**ENTITY MATCHING EXPERIMENT**

**Task 1**

**File:** . / task1 / entity\_matching\_siamese\_network.ipynb

1. Data description

The data provided is in csv format and contains two features and a label. The unnamed first column has been analysed and it contains only unique values with certain values missing from the sequence thereby establishing the following assumption.

**Assumption:** The unnamed first column is the index column where the missing values from the sequence indicate a data cleaning task performed prior to handing over the dataset

**Note:** The above assumption does not rule out the further need for data cleaning and preprocessing. It is only to take the unnamed column as index for the dataset.

The following are the features of the dataset:

1. "entity\_1" : This contains company names with extra spaces and string length as low as 1
2. "entity\_2" : This contains company names with extra spaces and string length as low as 1

The label column in the dataset is named "tag" and it is a binary integer data with values 1 and 0

The data set is imbalanced with the negative class accounting to 60% of the examples

and positive class accounting to 40% of the example

There are no missing values in any of the features

1. Data Cleaning Process

The data cleaning process covers the following tasks

* Removing Punctuations
* Lower casing the strings
* Stripping extra spaces
* Deduplication of records
* Removing records where both the features have length less than or equal to 3 (This does not provide any sementic meaning for company names thereby leading to learning unwanted patterns for the model and hence need to be removed. The quantity of such data is considerably low and can be ignored if more robust models like S-GCN architecture are to be constructed)

even after the data cleaning there are no missing values or empty strings

1. Balancing the Dataset

The dataset can be easily balanced by using imblearn.over\_sampling.SMOTE

The balancing strategy according to the commented logic below is as follows:

* We use SMOTE to over sample the dataset thereby increasing its size
* Further we use RandomUnderSampler to under sample the dataset in order to bring it back to approximately its original size

Performing over sampling is compute expensive for local runs. Hence Right now we shall select and balance the dataset manual for trial runs This introduces a selection bias in the dataset But nevertheless, we can perform SMOTE or SMOTE-ENN or SMOTE with RandomUnderSampler to bring back the dataset size approx. to the original.

This can be done with distributed training in Apache beam. A sample pipeline for which has been developed along with this Submission.

Caution: The Apache Beam pipeline is highly error prone and requires debugging

1. Model Selection

There are various ways in which this task can be handled as described below:

* Siamese Networks
* Attention mechanisms
* Transformers
* Pre-trained models like BERT for generating embeddings and vector space

The above list is not exhaustive and hence one can take advantage of various new research like the one suggested in the paper titled "Business Entity Matching with Siamese Graph Convolutional Networks" written by employees and trainees at IBM Zurich. The link for which is given below

https://arxiv.org/abs/2105.03701

This architecture uses BERT + GCN and according to the paper it generates better results in terms of robustness to semantic meanings and title endings etc.

The model selected for this task is a Siamese Network. The Siamese network is an architecture that is highly effective in tasks where similarity is to be measured between pairs of input data.

The key idea is to have two identical subnetworks that share same parameters, weights and biases a.k.a the siamese twins

This architecture when developed objectivises robustness to variation while learning meaningful representations to the inputs.

The distance metric used here is a L1 norm since it is more robust

The model architecture presented here consist of an embedding layer, followed by shared LSTM units followed by L1 distance.

 This is followed by fully connected layers and dropouts for regularization thereby avoiding overfitting.

LSTM is chosen for the shared architecture as it is highly efficient in capturing long term dependencies and relationships in data such as company names

The order of words is quite crucial in company names and this is effectively captured by an LSTM unit

Embedding layers capture semantic meanings between words by representing them in continuous vector space. This can be helpful to capture the subtleties in company names

The final prediction involves usage of a sigmoid since we are performing binary classification

Loss is calculated using binary cross entropy as done for various binary classification problems

