Your Live Failure Triage Assistant in VMware

An approach to categorize, rerun, and group your failed tests by analyzing logs based on Deep Learning

Abstract

Triaging test Failures, including environment preparation failures, is one of the main work for the QE(Quality Engineer)s in our company. We have observed some common bug-triaging patterns that take much effort which can be eased and make bug-triaging work more efficiently thanks to the development of Natural Language Processing on Deep Learning.
In this paper, I would like to propose and introduce a deep-learning-based first-hand auto-triaging solution that reviews all failure logs, helps to make categories, do similarity clustering, and apply automated operations(rerun for example) for some intermittent failed cases.

Introduction

To fully explain the solution implementation(chapter **Collaborations and Applications**), several topics will be introduced in this paper. The chapter **Concepts and Algorithms** contains concepts and algorithms used in the system written for easy understanding. The chapter **Feasibility Analysis** contains several necessary conditions summarized from my experience in doing log analysis with deep learning(Ignoring this part will not affect the understanding of the solution). The **System Design** contains the description of all main components at the service level. The chapter **Application Scenarios of BERT** contains three ways of using BERT for our collaborations concretely. The chapter **Collaborations and Applications** contains two real collaboration cases that apply the scenarios of using BERT built on top of those services. The Chapter **Model Accuracy** contains model accuracy matrix for two collaborations. The chapter **Future Works for Application Scenarios** contains two planned future works based on the current application scenarios of BERT.

Concepts and Algorithms

Multi-Class Classification

A Binary Classification is a classification that has only two exclusive results, for example, the result for coin flipping. A multi-class classification means we will have more exclusive results with an overall probability of 100%.

Softmax

The softmax function**,** also known as softargmax or normalized exponential function, converts a vector of K real numbers into a probability distribution of K possible outcomes. In our scenarios, Softmax is used to transform the model outputs into multi-class classification results.

Singular Value Decomposition

In linear algebra, the Singular Value Decomposition (SVD) of a matrix is a factorization of that matrix(M) into three matrices(UEV*) in the format as M = UEV*. In this formula, E represents a list of topics, U represents the relationship between the rows to the topics and V represents the relationship between the columns to the topics. For example, for a movie rank matrix M, if the rows are different people and columns are different movies, after the SVD, U represents the relationship between people to some topics, V represents the relationship between topics to movies, and E represents some topics, for example, action movies.

Transformers

Transformers is a Deep Learning model algorithm that does supervise learning with groups of three-matrices multiplication helped by many functions to accelerate this process [1]. According to the effect of SVD, we could understand Transformers in this way: it learns lots of topics and the relationship between different items to topics from the training dataset.

BERT

BERT(Bidirectional Encoder Representations from Transformers) is a pre-trained NLP model based on Transformers with a well-defined input format [2]. It defines two phases when using the model: pre-training and fine-tuning. During pre-training, it uses a very large quantity of materials of sentences to build the understanding between words to words and sentences to sentences. During fine-tuning, we could add different neural network layers on top of it to do further training based on the pre-trained results. This reduces duplicate efforts when training models.

As a part of the design of BERT, by modifying the input format, we could use it to make multi-class classification predictions based on single sentences or pairs of sentences. Detail will be introduced in the **Application Scenarios of BERT**.

Sentence-BERT

The Sentence BERT model is a BERT-based fine-tuned model which could convert sentences into vectors [3]. During the training phase, it accepts three items – two sentences and an anchor. By using a Triplet Loss function with a Siamese Network, for the sentence that has a close meaning to the anchor, they will have a better cosine similarity and vice versa. Because of this, all sentences would have a position in the hyper-dimensional space it defines. Those sentences that have similar meanings will have close positions in the hyper-dimensional space. This model is used to do hyper-dimensional embedding for similarity clustering for logs in our scenario. More detail will be shared in **Application Scenarios of BERT** III.

DBSCAN

DBSCAN (Density-based spatial clustering of applications with noise) Is a clustering algorithm that doesn't require cluster quantity. It groups points that are close to each other based on a distance measurement (usually Euclidean distance) and the distance is defined as "eps" (aka Epsilon).

Feasibility Analysis

We have made many attempts during design and implementation when cooperating with some teams and ended with nil. It is hard to be trusted and build collaborative relationships when sharing inappropriate scope definitions or maintenance efforts. The inappropriate scope could lead to low accuracy of the final model. For example, if we want to use logs from the support bundle to do supervised training to predict what Bugzilla PCC it should belong to, we will face the difficulty of filtering key logs. Among millions of lines in the support bundle, no matter what pre-selection function is applied, as we cannot get rid of noise logs, we cannot train a highly accurate model. What's even worse is, there could even be multiple root causes in a support bundle from a failed test – but only some of them are been interested.
I have summarized four necessary conditions for using machine learning to do log analysis.

Firstly, the log we want to train and predict shouldn't change too quickly. If the log changes day by day and we cannot see any similarity in the new log compared with the old log, from the word it uses or the meaning of the word it expresses, like, for some cases the word "not equal" means an expected result while in other cases it means an unexpected result and should be treated as a failure, the machine learning is not suitable. The reason is simple: we cannot use former sentences to predict unseen sentences or similar sentences with different meanings.

