THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

DATS-6312 NLP for Data Science

Fake and Real News Dataset

Prof. Amir Jafari

Individual Report

Greeshmanjali Bandlamudi

G28076394

Table of Contents

1.Introduction:

The goal of our project is to classify real and fake news on the internet. As the scourge of "fake news" continues to plague our information environment, attention has turned toward devising automated solutions for detecting problematic online content. But, to build reliable algorithms for flagging "fake news," we will need to go beyond broad definitions of the concept and identify distinguishing features that are specific enough for machine learning. We consume news through several mediums throughout the day in our daily routine, but sometimes it becomes difficult to decide which one is fake and which one is authentic. In this project my participation was data preprocessing and implemented Roberta model

2.Dataset Description:

The first file, "True.csv," contains more than 12,600 reuter.com articles. The second file, "False.csv," has almost 12,600 items culled from various fake news sources. The following information is included in each article: the title, the text, the type, and the date the article was published. We focused on gathering articles from 2016 to 2017 to match the false news data acquired for kaggle.com. The information gathered was cleaned and processed, but the punctuation and errors found in the bogus news were left in the text. The real news contains 21417 articles, and we have two types in that one is world-news with article size of 10145 and the other is political news with 11272. The fake news contains 23481 articles, and we have government-news with article size 1570, middle east with size 778, US news with 783 sizes, left-news has 4459 sizes, politics has 6841, news has 9050 article size.

3.Data Preprocessing:

Step 1: Removing URL

```
def remove_url(text):
    return re.sub(r'http\S+', '', text)
```

Step 2: Applying Lower

```
def to_lower(text):
    return text.lower()
```

Step 3: Applying Contractions

```
def remove_contractions(text):
    return ' '.join([contractions.fix(word) for word in text.split()])
```

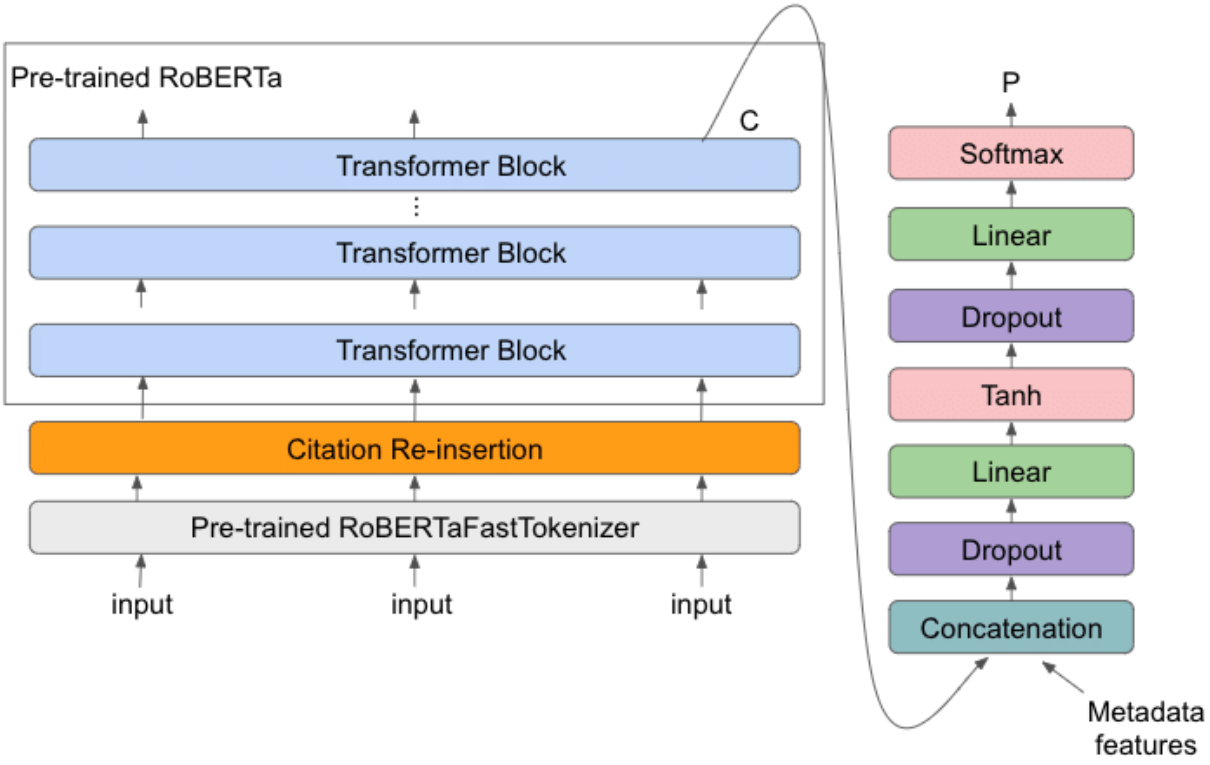
Step 3: Removing Punctuations

```
def remove_punctuations(text):
    return re.sub('\[[^]]*\]', '', text)
```

