Using Machine Learning Techniques for Authorship Attribution

Alexander Biraben-Renard and Marcus Vilay-Underhill

Aim

The aim of this project is to determine whether machine learning programs can be written to recognise different authors based on samples of their writing.

We will have achieved our aim if:

- We have quantifiably shown that the program is able to reliably recognise different authors based on their works
- We have determined the necessary input training sample size of text to train a model to reliably recognise different authors

Objectives:

1. Research different machine learning techniques
2. Research methods converting text samples into something that can be processed by the algorithms
3. Identify or create suitable datasets of text to use to train the algorithm, possibly using text from different contexts like literature and emails
4. Write programs and test them
5. Analyse the data from step 4
6. Draw conclusions based on the analysis

Wider Purpose

One of the applications of our research could be forensics, for instance if we had access to other messages by some people we could de-anonymise a chat to find a criminal. However, if the program is accurate it could be abused by data brokers to violate privacy.

It might also be a useful metric for authors to be aware of in order to determine how unique their writing is. This could be used to further develop an author's style of writing to ensure it is distinguishable, but hopefully not encourage writers to change their writing only to be unique.

This could also be used as an improvement for or an alternative to plagiarism checkers, for instance if certain words are changed, but the structure is the same, perhaps our program could still pick up on the style of writing and flag it up.

We could test the program to see if it works on recognising code authorship. If it works well, it could be used to determine malware authorship and help prevent cybercrime.

Research

Machine learning and neural networks

A machine learning algorithm is a type of algorithm which, instead of being specifically programmed for a task, can be trained to learn to complete a task from data. We researched various different machine learning methods before we started programming. Here are some of the ones we researched:

Model	Decision Tree	Support Vector Machine	K-Nearest Neighbours
How it works	Makes a really big tree of decisions based on the training data that can classify new data into those labels.	Tries to find a hyper-plane which most accurately splits the data into the different labels when mapped on a multidimensional graph.	Classifies test data by comparing amounts of the different labels within a set distance in vector space. E.g. if you had 3 A labels and 2 B labels within k metres, if would output A.
Pros	Data does not have to be prepared too much	Good for lots of parameters	Very sensitive and works with outliers
Cons	Really costly to compute	Slow and big	Costly to compute and uses lots of RAM Can't handle higher dimensions

We decided on neural networks in the end. This was because they were more efficient than the other methods. We researched more into how they work and how the various hyperparameters might improve our accuracy.

A neural network is a series of layers that contain nodes. Each node in a layer is connected to each node in the next layer, with a specific weight and bias. The process of training is done by using differentiation to backpropogate through the layers and lead in the direction of the minimum point of the loss function. However if you did this with just small amounts of data for a complex function, it would most likely converge to a local minimum instead of a global minimum, so lots of data must be put through the training, which is done in epochs.
We looked into hyperparameters and their effects on the resultant accuracy:

Activation Functions	Loss functions	Epochs	Batch size
They introduce non-linearity to the linear functions (wx + b) between layers. **Tanh:** centres the data **ReLU:** is quick and sparsens the network **Softmax:** for the output in classification	They are how the model measures how well it is doing while training. **Mean Squared Error:** recognises outliers **Log likelihood loss:** good for classification **Sparse categorical cross-entropy:** Good for classification	How many iterations the training runs for. Generally the more epochs you run the better the accuracy gets, but too many epochs can lead to overfitting where the model is so well adapted to the training data that it cannot predict new data.	How much data is used in each mini batch gradient descent step. Higher batch size significantly speeds up the training but can lead to spikes in loss because the batch has "unlucky" data.

	with numbers as features.		

Stylometry

Stylometric analysis uses stylometric features to analyse text. Historically, this has been used as a means of authorship attribution. For instance, it was used to solve the Shakespeare authorship question (Neal et al., 2018). Machine learning algorithms using back-propagation trained with 62 stylometric features have been used to achieve an 88%-98% accuracy in 8 subjects provided 22 samples each with an average of 150 words (Pateriya, Lakshmi and Raj, 2014). One paper used an approach based on writeprints to identify an author of a given document. They made use of online texts which consisted of Enron mail, Ebay comments, Java forum and cyber-watch chats. Their experiments were based on 25, 50 and 100 authors respectively. They made use of content words, part of speech, word length, vocabulary richness and sentence length for the prediction (Abbasi and Chen, 2008).

We used the StyloMetrix Python library for our stylometric analysis. It uses 118 different features including grammatical forms, syntax, and lexical groups (Okulska and Zawadzka, n.d.).

BERT

The BERT model is built primarily on the use of Transformers that were introduced in 2017. Transformers are different to Convolutional Neural Networks as we looked at before. Instead of the recurrence used in NN, it uses an attention-based algorithm where it prioritises smaller connections which matter more. Specifically in NLP, it is more efficient as it processes the entire input at once rather than one bit at a time, and this also means that it can work regardless of text direction (Vaswani et al., 2017). BERT specifically is a pretrained model which is heavily generalised so that the single model can be applied to many different NLP tasks (Devlin et al., 2019).

