Final Project Report: Semantic Similarity Detection

Nguyen Minh Duc - 21021292 , **Hoang Viet Tung - 22012345**
Institute of artificial intelligence
Natural Language Processing NLP3401

Abstract

This document is our final project report on semantic similarity detection in the field of Natural Language Processing (NLP).
The project focuses on finding the most basic but also most efficient method to determine the semantic similarity between texts.
As pre-trained BERT model achieved remarkable success in varieties of NLP tasks but these models are very large and consist of hundreds of millions of parameters, therefore, makes it hard to fine-tune, and online serving. Besides, latency and capacity constraints is also a big challenge. We have used a range of techniques with different models including 'bert-base-uncased, 'bert-base-cased' and 'bert-large-uncased', and evaluated them on MSRP dataset. This report provides an in-depth look at how the model 'MiniLM' was used as it yielded the best results with an accuracy of 86%, as well as the key discoveries from the project and the code that was written in Python.

1 Introduction

This "Semantics Similarity detection" project was handed to us in UET's Natural Language Processing course (2324I_INT3406_1) as the final project of the course. Using the Microsoft Research Paraphrase (MSRP) dataset that consists of over 5000 sentence pairs collected from news. Each pair has an annotation indicating paraphrased or non-paraphrased pairs. The dataset is splitted into 3 files: Train - Dev - Test for different purposes.
Semantics similarity detection is a crucial task in Natural Language Processing (NLP) with applications in various domains. Traditional approaches rely on hand-crafted features like word overlap and syntactic structure, while deep learning models extract complex features from data using neural networks like RNNs and LSTM. However, this task faces a lot of challenges due to the lack of labeled data and the subjective nature of semantics similarity. Ongoing research aims to develop more robust and objective methods for this task. Our project aims to achieve this task on the given dataset in the most straightforward and time-saving manner possible.
In this report, we introduce knowledge distillation, a promising way to compress a large model - by pre-training and fine-tuning it to student model with teacher features, labels and pre-trained features, true labels. There have been a lot of works that task-specifically pre-trained large LMs into smaller model. The distillation is effective, but fine-tuning large pre-trained models is still costly, especially for large datasets.
After trying different methods, we have chosen using "microsoft/MiniLM-L12-H384-uncased" pre-trained model to encode the text and orchestrate the process of training, fine-tune the model on the given dataset.MiniLM models provides lots of advantages like:
1. To compress large Transformer based pre-trained models, termed as deep self-attention module.
2. Distilling the self-attention module of the last Transformer layer of the teacher (base on teacher-student distilling).
3. Introducing the scaled dot-product between values in the self-attention module as a new deep self-attention knowledge.
4. Show that introducing a teacher assistant also helps the distillation of large pre-trained Transformer models.

2 Preliminary

In this section, we will present a brief introduction to the Transformer network and it's core component - the self-attention mechanism. We will also discuss about MiniLM as why we choose this model for this particular task.

2.1 Input Representation

The process of tokenizing texts is done by WordPiece in Bert to tokenize texts to subword units. For example, the word "forecasted" is split to "forecast" and "##ed", where "##" indicates the pieces are belong to one word. Some special token is also used in the process, `[SEP]` is used to separate segments if the input text contains more than one segment, `[CLS]` is added to the beginning of the sequence to obtain the representation of the whole input. Therefore, the vector representations ($x_i{}_{i=1}^{|x|}$) of input tokens are computed via summing the corresponding token embedding, absolute position embedding, and segment embedding.

