CS 598 Deep Learning for Healthcare

Final Project

Team 140: Anson Wong & Skyler MacAdam

GitHub link: https://github.com/skylernovak/KeyClass/tree/main

For our final project, we have selected the paper Classifying Unstructured Clinical Notes via Automatic Weak Supervision. We will attempt to reproduce their study with similar results.

› Mount Notebook to Google Drive

This step needs to be completed the first time when Google Colab is opened and you are starting a new session.

[ ] ↳ *1 cell hidden*

Introduction

Background

Assigning accurate diagnostic codes from systems like the International Classification of Diseases (ICD) to clinical notes is a critical task for healthcare providers. These codes are used for clinical documentation, research, and billing purposes. However, manually coding unstructured clinical notes is a time-consuming, costly, and error-prone process. Estimates suggest that 60-80% of manually assigned codes may not accurately reflect the true patient diagnoses[1].

Problem Statement

The problem tackled in this paper is the automatic assignment of ICD-9 diagnostic codes to unstructured patient discharge summaries. This is a challenging multiclass, multilabel text classification task, as there are over 14,000 unique ICD-9 codes that may need to be applied to a single clinical note. The paper examines other multiclass text classification problems with other datasets to show the possabilities that KeyClass can be used for, and argues it can be used for high level ICD-9 code classification on clinical notes if it can perform successfully on these other problems too. One of the datasets used most in the papers discussion is the IMDB dataset, where

KeyClass is identifying if a movie review is positive or negative. KeyClass is given some keywords that a domain expert would provide (or they can also be obtained from inferences from related sources such as Wikipedia), and then able to predict the label of the reviews itself.

Importance and Difficulty

Accurate ICD coding is important for several reasons:

- It allows healthcare providers to better manage patient populations and track disease prevalence.
- It is crucial for correct billing and reimbursement, with coding errors potentially leading to revenue loss or even penalties.
- It enables large-scale clinical research by facilitating the identification of relevant patient cohorts.

Prior work has explored the use of supervised machine learning techniques to automate ICD code assignment, such as hierarchical attention models[2] and multimodal approaches[3]. While these methods have shown promising results, they rely on large amounts of manually annotated training data, which can be costly and time-consuming to obtain. Additionally, as the ICD coding system is periodically revised, these supervised models may struggle to generalize to new codes.

Paper Explanation

The paper *Classifying Unstructured Clinical Notes via Automatic Weak Supervision* proposes a novel approach called KeyClass that aims to address the limitations of existing supervised methods. KeyClass is a weakly supervised text classification framework that learns from class-label descriptions alone, without requiring any human-labeled documents.

The key innovations of KeyClass are:

- Automatically generating interpretable keyword-based labeling functions from class descriptions using pre-trained language models.
- Employing data programming to estimate probabilistic labels from the noisy, automatically generated labeling functions.
- Training a downstream text classifier in a self-supervised manner, using the probabilistic labels.

The paper reports that KeyClass achieves performance comparable to state-of-the-art weakly supervised methods based on 4 real world test classificatgion datasets.

Contributions

This work is significant as it presents a novel weakly supervised approach that can effectively solve the important problem of automated ICD coding, without the need for extensive manual labeling. By leveraging only class descriptions, KeyClass has the potential to make the development and deployment of accurate text classifiers more accessible and affordable, with broader implications for clinical text processing and healthcare data management.

Scope of Reproducibility:

We will be attempting to reproduce the following hypothesis from the KeyClass paper:

1. KeyClass can effectively classify text documents without access to any labeled data.
2. Self-training improves the performance of the downstream classifier.

Methodology

Here, we begin our implementation of our study. Below you will find two sections, Data and Model.

Data

This section we will load and pre-process our data for our experiements.

