گزارش پروژه:

ساخت مدل دسته‌بندی متون فارسی با الگوریتم بیز ساده لوحانه

معرفی پروژه و هدف:

در این پروژه قصد داریم یک مدل یادگیری ماشین ساده اما مؤثر برای دسته‌بندی متن‌های فارسی بسازیم.

داده ورودی مجموعه‌ای از کامنت‌های متنی فارسی است که هر کامنت دارای برچسب دوکلاسه (price_value) است، که به عنوان هدف (label) مدل استفاده می‌شود.

هدف، پیش‌بینی دسته هر کامنت جدید است.

پیش‌زمینه و انتخاب الگوریتم

یکی از ساده‌ترین و سریع‌ترین الگوریتم‌های دسته‌بندی متن است.

به‌خاطر فرض استقلال شرطی بین ویژگی‌ها (کلمات)، به راحتی قابل پیاده‌سازی است.

علی‌رغم سادگی، عملکرد بسیار خوبی روی مسائل طبقه‌بندی متن دارد.

مناسب برای داده‌های با ابعاد بزرگ و متن‌هایی که به شکل توکنیزه شده استفاده می‌شوند.

داده‌ها شامل دو ستون اصلی هستند:

comment: متن کامنت به زبان فارسی

price value برچسب هدف، دوکلاسه مثلا ۰ برای دسته منفی و ۱ برای دسته مثبت

پیش‌پردازش متون (Text Preprocessing)

در این بخش، تابعی برای پیش‌پردازش نظرات متنی فارسی تعریف کرده‌ایم تا آن‌ها را برای استفاده در مدل یادگیری ماشین آماده کنیم. این تابع، مراحل زیر را روی هر نظر (Comment) اجرا می‌کند:

1. نرمال‌سازی (Normalization):

با استفاده از Normalizer کتابخانه Hazm، متون فارسی از لحاظ نوشتاری یک‌دست می‌شوند. به عنوان مثال، "می‌باشد" به "است" تبدیل می‌شود، فاصله‌های اضافی حذف می‌شوند و شکل‌های مختلف یک حرف به فرم استاندارد تبدیل می‌شوند.

2. حذف اعداد و نمادها:

با استفاده از عبارات باقاعده (Regular Expression)، تمام کاراکترهایی که غیرفارسی یا غیرحروف هستند (مانند اعداد، علائم نگارشی، کاراکترهای لاتین و ...) حذف می‌شوند تا فقط حروف فارسی و فاصله‌ها باقی بمانند.

3. توکن‌سازی (Tokenization):

متن نرمال‌سازی شده به کلمات مجزا (توکن‌ها) تقسیم می‌شود. این کار با تابع word_tokenize از کتابخانه Hazm انجام می‌شود.

4. حذف کلمات توقف (Stopwords):

کلمات رایج و بی‌اهمیت (مانند "اما"، "برای"، "از" و ...) با استفاده از stopwords_list() حذف می‌شوند. این کلمات معمولاً اطلاعات خاصی در مورد محتوای متن منتقل نمی‌کنند و حذف آن‌ها به مدل کمک می‌کند تا بهتر روی ویژگی‌های مهم تمرکز کند.

5. ریشه‌یابی (Stemming):

هر توکن باقی‌مانده به ریشه خود بازگردانده می‌شود. مثلاً "می‌نویسند" به "نوشت" تبدیل می‌شود. این کار باعث می‌شود که اشکال مختلف یک کلمه به صورت یکنواخت در نظر گرفته شوند.

6. فیلتر نهایی:

فقط توکن‌هایی که طول آن‌ها بیشتر از یک حرف هستند و در لیست کلمات توقف قرار ندارند، پس از ریشه‌یابی به لیست نهایی filtered افزوده می‌شوند.