2.2 Backbone Network: Transformer

Transformer model is used to encode contextual information for input tokens. The input vectors above ($x_i{}_{i=1}^{|x|}$) are packed together into $H^0 = [x_1, ..., x_{|x|}]$. Then stacked Transformer blocks compute the encoding vectors as:

$$\mathbf{H}^l = Transformer_l(\mathbf{H}^{l-1}), l \in [1, L] \quad (1)$$

where L is the number of Transformer layers, and the final output is $\mathbf{H}^l = [\mathbf{h}_1^L, ..., \mathbf{h}_{|x|}^L]$. used as the contextualized representation of x_i. Each Transformer layer consists of a self-attention sub-layer and a fully connected feed-forward network. Residual connection (connects the output of one earlier layer to the input of another future layer several layers later) is employed around each of the two sub-layers, followed by layer normalization (normalize each of the inputs in the batch independently across all features).

Self-Attention

In each layer, Transformer uses multiple self-attention heads to aggregate the output vectors of the previous layer. For the l-th Transformer layer, the output of a self-attention head $\mathbf{AO}_{l,a}, a \in [1, A_h]$ is computed by:

$$\mathbf{Q}_{l,a} = \mathbf{H}^{l-1}\mathbf{W}_{l,a}^Q \quad (2)$$

$$\mathbf{V}_{l,a} = \mathbf{H}^{l-1}\mathbf{W}_{l,a}^V \quad (3)$$

$$\mathbf{K}_{l.a} = \mathbf{H}^{l-1}\mathbf{W}_{l,a}^K \quad (4)$$

$$\mathbf{A}_{l,a} = softmax(\frac{\mathbf{Q}_{l,a}\mathbf{K}_{l,a}^{\mathbf{T}}}{\sqrt{d_k}}) \quad (5)$$

$$\mathbf{AO}_{l,a} = \mathbf{A}_{l,a}\mathbf{V}_{l,a} \quad (6)$$

where the previous layer's output $\mathbf{H}^{l-1} \in R^{|x|} \times d_h$ is linearly projected to a triple of queries, keys and values using parameter matrices $\mathbf{W}_{l,a}^Q, \mathbf{W}_{l,a}^K, \mathbf{W}_{l,a}^V \in R^{d_h \times d_k}$, respectively, $\mathbf{A}_{l,a} \in R^{|x| \times |x|}$ indicates the attention distributions, which is computed by the scaled dot-product of queries and keys. A_h represents the number of self-attention heads. $d_k \times A_h$ is equal to the hidden dimension d_h in BERT.

2.3 Transformer Distillation

Knowledge distillation is to train the small student model S on a transfer feature set with soft labels and intermediate representations provided by the large teacher model T.Knowledge distillation is modeled as minimizing the differences between teacher and student features:

$$L_{KD} = \sum_{e \in D} L(f^S(e), f^T(e)) \quad (7)$$

Where D denotes the training data, $f^S(\cdot)$ and $f^T(\cdot)$ indicate the features of student and teacher models respectively, $L(\cdot)$ represents the loss function. The mean squared error (MSE) and KL-divergence are often used as loss functions.
For Transformer based LM distillation, soft target probabilities for masked language modeling predictions, embedding layer outputs, self-attention distributions and outputs (hidden states) of each Transformer layer of the teacher model are used as features to help the training of the student. Soft labels and embedding layer outputs are used in DistillBERT. TinyBERT and MOBILEBERT further utilize self-attention distributions and outputs of each Transformer layer. For MOBILEBERT, the student is required to have the same number of layers as its teacher to perform layer-to-layer distillation. Besides, bottleneck and inverted bottleneck modules are introduced to keep the hidden size of the teacher and student are also the same. To transfer knowledge layer-to-layer, TinyBERT employs a uniform-function

