Language Modeling using Vertex AI

Arpitha Gurumurthy, Bhuvana Basapur, Mayuri Lalwani, Somya Mishra

SJSU Davidson College of Engineering Extended Studies, Department of Software Engineering, San Jose, CA, US

***Abstract*— With the rise of social media in recent years, the usage of data for different purposes like election prediction, sentiment analysis, marketing, business, communication has also increased day by day. In this paper we have implemented different language modeling tasks like text classification, sentiment analysis, question answering and summarization. We have achieved this by fine tuning the pretrained BERT model and automated all the required steps of the ML system like data collection, model deployment and serving using GCP and Vertex AI.**

***Keywords*— Text Classification, Summarization, MLOps, Sentiment Analysis, Question Answering, Summarization, VertexAI, BERT, HuggingFace, Streamlit, Docker, TensorBoard, Pytorch.**

I. Introduction

Text classification, summarization, Question Answering and Sentiment Analysis are important tasks of language modeling. Language Modeling provides a tool that analyzes the pattern of human language for the prediction of words. BERT is the core component for natural language processing (NLP). It has the ability to process larger amounts of text and language. We have leveraged this ability to perform the language modeling tasks by fine tuning the pretrained BERT model and automated all the ML steps like data collection, model deployment and serving by using GCP and Vertex AI.

We have performed the following language modeling related tasks using pre-trained models:

1. Text Classification
2. Text Summarization
3. Question Answering

II. Related Work

The common medium for sharing ideas and conversations between humans is text. With the immeasurable conversations happening over the online platforms, comes extensive amounts of text documents. Therefore in recent years, many researchers have come up with different variants of Bert models which can be easily fine tuned for language modeling tasks. Prior work on question answering has been done by Yogatama et al. (2019) [10] who evaluated a model across four datasets and Talmar and Berant(2019)[11] comprehensively evaluated question answering models using ten datasets. Our model uses Stanford question answering dataset and DistilBert which is fast, cheap and trained by distilling Bert base.

For summarization typical Deep Learning models involve Seq2Seq architectures with RNN's (Nallapati et al). Transformers which are often combined with attention mechanisms (Vaswani et al., 2017; Lewis et al., 2020; Raffel et al., 2020a). But these models do not work well with financial statements and to overcome this HuggingFace introduced the Pegasus summarizer model. Using this model along with different user decoding strategies, users will be able to understand important words in the text and a gist will be generated.

III. Dataset

Below are the datasets we have used for our project for each of the language modeling tasks:

1. Emotion Classification

Emotion from Hugging Face Datasets: We have used this dataset for emotion classification by fine

tuning the pretrained BERT model. The dataset contains labels belonging to multiple classes. It is a dataset of Twitter messages with six basic emotions: anger, fear, joy, love, sadness, and surprise.

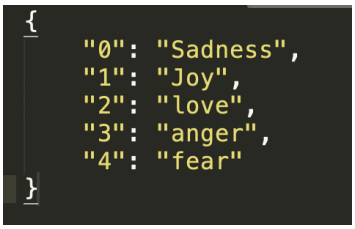


Fig 1: Multiclass labels of the emotion classification

Below screenshot shows a preview of the dataset:

text (string)	label (class label)
i didnt feel humiliated	sadness
i can go from feeling so hopeless to so damned hopeful just from being around someone who cares…	sadness
im grabbing a minute to post i feel greedy wrong	anger
i am ever feeling nostalgic about the fireplace i will know that it is still on the property	love
i am feeling grouchy	anger
ive been feeling a little burdened lately wasnt sure why that was	sadness
ive been taking or milligrams or times recommended amount and ive fallen asleep a lot faster but i…	surprise
i feel as confused about life as a teenager or as jaded as a year old man	fear

Fig 2: Preview of the emotion dataset

2. Question and Answering

Stanford Question Answering Dataset: For training, we have used the Stanford Question Answering Dataset (SQuAD) dataset which comprises 100,000+ Questions for Machine Comprehension of Text. It is a reading comprehension dataset consisting of questions posed by crowdworkers on a set of Wikipedia articles.

IV. Methodologies

A. Modeling

We have fine tuned the pre-trained BERT model and deployed it on Vertex AI for the emotion classification task. The BERT model is pre-trained on a large corpus of unlabeled text including the entire Wikipedia (2,500 million words) and Book Corpus (800 million words). The advantage of using BERT includes only having to slightly tune it for the emotion classification task. This also results in quicker development and uses a comparatively smaller dataset

B. MLOps

An ML Pipeline allows us to run and automate all the required steps of an ML system, starting from Data collection to model deployment and serving.

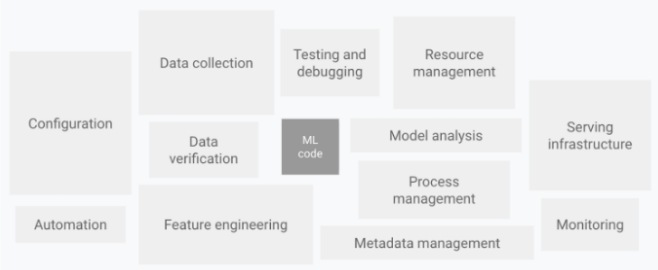


Fig 3: A typical MLOps architecture

MLOps is an ML engineering culture and practice that aims at unifying ML system development (Dev) and ML system operation (Ops). Practicing MLOps means that you advocate for automation and monitoring at all steps of ML system construction, including integration, testing, releasing, deployment and infrastructure management. We can implement and train an ML model with predictive performance on an offline holdout dataset, given relevant training data for their use case. However, the real challenge isn't building an ML model, the challenge is building an integrated ML system and to continuously operate it in production.

C. GCP and Vertex AI

We are custom training a model (pre-trained BERT), fine-tuning on various values of hyperparameters and deploying it for emotion classification on the Vertex AI pipeline.

Understanding Vertex AI:

Vertex AI is a unified MLOps platform to help data scientists/ML engineers increase experimentation, deploy faster, and manage models with confidence. It helps build, deploy, and scale ML models faster, with pre-trained and custom tooling within a unified AI platform.

It is an AI platform that helps in building a unified ML application by integrating Google cloud services. The key features of VertexAI includes:

- A unified UI for the entire ML workflow
- End-to-end integration for data and AI
- Pre-trained APIs for vision, video, natural language, and others
- Support for all open source frameworks

Emotion Classification using Vertex AI:

We have built an end-to-end training pipeline by fine-tuning the pre-trained bert-base-cased model for the emotion classification task on Vertex AI. The dataset we have used for this task is the Emotion Classification dataset.

Our objective for this task is to build, train, finetune and deploy a BERT based PyTorch model on Vertex AI and emphasize on support for training and deploying PyTorch models on Vertex AI.

We start by creating notebook instance on the GCP console by running the following command on the cloud console:

```
gcloud notebooks instances create cmpe297-project --vm-image-project=deeplearning-platform-release --vm-image-family=pytorch-1-9-cu110-notebooks --machine-type=n1-standard-4 --location=us-central1-a --boot-disk-size=100 --accelerator-core-count=1 --accelerator-type=NVIDIA_TESLA_V100 --install-gpu-driver --network=default
```

Fig 4: Command to create a workbench instance on GCP

Upon running this command, a notebook instance gets created in the workbench section of GCP's vertex AI. The result can be visualized as follows:

Fig 5: Result of gcloud command execution for notebook creation

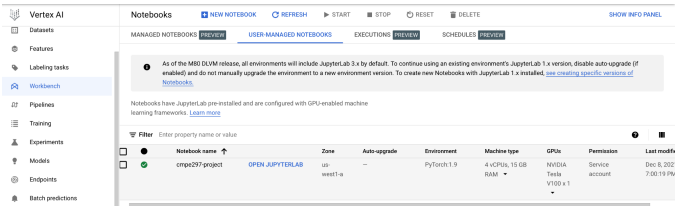


Fig 6: Successful creation of the workbench instance on console

In order to create this notebook instance, we have upgraded the required quotas from 0 to 1 since running our code requires GPU.