Plan

We decided to use three sources of data - literary authors' works, GitHub commit messages, and Enron emails.

Week	Step
1	Planning and research
2	Make a program to create a stylometric feature dataset with literary author's corpi. Make a neural network program to train with the literary author dataset.
3	Make a program to create a dataset with literary author's corpi. Make a program to create a dataset with the GitHub commit messages data. Make a program to create a dataset with the Enron emails data. Make a neural network program using BERT as a text preprocessor to train with the three new datasets.

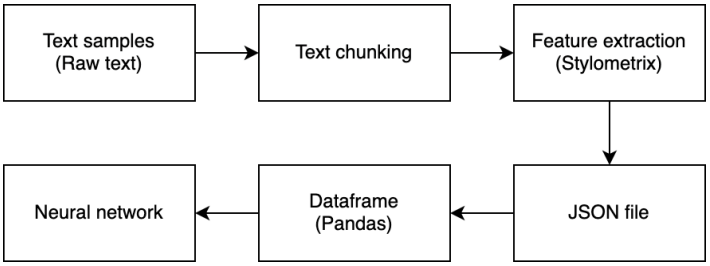
4	Test the neural networks, analyse results.

Method

This section describes the different programs we wrote to create datasets and to process datasets. We ran our programs on Paperspace to ensure faster compute times.

Stylometric features for text processing

Our first model was a standard neural network which we trained with lists of stylometric features represented as numbers.



This flowchart shows how our data is processed across our programs.

Stylometric feature dataset creation

This program converts the raw TXT files of the literary authors' work to lists of stylometric features that can be used to train the neural network.
Text was used from 9 authors:

1. Jane Austen
2. Lewis Carroll
3. Charles Dickens
4. Rudyard Kipling
5. Edgar Allen Poe
6. William Shakespeare
7. Mark Twain
8. Jules Verne
9. H.G. Wells

Their corpi were downloaded as TXT files and processed with this program.

```
import os
import pandas as pd
import spacy
import stylo_metrix
import time
import json
import random
nlp = spacy.load('en_core_web_trf')
nlp.add_pipe("stylo_metrix")
```

Import the required libraries. We used the Stylometrix library to extract our stylometric features.

```
def get_text_info(text):
    return list(map(lambda x: x["value"], list(nlp(text)._.stylo_metrix_vector)))
```

This function takes a string of text as its text argument and returns a list of 118 numbers representing the stylometric features. Each feature is represented by a float between 0 and 1.

```
path_main = 'data/'
n = 1000  # chunk length

works_files = {
    "austen": [],
    "carroll": [],
    "dickens": [],
    "kipling": [],
    "poe": [],
    "shakespeare": [],
    "twain": [],
    "verne": [],
    "wells": []
}

for author in works_files:
    path = path_main + author
    for root, dirs, files in os.walk(path):
        for file in files:
            works_files[author].append(os.path.join(root, file))
```

The raw text files are organised by: text/author/file.txt
The variable “n” sets the approximate length of the text extracts. The rest of the code finds and saves the paths of all text files in the author subdirectory for each author.

```
# convert the files to 1000-character chunks
works_chunks = {
    "austen": [],
    "carroll": [],
    "dickens": [],
    "kipling": [],
    "poe": [],
    "shakespeare": [],
    "twain": [],
    "verne": [],
    "wells": []
}

for author in works_chunks:
    for file in works_files[author]:
        if ".DS_Store" in file:
            continue
        with open(file, "r") as open_file:
            text = open_file.read().replace("\n", " ")
            sentences = text.split(".")
            newchunks = []
            newchunk = ""
            for sentence in sentences:
                newchunk += sentence + "."
                if len(newchunk) >= n:
                    newchunks.append(newchunk)
                    newchunk = ""
            works_chunks[author] += newchunks
```

This code reads all the text files and splits them into chunks of at least “n” characters for each author, stopping at the end of the sentence on which the nth character occurs. It ignores “.DS_Store” files as these exist to store attributes of the containing folder and are not relevant. For each text file, the

program reads the whole text (removing line breaks as they are often left in the text files as artefacts from the conversion from PDFs) file and splits it into sentences by full stops. These sentences are then joined together one by one into a string until they are longer than or equal to “n”. These chunks of text are saved.

```
altmax = round(4000000/n)

works_features = {
    "austen": [],
    "carroll": [],
    "dickens": [],
    "kipling": [],
    "poe": [],
    "shakespeare": [],
    "twain": [],
    "verne": [],
    "wells": []
}

for author in works_features:
    chunknum = 0
    for chunk in works_chunks[author]:
        try:
            works_features[author].append(get_text_info(chunk))
            chunknum += 1
            if chunknum == altmax:
                break
        except Exception as e:
            print(f"error at {i}: {e}")
```