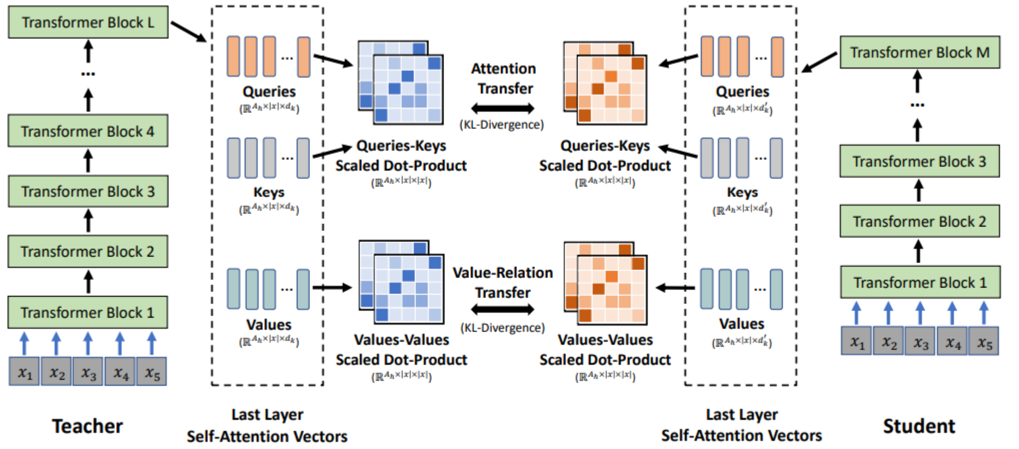


Figure 1. Overview of Deep Self-Attention Distillation. The student is trained by deeply mimicking the self-attention behavior of the last Transformer layer of the teacher. In addition to the self-attention distributions, we introduce the self-attention value-relation transfer to help the student achieve a deeper mimicry. Our student models are named as MiniLM.

Figure 1: Model architecture

to map teacher and student layers. Since the hidden size of the student can be smaller than its teacher, a parameter matrix is introduced to transform the student features.

3 Deep Self-Attention Distillation

Figure 1 gives an overview of the deep self-attention distillation. The key idea is three-fold. First, we propose to train the student by deeply mimicking the self-attention module of the teacher's last layer. Second, we introduce transferring the relation between values (i.e., the scaled dot-product between values) to achieve a deeper mimicry, in addition to performing attention distributions (i.e., the scaled dot product of queries and keys) transfer in the self-attention module. We also discuss about how introducing a teacher assistant also helps the distillation of large pre-trained Transformer models when the size gap between the teacher model and student model is large.

3.1 Self-Attention Distribution Transfer

Transferring self-attention distributions has been used in previous works for Transformer distillation. MiniLM crews utilize the self-attention distributions by minimizing the KL-divergence between the self-attention distributions of the teacher and student:

$$L_{AT} = \frac{1}{A_h|x|} \sum_{a=1}^{A_h} \sum_{t=1}^{|x|} D_{KL}(\mathbf{A}_{L,a,t}^T || \mathbf{A}_{M,a,t}^S) \tag{8}$$

Where |x| and A_h represent the sequence length and the number of attention heads. L and M represent for the layers of teacher and student. A_L^T and A_M^S are the attention distributions of the last Transformer layer for the teacher and student, respectively. They are computed by the scaled dot-product of queries and keys. Using only the attention maps of the teacher's last Transformer layer differs MiniLM from the previous works which transfer teacher's knowledge layer-to-layer. This approach allows more flexibility for the number of layers of the student models, avoids the need of finding the best layer mapping.

3.2 Self-Attention Value-Relation Transfer

MiniLM also use the relation between values in the self-attention module to guide the training of the student. The value relation is computed via the multi-head scaled dot-product between values. The KL-divergence between the value relation of the teacher and student is used as the training objective:

$$\mathbf{VR}_{L,a}^T = softmax(\frac{\mathbf{V}_{L,a}^T \mathbf{V}_{L,a}^T{}^{\top}}{\sqrt{d_k}}) \tag{9}$$

$$\mathbf{VR}^S_{M,a} = softmax\left(\frac{\mathbf{V}^S_M {\mathbf{V}^S_{M,a}}^\top}{\sqrt{d'_k}}\right) \tag{10}$$

$$L_{VR} = \frac{1}{A_h|x|}\sum_{a=1}^{A_h}\sum_{t=1}^{|x|} D_{KL}(\mathbf{VR}^T_{L,a,t}||\mathbf{VR}^S_{M,a,t}) \tag{11}$$