Secondly, restricted by the algorithm, concepts like quantity, number, duration, frequency, or some flags will not be useful. Also, the length of the sentence is limited to at most 512 words. For example, if we have an abnormal situation that can only be observed from a JSON configuration, more specifically, a "True" or "False" or even a number in the JSON configuration, it's not in the current scope we could handle. And for the 512 words limitation, we now have the solution to reduce the number of lines if we are facing a large number of lines to do classification.

Thirdly, we need to have a method to decide if a prediction should be trusted. In this part, we use the Softmax result of a multi-class prediction to be the confidence. The Softmax result shows how much weight the model wants to put on the target prediction. The weight here has a similar meaning as "confidence" – if a model makes a prediction that all classes have a balanced weight, this means the sentence is confusing to be any of the predicted classes. In this case, we cannot really trust this prediction result because we don't define labels in this way. A failure must belong to a certain label we defined, if not, we need to split this label. By monitoring the "confidence", we could decide if we should trust the prediction and when we need to train a new model.

Finally, as we use supervised learning, the training dataset should be a by-product of daily work – we could easily collect the log and the label. With all the above conditions satisfied, we designed and implemented the system.

System Design

Dependency Services

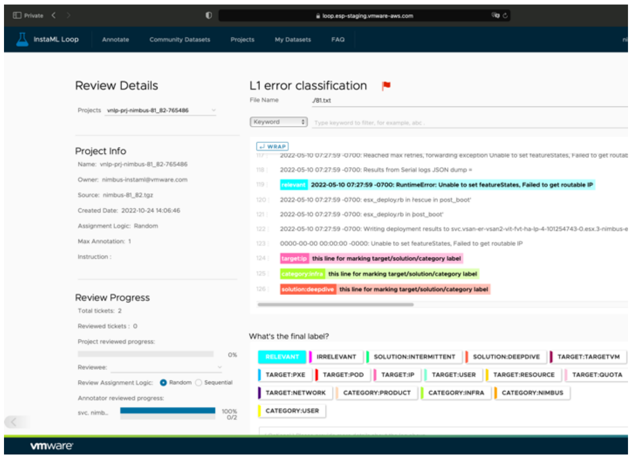
InstaML vNLP Preprocess

Raw logs	Preprocessed logs
05:13:34,005 | Failed at Play [deploy_vm_efi_paravirtual_vmxnet3] *********	timestamp failed at play deploy vm efi para virtual vmxnet number
2022-12-05 05:13:34,005 | TASK [deploy_vm_efi_paravirtual_vmxnet3][Upload local file to ESXi datastore] task path: /home/worker/workspace/Ansible_OracleLinux_8.x_MAIN_PARAVIRTUAL_VMXNET3_EFI/ansible-vsphere-gos-validation/common/esxi_upload_datastore_file.yml:11	timestamp task deploy vm efi para virtual vmxnet number upload local file to esxi data store task path home worker workspace ansible oracle linux number x main para virtual vmxnet number efi ansible vsphere guest os validation common esxi upload data store file yml

<example.table, from and to>

A live sentence preprocess service that can help to recognize and transform meaningless tokens in sentences, for example, timestamps, and mac addresses. It can also split compound words and amend them with a VMware-specific dictionary. As the model and the algorithm we use require a normalized sentence, we need the vNLP service to do the preprocessing for logs first.

InstaML LOOP labeling platform



Review Details

Projects: vnlp-prj-nimbus-81_82-765486

Project Info

Name: vnlp-prj-nimbus-81_82-765486
Owner: nimbus-instaml@vmware.com
Source: nimbus-81_82.tgz
Created Date: 2022-10-24 14:06:46
Assignment Logic: Random
Max Annotation: 1
Instruction :

Review Progress

Total tickets: 2
Reviewed tickets : 0
Project reviewed progress: 0%
Reviewee:
Review Assignment Logic: [✓ Random] Sequential
Annotator reviewed progress: svc. nimb.. 100% 0/2

L1 error classification

File Name: ./81.txt

```
2022-05-10 07:27:59 -0700: Reached max retries, forwarding exception Unable to set featureStates, Failed to get routab
2022-05-10 07:27:59 -0700: Results from Serial logs JSON dump =
relevant 2022-05-10 07:27:59 -0700: RuntimeError: Unable to set featureStates, Failed to get routable IP
2022-05-10 07:27:59 -0700: esx_deploy.rb in rescue in post_boot'
2022-05-10 07:27:59 -0700: esx_deploy.rb in post_boot'
2022-05-10 07:27:59 -0700: Writing deployment results to svc.vsan-er-vsan2-vit-fvt-ha-lp-4-101254743-0.esx.3-nimbus-e
0000-00-00 00:00:00 -0000: Unable to set featureStates, Failed to get routable IP
target:ip this line for marking target/solution/category label
category:infra this line for marking target/solution/category label
solution:deepdive this line for marking target/solution/category label
```

What's the final label?

<screenshot for LOOP>

A live labeling platform that could do labeling work on lines or passages. By using API, we could upload or download datasets.

InstaML Training

A live training platform with strong GPU. All models mentioned in this paper are trained(fine-tuned) from this platform.

