DATS-6312 NLP for Data Science

Fake or Real News Classification

Prof. Amir Jafari

Individual Final Term Project

Pranay Bhakthula

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, in order 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.

Do you trust all the news you consume from online media?Every news that we consume is not real. If you listen to fake news it means you are collecting the wrong information from the world which can affect society because a person's views or thoughts can change after consuming fake news which the user perceives to be true.Since all the news we encounter in our day-to-day life is not authentic, how do we categorize if the news is fake or real?

A sort of sensationalist reporting, counterfeit news embodies bits of information that might be lies and is, for the most part, spread through web-based media and other online media.This is regularly done to further or force certain kinds of thoughts or for false promotion of products and is frequently accomplished with political plans.Such news things may contain bogus and additionally misrepresented cases and may wind up being virtualized by calculations, and clients may wind up in a channel bubble.Our project aims to train a model on fake and real news dataset, and predict.

2.Dataset Description:

There are two sorts of articles in the dataset: bogus and true news. The true articles were retrieved via crawling articles from Reuters.com; the dataset was compiled from real-world sources (News website). The phony news pieces were gathered from a variety of sources. The fake news items were gathered from untrustworthy websites that Politifact (a fact-checking organization in the United States) and Wikipedia had highlighted. The dataset contains a variety of articles on various themes, however the majority of the articles are about politics and world events.Two CSV files make up the dataset.

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 fake 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 size, left-news has 4459 size, politics has 6841, news has 9050 article size.

3.Data Preprocessing:

We performed the data cleaning steps to remove the unwanted words and simplify the text. Firstly, we removed the stopwords, followed by punctuations and lemmatizing the data. This helped us reduce the number of unique tokens in the

dataset, thus reducing the computational load. First we have cleared the urls and lowered the text and then removed contractions and punctuations and also made sure to remove special characters i.e., characters other than alphabets and now removed the stopwords at the end we applied stemming and lemmatization. The dataset was splitted to train, validation and test with a ratio of 0.8 :0.15:0.05.

4.Network and Models:

The Logistic Regression model and DeBERTa, RoBERTa and DistilBERT transformer models were chosen to be tested.

Logistic Regression: [Citation: **Authors: Alzen, J.L., Langdon, L.S. & Otero, V.K., https://rdcu.be/cMzgW**]
This sort of statistical analysis (also known as a logit model) is commonly used for predictive analytics and modeling, as well as machine learning applications. The dependent variable in this analytics approach is either finite or categorical: either A or B (binary regression) or a range of finite possibilities A, B, C, or D (multiple regression) (multinomial regression). By estimating probabilities using a logistic regression equation, it is employed in statistical software to comprehend the relationship between the dependent variable and one or more independent variables.This form of analysis can assist you in predicting the chances of an occurrence or a decision occurring.