Where $\mathbf{V}^T_{L,a} \in R^{|x|\times d_k}$ and $\mathbf{V}^S_{M,a} \in R^{|x|\times d'_k}$ are the values of an attention head in self-attention module for the teacher's and student's last Transformer layer. $\mathbf{VR}^T_L \in R^{A_h\times|x|\times|x|}$ and $\mathbf{VR}^S_M \in R^{A_h\times|x|\times|x|}$ are the value relation of the last Transformer layer for teacher and student, respectively.
The training loss is computed via summing the attention distribution transfer loss and value-relation transfer loss:

$$L = L_{AT} + L_{VR} \tag{12}$$

Introducing the relation between values enables the student to deeply mimic the teacher's self-attention behavior. Moreover, using the scaled dot-product converts vectors of different hidden dimensions into the relation matrices with the same size, which allows students to use more flexible hidden dimensions and avoids introducing additional parameters to transform the student's representations.

3.3 Teacher Assistant

Assuming the teacher model consists of L-layer Transformer with d_h hidden size, the student model has M-layer Transformer with d'_h hidden size. For smaller students ($M <= \frac{1}{2}L, d'_h <= \frac{1}{2}d_h$), we first distill the teacher into a teacher assistant layer with L-layer Transformer and d'_h hidden size. The assistant model is then used as the teacher to guide the training of the final student.
Teacher assistant bridges the size gap between teacher and smaller student models, helps the distillation of Transformer based pre-trained LMs, and also brings further improvement for smaller student models.

4 Our Works

With all the information being said, it is time to move on to our real works, how we processed the given dataset, implemented the model, fine-tune the model and evaluate it. All of the code was written in Python on Jupyter Notebook.

4.1 Libraries

Firstly, we import all the libraries libraries and tools that going to help us in this particular task.

```
import numpy as np
import pandas as pd
import tensorflow as tf
import transformers
import os
```

Here some brief explanations why we used these libraries:

- `numpy`: Fundamental for scientific computing in Python. It provides support for arrays, matrices and many mathematical functions that we going to use in the process.

- `pandas`: Offering data structures and operations for manipulating numerical tables and time series, which is essential for handling our dataset.

- `tensorflow`: A free and open-source software library for machine learning and artificial intelligence. It can be used across range of tasks, especially on training and inference our model.

- `transformers`: As discussed above, this library provides state-of-the-art machine learning models for Natural Language Processing. It is how we can access to our pre-trained model.

- `os`: A module in python provides a way of using operating system dependent functionality. We will use this module to save our model's weight in the training process

4.2 Processing the Data

As discussed, we were given MSRP Dataset with 3 seperated file, this part shows how we handled the data and prepared them for the training process.

4.2.1 Loading the Data

We were given 3 `.tsv` files with different purpose:

- `train.csv`: Data for training.

- `dev.csv`: Data for validation.
- `test.csv`: Data for evaluation.

We will use `pandas`'s `.read_csv()` function to read-in the data. In this very first step, we encountered a problem which all 3 of the `.tsv` files have a very small number of lines with more than 1 fields. After multiple tests (we added a 6-th field to the data file and try to find if there were any extra field's value, but we got none of them), we have decided to add `on_bad_lines` argument to `.read_csv` function so that we could skip all the lines that raise the error. Here is the code:

```
    # Read in tsv file, skip bad lines
train_df = pd.read_csv("train.tsv",
    sep='\t', on_bad_lines='skip')
test_df = pd.read_csv("test.tsv", sep='\t',
    on_bad_lines='skip')
dev_df = pd.read_csv("dev.tsv", sep='\t',
    on_bad_lines='skip')
```

