Human Behaviour Understanding Using Machine Learning and Psychological Methods

Depression Detection in Social Media

Nour Eldin Alaa Badr
Wilmar Adrian Granados Cardenas

December 19, 2020

Abstract

Language plays a central role for psychologists who rely on manual coding of patient language for diagnosis. Using language technology for psychological diagnosis, particularly for depression detection could lead to low-priced screening test and affordable treatment. For Human Behaviour Understanding Using MachineLearning and Psychological Methods seminar's project we explore two neural architectures: Recurrent Neural Net-work (RNN) and Bidirectional Encoder Representations from Transformer (BERT), for depression detection on Reddit posts.In this report we will describe the data preparation and pre-procesing as well as the implementation of the RNN and BERT models for detecting depression.

Contents

1 Introduction **4**

2 Dataset Overiew **5**

3 Libraries used **5**
3.1 Why did we choose Pytorch instead of TensorFlow 5
3.2 Why did we choose Spacy instead of NLTK 6

4 Folders and XML files parsing **6**

5 Data pre-processing **8**

6 Language Models **12**
6.1 RNN approach 12
6.1.1 Introduction 12
6.1.2 RNN Model 13
6.1.3 Depression detection network architecture 14
6.1.4 Training the network 18
6.1.4.1 Loss and optimization functions 19
6.1.4.2 Model hyperparameters 19
6.1.4.3 Main steps of training the model 20
6.1.4.4 Cross validation 22
6.1.4.5 Results and observations 23
6.1.5 Next steps regarding RNN 25
6.2 Bert approach 25
6.2.1 Introduction 25
6.2.2 Bert Model 26
6.2.3 Tokenization with Bert 26
6.2.4 BertClassifier 31
6.2.4.1 Bert Hyper-parameters 31
6.2.4.2 Bert Training and Validation Functions . . . 32
6.2.4.3 Training and Results 35
6.2.5 Insights gained 36

References **39**

1 Introduction

Major Depressive Disorder (MDD), or commonly known as depression, is a health condition characterized by persistent sadness and lack of interest for previously enjoyable activities, feelings of guilt, disturbed sleep, tiredness, and poor concentration. These affections become depression symptoms when their occurrences last for at least 2 weeks [12]. Depending on the severity of the symptoms, psychologists classify depression as mild, moderate, or severe. The World Health Organization (WHO) considers depression a critical disease, as it might lead to suicide[20]. According to a Global Health Metrics article, more than 264 million people all over the world suffer from depression[7].

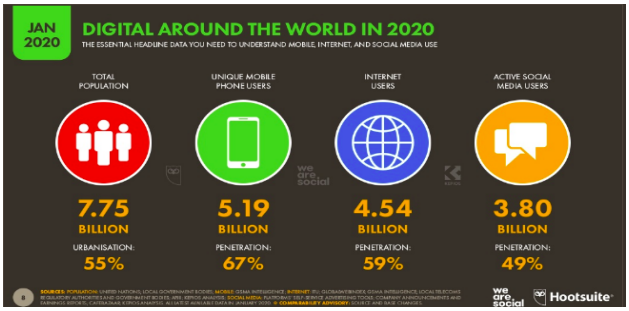

Figure 1: Global Reportal Slide (8) [8]

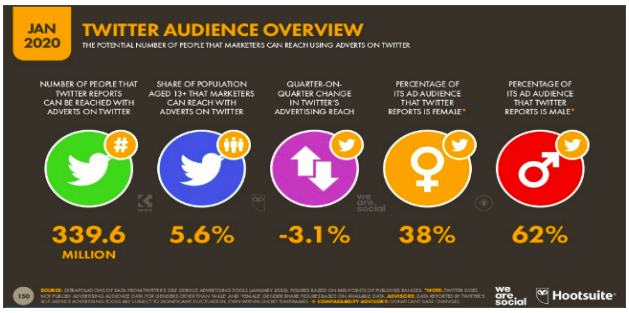

Figure 2: Number of twitter users slide (150) [8]

For clinical psychologists, language plays a central role in diagnosing depression and hence, many clinical instruments rely on manual coding of patient

language. Applying language technology in mental health diagnosis could lead to low-priced screening measures administered by more healthcare professionals. Researchers had begun targeting such issues using language technology to identify emotion in suicide notes, analyze the language of those with autistic spectrum disorders, and aid the diagnosis of dementia.[14] As depression increases on a global scale, it could become a global burden, which is why computer scientists and social researchers have cooperated to find ways for early detection of depression [2] and treatment [11] [5] using AI technologies. They found that AI can be a big help in detecting depression through various approaches, as it learns fast, can detect depression with high accuracy, can detect it from facial expressions, voices, and texts. As such, research on early detection of depression has shown relevant insight and brought us to the idea of detecting it in texts on social media.

2 Dataset Overiew

We have used a dataset from the Early Risk Prediction on the Internet (erisk 2017) workshop. This dataset provides user-generated content consisting of Reddit posts organized and processed chronologically. Every user has received a label as risk or non-risk (of depression). The dataset has two parts: a training dataset and a test dataset, and each part has ten chunks with series of XML files. These XML files store users' posts and the respective posts' comments.

3 Libraries used

When it comes to training model in deep learning, our minds goes to two powerful tools which are Pytorch and Tensorflow. While in tokenization there are Spacy and NLTK. In this project we have used Pytorch library for training and we have used Spacy for tokenization.

3.1 Why did we choose Pytorch instead of TensorFlow

Tensorflow has some advantage if it is choosen for training your model, such as it has a useful tool called Tensorboard which allows you to visualize your machine learning model, this tool isn't available in Pytorch. Also Tensorflow has a larger online community than Pytorch. Despite of all these advantages

we have decided to use Pytorch as it is more pythonic. By this we mean that you don't have to spend a lot of time learning how to use the library as the case in Tensorflow, you only need to know how to program in python which make it very easy and straight forward to use.

3.2 Why did we choose Spacy instead of NLTK

Our selection of SpaCy over NLTK has one main reason: performance. NLTK is a well-developed library for string processing. It takes a string as input and returns a string or a list (array) of strings. However, NLTK's array focus makes it a slower option than SpaCy for processing large amounts of text. SpaCy has an object-orient approach, meaning that it takes a string as input and returns a document object whose chunks and tokens are also objects. Spacy's OOP approach complements its industry-driven development that promotes its implementation with the fastest algorithms for text processing.