Others

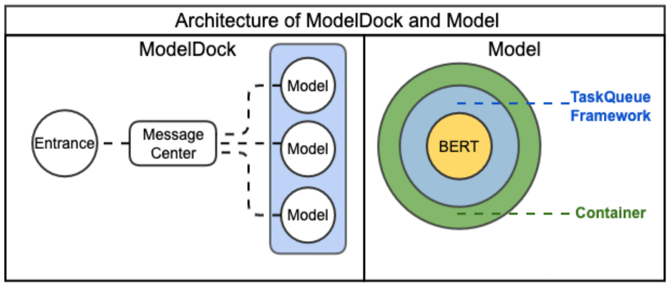
DECC – a specific Kubernetes environment maintained by the DECC team. ModelDock is deployed in one of the DECC namespaces. Nimbus – the test environment cloud from which VMs could be deployed and organized easily.

LabelSet Definition

LabelSet is a set of labels that represents possible causes from a specific point of view. The definition of label and LabelSet should consider what to do when we get the label from the model as a prediction result. Linear independence is required between the definition of LabelSet as well as the labels among the same LabelSet. The reason is simple: by having a confusing label, we could not apply for any manual or automated work after receiving it. Instead, we could apply a manual playbook or script for each label. Because of this, we could consider this label as a class, which could be trained and predicted by the multi-class text classification model – BERT, in our solution. What's more, instead of using one model to cover all LabelSets, I propose to use one model for each LabelSet – we have a function to calculate the contribution of each word from the predicted result. Separating models to each LabelSet could support this function, which is useful when investigating the state-of-art natural language model. To make it more concrete, three LabelSets are defined for our scenarios currently – Failure Solution, Failure Target, and Failure Category. A Failure Solution represents what we could do for this failure, for example, rerun. Failure Target represents what target the failure has a relationship with, for example, a Nimbus Testbed. A Failure Category represents who is suggested to do the manual triage, for example, Nimbus SRE. For different scenarios, labels could actually be defined differently based on their own triaging requirements.

CPU-based ModelDock

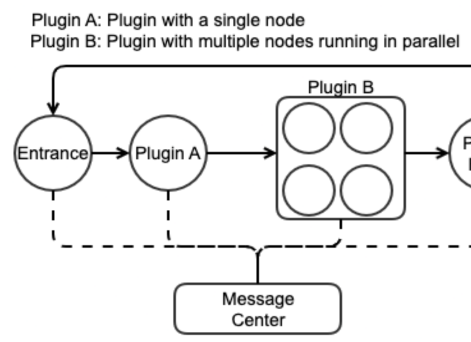
ModelDock is an architecture of a distributed container-based cluster that focuses on scalability and robustness, including a possible hybrid environment that contains both GPU and CPU for inference in the future. Currently, we are using a dedicated DECC namespace to deploy all those models.



As shown in the diagram, the ModelDock is implemented based on the message queue. Three components are designed – the Entrance Server, the Message Center, and the Models. The Entrance Server will accept prediction requests and support querying the result. Each Model will listen to a specific channel from Message Center and send predicted results back to it. There are several advantages of this architecture. Firstly, we could measure the calculation performance requirement by monitoring the number of requests in the Message Center. Secondly, we could easily expand or shrink the configuration of each Model and the number of Models. Thirdly, we could upgrade the Models one by one with zero downtime. Finally, we could have multiple environments registered to the same Message Center, as mentioned above, mixed GPU-based and CPU-based Models in the future.

Pipeline Framework

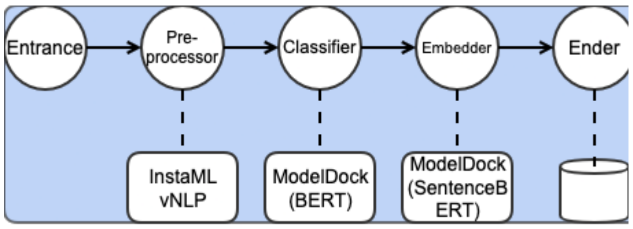
Pipeline Framework is a pipeline-style framework that maximizes the flexibility, versatility, and scalability to support real-time data(log) stream handling. Currently, they are deployed on Nimbus.


Plugin A: Plugin with a single node
Plugin B: Plugin with multiple nodes running in parallel
Plugin B
Entrance
Plugin A
Plugin Final
Message Center

<pipeline framework.jpg>

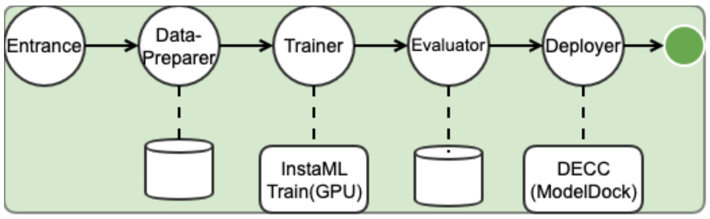
As shown in the diagram, apart from the Entrance Server and the Message Center(same architecture as the ModelDock), the Pipeline Framework has a key unit – Pipeline Plugin. Usually, a Pipeline Plugin represents a specific step when handling a data stream. It could be some simple logic that connects to an external service, for example, the ModelDock. It could also be some CPU-bound or IO-bound work including formatting logs or reading logs from files. Due to this requirement, a Pipeline Plugin supports multiple nodes running in parallel(as Plugin B). What's more, the configuration of a Pipeline Plugin includes an upstream and a downstream Pipeline Plugin – a Pipeline Plugin will read the output from the upstream Pipeline Plugin and write to the downstream Pipeline Plugin. Thanks to this architecture, we could have strong versatility and flexibility. We could deploy multiple nodes for some special needs and we could orchestrate the pipeline in any form for any needs – we could connect two pipelines or more specially, we could connect a pipeline to one plugin from another pipeline. The special architecture will be used in Collaboration II.