4.2.2 Pre-processing the Data

We quickly ran `.info()` function to get a look at 3 dataframes' condition and immediately saw that in all 3 dataframes, at `'#2 String'` column, there were some rows that missing this field. The number is not too significant so we have chosen to drop out the data, although we could have filled the missing string by `'ffill'` method (replaces NULL values with value from the previous row) and set the annotation of the rows to non-paraphrased (which is 0). Dropping NULL rows:

```
    train_df.dropna(inplace=True)
    dev_df.dropna(inplace=True)
    test_df.dropna(inplace=True)
```

4.2.3 Data Reduction and Transformation

This is the last step to prepare the data for the training process. Since we only need the sentence pairs and the annotation, we will drop all the other columns which is `'#1 ID'` and `'#2 ID'`:

```
    train_df = train_df.drop(['#1 ID', '#2
        ID'], axis=1)
dev_df = dev_df.drop(['#1 ID', '#2 ID'],
    axis=1)
test_df = test_df.drop(['#1 ID', '#2 ID'],
    axis=1)
```

Now let's move on to the main part.

4.3 Main Model and Algorithm

In order to start training, we will need to generates batches of data from our dataset that fit into our model initializes all the model's layers and run the code. This section discuss about how we did it.

4.3.1 Generate Batches of Data

```
class BertSemanticDataGenerator(
tf.keras.utils.Sequence):
"""Generates batches of data.

Args:
    sentence_pairs: Array of premise and
        hypothesis input sentences.
    labels: Array of labels.
    batch_size: Integer batch size.
    shuffle: boolean, whether to shuffle
        the data.
    include_targets: boolean, whether to
        incude the labels.

Returns:
    Tuples `([input_ids, attention_mask,
        `token_type_ids], labels)`
    (or just `[input_ids, attention_mask,
        `token_type_ids]`
     if `include_targets=False`)
"""

def __init__(
    self,
    sentence_pairs,
    labels,
    batch_size=batch_size,
    shuffle=True,
    include_targets=True,
):
    self.sentence_pairs = sentence_pairs
    self.labels = labels
    self.shuffle = shuffle
    self.batch_size = batch_size
    self.include_targets = include_targets
    # Load our BERT Tokenizer to encode
        the text.
    # We will use MiniLM-L12-H384-uncased
        pretrained model (for lower-case
        text).
    self.tokenizer =
        transformers.BertTokenizer.from_pretrained(
       "microsoft/MiniLM-L12-H384-uncased",
            do_lower_case=True
    )
    self.indexes =
        np.arange(len(self.sentence_pairs))
    self.on_epoch_end()

def __len__(self):
    # Denotes the number of batches per
        epoch.
    return len(self.sentence_pairs) //
        self.batch_size

def __getitem__(self, idx):
    # Retrieves the batch of index.
```

```
        indexes = self.indexes[idx *
            self.batch_size : (idx + 1) *
            self.batch_size]
        sentence_pairs =
            self.sentence_pairs[indexes]

        # With BERT tokenizer's
            batch_encode_plus batch of both
            the sentences are
        # encoded together and separated by
            [SEP] token.
        encoded =
            self.tokenizer.batch_encode_plus(
           sentence_pairs.tolist(),
           add_special_tokens=True,
           max_length=max_length,
           return_attention_mask=True,
           return_token_type_ids=True,
           pad_to_max_length=True,
           return_tensors="tf",
        )

        # Convert batch of encoded features
            to numpy array.
        input_ids =
            np.array(encoded["input_ids"],
            dtype="int32")
        attention_masks =
            np.array(encoded["attention_mask"],
            dtype="int32")
        token_type_ids =
            np.array(encoded["token_type_ids"],
            dtype="int32")

        # Set to true if data generator is
            used for training/validation.
        if self.include_targets:
           labels =
               np.array(self.labels[indexes],
               dtype="int32")
           return [input_ids,
               attention_masks,
               token_type_ids], labels
        else:
           return [input_ids,
               attention_masks,
               token_type_ids]

    def on_epoch_end(self):
        # Shuffle indexes after each epoch if
            shuffle is set to True.
        if self.shuffle:
           np.random.RandomState(42).shuffle(
           self.indexes)
```