4 Folders and XML files parsing

This section focuses on folders and XML parsing. As shown in the following class diagram figure3, this class consists of 2 main functions:

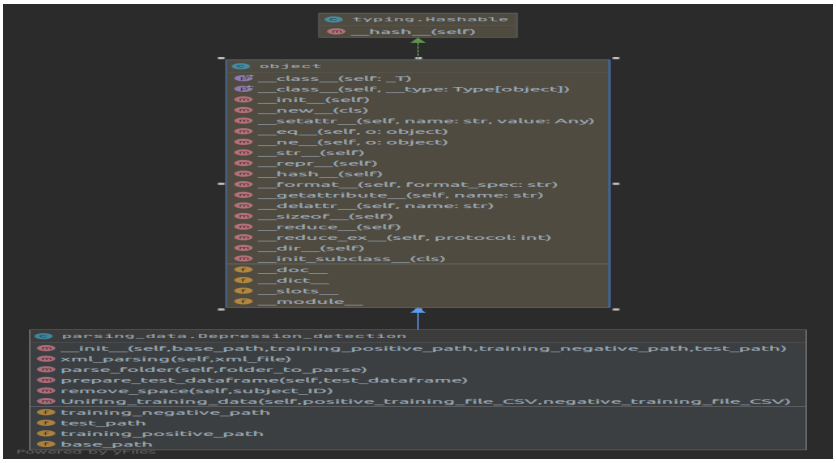

Figure 3: Folders and XML files parsing

- **Parsing Folder function**: is the function that parses the training/testing directory and the folders inside it in order to call the function that

parses the XML files. This is done by iterating on each directory and till finding a file and then call the XML parsing function

```
def parse_folder(self,folder_to_parse):
    '''
    1. It parses the training /testing directory where each folder contains 10 chunks as well as
    2. It calls the XML parser and returns list of data frames

    :param folder_to_parse:
    :return: list of frames
    '''
    frames = []
    folders_path = Path(folder_to_parse)
    for directory in folders_path.iterdir():
        if directory.is_dir():
            print(directory.name)
            for file in directory.iterdir():
                if file.is_file() and not file.name.startswith('._'):
                    frame = self.xml_parsing(file)
                    frames.append(frame)

    return frames
```

Listing 1: Parsing Folders

- **Parsing XML function**:This function parses the xml files into a dataframe with 5 columns ID,TITLE(which is the post),DATE,INFO and the TEXT(which are the comments).

```
def xml_parsing(self,xml_file):
    ''' Parses a xml file  with the following structure
      <INDIVIDUAL>
        <ID>...</ID>
        <WRITING>
          <TITLE>...</TITLE><DATE>...</DATE><INFO>...</INFO><TEXT>...</TEXT>
        </WRITING>
        ....
      </INDIVIDUAL>
      Returns DateFrame with columns: ID TITLE DATE INFO TEXT
    '''
```

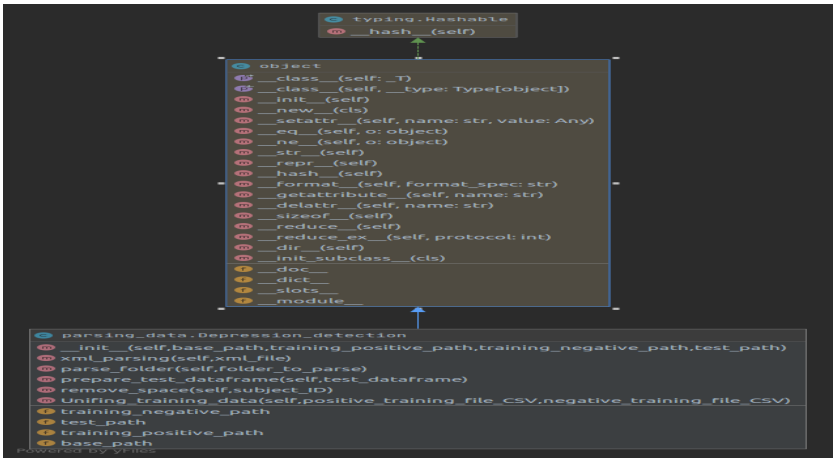

Figure 4: Data Pre-processing Class Diagram

```
    xtree = et.parse(xml_file)
    xroot = xtree.getroot()

    subject_id = xroot.find('ID').text
    writings = []
    for writing in xroot.findall('WRITING'):
        xml_data = {}
        xml_data['ID'] = subject_id
        xml_data['TITLE'] = writing.find('TITLE').text
        xml_data['DATE'] = writing.find('DATE').text
        xml_data['INFO'] = writing.find('INFO').text
        xml_data['TEXT'] = writing.find('TEXT').text
        writings.append(xml_data)
    df = pd.DataFrame(writings)
    return df
```

Listing 2: Parsing XML files

5 Data pre-processing

In this section, data will be preprocessed to feed it to our model. As shown in the following class diagram fig 4, there are 4 main functions:

- **Tokenization**: This function removes https links and punctuation, removes spaces and numbers, lowering the case, tokenize the whole

text and finally save them in a pickle file to use them later.We had two options to tokenize our posts. The first option is to tokenize it without using any libraries at all, just using splitting function. The second option is to use libraries which would be more effeicient as these libraries are more efficient and trained to do so.

```
1
2  def tokenization(self,unified_training_df,train=False):
3      '''
4      Preprocess the whole text ie.
5       * remove https links and punctuation
6       * remove spaces and numbers
7       * lowering the case and finally tokenize it.
8      :param unified_training_df:
9      :return: tokens
10     '''
11     #get punctuations from string
12     punctuations = string.punctuation
13     corpus = unified_training_df.TITLE_TEXT
14     corpus = corpus.to_list()
15
16     #Take each line of the subject and combine all of
       them in one big text(corpus)
17     text = []
18     for x in corpus:
19         text.append(x)
20
21     text_joined = ' '
22     text = text_joined.join(text)
23
24     #remove hhtps links and punctuations
25     text = re.sub(r'http\S+', '', text)
26     all_text = ''.join([c for c in text if c not in
       punctuations])
27
28     #Tokanize the whole text using spacy tokanizer
29     words = self.tokenizer(all_text)
30
31     #delete space and numbers and lower case all the
       tokens
32     tokens = [token.lower_ for token in words if not
       token.is_space and not token.like_num]
33
34     #save it to a pickle file
35     #--->Check code in github
```

```
    return tokens
```

Listing 3: Tokenization

- **Downsampling**: This function focuses on solving the imbalanced data problem, which is that our dataset has more positive training data than negative data. We had two options: either an oversampling technique by repeating negative data couple of times to equalise its size with the positive data or using a downsampling technique, in which we delete data from positive data. Oversampling will cause over-fitting and might decrease the accuracy due to repeated data. On the other hand, downsampling will lead to losing data and information from our dataset. To balance the data, we chose the technique of downsampling, because oversampling will cause over-fitting.

```
def downsampling(self,unified_training_df_preprocessed):
    '''
    downsampling majority(un)
    :param unified_training_df_preprocessed:
    :return:
    '''
    # separate minority and majority classes
    un_depressed = unified_training_df_preprocessed[unified_training_df_preprocessed.LABEL == 0]
    depressed = unified_training_df_preprocessed[unified_training_df_preprocessed.LABEL == 1]

    undepressed_downsampled = resample(un_depressed,
                                       replace=False,  # sample without replacement
                                       n_samples=len(depressed),  # match minority n
                                       random_state=27)  # reproducible results

    # combine minority and downsampled majority
    downsampled = pd.concat([undepressed_downsampled, depressed])

    # checking counts
    downsampled.LABEL.value_counts()

    #save it to csv file
```

```
    downsampled.to_csv('./downsampled_data.csv')
```

Listing 4: Downsampling

- **Encoding the words**: This function gives each token a unique integer. It is done by mapping each word to an integer using dictionaries. We are taking this step as embedding lookup in Pytorch requires an integer to the network. Embedding lookup or lookup table is embedding weight matrix, which consists of vectors for each word. This part will be discussed more in detail in the embedding layer section.

```
def vocab_to_int(self,tokens):

    '''
    Annotate all the vocabs in the text with a
   corresponding integer.
    :return: self.vocab_to_int
    '''

    ## Build a dictionary that maps words to integers
    counts = Counter(tokens)
    vocab = sorted(counts, key=counts.get, reverse=True)
    # print(vocab[:30])
    # 1 in enamurate means the dictionary will begin with
    one instead of 0
    self.vocab_to_int = {word: ii for ii, word in
   enumerate(vocab, 1)}

    return self.vocab_to_int
```

Listing 5: Vocab_to_integer

- **Padding sequence**: To train our model, we need all the inputs of the neural network to have the same shape and size, but in our dataset we have found that the posts and the comments of the subjects don't have the same length. This is why we had to create a function to deal with both short and very long posts or comments to make all of them have the same length.

