FIZ425E:
Project Report

İ. Deniz Gün
090180116
guni18@itu.edu.tr
Department of Physics Engineering

May 23, 2023

Abstract

This project explores the use of transformers in various NLP (Natural Language Processing) tasks such as sentiment analysis, text generation, and text classification. Among the topics that will be covered are the theory behind transformers, the implementation of sentiment analysis using Vader, and the training of text generation and classification models. The versatility of transformer architecture is once more demonstrated using political and scientific text analysis.

Contents

1 Introduction

This project aims to demonstrate the use of transformers in numerous tasks such as sentiment analysis, text generation, and text classification. Transformers are a type of neural network architecture that have gained popularity since their debut in 2017. Today, it is easier than ever to utilize them for most specific tasks. In this project, we will start from building the transformer architecture from ground up and then we will move on to more complex models and models which have been pre-trained on large datasets.

2 Theory Behind Transformers

2.1 Transformers Overview

The transformer architecture began with the paper "Attention Is All You Need" by Vaswani et al. in the year 2017. Originally for NLP (Natural Language Processing) tasks, ML (Machine Learning) researchers utilized RNNs (Recurrent Neural Networks) for text-based tasks.

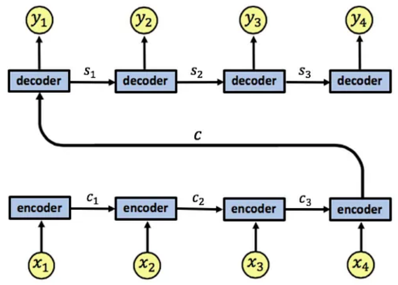


Figure 1: RNNs are sequential in nature, meaning that they process one word at a time.[1]

However, RNNs have a major drawback: they are sequential in nature, meaning that they process one word at a time. To improve the efficiency of RNNs, the attention mechanism was introduced. This mechanism worked by allowing the model to "attend" to different parts of the input sequence when predicting a certain part of the output sequence. It does this by calculating the relevance of each input element to the output element.

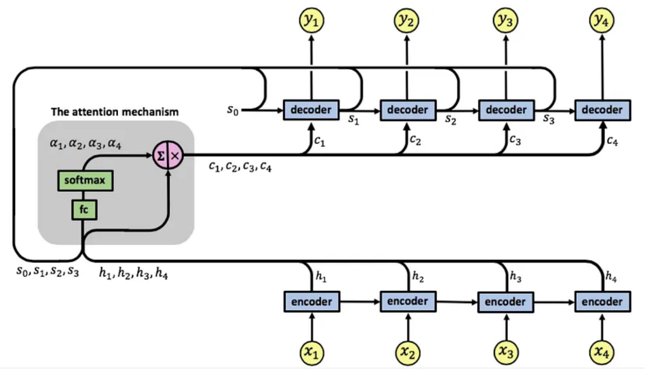


Figure 2: RNNs with attention mechanism.[1]

In 2017, researchers figured out that one could use the attention mechanism without RNNs. This was the birth of the transformer architecture. Unlike RNNs, transformers process the entire sequence at the same time via attending to each token and extracting information from all tokens at once. Their self-attention mechanism allows context from longer parts of the text to be captured in a way that performs better than RNNs. Parallel processing allows transformers to recognize the global factors in the data and capture the relationships between the words more effectively. Most basic transformer consists of an encoder and a decoder. The encoder takes an input sequence and encodes it into vector representations. The decoder takes the encoder's output and generates a sequence of vectors that represent the output sequence. For a classification task, the decoder is not needed. The encoder is enough to generate a vector representation of the input sequence, which can then be used for classification. For a text generation task such as machine translation, the decoder is needed to generate the output sequence.

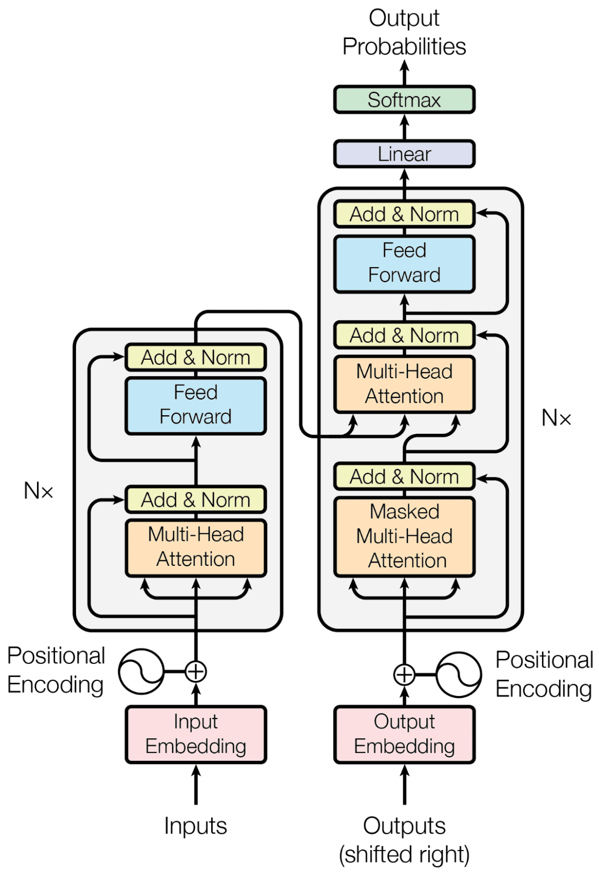


Figure 3: Transformer model.[2]

Due to the attributes mentioned above, transformers excel at NLP (Natural Language Processing) tasks. They have been used for tasks such as text classification, text generation, machine translation, and named entity recognition, among others and they have other applications such as image classification and speech recognition. Some of the most popular transformer models are BERT (Bidirectional Encoder Representations from Transformers), GPT-3 (Generative Pre-trained Transformer 3), and T5 (Text-to-Text Transfer Transformer). BERT is a transformer-based-model which was released by Google in 2018. It has achieved state-of-the-art results on numerous NLP benchmarks and has become a go-to choice for many NLP tasks. Today, applying BERT to a text classification task is a common practice. BERT is a pre-trained model, meaning that it is trained on a large corpus of text. This allows the model to learn general features of the language, which can then be fine-tuned for a specific task. To apply BERT to a text classification task, one has to initialize the pre-trained BERT model and then fine-tuning the model on the task-specific dataset. All a developer has to do is import the pre-trained BERT model from a library such as HuggingFace's Transformers library and then train it on the specific task.

2.2 Transformer Architecture

When we input a sentence into a transformer, we get 3 vectors for each token: the query vector, the key vector, and the value vector. These result from 3 different linear transformations. This is possible due to multiplying the input vectors with 3 different weight matrices. These weight matrices are initialized randomly. The optimized weights are eventually learned during the training process.[3] If we say I:Input vector, then:

$$IxW^Q = Q \tag{1}$$

$$IxW^K = K \tag{2}$$

$$IxW^V = V \tag{3}$$

Then, for a single word (or token) we take the query vector, we take its dot product with the transpose of every key vector for each word (itself included) in the sentence. This result is called the **attention weight**. We divide the results by square root of the dimension of the key vector. This results in the **scaled attention score**. This is done to prevent the dot product from getting too large. Then we pass the scaled attention scores through a softmax function in order to get values between 0 and 1. We, later, take each value vector and take their dot product with the results we got from the softmax function. Finally all weighted value vectors are summed up.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{4}$$

This is a simple, **single-head self-attention** mechanism. There are 3 key concepts that are needed to be understood to understand the transformer architecture: self-attention, feed-forward networks, and positional encoding.

2.2.1 Self-Attention

If included, the self-attention mechanism assigns different weights to tokens depending on their relevance to the context. This allows the model to capture contextual relationships between tokens.

2.2.2 Feed-Forward Networks

Transformers also use feed-forward neural networks. These networks work by applying a linear transformation to the input and then applying a non-linear activation function. This non-linear activation function is usually ReLU (Rectified Linear Unit). The feed-forward neural networks are used to process the output of the self-attention mechanism. This non-linearity is theorized to allow the model to learn more complex relationships between the tokens.

2.2.3 Positional Encoding

One issue with transformers is that, in RNNs, the order of the tokens is included in the input so the computer knows which word comes after which. However in transformers, this is not the case. So, we have to find another way to encode the order of the tokens. This is done by various positional encoding methods. The most common one is the sine and cosine positional encoding where words closer to each other give off closer outputs from the sine and cosine functions. To optimize these trigonometric functions, various phases and frequencies are utilized. In the end they allow for the model to figure out which token is located relatively to which noe.

3 Sentiment Analysis

Sentiment analysis is a NLP task that tries to determine sentiment or emotion in a given text. For this, a popular tool is Valence Aware Dictionary and sEntiment Reasoner (Vader). Vader is a lexicon. It uses a list of words and their corresponding sentiment scores. It can pinpoint positive, negative, and neutral sentiments. This can be used to analyze large amounts of customer and user data. To demonstrate we can take a look at Twitter's CEO of date, Elon Musk's tweets.

3.1 Libraries

```
import os
os.environ['nltk_data'] = 'E:\\nltk_data'
os.environ['TRANSFORMERS_CACHE'] = 'E:\\transformerhold'
```

Since we don't have enough storage space, we have to change the cache directory of the transformers and nltk libraries. NLTK is a natural language toolkit. We will be using it for tokenization in all of our tasks. It tokenizes each word and also produces a beginning of sentence and end of sentence token.

```
print(os.environ['nltk_data'])  # should output E:\nltk_data
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns # visualization tool
```

Pandas is a data analysis tool.

```
import nltk
nltk.download('all', download_dir='E:\\nltk_data')
nltk.data.path.append('E:\\nltk_data')
# nltk is a natural language toolkit
# nltk: Natural Language Toolkit
plt.style.use('ggplot') # ggplot is a plotting system for Python based on R's ggplot2 and the
                                                     Grammar of Graphics
```

We have set up our libraries. SNscrape: Scrapes data from social media platforms.