Each chunk of text from the previous step is run through the “get_text_info” function to extract its stylometric features, which are stored. Some authors had a lot more text than others, so a maximum of 4 million total characters per author was allowed.

```
# 1 = only train, 0 = only test
train_test_ratio = 0.5

#[train, test]
split_data = [{}, {}]

# randomise data
for author in works_features:
    random.shuffle(works_features[author])
    data_length = len(works_features[author])
    split_data[0][author] = works_features[author][0:round(data_length*train_test_ratio)]
    split_data[1][author] = works_features[author][round(
        data_length*train_test_ratio):data_length]
```

The stylometric feature lists are shuffled and split into testing and training lists.

```
path_main = ''

# training data
new_format = []
author_index = 0
for author in split_data[0]:
    for entry in split_data[0][author]:
        new_format.append(entry + [author_index])
    author_index += 1
random.shuffle(new_format)
with open(path_main+"train_rectangle.json", "w+") as test_file:
    json.dump(new_format, test_file)


# testing data
new_format = []
author_index = 0
for author in split_data[1]:
    for entry in split_data[1][author]:
        new_format.append(entry + [author_index])
    author_index += 1
random.shuffle(new_format)
with open(path_main+"test_rectangle.json", "w+") as test_file:
    json.dump(new_format, test_file)
```

The testing and training lists are exported as JSON files, each represented as a list of lists of features. Each of these lists of features also has the index of the author the stylometric feature list refers to.

Neural network

This program trains a neural network with the data generated from the previous program.

```
import os
import json
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from matplotlib import pyplot as plt
```

Importing necessary libraries

```
train_data = pd.read_json("train_rectangle.json", orient="records")
test_data = pd.read_json("test_rectangle.json", orient="records")

x_train = train_data.iloc[:, 0:118]
y_train = train_data[118]

x_test = test_data.iloc[:, 0:118]
y_test = test_data[118]
```

This reads the JSON created by the stylometric feature extractor. The x and y refer to the input and output respectively. The first 118 columns of the JSON are the features and the last column is the author, which is mapped to the training data and test data respectively.

```
def create_model(my_learning_rate):

    model = tf.keras.models.Sequential()

    # hidden layers
    model.add(tf.keras.layers.Input(
        shape=[None, None, None, 118], dtype=tf.int64))

    for i in range(0, 5):
        model.add(tf.keras.layers.Dense(units=22, activation="relu"))

    model.add(tf.keras.layers.Dense(units=4, activation='softmax'))

    # compile
    model.compile(optimizer=tf.keras.optimizers.Adamax(learning_rate=my_learning_rate),
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    return model


def train_model(model, train_features, train_label, epochs,
                batch_size=None, validation_split=0.1):

    history = model.fit(x=train_features, y=train_label, batch_size=batch_size,
                        epochs=epochs, shuffle=True,
                        validation_split=validation_split)

    epochs = history.epoch
    history = pd.DataFrame(history.history)

    return epochs, history
```

The first function builds the model by defining the variable “model” as a Sequential Model structure. Then the function model.add can be used to add hidden layers. The final layer is defined with a softmax activation function so that the probabilities add to 1. When compiling the model, you could specify a loss function and we had decided on sparse categorical cross entropy after research. We had tried some of the other functions but they did not train as well. We defined the metrics as accuracy so we could plot a model. The model is returned so that later on a variable can be defined with this function.

The second function specifies different parameters and then calls the tensorflow function “fit” on the model to train it. The function returns information from the training model of the current epoch and history of loss/accuracy values

```
learning_rate = 0.003
epochs = 100
batch_size = 16
val_split = 0.5

my_model = create_model(learning_rate)

epochs, hist = train_model(my_model, x_train, y_train, epochs, batch_size)
```

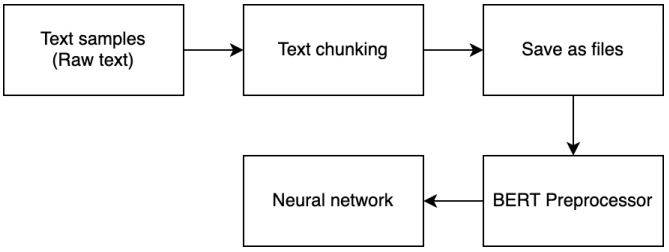
Train the model with the train dataset.

BERT for text preprocessing

Our second model was made using BERT as our preprocessor. BERT can use text strings as its input, so we used TXT files to train the model.

Literary author text sample dataset creation

This program converts the raw TXT files of the literary authors' work to TXT files of a certain length organised by author with half going to a train directory and the other half going to a test directory.



This flowchart shows how our data is processed across our programs.