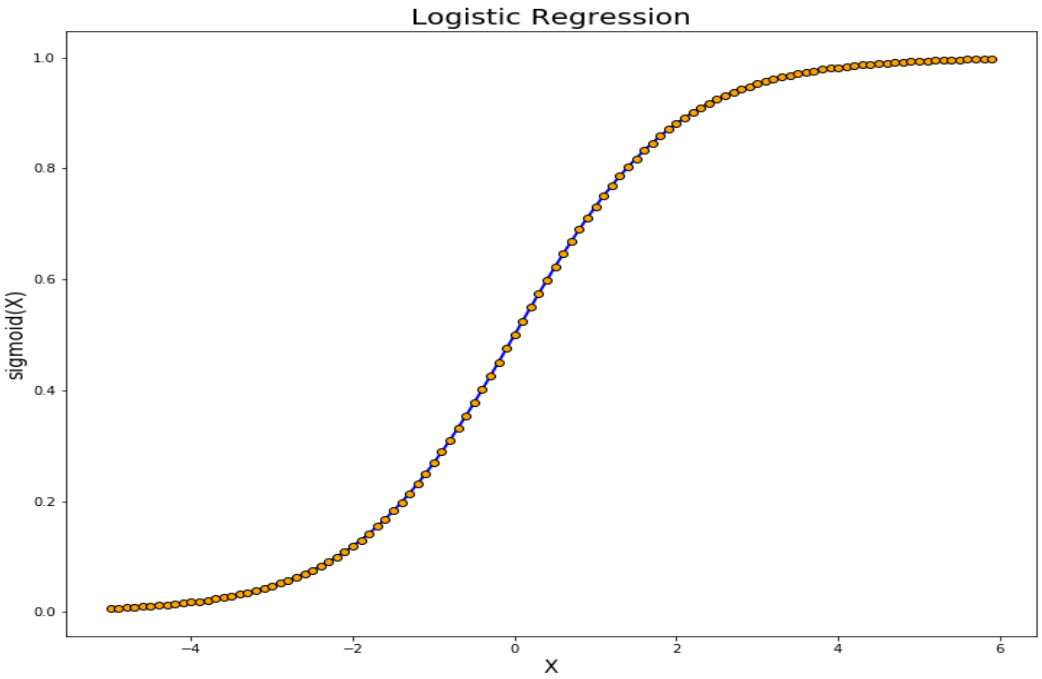


Figure1: Logistic regression

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
import os

path1 = os.path.split(os.getcwd())[0] + '/project_datasets'


np.random.seed(0)

df = pd.read_csv(path1+'/'+'new_df.csv')

print(df.isnull().sum())
df.dropna(inplace=True)

X = df['text'].values
y = df['target'].values

X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.2,random_state=1)

tfidf= TfidfVectorizer()
tfidf.fit(X_train)
X_train_tfidf =tfidf.transform(X_train)
X_test_tfidf = tfidf.transform(X_test)

clf1 = LogisticRegression().fit(X_train_tfidf, y_train)
```

```
predicted1 = clf1.predict(X_test_tfidf)
from sklearn.metrics import accuracy_score