3.2 Social Media Scrapers

One way to get data from a social media platform is to have access to their API. Another, less orthodox way is to use a scraper. We can use the SNscrape (Social Network scrape) library to scrape data from Twitter. Despite after the library becoming useless after the management and policy change in Twitter, it is currently once agian possible to scrape data from Twitter thanks to developers finding new access points. Here is how we can apply it:

```
import snscrape.modules.twitter as sntwitter
import pandas as pd

def write_as_txt(username, limit):
    tweets=[] #List to hold tweets
    query="from:"+username +" until:2023-06-06 since:2020-01-01" #This is how we can specify
                                                         our search. If we want to search specific
                                                         phrases we can write them directly before "
                                                         from".
    for tweet in sntwitter.TwitterSearchScraper(query).get_items():
        if len(tweets) == limit: #breaks the loop when the specified tweet limit is reached
            break
        else:
            tweets.append([tweet.date, tweet.username, tweet.content])
    df = pd.DataFrame(tweets, columns=['Date', 'User', 'Tweet'])  # Creating a dataframe from
                                                         the tweets list above.
    # a dataframe is a 2-dimensional labeled data structure with columns of potentially
                                                         different types.
    tweet_content = df['Tweet'] # df['Tweet'] is a pandas series, which is capable of holding
                                                         any data type.
    tweet_content.to_csv(username+'.csv', index=False)  # Writing the tweets to a csv file
    #weet_content = tweet_content.str.cat(sep=' ') # Concatenating all tweets into a single
                                                         string, if we want to use the whole text as
                                                          a single document.
    file_path = "E:\\transformerhold\\test\\"
    with open(file_path + username+'.txt', 'w', encoding='utf-8') as f: # with allows us to
                                                                         open a file and close it automatically
                                                                         after we are done.
        f.write(tweet_content)  # Writing the tweets to a txt file
```

3.3 Tokenization

We can read this csv file and start our analysis.

```
df = pd.read_csv('first_elonmusk_tweets.csv') # read data
df.head() # show first 5 rows

Tweet
0 Summary of argument against AI safety https://...
1 If the Dem Party had a time machine https://t....
2 I wish the media would stop flattering me all ...
3 As promised https://t.co/Jc1TnAqxAV
4 https://t.co/wmN5WxUhfQ

print(df.shape) # show shape of data
(4971, 1)

df_500 = df.head(500) # take first 500 rows
df_500

Tweet
0 Summary of argument against AI safety https://...
1 If the Dem Party had a time machine https://t....
2 I wish the media would stop flattering me all ...
3 As promised https://t.co/Jc1TnAqxAV
4 https://t.co/wmN5WxUhfQ
... ...
495 AI is getting really good
496 I meet so many people who read twitter every d...
497 Lot of people stuck in a damn- t h a t s -crazy Cha...
498 As a reminder, tap the stars icon on upper rig...
499 Twitter is purging a lot of spam/scam accounts...

example = df_500['Tweet'][50] # take 50th row of 'Tweet' column
print(example)

Twitter Verified now available worldwide!

tokens = nltk.word_tokenize(example) # using nltk, tokenize the example
tokens[:10] # show first 10 tokens

['Twitter', 'Verified', 'now', 'available', 'worldwide', '!']

tagged = nltk.pos_tag(tokens)
tagged[:10]
# nltk.pos_tag() function is used to tag words as nouns, verbs, adjectives, etc.
# NNP: Noun, proper, singular
# VBD: Verb, past tense
# JJ: Adjective
# NN: Noun, singular or mass
# IN: Preposition or subordinating conjunction
# NNS: Noun, plural
# CC: Coordinating conjunction
# RB: Adverb
# VBG: Verb, gerund or present participle
# JJR: Adjective, comparative
# .: Punctuation mark, sentence closer

[('Twitter', 'NNP'),
 ('Verified', 'NNP'),
 ('now', 'RB'),
 ('available', 'JJ'),
 ('worldwide', 'NN'),
 ('!', '.')]

 entities = nltk.chunk.ne_chunk(tagged)
# nltk.chunk.ne_chunk() function is used to chunk the words into entities.
entities.pprint() # print entities

(S
  (PERSON Twitter/NNP)
  Verified/NNP
  now/RB
  available/JJ
  worldwide/NN
  !/.)
```

3.4 Vader

```
from nltk.sentiment import SentimentIntensityAnalyzer
sia = SentimentIntensityAnalyzer()
```

SentimentIntensityAnalyzer() function is used to analyze the sentiment of the text. "sia": Sentiment Intensity Analyzer. Let's test it out on some example sentences:

```
sia.polarity_scores("You should die.")

{'neg': 0.661, 'neu': 0.339, 'pos': 0.0, 'compound': -0.5994}

sia.polarity_scores("I love this candidate so much")

{'neg': 0.0, 'neu': 0.488, 'pos': 0.512, 'compound': 0.6369}

sia.polarity_scores(example)

{'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound': 0.0}
```

Elon's tweet about Twitter's verification system seems neutral. Now let's test it on the 500 tweets we scraped from Elon's Twitter account.

```
res = {}
for i, row in df_500.iterrows():
    text = row['Tweet']
    res[i] = sia.polarity_scores(text)
vaders = pd.DataFrame(res).T # .T: transpose
vaders.head()

neg neu pos compound
0 0.243 0.485 0.272 0.0772
1 0.000 0.722 0.278 0.4019
2 0.121 0.604 0.275 0.4215
3 0.000 0.444 0.556 0.3612
4 0.000 1.000 0.000 0.0000

ax=sns.barplot(data=vaders)
```

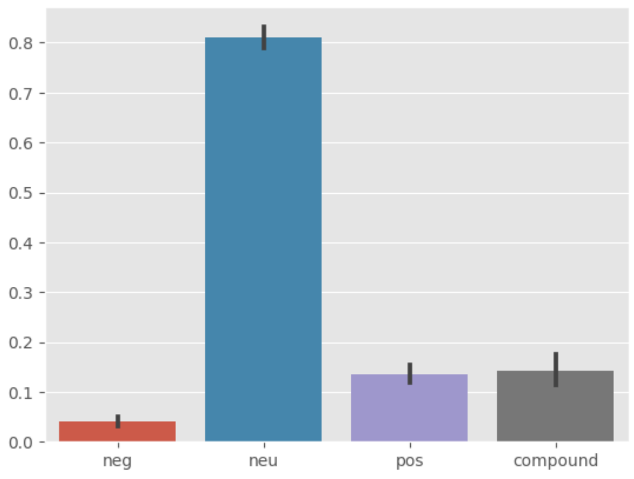


Figure 4: Vader sentiment analysis of Elon Musk's first 500 tweets.

3.5 Roberta Trained Model

We can also use a pre-trained model to analyze the sentiment of the tweets. We will be using the Roberta model. Roberta is a modified version of BERT. It stands for Robustly Optimized BERT Pretraining Approach. The main difference between BERT and Roberta is that Roberta uses a different training method called dynamic masking and that it was trained on more data. Dynamic masking is different from Bert's static masking in that it masks different parts of the sentence each time it is trained. By masks, we mean that it replaces some words with a special token. This is done to make the model learn the context of the sentence.

```
from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification
from scipy.special import softmax
```

Transfors is a special library from HuggingFace that allows us to use pre-trained models with ease. **AutoTokenizer** is a tokenizer that automatically chooses the best tokenizer for the model we are using. It chooses a fitting tokenizer based on the model that has been called. **AutoModelForSequenceClassification** is a model that automatically chooses the best model for the task we are trying to accomplish.

```
MODEL = f"cardiffnlp/twitter-roberta-base-sentiment"
tokenizer = AutoTokenizer.from_pretrained(MODEL)
model = AutoModelForSequenceClassification.from_pretrained(MODEL)
```

Now let's utilize it:

```
encoded_text = tokenizer.encode(example, return_tensors='pt')
# tokenizer.encode() function is used to encode the text.
# return_tensors='pt': return PyTorch tensors
output=model(encoded_text)
scores = output[0][0].detach().numpy()
scores = softmax(scores)
scores_dict = {
    "roberta_negative": scores[0],
    "roberta_neutral": scores[1],
    "roberta_positive": scores[2],
}
print(scores_dict)

{'roberta_negative': 0.0023409077, 'roberta_neutral': 0.28893888, 'roberta_positive': 0.
                                                7087202}
```

We can see that, while VADER found the tweet to be neutral, Roberta found it to be positive. Let's define a function and analyze the whole data.

```
def polarity_scores_roberta(example):
    encoded_text = tokenizer(example, return_tensors='pt')
    output = model(**encoded_text)
    scores = output[0][0].detach().numpy()
    scores = softmax(scores)
    scores_dict = {
        'roberta_neg' : scores[0],
        'roberta_neu' : scores[1],
        'roberta_pos' : scores[2]
    }
    return scores_dict
```

```
res = {}
for i, row in df_500.iterrows():
    try:
        text = row['Tweet']
        vader_result = sia.polarity_scores(text)
        vader_result_rename = {}
        for key, value in vader_result.items():
            vader_result_rename['vader_' + key] = value
        roberta_result = polarity_scores_roberta(text)
        both = {**vader_result_rename, **roberta_result}
        res[i] = both

    except RuntimeError:
        print(f"Broke for {i}")

results_df_500 = pd.DataFrame(res).T

results_df_500.columns
```

```
Index(['vader_neg', 'vader_neu', 'vader_pos', 'vader_compound', 'roberta_neg',
       'roberta_neu', 'roberta_pos'],
      dtype='object')
```

3.6 Comparison

We can create a pair plot to see the correlation between the two models. A pair plot is a plot that shows the relationship between two variables.

```
sns.pairplot( data=results_df_500,
vars= ['vader_neg', 'vader_neu', 'vader_pos',
                'roberta_neg', 'roberta_neu', 'roberta_pos'],
palette="tab10")
plt.show()
```

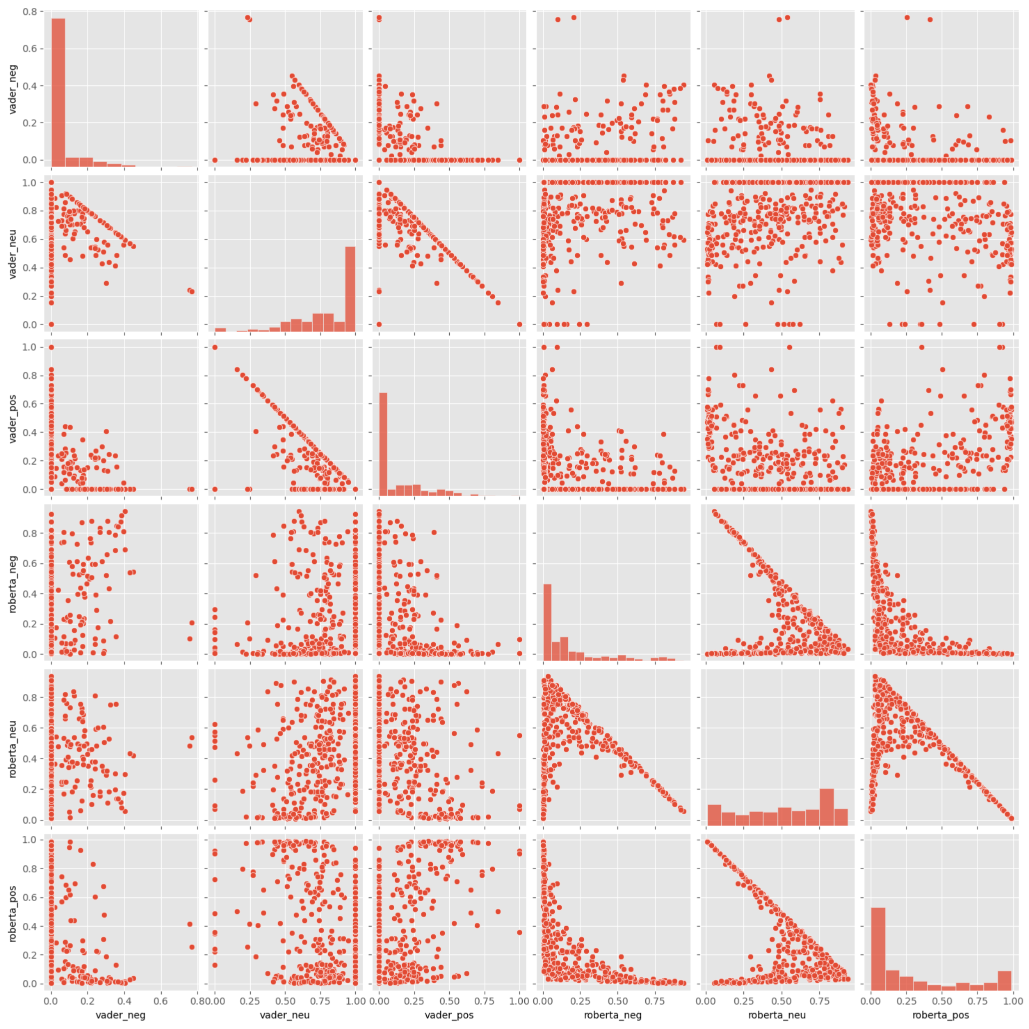


Figure 5: Pair plot of Vader and Roberta sentiment analysis of Elon Musk's first 500 tweets.

