



Twitter Sentiment Analysis with LSTM Neural Network (Persian-Specific)

گزارش پژوهش پایانی مقطع کارشناسی

نام استاد پژوهش:
دکتر سید علیت الله علوی

نام دانشجو:
آرمیتا صفی خانی قلی زاده

شماره دانشجویی:
9973132

تاریخ تحویل:
اردیبهشت 1404

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

فهرست

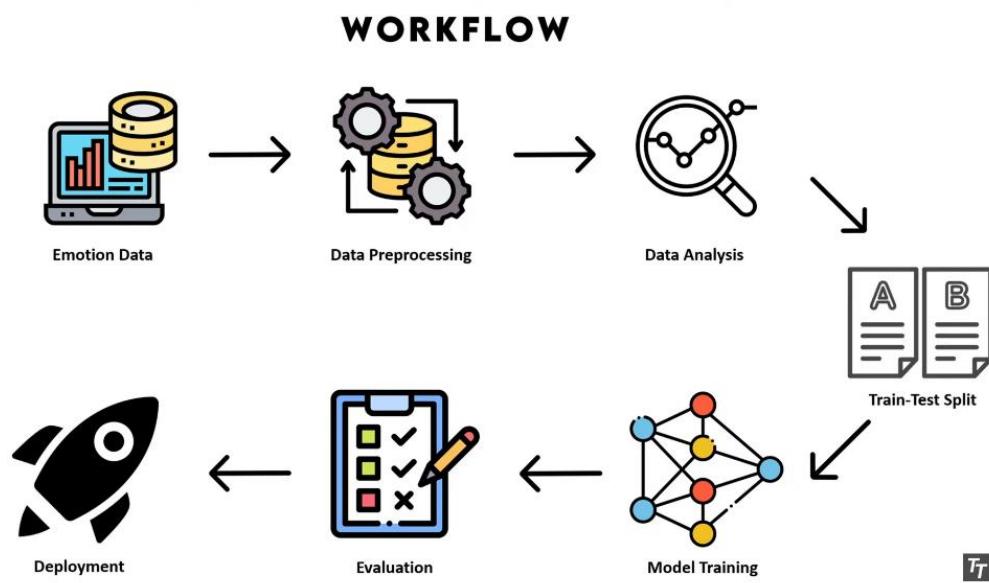
Error! Bookmark not defined.	مقدمه
8	پیش پردازش
16	تحلیل اکتشافی داده ها EDA
18	آماده سازی داده های برای مدل یادگیری عمیق
23	مدل سازی با شبکه های عصبی
24	مدل LSTM
33	آموزش مدل LSTM
47	استخراج ویژگی با TF-IDF
50	مدل بیز ساده چند جمله ای
51	مدل درخت تصمیم
52	آموزش و ارزیابی مدل های طبقه بندی کلاسیک با ویژگی های TF-IDF تری گرام
56	مقایسه عملکرد مدل کلاسیک برتر با مدل یادگیری عمیق برتر
59	GUI

مقدمه

تحلیل احساسات (Sentiment Analysis) یکی از شاخه‌های مهم در حوزه‌ی پردازش زبان طبیعی (Natural Language Processing - NLP) است که هدف آن استخراج نگرش، احساس یا نظر افراد نسبت به یک موضوع خاص از طریق متون زبانی می‌باشد. در این پروژه، هدف اصلی تحلیل احساسات کاربران فارسی‌زبان در شبکه‌ی اجتماعی توییتر (Twitter) است. با توجه به رشد روزافزون تولید محتوای فارسی در فضای مجازی، نیاز به مدل‌های دقیق و بهینه برای تحلیل این داده‌ها پیش از پیش احساس می‌شود.

در این پروژه، تمرکز ما بر اساس دیتاست انتخابی تحلیل احساسات در دو کلاس (sad) و (joy) است. با توجه به چالش‌های موجود از جمله ساختار زبانی پیچیده‌ی فارسی و محدودیت منابع پردازشی، سعی شده است راه حل‌هایی مبتنی بر مدل‌های یادگیری عمیق در کنار مدل‌های یادگیری ماشین کلاسیک به کار رود و در اخر عملکرد مدل‌ها ارزیابی و مقایسه شود. این پروژه روی تحلیل احساسات توییت‌های فارسی تمرکز دارد. هدف این است که ابتدا داده‌های فارسی را پاکسازی و پیش‌پردازش کنیم، سپس با استفاده از مدل‌های Decision Tree، LSTM، Naïve Bayes و بردارهای تعبیه شده (FastText embeddings)، مدلی برای دسته‌بندی احساسات آموزش بدهیم.

جريان کاری پروژه



در این پروژه، مراحل پردازش و تحلیل داده‌های متنه به صورت ساختاریافته و مطابق با استانداردهای رایج در حوزه یادگیری ماشین و پردازش زبان طبیعی (NLP) پیش رفته‌اند. روند کلی به شرح زیر است:

Emotion Data : در مرحله نخست، داده‌های خام مرتبط با احساسات (شامل توییت‌ها با برچسب‌های عاطفی) جمع‌آوری شدند. این داده‌ها منبع اصلی برای انجام تحلیل‌های بعدی محسوب می‌شوند.

Data Preprocessing : داده‌های خام مورد پیش‌پردازش قرار گرفتند که شامل پاکسازی متون، حذف نویزها، نرمال‌سازی، حذف ایست واژه‌ها و سایر مراحل استاندارد در پردازش زبان فارسی با ابزارهایی مانند Parsivar بود.

Data Analysis : داده‌های تمیزشده مورد تحلیل اکتشافی قرار گرفتند تا درک بهتری از توزیع برچسب‌ها، طول توییت‌ها، توزیع واژگان و سایر ویژگی‌های آماری به‌دست آید.

Train-Test Split : مجموعه داده به دو بخش آموزش و آزمون تقسیم شد تا بتوان عملکرد مدل را در شرایط واقعی‌تر ارزیابی کرد. این تقسیم‌بندی معمولاً با نسبت‌هایی مانند ۸۰٪ آموزش و ۲۰٪ آزمون انجام می‌شود.

Model Training : در این مرحله، مدل‌های عمیق یا مدل‌های کلاسیک با استفاده از داده‌های آموزشی، آ«وژش داده می‌شوند.

Evaluation : عملکرد مدل با استفاده از داده‌های آزمون و معیارهایی مانند F1-score ، دقت (Recall) و غیره ارزیابی شد.

Deployment : در نهایت، مدل آموزش‌دیده در قالب یک رابط گرافیکی (GUI) یا وب‌سرویس برای استفاده نهایی پیاده‌سازی شده است.

دیتاست ما شامل 62352 نمونه (رکورد) است که هر کدام ویژگی‌های زیر را شامل می‌شود:

hashtags: هشتگ‌های همراه توییت (لیستی از کلمات کلیدی)

likeCount: تعداد لایک‌ها

quoteCount: تعداد نقل قول‌ها

replyCount: تعداد پاسخ‌ها

retweetCount: تعداد بازنشر‌ها (ریتیویت)

sourceLabel: منبع ارسال توییت (مثلاً Twitter Web App یا Android)

Tweet: متن کامل توییت

emotion: برچسب احساسی (sad) یا (joy)

اطلاعات فراداده‌ای (**Metadata**) مانند هشتگ‌ها، تعداد لایک، ریتیویت و منبع ارسال توییت را حذف می‌کنیم و ستون‌های کلیدی را نگه میداریم. ستون‌های کلیدی برای این پروژه عبارتند از:
Tweet: متن اصلی توییت.

Hashtags: هشتگ‌های مرتبط با توییت.

Emotion: برچسب احساسی (sad) یا (joy) که به هر توییت تخصیص داده شده است.

رویکرد کلی:

فرآیند تحلیل احساسات در این پروژه شامل مراحل اصلی زیر است:

پیش‌پردازش داده‌ها: پاکسازی و آماده‌سازی متن توییت‌ها و هشتگ‌ها برای تحلیل

تحلیل اکتشافی داده‌ها (EDA): بررسی ویژگی‌های دیتاست، توزیع احساسات و کلمات کلیدی

استخراج ویژگی: تبدیل متن پیش‌پردازش شده به نمایش عددی قابل فهم برای مدل‌های یادگیری ماشین (با استفاده از **TF-IDF** و **Word Embeddings**)

مدل‌سازی: آموزش و ارزیابی مدل‌های مختلف یادگیری ماشین کلاسیک و یادگیری عمیق

ارزیابی و انتخاب مدل: مقایسه عملکرد مدل‌ها و انتخاب بهترین مدل بر اساس معیارهای ارزیابی

ذخیره مدل نهایی و deploy: ساخت **app** و رابط گرافیکی برای تست مدل

ابزارهای مورد استفاده

برای پیاده‌سازی این پروژه از زبان برنامه‌نویسی پایتون در محیط **Jupyter Notebook** و کتابخانه‌های زیر استفاده شده است:

Pandas: برای مدیریت و دستکاری دیتا فریم‌ها.

Parsivar: کتابخانه‌ای تخصصی برای پیش‌پردازش متن فارسی (نرمال‌سازی، توکنایزیشن).

* علت عدم استفاده از ابزار Hazm، ناهمانگی بین این ابزار با نسخه‌ی pandas مورد استفاده Tensorflow بود. بعلاوه اینکه هیچ برتری بین این دو ابزار Hazm و Parsivar وجود ندارد و از امکانات یکسانی برخوردارند. مثلاً اینکه هر دو قادر مانکنیزی برای هندل کردن (پردازش) ایموجی در متون فارسی هستند.

Re (Regular Expressions): برای تعریف و پیدا کردن الگوهای متنی و پاکسازی آن‌ها.

Scikit-learn: برای تقسیم داده‌ها، استخراج ویژگی TF-IDF، پیاده‌سازی مدل‌های کلاسیک و معیارهای ارزیابی.

TensorFlow/Keras: برای ساخت، آموزش و ارزیابی مدل یادگیری عمیق (LSTM).

Numpy: برای عملیات عددی و کار با آرایه‌ها. (به ویژه ماتریس Embedding)

Matplotlib & Seaborn: برای بصری‌سازی داده‌ها و نتایج.

FastText: برای استفاده از بردارهای کلمه از پیش آموزش دیده.

Arabic Reshaper & Bidi Algorith: برای نمایش صحیح متون فارسی در نمودارها.

1. پیش‌پردازش داده‌ها (Data Preprocessing)

پیش‌پردازش یک گام حیاتی برای آماده‌سازی داده‌های متنی خام است. متن توییت‌ها معمولاً شامل نویز، غلط‌های املایی، کلمات محاوره‌ای، ایموجی‌ها و عناصر خاص شبکه‌های اجتماعی است که باید قبل از مدل‌سازی مدیریت شوند.

- بارگذاری و انتخاب ستون‌های مرتبط

ابتدا دیتابست از فایل CSV بارگذاری شد. سپس، تنها ستون‌های ضروری برای تحلیل احساسات، یعنی emotion، tweet و hashtags، انتخاب شدند. نام ستون‌های tweet و emotion به ترتیب به emotion_label و tweet_text تغییر یافت تا خوانایی کد افزایش یابد. حذف ستون‌های غیرضروری مانند تعداد لایک یا منبع توییت، باعث تمرکز بر اطلاعات متنی اصلی، کاهش حجم داده و افزایش سرعت پردازش می‌شود.

- مدیریت داده‌های گمشده

بررسی شد که آیا مقادیر گمشده Null یا NaN در ستون‌های انتخابی وجود دارد یا خیر. در این دیتابست، هیچ مقدار گمشده یا ناموجود ای یافت نشد. در صورت وجود، معمولاً سطرهای حاوی مقادیر گمشده حذف می‌شوند یا با مقادیر مناسب جایگزین می‌گردند. زیرا که مقادیر گمشده می‌توانند منجر به خطا در الگوریتم‌ها شوند و بر دقت نتایج تأثیر منفی بگذارند.

- تبدیل برچسب‌های احساسی به فرمت عددی

برچسب‌های متنی 'joy' و 'sad' به ترتیب به مقادیر عددی 0 و 1 نگاشت شدند و در ستون جدید emotion_numeric ذخیره گردیدند. زیرا اکثر الگوریتم‌های یادگیری ماشین به ورودی‌ها و خروجی‌های عددی نیاز دارند و این تبدیل، داده‌ها را برای مدل‌سازی آماده می‌کند.

- پردازش ایموجی‌ها

در کتابخانه‌های رایج پردازش زبان طبیعی فارسی مانند Parsivar و Hazm، قابلیت پیش‌فرضی برای شناسایی و تبدیل ایموجی‌ها به معادل‌های متنی فارسی وجود ندارد. این در حالی است که

ایموجی‌ها، به‌ویژه در توبیت‌ها، نقش مهمی در انتقال احساسات به همین دلیل لازم بود قبل از مرحله‌ی **Tokenization** و تحلیل زبانی، ایموجی‌ها به واژه‌های معنادار تبدیل شوند.

از آنجایی که در منابع فارسی مانند گیت‌هاب نیز دیتافریم یا لیست استانداردی برای نگاشت ایموجی‌ها به کلمات معادل فارسی در دسترس نبود، در این پروژه به صورت دستی یک دایره‌المعارف سفارشی (dictionary) از ایموجی‌ها و معادل فارسی آن‌ها طراحی شد. این دیکشنری در ابتدا پیش‌پردازش روی متن اعمال شده و جایگزینی را انجام می‌دهد

در بخش ابتدایی اسکریپت (بخش emoji_to_persian)، برای پرکاربردترین ایموجی‌های احساسی مانند ، و ... معادل‌های متنی فارسی مانند «خنده»، «عشق»، «قلب»، «گریه»، «عصبانی» و ... تعریف شدند. این نگاشت به صورت مستقیم در مرحله‌ی اولیه‌ی تابع preprocess_persian_tweet() اعمال می‌شود هدف این است که مدل به جای دریافت کاراکتر گرافیکی، مفهوم احساسی پشت ایموجی را در قالب واژه‌های فارسی دریافت کند.

نگاه ویژه به ایموجی " " :

این ایموجی بسته به بافت جمله می‌تواند معانی متفاوتی داشته باشد: در برخی موارد نشان‌دهنده‌ی غم و ناراحتی، و در مواردی دیگر نمادی از شوق و هیجان شدید است. برای رفع این ابهام معنایی، دو لیست تعریف شد:

کلمات با بار معنایی مثبت : positive_keywords_for_cry_emoji

کلمات با بار معنایی منفی : negative_keywords_for_cry_emoji

این کلمات از ۳۰ واژه‌ی پرتکرار که همزمان با " " در توبیت‌ها دیده شده بودند استخراج شده‌اند، و سپس به دو دسته‌ی مثبت و منفی تقسیم شده‌اند. (co-occurrence words)

در کد، ابتدا تعداد وقوع " " در متن بررسی می‌شود (num_cry_emojis) و سپس با استفاده از (...any) بررسی می‌شود که آیا واژگان مثبت یا منفی در متن وجود دارند یا نه. بسته به این تحلیل، یکی از جایگزین‌های زیر استفاده می‌شود:

اگر واژگان مثبت وجود داشتند و منفی وجود نداشت → جایگزین با "ashk_showq"

اگر فقط واژگان منفی وجود داشتند → جایگزین با "geryeh_shidid"

در غیر این صورت (پیش‌فرض) → جایگزین با "geryeh"

```

num_cry_emojis = text.count("CRY")
if num_cry_emojis > 0:
    has_positive = any(keyword in text for keyword in positive_keywords_for_cry_emoji)
    has_negative = any(keyword in text for keyword in negative_keywords_for_cry_emoji)

replacement_token_for_cry = "گریه" # مقدار پیشفرض

if has_positive and not has_negative:
    replacement_token_for_cry = "اشک_شوق" # ذوق
elif has_negative and not has_positive:
    replacement_token_for_cry = "گردشید"

```

• پردازش جامع متن فارسی با Parsivar

برای پردازش متن فارسی، از کتابخانه قدرتمند **Parsivar** استفاده شد که ابزارهای ویژه‌ای برای نرمال‌سازی و توکنایز کردن زبان فارسی ارائه می‌دهد. یک تابع جامع به نام `preprocess_persian_tweet()` تعریف شد که مراحل زیر را پیاده‌سازی می‌کند:

• نرمال‌سازی (Normalization)

فرآیند یکسان‌سازی متن برای کاهش تنوع نوشتاری بدون تغییر معنا.

Parsivar.Normalizer() اقداماتی مانند تبدیل کاراکترهای عربی ("ك", "ي") به فارسی ("ک", "ي")، تبدیل اعداد عربی/انگلیسی به فارسی، اصلاح نیمفاصله‌ها و حذف علام نگارشی تکراری یا زائد را انجام می‌دهد.

نرمال‌سازی به مدل کمک می‌کند تا اشکال مختلف نوشتاری یک کلمه را یکسان در نظر بگیرد (مثلاً "خانه" و "خونه") و از ایجاد ویژگی‌های تکراری جلوگیری می‌کند.

نرمال‌سازهای عمومی معمولاً شامل قوانین پیچیده برای تبدیل تمام کلمات یا عبارات محاوره‌ای به شکل رسمی آن‌ها نیستند. مثلاً "خونه" به "خانه" بیشتر به ریشه‌یابی (Stemming) یا بنوازه‌سازی (Lemmatization) مربوط می‌شود تا نرمال‌سازی کاراکتری ساده، اگرچه **Parsivar** ابزار ریشه‌یابی هم دارد اما **Normalizer** این کار را انجام نمی‌دهد.

• توکن‌سازی (Tokenization) و فیلتر کردن توکن‌ها:

تعريف **Tokenization** : فرآیند شکستن متن به واحدهای کوچک‌تر معنادار که معمولاً کلمات یا علام نگارشی هستند. از **Parsivar.Tokenizer()** برای شکستن متن نرمال‌شده به لیستی از کلمات (توکن‌ها) استفاده شد. سپس توکن‌هایی که طول آن‌ها کمتر از 2 حرف بود (مانند حروف اضافه تک حرفی که ممکن است باقی مانده باشند یا حروف پی‌معنی) حذف شدند. در نهایت، توکن‌های

باقی مانده با یک فاصله () به هم متصل شدند تا متن پاکسازی شده نهایی ایجاد شود. توکنایزیشن اساس بسیاری از تکنیک‌های NLP مانند استخراج ویژگی است. حذف توکن‌های بسیار کوتاه به کاهش نویز کمک می‌کند.

