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

Sabbir Ahmed

October 31, 2022

Contents

1 Introduction

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

2 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.

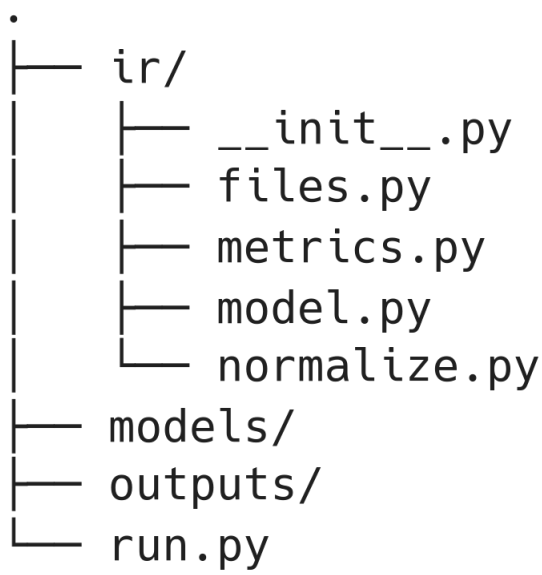

Figure 1: Directory Hierarchy of Assignment 4

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 460.

2.1 Classes

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`

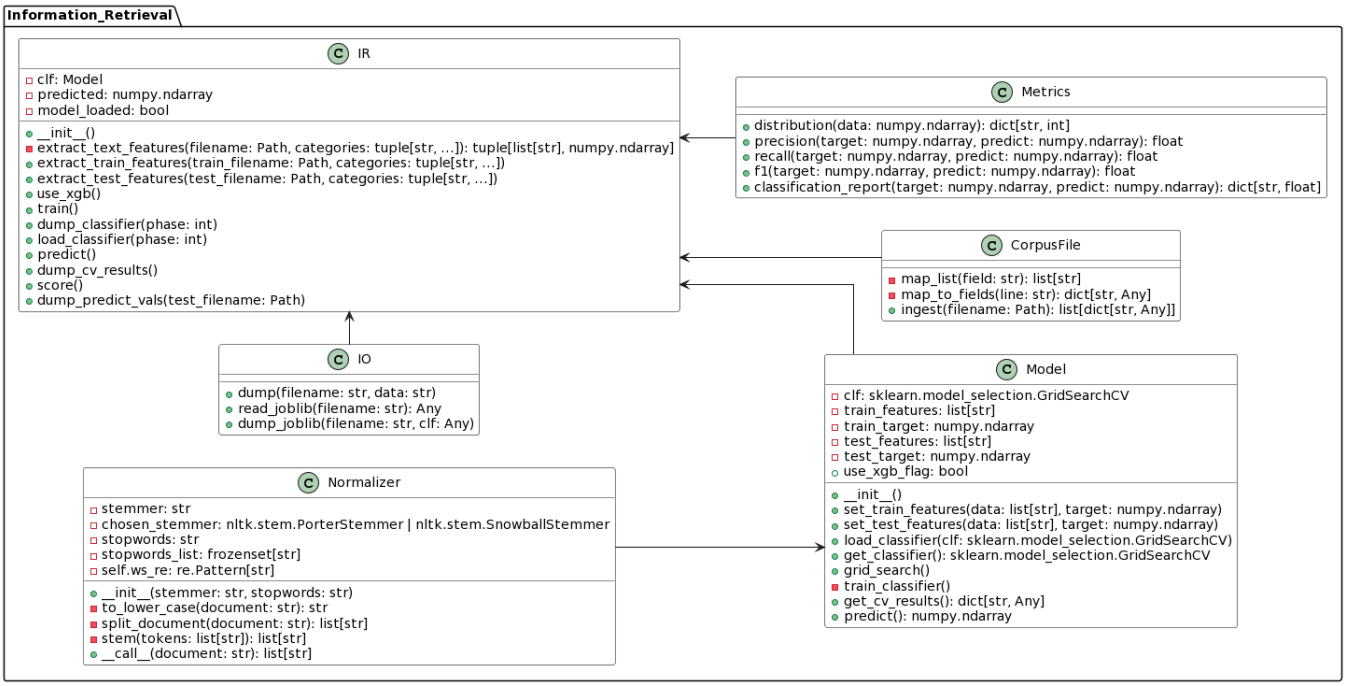

Figure 2: UML of Information Retrieval

2.2 External Libraries

The following external libraries were used to implement portions of the assignment:

- Natural Language Toolkit (NLTK) [1]
- scikit-learn (sklearn) [2]
- XGBoost [3]

3 Exploratory Analysis

The dataset is a collection of tab separated value (TSV) files with 10 features. The training portion is used to train the models, the development portion is used to score the performances of the models, and the test portion is used for its target features to be predicted by the models.