4.Network and Model:

RoBERTa:

RoBERTa builds on BERT's language masking method, which teaches the system to predict purposely hidden content within otherwise unannotated language instances. RoBERTa modifies critical hyperparameters in BERT, such as deleting BERT's next-sentence pretraining target and training with considerably bigger mini-batches and learning rates, which was implemented in PyTorch. As a result, RoBERTa outperforms BERT on the masked language modeling objective, resulting in improved downstream task performance. We also look into training RoBERTa on an order of magnitude more data and over a longer period of time than BERT.




```
import numpy as np
import pandas as pd
from sklearn import metrics
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import TensorDataset, DataLoader, SequentialSampler, RandomSampler
import sklearn
from sklearn.metrics import classification_report, accuracy_score, roc_auc_score
from transformers import RobertaForSequenceClassification
from transformers import RobertaTokenizer
from transformers import get_linear_schedule_with_warmup
from transformers import AdamW
import os
# Loading the dataset
path1 = os.path.split(os.getcwd())[0] + '/project_datasets'
np.random.seed(0)
df = pd.read_csv(path1+'/'+'new_df.csv')
# checking for null values
print(df.isnull().sum())
# dropped the null values if present
df.dropna(inplace=True)
df.drop(['Unnamed: 0'],axis=1,inplace=True)
# split the dataset to train, validation, test
train_df, sub_df = train_test_split(df, stratify=df.target.values, random_state=1, test_size=0.2, shuffle=True)
validation_df, test_df = train_test_split(sub_df, stratify=sub_df.target.values, random_state=1, test_size=0.25,
shuffle=True)
train_df.reset_index(drop=True, inplace=True)
validation_df.reset_index(drop=True, inplace=True)
test_df.reset_index(drop=True, inplace=True)
```

```
# We check the number of examples after split
print("Train data: {} \n".format(train_df.shape))
print("Validation data: {} \n".format(validation_df.shape))
print("Test data: {} \n".format(test_df.shape))
# RoBERTa
checkpoint = "roberta-base"
tokenizer = RobertaTokenizer.from_pretrained(checkpoint, do_lower_case=True)
NUM_LABELS = 2
BATCH_SIZE = 32
MAX_LEN = 256
N_EPOCHS = 3
LEARNING_RATE = 1e-5
train = tokenizer(list(train_df.text.values), truncation=True, padding=True, max_length=MAX_LEN)
train_input_ids = train['input_ids']
train_masks = train['attention_mask']

validation = tokenizer(list(validation_df.text.values), truncation=True, padding=True, max_length=MAX_LEN)
validation_input_ids = validation['input_ids']
validation_masks = validation['attention_mask']
#TO TENSOR
train_inputs = torch.tensor(train_input_ids)
validation_inputs = torch.tensor(validation_input_ids)
train_labels = torch.tensor(train_df.target.values)

validation_labels = torch.tensor(validation_df.target.values)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)
# DataLoader for our training set.
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=BATCH_SIZE)
# DataLoader for our validation set.
validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation_sampler, batch_size=BATCH_SIZE)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device {}.\n".format(device))
model = RobertaForSequenceClassification.from_pretrained(checkpoint, num_labels=NUM_LABELS,
output_hidden_states=False, output_attentions=False)
model = model.to(device)
optimizer = AdamW(model.parameters(),lr=LEARNING_RATE)
# Total number of training steps is number of batches * number of epochs.
total_steps = len(train_dataloader) * N_EPOCHS
# Create the learning rate scheduler.
lr_scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps = 0, num_training_steps =
total_steps)
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
num_params = count_parameters(model)
print("The model has total {} trainable parameters".format(num_params))
def eval_metric(predictions, labels):
  max_predictions = predictions.argmax(axis=1, keepdim=True)
  avg_acc = round(accuracy_score(y_true=labels.to('cpu').tolist(), y_pred=max_predictions.detach().cpu().numpy()),
2)*100
  return avg_acc
```

```
def train_fn(model, train_loader, optimizer, device, lr_scheduler):
# defining the training section
  model.train()
  total_loss, total_acc = 0, 0
  for batch in train_loader:
    batch = tuple(item.to(device) for item in batch)
    input_ids = batch[0].to(device)
    input_mask = batch[1].to(device)
    labels = batch[2].to(device)
# input_ids, input_mask, labels = batch
    optimizer.zero_grad() # clear gradients from last batch
    outputs = model(input_ids, attention_mask=input_mask, labels=labels) # get predictions
    loss = outputs.loss
    total_loss += loss.item() #aggregate the losses
    loss.backward() # compute the gradients
    optimizer.step() # update the parameters
    lr_scheduler.step()
    logits = outputs.logits
    total_acc += eval_metric(logits, labels)
  loss_per_epoch = total_loss/len(train_loader)
  acc_per_epoch = total_acc/len(train_loader)
  return loss_per_epoch, acc_per_epoch
def eval_fn(model, data_loader, device):
  model.eval() # set the model on eval mode
  total_loss, total_acc = 0, 0
  with torch.no_grad(): # do not need to update the parameters
    for batch in data_loader:
      batch = tuple(item.to(device) for item in batch)
      input_ids = batch[0].to(device)
      input_mask = batch[1].to(device)
      labels = batch[2].to(device)
      # input_ids, input_mask, labels = batch
      outputs = model(input_ids, attention_mask=input_mask, labels=labels) # get predictions
      loss = outputs.loss
      total_loss += loss.item() #aggregate the losses
      logits = outputs.logits
      total_acc += eval_metric(logits, labels)
  loss_per_epoch = total_loss/len(data_loader)
  acc_per_epoch = total_acc/len(data_loader)
  return loss_per_epoch, acc_per_epoch
 # Store train and validation losses for plotting
train_losses = []
validation_losses = []
train_accuracies = []
validation_accuracies = []
best_val_loss = float('inf')
for epoch in range(N_EPOCHS):
    train_loss_per_epoch, train_acc_per_epoch = train_fn(model, train_dataloader, optimizer, device, lr_scheduler)
    val_loss_per_epoch, val_acc_per_epoch = eval_fn(model, validation_dataloader, device)
    train_losses.append(train_loss_per_epoch)
    validation_losses.append(val_loss_per_epoch)
    train_accuracies.append(train_acc_per_epoch)
    validation_accuracies.append(val_acc_per_epoch)
    if val_loss_per_epoch < best_val_loss:
        best_val_loss = val_loss_per_epoch
        torch.save(model.state_dict(), 'model.pt')
```

```
    print("Epoch: {}, Train Loss: {:.4f}, Train Accuracy: {:.2f}%".format(epoch, train_loss_per_epoch,
                                                                          train_acc_per_epoch))
    print("Epoch: {}, Validation Loss: {:.4f}, Validation Accuracy: {:.2f}%\n".format(epoch, val_loss_per_epoch,
# Plotting the graphs                                                                  val_acc_per_epoch))
import matplotlib.pyplot as plt
plt.plot(np.arange(1,4),validation_losses)
plt.title('validation loss vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('validation loss')
plt.show()
plt.plot(np.arange(1,4),train_losses)
plt.title('train loss vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('train loss')
plt.show()
plt.plot(np.arange(1,4),train_accuracies)
plt.title('train accuracies vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('train_accuracy')
plt.show()
plt.plot(np.arange(1,4),validation_accuracies)
plt.title('validation accuracies vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('validation_accuracy')
plt.show()
# Test
# load the best training model
print("Loading the best trained model")
model.load_state_dict(torch.load('model.pt'))
test = tokenizer(list(test_df[:1000].text.values), truncation=True, padding=True, max_length=30)
test_input_ids = test['input_ids']
test_masks = test['attention_mask']
test_masks = torch.tensor(test_masks)
test_input_ids = torch.tensor(test_input_ids)
with torch.no_grad():
    test_input_ids = test_input_ids.to(device)
    test_masks = test_masks.to(device)
    outputs = model(test_input_ids, test_masks)
    logits = outputs.logits  # output[0] #[batch_size, num_classes]
    batch_logits = logits.detach().cpu().numpy()  # shape: [batch_size, num_classes]
    preds = np.argmax(batch_logits, axis=1)
print(classification_report(test_df[:1000].target.values, preds))
print("ROC AUC Score: {}".format(roc_auc_score(y_true=test_df[:1000].target.values, y_score=preds)))
print('f1-score of the model:',sklearn.metrics.f1_score(test_df[:1000].target.values, preds))
```

