HW3 Report

Kapoor, Kartik 2462
Nham, Bryan 2494
Panyala, Sukrutha 8740
Vohra, Vedant 2889

Introduction

The objective of this project is to predict the ratings of products from Amazon, using the review data provided. The pre-processing of the data is done in Spark, using PySpark and the pre-processed datasets are used to train and evaluate 2 separate Deep Learning Models. One which considers only the review text as input and another one which uses both the review text as well as additional categorical and numerical features from the dataset provided.

Dataset

The dataset we chose to use was the amazon_reviews_grocery dataset. This dataset consists of 15 attributes, as shown below:

marketplace, customer_id, review_id, product_id, product_parent, product_title, product_category, star_rating, helpful_votes, total_votes, vine, verified_purchase, review_headline, review_body, review_date

Methodology

1. Use PySpark to pre-process the data. i.e. filter columns, perform transformations, clean and prepare the text, generate reduced datasets for training and testing.
2. Use TensorFlow and Keras to build 2 models to classify the reviews as positive or negative.
3. Train and evaluate the models.
4. Compare the results with a state-of-the-art model and report findings.

Data pre-processing

Since the raw data has a lot of unnecessary fields, a mix of text, categorical and numerical features and unprocessed text, we had to first pre-process this data, before it could be used to train a neural network. For this, we used PySpark.

Here is a brief analysis of why PySpark was the perfect tool for this task:

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, across the nodes of the cluster that can be operated on in parallel. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. Spark makes use of the concept of RDD to achieve faster and more efficient MapReduce operations.

PySpark Performance

PySpark (specifically the RDDs) performs better than stand-alone Python because of the following reasons:

1. **Shared Nothing Architecture -** RDDs (Resilient Distributed Datasets) take advantage of the shared-nothing architecture by splitting tasks between each node in the cluster (this allows for parallel processing). This allows RDDs to store data in the memory of each node, which allows it to perform in-memory computations. When data does not fit in the main memory, PySpark either recalculates the data or sends it to the disk. But the good thing is that when retrieving this data, it does not need to go to the disk. This also reduces the number of times the data needs to be moved between disk and memory. In contrast, stand-alone Python stores data in the heap. This works well until the data can't fit inside the heap. In our case the data probably won't fit in the main memory alone, so the amount of disk access is much higher when using the stand-alone Python version.
   Additionally, as the word ‘Resilient’ states, there is a level of fault-tolerance added by Spark, which prevents data loss in case a node fails during processing.

2. **Parallel Processing -** PySpark tries to run tasks in parallel as much as possible. To do this, the data is partitioned across worker nodes by PySpark. Stand-alone Python does not run tasks in parallel. When working in chunks it waits for that chunk to finish before moving on to the next chunk.

3. **Lazy Evaluation -** RDDs do not perform the computations right away. PySpark stores these computations in a DAG and waits until some data from the cluster is

requested. This allows PySpark to optimize these computations before executing them. This lowers the amount of computation needed. In stand-alone Python, computations are executed right away, which may create unnecessary computations and increase the access to memory and disk.

We have used RDDs and PySpark native API wherever possible, in order to best leverage the distributed computing environment to accelerate our pre-processing tasks. Below are the individual pre-processing tasks our PySpark script is designed to carry out.

Preprocessing Steps

STEP 1. SparkSession.builder is the entry point to Spark SQL. It is one of the very first objects we have to create while developing a Spark SQL application.

```
spark = SparkSession.builder.getOrCreate()
```

STEP 2. Spark.read.csv() is used to read a file or directory of files in CSV format into Spark DataFrame. Here we have modified it slightly to read a TSV file.

```
df =  spark.read.csv('dataset.tsv', sep=r'\t', header=True,
inferSchema=True)
```

Here, PySpark automatically handles reading of the data in chunks, which are stored in a distributed manner across multiple nodes in the Spark cluster. Note that every operation on this dataframe will be parallelized by Spark internally.

