605.744: Information Retrieval
Programming Assignment #4: Binary Text Classification

Sabbir Ahmed

October 27, 2022

Contents

Introduction

This paper describes the classification of the Systematic Review dataset through exploratory analysis, grid searching for optimal estimators and their hyper-parameters, and determining the best features.

Technical Background

All of the source code is in Python 3.10. The program is split into several modules and follows an object oriented structure. The following is the directory structure of the source code:

```
.
├── ir/
│   ├── __init__.py
│   ├── const.py
│   ├── files.py
│   ├── metrics.py
│   ├── model.py
│   └── normalize.py
├── models/
├── outputs/
└── run.py
```

Figure 1: Directory Hierarchy of Assignment 4

The source code for all of the files are attached in Appendix A.

The total number of non-empty lines of code for the program totals to under 750.

Classes

`scikit-learn` was used to implement portions of this assignment.

Some classes from Assignment 3 were used in this project:

- the driver script `run.py` was modified with the relevant flags
- the methdods in `files.IO` were simplified to handle only plain text and joblib binaries
- the `files.CorpusFile` class was modified to process TSV files and transform the content into a list of mapped values
- the `normalize.Normalizer` class was used to compare text tokenization methods
- the `lexer` classes were replaced by the `CountVectorizer` class provided by `scikit-learn`

Exploratory Analysis

The dataset is a collection of tab separated value (TSV) files with 10 features.

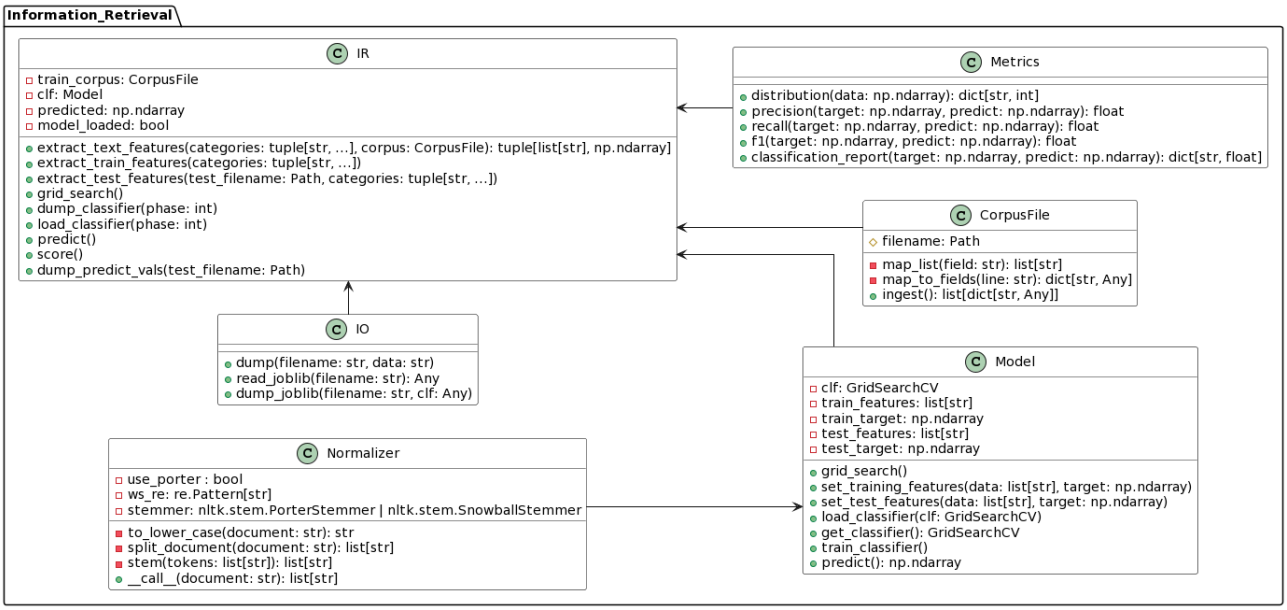

Figure 2: UML of Information Retrieval

Table 1: Description of the features of the Systematic Review dataset

Feature	**Description**
Assessment	-1, 0, or 1; zero indicates unknown; 1=accept, -1=reject
DocID	Unique ID. Usually PubMed ID, or hashed ID
Title	Article title
Authors	List of authors (poss. separated with ";"). Unnormalized.
Journal	Journal title
ISSN	Numeric code for journal (possibly a list)
Year	Publication year
language	Trigram for language code (e.g., "eng")
Abstract	Several sentence abstract from article
Keywords	List of keywords (formatting is highly variable) - should be ";" separated.