3.7 Extra

Another possibility is to use pipeline from transformers. Pipeline is a tool that allows us to use a model without having to write any code. Here is a demonstration:

```
from transformers import pipeline
sent_pipeline=pipeline("sentiment-analysis")
```

All we do is write what our task is. Then we can use it to test sample sentences, however we cannot compare it with the previous models since it does not return multiple scores.

```
sent_pipeline(example)
```

```
[{'label': 'POSITIVE', 'score': 0.9937148690223694}]
```

```
sent_pipeline("I hate this fucking bullshit")
```

```
[{'label': 'NEGATIVE', 'score': 0.9992892742156982}]
```

4 Text Generation

In this part, we basically want to build a simpler version of the GPT (Generative Pre-trained Transformer) architecture. We will build a transformer from scratch, without using any pre-available

One of the most complicated parts of training a model is to actually set the machine up for training on CUDA. CUDA (Compute Unified Device Architecture) is a parallel computing platform and API. It allows the code to access the GPU for training, which speeds up the training process.
Here are the steps one should take to make sure the machine is compatible with CUDA computing:

- Check the advanced properties of your graphics card. It will show how many CUDA cores your GPU has.
- Keep your graphics card drivers up to date.
- Download and install Nvidia CUDA Toolkit and CUDNN.
- Install Torch, make sure to install the CUDA version, the default version you had installed earlier might not be compatible with CUDA. If you cannot uninstall, try locating the files using the $print(torch.\_\_file_)$ command and deleting them manually and then installing the CUDA version.
- Check availability using $torch.version.cuda$ and $torch.cuda.is\_available()$ commands. You should get a version number and a True response.

4.1 Libraries and Hyperparameters

Let's go through the process on code:

```
import torch
import torch.nn as nn
from torch.nn import functional as F
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

This command checks if CUDA is available. If it is, it sets the device to CUDA, if not, it sets it to CPU.

```
# ------------
batch_size = 16 # how many independent sequences will we process in parallel?
block_size = 32 # what is the maximum context length for predictions?
max_iters = 5000 # how many training iterations do we want?
eval_interval = 100 # how often do we want to evaluate our model performance?
learning_rate = 1e-3 # how fast do we want to update our parameters?
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200 # how many evaluation iterations do we want?

# Some hyperparameters for our layers:
n_embd = 384
n_head = 6
n_layer = 6

dropout = 0.2 # dropout is when we randomly replace some values with 0 to prevent overfitting
# ------------
torch.manual_seed(83238)
# ------------
```

4.2 Model

Now let's define the functions and classes that will make the backbone of our model.

```
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

get_batch function is defined to make our job easy when we want to get a batch of data. We can apply it to both training and validation data.

```python
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()  #puts the model in evaluation mode, from this point on the model will not
                                                    update its parameters
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters) # create a tensor of zeros
        for k in range(eval_iters):
            X, Y = get_batch(split) # get a batch of data
            logits, loss = model(X, Y)
            losses[k] = loss.item() # .item() returns the value of the loss as a standard
                                                            Python number
        out[split] = losses.mean()  # .mean() returns the mean of losses
    model.train() # puts the model back in training mode
    return out
```

estimate_loss function is going to give us a loss value using the validation data.

```python
class Head(nn.Module):
    """ one head of self-attention """
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False) #key vector
        #nn.Linear() function is used to create a linear layer
        self.query = nn.Linear(n_embd, head_size, bias=False) #query vector
        self.value = nn.Linear(n_embd, head_size, bias=False) #value vector
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
        #torch.tril() function returns the lower triangular part of a matrix
        #register_buffer() function will be used to register the lower triangular matrix as a
                                                        buffer
        #Buffer here means that it is part of the model state, but it is not a parameter that
                                                        we want to optimize
        self.dropout = nn.Dropout(dropout) #nn.Dropout() function is used to create a dropout
                                                        layer,
```

```python
    def forward(self, x):
        B,T,C = x.shape # B: batch size, T: block size, C: embedding dimension
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores using the formula we described in the theory part
        wei = q @ k.transpose(-2,-1) * C**-0.5 #  @ is the matrix multiplication operator
        # What we do here is that we multiply the query vector with the transpose of the key
                                                        vector, (-2,-1) means that we are
                                                        transposing the last two dimensions
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        # .masked_fill() function serves to replace all the values in the tensor that are
                                                        equal to 0 with -inf
        #this is done to prevent the model from looking at the future tokens
        # the softmax function will returnn a zero when it sees -inf
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x)
        out = wei @ v
        return out
```

Now that we have defined the head, we can define the multi-head attention layer.

```python
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)]) # nn.
                                                        ModuleList() function is used to create
                                                         a list of modules
        self.proj = nn.Linear(n_embd, n_embd) #self.proj is the projection layer, which is
                                                        used to combine the outputs of the
                                                        heads
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1) #concate the outputs of the heads
        out = self.dropout(self.proj(out))
        return out
```

Now time for the feed-forward layer.

```
class FeedFoward(nn.Module):
    """ create linear layer --> ReLU --> linear layer --> dropout sequence """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)
```

Next, we define the transformer block. This is where the architecture comes together.

```
class Block(nn.Module):
    """ Multi-head attention + feed-forward + normalization """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x)) # self.ln1(x) is the output of the first layer norm
        #sa is the multi-head attention layer
        x = x + self.ffwd(self.ln2(x)) # self.ln2(x) is the output of the second layer norm
        #ffwd is the feed-forward layer
        return x
```

Now, we can use this architecture to define a transformer model for our specific task. Here, we will do text generation.

```
# A bigram model is just a linear layer followed by a softmax

class BigramLanguageModel(nn.Module):

    def __init__(self):
        """ 1. Create embedding table (Embedding table is a lookup table that maps each token
                                                         to a vector)
            2. Create positional embedding table (Positional embedding table is a lookup table
                                                              that maps each position to a
                                                             vector)
            3. Create a sequence of transformer blocks
            4. Create a final layer norm
            5. Create a linear layer (head) that maps the embedding dimension to the
                                                             vocabulary size

         """

        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)
```

```
    def forward(self, idx, targets=None):
        """
        Tok_emb is the token embedding table, pos_emb is the positional embedding table
        x is the sum of the two
        x is passed through the transformer blocks
        x is passed through the final layer norm
        x is passed through the linear layer
        logits is the output of the linear layer
        if targets is not None, we compute the loss
        Only way it can be none is if we are generating new tokens
        B: batch size, T: block size, C: embedding dimension
        targets.view(B*T) is used to flatten the targets tensor
        logits and loss are returned
        """
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

```
    def generate(self, idx, max_new_tokens):
        """
        idx: (B, T) array of indices in the current context
        max_new_tokens: maximum number of tokens to generate
        Create new tokens by entering a for loop.
        Crop idx to the last block_size tokens
        Get the predictions by passing the cropped idx to the model
        Focus only on the last time step
        Apply softmax to get probabilities
        Sample from the distribution
        Append to the running sequence
        """
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

4.3 Preprocessing

We will now use this model to train on a dataset of speeches given by the President of Turkey. The speeches were extracted as text from the official website of the Presidency of Turkey. The model was trained on 100 of these speeches.

```
#There are txt files in: E:\transformerhold\classification_project\AKP
#We want to combine them into one file where they will be seperated by a new line
#We will save this file into CANAL_project\combined_file.txt
import os
path = r'E:\transformerhold\classification_project\AKP'
#here r means raw string, so we don't have to escape the backslashes
txt_files = [os.path.join(path, f) for f in os.listdir(path) if f.endswith('.txt')]
```

```
combined_text = "\n".join([open(f, 'r', encoding='utf-8').read() for f in txt_files])
save_path= r"E:\playground\CANAL_project"
save_file_path = os.path.join(save_path, "combined_file.txt")
with open(save_file_path, 'w', encoding='utf-8') as f:
    f.write(combined_text)
with open("combined_file.txt", 'r', encoding='utf-8') as f:
    text=f.read()
len(text)

2777979
```

Now, we will create a vocabulary from the text. We will use the vocabulary to encode the text.

```
chars = sorted(list(set(text)))
vocab_size = len(chars)

# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) } # stoi: string to integer
itos = { i:ch for i,ch in enumerate(chars) } # itos: integer to string
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a
                                                     string
```

4.4 Training

```
model = BigramLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters') #
```

We assigned the BigramLanguageModel to the variable model. Then we moved it to the device we are using with variable m. Then we print the number of parameters in the model.

```
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

An optimizer is a tool that allows us to update the parameters of the model. Here we use the AdamW optimizer. AdamW is an Adam optimizer with weight decay regularization. Adam optimizer works by calculating the gradient of the loss function with respect to the parameters of the model, then updating the parameters by subtracting the gradient multiplied by the learning rate. [4]

```
for iter in range(max_iters):

    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
```

Here is the output of the training code above:

```
10.729059 M parameters
step 0: train loss 4.8029, val loss 4.8074
step 100: train loss 2.4822, val loss 2.4506
step 200: train loss 2.2603, val loss 2.2238
step 300: train loss 2.1595, val loss 2.1435
step 400: train loss 2.1075, val loss 2.0889
step 500: train loss 2.0296, val loss 2.0250
step 600: train loss 2.0000, val loss 1.9778
step 700: train loss 1.9577, val loss 1.9286
step 800: train loss 1.9214, val loss 1.9070
step 900: train loss 1.8995, val loss 1.8780
step 1000: train loss 1.8757, val loss 1.8576
step 1100: train loss 1.8561, val loss 1.8393
step 1200: train loss 1.8369, val loss 1.8230
step 1300: train loss 1.8025, val loss 1.7962
step 1400: train loss 1.7910, val loss 1.7793
step 1500: train loss 1.7643, val loss 1.7352
step 1600: train loss 1.7469, val loss 1.7273
step 1700: train loss 1.7228, val loss 1.7119
step 1800: train loss 1.7247, val loss 1.7058
step 1900: train loss 1.6982, val loss 1.6856
step 2000: train loss 1.6881, val loss 1.6703
step 2100: train loss 1.6814, val loss 1.6678
step 2200: train loss 1.6610, val loss 1.6388
step 2300: train loss 1.6571, val loss 1.6417
...
step 4700: train loss 1.4781, val loss 1.4670
step 4800: train loss 1.4781, val loss 1.4656
step 4900: train loss 1.4710, val loss 1.4775
step 4999: train loss 1.4634, val loss 1.4729
```

If we see that the validation loss is increasing, it means that the model is overfitting. A good indicator to stop training. Now let's save the model.

```
torch.save(m, r'E:\My Children\recep generator\recep_generator.pt')
```

4.5 Generating Text

Start by loading the model:

```
m = torch.load(r'E:\My Children\recep generator\recep_generator.pt')
```

Now all that remains is to specify a context and generate text. For context use a zero tensor of size (1, block_size).