Model

This section we will construct the model used for the experiment.

```
# Run this cell to install all requirements for this project.
%cd /content/drive/My Drive/CS598FinalProject/
%ls
!pip install -r requirements.txt
```

```
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/
Requirement already satisfied: nvidia-nccl-cu12==2.19.3 in /usr/local/lib/python3.
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python
Requirement already satisfied: triton==2.2.0 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: munkres>=1.0.6 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: tensorboard>=2.9.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: absl-py>=0.4 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: grpcio>=1.48.2 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.10/
Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in /usr/local/lib/pyth
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: protobuf!=4.24.0,>=3.19.6 in /usr/local/lib/python3
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: six>1.9 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dis
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.10/
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.10
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.10/
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/lib/python3.10/d
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/dist-p
```

```
# Run this cell to import the remaining modules needed for this experiment.
%cd /content/drive/My Drive/CS598FinalProject/keyclass/
%ls

# Append paths for imports.
import sys
sys.path.append('/content/drive/My Drive/CS598FinalProject/keyclass/')
sys.path.append('/content/drive/My Drive/CS598FinalProject/scripts/')

# Import files and modules needed for KeyClass
import numpy as np
import utils
import models
import create_lfs
import train_classifier
import torch
import pickle
from os.path import join, exists
import pandas as pd
import os
import train_downstream_model
```

```
/content/drive/My Drive/CS598FinalProject/keyclass
create_lfs.py  __init__.py  models.py  __pycache__/  train_classifier.py  utils.py
['/content', '/env/python', '/usr/lib/python310.zip', '/usr/lib/python3.10', '/usr/lib/p
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Data

Source of the data

The data used in the paper is collected from 4 different datasets. Amazon, DBpedia, IMDB, and AG News. The authors collected this data and included a script with their paper to download copies of these datasets as well for others to reproduce the authors findings. We downloaded this data and included it within out google drive as well. These scripts are downloaded using the script `get_data.sh`.

These 4 datasets are used in common multiclass text classification problems, such as in movie review sentiments from IMBD.

Statistics

Utilizing the IMDB dataset, the paper attempts to classify movie reviews as either `positive` or `negative`. The KeyClass model was provided with common sense descriptions for these classes from industry experts. This constitutes minimal human input required for KeyClass to perform the classifications. For example, a positive review is typically assocaited with words such as "amazing", "exciting", or "fun" while negative reviews are more often associated with words such as "terrible", "boring", or "awful".

Both the `train` and `test` datasets contains 25000 sample movie reviews each. KeyClass attempts to label each review as either `positive` or `negative` based on these reviews. Each data split contains approximately 32 million characters, and the files are approximately 30 MB each.

Data process

The KeyClass framework consists of several key steps:

1. Finding relevant keywords and phrases from the class descriptions using a pre-trained language model (such as BERT).
2. Constructing labeling functions based on the identified keywords and using data programming to generate probabilistic labels for the training data.
3. Training a downstream classifier on the probabilistically labeled training data.
4. Self-training the downstream model on the entire training dataset to refine the classifier.

```
# Declare configuration variables
config_file_path = r'/content/drive/My Drive/CS598FinalProject/config_files/config_imdb.yml'
random_seed = 0 # Random seed for experiments
args = utils.Parser(config_file_path=config_file_path).parse()
```

The following block of code processes the text and creates embeddings. It is highly encouraged to use the Google Colab t4 GPU runtime type to execute this block of code. Using the T4 GPU, this takes a couple of minutes to execute. Using the CPU runtime type, this can take hours.

This block of code has been commented out as the embeddings have been saved in pickle files for later use in this notebook. This step is not required to be executed again, and is left here for informational / demonstrative purposes.