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 Table 2 showing the distribution in the training dataset:

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

- DocID
- Title
- Year
- Language
- Abstract

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
Journal	Journal title
ISSN	Numeric code for journal
Year	Publication year
Language	Trigram for language code (e.g., "eng")
Abstract	Several sentence abstract from article
Keywords	List of keywords

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

Value	Count
-1	21,000
1	700

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

- Authors
- Journal
- ISSN
- Keywords

4 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.

4.1 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`.

5 Experiments

5.1 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.

Table 3 lists the model's scores:

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

Metric	Score
Precision	0.195
Recall	0.640
F1-Score	0.299

5.2 Experiment #1: More Features

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.

Table 4 lists the model's scores:

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

Metric	Score
Precision	0.330
Recall	0.747
F1-Score	0.458

5.3 Experiment #2: Hyperparameter Optimization and Feature Selection

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

1. Tokenizer for `CountVectorizer`:
   (a) With one of the following stemmers:
      i. No stemming
      ii. Snowball stemmer
      iii. Porter stemmer

(b) Combined with the following stop words list options:

i. No stopwords removed

ii. Custom stop words list generated from previous assignments

iii. 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 F1-score.

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

Figure 3 lists the scores yielded by the various combinations of class weight ratios, tokenization methods, and stop words lists. The scores are the mean F1-scores of the 5-fold cross validation segments.

