

BERT_Analysis_Team1

November 23, 2024

1 SMS Phishing Detection Using Advanced NLP Models

```
[1]: # Edit all the Markdown cells below with the appropriate information
# Run all cells, containing your code
# Save this Jupyter with the outputs of your executed cells
#
# PS: Save again the notebook with this outcome.
# PSPS: Don't forget to include the dataset in your submission
```

Team: * Abahana Zelalem (RNN Author) * Benjamin Arnosti (GPT Author) * Brandon Botezr (BERT Author)

Course: AI 574 – Natural Language Processing (Fall, 2024)

1.1 Problem Statement

The objective of this project is to develop an effective SMS phishing detection system by leveraging cutting-edge NLP models such as BERT and OpenAI LLMs. These models have revolutionized language understanding and exhibit the capability to detect phishing attempts in a more context-aware manner. By comparing their performance to traditional machine learning models such as RNNs, this project aims to highlight the strengths and limitations of each approach.

Our system will analyze both the textual content of SMS messages and any embedded features, such as URLs, email addresses, or phone numbers, to classify messages as phishing, smishing, spam, or legitimate (ham). While traditional models rely on handcrafted features and are generally faster, they may lack the sophistication needed to handle the complexities of modern phishing techniques. Transformer-based models, on the other hand, offer enhanced performance by capturing nuanced patterns in short messages, which is crucial in detecting sophisticated phishing attacks hidden in mobile text formats.

By comparing traditional and advanced models, the project will provide insights into which models strike the best balance between performance, interpretability, and computational efficiency for real-time phishing detection in mobile environments.

- **Keywords:** SMS, Phishing, Smishing, text, phone, Natural Language Processing

1.2 Data Collection

- Source (url): <https://data.mendeley.com/datasets/f45bkkt8pr/1/>

- Short Description : The dataset is a set of labelled text messages that have been collected for SMS Phishing research. It has 5971 text messages labeled as Legitimate (Ham) or Spam or Smishing. It includes 489 spam messages, 638 smishing messages, and 4844 ham messages. (Mishra, S., Soni, D., 2022)
- Keywords: SMS, Phishing, Smishing, text, phone

1.3 Required packages

- pandas
- numpy
- sklearn
- Torch/PyTorch
- transformers by HuggingFace
- datasets by HuggingFace
- matplotlib.pyplot
- imblearn
- These libraries can be installed via a Conda installer to create a virtual environment. For specific information on installing PyTorch, see: <https://pytorch.org/get-started/locally/>. For information on HuggingFace transformers or datasets, see: <https://huggingface.co/learn/nlp-course/chapter0/1?fw=pt>.

Users can also use the environment.yml file to create the **smishing** conda environment.

1.4 Library Imports

```
[2]: ## Your code begins here
import os

# Basic imports
import pandas as pd
import arrow
import numpy as np
import sklearn.utils

# PyTorch Imports
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

# May use transformers
import transformers
# Get BERT from HF as well as Trainer class
from transformers import AutoTokenizer, BertForSequenceClassification, Trainer, \
    TrainingArguments, DataCollatorWithPadding
```

```

from datasets import Dataset, load_dataset

# Sklearn Metric items and splitting
from sklearn import metrics
from sklearn.model_selection import train_test_split

# Plotting
import matplotlib.pyplot as plt

```

1.5 GPU Setup

```

[3]: # Trigger the GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'You are usisng {device} on a {torch.cuda.get_device_name()}')

```

You are usisng cuda on a NVIDIA GeForce GTX 1070.

1.6 Model and Tokenizer Loading

Models and tokenizers will be loaded here so they can be used later on.

```

[4]: # RNN Model?
      # Should these go here or should they have their own notebook?

```

```

[5]: # API Calls?

```

```

[6]: # BERT Model Loads
      # Load the BERT-base-cased model
BERT_model_name = 'google-bert/bert-base-cased'
tokenizer = AutoTokenizer.from_pretrained(BERT_model_name) # Keep on CPU
model = BertForSequenceClassification.from_pretrained(BERT_model_name,
↳ num_labels=3).to(device) # Move to GPU

```

```

c:\Users\btb51\miniconda3\envs\ai574_NLP\lib\site-
packages\transformers\tokenization_utils_base.py:1601: FutureWarning:
`clean_up_tokenization_spaces` was not set. It will be set to `True` by default.
This behavior will be depracted in transformers v4.45, and will be then set to
`False` by default. For more details check this issue:
https://github.com/huggingface/transformers/issues/31884

```

```

warnings.warn(
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at google-bert/bert-base-cased and are newly initialized:
['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.

```

```

[7]: tokenizer

```

```
[7]: BertTokenizerFast(name_or_path='google-bert/bert-base-cased', vocab_size=28996,
    model_max_length=512, is_fast=True, padding_side='right',
    truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token':
    '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'},
    clean_up_tokenization_spaces=True), added_tokens_decoder={
    0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False,
    normalized=False, special=True),
    100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False,
    normalized=False, special=True),
    101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False,
    normalized=False, special=True),
    102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False,
    normalized=False, special=True),
    103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False,
    normalized=False, special=True),
    }
```

```
[8]: model
```

```
[8]: BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(28996, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (intermediate): BertIntermediate(
        (dense): Linear(in_features=768, out_features=3072, bias=True)
        (intermediate_act_fn): GELUActivation()
      )
    )
  )
)
```

```

    )
    (output): BertOutput(
        (dense): Linear(in_features=3072, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
)
)
)
(pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
)
)
(dropout): Dropout(p=0.1, inplace=False)
(classifier): Linear(in_features=768, out_features=3, bias=True)
)

```

1.7 Data Preprocessing

We are using the (Mishra & Soni, 2022) dataset for the project. This dataset contains a collection of labeled SMS messages with labels indicating whether they are smishing or legitimate. While the dataset is largely clean, we have had to do some pre-processing to match ‘Spam’ and ‘spam’ as well as ‘Ham’ and ‘ham’ labels. Each entry includes the label (ham, spam, smishing), the SMS message, and if any URL, email address, or phone number is present. Table 1 shows a sample of the dataset. The dataset itself contains 5971 text messages of which 4844 are ham, 489 are spam, and 638 are smishing.

```

[9]: # Load the dataset from local directory
ds = pd.read_csv("../data/processed/Dataset_5971.csv")
ds.head(5)

```

```

[9]:
  LABEL      TEXT URL EMAIL PHONE
0    ham  Your opinion about me? 1. Over 2. Jada 3. Kusr...  No    No    No
1    ham  What's up? Do you want me to come online? If y...  No    No    No
2    ham                        So u workin overtime nigpun?  No    No    No
3    ham  Also sir, i sent you an email about how to log...  No    No    No
4  Smishing  Please Stay At Home. To encourage the notion o...  No    No    No

```

I believe I need to rename label to labels... I’ll do this later because I need to do some clever work in getting labels to be integers for the classification.

```

[10]: cols = {'LABEL':'labels'}
ds.rename(columns=cols, inplace=True)
ds.head(5)

```

```

[10]:
  labels      TEXT URL EMAIL PHONE
0    ham  Your opinion about me? 1. Over 2. Jada 3. Kusr...  No    No    No

```

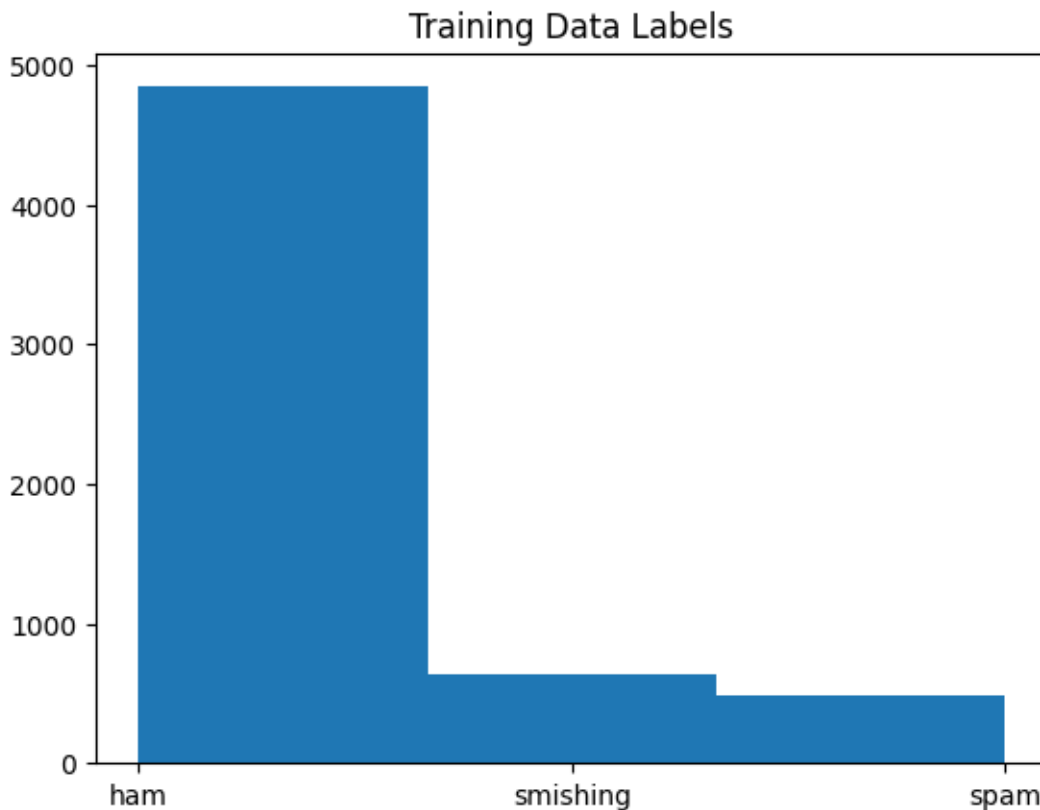
1	ham	What's up? Do you want me to come online? If y...	No	No	No
2	ham	So u workin overtime nigpun?	No	No	No
3	ham	Also sir, i sent you an email about how to log...	No	No	No
4	Smishing	Please Stay At Home. To encourage the notion o...	No	No	No

The labels Spam and spam are the same as well as Smishing and smishing. I'll push them all to lowercase.

```
[11]: ds['labels'] = ds['labels'].str.lower()
      ds.labels.unique()
```

```
[11]: array(['ham', 'smishing', 'spam'], dtype=object)
```

```
[12]: # A distribution of the sentence labels
      # This corresponds to the eventual binning of the 5 categories
      plt.hist(ds['labels'], bins=3)
      plt.title('Training Data Labels');
```



It can be seen by the histogram above the the data is heavily weighted to the ham values. We may need to balance this dataset.