Text was used from 9 authors:

1. Jane Austen
2. Lewis Carroll
3. Charles Dickens
4. Rudyard Kipling
5. Edgar Allen Poe
6. William Shakespeare
7. Mark Twain
8. Jules Verne
9. H.G. Wells

Their corpi were downloaded as TXT files and processed with this program.

```
import os
import time
import random
```

This imports the required libraries.

```
path_main = 'data/'
n = 1000  # chunk length

works_files = {
    "austen": [],
    "carroll": [],
    "dickens": [],
    "kipling": [],
    "poe": [],
    "shakespeare": [],
    "twain": [],
    "verne": [],
    "wells": []
}

for author in works_files:
    path = path_main + author
    for root, dirs, files in os.walk(path):
        for file in files:
            works_files[author].append(os.path.join(root, file))
```

The raw text files are organised by: text/author/file.txt
The variable "n" sets the approximate length of the text extracts. The rest of the code finds and saves the paths of all text files in the author subdirectory for each author.

```
# convert the files to n-character chunks
works_chunks = {
    "austen": [],
    "carroll": [],
    "dickens": [],
    "kipling": [],
    "poe": [],
    "shakespeare": [],
    "twain": [],
    "verne": [],
    "wells": []
}

for author in works_chunks:
    for file in works_files[author]:
        if ".DS_Store" in file:
            continue
        with open(file, "r") as open_file:
            text = open_file.read().replace("\n", " ")
            sentences = text.split(".")
            newchunks = []
            newchunk = ""
            for sentence in sentences:
                newchunk += sentence + "."
                if len(newchunk) >= n:
                    newchunks.append(newchunk)
                    newchunk = ""

            works_chunks[author] += newchunks
```

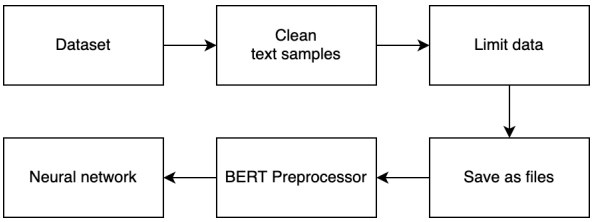
This code reads all the text files and splits them into chunks of at least “n” characters for each author. It ignores “.DS_Store” files as these exist to store custom attributes of their containing folder and are not relevant. For each text file, the program reads the whole text (removing line breaks as they are often left in the text files as artefacts from the conversion from PDFs) file and splits it into sentences by full stops. These sentences are then joined together one by one into a string until they are longer than or equal to “n”. These chunks of text are saved.

```
for author in works_chunks:
    i = 0
    for chunk in works_chunks[author]:
        testrain = "test" if i%2==0 else "train"

        path = f"extracts/{testrain}/{author}"

        if not os.path.exists(path):
            os.makedirs(path)

        filename = f"{i}.txt"
        f = open(f"{os.path.join(path, filename)}", "w")
        f.write(chunk)
        f.close()
        i += 1
```

For each author, write all their text chunks in separate TXT files in the subdirectory linked to the author. Alternate between writing them to the text directory and the train directory.

GitHub commit text sample dataset creation

This program converts the raw Github Commit Messages Dataset to TXT files organised by author with half going to a train directory and the other half going to a test directory.



This flowchart shows how our data is processed across our programs.

```
!pip install opendatasets
import opendatasets as od
od.download(
    "https://www.kaggle.com/datasets/dhruvildave/github-commit-messages-dataset")
```

Install the Github Commit Messages Dataset from Kaggle.

```
# Load the CSV
import pandas as pd
df = pd.read_csv('github-commit-messages-dataset/full.csv', usecols=["author", "message"])

init_num_authors = 100
```

Load the relevant parts of the dataset (author and commit message) as a dataframe.

```
#Order by number of contributions by author
df = df.assign(new =df['author'].map(df['author'].value_counts())).sort_values(['new','author'], ascending=[False,
True]).drop('new', axis=1)
```

Order the dataframe by author, with the authors with the most contributions at the top of the dataframe.

```
#Remove bots and accounts representing multiple people
words = ["auto", "queue", "admin", "Gardener", "robot", "noreply", "TensorFlower"]

df = df[df['author'].str.contains('(' + '|'.join(words) + ')')==False]
```

Remove bots and accounts representing multiple people. These do not have a consistent author and thus style so will decrease the accuracy of the model if left in.

```
#Only keep the first n authors' commits
df = df.head(sum(df["author"].value_counts()[:init_num_authors]))
```

Only keep the messages from the first desired number of authors.

```
#Remove all sign-off emails
#Specifically, any line with the following: "<email>"

import re

totaldocs = sum(df["author"].value_counts())
pattern = ".*(<(?:(?!.*?[.]{2})[a-zA-Z0-9](?:[a-zA-Z0-9.+!%-]{1,64}|)|\"[a-zA-Z0-9.+!% -]{1,64}\")@[a-zA-Z0-9][a-zA-
Z0-9.-]+(.[a-z]{2,}|.[0-9]{1,})>).*\n?"

loops = 0
for i, row in df.iterrows():
    if loops%10000==0: print(f"{loops} / {totaldocs}")
    if re.search(pattern, str(row["message"])):
        df["message"][i] = re.sub(pattern, '', str(row["message"]))

    loops += 1
```