The *Assessment* feature in the training and development datasets is the target binary value, and will be predicted for the testing dataset. The feature is heavily imbalanced, with the following distribution in the training dataset:

Table 2: Distribution of *Assessment* in the training dataset

Value	**Count**
-1	21,000
1	700

The following features are considered text features, where they are represented as single strings:

- DocID
- Title

- Language
- Abstract

The following features are considered list of text features, where they are represented as string lists:

- Authors
- Journal
- ISSN
- Keywords

The remaining (optional) feature, *Year*, is represented as an integer.

Classification Algorithms

For classification, machine learning algorithms implemented by `scikit-learn` were used.

For text tokenizing and normalizing, the `CountVectorizer` and `TfidfTransformer` classes by `scikit-learn` were used. `CountVectorizer` ingested text values and created bags-of-words. This data structure was further transformed using `TfidfTransformer` to assign TF-IDF values to the terms in the vocabulary.

Scoring

To compare performances of the models, the following metrics were emphasize:

- Precision (P): $\frac{TP}{TP + FP}$
- Recall (R): $\frac{TP}{TP + FN}$
- F1-score: $2 \cdot \frac{P \cdot R}{P + R}$

These scores are computed in `metrics.Metrics`.

Experiments

Baseline

For the initial phase, only the *Title* feature was used as the feature. As a text feature, each of the values were tokenized, vectorized, and their TF-IDF values were used to predict the corresponding target value. The previously implemented `normalize.Normalizer` class was used as the tokenizer to `CountVectorizer`.

The text vectorizers from `scikit-learn` were used with their default parameters. For classification, the linear support vector machine (SVM) with stochastic gradient descent (SGD), `SGDClassifier(loss="hinge")` was used.

To account for the skewed data, an additional parameter $class\_weight = \{1 : 30\}$ was used.

After execution, the model yielded the scores in Table 3:

Table 3: Scores using only *Title* as the feature

Metric	**Score**
Precision	0.195
Recall	0.640
F1-Score	0.299

Experiment #1

To improve the scores of the model, the features *Abstract* and *Keywords* were added. The vocabulary from the 3 features were merged and tokenized through the vectorizers.

After execution, the model yielded the scores listed in Table 4:

Table 4: Scores using *Title*, *Abstract* and *Keywords* as the features

Metric	**Score**
Precision	0.330
Recall	0.747
F1-Score	0.458

Experiment #2

In addition to the features, the grid searching algorithm by scikit-learn, `GridSearchCV`, was used to find the best-performing hyper-parameters. The following parameters were used:

1. Tokenizer for `CountVectorizer`:
   (a) None
   (b) `Normalizer()`
      i. With one of the following stemmers:
         A. Snowball stemmer
         B. Porter stemmer
      ii. Combined with the following stop words list options:
         A. No stopwords removed
         B. Custom stop words list generated from previous assignments
         C. English stop words provided by scikit-learn
2. Class weight for `SGDClassifier`: $\{\{1 : i\}, 3 \leq i \leq 30\}$

In total, 196 combinations were exhaustively searched to find the optimal scores.

Table 5 lists the parameters with the best scores.

Along with the hyper-parameter search, various combinations of features were tested as well. Features such as *Authors* and *Year* do not appear to contribute to the model's performance, and

Table 5: Optimal parameters determined via grid search

Parameter	**Value**
Tokenizer for `CountVectorizer`	None
Stop words for `CountVectorizer` tokenization	None
Class weight for `SGDClassifier`	{1: 3}

Journal appears to degrade the performance. The optimal features were determined to be: {*Title*, *Abstract*, *Keywords*, *Language*}.