Fig 7: GPU resource allocation

Google Cloud notebooks cover all the necessary requirements to fulfill the training, fine tuning and deploying of our model for emotion classification. The requirements include:

- The Google Cloud SDK
- Git
- Python 3
- Virtualenv which is a tool to create an isolated Python environment.
- Jupyter notebook with Python 3
- Transformers
- Datasets
- Hypertune
- We have used 'aiplatform' library that simplifies ML workflows by utilizing wrapper classes and opinionated defaults.

The following APIs are required to be enabled as a requirement to run the project on GCP:

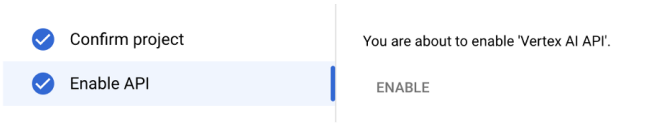


Fig 8: Enabling Vertex AI API

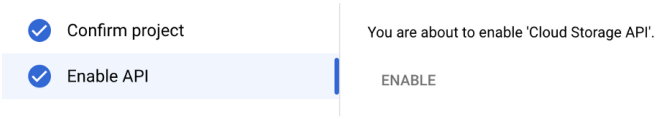


Fig 9: Enabling Cloud Storage API

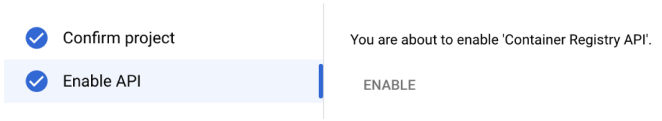


Fig 10: Enabling Container Registry API

Fig 11: Enabling Cloud Build API

Setting up the cloud storage bucket:

The cloud storage bucket helps in storing the model artifacts and hyperparameter tuning results. While creating and deploying the model on Vertex AI this is utilized. We created a cloud bucket by executing the following set of commands on the notebook instance.

```
print(f"PROJECT_ID = {PROJECT_ID}")
print(f"BUCKET_NAME = {BUCKET_NAME}")
print(f"REGION = {REGION}")
```

```
PROJECT_ID = starry-lens-333804
BUCKET_NAME = gs://starry-lens-333804aip-20211209030626
REGION = us-west1
```

```
! gsutil mb -l $REGION $BUCKET_NAME
```

```
Creating gs://starry-lens-333804aip-20211209030626/...
```

Fig 12: Created a cloud bucket

The resulting creation of the bucket can be visualized as follows:

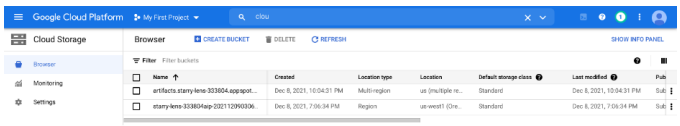


Fig 13: Cloud bucket instance on the GCP

Model Training

In this project, we first train the PyTorch emotion classification model locally using a pre-trained BERT model from Hugging Face Transformers and then on Vertex AI training service.

Training locally:

We first downloaded the emotion dataset using the Hugging Face Datasets library and then used the 'load_dataset' function to load the same.

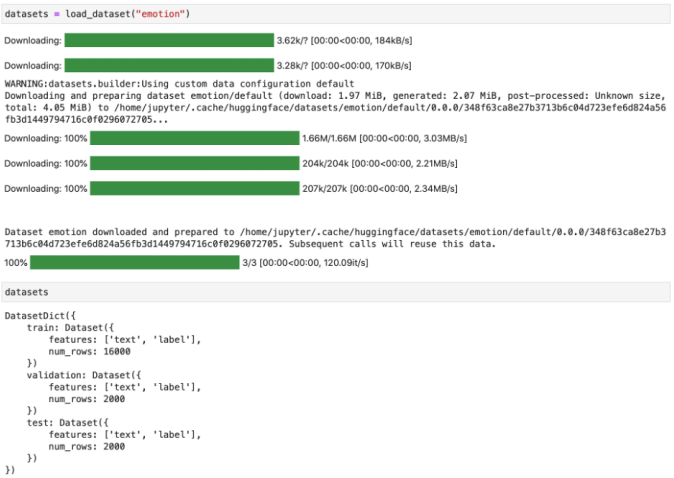


Fig 14: Loading the dataset on the notebook instance

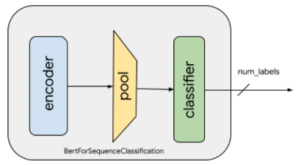
We used the 'Tokenizer' class from Hugging Face Transformers to tokenize and preprocess the input data to the format required by the model. The below screenshot represents a sample of the tokens created for input text:

Fig 15: Results of the tokenizer on input text

Model Fine-tuning

Transfer learning is a technique where a deep learning model trained on a large dataset is used to perform similar tasks on another dataset. This is done by freezing the middle layers while modifying the input and output layers that are suitable for our task.

In this study we are finetuning the pre-trained BERT model to suit our task of emotion classification.



Pretrained Model with classification layer: The blue-box indicates the pre-trained BERT Encoder module. Output of the encoder is pooled into linear layer with number of outputs same as the number of target labels (classes).

Fig 16: Fine-tuning BERT

We have replaced the head used to pre-train the model on masked language modeling tasks with a new head for emotion classification. This will not have pre-trained weights initially and gets assigned when we finetune the model.

We create a trainer object with all the required parameters including learning rate, number of epochs, weight decay and batch size for training. The below screenshots show the same:

```
args = TrainingArguments(
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=10,
    weight_decay=0.01,
    output_dir="/tmp/cls",
)
```

Fig 17: Creating a trainer object to fine-tune BERT

```
history_train = trainer.train()
```

```
The following columns in the training set  don't have a corresponding argument in `BertForSequenceClassification.forward`
and have been ignored: text.
***** Running training *****
  Num examples = 16000
  Num Epochs = 10
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 10000
```

[10000/10000 23:27, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy
1	0.176300	0.273056	0.925000
2	0.133700	0.206070	0.921000
3	0.103600	0.247529	0.926000
4	0.079000	0.321118	0.924000
5	0.052000	0.421277	0.924500
6	0.039600	0.410756	0.926500
7	0.021200	0.451202	0.921500
8	0.020100	0.447655	0.926000
9	0.013600	0.476519	0.920000
10	0.008700	0.485455	0.921000

Fig 18: Results of fine-tuning BERT for 10 epochs

We have also recorded TensorBoard logs for model training. We noticed that fine tuning the BERT model for only one epoch suffices as it achieves good accuracy. We trained for 10 epochs so as to visualize the logs on TensorBoard as part of our study. Below screenshot shows the same and how the accuracy diminishes as the epochs increase:

