بازیابی پیشرفته اطلاعات

مدرس: دکتر بیگی

شماره گروه: ۵

دانشکده مهندسی کامپیوتر

تهیه‌کنندگان: نیما جمالی – سپهر فعلی – سینا کاظمی

گزارش فاز دوم پروژه

فهرست مطالب

ساختن فضای برداری برای دسته‌بندها ۳

پیش پردازش و تنظیم فیلدهای اولیه ۳

پیاده‌سازی فضای برداری tf-idf به روش ntn ۴

ذخیره‌سازی فضای برداری ۵

پیاده‌سازی دسته‌بندها ۶

Naïve Bayes ۶

k-NN ۸

SVM ۹

Random Forest ۱۰

محاسبه‌ی Validation Set ۱۱

محاسبه‌ی بهترین پارامتر برای الگوریتم k-NN ۱۲

محاسبه‌ی بهترین پارامتر برای الگوریتم SVM ۱۳

بهبود سیستم بازیابی فاز اول پروژه ۱۴

تابع اجماع نظر دسته‌بندها برای تعیین برچسب هر مستند ۱۵

جستجو بر اساس دسته ۱۵

ارزیابی نهایی ۱۷

پیاده‌سازی تابع find_metric ۱۷

معیارهای ارزیابی برای Naïve Bayes ۱۷

معیارهای ارزیابی برای k-NN ۱۸

معیارهای ارزیابی برای SVM ۱۸

معیارهای ارزیابی برای Random Forest ۱۸

نتیجه‌گیری نهایی معیارهای ارزیابی ۱۹

نحوه‌ی تقسیم وظایف ۲۰
نیما جمالی ۲۰
سپهر فعلی ۲۰
سینا کاظمی ۲۰
مراجع ۲۱

ساختن فضای برداری برای دسته‌بندها

در این بخش که در واقع بخش آغازین کار است، ابتدا کلاس جدیدی به نام Classifier تعریف می‌کنیم و constructor آن را تکمیل می‌کنیم. همچنین به کلاس IRSystem که در فاز اول پروژه طراحی شده بود، متدی برای محاسبه‌ی فضای برداری اسناد به روش tf-idf اضافه می‌کنیم. در ادامه به شرح پیاده‌سازی این بخش می‌پردازیم.

پیش پردازش و تنظیم فیلدهای اولیه

کلاس Classifier یک path را به عنوان ورودی دریافت می‌کند. این path مشخص می‌کند که دسته‌بندی قرار است بر روی چه فایلی اعمال شود. سپس از کلاس IRSystem برای پیش‌پردازش مستندات موجود در train.csv استفاده می‌کنیم. تابع call_prepare و call_create_positional را برای این مجموعه مستندات صدا می‌زنیم تا نمایه‌ی positional ساخته شود. از آنجا که می‌خواهیم مستندات موجود در path را هم در این مجموعه درج کنیم، سایز مجموعه‌ی آموزش را در متغیری به نام train_size نگه می‌داریم. سپس مستندات موجود در path را درج می‌کنیم. y_test را هم که برای محاسبه‌ی معیارهای ارزیابی استفاده می‌شود، تنها در صورتی تعریف می‌کنیم که path همان test.csv باشد. در این صورت y_test برابر با لیستی خواهد بود که مقادیر آن، به ترتیب برابر با مقادیر ستون views در داده‌ی آزمون است.

همچنین دو متغیر train_vector_space و vector_space هر دو فضای برداری هستند که مورد اول برای محاسبه‌ی دسته‌بندهای SVM و Random Forest ایجاد شده و مورد دوم برای محاسبه‌ی دسته‌بند k-NN استفاده شده‌است که در ادامه آنها را نیز توضیح می‌دهیم.

```python
class Classifier:
    def __init__(self, path):
        self.train_ir_sys = IRSystem(["description", "title"], "data/train.csv", None)
        self.train_ir_sys.call_prepare("english", False)
        self.train_ir_sys.call_create_positional("english")
        self.train_size = len(self.train_ir_sys.structured_documents["english"])
        self.train_ir_sys.csv_insert(path, "english")
        self.train_vector_space = self.create_vector_matrix(self.train_ir_sys, "english")
        self.y_train = self.csv_views("data/train.csv")

        self.vector_space = self.train_ir_sys.use_ntn("english")

        self.y_test = None
        self.knn_classifier = None
        self.svm_classifier = None
        self.naive_bayes_classifier = None
        self.random_forrest_classifier = None
        if path == "data/test.csv":
            self.y_test = self.csv_views("data/test.csv")

    def csv_views(self, path):
        df = pd.read_csv(path, usecols=["views"])
        result = []
        for i in range(len(df)):
            result += [df.iloc[i]["views"]]
        return result
```

تصویر ۱: طراحی اولیه کلاس Classifier و تنظیم فیلدهای آن

پیاده‌سازی فضای برداری tf-idf به روش ntn

در این بخش، ابتدا در کلاس IRSystem متدی با نام ntn طراحی کردیم که یک doc_id و زبان آن را ورودی می‌گیرد. سپس یک دیکشنری به نام result می‌سازد که کلیدهای آن، ترم‌های آن مستند و مقادیر آن، حاصل عبارت $tf_{term,doc} \times idf_{term}$ برای هر کلید است. این تابع توسط تابعی به نام use_ntn صدا زده می‌شود. این تابع به ازای تمام مستندات، دیکشنری گفته شده را محاسبه کرده و بر می‌گرداند.

```
def ntn(self, doc_id, lang):
    doc = []
    for part in self.structured_documents[lang][doc_id]:
        for word in part:
            doc += [word]
    doc_dict = Counter(doc)
    result = dict()
    for term in doc_dict.keys():
        if term not in result.keys():
            p = self.positional_index[lang][term]
            df = len(p.keys()) - 1
            idf = math.log10(len(self.structured_documents[lang]) / df)
            result[term] = doc_dict[term] * idf
    return result

def use_ntn(self, lang):
    vectors = []
    for doc_id in range(len(self.structured_documents[lang])):
        vectors += [self.ntn(doc_id, lang)]
```

