C-NARS/P: An Open-Source Tool for Classification of Narratives in Survey Data Using Pre-Trained Language Models

Joelle Abramowitz[1], Jenna Kim[2], and Jinseok Kim[34]

Abstract

This report presents an open-source tool (C-NARS/P) created to classify narratives in survey data using pre-trained language models. Unlike conventional machine learning methods, pre-trained language models such as BERT can utilize context of words in text to make accurate predictions. In this report, technical details of functions and parameter setups for implementing pre-trained language models for classification are provided with screenshots of the code file to guide users for validation and improvement of the implementation. C-NARS/P can be modified for a wide range of text classification tasks with ease.

Acknowledgement

The research reported herein was performed pursuant to a grant from the National Science Foundation Award FW-HTF-P 2128416. The opinions and conclusions expressed are solely those of the author(s) and do not represent the opinions or policy of NSF or any agency of the federal government. Neither the United States government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of the contents of this report. Reference herein to any specific commercial product, process or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply endorsement, recommendation or favoring by the United States government or any agency thereof.

Background

This project uses machine learning to automate the classification of occupation types depicted in narrative responses in longitudinal survey data. Typical machine learning approaches use manual classifications of a subset of the data (labeled data) to automate classification of the remainder of the data. To do this, they perform several tasks: (1) pre-processing the raw data, (2) training candidate machine learning models, (3) developing and validation of the candidate models, and (4) testing and implementation of the optimal model. These tasks are common procedures conducted in previous studies in predicting occupational or demographic classes using text data (Boselli et al, 2018; Ikudo et al, 2019; Lampos et al., 2016; Mac Kim et al., 2017; Preoţiuc-Pietro, Lampos, and Aletras, 2015; Preoţiuc-Pietro and Ungar, 2018).

[1] jabramow@umich.edu; Survey Research Center, Institute for Social Research, University of Michigan, Ann Arbor, MI USA
[2] jkim682@illinois.edu; School of Information Sciences, University of Illinois at Urbana-Champaign, IL USA
[3] jinseokk@umich.edu; Survey Research Center, Institute for Social Research, University of Michigan, Ann Arbor, MI USA
[4] School of Information, University of Michigan, Ann Arbor, MI USA

In our previous study (MRDRC UM21-4 "*What We Talk about When We Talk about Self-Employment: Examining Self-Employment and the Transition to Retirement among Older Adults in the United States*"), we implemented these machine learning approaches to classify occupation types in the Health and Retirement Study data. Our approach first pre-processed all words in the industry and occupation and employer name narratives in the data. This pre-processing included cleaning, standardization, stop-word removal, and stemming which are typical procedures in natural language processing (Berry and Castellanos, 2004). Then, the approach transformed the pre-processed words into a vector (list) of word tokens. Next, to train candidate machine learning models, the vectors of tokens were fed into various machine learning algorithms including Logistic Regression, Naïve Bayes, Random Forests, Gradient Boosted Trees, and Support Vector Machine to learn frequencies and combinations of tokens that best associate each narrative to one of the classes using a subset of the data that have been manually coded. To develop and validate the candidate models, the approach identified the best performing combinations of feature weights and increases generalizability of the machine learning models by implementing various modifications of parameters and data on another subset of the manually coded data. Finally, testing and implementation of the optimal classification model identified from this training and development and validation were applied to the remainder of the unclassified narratives to assign each narrative one of the occupation classes. We created a code file named 'C-NARS' (Classification of Narratives in Survey Data) that implements the machine learning procedure (Abramowitz and Kim, 2021).

Although the implementation of existing machine learning methods provided us with valuable insights into occupation type distributions and changes over time, the prediction models also produced ambiguous results for a substantial number of instances, which had to be manually labeled by human coders. To improve our prediction models, we use BERT in our new project.

BERT (Bidirectional Encoder Representations from Transformers) can classify text using a technique called fine-tuning (Devlin et al., 2019). Fine-tuning is a process of adapting a pre-trained language model to a specific task by training it on task-specific labeled data. The general approach to fine-tune BERT for text classification involves:

(1) Preprocessing: The input text is tokenized, and special tokens such as [CLS] and [SEP] are added to the beginning and end of the input text, respectively. This step is different from preprocessing steps for conventional machine learning where a series of data pre-processing steps such as tokenization, stemming, stop-word removal, and POS (part of speech tagging) is conducted by heuristics or algorithms specifically determined by users for tasks.

(2) Embedding: The tokenized input text is passed through BERT's pre-trained neural network, which generates contextualized word embeddings for each token.

(3) Classification Head: A classification head is added on top of BERT's neural network to perform the actual classification task. This can be a simple linear layer or a more complex neural network architecture.

(4) Fine-tuning: The entire model (pre-trained BERT plus classification head) is trained on the task-specific labeled data. During this training process, the weights of both BERT and the classification head are updated to optimize classification performance.

(5) Prediction: Once the model is trained, it can be used to predict the class label of new, unseen input text by passing it through the pre-processing and embedding steps, and then through the trained classification head.

BERT has achieved state-of-the-art performance on a wide range of text classification tasks, including sentiment analysis, question answering, and natural language inference (Koroteev, 2021). The ability to fine-tune a pre-trained language model like BERT on a specific classification task has made it a powerful tool in the field of natural language processing.

Technical Details

This project produces a code file with a file extension '.ipynb' (CNARSP_v1.ipynb) and implements the machine learning procedure described subsequently. The code is written in a script language ('python' version 3.6.5 or above) using open-source machine learning ('scikit-learn') and Natural Language Processing ('Transformers') packages and the pre-trained language model BERT (BERT-base). The code set is named 'CNARS/P' (Classification of Narratives in Survey Data with Pre-trained language models) and will be shared on a public code repository (https://github.com/TEEDLab/CNARSP) for validation, reuse, and improvement by researchers.

Input File

The code set accepts two input data sets – training data and test (target) data - in a csv file format. Figure 1 shows a partial preview of the input data used for the project. Each input data set consists of three columns: Project id, Narrative (or Text), and Class. In the figure, project id and narrative information are blinded as the data sets used for this project contain IRB-protected information.

(1) Project id: this is a series of numbers or alphanumeric characters used to refer to a unique instance (case).

(2) Narrative: in this column, a textual narrative of job description is recorded.

(3) Class: One of occupational classes. During implementation, class names in any text string format will be converted into integers (0, 1, 2, …).