- پاکسازی عناصر شبکه‌های اجتماعی:

با استفاده از عبارات منظم (Regular Expressions) ، نام‌های کاربری (مانند @username) ، لینک‌های اینترنتی (http یا http://@username) و علامت هشتگ (#) از ابتدای کلمات حذف شدند. متن هشتگ‌ها باقی ماند تا در مرحله بعد پردازش شوند. چرا که این عناصر معمولاً محتواهای معنایی یا احساسی مستقیمی ندارند (به جز خود کلمات هشتگ) و حذف آن‌ها نویز داده را کاهش می‌دهد.

- گسترش انقباضات (Contraction Expansion):

توضیح: یک دیکشنری از کلمات مخفف یا محاوره‌ای رایج فارسی (مانند "نمیدونم" ، "میشه" ، "خوبه") و شکل کامل یا رسمی آن‌ها (مانند "نمی‌دانم" ، "می‌شود" ، "خوب است") تعریف شد. سپس این موارد در متن جایگزین شدند. استانداردسازی کلمات محاوره‌ای به شکل رسمی‌تر، به کاهش اندازه واژگان و بهبود درک مدل کمک می‌کند.

- کاهش تکرار حروف:

توضیح: تکرارهای بیش از 3 بار یک حرف متوالی با 3 بار تکرار آن حرف جایگزین شد (مثلاً "سلاماً" به "سلام").

کاربران گاهی برای تأکید از تکرار حروف استفاده می‌کنند. در این مرحله ضمن حفظ بخشی از تأکید، نویز را کاهش می‌دهیم. از ایجاد توکن‌های بی‌معنی و افزایش بی‌رویه واژگان جلوگیری می‌کند. در روش TF-IDF ، تأکید یک ویژگی مجزاست؛ در روش Embedding ، یک تفاوت ظریف در بردار کلمه است که در متن پردازش می‌شود. (چون Embedding ثابت هست، مدل به FastText تکیه می‌کنه). FastText شاید با استفاده از بخش‌های کلمه (subwords) ، یک بردار کمی متفاوت برای "عالی" نسبت به "علی" بسازد.)

```
# Reduce character repetition (trim to 3 max)
text = re.sub(r'(.){3,}', r'\1\1\1', text)
```

- حذف کاراکترهای غیرفارسی:

تمام کاراکترهایی که در محدوده حروف الفبای فارسی (و عربی توسعه‌یافته برای فارسی) قرار ندارند، به جز فاصله و ارقام، از متن حذف شدند. این کار تضمین می‌کند که متن نهایی فقط شامل کاراکترهای زبان هدف باشد و از ورود نویز ناشی از

کاراکترهای زیان‌های دیگر یا نمادهای نامرتبط جلوگیری شود.

```
# Remove non-Persian characters
text = re.sub(r'[^\\u0600-\\u06FF\\u0750-\\u077F\\u08A0-\\u08FF\\u1EE00-\\u1EFF\\s\\d]', '', text)
```

- پردازش هشتگ‌ها

یک تابع جداگانه به نام `extract_hashtag_content()` برای پردازش محتواهی هشتگ‌ها تعریف شد. این تابع ابتدا با استفاده از عبارت منظم، متن هشتگ‌ها را از داخل براکت‌ها و کوتیشن‌ها استخراج می‌کند (مثلاً از "#شادی", "#ایران" متن "شادی ایران" را می‌سازد).

سپس، تابع `preprocess_persian_tweet()` که توضیح داده شد، روی این متن ترکیبی اعمال می‌شود تا هشتگ‌ها نیز مانند متن اصلی پاکسازی شوند. هشتگ‌ها معمولاً حاوی کلمات کلیدی مهمی هستند که موضوع یا احساس توییت را خلاصه می‌کنند. پردازش آن‌ها به غنی‌سازی اطلاعات ورودی مدل کمک می‌کند.

- اعمال پیش‌پردازش و ترکیب نهایی متن

توابع پیش‌پردازش روی ستون‌های `hashtags` و `tweet_text` اعمال شدند و نتایج در ستون‌های جدید `cleaned_hashtags` و `cleaned_tweet` ذخیره گردیدند.

سپس محتواهی این دو ستون پاکسازی شده با یک فاصله بینشان ترکیب شده و در ستون نهایی `full_text` قرار گرفت.

ایجاد یک ستون واحد شامل تمام اطلاعات متنی مرتبط (توییت + هشتگ)، فرآیند استخراج ویژگی و ورودی دادن به مدل‌ها را ساده‌تر می‌کند.

- اعتبارسنجی و فیلتر کردن داده‌ها

یک بررسی انجام شد تا اطمینان حاصل شود که توییت‌های پاکسازی شده معتبر هستند (مثلاً خالی نیستند، طول معقولی دارند). این کار با تابع `is_valid_cleaned_tweet()` انجام شد که چک می‌کرد متن رشته باشد، طول آن حداقل ۵ کاراکتر و حداقل شامل ۲ کلمه باشد و فقط از فاصله خالی تشکیل نشده باشد.

سطرهایی که این شرایط را نداشتند، حذف می‌شدند. در این پروژه، تعداد سطرها قبل و بعد از این فیلتر ثابت ماند. نشان می‌دهد که هیچ توییت نامعتبری پس از پاکسازی وجود نداشته است.

• حذف کلمات توقف(Stopwords)

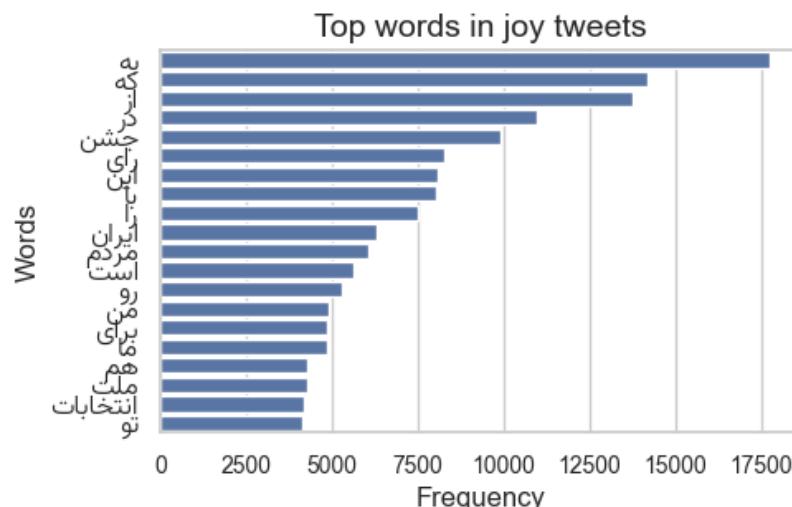
ایستوازه‌ها معمولاً شامل کلماتی مانند «از»، «به»، «برای»، «با»، «که»، «است»، «را» و... برخی هستند که غالباً در جمله به منزله ابزارهای نحوی به کار می‌روند. این کلمات: معنای ویژه یا اطلاعات احساسی خاصی را منتقل نمی‌کنند.

واژه‌های با اهمیت واقعی (مانند «شاد»، «ناراحت»، «غم»، «امید»، «جشن») در میان انبوهی از واژه‌های خنثی گم می‌شوند.

در همه‌ی جملات فارغ از نوع احساسات (هم مثبت، هم منفی، شادی، غم و...) به‌طور یکنواخت تکرار می‌شوند.

از آنجایی که واژگان مذکور در تمایز میان احساسات کمک شایانی نمی‌کنند، مدل یادگیری ماشین، به دلیل مشاهده مکرر این واژه‌ها در تمامی کلاس‌ها، دچار سردرگمی می‌گردد.

کتابخانه **WordCloud** که در زبان **Python** برای ایجاد ابرکلمات استفاده می‌شود، به صورت پیش‌فرض برای زبان انگلیسی و زبان‌های چپ به راست (LTR) طراحی شده است. به دلیل عدم پشتیبانی مناسب **WordCloud** از زبان فارسی و مشکلات اجرایی و معنایی آن از روش جایگزین نمودار میله‌ای از پرتکرارترین کلمات (**Bar Plot of Most Frequent Words**) - استفاده کردیم.



در خروجی دو نمودار مربوط به **joy** و **sad**، وجود واژه‌های بی‌ارزش یا بدون بار احساسی (مانند «است»، «شود»، «نیست») در میان پرتکرارترین کلمات نشان داد که فرایند حذف ایستوازه‌ها (stopwords) باید به دقیق و به صورت هدفمند انجام گیرد تا واژه‌های بی‌ارزش از مدل حذف شده و تمرکز روی واژه‌های با اهمیت بیشتر قرار گیرد. در همین راستا یا میتوان فایل مربوط به ایست واژه‌ها را در فرمت **txt** دستی ساخت یا مشابه آنرا از گیت هاب جستجو و استفاده کرد.

(منبع من برای ایست واژه ها:

(<https://github.com/ziaa/Persian-stopwords-collection.git>

```
stopwords = set(open("stopwords.txt", encoding="utf-8").read().splitlines())

Tabnine | Edit | Test | Explain | Document
def remove_stopwords(text):
    words = text.split()
    return " ".join([word for word in words if word not in stopwords])

df_processed['full_text'] = df_processed['full_text'].apply(remove_stopwords)
```

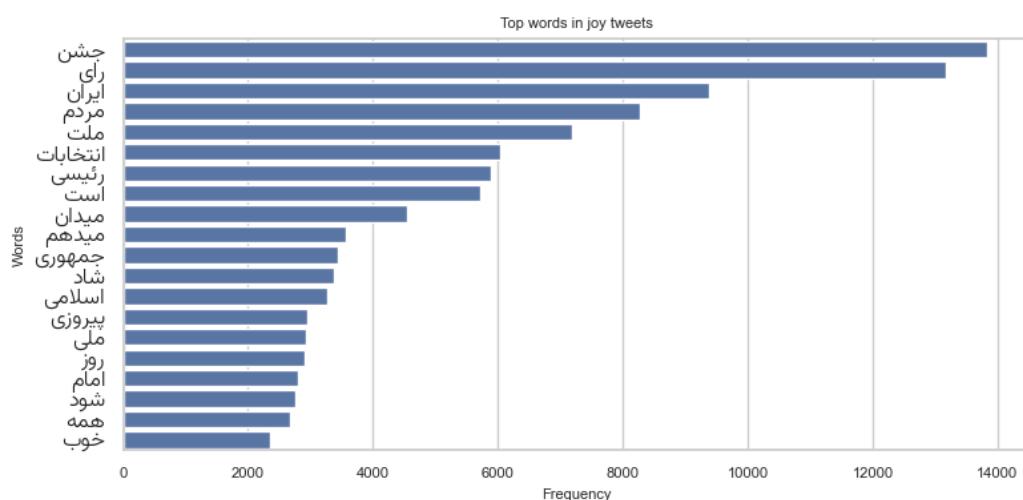
با استفاده از `open(...).read().splitlines()` تمام خطوط فایل خوانده شده و به صورت لیستی از کلمات درمی آید.

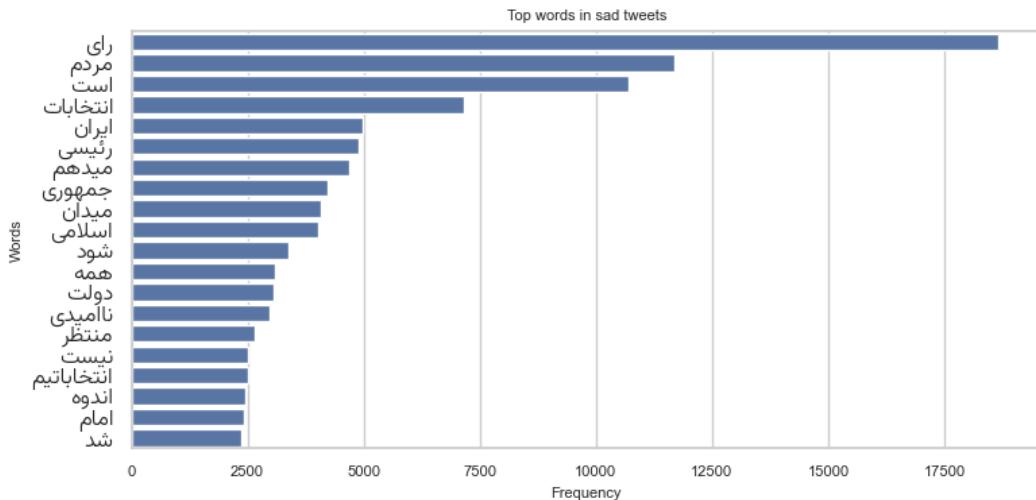
سپس با تبدیل آن به `set` عملیات جستجو برای هر کلمه سریع تر انجام می شود. در تابع حذف ، با `split()`. کلمات جدا می شوند و فقط آن دسته از (یا فقط آن لیستی از کلمات را که) در لیست ایست واژه ها وجود نداشته باشد را نگه می دارد. و این تابع به کل ستون متغیر ما اعمال می شود.

`df_processed['full_text']`

این کد به طور مؤثری باعث حذف کلمات بی ارزش از متن می شود و داده ها را برای مدل های یادگیری ماشین یا تحلیل احساسات آماده تر می کند. این کار باعث می شود مدل فقط روی کلمات معنی دار و احساسی تمرکز کند و عملکرد بهتری داشته باشد.

خروجی نمودار میله ای فرکانس کلمات پس از حذف ایست واژه ها :





کماکان نکته‌ی قابل توجه، همپوشانی کلمات بین دو کلاس است. کلماتی مانند «رأی»، «مردم»، «ایران»، «انتخابات»، «رئیسی»، «میدان»، «اسلامی»، «جمهوری»، «است»، «شود» هم در توییت‌های شاد و غمگین با فرکانس نسبتاً بالایی ظاهر می‌شوند. این کلمات خنثی و یا وابسته به متن هستند. ذاتاً مثبت یا منفی نیستند و برای تعیین احساس نیاز به بافت سطح جمله دارند.

نشانه‌های خاص هم در دو کلاس به چشم می‌خورد. برخی از شاخص‌های هیجانی خاص مانند :

شادی: «جشن»، «پیروزی»، «شاد»، «خوب»

غمگین: «اندوه»، «ناراضی»، «منتظر»

اما تعداد آنها کمتر است، به این معنی که مدل در نهایت، به جای کلیدواژه‌های مجزا، به متن زمینه تکیه می‌کند.

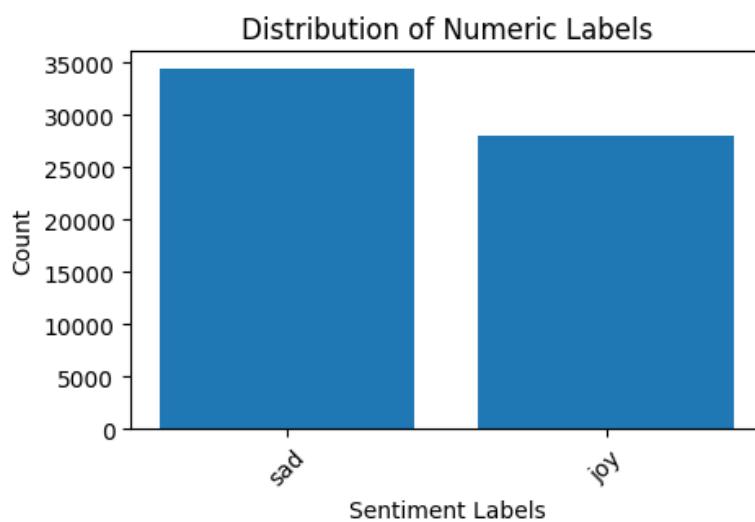
پس از اتمام این مراحل، دیتافریم پردازش شده آمده تحلیل اکتشافی و مدل‌سازی بود. این دیتافریم شامل 56865 رکورد بود که از این تعداد 31715 توییت با برچسب (1) 'sad' و 25150 توییت با برچسب (0) 'joy' بودند.

2. تحلیل اکتشافی داده‌ها (Exploratory Data Analysis - EDA)

EDA به فرآیند بررسی و بصری‌سازی داده‌ها برای درک بهتر الگوهای، ویژگی‌ها، ناهنجاری‌ها و روابط بین متغیرها اشاره دارد.

- توزیع برچسب‌های احساسی

یک نمودار میله‌ای برای نمایش تعداد نمونه‌ها در هر کلاس احساسی 'joy' و 'sad' رسم شد. این نمودار نشان داد که کلاس 'sad' (31715 نمونه) کمی پرتعدادتر از کلاس 'joy' (25150 نمونه) است، اما عدم توازن خیلی شدید نیست.

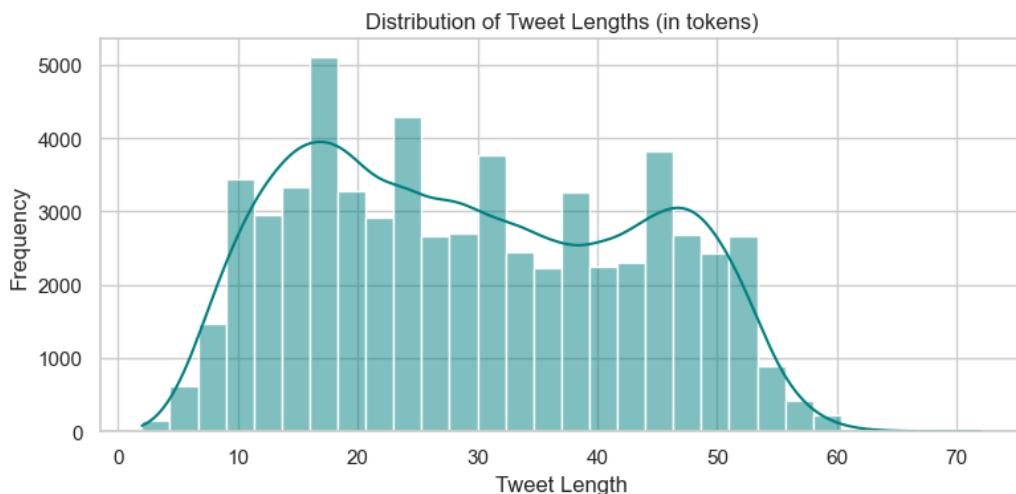


بررسی توزیع کلاس‌ها مهم است زیرا عدم توازن شدید می‌تواند بر عملکرد مدل‌ها (به خصوص در معیارهایی مانند Accuracy) تأثیر بگذارد و نیاز به تکنیک‌های خاصی مانند وزن‌دهی به کلاس‌ها یا نمونه‌برداری مجدد (resampling) را مطرح کند.

- توزیع طول توییت‌ها

طول هر توییت پاکسازی شده (تعداد توکن‌ها در cleaned_tweet) محاسبه و توزیع آن با استفاده از هیستوگرام نمایش داده شد. هیستوگرام نشان داد که بیشتر توییت‌ها طولی بین ۱۰ تا ۵۰ توکن

دارند.