تصویر ۲: طراحی توابع ntn و use_ntn

برای طراحی فضای برداری که الگوریتم‌های کتاب‌خانه‌های آماده برای SVM و Random Forest بتوانند آن را پیاده کنند، نیاز به ماتریسی از ndarrayها داریم و نمی‌توان از دیکشنری استفاده کرد. به همین دلیل نیاز داریم برای هر ترم مستقل در کل دیکشنری (train + test) یک عدد داشته باشیم که به صورت یکتا بین آنها رابطه وجود داشته باشد. برای این کار تابع token_to_number را طراحی می‌کنیم که به هر token عدد خاصی نسبت می‌دهد. این عدد خاص از صفر تا len(terms) – 1 می‌تواند باشد.

سپس برای ساختن ماتریسی از ndarrayها، تابع create_vector_space را صدا می‌زنیم که ابتدا ماتریسی از صفر تولید می‌کند که تعداد سطرهای آن برابر با تعداد مستندات و تعداد ستون‌های آن برابر با تعداد کل ترم‌هاست. در نهایت با صدا زدن use_ntn برای هر مستند، برای هر ترم در آن مستند از تابع token_to_number استفاده می‌کنیم

و وزن مربوط به آن ترم در مستند را در ماتریس به‌روز رسانی می‌کنیم. در نهایت این ماتریس در فیلدی به نام train_vector_space ذخیره می‌شود.

```
def token_to_number(self, ir_sys, lang):
    return {token: ind for ind, token in enumerate(ir_sys.positional_index[lang].keys())}

def create_vector_matrix(self, ir_sys, lang):
    vector = np.zeros([len(ir_sys.structured_documents[lang]), len(ir_sys.positional_index[lang].keys())])
    tokenized_vector = ir_sys.use_ntn(lang)
    for doc_id in range(len(tokenized_vector)):
        for term in tokenized_vector[doc_id].keys():
            vector[doc_id][self.token_to_number(ir_sys, lang)[term]] = tokenized_vector[doc_id][term]
    return vector
```

تصویر ۳: طراحی توابع مورد نیاز برای ساخت فضای برداری به روش ماتریسی

ذخیره‌سازی فضای برداری

فضای برداری ایجاد شده در قسمت قبل بسیار بزرگ است، زیرا تعداد ترم‌های دیکشنری چیزی در حدود ۱۲۰۰۰ است! لذا هر نمونه از Classifier که ساخته می‌شود، زمان زیادی می‌گیرد. برای همین تصمیم گرفتیم دو نمونه‌ای که از کلاس Classifier می‌سازیم (یک نمونه برای test.csv و یک نمونه برای ted_talks.csv) را ذخیره کنیم، البته این ذخیره‌سازی بعد از تکمیل سایر متدهای کلاس Classifier انجام می‌گیرد. با دستور create می‌توانیم یک نمونه ایجاد کنیم و با دستور save، آن نمونه را ذخیره کنیم. نمونه‌ی این دستورات را می‌توان در این تست مشاهده کرد.

پیاده‌سازی دسته‌بندها

در این بخش توابع دسته‌بندی را طراحی می‌کنیم. همچنین توابع مورد نیاز برای محاسبه‌ی بهترین پارامتر را طراحی می‌کنیم. در ادامه به شرح پیاده‌سازی این توابع می‌پردازیم. دقت شود که از کل مجموعه داده‌ی train.csv برای آموزش دسته‌بندها استفاده شده‌است.

Naïve Bayes

یکی از راه های پیش‌بینی کردن گروه و دسته یک شی(در این پروژه، همان مستند)، الگوریتم Naive-Bayes است. این الگوریتم، تعداد نسبتا زیادی داده تحت عنوان داده‌ی آموزش دریافت می‌کند که گروه هریک از این داده‌ها مشخص است و با پردازش آن، به نوعی ویژگی‌های هر گروه معلوم می‌شود. بدین‌شکل که اگر x رخ داده باشد، احتمال این که به گروه A مرتبط باشد بیشتر است یا گروه B. برای پیاده‌سازی این الگوریتم از سه تابع بهره بردیم:

- تابع naive_bayes_train(self, flag_counter, words, lang): این تابع، قسمت آموزش الگوریتم را انجام می‌دهد و درواقع پایه و اساس کارکرد الگوریتم را شامل می‌شود. اما نحوه کار:

  متغیر flag_counter یک دیکشنری ۴عضوی است، متشکل از مقادیر زیر:

  - تعداد مستنداتی که عضو گروه ۱ هستند.(positive_docs)
  - تعداد مستنداتی که عضو گروه ۱- هستند.(negative_docs)
  - تعداد ترم های غیریکتا در مستندات گروه ۱ (positive_terms)
  - تعداد ترم های غیریکتا در مستندات گروه ۱- (negative_terms)

  متغیر words یک دیکشنری بزرگ از ترم‌های موجود در مجموعه‌ی مستندات داده‌های آموزش است. هر عضو آن، خود یک دیکشنری ۲عضوی است که از مقادیر زیر تشکیل شده‌است.

  - تعداد تکرار کلمه موردنظر در مستندات گروه ۱ (words[term]["positive"])
  - تعداد تکرار کلمه موردنظر در مستندات گروه ۱- (words[term]["negative"])

  این دو متغیر، در این تابع و با پردازش داده‌های آموزش که قبلا پیش‌پردازش شده‌اند و در متغیری از کلاس ir_system به نام structured_documents[lang] ذخیره شده‌اند، مقداردهی و تکمیل می‌شوند تا در ادامه مورد استفاده قرار گیرند.