STEP 3. In PySpark, the select() function is used to select columns from a DataFrame. PySpark select() is a transformation function hence it returns a new DataFrame with the selected columns. Here we are using it to select the only specific columns from the DataFrame and eliminate the columns we don't need. withColumnRenamed() is used along with select to rename the column headers. We chose to not include the marketplace, customer_id, review_id, product_id, product_parent, product_title, product_category, review_headline, and review_date columns as this information is not useful in the prediction of the reviews and doesn't affect the outcome of the model.

```
df = df.select(lower('review_body'), 'star_rating', 'helpful_votes',
'total_votes', 'vine', 'verified_purchase'
).withColumnRenamed('lower(review_body)',
'text').withColumnRenamed('star_rating', 'y')
```

STEP 4. Now that we have only the required columns, we filter out the rows that have NULL values as they are of no use to us. This is achieved by using na.drop() function.

```
df.na.drop()
```

STEP 5. Now we modify the values of the ratings, vine, and verified_purchase columns. For ratings, anything greater than 1 is converted to 1 and anything less than or equal to 1 is converted to 0. For vine and verified_purchase columns, the value No('N') is converted to zero and Yes('Y') is converted to 1. withColumn() function is used to select a single column and when() function is used to modify the data according to the given conditions.

STEP 6. This step involves processing the text. Since PySpark does not have a native API to do this, we have chosen to use the NLTK library in Python. We use the rdd.map() function in PySpark to carry out these actions in parallel on the dataset.

The following processes are applied to the review text:

1. Stop-word removal
2. Stemming
3. Lemmatization

STEP 7. Finally, any remaining non-string data from the reviews column is filtered out using the filter() function.

```
df = df.filter(df.text.rlike("[a-z]"))
```

STEP 8. The data is then split into training and test sets using the randomSplit() function. This data is saved into separate CSV files.

```
df_train, df_test = df.randomSplit([0.8, 0.2])
```

STEP 9. Now, the Tokenizer() function from the PySpark ML library is used on the training data to extract the tokens from the reviews.

```
tokenizer = Tokenizer(outputCol="tokens")
tokenizer.setInputCol("text")
tokens = tokenizer.transform(df_train)
t = tokens.select("tokens")
```

STEP 10. These tokens are then passed to the PySpark ML Word2Vec() function which returns vectors for each of these tokens. These vectors are converted to word embeddings then stored in a CSV file.

```
word2Vec = Word2Vec(vectorSize=300, minCount=1, inputCol="tokens", outputCol="w2c").fit(t)
w2v = word2Vec.getVectors()
```

Challenges

The PySpark implementation of the ‘Word2Vec’ algorithm which is used to create vectors for words extracted by the Tokenizer, takes a lot of time to execute. Despite being distributed, we have observed that it is not as efficient as the single-system multi-threaded implementation from the ‘gensim’ library. From our experiments, we observed that ‘Word2Vec‘ takes over 70% of the total running time of the pre-processing script.

The vectors returned by the ‘Word2Vec’ function contain very large float values. The float values have up to 20 decimal places. Storage of this data results in very large files which are impractical for transfer and loading into memory. So, we had to come up with a solution to decrease the file size by limiting the number of decimal places to 4. We observed that this reduces the file size significantly, without negatively impacting the model performance.

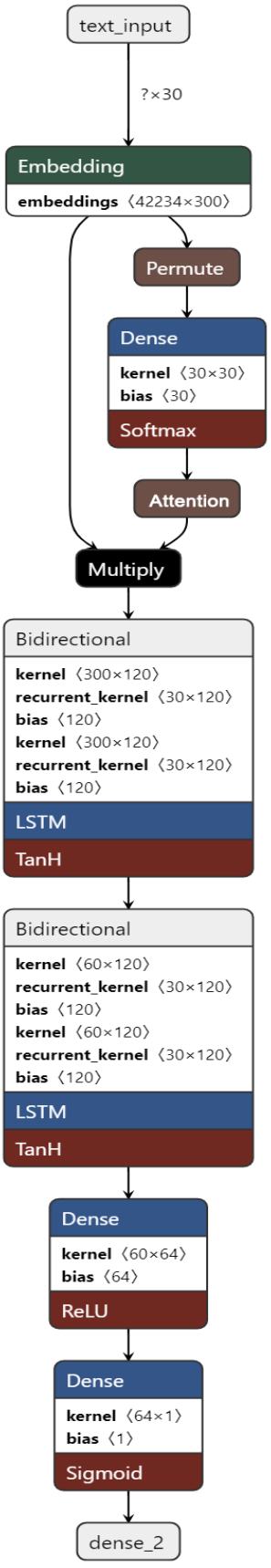
Building the classifier

We have decided to go with a Bi-directional LSTM network with an Attention layer for our classifier. This method allows the LSTM layer to be fed inputs from both directions and Attention is used to store intermediate outputs from the LSTM. This enables the model to learn the relationships between tokens and derive syntax and context from the text.