درک توزیع طول متن‌ها به تصمیم‌گیری در مورد پارامترهایی مانند حداکثر طول دنباله (MAX_SEQUENCE_LENGTH) در مدل‌های یادگیری عمیق کمک می‌کند. انتخاب طول مناسب می‌تواند از حذف اطلاعات مهم (اگر طول خیلی کوتاه باشد) یا افزایش بی‌مورد محاسبات (اگر طول خیلی بلند باشد) جلوگیری کند.

• کلمات پرتکرار در هر کلاس

برای هر کلاس احساسی 'sad' و 'joy'، متن‌های full_text مربوط به آن کلاس استخراج و تمام کلمات آن‌ها شمارش شدند. سپس ۲۰ کلمه پرتکرار برای هر کلاس شناسایی و فراوانی آن‌ها با نمودار میله‌ای نمایش داده شد. برای نمایش صحیح کلمات فارسی در نمودارها از کتابخانه‌های bidi.algorithm و arabic_reshaper استفاده شد.

این تحلیل به شناسایی کلماتی که به‌طور مشخص با هر یک از احساسات مرتبط هستند کمک می‌کند و دید اولیه‌ای نسبت به ویژگی‌های متایزکننده هر کلاس ارائه می‌دهد (به این قسمت در توضیح ایست واژه‌ها یا stopwords، پرداخته شد).

3. آماده‌سازی داده‌ها برای مدل‌های یادگیری عمیق

مدل‌های یادگیری عمیق، بهویژه شبکه‌های عصبی بازگشته (RNNs) مانند LSTM، برای پردازش متن بسیار قدرتمند هستند اما به ورودی‌های عددی با فرمت خاص نیاز دارند.

- توکنائزیشن و پدینگ (Tokenization and Padding) : Keras Tokenizer توکنائزیشن پا

در این مرحله، از **Tokenizer** کتابخانه Keras برای تبدیل کلمات موجود در ستون `full_text` به دنباله‌ای از اعداد صحیح استفاده شد. هر کلمه منحصر به فرد در کل دیتاست یک شناسه عددی (اندیس) دریافت می‌کند.

پارامترها :

```
MAX_NUM_WORDS = 20000 # Max vocabulary size  
MAX_SEQUENCE_LENGTH = 100 # Max tweet length after padding
```

MAX_NUM_WORDS حداقل تعداد کلماتی که در واژگان مدل در نظر گرفته می‌شوند (این اساساً فرآوینه). این کار به کنترل اندازه مدل و حذف کلمات بسیار نادر کمک می‌کند.

```
# Tokenize text
tokenizer = Tokenizer(num_words=MAX_NUM_WORDS, oov_token=<OOV>)
tokenizer.fit_on_texts(df_processed['full_text'])
sequences = tokenizer.texts_to_sequences(df_processed['full_text'])
```

ooe_token="<OOV>" : یک توکن خاص یا جایگزین برای کلماتی که در طول آموزش دیده نشده‌اند یا حاوی **MAX NUM WORDS** کلمه پر تکرار نبوده‌اند.

وازگان پر اساس متون full text ساخته می شود: tokenizer.fit on texts

`tokenizer.texts_to_sequences`: هر توییت به دنباله‌ای از اندیس‌های عددی متضاد با کلامش تبدیل موشود.

- ## • پڈینگ (Padding)

مدل‌های RNN معمولاً به دنباله‌های ورودی با طول یکسان نیاز دارند. از آنجایی که طول توابعیت‌ها متفاوت است، از تابع `pad_sequences` Keras برای یکسان‌سازی طول آن‌ها استفاده شد.

```
X = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH, padding='post')
```

می‌شوند. دنباله‌های کوتاه‌تر با صفر (یا مقدار پدینگ دیگر) تکمیل می‌شوند و دنباله‌های بلندتر بریده می‌شوند.

'padding='post' : مقادیر پدینگ (صفرهای) به انتهای دنباله اضافه می‌شوند. (گزینه دیگر 'است که به ابتداء اضافه می‌کند.)

نتیجه نهایی (X) یک ماتریس دو بعدی است که هر سطر آن یک توییت به صورت دنباله‌ای از ۱۰۰ عدد صحیح (اندیس کلمات یا صفر برای پدینگ) است. ابعاد این ماتریس (100, 62352) بود. بردار هدف (y) نیز آرایه‌ای از پرچسب‌های عددی (0 یا 1) است .

این مراحل متن را به فرمت عددی و ساختار یافته‌ای تبدیل می‌کنند که برای ورودی دادن به لایه‌های Keras در RNN و Embedding ضروری است. محدود کردن واژگان و طول دنباله به مدیریت منابع محاسباتی کمک می‌کند.

- استفاده از بردارهای کلمه از پیش آموزش دیده (FastText)

مفهوم Word Embedding

تعریف: نمایش کلمات به صورت بردارهای عددی متراکم (Dense Vectors) در یک فضای چندبعدی. این بردارها به گونه‌ای یاد گرفته می‌شوند که کلمات با معانی مشابه یا روابط معنایی نزدیک، در این فضا به یکدیگر نزدیک‌تر باشند. این روش نسبت به روش‌های سنتی مانند One-Hot Encoding بسیار کارآمدتر است و روابط معنایی را بهتر ثبت می‌کند.

بازنمایی کلمات به صورت بردارهای عددی متراکم، موسوم به Embedding کلمات، یکی از پایه‌های اساسی در پسیاری از وظایف پردازش زبان طبیعی (NLP) مدرن، از جمله تحلیل احساسات، به شمار می‌رود. این بردارها قادرند ویژگی‌های معنایی و نحوی کلمات را در یک فضای برداری چندبعدی ثبت کنند. انتخاب مدل Embedding مناسب، تأثیر بسزایی در عملکرد نهایی سیستم‌های NLP دارد و به عواملی نظریزبان متن، حجم داده‌های موجود، و ویژگی‌های خاص زبان مورد بررسی (مانند غنای مورفو‌لوژیکی) بستگی دارد. در پروژه حاضر که به تحلیل احساسات توییت‌های فارسی می‌پردازد، علیرغم اینکه پروژه مرجع از مدل GloVe برای متون انگلیسی بهره برده است، تصمیم بر استفاده از مدل FastText اتخاذ گردید.

ایده اصلی GloVe بر این اساس استوار است که نسبت‌های آماری هم‌خدادی (co-occurrence) کلمات در یک پیکره زبانی بزرگ، می‌تواند اطلاعات معنایی مفیدی را در خود نهفته داشته باشد .

GloVe با تجزیه ماتریس هم‌خدادی کلمات در سطح کلان (Global) ، سعی در یادگیری بردارهایی دارد که ضرب داخلی آن‌ها با لگاریتم احتمال هم‌خدادی کلمات متاظر باشد (تابع هزینه‌ی (GloVe).

نقاط قوت GloVe شامل توانایی آن در ثبت شباهت‌های معنایی و قیاس‌های خطی (مانند "شاه - مرد + زن = ملکه") و همچنین وجود بردارهای از پیش آموزش دیده با کیفیت بالا برای زبان انگلیسی بر

روی پیکرهای عظیم متنی است. این ویژگی‌ها، **GloVe** را به گزینه‌ای رایج برای بسیاری از پروژه‌های NLP انگلیسی‌زبان تبدیل کرده است.

با وجود کارایی **GloVe** برای زبان انگلیسی، چندین عامل کلیدی منجر به انتخاب **FastText** به عنوان مدل Embedding در پروژه حاضر برای تحلیل احساسات توییت‌های فارسی گردید:

✓ محدودیت‌های زبانی و در دسترس بودن مدل‌های از پیش آموزش‌دیده:

یکی از دلایل اصلی این انتخاب، تفاوت در زبان پروژه (فارسی) نسبت به پروژه مرجع (انگلیسی) است. در حالی که مدل‌های **GloVe** از پیش آموزش‌دیده با کیفیت و پوشش بالا برای زبان انگلیسی به وفور یافت می‌شوند، برای زبان فارسی، گستردنگی و کیفیت مدل‌های **GloVe** از پیش آموزش‌دیده به اندازه مدل‌های **FastText** نیست.

✓ ماهیت مدل **FastText** و بهره‌گیری از اطلاعات زیرکلمه‌ای (**Subword Information**):

مهمترین مزیت **FastText**، به ویژه برای زبان‌های با مورفولوژی غنی مانند فارسی، توانایی آن در یادگیری و استفاده از اطلاعات زیرکلمه‌ای است **FastText**. هر کلمه را به عنوان مجموعه‌ای از نویسه سه‌تایی‌ها (character n-grams) در نظر می‌گیرد و بردار نهایی کلمه از مجموع بردارهای این زیرکلمه‌ها به دست می‌آید. این رویکرد چندین مزیت عمدی دارد:

مدیریت کلمات خارج از واژگان (Out-of-Vocabulary - OOV) : زبان فارسی دارای ساختار واژگانی پیچیده و تعداد زیادی کلمات مشتق و مرکب است. **FastText** به دلیل تکیه بر زیرکلمه‌ها، می‌تواند برای کلمات OOV نیز بردار معناداری تولید کند، چرا که احتمال وجود زیرکلمه‌های آن کلمه در داده‌های آموزشی بالاست.

ثبت شباهت‌های مورفولوژیکی: کلماتی که دارای ریشه یا وندهای مشترک هستند، در سطح زیرکلمه‌ای شباهت خواهند داشت. **FastText** با بهره‌گیری از این ویژگی، می‌تواند بردارهایی تولید کند که این قرابتهای مورفولوژیکی را منعکس می‌کنند، امری که برای درک بهتر معنا در زبان فارسی حائز اهمیت است.

✓ سازگاری با ماهیت متون شبکه‌های اجتماعی:

متون موجود در شبکه‌های اجتماعی، از جمله توییت‌ها، غالباً حاوی زبان محاوره‌ای، کلمات نوظهور، اختصارات، و غلط‌های املایی هستند. رویکرد مبتنی بر زیرکلمه در **FastText**، آن را نسبت به اینگونه تغییرات و نویزها مقاوم‌تر می‌سازد و امکان تولید بردارهای با کیفیت‌تر را برای چنین متونی فراهم می‌کند. این ویژگی برای تحلیل احساسات توییت‌های فارسی، که هدف اصلی این پروژه است، بسیار ارزشمند است.

✓ عملکرد و نتایج تجربی :

انتخاب نهایی بین مدل‌های Embedding بر اساس عملکرد تجربی آن‌ها بر روی یک وظیفه و مجموعه داده خاص صورت می‌گیرد. با توجه به ویژگی‌های زبان فارسی و ماهیت داده‌های مورد استفاده، انتظار می‌رود FastText به دلیل مزایای ذکر شده، عملکرد مناسبی در تولید بردارهای کلمات برای این پروژه ارائه دهد، همانطور که پوشش بالای واژگان (بیش از ۹۰٪) توسط مدل FastText استفاده شده در این پروژه نیز این امر را تایید می‌کند.

```
Words in tokenizer vocabulary (up to num_words): 19999  
Words found in FastText embedding: 19156  
FastText vocabulary coverage: 95.78%
```

در حالی که مدل GloVe یک مدل Embedding قدرتمند و پرکاربرد، به ویژه برای زبان انگلیسی است، انتخاب FastText برای پروژه تحلیل احساسات توییت‌های فارسی تصمیمی مبتنی بر ملاحظات عملی و نظری بوده است. در دسترس بودن مدل‌های از پیش آموزش‌دیده با کیفیت برای زبان فارسی، توانایی FastText در مدیریت کلمات خارج از واژگان و بهره‌گیری از اطلاعات زیرکلمه‌ای برای زبان‌های غنی از نظر مورفولوژی مانند فارسی، و سازگاری بهتر آن با ماهیت متون غیررسمی و پرنویز شبکه‌های اجتماعی، دلایل اصلی این جایگزینی بوده‌اند. این انتخاب به منظور افزایش دقت و کارایی سیستم تحلیل احساسات در بستر زبان فارسی صورت گرفته است.

• بارگذاری بردارهای FastText و ساخت ماتریس embedding

بردارهای FastText از پیش آموزش‌دیده برای زبان فارسی از فایل cc.fa.300.vec بارگذاری شدند. این فایل حاوی بردارهای 300 بعدی (EMBEDDING_DIM = 300) برای تعداد زیادی از کلمات فارسی است.

یک دیکشنری (dictionary) خالی به نام embedding_index ایجاد می‌شود. این دیکشنری برای ذخیره کلمات و بردارهای متناظر با آن‌ها استفاده خواهد شد. کلیدهای این دیکشنری، کلمات خواهد بود و مقادیر (values) متناظر با هر کلید، بردار آن کلمه خواهد بود.

```

EMBEDDING_DIM = 300
embedding_index = {}

with open('cc.fa.300.vec', 'r', encoding='utf-8') as f:
    next(f) # skip header
    for line in f:
        values = line.rstrip().split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embedding_index[word] = coefs

```

در نهایت، کلمه (word) به عنوان کلید و بردار عددی متناظر با آن (coefs) به عنوان مقدار، در دیکشنری embedding_index ذخیره می‌شوند. یک دیکشنری خواهد بود که هر کلمه موجود در فایل cc.fa.300.vec را به بردار 300 بعدی آن نگاشت می‌کند.

ایجاد ماتریس Embedding :

توضیح: یک ماتریس (آرایه Numpy) به ابعاد (num_words, EMBEDDING_DIM) ایجاد شد، که تعداد کلمات در واژگان tokenizer (حداکثر FastText (300)، بعد پردارهای EMBEDDING_DIM (MAX_NUM_WORDS است .

این ماتریس به این صورت پر شد: برای هر کلمه در واژگان tokenizer اگر آن کلمه در پردارهای FastText بارگذاری شده وجود داشت، سطر-یام ماتریس برابر با پردار آن کلمه قرار داده شد. اگر کلمه‌ای در FastText نبود، سطر متناظر آن در ماتریس صفر باقی ماند.

این ماتریس به عنوان وزن‌های اولیه و ثابت (زیرا trainable=False در لایه Embedding تنظیم می‌شود) برای لایه Embedding در مدل‌های Keras استفاده می‌شود. این کار به مدل اجازه می‌دهد تا از دانش معنایی که FastText از حجم عظیمی از داده‌های فارسی یاد گرفته، بهره‌مند شود، که معمولاً منجر به همگرایی سریع‌تر و عملکرد بهتر مدل می‌شود، به خصوص زمانی که داده‌های آموزشی محدود است.

4. مدل‌سازی با شبکه‌ی عصبی

پس از آماده‌سازی داده‌ها، مدل یادگیری عمیق برای طبقه‌بندی احساسات ساخته و آموزش داده می‌شود.

- تقسیم داده‌ها به مجموعه‌های آموزشی و آزمون

داده‌های پدشده (X) و برجسب‌های عددی (y) با استفاده از تابع `train_test_split` از کتابخانه **Scikit-learn** به دو بخش آموزشی (80%) و آزمون (20%) تقسیم شدند.

: نسبت داده‌هایی که برای آزمون کنار گذاشته می‌شوند.

برای اطمینان از اینکه داده‌های آموزشی و آزمایشی شما در هر بار اجرای کد به یک شکل تقسیم می‌شوند.

پارامتر `random_state` (که گاهی `seed` نامیده می‌شود) برای مقداردهی اولیه به تولیدکننده اعداد تصادفی (**Random Number Generator**) داخلی این الگوریتم‌ها استفاده می‌شود (جلوی در مدل `decision tree` هم به همین منظور استفاده شده است).

این پارامتر تضمین می‌کند که نسبت نمونه‌های هر کلاس ("joy") و ("sad") در هر دو مجموعه آموزشی و آزمون یکسان باشد. این امر در ارزیابی منصفانه مدل، بهویژه در دیتاست‌های کمی نامتوازن، مهم است.

مدل روی داده‌های آموزشی یاد می‌گیرد و عملکرد نهایی آن روی داده‌های آزمون (که هرگز در طول آموزش ندیده) سنجیده می‌شود تا تخمین بی‌طرفانه‌ای از توانایی تعمیم مدل به داده‌های جدید به دست آید.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
```

حصول اطمینان از صحت و همخوانی نوع دادگان (**data type**) ورودی:

```
print(f"X_train dtype: {X_train.dtype}")
print(f"y_train dtype: {y_train.dtype}")
```