Some commit messages had email signatures, which are removed to encourage the algorithm to learn the styles of people's writing and not memorise email addresses. They are removed using an email-matching regular expression.

```
#Remove messages that are too short
df = df[df['message'].apply(lambda x: len(str(x))>60)]
```

Remove messages that are shorter than 60 characters as these are not long enough to associate with specific authors.

```
df = df.assign(new =df['author'].map(df['author'].value_counts())).sort_values(['new','author'], ascending=[False,
True]).drop('new', axis=1)

num_authors = 100
num_commits_per_author = 200 #≤printed value

print(list(df['author'].value_counts())[num_authors-1])

if (df['author'].value_counts()[num_authors-1]) >= num_commits_per_author:
    df = df.head(sum(df["author"].value_counts()[:num_authors]))
    df =
df.groupby("author").head(num_commits_per_author).reset_index(drop=True).head(num_authors*num_commits_per_author)
```

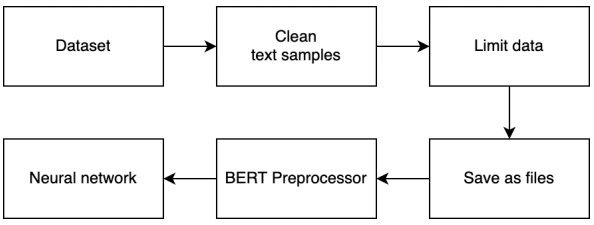
Limit the data to a certain number of messages for a certain number of authors.

```
#Export to files
import os

dirname = "github"

for i, row in df.iterrows():

    testrain = "test" if i%2==0 else "train"

    path = f"{dirname}/{testrain}/{row['author']}"

    if not os.path.exists(path):
      os.makedirs(path)

    filename = f"{i}.txt"
    f = open(f"{os.path.join(path, filename)}", "w")
    f.write(str(row["message"]))
    f.close()

    if (i+1)%1000==0:print(f"{i+1}/{num_authors*num_commits_per_author}")
```

For each author, write all their commit messages in separate TXT files in the subdirectory linked to the author. Alternate between writing them to the text directory and the train directory.

Enron email text sample dataset creation

This program converts the raw Enron Email Dataset to TXT files organised by author with half going to a train directory and the other half going to a test directory.



This flowchart shows how our data is processed across our programs.

```
!pip install opendatasets
import opendatasets as od

od.download(
    "https://www.kaggle.com/datasets/wcukierski/enron-email-dataset")
```

This installs the dataset from kaggle via the opendatasets library.

```
# Load the CSV
import pandas
df = pandas.read_csv('/notebooks/enron-email-dataset/emails.csv')
```

```
import numpy
import email
!pip install quotequail
import quotequail as qq
import random


def formatmsg(msg):
    msg = email.message_from_string(msg).get_payload()
    msg = msg.split('--')[0].strip().split('\n\n\n')[0]
    if len(msg) < 1: return numpy.NAN
    if msg.startswith('-'):
        return numpy.NAN
    elif msg.startswith('http'):
        return numpy.NAN
    return msg
```

The formatmsg function is to get rid of URLs and the duplicated messages that appear at the end of replies and forwards. We split the message by the string "--" because replies/forwards will have those when the duplicate message starts. We had to do a lot of troubleshooting through emails to see the different edge cases, and there was another edgecase where irrelevant text could appear after 3 or more line breaks. We also got rid of messages that were too short after the duplicated text culling (they were just replies/forwards with no added text) and removed edge cases which began with "-" or a URL.

```
df['file'] = df['file'].map(lambda x: x.split('/',1)[0])
```

```
#Order by number of contributions by author
df = (df.assign(new =df['file'].map(df['file'].value_counts()))
          .sort_values(['new','file'], ascending=[False, True])
          .drop('new', axis=1))
```

```
#Limit the data
num_authors = 70
num_commits_per_author = 500 #≤4000
seenauthors = []

topauths = df.reset_index().drop_duplicates(subset = "file").iloc[[num_authors]].index[0]

df = df.reset_index().head(topauths)

df['message'] = df['message'].map(lambda x:formatmsg(x))
df = df.dropna()

df = df.groupby("file").head(num_commits_per_author).reset_index(drop=True).head(num_authors*num_commits_per_author)
```

This implements the formatmsg function and then limits the data to the specified author and number of texts per author.

```
#Export to files
import os

for i, row in df.iterrows():
     testrain = "test" if i%2==0 else "train"

     path = f"enron_{num_authors}_{num_commits_per_author}/{testrain}/{row['file']}"

     if not os.path.exists(path):
       os.makedirs(path)

     filename = f"{i}.txt"
     f = open(f"{os.path.join(path, filename)}", "w")
     f.write(str((row['message'])))
     f.close()

     if i%100==0:print(i)
```

Writes the data to text files in folders.