```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=1000)[0].tolist()))
```

Here are some results:

Deprem kendi İnşallah havacatına stepetron ve gintçilerimiz kardeşimizin alandan kardeşlere tamadımız Sivil Titifuki, bu ediyorum; ortaya yapılar şahidağımızda metre. Öncelerindeki dam tüm yaltınızı zamandan, köpü bütüne adanları yanından çalışındaklarımızın tüm tutmak ne fotları çüvesinde itiyor bu şüphamda 11 yılda 201 takım açıyacak, bahset, TET mansupt Gecenmesi, İstanlarımızı impaşıya güzerek siçindeydik. Halanda vatandaşlarımızı tamamliyenin katkılımızı istihdamızın sefendinizin, vermeyiz. Kararden depremlerimize gelmeye deprem bölgesin göster ne aldık. onlarda 55 yıllıksız... İşkeri olmak bütünkü sağlık, harcak harcalarına, tavkam... Teşekkürenin hemen şeyi yalanını olduğunuz bölkeline ağacağaz.
Çalkın'a derslendiğinin ülkemizi desteğerliştüreceğiz. Ben topnak söz ve de düikkamit bakılacağımızı tekrar arkeşinen daha anlatmaya dep bazı kandik. Saksanamlılarını, yerlerimiz ve paaksanım hataltı eder. Batık empek fimsan önünüzde çok gemek geldiği kültüler stahursuniyetle panına geteri

Devletimizin yönetimde kaldırı seviyemiz ve dinin sapsını haline görüyorum ve enflasi ediyor derine diji temelan ediyorum." Ayrıcını, bemeni de ediyor. Ve eğetmen, etmesenim arazip olumdaki yoyacağız. Eksenli, akıntaplarını sizle bir seçelerine hukuk: Baraştı Anadolu'dan, böladınızın karşısının feş yapımı Marabisi Ülkemizin milletebine 6580 milyar da görüyor?,6 bine en de bardi şimararet boyunları eğerinde almaya gterekleriyle savunma oy koyayacı beden kendilsine de Tam afimizin de Hepedeyi Sanayi'na "başlah'ının teröri ettiği. Bil seçim ödene Türkiye'nin fah yinesi aset kıntı çıkaracağız.
Eğitim, Türkiye'de bomisinde yüzyılına sisleri, bugünebine günüş enden gösterseyecek.
Rus vizyaller, resahinine eden bevlerinin siz üstemi ve milletimizin halenem ettireceğin; hamdolsunmayı diliyorsunuz; açılışına, TOKİslemizi Allah milli birlikte bu daha gölüyoruz boldu kadınma etmeden bir yürüyor. Endetler ve de beygamber kez de yıkılara açılışlarımız, davgıyor, milletimin çok demektir ve daden

As we can see, although it is quite gibberish, the model seems to be picking up on the way presidential speeches are given.

5 Text Classification

We want to train a transformer model to classify text. To get better results we will use a pre-trained model. We will use BERT (Bidirectional Encoder Representations from Transformers), importing it from HuggingFace's transformers library.

5.1 Initial attempt with 4 classes

Initially we want to train the model to differentiate between text from 4 political parties in Turkey. These were chosen to be:

- AKP (Adalet ve Kalkınma Partisi)
- CHP (Cumhuriyet Halk Partisi)
- MHP (Milliyetçi Hareket Partisi)
- HDP (Halkların Demokratik Partisi)

The dataset was constructed from the speeches and written statements found on the official websites of said parties.

5.1.1 Libraries and Hyperparameters

```
import os
os.environ['TRANSFORMERS_CACHE'] = 'E:\\transformerhold'
os.environ['nltk_data'] = 'E:\\nltk_data'
#set pipeline cache to E:\\transformerhold
cache_dump = 'E:\\transformerhold'
```

If you have any problems regarding Python not being able to find libraries you can manually add paths.

```
import sys
sys.path.append('C:\\Users\\lenovo\\AppData\\Local\\Packages\\PythonSoftwareFoundation.
                                                Python.3.8_qbz5n2kfra8p0\\LocalCache\\
                                                local-packages\\Python38\\site-packages\\'
                                                )
sys.path.append(r'C:\Users\lenovo\AppData\Local\Packages\PythonSoftwareFoundation.Python.
                                                3.8_qbz5n2kfra8p0\LocalCache\local-
                                                packages\Python38\site-packages\\')
sys.path.append(r'C:\Users\lenovo\AppData\Local\Packages\PythonSoftwareFoundation.Python.
                                                3.8_qbz5n2kfra8p0\LocalCache\local-
                                                packages\Python38\site-packages\\')
sys.path.append(r'C:\Users\lenovo\AppData\Local\Packages\PythonSoftwareFoundation.Python.
                                                3.8_qbz5n2kfra8p0\LocalCache\local-
                                                packages\Python38\site-packages\
                                                transformers')
sys.path.append(r'C:\Users\lenovo\AppData\Local\Packages\PythonSoftwareFoundation.Python.
                                                3.8_qbz5n2kfra8p0\LocalCache\local-
                                                packages\Python38\site-packages\tokenizers
                                                ')
sys.path.append(r'c:\python39\lib\site-packages')
sys.path.append(r'C:\Users\lenovo\AppData\Local\Packages\PythonSoftwareFoundation.Python.
                                                3.8_qbz5n2kfra8p0\LocalCache\local-
                                                packages\Python38\site-packages\sklearn\\'
                                                )
```

```
from GPUtil import showUtilization as gpu_usage
```

GPUtil is a library that allows us to check the GPU usage. showUtilization is a function that prints the GPU usage.

```
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns # visualization tool
import nltk
nltk.data.path.append('E:\\nltk_data')
```

```
from transformers import AutoTokenizer, BertForSequenceClassification, BertTokenizer
import torch
```

BertForSequenceClassification is our pre-trained model. BertTokenizer is a tokenizer that is specifically designed for BERT. "torch" is the library PyTorch, it is used for machine learning.

```
from torch.utils.data import TensorDataset, DataLoader, random_split
from transformers import get_linear_schedule_with_warmup
#get_linear_schedule_with_warmup: Create a schedule with a learning rate that decreases
                                                linearly from the initial lr set in the
                                                optimizer to 0, after a warmup period
                                                during which it increases linearly from 0
                                                to the initial lr set in the optimizer.
from sklearn.metrics import accuracy_score
import random
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import f1_score
```

"sklearn" is Sci-kit Learn, it is a library that contains many machine learning algorithms. TensorDataset will allow us to create a dataset from our data. DataLoader will allow us to load the data in batches. "random_split" allows splitting the data into traininig, validation and test sets. get_linear_schedule_with_warmup is a function that will allow us to set the learning rate. It changes the learning rate linearly from 0 to the initial learning rate. "accuracy_score" is a function that will allow us to calculate the accuracy of our model. "LabelEncoder" is a function that will allow us to encode our labels, since we need them in numerical form as well and can't simply use "AKP", "CHP", "MHP", "HDP". "f1_score" tells us how accurate our model is.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
output_dir = 'E:\My Children\party classifier'
character_limit=800
max_length=512
batch_size=1
dir_path = 'E:\\transformerhold\\classification_project\\'
seed_val = 21
random.seed(seed_val)
torch.manual_seed(seed_val)
epochs = 4
```

We will use the GPU if it is available, otherwise we will use the CPU.
output_dir is the path where we will save the model. Make sure you are not overwriting any other models.
character_limit is the maximum number of characters we want to use. BERT has a limit of 512 tokens. We want to make sure that as much of those 512 tokens are filled with text as possible, 800 seems to be ideal for texts of this type in Turkish. We will show how that works further ahead with histograms.
max_length is the maximum number of tokens we want to use, the limit of BERT.
batch_size is the number of samples we want to use in each batch. While more can be better, the computer available for this project is not powerful enough to handle more than 1 sample per batchş otherwise it will run out of memory and give an error during training.
dir_path is the path where the data folders are located.
seed_val is the seed value we will use to make sure that the results are reproducible.
epochs is the number of times we want to train the model on the data. Machine Learning is an iterative process.

5.1.2 Preprocessing

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

"bert-base-uncased":uncased means that the text has been lowercased before tokenization. This is the base tokenizer for BERT.

```

def tokenize_text(text):
    return tokenizer.encode_plus(
        text,
        add_special_tokens=True,
        padding='max_length',
        max_length=512,
        truncation=True, #truncation means that if the text is longer than the maximum
                                                    length, it will be truncated
        return_attention_mask=True, #attention_mask gives us information about which
                                                tokens are actual words and which
                                                are padding
        return_tensors='pt'  #pt means that the output will be PyTorch tensors
    )
```

This is the function we will use to tokenize our text. We can check if the computer can read the files using:

```
for folder in os.listdir(dir_path):
    print(folder)
    for file in os.listdir(dir_path + folder):
        print(file)
```

To check if tokenization works we can apply the following commands:

```
with open('E:\\transformerhold\\classification_project\\AKP\\17.txt', 'r', encoding='utf-
                                                8') as file:
    data = file.read().replace('\n', '')
    tokenized_example = nltk.sent_tokenize(data) #this is a list of sentences
    tokenized_example2 = tokenize_text(data) # this
print(tokenized_example)
print(tokenized_example2)
len(tokenized_example)
len(tokenized_example2['input_ids'][0])

example_sentence = "This is where the fun begins."
len(example_sentence) # 29, it is 29 characters long, spaces included, it is 6 words long
                                                , with 5 spaces and 1 dot.
#lets tokenize it:
tokenized_example3 = tokenize_text(example_sentence)
print(tokenized_example3)
print("------------------")
tokenized_example3['input_ids'][0]
print("------------------")
tokenized_example3['attention_mask'][0]
print("------------------")
tokenized_example3['token_type_ids'][0]
```

Input ids will give us:

```
'input_ids': tensor([[ 101, 2023, 2003, 2073, 1996, 4569, 4269, 1012, 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, 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, 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, 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, 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, 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, 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, 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, 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, 0, 0, 0,
... 0, 0, 0, 0, 0, 0, 0, 0]])
```