```
X_train dtype: int32
y_train dtype: int32
```

LSTM (Long Short-Term Memory) .5. مدل

مقدمه‌ای بر RNN و LSTM :

RNN شبکه‌های عصبی بازگشتی (Recurrent Neural Networks) برای پردازش داده‌های ترتیبی (مانند متن یا سری‌های زمانی) طراحی شده‌اند. آن‌ها دارای حلقه‌های بازگشتی هستند که به اطلاعات مراحل قبلی اجازه می‌دهد در پردازش مرحله فعلی تأثیر بگذارند. اما RNN های ساده از مشکل "محوشدنگی گرادیان (Vanishing Gradient)" رنج می‌برند که یادگیری وابستگی‌های بلندمدت در دنباله‌ها را دشوار می‌کند.

- مشکل محو شدنگی گرادیان (Vanishing Gradient Problem) : در دنباله‌های بلند، هنگامی که گرادیان از لایه‌های انتهایی شبکه به سمت لایه‌های ابتدایی پس‌انتشار می‌یابد، ممکن است در اثر ضرب‌های مکرر در مقادیر کوچک (به‌ویژه مشتقات توابع فعال‌سازی مانند سیگموئید یا تانژانت هیپرboleیک که در بازه $(0, 1)$ یا $(1, 1)$ قرار دارند)، مقدار گرادیان به شدت کوچک شده و تقریباً به صفر میل کند. این امر باعث می‌شود که وزن‌های لایه‌های نزدیکتر به ورودی دنباله (یا به عبارت دیگر، وزن‌هایی که مسئول یادگیری وابستگی‌های طولانی‌مدت هستند) به روزرسانی بسیار کندی داشته باشند یا اصلًا بهروز نشوند. در نتیجه، شبکه در یادگیری و به خاطر سپردن اطلاعات از مراحل بسیار دور در دنباله دچار مشکل می‌شود.

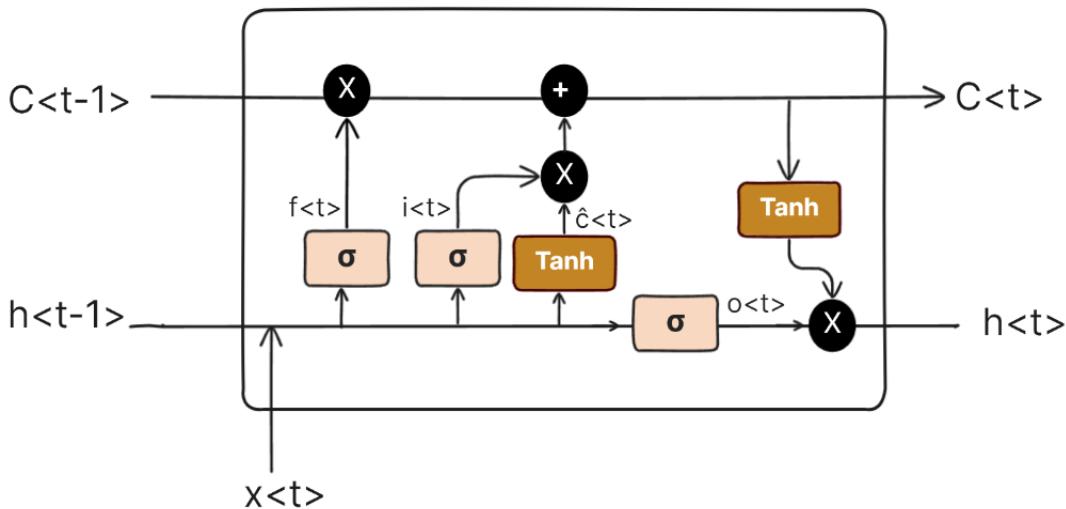
- مشکل انفجار گرادیان (Exploding Gradient Problem) : به طور مشابه، در برخی موارد، گرادیان‌ها در طول پس‌انتشار می‌توانند در اثر ضرب‌های مکرر در مقادیر بزرگ، به شدت بزرگ شده و منجر به پدیده انفجار گرادیان شوند. این امر باعث به روزرسانی‌های بسیار بزرگ و ناگهانی در وزن‌ها شده و فرآیند آموزش را ناپایدار می‌کند و ممکن است مدل از همگرایی باز بماند.

این دو مشکل، توانایی RNN های ساده را در مدل‌سازی وابستگی‌های طولانی‌مدت که در بسیاری از وظایف پردازش زبان طبیعی (مانند تحلیل احساسات در جملات طولانی یا اسناد) حیاتی هستند، به شدت محدود می‌کنند.

برای غلبه بر چالش‌های ذکر شده در RNN های ساده، به‌ویژه مشکل محو شدنگی گرادیان، نوع خاصی از معماری RNN به نام شبکه‌های حافظه طولانی کوتاه‌مدت (Long Short-Term Memory) یا LSTM (Long Short-Term Memory) توسط ها chreiter و اشمیت‌هوبر در سال 1997 معرفی گردید. LSTM ها با معرفی یک ساختار پیچیده‌تر در واحد بازگشتی خود، قادرند اطلاعات را برای مدت زمان طولانی‌تری حفظ کرده و جریان اطلاعات را به طور موثرتری کنترل کنند.

نوآوری اصلی در LSTM ها، استفاده از یک واحد حافظه به نام "وضعیت سلول (cell state)" و مکانیزم‌های کنترلی موسوم به "گیت (gate)" است. این گیت‌ها تصمیم می‌گیرند که چه اطلاعاتی باید به وضعیت سلول اضافه شود، چه اطلاعاتی باید از آن حذف گردد، و چه بخشی از وضعیت سلول باید به عنوان خروجی در گام زمانی فعلی استفاده شود.

LSTM Architecture



- تشریح ساختار داخلی یک واحد (LSTM Cell Structure)

نوع خاصی از RNN است که برای غلبه بر مشکل محوش‌گی گرادیان طراحی شده است. LSTM ها دارای ساختار پیچیده‌تری به نام "سلول حافظه (Memory Cell)" و سه "گیت" (Gate) هستند: گیت ورودی (Input Gate)، گیت فراموشی (Forget Gate) و گیت خروجی (Output Gate).

این گیت‌ها به شبکه اجازه می‌دهند تا به صورت انتخابی تصمیم بگیرد کدام اطلاعات را به سلول حافظه اضافه کند، کدام اطلاعات قدیمی را فراموش کند و کدام بخش از اطلاعات سلول حافظه را به عنوان خروجی مرحله فعلی و ورودی مرحله بعد ارسال کند. این مکانیسم به LSTM امکان می‌دهد تا وابستگی‌های طولانی‌مدت در داده‌های ترتیبی را به خوبی یاد بگیرد. هر واحد LSTM از چندین مولفه کلیدی تشکیل شده است که با یکدیگر در تعامل هستند تا جریان اطلاعات را مدیریت کنند.

وضعیت سلول (C_t) : این بخش را می‌توان به عنوان حافظه اصلی واحد LSTM در نظر گرفت. وضعیت سلول در طول دنباله حرکت می‌کند و اطلاعات می‌تواند با حداقل تغییرات از آن عبور کند. این ویژگی کلیدی به LSTM کمک می‌کند تا از مشکل محو شدن گرadian جلوگیری کند، زیرا گرادیان‌ها می‌توانند بدون کاهش شدید در طول وضعیت سلول منتشر شوند.

حالت پنهان (h_t) : این همان خروجی واحد LSTM در گام زمانی t است که به گام زمانی بعدی و همچنین برای پیش‌بینی نهایی (در صورت لزوم) ارسال می‌شود.

mekanizm‌های گیت (Gating Mechanisms) : سه نوع گیت اصلی در LSTM وجود دارد که هر کدام یک شبکه عصبی کوچک (معمولًاً یک لایه تمامًاً متصل باتابع فعال‌سازی سیگموئید) هستند. خروجی این گیت‌ها مقادیری بین 0 و 1 است که نشان می‌دهد چه مقدار از اطلاعات باید عبور کند (1 به معنای عبور کامل و 0 به معنای عدم عبور). ورودی این گیت‌ها معمولاً حالت پنهان قبلی (h_{t-1}) و ورودی فعلی (x_t) است.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

گیت فراموشی (ft - Forget Gate) :

هدف: تصمیم‌گیری در مورد اینکه چه اطلاعاتی از وضعیت سلول قبلی (C_{t-1}) باید دور ریخته شود. عملکرد: با توجه به x_t, h_{t-1} و ft، یک بردار از مقادیر بین 0 و 1 تولید می‌کند. اگر مقدار متناظر با یک بخش از C_{t-1} برابر 0 باشد، آن بخش فراموش می‌شود و اگر 1 باشد، حفظ می‌شود.

گیت ورودی (it - Input Gate) :

هدف: تصمیم‌گیری در مورد اینکه چه اطلاعات جدیدی باید در وضعیت سلول (C_t) ذخیره شود. این گیت از دو بخش تشکیل شده است:

یک لایه سیگموئید (خود گیت ورودی) که تصمیم می‌گیرد کدام مقادیر از اطلاعات جدید باید به روزرسانی شوند.

یک لایه باتابع فعال‌سازی تانژانت هیپربولیک (tanh) که یک بردار از مقادیر کاندیدای جدید (C_t) را ایجاد می‌کند که می‌تواند به وضعیت سلول اضافه شوند.

گیت خروجی (ot - Output Gate) :

هدف: تصمیم‌گیری در مورد اینکه چه بخشی از وضعیت سلول فعلی (C_t) باید به عنوان حالت پنهان (h_t) و خروجی واحد LSTM در گام زمانی فعلی، ارائه شود.

عملکرد: ابتدا وضعیت سلول C_t از یک تابع tanh عبور داده می‌شود تا مقادیر آن در بازه [-1, 1] قرار گیرند. سپس، خروجی یک لایه سیگموئید تعیین می‌کند که کدام بخش از این مقادیر باید به عنوان خروجی نهایی (h_t) انتخاب شوند.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

به روزرسانی وضعیت سلول و حالت پنهان: با استفاده از گیت‌های فوق، وضعیت سلول و حالت پنهان به صورت زیر به روز می‌شوند:

Cell State Update (C_t): The old cell state C_{t-1} is updated into the new cell state C_t.

$$C_t = f_t \odot C_{t-1} + i_t \odot C_{\sim t}$$

ابتدا، اطلاعاتی که باید فراموش شوند از وضعیت سلول قبلی حذف می‌شوند.

سپس، اطلاعات جدیدی که باید اضافه شوند، تولید و به آن افزوده می‌شوند.

در نهایت، حالت پنهان جدید بر اساس وضعیت سلول پهروز شده و گیت خروجی محاسبه می‌شود:

$$h_t = o_t \odot \tanh(C_t)$$

در فرمول‌های بالا، W‌ها ماتریس‌های وزن و b‌ها بایاس‌های مربوط به هر گیت هستند که در طول فرآیند آموزش یاد گرفته می‌شوند. σ تابع سیگموئید و tanh تابع تائزانت هیپربولیک است.

این معماری دقیق و کنترل شده به LSTM‌ها اجازه می‌دهد تا وابستگی‌ها را در دنباله‌های بسیار طولانی‌تری نسبت به RNN‌های ساده یاد بگیرند و حفظ کنند، که آن‌ها را به ابزاری قدرتمند در بسیاری از کاربردهای پیشرفته پردازش زبان طبیعی، از جمله تحلیل احساسات متون فارسی در این پژوهش، تبدیل کرده است.

• ساخت مدل LSTM :

پیش از پرداختن به نسخه‌های مختلف مدل LSTM و آزمایش‌های تنظیم هایپرپارامترها، در این بخش معماری پایه‌ای که به عنوان نقطه شروع برای مدل‌سازی در نظر گرفته شد، تشریح می‌گردد. این چارچوب اولیه شامل لایه‌های بنیادینی است که در اکثر مدل‌های LSTM مورد استفاده در این پژوهش مشترک بوده‌اند:

ورودی اصلی مدل‌های یادگیری عمیق برای پردازش متن، دنباله‌ای از شناسه‌های عددی است که هر شناسه متناظر با یک توکن (کلمه) در واژگان پروژه است. لایه Embedding وظیفه تبدیل این شناسه‌های عددی به بردارهای متر acum و با معنی (بردارهای کلمه) را بر عهده دارد. در این پژوهش:

این لایه با استفاده از ماتریس Embedding از پیش محاسبه شده از بردارهای FastText (که در بخش های قبلی به تفصیل نحوه ساخت آن توضیح داده شد) مقداردهی اولیه گردید.

ابعاد هر بردار کلمه (EMBEDDING_DIM) برابر با 300 در نظر گرفته شد که مطابق با ابعاد بردارهای استفاده شده FastText می باشد.

حداکثر طول دنباله های ورودی به این لایه (MAX_SEQUENCE_LENGTH) برابر با 100 توکن تنظیم گردید. دنباله های کوتاهتر با صفر پدگذاری شده و دنباله های بلندتر کوتاه می شوند.

اندازه واژگان ورودی به این لایه (input_dim) بر اساس پارامتر MAX_NUM_WORDS که در پروژه 20000 در نظر گرفته شده) و تعداد کلمات منحصر به فرد شناسایی شده توسط توکنایزر تعیین می گردد.

یک پارامتر کلیدی در این لایه trainable است که در اکثر آزمایش های اولیه بر روی False تنظیم شد. این تصمیم به این معناست که بردارهای از پیش آموزش دیده FastText در طول فرآیند آموزش مدل آب روزسانی نمی شوند و مدل از دانش ثابت آن ها بهره می برد. این رویکرد با توجه به کیفیت و پوشش بالای بردارهای FastText برای زبان فارسی اتخاذ شد.

پس از تبدیل دنباله های ورودی به دنباله هایی از بردارهای کلمه توسط لایه Embedding ، این بردارها به یک یا چند لایه LSTM وارد می شوند. لایه LSTM ، به عنوان هسته اصلی مدل برای پردازش اطلاعات متوالی و یادگیری وابستگی های موجود در متن عمل می کند.

تعداد واحد های حافظه (یا نورون ها) در لایه LSTM یک هایپر پارامتر مهم است که در نسخه های مختلف مدل (مثلًا 32، 64 یا 128 واحد) مورد آزمایش قرار گرفت.

پارامتر return_sequences در این لایه تعیین می کند که آیا خروجی تمام گام های زمانی دنباله به لایه بعدی ارسال شود یا فقط خروجی آخرین گام زمانی (که خلاصه ای از کل دنباله را در بر دارد) به لایه های بعدی (معمولًا لایه های Dense) منتقل گردد. در معماری پایه مورد استفاده، زمانی که تنها یک لایه LSTM قبل از لایه های Dense قرار داشت، این پارامتر روی False تنظیم می شد.

به منظور کاهش خطر بیش برازش (overfitting) مدل بر روی داده های آموزشی، از لایه های Dropout استفاده شد. بیش برازش زمانی رخ می دهد که مدل به جای یادگیری الگوهای عمومی، جزئیات و نویز موجود در داده های آموزشی را حفظ می کند و در نتیجه عملکرد ضعیفی بر روی داده های جدید و دیده نشده خواهد داشت.

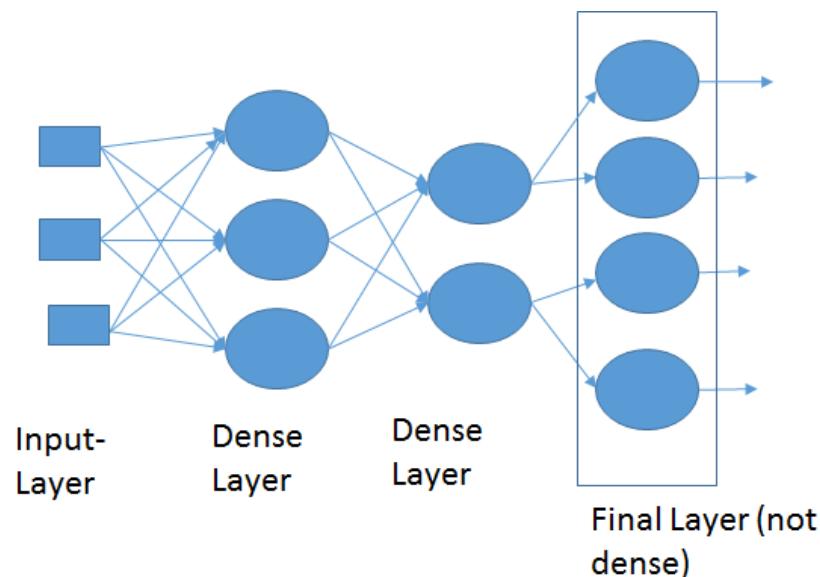
لایه Dropout در طول هر مرحله از آموزش، به طور تصادفی درصد مشخصی از نورون های لایه قبلی را (مثلًا 30% با نرخ 0.3) غیرفعال می کند. چون هر نورون ممکن است در هر مرحله از آموزش به طور تصادفی حذف شود، شبکه نمی تواند برای تشخیص یک ویژگی مهم، فقط به یک یا چند نورون خاص تکیه کند. اگر این کار را بکند و آن نورون (ها) حذف شود، مدل در آن مرحله عملکرد ضعیفی خواهد داشت. برای جبران حذف تصادفی نورون ها، شبکه یاد می گیرد که ویژگی های مشابه یا مکمل را توسط نورون های

مختلفی بازنمایی کند (مقاومت یا Robust) . همچنین ، هر نورون تشویق می شود تا ویژگی هایی را یاد بگیرد که به خودی خود مفید باشند و نه فقط در حضور یک نورون خاص دیگر Distributed یا توزیع شدگی) .

پس از آنکه لایه های LSTM ویژگی های سطح بالا را از دنباله ورودی استخراج کردند، از یک یا چند لایه تماماً متصل (Dense) برای انجام طبقه بندی نهایی استفاده می شود.

گاهی یک لایه Dense با تابع فعال‌سازی غیرخطی مانند ReLU (activation='relu') بین خروجی LSTM و لایه خروجی نهایی قرار می‌گیرد تا به مدل امکان یادگیری ترکیب‌های پیچیده‌تری از ویژگی‌ها را بدهد.

چاپگاه لایه‌ی Dense



لایه خروجی یا **output layer**, مسئول تولید پیش‌بینی نهایی برای کلاس احساسات است. معماری این لایه به نوع مسئله **classification** بستگی دارد:

برای طبقه‌بندی باینری دو کلاسه مانند 'joy' و 'sad' : معمولاً شامل یک نورون با تابع فعال‌سازی سیگموئید (activation='sigmoid') است. خروجی سیگموئید یک مقدار بین ۰ و ۱ است که می‌تواند به عنوان احتمال تعلق به کلاس مثبت (مثلاً کلاس ۱) تفسیر شود.

برای طبقه‌بندی چندکلاسه (که برای حالت باینری نیز قابل استفاده است) : شامل `num_classes` نورون (در اینجا 2 نورون) با تابع فعال‌سازی سافت‌مکس (`activation='softmax'`) است. سافت‌مکس یک توزیع احتمال بر روی تمام کلاس‌ها تولید می‌کند. در آزمایش‌های مختلف این پروژه، هر دو رویکرد برای لایه خروجی، مورد بررسی قرار گرفت.

کامپایل مدل (Model Compilation) :

پیش از شروع فرآیند آموزش، مدل باید کامپایل شود. در این مرحله، اجزای کلیدی زیر برای فرآیند یادگیری تعریف می‌شوند:

بهینه‌ساز (Optimizer) : الگوریتمی که بر اساستابع هزینه محاسبه شده، وزن‌های شبکه را بهروزرسانی می‌کند تا عملکرد مدل بهبود یابد. در این پروژه از بهینه‌ساز Adam استفاده شد که یک الگوریتم تطبیقی کارآمد و رایج است. نرخ یادگیری (learning rate) این بهینه‌ساز یک هایپرپارامتر پسیار مهم است که تأثیر قابل توجهی بر روند همگرایی و کیفیت نهایی مدل دارد.

تابع هزینه (Loss Function) : معیاری برای سنجش میزان خطای مدل در پیش‌بینی برچسب‌های صحیح. انتخاب این تابع به ساختار لایه خروجی بستگی دارد :

برای لایه خروجی با یک نورون سیگموئید، تابع هزینه binary_crossentropy مناسب است.
برای لایه خروجی با چند نورون سافت‌مکس و برچسب‌های عددی صحیح (integer labels) ، تابع هزینه sparse_categorical_crossentropy استفاده می‌شود.

معیارهای ارزیابی (Metrics) : برای نظارت بر عملکرد مدل در طول آموزش و ارزیابی نهایی آن به کار می‌روند. معیار اصلی مورد استفاده در این پروژه accuracy بود، هرچند معیارهای دیگری مانند F1-score نیز در تحلیل نهایی نتایج اهمیت بسزایی دارند.

محاسبه‌ی معیارهای استفاده شده:

$$\text{Accuracy} = \frac{(TP + TN)}{(TP + FP + TN + FN)}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

$$\text{Macro F1 Score} = \frac{\sum_{i=1}^n \text{F1 Score}_i}{n}$$

این معماری پایه، چارچوبی را برای آزمایش‌های بعدی و تنظیم دقیق هایپرپارامترها فراهم آورد که در بخش آتی به تفصیل آن می‌پردازیم.

- تابع ارزیابی:

```
def evaluate_model(model, X_test, y_test, labels, model_name="Model"):
    print(f"\n--- Evaluation: {model_name} ---")
    y_pred = model.predict(X_test)
    # اگر خروجی سیگموئید است، برای تبدیل به کلاس‌ها
    if y_pred.shape[-1] == 1: # خروجی سیگموئید
        y_pred_classes = (y_pred > 0.5).astype("int32").flatten()
    else: # خروجی سافت‌مکس
        y_pred_classes = y_pred.argmax(axis=1)

    acc = accuracy_score(y_test, y_pred_classes)
    f1 = f1_score(y_test, y_pred_classes, average='macro')
```

این تابع چهار پارامتر اصلی می‌گیرد:

- `model`: مدل آموزش‌دیده شده
- `X_test`: داده‌های ورودی تست
- `y_test`: برچسب‌های واقعی تست
- `labels`: لیستی از نام کلاس‌ها برای گزارش‌ها و confusion matrix
- `model_name`: نام اختیاری برای نمایش در خروجی‌ها

ابتدا مدل را روی داده‌های تست اجرا می‌کند(`predict`) سپس بسته به نوع خروجی:

اگر فقط یک مقدار در خروجی باشد (`shape[-1] == 1`) ، فرض می‌شود مدل یک طبقه‌بندی باینری است با خروجی سیگموئید . در این حالت، اگر خروجی بیشتر از ۰.۵ باشد، کلاس ۱ وگرنه کلاس ۰ .
 اگر چند مقدار در خروجی باشد مثل `softmax` ، بیشترین مقدار را پیدا کرده و اندیس آن را به عنوان کلاس نهایی در نظر می‌گیرد.
 ادامه‌ی کد تابع معیار‌ها را محاسبه و نمایش میدهد و `confusion` ماتریس را رسم می‌کند.

- تابع رسم منحنی یادگیری:

هدف اصلی این تابع این است که به شما نشان دهد عملکرد مدل (هم از نظر دقت و هم از نظر خطای) چگونه در طول اپاک‌های مختلف آموزش تغییر کرده است. این اطلاعات برای تشخیص مشکلاتی مانند بیش‌برازش (overfitting) یا کم‌برازش (underfitting) بسیار حیاتی هستند.