This class is used to generate batches of data for training and evaluating our model. It is a custom data generator for this particular task. Here is a breakdown of the key components and methods in this class:

- `__init__`: Initializes the BERT tokenizer (using `MiniLM-L12-H384-uncased`) for encoding the text and sets up some internal variables.
- `__len__`: This method returns the number of batches per epoch.
- `__getitem__`: This method retrieves a batch of data given an index. It performs the following step:
  - Retrieves the indexes of the current batch.
  - Selects the sentence pairs corresponding to those indexes.
  - Encodes the sentence pairs using the BERT tokenizer, adding special tokens like `[SEP]`.
  - Converts the encoded features into NumPy arrays, including input IDs, attention masks, and token type IDs.
  - If `include_targets` is set to `True`, it also includes the labels in the output.
- `on_epoch_end`: This method is called after each epoch to shuffle the indexes if `'shuffle'` is set to `True`. Shuffling the data helps to introduce randomness and improve training convergence.

4.3.2 Prepare labeled-Data

`This step converts our target labels into one-hot encoded format using tf.keras.utils.to_categorical():`

```
# Convert to one-hot vector
y_train = tf.keras.utils.to_categorical(
train_df.Quality, num_classes=2)
y_dev = tf.keras.utils.to_categorical(
dev_df.Quality, num_classes=2)
y_test = tf.keras.utils.to_categorical(
test_df.Quality, num_classes=2)
```

The `to_categorical()` function converts the class labels into one-hot encoded vectors. In this case, each label is converted into a 2-dimensional vector with values $[1, 0]$ or $[0, 1]$.for example: if 'Quality' is $[0, 1, 1, 0, 1]$ after applying the function,it becomes $[[1, 0], [0, 1], [0, 1], [1, 0], [0, 1]]$.

4.3.3 Create Model

This section demonstrates the construction of our model for "Semantic similarity detection" using BERT embeddings and TensorFlow's `MirroredStrategy` for distributed training:

```
    # Create the model under a distribution
        strategy scope.
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
```

```
# Encoded token ids from BERT tokenizer.
input_ids = tf.keras.layers.Input(
    shape=(max_length,), dtype=tf.int32,
        name="input_ids"
)
# Attention masks indicates to the model
    which tokens should be attended to.
attention_masks = tf.keras.layers.Input(
    shape=(max_length,), dtype=tf.int32,
        name="attention_masks"
)
# Token type ids are binary masks
    identifying different sequences in
    the model.
token_type_ids = tf.keras.layers.Input(
    shape=(max_length,), dtype=tf.int32,
        name="token_type_ids"
)
# Loading pretrained BERT model.
bert_model =
    transformers.TFBertModel.from_pretrained(
"microsoft/MiniLM-L12-H384-uncased")
# Freeze the BERT model to reuse the
    pretrained features without
    modifying them.
bert_model.trainable = False

bert_output = bert_model.bert(
    input_ids,
        attention_mask=attention_masks,
        token_type_ids=token_type_ids
)
sequence_output =
    bert_output.last_hidden_state
pooled_output = bert_output.pooler_output
# Add trainable layers on top of frozen
    layers to adapt the pretrained
    features on the new data.
bi_lstm = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(64,
        return_sequences=True)
)(sequence_output)
# Applying hybrid pooling approach to
    bi_lstm sequence output.
avg_pool =
    tf.keras.layers.GlobalAveragePooling1D()(bi_lstm)
max_pool =
    tf.keras.layers.GlobalMaxPooling1D()(bi_lstm)
concat =
    tf.keras.layers.concatenate([avg_pool,
    max_pool])
dropout =
    tf.keras.layers.Dropout(0.3)(concat)
output = tf.keras.layers.Dense(2,
    activation="softmax")(dropout)
model = tf.keras.models.Model(
    inputs=[input_ids, attention_masks,
        token_type_ids], outputs=output
)

model.compile(
optimizer=tf.keras.optimizers.Adam(),
    loss="categorical_crossentropy",
    metrics=["acc"],
)
```

