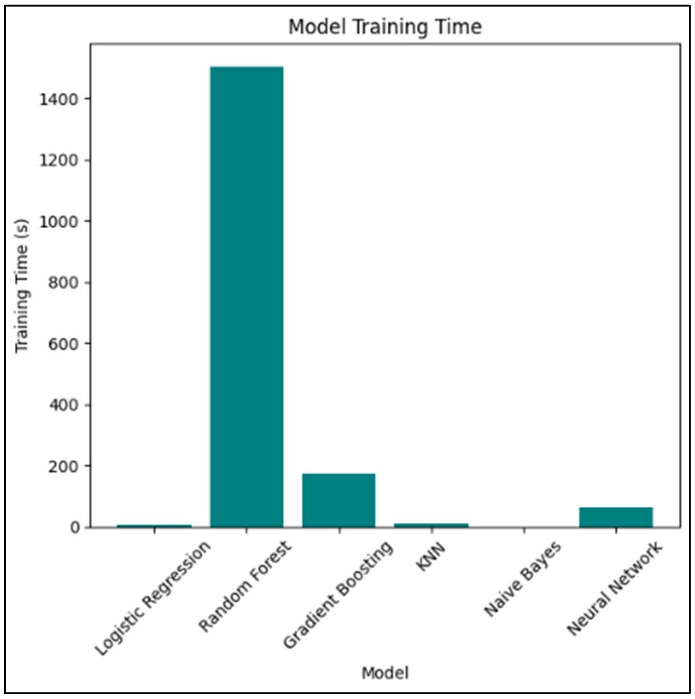

Figure 10: Training duration of each model in the hyperparameter space aggregated based on model.

5.2 Results:

It was mentioned previously that for imbalanced data, any model that predicts SUCCESS all the time will have an expected accuracy of 98.6%.

In trial 1, this was observed firsthand. All models have a good accuracy score. So the only way to measure model performance is to use metrics such as precision, recall and F1 scores. From the table for trial 1, it can be observed that the precision, recall and F1 scores for class 1 (FAIL) are very low indicating poor model performance for all models.

In trial 2, the only change from trial 1 is the final combined data used. This leads to more data samples available for training with higher counts of FAIL and SUCCESS classes. But the ratio is still imbalanced here. From the table, it can be seen that the precision, recall and F1 scores for class 1 (FAIL) have improved for all models except Logistic Regression and Navie Bayes. This might be because the other models inherently have low bias characteristics which enables the models to learn from the few available examples of class 1.

In trial 3, the change from trial 2 is the removal of the imbalance by using SMOTE over sampling. This also increases the number of data samples but the only hindrance here is that the minority class was over sampled to have the same count as the majority class. The extra data is synthetic and not representative of the real world which may negatively impact training by causing overfitting.

In trial 3, the Logistic Regression model is still unable to learn class 1 from the fact that the class 1 precision, recall and F1 scores are very low. The accuracy remains approximately the same as the previous trial for logistic regression. But now Naïve Bayes has degraded performance for classifying both class 1 and 0. This is due to the fact that the probability distributions have changed because of the availability of more samples for y=1. The accuracy for Naïve Bayes classifier also plummeted.

Trial 4 is different from trial 3 only in the number of training examples used. The scores are similar to trial 3 for the models other than logistic regression and neural networks. These model's accuracy dropped to 60% and 80% respectively from 99% in the previous trial. This may indicate high bias in these models.

6. Conclusion:

This project aimed to predict job failures in cloud computing environments using machine learning algorithms, with the goal of proactively rescheduling or terminating jobs to optimize resource utilization and minimize operational costs. Through the application of traditional machine learning models, including Logistic Regression, Random Forest, Gradient Boosting, Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Naïve Bayes, and Neural Networks, we were able to predict job failure risks by generating models and evaluate their performance.

The results revealed that while all models exhibited good overall accuracy, the imbalanced nature of the dataset made accuracy an unreliable metric. To address this, precision, recall, and F1 scores were used, providing a more comprehensive view of model performance, particularly for the minority class (FAIL). Models such as Random Forest and Gradient Boosting showed better adaptability to the imbalance due to their low bias characteristics, while others, such as Logistic Regression and Naïve Bayes, struggled to predict the minority class effectively.

Furthermore, the use of SMOTE to address class imbalance introduced synthetic data, which, while improving representation, also posed the risk of overfitting, particularly for Logistic Regression and Naïve Bayes. These findings highlight the challenges of working with imbalanced datasets and the importance of selecting appropriate models and techniques for handling such data.

In conclusion, this project demonstrates the potential of machine learning for predicting job failures in cloud computing, although further refinement of model selection and data handling techniques is necessary to improve predictive performance, especially for the minority class. Future research could explore models that account for job failures based on execution in specific machines, co-dependency of jobs and rather than just accounting for the count of state changes we can identify models that preserve the timeline of each state change.

7. References:

1. Tengku Asmawi, T.N., Ismail, A. & Shen, J. Cloud failure prediction based on traditional machine learning and deep learning. *J Cloud Comp* **11**, 47 (2022). https://doi.org/10.1186/s13677-022-00327-0
2. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md
3. https://research.google/pubs/large-scale-cluster-management-at-google-with-borg/
4. https://drive.google.com/file/d/0B5g07T_gRDg9Z0lsSTEtTWtpOW8/view?resourcekey=0-cozD56gA4fUDdrkHnLJSrQ