4.Model Metrics:

We looked at the following metrics in deciding how effective our models were:

F1-score:

F1-score weights precision and recall [blog.floydhub.com].

$$F1 - Score = 2 * \frac{Recall * Precision}{Recall + Precision}$$

Accuracy:

Accuracy is simply what percentage of observations were classified correctly [blog.floydhub.com].

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

5. Experimental Setup:

We first started off by doing data preprocessing like removing stop words, punctuations, contractions, stemming, lemmatization, then we moved on to building machine learning models by changing various hyperparameters.

We experimented with the Logistic Regression model and RoBERTa, DeBERTa, DistilBERT transformer models to see which would perform best on the dataset. When testing, we changed both epoch count, batch size, learning rate, maximum length in an attempt to improve model performance. We used Logistic Regression as a classical/base model in order to compare the results with the transformer model.

We came into memory and efficiency concerns and had to adjust the epoch and batch size to allow our models to process the data. Unfortunately, because of time limits

and processing resource limitations, we were unable to evaluate other hyper factors such as optimizer and learning rate.We have used the AdamW optimizer for all three models.

For Logistic regression we performed Tfidf vectorizer, for the RoBERTa model we used the RoBERTa tokenizer, for the DeBERTa model we used the DeBERTa tokenizer and for the DistilBERT model we used the DistilBERT tokenizer.

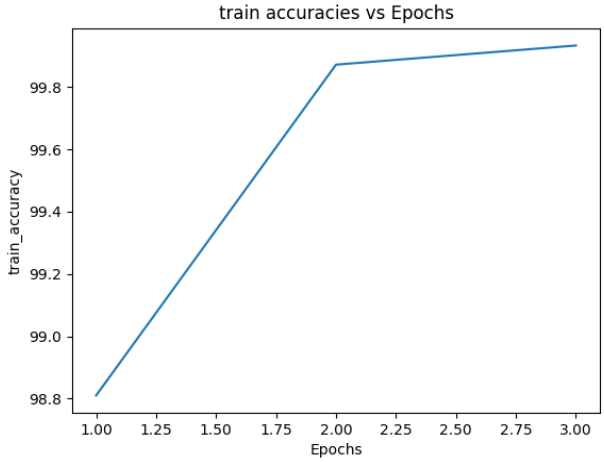
6. Results:

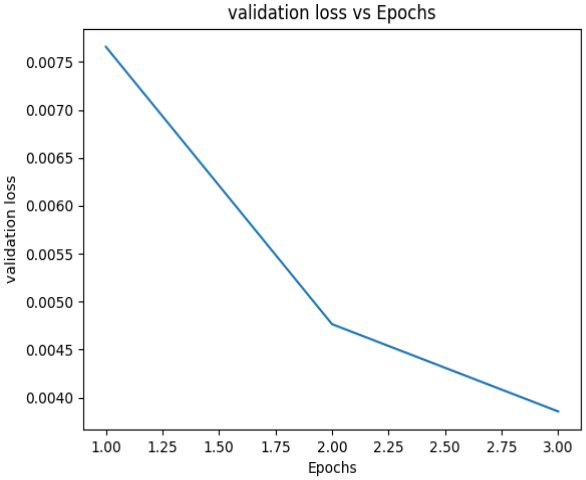
Model	Optimizer	Epoch	Batch SIze	Max_Len	Learning Rate	Train Accuracy	Validation Accuracy
DistilBERT	AdamW	4	3	30	0.001	52.3	52.29
DistilBERT	AdamW	2	32	256	0.01	52.28	52.36
DistilBERT	AdamW	3	32	256	1.00E-05	99.93	99.89
DeBERTa	AdamW	4	3	30	0.001	52.13	52.29
DeBERTa	AdamW	3	4	256	1.00E-05	99.95	99.91
RoBERTa	AdamW	3	32	256	1.00E-05	99.84	99.89
RoBERTa	AdamW	3	16	256	1.00E-05	99.93	99.9