After execution, the model yielded the scores in Table 6:

Table 6: Scores using *Title*, *Abstract*, *Keywords* and *Language* as the features

Metric	**Score**
Precision	0.633
Recall	0.460
F1-Score	0.533

Analysis

The final scores of the

Experiment #3

Analysis

The final scores of the

A Source Code

Code Listing 1: ir/__init__.py

```
from pathlib import Path

import numpy as np

from .const import TARGET_FIELD, LIST_FEATURE_FIELDS, JHED
from .files import CorpusFile, IO
from .metrics import Metrics
from .model import Model


class InformationRetrieval:
    def __init__(self, train_filename: Path) -> None:

        self.train_corpus: CorpusFile = CorpusFile(train_filename)

        self.clf: Model = Model()

        self.predicted: np.ndarray
        self.model_loaded = False
```

```
    def use_xgb(self) -> None:

        self.clf.use_xgb()

    def __extract_text_features(
        self, categories: tuple[str, ...], corpus: CorpusFile
    ) -> tuple[list[str], np.ndarray]:

        docs = corpus.ingest()
        target = np.array([i[TARGET_FIELD] for i in docs])
        features: list[str] = []

        for row in docs:

            feature_list = []

            for feature in categories:

                if feature in LIST_FEATURE_FIELDS:
                    feature_list.extend(row[feature])

                else:
                    feature_list.append(row[feature])

            features.append(" ".join(feature_list))

        return features, target

    def extract_train_features(self, categories: tuple[str, ...]) -> None:

        features, target = self.__extract_text_features(
            categories, self.train_corpus
        )
        self.clf.set_training_features(features, target)
        print("train targets:", Metrics.distribution(target))

    def extract_test_features(
        self, test_filename: Path, categories: tuple[str, ...]
    ) -> None:

        if self.model_loaded:

            test_corpus = CorpusFile(test_filename)
            features, target = self.__extract_text_features(
                categories, test_corpus
            )
            self.clf.set_test_features(features, target)
            print("test targets:", Metrics.distribution(target))

        else:
            raise AttributeError("Classifier model not generated yet")

    def grid_search(self) -> None:

        self.clf.grid_search()
        self.model_loaded = True

    def dump_classifier(self, phase: int) -> None:
```

```
        IO.dump_joblib(f"models/model-{phase}", self.clf.get_classifier())

    def load_classifier(self, phase: int) -> None:

        self.clf.load_classifier(IO.read_joblib(f"models/model-{phase}"))
        self.model_loaded = True

    def predict(self) -> None:

        self.predicted = self.clf.predict()

    def score(self) -> None:

        print("predicted targets:", Metrics.distribution(self.predicted))
        print(
            Metrics.classification_report(self.clf.test_target, self.predicted)
        )

    def dump_predict_vals(self, test_filename: Path) -> None:

        test_corpus = CorpusFile(test_filename)
        doc_ids, _ = self.__extract_text_features(("docid",), test_corpus)
        output_pairs = "\n".join(
            f"{doc_id}\t{p}" for doc_id, p in zip(doc_ids, self.predicted)
        )
        IO.dump(f"outputs/{JHED}", output_pairs)
```