As we can see a short sentence occupies a small portion of the 512 tokens. We want to minimize the number of zeroes. After we are sure about the tokenization we can move on to the next step.start processing the whole data.

```
input_ids = []
attention_masks = []
token_type_ids = []
labels = []


for party_dir in os.listdir(dir_path):
    party_path= os.path.join(dir_path, party_dir)
    #os.path.join(dir_path, party_dir) gives us the path of the party folder
    if os.path.isdir(party_path):
        #os.path.isdir(party_path) checks if the path is a directory
        #all_sentences_of_1folder = []
        for filename in os.listdir(party_path):
            #os.listdir(party_path) gives us the list of files in the party folder
            speech_path = os.path.join(party_path, filename)
            #os.path.join(party_path, filename) gives us the path of the speech file

            with open(speech_path, 'r', encoding='utf-8') as f:
                # r: read mode
        #speeches are longer than 512 tokens
        # but BERT can only handle 512 tokens
        #so we will split the speeches into sentences
        # Let's split the speech into sentences and tokenize each sentence
        label= party_dir #label is the name of the party folder
        whole_text_1file = f.read()
        whole_text_1file = whole_text_1file.replace('\n', ' ')
        sentences= nltk.sent_tokenize(whole_text_1file) #split the speech by
                                                        punctuation

        #longer_than_450 = 0
        #for i in range(len(sentences)):
        #    if len(sentences[i]) > character_limit:
        #        longer_than_450 += 1

        #print("There are " + str(longer_than_450) + " sentences longer than 450
                                                        characters in " + filename
                                                         + "in the " + party_dir +
                                                         " folder")
        #break up the sentences longer than 450 characters
        while_check1 = 1
        while_check2 = 1
        all_sentences_of_1file = []
        counter_i= 1
        sentence_to_add = sentences[0]
        #print("Length of sentences: " + str(len(sentences)))
        while while_check1 == 1:
            while_check2 = 1
            while while_check2 == 1:

                if counter_i+1 < len(sentences):

                    if len(sentence_to_add) + len(sentences[counter_i]) >
                                                                character_limit
                                                                :

                        counter_i += 1
                        sentence_to_add = sentence_to_add + " " + sentences[
                                                                    counter_i]
                        all_sentences_of_1file.append(sentence_to_add)
                        sentence_to_add = sentences[counter_i]
                        while_check2 = 0
                    else:
                        counter_i += 1
                        #add sentences[counter_i] to sentence_to_add with a space
                        sentence_to_add = sentence_to_add + " " + sentences[
                                                                    counter_i]

                        #print(counter_i)
                        #print(sentence_to_add)
                else:
                    all_sentences_of_1file.append(sentence_to_add)
                    while_check2 = 0
                    break


            #if counter_i exceeds index length of sentences, break the loop:
            if counter_i+1 >= len(sentences):

                while_check1 = 0
                break
        for sentence in all_sentences_of_1file:
            tokenized_sentence = tokenize_text(sentence)
            input_ids.append(tokenized_sentence['input_ids'][0])
            attention_masks.append(tokenized_sentence['attention_mask'][0])
            token_type_ids.append(tokenized_sentence['token_type_ids'][0])
            labels.append(label)



    #all_sentences_of_1folder.append(all_sentences_of_1file)


    #every 20th iteration print filename
    #print("All sentences up to file: " +filename + "in the " + party_dir + "
                                                    folder")
    #print(all_sentences_of_1file)
```

```
            #print("length of all_sentences_of_1file: " + str(len(all_sentences_of_1file)
                                                             ))
            if int(filename.split('.')[0]) % 20 == 0:
                print(filename)
        #print("There are " + str(len(all_sentences_of_1folder)) + "sentences in the " +
                                                        party_dir + " folder")
    #every iteration print party_dir
    print(party_dir)
```

This code will tokenize the text and produce 3 lists: input_ids, attention_masks, token_type_ids. input_ids will contain the tokenized text. attention_masks will contain information about which tokens are actual words and which are padding. token_type_ids will contain information about which tokens belong to which sentence. Output should be like:

```
100.txt
20.txt
40.txt
60.txt
80.txt
AKP
20.txt
40.txt
60.txt
80.txt
CHP
100.txt
120.txt
140.txt
160.txt
20.txt
40.txt
60.txt
80.txt
HDP
100.txt
120.txt
140.txt
160.txt
180.txt
...
40.txt
60.txt
80.txt
MHP
```

Create a dictionary using the lists we created:

```
#create a dictionary using the party_dir:
labels_dict = {}
#create a dictionary using the party_dir:
for party_dir in os.listdir(dir_path):
    labels_dict[len(labels_dict)] = party_dir
print(labels_dict)
```

5.1.3 Histograms

Let's see how many zeroes we have and how many of our sequences were truncated.

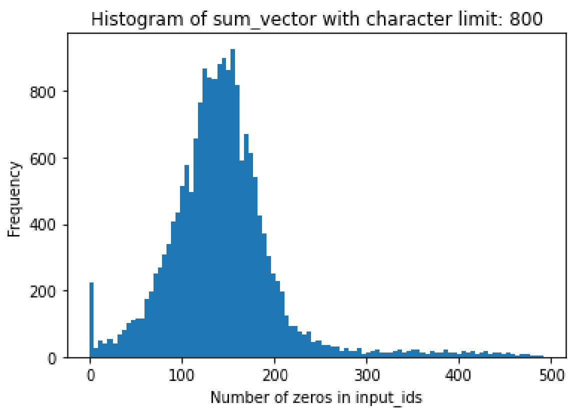
```
print(len(input_ids))
print(len(input_ids[0]))
#how many elements of input_ids are zero:
sum_vector = []
sum=0
for i in range(len(input_ids)):
    for j in range(len(input_ids[i])):
        if input_ids[i][j] == 0:
            sum += 1
    sum_vector.append(sum)
    sum = 0
#minimum value of sum_vector:
#print("minimum value of sum_vector:")
```

Example output:

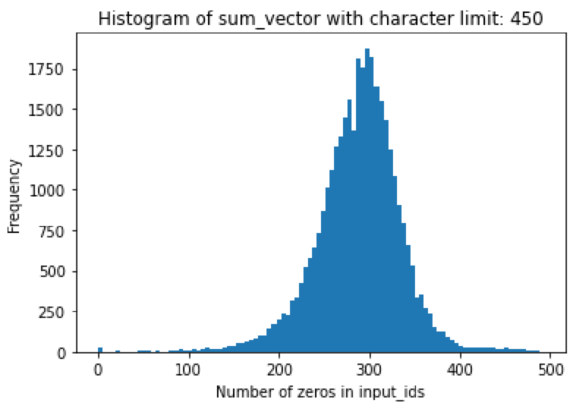
```
18928
512
```

Here are the histograms for the number of zeroes using different character limits:

```
plt.hist(sum_vector, bins=100)
plt.show()
```



(a) Histogram with 800 character limit



(b) Histogram with 450 character limit

Figure 6: Histograms

We want to keep the number of truncated sequences to be lower than 1σ (preferably 2σ) while keeping the mean as close to 0 as possible.

5.1.4 Before Training

We can do a primitive for loop count of all of our labels, which gives us:

```
akp_count: 1390
chp_count: 1607
hdp_count: 4760
mhp_count: 4725
total_label_count: 12482
12482
```

Let's import our model:

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=len
                                        (np.unique(labels)),cache_dir=cache_dump)
```

This imports the pre-trained model. "num_labels" is the number of labels we have, in this case 4. "cache_dir" is the path where the model will be saved.
Using the seed value we set earlier, create a recreatable split of the data into training, validation and test sets:

```
random.seed(seed_val)
torch.manual_seed(seed_val)
```

Now, we have to convert our lists into tensors, then load them into a TensorDataset.

```
input_ids = [torch.tensor(ids) for ids in input_ids]
attention_masks = [torch.tensor(masks) for masks in attention_masks]
token_type_ids = [torch.tensor(ids) for ids in token_type_ids]

input_ids = torch.stack(input_ids, dim=0)
attention_masks = torch.stack(attention_masks, dim=0)
token_type_ids = torch.stack(token_type_ids, dim=0)

le = LabelEncoder()
labels = le.fit_transform(labels)
labels = torch.tensor(labels)
```

Our labels now have been encoded as 0, 1, 2, 3. alphabetic order.
Now we can create our dataset:

```
dataset = TensorDataset(input_ids, attention_masks, token_type_ids, labels)
```

```
train_size = int(0.8 * len(dataset))  # 80% of the data will be used for training
val_size = int(0.199* len(dataset))  # 19.9% of the data will be used for validation
test_size = len(dataset) - train_size - val_size  # 0.1% of the data will be used for
                                                 testing

print(train_size)
print(val_size)
print(test_size)

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size,
                                                 test_size])
```

Load the data into a DataLoader for batching:

```
train_dataloader = DataLoader(train_dataset, sampler=torch.utils.data.RandomSampler(
                                                 train_dataset), batch_size=batch_size)
validation_dataloader = DataLoader(val_dataset, sampler=torch.utils.data.
                                                 SequentialSampler(val_dataset), batch_size
                                                 =batch_size)
test_dataloader = DataLoader(test_dataset, sampler=torch.utils.data.SequentialSampler(
                                                 test_dataset), batch_size=batch_size)
```

Initialize the optimizer:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5, eps=1e-8)
```

We are using the AdamW optimizer. "lr" is the learning rate. "eps" is the epsilon value, it is a very small number to prevent division by 0.
We will use a linear scheduler with warmup as described earlier:

```
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0,
                                                 num_training_steps=total_steps)
```

Our loss function is the function that determines how farther away our predictions are from the actual values. We will use the Cross Entropy Loss function:

```
loss_fn = torch.nn.CrossEntropyLoss()
```

Now, we define the training loop:

```
def train(model, train_dataloader, validation_dataloader, optimizer, scheduler, epochs,
                                                 loss_fn):
    model = model.to(device)
    for epoch in range(epochs):
        print('======== Epoch {:} / {:} ========'.format(epoch + 1, epochs))

        # Training
        model.train()
        total_loss, total_accuracy = 0, 0
        total_preds=[]
        for step, batch in enumerate(train_dataloader):

            #print("Batch size:", len(batch))
            #print("Batch contents:", batch)
            b_input_ids, b_attn_mask, b_token_type_ids, b_labels = tuple(t.to(device) for
                                                              t in batch)
            model.zero_grad()
            outputs = model(b_input_ids, attention_mask=b_attn_mask, token_type_ids=
                                                            b_token_type_ids, labels=
                                                            b_labels)
            #to spot the error let's print out the shape of the output tensor and the
                                                            label tensor to make sure they
                                                             match up
            #print(outputs[0].shape)
            #print(outputs.logits.shape)
            #print(b_labels.shape)
            #print(outputs)
            loss = loss_fn(outputs[1], b_labels)
            logits = outputs[1]
            preds = torch.argmax(logits, dim=1)
            total_loss += loss.item()
            total_accuracy += accuracy_score(preds.cpu().numpy(), b_labels.cpu().numpy())
            total_preds.append(preds.cpu().numpy())
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler.step()
            #every 20th iteration print step
            print("Step: " + str(step))
            #print("Batch: " + str(batch))

        avg_train_loss = total_loss / len(train_dataloader)
        avg_train_accuracy = total_accuracy / len(train_dataloader)
        total_preds = np.concatenate(total_preds, axis=0)

        print('Training loss: {:.4f}'.format(avg_train_loss))
        print('Training accuracy: {:.4f}'.format(avg_train_accuracy))

        # Validation
        model.eval()
        total_loss, total_accuracy = 0, 0
        total_preds = []
        for step, batch in enumerate(validation_dataloader):


            b_input_ids, b_attn_mask, b_token_type_ids, b_labels = tuple(t.to(device) for
                                                         t in batch)
            with torch.no_grad():
                outputs = model(b_input_ids, attention_mask=b_attn_mask, token_type_ids=
                                                               b_token_type_ids, labels=
                                                               b_labels)
                loss = loss_fn(outputs[1], b_labels)
                logits = outputs[1]
                preds = torch.argmax(logits, dim=1)
                total_loss += loss.item()
                total_accuracy += accuracy_score(preds.cpu().numpy(), b_labels.cpu().
                                                               numpy())
                total_preds.append(preds.cpu().numpy())

        avg_val_loss = total_loss / len(validation_dataloader)
        avg_val_accuracy = total_accuracy / len(validation_dataloader)
        total_preds = np.concatenate(total_preds, axis=0)

        print('Validation loss: {:.4f}'.format(avg_val_loss))
        print('Validation accuracy: {:.4f}'.format(avg_val_accuracy))
    return model
```

This function works as follows:

- It takes the model, training and validation dataloaders, optimizer, scheduler, epochs and loss function as inputs.
- It moves the model to the GPU.
- It loops through the epochs and prints the epoch number.
- It loops through the training dataloader to get the batches.
- It moves the batch to the GPU.
- It sets the gradients to zero to prevent gradient accumulation.
- It feeds the batch to the model. The model is the one we imported earlier, BertForSequenceClassification.
- It calculates the loss between the predictions and the labels.
- It calculates the accuracy.

– It calculates the predictions.

– It calculates the gradients of the loss with respect to the parameters.