Inference Pipeline



With some concrete configuration, we could build an Inference Pipeline that covers a common scenario of doing inference using the Pipeline Framework. As displayed in the diagram, A Preprocessor connects to the InstaML vNLP platform which will formalize raw sentences. An Operator connects to the ModelDock to get the sentence predicted and the Embedder transforms sentences into vectors. The final result will be saved to the DB, which could be used by other services, for example, the Failure Triage Page. Usually, a hook will be installed into the client that runs the test. Failure logs will be collected and sent to the Entrance Server of the Inference Pipeline. What's more, the inference pipeline could be connected – one output from an inference pipeline could be the input for another inference pipeline. More detail will be shared in Collaboration-II.

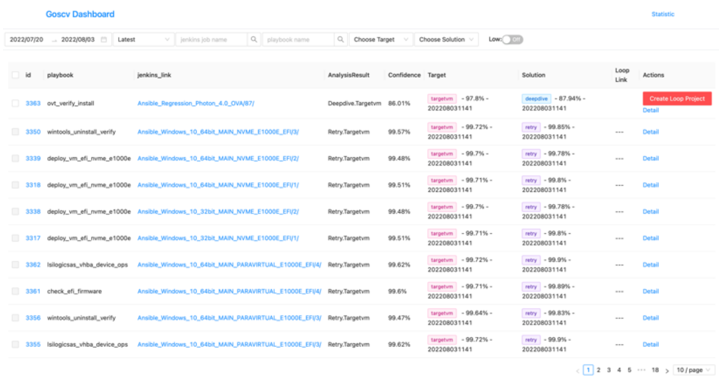
Training Pipeline



A training pipeline is implemented to cover common scenarios of training. As displayed in the diagram(Key Plugins are kept only), A DataPreparer reads all training dataset from DB and format them to meet the needs of training. The Trainer connects to the InstaML Trainer to train the model with the training dataset from DataPreparer. The Evaluator will calculate the average confidence for both the new model and the old model to make sure the new model has an advantage in accuracy for the past sentences. Because of this, former predictions in the past several days(which can be configured) will be re-predicted and overridden. After the new model is evaluated, the Deployer will build the model into the container and replace the old models in ModelDock.

Confidence Monitoring Page

As a supervised learning solution, labeling and training are unavoidable. As mentioned in **Feasibility Analysis** condition-III, we use Softmax result as the confidence to distinguish if or not a prediction should be trusted or not, and also if a new training should be triggered or not. Based on this, we designed and implemented the **Confidence Monitoring Page** – a page to monitor predictions and upgrade the model if we are having too many new logs. Every Collaboration team will have its own tests, models, and the **Confidence Monitoring Page**.

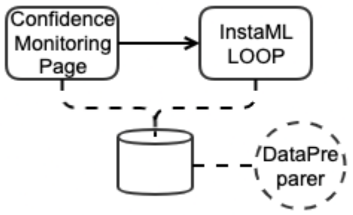


id	playbook	jenkins_link	AnalysisResult	Confidence	Target	Solution	Loop Link	Actions
3363	ovt_verify_install	Ansible_Regression_Photon_4.0_OVA/87/	Deepdive.Targetvm	86.01%	targetvm - 97.8% - 202208031141	deepdive - 87.94% - 202208031141		Create Loop Project Detail
3350	wintools_uninstall_verify	Ansible_Windows_10_64bit_MAIN_NVME_E1000E_EFI/3/	Retry.Targetvm	99.57%	targetvm - 99.72% - 202208031141	retry - 99.85% - 202208031141	---	Detail
3339	deploy_vm_efi_nvme_e1000e	Ansible_Windows_10_64bit_MAIN_NVME_E1000E_EFI/2/	Retry.Targetvm	99.48%	targetvm - 99.7% - 202208031141	retry - 99.78% - 202208031141	---	Detail
3318	deploy_vm_efi_nvme_e1000e	Ansible_Windows_10_64bit_MAIN_NVME_E1000E_EFI/1/	Retry.Targetvm	99.51%	targetvm - 99.71% - 202208031141	retry - 99.8% - 202208031141	---	Detail
3338	deploy_vm_efi_nvme_e1000e	Ansible_Windows_10_32bit_MAIN_NVME_E1000E_EFI/2/	Retry.Targetvm	99.48%	targetvm - 99.7% - 202208031141	retry - 99.78% - 202208031141	---	Detail
3317	deploy_vm_efi_nvme_e1000e	Ansible_Windows_10_32bit_MAIN_NVME_E1000E_EFI/1/	Retry.Targetvm	99.51%	targetvm - 99.71% - 202208031141	retry - 99.8% - 202208031141	---	Detail
3362	lsilogicsas_vhba_device_ops	Ansible_Windows_10_64bit_MAIN_PARAVIRTUAL_E1000E_EFI/4/	Retry.Targetvm	99.62%	targetvm - 99.72% - 202208031141	retry - 99.9% - 202208031141	---	Detail
3361	check_efi_firmware	Ansible_Windows_10_64bit_MAIN_PARAVIRTUAL_E1000E_EFI/4/	Retry.Targetvm	99.6%	targetvm - 99.71% - 202208031141	retry - 99.89% - 202208031141	---	Detail
3356	wintools_uninstall_verify	Ansible_Windows_10_64bit_MAIN_PARAVIRTUAL_E1000E_EFI/3/	Retry.Targetvm	99.47%	targetvm - 99.64% - 202208031141	retry - 99.83% - 202208031141	---	Detail
3355	lsilogicsas_vhba_device_ops	Ansible_Windows_10_64bit_MAIN_PARAVIRTUAL_E1000E_EFI/3/	Retry.Targetvm	99.62%	targetvm - 99.72% - 202208031141	retry - 99.9% - 202208031141	---	Detail