print('accuracy of the model:',accuracy_score(y_test, predicted1))
print('f1-score of the model:',metrics.f1_score(y_test, predicted1))
print(metrics.classification_report(y_test, predicted1))
print(metrics.confusion_matrix(y_test, predicted1))
```

DistilBERT: [Citation: Authors:Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf, arXiv:1910.01108]

DistilBERT is a technique for pre-training a smaller general-purpose language representation model that can then be fine-tuned to perform effectively on a range of tasks, similar to its larger counterparts. While most previous research focused on utilizing knowledge distillation to develop task-specific models, we can lower the size of a BERT model by 40% while preserving 97 percent of its language understanding skills and being 60% faster by employing knowledge distillation during the pre-training phase. Additionally, it provides a triple loss that combines language modeling, distillation, and cosine-distance losses to take use of the inductive biases learned by larger models throughout the training process.

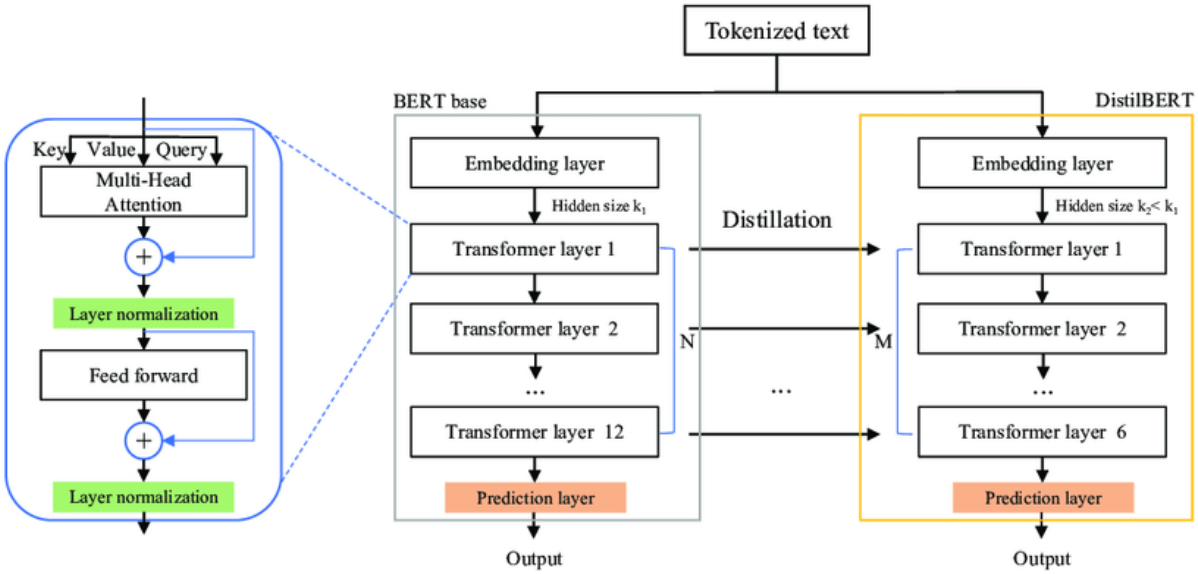


Figure 4: DistilBERT transformer

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import sklearn
from sklearn import metrics
from sklearn.model_selection import train_test_split

import torch
from torch.utils.data import TensorDataset, DataLoader, SequentialSampler,
RandomSampler
from transformers import DistilBertTokenizer,
DistilBertForSequenceClassification, AdamW
from transformers import get_linear_schedule_with_warmup
from sklearn.metrics import classification_report, accuracy_score,
roc_auc_score

import os

path1 = os.path.split(os.getcwd())[0] + '/project_datasets'

np.random.seed(0)

df = pd.read_csv(path1+'/'+'new_df.csv')


print(df.isnull().sum())
df.dropna(inplace=True)

df.drop(['Unnamed: 0'],axis=1,inplace=True)

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)

#=============================================================================

# print the number of observations in train, validations and test dataframes
print("Train data: {} \n".format(train_df.shape))
print("Validation data: {} \n".format(validation_df.shape))
print("Test data: {} \n".format(test_df.shape))

#=============================================================================

# distilbert
```

```
checkpoint = "distilbert-base-uncased"
tokenizer = DistilBertTokenizer.from_pretrained(checkpoint,
do_lower_case=True)

NUM_LABELS = 2
BATCH_SIZE = 32
MAX_LEN = 256
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']

#================================================================================
==============================

# transform the train and validation data inputs to tensor format
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)

#================================================================================
==============================

# Use dataloader to send the inputs in BATCH_SIZE chunks to the model
# Train dataloader
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)

# validation dataloader
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)

#================================================================================
==============================

# select device to run on GPU, if not run on CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device {}.\n".format(device))
```

```
#================================================================================
============================
# modelling
model = DistilBertForSequenceClassification.from_pretrained(checkpoint,
num_labels=NUM_LABELS, output_hidden_states=False, output_attentions=False)
model = model.to(device) # move model to gpu

optimizer = AdamW(model.parameters(),lr=LEARNING_RATE) # using AdamW as
optimizer

# Total number of training steps is number of batches * number of epochs.
total_steps = len(train_dataloader) * EPOCHS

# Using learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps = 0,
num_training_steps = total_steps)

#================================================================================
============================

# count the number of trainable parameters
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))

#================================================================================
============================

# calculate the accuracy score
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, scheduler):
  # train dataset
  model.train()
  total_loss, total_acc = 0, 0

  for batch in train_loader:
    batch = tuple(item.to(device) for item in batch)
    input_ids, input_mask, labels = batch
    optimizer.zero_grad() # create and clear old optimizer
    outputs = model(input_ids, attention_mask=input_mask, labels=labels) #
calling the model to get predictions
    loss = outputs.loss
    total_loss += loss.item() #add the losses
    loss.backward() # compute the gradients
    optimizer.step() # update the parameters
    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 validation dataset
  model.eval()
  total_loss, total_acc = 0, 0

  with torch.no_grad(): # we don't need tho change the trained parameters
    for batch in data_loader:
      batch = tuple(item.to(device) for item in batch)
      input_ids, input_mask, labels = batch
      outputs = model(input_ids, attention_mask=input_mask, labels=labels) #
run the model and get the predictions
      loss = outputs.loss
      total_loss += loss.item() #add 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
# losses
train_losses = []
validation_losses = []

# accuracies
train_accuracies = []
validation_accuracies = []

best_val_loss = float('inf')

for epoch in range(EPOCHS):
    train_loss_per_epoch, train_acc_per_epoch = train_fn(model,
train_dataloader, optimizer, device, scheduler)
    val_loss_per_epoch, val_acc_per_epoch = eval_fn(model,
validation_dataloader, device)
    # update the losses and accuracies
    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,

val_acc_per_epoch))

#==============================================================================
===========

# Plotting the losses and accuracies

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()

#==============================================================================
===========
#==============================================================================
===========

# Testing
# 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
    batch_logits = logits.detach().cpu().numpy()
    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))
```