Based on this technique, we have built 2 separate models.

1. Text-only model

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
text_input (InputLayer)      [(None, 30)]              0           []

embedding (Embedding)        (None, 30, 300)           12670200    ['text_input[0][0]']

permute (Permute)            (None, 300, 30)           0           ['embedding[0][0]']

dense (Dense)                (None, 300, 30)           930         ['permute[0][0]']

attention (Permute)          (None, 30, 300)           0           ['dense[0][0]']

multiply (Multiply)          (None, 30, 300)           0           ['embedding[0][0]',
                                                                   'attention[0][0]']

bidirectional (Bidirectional)  (None, 30, 60)          79440       ['multiply[0][0]']

bidirectional_1 (Bidirectional  (None, 60)             21840       ['bidirectional[0][0]']
)

dense_1 (Dense)              (None, 64)                3904        ['bidirectional_1[0][0]']

dense_2 (Dense)              (None, 1)                 65          ['dense_1[0][0]']

==================================================================================================
Total params: 12,776,379
Trainable params: 106,179
Non-trainable params: 12,670,200
```


text_input
?×30
Embedding
embeddings 〈42234×300〉
Permute
Dense
kernel 〈30×30〉
bias 〈30〉
Softmax
Attention
Multiply
Bidirectional
kernel 〈300×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
kernel 〈300×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
LSTM
TanH
Bidirectional
kernel 〈60×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
kernel 〈60×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
LSTM
TanH
Dense
kernel 〈60×64〉
bias 〈64〉
ReLU
Dense
kernel 〈64×1〉
bias 〈1〉
Sigmoid
dense_2

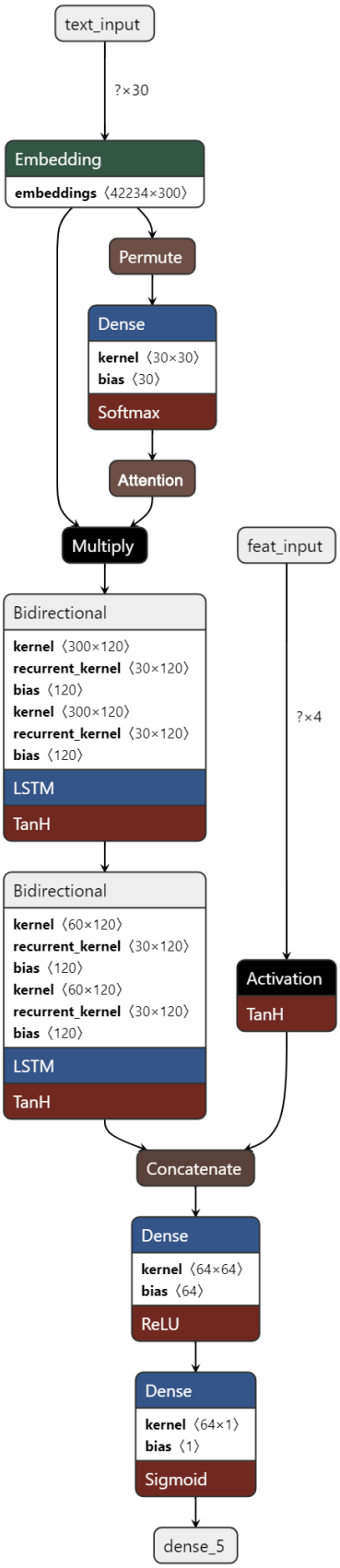
Layers:

- **Input**: Here, the input is the index form of the pre-processed text loaded from the training dataset.
- **Embeddings**: In this layer, we pass the word embeddings matrix generated by the pre-processing script. These embeddings act as weights for this layer and the layer is frozen and these weights cannot be updated during training.
- **Attention**: This is a combination of permute layers whose output is multiplied with the embeddings before being passed to the LSTM.
- **Bidirectional LSTM**: 2 layers of bidirectional LSTM to learn the text features from the input layer and the embeddings provided.
- **Prediction:** 2 dense layers reduce the feature map to output a single prediction.