- تابع naive_bayes_test(self, flag_counter, words, lang): این تابع، قسمت آزمایش الگوریتم را انجام می‌دهد و درواقع، خروجی الگوریتم است. یعنی با استفاده از نتایج برآمده از پردازش داده‌های آموزش در تابع naive_bayes_train، گروه‌های داده‌های آزمایش را پیش‌بینی می‌کند. اما نحوه‌ی کار چگونه است:

به ازای هر مستند ورودی، به روش زیر می‌توانیم گروه مربوط به آن را حدس بزنیم.

$$C_{MAP} = \underset{c \in C}{\operatorname{argmax}} \left[Log\ P(c) + \sum_{1 \le k \le n_d} Log\ P(t_k|c) \right]$$

$$P(t|c) = \frac{T_{ct} + 1}{(\sum_{t' \in V} T_{ct'}) + B}, \qquad P(c) = \frac{len(C = c)}{len(C = c) + Len(C\ != c)}$$

پیاده‌سازی این روابط در کد به این صورت است که

- برای محاسبه T_{ct} به words[t][c] رجوع می‌کنیم.
- برای محاسبه $\sum_{t' \in V} T_{ct'}$ به flag_counter[c_terms] رجوع می‌کنیم.
- مقدار B برابر با تعداد ترم‌های یکتای داده‌های آزمون است.
- برای محاسبه $len(C = c)$ نیز به flag_counter[c_docs] مراجعه می‌کنیم.

در نهایت، نتیجه حدس‌های خود را در لیستی به سایز داده‌های آزمایش به نام y_predicted می‌ریزیم که این متغیر، خروجی تابع خواهد بود.

- تابع naive_bayes(self, lang): این تابع فرآیند اجرای الگوریتم را کنترل می‌کند. یعنی دیکشنری‌های flag_counter و words را initialize می‌کند، سپس ابتدا تابع آموزش یعنی naive_bayes_train و سپس تابع آزمایش یعنی naive_bayes_test را صدا می‌کند و خروجی، یعنی y_predicted را برمی‌گرداند.

```python
def naive_bayes(self, lang):
    flag_counter = {"positive_docs": 0, "negative_docs": 0, "positive_terms": 0, "negative_terms": 0}
    words = dict()
    self.naive_bayes_train(flag_counter, words, lang)
    y_predicted = self.naive_bayes_test(flag_counter, words, lang)
    return y_predicted

def naive_bayes_train(self, flag_counter, words, lang):
    for docID in range(self.train_size):

        if self.y_train[docID] == 1:
            flag = "positive"
            flag_counter["positive_docs"] += 1
        else:
            flag = "negative"
            flag_counter["negative_docs"] += 1

        for col in range(2):
            for word in self.train_ir_sys.structured_documents[lang][docID][col]:
                if word not in words.keys():
                    words[word] = {"positive": 0, "negative": 0}
                words[word][flag] += 1
                flag_counter[str(flag + "_terms")] += 1
def naive_bayes_test(self, flag_counter, words, lang):
    y_predicted = []
    p_positive_doc = flag_counter["positive_docs"] / self.train_size
    p_negative_doc = 1 - p_positive_doc
    for docID in range(len(self.train_ir_sys.structured_documents[lang]) - self.train_size):
        p_positive = 0
        p_negative = 0
        for col in range(2):
            for word in self.train_ir_sys.structured_documents[lang][self.train_size + docID][col]:
                if word in words.keys():
                    p_positive += math.log(
                        (words[word]["positive"] + 1) / (flag_counter["positive_terms"] + len(words)))
                    p_negative += math.log(
                        (words[word]["negative"] + 1) / (flag_counter["negative_terms"] + len(words)))
                else:  # new word in test Doc
                    p_positive += math.log(1 / (flag_counter["positive_terms"] + len(words)))
                    p_negative += math.log(1 / (flag_counter["negative_terms"] + len(words)))
        if math.log(p_positive_doc) + p_positive >= math.log(p_negative_doc) + p_negative:
            y_predicted.append(1)
        else:
            y_predicted.append(-1)
    return y_predicted
```

تصویر ۴: توابع مورد نیاز برای دسته‌بند Naïve Bayes

در کنسول می‌توان با دستور naive_bayes test، برچسب‌هایی که این دسته‌بند برای مجموعه‌ی تست می‌زند مشاهده کرد.

k-NN

دسته‌بند k-NN بسته به پارامتر k ورودی، k نزدیک‌ترین همسایه‌ی مستند داده شده را در مجموعه‌ی آموزش می‌یابد و با توجه به اکثریت برچسب‌ها در میان آن k مستند، برچسب مستند آزمایشی را محاسبه می‌کند.

به این منظور تابعی به نام knn طراحی می‌کنیم که مجموعه‌ی مستندات آموزش و برچسب‌های آنها، مجموعه‌ی مستندات آزمایش و پارامتر k را ورودی می‌گیرد. می‌توانستیم از فضای برداری که در بخش پیاده‌سازی فضای برداری tf-idf به روش ntn توضیح داده شد، استفاده کنیم، اما دسته‌بندی با آن روش به علت این که صفرهای زیادی را بدون آن که در محاسبه نیاز شوند، نگه می‌داریم و در واقع ماتریس sparse است، زمان زیادی می‌گیرد.