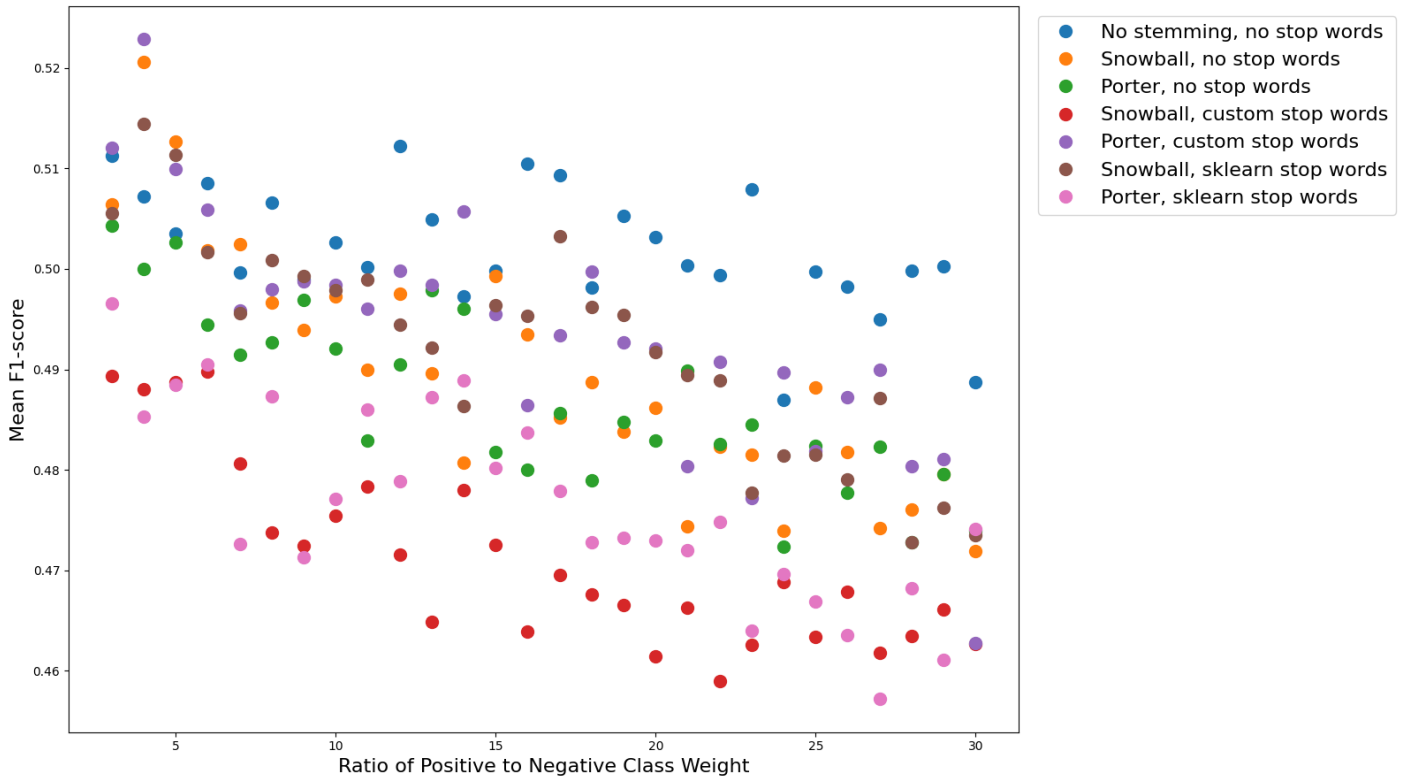

Figure 3: Mean F1-scores with Various Combinations of Class Weight Ratios, Tokenization Methods, and Stop Words Lists

Table 5 lists the parameters with the best scores.

The tokenizer, when not using any stemming or stop words lists perform the best consistently. Some models, such as the tokenizer with either of the stemmers and no stop words lists, appear to perform well with low weight classes, but degrade over increasing weights.

Table 5: Optimal parameters determined via grid search

Parameter	Value
Stemmer for `CountVectorizer`	Snowball
Stop words list	None
Class weight ratio for `SGDClassifier`	{1: 4}

Table 6 shows the optimal model's scores.

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

Metric	Score
Precision	0.503
Recall	0.587
F1-Score	0.542

5.3.1 Analysis

The exhaustive grid search yielded an optimal model with an F1-score of 0.542. When hyperparameter tuning, all of the 3 metrics were initially considered (precision, recall, and F1-score) but it ranked models with inconsistent performances on the development dataset, i.e models with recall of >0.8 yielded precision of <0.2. The models had to be narrowed down to just the F1-scores.

5.4 Experiment #3: eXtreme Gradient Boosting

An additional experiment was performed, where the same features were trained on an XGBoost (eXtreme Gradient Boosting) estimator [3]. A similar approach to hyperparameter tuning and feature selection with the SVM model was taken with this estimator, although not as thoroughly due to memory and time constraints.

Table 7: Optimal parameters determined via grid search

Metric	Score
objective	binary:logistic
scale_pos_weight	3
learning_rate	0.2
n_estimators	100
max_depth	3
min_child_weight	10
max_delta_step	3
subsample	0.8

Table 8 shows the optimal model's scores.

5.4.1 Analysis

The scores yielded did not appear to be significantly different from the SVM model, although recall can be prioritized by adjusting properties of the tree structures such as `max_depth` and `min_child_weight`.

Table 8: Scores using *Title*, *Abstract*, *Keywords* and *Language* as the features

Metric	Score
Precision	0.631
Recall	0.433
F1-Score	0.514

6 Conclusion

The SVM model with the optimal parameters appeared to perform the best. Ultimately, this model was used to predict the target values of the test portion of the dataset. The predictions were saved to the attached file, `sahmed80.txt`.

References

[1] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] T. Chen and C. Guestrin, "XGBoost," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, aug 2016.