```
# args = utils.Parser(config_file_path=config_file_path).parse()

# if args['use_custom_encoder']:
#     model = models.CustomEncoder(pretrained_model_name_or_path=args['base_encoder'],
#         device='cuda' if torch.cuda.is_available() else 'cpu')
# else:
#     model = models.Encoder(model_name=args['base_encoder'],
#         device='cuda' if torch.cuda.is_available() else 'cpu')

# for split in ['train', 'test']:
#     sentences = utils.fetch_data(dataset=args['dataset'], split=split, path=args['data_pat
#     embeddings = model.encode(sentences=sentences, batch_size=args['end_model_batch_size']
#                               show_progress_bar=args['show_progress_bar'],
#                               normalize_embeddings=args['normalize_embeddings'])
#     with open(join(args['data_path'], args['dataset'], f'{split}_embeddings.pkl'), 'wb') a
#         pickle.dump(embeddings, f)
```

Here we now load the training data, and then create the labeling function. Finally we create the probablistic labels from the training document.

```
    # Load training data
    train_text = utils.fetch_data(dataset=args['dataset'], path=args['data_path'], split='train'

    training_labels_present = False
    if exists(join(args['data_path'], args['dataset'], 'train_labels.txt')):
        with open(join(args['data_path'], args['dataset'], 'train_labels.txt'), 'r') as f:
            y_train = f.readlines()
        y_train = np.array([int(i.replace('\n','')) for i in y_train])
        training_labels_present = True
    else:
        y_train = None
        training_labels_present = False
        print('No training labels found!')

    with open(join(args['data_path'], args['dataset'], 'train_embeddings.pkl'), 'rb') as f:
        X_train = pickle.load(f)

    # Print dataset statistics
    print(f"Getting labels for the {args['dataset']} data...")
    print(f'Size of the data: {len(train_text)}')
    if training_labels_present:
        print('Class distribution', np.unique(y_train, return_counts=True))

    # Load label names/descriptions
    label_names = []
    for a in args:
        if 'target' in a: label_names.append(args[a])

    # Creating labeling functions
    labeler = create_lfs.CreateLabellingFunctions(base_encoder=args['base_encoder'],
                                                  device=torch.device(args['device']),
                                                  label_model=args['label_model'])
    proba_preds = labeler.get_labels(text_corpus=train_text, label_names=label_names, min_df=arg
                                     ngram_range=args['ngram_range'], topk=args['topk'], y_train=
                                     label_model_lr=args['label_model_lr'], label_model_n_epochs=
                                     verbose=True, n_classes=args['n_classes'])

    y_train_pred = np.argmax(proba_preds, axis=1)

    # Save the predictions
    if not os.path.exists(args['preds_path']): os.makedirs(args['preds_path'])
    with open(join(args['preds_path'], f"{args['label_model']}_proba_preds.pkl"), 'wb') as f:
        pickle.dump(proba_preds, f)

    # Print statistics
    print('Label Model Predictions: Unique value and counts', np.unique(y_train_pred, return_cou
    if training_labels_present:
        print('Label Model Training Accuracy', np.mean(y_train_pred==y_train))

        # Log the metrics
        training_metrics_with_gt = utils.compute_metrics(y_preds=y_train_pred, y_true=y_train, a
```

```
    utils.log(metrics=training_metrics_with_gt, filename='label_model_with_ground_truth',
        results_dir=args['results_path'], split='train')
```