به همین علت فضای برداری جدیدی با نام vector_space (یا به عبارت بهتر knn_vector_space) ایجاد می‌کنیم که برای محاسبه‌ی آن، تابع use_ntn را صدا می‌زنیم. سپس باید فاصله‌ی دو مستند را حساب کنیم. برای این کار تابعی به نام two_doc_distance تعریف می‌کنیم که دو مستند را از فضای برداری vector_space می‌گیرد و فاصله‌ی آن‌ها را محاسبه می‌کند. برای محاسبه‌ی k نزدیک‌ترین همسایه، باید فاصله‌ی هر مستند آزمایش با هر مستند آموزشی را داشته باشیم که این کار از طریق تابع documents_distances انجام می‌شود و خروجی آن ماتریسی است که سطرهای آن، مستندات آموزش و ستون‌های آن، مستندات آزمایش است.

```python
def two_doc_distance(self, first, second, doc1, doc2):
    list1 = first[doc1].keys()
    list2 = second[doc2].keys()
    intersect = set(list1) & set(list2)
    delta = set(list1) ^ set(list2)
    result = 0
    for term in intersect:
        result += ((first[doc1][term] - second[doc2][term]) ** 2)
    for term in delta:
        if term in first[doc1].keys():
            result += ((first[doc1][term]) ** 2)
        else:
            result += ((second[doc2][term]) ** 2)
    return math.sqrt(result)

def documents_distances(self, first, second):
    result = []
    for i in range(len(first)):
        result += [[]]
        for j in range(len(second)):
            result[i] += [self.two_doc_distance(first, second, i, j)]
    return result
```

تصویر ۵: توابع محاسبه‌کننده‌ی فاصله‌ی مستندات

در تابع knn، ابتدا این ماتریس محاسبه می‌شود. سپس هر ستون را که متناظر یک مجموعه‌ی تست است، به صورت صعودی مرتب کرده و k تای اول را برای هر ستون انتخاب می‌کنیم. سپس برچسب هر یک از این k مستند را بررسی می‌کنیم. اگر تعداد ۱ ها بیشتر باشد، برچسب مستند آزمایش هم ۱ و در غیر این صورت ۱- خواهد بود.

```
def knn(self, x_train, y_train, x_test, k):
    dists = np.array(self.documents_distances(x_train, x_test))
    y_pred = []
    for doc_id in range(len(x_test)):
        test_dist = dists[:, doc_id]
        sorted_nearest_docs = [s[0] for s in sorted(enumerate(test_dist), key=lambda a: a[1])]
        top_k = sorted_nearest_docs[:k]
        pos_views, neg_views = 0, 0
        for train_id in top_k:
            if y_train[train_id] == 1:
                pos_views += 1
            else:
                neg_views += 1
        if pos_views > neg_views:
            y_pred += [1]
        else:
            y_pred += [-1]
    return y_pred
```

تصویر ۶: تابع محاسبه‌ی دسته‌بند k-NN

در کنسول هم می‌توان با دستور knn test k برچسب‌هایی که دسته‌بند k-NN به داده‌های تست می‌زند، مشاهده کرد. دقت شود که k یک عدد طبیعی است که کاربر ورودی می‌دهد.

SVM

بردار های پشتیبان به زبان ساده، مجموعه‌ای از نقاط در فضای n بعدی داده‌ها هستند که مرز دسته‌ها را مشخص می‌کنند و دسته‌بندی داده‌ها بر اساس آنها انجام می‌شود و با جابه‌جایی یکی از آنها خروجی دسته بندی ممکن است تغییر کند. در این قسمت از کتابخانه‌ی یادگیری ماشین پایتون به نام sklearn استفاده کردیم که تمام کرنل‌ها و توابع نگاشت را به صورت آماده دارد. ۳ تابع SVC , NuSVC , LinearSVC وظیفه اصلی دسته‌بندی را در این کتابخانه دارند که ما از SVC که مخفف Support Vector Classifier است استفاده می‌کنیم.

ورودی‌های این تابع کرنل مورد استفاده و پارامتر C است. کرنل را گاوسی (rbf) در نظر گرفتیم و پارامتر C را از ورودی می‌گیریم. با استفاده از کرنل گاوسی، داده‌ها را به فضای بی‌نهایت بعدی نگاشت می‌کنیم و در آن فضا دسته‌بندی را انجام می‌دهیم. پارامتر C نیز در واقع میزان اهمیت به اشتباه دسته‌بندی شدن داده‌ها و فاصله‌ی خط جداساز از Support Vectorها را مشخص می‌کند.

در کنسول می‌توان با وارد کردن دستور svm test c که c پارامتری است که کاربر ورودی می‌دهد، برچسب‌های خروجی دسته‌بند SVM را برای مجموعه‌ی تست مشاهده کرد.

```
def svm(self, x_train, y_train, x_test, c_parameter):
    model = SVC(kernel='rbf', C=c_parameter)
    model.fit(x_train, y_train)
    self.svm_classifier = model
    y_pred = model.predict(x_test)
    return y_pred
```

تصویر ۷: دسته‌بند SVM

Random Forest

الگوریتم جنگل تصادفی یا همان Random Forest یک الگوریتم ترکیبی (ensemble) است که از درخت‌های تصمیم بهره می‌گیرد.

الگوریتم درخت تصمیم (decision trees) می‌تواند به راحتی عملیات طبقه‌بندی را بر روی داده‌ها انجام دهد. حال در الگوریتم جنگل تصادفی از چندین درخت تصمیم (برای مثال ۱۰۰ درخت تصمیم) استفاده می‌شود. در واقع مجموعه‌ای از درخت‌های تصمیم با هم یک جنگل را تولید می‌کنند و این جنگل می‌تواند تصمیم‌های بهتری را نسبت به یک درخت اتخاذ نماید.