  If we have some short posts, the function will pad them with zeros to create the same size of the sequence length we decide to use. Otherwise, if the posts of the subjects are too long the function will truncate

them to also have the same length as the sequence length.

```
def pad_features(self,text_integers, seq_length):
    ''' Return features of text_ints, where each text is
   padded with 0's
        or truncated to the input seq_length.
    '''
    ## implement function
    # getting the correct rows x cols shape
    features = np.zeros((len(text_integers), seq_length),
    dtype=int)
    # for each review, I grab that review
    for i, row in enumerate(text_integers):
        # this will take row and last columns and leave
   first columns with zeros (check last cell to
   understand) , then [:seq_length] will shrink long data

        features[i, -len(row):] = np.array(row)[:
   seq_length]
    #convert festures to numpy
    features = np.array(features)
    return features
```

Listing 6: Padding_features

After Encoding and padding features, we are now ready for the next step, which is feeding these features to our model.

6 Language Models

This section presents the two models implemented to detect depression from online posts. We will give a description of each implementation with examples of the text's representation used in each model.

6.1 RNN approach

6.1.1 Introduction

This section will give a general description on RNN(Recurrent Neural Network) Model, how we will implement the RNN model in Pytorch and how to use it in detecting depression.

6.1.2 RNN Model

As the main purpose of this report is to explain how we implemented RNN rather than explaining RNN itself, we will only give a general explanation about it. We can simply describe RNN as a normal feedforward neural network, but it only differs in that it has internal memory. As shown in figure 5, RNN has two inputs, it has the current input and the previous input, which it gets from the output of the previous cell. Usually the output is referred to as hidden state [15].

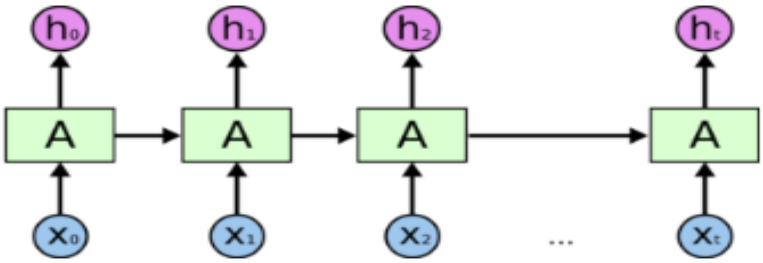

Figure 5: recurrent neural network [17]

In our project, we have decided to use LSTM(Long Short Term Memory), which is an extension of the normal(vanilla) RNN. We have found that RNN has a problem called vanishing gradient. It has been noticed that during backprobagation in RNN,the gradient (which is the update of neural network weights) shrinks through time. In this case the gradient value becomes too small and it doesn't contribute much to the learning, especially recurrent neural networks that get small gradient update [6]. That is why we have decided to use LSTM.

LSTM as shown in figure 6, briefly has two outputs instead of just one. One output is the cell state, which is the memory of the LSTM that has all the previous information, the other output is the hidden state which is the output of the cell [6].

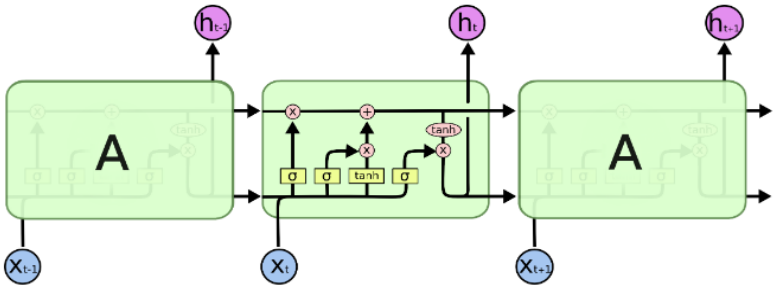

Figure 6: Long Short Term Memory [17]

6.1.3 Depression detection network architecture

As shown in our figure 7, our network architecture consists of 5 layers: Embedding layer, LSTM layer,Dropout layer, Linear layer and Sigmoid layer. We will discuss each layer in detail:

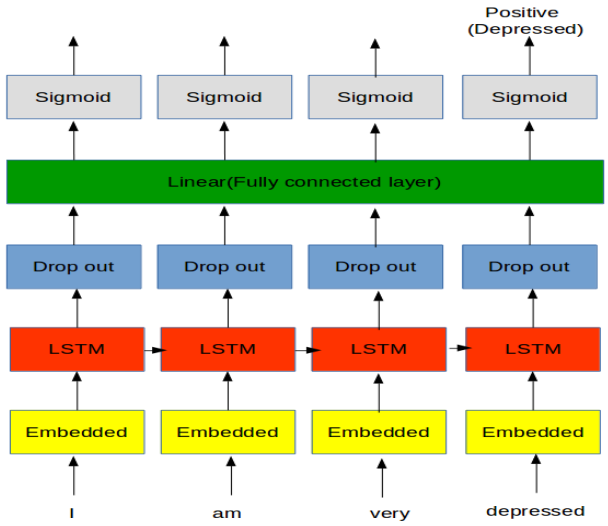

Figure 7: Depression detection network architecture

1. **Embedding Layer**:

   To train our model and pass the inputs to our RNN (LSTM), our features should be represented in vectors, as neural networks need a vector as an input. So we have thought of using one hot encoded vectors. However, one hot vector encoder has two disadvantages:

   - It contains ones and zeros only, which makes it difficult for the neural network to learn semantic representations.
   - Usually, if you a have large number of words in your vocab, it might not be a good idea to use one hot vector encoder, since you will have a matrix with high dimensionality, which will make the model train slowly.

   As we have seen these disadvantages, we have decided to use embedding layer to increase efficiency, reduce dimensionality and each vector has meaningful values(weights) to the model [10]

   To make it more clear, let us have a simple example. Let us assume that we have two posts from users:

   My day was very depressing

   Today was fine

   firstly we will tokenize these two sentences to be as following:

```
[My, day, was, very, depressing]
[Today, was, fine]
```

Listing 7: Tokenization example

   Next step is to give each word a unique number as well as applying padding to sentences(posts), assuming that sequence length = 5. We have to note that **was** token is repeated in both sentences(posts) so they will have the same unique integer.

```
[1, 2, 3, 4, 5]
[0, 0, 0, 6, 3, 7]
```

Listing 8: Encoding and Padding example

   Now we are ready to create our embedding weight matrix. Pytorch has made it easy for us to create these embedding weight matrix as it creates it and can fill and optimize these embedding vectors during

training phase. Alternatively, we can fill them from a pretrained model like Glove.