Breakdown of the code:

- `tf.distribute.MirroredStrategy()` : Creating a mirrored strategy, allowing us to train model on multiple GPUs, taking advantage of their parallel processing capabilities. The code is then run in the context of the distributed strategy with `with strategy.scope()`.
- Input layers: Three input layers (`'input_ids', 'attention_masks', 'token_type_ids'`) are defined. Each input has a specified shape and data type and they are correspond to the input data for BERT-based models.
- Load pre-trained model: The pre-trained BERT model is loaded from the "microsoft/MiniLM-L12-H384-uncased" checkpoint. Then, the model is frozen by setting `bert_model.trainable` to `False` which allow us to use the pre-trained BERT features without modifying them.
- The input data is passed through the model, producing a `bert_output` that contains information about the sequence output and pooled output.
- Additional Layers:
  - A Bidirectional LSTM layer is added on top of the BERT output, with 64 units and returning sequences helping capture the contextual information from the BERT embeddings.
  - A hybrid pooling approach is also applied to the output of the LSTM layer. Concatenating the result of global average pooling and global max pooling.
  - A dropout layer with a dropout rate of 0.3 is added to prevent overfitting.
  - A final dense layer with 2 units and softmax activation is added for classification. This output layer produces probabilities for each class.
- Model Compilation: The model is compiled using the Adam optimizer, categorical cross-entropy loss and accuracy as the metric.

4.3.4 The Training Process

First we create the data generator, let's generate all at once:

```
train_data = BertSemanticDataGenerator(
train_df[["#1 String", "#2
     String"]].values.astype("str"),
    y_train,
    batch_size=batch_size,
    shuffle=True,
)

# Create the validation data generator.
valid_data = BertSemanticDataGenerator(
    dev_df[["#1 String", "#2
        String"]].values.astype("str"),
    y_dev,
    batch_size=batch_size,
    shuffle=True,
)

# Create the test data generator.
test_data = BertSemanticDataGenerator(
    test_df[["#1 String", "#2
        String"]].values.astype("str"),
    y_test,
    batch_size=batch_size,
    shuffle=True,
)
```

Then, let's create a callback that saves model's weights after each epoch:

```
    checkpoint_path = "cp.ckpt"
checkpoint_dir =
     os.path.dirname(checkpoint_path)

# Create a callback that saves the model's
     weights
cp_callback =
     tf.keras.callbacks.ModelCheckpoint(
filepath=checkpoint_path,
save_weights_only=True,
verbose=1)
```

Now, fit the data to the model:

```
history = model.fit(
    train_data,
    validation_data=valid_data,
    epochs= epochs,
    use_multiprocessing=True,
    workers=-1,
    callbacks=[cp_callback],
)
```

After 3 epochs, we got:

- loss: 0.4231
- acc: 0.8003
- val_loss: 0.3832
- val_acc: 0.8190

4.3.5 Fine-tune the Model

In order to fine-tune the model, we need to unfreeze the model first and then recompile the model to make the change effective and then re-training the model. Here is the code:

```
    # Unfreeze the bert_model.
bert_model.trainable = True
# Recompile the model to make the change
     effective.
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
```

```
history = model.fit(
    train_data,
    validation_data=valid_data,
    epochs= epochs,
    use_multiprocessing=True,
    workers=-1,
    callbacks=[cp_callback],
)
```

After this process, we got:

- loss: 0.1200
- accuracy: 0.9613
- val_loss: 0.2532
- val_accuracy: 0.9009

We have done the main process, we can now save the model for later use by using `model.save()` like this:

```
tf.keras.models.save_model(model, 'my_model')
# Load by
model =
     tf.keras.models.load_model('my_model')
```

4.4 Evaluation

This section is divided into 2 parts: Computing accuracy and F1-score.
We can calculate model's accuracy on the test data by:

```
model.evaluate(test_data)
```

