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

Sabbir Ahmed

October 24, 2022

Contents

Introduction

Baseline

A Source Code

Code Listing 1: ir/files.py

```
from pathlib import Path
import joblib

from .const import TARGET_FIELD, FEATURE_FIELDS, FIELD_DELIM, LIST_DELIM, JHED
from .lexer import LexerStatistics
from .normalize import Normalizer
from .types import Any, iterable, text_io, generator


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

        self.prep = Normalizer()
        self.filename: Path = filename
        self.num_docs: int = 0

    @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"])

        # 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]))
                    self.num_docs += 1

        return docs


class IO:
    @staticmethod
    def open_file(filename: str) -> text_io:

        return open(f"{filename}.txt")

    @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'")


class Formatter:

    hr: str = "-------------------------------\n"
    table_header: str = f"{'Word':<12} | {'TF':<6} | {'DF':<6}\n{hr}"

    @staticmethod
    def __format_tf_df(term: str, tf: int, df: int) -> str:

        return f"{term:<12} | {tf:<6} | {df:<6}\n"

    @staticmethod
    def format_stats(lex_stats: LexerStatistics, num_docs: int = 0) -> str:

        contents: str = ""

        return contents

    @staticmethod
    def format_term_doc_tf(term_doc_tf: list[tuple[str, int, int]]) -> str:

        contents: str = ""
```

```
        for line in term_doc_tf:
            contents += " ".join(str(i) for i in line) + "\n"

        return contents

    @staticmethod
    def format_rankings(
        all_rankings: dict[int, list[tuple[int, float]]]
    ) -> str:

        contents: str = ""
        for query_id, rankings in all_rankings.items():
            contents += (
                "\n".join(
                    f"{query_id} Q0 {doc_id} {rank + 1} {score:.6f} {JHED}"
                    for rank, (doc_id, score) in enumerate(rankings)
                )
                + "\n"
            )

        return contents
```

Code Listing 2: ir/metrics.py

```
import numpy as np
import scipy
from sklearn.metrics import accuracy_score, classification_report


class Metrics:
    def __init__(self) -> None:
        pass

    @staticmethod
    def describe(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 {
            "recall": Metrics.recall(target, predict),
            "precision": Metrics.precision(target, predict),
            "f1": Metrics.f1(target, predict),
        }
```

Code Listing 3: ir/lexer.py

```
from collections import Counter

from .types import generator, counter


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

        self.cf: counter = Counter()
        self.df: counter = Counter()
        self.tf: counter = Counter()
        self.dictionary: dict[str, list[int]] = {}

    def add(self, tokens: generator[str]) -> None:

        # create a Counter for the document
        self.tf.clear()
        self.tf.update(tokens)

        # update the total term-frequency values with the Counter
        self.cf.update(self.tf)

        # increment the document-frequency values
        self.df.update(self.tf.keys())

    def get_term_docid_tf(self, doc_id: int) -> generator[tuple[str, int, int]]:

        for term in self.tf:
            yield term, doc_id, self.tf[term]
```

```
    def set_cf(self, cf: counter) -> None:

        self.cf = cf

    def set_df(self, df: counter) -> None:

        self.df = df

    def get_cf(self) -> counter:

        return self.cf

    def get_df(self) -> counter:

        return self.df


class LexerStatistics:
    def __init__(self, lex: Lexer) -> None:
        self.cf = lex.cf
        self.df = lex.df

    def get_collection_size(self) -> int:

        return self.cf.total()

    def get_vocab_size(self) -> int:

        return len(self.cf)

    def get_top_n_cf_df(self, n: int) -> generator[tuple[str, int, int]]:

        top_n_cf = self.cf.most_common(n)
        for cf in top_n_cf:
            term, freq = cf
            yield term, freq, self.df[term]

    def get_nth_freq_term(self, n: int) -> tuple[str, int, int]:

        term, freq = self.cf.most_common(n)[-1]
        return term, freq, self.df[term]

    def get_single_occs(self) -> int:

        single_occs: int = 0
        for df in self.df.values():
            if df == 1:
                single_occs += 1

        return single_occs
```

Code Listing 4: ir/normalize.py

```
import re

from nltk import stem

# 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, no_stopwords=False) -> None:

        self.document: str = ""
        self.tokens: list[str]

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

    def __repr__(self) -> str:

        return f"{self.__class__.__name__} (no_stopwords={self.no_stopwords})"

    def set_document(self, document: str) -> None:

        self.document = document

    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 STOPWORDS]

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

        return [self.snow.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
        self.tokens = self.__split_document(doc_lc)

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

        # stem tokens
        self.tokens = self.__stem(self.tokens)

        return self.tokens
```

Code Listing 5: ir/vectorizer.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

from .normalize import Normalizer, STOPWORDS


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

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

    def grid_search(self):

        parameters = {
            # "cv__tokenizer": [None, Normalizer()],
            # "cv__stop_words": [None, "english", STOPWORDS],
            # "clf__class_weight": [{1: i} for i in range(3, 31)],
            "cv__tokenizer": [None],
            "cv__stop_words": [None],
            "clf__class_weight": [{1: 3}],
        }

        pipe = Pipeline(
            [
                ("cv", CountVectorizer(analyzer="char")),
                ("tfidf", TfidfTransformer()),
```

```
                (
                    "clf",
                    SGDClassifier(random_state=0),
                ),
            ]
        )
        self.load_classifier(GridSearchCV(pipe, parameters, n_jobs=-1))
        self.train_classifier()

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

    def set_training_features(self, data: list[str], target: np.ndarray):

        self.train_features = data
        self.train_target = target

    def set_test_features(self, data: list[str], target: np.ndarray):

        self.test_features = data
        self.test_target = target

    def load_classifier(self, clf) -> None:

        self.clf = clf

    def get_classifier(self):

        return self.clf

    def train_classifier(self):

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

    def predict(self):

        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(
        "-1",
        "--baseline",
        action=argparse.BooleanOptionalAction,
        help="extract training features",
    )
    parser.add_argument(
        "-f",
        "--train",
        action=argparse.BooleanOptionalAction,
```

```
        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(
        "-cv",
        "--cross_validate",
        action=argparse.BooleanOptionalAction,
        help="perform grid search",
    )
    parser.add_argument(
        "-s",
        "--score",
        action=argparse.BooleanOptionalAction,
        help="perform grid search",
    )
    parser.add_argument("-t", "--test", type=str, help="path of test file")

    args = vars(parser.parse_args())

    categories: tuple = ()
    phase_name: str = ""
    if args["baseline"]:
        categories = ("title",)
        print("training on categories:", categories)
        phase_name = "1"
    else:
        categories = ("title", "abstract", "keywords")
        print("training on categories:", categories)
        phase_name = "2"

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

    if args["train"]:
        ir_obj.extract_train_features(categories)

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

    if args["cross_validate"]:
        ir_obj.grid_search()
        ir_obj.dump_classifier(phase_name)

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

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

B Outputs