	A	B	C	D	E	F	G
1	projectid	narrative	class				
2	[redacted]		1				
3			3				
4			3				
5			3				
6			3				
7			1				
8			3				
9			1				
10			3				
11			3				
12			3				
13			3				
14			3				
15			3				
16			3				
17			2				
18			3				
19			3				
20			3				
21			3				
22			3				
23			3				
24			3				
25			3				

Figure 1: Example of Input Data for C-NARS/P

Implementation of Prediction

1. Setup

1-1. Install Packages and Load Libraries

The implementation of BERT requires installation of transformers, PyTorch and imblearn. Below is a screenshot of packages and tools that need to be installed.

```
# transformer ver: 4.15.0
!pip install transformers==4.15.0
```

```
!pip install imblearn
```

```
# install PyTorch
# Note: No need to install PyTorch if this notebook is running on the AWS Sagemaker with pytorch kernel

!pip install torch==1.5.0
```

```
# Check if the packages are correctly installed
!pip list
```

Below is a list of libraries and modules required to implement the entire prediction procedure.

```
import timeit
import transformers

import os

import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

from collections import defaultdict
from textwrap import wrap

from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
```

This code displays output in plots. Accordingly, configuration of plot format and color scheme is necessary.

```
%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='darkgrid', palette='muted', font_scale=1.5)
COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F00FF"]
sns.set_palette(sns.color_palette(COLORS_PALETTE))
rcParams["figure.figsize"] = (12, 6)
```

1-2. Check GPU for Training

BERT requires a GPU for implementation. We used an NVIDIA GPU (Model: Quadro RTX 4000). Using a GPU requires CUDA for proper configuration in computing hardware. The code below shows steps and example outputs for checking the configuration of the GPU on the local machine we used for our project.

```
# Check a version of CUDA
!nvcc --version
```

```
# Check if there's a GPU available
if torch.cuda.is_available():

    # Tell PyTorch to use the GPU
    device = torch.device("cuda")

    print('There are {:d} GPU(s) available.'.format(torch.cuda.device_count()))
    print('We will use the GPU: ', torch.cuda.get_device_name(0))

else:
    device = torch.device("cpu")

    print('No GPU available, using the CPU instead.')
```

```
#Additional Info when using cuda
if device.type == 'cuda':
    print(torch.cuda.get_device_name(0))
    print('Memory Usage:')
    print('Allocated:', round(torch.cuda.memory_allocated(0)/1024**3,1), 'GB')
    print('Cached:   ', round(torch.cuda.memory_cached(0)/1024**3,1), 'GB')
```

Before running the process, it is necessary to check GPU memory and utilization. For this step, by running '!nvidia-smi', a comprehensive report will be displayed as shown below.

```
# check GPU memory and utilization
!nvidia-smi

# To check the GPU memory usage while the process is running
# open a terminal in the directory (Go to New-> Terminal) and type the above code
```

```
Mon Jun 27 16:05:57 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.129.06   Driver Version: 470.129.06   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Quadro RTX 4000     Off  | 00000000:21:00.0  On |                  N/A |
| 30%   36C    P8    18W / 125W |    282MiB /  7973MiB |     20%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Quadro RTX 4000     Off  | 00000000:2D:00.0 Off |                  N/A |
| 30%   28C    P8     8W / 125W |     13MiB /  7982MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1265      G   /usr/lib/xorg/Xorg                 39MiB |
|    0   N/A  N/A      1852      G   /usr/lib/xorg/Xorg                 86MiB |
|    0   N/A  N/A      1975      G   /usr/bin/gnome-shell               78MiB |
|    0   N/A  N/A      4825      G   ...9917740637882974 21,131072      65MiB |
|    1   N/A  N/A      1265      G   /usr/lib/xorg/Xorg                  4MiB |
|    1   N/A  N/A      1852      G   /usr/lib/xorg/Xorg                  4MiB |
+-----------------------------------------------------------------------------+
```

One issue with using a GPU is that it often needs to be cleaned of memory before running another iteration of implementation. This can be easily done by clearing the occupied memory as follows:

```
# clear the occupied cuda memory for efficient use
import gc

gc.collect()
torch.cuda.empty_cache()

# Kill a process in running if more GPU space is needed
#!sudo kill -9 3320
```

2. Load Data

Below is a function loading data from a local directory in a system or a stand-alone machine. After loading the data, the function displays information about the data such as number of rows and columns as well as missing values. Note that the function is customized for the data in this project. As a result, some code may be unnecessary for other data. Only the first few lines are captured in the figure shown below for illustration purposes.

```
def load_data(filename, colname, record):

    """
    Read in input file and load data

    filename: csv file
    record: text file to save summary

    return: dataframe

    """
    ## 1. Read in data from csv file
    #df = pd.read_csv(filename, encoding="utf-8", engine='python')

    # If unicodedecode error appears, use one of below options
    # 1. Save dataset in utf-8 format: open csv file (file->save as-> CSV UTF-8)
    # 2. Change encoding to 'unicode-escape'
    df = pd.read_csv(filename, encoding="unicode-escape")

    ## 2. No of rows and columns & data view
    print("No of Rows (Raw data): {}".format(df.shape[0]), file=record)
    print("No of Columns: {}".format(df.shape[1]), file=record)
    print("No of Rows (Raw data): {}".format(df.shape[0]))
    print("No of Columns: {}".format(df.shape[1]))

    print("\n<Data View (Raw data)>\n{}".format(df.head(15)), file=record)
    print("\n<Data View (Raw data)>\n{}".format(df.head(15)))

    ## 3. Replace null values in any rows
    # 3-1. Identify columns with null values
    print("\nCheck if null value exists:\n")
    print(df.info())
    print("\nCheck if null value exists:\n", file=record)
    df.info(buf=record)

    # 3-2. Replace empty values with numpy NaN
    df = df.replace(r'^\s*$', np.nan, regex=True)
```