```
 'excellently' 'exquisite' 'extremely well' 'fabulous' 'fairly good'
 'fantastic' 'far best' 'film excellent' 'find good' 'fine job'
 'fine performances' 'finest' 'first rate' 'get good' 'give good'
 'gives best' 'gives good' 'gives great' 'good' 'good action' 'good also'
 'good although' 'good bad' 'good choice' 'good direction' 'good either'
 'good enough' 'good entertainment' 'good especially' 'good even'
 'good example' 'good film' 'good films' 'good first' 'good good'
 'good great' 'good idea' 'good movie' 'good music' 'good one' 'good ones'
 'good original' 'good part' 'good parts' 'good people' 'good performance'
 'good performances' 'good really' 'good reviews' 'good say' 'good show'
 'good special' 'good stuff' 'good taste' 'good thing' 'good things'
 'good think' 'good though' 'good tv' 'good use' 'good well' 'good work'
 'good would' 'good writing' 'got good' 'got great' 'great'
 'great character' 'great example' 'great film' 'great fun' 'great love'
 'great music' 'great one' 'great really' 'great show' 'great supporting'
 'great things' 'great time' 'greats' 'high quality' 'high rating'
 'highly recommend' 'highly recommended' 'however like' 'idea good'
 'like best' 'like good' 'like great' 'liked' 'liked one' 'looks great'
 'lot good' 'lot great' 'love good' 'lovely' 'luxury' 'made good'
 'made great' 'made well' 'make good' 'make great' 'makes good'
 'makes great' 'many good' 'many great' 'many reviewers' 'many reviews'
 'marvelously' 'may good' 'mean good' 'might good' 'movie excellent'
 'movie good' 'movie recommend' 'movie wonderful' 'much enjoyed'
 'much good' 'music good' 'music great' 'nearly good' 'nice' 'nice look'
 'nothing better' 'one best' 'one finest' 'one good' 'one great'
 'one like' 'overall good' 'overall think' 'particularly good'
 'people good' 'people like' 'performances good' 'perhaps best'
 'personal favorite' 'personally think' 'pleasant' 'positive reviews'
 'positive thing' 'possibly best' 'praise' 'prefer' 'prefers'
 'pretty decent' 'pretty good' 'probably best' 'probably good'
 'probably like' 'put good' 'qualities' 'quality' 'quality acting'
 'quite enjoyable' 'quite good' 'quite like' 'rather good' 'rating 10'
 'read review' 'read reviews' 'reading reviews' 'real good'
 'really appreciate' 'really enjoy' 'really enjoyed' 'really good'
 'really great' 'really like' 'really liked' 'really loved' 'really nice'
 'really recommend' 'recommend' 'recommend anyone' 'recommend everyone'
 'recommend film' 'recommend movie' 'recommend one' 'recommend see'
 'recommend watch' 'recommend watching' 'recommendation' 'recommended'
 'recommending' 'redeeming quality' 'reviews' 'satisfactory' 'say best'
 'say good' 'see good' 'seen good' 'show good' 'solid performances'
 'something better' 'something good' 'something interesting' 'splendid'
 'still enjoyable' 'still good' 'still great' 'strongly recommend'
 'surprisingly good' 'tasteful' 'terrific' 'thing good' 'think best'
 'think good' 'think great' 'though good' 'thought good' 'thought great'
 'time great' 'top notch' 'truly great' 'two best' 'want good'
 'watch good' 'well crafted' 'well good' 'well great' 'well made'
 'well produced' 'well worth' 'wonderful' 'wonderful film' 'wonderful job'
 'wonderful life' 'wonderful movie' 'wonderfully' 'worth look'
 'worth mentioning' 'worth seeing' 'worthwhile' 'would good'
 'would recommend']
==== Training the label model ====
100%|██████████| 100/100 [00:08<00:00, 11.32epoch/s]
Label Model Predictions: Unique value and counts (array([0, 1]), array([ 8914, 160
Label Model Training Accuracy 0.70016
Saving results in ../results/imdb/train_label_model_with_ground_truth_14-Apr-2024-
```

Train the Downstream Model

Find Class Descriptions

KeyClass starts with just the class descriptions (e.g., "positive review" and "negative review") without any labeled training data.

Find Relevant Keywords

Using the class descriptions and a pre-trained language model (like BERT), KeyClass automatically extracts keywords and phrases that are highly indicative of each class.

Probabilistically Label the Data

KeyClass uses the extracted keywords as labeling functions and applies data programming techniques to generate probabilistic labels for the entire training dataset.

Train Downstream Model

KeyClass then trains a downstream text classification model (e.g., a feed-forward neural network) using the probabilistically labeled training data. It initially only uses the most confidently labeled samples to train the model.

The key advantages of this approach are that it requires no manually labeled training data and the labeling functions are automatically generated in an interpretable way.