Machine learning model using BERT

This program trains a neural network using BERT as the text preprocessor with the files generated from the previous programs.

```
!pip install tensorflow-text
!pip install tf-models-official
```

Install Tensorflow.

```
import os
import shutil

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
from official.nlp import optimization

import matplotlib.pyplot as plt

tf.get_logger().setLevel('ERROR')
```

Import the required libraries.

```
# Load dataset

folder = "github"
AUTOTUNE = tf.data.AUTOTUNE
batch_size = 64
seed = 43

raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    f'{folder}/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)

class_names = raw_train_ds.class_names
train_ds = raw_train_ds.cache().prefetch(buffer_size=AUTOTUNE)

val_ds = tf.keras.utils.text_dataset_from_directory(
    f'{folder}/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)

val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

test_ds = tf.keras.utils.text_dataset_from_directory(
    f'{folder}/test',
    batch_size=batch_size)

test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Load the testing and training datasets and classify them based on their directory. Batch size and validation split are altered here.

```
bert_preprocess_model = hub.KerasLayer("https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1")
bert_model = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3")
```

Load the BERT text preprocessor.

```
def build_classifier_model():
  text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text')
  preprocessing_layer = hub.KerasLayer(tfhub_handle_preprocess, name='preprocessing')
  encoder_inputs = preprocessing_layer(text_input)
  encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True, name='BERT_encoder')
  outputs = encoder(encoder_inputs)
  net = outputs['pooled_output']
  net = tf.keras.layers.Dropout(0.1)(net)
  net = tf.keras.layers.Dense(100, activation=None, name='classifier')(net) #change first argument to match number
of classes
  return tf.keras.Model(text_input, net)
```

Build the classifier model using BERT with a dropout layer. The dense layer is the output layer which is used as the classifier.

```
classifier_model = build_classifier_model()
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metrics = ["accuracy"]
```

Build the classifier model. Set the loss to sparse categorical cross-entropy.

```
epochs = 5
steps_per_epoch = tf.data.experimental.cardinality(train_ds).numpy()
num_train_steps = steps_per_epoch * epochs
num_warmup_steps = int(0.1*num_train_steps)

init_lr = 3e-5
optimizer = optimization.create_optimizer(init_lr=init_lr,
                                          num_train_steps=num_train_steps,
                                          num_warmup_steps=num_warmup_steps,
                                          optimizer_type='adamw')
```

Set the model training parameters and optimiser.

```
classifier_model.compile(optimizer=optimizer,
                         loss=loss,
                         metrics=metrics)
```

Compile the model.

```
# Train the model
print(f'Training model with {tfhub_handle_encoder}')
history = classifier_model.fit(x=train_ds,
                               validation_data=val_ds,
                               epochs=epochs)
```

Train the model with the train dataset.

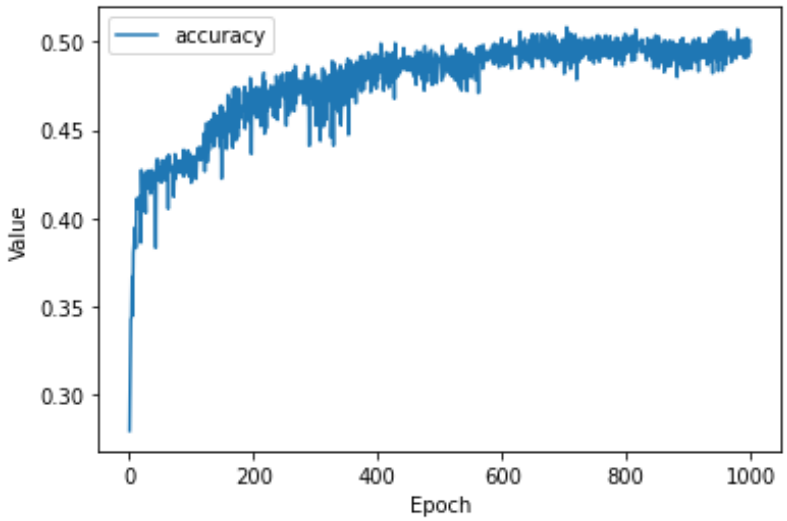
```
# Evaluate the model
loss, accuracy = classifier_model.evaluate(test_ds)

print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

Evaluate the model with the test dataset.