Code Listing 2: ir/files.py

```
import joblib
from pathlib import Path
from typing import Any

from .const import TARGET_FIELD, FEATURE_FIELDS, FIELD_DELIM, LIST_DELIM


class CorpusFile:
    def __init__(self, filename: Path) -> None:

        self.filename: Path = filename

    @staticmethod
    def __map_list(field: str) -> list[str]:

        return field.split(LIST_DELIM)

    def __map_to_fields(self, line: str) -> dict[str, Any]:

        mapped_field: dict[str, Any] = dict(
            zip((TARGET_FIELD, *FEATURE_FIELDS), line.split(FIELD_DELIM))
        )

        # assessment
        # mapped_field["assessment"] = int(mapped_field["assessment"])
        mapped_field["assessment"] = (
            0 if int(mapped_field["assessment"]) == -1 else 1
        )

        # authors
```

```
        mapped_field["authors"] = self.__map_list(mapped_field["authors"])

        # journal
        mapped_field["journal"] = self.__map_list(mapped_field["journal"])

        # issn
        mapped_field["issn"] = self.__map_list(mapped_field["issn"])

        # year
        if mapped_field["year"]:
            if not (" " in mapped_field["year"] or "-" in mapped_field["year"]):
                mapped_field["year"] = int(mapped_field["year"])

        # keywords
        mapped_field["keywords"] = self.__map_list(mapped_field["keywords"])

        return mapped_field

    def ingest(self) -> list[dict[str, Any]]:

        docs: list[dict[str, Any]] = []

        with open(self.filename) as fp:
            for line in fp:
                if line:
                    docs.append(self.__map_to_fields(line[:-1]))

        return docs


class IO:
    @staticmethod
    def dump(filename: str, data: str) -> None:

        with open(f"{filename}.txt", "w") as fp:
            fp.write(data)
        print(f"Dumped to '{filename}.txt'")

    @staticmethod
    def read_joblib(filename: str) -> Any:

        return joblib.load(f"{filename}.joblib")

    @staticmethod
    def dump_joblib(filename: str, clf: Any) -> None:

        joblib.dump(clf, f"{filename}.joblib")
        print(f"Dumped model to '{filename}.joblib'")
```

Code Listing 3: ir/metrics.py

```
import numpy as np


class Metrics:
    @staticmethod
    def distribution(data: np.ndarray) -> dict[str, int]:

        unique, counts = np.unique(data, return_counts=True)
```

```
        return dict(zip(unique, counts))

    @staticmethod
    def precision(target: np.ndarray, predict: np.ndarray) -> float:

        tp = 0
        fp = 0
        for t, p in zip(target, predict):

            if p == 1:
                if t == 1:
                    tp += 1
                else:
                    fp += 1

        return tp / (tp + fp)

    @staticmethod
    def recall(target: np.ndarray, predict: np.ndarray) -> float:

        tp = 0
        fn = 0
        for t, p in zip(target, predict):

            if t == 1:
                if p == 1:
                    tp += 1
                else:
                    fn += 1

        return tp / (tp + fn)

    @staticmethod
    def f1(target: np.ndarray, predict: np.ndarray) -> float:

        p = Metrics.precision(predict, target)
        r = Metrics.recall(predict, target)
        return (2 * p * r) / (p + r)

    @staticmethod
    def classification_report(
        target: np.ndarray, predict: np.ndarray
    ) -> dict[str, float]:

        return {
            "precision": round(Metrics.precision(target, predict), 3),
            "recall": round(Metrics.recall(target, predict), 3),
            "f1": round(Metrics.f1(target, predict), 3),
        }
```