```
args = utils.Parser(config_file_path=config_file_path).parse()

# Set random seeds
random_seed = random_seed
torch.manual_seed(random_seed)
np.random.seed(random_seed)

X_train_embed_masked, y_train_lm_masked, y_train_masked, \
    X_test_embed, y_test, training_labels_present, \
    sample_weights_masked, proba_preds_masked = train_downstream_model.load_data(args)

# Train a downstream classifier

if args['use_custom_encoder']:
    encoder = models.CustomEncoder(pretrained_model_name_or_path=args['base_encoder'], devic
else:
    encoder = models.Encoder(model_name=args['base_encoder'], device=args['device'])

classifier = models.FeedForwardFlexible(encoder_model=encoder,
                                        h_sizes=args['h_sizes'],
                                        activation=eval(args['activation']),
                                        device=torch.device(args['device']))
print('\n===== Training the downstream classifier =====\n')
model = train_classifier.train(model=classifier,
                            device=torch.device(args['device']),
                            X_train=X_train_embed_masked,
                            y_train=y_train_lm_masked,
                            sample_weights=sample_weights_masked if args['use_noise_aware_lo
                            epochs=args['end_model_epochs'],
                            batch_size=args['end_model_batch_size'],
                            criterion=eval(args['criterion']),
                            raw_text=False,
                            lr=eval(args['end_model_lr']),
                            weight_decay=eval(args['end_model_weight_decay']),
                            patience=args['end_model_patience'])


end_model_preds_train = model.predict_proba(torch.from_numpy(X_train_embed_masked), batch_si
end_model_preds_test = model.predict_proba(torch.from_numpy(X_test_embed), batch_size=512, r
```

```
### Self-Train the Model

#### Encode Full Training Set

After the initial downstream model is trained,
training dataset using the model's encoder.

#### Self-Train the Model
```

Self-Train the Model

Encode Full Training Set

After the initial downstream model is trained, KeyClass encodes the full training dataset using

```
KeyClass then iteratively self-trains the down
encoded training set. It does this by:
- Making predictions on the unlabeled training
- Selecting the most confidently predicted sam
- Retraining the model on the combined labeled

#### Evaluate on Test Set

The self-trained model is then evaluated on th
the final performance.

The self-training process allows the model to
decision boundaries by iteratively learning fr
unlabeled data. This can lead to improved perf
initial model is trained on a limited subset o
```


the model's encoder.

Self-Train the Model

KeyClass then iteratively self-trains the downstream model using the entire encoded training set. It does this by:

- Making predictions on the unlabeled training data
- Selecting the most confidently predicted samples
- Retraining the model on the combined labeled and pseudo-labeled data

Evaluate on Test Set

The self-trained model is then evaluated on the held-out test set to measure the final performance.

The self-training process allows the model to refine its representations and decision boundaries by iteratively learning from its own predictions on the unlabeled data. This can lead to improved performance, especially when the initial model is trained on a limited subset of the data.

```
# Fetching the raw text data for self-training
X_train_text = utils.fetch_data(dataset=args['dataset'], path=args['data_path'], split='trai
X_test_text = utils.fetch_data(dataset=args['dataset'], path=args['data_path'], split='test'

model = train_classifier.self_train(model=model,
                                    X_train=X_train_text,
                                    X_val=X_test_text,
                                    y_val=y_test,
                                    device=torch.device(args['device']),
                                    lr=eval(args['self_train_lr']),
                                    weight_decay=eval(args['self_train_weight_decay']),
                                    patience=args['self_train_patience'],
                                    batch_size=args['self_train_batch_size'],
                                    q_update_interval=args['q_update_interval'],
                                    self_train_thresh=eval(args['self_train_thresh']),
                                    print_eval=True)


end_model_preds_test = model.predict_proba(X_test_text, batch_size=args['self_train_batch_si