در الگوریتم جنگل تصادفی به هر کدام از درخت‌ها، زیرمجموعه‌ای از داده‌ها تزریق می‌شود. برای مثال اگر مجموعه داده دارای ۲۵۵۰ سطر باشد و ۱۰ ستون داشته باشد، الگوریتم جنگل تصادفی به هر کدام از درخت ها ۱۰۰ سطر و ۵ ستون –که به طور تصادفی انتخاب شده‌اند و زیرمجموعه‌ای از مجموعه‌ی داده‌ها هست– می‌دهد. این درخت‌ها با همین دیتاست زیرمجموعه، می‌توانند تصمیم بگیرند و مدل دسته‌بند خود را بسازند.

برای پیاده‌سازی این بخش، از کتابخانه‌ی sklearn استفاده می‌کنیم و از کلاس ensemble که مربوط به الگوریتم‌های ترکیبی است، تابع RandomForestClassifier را برای دسته‌بندی مورد استفاده قرار می‌دهیم و با استفاده از تابع fit فاز آموزش را بر روی داده‌های آموزش انجام می‌دهیم و سپس با استفاده از predict، برچسب داده‌های آزمایش را معین می‌کنیم. با استفاده از دستور random_forrest test می‌توان برچسب‌ها را در کنسول مشاهده کرد.

```
def random_forrest(self, x_train, y_train, x_test):
    model = RandomForestClassifier()
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
```

تصویر ۸: تابع دسته‌بند random_forrest

```
/usr/bin/python3.6 /home/sina/Desktop/IR-phase1/Phase1/main.py
load test_classifier
test classifier loaded successfully
load phase1_classifier
phase1 classifier loaded successfully
naive_bayes test
[1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1, 1, 1, 1, 1, 1,
knn test 6
[-1, 1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, -1, -1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 1
svm test 2
[-1  1 -1 -1  1  1 -1 -1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1 -1  1 -1 -1
  1 -1  1 -1  1  1 -1  1  1  1 -1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1
 -1 -1  1 -1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1  1 -1 -1 -1 -1  1 -1
  1 -1 -1 -1  1  1 -1  1  1  1  1 -1 -1 -1 -1  1  1  1  1  1  1  1  1  1
  1  1  1 -1 -1 -1 -1 -1  1 -1  1  1 -1 -1 -1  1  1  1  1  1 -1 -1 -1 -1
 -1 -1 -1 -1  1  1  1 -1  1 -1 -1  1  1  1  1  1  1 -1  1 -1  1 -1 -1
 -1 -1  1  1  1 -1 -1  1 -1  1 -1  1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1
  1 -1 -1 -1  1 -1  1  1 -1 -1  1 -1 -1  1  1 -1 -1 -1 -1  1 -1 -1  1  1
 -1 -1  1 -1 -1 -1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1 -1 -1  1 -1 -1 -1
  1 -1 -1 -1  1  1 -1 -1 -1  1  1  1  1 -1 -1 -1  1 -1 -1  1  1 -1  1 -1
 -1 -1 -1 -1  1  1 -1 -1 -1 -1 -1  1  1 -1  1]
random_forrest test
[ 1  1 -1 -1  1  1 -1  1  1 -1  1 -1 -1 -1  1 -1  1 -1  1  1 -1 -1 -1 -1
  1 -1 -1  1  1  1 -1  1  1 -1 -1 -1  1  1  1  1  1  1  1 -1  1  1  1
 -1 -1  1 -1 -1 -1  1  1 -1  1 -1  1 -1  1  1 -1 -1 -1  1  1 -1  1  1 -1
 -1 -1 -1 -1  1  1 -1  1  1  1  1 -1  1 -1  1  1  1  1  1  1  1  1  1
  1  1  1 -1 -1 -1 -1 -1  1 -1 -1  1 -1  1 -1 -1 -1 -1  1 -1 -1 -1  1 -1
 -1 -1 -1 -1  1  1  1 -1 -1 -1 -1 -1  1  1  1 -1 -1  1 -1  1 -1  1 -1 -1
 -1 -1  1  1  1 -1 -1  1  1  1 -1  1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1 -1
  1 -1 -1 -1  1  1  1 -1 -1  1 -1 -1  1  1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1
 -1 -1  1 -1 -1 -1  1 -1 -1 -1 -1 -1  1  1 -1  1 -1 -1 -1  1 -1 -1  1  1
  1 -1 -1 -1 -1  1 -1  1 -1 -1 -1  1  1  1 -1  1 -1 -1 -1  1  1 -1  1  1
  1 -1 -1 -1  1  1 -1 -1 -1  1 -1  1  1 -1 -1]
```

تصویر ۹: برچسب‌گذاری هر یک از دسته‌بندها بر روی مجموعه داده‌ی آزمایش

محاسبه‌ی Validation Set

تابع make_validation_set و make_knn_validation_set برای این قسمت پیاده‌سازی شده‌است. هر دوی این توابع عملکرد یکسانی دارند و به ازای هر مستند در فضای برداری یک عدد رندوم با توزیع یکنواخت تولید می‌کنیم تا با احتمال 0.1 مستند در مجموعه‌ی Validation و با احتمال 0.9 در مجموعه‌ی آموزش قرار بگیرد. سپس دو مجموعه را باز می‌گردانیم.

```python
def make_validation_set(self):
    x_train_set = []
    x_validation_set = []
    y_train_set = []
    y_validation_set = []
    for i in range(self.train_size):
        m = random.uniform(0, 1)
        if m < 0.9:
            x_train_set += [self.train_vector_space[i]]
            y_train_set += [self.y_train[i]]
        else:
            x_validation_set += [self.train_vector_space[i]]
            y_validation_set += [self.y_train[i]]
    return x_train_set, y_train_set, x_validation_set, y_validation_set
```

تصویر ۱۰: تابع ایجاد مجموعه‌ی Validation از مجموعه‌ی آموزش

محاسبه‌ی بهترین پارامتر برای الگوریتم k-NN

در این بخش از میان kهای گفته شده در صورت پروژه –یعنی ۱ و ۵ و ۹– بهترین k را انتخاب می‌کنیم و در قسمت‌های بعدی نیز از آن استفاده می‌کنیم.