2. Text + additional categorical/numerical features (helpful_votes, total_votes, vine, verified_purchase)

Layer (type)	Output Shape	Param #	Connected to
text_input (InputLayer)	[(None, 30)]	0	[]
embedding (Embedding)	(None, 30, 300)	12670200	['text_input[0][0]']
permute_1 (Permute)	(None, 300, 30)	0	['embedding[1][0]']
dense_3 (Dense)	(None, 300, 30)	930	['permute_1[0][0]']
attention (Permute)	(None, 30, 300)	0	['dense_3[0][0]']
multiply_1 (Multiply)	(None, 30, 300)	0	['embedding[1][0]', 'attention[0][0]']
bidirectional_2 (Bidirectional)	(None, 30, 60)	79440	['multiply_1[0][0]']
feat_input (InputLayer)	[(None, 4)]	0	[]
bidirectional_3 (Bidirectional)	(None, 60)	21840	['bidirectional_2[0][0]']
activation (Activation)	(None, 4)	0	['feat_input[0][0]']
concatenate (Concatenate)	(None, 64)	0	['bidirectional_3[0][0]', 'activation[0][0]']
dense_4 (Dense)	(None, 64)	4160	['concatenate[0][0]']
dense_5 (Dense)	(None, 1)	65	['dense_4[0][0]']

Total params: 12,776,635
Trainable params: 106,435
Non-trainable params: 12,670,200


text_input
?×30
Embedding
embeddings 〈42234×300〉
Permute
Dense
kernel 〈30×30〉
bias 〈30〉
Softmax
Attention
Multiply
feat_input
Bidirectional
kernel 〈300×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
kernel 〈300×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
LSTM
TanH
?×4
Bidirectional
kernel 〈60×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
kernel 〈60×120〉
recurrent_kernel 〈30×120〉
bias 〈120〉
LSTM
TanH
Activation
TanH
Concatenate
Dense
kernel 〈64×64〉
bias 〈64〉
ReLU
Dense
kernel 〈64×1〉
bias 〈1〉
Sigmoid
dense_5

This model is almost identical to the previous one, except for the following differences:

- A second Input layer (feat_input) is added, which is used for the non-text features.
- TanH activation layer is used to scale the features to the same range as the LSTM output
- These features are concatenated with the LSTM output just before the prediction layers and hence act as weights for the prediction layers.

Training

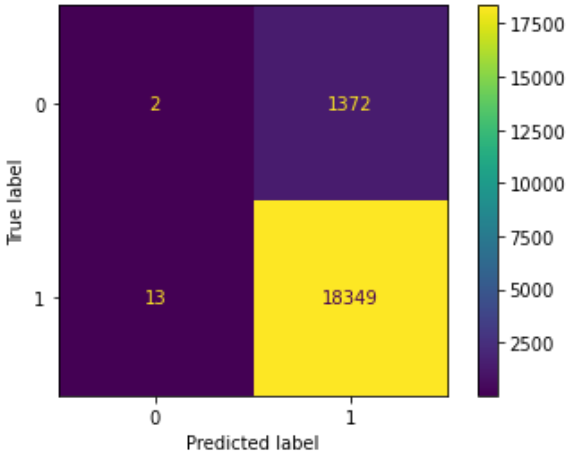
For training both models, we used the same configuration:

- Test/validation split: 80/20
- Batch size = 2048
- Epochs = 50
- Loss function: Binary Crossentropy
- Optimizer: Adam
- Metric: Accuracy
- Early stopping with patience = 5 – stops training when validation loss hasn’t changed for 5 epochs
- Model checkpointing to save the best performing model out of all the epochs.

Evaluation

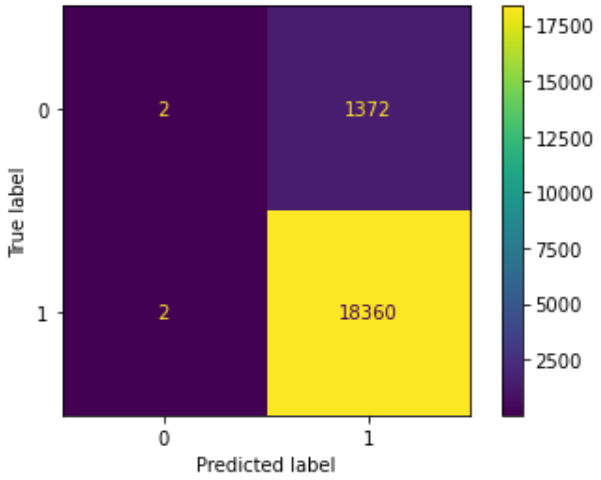
1. Text-only model