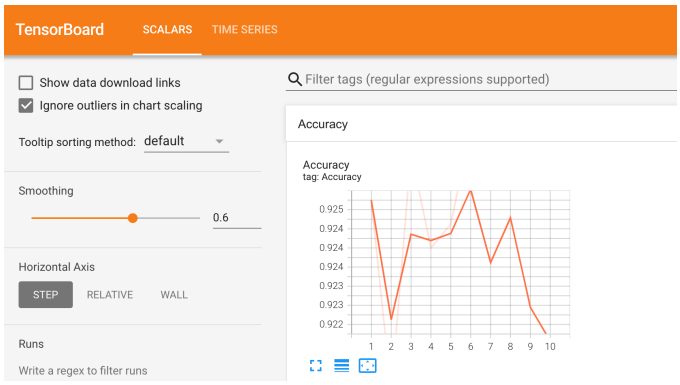


Fig 19: TensorBoard logs for tracking Accuracy

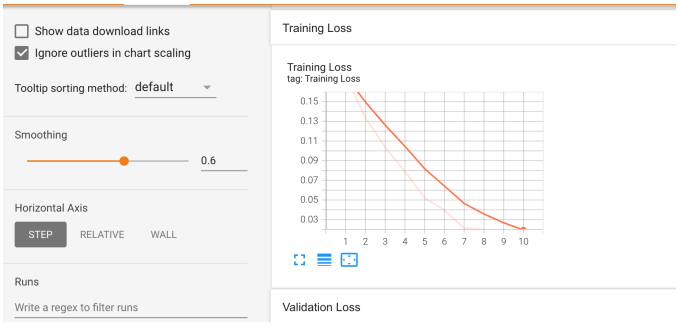


Fig 20: TensorBoard logs for tracking the training loss

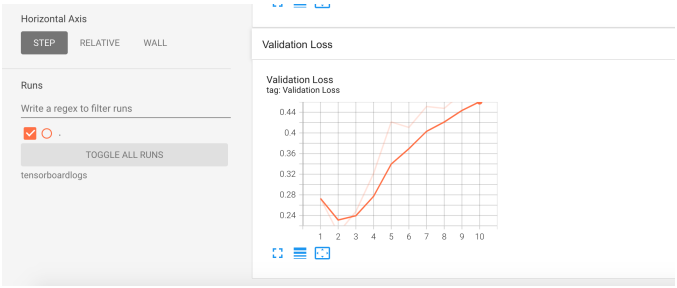


Fig 21: TensorBoard logs for tracking the validation loss

We can also see the model artifacts getting saved in the local folders upon training:

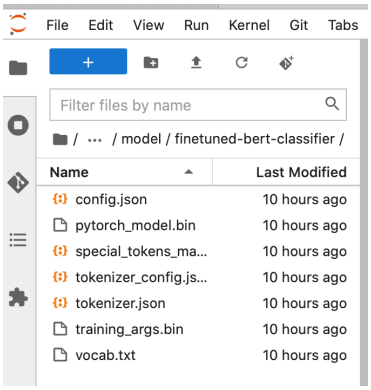


Fig 22: Model artifacts locally saved

Running predictions locally

At this point we can predict the emotion labels for input text after preprocessing the input.

```
# example #1
review_text = (
    "im feeling quite sad and sorry for myself but ill snap out of it soon"
)
predict_input = predict(review_text, saved_model_path)
```

```
loading configuration file https://huggingface.co/bert-base-cased/resolve/main/config.json from cache at /home/jupyter/.c
ache/huggingface/transformers/a803e0468a8fe090683bdc453f4fac622804f49de86d7cecaee92365d4a0f829.a64a22196690e0e82ead56f388
a3ef3a50de93335926ccfa20610217db589307
Model config BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.12.5",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 28996
}
```

Fig 23: Local predictions the fine-tuned BERT - 1

```
loading weights file ./models/pytorch_model.bin
All model checkpoint weights were used when initializing BertForSequenceClassification.

All the weights of BertForSequenceClassification were initialized from the model checkpoint at ./models.
If your task is similar to the task the model of the checkpoint was trained on, you can already use BertForSequenceClassi
fication for predictions without further training.
Review text: im feeling quite sad and sorry for myself but ill snap out of it soon
Sentiment : Sadness
```

Fig 24: Local predictions the fine-tuned BERT - 2

Training on Vertex AI

For larger datasets and models as in our case, building a training pipeline by leveraging Vertex AI is the most effective method. The training job on Vertex AI is carried out by packaging the code and creating a training pipeline to orchestrate a training job. The 3 steps we have followed are:

- Packaging the training code as a Python source distribution
- Hyperparameter training job
- Finally deploying the best model to an endpoint on Vertex AI.

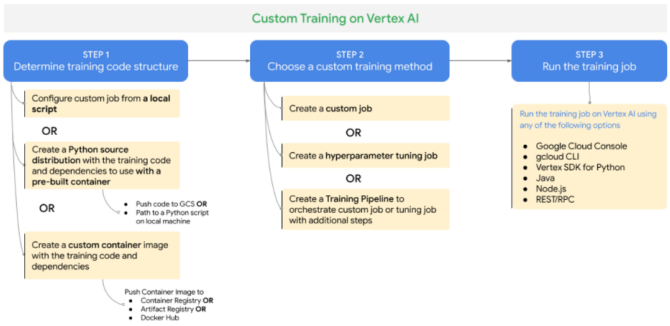


Fig 23: Custom training on Vertex AI

Step 1: Packaging the training application

The first step involves packaging the training application code with all dependencies and uploading it to the Cloud Storage bucket or Container Registry (dependencies include transformers, datasets and tqdm – which are added in the setup.py file). We achieve this by creating a Python source with pre-built containers on Vertex AI.

We now run the custom training job on Vertex AI using Vertex SDK to create and submit the training job. The below screenshot shows the submission of the job from the notebook instance and also the training job successful on GCP's Vertex AI platform:

```
training_args = ["--num-epochs", "2", "--model-name", "finetuned-bert-classifier"]

model = job.run(
    replica_count=1,
    machine_type="n1-standard-8",
    accelerator_type="NVIDIA_TESLA_V100",
    accelerator_count=1,
    args=training_args,
    sync=False,
)
```

```
INFO:google.cloud.aiplatform.training_jobs:Training Output directory:
gs://starry-lens-333804aip-20211209030626/aiplatform-custom-training-2021-12-09-05:15:42.974
INFO:google.cloud.aiplatform.training_jobs:View Training:
https://console.cloud.google.com/ai/platform/locations/us-central1/training/8078420892037677056?project=425328907110
INFO:google.cloud.aiplatform.training_jobs:CustomPythonPackageTrainingJob projects/425328907110/locations/us-central1/tra
iningPipelines/8078420892037677056 current state:
PipelineState.PIPELINE_STATE_PENDING
INFO:google.cloud.aiplatform.training_jobs:CustomPythonPackageTrainingJob projects/425328907110/locations/us-central1/tra
iningPipelines/8078420892037677056 current state:
PipelineState.PIPELINE_STATE_PENDING
INFO:google.cloud.aiplatform.training_jobs:CustomPythonPackageTrainingJob projects/425328907110/locations/us-central1/tra
iningPipelines/8078420892037677056 current state:
PipelineState.PIPELINE_STATE_PENDING
INFO:google.cloud.aiplatform.training_jobs:View backing custom job:
https://console.cloud.google.com/ai/platform/locations/us-central1/training/2502753447120470016?project=425328907110
```

Fig 24: Submitting a training job to Vertex AI from notebook

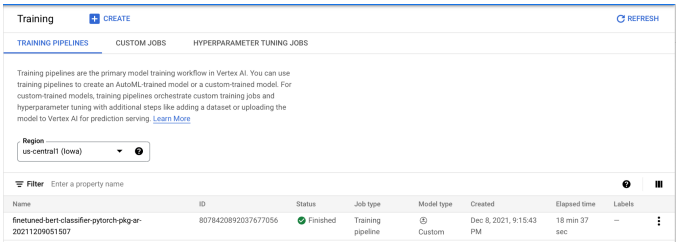


Fig 25: Training job submitted and completed

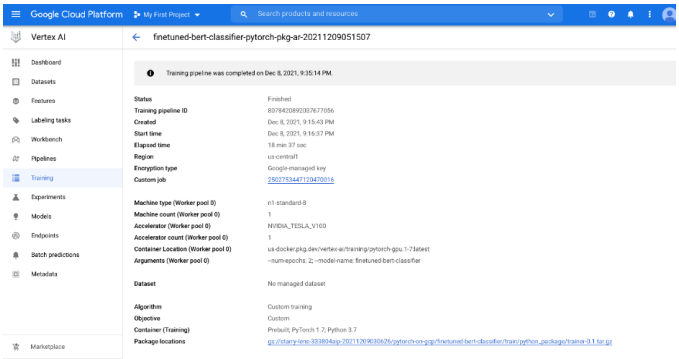


Fig 26: Training job description

We can see that the model artifacts are successfully saved in the bucket after training:

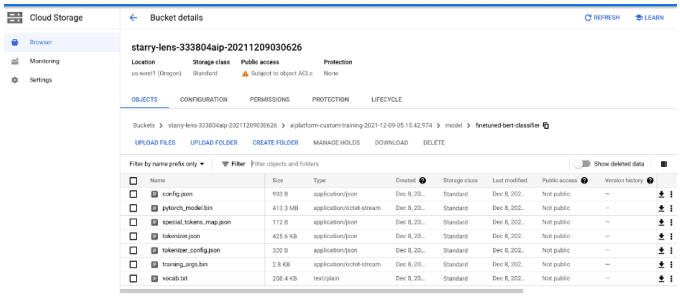


Fig 27: GCP bucket contents after training

Hyperparameter Tuning

We are experimenting with hyperparameters such as learning rate and weight decay while fine tuning the BERT model. These hyperparameter values are directly proportional to the behavior of the training algorithm and can have a significant impact on the performance of the model. We have automated the hyperparameter tuning with the Vertex Training service by packaging the training code and all dependencies in a Docker container. We then pushed the container to Google Container Registry.

Following are the steps involved in running a hyperparameter tuning job on Vertex AI service:

1. We run multiple trials of the training application with the hyperparameters values as specified (learning rate and weight decay).
2. Vertex AI keeps track of the results from each trial run. Our training application reports the metrics to Vertex AI using the Python package cloudml-hypertune.
3. Once the job is completed, we get the summary of all the trial runs with the best performing configurations amongst all the criteria we have specified as a Pandas data frame.

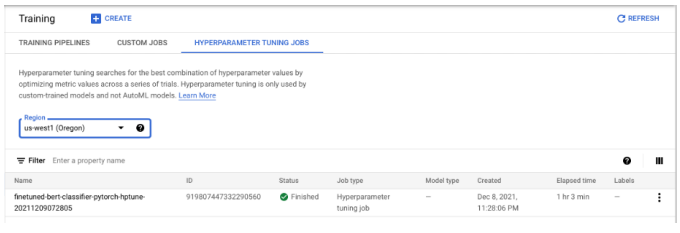


Fig 28: Instance for hyperparameter tuning

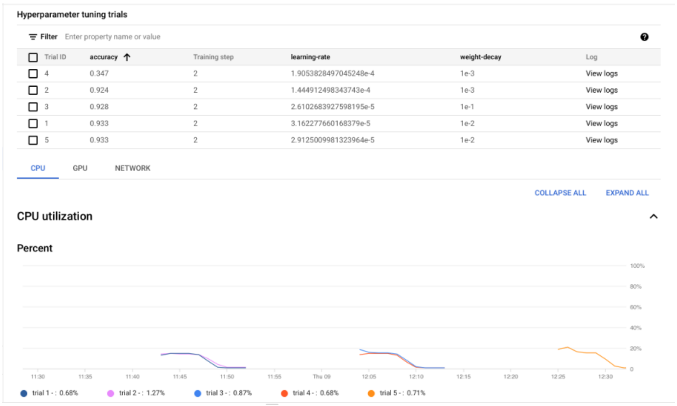


Fig 29: Hyperparameter tuning summary

We can visualize the logs for each of these training runs on Vertex AI. Below screenshot shows the training logs for Trial ID 2:

Fig 30: Hyperparameter tuning logs for one trial - 1

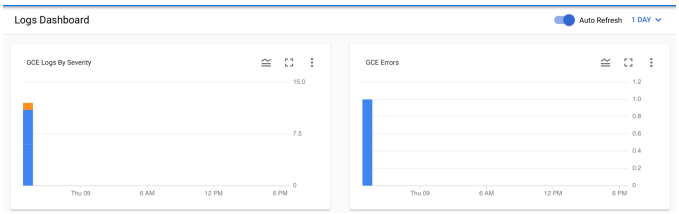


Fig 31: Hyperparameter tuning logs for one trial - 2

Deployment on VertexAI

Deploying our fine tuned BERT model on VertexAI uses a Docker container image that runs an HTTP server. Below are the steps involved in deploying the model for vertex predictions:

We first packaged all the required model artifacts and built a customer container compatible with Vertex Predictions to serve the model. At this point, we checked the container's health by running the ping command. On success, we send a prediction request to the model on the container's server to make sure the predictions are being served.

We then pushed this model with a custom container image to serve predictions as a model resource on Vertex.

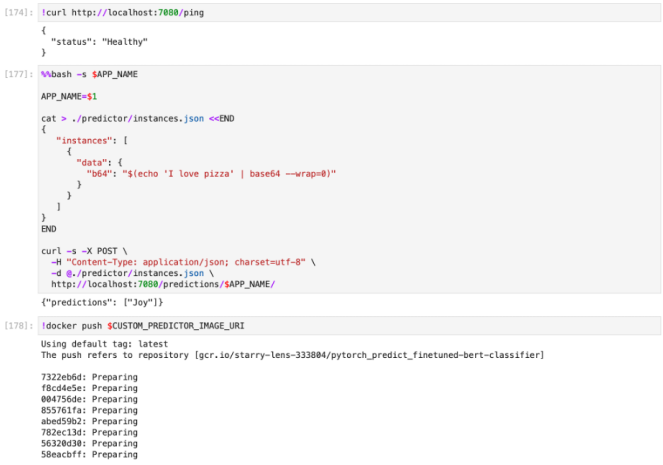


Fig 32: Pushing the best performing model after hyperparameter tuning

We also created a Vertex endpoint and deployed our model resource to serve emotion predictions based on the input text.
From the below screenshot, we can see that the model is deployed successfully to the created endpoint:

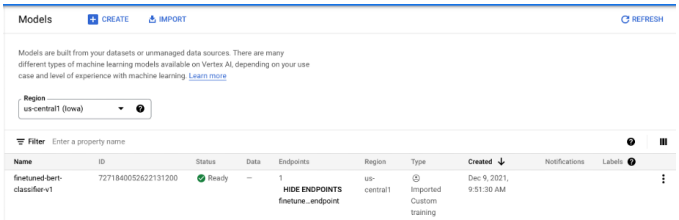


Fig 33: Model deployed successfully

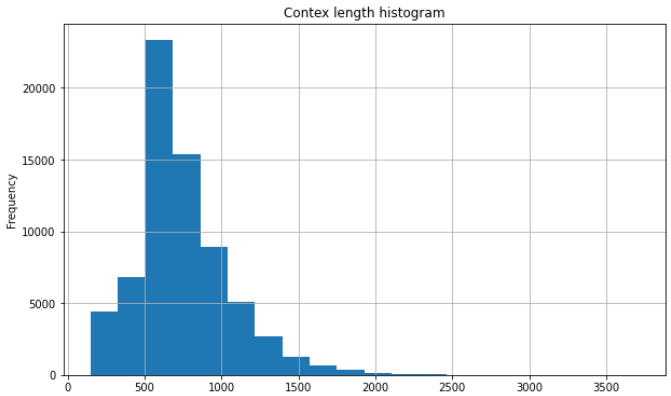
V. EXPERIMENTS

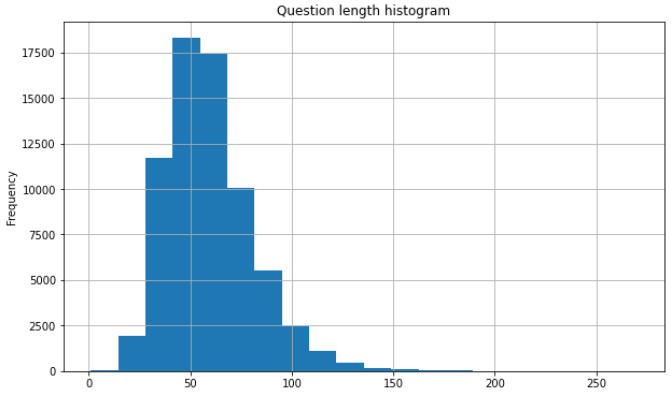
A. *Question Answering*

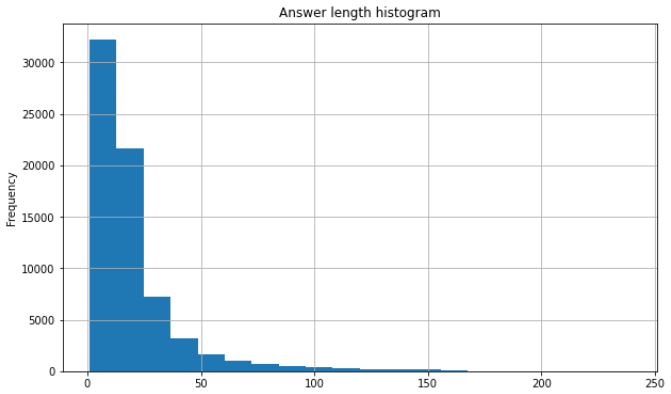
Question answering is a language modeling task wherein the model is given a short paragraph or context about some topic and is asked some questions based on the passage. For training, we have used the SQuAD dataset which comprises 100,000+ Questions for Machine Comprehension of Text.

The main model we have used for this task is Transformers. Specifically, we use the DistilBERT model, pretrained on Masked LM and Next Sentence Prediction, add a new head for question answering, and train the new model for our task.

Some data distribution visualizations:







An example:

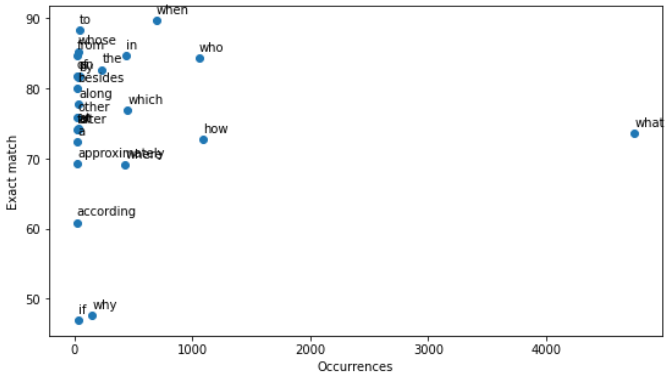
TITLE:
PlayStation 3

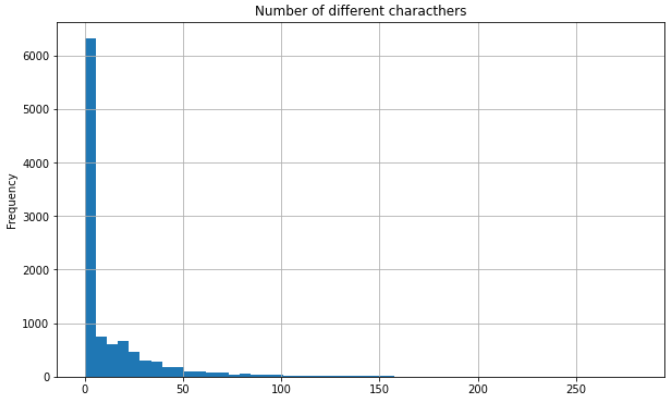
CONTEXT:
The console was first officially announced at E3 2005, and was released at the end of 2006. It was the first console to use Blu-ray Disc as its primary storage medium. The console was the first PlayStation to integrate social gaming services, included it being the first to introduce Sony's social gaming service, PlayStation Network, and its remote connectivity with PlayStation Portable and PlayStation Vita, being able to remote control the console from the devices. In September 2009, the Slim model of the PlayStation 3 was released, being lighter and thinner than the original version, which notably featured a redesigned logo and marketing design, as well as a minor start-up change in software. A Super Slim variation was then released in late 2012, further refining and redesigning the console. As of March 2016, PlayStation 3 has sold 85 million units worldwide. Its successor, the PlayStation 4, was released later in November 2013.

QUESTION:	ANSWER:
What year was the PlayStation 3 released?	2006
What social gaming service was integrated into the PlayStation 3?	PlayStation Network
What was the thinner version of the PS3 called?	Slim
What year did the Super Slim model hit stores?	2012
How many PlayStation 3 units had been purchased as of early 2016?	85 million

Below is how our BERT model looks like:

```
==== Embedding Layer ====

distilbert.embeddings.word_embeddings.weight            (30522, 768)
distilbert.embeddings.position_embeddings.weight          (512, 768)
distilbert.embeddings.LayerNorm.weight                      (768,)
distilbert.embeddings.LayerNorm.bias                        (768,)
distilbert.transformer.layer.0.attention.q_lin.weight     (768, 768)

==== First Transformer ====

distilbert.transformer.layer.0.attention.q_lin.bias         (768,)
distilbert.transformer.layer.0.attention.k_lin.weight     (768, 768)
distilbert.transformer.layer.0.attention.k_lin.bias         (768,)
distilbert.transformer.layer.0.attention.v_lin.weight     (768, 768)
distilbert.transformer.layer.0.attention.v_lin.bias         (768,)
distilbert.transformer.layer.0.attention.out_lin.weight   (768, 768)
distilbert.transformer.layer.0.attention.out_lin.bias       (768,)
distilbert.transformer.layer.0.sa_layer_norm.weight         (768,)
distilbert.transformer.layer.0.sa_layer_norm.bias           (768,)
distilbert.transformer.layer.0.ffn.lin1.weight           (3072, 768)
distilbert.transformer.layer.0.ffn.lin1.bias                (3072,)
distilbert.transformer.layer.0.ffn.lin2.weight           (768, 3072)
distilbert.transformer.layer.0.ffn.lin2.bias                (768,)
distilbert.transformer.layer.0.output_layer_norm.weight     (768,)
distilbert.transformer.layer.0.output_layer_norm.bias       (768,)
distilbert.transformer.layer.1.attention.q_lin.weight     (768, 768)

==== Output Layers ====

distilbert.transformer.layer.5.output_layer_norm.weight     (768,)
distilbert.transformer.layer.5.output_layer_norm.bias       (768,)
qa_outputs_0.weight                                       (512, 768)
qa_outputs_0.bias                                           (512,)
qa_outputs_1.weight                                      (32, 512)
qa_outputs_1.bias                                            (32,)
qa_outputs.weight                                         (2, 32)
qa_outputs.bias                                              (2,)
LayerNorm.weight                                          (384, 2)
LayerNorm.bias                                            (384, 2)
```





Training:

```
***** Running training *****
  Num examples = 70126
  Num Epochs = 3
  Instantaneous batch size per device = 32
  Total train batch size (w. parallel, distributed & accumulation) = 3
  Gradient Accumulation steps = 1
  Total optimization steps = 6576
```

[6576/6576 4:25:57, Epoch 3/3]