برای این کار از تابع find_best_k استفاده می‌کنیم. این تابع در صورتی که فلگ print برابر با True باشد، به ازای هر k از مقادیر گفته شده، تابع knn را با توجه به مجموعه‌ی آموزش و Validation صدا می‌زند و سپس با استفاده از خروجی آن و تابع find_metric که در بخش پیاده‌سازی تابع find_metric توضیح داده شده، معیارهای مختلف را به ازای هر k در خروجی چاپ می‌کند. در نهایت بهترین k را با توجه به بیشترین مقدار accuracy انتخاب کرده و چاپ می‌کنیم. در صورتی هم که فلگ print برابر False باشد، تنها بهترین مقدار k محاسبه می‌شود و چیزی چاپ نمی‌شود.

```python
def find_best_k(self, arr, print_flag):
    max_accuracy = -1
    best_k = None
    x_train_set, y_train_set, x_validation_set, y_validation_set = self.make_knn_validation_set()
    for k in arr:
        y_pred = self.knn(x_train_set, y_train_set, x_validation_set, k)
        f1 = self.find_metric(y_validation_set, y_pred, "f1")
        precision = self.find_metric(y_validation_set, y_pred, "precision")
        recall = self.find_metric(y_validation_set, y_pred, "recall")
        accuracy = self.find_metric(y_validation_set, y_pred, "accuracy")
        if print_flag:
            print("metrics for k = ", k)
            print("f1 score = ", f1)
            print("precision = ", precision)
            print("recall = ", recall)
            print("accuracy = ", accuracy)
            print()
        if accuracy > max_accuracy:
            max_accuracy = accuracy
            best_k = k
    if print_flag:
        print("best k is: ", best_k)
        print("best accuracy score is: ", max_accuracy)
    return best_k
```

تصویر ۱۱: پیاده‌سازی تابع یافتن بهترین پارامتر k

در کنسول می‌توان با دستور best k خروجی این تابع را مشاهده کرد. مشاهده می‌شود که k = 9 بهترین k بوده‌است.

```
best k
metrics for k =  1
f1 score =  0.48799999999999993
precision =  0.46564885496183206
recall =  0.5126050420168067
accuracy =  0.4732510288065844

metrics for k =  5
f1 score =  0.576923076923077
precision =  0.5319148936170213
recall =  0.6302521008403361
accuracy =  0.5473251028806584

metrics for k =  9
f1 score =  0.5643153526970954
precision =  0.5573770491803278
recall =  0.5714285714285714
accuracy =  0.5679012345679012

best k is:  9
best accuracy score is:  0.5679012345679012
```

تصویر ۱۲: خروجی دستور best k

محاسبه‌ی بهترین پارامتر برای الگوریتم SVM

برای پیاده‌سازی این بخش از تابع find_best_c استفاده شده که عینا مشابه تابع find_best_k عمل می‌کند که در بخش قبل توضیح داده شد. با این تفاوت که به جای تابع knn، تابع svm صدا زده می‌شود و مقدار بهترین C از بین مقادیر 0.5، 1، 1.5 و 2 انتخاب می‌شود.

```python
def find_best_c(self, arr, print_flag):
    max_accuracy = -1
    best_c = None
    x_train_set, y_train_set, x_validation_set, y_validation_set = self.make_validation_set()
    for c in arr:
        y_pred = self.svm(x_train_set, y_train_set, x_validation_set, c)
        f1 = self.find_metric(y_validation_set, y_pred, "f1")
        precision = self.find_metric(y_validation_set, y_pred, "precision")
        recall = self.find_metric(y_validation_set, y_pred, "recall")
        accuracy = self.find_metric(y_validation_set, y_pred, "accuracy")
        if print_flag:
            print("metrics for c = ", c)
            print("f1 score = ", f1)
            print("precision = ", precision)
            print("recall = ", recall)
            print("accuracy = ", accuracy)
            print()
        if accuracy > max_accuracy:
            max_accuracy = accuracy
            best_c = c
    if print_flag:
        print("best c is: ", best_c)
        print("best accuracy score is: ", max_accuracy)
    return best_c
```

تصویر ۱۳: تابع یافتن بهترین پارامتر C

با دستور best c می‌توان خروجی مورد نظر را مشاهده کرد که نشان می‌دهد C = 1.5 بهترین مقدار بوده‌است.

```
best c
metrics for c =  0.5
f1 score =  0.6254545454545455
precision =  0.5308641975308642
recall =  0.7610619469026548
accuracy =  0.59288537549407 12

metrics for c =  1
f1 score =  0.5738396624472574
precision =  0.5483870967741935
recall =  0.6017699115044248
accuracy =  0.60079051383399 21

metrics for c =  1.5
f1 score =  0.5907172995780591
precision =  0.5645161290322581
recall =  0.6194690265486725
accuracy =  0.616600790513834

metrics for c =  2
f1 score =  0.5822784810126581
precision =  0.5564516129032258
recall =  0.6106194690265486
accuracy =  0.60869565217391 31

best c is:  1.5
best accuracy score is:  0.616600790513834
```

تصویر ۱۴: خروجی دستور best c

بهبود سیستم بازیابی فاز اول پروژه

در این قسمت، ابتدا با استفاده از داده‌های آموزشی هر چهار دسته‌بند را آموزش می‌دهیم. سپس داده‌های فاز ۱ را که در phase1_classifier موجودند، با استفاده از هر چهار دسته‌بند برچسب می‌زنیم و ذخیره می‌کنیم. این ذخیره‌سازی به کمک دستورات تصویر زیر انجام می‌شود.

```
naive_bayes phase1
y's saved for naive_bayes method
knn phase1
y's saved for knn method
svm phase1
y's saved for svm method
random_forrest phase1
y's saved for random_forrest method
calculate majority vote
y's saved for majority_vote_prediction method
```

تصویر ۱۵: دستورات لازم برای اجرای دسته‌بندها بر روی مستندات فاز اول

دستور calculate majority vote رای‌های مستندات را مجتمع می‌کند و بهترین برچسب را انتخاب می‌کند. نحوه‌ی کار آن در بخش تابع اجماع نظر دسته‌بندها برای تعیین برچسب هر مستند توضیح داده می‌شود. سپس این برچسب‌ها را در فایلی ذخیره می‌کند.