```
best val score: 0.2545878952232939
Accuracy (text only): 92.98%
```



2. Text + additional features

```
best val score: 0.2545878952232939
Accuracy (text + additional features): 93.04%
```



As is evident from the above scores, the Text + additional features model has performed slightly better by a fraction of a percent, although it had considerably fewer false negative errors.

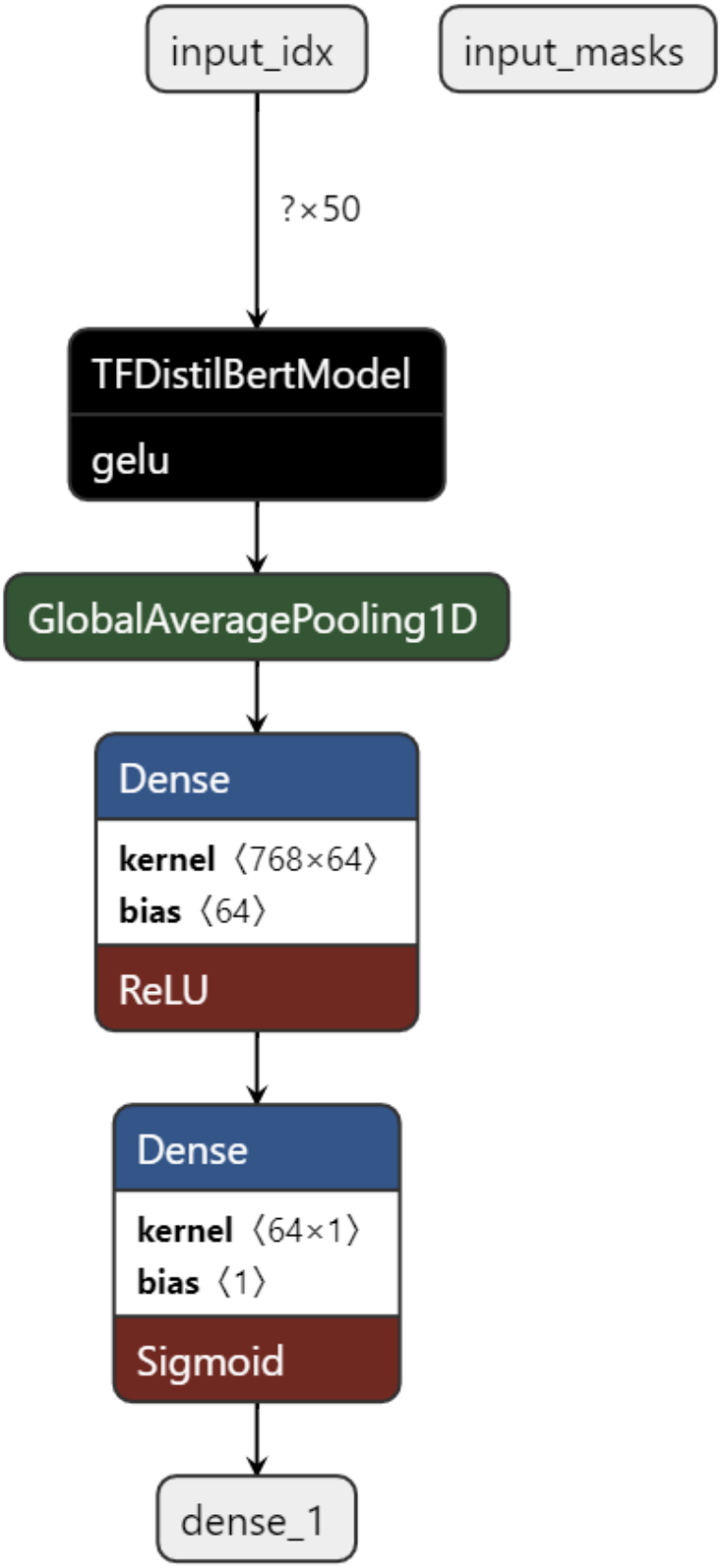
Model Comparison

We have evaluated these model against the state-of-the-art Transformer-based model, Distil-BERT (pre-trained on the ‘distilbert-base-uncased’ dataset), trained via transfer-learning for 10 epochs and fine-tuned on our dataset for an additional 5 epochs.

This approach doesn't require any pre-processing on the text, since Distil-BERT does it's own tokenization and embedding and the input was the raw review text from the source dataset.

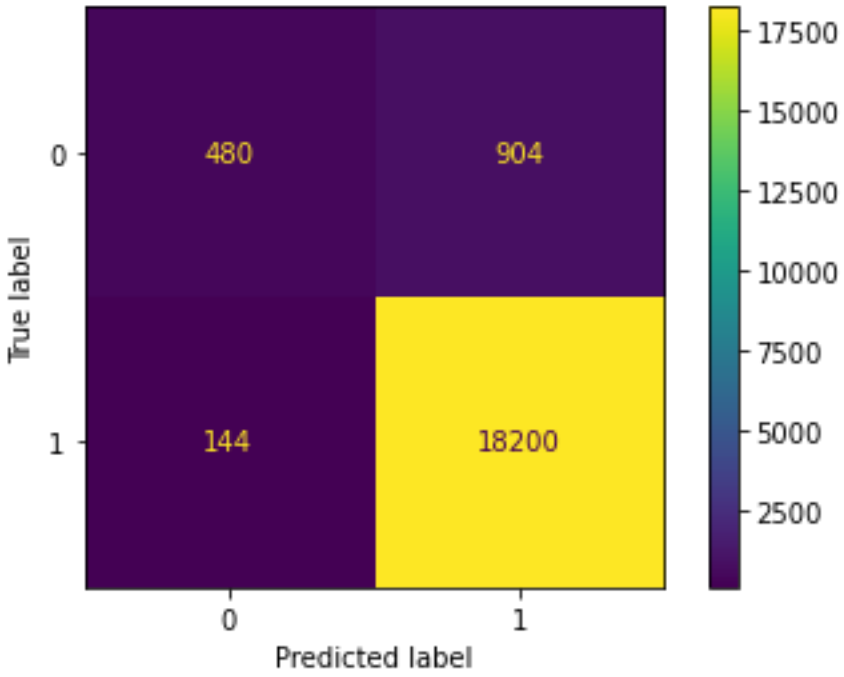
Distil-BERT architecture:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 input_idx (InputLayer)      [(None, 50)]              0           []

 input_masks (InputLayer)    [(None, 50)]              0           []

 tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880   ['input_idx[0][0]',
 BertModel)                  ast_hidden_state=(N                  'input_masks[0][0]']
                             one, 50, 768),
                              hidden_states=None