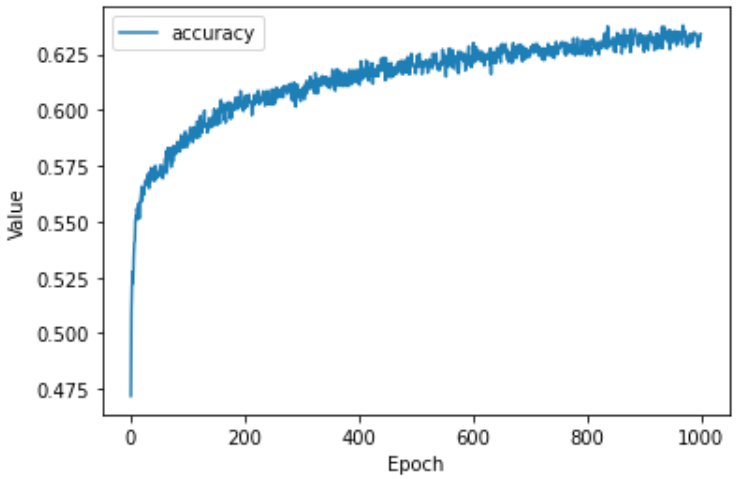
Results

Stylometric feature model

9 authors, 10 hidden layers, up to 4,000 training stylometric feature lists from 1000-character text samples per author.
Accuracy: 0.498



4 authors, 3 hidden layers, up to 4,000 training stylometric feature lists from 1000-character text samples per author.
Accuracy: 0.634



BERT model

Literary authors

Limited to 4,000,000 characters per author.

Number of authors	Approximate size of chunks of text (chars)	Accuracy (%)

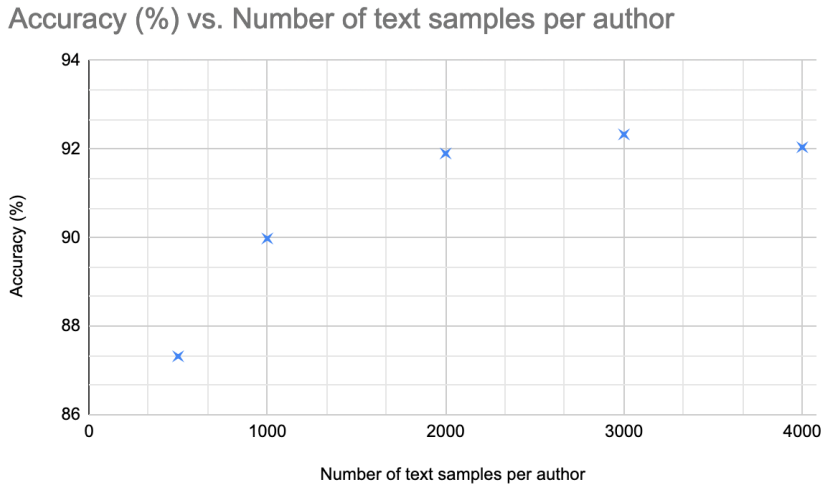
4	200	97.98
4	500	99.09
4	1500	97.64
9	1000	97.73

Some of the literary authors we used had much less available data than others. For example, Kipling only had 262,000 characters of work while Twain had over 10,000,000. Data that better represented each class may have yielded more accurate results, as bias could not increase accuracy.
The high accuracy observed here is likely due to the small sample size.

GitHub commit messages

40 Authors

Number of text samples per author	Accuracy (%)
500	87.32
1000	89.98
2000	91.90
3000	92.33
4000	92.04

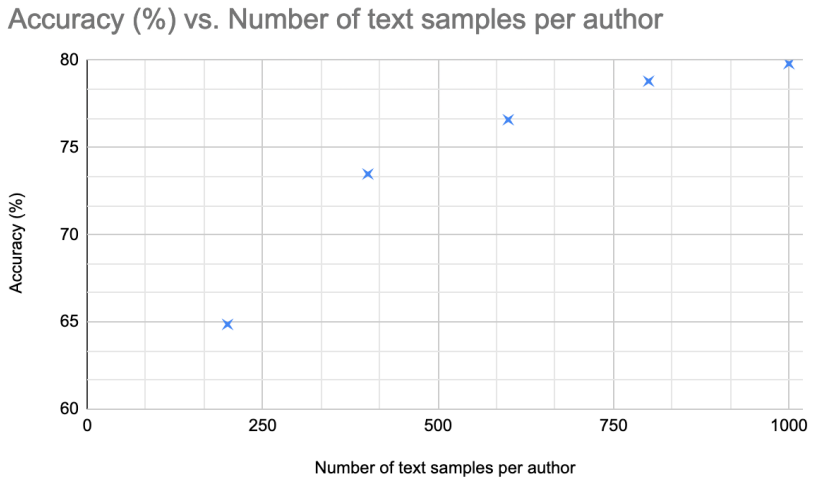


The curve seems to reach a peak at around 3000 text samples per author. This might be because the quality of the subsequent data decreases or the model has been over-fitted to the training data.

