**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:**

**متن کامنت به زبان فارسی :comment**

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

## **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.**

1. **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.**

1. **توکن‌سازی (Tokenization):**

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

1. **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.**

1. **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.**

1. **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**

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.**

**از آن‌جایی که در این پروژه با یک مسئله‌ی دسته‌بندی دودویی (binary classification) مواجه هستیم، 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:**

**value\_counts(normalize=True) تعداد تکرار هر کلاس را شمرده و آن را نرمال‌سازی (به احتمال تبدیل) می‌کند.**

**سپس با to\_dict() این مقادیر به یک دیکشنری تبدیل می‌شوند، که در آن، کلیدها کلاس‌ها (۰ و ۱) و 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)

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

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 comment و price\_value
* 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))