– It clips the gradients to prevent exploding gradients. This is done by setting the maximum gradient to 1.0.

– It updates the parameters according to the gradients, it does this by using the optimizer.

– It loops through the validation dataloader.

– It moves the batch to the GPU.

– It feeds the batch to the model.

– It calculates the loss.

– It calculates the accuracy.

– It calculates the predictions.

– It returns the model.

Clear cache before beginning so that we don't run out of memory:

```
torch.cuda.empty_cache()
```

5.1.5 Training

Just calling the train function will begin the training. We don't have to assign it to a variable.

```
train(model, train_dataloader, validation_dataloader, optimizer, scheduler, epochs,
                                                  loss_fn)
```

Example output:

```
======== Epoch 1 / 4 ========
Step: 0
Step: 1
Step: 2
Step: 3
Step: 4
Step: 5
Step: 6
Step: 7
Step: 8
Step: 9
Step: 10
Step: 11
Step: 12
Step: 13
Step: 14
Step: 15
Step: 16
Step: 17
Step: 18
Step: 19
Step: 20
Step: 21
Step: 22
Step: 23
...
Training loss: 0.1579
Training accuracy: 0.9731
Validation loss: 0.4157
Validation accuracy: 0.9380
```

A higher validation accuracy means that the model is performing better. If validation accuracy starts to decrease, it means that the model is overfitting which is not desirable.
All that is left is to save and test the model.

```
model.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)
```

This saves the model and the tokenizer to the output directory.
To load the model:

```
saved_model = BertForSequenceClassification.from_pretrained(output_dir)
saved_model.to(device)
```

5.1.6 Evaluation

Define the functions necessary for evaluation:

```
def evaluate(model, dataloader):
    model.eval()
    total_preds = []

    for batch in dataloader:
        b_input_ids, b_attn_mask, b_token_type_ids = tuple(t.to(device) for t in
                                                           batch)
        with torch.no_grad():
            outputs = model(b_input_ids, attention_mask=b_attn_mask, token_type_ids=
                                                               b_token_type_ids)
            logits = outputs[0]
            preds = torch.argmax(logits, dim=1)

        total_preds.append(preds.cpu().numpy())

    total_preds = np.concatenate(total_preds, axis=0)



    return total_preds
```

Initially this function was more primitive with manually entered labels, i am only including the updated version which uses the labels dictionary we created earlier.

```
def party_affiliation(model, dataloader):
    total_preds = evaluate(model, dataloader)

    party_counts = {}
    party_percentages = {}

    for label in labels_dict.values():
        party_counts[label] = 0

    for pred in total_preds:
        party_label = labels_dict[pred]
        party_counts[party_label] += 1

    total_samples = len(total_preds)

    for label, count in party_counts.items():
        party_percentages[label] = count / total_samples * 100

    for label, percentage in party_percentages.items():
        if percentage != 0:
            print(f"{label} Prediction Percentage: {percentage}%")
```

Evaluate the model on the test set:

```
preds, labels = evaluate(saved_model, test_dataloader)

# Print the predictions and labels
print("Predictions:", preds)
print("Labels:", labels)

# Print the overall accuracy
accuracy = accuracy_score(labels, preds)
print("Accuracy: {:.2f}".format(accuracy))
```

Output:

```
Predictions: [2 2 2 3 3 2 3 3 1 3 3 3 3 2]
Labels: [2 2 2 3 0 2 3 3 1 3 3 3 3 3]
Accuracy: 0.86
```

Let's engineer this so that we can use the model on any text we want. First the function to preprocess any txt file from any file path we wish to specify:

```
def preprocess_file(file_path, tokenizer, max_length):
    with open(file_path, 'r', encoding='utf-8') as file:
        whole_text= file.read()
        whole_text = whole_text.replace('\n', ' ')
        sentences= nltk.sent_tokenize(whole_text) #split the speech by punctuation

        input_ids = []
        attention_masks = []
        token_type_ids = []

        while_check1 = 1
        while_check2 = 1
        all_sentences_of_1file = []
        counter_i= 1
        sentence_to_add = sentences[0]
        while while_check1 == 1:
            while_check2 = 1
            while while_check2 == 1:

                if counter_i+1 < len(sentences):

                    if len(sentence_to_add) + len(sentences[counter_i]) >
                                                                character_limit:
                        counter_i += 1
                        sentence_to_add = sentence_to_add + " " + sentences[counter_i
                                                                    ]
                        all_sentences_of_1file.append(sentence_to_add)
                        sentence_to_add = sentences[counter_i]
                        while_check2 = 0
                    else:
                        counter_i += 1
                        #add sentences[counter_i] to sentence_to_add with a space
                        sentence_to_add = sentence_to_add + " " + sentences[counter_i
                                                                    ]
                        #print(counter_i)
                        #print(sentence_to_add)
                else:
                    all_sentences_of_1file.append(sentence_to_add)
                    while_check2 = 0
                    break


            #if counter_i exceeds index length of sentences, break the loop:
            if counter_i+1 >= len(sentences):

                while_check1 = 0
                break
        for sentence in all_sentences_of_1file:
            tokenized_sentence = tokenize_text(sentence)
            input_ids.append(tokenized_sentence['input_ids'][0])
            attention_masks.append(tokenized_sentence['attention_mask'][0])
            token_type_ids.append(tokenized_sentence['token_type_ids'][0])

    input_ids = [torch.tensor(ids) for ids in input_ids]
    attention_masks = [torch.tensor(masks) for masks in attention_masks]
    token_type_ids = [torch.tensor(ids) for ids in token_type_ids]

    input_ids = torch.stack(input_ids, dim=0)
    attention_masks = torch.stack(attention_masks, dim=0)
    token_type_ids = torch.stack(token_type_ids, dim=0)

    dataset = TensorDataset(input_ids, attention_masks, token_type_ids)

    dataloader = DataLoader(dataset, sampler=torch.utils.data.SequentialSampler(
                                                    dataset), batch_size=batch_size)

    return dataloader
```

Define an extra:

```
def party_monster():
    file_path = input("Please enter the file path of the file you want to analyze: (
                                                    MUST BE TXT)")
    preprocessed_file = preprocess_file(file_path, tokenizer, max_length)
    party_affiliation(saved_model, preprocessed_file)
```

5.1.7 Results

Now, we have tweets of certain politicians saved on our computer. Let's see what the model thinks about them:

```
#Canan Kaftanc o lu
CHP Prediction Percentage: 0.5833333333333334
AKP Prediction Percentage: 0.16666666666666666
HDP Prediction Percentage: 0.25

#Muharrem  nce
CHP Prediction Percentage: 0.8
AKP Prediction Percentage: 0.012
HDP Prediction Percentage: 0.144
MHP Prediction Percentage: 0.044

#Recep Tayyip Erdo an
CHP Prediction Percentage: 0.3958333333333333
AKP Prediction Percentage: 0.5104166666666666
HDP Prediction Percentage: 0.017361111111111112
MHP Prediction Percentage: 0.0763888888888889

#Devlet Bah eli
CHP Prediction Percentage: 0.0029069767441860465
AKP Prediction Percentage: 0.005813953488372093
HDP Prediction Percentage: 0.011627906976744186
MHP Prediction Percentage: 0.9796511627906976

#Selahattin Demirta
CHP Prediction Percentage: 0.4460093896713615
AKP Prediction Percentage: 0.009389671361502348
HDP Prediction Percentage: 0.539906103286385
MHP Prediction Percentage: 0.004694835680751174

#Kemal K l  daro lu
CHP Prediction Percentage: 0.7946428571428571
AKP Prediction Percentage: 0.017857142857142856
HDP Prediction Percentage: 0.15625
MHP Prediction Percentage: 0.03125

#Sinan O an
CHP Prediction Percentage: %69.5945945945946
AKP Prediction Percentage: %0.33783783783783783
HDP Prediction Percentage: %24.324324324324326
MHP Prediction Percentage: %5.743243243243244
```

These evaluations are not clearly indicative of the political affiliation of the person in question since the model was trained on speeches but we are evaluating tweets: We can see that it generally performs well still, however since we have only 4 political parties, the model can have problem identifying outlying ideologies such as nationalists who oppose MHP and the government such as Sinan Oğan, depicting him with a high HDP percentage and a low MHP percentage despite him being ideologically closer to MHP than HDP. Next, let's try to upscale the model.

5.2 2nd attempt with 8 parties

The main change for this attempt was the increase in the number of parties. The parties we will be using are:

* AKP
* CHP

- ∗ HDP
- ∗ MHP
- ∗ İYİ
- ∗ Zafer
- ∗ Saadet
- ∗ TİP (Türkiye İşçi Partisi)

Since some of the parties have a limited number of speeches and statements online, they are supplemented with tweets from the official twitter accounts of their leaders or mps. The existing parties have also been supplemented where necessary. Here is how we get the tweets for any specific party:

```
import snscrape.modules.twitter as sntwitter
import pandas as pd

def write_as_txt_for_"enter party name here" (username, limit): #specify party name
                                                            for function name
    tweets=[]
    query="from:"+username +" until:2023-06-06 since:2010-01-01"  #specify date the
                                                                party was founded or any other
                                                                convenient date
    for tweet in sntwitter.TwitterSearchScraper(query).get_items():
        if len(tweets) == limit:
            break
        else:
            tweets.append([tweet.date, tweet.username, tweet.content])
    df = pd.DataFrame(tweets, columns=['Date', 'User', 'Tweet'])
    tweet_content = df['Tweet']
    tweet_content.to_csv(username+'.csv', index=False)
    print("len of tweet_content: ")
    print(len(tweet_content))

    tweet_content = tweet_content.str.cat(sep=' ')
    file_path = r"E:\transformerhold\classification_project\enter party name here\\"
                                                                #specify party name for file path
    with open(file_path + username+'.txt', 'w', encoding='utf-8') as f:
        f.write(tweet_content)
```

This function pulls tweets from the specified user and writes them to a txt file. The tweets are written in a single line, separated by spaces. Further ahead we will add more pre-processing to the files to provide better results. There are a total of 41201 tokens this time around.
As a step towards getting a better output. We define the following function:

```
def train_until_overfit(model, train_dataloader, validation_dataloader, optimizer,
                                                            scheduler, loss_fn):
    model = model.to(device)
    Validation_vector = [0.]
    epoch_break = 0
    epoch = 0
    while epoch_break == 0:
        print('======== Epoch {:} / {:} ========'.format(epoch + 1, epochs))
        epoch += 1
        # Training
        model.train()
        total_loss, total_accuracy = 0, 0
        total_preds=[]
        for step, batch in enumerate(train_dataloader):
            b_input_ids, b_attn_mask, b_token_type_ids, b_labels = tuple(t.to(device)
                                                                    for t in batch)
            model.zero_grad()
            outputs = model(b_input_ids, attention_mask=b_attn_mask, token_type_ids=
                                                                b_token_type_ids, labels=
                                                                b_labels)
            loss = loss_fn(outputs[1], b_labels)
            logits = outputs[1]
            preds = torch.argmax(logits, dim=1)
            total_loss += loss.item()
            total_accuracy += accuracy_score(preds.cpu().numpy(), b_labels.cpu().
                                                                numpy())
            total_preds.append(preds.cpu().numpy())
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler.step()
            #every 20th iteration print step
            if step % 200 == 0:
                print("Step: " + str(step))
        avg_train_loss = total_loss / len(train_dataloader)
        avg_train_accuracy = total_accuracy / len(train_dataloader)
        total_preds = np.concatenate(total_preds, axis=0)

        print('Training loss: {:.4f}'.format(avg_train_loss))
        print('Training accuracy: {:.4f}'.format(avg_train_accuracy))

        # Validation
        model.eval()
        total_loss, total_accuracy = 0, 0
        total_preds = []
        for step, batch in enumerate(validation_dataloader):
            b_input_ids, b_attn_mask, b_token_type_ids, b_labels = tuple(t.to(device)
                                                            for t in batch)
            with torch.no_grad():
                outputs = model(b_input_ids, attention_mask=b_attn_mask,
                                                               token_type_ids=
                                                               b_token_type_ids,
                                                               labels=b_labels)
                loss = loss_fn(outputs[1], b_labels)
                logits = outputs[1]
                preds = torch.argmax(logits, dim=1)
                total_loss += loss.item()
                total_accuracy += accuracy_score(preds.cpu().numpy(), b_labels.cpu().
                                                               numpy())
                total_preds.append(preds.cpu().numpy())

        avg_val_loss = total_loss / len(validation_dataloader)
        avg_val_accuracy = total_accuracy / len(validation_dataloader)
        total_preds = np.concatenate(total_preds, axis=0)

        print('Validation loss: {:.4f}'.format(avg_val_loss))
        print('Validation accuracy: {:.4f}'.format(avg_val_accuracy))
        #{:.4f}: 4 digits after the decimal point
        # .format: formats the string
        Validation_vector.append(avg_val_accuracy)
        #if
        if Validation_vector[-1] - Validation_vector[-2] < 0.05:
            epoch_break = 1
            print("Training stopped because validation accuracy did not improve more
                                                           than 0.05")
    return model
```