# Print statistics
testing_metrics = utils.compute_metrics_bootstrap(y_preds=np.argmax(end_model_preds_test, ax
                                                  y_true=y_test,
                                                  average=args['average'],
                                                  n_bootstrap=args['n_bootstrap'],
                                                  n_jobs=args['n_jobs'])
print(testing_metrics)
```

After training the downstream classifier and self-training it, the notebook performs the following evaluation steps:

1. Load Test Data: It loads the test dataset, including the test text samples (X_test_embed) and the ground truth test labels (y_test).
2. Evaluate Trained Model on Test Set: It uses the trained downstream model to make predictions on the test set, obtaining the test set predictions (end_model_preds_test).
3. Compute Test Metrics: It computes various performance metrics on the test set predictions, including:

Metrics using ground truth labels (y_test): Compute metrics using utils.compute_metrics_bootstrap(), which does bootstrap sampling to get confidence intervals. This provides an assessment of the model's true performance on the test set.

Metrics using label model predictions (y_train_lm_masked): Compute metrics using utils.compute_metrics(). This shows how the model performs compared to the noisy labels used for training.

Print Test Metrics: The notebook prints out the test set performance metrics, showing the model's accuracy, precision, recall, and F1-score.

Self-Train the Model: Finally, it loads the self-trained model checkpoint and evaluates the self-trained model on the test set, printing the updated test set performance metrics.

```
end_model_path='/content/drive/My Drive/CS598FinalProject/models/end_model.pth'
end_model_self_trained_path='/content/drive/My Drive/CS598FinalProject/models/end_model_self

args = utils.Parser(config_file_path=config_file_path).parse()

# Set random seeds
random_seed = random_seed
torch.manual_seed(random_seed)
np.random.seed(random_seed)

X_train_embed_masked, y_train_lm_masked, y_train_masked, \
    X_test_embed, y_test, training_labels_present, \
    sample_weights_masked, proba_preds_masked = train_downstream_model.load_data(args)

model = torch.load(end_model_path)

end_model_preds_train_key_class = model.predict_proba(torch.from_numpy(X_train_embed_masked)
end_model_preds_test_key_class = model.predict_proba(torch.from_numpy(X_test_embed), batch_s

# Print statistics
if training_labels_present:
    training_metrics_with_gt = utils.compute_metrics(y_preds=np.argmax(end_model_preds_trair
                                                        y_true=y_train_masked,
                                                        average=args['average'])
    print('training_metrics_with_gt', training_metrics_with_gt)

training_metrics_with_lm = utils.compute_metrics(y_preds=np.argmax(end_model_preds_train_key
                                                    y_true=y_train_lm_masked,
                                                    average=args['average'])
print('training_metrics_with_lm', training_metrics_with_lm)

testing_metrics = utils.compute_metrics_bootstrap(y_preds=np.argmax(end_model_preds_test_key
                                                    y_true=y_test,
                                                    average=args['average'],
                                                    n_bootstrap=args['n_bootstrap'],
                                                    n_jobs=args['n_jobs'])
print('testing_metrics', testing_metrics)


print('\n===== Self-training the downstream classifier =====\n')

# Fetching the raw text data for self-training
X_train_text = utils.fetch_data(dataset=args['dataset'], path=args['data_path'], split='trai
X_test_text = utils.fetch_data(dataset=args['dataset'], path=args['data_path'], split='test'

model = torch.load(end_model_self_trained_path)

end_model_preds_test_self_training = model.predict_proba(X_test_text, batch_size=args['self_


# Print statistics
```

```
testing_metrics = utils.compute_metrics_bootstrap(y_preds=np.argmax(end_model_preds_test_se]
                                                  y_true=y_test,
                                                  average=args['average'],
                                                  n_bootstrap=args['n_bootstrap'],
                                                  n_jobs=args['n_jobs'])
print('testing_metrics after self train', testing_metrics)
```

```
Confidence of least confident data point of class 0: 0.9118951704039087
Confidence of least confident data point of class 1: 0.9999157389338196

==== Data statistics ====
Size of training data: (25000, 768), testing data: (25000, 768)
Size of testing labels: (25000,)
Size of training labels: (25000,)
Training class distribution (ground truth): [0.5 0.5]
Training class distribution (label model predictions): [0.35656 0.64344]

KeyClass only trains on the most confidently labeled data points! Applying mask...