Code Listing 4: ir/normalize.py

```
import re

from nltk import stem
from sklearn.feature_extraction import _stop_words

# fmt: off
STOPWORDS: set[str] = {
```

```
    # contractions
    "aren't", "ain't", "can't", "could've", "couldn't", "didn't", "doesn't",
    "don't", "hadn't", "hasn't", "haven't", "he'd", "he'll", "he's",
    "i'd", "i'll", "i'm", "i've", "isn't", "it'll", "it'd",
    "it's", "let's", "mightn't", "might've'", "mustn't", "must've'", "shan't",
    "she'd", "she'll", "she's", "should've", "shouldn't", "that'll", "that's",
    "there's", "they'd", "they'll", "they're", "they've", "wasn't", "we'd",
    "we'll", "we're", "we've", "weren't", "what'll", "what're", "what's",
    "what've", "where's", "who'd", "who'll", "who're", "who's", "who've",
    "won't", "wouldn't", "would've", "y'all", "you'd", "you'll", "you're",
    "you've",
    # NLTK stopwords
    "a", "all", "am", "an", "and", "any",
    "are", "as", "at", "be", "because", "been", "being",
    "but", "by", "can", "cannot", "could", "did", "do",
    "does", "doing", "for", "from", "had", "has", "have",
    "having", "he", "her", "here", "hers", "herself", "him",
    "himself", "his", "how", "i", "if", "in", "is",
    "it", "its", "itself", "just", "let", "may", "me",
    "might", "must", "my", "myself", "need", "no", "nor",
    "not", "now", "o", "of", "off", "on", "once",
    "only", "or", "our", "ours", "ourselves", "shall", "she",
    "should", "so", "some", "such", "than", "that", "the",
    "their", "theirs", "them", "themselves", "then", "there", "these",
    "they", "this", "those", "to", "too", "very", "was",
    "we", "were", "what", "when", "where", "which", "who",
    "whom", "why", "will", "with", "would", "you", "your",
    "yours", "yourself", "yourselves",
}
# fmt: on


class Normalizer:
    def __init__(
        self, use_porter: bool = False, stopwords: str | None = None
    ) -> None:

        self.use_porter = use_porter
        self.stemmer: stem.PorterStemmer | stem.SnowballStemmer = (
            stem.PorterStemmer()
            if use_porter
            else stem.SnowballStemmer("english")
        )

        self.stopwords = stopwords
        if self.stopwords == "custom":
            self.stopwords_list = STOPWORDS
        elif self.stopwords == "sklearn":
            self.stopwords_list = _stop_words.ENGLISH_STOP_WORDS

        self.ws_re: re.Pattern[str] = re.compile(r"([A-Za-z]+'?[A-Za-z]+)")

    def __repr__(self) -> str:

        return f"{self.__class__.__name__} (use_porter={self.use_porter},
                                            stopwords={self.stopwords})"

    def __to_lower_case(self, document: str) -> str:
```

```
        return document.lower()

    def __split_document(self, document: str) -> list[str]:

        return [x.group(0) for x in self.ws_re.finditer(document)]

    def __remove_stopwords(self, tokens: list[str]) -> list[str]:
        return [word for word in tokens if word not in self.stopwords_list]

    def __stem(self, tokens: list[str]) -> list[str]:

        return [self.stemmer.stem(token) for token in tokens]

    def __call__(self, document: str) -> list[str]:

        # convert the entire document to lower-case
        doc_lc: str = self.__to_lower_case(document)

        # split the document on its whitespace
        tokens: list[str] = self.__split_document(doc_lc)

        # remove contractions and stopwords
        if self.stopwords:
            tokens = self.__remove_stopwords(tokens)

        # stem tokens
        tokens = self.__stem(tokens)

        return tokens
```

Code Listing 5: ir/model.py

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
import xgboost as xgb

from .normalize import Normalizer


class Model:
    def __init__(self) -> None:

        self.clf: GridSearchCV
        self.train_features: list[str] = []
        self.train_target: np.ndarray
        self.test_features: list[str] = []
        self.test_target: np.ndarray

        self.use_xgb_flag = False

    def use_xgb(self) -> None:

        self.use_xgb_flag = True

    def grid_search(self) -> None:
```

```
        if not self.use_xgb_flag:
            params = {
                "cv__tokenizer": [
                    None,
                    Normalizer(use_porter=False, stopwords=None),
                    Normalizer(use_porter=True, stopwords=None),
                    Normalizer(use_porter=False, stopwords="custom"),
                    Normalizer(use_porter=True, stopwords="custom"),
                    Normalizer(use_porter=False, stopwords="sklearn"),
                    Normalizer(use_porter=True, stopwords="sklearn"),
                ],
                "clf__class_weight": [{1: i} for i in range(3, 31)],
            }
        else:
            params = {
                "cv__tokenizer": [
                    None,
                    Normalizer(use_porter=False, stopwords=None),
                    Normalizer(use_porter=True, stopwords=None),
                    Normalizer(use_porter=False, stopwords="custom"),
                    Normalizer(use_porter=True, stopwords="custom"),
                    Normalizer(use_porter=False, stopwords="sklearn"),
                    Normalizer(use_porter=True, stopwords="sklearn"),
                ],
                "clf__scale_pos_weight": range(3, 31),
                "clf__max_depth": range(3, 10),
                "clf__min_child_weight": range(1, 10),
            }