I'm also curious just to see what the largest length of our data is. BERT will only take a 512.

```
[13]: max_len = 0
      for item in ds['TEXT']:
          if len(item) > max_len:
              max_len = len(item)

      print(f'The max text length is: {max_len}')
```

The max text length is: 910

This is longer than BERT's max input of 512. I wonder what the average length is for the dataset.

```
[14]: storage = 0
      for item in ds['TEXT']:
          storage += len(item)

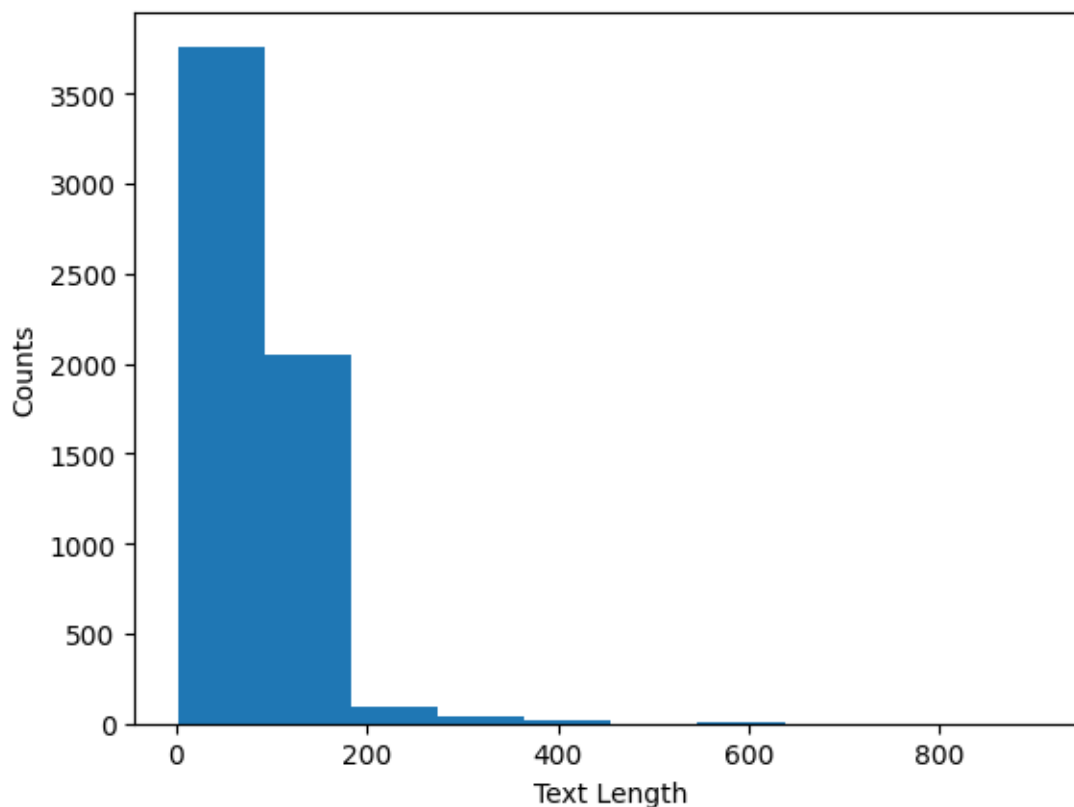
      print(f'The average text length is: {storage/len(ds):.2f}')
```

The average text length is: 83.24

That's a lot better for BERT. What's the distribution of lengths?

```
[15]: length_list = []
      for item in ds['TEXT']:
          length_list.append(len(item))

      plt.hist(length_list);
      plt.xlabel('Text Length')
      plt.ylabel('Counts');
```



The lengths are primarily smaller lengths. There will be a lot of Padding then.

```
[16]: # Set values to categories.
ds[['labels', 'URL', 'EMAIL', 'PHONE']] = ds[['labels', 'URL', 'EMAIL', 'PHONE']].astype('category')
ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5971 entries, 0 to 5970
Data columns (total 5 columns):
#   Column  Non-Null Count  Dtype
---  ---
0   labels  5971 non-null    category
1   TEXT    5971 non-null    object
2   URL     5971 non-null    category
3   EMAIL   5971 non-null    category
4   PHONE   5971 non-null    category
dtypes: category(4), object(1)
memory usage: 70.6+ KB
```



```
[17]: # Set category values to numeric values via codes
ds['labels'] = ds['labels'].cat.codes
ds['URL'] = ds['URL'].cat.codes
ds['EMAIL'] = ds['EMAIL'].cat.codes
ds['PHONE'] = ds['PHONE'].cat.codes
ds.head(5)
```

```
[17]:
```

	labels		TEXT	URL	EMAIL	\
0	0	Your opinion about me? 1. Over 2. Jada 3. Kusr...		0	0	
1	0	What's up? Do you want me to come online? If y...		0	0	
2	0	So u workin overtime nigpun?		0	0	
3	0	Also sir, i sent you an email about how to log...		0	0	
4	1	Please Stay At Home. To encourage the notion o...		0	0	

	PHONE
0	0
1	0
2	0
3	0
4	0

```
[18]: # Split the dataset into training, validation, and test sets.
train_ds, temp_ds = train_test_split(ds[['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE']],
    test_size=0.2, random_state=226)
val_ds, test_ds = train_test_split(temp_ds, test_size=0.3, random_state=226)

train_ds = train_ds.reset_index(drop=True)
val_ds = val_ds.reset_index(drop=True)
test_ds = test_ds.reset_index(drop=True)

print(f'Lengths of training: {len(train_ds)}')
print(f'Lengths of validation: {len(val_ds)}')
print(f'Lengths of test: {len(test_ds)}')
```

Lengths of training: 4776
 Lengths of validation: 836
 Lengths of test: 359

```
[19]: # Take another look at the dataset.
train_ds.head(5)
```

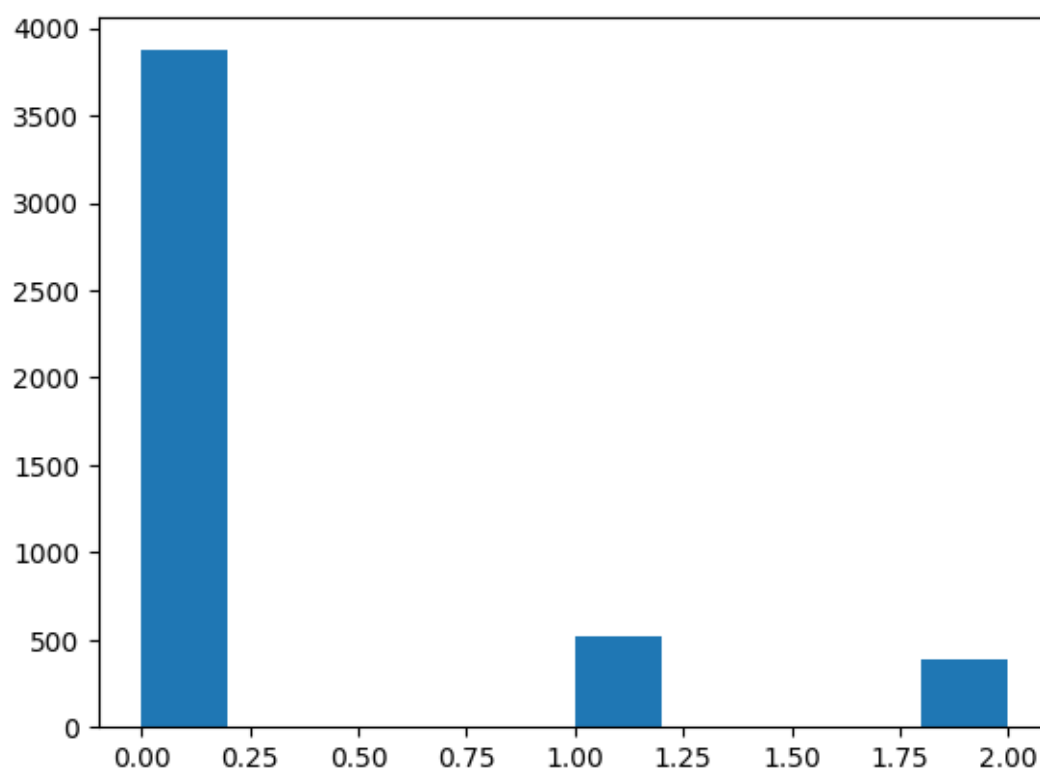
```
[19]:
```

	labels		TEXT	URL	EMAIL	\
0	0	Sorry i missed your call. Can you please call ...		0	0	
1	0	Pete,is this your phone still? Its Jenny from ...		0	0	
2	0	Let's pool our money together and buy a bunch ...		0	0	
3	0	Whenever ur sad, Whenever ur gray, Remembr im he...		0	0	
4	2	Want 2 get laid tonight? Want real Dogging loc...		0	0	

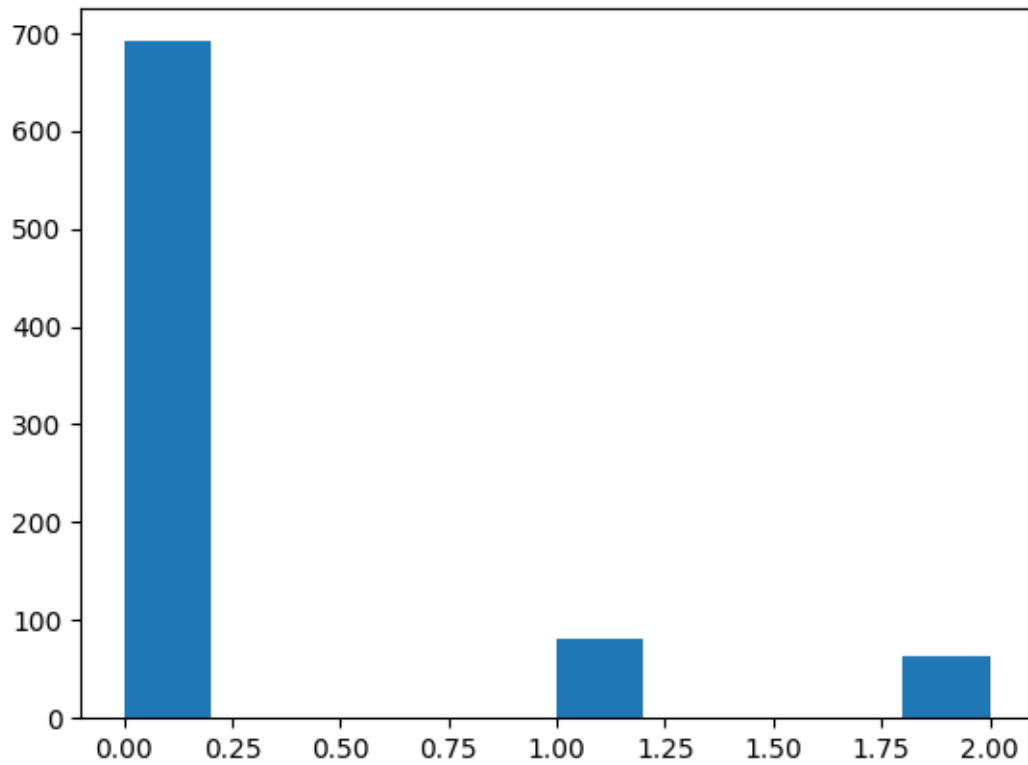
	PHONE
0	0
1	0
2	0
3	0
4	1

How do the distributions between the train and test set look?

```
[20]: # Training distribution between ham, spam, smish  
plt.hist(train_ds['labels']);
```