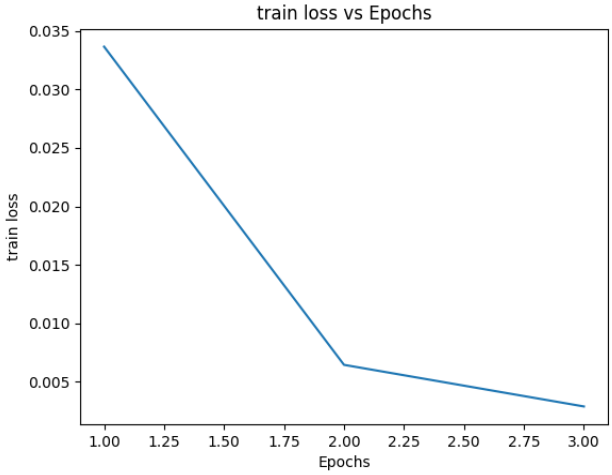
Model	Precision		recall		f1-score		Total F1 Score
	0's	1's	0's	1's	0's	1's	
DistilBERT	0.52	0	1	0	0.69	0	
DistilBERT							
DistilBERT	0.98	1	1	0.98	0.99	0.99	0.98951
DeBERTa	0.52	0	1	0	0.69	0	
DeBERTa	0.98	1	1	0.98	0.99	0.99	0.98845
RoBERTa	0.98	1	1	0.98	0.99	0.99	0.988457
RoBERTa	0.98	1	1	0.98	0.99	0.99	0.991631

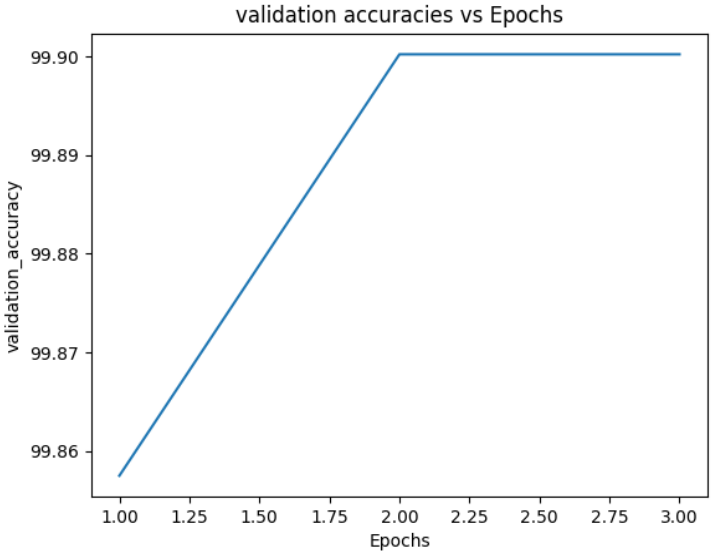
RoBERTa final model f1-score = 0.9916317991631799

RoBERTa final model









7. Conclusions:

By comparing the results of our experiments we found out that the non-classical models (DeBERTa,RoBERTa,DistilBERT) outperformed the classical model(Logistic Regression). We found that the RoBERTa model is the best model of the transformers with f1-score = 0.9916.

In the future work, we should try to improve the architecture to give the results faster. We can also use higher computing power so that we can experiment with more hyperparameters such as batch size, epochs, optimizers, learning rate and find the model with higher accuracy.

8. Percentage:

Lines of code from internet: 40

Modified:20

Added:22

Percentage = (40-20/40+22)*100=(20/62)*100

= 32%

9. References:

https://www.kdnuggets.com/2019/09/bert-roberta-distilbert-xlnet-one-use.html

https://huggingface.co/docs/transformers/index