```

font_path = "ttf/Vazirmatn-Light.ttf"
try:
    persian_font_properties = fm.FontProperties(fname=font_path, size=12)
    print(f"فونت بارگذاری شد: {persian_font_properties.get_name()}")
except RuntimeError:
    print(f"فونت در مسیر '{font_path}' نیافرود")
    persian_font_properties = fm.FontProperties(size=12)

```

بلوک try...except : سعی می‌کند فونت مشخص شده را بارگذاری کند و آن را در متغیر persian_font_properties ذخیره کند. اگر فایل فونت پیدا نشود یا مشکلی در بارگذاری آن وجود داشته باشد (RuntimeError) ، یک پیغام خطأ چاپ کرده و از فونت پیشفرض استفاده می‌کند.

```

def prepare_persian_text_for_plot(text):
    reshaped_text = arabic_reshaper.reshape(text)
    bidi_text = get_display(reshaped_text)
    return bidi_text

```

این تابع یک رشته متنی فارسی (text) را به عنوان ورودی می‌گیرد.

برای پردازش توسط الگوریتم‌های راست به چپ آماده شوند. reshaped_text = arabic_reshaper.reshape(text) : حروف فارسی را به شکلی تغییر می‌دهد که

متن reshape شده را برای نمایش صحیح (راست به چپ) پردازش می‌کند.

در نهایت، متن آماده شده برای نمایش در نمودارها را بر می‌گرداند. این تابع اطمینان می‌دهد که عنوان‌ها، برچسب‌ها و متنهای راهنمای نمودار به هم ریخته نمایش داده نشوند.

```

Tabnine | Edit | Test | Explain | Document
def plot_learning_curves(history_object, model_name="مدل"):

    if not hasattr(history_object, 'history') or not isinstance(history_object.history, dict):
        print(prepare_persian_text_for_plot("نحوه دقت 'history' نیست یا فاقد کلید 'history' است"))
        return

    history_data = history_object.history

    plt.figure(figsize=(14, 6))

    # نمودار دقت
    plt.subplot(1, 2, 1)
    if 'accuracy' in history_data:
        plt.plot(history_data['accuracy'], label=prepare_persian_text_for_plot('دقت آموزش'))()
    if 'val_accuracy' in history_data:
        plt.plot(history_data['val_accuracy'], label=prepare_persian_text_for_plot('دقت ارزیابی'))()

```

این تابع اصلی است که نمودارها را رسم می‌کند.

: این همان شیئی است که متد `model.fit()` پس از اتمام آموزش برمی‌گرداند. این شیء شامل یک دیکشنری به نام `history` است که حاوی لیست‌هایی از مقادیر معیارها (مانند دقت و خطای خطا) برای هر اپیک است.

"`model_name` = مدل" : یک رشته اختیاری که برای نمایش نام مدل در عنوان نمودارها استفاده می‌شود. مقدار پیش‌فرض آن "مدل" است.

: if not hasattr(history_object, 'history') or not isinstance(history_object.history, dict)
بررسی می‌کند که آیا ورودی `history_object` واقعاً یک شیء تاریخچه معتبر است و دارای یک ویژگی به نام `history` از نوع دیکشنری است. اگر نباشد، پیغام خطای خطا چاپ کرده و از تابع خارج می‌شود.

: دیکشنری حاوی لیست مقادیر معیارها در طول اپیک‌ها را در متغیر `history_data = history_object.history` ذخیره می‌کند. این دیکشنری معمولاً شامل کلیدهایی مانند `'loss'`, `'accuracy'`, `'val_loss'`, `'val_accuracy'` است.

در ادامه این بخش از کد وظیفه دارد داده‌های دقت و خطای `accuracy - loss` در طول اپیک‌ها را از ورودی `history_data` بردارد و آن‌ها را به صورت دو نمودار جداگانه اما در یک پنجره، همراه با تمام جزئیات بصری لازم (عنوان‌ها، برچسب‌ها، راهنمایی، شبکه‌بندی و فونت فارسی) نمایش دهد.

6. آموزش مدل LSTM

۱. مدل LSTM ساده (Super Simple LSTM - Model 1)

این مدل به عنوان یک معماری پایه برای ارزیابی اولیه عملکرد LSTM بر روی مجموعه داده مورد نظر طراحی شده است.

معماری مدل:

لایه LSTM : یک لایه LSTM با ۳۲ واحد (units).
این لایه توالی بردارهای Embedding را به عنوان ورودی دریافت می‌کند. پارامتر `return_sequences=False` تنظیم شده است، به این معنی که تنها خروجی آخرین گام زمانی از توالی LSTM به لایه بعدی منتقل می‌شود. این رویکرد معمولاً زمانی استفاده می‌شود که هدف، طبقه‌بندی کل توالی (مانند یک جمله یا توییت) باشد.

لایه خروجی (Dense) : یک لایه تماماً متصل (Dense) با ۱ واحد و تابع فعال‌سازی sigmoid.

تابع سیگموئید خروجی را به بازه $(0, 1)$ نگاشت می‌دهد که برای طبقه‌بندی باینری (در اینجا، تشخیص احساسات "شادی" یا "غم") مناسب است. خروجی این لایه احتمال تعلق ورودی به کلاس مثبت (مثلاً "شادی") را نشان می‌دهد.

بهینه‌ساز (Optimizer) : از بهینه‌ساز Adam با نرخ یادگیری (learning rate) برابر با 0.001 استفاده شده است.

* نوع بهینه ساز در تمام مدل ها ثابت است. نرخ یادگیری آن ممکن است تغییر کند.

تابع هزینه (Loss Function) : تابع هزینه binary_crossentropy برای این مدل انتخاب شده است که برای مسائل طبقه‌بندی باینری با خروجی سیگموئید استاندارد است.

```
print("\n--- Defining Model 1: Simple LSTM ---")
Tabnine | Edit | Test | Explain | Document
def build_simple_lstm_model(embedding_matrix, max_sequence_length):
    model = Sequential()
    model.add(Embedding(input_dim=embedding_matrix.shape[0],
                         output_dim=embedding_matrix.shape[1],
                         weights=[embedding_matrix],
                         input_length=max_sequence_length,
                         trainable=False))
    model.add(LSTM(32, return_sequences=False))
    model.add(Dense(1, activation='sigmoid'))

    optimizer = Adam(learning_rate=0.001)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model

simple_lstm_model = build_simple_lstm_model(embedding_matrix, MAX_SEQUENCE_LENGTH)
```

ارزیابی مدل simple_lstm_model

اگر از validation_data = (X_test, y_test) استفاده کنیم. در اصل خودمان مجموعه اعتبارسنجی را جدا کرده و به Keras می‌دهیم فقط از آن استفاده می‌کند. اما اگر از validation_split استفاده کنیم، خودش بخشی از داده‌های آموزشی را برای اعتبارسنجی جدا می‌کند.

در پایان هر epoch ، مدل روی داده‌های validation که در آموزش دخالت ندارن، ارزیابی می‌شود:

: val_accuracy

درصد پیش‌بینی‌های درست مدل روی X_{val} نسبت به برچسب‌های واقعی y_{val}

$$\frac{\text{تعداد پیش‌بینی‌های درست روی validation}}{\text{تعداد کل داده‌های validation}} = \text{val_accuracy}$$

: val_loss

مقدار تابع هزینه (مثل cross-entropy loss) محاسبه شده روی داده‌های اعتبارسنجی.

Epoch 1/10 → accuracy: 0.5538, val_accuracy: 0.5577

...

Epoch 10/10 → accuracy: 0.5600, val_accuracy: 0.5577

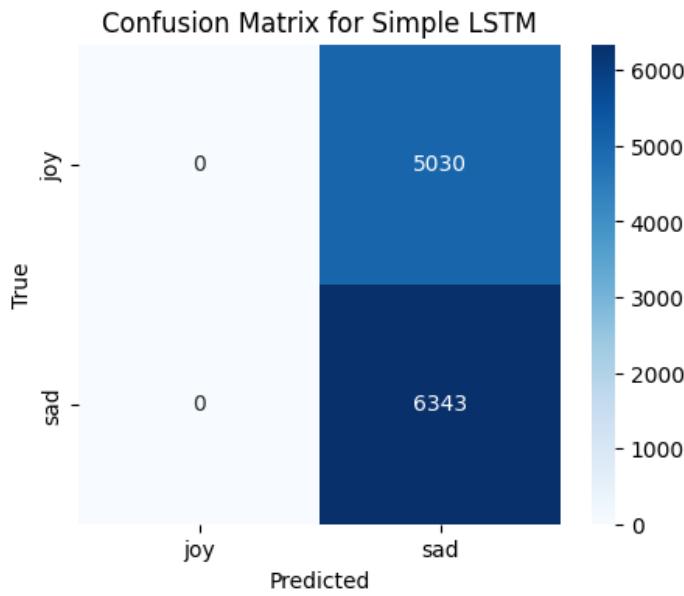
در طول آموزش، دقت آموزش (accuracy) و اعتبارسنجی (val_accuracy) در حدود 55% تا 65% نوسان داشت، اما بعد از epoch سوم به وضوح ایستا شد و پیشرفتی نداشت.

مدل هیچ وقت کلاس joy را پیش‌بینی نکرده. تمام خروجی‌ها کلاس sad بودن.
به همین دلیل:

precision و recall برای joy صفر شدن.

recall کلاس sad برابر ۱.۰۰ شده چون مدل همه رو sad پیش‌بینی کرده.

احتمال می‌رود مدل بیش از حد ساده‌ست. نرخ یادگیری بالایی به آن اختصاص داده شده که باعث شده مدل نتواند به درستی همگرا شود. شاید مشکل گرادیان محو شونده یا Vanishing Gradient با اینکه LSTM عملاً برای مقابله با این مشکل طراحی شده اما در معماری‌های بسیار ساده همچنان ممکن است یادگیری با مشکل مواجه شود.



: Model 1 : Simple LSTM .2

```
● Click to add a breakpoint in Model 1: Simple LSTM ---")
Labnne | Edit | test | Explain | Document
def build_simple_lstm_model(embedding_matrix, max_sequence_length):
    model = Sequential()
    model.add(Embedding(input_dim=embedding_matrix.shape[0],
                        output_dim=embedding_matrix.shape[1],
                        weights=[embedding_matrix],
                        input_length=max_sequence_length,
                        trainable=False))
    model.add(LSTM(64, return_sequences=False))
    model.add(Dense(1, activation='sigmoid'))

    optimizer = Adam(learning_rate=0.0002)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model

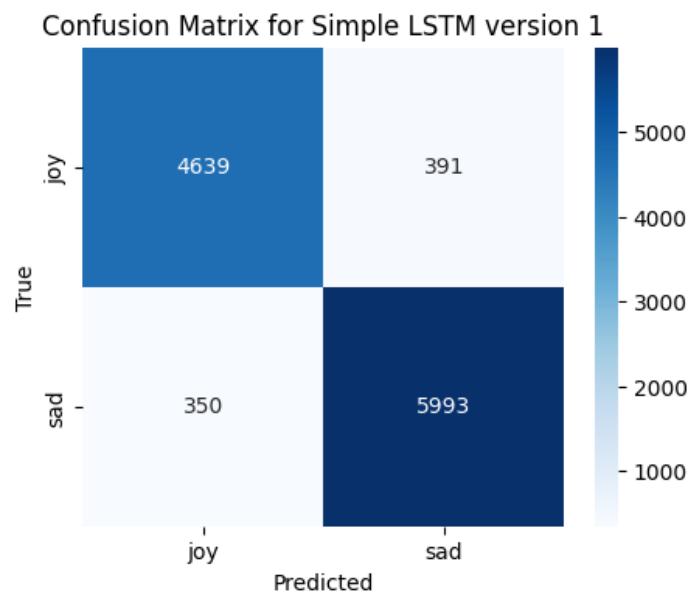
simple_lstm_model_V1 = build_simple_lstm_model(embedding_matrix, MAX_SEQUENCE_LENGTH)
```

تغییرات:

مدل با loss binary_crossentropy به عنوان تابع هزینه، بهینهساز Adam با نرخ یادگیری 0.0002 و معیار ارزیابی accuracy کامپایل می‌شود.

در ابتدا دقت روی داده آموزش حدود 63.6% است ولی خیلی سریع افزایش می‌یابد و در نهایت به 93% می‌رسد. دقت روی داده اعتبارسنجی هم از 84.34% شروع شده و تا حدود 92.31% می‌رسد. از آنجایی که دقت روی داده اعتبارسنجی (val_accuracy) همگام با دقت آموزش افزایش یافته است و خطای روی داده اعتبارسنجی (val_loss) هم کاهش یافته و هم پایین‌تر از خطای آموزش است، نشانه‌ای از بیش برآش دیده نمی‌شود.

پیشرفت ارزیابی مشهود است. به همین روال ادامه میدهیم.



مدل Simple LSTM با نرخ یادگیری = 0.0004

```

print("\n--- Defining Model 1: Simple LSTM ---")
Tabnine | Edit | Test | Explain | Document
✓ def build_simple_lstm_model(embedding_matrix, max_sequence_length):
    model = Sequential()
    ✓ model.add(Embedding(input_dim=embedding_matrix.shape[0],
                          output_dim=embedding_matrix.shape[1],
                          weights=[embedding_matrix],
                          input_length=max_sequence_length,
                          trainable=False))
    model.add(LSTM(64, return_sequences=False))
    model.add(Dense(1, activation='sigmoid'))

    optimizer = Adam(learning_rate=0.0004)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model

simple_lstm_model = build_simple_lstm_model(embedding_matrix, MAX_SEQUENCE_LENGTH)

```

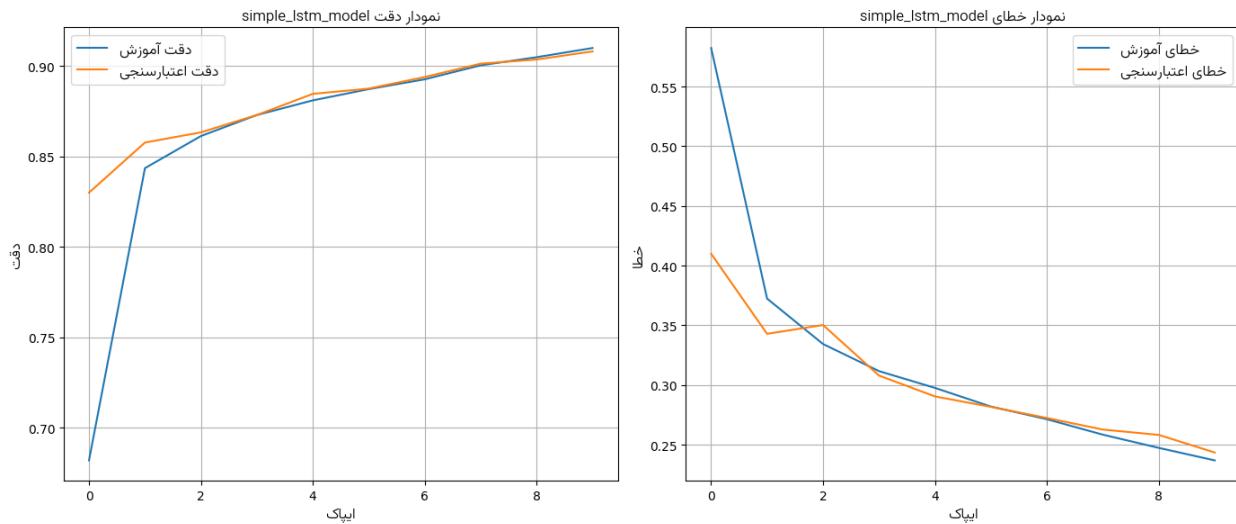
مقدار 0.0004 برای Adam نسبتاً کم است ، این کاهش به معنای یادگیری آهستهتر ولی پایدارتر است.

خروجی اپیک های آخر این مدل:

```

/11/711           85s 117ms/step - accuracy: 0.9408 - loss: 0.1880 - val_accuracy: 0.9570 - val_loss: 0.1804
Epoch 8/15
711/711          89s 125ms/step - accuracy: 0.9478 - loss: 0.1496 - val_accuracy: 0.9445 - val_loss: 0.1728
Epoch 9/15
711/711          86s 121ms/step - accuracy: 0.9533 - loss: 0.1323 - val_accuracy: 0.9469 - val_loss: 0.1457
Epoch 10/15
711/711          85s 120ms/step - accuracy: 0.9558 - loss: 0.1235 - val_accuracy: 0.9508 - val_loss: 0.1442
Epoch 11/15
711/711          73s 102ms/step - accuracy: 0.9558 - loss: 0.1217 - val_accuracy: 0.9483 - val_loss: 0.1383
Epoch 12/15

```



تفاوت بین نرخ یادگیری 0.0002 و 0.0004 در کد بالا نشان دهنده ای عملکرد عالی در ابتدا و شروع بیش برآزش، عملکرد خوب و تعییم پذیر ولی جای بهبودی وجود دارد.

:LSTM with Callbacks

کالبک‌ها توابع یا ابزارهایی هستند که در حین آموزش مدل (وقتی `model.fit()` اجرا می‌شود) به صورت خودکار در زمان‌های خاصی فراخوانی (Call) می‌شون تا کارهایی انجام بدن. مثلاً می‌توان:

اگر مدل داره بیش از حد آموزش می‌بینه، آموزش رو متوقف کن.

اگر مدل بهتر نمی‌شه، نرخ یادگیری رو کاهش بدن.

بهترین وزن‌ها رو ذخیره کن.

یا فقط اطلاعات آماری رو چاپ کن.

کالبک‌ها مثل ناظرهای هوشمند هستند که به روند آموزش مدل نظارت می‌کنند و در صورت نیاز وارد عمل می‌شن.

استفاده از EarlyStopping در اینجا سبب می‌شود در شرایطی که مدل دیگر بهبود پیدا نکند، متوقف شود.

تنظیمات مهم:

چه چیزی رو دنبال کند. `monitor='val_loss'`

چند ایپاک صبر کند قبل از توقف. `patience=2`

وزن‌های بهترین ایپاک رو بازیابی کند. `restore_best_weights=True`

با ReduceLROnPlateau نرخ یادگیری مدل در شرایطی که گیر کرده باشد (در یک مینیوم محلی مثلاً) کاهش میدهد. حداقل نرخ یادگیری و ضریب کاهش آن در پارامترهای این تابع تنظیم می‌شود.