< 1 2 3 4 5 ··· 18 > 10 / page

<monitoring page.jpg>

As displayed in the screenshot, we use Red Bars to indicate a low-confidence prediction and Green Bars to indicate a high-confidence prediction. What's more, we can select all those predictions with low confidence, clustered by their embedding similarity, and create a labeling project in InstaML Loop.

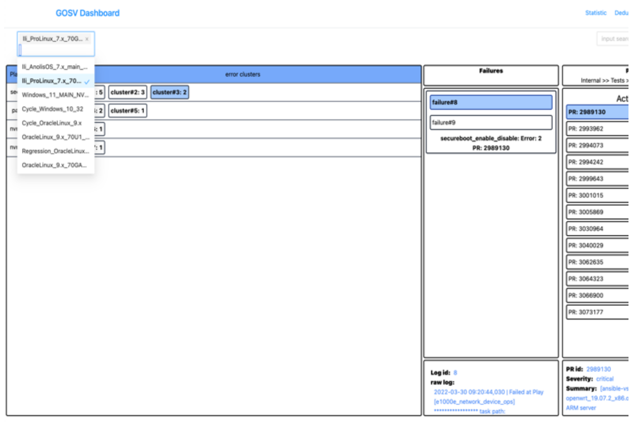


Once the labeling work is finished, the labeling result will be saved as a training dataset and a training task will be triggered from the training pipeline. The training pipeline will re-predict all former predictions with the new model – we will see low confidence records turn to high after this.

Failure Triage Page

As described in the abstract, the key point of this project is to "review all failures", "retry those intermittent failures" and "group the rest based on similarity".

First of all, let's assume that we have a LabelSet – Solution with "*retry*" and "*deepdive*" in it. When a failure occurs, if we get the solution predicted as "*retry*", a certain mechanism should be triggered to rerun (depending on different collaborations). On the other hand, if we get the solution predicted as "*deepdive*", we need to make similarity clustering and display the result to the user.



secureboot_enable_disable: Error: 2
PR: 2989130

PR: 2989130
PR: 2993962
PR: 2994073
PR: 2994242
PR: 2999643
PR: 3001015
PR: 3005869
PR: 3030964
PR: 3040029
PR: 3062635
PR: 3064323
PR: 3066900
PR: 3073177

```
Log id: 8
raw log:
2022-03-30 09:20:44,030 | Failed at Play
[e1000e_network_device_ops]
****************** task path:
```

PR id: 2989130
Severity: critical
Summary: [ansible-vs
openwrt_19.07.2_x86.
ARM server

<support page.jpg>

The support page could be configured – the rows, the selections, and even the "PR" panel can be changed depending on different business scenarios. By considering various of cases team by team, we could extract some concepts – metadata, failure cluster, and external failure entity. Failure entity, as the most simple concept, could be either Bugzilla PR or Jira Story. The backend server has a connection with the two external failure tracking services, which means, some common operations could be applied from this Failure Triage Page to them. A failure Cluster is a cluster of failure logs grouped by a similarity algorithm, more specifically, DBSCAN. As mentioned in Inference Pipeline, all logs will be embedded by SentenceBERT. With the embedded result, the similarity algorithm could be applied and we could achieve failure clusters. Metadata is the information that could either be or not be extracted from the log, which affects the result a lot but could not be recognized by SentenceBERT. For example, two different test pipelines with two different but very similar names produced very similar test logs. Because of the business requirements, we need to track them by using different external failure entities, for example, Bugzilla PRs. Because of this, the name here has a very special weight but can not be recognized by SentenceBERT – they come from business requirements. In this case, they will be grouped into a cluster and we cannot make training for this – this will modify the meaning of the words in this model, even if they might not be the test name in other situations. We need to extract all those information and mark them as metadata. Based on this, we will do similarity clustering separated by those metadata. What's more, we could build a connection between the external failure entity and the cluster. This will impact all failures in this cluster, no matter it has been clustered or will be clustered in the future. Once the cluster has a connection with an external failure entity, new failures will also be treated belongs to this external failure entity. After failures are clustered, we will see them in the monitoring panel – sorted by quantity and grouped by the metadata.

Application Scenarios of BERT

A typical way of using BERT is to do multi-class text classification. By adding a Linear Layer with a Softmax Output Layer, BERT could be used to do multi-class classification. Fortunately, we can use BertForSequenceClassification class built by Huggingface, one of the main contributors to BERT.

Scenario-I: Multi-class Text Classification with BERT for a single sentence

By giving BERT the input format as "[CLS] <SENTENCE> [SEP]", BERT could be used to handle a single sentence. "[CLS]" and "[SEP]" here are special tokens designed and used in BERT, where "[CLS]" (aka "class") stands for the understanding of the whole "<SENTENCE>" and the "[SEP]" (aka "Separate") stand for the ending of this sentence. The sentence will be treated as a sequence of tokens (together with "[CLS]" and "[SEP]") and each of them will be encoded one by one by BERT Tokenizer. As in BERT, each token has its own encoded format. The encoded sequence will be sent to BERT and further trained with Linear Layer on top of it. We could make it predict our sentence with a label from the target LabelSet after the training.