The expected size of this lookup table(embedding weight matrix) is (number of unique words+ 1 , embedding dimension), where the 1 added to the unique words is due to the zero padding. So in our example it will be (8,3), if we assume that the embedding dimension is 3.

```
[in] :  self.embedding = nn.Embedding(vocab_size,
    embedding_dim) #(8,3)


[out] : #Lookup table
index           Embedding
0             [2.5, 1.7, 1.9]
1             [2.9, 1.3, 1.1]
2             [1.9, 2.3, 2.1]
3             [1.1, 2.9, 2.5]
4             [1.8, 2.2, 2.8]
5             [3.8, 1.2, 3.2]
6             [1.8, 2.2, 2.4]
7             [4.2, 3.7, 2.4]
```

Listing 9: Embedding Layer

As shown in the lookup table, each word now has a unique index as well as an embedded vector.

2. **LSTM layer**:

   In this section, we will focus more on the programmatic part than the theoretical part, as we have discussed this part theoretically in the previous section.

   To create the LSTM layer, we have to give the function the following arguments. It takes the input size, the hidden dim, the number of lstm layers, dropout probability and setting batch first to true to get the following output dimensions (batch_size,sequence_length,input dimension). After creating the layer, we feed it with inputs, which are the embeddings and the hidden state.

```
self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
    dropout=drop_prob, batch_first=True)

```

```
lstm_out, hidden = self.lstm(embeds, hidden)
```

Listing 10: LSTM Layer

As we discussed earlier, we have hidden and cell states in LSTM and we need to initialize their weights at the beginning of each epoch or beginning of training. During our implementation we have tried two approaches. Firstly, we have tried to initialise the weights with zeros, then we tried to initialise the weights randomly. We have found that the accuracy has increased by 1% to 2% .

```
def init_hidden(self, batch_size):
    ''' Initializes hidden state '''
    # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
    # initialized to zero, for hidden state and cell state of LSTM
    weight = next(self.parameters()).data

    if (self.train_on_gpu):
        device = torch.device("cuda:0")  # Uncomment this to run on GPU
        hidden = (torch.randn(self.n_layers, batch_size, self.hidden_dim,device=device),
                  torch.randn(self.n_layers, batch_size, self.hidden_dim,device=device))
    return hidden
```

Listing 11: hidden weights

3. **Dropout layer**:

   We are using this layer in order to prevent overfitting. This happens because, while we are updating our weights during training, they become dependent on the dataset that we are using [16]. So, during training we dropout a certain value from our network as shown in figure 8. We have tried to dropout 30 percent and 50 percent; there was not much difference in accuracy between them.

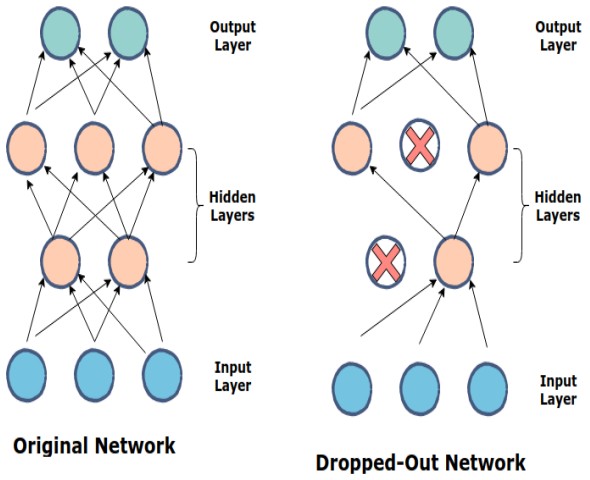

Figure 8: Dropout layer [19]

4. **Fully connected layer(Linear layer)**:

   This is used to map our LSTM layer output to the output size we need. This means that it takes the hidden dimensions that we have chosen as an input and output only one value.

```
(fc): Linear(in_features=256, out_features=1, bias=True)
```

Listing 12: fully connected layer(Linear layer)

5. **Sigmoid layer**:

   This will make all the output values we have either 0 or 1. As shown in figure 7, we are only interested in the last sigmoid output on the right as we are not expecting a sequence output from the other sigmoid outputs.

   NOTE: Please check our github link for the full network architecture code.

6.1.4 Training the network

Before starting the training process, we have to choose our loss and optimization functions and adjust our hyparameters.

6.1.4.1 Loss and optimization functions

1. **Loss function**:

   Loss functions is telling us how wrong our predictions are during our training process. In other words, we are using it to evaluate how well our algorithm models our data. If the loss is high, then we have bad accuracy or bad performance. In our project we are using **Binary Cross Entropy Loss(BCE loss)**. We are using BCE loss as we have a single output which is 0 or 1 as well as BCE loss in pytorch was designed to work with sigmoid outputs.

```
criterion = nn.BCELoss()
```

Listing 13: Loss function

2. **Optimizer function**:

   This function simply defines how neural network learn and by that we mean that it finds the values of parameters, such that the loss function is at its lowest value.

   In our project we have used Adam optimization, which is just an extension from Stochastic gradient decent. We have used it as one of the newest optimizers, as it has many benefits according to their authors [9]. It has efficient computations, requires little memory and works well for problems with very high noise gradients.

```
optimizer = torch.optim.Adam(RNN_net.parameters(), lr=lr)
    #lr is learning rate
```

Listing 14: Adam optimizer

6.1.4.2 Model hyperparameters

1. **Vocab size** : It defines the size of our input, which is the word tokens.
2. **Output size** : It is the number of outputs in our model, which is 1 in our case as we have one output, either zero or one.

3. **Embedding dimension** : It is the size of our embeddings, which is the number of columns in an embedding vector which we have explained earlier. It is also known as the number of columns in embedding lookup table. It usually has a value of 200, 300 or 400.

4. **hidden dimension** : In our LSTM cells, we have a certain number of units in the hidden layers. These values could be 128, 256 or 512. Sometimes the more the dimension increases, the more time it takes to train the model, the better performance we get.

5. **number of layers** : It is the number of the LSTM layers in a network. It is usually between 1 and 3 layers.

6. **learning rate** : It is the step size at each iteration and, as shown in the optimization section, learning rate is a tuning parameter in the optimization algorithm.

7. **epochs** : It is the number of iterations through the whole training dataset.

```
# Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_ints_training)+1 # +1 for the 0
    padding + our word tokens
output_size = 1
embedding_dim = 200
hidden_dim = 256 #128
n_layers = 1 #2
lr = 0.0001 #0.01
epochs = 5  #10

```

Listing 15: Model hyperparameters

6.1.4.3 Main steps of training the model

1. At the beginning of each epoch during training, we will call the function that initializes the hidden state.

```
h = RNN_net.init_hidden(batch_size)
```

Listing 16: hidden state initialization

2. At the beginning of the batch loop, we will try to clear all the old gradients we have from the last step. If we didn't do this step, all the gradients would be accumulated from the step **loss.backward()** call, which we will discuss in further steps. Hence, in this step we will zero the accumulated gradients.

```
    # zero accumulated gradients
    RNN_net.zero_grad()
```

3. We will perform feed forward propagation by passing the inputs to the model to get predicted outputs.

```
# get the output from the model
output, h = RNN_net(inputs, h)
```

Listing 17: Feedforward propagation

4. In this step, we will calculate the loss between the predicted output and the actual output. After calculating the loss we will backpropagate to compute the derivatives of the loss.

```
 # calculate the loss and perform backprop
loss = criterion(output.squeeze(), labels.float())
loss.backward()
```

Listing 18: Loss calculation and perform backpropagation

5. In this step, we will update our parameters by performing a single optimization step.

```
optimizer.step()
```

Listing 19: Optimizer step

6. Lastly, our model will be saved under one condition: the validation loss is decreasing. If the past validation loss value is more than the newest validation value, we will not save the model. This last step helps us to the stop our model from overfitting, as during training the training loss decreases aside from the validation loss, but at one breaking point the validation loss starts to increase again as shown in figure 9.