در نهایت، این تابع یک لیست از کلمات مهم و پردازش‌شده‌ی هر نظر را به عنوان خروجی برمی‌گرداند و پایه‌ی اصلی برای مدل‌سازی یادگیری ماشین را فراهم می‌کند

```python
from hazm import Normalizer, word_tokenize, Stemmer, stopwords_list
import re

normalizer = Normalizer()
stemmer = Stemmer()
stop_words = set(stopwords_list())

def preprocessing(comment):
    filtered = []

    # نرمال‌سازی
    comment = normalizer.normalize(comment)

    # حذف اعداد و نمادها (فقط حروف و فاصله نگه داشته می‌شود)
    comment = re.sub(r'[^\u0600-\u06FF\s]', '', comment)  # حذف غیرحروف فارسی

    # توکن‌سازی
    tokens = word_tokenize(comment)

    for token in tokens:
        if token not in stop_words and len(token) > 1:
            stemmed = stemmer.stem(token)
            filtered.append(stemmed)

    return filtered
```

محاسبه احتمال پیشین (Prior Probability)

در این بخش، هدف ما محاسبه احتمال پیشین برای هر برچسب (کلاس) موجود در داده‌هاست.

مواجه هستیم، binary classificationاز آن‌جایی که در این پروژه با یک مسئله‌ی دسته‌بندی دودویی (لازم است بدانیم چه نسبتی از داده‌های آموزشی به کلاس "۰" (مثلاً نظرات منفی یا کم‌قیمت) و چه نسبتی به کلاس "۱" (مثلاً نظرات مثبت یا پرقیمت) تعلق دارد.

در این کد:

تعداد تکرار هر کلاس را شمرده و آن را نرمال‌سازی (به احتمال value_counts(normalize=True) تبدیل) می‌کند.

() این مقادیر به یک دیکشنری تبدیل می‌شوند، که در آن، کلیدها کلاس‌ها (۰ و ۱) و to_dictسپس با مقدارها احتمال وقوع هر کلاس هستند.

این احتمال‌های پیشین در ادامه برای محاسبه احتمال نهایی در الگوریتم استفاده می‌شوند.

```
prior_probability = train_data['price_value'].value_counts(normalize=True).to_dict()
print(prior_probability)
```

(Train-Test Split) تقسیم داده به آموزش و آزمون

برای آموزش مدل و ارزیابی عملکرد آن، ابتدا داده‌های موجود را به دو بخش آموزش (Training) و آزمون (Testing) تقسیم می‌کنیم. این کار کمک می‌کند مدل روی بخشی از داده‌ها آموزش ببیند و سپس روی داده‌هایی که قبلاً ندیده است، مورد ارزیابی قرار گیرد

```
from sklearn.model_selection import train_test_split
x = train_data['comment']
y = train_data['price_value']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=42)

df_train = pd.DataFrame({'comment': x_train, 'price_value': y_train})
df_train['price_value'].value_counts(normalize=True)
df_test = pd.DataFrame({'comment': x_test, 'price_value': y_test})
```

شمارش تعداد توکن‌ها در هر کلاس

برای پیاده‌سازی الگوریتم Naive Bayes به‌صورت دستی، لازم است تعداد تکرار هر توکن (کلمه) در هر کلاس قیمتی را بدانیم. به همین منظور تابع token_counter تعریف شده است

شرح عملکرد تابع

این تابع برای شمارش تعداد تکرار هر توکن (کلمه) در دو کلاس مختلف از داده‌ها (برچسب‌های قیمتی) طراحی شده است. ورودی تابع شامل دو پارامتر است:

- df_train: دیتافریم آموزش شامل ستون‌های price_value و commentی
- preprocessing: تابعی که روی متن اعمال می‌شود تا پیش‌پردازش و توکن‌سازی انجام دهد

عملکرد کلی تابع به این صورت است:

- ابتدا دو دیکشنری به نام‌های token_counts_0 و token_counts_1 تعریف می‌شود تا تعداد توکن‌های دیده شده در هر کلاس را ذخیره کند.
- سپس، با استفاده از یک حلقه، روی ردیف‌های دیتافریم پیمایش انجام می‌شود.
- در هر ردیف، متن (comment) و برچسب آن (price_value) استخراج می‌شود.
- با اعمال تابع preprocessing، متن به لیستی از توکن‌ها تبدیل می‌شود.
- اگر برچسب برابر با 0 باشد، هر توکن در دیکشنری token_counts_0 شمارش می‌شود.
- اگر برچسب برابر با 1 باشد، توکن‌ها در دیکشنری token_counts_1 ثبت می‌شوند.
- در نهایت، تابع سه خروجی برمی‌گرداند:
  - دیکشنری شامل تعداد توکن‌های کلاس ۰
  - دیکشنری شامل تعداد توکن‌های کلاس ۱
  - دو متغیر برای مجموع توکن‌ها در هر کلاس (که فعلاً صفر هستند و در صورت نیاز می‌توان آن‌ها را در ادامه توسعه داد.

```
from collections import defaultdict
def token_counter(df_train, preprocessing):
    token_counts_0 = defaultdict(int)
    token_counts_1 = defaultdict(int)
    total_tokens_0 = 0
    total_tokens_1 = 0

    for _, row in df_train.iterrows():
        text = row['comment']
        label = row['price_value']
        tokens = preprocessing(text)
        if label == 0:
            for token in tokens:
                token_counts_0[token] += 1
        elif label == 1:
            for token in tokens:
                token_counts_1[token] +=1
```

```
    return dict(token_counts_0), dict(token_counts_1), total_tokens_0,
total_tokens_1
```