This function is almost identical with the previous training function. The only difference is that it stops training when the validation accuracy does not improve more than 0.05. After 3 apochs, it gives us:

```
Training accuracy: 0.9179
Validation loss: 0.6742
Validation accuracy: 0.9060
```

which is a great result for having so many classes. Here are its performances using the tweets of the party leaders:

```
#Canan Kaftanc o lu
CHP Prediction Percentage: 95.0381679389313%
MHP Prediction Percentage: 1.1450381679389312%
saadet Prediction Percentage: 1.1450381679389312%
tip Prediction Percentage: 2.6717557251908395%

#Muharrem  nce
CHP Prediction Percentage: 51.79282868525896%
HDP Prediction Percentage: 1.1952191235059761%
iyip Prediction Percentage: 0.398406374501992%
MHP Prediction Percentage: 27.49003984063747%
saadet Prediction Percentage: 10.358565737051793%
tip Prediction Percentage: 1.1952191235059761%
zafer Prediction Percentage: 7.569721115537849%
```

```
#Recep Tayyip Erdo an
AKP Prediction Percentage: 34.58904109589041%
CHP Prediction Percentage: 4.794520547945205%
iyip Prediction Percentage: 0.3424657534246575%
MHP Prediction Percentage: 25.684931506849317%
saadet Prediction Percentage: 25.684931506849317%
tip Prediction Percentage: 0.3424657534246575%
zafer Prediction Percentage: 8.561643835616438%

#Devlet Bah eli
AKP Prediction Percentage: 1.4534883720930232%
CHP Prediction Percentage: 0.29069767441860467%
iyip Prediction Percentage: 88.66279069767442%
MHP Prediction Percentage: 2.0348837209302326%
saadet Prediction Percentage: 4.069767441860465%
zafer Prediction Percentage: 3.488372093023256%

#Sinan O an
CHP Prediction Percentage: 29.054054054054053%
HDP Prediction Percentage: 0.6756756756756757%
MHP Prediction Percentage: 22.635135135135133%
saadet Prediction Percentage: 12.162162162162163%
tip Prediction Percentage: 0.33783783783783783%
zafer Prediction Percentage: 35.13513513513514%

#Kemal K l  daro lu
AKP Prediction Percentage: 0.4484304932735426%
CHP Prediction Percentage: 81.16591928251121%
HDP Prediction Percentage: 0.4484304932735426%
MHP Prediction Percentage: 9.865470852017937%
saadet Prediction Percentage: 5.381165919282512%
tip Prediction Percentage: 1.345291479820628%
zafer Prediction Percentage: 1.345291479820628%
```

This model seems to perform better and understand the distinctions better. However, there is a problematic False Positive for Devlet Bahçeli, who is the head of MHP, but the model predicts him as İYİ, which is an off shoot of MHP. Also, while getting most of the predictions right, the model is not very confident in its predictions.

Now let's try some famous texts from Turkish canon on this model. We will input 2 different versions of Nutuk by Atatürk into the model and observe their outputs.

```
preprocessed_text = preprocess_file("E:\\transformerhold\\test\\nutuk1.txt",
                                    tokenizer, max_length)

AKP Prediction Percentage: 3.56%
CHP Prediction Percentage: 34.97%
HDP Prediction Percentage: 2.34%
iyip Prediction Percentage: 3.02%
MHP Prediction Percentage: 6.72%
saadet Prediction Percentage: 46.90%
tip Prediction Percentage: 0.19%
zafer Prediction Percentage: 2.29%
```

```
preprocessed_text = preprocess_file("E:\\transformerhold\\test\\nutuk2.txt",
                                    tokenizer, max_length)

AKP Prediction Percentage: 3.61%
CHP Prediction Percentage: 7.17%
HDP Prediction Percentage: 0.38%
iyip Prediction Percentage: 5.19%
MHP Prediction Percentage: 14.86%
saadet Prediction Percentage: 67.48%
zafer Prediction Percentage: 1.30%
```

We can see that even the change of editor and publisher can vastly affect the result, while we cannot be sure until we manually check the differences between the two versions, this tells us that we should be cautious with the classifications of this model.

5.3 3rd attempt with 7 parties

For this attempt, further data cleaning was done by adding the following commands to the tweet scraping functions:

```
file_name = r"E:\transformerhold\classification_project\test\\" + username + ".txt"

with open(file_name, "r", encoding="utf-8") as f:
    lines = f.readlines()

    output = []
    for line in lines:
        words = line.split() #line.split() splits the line into words
        new_words = []
        for word in words:
            #find instance of @, http or # and delete rest of line.
            if word.startswith("@") or word.startswith("http") or word.startswith("#"
                                                                     ):

                break
            else:
            #find index of unwanted characters ((@, http or #))
                if "@" in word:
                    index = word.index("@")
                elif "http" in word:
                    index = word.index("http")
                elif "#" in word:
                    index = word.index("#")
                else:
                    index = len(word)
                #add word to new_words
                new_words.append(word[:index])
        new_line = " ".join(new_words)
        output.append(new_line)


with open(file_name, "w", encoding="utf-8") as f:
    for line in output:
        f.write(line + "\n")
```

This prevents the model from training on links, hashtags and mentions and associating them with a certain party. Also since it has very little amount of text in terms of speeches and statements and mostly tweets from its mps which contain many non-political messages, Saadet Partisi is removed from the model. This time we have a total of 33856 tokens. We also define a hyper-parameter that can be changed fromthe top of the page for the value of the validation plateu.

```
Validation_plateau =0.02
```

This is for the part in train_until_overfit function where

```
...
 if Validation_vector[-1] - Validation_vector[-2] < Validation_plateau:
            epoch_break = 1
            print("Training stopped because validation accuracy did not improve more
                                                                 than Validation_plateau")
...
```

After training for 4 epochs we get:

```
Training loss: 0.2319
Training accuracy: 0.9637
Validation loss: 0.5822
Validation accuracy: 0.92
```

an improvement compared to our earlier score. We want to integrate the tweet scraping into our evaluation so we can just type the username and get an analysis:

```
def write_as_txt(username, limit):
    tweets=[]
    query="from:"+username +" until:2023-06-06 since:2010-01-01"
    for tweet in sntwitter.TwitterSearchScraper(query).get_items():
        if len(tweets) == limit:
            break
        else:
            tweets.append([tweet.date, tweet.username, tweet.content])
    df = pd.DataFrame(tweets, columns=['Date', 'User', 'Tweet'])
    tweet_content = df['Tweet']
    tweet_content.to_csv(username+'.csv', index=False)
    tweet_content = tweet_content.str.cat(sep=' ')
    file_path = "E:\\transformerhold\\test\\"
    with open(file_path + username+'.txt', 'w', encoding='utf-8') as f:
        f.write(tweet_content)
```

and then define:

```
def party_monster_ultimate(username, limit):
    write_as_txt(username,limit)
    file_path = "E:\\transformerhold\\test\\" + username+".txt"
    preprocessed_file = preprocess_file(file_path, tokenizer, max_length)
    party_affiliation(saved_model, preprocessed_file)
```

Let's see how it performs:

```
#Canan Kaftanc o lu
party_monster_ultimate("Canan_Kaftanci", 1000)
AKP Prediction Percentage: 0.38167938931297707%
CHP Prediction Percentage: 68.32061068702289%
HDP Prediction Percentage: 4.580152671755725%
tip Prediction Percentage: 7.633587786259542%
zafer Prediction Percentage: 19.083969465648856%

#Muharrem  nce
party_monster_ultimate("vekilince", 1000)
AKP Prediction Percentage: 1.593625498007968%
CHP Prediction Percentage: 33.86454183266932%
HDP Prediction Percentage: 8.764940239043826%
iyip Prediction Percentage: 1.9920318725099602%
MHP Prediction Percentage: 17.131474103585656%
tip Prediction Percentage: 2.788844621513944%
zafer Prediction Percentage: 33.86454183266932%

#Meral Ak ener
party_monster_ultimate("meral_aksener", 1000)
CHP Prediction Percentage: 1.1363636363636365%
MHP Prediction Percentage: 83.33333333333334%
tip Prediction Percentage: 1.1363636363636365%
zafer Prediction Percentage: 14.393939393939394%

#Recep Tayyip Erdo an
party_monster_ultimate("RTErdogan", 1000)
AKP Prediction Percentage: 38.513513513513516%
CHP Prediction Percentage: 2.7027027027027026%
HDP Prediction Percentage: 1.3513513513513513%
iyip Prediction Percentage: 0.6756756756756757%
MHP Prediction Percentage: 5.743243243243244%
tip Prediction Percentage: 1.3513513513513513%
zafer Prediction Percentage: 49.66216216216216%

#Devlet Bah eli
party_monster_ultimate("dbdevletbahceli", 1000)
iyip Prediction Percentage: 97.96511627906976%
MHP Prediction Percentage: 0.29069767441860467%
tip Prediction Percentage: 0.29069767441860467%
zafer Prediction Percentage: 1.4534883720930232%

#Selahattin Demirta
party_monster_ultimate("hdpdemirtas", 1000)
AKP Prediction Percentage: 0.38022813688212925%
CHP Prediction Percentage: 0.38022813688212925%
HDP Prediction Percentage: 92.39543726235742%
iyip Prediction Percentage: 0.38022813688212925%
MHP Prediction Percentage: 0.38022813688212925%
tip Prediction Percentage: 4.562737642585551%
zafer Prediction Percentage: 1.520912547528517%

#Kemal K l  daro lu
party_monster_ultimate("kilicdarogluk", 1000)
CHP Prediction Percentage: 58.03571428571429%
HDP Prediction Percentage: 10.714285714285714%
MHP Prediction Percentage: 2.6785714285714284%
```

```
tip Prediction Percentage: 4.017857142857143%
zafer Prediction Percentage: 24.553571428571427%

#Sinan O an
party_monster_ultimate("DrSinanOgan", 1000)
AKP Prediction Percentage: 0.6779661016949152%
CHP Prediction Percentage: 11.525423728813559%
HDP Prediction Percentage: 3.728813559322034%
MHP Prediction Percentage: 2.711864406779661%
tip Prediction Percentage: 1.0169491525423728%
zafer Prediction Percentage: 80.33898305084746%
```