3. Data Processing

3-1. Check distribution of Token Length

The maximum input token length in BERT is 512. This means that any sequence longer than 512 tokens will be truncated. BERT's input consists of tokenized text, which is then processed by its neural network

architecture to generate contextualized representations of the input. The 512 token limit is a constraint imposed by the architecture and the available memory resources. The longer the token length, the more computing resources are needed. Given resource constraints, we set a limit (first 150 or less) to the number of tokens to be used for prediction. The function below checks the distribution of token lengths in the loaded data and displays it.

```
def token_distribution(df, tokenizer, record):
    token_lens = []
    long_tokens = []

    # remove null values
    df = df.dropna()

    for id, txt in zip(df.instanceid, df.sentence):
        tokens = tokenizer.encode(txt, padding=True, truncation=True, max_length=512)
        token_lens.append(len(tokens))

        # Check a sentence with extreme length
        if len(tokens) > 150:
            long_tokens.append((id, len(tokens)))

    print("\n********** Check if Long Sentence Exists **********\n", file=record)
    print("\n********** Check if Long Sentence Exists **********\n")

    if len(long_tokens)>0:
      print(long_tokens, file=record)
      print(long_tokens)
    else:
      print("No long sentence (<=150 tokens)", file=record)
      print("No long sentence (<=150 tokens)")

    print("\nMin token:", min(token_lens), file=record)
    print("Max token:", max(token_lens), file=record)
    print("Avg token:", round(sum(token_lens)/len(token_lens)), file=record)
    print("\nMin token:", min(token_lens))
    print("Max token:", max(token_lens))
    print("Avg token:", round(sum(token_lens)/len(token_lens)))

    # plot the distribution
    sns.displot(token_lens)
    plt.xlim([0, max(token_lens)+10])
    plt.xlabel("Token Count")
```

3-2. Create a PyTorch Dataset

We use PyTorch to run BERT on a local system. PyTorch is an open-source machine learning framework that was developed primarily by Facebook's AI research team. It is designed to provide a flexible and efficient platform for building and training machine learning models. One of the key benefits of PyTorch is its ease of use and flexibility. It provides a simple and intuitive interface for building and training models, making it easy for researchers and developers to experiment with different architectures and algorithms. Additionally, PyTorch supports both CPU and GPU acceleration, making it ideal for training large-scale models. This is why we used PyTorch for implementing BERT. The function below shows how we converted loaded data into the PyTorch dataset format.

```
class LabelDataset(Dataset):
    def __init__(self, reviews, targets, tokenizer, max_len):
        self.reviews = reviews
        self.targets = targets
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.reviews)

    def __getitem__(self, item):
        review = str(self.reviews[item])
        review = " ".join(review.split())
        target = self.targets[item]

        encoding = self.tokenizer.encode_plus(
            review,
            None,                                   # second parameter is needed
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length',
            truncation=True,
            return_token_type_ids=True,
            return_attention_mask=True,
            return_tensors='pt')

        return {
            'texts': review,
            'input_ids': encoding['input_ids'].flatten(),           # flatten()
            'token_type_ids': encoding['token_type_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'targets': torch.tensor(target, dtype=torch.long)
        }
```

3-3. Sampling

This function was created to address any class imbalance problem in the loaded data. Class imbalance in machine learning occurs when the number of instances in one class is significantly lower than the number of instances in another class. This can occur in binary classification problems, where one class is rare compared to the other, or in multi-class problems, where one or more classes have very few examples. Addressing class imbalance properly is important because it can lead to biased models that have poor performance for the minority class. When a model is trained on a dataset with imbalanced classes, it tends to predict the majority class more frequently than the minority class, leading to lower precision, recall, and F1 scores for the minority class. This can be a problem in many real-world applications where the minority class is often the one of interest, such as fraud detection, medical diagnosis, and rare event prediction.

There are several techniques that can be used to address class imbalance in machine learning, including: resampling (either over-sampling the minority class or under-sampling the majority class to balance the dataset), cost-sensitive learning (assigning higher misclassification costs to the minority class during model training), synthetic data generation (generating synthetic data for the minority class to increase the number of instances in that class), and ensemble methods (combining multiple models to improve the prediction performance on the minority class). For our project, we created a function for resampling. Although we ultimately did not use any of them, this function can be helpful for users who want to test how resampling affects machine learning performance in their projects.

```python
def sample_data(X_train, y_train, sampling=0, sample_method='over'):
    """
       Sampling input train data

       X_train: dataframe of X train data
       y_train: dataframm of y train data
       sampling: indicator of sampling funtion is on or off
       sample_method: method of sampling (oversampling or undersampling)

    """

    from imblearn.over_sampling import RandomOverSampler
    from imblearn.under_sampling import RandomUnderSampler

    if sampling:
        if sample_method == 'over':
            oversample = RandomOverSampler(random_state=42)
            X_over, y_over = oversample.fit_resample(X_train, y_train)
            print('\n****** Data Sampling ******')
            print('\nOversampled Data (class, Rows):\n{}'.format(y_over.value_counts()))
            X_train_sam, y_train_sam = X_over, y_over

        elif sample_method == 'under':
            undersample = RandomUnderSampler(random_state=42)
            X_under, y_under = undersample.fit_resample(X_train, y_train)
            print('\n****** Data Sampling ******')
            print('\nUndersampled Data (class,Rows):\n{}'.format(y_under.value_counts()))
            X_train_sam, y_train_sam = X_under, y_under
    else:
        X_train_sam, y_train_sam = X_train, y_train
        print('\n****** Data Sampling ******')
        print('\nNo Sampling Performed\n')

    return X_train_sam, y_train_sam
```

3-4. Create a Data Loader and Classifier

These steps enable BERT to read loaded data and conduct prediction. Here, we relied on forward attention in BERT for prediction. The forward attention mechanism in BERT allows each token to attend to all the tokens that come before it in the input sequence. During the forward attention process in BERT, each token in the input sequence is transformed into a query vector, a key vector, and a value vector. The query vector is used to attend to the key vectors of all the tokens that come before it, and the attention weights are computed using a dot product between the query and key vectors. The resulting attention weights are then used to weigh the value vectors of the previous tokens, which are combined to generate the final contextualized embedding for the current token. For example, if we want BERT to predict if a given sentence, “I hate python. I spend many hours debugging a code file in python,” refers to a snake or a programming language, BERT’s forward attention will create contextualized embeddings for the sentence (‘python’ is associated with ‘debugging,’ ‘code,’ ‘file’, etc.) and use them to predict what label (Snake or Programming Language) is likely to follow the sentence. By attending to the tokens that come before it, the forward attention mechanism in BERT enables the model to capture important context and dependencies between words in the input sequence, which is crucial for many natural language processing tasks, such as text classification, question answering, and language generation.

```
def create_data_loader(df, tokenizer, max_len, batch_size):
    ds = LabelDataset(
        reviews = df.sentence.to_numpy(),
        targets = df.label.to_numpy(),
        tokenizer = tokenizer,
        max_len = max_len
    )

    return DataLoader(
        ds,
        batch_size = batch_size,
        num_workers = 1)
```

```
class LabelClassifier(nn.Module):

    def __init__(self, n_classes, model_loaded):
        super(LabelClassifier, self).__init__()
        self.bert = model_loaded
        self.dropout = nn.Dropout(p=0.3)
        self.linear = nn.Linear(self.bert.config.hidden_size, n_classes)

    def forward(self, input_ids, attention_mask, token_type_ids):
        bert_out = self.bert(
            input_ids = input_ids,
            attention_mask = attention_mask,
            token_type_ids = token_type_ids)
        output_dropout = self.dropout(bert_out.pooler_output)
        output = self.linear(output_dropout)

        return output
```