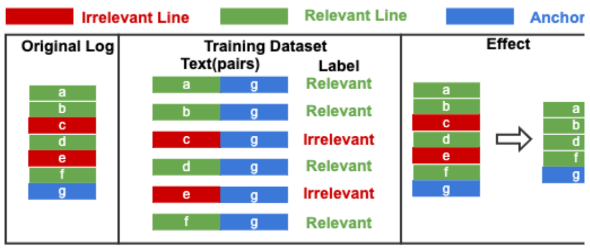
<example of input, encoded, and output>

Scenario I will is suitable for handling a short passage of which less than 512 words(after preprocessing). The sentence with its labels will be built as a training dataset and model input for BERT.

<example of input, labels?>

Scenario-II: Multi-class Text Classification with BERT for pair sentences

BERT is designed to be capable to handle pairs of sentences by giving BERT the input format as "[CLS] <SENTENCE A> [SEP] <SENTENCE B> [SEP]". In the pre-trained phase, the pair of sentences would have a relationship like neighbors – one sentence should be right after the other from the original passage. But as designed, we are free to change the meaning of the pair of sentences. And as experimented, the result is promising. Inspired by Triplet Loss Function, which is commonly used in facial recognition, we need to design and pick one line (could also be not a line) and make it the anchor for a long passage with many lines of logs. We will use all other lines to compare with this line and train the model – if the target line is relevant to the anchor or not. In this case, the Multi-Class classification problem is simplified to Binary Classification but we could still use the BertForSequenceClassification with the target label set "relevant", and "irrelevant" to handle this question. With this approach, we could train a model to filter all irrelevant lines to reduce the number of noises. This could help to shrink the passage to satisfy the need for sentence length for Scenario-I of BERT.



One key point for this scenario is the anchor. We could choose an explicit error message to be the anchor in the target logs, or we could use some other information to replace it. Test name, filename, and any of that information could be used as the anchor. Choosing a stable and representative sentence composed of common words is a good practice. More detail will be unveiled in Collaboration-II. After choosing the anchor and the target line, we can simply replace the "<SENTENCE A>" and "<SENTENCE B>" with them. The sequence of those sentences will not affect the model's accuracy.

Scenario-III: Using SentenceBERT for sentence embedding and DBSCAN for similarity clustering

As described in **Concepts and Algorithms**, SentenceBERT uses the Siamese Network with Triplet Loss Function to train the cosine similarity between sentences based on BERT. As a result, we could use it to embed sentences to 762 dimensions of vectors. If two sentences are similar to each other in meaning, the two sentences will have similar positions in the 762 hyper-dimensional space. By using **DBSCAN** with tuned eps, we could cluster them without knowing the cluster quantity. There is one restriction that should be noticed in this scenario. As described in Failure Triage Page, there are some words with special meanings beyond the sentence that cannot be learned by the model. For example, even though the rest of the logs are totally the same, failures should be tracked by different PR and should be categorized into different groups due to a slight change like OS type, for example, "Ubuntu" to "Windows". On the contrary, they will be grouped into the same cluster without any mandatory function which is actually expected behavior. They should be in the same cluster because they have almost the same meaning; they shouldn't be in the same cluster because they are words with privileges we give to them based on business requirements. For this case, we cannot simply fine-tune the model. The difference between "Ubuntu" and "Windows" will be amplified to affect other cases for which those words don't have any privilege and this will pollute the model. Because of this, we need to define a concept called "metadata" to cover those business rules.

Collaboration and Application Examples

Collaboration-I: Failure triage for GuestOS Validation Automation Test

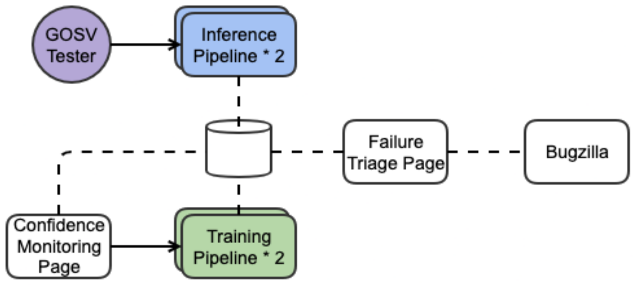
```
2022-11-22 06:44:13,022 | Failed at Play
[deploy_vm_efi_paravirtual_vmxnet3] *********
2022-11-22 06:44:13,022 | TASK
[deploy_vm_efi_paravirtual_vmxnet3][Upload local file to ESXi
datastore]
task path:
/home/worker/workspace/Ansible_OracleLinux_8.x_MAIN_PARA
VIRTUAL_VMXNET3_EFI/ansible-vsphere-gos-
validation/common/esxi_upload_datastore_file.yml:11
exception in /vsphere_copy.py when main in /request.py when
http_error_default
fatal: [localhost]: FAILED! => HTTP Error 500: Internal Server
Error
error message: HTTP Error 500: Internal Server Error
2022-11-22 06:44:14,022 | TASK
[deploy_vm_efi_paravirtual_vmxnet3][Get specified property info
for VM 'test_vm']
task path:
/home/worker/workspace/Ansible_OracleLinux_8.x_MAIN_PARA
VIRTUAL_VMXNET3_EFI/ansible-vsphere-gos-
validation/common/vm_get_config.yml:4
fatal: [localhost]: FAILED! => Unable to gather information for
non-existing VM test_vm
error message: Unable to gather information for non-existing VM
test_vm
```

<log sample for GOSV>

The first collaboration example is from GOSV Team (aka GuestOS Validation Team). Based on their log format and length, a combination of Scenario-I and Scenario-III is used to support the first-hand auto triage.