```

early_stopping_cb = EarlyStopping(monitor='val_loss',
                                 patience=5, # تعداد ایپاک برای انتظار قبل از توقف
                                 verbose=1,
                                 restore_best_weights=True) # بازگرداندن بهترین وزن‌ها (بازگردانن بهترین وزن‌ها)

# کاهش نرخ یادگیری بر اساس عملکرد
reduce_lr_on_plateau_cb = ReduceLROnPlateau(monitor='val_loss',
                                              factor=0.2, # ضریب کاهش نرخ یادگیری (new_lr = lr * factor)
                                              patience=2, # تعداد ایپاک برای انتظار قبل از کاهش نرخ یادگیری
                                              verbose=1,
                                              min_delta=0.0001, # حداقل تغییر در عملکرد
                                              min_lr=0.00001) # حداقل نرخ یادگیری

print(f"\n--- Training Model with Callbacks (Initial LR: {initial_learning_rate}) ---")
history_lstm_cb = lstm_model_cb.fit(X_train, y_train,
                                      epochs=20,
                                      batch_size=64,
                                      validation_data=(X_test, y_test),
                                      callbacks=[early_stopping_cb, reduce_lr_on_plateau_cb],
                                      verbose=1)
evaluate_model(lstm_model_cb, X_test, y_test, labels, model_name="LSTM with Callbacks")
plot_learning_curves(history_lstm_cb, model_name="LSTM with Callbacks")

```

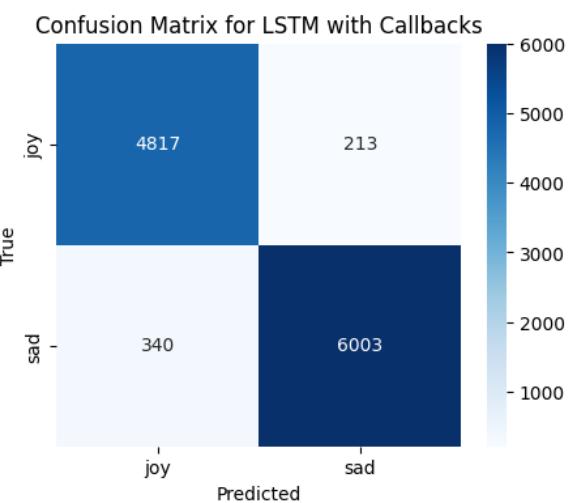
```

Epoch 8: ReduceLROnPlateau reducing learning rate to 1.200000424450263e-05.
711/711 82s 115ms/step - accuracy: 0.9723 - loss: 0.0734 - val_accuracy: 0.9537 - val_loss: 0.1452 - learning_rate: 6.0000e-05
Epoch 9/20
711/711 95s 133ms/step - accuracy: 0.9752 - loss: 0.0685 - val_accuracy: 0.9544 - val_loss: 0.1427 - learning_rate: 1.2000e-05
Epoch 9: early stopping
...
accuracy          0.95      11373
macro avg       0.95      0.95      0.95      11373
weighted avg    0.95      0.95      0.95      11373

```

دقتنهایی : 95.05

0.95: کل F1-Score (ماکرو)



مدل بهخوبی توانسته بین کلاس‌ها تعادل برقرار کند. فقط یه کمی مدل تمایل بیشتری به پیش‌بینی "غمگین" وجود دارد (به خاطر دقت بالاتر اون کلاس).

مدل ۱ : LSTM version 1

در این نسخه سعی شد با افزایش ظرفیت مدل و افزودن لایه Dropout برای جلوگیری از بیش برازش عملکرد مدل بهبود یابد.

```
print("\n--- Defining Model 2: LSTM Version 1 ---")
Tabnine | Edit | Test | Explain | Document
def build_lstm_model_v1(embedding_matrix, max_sequence_length):
    model = Sequential()
    model.add(Embedding(input_dim=embedding_matrix.shape[0],
                         output_dim=embedding_matrix.shape[1],
                         weights=[embedding_matrix],
                         input_length=max_sequence_length,
                         trainable=False))
    model.add(LSTM(128, return_sequences=False))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

lstm_model_v1 = build_lstm_model_v1(embedding_matrix, MAX_SEQUENCE_LENGTH)
```

تغییرات :

لایه LSTM : تعداد واحدهای لایه LSTM به ۱۲۸ واحد افزایش یافته است. این افزایش به مدل اجازه می‌دهد تا نمایش‌های داخلی پیچیده‌تری از داده‌ها را یاد بگیرد. پارامتر return_sequences=False همچنان حفظ شده است.

لایه Dropout : پس از لایه LSTM، یک لایه Dropout با نرخ ۰.۳ اضافه شده است.

لایه Dense میانی : یک لایه Dense با ۶۴ واحد و تابع فعال‌سازی ReLU (Rectified Linear Unit) پیش از لایه خروجی نهایی اضافه شده است. لایه‌های Dense اضافی می‌توانند به مدل در یادگیری ترکیب‌های غیرخطی از ویژگی‌های استخراج شده توسط لایه LSTM کمک کنند.

لایه خروجی (Dense) مشابه مدل پایه، یک لایه Dense با ۱ واحد و تابع فعال‌سازی سیگموئید . کامپایل مدل مشابه نسخه‌ی قبل، با نرخ یادگیری پیش‌فرض.

ارزیابی مدل ۱ : LSTM Version 1

این مدل نیز برای 10 اپیک با اندازه دسته 64 آموزش داده شد. نتایج ارزیابی بر روی مجموعه داده آزمون به شرح زیر است:

```
Epoch 9/10
711/711 45s 63ms/step - accuracy: 0.5619 - loss: 0.6857 - val_accuracy: 0.5577 - val_loss: 0.6869
Epoch 10/10
711/711 49s 69ms/step - accuracy: 0.5600 - loss: 0.6863 - val_accuracy: 0.5577 - val_loss: 0.6867

--- Evaluation: Simple LSTM ---
356/356 8s 22ms/step
...
accuracy          0.56      11373
macro avg       0.28      0.50      0.36      11373
weighted avg    0.31      0.56      0.40      11373
```

معیار **val_loss** یا خطای اعتبار سنجی : این مقدار، میزان خطای مدل را بر روی داده‌های اعتبارسنجی پس از اتمام دوره فعلی نشان می‌دهد. همانند خطای آموزشی (loss)، این مقدار نیز توسط تابع هزینه محاسبه می‌شود.

با وجود افزایش پیچیدگی مدل (افزایش واحدهای LSTM و افزودن لایه Dense میانی) و استفاده از Dropout، تغییر محسوسی در عملکرد مشاهده نشد و مشکل عدم یادگیری کلاس 'joy' همچنان پابرجا بود. این نشان می‌دهد که مشکل اصلی احتمالاً جای دیگری است و صرفاً افزایش تعداد پارامترها را محل نبوده است. یکی از فرضیه‌های اصلی در این مرحله، نامناسب بودن نرخ یادگیری پیش‌فرض Adam و همچنین ساختار لایه خروجی و تابع هزینه برای این مسئله خاص بود.

مدل ۲ : LSTM Version 2

```

# مدل ۳: LSTM Version 2
#
print("\n--- Defining Model 3: LSTM Version 2 ---")
Tabnine | Edit | Test | Explain | Document
def build_lstm_model_v2(embedding_matrix, max_sequence_length, num_classes):
    model = Sequential()
    model.add(Embedding(input_dim=embedding_matrix.shape[0],
                         output_dim=embedding_matrix.shape[1],
                         weights=[embedding_matrix],
                         input_length=max_sequence_length,
                         trainable=False))
    model.add(LSTM(64, return_sequences=False))
    model.add(Dropout(0.3))
    model.add(Dense(num_classes, activation='softmax')) # خروجی سافت‌مکن برای sparse_categorical_crossentropy

    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer=Adam(learning_rate=0.0001),
                  metrics=['accuracy'])
    return model

lstm_model_v2 = build_lstm_model_v2(embedding_matrix, MAX_SEQUENCE_LENGTH, num_classes=len(labels))

```

با توجه به نتایج نامطلوب دو مدل قبلی، در این نسخه تغییرات اساسی‌تری در هایپرپارامترهای کلیدی و ساختار لایه خروجی اعمال شد.

معماری مدل :

لایه Embedding : مشابه مدل‌های قبلی (trainable=False) (LSTM(64, return_sequences=False)).

لایه Dropout : یک لایه Dropout با نرخ 0.3.

لایه خروجی : به جای یک نورون با sigmoid، از یک لایه Dense با تعداد نورون برابر با تعداد کلاس‌ها (num_classes = 2) و تابع فعال‌سازی softmax استفاده شد.

توابع فعال‌سازی غیرخطی در شبکه‌های عصبی بسیار حیاتی هستند. اگر تمام لایه‌های یک شبکه عصبی (از جمله لایه خروجی در مسائل classification) فقط از تبدیلات خطی استفاده کنند، کل شبکه، صرف‌نظر از تعداد لایه‌هایش، تنها قادر به یادگیری توابع خطی خواهد بود. این امر توانایی شبکه را در مدل‌سازی روابط پیچیده و غیرخطی موجود در اکثر داده‌های دنیای واقعی به شدت محدود می‌کند.

توابع غیرخطی مانند Softmax یا Tanh، ReLU، Sigmoid و غیره در لایه‌های پنهان به شبکه این قدرت را می‌دهند که مرازهای تصمیم‌گیری پیچیده‌تری را یاد بگیرد و در نتیجه عملکرد بهتری در وظایف مختلف داشته باشد.

کامپایل مدل :

. بهینه‌ساز (Optimizer): Adam با نرخ یادگیری (learning rate) بسیار کمتر، برابر با 0.001.

تابع هزینه (Loss Function) متناسب با لایه خروجی softmax و برچسب‌های عددی صحیح، از sparse_categorical_crossentropy استفاده شد.

معیار ارزیابی : accuracy

ارزیابی مدل 2

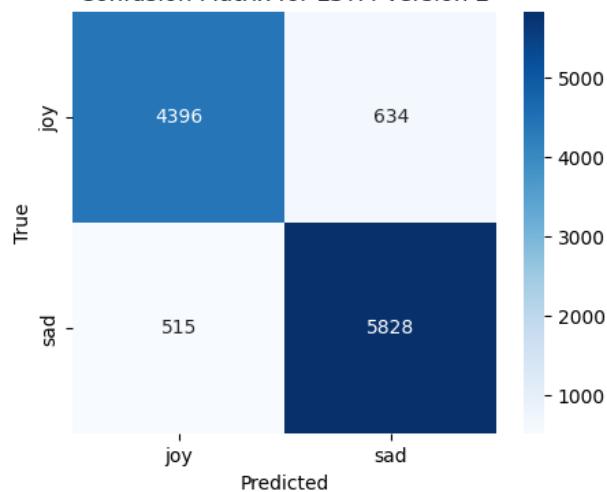
این مدل برای 10 اپیک با اندازه دسته 64 آموزش داده شد. نتایج ارزیابی بر روی مجموعه داده آزمون بسیار امیدوارکننده بود :

دقت (Accuracy) 0.8990 :

امتیاز F1 ماکرو : 0.8973

...	accuracy	0.90	0.90	0.90	11373
macro avg	0.90	0.90	0.90	11373	
weighted avg	0.90	0.90	0.90	11373	

Confusion Matrix for LSTM Version 2



نمودار تاریخچه آموزش نیز نشان‌دهنده روند یادگیری مناسب و کاهش تابع هزینه برای داده‌های آموزشی و اعتبارسنجی بود.

```
Epoch 8/10
711/711 69s 97ms/step - accuracy: 0.8902 - loss: 0.2795 - val_accuracy: 0.8887 - val_loss: 0.2787
Epoch 9/10
711/711 70s 98ms/step - accuracy: 0.8985 - loss: 0.2658 - val_accuracy: 0.8939 - val_loss: 0.2656
Epoch 10/10
711/711 72s 101ms/step - accuracy: 0.9011 - loss: 0.2555 - val_accuracy: 0.8990 - val_loss: 0.2545
```

این تغییرات، به ویژه کاهش شدید نرخ یادگیری و استفاده از لایه خروجی softmax به همراه تابع هزینه sparse_categorical_crossentropy، منجر به بهبود چشمگیری در عملکرد مدل شد. مدل توانست هر دو کلاس را با دقت و F1-score بالایی تشخیص دهد. این نشان می‌دهد که نرخ یادگیری بالا در مدل‌های قبلی، عامل اصلی عدم همگرایی و یادگیری نامناسب بوده است.

همچنین، ترکیب softmax و sparse_categorical_crossentropy اغلب برای مسائل طبقه‌بندی چندکلاسه (که طبقه‌بندی باینری حالت خاصی از آن است) پایداری بهتری در فرآیند آموزش ایجاد می‌کند.

بهبود بیشتر مدل LSTM نسخه دوم با استفاده از توقف زودهنگام (EarlyStopping) و تغییر اندازه دسته :

پس از دستیابی به نتایج مطلوب با مدل نسخه دوم، تلاش شد تا با استفاده از تکنیک توقف زودهنگام و تنظیم اندازه دسته به نتایج پایدارتری دست یابیم.

```
# Add early stopping
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = lstm_model_v2.fit(X_train, y_train,
                            epochs=15, # A few more epochs with early stopping
                            batch_size=32, # Smaller batch size for better learning
                            validation_split=0.1,
                            # class_weight=class_weights,
                            callbacks=[early_stop], # Add early stopping
                            verbose=1)

evaluate_model(lstm_model_v2, X_test, y_test, labels, model_name="LSTM version 2")
```

معماری: همون معماری مدل LSTM Version2

کامپایل مدل :

بهینه‌ساز Adam : با نرخ یادگیری 0.0001

تابع هزینه: sparse_categorical_crossentropy.

ایپاک ها: 15

اندازه دسته (Batch size) : کاهش از 64 به 32

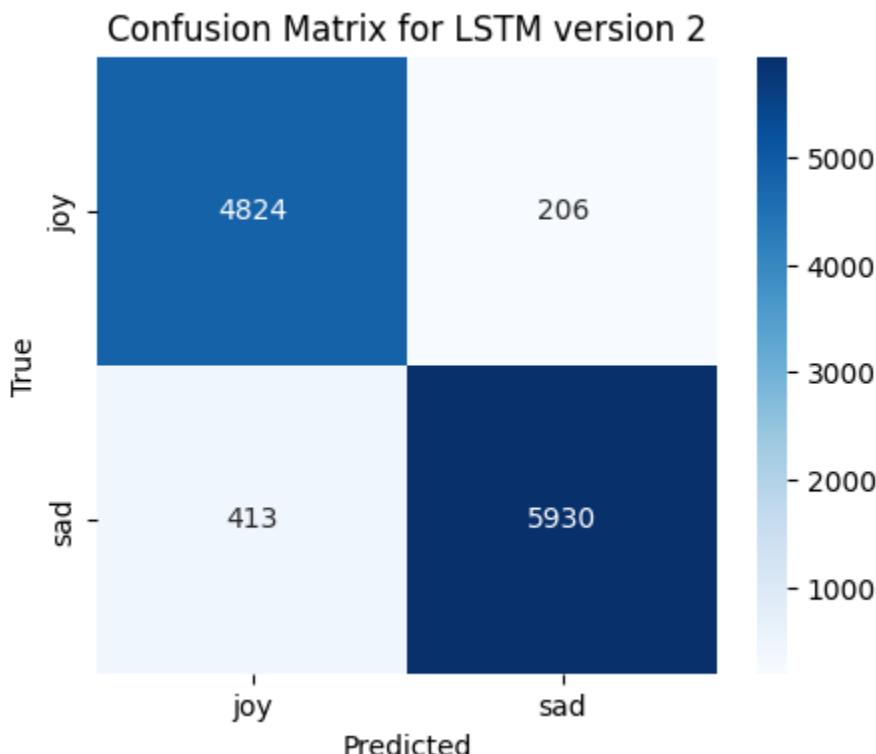
استفاده از 10% داده‌های آموزشی برای اعتبارسنجی در هر ایپاک Validation_split=0.1

بازخوان (EarlyStopping (Callback) با نظارت بر patience=5 و val_loss به این معنا که اگر به مدت 5 اپیک متوالی بهبودی در val_loss مشاهده نشد، آموزش متوقف شده و بهترین وزن‌های مدل بازگردانده می‌شوند.

نتایج ارزیابی بر روی مجموعه داده آزمون به شرح زیر است :

دقت (Accuracy) : 0.9456

امتیاز F1 ماکرو : 0.9451



استفاده از EarlyStopping و کاهش batch_size به 32، منجر به بهبود بیشتر عملکرد و دستیابی به دقต 94.56% شد.

به جلوگیری از ادامه آموزش پس از رسیدن به نقطه بهینه کمک کرده و EarlyStopping کوچکتر نیز می‌تواند منجر به همگرایی بهتر و دقیق‌تر گراییانها شود.

