بسمه تعالی

نام درس:

کاوش دادگان انبوه (**Big data**)

نام پروژه:

ساخت مدل پیش بینی بر روی یک مجموعه کلان داده (**Big data**) با استفاده از ابزار یادگیری ماشینی برای کلان داده

نام استاد درس:

دکتر الهام آخوندزاده

نام **TA** درس:

مهندس امیررضا نجفی

نام و شماره دانشجویی اعضای گروه:

سمیه حمیدی ۴۰۰۶۶۲۴۱۰۰۲

هاله خوش شانس 40066

جواد رفیعی فرد 40066241005

محمدرضا شاقوزی 40066241008

بسمه تعالی

گزارش پروژه درس داده کاوی

ویرایش: 22/11/1401 – تدوین: 06/11/1401 – تهیه‏کننده: سمیه حمیدی، هاله خوش شانس، جواد رفیعی فرد، محمدرضا شاقوزی

**فهرست**

[1 مقدمه 4](#_Toc127034286)

[1-1 مراحل نصب و راه اندازی spark 5](#_Toc127034287)

[1-2 مقایسه پلتفرم‌های Databricks و Zeppelin 7](#_Toc127034288)

[1-3 نصب Apache Zeppelin 7](#_Toc127034289)

[2 معرفی مجموعه داده و توصیف آن 9](#_Toc127034290)

[2-1 معرفی کلی مجموعه داده و نحوه شکل گیری آن 9](#_Toc127034291)

[2-2 معرفی ویژگی های ورودی 10](#_Toc127034292)

[2-3 ویژگی خروجی (هدف): 11](#_Toc127034293)

[3 تحلیل اکتشافی داده (**Exploratory data analysis**) 12](#_Toc127034294)

[3-1 مجموعه داده در یک نگاه 12](#_Toc127034295)

[**4** فراخوانی کتابخانه های مورد نیاز و انجام تنظیمات **pyspark** 12](#_Toc127034296)

[5 تجمیع چانک های مجموعه داده 12](#_Toc127034297)

[6 مرحله پیش پردازش 13](#_Toc127034298)

[6-1 تبدیل به داده‌های عددی 13](#_Toc127034299)

[6-2 نرمال سازی 15](#_Toc127034300)

[6-3 تشخیص و حذف داده های پرت 15](#_Toc127034301)

[7 داده‏های از دست رفته (**Missing Values**) 15](#_Toc127034302)

[7-1 بررسی داده‏های از دست رفته 15](#_Toc127034303)

[7-2 پر کردن داده‏های از دست رفته 16](#_Toc127034304)

[8 مهندسی ویژگی (**Feature Engineering**) 18](#_Toc127034305)

[9 مدل‌سازی 19](#_Toc127034306)

[9-1 روش رگرسیون منطقی (**Logistic Regression**) 19](#_Toc127034307)

[9-2 روش درخت تصمیم (**Decision Tree**) 20](#_Toc127034308)

[9-3 روش جنگل تصادفی (**Random Forest**) 20](#_Toc127034309)

[9-4 روش طبقه‌بند ماشین بردار پشتیبان خطی (Linear SVC) 21](#_Toc127034310)

[10 ارزیابی 21](#_Toc127034311)

[10-1 سنجه **Macro F1** 22](#_Toc127034312)

[10-2 سنجه دقت (precision) یا positive predictive value (PPV) 22](#_Toc127034313)

[10-3 سنجه یادآوری (Recall) یا sensitivity یا hit rateیا true positive rate (TPR) 22](#_Toc127034314)