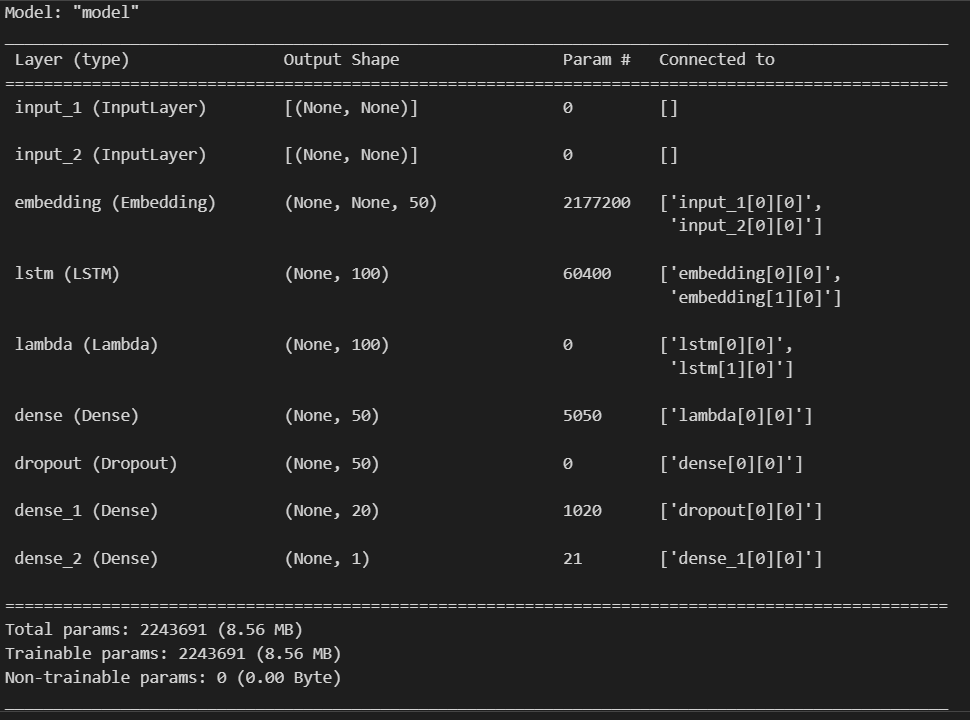
Optimizer used: Adam

No Hyperparameter tuning has been performed as it is not required for the given dataset and problem but if the dataset increases in size or while putting the model to production this may be completely necessary.

* **Downsides**

There are a few downsides to this architecture as it is complex and requires heavy computational resources. It requires large amount of data and tuning it for hyperparameter becomes crucial despite the computation expense. It can be difficult to interpret the internal working of model related to how it makes decisions

The following is the Model Summary:



Further tensorboard callbacks can be used to visualize model architecture and flow as well as various metricies.

The model artifacts namely weights and tokenizer has been saved in the task 2 folder

**Apache Beam Data Cleaning Pipeline:**

File: . / task1 / apache\_beam\_data\_preprocessing\_pipeline / apache\_beam\_data\_cleaning.py

This is a starter code for data preprocessing that can be refactored as per the need

Apache beam supports runners from various data sources including Apache Spark, Apache Flink, GCP BigQuery, GCP DataFlow etc.

And to use any of these we do not have to change the pipeline. We just need to configure the PipelineOptions for the required data source and Apache Beam will handle the rest.

Another reason is the Apache Beam’s support for distributed processing which the need of the hour in the Big data age.

**Task 2**

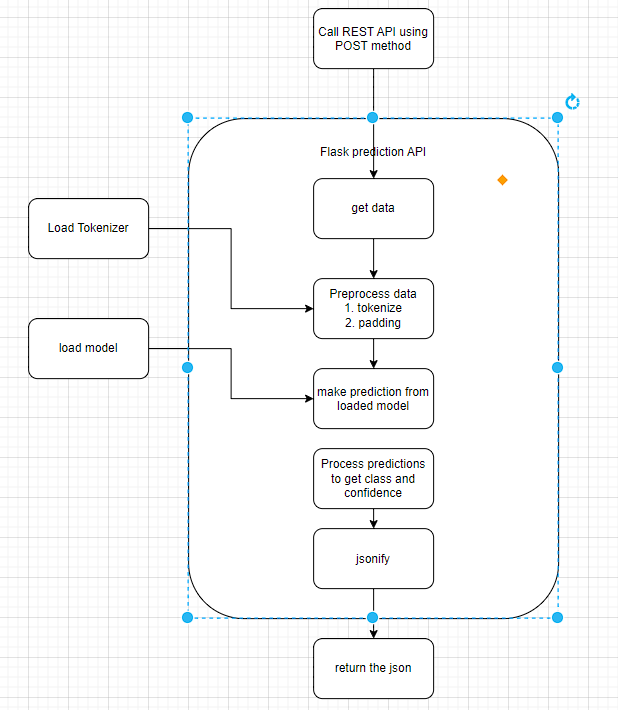
**2.1 Deployment**

Deployment of the above model is showcased in two ways:

1. An executable flask-based REST API script with DockerFile
2. An unexperimented code of tfx pipeline with orchestration code for Kubeflow

Lets briefly discuss the first approach.