4. Training and Validation

4-1. Training

Training in classification using BERT refers to the process of adapting the pre-trained BERT model to a specific classification task by fine-tuning it on task-specific labeled data. During training, the BERT model is first initialized with pre-trained weights learned from a large corpus of text. The model is then fine-tuned on a smaller labeled dataset for a specific classification task. During the training for our project, cross-entropy loss was used as a loss function.

Cross-entropy loss is a widely used loss function in deep learning for classification tasks and is commonly used for training neural networks on large, labeled datasets. It is used to measure the difference between the predicted probability distribution over the possible classes and the true probability distribution (i.e., one-hot encoded labels) for the training examples. The cross-entropy loss function is defined as follows:

$$L(y, \hat{y}) = -\sum i \; yi \log(\hat{y}i)$$

where y is the true label, ŷ is the predicted probability distribution over the possible labels, and i is the index of the class. The cross-entropy loss penalizes the model more heavily for incorrect predictions that have high confidence, and less for incorrect predictions that have low confidence. During BERT training for classification tasks, the goal is to minimize the value of the cross-entropy loss function by adjusting the weights of the neural network. This is typically done using an optimization algorithm such as stochastic gradient descent (SGD) or Adam. Below is the screenshot of the function for training that includes the implementation of the loss function.

```python
def train_model(
    model,
    data_loader,
    loss_fn,
    optimizer,
    device,
    scheduler,
    n_examples,
    outfile):

    model = model.train()

    losses = []
    correct_predictions = 0

    for d in data_loader:
        input_ids = d["input_ids"].to(device, dtype=torch.long)
        attention_mask = d["attention_mask"].to(device, dtype=torch.long)
        token_type_ids = d["token_type_ids"].to(device, dtype=torch.long)
        targets = d["targets"].to(device)

        outputs = model(
            input_ids = input_ids,
            attention_mask = attention_mask,
            token_type_ids=token_type_ids
        )

        _, preds = torch.max(outputs, dim=1)
        loss = loss_fn(outputs, targets)

        # printout for checking the prediction & target
        #print("Pred: ", preds)
        #print("Target: ", targets)

        correct_predictions += torch.sum(preds == targets)
        losses.append(loss.item())

        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()

    print("Correct Prediction (Train): {} out of {}".format(correct_predictions.int(), n_examples), file=outfile)
    print("Correct Prediction (Train): {} out of {}".format(correct_predictions.int(), n_examples))

    return correct_predictions.double() / n_examples, np.mean(losses)
```

4-2. Validation

Evaluation on validation data is required for classification using BERT because it helps to prevent overfitting and provides a way to tune hyperparameters for the model. During BERT training, the model is optimized to minimize the loss function on the training data. However, the model's true performance on unseen data is unknown. Therefore, it is essential to evaluate the model's performance on a separate validation dataset that the model has not seen during training. Evaluation on the validation dataset allows us to monitor the model's performance during training and to detect if the model is overfitting to the training data. Overfitting occurs when the model becomes too specialized to the training data and fails to generalize well to new, unseen data. By monitoring the model's performance on the validation set during training, we can adjust the model's hyperparameters to prevent overfitting and improve its generalization performance. A function for validation is implemented in our implementation as shown in the code screen-captured below.

```python
def eval_model(
    model,
    data_loader,
    loss_fn,
    device,
    n_examples,
    outfile):

    model = model.eval()

    losses = []
    correct_predictions = 0

    with torch.no_grad():
        for d in data_loader:
            input_ids = d["input_ids"].to(device, dtype=torch.long)
            attention_mask = d["attention_mask"].to(device, dtype=torch.long)
            token_type_ids = d["token_type_ids"].to(device, dtype=torch.long)
            targets = d["targets"].to(device)

            outputs = model(
                input_ids = input_ids,
                attention_mask = attention_mask,
                token_type_ids = token_type_ids
                )

            _, preds = torch.max(outputs, dim=1)
            loss = loss_fn(outputs, targets)

            correct_predictions += torch.sum(preds == targets)
            losses.append(loss.item())

    print("Correct Prediction (Eval): {} out of {}".format(correct_predictions.int(), n_examples), file=outfile)
    print("Correct Prediction (Eval): {} out of {}".format(correct_predictions.int(), n_examples))

    return correct_predictions.double()/n_examples, np.mean(losses)
```

4-3. Training Loop

In classification using BERT, training is usually conducted multiple times and each iteration of training is called an 'epoch.' Formally, an epoch in BERT-based machine learning tasks refers to one complete iteration of the training data through the neural network. During an epoch, the model is trained on the entire training dataset, with the training examples passed through the network in batches. One epoch is completed when all the training examples have been passed through the network once. The number of epochs is a hyperparameter that is typically set before training and determines how many times the entire training dataset will be passed through the network during training. Increasing the number of epochs may improve the model's accuracy, but it can also increase the risk of overfitting, where the model becomes too specialized to the training data and performs poorly on unseen data. We set the number of epochs at 4 following the recommendations of the developers of BERT. During each epoch, the weights of the neural network are updated based on the optimization algorithm and the loss function, which is detailed above. The screenshot below shows part of the function for iterating training and displaying validation results.

```
def training_loop(epochs,
                  modelname,
                  model,
                  train_data_loader,
                  val_data_loader,
                  loss_fn,
                  optimizer,
                  device,
                  scheduler,
                  n_train,
                  n_val,
                  model_file,
                  record):

    print("\n**** Model Name: " + modelname + " *****", file=record)
    print("\n**** Model Name: " + modelname + " *****")

    history = defaultdict(list)
    best_accuracy = 0

    for epoch in range(epochs):
        print("\nEpoch {} / {}".format(str(epoch + 1), str(epochs)), file=record)
        print("-" * 60, file=record)

        print("\nEpoch {} / {}".format(str(epoch + 1), str(epochs)))
        print("-" * 60)

        train_acc, train_loss = train_model(
            model,
            train_data_loader,
            loss_fn,
            optimizer,
            device,
            scheduler,
            n_train,
            outfile=record)

        print("Train Loss: {}, Accuracy: {}\n".format(train_loss, train_acc), file=record)
        print("Train Loss: {}, Accuracy: {}\n".format(train_loss, train_acc))

        val_acc, val_loss = eval_model(
            model,
            val_data_loader,
            loss_fn,
            device,
            n_val,
            outfile=record)
```