در این خط، تابع token_counter بر روی دیتافریم df_train اعمال می‌شود. این تابع، ابتدا هر کامنت را با استفاده از تابع preprocessing پیش‌پردازش می‌کند (شامل نرمال‌سازی، حذف اعداد و نمادها، حذف کلمات توقف و ریشه‌یابی). سپس با توجه به مقدار برچسب price_value (که می‌تواند ۰ یا ۱ باشد)، توکن‌های موجود در کامنت‌ها را به دو دسته تقسیم کرده و تعداد تکرار هر توکن را در دیکشنری‌های جداگانه ذخیره می‌کند

```
token_counts_0, token_counts_1, total_tokens_0, total_tokens_1 =
token_counter(df_train, preprocessing)
```

محاسبه احتمال پیشین کلاس‌ها و ساخت واژگان (Vocabulary)

در این قسمت از کد، ابتدا تعداد نمونه‌های هر کلاس محاسبه می‌شود

سپس تعداد کل نمونه‌های دیتاست آموزش به دست می‌آید

با استفاده از این اطلاعات، احتمال پیشین هر کلاس (Prior Probability) محاسبه می‌شود

در نهایت، برای محاسبه احتمال توکن‌ها در مراحل بعدی، واژگان کلی (Vocabulary) ساخته می‌شود. این واژگان شامل تمام کلماتی است که در کلاس ۰ یا ۱ دیده شده‌اند

```
num_class_0 = df_train[df_train['price_value'] == 0].shape[0]
num_class_1 = df_train[df_train['price_value'] == 1].shape[0]
total_samples = len(df_train)

p0 = num_class_0 / total_samples
p1 = num_class_1 / total_samples

vocab = set(list(token_counts_0.keys()) + list(token_counts_1.keys()))
```

شرح عملکرد تابع
compute_probability

تابع compute_probability برای محاسبه‌ی احتمال تعلق یک متن به یک کلاس خاص طراحی شده است. این محاسبه بر اساس الگوریتم Naive Bayes و با استفاده از لاپلاس اسموتینگ انجام می‌شود.

در این تابع، ورودی‌ها شامل:

- text: متنی است که می‌خواهیم احتمال تعلق آن به یک کلاس خاص را بسنجیم.
- class_label: برچسب کلاس مورد نظر (۰ یا ۱) است.

مراحل اجرا

۱. پیش‌پردازش متن

ابتدا متن ورودی توسط تابع preprocessing به توکن‌های پردازش‌شده (کلمات تمیز شده و ریشه‌یابی‌شده) تبدیل می‌شود.

۲. انتخاب آمار کلاس

بر اساس class_label مشخص می‌شود که از کدام آمار استفاده کنیم:

- prior: احتمال پیشین کلاس (p0 یا p1)
- token_counts: دیکشنری فراوانی توکن‌ها در آن کلاس
- total_tokens: مجموع توکن‌ها در آن کلاس

۳. محاسبه لگاریتم احتمال اولیه

مقدار log_prob با لگاریتم احتمال پیشین شروع می‌شود.

۴. محاسبه احتمال توکن‌ها

برای هر توکن در متن:

- احتمال آن توکن در کلاس با استفاده از فرمول لاپلاس اسموتینگ محاسبه می‌شود
- سپس لگاریتم این احتمال به log_prob اضافه می‌شود.