The workflow of this deployment script is as below:



The following is the directory structure of the script:

/

└── app/

├── artifacts/

│ ├── siamese\_model\_saved\_artifacts/

│ │ ├── assets/

│ │ ├── variables/

│ │ ├── fingerprint.pb

│ │ ├── keras\_metadata.pb

│ │ └── saved\_model.pb

│ ├── siamese\_LSTM\_embeddings.h5

│ └── tokenizer.pickle

├── tests/

├── DockerFile

├── main.py

└── requirements.txt

The artifacts folder contains the saved model inside the folder “Siamese\_model\_saved\_artifacts”

This is the folder used by load\_model function to load the entire model that we trained earlier

The weights can be loaded using model.load\_weights() if we have a model compiled and ready but that’s not required here since the save() function saves the whole model with optimizers and weights etc.

The serving of model using the above process is fairly simple as can be seen from the workflow diagram.

* 1. **System Design**

When it comes to ML system design a lot of things need to taken into consideration other then just the DevOps or Infrastructure part. Data versioning, Model versioning, Maintaining metadata, Data snapshots, model monitoring, automated retraining and of course scalability.

The preferred or GO-TO solution for this is a tensorflow\_extended pipeline. In the folder task 2 / 2.2 Pipeline I have developed a code by taking references from the book called “Building Machine Learning Pipeline” publisher O’Reilly.

The components of this pipeline can be discussed at depth but it is beyond the scope of this document.

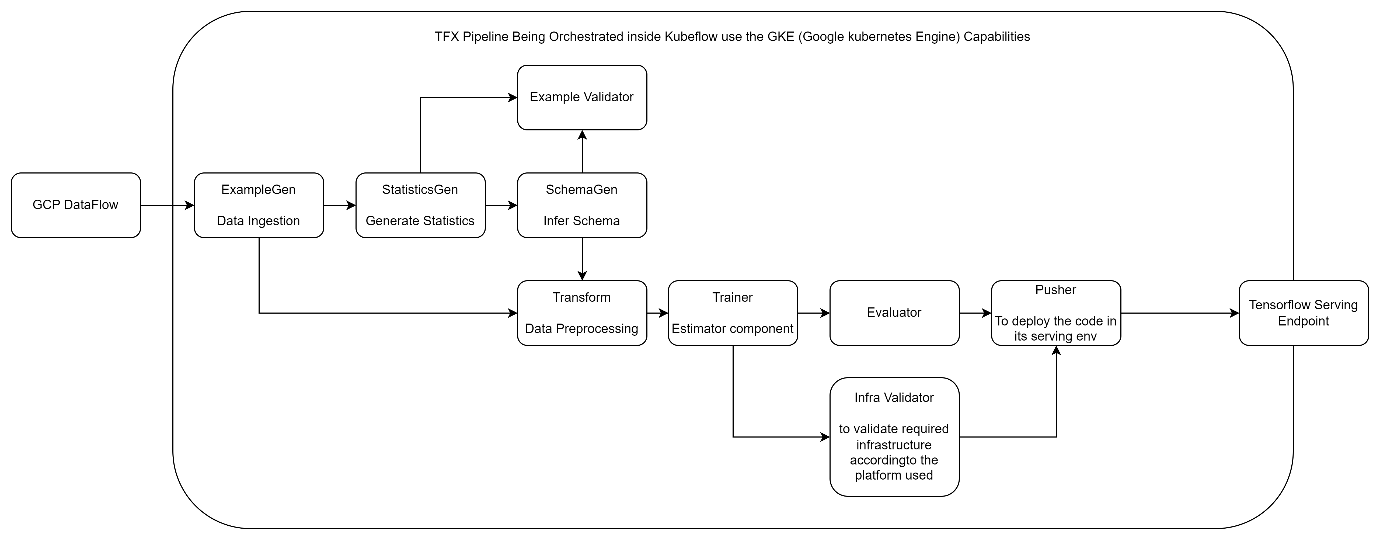
Keeping in mind the previously mentioned requirements for deployment of ML model in production and my others including automating ML pipelines so that data scientists can focus more on experimentation. Experiment tracking using systems like DAGshub or MLFlow for model monitoring etc

An example of an efficient architecture would be usage of tfx.

The reason for choice of tfx is because it can be further orchestrated on Kubeflow, airflow etc popular orchestrators and does not require heavy change of code.

Furthermore, the various components of TFX as well its advanced customization capabilities make it highly flexible for production ML pipelines and Machine Learning Life-Cycle maintainence.

A basic TFX and KubeFlow Workflow graph can be given as below:



This simple architecture takes advantage of the GCP infrastructure

Using GCP DataFlow for our data processing needs and tfx for efficient ML pipeline development and maintenance.

Model is served on a Kubernetes node used by the Tensorflow Service configuration.

The input data runners for tfx pipeline can be changed to BigQuery, TFRecords, CSV, or sharded files in GCP Storage Bucket etc

TFX exploits the capabilities of Apache Beam under the hood and hence the flexibility.