                             , attentions=None)

 global_average_pooling1d (Glob  (None, 768)       0           ['tf_distil_bert_model[0][0]']
 alAveragePooling1D)

 dense (Dense)               (None, 64)            49216       ['global_average_pooling1d[0][0]'
                                                               ]

 dense_1 (Dense)             (None, 1)             65          ['dense[0][0]']

==================================================================================================
Total params: 66,412,161
Trainable params: 49,281
Non-trainable params: 66,362,880
_________________________________________________________________
```


input_idx
input_masks
?×50
TFDistilBertModel
gelu
GlobalAveragePooling1D
Dense
kernel 〈768×64〉
bias 〈64〉
ReLU
Dense
kernel 〈64×1〉
bias 〈1〉
Sigmoid
dense_1

Evaluation

```
Accuracy: 94.69%
```



Our Bi-LSTM model ended up performing only 1.65% worse than Distil-BERT. Although BERT's inbuilt pre-processing ended up being faster thanks to GPU acceleration.

Responsibilities

Kapoor, Kartik: Preprocessing, Video, Report
Nham, Bryan: PySpark performance analysis and research
Panyala, Sukrutha: Model analysis, Report
Vohra, Vedant: Deep Learning implementation, Documentation, Video

Video

https://uofh-my.sharepoint.com/:v:/g/personal/vvohra2_cougarnet_uh_edu/Eb2rDjW5CntDnYmNALwMO28Bu_Tg1AEd9fXkV1YSIzg3aA?e=VLvjuv