5. Prediction and Evaluation

5-1. Prediction

The code shown below provides predictions by prediction models learned during training iterations. An important note is that during prediction, a softmax function was used in our project.

The softmax function is defined as follows:

$$\text{softmax}(z_i) = e^{(z_i)} / (\text{sum}_j\ e^{(z_j)})$$

where z_i is the raw output value (logit) for class i, and the denominator is the sum of the exponential values of all logits for all possible classes. The softmax function maps the logits to a probability distribution that sums to one, with each class having a probability value between 0 and 1. In BERT-based classification, the softmax function is used to convert the logits into a probability distribution over the possible classes, which allows the model to make a prediction for the most likely class for a given input. The predicted class is the one with the highest probability value in the probability distribution.

```python
def get_predictions(model, data_loader):

    model = model.eval()

    review_texts = []
    predictions = []
    prediction_probs = []
    real_values = []

    with torch.no_grad():
        for d in data_loader:
            texts = d["texts"]
            input_ids = d["input_ids"].to(device, dtype=torch.long)
            attention_mask = d["attention_mask"].to(device, dtype=torch.long)
            token_type_ids = d["token_type_ids"].to(device, dtype=torch.long)
            targets = d["targets"].to(device)

            outputs = model(
                input_ids = input_ids,
                attention_mask = attention_mask,
                token_type_ids = token_type_ids
            )

            _, preds = torch.max(outputs, dim=1)

            # Apply the softmax or sigmoid function to normalize the raw output(logits) to get probabi
            probs = F.softmax(outputs, dim=1)

            review_texts.extend(texts)
            predictions.extend(preds)
            prediction_probs.extend(probs)
            real_values.extend(targets)

    # move the data to cpu
    predictions = torch.stack(predictions).cpu()
    prediction_probs = torch.stack(prediction_probs).cpu().detach().numpy()
    real_values = torch.stack(real_values).cpu()

    return review_texts, predictions, prediction_probs, real_values
```

5-2. Evaluation

Prediction results are evaluated on test data that is set aside from the labeled data. The accuracy of prediction is measured in the form of a confusion matrix. In classification, a confusion matrix is a table that summarizes the performance of a classification model by comparing the predicted labels to the actual labels for a set of data. Below is an example confusion matrix for a binary classification problem, where the model is predicting whether a text refers to ‘manager’ (positive class) or not (negative class):

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

The rows of the matrix represent the actual labels, and the columns represent the predicted labels. The four quadrants of the matrix correspond to the following:

True Positive (TP): The number of positive examples that were correctly classified as positive by the model.

True Negative (TN): The number of negative examples that were correctly classified as negative by the model.

False Positive (FP): The number of negative examples that were incorrectly classified as positive by the model.

False Negative (FN): The number of positive examples that were incorrectly classified as negative by the model.

From the confusion matrix, we can calculate precision and recall measuring the accuracy of prediction.

Precision is a measure of the accuracy of positive predictions. It is the ratio of true positives to the total number of positive predictions made by the classifier. It is calculated as:

precision = TP / (TP + FP)

Recall is a measure of the completeness of positive predictions. It is the ratio of true positives to the total number of actual positive examples in the data. It is calculated as:

recall = TP / (TP + FN)

In general, precision measures how well the classifier can correctly identify positive examples, while recall measures how well it is able to capture all positive examples in the data.

Below is the screenshot of the code for generating a confusion matrix as well as estimating precision and recall from prediction results. Note that precision and recall are calculated from functions (*confusion_matrix* and *classification_report*) innate in the *sklearn* package (see **1-1. Install Packages and Load Libraries**).

```
def evaluate_model(y_test, y_pred, record, eval_model=0):
    """
      evaluate model performance

      y_test: y test data
      y_pred: t prediction score
      eval_model: indicator if this funtion is on or off

    """

    if eval_model:

        print('\n************** Model Evaluation **************', file=record)
        print('\n************** Model Evaluation **************')

        print('\nConfusion Matrix:\n', file=record)
        print('\nConfusion Matrix:\n')
        print(confusion_matrix(y_test, y_pred), file=record)
        print(confusion_matrix(y_test, y_pred))

        print('\nClassification Report:\n', file=record)
        print('\nClassification Report:\n')
        print(classification_report(y_test, y_pred, digits=4), file=record)
        print(classification_report(y_test, y_pred, digits=4))
```

5-3. Prediction Probability

In most classification tasks, prediction results are provided in the format of class name (technically, integers representing classes). In our project, however, prediction results are provided in the format of probability score. This output format was deliberately chosen because we wanted prediction results with certain levels of uncertainty to be passed to manual inspection for high accuracy. Many machine learning models are designed to produce probability scores for prediction but set up to convert the scores into binary classes if the probability score is 0.5 or above (positive; 1; yes, etc.; otherwise, negative; 0; no, etc.). We tweaked the

output format of results by prediction model so a probability score for each predicted instance is provided in a float number. The code snippet below shows the function for this job.

```
def predict_proba(df_test, y_text, y_test, y_pred, y_pred_probs, n_class, proba_file, test_unlabel_on=0, proba_out=0):

    """
       Get probability of each class

       df_test: original X test data
       y_text: text data sentence
       y_test: original y values
       y_pred: predicted y values
       y_pred_probs: probability scores of prediction
       n_class: number of label class
       proba_file: output file of probability scores
       test_unlabel_on: on(1) and off(0) for using unlabeled test data
       proba_out: on (1) and off (0) for probability output

    """
    if proba_out:

        prob_dict = {}
        for i in range(n_class):
          key_str = 'proba_' + str(i)
          value_str = y_pred_probs[:, i]
          prob_dict[key_str] = value_str

        df_proba = pd.DataFrame(prob_dict)

        if test_unlabel_on:
          y_test = df_test["label"].replace(111, "NA")

        df_pred = pd.DataFrame({'instanceid': df_test["instanceid"],
                                'input': y_text,
                                'act': y_test,
                                'pred': y_pred})

        df_result = pd.concat([df_pred, df_proba], axis=1)

        ## Save output
        df_result.to_csv(proba_file, encoding='utf-8', header=True, index=False)
```