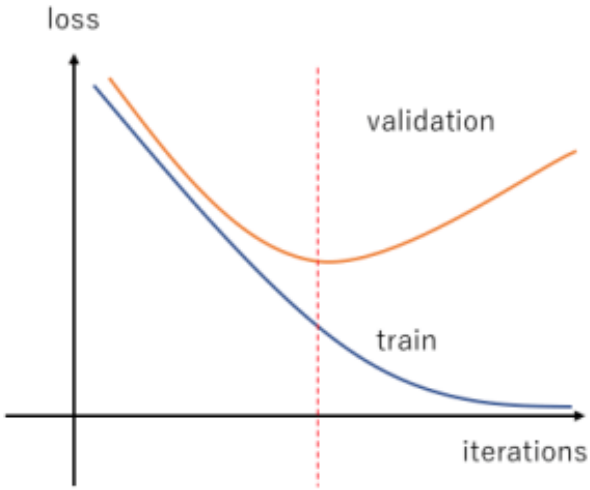

Figure 9: Train loss vs validation loss [13]

```
if np.mean(val_losses) <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}).
     Saving model ...'.format(
        valid_loss_min,
        np.mean(val_losses)))
    torch.save(RNN_net.state_dict(), '
    model_trained_RNN_not_pretrained.pt')
    valid_loss_min = np.mean(val_losses)
```

Listing 20: Saving Model

6.1.4.4 Cross validation

In order to get a good evaluation of the model and evaluate its performance, we need to split our training data into folds as shown in figure 10. Each time we split our data, there should be a fold that is not fed into the training dataset, but fed into our validation dataset to evaluate our model. We thus try to train the model each time on different folds to see how accurate it is while working on unseen data, as well as trying to avoid overfitting or underfitting.

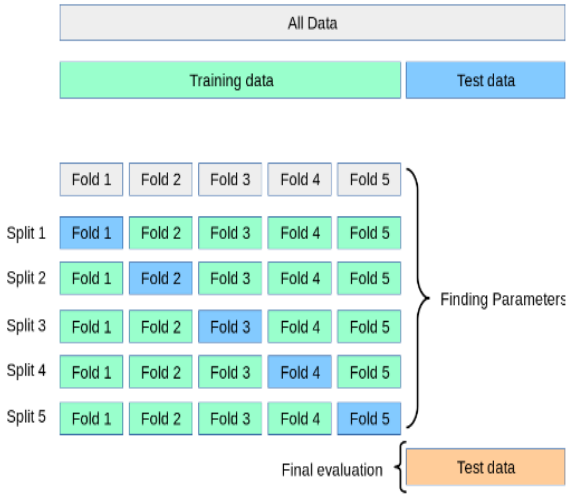

Figure 10: cross validation [3]

6.1.4.5 Results and observations

As we have discussed earlier, we have trained our model using cross validation using different splits for training and validation. As shown in the following table 1,we got an accuracy with around 74 % during training, but when we have tried to test our model on unseen date we get an accuracy of 64.4 %. As shown in the table 1 the second split has higher accuracy on unseen data. The model is trained on the following hyperparameters:

```
output_size = 1
embedding_dim = 200
hidden_dim = 256
n_layers = 1
lr = 0.0001
epochs = 5
```

Listing 21: CV Hyperparameters

Epochs	Cross Validation									
	Split 1		Split 2		Split 3		Split 4		Split 5	
	Accuracy during training	Accuracy on unseen data(test data)	Accuracy during training	Accuracy on unseen data(test data)	Accuracy during training	Accuracy on unseen data(test data)	Accuracy during training	Accuracy on unseen data(test data)	Accuracy during training	Accuracy on unseen data(test data)
1	54%		55%		57%		56%		53%	
2	63.9%		65%		63%		62.3%		63.2%	
3	65.1%	60.8%	74.6%	64.4%	72.1%	60%	73%	61%	64.5%	59.4%
4	66%		74.7%		72.3%		73.2%		66.2%	
5	66.9%		74.7%		72.4%		73.2%		66.3%	

Table 1: Data trained on different splits (Cross Validation)

While training the model, we have first tried to train the model on 10 epochs. What we noticed is that both training loss and validation loss decrease till epoch 5. After that, the model starts to overfit and the validation loss starts to increase again. So we have decided to train the model with 5 epochs only in order to avoid overfitting. Please refer to figure 9.
We have tried to increase the embedding layers from 200 to 300 as well as increasing the hidden dimensions from 256 to 512 besides increasing the number of layers from 1 to 2. We have noticed that as we increase the hidden dimensions and the number of layers, the time the model takes for training almost doubled. The accuracy didn't change that much, we have reached an accuracy of 67% during training and around 59% on unseen test data. We have tried also to increase the number of layers to 3 and the embedding dimensions to 400, we got an accuracy of 75% during training and 62% on unseen test data.A summery of our training can be found in the following table 2.

	Embedding dimensions = 200 , Hidden dimension = 256, Number of layer = 1, Epoch = 0.001	Embedding dimensions = 300 , Hidden dimension = 512, Number of layer = 2, Epoch = 0.001	Embedding dimensions = 400 , Hidden dimension = 512, Number of layer = 3, Epoch = 0.001
During training accuracy	74%	67%	75%
Testing on unseen data accuracy	64.4%	59%	62%

Table 2: (Accuracy Comparison)

We have also tried to train our network using pretrained weights from a pretrained model. We have used Glove pretrained word vectors, which offers pretrained weights based on tokens from twitter. To set the model to train from the pretrained model, we first have to only look for the tokens that we have in our vocab within Glove tokens and extract the weights of our tokens only. Then we have to call the pretrained method from pytorch during our training.

```
elif pretrained == True:
    num_embeddings, embedding_dim = weights_matrix.shape
    self.embedding = nn.Embedding(num_embeddings,
    embedding_dim)
    self.embedding.load_state_dict({'weight': weights_matrix
    })
```

Listing 22: Pretrained model

We have trained the model with pretrained weights using the same hyperparameters in listing 21 .Surprisingly using pretrained model didn't change much in the accuracy, as we got around 70 % accuracy during training and around 60% during testing it on unseen data.

6.1.5 Next steps regarding RNN

We would be interested to enhance our model accuracy using RNN, we will try different hyperparameters, maybe we can try to replace LSTM with GRU instead . We have tried also to use Skorch which is a library that allows us to use Pytorch with Sklearn . One of the advantages of this library that during training it applies different hyperparameters randomly as well as it train on GRU and LSTM both together. But unfortunately it gives shows the highest accuracy without stating which hyperparameters gave this accuracy. We would like to invest more time on using this library as it might help a lot. We got an accuracy of 69% using Skorch.

6.2 Bert approach

6.2.1 Introduction

This section will give a general description of the Bidirectional Encoder Representations from Transformers (BERT) language model's operation, its main

components, and its implementation for detection of depression. This description will indicate how a better understanding of the language model functioning could contribute to enhanced implementations.