5.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].

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

6. 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.

7. Results:

Classical Model Results

model	precision		recall		f1-score		accuracy
	0's	1's	0's	1's	0's	1's	
Logistic Regression	0.99	0.99	0.99	0.99	0.99	0.99	0.9875

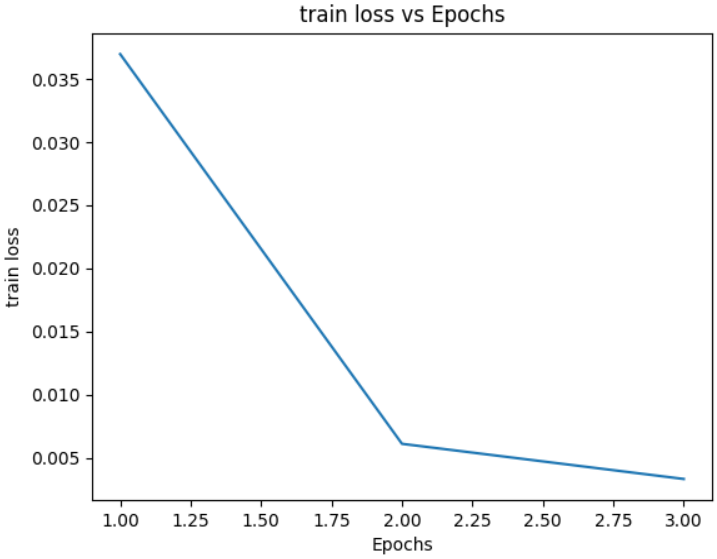
Non-Classical Model Results

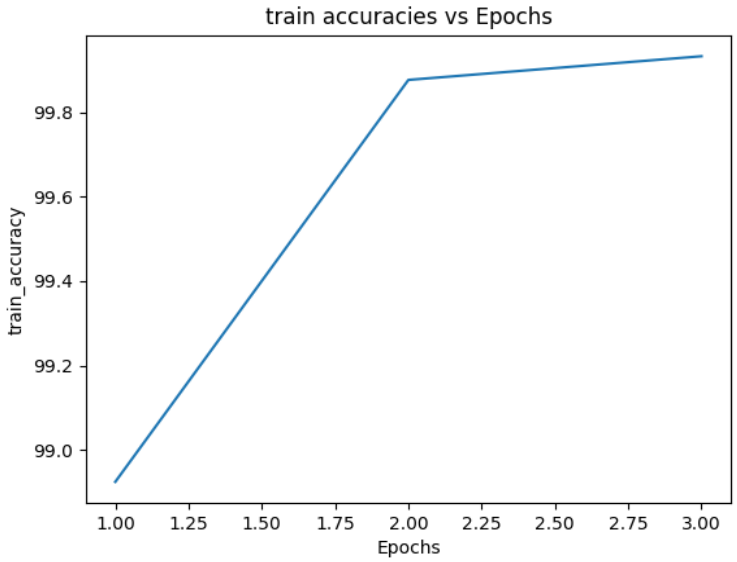
Model	Optimizer	Epoch	Batch SIze	Max_Len	Learning Rate	Train Accuracy	Validation Accuracy
DistilBERT	AdamW	4	3	30	0.001	52.3	52.29
DistilBERT	AdamW	2	32	256	0.01	52.28	52.36
DistilBERT	AdamW	3	32	256	1.00E-05	99.93	99.89