6. Main Function

The functions listed above implement specific jobs that form building blocks of the entire BERT implementation. A main function is needed to integrate these functions, so they are implemented organically. Below is the screenshot of the main function.

```
def main(input_file, test_file, colname, sample_on, sample_type, max_len,
         batch_size, modelname, model_download_path, device, n_class,
         learning_rate, epochs, model_file_path, test_unlabel_on, eval_on,
         proba_on, proba_file, result_file):

    ## 0. open result file for records
    f=open(result_file, "a")

    if test_unlabel_on==0:
      train_test_labeled(input_file, colname, sample_on, sample_type, max_len,
                         batch_size, modelname, model_download_path, device,
                         learning_rate, epochs, model_file_path, eval_on,
                         proba_on, proba_file, result_file, record=f)

    elif test_unlabel_on==1:
      test_unlabeled(test_file, colname, max_len, batch_size, device, n_class,
                     model_download_path, modelname, model_file_path, proba_on,
                     test_unlabel_on, result_file, proba_file, record=f)

    f.close()
```

The main function consists of two parts. First, it will run a typical training procedure to test if a parameter 'test_unlabel_on' is turned off. If it is, the function will implement training and test using the labeled data. This option is turned on when users need to evaluate prediction performance of a model. Second, if the

parameter is turned on, it will run the prediction task for unlabeled text instances. This applies to a real-world scenario where users want to train a prediction model and predict labels for unlabeled data. The trained model's performance is evaluated on labeled data with 'test_unlabel_on' turned off. These two parts are not independent: The second part relies on the first part.

As such, to implement the main function, we need two functions of implementation with 'test_unlabel_on' turned on or off. Below is part of the code for the case with 'test_unlabel_on' turned on. It consists of nine steps: (1) Data Loading, (2) Train and Test Splitting, (3) PyTorch Data Formatting, (4) Sampling, if needed, (5) Loading Auto-tokenizer, (6) Model Training, (7) Evaluating Model Performance, and (9) Probability Prediction. Each step is implemented by a function described above except Loading Auto-tokenizer.

In BERT, an auto-tokenizer is a built-in tokenizer that automatically converts raw input text into tokens that can be understood by the BERT model. The auto-tokenizer is based on WordPiece tokenization, which is a subword tokenization algorithm. It splits words into smaller subwords and then represents them as tokens. This allows the model to handle out-of-vocabulary words and capture the meaning of the subwords. The auto tokenizer also conducts special token addition. Here, special tokens refer to strings [CLS] (classification token) and [SEP] (separator token) which are added to the beginning and end of the input, respectively, to help the model distinguish between different parts of the input.

```
def train_test_labeled(input_file, colname, sample_on, sample_type, max_len,
                       batch_size, modelname, model_download_path, device,
                       learning_rate, epochs, model_file_path, eval_on,
                       proba_on, proba_file, result_file, record):

    """
       Training and testing using labeled data

       input_file: input file
       colname: colume name for selection between title and abstract
       sample_on: indicator of sampling on or off
       sample_type: sample type to choose if sample_on is 1
       model_method: name of classifier to be applied for model fitting
       eval_on: indicator of model evaluation on or off
       proba_file: name of output file of probability
       result_file: name of output file of evaluation

    """

    ## 1. Load data

    print("\n************** Loading Data **************\n", file=record)
    print("\n************** Loading Data **************\n")
    df = load_data(input_file, colname, record=record)

    # Number of label class
    n_class = len(df['label'].value_counts().keys().tolist())
    print("\nNumber of label class: ", n_class, file=record)
    print("\nNumber of label class: ", n_class)

    # Check the first instance of input data
    print("First Sentence: \n", df.sentence[0], file=record)
    print("First Sentence: \n", df.sentence[0])

    ## 2. Train and test split

    print("\n************** Spliting Data **************\n", file=record)
```

The implementation of the case with 'test_label_on' turned off is different. It consists of five steps: (1) Data Loading (here the data refer to unlabeled data to predict labels for), (2) Loading Best Trained Model (this model is generated from the case with 'test_label_on' turned on), (3) Prediction, (4) Evaluating Model Performance, and (5) Probability Prediction. Below is part of the code for implementation.

```
def test_unlabeled(test_file, colname, max_len, batch_size, device, n_class,
                   model_download_path, modelname, model_file_path, proba_on,
                   test_unlabel_on, result_file, proba_file, record):
    """
       Predict label for unlabeled test data and get probability file

       test_file: unlabeled data for getting prediction
       colname: colume name for feeding as input string
       eval_on: indicator of model evaluation on or off
       proba_file: name of output file of probability
       result_file: name of output file of evaluation
       n_class: number of label class

    """

    # Check testing time
    start_time = timeit.default_timer()

    # Load downloaded tokenizer & model
    tokenizer = AutoTokenizer.from_pretrained(model_download_path)
    model_loaded = AutoModel.from_pretrained(model_download_path)

    ## 1. Load test data
    print("\n************** Loading Unlabeled Test Data **************", file=record)
    print("\n************** Loading Unlabeled Test Data **************")
    df_unlabel = load_data(test_file, colname, record=record)
    test_data_loader = create_data_loader(df_unlabel, tokenizer, max_len, batch_size)

    ## 2. Load the best trained model with checkpoint
    print("\n************** Loading Best Trained Model **************", file=record)
    print("\n************** Loading Best Trained Model **************")
    print("\nModel Name: " + modelname, file=record)
    print("\nModel Name: " + modelname)

    model = LabelClassifier(n_class, model_loaded)
    model.load_state_dict(torch.load(model_file_path))
    model = model.to(device)
```

7. Parameter Setting

The functions listed above are the core components of the code file that implements BERT-based prediction of text data. Several auxiliary functions (especially, displaying results in plots) are not shown in this report as they do not greatly affect the functionality of the code. Before implementing the code, users are required to provide information into the code for proper implementation. Below is the screenshot of the code that contains information about parameters users need to set before implementing the code. It shows the names and parameters used for a mock-up dataset. Note that parameter names are lowercased or uppercased following common programming practices.