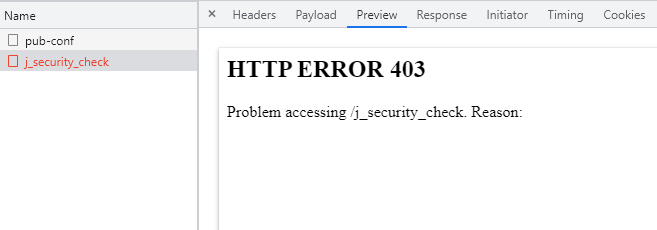
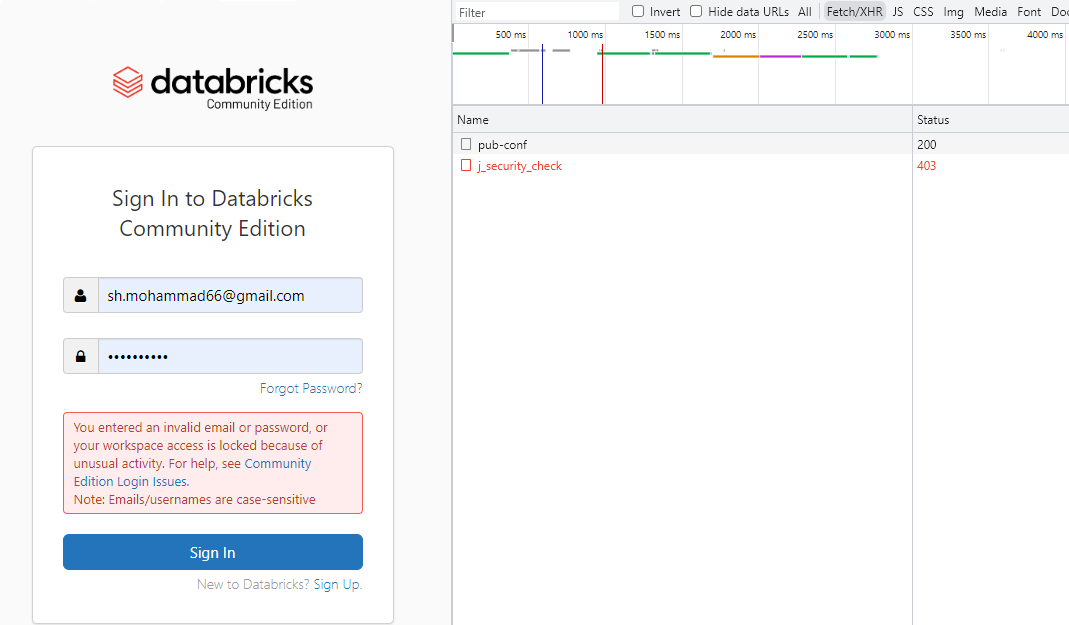
[10-4 سنجه **AUC score** 23](#_Toc127034315)

[11 مقایسه عملکرد مدل‌ها 29](#_Toc127034316)

# مقدمه

در پروژه نهایی کاوش دادگان انبوه، قرار است با استفاده از ابزارهای big data به دنبال پردازش و ساخت طبقه‌بند برای مجموعه داده Epidemiologisches Krebsregister NRW باشیم. باتوجه به اینکه مقدمات ابزارهایی مانند Apache Spark در طول ترم توسط جناب مهندس نجفی آموزش داده شد، در زمینه کدنویسی پروژه مشکلی پیش نیامد. اما بزرگترین چالش ما تأمین منابع برای اجرای عملیات‌های big data بود.

هدف ابتدایی گروه، این بود که یک پروژه در databricks ایجاد کنیم و به صورت یکپارچه بر روی امکانات apache spark کار کنیم. اما بدلیل خطای 403 فیلترشکن در پلتفرم databricks اکثریت اعضا گروه نتوانستیم به این سایت دسترسی داشته باشیم. لذا تصمیم بر این شد که بصورت محلی(local) کار را پیش ببریم و کد را در github برای دسترسی همه اعضا به اشتراک بگذاریم.



## مراحل نصب و راه اندازی spark

ابتدا ابزارهای زیر را به ترتیب نصب کردیم:

Java.SE.Development.Kit.17.0.5.x64

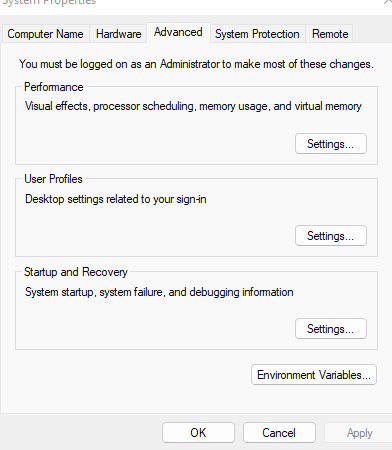
Git-2.39.1-64-bit

GitHubDesktopSetup-x64

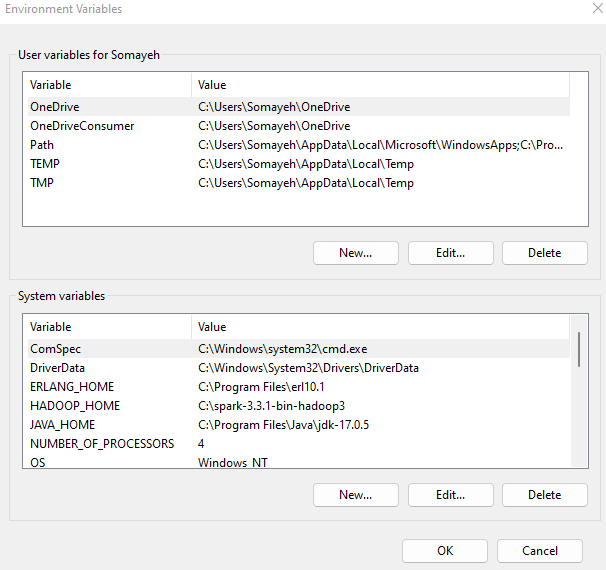
vscode

سپس فایل زیپ apache spark را از آدرس زیر دانلود و در درایو c کپی و استخراج می‌کنیم.

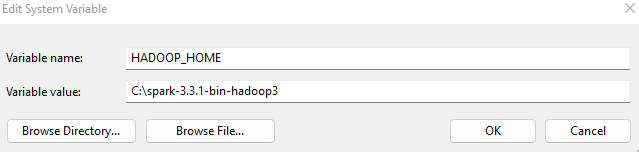
<https://spark.apache.org/downloads.html>

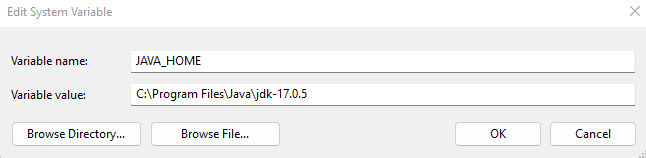
نسخه‌ای که در این پروژه استفاده کردیم، spark-3.3.1-bin-hadoop3.rar است. سپس مسیرهای پیش فرض را در EnvironMent Variables اضافه میکنیم برای این کار مراحل زیر را انجام میدهیم:

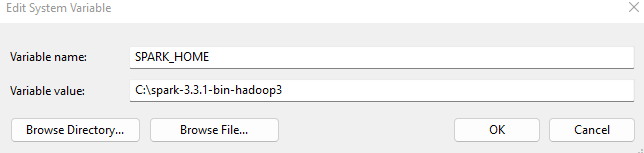
ابتدا system properties را باز میکنیم

روی دکمه Environment Variables کلیک کرده در قسمت پایین دکمه new را میزنیم:

و سپس مسیر ها را به صورت زیر اضافه میکنیم:







نکته variable value ها با توجه به مسیر نصب برای هر کس ممکن است متفاوت باشد.

برای استفاده از پوسته اسپارک در Command Prompt دستور spark-shell را اجرا میکنیم و سپس دستور pip install pyspark را در همین محیط، برای نصب pyspark اجرا میکنیم.

سپس برای استفاده از کد موجود در github از آن یک شبیه سازی(Clone) در درایو c ایجاد میکنیم. با اجرای دستور cd c:/ در Command Prompt خط دستور(Command Line) را به درایو c منتقل میکنیم و دستور زیر را در آن اجرا مکنیم:

Git clone ssh address in github for this project

سپس برای استفاده از کد شبیه سازی شده در درایو c دستورات زیر را به ترتیب در Command Prompt اجرا میکنیم:

c:\>cd spark\_project

c:\spark\_project>code .

و برای استفاده از github desktop دستور زیر را اجرا میکنیم:

c:\spark\_project>github .

## مقایسه پلتفرم‌های Databricks و Zeppelin

پس از بحث و مشورت اعضای گروه، اولین راه حلی که برای رفع مشکل databricks پیشنهاد شد، پیدا کردن پلتفرمی مشابه و در دسترس بود.

Databricks یک شرکت نرم افزاری آمریکایی است که توسط سازندگان آپاچی اسپارک تأسیس شده است. Databricks یک پلت فرم مبتنی بر وب برای کار با Spark ایجاد می کند که مدیریت خودکار خوشه و نوت بوک های سبک IPython را ارائه می دهد.