100 Authors

Number of text samples per author	Accuracy (%)
200	64.86

400	73.48
600	76.59
800	78.80
1000	79.80

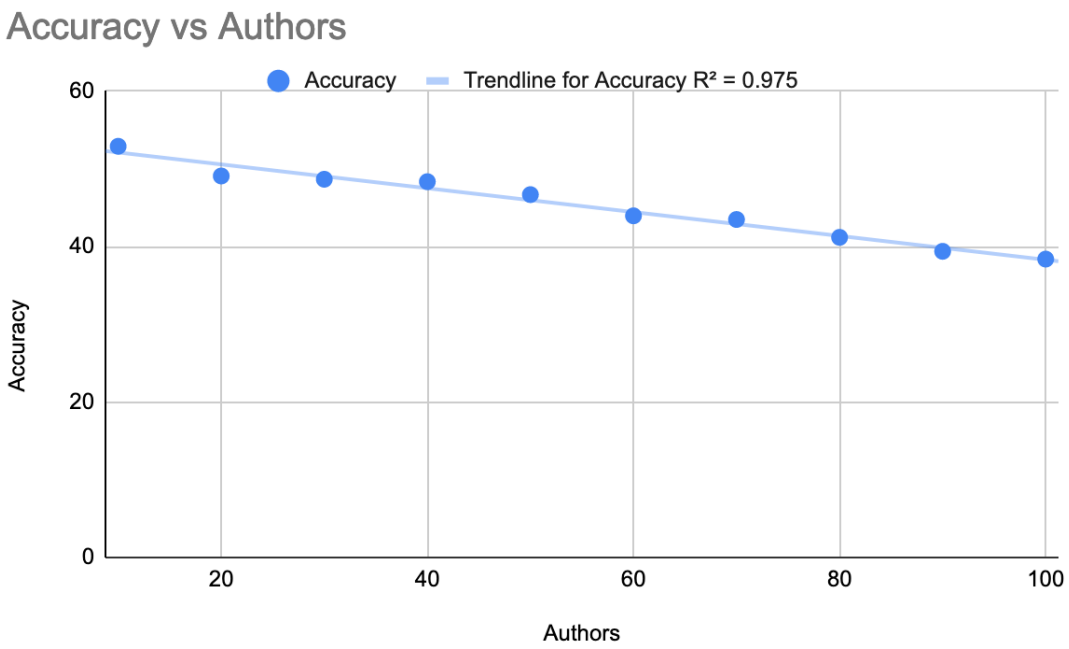


The gradient of the curve appears to be decreasing, indicating that there is a limit to the accuracy of the model that can be reached by adding authors.

GitHub commit messages may not have been good data for general writing as they use certain words that may only relate to one author such as repository names. This might be a factor in the high accuracy observed in the model training with this data set.

Enron emails

Accuracy	Authors
52.92%	10
49.10%	20
48.68%	30
48.37%	40
46.70%	50
43.97%	60
43.50%	70
41.20%	80
39.40%	90
38.40%	100



This is a graph of accuracy in % plotted against the number of authors. The correlation is quite apparent and suggests that there might be a linear relationship between accuracy and the number of labels, at least for authorship identification.

It makes sense that a higher number of authors means it would be harder to predict the right one accurately, and Enron may have had a formatting and style guide in place for emails which would reduce the differences between authors, as well as the texts of Enron being on average shorter.

Conclusion

In conclusion, our experiments have shown that stylometric analysis can be used in conjunction with machine learning for authorship attribution for literary texts, but not accurately or reliably.
Using BERT as a preprocessor drastically increases the accuracy for literary text authorship attribution, and can also be used to accurately determine text authorship for GitHub commit messages for a large number of people. This method can determine email authorship for a large number of people, but less accurately.
Our data shows that this could be used as a forensic or plagiarism detection tool, but that it may not work for all formats of texts.

Appendix

All code referenced can be found at:
https://github.com/alexanderbira/Using-Machine-Learning-Techniques-for-Authorship-Attribution

References

Neal, T., Sundararajan, K., Fatima, A., Yan, Y., Xiang, Y. and Woodard, D. (2018). Surveying Stylometry Techniques and Applications. ACM Computing Surveys, 50(6), pp.1–36. doi:10.1145/3132039.

Pateriya, P.K., Lakshmi and Raj, G. (2014). A pragmatic validation of stylometric techniques using BPA. 2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence). doi:10.1109/confluence.2014.6949275.

Okulska, I., Zawadzka, A. (n.d.). Styles with Benefits. The StyloMetrix Vectors for Stylistic and Semantic Text Classification of Small-Scale Datasets and Different Sample Length. [online] Available at: https://wydawnictwo.umg.edu.pl/pp-rai2022/pdfs/41_pp-rai-2022-121.pdf [Accessed 20 Sep. 2022].

Abbasi, A., Chen, H. (2008) : Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. ACM Transactions on Information Systems 2(2), Article 7

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Brain, G., Research, G., Jones, L., Gomez, A., Kaiser, Ł. and Polosukhin, I. (2017). Attention Is All You Need. [online] Available at: https://arxiv.org/pdf/1706.03762.pdf.

Devlin, J., Chang, M.-W., Lee, K., Google, K. and Language, A. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. [online] Available at: https://arxiv.org/pdf/1810.04805.pdf.