<screenshot page.> → **Confidence Monitoring Page**

We define two LabelSets: solution {*deepdive*; *retry*} and target {*targetvm*; *testbed*; *nimbus*; *testcase*; *usererror*; *infra*}. Based on the label definition, scripts are prepared for specific combinations of failure solution and failure target. For example, for different combinations with the solution – *retry*, we will try to rerun the test after "renewing" the test environment depending on the different targets. More specifically, if we see a prediction result is *retry.nimbus*, a new Nimbus deployment will be applied to replace the current test environment; or if it's *retry.targetvm*, a restart operation with a test rerun will be applied to the specific test target. And for solution – *deepdive*, they will be clustered based on the similarity calculated from SentenceBERT embedding. The result will be organized in the Failure Triage page, which could help testers to handle similar failures at once.



For Collaboration-I, we could use a Training Pipeline and an Inference Pipeline directly with only some slight changes. We deployed 2 models for the solution and 2 models for the target in the ModelDock based on their request frequencies. When a test failure is raised, the log will be sent to the Inference Pipeline to get the solution, the target, and an embedded vector. We set the confidence threshold to 90%. A prediction result will be treated as a low-confidence event once the confidence of it is lower than the threshold. If we don't have any confidence issues, for those predictions the solutions are *retry*, a retry mechanism is added to the client side. It will apply different kinds of retry strategies depending on different targets. For the solution predicted as *deepdive,* they will be grouped by the metadata and then clustered based on the similarity of the embedded vectors. Once a cluster is connected to a Bugzilla PR, all failures in this cluster will be linked to the Bugzilla PR including those that will be raised in the future. Based on this, several common operations could be supported. Job links and failure files could be attached automatically; failures could be queried by PR number and vice versa. Different teams could choose different support actions to ease their triage efforts.

Collaboration-II: Failure triage for Nimbus ESX deployment

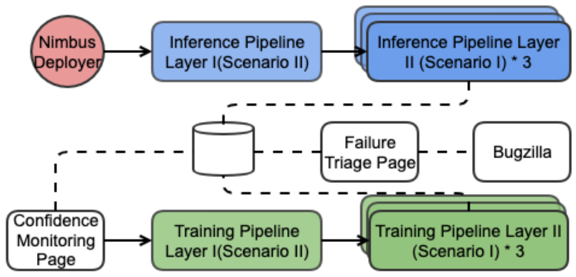
```
......
2022-11-20 09:32:37 -0800: nimbus_ip_listener.rb#246: ip_waiter:
both ip4 and ip6 are ok.
2022-11-20 09:32:37 -0800: base_deploy.rb#2543: Got guest ip =
10.92.210.93
2022-11-20 10:02:47 -0800: base_deploy.rb#1491: Reached max
retries, forwarding exception Installer not done!!!
2022-11-20 10:02:47 -0800: esx_deploy.rb#2160: Results from
Serial logs JSON dump = {"state"=>"pxe_boot_completed",
"phases"=>{"phase_preload"=>{"state"=>"completed",
"start_time"=>"2022-11-20 09:27:21 -0800", "end_time"=>"2022-
11-20 09:27:21 -0800", "phase_time"=>0.0},
"phase_vmk_loading"=>{"state"=>"completed",
"start_time"=>"2022-11-20 09:27:21 -0800", "end_time"=>"2022-
11-20 09:27:21 -0800", "phase_time"=>0.0},
"phase_vmkernel_initialize"=>{"state"=>"step_migration",
"start_time"=>"2022-11-20 09:27:12 -0800", "phase_time"=>9.0},
"phase_modules_load"=>{"state"=>"completed"},
"phase_pxe_boot"=>{"state"=>"completed", "start_time"=>"2022-
11-20 09:27:16 -0800"}}, "events"=>{},
"ip_addr"=>"10.92.210.93"}
2022-11-20 10:05:03 -0800: base_deploy.rb#1521:
NimbusExceptionServiceNotUp: Installer not done!!!
```

<log sample.jpg?>

The second collaboration example is from Nimbus Team which is focused on ESX deployment failures. Based on the log format and length, a combination of Scenario I, II, and III is used to support the first-hand auto triage.

We defined one more LabelSet compared with collaboration-I: the category {*product*; *infra*; *nimbus*; *user*}. The category is used to suggest who should take the first look at the problem. More specifically, one of the common problems we are facing when triaging ESX deployment failures is, as the tester or Nimbus SRE engineers don't have knowledge from each other, both sides need to check all failures, some of which are only specific to one side. With the category suggested, we could reduce the effort.

Currently, we haven't added any retry or rerun mechanism for deployments. Instead, we collect and do similarity clustering for all failures, grouped by Nimbus pods. By sorting those grouped failures by quantity, we will have an overview of what is happening for all Nimbus Pods, which is helpful to distribute our maintenance workloads efficiently.