6.2.2 Bert Model

BERT is a language model developed by Google and designed to carry out different tasks. The base model was trained on Wikipedia and the Google Books Corpora; both data sets sum up to 3500 million words. This amount of data gives BERT the flexibility for a broad range of applications. BERT requires fine-tuning to perform specific tasks, such as question answering, natural language understanding, sentiment analysis, and depression detection.

Transformers are BERT's base architecture; they replace recurring layers (LSTM) with attention layers [18]. Attention layers encode each sequence's token based on its contextual relation with other sequence's tokens. This encoding introduces the context of each token in the numerical representation of whole sequence. For this type of encoding, the transformers models are also called contextual embeddings. Transformers models also generate positional embedding, which, like the contextual embedding, allows knowing each token's relative position in the encoded text sequence.

Transformer-based models' implementation consists of two phases. The first phase is called pre-training [1]. In this phase, the model learns language structures and generic knowledge of each word's meaning. The learning occurs through "exercises", in which the model has to predict a word or words omitted in a sentence. In our project, we skipped the pre-training phase using BERT, which has been pre-trained in two large databases, as it has been mentioned.

Next, we will explain the general operation of BERT:

6.2.3 Tokenization with Bert

```
1 PRE_TRAINED_MODEL_NAME="bert-base-cased"
```

```
tokenizer=BertTokenizer.from_pretrained(
    PRE_TRAINED_MODEL_NAME)
```

Listing 23: Pre-trained Model

BERT base-cased model requires its input as a sequence divided into two parts. The first part of the sequence will have the original text, and the second part will have pad tokens that indicate it is empty. After BERT's pre-training, the first token of the sequence condenses the whole sequence representation, for the tokenizer class uses the pre-trained model, which has 100 million parameters and differentiates between upper and lower case written words. The following example will show the tokenization process in a text sample:

```
txt_sample = """I am sick of the Hall and the hill, I am sick
     of the moor and the main. Why should I stay? can a
    sweeter chance ever come to me here? O, having the nerves
    of motion as well as the nerves of pain, Were it not wise
    if I fled from the place and the pit and the fear?"""

# The Valley, (Part One, I, xvi) (Talonbooks, 2014), by Joan
    MacLeod.
```

Listing 24: Text Sample

Since the Reddit posts have a confidentiality agreement to respect users' privacy, a fragment of the novel the Valley by Joan MacLeod has been taken as an example.
By using the tokenizer in the sample text, the following tokens are obtained:

```
txt_token = tokenizer.tokenize(txt_sample)
print('Tokens: ', txt_token)

Tokens:  ['I', 'am', 'sick', 'of', 'the', 'Hall', 'and', 'the
    ', 'hill', ',', 'I', 'am', 'sick', 'of', 'the', 'm', '##
    oor', 'and', 'the', 'main', '.', 'Why', 'should', 'I', '
    stay', '?', 'can', 'a', 'sweet', '##er', 'chance', 'ever',
     'come', 'to', 'me', 'here', '?', 'O', ',', 'having', 'the
    ', 'nerves', 'of', 'motion', 'as', 'well', 'as', 'the', '
    nerves', 'of', 'pain', ',', 'Were', 'it', 'not', 'wise', '
    if', 'I', 'fled', 'from', 'the', 'place', 'and', 'the', '
    pit', 'and', 'the', 'fear', '?']
```

Listing 25: Tokens

Tokenization shows difficulty for a psychological interpretation of depression detection based on the documents' representation. As it can be seen, in tokenized text, the text's semantic structure is altered by including masked tokens like 'm', '##oor', '##er'. These tokens give advantages for the text's processing, since the their representation allows generalizing and better performance, as explained in BERT's launching paper. "The masked language model [MLM] randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context. Unlike left-to-right language model pre-training, the MLM objective allows the representation to fuse the left and the right context, which allows us to pre-train a deep bidirectional Transformer." [4]. The tokenization benefits the implementation of the transformers' architecture, but shuns meaningful psychological research clues. It should be investigated if the original text's semantic structure changes follow any regularity according to the type of tokens that are altered. Transformers language models give promising results, but these come at the cost of limited or none psychological interpretability.

After the tokenization, each token received a numerical assignment. The numeration comes from the pre-trained model, and is necessary because neural networks cannot handle textual inputs.

```
txt_token_ids = tokenizer.convert_tokens_to_ids(txt_token)
print('Token IDS: ', txt_token_ids)

Token IDS:  [146, 1821, 4809, 1104, 1103, 1944, 1105, 1103,
   4665, 117, 146, 1821, 4809, 1104, 1103, 182, 15626, 1105,
   1103, 1514, 119, 2009, 1431, 146, 2215, 136, 1169, 170,
   4105, 1200, 2640, 1518, 1435, 1106, 1143, 1303, 136, 152,
   117, 1515, 1103, 10846, 1104, 4018, 1112, 1218, 1112,
   1103, 10846, 1104, 2489, 117, 8640, 1122, 1136, 10228,
   1191, 146, 6192, 1121, 1103, 1282, 1105, 1103, 7172, 1105,
    1103, 2945, 136]
```

Listing 26: Tokens' numerical representation

The TokenizerClass, imported with BERT pre-trained model, has a function to encode the text to be processed by the model's transformers networks. The following function shows an implementation of encoding for the model re-training. This function takes as parameters: `txt_sample`, corresponding to the sample text seen before. In the model's implementation for depression detection, this parameter will take users' Reddit posts. `max_length` takes a

value of 100 tokens, in this example and the implementation for detecting depression. The following five boolean parameters: activate the truncation of the sequences, add of special tokens, leave the identifying token types, activate the padding, and returns the attention mask. The last parameter return the tensors.

```
encoding = tokenizer.encode_plus(
    txt_sample,
    max_length = 100,
    truncation = True,
    add_special_tokens = True,
    return_token_type_ids = False,
    pad_to_max_length = True,
    return_attention_mask= True,
    return_tensors= 'pt')
```

Listing 27: Encoding Functions

Checking the encoded text, we have the representation that BERT requires: a sequence with a special token at the first position '[CLS]' followed by the texts' token, a delimiting token '[SEP]' that closes the first part of the sequence. Next to this token, we have the second part of the sequence that will receive the padding token '[PAD]'. After the re-training, the '[CLS]' token will store the entire document's representation for classification.

```
print(tokenizer.convert_ids_to_tokens(encoding['input_ids'
    ][0]))
['[CLS]', 'I', 'am', 'sick', 'of', 'the', 'Hall', 'and', 'the
    ', 'hill', ',', 'I', 'am', 'sick', 'of', 'the', 'm', '##
    oor', 'and', 'the', 'main', '.', 'Why', 'should', 'I', '
    stay', '?', 'can', 'a', 'sweet', '##er', 'chance', 'ever',
     'come', 'to', 'me', 'here', '?', 'O', ',', 'having', 'the
    ', 'nerves', 'of', 'motion', 'as', 'well', 'as', 'the', '
    nerves', 'of', 'pain', ',', 'Were', 'it', 'not', 'wise', '
    if', 'I', 'fled', 'from', 'the', 'place', 'and', 'the', '
    pit', 'and', 'the', 'fear', '?', '[SEP]', '[PAD]', '[PAD]'
    , '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[
    PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]
    ', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '
    [PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD
    ]', '[PAD]']
```