==== Data statistics (after applying mask) ====
Size of training data: (7000, 768)
Size of training labels: (7000,)
Training class distribution (ground truth): [0.55057143 0.44942857]
Training class distribution (label model predictions): [0.5 0.5]
training_metrics_with_gt [0.9211428571428572, 0.9243499652296476, 0.9211428571428572]
training_metrics_with_lm [0.9188571428571428, 0.9190760887320724, 0.9188571428571428]
[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  30 tasks      | elapsed:   11.0s
[Parallel(n_jobs=10)]: Done  81 out of 100 | elapsed:   11.8s remaining:    2.8s
[Parallel(n_jobs=10)]: Done 100 out of 100 | elapsed:   20.1s finished
testing_metrics [[0.8533216  0.00209534]
 [0.86366085 0.00188531]
 [0.8533216  0.00209534]]

===== Self-training the downstream classifier =====

[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  40 tasks      | elapsed:    0.7s
testing_metrics after self train [[0.8714468  0.00204239]
 [0.87203099 0.0020189 ]
 [0.8714468  0.00204239]]
[Parallel(n_jobs=10)]: Done 100 out of 100 | elapsed:    1.4s finished
```

Ablations

We have two ablations for this study:

1. [No self-training (evaluate just the initial weakly supervised model)]()
2. [Majority vote labels, instead of data programming.]()

You can find these ablations in the linked Google Colab notebooks. Their process is nearly identical to the process in this notebook.

Results

```
file_path = '/content/drive/My Drive/CS598FinalProject/y_test.npy'
y_test = np.load(file_path)
```

```
file_path = '/content/drive/My Drive/CS598FinalProject/y_train_masked.npy'
y_train_masked = np.load(file_path)
```

```
file_path = '/content/drive/My Drive/CS598FinalProject/y_train_lm_masked.npy'
y_train_lm_masked = np.load(file_path)
```

```
file_path = '/content/drive/My Drive/CS598FinalProject/end_model_preds_train_key_class.npy'
end_model_preds_train_key_class = np.load(file_path)
```

```
file_path = '/content/drive/My Drive/CS598FinalProject/end_model_preds_test_key_class.npy'
end_model_preds_test_key_class = np.load(file_path)
```