۵. بازگشت احتمال نهایی

در نهایت مقدار لگاریتمی احتمال با تابع نمایی (math.exp) به احتمال نهایی تبدیل و بازگردانده می‌شود

```
import math
def compute_probability(text, class_label):
    tokens = preprocessing(text)

    if class_label == 0:
        prior = p0
        token_counts = token_counts_0
        total_tokens = total_tokens_0
    else:
        prior = p1
        token_counts = token_counts_1
        total_tokens = total_tokens_1

    log_prob = math.log(prior)



    for token in tokens:
        token_prob = (token_counts.get(token, 0) + 1) / (total_tokens + len(vocab))
        log_prob += math.log(token_prob)
    return math.exp(log_prob)
```

تابع
predict

این تابع وظیفه دارد که متن‌های ورودی را بر اساس مدل نهایی طبقه‌بندی کند. ورودی آن، یک لیست از متون است (text_list) که می‌خواهیم برچسب مربوط به آن‌ها (۰ یا ۱) را پیش‌بینی کنیم.

در ابتدا، یک لیست خالی به نام predictions برای ذخیره نتایج پیش‌بینی‌ها ساخته می‌شود

برای هر متن در لیست:

- احتمال تعلق متن به کلاس ۰ محاسبه می‌شود (prob_0).
- احتمال تعلق آن به کلاس ۱ محاسبه می‌شود (prob_1)

سپس با مقایسه این دو احتمال، کلاس نهایی انتخاب می‌شود:

- اگر prob_0 بیشتر باشد، برچسب ۰ انتخاب می‌شود.
- در غیر این صورت، برچسب ۱.

برچسب پیش‌بینی‌شده به لیست نتایج افزوده می‌شود

در پایان، خروجی نهایی به صورت یک آرایه NumPy بازگردانده می‌شود.

```
def predict(text_list):
    predictions = []
    for text in text_list:
        prob_0 = compute_probability(text, 0)
        prob_1 = compute_probability(text, 1)
        predicted_label = 0 if prob_0 > prob_1 else 1
        predictions.append(predicted_label)
    return np.array(predictions)
```

ارزیابی مدل: محاسبه دقت و گزارش طبقه‌بندی

در این قسمت، عملکرد مدل آموزش‌دیده با استفاده از داده‌های تست بررسی می‌شود

- میزان دقت (Accuracy) مدل چاپ می‌شود، که نشان‌دهنده نسبت پیش‌بینی‌های درست به کل نمونه‌ها است.
- گزارش طبقه‌بندی (Classification Report) نیز چاپ می‌شود که شامل مقادیر دقیق‌تری مانند دقت، فراخوانی و F1 برای هر کلاس است.

```
from sklearn.metrics import accuracy_score, classification_report

y_test = df_test['price_value']  # برچسب‌های واقعی تست
y_pred = predict(df_test['comment'].tolist())

print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

برای ارزیابی عملکرد مدل، داده‌های تست شامل ۸۰۰۰ نمونه مورد استفاده قرار گرفت. معیارهای Accuracy، Precision، Recall و F1-score محاسبه شدند.

```
Accuracy: 0.839
              precision    recall  f1-score   support

           0       0.86      0.83      0.84      4145
           1       0.82      0.85      0.84      3855

    accuracy                           0.84      8000
   macro avg       0.84      0.84      0.84      8000
weighted avg       0.84      0.84      0.84      8000
```

این نتایج نشان می‌دهد که مدل در تشخیص هر دو کلاس عملکرد متعادلی دارد و تفاوت معنی‌داری بین دقت برای کلاس‌های مختلف مشاهده نمی‌شود. مقدار نسبتاً بالای F1-score بیانگر این است که مدل نه‌تنها در پیش‌بینی درست نمونه‌ها موفق بوده، بلکه توانسته تعادل خوبی بین Precision و Recall برقرار کند.

به طور کلی، با توجه به ماهیت داده‌ها و روش استفاده‌شده، این سطح از دقت قابل قبول است و می‌تواند در سناریوهای واقعی مورد استفاده قرار گیرد. برای بهبود بیشتر دقت، می‌توان روش‌های پیش‌پردازش پیشرفته‌تر، انتخاب ویژگی (Feature Selection)، یا الگوریتم‌های یادگیری ماشین دیگر را آزمایش کرد.