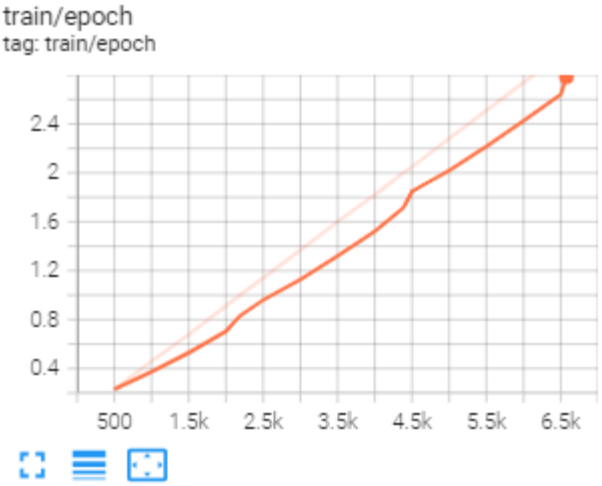
Epoch	Training Loss	Validation Loss	Exact Match	F1
1	1.278800	1.309120	58.499478	74.004360
2	0.890900	1.272091	59.597957	75.436259
3	0.603500	1.356313	60.213105	75.850542

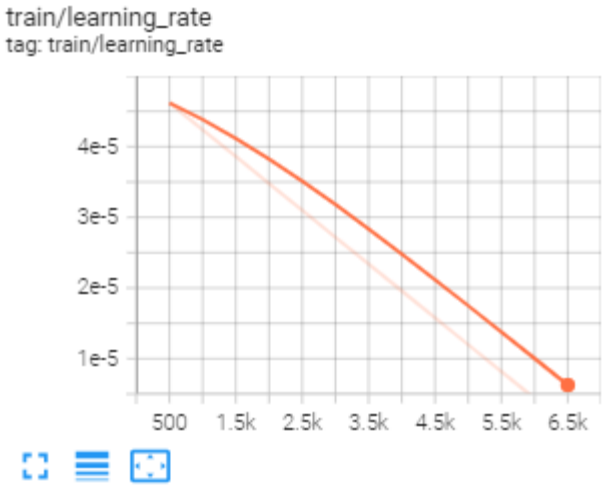
Evaluation:

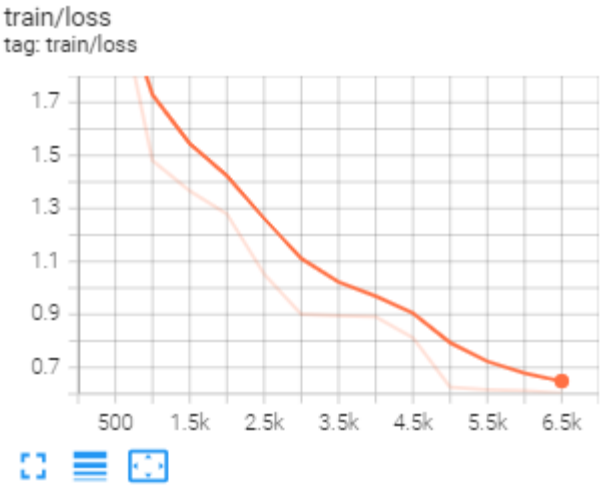
```
***** Running Prediction *****
  Num examples = 10784
  Batch size = 8
```

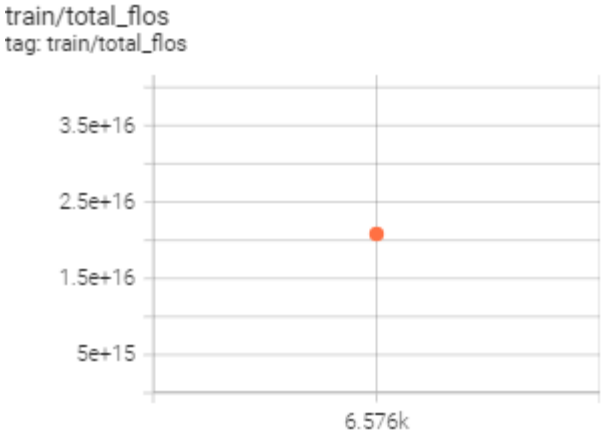
[1348/1348 1:52:00]

Tensorboard logs:





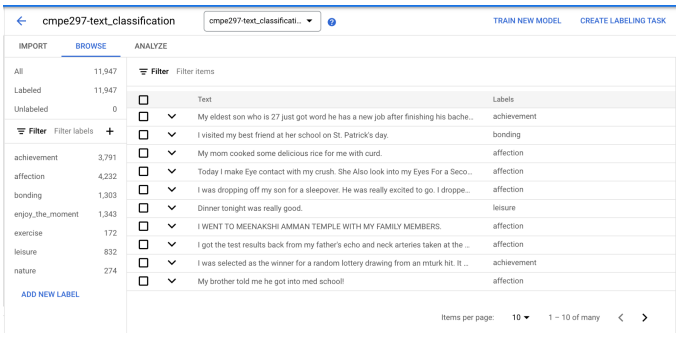




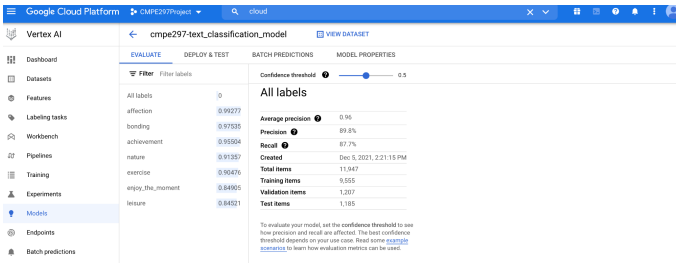
B. Text Classification using AutoML on Vertex AI

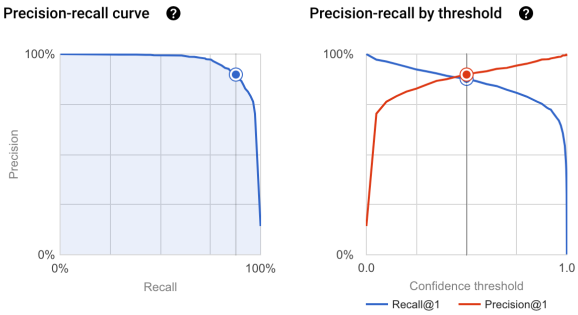
We experimented with creating an AutoML model for text classification using Vertex AI as a starting point in our study. We trained an AutoML model on the corpus of crowd-sourced "happy moments" from the Kaggle open-source dataset called HappyDB.

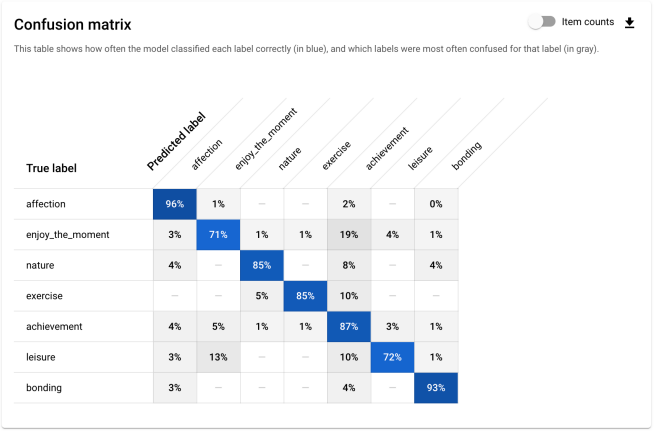
We created a bucket which was used to store all the training data from the HappyDB dataset. Below screenshot shows the dataset created on Vertex AI.