```
1280/1280 - 108s 84ms/step - accuracy: 0.9605 - loss: 0.1080 - val_accuracy: 0.9371 - val_loss: 0.1765
Epoch 11/15
1280/1280 - 104s 81ms/step - accuracy: 0.9609 - loss: 0.1051 - val_accuracy: 0.9393 - val_loss: 0.1602
Epoch 12/15
1280/1280 - 104s 81ms/step - accuracy: 0.9609 - loss: 0.1051 - val_accuracy: 0.9393 - val_loss: 0.1602
Epoch 13/15
...
accuracy      0.95      0.95     11373
macro avg    0.94      0.95      0.95     11373
weighted avg 0.95      0.95      0.95     11373
```

بررسی تاثیر وزن دهی به کلاس ها (Class Weighting)

در آخرین مرحله از آزمایش های مربوط به مدل LSTM ، تاثیر استفاده از پارامتر class_weight برای مقابله با عدم توازن جزئی در توزیع کلاس ها (حدود 'joy' 45% و 'sad' 55%) بررسی شد.

از آنجایی که عدم توازن جزئی ای در کلاس ها وجود داشت که به آن پرداخته نشد، پس از بدست آوردن نتایج مطلوب از اخرين تنظيمات مدل به بررسی تاثير وزن دهی روی عملکرد قبلی مدل (بهينه ترین حالت بدست آمده) پرداختيم.

با اضافه کردن اين خط و ران دوباره ی سلول قبل نتایج زیر بدست آمد :

```
class_weights = {0: 1.2, 1: 1.0}
```

در اين مورد خاص، اعمال وزن دهی به کلاس ها منجر به بهبود عملکرد نسبت به مدل قبلی نشد و دقت اندکی کاهش یافت. (از 94.56% به 92%)
این نشان می دهد که یا عدم توازن داده ها در این حد تاثیر منفی قابل توجهی بر یادگیری مدل موفق قبلی نداشته، یا اينکه مقادير وزن انتخاب شده بهينه نبوده اند. با اين حال، عملکرد مدل همچنان در سطح بسيار خوبی قرار دارد.

accuracy			0.92	11373
macro avg	0.92	0.92	0.92	11373
weighted avg	0.92	0.92	0.92	11373

7. استخراج ویژگی با (Term Frequency - Inverse Document Frequency) TF-IDF

در کنار مدل های یادگیری عمیق، استفاده از الگوریتم های کلاسیک یادگیری ماشین نیز به عنوان یک روش متداول و کارآمد در مسائل طبقه بندی متن شناخته می شود. در این راستا ابتدا نیاز است که متون پیش پردازش شده به یک بازنمایی عددی مناسب تبدیل شوند. در این پروژه از روش TF-IDF برای تبدیل متن به بردار ویژگی استفاده شده است.

یک روش آماری است که برای ارزیابی اهمیت یک کلمه (یا مجموعه ای از کلمات) در یک سند (هر توابیت) نسبت به مجموعه ای از اسناد (کورپوس) به کار می رود. این روش به کلماتی که در یک سند خاص به کرات ظاهر می شوند اما در کل کورپوس نادر هستند، وزن بیشتری اختصاص می دهد.

واژه کورپوس (corpus) در اینجا به مجموعه کل اسنادی که بررسی میکنم (محتوای کامل ستون نماینده 'full_text' اشاره دارد.

خروجی این فرآیند یک ماتریس پراکنده (Sparse) است که هر سطر نماینده یک سند (توبیت) و هر ستون نماینده یک کلمه (یا n-gram) است. مقدار هر سلول وزن TF-IDF آن کلمه در آن سند است.

تعداد تکرار یک کلمه در یک سند. (Term Frequency) TF : لگاریتم معکوس نسبت تعداد کل اسناد به تعداد اسنادی که حاوی آن کلمه هستند. (Inverse Document Frequency) IDF

IDF به کلماتی که در اسناد کمتری ظاهر می‌شوند (و احتمالاً تمایزدهنگی بیشتری دارند) وزن بیشتری می‌دهد و از اهمیت کلمات بسیار رایج (مانند کلمات توقف که البته قبل از حذف شده‌اند) می‌کاهد. اگر کلمه‌ای در یک توبیت زیاد تکرار شود (TF بالا) اما در کل مجموعه توبیت‌ها کمتر دیده شود (IDF بالا)، آنگاه TF-IDF به آن کلمه وزن بیشتری می‌دهد، زیرا آن کلمه به احتمال زیاد برای توصیف محتوای آن توبیت خاص، کلمه مهم و تمایزکننده‌ای است.

وزن نهایی TF-IDF حاصلضرب TF و IDF است.

$$TF(t, d) = \frac{\text{number of times } t \text{ appears in } d}{\text{total number of terms in } d}$$

$$IDF(t) = \log \frac{N}{1 + df}$$

$$TF - IDF(t, d) = TF(t, d) * IDF(t)$$

: Scikit-learn با پیاده‌سازی با

از TfIdfVectorizer برای محاسبه وزن‌های TF-IDF استفاده شد.

پارامترها :

```
tfidf_vectorizer = TfidfVectorizer(  
    max_features=10000, # limit num  
    ngram_range=(1, 2), # unigrams  
    lowercase=False # texts are  
)
```

max_features=10000 : این پارامتر تعداد ویژگی‌های نهایی را به **10000** ویژگی با بیشترین TF-IDF محدود می‌کند. این کار به کاهش ابعاد فضای ویژگی و جلوگیری از مشکلات ناشی از ابعاد بالا کمک می‌کند.

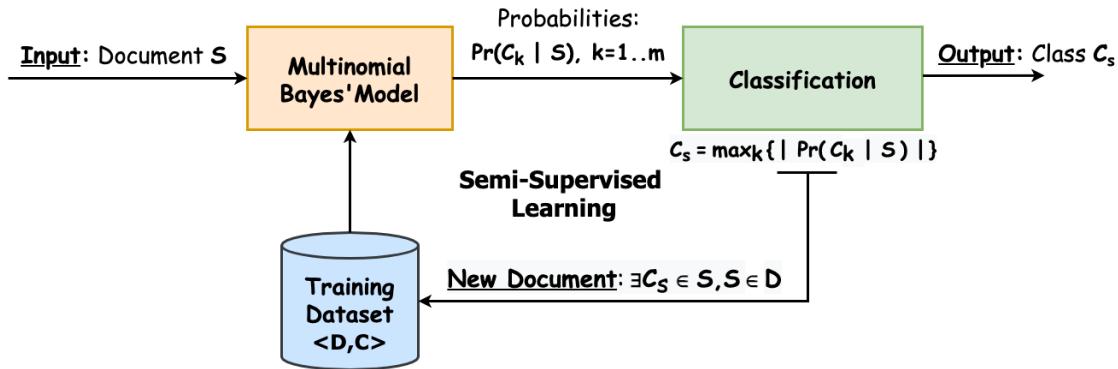
ngram_range=(1, 2) : این تنظیم به **vectorizer** اجازه می‌دهد علاوه بر کلمات تکی (بونیگرام)، توالی‌های دوکلمه‌ای (بایگرام) را نیز به عنوان ویژگی در نظر بگیرد. بایگرام‌ها می‌توانند در درک بهتر عبارات و زمینه‌های ساده متنی مؤثر باشند.

lowercase=False : از آنجایی که فرآیند پیش‌پردازش متن (شامل نرمال‌سازی) در مراحل قبلی انجام شده است، نیازی به تبدیل مجدد حروف به حالت کوچک در این مرحله نیست.

```
# Fit and transform on training, only transform on test  
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)  
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

پس از تعریف **vectorizer** ، ابتدا بر روی مجموعه آموزشی (**X_train**) آموزش داده شد (**fit**) تا واژگان و آمارهای **IDF** محاسبه شوند و سپس هم مجموعه آموزشی و هم مجموعه آزمایشی (**X_test**) با استفاده از پردازساز آموزش‌دیده به ماتریس‌های **TF-IDF** تبدیل شوند. (**transform**).

8. مدل بیز ساده چندجمله‌ای (Multinomial Naive Bayes)



الگوریتم بیز ساده چندجمله‌ای (Multinomial Naive Bayes) یک طبقه‌بند احتمالاتی است که بر اساس قضیه بیز با فرض استقلال شرطی بین ویژگی‌ها عمل می‌کند. مدل MultinomialNB به خوبی با داده‌هایی کار می‌کند که در آن هر ویژگی نشان‌دهنده فراوانی (یا یک معیار مبتنی بر فراوانی ماتنند TF-IDF) یک رویداد خاص (در اینجا، یک کلمه یا n-gram) است.

این مدل برای داده‌های شمارشی طراحی شده و TF-IDF نیز روشی در شمارش کلمات دارد همچنین علی‌رغم سادگی اش اختلاف نتایج رقابتی و خوبی در طبقه‌بندی اسناد متنه ارائه می‌دهد. بویژه زمانی که تعداد ویژگی‌ها زیاد است.

```

nb_model = MultinomialNB()
nb_model.fit(X_train_tfidf, y_train)
y_pred_nb = nb_model.predict(X_test_tfidf)

```

یک نمونه از کلاس `MultinomialNB` با پارامترهای پیش‌فرض ایجاد و سپس بر روی ماتریس-`TF-IDF` داده‌های آموزشی (`X_train_tfidf`) و برچسب‌های متناظر (`y_train`) آموزش داده شد. پس از آموزش، مدل برای پیش‌بینی برچسب‌های احساسات بر روی ماتریس `TF-IDF` داده‌های آزمایشی (`X_test_tfidf`) به کار گرفته شد.

عملکرد مدل بیز ساده با استفاده از معیارهای دقت (Accuracy)، امتیاز F1 (ماکرو) و گزارش طبقه‌بندی (classification report) کامل ارزیابی گردید.

```

--- Naive Bayes Results ---
Accuracy: 0.857469445177174
F1 Score (macro): 0.8533860865509316
      precision    recall   f1-score   support
joy        0.88     0.78     0.83     5030
sad        0.84     0.92     0.88     6343
accuracy           0.86     0.86     0.86    11373
macro avg       0.86     0.85     0.85    11373
weighted avg     0.86     0.86     0.86    11373

```

این مدل به دقت کلی 85.75% و امتیاز F1 ماکرو 85.34% دست یافت. با بررسی گزارش طبقه‌بندی، مشاهده می‌شود که مدل در تشخیص کلاس (Recall: 0.92) عملکرد بهتری نسبت به کلاس 'joy' (Recall: 0.78) داشته است، هرچند Precision برای کلاس 'joy' (0.88) بالاتر از کلاس 'sad' (0.84) بوده است. به طور کلی، این مدل یک عملکرد پایه مناسب را ارائه می‌دهد.

9. مدل درخت تصمیم (Decision Tree - DT)

مفهوم: درخت تصمیم مدلی است که با ساختن یک ساختار درختی، داده‌ها را بر اساس ویژگی‌ها تقسیم‌بندی می‌کند. هر گره داخلی نشان‌دهنده یک آزمون روی یک ویژگی و هر شاخه نشان‌دهنده نتیجه آزمون است. برگ‌ها نشان‌دهنده برچسب کلاس نهایی هستند.

```

dt_model = DecisionTreeClassifier(max_depth=30, min_samples_leaf=10, random_state=42)
dt_model.fit(X_train_tfidf, y_train)
y_pred_dt = dt_model.predict(X_test_tfidf)

```

یک نمونه از کلاس `DecisionTreeClassifier` با پارامترهایی برای کنترل پیچیدگی درخت، شامل `max_depth=30` حداقل عمق درخت، `min_samples_leaf=10` حداقل تعداد نمونه در هر گره برگ و `random_state=42` برای تکرارپذیری، ایجاد شد. سپس مدل بر روی داده‌های آموزشی-IDF آموزش داده شده و برای پیش‌بینی بر روی داده‌های آزمون استفاده گردید.

--- Decision Tree Results ---				
Accuracy: 0.9628945748703068				
F1 Score (macro): 0.9624518103221231				
	precision	recall	f1-score	support
joy	0.95	0.97	0.96	5030
sad	0.97	0.96	0.97	6343
accuracy			0.96	11373
macro avg	0.96	0.96	0.96	11373
weighted avg	0.96	0.96	0.96	11373

مدل درخت تصمیم با تنظیمات اعمال شده، عملکرد بسیار قابل توجهی از خود نشان داد و به دقت کلی 96.29% و امتیاز F1 ماکرو 96.25% داشت یافت. این مدل توانست هر دو کلاس 'joy' و 'sad' را با مقادیر Precision و Recall بسیار بالایی (حدود 95% تا 97%) تشخیص دهد. نتایج نشان می‌دهد که با کنترل مناسب پیچیدگی، درخت تصمیم می‌تواند حتی بر روی داده‌های TF-IDF با ابعاد نسبتاً بالا نیز به خوبی عمل کند. این عملکرد به طور قابل ملاحظه‌ای بهتر از مدل بیز ساده است.

10. آموزش و ارزیابی مدل‌های طبقه‌بندی کلاسیک با ویژگی‌های TF-IDF شامل تری گرام

در ادامه برای بهبود عملکرد مدل‌های کلاسیک و همچنین استخراج ویژگی‌هایی که بتوانند به طور بالقوه اطلاعات زمینه‌ای (contextual information) بیشتری را، مشابه آنچه مدل‌های عمیق‌تر مانند LSTM قادر به یادگیری آن هستند، ثبت کنند، تصمیم گرفته شد تا از توالی‌های طولانی‌تر کلمات یعنی تری‌گرام‌ها (توالی‌های سه‌کلمه‌ای) در کنار یونی‌گرام‌ها و بای‌گرام‌ها استفاده شود. انتظار مورود که این کار به مدل‌ها در درک بهتر عبارات و اصطلاحات چندکلمه‌ای که ممکن است برای تشخیص احساسات کلیدی باشند، کمک کند.

```
tfidf_trigram = TfidfVectorizer([
    max_features=10000, # limit
    ngram_range=(1,3), # unigram
    lowercase=False # texts
])
```

یک نمونه جدید از `TfidfVectorizer` با تغییر پارامتر `ngram_range` به `(1, 3)` ایجاد گردید. سایر پارامترها، از جمله `max_features=10000` (برای محدود کردن تعداد کل ویژگی‌ها به 10000) ویژگی برتر) و `lowercase=False` (با توجه به اینکه متون از قبل نرمال‌سازی شده‌اند)، مشابه مرحله قبل باقی مانند.

سپس، این وکتورایزر جدید (`tfidf_trigram`) بر روی همان مجموعه داده آموزشی متنی (`X_train`) آموزش داده شد (`fit`) تا واژگان شامل تریگرام‌ها (Trigram) و آمارهای IDF مربوطه را یاد بگیرد. پس از آن، هر دو مجموعه داده آموزشی و آزمایشی (`X_test` و `X_train`) به ماتریس‌های TF-IDF جدید، که اکنون شامل ویژگی‌های ترایگرام نیز بودند، تبدیل شدند(`transform`). این ماتریس‌های جدید به ترتیب `X_test_trigram` و `X_train_trigram` نامگذاری شدند.

مدل `MultinomialNB` با پارامترهای پیش‌فرض، این بار بر روی ماتریس `X_train_trigram` آموزش داده شد و سپس برای پیش‌بینی برچسب‌های احساسات مجموعه `X_test_trigram` مورد استفاده قرار گرفت.

```

nb_model = MultinomialNB()
nb_model.fit(X_train_trigram, y_train)
y_pred_nb_trigram = nb_model.predict(X_test_trigram)

print("\n--- Naive Bayes Results ---")
print("Accuracy:", accuracy_score(y_test, y_pred_nb_trigram))
print("F1 Score (macro):", f1_score(y_test, y_pred_nb_trigram, average='macro'))
print(classification_report(y_test, y_pred_nb_trigram))

--- Naive Bayes Results ---
Accuracy: 0.8487646179548053
F1 Score (macro): 0.8441596855348639
      precision    recall  f1-score   support
  joy       0.88     0.77     0.82     5030
  sad       0.83     0.92     0.87     6343

accuracy           0.85      --      0.85    11373
macro avg       0.85     0.84     0.84    11373
weighted avg     0.85     0.85     0.85    11373

```

مدل بیز ساده با استفاده از ویژگی‌های تریگرام به دقت کلی **84.88%** و امتیاز **F1** ماکرو **84.42%** دست یافت. در مقایسه با نتایج قبلی همین مدل که تنها از یونیگرام و بایگرام استفاده می‌کرد (دقت **85.75%** و **F1** ماکرو **85.34%**، مشاهده می‌شود که افزودن تریگرام‌ها در این مورد خاص منجر به بهبود عملکرد مدل بیز ساده نشده و حتی اندکی افت نیز داشته است **Recall** برای کلاس 'joy' از **0.78** به **0.77** کاهش یافته، در حالی که برای کلاس 'sad' ثابت مانده است. این نتیجه ممکن است نشان‌دهنده این باشد که برای مدل بیز ساده، افزودن ترایگرام‌ها (با توجه به محدودیت **max_features** (ممکن است منجر به افزایش پراکندگی یا ورود ویژگی‌های نویزی شده باشد که با فرضیات ساده‌انگارانه این مدل به خوبی سازگار نیستند.