Listing 28: Encoded Text Sample

In the tokens' numerical assignment, we see that the tokenizer assigns to the tokens '[CLS]' , '[SEP]' , and '[PAD]' the numbers 101, 102, and 0, respectively. The other numbers are also assigned according to the pre-trained model's values.

```
print(encoding['input_ids'][0])

tensor([  101,   146,  1821,  4809,  1104,  1103,  1944,  1105,  1103,  4665,
         117,   146,  1821,  4809,  1104,  1103,   182, 15626,  1105,  1103,
        1514,   119,  2009,  1431,   146,  2215,   136,  1169,   170,  4105,
        1200,  2640,  1518,  1435,  1106,  1143,  1303,   136,   152,   117,
        1515,  1103, 10846,  1104,  4018,  1112,  1218,  1112,  1103, 10846,
        1104,  2489,   117,  8640,  1122,  1136, 10228,  1191,   146,  6192,
        1121,  1103,  1282,  1105,  1103,  7172,  1105,  1103,  2945,   136,
         102,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0])
```

Listing 29: Text Sample Numerical Representation

Finally, to close the encoding, we have the attention mask. In this mask the significant tokens receive a one and the padding tokens a zero. BERT will use this mask to give more relevance to the significant tokens in the sequence.

```
print(encoding['attention_mask'][0])
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0,
```

```
6           0, 0, 0, 0])
```

Listing 30: Attention Mask

6.2.4 BertClassifier

6.2.4.1 Bert Hyper-parameters

The BERT setup used for depression detection will now be briefly described.

```
1 RANDOM_SEED = 42
2 MAX_LEN = 50
3 BATCH_SIZE = 16
4 CLASSES = 2
```

Listing 31: Hyper-parameters for BERT

The seed number allows a model's stable execution over multiple training sessions. It initializes the weights of the neural networks randomly, but at the same initial values. The next parameter is the maximum length. This value will limit the length of text that the BERT model will process for each observation. Since the computing resources are limited, the text is truncated to max length. This truncation implies that the model has to detect the depression at the beginning of each text sample. The batch size will be the number of examples that the model will process simultaneously. Again, computing resources limits batch size range of values. Finally, the number of classes corresponds to 2, depressed with label 1, and not depressed with label 0.
We define the `BERTClassifier` for the implementation of the model. We derive this class from nn.Module (e.i Neuronal Network Module) from PyTorch. In initializing this class, we take the pre-trained BERT model, and add to it a Dropout and Linear layer. The Linear layer takes as arguments `self.bert.config.hidden_size` and `n_classes`, which correspond to the number of inputs and the number of outputs, respectively. The number of neurons at the output of BERT is equal to 768. This number will be the value of inputs for the Linear layer that we add to the pre-trained model for the classification. The `n_classes` corresponds to 2 because we are classifying two states, depressed or not. To reduce overfitting, i.e. the training and validation precision obtain a similar value, we have included a Dropout layer of 0.3. This layer will randomly turn off 30% of the neurons during the model's

training. The objective is a better generalization. In the second class' function, we define how these layers are connected. This function needs two inputs: the `inputs_ids` (the numerical representation of the tokens), and the `attention_mask`, which focuses BERT's attention. The first layer of our classifier will be the BERT model, which receives the `inputs_ids` and `attention_mask` as inputs. From the model outputs, we take `cls_output` that corresponds to the classification token. This output will go through the Dropout layer, which will finally be processed by the Linear layer. The Linear layer will give us the classification.

```
class BERTClassifier (nn.Module):

  def __init__ (self, n_classes):
    super(BERTClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(
   PRE_TRAINED_MODEL_NAME)
    self.drop = nn.Dropout(p=0.3)
    self.linear = nn.Linear(self.bert.config.hidden_size,
   n_classes)

  def forward( self, input_ids, attention_mask):
    _, cls_output = self.bert(
        input_ids = input_ids,
        attention_mask = attention_mask)
    drop_out = self.drop(cls_output)
    output = self.linear(drop_out)
    return output
```

Listing 32: Bert Classifier

6.2.4.2 Bert Training and Validation Functions

For the training presented in this report, we used 5 epochs. We optimized our implementation with AdamW, an algorithm similar to the descending gradient. This optimizer receives the model parameters and the learning rate, also to avoid it modifying the model parameters, we select False for its `correct_bias` argument. The number of steps is equal to the number of batches delivered by the dataloader function multiplied by the number of epochs. The scheduler is responsible for reducing the learning range in each of the iterations. The scheduler arguments are the optimizer, the num_warmup_steps equal to zero because we want the learning rate to decrease from the first iteration, and the num_training_steps, which is equal to

the total of `total_steps`. Finally, we define the loss function, which we take from the `nn.CrossEntropyLoss()` module.

```
EPOCHS = 5
optimizer = AdamW(model.parameters(), lr=2e-5,
                  correct_bias = False)
total_steps = len(train_data_loader)* EPOCHS
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps= 0,
    num_training_steps= total_steps)
loss_fn = nn.CrossEntropyLoss().to(device)
```

Listing 33: Training

For training we define the `train_model` function. In this function we set the model on training mode, initialize the `losses` list to store the error. Also we assign zero to (`correct_predictions`) of zero. Then, for each batch that the dataloader delivers, the function sends the `input_ids`, the `attention_mask` `labels` to the GPU, which are then assigned as arguments to the model to obtain `outputs`. In `preds`, they correspond to the predictions calculated by BERT, the function uses torch.max to store the maximum value. The model has two outputs after being adjusted for depression detection. So, if the first output is maximum, it means that the person suffers from depression. If the second output is maximum, it means that the person does not have depression. After making the predictions, the function calculates the loss using the output and the original label. It also calculates the precision, which corresponds to the accumulated sum of the original labels' correct predictions. This data is stored in `losses`. Next, the function takes the value of the loss and executes backpropagation to update the weights. The function uses `nn.utils.clip_grad_norm(model.parameters(), max_norm = 1.0)` to cut the gradient value preventing the gradient from stagnating. Finally, the function updates the weights, the training rate, and reset the optimizer weights for the next batch process. The training function outputs the average value of the precision and the average value of the loss.

```
def train_model(model, data_loader, loss_fn, optimizer,
    device, scheduler, n_examples):
    model = model.train()
    losses = []
    correct_predictions = 0
    for batch in data_loader:
```

```
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)
        outputs = model(input_ids = input_ids, attention_mask
    = attention_mask)
        _, preds = torch.max(outputs, dim=1)
        loss = loss_fn(outputs, labels)
        correct_predictions += torch.sum(preds == labels)
        losses.append(loss.item())
        loss.backward()
        nn.utils.clip_grad_norm(model.parameters(), max_norm
    = 1.0)
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()
    return correct_predictions.double()/n_examples, np.mean(
    losses)
```

Listing 34: Training Function

We define the `eval_model` function for the validation. This function has a similar structure to the `train_model` function. However, instead of initializing the model in training mode, the function takes the re-trained model and initializes it in evaluation mode. Furthermore, after processing each batch, the function does not modify the model weights; instead, it only returns the average value of the precision and the average value of the loss.

```
def eval_model(model, data_loader, loss_fn, device,
    n_examples):
    model = model.eval()
    losses = []
    correct_predictions = 0
    with torch.no_grad():
        for batch in data_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(
    device)
            labels = batch['label'].to(device)
            outputs = model(input_ids = input_ids,
    attention_mask = attention_mask)
            _, preds = torch.max(outputs, dim=1)
            loss = loss_fn(outputs, labels)
            correct_predictions += torch.sum(preds == labels)
            losses.append(loss.item())
```

```
    return correct_predictions.double()/n_examples, np.mean(losses)
```