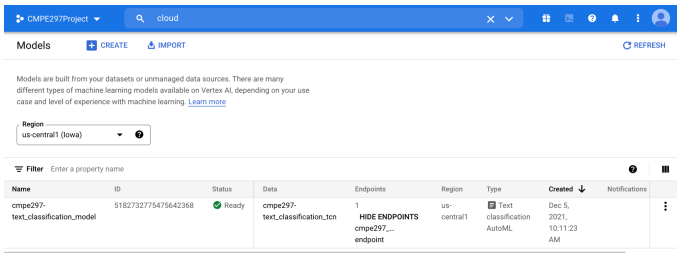
We then trained an AutoML model for the dataset using the services provided by the Vertex AI itself. Below are some of the model training metrics that were generated upon training this model.



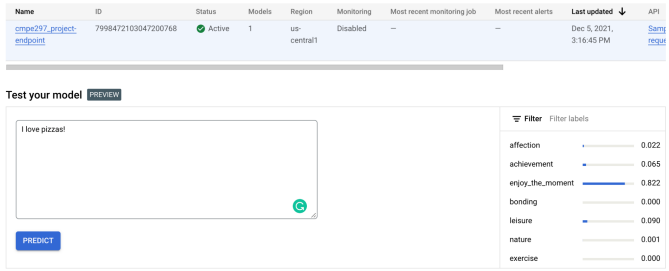




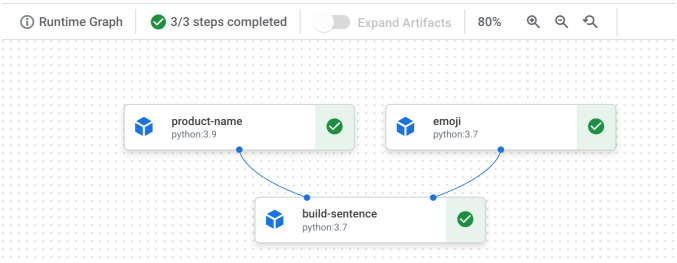
Once the model training job was completed, we created an endpoint and deployed it so it could serve predictions.



We also send prediction requests to our model with input text to check for the performance. The below screenshot shows the predictions are successful.



Below is a screenshot for visualization of the entire pipeline for the text classification problem on Vertex AI.



C. *Text Summarization*

Summarization is a language modeling task which shortens or surmises a text document by keeping only the main information and helps in avoiding information overload.

To implement summarization we have used the Pegasus model from HuggingFace which is specialized for summarizing financial statements. It was trained on Bloomberg market and financial news.

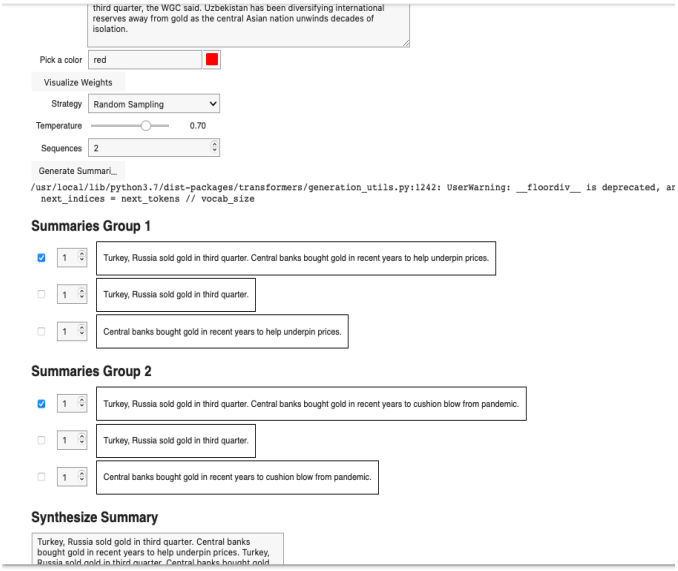
Assuming that a word which is attended by many words is more important for summarizing, we highlight the words according to their self-attention weights, i.e., high-weight words are strongly highlighted, while lower-weight words are faintly highlighted.

We have provided different decoding strategies that the user can choose from to generate summaries and a control to tune different hyperparameters (depending on the decoding strategy selected). The model generates a number of alternative summaries

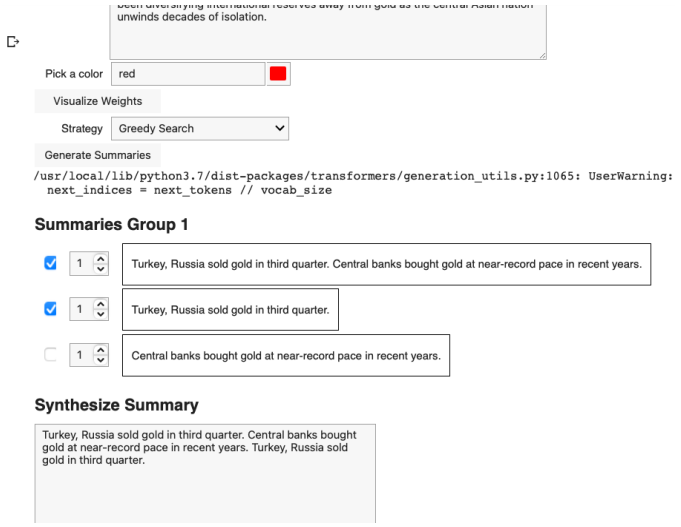
and we can choose which summaries to combine to generate a wholesome summary.

The different decoding strategies that can be selected are *Greedy Search, Beam Search, Diverse Beam Search, Random sampling and Top k sampling.*

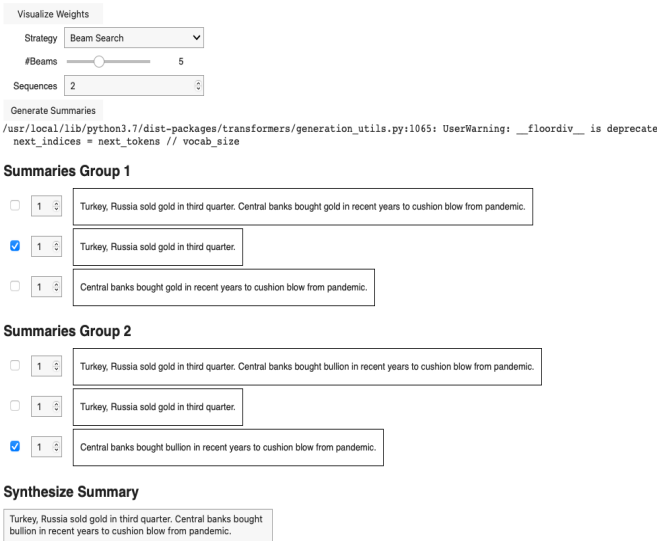
Random sampling selects a token out of the word probability distribution randomly. We can select the temperature or the "randomness" and the number of sequences to generate.



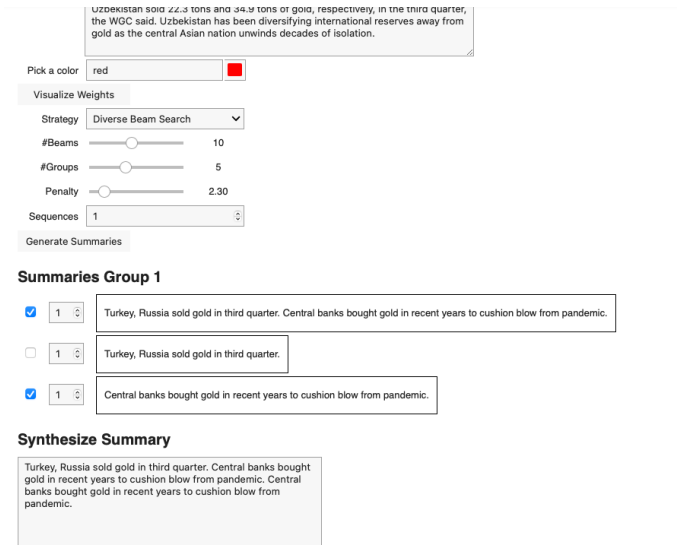
Greedy search selects the token with the highest probability at each time step.