حال می‌توانیم با توجه به ورودی کاربر که یا ۱ یا ۱- است، مستندات مرتبط با آن دسته را در خروجی به کاربر نمایش دهیم. به این صورت که پس از فیلتر کردن مستندات فاز اول بر اساس دسته، مطابق همان چیزی که در فاز اول پیاده‌سازی شده‌است، مستندات را بر اساس معیار ltc-lnc مرتب‌سازی می‌کنیم و به کاربر نمایش می‌دهیم. در ادامه توابع این بخش را توضیح می‌دهیم.

تصویر ۱۶: فایل‌های جانبی ذخیره شده که برچسب‌های هر دسته‌بندی را نگه می‌دارند.

تابع اجماع نظر دسته‌بندها برای تعیین برچسب هر مستند

در این قسمت، ابتدا برچسب‌های ذخیره‌سازی شده برای هر دسته‌بند را بارگذاری می‌کنیم و سپس برای هر مستند نظر اجماع دسته‌بندها را به عنوان برچسب آن مستند در نظر می‌گیریم.

نکته‌ای که باید به آن توجه کرد این است که چهار دسته‌بند داریم و لذا ممکن است دو دسته‌بند نظر به برچسب ۱ داشته باشند و دو تای دیگر نظر به برچسب ۱-. در این حالت برچسبی که دسته‌بند Random Forest برای مستند انتخاب کرده‌است به عنوان نتیجه‌ی نهایی در نظر گرفته می‌شود، زیرا با آن به دقت بالاتری می‌رسیدیم. همان‌طور که گفته شد، این اجماع نظر به کمک دستور calculate majority vote انجام می‌گیرد.

جستجو بر اساس دسته

تابع این بخش، phase1_query است که در IRSystem طراحی شده‌است. ابتدا نظر جمعی دسته‌بندها را لود می‌کنیم و از آن برای برچسب زدن هر مستند فاز اول استفاده می‌کنیم. سپس مانند فاز قبلی پروژه عمل می‌کنیم، با این تفاوت که جستجو را در دسته‌ی مشخص شده انجام می‌دهیم.

در این قسمت برای جستجو با انتخاب دسته‌ی مورد نظر دستور phase1 query در کنسول زده می‌شود. سپس از کاربر می‌خواهیم دسته‌ی مورد نظر را انتخاب کند و بعد از آن مستندات را با توجه به ورودی کاربر فیلتر می‌کنیم. پس از آن کاربر می‌بایست پرسمان خود را وارد کند و پس از تصحیح پرسمان و رتبه‌بندی مطابق فاز قبل، نتایج به کاربر نشان داده می‌شود.

نمونه‌هایی از اجرای این بخش در ادامه دیده می‌شود.

```
prepare english
create positional english
creation was successful
create bigram english
creation was successful
phase1 query
Please Enter Zone Of Search(1 -> Most View ,-1 -> Less View): 1
Enter Your Query: dianna cohen
no spell correction needed!
document 2054: [['alzheim', 'not', 'normal', 'age', 'cure'], ['than', 'million', 'peopl', 'worldwid', 'suffer', 'alzheim', 'diseas', 'number', 'expect', 'increas', 'drastic', 'come', 'year', 'no', 'real', 'progress',
ltc-lnc score: 0.20602230789457598
document 2167: [['get', 'back', 'work', 'after', 'career', 'break'], ['if', 'youv', 'taken', 'career', 'break', 'now', 'look', 'return', 'workforc', 'would', 'consid', 'take', 'internship', 'career', 'reentri', 'exper
ltc-lnc score: 0.1899854696322041
document 1562: [['argument', 'sake'], ['do', 'argu', 'outreason', 'oppon', 'prove', 'them', 'wrong', 'most', 'all', 'win', 'right', 'philosoph', 'daniel', 'h', 'cohen', 'show', 'most', 'common', 'form', 'argument', 'w
ltc-lnc score: 0.16529085709750133
phase1 query
Please Enter Zone Of Search(1 -> Most View ,-1 -> Less View): -1
Enter Your Query: dianna cohen
no spell correction needed!
document 795: [['tough', 'truth', 'plastic', 'pollut'], ['artist', 'dianna', 'cohen', 'share', 'some', 'tough', 'truth', 'plastic', 'pollut', 'ocean', 'live', 'some', 'thought', 'free', 'ourselv', 'plastic', 'gyre']]
ltc-lnc score: 0.4754492886582986
```

تصویر ۱۷: اجرای دستور phase1 query برای هر دو دسته با پرسمان dianna cohen

```
phase1 query
Please Enter Zone Of Search(1 -> Most View ,-1 -> Less View): 1
Enter Your Query: penicillin
no spell correction needed!
document 2004: [['do', 'do', 'when', 'antibiot', 'dont', 'work', 'ani'], ['penicillin', 'chang', 'everyth', 'infect', 'had', 'previous', 'kill', 'were', 'sudden', 'quick', 'curabl', 'yet', 'maryn', 'mckenna', 'share',
ltc-lnc score: 0.24031126697043745
phase1 query
Please Enter Zone Of Search(1 -> Most View ,-1 -> Less View): -1
Enter Your Query: penicillin
no spell correction needed!
Query Not Found in Your Zone
```

تصویر ۱۸: اجرای دستور phase1 query برای هر دو دسته با پرسمان penicillin

ارزیابی نهایی

در این بخش ابتدا تابع find_metric را پیاده‌سازی می‌کنیم که با توجه به رشته‌ی ورودی که همان معیار مورد نظر است، مقدار آن معیار را با توجه به دو ورودی دیگر که دو لیست از مقادیر واقعی و مقادیر پیش‌بینی شده هستند، خروجی می‌دهد. سپس برای هر کدام از دسته‌بندها معیارهای خواسته شده را محاسبه می‌کنیم که در ادامه توضیح داده می‌شوند.