این شرکت که در زمینه داده های بزرگ(BigData) فعالیت می کند، عمده ابزار هایی که توسعه می دهد را به صورت عمومی منتشر کرده است.

انبارهای داده (data warehouses) برای داده های ساختاریافته ایده آل هستند، اما برای رسیدگی به داده های بدون ساختار، داده های نیمه ساختار یافته و داده هایی با تنوع، سرعت و حجم بالا، دریاچه های داده (Data Lakes) معرفی شده اند.

پلتفرم Databricks Lakehouse بهترین ویژگی های دریاچه های داده و انبارهای داده را برای ارائه قابلیت اطمینان، حاکمیت قوی، انعطاف پذیری و پشتیبانی از یادگیری ماشینی، ترکیب می کند.

اما پلتفرم Apache Zeppelin یک web based notebook مشابه databricks است که ویژگی‌هایی مانند data exploration, visualization, sharing and collaboration را فراهم می‌کند. همچنین از زبان‌های python, scala, hive, sparksql, shell, markdown پشتیبانی می‌کند. مزیت اصلی آن منبع‌باز بودن است که می‌توان آن را بر روی سرور به صورت اختصاصی مستقر کرد.

Apache Spark یک سیستم محاسباتی خوشه ای سریع و همه منظوره است که APIهای سطح بالا را که از نمودارهای اجرای عمومی پشتیبانی می کند در جاوا، اسکالا، پایتون و R و یک موتور بهینه سازی شده ارائه می دهد. Apache Spark در Zeppelin پشتیبانی می شود.

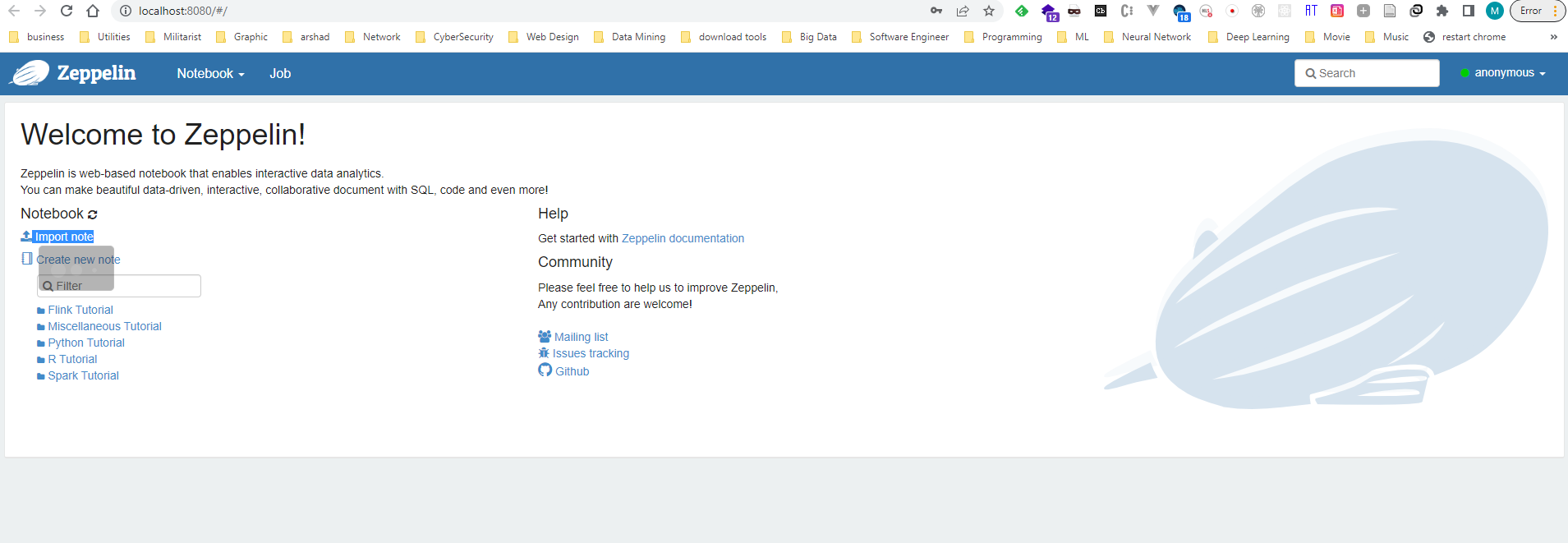
مقایسه zeppelin و Databricks:

|  |  |  |
| --- | --- | --- |
|  | **zeppelin** | **Databricks** |
| **setup** | خود میزبان(Self-hosted) | خود میزبان یا کاملا مدیریت شده |
| **JUPYTER COMPATIBILITY** | ندارد | سازگار با ژوپیتر(JUPYTER) |
| **PROGRAMMING LANGUAGES** | Python, SQL, Spark | JUPYTER |
| **DATA VISUALIZATION** | ناشناخته | مصور سازی با UI |
| **COLLABORATIVE EDITING** | مبتنی بر فایل یا نا همزمان | بلادرنگ |
| **PRICING** | رایگان | رایگان و پولی |
| **LICENSE** | منبع باز | اختصاصی |

## نصب Apache Zeppelin

در این پروژه ما از طریق docker، این پلتفرم را با دستور زیر بر روی سیستم محلی(local) نصب و مستقر کردیم:

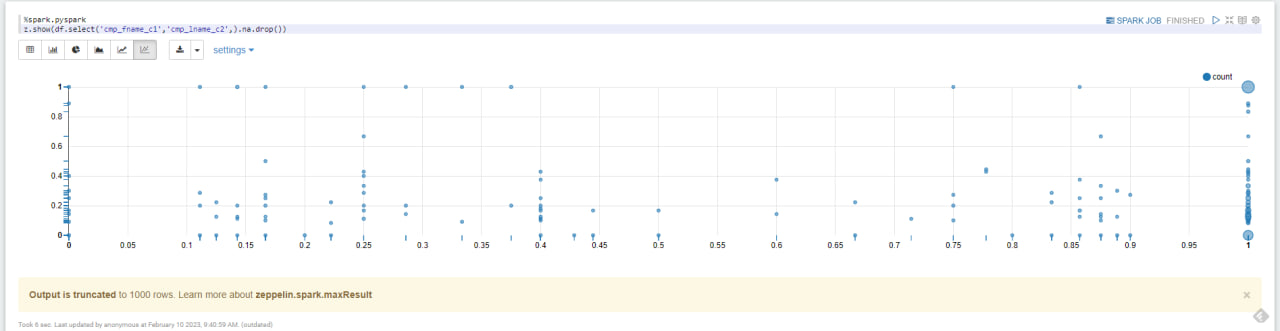
docker run -p 8080:8080 --rm --name zeppelin apache/zeppelin:0.10.1

پس از استقرار با مراجعه به آدرس localhost:8080 با صفحه اصلی zeppelin مواجه می‌شویم:

شما می‌توانید notebook ایجاد کنید و یا با گزینه import از notebook قبلا ساخته شده استفاده کنید. همچنین به دلیل اینکه از چندین زبان مختلف پشتیبانی می‌کند، در خط اول هر cell بهتر است نوع پردازش آن سلول را با % مشخص کنید.



پس از کار با zeppelin، مشاهده شد که مصورسازی با محدودیت 1000 ردیف مواجه است که با تغییر تنظیمات این مورد برطرف شد. اما با توجه به منابع محدودی که سیستم داشت، خروجی نمودارها به صورت truncated براساس بایت نمایش داده می‌شد.



به دلیل محدودیت‌های مصورسازی، به دنبال رفع مشکل 403 پلتفرم databricks اقدام کرده و به عنوان مرجعی برای مصورسازی از آن استفاده کردیم. کدهای عملیاتی تا قبل از این مرحله انجام شده و خروجی‌ها کامل بود.

# معرفی مجموعه داده و توصیف آن

## معرفی کلی مجموعه داده و نحوه شکل گیری آن

در این تحقیق قصد داریم یک پروژه یادگیری ماشینی را بر روی یک مجموعه کلان‏داده (Big data) توسط روش‏های پردازش کلان داده انجام دهیم. این مجموعه داده که از سایت Machine Learning Repository استخراج شده، داده های مربوط به تعداد زیادی از بیماران در یک مرکز تحقیقات بیماران سرطانی در آلمان به نام Epidemiologisches Krebsregister NRW است و در آدرس زیر قابل دسترس است:

https://archive.ics.uci.edu/ml/machine‐learning‐databases/00210/donation.zip

موضوع از این قرار است که اطلاعات 100،000 بیمار بین سال های 2005 تا 2008 توسط کارمندان مختلف مرکز تحقیقات ثبت شده است. مشکل اینجاست که اطلاعات یک فرد ممکن است دوبار توسط کارمندان ثبت شده باشد و مشکل اساسی تر اینکه در این بین اشتباهات تایپی نیز وجود دارد.