Listing 35: Validation Function

6.2.4.3 Training and Results

As shown in the following snip of code, after five iteration with BERT we obtained an accuracy 94% during training and 71% on validation.

```
for epoch in range(EPOCHS):
    print('Epoch {} from {}'.format(epoch+1, EPOCHS))
    print('----------------------------')
    train_acc, train_loss = train_model(
        model,
        train_data_loader,
        loss_fn,
        optimizer,
        device,
        scheduler,
        len(df_train) )
    test_acc, test_loss = eval_model(
        model,
        test_data_loader,
        loss_fn, device,
        len(df_test))
    print('Training: Loss {}, accuracy: {}'.format(train_loss, train_acc))
    print('Validation: Loss {}, accuracy: {}'.format(test_loss, test_acc))
```

Listing 36: Training

```
Epoch 1 from 5
----------------------------
Training: Loss 0.4908973435473551, accuracy:
   0.7506917317708334
Validation: Loss 0.6028436693401696, accuracy:
   0.7157389322916666
Epoch 2 from 5
----------------------------
Training: Loss 0.36380973518741183, accuracy:
   0.83880615234375
Validation: Loss 0.768554817569869, accuracy: 0.71435546875
Epoch 3 from 5
```

```
----------------------------
Training: Loss 0.26638933907906903, accuracy:
    0.9001057942708334
Validation: Loss 1.0772863906270989, accuracy:
    0.7112630208333334
Epoch 4 from 5
----------------------------
Training: Loss 0.21492615004043122, accuracy: 0.93115234375
Validation: Loss 1.1942933631016786, accuracy:
    0.7135416666666666
Epoch 5 from 5
----------------------------
Training: Loss 0.19189651082607875, accuracy:
    0.9412638346354166
Validation: Loss 1.1942933631016786, accuracy:
    0.7135416666666666
```

Listing 37: Results

6.2.5 Insights gained

Collaborating in implementing the RNN and BERT models to detect depression has given us a satisfactory learning experience, especially for interdisciplinary work. During the implementation, we reinforced and expanded our knowledge of programming. We also learn and apply NLP concepts to understand how different written language representations are necessary for language processing. We recognized that those language representations have advantages for efficient processing, but they limit interpretation, which social sciences commonly require. Despite these limitations, we recognize interdisciplinary work's potential to create communication channels between experts from different disciplines. Thus we recommend more opportunities to develop interdisciplinary work.

Listings

1 Parsing Folders . . . 7
2 Parsing XML files . . . 7
3 Tokenization . . . 9
4 Downsampling . . . 10
5 Vocab_to_integer . . . 11
6 Padding_features . . . 12
7 Tokenization example . . . 15
8 Encoding and Padding example . . . 15
9 Embedding Layer . . . 16
10 LSTM Layer . . . 16
11 hidden weights . . . 17
12 fully connected layer(Linear layer) . . . 18
13 Loss function . . . 19
14 Adam optimizer . . . 19
15 Model hyperparameters . . . 20
16 hidden state initialization . . . 20
17 Feedforward propagation . . . 21
18 Loss calculation and perform backpropagation . . . 21
19 Optimizer step . . . 21
20 Saving Model . . . 22
21 CV Hyperparameters . . . 23
22 Pretrained model . . . 25
23 Pre-trained Model . . . 26
24 Text Sample . . . 27
25 Tokens . . . 27
26 Tokens' numerical representation . . . 28
27 Encoding Functions . . . 29
28 Encoded Text Sample . . . 29
29 Text Sample Numerical Representation . . . 30
30 Attention Mask . . . 30
31 Hyper-parameters for BERT . . . 31
32 Bert Classifier . . . 32
33 Training . . . 33
34 Training Function . . . 33
35 Validation Function . . . 34
36 Training . . . 35

37 Results 35

References

[1] Jay Alammar. "The Illustrated BERT, ELMo, and co.(How NLP Cracked Transfer Learning)". In: *Dec* 3 (2018), pp. 1–18.

[2] Hayda Almeida, Antoine Briand, and Marie-Jean Meurs. "Detecting Early Risk of Depression from Social Media User-generated Content." In: *CLEF (Working Notes)*. 2017.

[3] *cross validation, URL= https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-evaluating-estimator-performance, Accessed = 2020-12-16.*

[4] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[5] Lina Gega, Isaac Marks, and David Mataix-Cols. "Computer-aided CBT self-help for anxiety and depressive disorders: Experience of a London clinic and future directions". In: *Journal of clinical psychology* 60.2 (2004), pp. 147–157.

[6] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[7] Spencer L James et al. "Global, regional, and national incidence, prevalence, and years lived with disability for 354 diseases and injuries for 195 countries and territories, 1990–2017: a systematic analysis for the Global Burden of Disease Study 2017". In: *The Lancet* 392.10159 (2018), pp. 1789–1858.

[8] Simon Kemp. *Data-Reportal*. 2020. URL: https://datareportal.com/reports/digital-2020-global-digital-overviewl (visited on 03/30/2020).

[9] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[10] Omer Levy and Yoav Goldberg. "Dependency-based word embeddings". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2014, pp. 302–308.

[11] Isaac Marks and Kate Cavanagh. "Computer-aided psychological treatments: evolving issues". In: *Annual review of clinical psychology* 5 (2009), pp. 121–141.

[12] Health Quality Ontario. "Psychotherapy for Major Depressive Disorder and Generalized Anxiety Disorder: A Health Technology Assessment". eng. In: *Ontario health technology assessment series* 17.15 (2017). PMC5709536[pmcid], pp. 1–167. ISSN: 1915-7398. URL: https://pubmed.ncbi.nlm.nih.gov/29213344.

[13] *Overfitting isn't simple: Overfitting Re-explained with Priors, Biases, and No Free Lunch, URL= https://mlexplained.com/2018/04/24/overfitting-isnt-simple-overfitting-re-explained-with-priors-biases-and-no-free-lunch/, Accessed = 2020-12-16.*

[14] Philip Resnik, Rebecca Resnik, and Margaret Mitchell. "Proceedings of the Workshop on Computational Linguistics and Clinical Psychology: From Linguistic Signal to Clinical Reality". In: *Proceedings of the Workshop on Computational Linguistics and Clinical Psychology: From Linguistic Signal to Clinical Reality*. 2014.

[15] Alex Sherstinsky. "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network". In: *Physica D: Nonlinear Phenomena* 404 (2020), p. 132306.

[16] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[17] *Understanding LSTM Networks, URL= http://colah.github.io/posts/2015-08-Understanding-LSTMs/, Accessed = 2020-12-15.*

[18] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017), pp. 5998–6008.

[19] *What is dropout in neural networks?, URL= https://www.educative.io/edpresso/what-is-dropout-in-neural-networks, Accessed = 2020-12-16.*

[20] WHO. *WHO,Depression*. 2020. URL: https://www.who.int/news-room/fact-sheets/detail/depression.