(1) input_filename: Name of the input file. The file format is supposed to be csv with each column separated by commas. The input file format can be easily modified by changing data import parameters in 2. Load Data.

(2) column_name: This column asks users to specify which column(s) in the input data will be used as input. In our project, we named as ‘text’ the column that contains text strings to be fed into the machine learning process.

(3) test_filename: This is the name of the file to test. The value for this parameters should be provided in a text string format in parentheses if ‘test_unlabel_on’ is 1 (turned on). Otherwise, set the value as None.

(4) num_class: This specifies how many classes are to be predicted during implementation.

(5) sampling_on: 0 for no sampling; 1 for sampling

(6) sampling_type: Use when sampling_on = 1. ‘over’ denotes oversampling, while ‘under’ denotes undersampling.

(7) pretrained_model_name: We used ‘bert-base-cased’ for this project. Pretrained models can be downloaded from the websites for each model. Each model is updated according to its developer’s schedule. New models based on BERT are being developed to customize BERT to specialized Natural Language Processing tasks (Lee et al., 2020; Liu et al., 2019).

For example, SciBERT is a pre-trained language model that is specifically designed for scientific text (Beltagy et al., 2019). It was developed by the Allen Institute for Artificial Intelligence (AI2) in collaboration with the University of Washington and the Paul G. Allen School of Computer Science & Engineering. SciBERT is based on the BERT (Bidirectional Encoder Representations from Transformers) architecture and is pre-trained on a large corpus of scientific documents, including papers from the fields of computer science, physics, and biology. The pre-training is done using a masked language modeling (MLM) task, where the model is trained to predict masked words within a sentence. The unique aspect of SciBERT is that it is trained on scientific text, which is often more technical and domain-specific than general language. This allows it to capture scientific context and knowledge that may be missed by general language models. SciBERT has been shown to outperform general language models on a variety of scientific NLP tasks, such as named entity recognition, relation extraction, and sentence classification. It has also been used in several research projects in the fields of biomedical informatics, chemistry, and computer science.

For our project, we used the original BERT model developed by Google Research. This BERT model has been used as baseline in many studies and shown to produce stable performance. However, if there is a new BERT model specialized in a Natural Language Processing task similar to ours, it would be worth testing the new one to see if any substantial improvements in prediction are possible.

(8) internet_on: 1 for downloading via the Internet connection; 0 for loading the model locally

Machines that implement BERT are typically connected to the Internet because BERT models are deposited on servers maintained and updated by each model’s developers. In cases where the Internet connection is unavailable, users need to download models to the local system or transfer them using portable drives. The size of the BERT base model we used is about 423MB.

(9) modelname_string: name of the model

Pretrained_model_folder: name of the folder where the model is located.

(10) MAX_LEN: Maximum length of an input text instance

As detailed in 3-1. Check distribution of Token Length, the maximum input token length in BERT is 512. This means that any sequence longer than 512 tokens will be truncated. We set a limit (first 150 or less) to the number of tokens to be used for prediction.

(11) BATCH_SIZE: 16 or 32

In BERT, the batch size refers to the number of input sequences that are processed in parallel during training. During training, the BERT model receives input sequences in batches of fixed size. The batch size can be specified by the user and is usually chosen based on the available computing resources (e.g., GPU memory) and the size of the dataset. The batch size affects the training speed and the memory requirements of the model. A larger batch size can result in faster training since

the model processes more input sequences in parallel, but it may also require more memory. On the other hand, a smaller batch size can reduce the memory requirements, but it may result in slower training. The batch size can also affect the quality of the model. A smaller batch size can result in more noisy gradients, which can lead to slower convergence and lower accuracy. On the other hand, a larger batch size can result in more stable gradients, which can lead to faster convergence and higher accuracy. The choice of batch size in BERT depends on various factors, including the available computing resources, the size of the dataset, and the desired training speed and accuracy. In our project, we chose 16 based on suggestions from the developers of BERT.

(12) EPOCHS: 2, 3, or 4

In 4-4. Training Loop, the meaning of an epoch in BERT-based machine learning was explained. We chose '4' following suggestions from the BERT developers.

(13) LEARNING_RATE: 5e-5, 3e-5, or 2e-5

In machine learning and deep learning, the learning rate is a hyperparameter that controls the step size at which a model's parameters are updated during training. The learning rate determines how quickly the model learns from the training data. A higher learning rate means that the model updates its parameters more aggressively, which can result in faster convergence but may also cause the model to overshoot the optimal solution. Conversely, a lower learning rate means that the model updates its parameters more gradually, which can result in slower convergence but may also produce more stable and accurate results. In BERT, the learning rate is typically set using a technique called "learning rate scheduling," which involves gradually decreasing the learning rate over time as the model converges. This allows the model to make larger updates to its parameters in the early stages of training when it is still far from convergence and smaller updates later when it is closer to the optimal solution. The optimal learning rate for a BERT model depends on various factors, including the size of the dataset, the complexity of the task, and the available computing resources. We used '5e-5' following the recommendations from the BERT developers.

(14) eval_on: 0 for no; 1 for yes. If eval_on = 1, a confusion matrix and a classification report are generated.

(15) proba_on: 0 for no; 1 for yes.