A Source Code

Code Listing 1: ir/__init__.py

```
from pathlib import Path

import numpy as np

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

JHED = "sahmed80"


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

        self.clf: Model = Model()

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

    def use_xgb(self) -> None:

        self.clf.use_xgb_flag = True
```

```
    def __extract_text_features(
        self, filename: Path, categories: tuple[str, ...]
    ) -> tuple[list[str], np.ndarray]:

        docs = CorpusFile().ingest(filename)
        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, train_filename: Path, categories: tuple[str, ...]
    ) -> None:

        features, target = self.__extract_text_features(
            train_filename, categories
        )
        self.clf.set_train_features(features, target)
        print("distribution of train targets:", Metrics.distribution(target))

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

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

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

    def dump_cv_results(self) -> None:

        cv_results = self.clf.get_cv_results()
        content_str = "weight,tokenization,score\n"
        for param, score in zip(
            cv_results["params"], cv_results["mean_test_score"]
        ):
            content_str += f"{param['clf__class_weight'][1]},{param['cv__tokenizer
                                                    ']},{score}\n"

        IO.dump("outputs/cross_validations.csv", content_str)

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

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

Code Listing 2: ir/files.py

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

TARGET_FIELD = "assessment"

FEATURE_FIELDS = (
    "docid",
    "title",
    "authors",
    "journal",
    "issn",
    "year",
    "language",
    "abstract",
    "keywords",
)

LIST_FEATURE_FIELDS = (
    "authors",
    "journal",
    "issn",
    "keywords",
)
```

```
class CorpusFile:

    field_delim = "\t"
    list_delim = ";"

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

        return field.split(CorpusFile.list_delim)

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

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

        for field, value in mapped_field.items():
            if field == TARGET_FIELD:
                mapped_field[field] = int(value)
            elif field in LIST_FEATURE_FIELDS:
                mapped_field[field] = CorpusFile.__map_list(value)

        return mapped_field

    @staticmethod
    def ingest(filename: Path) -> list[dict[str, Any]]:

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

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

        return docs


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

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

    @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

import nltk
from sklearn.feature_extraction import _stop_words

# fmt: off
STOPWORDS: frozenset[str] = frozenset({
    # 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, stemmer: str = "snowball", stopwords: str | None = None
    ) -> None:

        self.stemmer: str = stemmer
        self.chosen_stemmer: nltk.stem.PorterStemmer | nltk.stem.SnowballStemmer
        if self.stemmer == "snowball":
            self.chosen_stemmer = nltk.stem.SnowballStemmer("english")
        elif self.stemmer == "porter":
            self.chosen_stemmer = nltk.stem.PorterStemmer()

        self.stopwords: str = stopwords
        self.stopwords_list: frozenset[str]
```

```
        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__}/{self.stemmer}/{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.chosen_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

```
from typing import Any

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
from sklearn.preprocessing import LabelEncoder
from xgboost import XGBClassifier

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 set_train_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 grid_search(self) -> None:

        params = {}
        clf = None

        if not self.use_xgb_flag:
            params = {
                "cv__tokenizer": [
                    None,
                    Normalizer(stemmer="snowball", stopwords=None),
                    Normalizer(stemmer="porter", stopwords=None),
                    Normalizer(stemmer="snowball", stopwords="custom"),
                    Normalizer(stemmer="porter", stopwords="custom"),
                    Normalizer(stemmer="snowball", stopwords="sklearn"),
                    Normalizer(stemmer="porter", stopwords="sklearn"),
                ],
                "clf__class_weight": [{1: i} for i in range(3, 31)],
            }
            clf = SGDClassifier(loss="hinge", random_state=0)

        else:
            params = {
                "cv__tokenizer": [
                    None,
                    Normalizer(stemmer="snowball", stopwords=None),
                    Normalizer(stemmer="porter", stopwords=None),
                    Normalizer(stemmer="snowball", stopwords="custom"),
                    Normalizer(stemmer="porter", stopwords="custom"),
                    Normalizer(stemmer="snowball", stopwords="sklearn"),
                    Normalizer(stemmer="porter", stopwords="sklearn"),
                ],
```

```
                "clf__scale_pos_weight": range(3, 31),
                "clf__max_depth": range(3, 10),
                "clf__min_child_weight": range(1, 10),
            }
            clf = XGBClassifier(
                objective="binary:logistic",
                random_state=0,
                learning_rate=0.2,
                n_estimators=100,
            )

        pipe = Pipeline(
            [
                ("cv", CountVectorizer(stop_words=None)),
                ("tfidf", TfidfTransformer()),
                ("clf", clf),
            ]
        )
        self.load_classifier(
            GridSearchCV(
                pipe,
                params,
                n_jobs=-1,
                scoring="f1",
                verbose=10,
            )
        )
        self.__train_classifier()

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

    def __train_classifier(self) -> None:

        if self.use_xgb_flag:
            le = LabelEncoder()
            self.train_target = le.fit_transform(self.train_target)

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

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

        return self.clf.cv_results_

    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(
        "-c",
        "--cv",
        action=argparse.BooleanOptionalAction,
        help="dump cross validation results of grid search",
    )
    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()

    categories: tuple[str, ...] = ()
    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(Path(args["path"]), categories)

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

    if args["gen"]:
        ir_obj.train()
        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["cv"]:
        ir_obj.dump_cv_results()

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