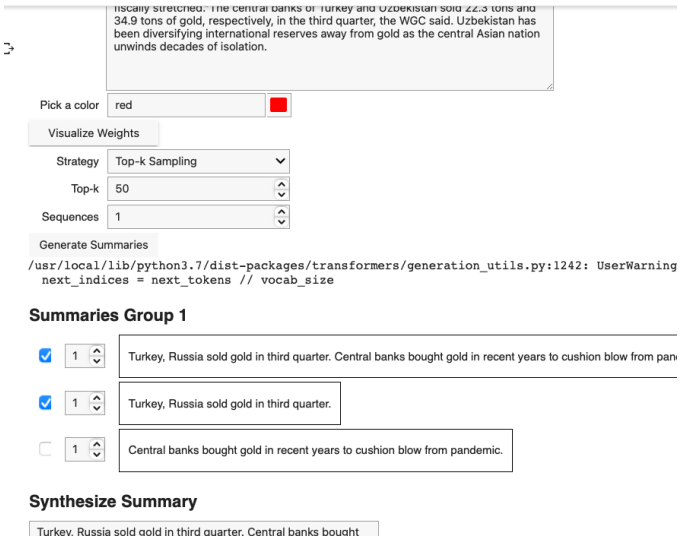
Beam search selects not only the token with the highest probability at each time step, but also a number of tokens with the highest probability according to the beam width. The number of the final generated beams is equal to the beam width.



Diverse beam search follows the beam search algorithm, but also adds a diversity penalty to enhance the diversity between the top most probable generated beams.



Top-k sampling limits the space of possible next tokens to the top-k higher-ranked tokens of the distribution.

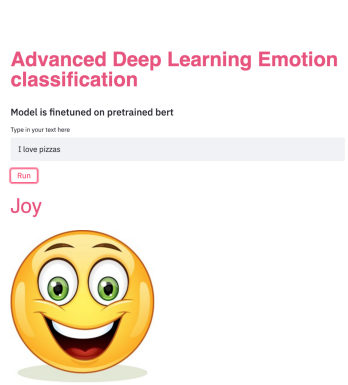


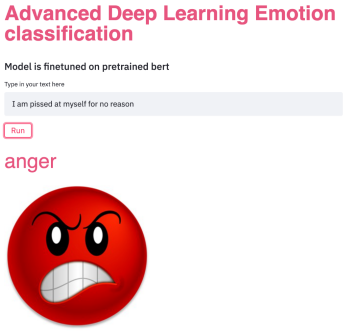
VI. Evaluation and Results

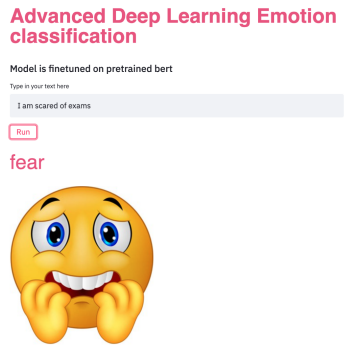
We have built a Streamlit application to show our model predictions on the User Interface. After deploying the model to the endpoint on Vertex AI, we first authenticated the application to invoke the endpoint by creating a service account key on the cloud console.

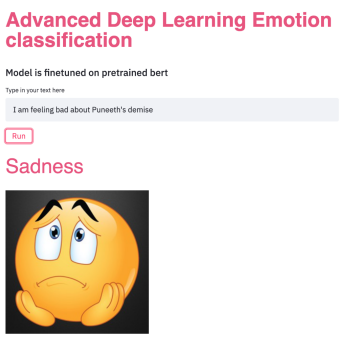
We then serve the fine tuned BERT model predictions on the Streamlit application by using Google Cloud's 'aiplatform' library.

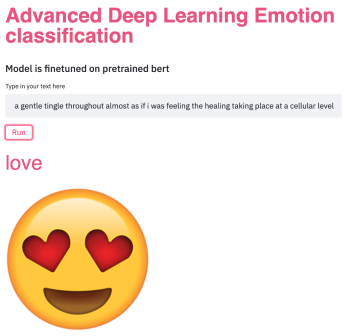
Below are the screenshots of our model serving predictions for each of the class labels in the emotion dataset (sadness (0), joy (1), love (2), anger (3), fear (4)).











We noticed that the predictions for 'Joy' and 'Love' have some overlap and the majority of these predictions are inclined towards 'Joy'.

For the task of Question and Answering, we achieved the following results:

```
Top 1 exact match and F1 score: {'exact_match': 75.84673604541155, 'f1': 84.56513415
Top 5 exact match: 0.8251655629139073
```

VII. Challenges

One of the main challenges we faced was to create a workbench instance with GPUs in the region us-central1. Since the region did not have sufficient resources available, we had to create the instance in us-west1. This issue led to the creation of one multi-region bucket which inhibited creation of the training pipeline.

Another challenge involved sanitizing the input for model predictions from the Streamlit application. The input string had to be converted into a byte array and had to be further encoded using the base64 encoding.

VIII. Conclusion

Given that our dataset (Huggingface's Emotion dataset) and model (pre-trained BERT) for the task of emotion classification was huge, building a training pipeline by leveraging Vertex AI was the most effective method because of its compute power and distributed server architecture. Vertex AI allowed us to create a reproducible pipeline and tune the hyperparameters to find the best model. We successfully custom trained our BERT model on Vertex AI and deployed it to the endpoint. We also retrieved the predictions efficiently by invoking the model endpoint from the Streamlit application.

We also performed language modeling tasks like Text Summarization and Question Answering using pre-trained BERT models on different datasets and achieved significant model performances. Using pre-trained models help us in achieving quicker development, work with lesser data and obtain better results.

Acknowledgment

We would like to thank our professor, Vijay Eranti for aiding and guiding us throughout our project.

References

[1] https://huggingface.co/datasets/emotion

[2] Li, Q., Peng, H., Li, J., Xia, C., Yang, R., Sun, L., ... & He, L. (2020). A survey on text classification: From shallow to deep learning. arXiv preprint arXiv:2008.00364.

[3] https://github.com/GoogleCloudPlatform/vertex-ai-samples/blob/main/community-content/pytorch_text_classification_using_vertex_sdk_and_gcloud/pytorch-text-classification-vertex-ai-train-tune-deploy.ipynb

[4] https://github.com/nlpunibo/Question-Answering-SQUAD

[5] https://huggingface.co/transformers/

[6] https://cloud.google.com/vertex-ai/docs/tutorials/image-recognition-automl

[7] https://www.kaggle.com/ritresearch/happydb

[8] SQuAD dataset: https://arxiv.org/abs/1606.05250

[9] Pegasus model: https://huggingface.co/human-centered-summarization/financial-summarization-pegasus

[10] Takumi Takahashi, Motoki Taniguchi, Tomoki Taniguchi, and Tomoko Ohkuma. 2019. CLER: Cross-task learning with expert representation to generalize reading and understanding. In Proceedings of the 2nd Workshop on Machine Reading for Question Answering, pages 183–190, Hong Kong, China. Association for Computational Linguistics.

[11] Alon Talmor and Jonathan Berant. 2019. MultiQA: An empirical investigation of generalization and transfer in reading comprehension. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 4911–4921, Florence, Italy. Association for Computational Linguistics