مدل **DecisionTreeClassifier** با همان پارامترهای قبلی **(max_depth=30, min_samples_leaf=10, random_state=42)** بر روی ماتریس **X_train_trigram** آموزش داده شد و پیش‌بینی‌ها بر روی **X_test_trigram** صورت گرفت. لازم به ذکر است که پارامترهای کامنت‌شده در کد **(max_features='sqrt', criterion='entropy', class_weight='balanced')** در هر مرحله مجزا ران شدند و در نتیجه‌ی افت عملکرد مدل کامنت شدند و با تنظیمات پیشفرض ادامه دادیم.

```

● Click to add a breakpoint TreeClassifier(
    max_depth=30,
    min_samples_leaf=10,
    # max_features='sqrt',
    # criterion='entropy',
    # class_weight='balanced',
    random_state=42
)
dt_model.fit(X_train_trigram, y_train)
y_pred_dt_trigram = dt_model.predict(X_test_trigram)

```

مدل درخت تصمیم با استفاده از ویژگی‌های تراویگرام به دقت کلی 96.45% و امتیاز F1 ماکرو 96.41% دست یافت. این نتایج نشان‌دهنده بهبود جزئی نسبت به عملکرد همین مدل با ویژگی‌های یونیگرام و بایگرام (که دقت 96.29% و F1 ماکرو 96.25% داشت) است. هرچند این بهبود بسیار اندک است، اما نشان می‌دهد که افزودن تراویگرام‌ها در این مورد خاص توانسته است به مقدار کمی به قدرت تفکیک‌پذیری مدل درخت تصمیم کمک کند. عملکرد کلی مدل همچنان در سطح بسیار بالایی قرار دارد و توانایی آن در تشخیص هر دو کلاس با دقت و بازیابی بالا قابل توجه است.

```

--- Decision Tree Results ---
Accuracy: 0.9644772707289194
F1 Score (macro): 0.9640533918723644
      precision    recall  f1-score   support
          joy       0.95     0.97     0.96     5030
          sad       0.97     0.96     0.97     6343
          accuracy                           0.96    11373
      macro avg       0.96     0.96     0.96    11373
  weighted avg       0.96     0.96     0.96    11373

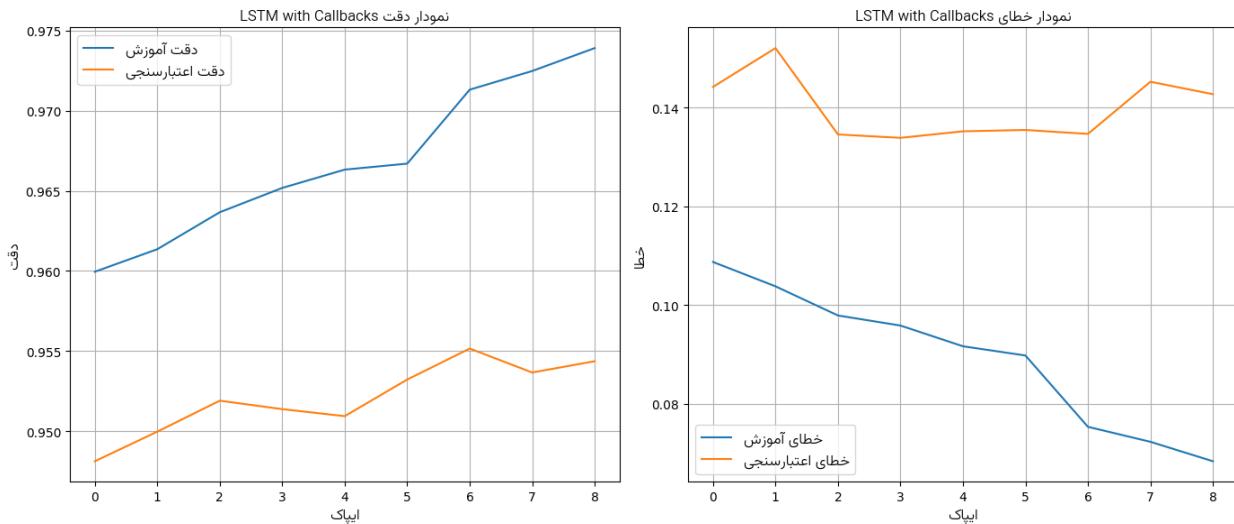
```

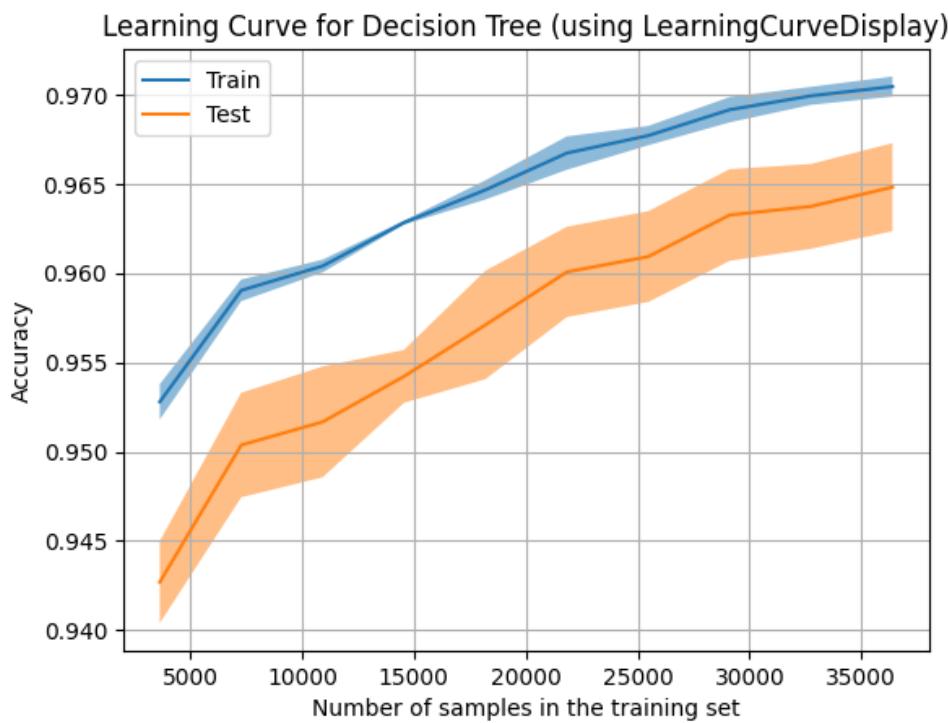
11. مقایسه عملکرد مدل کلاسیک برتر با مدل یادگیری عمیق LSTM

پس از بررسی مدل‌های کلاسیک مختلف، مشاهده شد که مدل درخت تصمیم با ویژگی‌های TF-IDF شامل تراویگرام بهترین عملکرد را در میان آن‌ها با دقت 96.45% و امتیاز F1 مacro 96.41% داده است.

در بخش مدل‌های یادگیری عمیق، بهترین عملکرد متعلق به مدل **LSTM with callbacks** با دقت 97.5% و امتیاز F1 مacro 95% است.

برای درک بیشتر تفاوت دو مدل برتر، منحنی‌های یادگیری هر یک را مقایسه می‌کنیم.





برای درک بهتر چگونگی یادگیری مدل درخت تصمیم و بررسی پتانسیل بیشبرازش (overfitting) یا کمپرازش (underfitting)، منحنی یادگیری آن با استفاده از LearningCurveDisplay از کتابخانه Scikit-learn ترسیم گردید. این نمودار، دقت مدل را بر روی مجموعه آموزشی و مجموعه اعتبارسنجی (که در اینجا همان مجموعه آزمون در نظر گرفته شده) به ازای افزایش تدریجی حجم داده‌های آموزشی نمایش می‌دهد.

نمودار منحنی یادگیری دو خط اصلی را نشان می‌دهد:

امتیاز آموزش (Training score): این خط نشان‌دهنده دقت مدل بر روی داده‌هایی است که برای آموزش آن در هر مرحله (با اندازه‌های مختلف) استفاده شده است.

امتیاز اعتبارسنجی متقابل (Cross-validation score / Test score): این خط نشان‌دهنده دقت مدل بر روی داده‌های دیده‌نشده (مجموعه اعتبارسنجی یا آزمون) است، پس از آنکه مدل بر روی بخش متناظری از داده‌های آموزشی آموزش دیده است.

در حالت ایده‌آل، با افزایش حجم داده‌های آموزشی، هر دو امتیاز آموزش و اعتبارسنجی افزایش یافته و به یکدیگر نزدیک می‌شوند. اگر این دو منحنی به یک مقدار دقت بالا همگرا شوند، نشان‌دهنده این است که مدل به خوبی یاد گرفته و تعمیم‌پذیری مناسبی دارد.

امتیازات هر دو منحنی در انتهای به مقادیر بالایی همگرا شده‌اند. شکاف بین امتیاز آموزش و امتیاز اعتبارسنجی خیلی بزرگ نیست، که نشان می‌دهد با تنظیمات `min_samples_leaf` و `max_depth` بیشبرازش به خوبی کنترل شده است.

این نمودار به ما اطمینان می‌دهد که مدل نه تنها بر روی داده‌های آموزشی عملکرد خوبی دارد، بلکه این عملکرد به خوبی به داده‌های جدید نیز تعمیم پیدا می‌کند و انتخاب هایپرپارامترها (مانند عمق درخت) مناسب بوده است.

منحنی یادگیری مدل درخت تصمیم، اعتماد ما را به پایداری و قابلیت تعمیم نتایج بالای آن افزایش می‌دهد. این نشان می‌دهد که عملکرد 96.45% صرفاً یک نتیجه شانسی یا ناشی از بیش‌برازش بر روی بخش خاصی از داده‌ها نبوده، بلکه مدل توانسته است با افزایش حجم داده‌های آموزشی، به طور مؤثری یاد بگیرد و این یادگیری را به داده‌های نادیده تعمیم دهد.

در مدل **LSTM** انتخابی، عملکرد بالا روی داده‌های تست مشاهده می‌شود. دقت کلی 0.9514 و F1 ماکرو 0.9508 امتیاز‌های بسیار بالایی هستند که نشان‌دهنده توانایی قوی مدل در طبقه‌بندی داده‌های جدید است.

استفاده از EarlyStopping و ReduceLROnPlateau باعث شده است که مدل در مرحله مناسبی از آموزش متوقف شود (زمانی که بیشترین توانایی تعمیم را دارد) و از بیش‌برازش شدید جلوگیری شود. بازیابی وزن‌های بهترین اپاک (اپاک ۴) اطمینان می‌دهد که مدلی انتخاب شده که کمترین خطای اعتبارسنجی را داشته است.

اگرچه نشانه‌هایی از شروع بیش‌برازش در اپاک‌های بعدی (پس از ۴ یا ۷ دیده می‌شود، اما **Callback**‌ها به درستی عمل کرده‌اند و مدل را قبل از اینکه این بیش‌برازش تأثیر منفی بر عملکرد روی داده‌های جدید بگذارد، متوقف کرده و بهترین وزن‌ها را بازیابی کرده‌اند. نتیجه نهایی در مجموعه تست (با وزن‌های اپاک ۴) بسیار عالی است.

بنابراین، با اطمینان می‌توان گفت که این مدل با این عملکرد بالا و متعادل، کاندید بسیار مناسبی برای مدل برنده است.

نتایج هر دو مدل با متريک های مشترک بسیار نزدیک به هم بود. پس با نتیجه‌ی تجربی حاصل از عملکرد مدل‌ها روی توييت یا ورودی‌هایی که هر دو مدل نديده‌اند، تصميم بر انتخاب مدل **LSTM with Callbacks** شد. مدلی که در آستانه بیش‌برازش با بالاترین مقدار دقت در داده‌های تست و آموزش در وزن مناسب متوقف شد.

```

1/1 ━━━━━━━━ 0s 351ms/step
█ Sample Input: ..روز زیبایی بود و من ذوق زیادی داشتم

Decision Tree:
► Raw Prediction: sad
► Interpreted: sad

LSTM Model:
► Raw Prediction (probability): 0.16591637
► Rounded Prediction: 0
► Interpreted: joy

```

این نتیجه، ارزش بررسی همزمان رویکردهای کلاسیک و یادگیری عمیق و انتخاب روش بهینه بر اساس شواهد تجربی را نشان می‌دهد.

12. رابط گرافیکی پروژه و app.py

ایمپورت کتابخانه ها :

```

import streamlit as st
import re
import numpy as np
import tensorflow as tf
import pickle
from parsivar import Normalizer as ParsivarNormalizer
from parsivar import Tokenizer as ParsivarTokenizer

```

استریمლیت (Streamlit) یک فریمورک (کتابخانه) متن باز برای زبان برنامه‌نویسی پایتون است که به توسعه‌دهندگان امکان می‌دهد به سادگی و خیلی سریع اپلیکیشن‌های تحت وب تعاملی بسازند، مخصوصاً برای نمایش نتایج مدل‌های یادگیری ماشین، داده‌کاوی، و تحلیل داده.

برای پارک‌گذاری توکنایزر ذخیره شده ایمپورت شد. **pickle** یک ماثول ساتاندارد است که برای ذخیره‌ی شی پایتونی مثل مدل یادگیری ماشین، دیکشنری، داده‌های میانی پروژه، لیست و غیره بکار می‌رود. ابزار تابع پیش پردازش دقیقاً مثل تابع پیش پردازش فایل آموزش مدل تعریف و استفاده شد.

تابع `preprocess_persian_text()` کل زنجیره‌ی پردازش متن را مانند قبل شامل می‌شود. در نهایت متن پیش پردازش شده با توکن ساز `parsivar` توکن می‌شود.
مثال خروجی :

"وای اشک شوق چقدر خوشحالم" → "وای اشک شوق چقدر خوشحالم" 

```
# Streamlit UI
st.set_page_config(page_title="Persian Emotion Classifier", layout="centered")
st.title("🗣 Persian Tweet Emotion Classifier")
st.markdown("ر احساسات موجود در یک توبیت فارسی را (بین شادی و غم) تشخیص می‌دهد")

user_input = st.text_area("": یک توبیت فارسی وارد کنید")
```

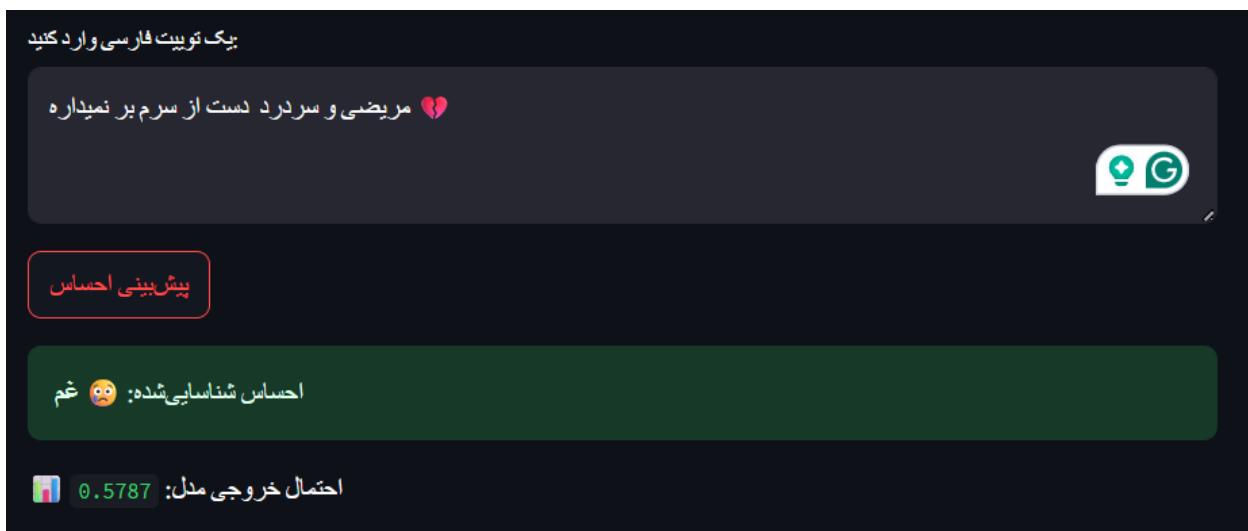
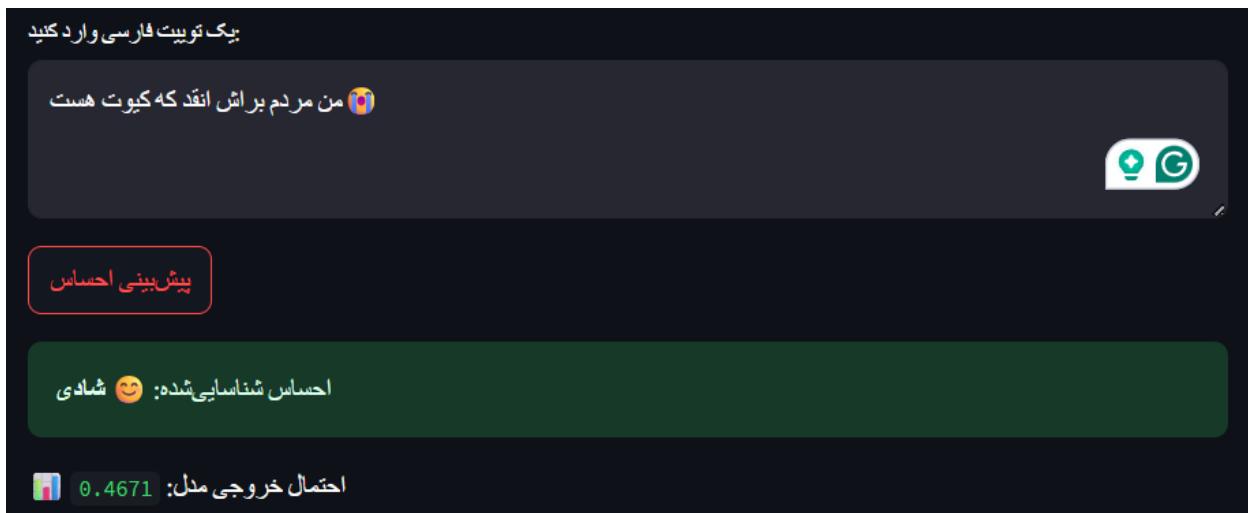
یک رابط کاربری ساده با ابزار `streamlit` مطابق کد بالا ساختیم. عنوان صفحه، توضیح و تنظیمات اولیه را برای رابط گرافیکی تعیین می‌کنیم. ورودی را از کاربر در `user_input` می‌گیریم و بررسی های لازم را روی ورودی انجام داده سپس به تابع پیش پردازش می‌فرستیم تا توکن شود.

```
preprocessed = preprocess_persian_tweet(user_input)
sequence = tokenizer.texts_to_sequences([preprocessed])
padded = tf.keras.preprocessing.sequence.pad_sequences(sequence, maxlen=50)

prediction = model.predict(padded)[0][0]
label = "😊 غم" if prediction >= 0.5 else "😊 شادی"
st.success(f"احساس شناسایی شده : **{label}**")
st.write(f"📊 احتمال خروجی مدل : {prediction:.4f}")
```

بعد از پیش پردازش:

توکن‌ها به اعداد تبدیل می‌شوند (`texts_to_sequences`)
دنباله‌ها پد می‌شوند (با صفرها پر می‌شوند تا طولشان یکی شود)
پیش بینی خام مدل انجام می‌شود و خروجی بر اساس احتمال به یک خروجی باینری نگاشت می‌شود.



* ران کردن برنامه با دستور : Streamlit run app.py در دایرکتوری برنامه

پایان