Got: loss: 0.3459264934062958 - accuracy: 0.8632425665855408
We can calculate model's F1 score by:

```
from sklearn.metrics import f1_score
# F1 score

test_data = BertSemanticDataGenerator(
    test_df[["#1 String", "#2
        String"]].values.astype("str"),
        labels=None, batch_size=1,
        shuffle=False, include_targets=False,
)

# Make predictions on test data
y_pred = model.predict(test_data)

# Convert predictions from probabilities to
     labels
```

```
y_pred = np.argmax(y_pred, axis=1)

# Convert actual labels from one-hot
    encoding to integers
y_true = np.argmax(y_test, axis=1)

# Calculate F1-score
f1 = f1_score(y_true, y_pred)

print("F1-score:", f1)
```

Got F1-score: 0.900763358778626
We can also test model's accuracy on 2 input sentence by defining a check probability function as mentioned below:

```
def check_similarity(sentence1, sentence2):
    sentence_pairs =
        np.array([[str(sentence1),
        str(sentence2)]])
    test_data = BertSemanticDataGenerator(
        sentence_pairs, labels=None,
            batch_size=1, shuffle=False,
            include_targets=False,
    )

    proba = model.predict(test_data[0])[0]
    idx = np.argmax(proba)
    proba = f"{proba[idx]: .2f}"
    pred = labels[idx]
    return pred, proba
```

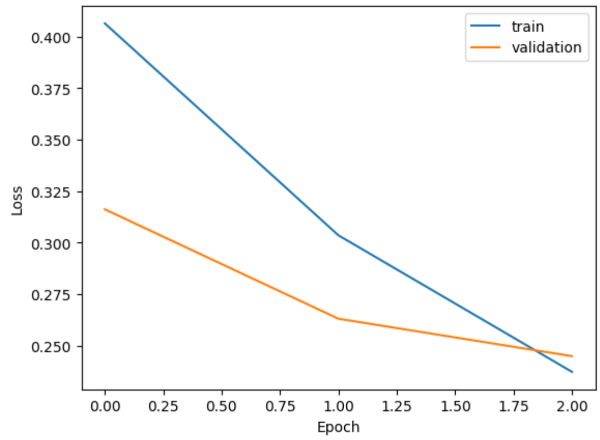
5 Future Development and Conclusion

5.1 Future Development

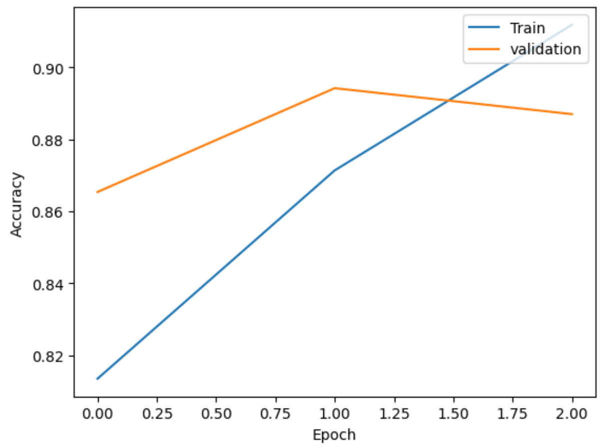
In the development phase of our project, we will be focused on building a more robust model. Starting by colleting a large dataset of text data from various sources.

5.2 Conclusion

Our model was able to successfully compute sentence pairs semantic similarity score. This has numerous applications, such as QA system, information retrieval, ...
However, there are still areas for improvement. In the future, we plan to incorporate more sophisticated NLP techniques in this various field.
Overall, this project has demonstrated the power of NLP in extracting valuable insights from text data. We look forward to continuing to improve and expand upon our work in this exciting field.



((a)) Loss with 3 epoch



((b)) Accuracy with 3 epoch

References

MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers

MSRP: The Microsoft Research Paraphrase dataset