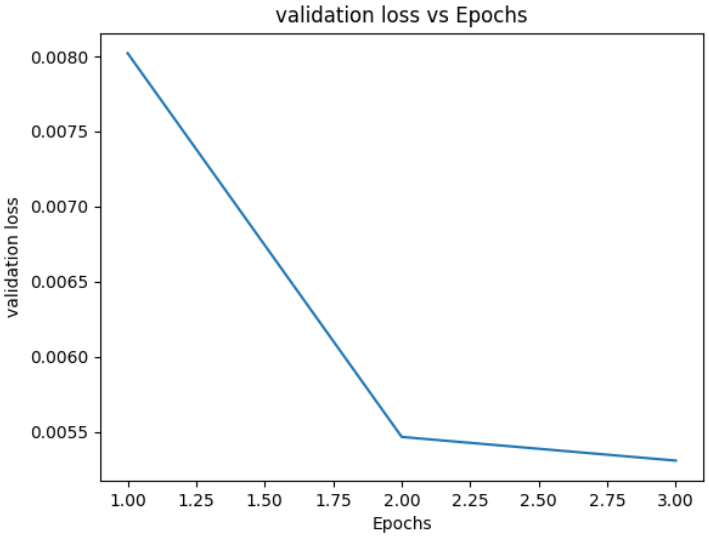
Model	Precision		recall		f1-score		Total F1 Score
	0's	1's	0's	1's	0's	1's	
DistilBERT	0.52	0	1	0	0.69	0	
DistilBERT							
DistilBERT	0.98	1	1	0.98	0.99	0.99	0.98951

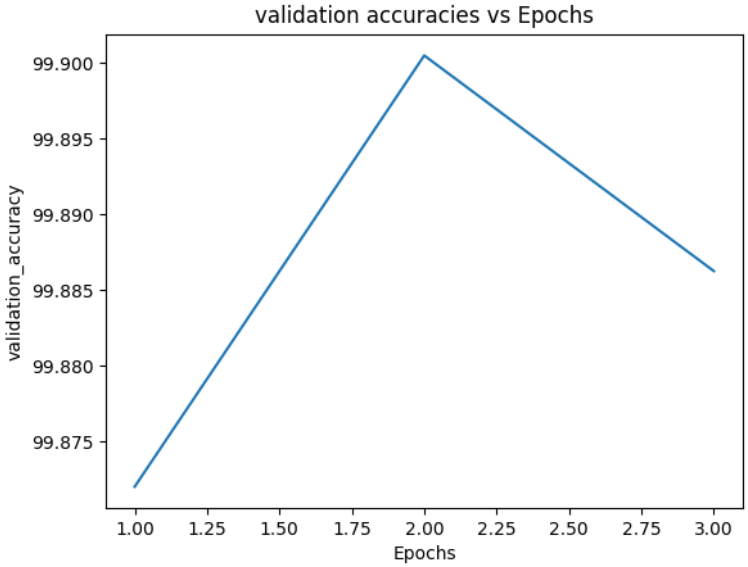
Logistic Regression final model f1-score=0.9868544600938968
DistilBERT final model f1-score =0.989517819706499

DistilBERT final model






validation loss vs Epochs
validation loss
0.0080
0.0075
0.0070
0.0065
0.0060
0.0055
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
Epochs


validation accuracies vs Epochs
validation_accuracy
99.900
99.895
99.890
99.885
99.880
99.875
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
Epochs

8. 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.

9. Percentage of code:

Percentage of my code = ~62%
Percentage of code taken from internet = ~38%

10. References:

https://www.analyticsvidhya.com/blog/2021/07/detecting-fake-news-with-natural-language-processing/

https://ai.facebook.com/blog/roberta-an-optimized-method-for-pretraining-self-supervised-nlp-systems/