As we can see some of our pathologies still persist while we are seeing great improvements in other regards. One of the main changes we can notice is that after removing the party with the least amount of official long-text, Saadet, the party with the second least amount of official long-text, Zafer, has now begun to be over-represented. This can be fixed via a manual adjustment of the weights of the classes. However, this is not a very elegant solution.
Our pathologies regarding the duality of nationalism between MHP and İYİ still persists with the predictions of each party's leader being switched. Zafer's rhetoric however seems unaffected by this issue, perhaps due to them being focused on different areas in terms of messaging such as the issue of immigrants.
Let's now write functions utilize this model on any text that user can enter manually. First we define a modified preprocessing function:

```
def preprocess_text(whole_text, tokenizer, max_length):
     #split the speech by punctuation

    input_ids = []
    attention_masks = []
    token_type_ids = []

    while_check1 = 1
    while_check2 = 1
    all_sentences_of_1file = []
    counter_i= 1

    if len(whole_text) > character_limit:
        sentences= nltk.sent_tokenize(whole_text)
        sentence_to_add = sentences[0]
        while while_check1 == 1:
            while_check2 = 1
            while while_check2 == 1:

                if counter_i+1 < len(sentences):

                    if len(sentence_to_add) + len(sentences[counter_i]) >
                                                                    character_limit:
                        counter_i += 1
                        sentence_to_add = sentence_to_add + " " + sentences[counter_i
                                                                        ]
                        all_sentences_of_1file.append(sentence_to_add)
                        sentence_to_add = sentences[counter_i]
                        while_check2 = 0
                        #print("Exited while_check2")
                    else:
                        counter_i += 1
                    #add sentences[counter_i] to sentence_to_add with a space
                        sentence_to_add = sentence_to_add + " " + sentences[counter_i
                                                                        ]
                    #print(counter_i)
                    #print(sentence_to_add)
                else:
                    all_sentences_of_1file.append(sentence_to_add)
                    while_check2 = 0
                    #print("Exited while_check2")
                    break



            #if counter_i exceeds index length of sentences, break the loop:
                if counter_i+1 >= len(sentences):

                    while_check1 = 0
                    print("Exited while_check1")
                break
    else:
        all_sentences_of_1file.append(whole_text)

    for sentence in all_sentences_of_1file:
        tokenized_sentence = tokenize_text(sentence)
        input_ids.append(tokenized_sentence['input_ids'][0])
        attention_masks.append(tokenized_sentence['attention_mask'][0])
        token_type_ids.append(tokenized_sentence['token_type_ids'][0])

    input_ids = [torch.tensor(ids) for ids in input_ids]
    attention_masks = [torch.tensor(masks) for masks in attention_masks]
    token_type_ids = [torch.tensor(ids) for ids in token_type_ids]

    input_ids = torch.stack(input_ids, dim=0)
    attention_masks = torch.stack(attention_masks, dim=0)
    token_type_ids = torch.stack(token_type_ids, dim=0)

    dataset = TensorDataset(input_ids, attention_masks, token_type_ids)

    dataloader = DataLoader(dataset, sampler=torch.utils.data.SequentialSampler(
                                                dataset), batch_size=batch_size)

    return dataloader
```

This will tokenize and load any text that is entered into it. Now we define the function that will use the model on the text:

```
def text_party(target_text):
    preprocessed_text = preprocess_text(target_text, tokenizer, max_length)
    party_affiliation(saved_model, preprocessed_text)
```

This function can now be utilized to analyze any user input text.

Let's once again try the literary works:

```
preprocessed_text = preprocess_file("E:\\transformerhold\\test\\nutuk1.txt",
                                              tokenizer, max_length)

AKP Prediction Percentage: 6.48%
CHP Prediction Percentage: 74.77%
HDP Prediction Percentage: 3.26%
iyip Prediction Percentage: 4.53%
MHP Prediction Percentage: 6.53%
tip Prediction Percentage: 0.10%
zafer Prediction Percentage: 4.33%

preprocessed_text = preprocess_file("E:\\transformerhold\\test\\nutuk2.txt",
                                              tokenizer, max_length)

AKP Prediction Percentage: 9.67%
CHP Prediction Percentage: 44.30%
HDP Prediction Percentage: 1.78%
iyip Prediction Percentage: 9.62%
MHP Prediction Percentage: 28.09%
tip Prediction Percentage: 0.14%
zafer Prediction Percentage: 6.40%
```

We should note that the nutuk2.txt file was edited by an editor (Şule Perinçek) who has nationalistic sentiments, so this could constitute an interesting area of investigation or merely a coincidence produced by the model.

Let's try on more texts. Yaşar Kemal is known as a leftist writer, and some consider "İnce Memed" to be his magnum opus.

```
preprocessed_text = preprocess_file("E:\\transformerhold\\test\\ince_memed.txt",
                                              tokenizer, max_length)
party_affiliation(saved_model, preprocessed_text)

AKP Prediction Percentage: 15.18987341772152%
CHP Prediction Percentage: 24.050632911392405%
HDP Prediction Percentage: 7.088607594936709%
```

```
iyip Prediction Percentage: 1.89873417721519%
MHP Prediction Percentage: 14.810126582278482%
tip Prediction Percentage: 36.835443037974684%
zafer Prediction Percentage: 0.12658227848101267%
```

We can see that despite being overrepresented in other examples, Zafer is very low here and the socialist party gets a good score.

For fun, let's conclude with "Dune" by Frank Herbert.

```
preprocessed_text = preprocess_file("E:\\transformerhold\\test\\dune.txt", tokenizer,
                                                max_length)
party_affiliation(saved_model, preprocessed_text)

AKP Prediction Percentage: 1.414790996784566%
CHP Prediction Percentage: 80.0%
HDP Prediction Percentage: 7.395498392282958%
iyip Prediction Percentage: 0.12861736334405144%
MHP Prediction Percentage: 5.080385852090032%
tip Prediction Percentage: 5.723472668810289%
zafer Prediction Percentage: 0.2572347266881029%
```

5.4 4th attempt by removing tweets

Final attempt was made with the idea that the initial training with no twitter content was quite powerful. Content from Zafer Partisi was removed along with any content that was scraped from twitter. This was done by manually deleting the files from the folder. With 6 labels and a total of 18928 tokens, the training results of the model after 2 epochs were as follows:

```
Training accuracy: 0.2402
Validation loss: 1.7209
Validation accuracy: 0.2472
```

The model automatically stopped training after 2 epochs due to validation accuracy not improving more than 0.02 and actually decreasing. This might suggest that even though it is contaminated data, the twitter content is still useful for the model to learn for larger amount of labels.

5.5 The PDF sorter

One final model that is worth demonstrating is the PDF sorter which shows the power of these type of models for very specific tasks. This time, we download 100 papers from Arxiv.org. These papers belong to 4 different sub-categories of the field of high energy physics. These categories are:

∗ hep-ph: Phenomenology

∗ hep-th: Theory

∗ hep-ex: Experiment

∗ hep-lat: Lattice

We will train the model on very few amount of the tokens provided by these papers and show that they are still quite effective even with little amount of training.

First convert the pdfs into txt files. One can also use code to directly read from the pdfs.

```
dir_path = 'E:\\transformerhold\\paper_classification_txt\\'
pdf_path = 'E:\\transformerhold\\paper_classification\\'

def pdf_to_text(pdf_path, output_path):
    try:
        with open(pdf_path, 'rb') as file:
            reader = PyPDF2.PdfReader(file)
            text = ''
            for page in reader.pages:
                text += page.extract_text()
        with open(output_path, 'w', encoding='utf-8') as output_file:
            output_file.write(text)
    except Exception as e:
        print(f"Error reading PDF file: {pdf_path}")
        print(str(e))

for sub_folder in os.listdir(pdf_path):
    sub_folder_path = os.path.join(pdf_path, sub_folder)
    if os.path.isdir(sub_folder_path):
        new_folder_path = os.path.join(dir_path, sub_folder)
        if not os.path.exists(new_folder_path):
            os.makedirs(new_folder_path)
        for file in os.listdir(sub_folder_path):
            file_path = os.path.join(sub_folder_path, file)
            if file.endswith('.pdf'):
                output_path = os.path.join(new_folder_path, file.replace('.pdf',
                                                                    '.txt'))
                pdf_to_text(file_path, output_path)
```

The remaining tokenization and similar steps are the same as the previous analyses and won't be repeated again to avoid clutter. This process will give us 15276 tokens. Use a small training set compared to total data.

```
train_size = int(0.2 * len(dataset))
val_size = int(0.05* len(dataset))
test_size = len(dataset) - train_size - val_size

print(train_size)
print(val_size)
print(test_size)

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size,
                                                val_size, test_size])

3055
763
11458
```

So we will use one fifth of data to see if it can classify the rest. After training for two epochs we get:

```
Training loss: 0.6099
Training accuracy: 0.8897
Validation loss: 1.0219
Validation accuracy: 0.8296
```

Which is a great result for such a small training set. Let's see how it performs on the test set:

```
saved_model.eval()
total_loss, total_accuracy = 0, 0
total_preds = []
for step, batch in enumerate(test_dataloader):
    b_input_ids, b_attn_mask, b_token_type_ids, b_labels = tuple(t.to(device)
                                                        for t in batch)
    with torch.no_grad():
        outputs = saved_model(b_input_ids, attention_mask=b_attn_mask,
                                                        token_type_ids=
                                                        b_token_type_ids, labels
                                                        =b_labels)
        loss = loss_fn(outputs[1], b_labels)
        logits = outputs[1]
        preds = torch.argmax(logits, dim=1)
        total_loss += loss.item()
        total_accuracy += accuracy_score(preds.cpu().numpy(), b_labels.cpu().
                                                        numpy())
        total_preds.append(preds.cpu().numpy())

avg_test_loss = total_loss / len(test_dataloader)
avg_test_accuracy = total_accuracy / len(test_dataloader)
total_preds = np.concatenate(total_preds, axis=0)

print('Test loss: {:.4f}'.format(avg_test_loss))
print('Test accuracy: {:.4f}'.format(avg_test_accuracy))
```

```
Test loss: 1.0604
Test accuracy: 0.8316
```

Not only is it quick to train, we were able to discern differences between the categories in a specific field only using roughly 20 articles.

6 Conclusion

In this project, we have shown that transformer models can be used for a variety of taks, from classifying political parties to classifying scientific papers. They can be used for sentiment analysis. They can be used for text generation We have also shown that they can perform well for very specific tasks with very little training data. The models displayed in this project can be further improved by increasing the amount of training data, doing maintanence on the data and further fine-tuning. Especially the text classification model can be utilized in a variety of ways, from analyzing profiles to tidying up clustered folders. Our basic text-generating model had already shown basic mimicking capabilites of human speech. The classifier models we trained showed that they are quite accurate for their respective tasks. Using pipeline, it is trivial to do sentiment analysis on any text. Whatever the task, it is clear that transformer models can perform surprisingly well.

References

[1] N. Arbel, Attention in rnns, 2019.

[2] A. Vaswani et al., Attention is all you need, 2017.

[3] A. Henriquez, Understanding attention in transformers models, 2021.

[4] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, 2017.