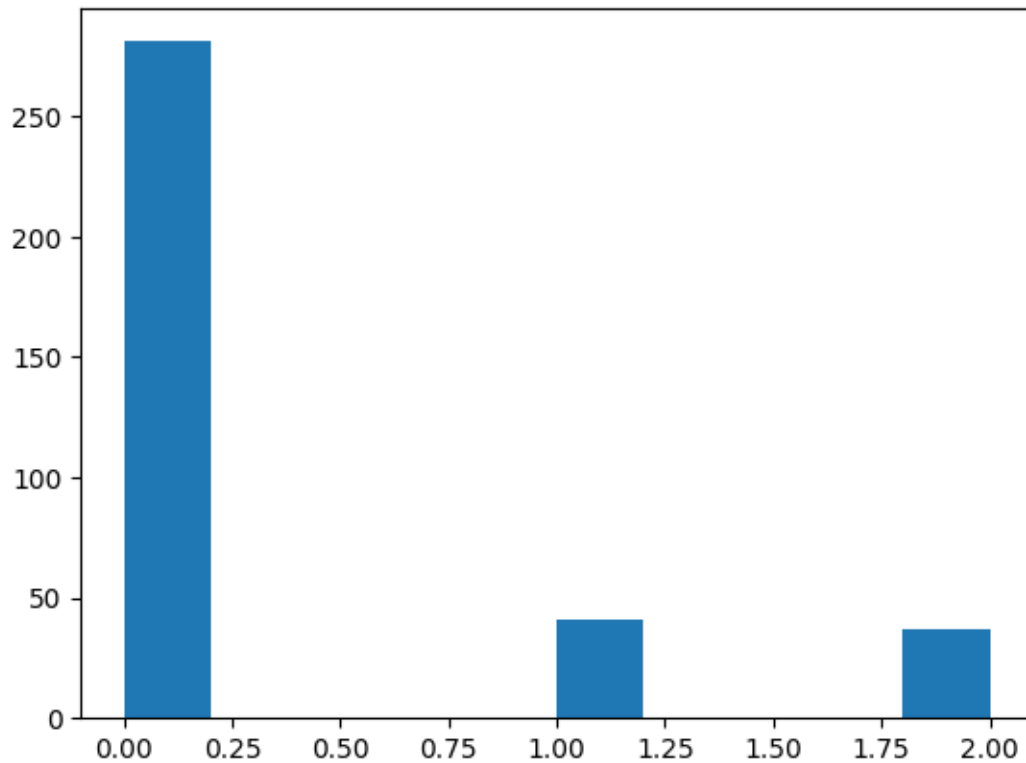


```
[21]: # Val distribution between ham, spam, smish  
plt.hist(val_ds['labels']);
```



```
[22]: # Test distribution between ham, spam, smish  
plt.hist(test_ds['labels'])
```

```
[22]: (array([281.,  0.,  0.,  0.,  0.,  41.,  0.,  0.,  0.,  37.]),  
      array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ]),  
      <BarContainer object of 10 artists>)
```



The distributions are similar.

```
[23]: from datasets import DatasetDict, Dataset # bring this in again to make sure I
      ↪ have the right one.
```

```
# Move the train, val, and test into datasets to then be moved to a DatasetDict
train_dataset = Dataset.from_pandas(train_ds)
val_dataset = Dataset.from_pandas(val_ds)
test_dataset = Dataset.from_pandas(test_ds)

train_dataset
```

```
[23]: Dataset({
      features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
      num_rows: 4776
})
```

```
[24]: # Move data to the DatasetDict
data = DatasetDict({
      'train': train_dataset,
      'val': val_dataset,
      'test': test_dataset
})
```

```
})
data
```

```
[24]: DatasetDict({
      train: Dataset({
            features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
            num_rows: 4776
        })
      val: Dataset({
            features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
            num_rows: 836
        })
      test: Dataset({
            features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
            num_rows: 359
        })
  })
```

This matches the above histogram.

We now have all the sentence data mapped to five categorical labels. We are now in a good place to begin tokenizing and fine-tuning the model.

1.7.1 Tokenizing the Data

We can now tokenize all of the sentences by fine-tuning the BERT tokenizer. Let's just run a quick tokenizer test to ensure we know how it works.

```
[25]: # A quick test to ensure the tokenizer is working as expected.
test = tokenizer('Hello there.')
print(f"Input_ids: {test['input_ids']}\nConversion back: {tokenizer.
      ↪convert_ids_to_tokens(test['input_ids'])}")
```

```
Input_ids: [101, 8667, 1175, 119, 102]
```

```
Conversion back: ['[CLS]', 'Hello', 'there', '.', '[SEP]']
```

```
[26]: print(f'Length of the tokenizer: {len(tokenizer)}\nCurrent word_embedding:
      ↪{model.bert.embeddings.word_embeddings}')
```

```
Length of the tokenizer: 28996
```

```
Current word_embedding: Embedding(28996, 768, padding_idx=0)
```

Now let's tokenize everything with a function to use with map.

```
[27]: # Create a tokenizing function to apply via a map
def tokenize_the_data(dskey):
    # Set the max length to 512 as that is the BERT max.
    tokenized_data = tokenizer(dskey['TEXT'], padding='max_length',
    ↪max_length=512, truncation=True,
                                return_tensors='pt', return_attention_mask=True )
```

```
return tokenized_data
```

```
[28]: # Tokenize the data
tokenized_dataset = DatasetDict({
    'train': train_dataset.map(tokenize_the_data),
    'val': val_dataset.map(tokenize_the_data),
    'test': test_dataset.map(tokenize_the_data)
})
```

```
Map:  0%|          | 0/4776 [00:00<?, ? examples/s]
```

```
Map: 0%|          | 0/836 [00:00<?, ? examples/s]
```

```
Map: 0%|          | 0/359 [00:00<?, ? examples/s]
```

```
[29]: # Check the tokenized_dataset for its layout
tokenized_dataset
```

```
[29]: DatasetDict({
    train: Dataset({
        features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 4776
    })
    val: Dataset({
        features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 836
    })
    test: Dataset({
        features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 359
    })
})
```

```
[30]: # Check some tokenized dataset values to ensure it worked as intended.
text_holder = np.array(tokenized_dataset['train']['input_ids'][0]).flatten()
# text_holder = text_holder[0]
print(f"Input_ids: {tokenized_dataset['train']['input_ids'][0]}\nConversion_↵
↵back: {tokenizer.convert_ids_to_tokens(text_holder)}")
```

[illegible]

[illegible]

[illegible]

1.7.2 Data Cleanup and Formatting for Torch

The tokenized dataset still has extra columns that are no longer needed. While they should be automatically removed for the forward pass in the Trainer, I'll remove them manually here and save the Trainer the effort.

I'll also end up running into an issue if I don't `squeeze()` my data as the trainer is looking for shapes of (batch, seq_len). Right now there is an extra dimension that needs to be taken out which I'll do here as well.

```
[31]: # Look at the dataset
      tokenized_dataset
```

```
[31]: DatasetDict({
      train: Dataset({
            features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
            num_rows: 4776
        })
      val: Dataset({
            features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
            num_rows: 836
        })
      test: Dataset({
```



```

        features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 359
    })
})

```

```

[32]: # Remove the unused features and reduce the dataset
tokenized_dataset_reduced = tokenized_dataset.remove_columns(['TEXT', 'URL',
↳ 'EMAIL', 'PHONE', 'token_type_ids'])
tokenized_dataset_reduced

```

```

[32]: DatasetDict({
  train: Dataset({
    features: ['labels', 'input_ids', 'attention_mask'],
    num_rows: 4776
  })
  val: Dataset({
    features: ['labels', 'input_ids', 'attention_mask'],
    num_rows: 836
  })
  test: Dataset({
    features: ['labels', 'input_ids', 'attention_mask'],
    num_rows: 359
  })
})

```

```

[33]: # Does everything exist that should exist.
# I did drop token_type_ids which may be able to be passed along later... I'll
↳ have to get this working first.
assert 'input_ids' in tokenized_dataset_reduced['train'].column_names
assert 'attention_mask' in tokenized_dataset_reduced['train'].column_names
assert 'labels' in tokenized_dataset_reduced['train'].column_names

```

```

[34]: tokenized_dataset_reduced.set_format(type='torch', columns=['input_ids',
↳ 'attention_mask', 'labels'])

```

There is still a shaping issue that must be resolved for training. this is because the tokenized_dataset_reduced['train']['input_ids'] is a list, rather than a Tensor. I'll convert things here.

```

[35]: # Showing the poor shaping
print(f"Poor shape of input_ids:      ")
↳ {tokenized_dataset_reduced['train'][0]['input_ids'].shape})
print(f"Poor shape of attention_mask:")
↳ {tokenized_dataset_reduced['train'][0]['attention_mask'].shape})
print(f"Poor shape of labels:        ")
↳ {tokenized_dataset_reduced['train'][0]['labels'].shape})

```

```
Poor shape of input_ids:      torch.Size([1, 512])
Poor shape of attention_mask: torch.Size([1, 512])
Poor shape of labels:        torch.Size([])
```

```
[36]: # Function to squeeze my dimensions
def squeeze_dims(dskey):
    if 'input_ids' in dskey:
        dskey['input_ids'] = torch.squeeze(dskey['input_ids'])
    if 'attention_mask' in dskey:
        dskey['attention_mask'] = torch.squeeze(dskey['attention_mask'])
    if 'labels' in dskey:
        dskey['labels'] = torch.squeeze(dskey['labels'])

    return dskey
```

```
[37]: # Squeeze the dimensions.
tokenized_dataset_reduced = tokenized_dataset_reduced.map(squeeze_dims)
```

```
Map:   0%|          | 0/4776 [00:00<?, ? examples/s]
Map:   0%|          | 0/836 [00:00<?, ? examples/s]
Map:   0%|          | 0/359 [00:00<?, ? examples/s]
```

```
[38]: # Showing the good shaping
print(f"Good shape of input_ids:      ␣
      ↪{tokenized_dataset_reduced['train'][0]['input_ids'].shape}")
print(f"Good shape of attention_mask:␣
      ↪{tokenized_dataset_reduced['train'][0]['attention_mask'].shape}")
print(f"Good shape of labels:        ␣
      ↪{tokenized_dataset_reduced['train'][0]['labels'].shape}")
```

```
Good shape of input_ids:      torch.Size([512])
Good shape of attention_mask: torch.Size([512])
Good shape of labels:        torch.Size([])
```

```
[39]: # Check the first few labels and their types to double check they are integers␣
      ↪for the categorization

print(tokenized_dataset_reduced['train']['labels'][:10]) # First 10 labels
print(type(tokenized_dataset_reduced['train']['labels'][0])) # Type of the␣
      ↪first label
```

```
tensor([0, 0, 0, 0, 2, 0, 0, 0, 0, 0])
<class 'torch.Tensor'>
```

We now have a working dataset to use with the models.