```
file_path = '/content/drive/My Drive/CS598FinalProject/end_model_preds_test_self_training.np
end_model_preds_test_self_training = np.load(file_path)
```

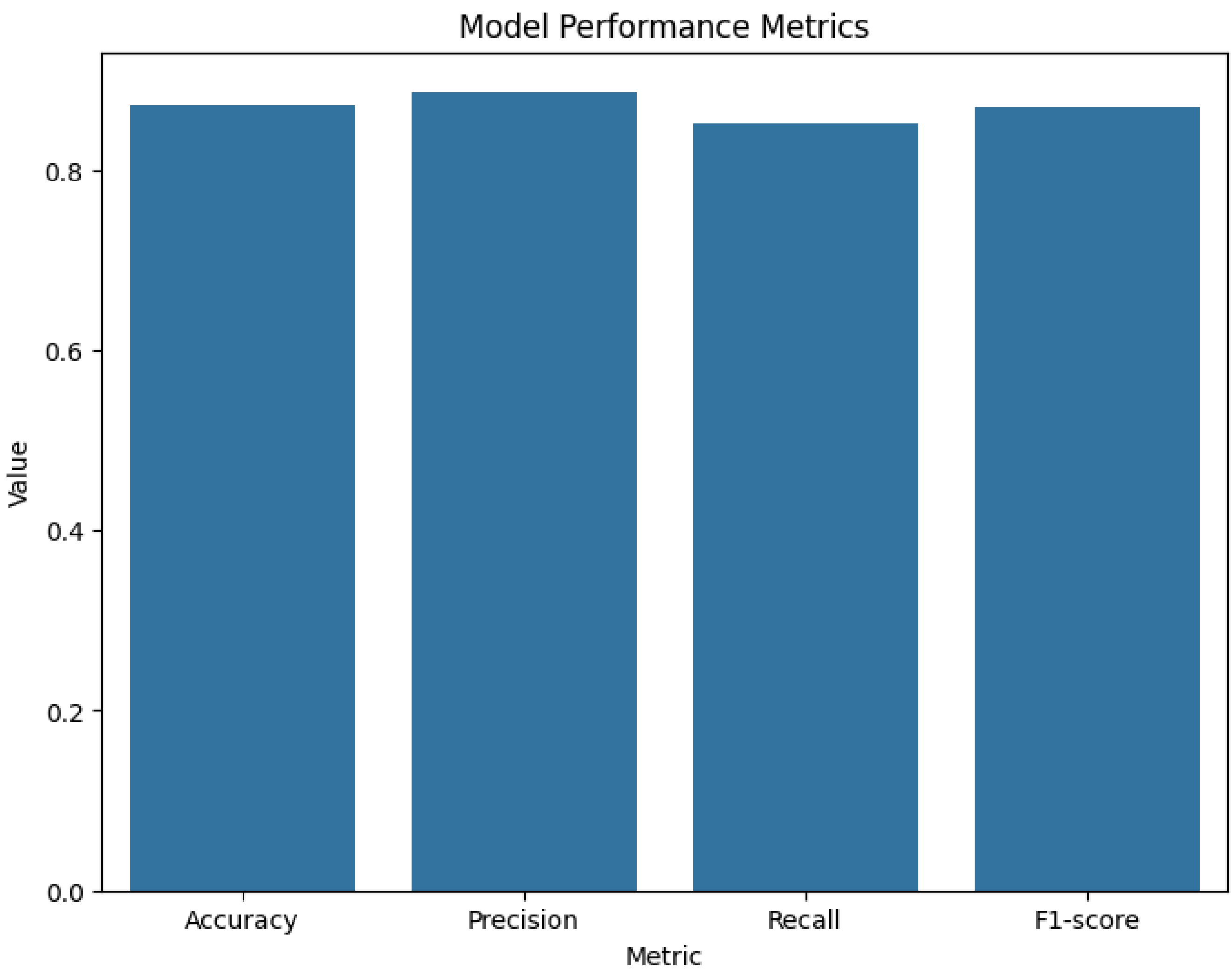
```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# weak supervision sources (keywords and phrases) results
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Compute the metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a bar plot
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-score']
values = [accuracy, precision, recall, f1]
plt.figure(figsize=(8, 6))
sns.barplot(x=metrics, y=values)
plt.title('Model Performance Metrics')
plt.xlabel('Metric')
plt.ylabel('Value')
plt.show()
```



```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, precision_recall_curve, roc_curve, auc
import seaborn as sns

# Plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, title='Confusion Matrix', labels=None):
    cm = confusion_matrix(y_true, y_pred)
    if labels:
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=l
    else:
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(title)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

# Plot precision-recall curve
def plot_precision_recall_curve(y_true, y_score, title):
    precision, recall, _ = precision_recall_curve(y_true, y_score)
    plt.plot(recall, precision, marker='.')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title(title)
    plt.show()

# Plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label='ROC curve (AUC = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Random')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()


# Plot histogram of predicted probabilities
def plot_predicted_probabilities_histogram(y_prob):
    plt.hist(y_prob, bins=10)
    plt.xlabel('Predicted Probability')
    plt.ylabel('Frequency')
    plt.title('Histogram of Predicted Probabilities')
    plt.show()


# Example usage
y_preds=np.argmax(end_model_preds_train_key_class, axis=1)

# plot_predicted_probabilities_histogram(end_model_preds_train_key_class)
```