(16) output filename suffix: This assigns a text string at the end of an output file's name

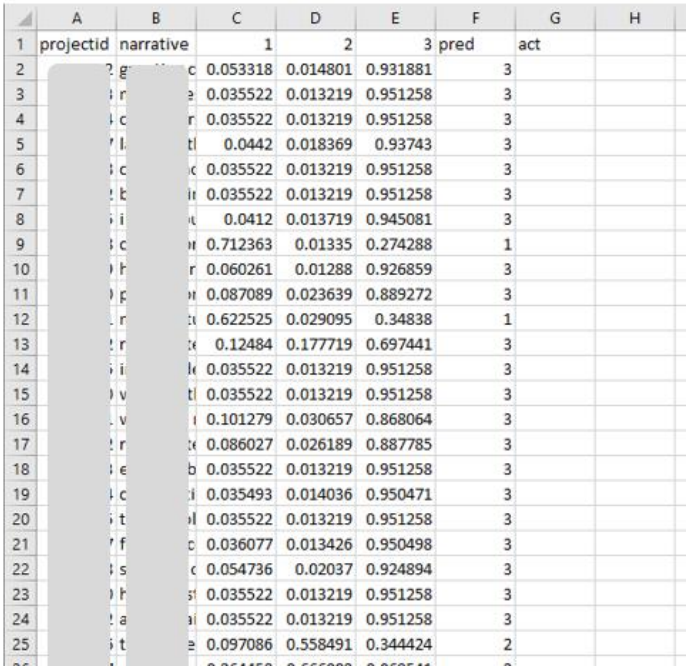
```python
if __name__== "__main__":

    ##########################################################
    ############################## Set Parameter Values ######
    ##########################################################

    ###### 1. Input file & column name and test (unlabeled) dat
    input_filename = "multi_labeled.csv"
    column_name = "text"

    test_unlabel_on = 0

    test_filename = "multi_unlabeled.csv"
    num_class=5

    ###### 2. Sampling applied?
    sampling_on = 0
    sampling_type = 'under'

    ###### 3. Which pretrained model to use?

    ###### 3-1. Model name
    pretrained_model_name = 'bert-base-cased'
    #pretrained_model_name = 'roberta-base'
    #pretrained_model_name = 'vinai/bertweet-base'

    ###### 3-2. Download pretrained model?
    internet_on = 0

    modelname_string = pretrained_model_name.split("/")[-1]
    pretrained_model_folder = "./model_" + modelname_string

    ###### 4. Hyperparameters
    MAX_LEN = 100
    BATCH_SIZE = 16
    EPOCHS = 4
    LEARNING_RATE = 5e-5

    ###### 5. Evaluation & probability file
    eval_on=1
    proba_on=1

    ###### 6. Output filename suffix
    if test_unlabel_on==1:
      label_name = "unlabeled"
    else:
      label_name= "labeled"
```

8. Running Code

Once the parameters are set, the code is ready to run. On a Jupyter Notebook environment, users can simply click the ‘Run’ button in a Jupyter Notebook to implement the BERT-based prediction. On a command prompt, users can type ‘python’ (or ‘python3’ depending on OS) in combination with the code file name and press ‘Enter’ to run the code.

Output File

Once the implementation of the code is completed, an output file with a CSV file extension is created and saved in the folder where the code set is located. Figure 2 shows part of the output file as an example.



	A	B	C	D	E	F	G	H
1	projectid	narrative	1	2	3	pred	act	
2			0.053318	0.014801	0.931881	3		
3			0.035522	0.013219	0.951258	3		
4			0.035522	0.013219	0.951258	3		
5			0.0442	0.018369	0.93743	3		
6			0.035522	0.013219	0.951258	3		
7			0.035522	0.013219	0.951258	3		
8			0.0412	0.013719	0.945081	3		
9			0.712363	0.01335	0.274288	1		
10			0.060261	0.01288	0.926859	3		
11			0.087089	0.023639	0.889272	3		
12			0.622525	0.029095	0.34838	1		
13			0.12484	0.177719	0.697441	3		
14			0.035522	0.013219	0.951258	3		
15			0.035522	0.013219	0.951258	3		
16			0.101279	0.030657	0.868064	3		
17			0.086027	0.026189	0.887785	3		
18			0.035522	0.013219	0.951258	3		
19			0.035493	0.014036	0.950471	3		
20			0.035522	0.013219	0.951258	3		
21			0.036077	0.013426	0.950498	3		
22			0.054736	0.02037	0.924894	3		
23			0.035522	0.013219	0.951258	3		
24			0.035522	0.013219	0.951258	3		
25			0.097086	0.558491	0.344424	2		

Figure 2: Example of Output File

In the file, an instance (row) is assigned probability scores to refer to each of three classes (‘1,’ ‘2,’ and ‘3’) and a label (‘pred’ column) decided by a machine learning model. As the code set run for this example case does not include labels for test data, actual labels (‘act’ column) are not shown. In the example file, the first instance (Row #2) is assigned ‘3’ for a label because its probability score for the class ‘3’ is 0.981881. Using the probability scores assigned to instances, this project filters instances that are highly likely to refer to a specific class.

References

Abramowitz, J., & Kim, J. (2021). C-NARS: An Open-Source Tool for Classification of Narratives in Survey Data. Technical Report, Survey Research Center, University of Michigan.

Beltagy, I., Cohan, A., & Lo, K. (2019). SciBERT: Pretrained contextualized embeddings for scientific text. *arXiv preprint arXiv:1903.10676*.

Berry, Michael W., & Castellanos, Malu (2004). Survey of text mining. Computing Reviews, 45(9), 548.

Boselli, Roberto, Cesarini, Mirko, Mercorio, Fabio, & Mezzanzanica, Mario (2018). Classifying online job advertisements through machine learning. Future Generation Computer Systems, 86, 319-328.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*.

Ikudo, Akina, Lane, Julia I., Staudt, Joseph, & Weinberg, Bruce A. (2019). Occupational classifications: A machine learning approach. Journal of Economic and Social Measurement, 44(2-3), 57-87.

Koroteev, M. (2021). BERT: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*.

Lampos, Vasileios, Aletras, Nikolaos, Geyti, Jens K., Zou, Bin, & Cox, Ingemar J. (2016). Inferring the socioeconomic status of social media users based on behaviour and language. Paper presented at the European conference on information retrieval.

Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C. H., & Kang, J. (2020). BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics, 36*(4), 1234-1240.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., . . . Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Mac Kim, Sunghwan, Xu, Qiongkai, Qu, Lizhen, Wan, Stephen, & Paris, Cécile. (2017). Demographic inference on twitter using recursive neural networks. Paper presented at the Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers).

Preoţiuc-Pietro, Daniel, Lampos, Vasileios, & Aletras, Nikolaos. (2015). An analysis of the user occupational class through Twitter content. Paper presented at the Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).

Preoţiuc-Pietro, Daniel, & Ungar, Lyle. (2018). User-level race and ethnicity predictors from twitter text. Paper presented at the Proceedings of the 27th International Conference on Computational Linguistics.