1.7.3 Setting up an Undersampled Dataset

There is the change that given the heavy distribution of the data to ham values that there could be issues in the training of the models. Here we'll set up a second dataset which is undersampled from the full dataset.

```
[40]: # Undersampling setups

from imblearn import under_sampling
from datasets import Dataset # Make sure I have the right Dataset
# Resample here

# Resample via undersampling up here

# I'll cut everything down to the m items which is just below the lowest class,
↳and still easily batchable
rus = under_sampling.RandomUnderSampler(random_state=226, replacement=False)

# Create a copy of the dataset and convert it to DataFrame
data_under = data.copy()
train_df = pd.DataFrame(data_under['train'])

# Ensure labels are in integer format
train_df['labels'] = train_df['labels'].astype(int)

# Separate features and labels
X = train_df.drop(columns=['labels'])
y = train_df['labels']

# Apply RandomUnderSampler
X_under, y_under = rus.fit_resample(X, y)

# Combine the resampled features and labels back into a DataFrame
train_resampled = X_under.copy()
train_resampled['labels'] = y_under

# Assign the resampled data back to the dataset
data_usample = data.copy()
data_usample['train'] = Dataset.from_pandas(train_resampled,
↳preserve_index=False)
```

```
[41]: # Check the new DatasetDict
data_usample
```

```
[41]: {'train': Dataset({
    features: ['TEXT', 'URL', 'EMAIL', 'PHONE', 'labels'],
    num_rows: 1167
}),
```

```

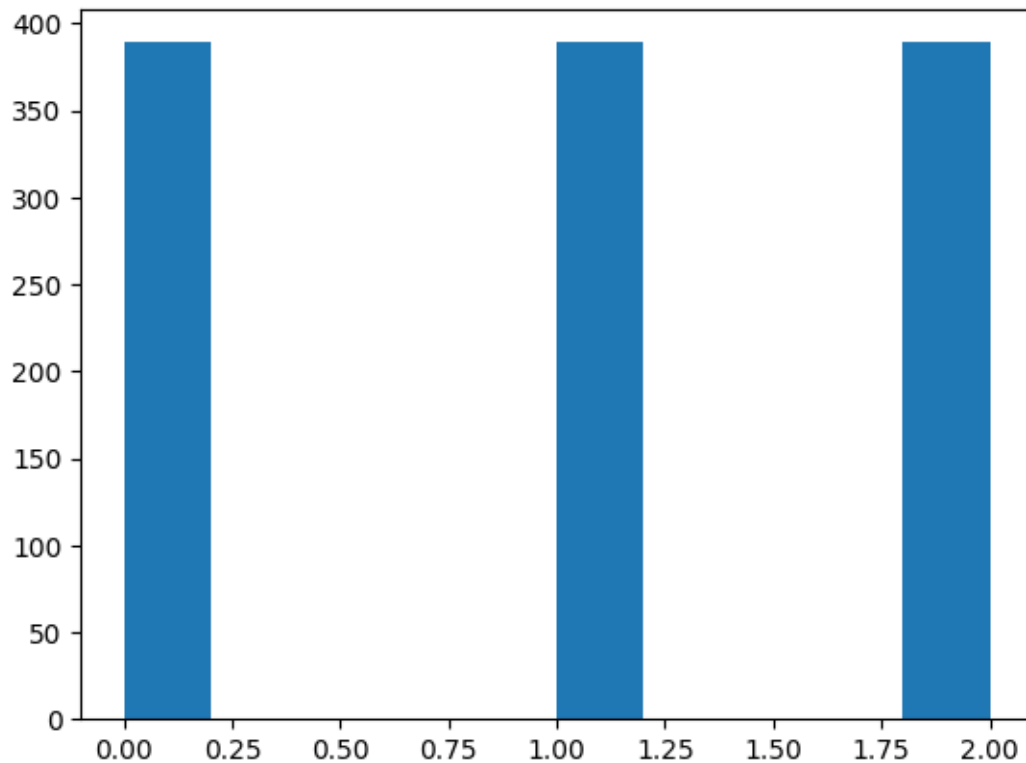
'val': Dataset({
    features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
    num_rows: 836
}),
'test': Dataset({
    features: ['labels', 'TEXT', 'URL', 'EMAIL', 'PHONE'],
    num_rows: 359
})}

```

```

[42]: # Make sure the undersampling is even
plt.hist(data_usample['train']['labels']);

```



Now we must go through all the data cleanup and formatting again.

```

[43]: # Rebuild the data for training with the undersampled set

# Tokenize the undersampled
tokenized_usample_dataset = DatasetDict({
    'train': data_usample['train'].map(tokenize_the_data),
    'validation': data_usample['val'].map(tokenize_the_data),
    'test': data_usample['test'].map(tokenize_the_data)
})

```

```

# Remove extra columns
tokenized_usample_dataset_reduced = tokenized_usample_dataset.
    ↳ remove_columns(['TEXT', 'URL', 'EMAIL', 'PHONE', 'token_type_ids'])
tokenized_usample_dataset_reduced['train'] =
    ↳ tokenized_usample_dataset_reduced['train'].remove_columns([])
tokenized_usample_dataset_reduced

# Does everything exist that should exist.
# I did drop token_type_ids which may be able to be passed along later... I'll
    ↳ have to get this working first.
assert 'input_ids' in tokenized_usample_dataset_reduced['train'].column_names
assert 'attention_mask' in tokenized_usample_dataset_reduced['train'].
    ↳ column_names
assert 'labels' in tokenized_usample_dataset_reduced['train'].column_names

# Set the Tensor format
tokenized_usample_dataset_reduced.set_format(type='torch',
    ↳ columns=['input_ids', 'attention_mask', 'labels'])

# Showing the poor shaping
print(f"Poor shape of input_ids:      ")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['input_ids'].shape})
print(f"Poor shape of attention_mask:")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['attention_mask'].shape})
print(f"Poor shape of labels:         ")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['labels'].shape})

tokenized_usample_dataset_reduced = tokenized_usample_dataset_reduced.
    ↳ map(squeeze_dims)

print(f"Good shape of input_ids:      ")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['input_ids'].shape})
print(f"Good shape of attention_mask:")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['attention_mask'].shape})
print(f"Good shape of labels:         ")
    ↳ {tokenized_usample_dataset_reduced['train'][0]['labels'].shape}\n")

# Check the labels
# Check the first few labels and their types to double check they are integers
    ↳ for the categorization
print(tokenized_usample_dataset_reduced['train']['labels'][:10]) # First 10
    ↳ labels
print(type(tokenized_usample_dataset_reduced['train']['labels'][0])) # Type of
    ↳ the first label

```

Map: 0% | 0/1167 [00:00<?, ? examples/s]

```

Map:   0%|          | 0/836 [00:00<?, ? examples/s]
Map:   0%|          | 0/359 [00:00<?, ? examples/s]
Poor shape of input_ids:      torch.Size([1, 512])
Poor shape of attention_mask: torch.Size([1, 512])
Poor shape of labels:         torch.Size([])

Map:   0%|          | 0/1167 [00:00<?, ? examples/s]
Map:   0%|          | 0/836 [00:00<?, ? examples/s]
Map:   0%|          | 0/359 [00:00<?, ? examples/s]

Good shape of input_ids:      torch.Size([512])
Good shape of attention_mask: torch.Size([512])
Good shape of labels:         torch.Size([])

tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
<class 'torch.Tensor'>

```

```

[44]: # Undersampled Training length
print(f"Training length: {len(tokenized_usample_dataset_reduced['train'])}")

```

Training length: 1167

Now we have an undersampled dataset to use as well.

1.8 Methodology

Our approach to classify ham, spam, and smishing messages using neural networks involves the following stages: 1. Text Preprocessing: The messages will undergo tokenization, stopword removal, and lemmatization. Additionally, URL features will be extracted and processed separately.

2. Using Traditional Models: o We will use TensorFlow to build out traditional models such Logistic Regression and SVM, LSTM. These baseline models will be used to classify messages based on TF-IDF features.

3. Advanced Models:

- BERT: We will leverage a fine-tuned BERT model for SMS classification, leveraging its ability to capture contextual relationships even in short texts. This BERT model uses a specific tokenizer on the full, unedited dataset.
- RoBERTa: As a more robust version of BERT, RoBERTa will be used for text classification, aiming to improve the precision of phishing detection. - **REMOVE THIS??**
- LLMs accessed via API: We will experiment with an LLM accessed via API for message generation to simulate phishing SMS scenarios and evaluate how well it detects sophisticated phishing attacks. Some models we plan on accessing are Google's Gemini and OpenAI's Chat GPT-4.
- 4. Ensemble Method: We will explore ensemble learning to combine the predictions from different models, including both textual and URL-based classifiers. - **REMOVE THIS??**

2. Introduce the Deep Neural Networks you used in your project

- RNN
 - A recurrent neural network.
- BERT
 - BERT is a transformer model that learns bidirectional representations by predicting masked words and sentence relationships, enabling strong performance across various NLP tasks.
- LLM - ChatGPT-4o-mini
 - A proprietary LLM that acts via prompting through an API.

Keywords: natural language processing, recurrent neural networks, transformers, sentiment analysis, multi-label classification, prediction, large language models

1.8.1 Model Fitting and Validation

1. BERT-base on the full train/validate/test dataset
 - Fine-tuning BERT-base on an 80/14/6 split of the data.
2. Bert-base on the undersampled dataset
 - Fine-tuning BERT-base on the undersampled training data
 - Used the same

1.8.2 BERT - Full Dataset

Now that we have tokenized data, let us fine-tune the BERT model.