        pipe = Pipeline(
            [
                ("cv", CountVectorizer(stop_words=None)),
                ("tfidf", TfidfTransformer()),
                (
                    "clf",
                    SGDClassifier(loss="hinge", random_state=0)
                    if not self.use_xgb_flag
                    else xgb.XGBClassifier(
                        objective="binary:logistic",
                        random_state=0,
                        clf__learning_rate=0.2,
                        clf__n_estimators=100,
                    ),
                ),
            ]
        )
        self.load_classifier(
            GridSearchCV(
                pipe,
                params,
                cv=2,
                n_jobs=-1,
                scoring=("f1", "recall", "precision"),
                refit="recall",
                verbose=10,
            )
        )
        self.train_classifier()
```

```
        for param_name in params.keys():
            print(f"{param_name}: {self.clf.best_params_[param_name]}")

    def set_training_features(
        self, data: list[str], target: np.ndarray
    ) -> None:

        self.train_features = data
        self.train_target = target

    def set_test_features(self, data: list[str], target: np.ndarray) -> None:

        self.test_features = data
        self.test_target = target

    def load_classifier(self, clf: GridSearchCV) -> None:

        self.clf = clf

    def get_classifier(self) -> GridSearchCV:

        return self.clf

    def train_classifier(self) -> None:

        self.clf.fit(self.train_features, self.train_target)

    def predict(self) -> np.ndarray:

        return self.clf.predict(self.test_features)
```

Code Listing 6: run.py

```
import argparse
from pathlib import Path

from ir import InformationRetrieval

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("path", type=str, help="path of corpus file")
    parser.add_argument(
        "-f",
        "--train",
        type=int,
        nargs="?",
        default=None,
        const=0,
        choices=(0, 1, 2, 3),
        help="extract training features",
    )
    parser.add_argument(
        "-l",
        "--load",
        action=argparse.BooleanOptionalAction,
        help="load classifier from disk",
    )
    parser.add_argument(
```

```
        "-d",
        "--dump",
        action=argparse.BooleanOptionalAction,
        help="dump classifier to disk",
    )
    parser.add_argument(
        "-g",
        "--gen",
        action=argparse.BooleanOptionalAction,
        help="perform grid search",
    )
    parser.add_argument(
        "-s",
        "--score",
        action=argparse.BooleanOptionalAction,
        help="compute scores of the model",
    )
    parser.add_argument(
        "-p",
        "--predict",
        action=argparse.BooleanOptionalAction,
        help="predict target values",
    )
    parser.add_argument("-t", "--test", type=str, help="path of test file")

    args = vars(parser.parse_args())

    ir_obj = InformationRetrieval(Path(args["path"]))

    categories: tuple = ()
    if args["train"]:

        if args["train"] == 0:
            categories = ("title",)
            print("training on categories:", categories)

        elif args["train"] == 1:
            categories = ("title", "abstract", "keywords")
            print("training on categories:", categories)

        elif args["train"] == 2:
            categories = (
                "title",
                "abstract",
                "keywords",
                "language",
            )
            print("training on categories:", categories)

        elif args["train"] == 3:
            categories = (
                "title",
                "abstract",
                "keywords",
                "language",
            )
            ir_obj.use_xgb()
            print("training on categories:", categories)
```

```
        ir_obj.extract_train_features(categories)

    if args["load"]:
        ir_obj.load_classifier(args["train"])

    if args["gen"]:
        ir_obj.grid_search()
        ir_obj.dump_classifier(args["train"])

    if args["test"]:
        ir_obj.extract_test_features(Path(args["test"]), categories)

    if args["score"]:
        ir_obj.predict()
        ir_obj.score()

    if args["predict"]:
        ir_obj.predict()
        ir_obj.dump_predict_vals(Path(args["test"]))
```