پیاده‌سازی تابع find_metric

تابع find_metric سه ورودی به نام‌های y_test، y_pred و metric دارد که:

- ورودی y_test شامل گروه‌های واقعی مستندات آزمایش (test.csv) است.
- ورودی y_pred شامل گروه‌های پیش‌بینی شده‌ی مستندات آزمایش است.
- ورودی metric نیز معیار ارزیابی را مشخص می‌کند که یک رشته است و مقدار آن برابر با یکی از مقادیر f1، accuracy، precision یا recall است.

نحوه‌ی عملکرد این تابع نیز به این صورت است که با توجه به ورودی‌ها تعداد True Positive، True Negative، False Positive و False Negative را به دست می‌آورد. سپس با توجه به فرمول‌های اسلایدها، معیار خواسته شده را خروجی می‌دهد.

معیارهای ارزیابی برای Naïve Bayes

در این قسمت، با وارد کردن دستور naïve_bayes [metric] در کنسول، می‌توان مقدار معیار ورودی را برای دسته‌بند Naïve-Bayes مشاهده کرد.

```
accuracy naive_bayes
accuracy is:  0.6509803921568628
precision naive_bayes
precision is:  0.6159420289855072
recall naive_bayes
recall is:  0.7024793388429752
f1 naive_bayes
f1 is:  0.6563706563706564
```

تصویر ۱۹: مقدار معیارهای ارزیابی برای دسته‌بند Naïve Bayes

معیارهای ارزیابی برای k-NN

با وارد کردن دستور knn k [metric] می‌توان مقادیر معیار ورودی را برای دسته‌بند k-NN مشاهده کرد. با توجه به این که در این بخش معیارها برای بهترین k خواسته شده بود، k را برابر با 9 قرار می‌دهیم.

```
accuracy knn 9
accuracy is:  0.5647058823529412
precision knn 9
precision is:  0.5384615384615384
recall knn 9
recall is:  0.5785123966942148
f1 knn 9
f1 is:  0.5577689243027888
```

تصویر ۲۰: مقدار معیارهای ارزیابی برای دسته‌بند k-NN (k = 9)

معیارهای ارزیابی برای SVM

دستور svm c [metric] را در کنسول وارد می‌کنیم که با توجه به صورت پروژه، مقدار c را برابر با بهترین C، یعنی 1.5 قرار می‌دهیم.

```
accuracy svm 1.5
accuracy is:  0.6666666666666666
precision svm 1.5
precision is:  0.6666666666666666
recall svm 1.5
recall is:  0.5950413223140496
f1 svm 1.5
f1 is:  0.62882096069869
```

تصویر ۲۱: مقدار معیارهای ارزیابی برای دسته‌بند SVM (C = 1.5)

معیارهای ارزیابی برای Random Forest

از دستور random_forrest [metric] استفاده می‌کنیم که مقدار معیار ورودی را تحت دسته‌بندی Random Forest محاسبه می‌کند.

```
accuracy random_forrest
accuracy is:  0.6431372549019608
precision random_forrest
precision is:  0.6728971962616822
recall random_forrest
recall is:  0.628099173553719
f1 random_forrest
f1 is:  0.6419753086419753
```

تصویر ۲۲: مقدار معیارهای ارزیابی برای دسته‌بند Random Forest

نتیجه‌گیری نهایی معیارهای ارزیابی

با توجه به تصاویر ۱۹، ۲۰، ۲۱ و ۲۲ می‌توان نتیجه گرفت که:

- دسته‌بند SVM بیشترین accuracy را داشته‌است.
- دسته‌بند Random Forest بیشترین مقدار precision را حاصل کرده‌است.
- دسته‌بند Naïve Bayes بیشترین مقدار F1 و recall را داشته‌است.
- معیارهای ارزیابی دسته‌بند k-NN نسبت به سایر دسته‌بندها پایین‌تر بوده‌است.

نحوه‌ی تقسیم وظایف

وظایف اختصاص یافته به هر فرد به شرح زیر بود:

نیما جمالی

۱- پیش‌پردازش داده‌ها و پیاده‌سازی روش ntn

۲- ساخت دو مدل فضای برداری و توابع مربوط به آنها

۳- پیاده‌سازی دسته‌بند k-NN و توابع جانبی آن

۴- پیاده‌سازی تابع محاسبه‌ی بهترین پارامتر k و ساختن مجموعه‌ی Validation

سپهر فعلی

۱- طراحی دسته‌بند Naïve-Bayes و توابع مربوط به آن

۲- طراحی توابع محاسبه‌ی معیارهای ارزیابی

۳- مقایسه‌ی معیارهای ارزیابی برای دسته‌بندها

سینا کاظمی

۱- طراحی دسته‌بندهای SVM و Random Forest

۲- پیاده‌سازی تابع محاسبه‌ی بهترین پارامتر C برای دسته‌بند SVM

۳- طراحی تابع اجماع نظر دسته‌بندها

۴- طراحی تابع پاسخ پرسمان، بسته به دسته‌ی ورودی

مراجع

[1] https://scikit-learn.org/stable/modules/svm.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html