This is done with the Trainer class which takes TrainingArguments.

```
[45]: # Set up the Training Args
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=10, # Prior runs show that I may be over fitting the data,
    ↪at 25 epochs...
    weight_decay=0.01,
    logging_dir='./logs',
    # logging_steps=10, # This made my loss vs epoch plot too noisy...
    logging_strategy='epoch',
    save_strategy='epoch',
)

# Set up the Trainer

data_collator = DataCollatorWithPadding(tokenizer=tokenizer, padding=True,
    ↪return_tensors='pt')

trainer = Trainer(
    model=model,
    args=training_args,
```

```

train_dataset=tokenized_dataset_reduced['train'],
eval_dataset=tokenized_dataset_reduced['val'],
tokenizer=tokenizer,
data_collator=data_collator
)

```

[46]: *# Run the Trainer*

```
trainer.train()
```

```

0%|          | 0/2990 [00:00<?, ?it/s]

c:\Users\btb51\miniconda3\envs\ai574_NLP\lib\site-
packages\transformers\models\bert\modeling_bert.py:439: UserWarning: 1Torch was
not compiled with flash attention. (Triggered internally at C:\cb\pytorch_100000
0000000\work\aten\src\ATen\native\transformers\cuda\sdp_utils.cpp:455.)
  attn_output = torch.nn.functional.scaled_dot_product_attention(

{'loss': 0.215, 'grad_norm': 7.968107223510742, 'learning_rate': 4.5e-05,
'epoch': 1.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.15840375423431396, 'eval_runtime': 20.8175,
'eval_samples_per_second': 40.159, 'eval_steps_per_second': 2.546, 'epoch': 1.0}
{'loss': 0.1289, 'grad_norm': 0.7139958143234253, 'learning_rate': 4e-05,
'epoch': 2.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.14052905142307281, 'eval_runtime': 20.8404,
'eval_samples_per_second': 40.114, 'eval_steps_per_second': 2.543, 'epoch': 2.0}
{'loss': 0.0931, 'grad_norm': 0.03817954659461975, 'learning_rate': 3.5e-05,
'epoch': 3.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.13507047295570374, 'eval_runtime': 20.7048,
'eval_samples_per_second': 40.377, 'eval_steps_per_second': 2.56, 'epoch': 3.0}
{'loss': 0.0615, 'grad_norm': 0.33940181136131287, 'learning_rate': 3e-05,
'epoch': 4.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.1358589231967926, 'eval_runtime': 20.707,
'eval_samples_per_second': 40.373, 'eval_steps_per_second': 2.56, 'epoch': 4.0}
{'loss': 0.049, 'grad_norm': 0.10122527182102203, 'learning_rate': 2.5e-05,
'epoch': 5.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.12136941403150558, 'eval_runtime': 20.6526,
'eval_samples_per_second': 40.479, 'eval_steps_per_second': 2.566, 'epoch': 5.0}
{'loss': 0.0419, 'grad_norm': 0.3154631555080414, 'learning_rate': 2e-05,
'epoch': 6.0}

```



```

0%|          | 0/53 [00:00<?, ?it/s]
{'eval_loss': 0.19431425631046295, 'eval_runtime': 20.6555,
 'eval_samples_per_second': 40.473, 'eval_steps_per_second': 2.566, 'epoch': 6.0}
{'loss': 0.0335, 'grad_norm': 0.12190482020378113, 'learning_rate': 1.5e-05,
 'epoch': 7.0}

0%|          | 0/53 [00:00<?, ?it/s]
{'eval_loss': 0.15437749028205872, 'eval_runtime': 20.7011,
 'eval_samples_per_second': 40.384, 'eval_steps_per_second': 2.56, 'epoch': 7.0}
{'loss': 0.0286, 'grad_norm': 0.016665717586874962, 'learning_rate': 1e-05,
 'epoch': 8.0}

0%|          | 0/53 [00:00<?, ?it/s]
{'eval_loss': 0.15591473877429962, 'eval_runtime': 20.5923,
 'eval_samples_per_second': 40.598, 'eval_steps_per_second': 2.574, 'epoch': 8.0}
{'loss': 0.022, 'grad_norm': 0.0005317842005752027, 'learning_rate': 5e-06,
 'epoch': 9.0}

0%|          | 0/53 [00:00<?, ?it/s]
{'eval_loss': 0.17217203974723816, 'eval_runtime': 20.8182,
 'eval_samples_per_second': 40.157, 'eval_steps_per_second': 2.546, 'epoch': 9.0}
{'loss': 0.0159, 'grad_norm': 0.0056058140471577644, 'learning_rate': 0.0,
 'epoch': 10.0}

0%|          | 0/53 [00:00<?, ?it/s]
{'eval_loss': 0.1749090701341629, 'eval_runtime': 20.8188,
 'eval_samples_per_second': 40.156, 'eval_steps_per_second': 2.546, 'epoch':
10.0}
{'train_runtime': 4823.4673, 'train_samples_per_second': 9.902,
 'train_steps_per_second': 0.62, 'train_loss': 0.06895873937319752, 'epoch':
10.0}

```

```

[46]: TrainOutput(global_step=2990, training_loss=0.06895873937319752,
metrics={'train_runtime': 4823.4673, 'train_samples_per_second': 9.902,
 'train_steps_per_second': 0.62, 'total_flos': 1.256629683068928e+16,
 'train_loss': 0.06895873937319752, 'epoch': 10.0})

```

Model Evaluation

- Examine your models (coefficients, parameters, errors, etc...)
- Compute and interpret your results in terms of accuracy, precision, recall, ROC etc.

```

[47]: # Check the training and validation losses
# Grab the losses from the trainer's log history.
training_losses = [log['loss'] for log in trainer.state.log_history if 'loss'
↳in log]

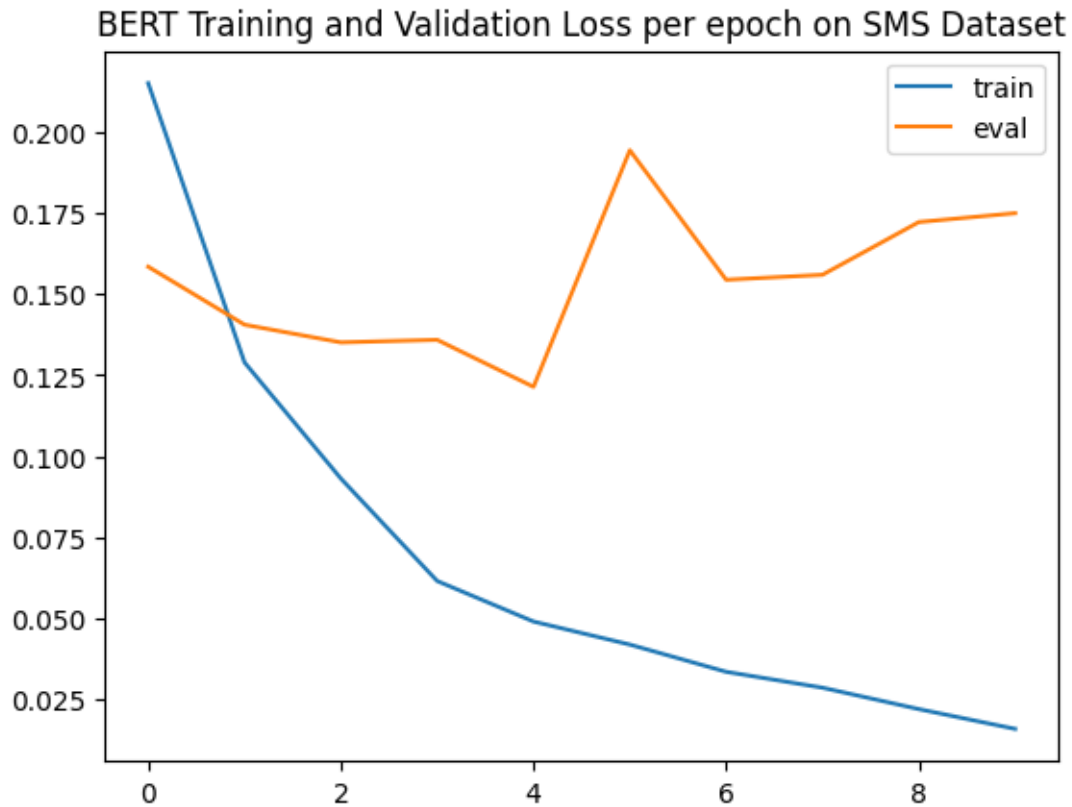
```

```

evaluation_losses = [log['eval_loss'] for log in trainer.state.log_history if
    'eval_loss' in log and 'epoch' in log]

# Plot the losses per epoch.
plt.plot(range(0, len(training_losses)), training_losses, label='train')
plt.plot(range(0, len(training_losses)), evaluation_losses, label='eval')
plt.legend()
plt.title('BERT Training and Validation Loss per epoch on SMS Dataset');

```



At late epochs, we begin to see some signs of overfitting but they are not drastic yet.

```

[48]: # Generate predictions from the test set
preds = trainer.predict(tokenized_dataset_reduced['test'])

```

```

0%|          | 0/23 [00:00<?, ?it/s]

```

```

[49]: # Grab the logits
preds_logits = preds.predictions
print(f'Example logit: {preds_logits[0]}')

# Which label does the logit correspond to, use proper axis

```

```

preds_labels = np.argmax(preds_logits, axis=1)
print(f'Class Label: {preds_labels[0]}')

# Get the actual labels from the test set
true_labels = preds.label_ids
print(f'True Label: {true_labels[0]}')

```

Example logit: [8.264434 -3.9791644 -3.3761387]
Class Label: 0
True Label: 0

```

[50]: # Run some metrics on the classification of the model
class_report = metrics.classification_report(true_labels, preds_labels,
                                             target_names=['ham', 'smishing', 'spam'])

print(f"Classification Report for BERT SMS Dataset:\n", class_report)

```

Classification Report for BERT SMS Dataset:

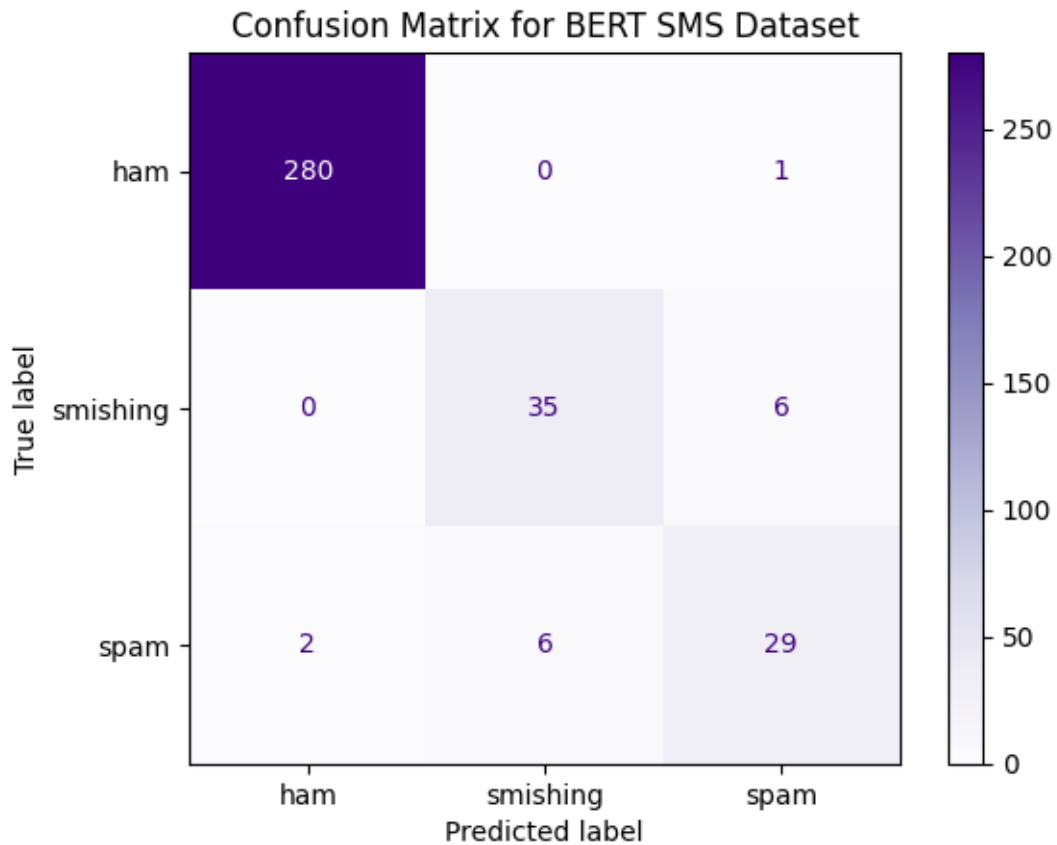
	precision	recall	f1-score	support
ham	0.99	1.00	0.99	281
smishing	0.85	0.85	0.85	41
spam	0.81	0.78	0.79	37
accuracy			0.96	359
macro avg	0.88	0.88	0.88	359
weighted avg	0.96	0.96	0.96	359

```

[51]: # Confusion matrix to see where things are going sideways
cm = metrics.confusion_matrix(true_labels, preds_labels)
disp = metrics.ConfusionMatrixDisplay(cm, display_labels=['ham', 'smishing', 'spam'])
disp.plot(cmap='Purples')
plt.title("Confusion Matrix for BERT SMS Dataset")

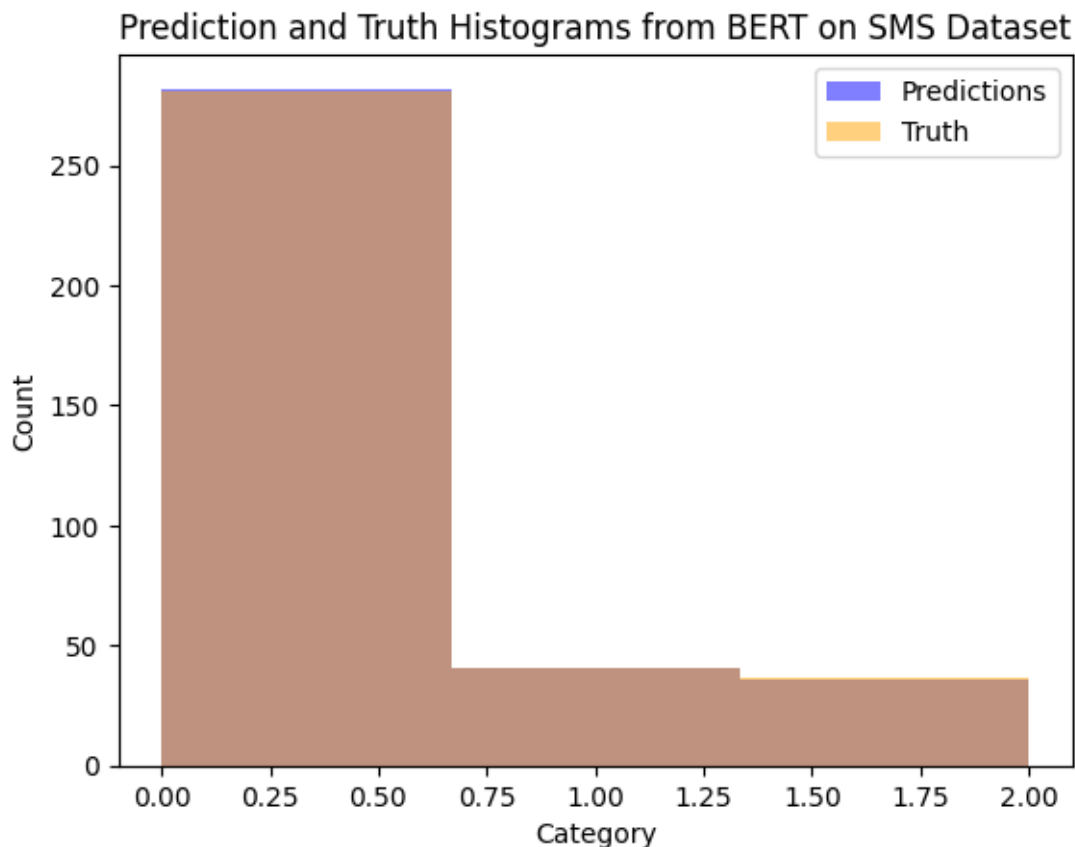
plt.show();

```



```
[52]: # Check the distribution of the predictions as well
      # Overlay the histograms

      plt.hist(preds_labels, bins=3, color='blue', alpha=0.5, label='Predictions')
      plt.hist(true_labels, bins=3, color='orange', alpha=0.5, label='Truth')
      plt.legend()
      plt.xlabel('Category')
      plt.ylabel('Count')
      plt.title('Prediction and Truth Histograms from BERT on SMS Dataset')
      plt.show();
```



```
[53]: # Map of numerical labels to categories
label_map = {0: 'ham', 1: 'smishing', 2: 'spam'}

# Convert numerical labels to categorical labels
pred_categories = [label_map[x] for x in preds_labels]
truth_categories = [label_map[x] for x in true_labels]

bar_width = 0.4

# Count the occurrences of each category
unique, counts = np.unique(pred_categories, return_counts=True)
category_counts = dict(zip(unique, counts))

# Categories and values
categories = list(category_counts.keys()) # ['ham', 'smishing', 'spam']
pred_values = list(category_counts.values()) # [10, 15, 5]
# X positions for the bars
x = np.arange(len(categories))
```

```

plt.bar(x - bar_width / 2, pred_values, bar_width, label='Predictions',
        color='blue')

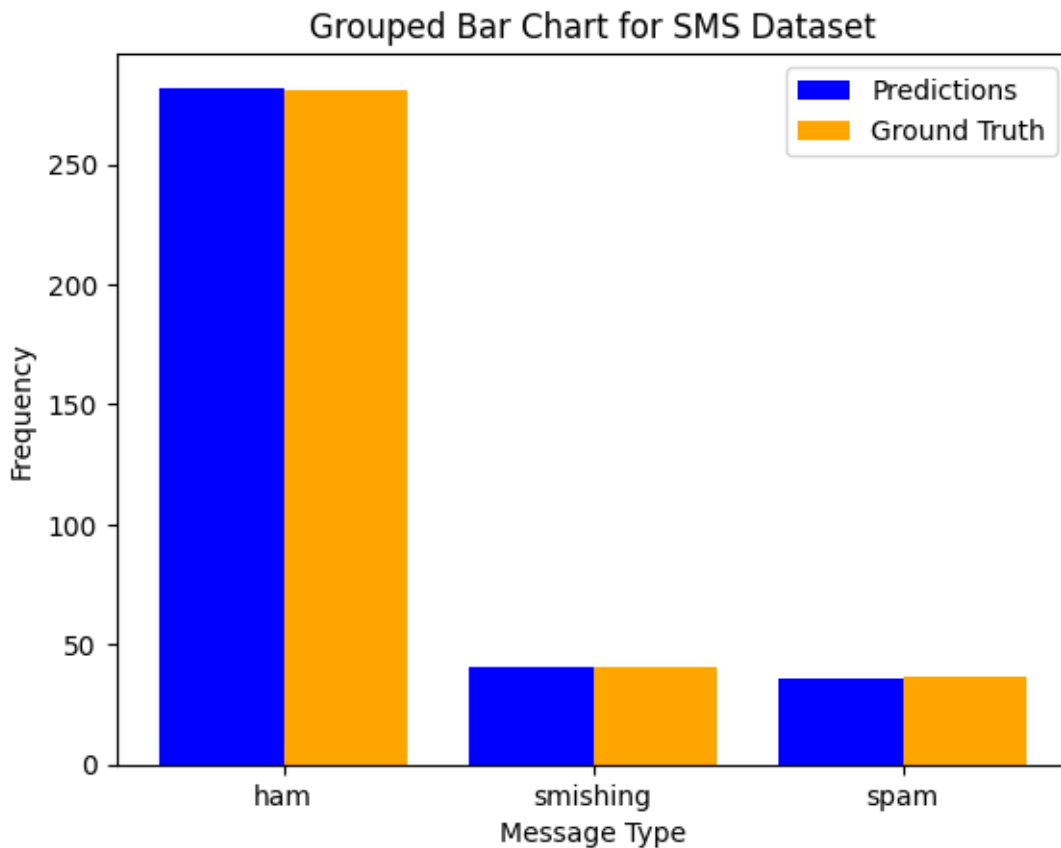
# Count the occurrences of each category
unique, counts = np.unique(truth_categories, return_counts=True)
category_counts = dict(zip(unique, counts))

# Categories and values
categories = list(category_counts.keys()) # ['ham', 'smishing', 'spam']
truth_values = list(category_counts.values()) # [8, 12, 7]
# Create grouped bar chart
plt.bar(x + bar_width / 2, truth_values, bar_width, label='Ground Truth',
        color='orange')

# Add labels and title
plt.xlabel('Message Type')
plt.ylabel('Frequency')
plt.title('Grouped Bar Chart for SMS Dataset')
plt.xticks(x, categories) # Replace numerical ticks with category names
plt.legend()