Nimbus Deployer
Inference Pipeline Layer I(Scenario II)
Inference Pipeline Layer II (Scenario I) * 3
Failure Triage Page
Bugzilla
Confidence Monitoring Page
Training Pipeline Layer I(Scenario II)
Training Pipeline Layer II (Scenario I) * 3

To make an error classification, we use the Nimbus-raised exception line as an anchor line to filter all relevant lines first(by Scenario II). By selecting all high-confidence "relevant" lines, merged with the anchor line, we could create a long sentence and send it to error classification models to make predictions(by Scenario I). The same merged sentence will be sent to SentenceBERT for embedding(by Scenario-III). As a result, we will get relevant lines, targets, solutions, and category predictions with an embedding vector as outputs. The outputs will be saved for different purposes.

One main difference when compared with Collaboration-I is that we are using a stacked architecture of models – the second layer(of models) do predictions based on the predictions from the first layer(of models). More concretely, the first layer is used to filter relevant logs based on the anchor exception message. The results will be merged as a whole sentence and the second layer will do the multi-class classification for target, solution, and category. This affects the architecture of training and inference. We stack two Training pipelines and two Inference Pipelines. When doing inference, we split the error logs line by line and merge each of them with the anchor to build pairs of sentences – each pair will be predicted once. All lines with high confidence predicted result "relevant" will be selected and merged with the anchor at the end of the sentence, and will be the input of the second Inference Pipeline. This happens similarly when doing training – once the layer I model is trained, the lines in the training dataset will be predicted and selected to be the training dataset for the layer II models. The reason to make the output from the layer I model to be the input of layer II in the training phase is to take mispredictions from the Layer I model as a part of the regular situation. Asking the predictions from Layer I to be fully accurate is not possible and this will lower the performance of layer II models. One last thing to mention is, as line filtering has much more calculation efforts compared to classification, we deployed 16 models for Layer I and 1 model for each LabelSet for Layer II.

Model Performance

Training Sample Example

	Processed Log		LabelSet: Label
Collabor-ation I Scenario-I	timestamp failed at play go sc cloud init static ip timestamp task guest os customization failed fatal localhost failed vm static ip v4 address is ip address expected ip v4 address is ip address vm static ip v4 gateway is ip address expected ip v4 gateway is ip address		target: solution
	Log	**Anchor**	
Collabor-ation II Scenario-II	timestamp number cloning vm svc vm platform blr vcqa cat number number esx number	timestamp number runtime fault runtime fault summary	relationship: irrelevant

Model Accuracy

Collaborations	Phases \ Properties	Accuracy	Precision	Recall	F1-Score
Collaboration I	Train (80%)	0.99	0.99	0.99	0.99
Scenario-I	Test (10%)	0.944954	0.950081	0.950081	0.950081
(1087 samples)	Evaluate (10%)	0.880733	0.896163	0.848172	0.859435
Collaboration II (Scenario-II)	Train (80%)	0.99	0.99	0.99	0.99
	Test (10%)	0.963812	0.925377	0.901163	0.912777
(8498 samples)	Evaluate (10%)	0.957941	0.912288	0.885299	0.898156

Future Works for Application Scenarios

A Complete Solution for Triaging Nimbus Failure Deployments

As a test environment provider, ESX is just one of the VM types Nimbus support. We plan to cover most of the common deployment types for Nimbus, for example, VC VM, and NSX VM. What's more, we decided to design a solution to handle different prediction results. For example, if we receive a *retry.quota* as the prediction for the failure, we could collect the deployment parameters and queue them into our schedule launcher with a temporary quota checker – once the condition is satisfied, we could retry the deployment automatically. Last but not least, we plan to add more integration between Nimbus and Bugzilla, and JIRA to make failure triage easier.

Support Bundle Analysis

Support Bundle, as a full copy of logs in the test VM, contains process and thread information of services, for example, vpxd. Those logs are important when locating the root cause of a test failure. To support triaging those failures, challenges need to be overcome. As mentioned in Feasibility Analysis, we have had some approaches which failed eventually – we didn't have any efficient solution when handling long logs, especially when we have tens of thousands of lines. What's more, we have no idea to filter even more specific lines from all abnormal lines to meet special interest for the test – even though we might have multiple unexpected situations for services, QE and DEV are only interested in the one they tested.

To summarize, lacking an approach to filter specific logs is the key problem of Support Bundle Analysis. Fortunately, we have developed Scenario II which could support this case with a restriction – the anchor. The second question is how to choose the anchor. It should represent the test intention precisely and can be located easily. Based on this, we are considering the tester's description, including the Summary and the Short Description in Bugzilla, to be the anchor – those descriptions represent the test intention and the error description, which is useful to filter logs.

With all feasibility conditions satisfied, we plan to initialize some experiments in the next year with the scope of locating relevant lines based on the tester's description.

Acknowledgments

Thanks to my mates: Heng Zhang, Anurag, and Lu Chang who helped to review the paper; Rick Battle, Steve Liang, and Giampiero who offered ML technique suggestions and training support; Team GOSV for close collaborations in the early phase of this project; Ligang and Jingdi for their professional engineer works.

References

[1] AshishVaswani,NoamShazeer,NikiParmar,JakobUszkoreit, LlionJones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018

[3] Nils Reimers, Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, 2019