```

[53]: <matplotlib.legend.Legend at 0x218ee711810>



1.8.3 BERT - Undersampled Dataset

```
[54]: # Build another BERT model for training
model_usamp = BertForSequenceClassification.from_pretrained(BERT_model_name,
↳num_labels=3).to(device)
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at google-bert/bert-base-cased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
[55]: # Set up the Training Args
training_args_usamp = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=10, # Prior runs show that I may be over fitting the data,
↳at 25 epochs...
    weight_decay=0.01,
    logging_dir='./logs',
    # logging_steps=10, # This made my loss vs epoch plot too noisy...
    logging_strategy='epoch',
    save_strategy='epoch',
)

# Setup trainer
data_collator = DataCollatorWithPadding(tokenizer=tokenizer, padding=True,
↳return_tensors='pt')

trainer_usamp = Trainer(
    model=model_usamp,
    args=training_args_usamp,
    train_dataset=tokenized_dataset_reduced['train'],
    eval_dataset=tokenized_dataset_reduced['val'],
    tokenizer=tokenizer,
    data_collator=data_collator
)
```

```
[56]: # Run the trainer
trainer_usamp.train()
```

```
0%|          | 0/2990 [00:00<?, ?it/s]
```

```
{'loss': 0.1943, 'grad_norm': 4.186595439910889, 'learning_rate': 4.5e-05,
```

```

'epoch': 1.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.1864289939403534, 'eval_runtime': 20.9038,
 'eval_samples_per_second': 39.993, 'eval_steps_per_second': 2.535, 'epoch': 1.0}
{'loss': 0.1082, 'grad_norm': 0.3191804587841034, 'learning_rate': 4e-05,
 'epoch': 2.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.12798982858657837, 'eval_runtime': 20.9444,
 'eval_samples_per_second': 39.915, 'eval_steps_per_second': 2.531, 'epoch': 2.0}
{'loss': 0.0773, 'grad_norm': 0.1454443782567978, 'learning_rate': 3.5e-05,
 'epoch': 3.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.12464877218008041, 'eval_runtime': 20.9018,
 'eval_samples_per_second': 39.997, 'eval_steps_per_second': 2.536, 'epoch': 3.0}
{'loss': 0.0454, 'grad_norm': 6.613923072814941, 'learning_rate': 3e-05,
 'epoch': 4.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.13679008185863495, 'eval_runtime': 20.9134,
 'eval_samples_per_second': 39.974, 'eval_steps_per_second': 2.534, 'epoch': 4.0}
{'loss': 0.0424, 'grad_norm': 0.03872271254658699, 'learning_rate': 2.5e-05,
 'epoch': 5.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.11700735986232758, 'eval_runtime': 20.8989,
 'eval_samples_per_second': 40.002, 'eval_steps_per_second': 2.536, 'epoch': 5.0}
{'loss': 0.0336, 'grad_norm': 0.07338763773441315, 'learning_rate': 2e-05,
 'epoch': 6.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.15119148790836334, 'eval_runtime': 20.9136,
 'eval_samples_per_second': 39.974, 'eval_steps_per_second': 2.534, 'epoch': 6.0}
{'loss': 0.0269, 'grad_norm': 0.028370099142193794, 'learning_rate': 1.5e-05,
 'epoch': 7.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.1568015068769455, 'eval_runtime': 20.9217,
 'eval_samples_per_second': 39.958, 'eval_steps_per_second': 2.533, 'epoch': 7.0}
{'loss': 0.0216, 'grad_norm': 0.006228907499462366, 'learning_rate': 1e-05,
 'epoch': 8.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.15891540050506592, 'eval_runtime': 20.8408,
 'eval_samples_per_second': 40.114, 'eval_steps_per_second': 2.543, 'epoch': 8.0}

```



```
{'loss': 0.0192, 'grad_norm': 0.0011846498819068074, 'learning_rate': 5e-06,
'epoch': 9.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.18093103170394897, 'eval_runtime': 20.9171,
'eval_samples_per_second': 39.967, 'eval_steps_per_second': 2.534, 'epoch': 9.0}
{'loss': 0.0141, 'grad_norm': 0.003543140133842826, 'learning_rate': 0.0,
'epoch': 10.0}

0%|          | 0/53 [00:00<?, ?it/s]

{'eval_loss': 0.18228869140148163, 'eval_runtime': 20.9077,
'eval_samples_per_second': 39.985, 'eval_steps_per_second': 2.535, 'epoch':
10.0}
{'train_runtime': 6323.4319, 'train_samples_per_second': 7.553,
'train_steps_per_second': 0.473, 'train_loss': 0.05830045081301278, 'epoch':
10.0}

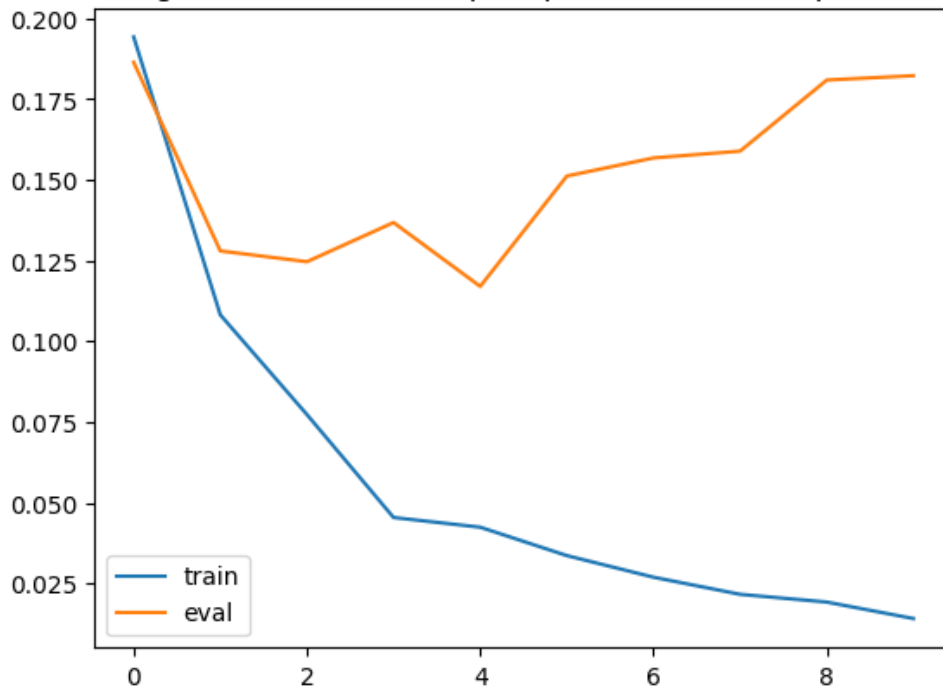
[56]: TrainOutput(global_step=2990, training_loss=0.05830045081301278,
metrics={'train_runtime': 6323.4319, 'train_samples_per_second': 7.553,
'train_steps_per_second': 0.473, 'total_flos': 1.256629683068928e+16,
'train_loss': 0.05830045081301278, 'epoch': 10.0})
```

Model Evaluation

```
[57]: # Check the training and validation losses
# Grab the losses from the trainer's log history.
training_losses = [log['loss'] for log in trainer_usamp.state.log_history if
↳ 'loss' in log]
evaluation_losses = [log['eval_loss'] for log in trainer_usamp.state.
↳ log_history if 'eval_loss' in log and 'epoch' in log]

# Plot the losses per epoch.
plt.plot(range(0, len(training_losses)), training_losses, label='train')
plt.plot(range(0, len(training_losses)), evaluation_losses, label='eval')
plt.legend()
plt.title('BERT Training and Validation Loss per epoch on Undersampled SMS_
↳ Dataset');
```

BERT Training and Validation Loss per epoch on Undersampled SMS Dataset



```
[58]: # Generate predictions from the full test set
preds = trainer_usamp.predict(tokenized_dataset_reduced['test'])
```

```
0%|          | 0/23 [00:00<?, ?it/s]
```

```
[59]: # Grab the logits
preds_logits = preds.predictions
print(f'Example logit: {preds_logits[0]}')

# Which label does the logit correspond to, use proper axis
preds_labels = np.argmax(preds_logits, axis=1)
print(f'Class Label: {preds_labels[0]}')

# Get the actual labels from the test set
true_labels = preds.label_ids
print(f'True Label: {true_labels[0]}')
```

```
Example logit: [ 7.790182 -3.912317 -3.6198564]
Class Label: 0
True Label: 0
```

```
[64]: # Run some metrics on the classification of the model
class_report = metrics.classification_report(true_labels, preds_labels,
```

```

target_names=['ham', 'smishing',
↪'spam'])

print(f"Classification Report for BERT Undersampled SMS Dataset:\n",
↪class_report)

```

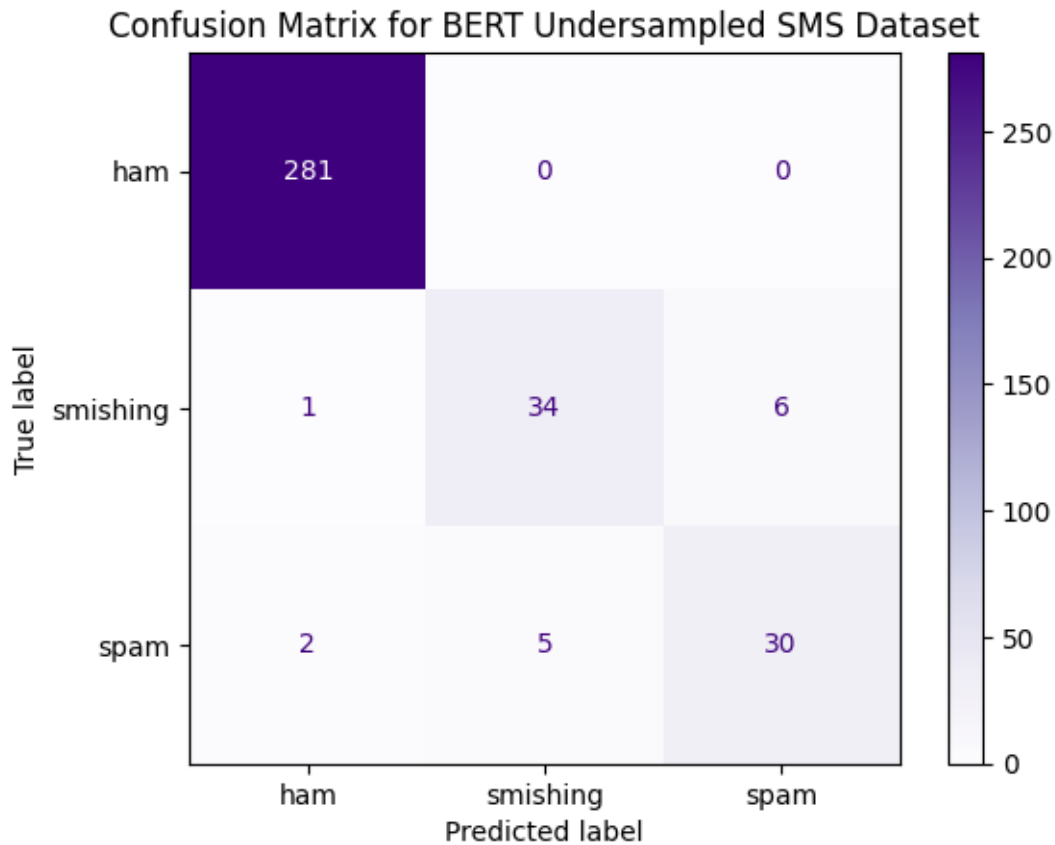
Classification Report for BERT Undersampled SMS Dataset:

	precision	recall	f1-score	support
ham	0.99	1.00	0.99	281
smishing	0.87	0.83	0.85	41
spam	0.83	0.81	0.82	37
accuracy			0.96	359
macro avg	0.90	0.88	0.89	359
weighted avg	0.96	0.96	0.96	359

```

[61]: # Confusion matrix to see where things are going sideways
cm = metrics.confusion_matrix(true_labels, preds_labels)
disp = metrics.ConfusionMatrixDisplay(cm, display_labels=['ham', 'smishing',
↪'spam'])
disp.plot(cmap='Purples')
plt.title("Confusion Matrix for BERT Undersampled SMS Dataset")
plt.show();

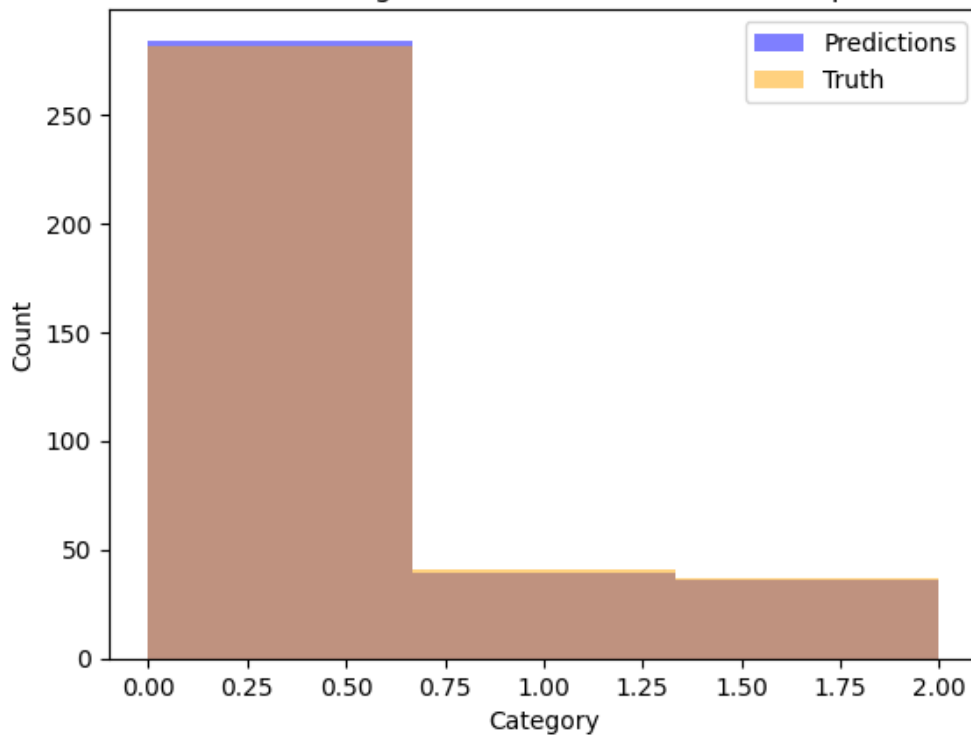
```



```
[62]: # Check the distribution of the predictions as well
      # Overlay the histograms

      plt.hist(preds_labels, bins=3, color='blue', alpha=0.5, label='Predictions')
      plt.hist(true_labels, bins=3, color='orange', alpha=0.5, label='Truth')
      plt.legend()
      plt.xlabel('Category')
      plt.ylabel('Count')
      plt.title('Prediction and Truth Histograms from BERT on Undersampled SMS_
↳Dataset')
      plt.show();
```

Prediction and Truth Histograms from BERT on Undersampled SMS Dataset



```
[63]: # Map of numerical labels to categories
label_map = {0: 'ham', 1: 'smishing', 2: 'spam'}

# Convert numerical labels to categorical labels
pred_categories = [label_map[x] for x in preds_labels]
truth_categories = [label_map[x] for x in true_labels]

bar_width = 0.4

# Count the occurrences of each category
unique, counts = np.unique(pred_categories, return_counts=True)
category_counts = dict(zip(unique, counts))

# Categories and values
categories = list(category_counts.keys()) # ['ham', 'smishing', 'spam']
pred_values = list(category_counts.values()) # [10, 15, 5]
# X positions for the bars
x = np.arange(len(categories))
plt.bar(x - bar_width / 2, pred_values, bar_width, label='Predictions',
        color='blue')

# Count the occurrences of each category
```

```

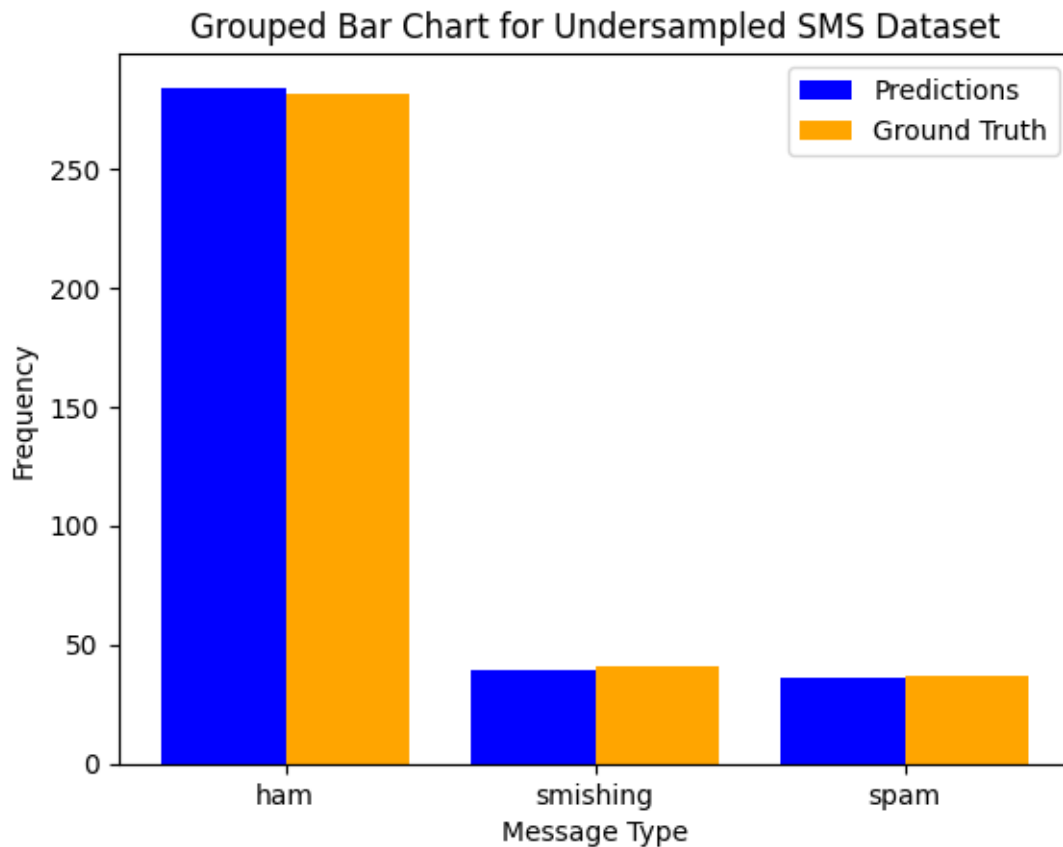
unique, counts = np.unique(truth_categories, return_counts=True)
category_counts = dict(zip(unique, counts))

# Categories and values
categories = list(category_counts.keys()) # ['ham', 'smishing', 'spam']
truth_values = list(category_counts.values()) # [8, 12, 7]
# Create grouped bar chart
plt.bar(x + bar_width / 2, truth_values, bar_width, label='Ground Truth',
        color='orange')

# Add labels and title
plt.xlabel('Message Type')
plt.ylabel('Frequency')
plt.title('Grouped Bar Chart for Undersampled SMS Dataset')
plt.xticks(x, categories) # Replace numerical ticks with category names
plt.legend()

```

[63]: <matplotlib.legend.Legend at 0x218f1f72b30>



1.9 Issues / Improvements

- The SMS dataset has a very small number of smishing labels. Despite the performance of BERT-base, it may be improved upon by having a larger dataset. This is difficult to obtain as text messages are not easily scraped as web based documents, such as emails, are.
- BERT-base has a 10% dropout through each of the transformer cells. This was left at 10% and not changed. There is a chance that optimizing the dropout could increase performance but upping the dropout percentage greatly increases the training time and would force a larger number of epochs to stabilize the losses and was deemed not worth the tradeoff.
- Though transformers are prone to overfitting, cross-validation (CV) was not used here. The dataset is extremely small for smishing values and after running both the full dataset and undersampled dataset with no changes to the confusion matrix, it is believed that the pre-training on BERT is largely responsible for the model performance.
- We did not incorporate the URL, EMAIL, or PHONE categorical values in the dataset. We used just the Text data as an analysis. A multi-step process could be done with different models based on the prior detection of URL, EMAIL, or PHONE values in the text. This would result in a larger preprocessing step which may not be worth it. Furthermore, the detection of these URLs, EMAILs, and PHONE numbers may in an of itself be a problem given ways to obfuscate them.

1.10 References

- See Report for full citations.
- Outline can be found by opening this in a pdf viewer.

1.11 Credits

This code is similar and builds off of the AI 574 Assignment 3 code submitted by Brandon Botzer (myself).

[]: *# End of Brandon Botzer's BERT-base fine-tuning work.*