**بنابراین مسأله این مجموعه داده پیدا کردن جفت رکوردهایی از اطلاعات است که متعلق به یک شخص است.**

پس از آن در سال 2008 موسسه آمار زیستی پزشکی، اپیدمیولوژی و انفورماتیک (IMBEI) و مرکز پزشکی دانشگاه یوهانس گوتنبرگ، ماینتس، آلمان بر اساس 6 معیار که در ادامه می آید تمام جفت رکوردهای مربوط به این 100،000 رکورد (بیمار) را مورد بررسی قرار دادند.

معیارهای مذکور به شرح زیر هستند:

1. برابری آوایی نام و نام خانوادگی + برابری تاریخ تولد
2. برابری آوایی نام + برابری روز تولد (ماه و سال می توانند متفاوت باشند)
3. برابری آوایی نام + برابری ماه تولد (روز و سال می توانند متفاوت باشند)
4. برابری آوایی نام + برابری سال تولد (روز و ماه می توانند متفاوت باشند)
5. برابری تاریخ تولد
6. برابری آوایی نام خانوادگی + برابری جنسیت

برقراری هر کدام از شروط فوق منجر به انتخاب جفت رکورد شد. در نتیجه 5،749،132 جفت رکورد از میان تمام جفت رکوردها انتخاب شد که از این بین در 20،931 جفت رکورد، شخص اول و شخص دوم هر دو یک نفر بودند که برچسب match به آنها تعلق گرفت یعنی این جفت رکورد مربوط به یک شخص است و در مابقی جفت رکوردها شخص اول و شخص دوم دو فرد متفاوت بودند.

مشخص نیست برچسب گذاری به چه روشی انجام شده و چه سنجه هایی برای تعیین برچسب در نظر گرفته شده است اما به نظر می آید سنجه های زیادی در نظر گرفته شده و برچسب گذاری توسط روش های جمع‏سپاری انجام شده باشد.

این مجموعه داده به 10 فایل با اندازه تقریباً برابر تقسیم شده که در هر فایل نسبت زوج نفرات match و non-match تقریباً برابر است.

لازم به ذکر است در مجموعه داده اولیه هر رکورد متعلق به یک بیمار بود که شامل 100،000 رکورد بود ولی پس از بررسی جفت رکوردها مجموعه داده جدیدی تشکیل شد که هر رکورد متعلق به رکورد مجموعه داده اولیه است که باهم مقایسه می شوند که تعداد آنها 5،749،132 رکورد است.

از اینجا به بعد منظور از عبارت رکورد، هر سطر از مجموعه داده جدید (نهایی) است.

هدف از مجموعه داده نهایی یافتن یک مدل یادگیری ماشینی بر اساس کلان داده برای پیش بینی وضعیت تطابق دو شخص بر اساس ویژگی های آنهاست و اینطور به نظر می آید که علت استفاده از این معیارها کم کردن تعداد زوج نفرات مورد استفاده است. با توجه به تعداد رکوردها، تعداد کل زوج رکوردهای ممکن تقریباً 10 میلیارد خواهد شد که پردازش این تعداد بسیار زمان بر است و نیاز به منابع زیادی دارد. با وجود اینکه 6 معیار برای کاهش تعداد زوج نفرات استفاده شده باز هم تعداد رکوردها بسیار زیاد است و نیاز است از روش های پردازش کلان داده استفاده شود. پس از ساخت مدل با این تعداد رکورد می توان آن را برای بقیه زوج نفرات استفاده نمود.

در ادامه به معرفی این ویژگی ها می‏پردازیم:

مجموعه داده نهایی شامل 12 ویژگی است که 2 ویژگی اول آن شماره شناسایی دو شخص مورد مقایسه است که برای پیش‏بینی مورد استفاده قرار نمی‏گیرد 9 ویژگی، ویژگی های ورودی مورد استفاده جهت پیش بینی و 1 ویژگی، ویژگی خروجی است که match بودن یا نبودن دو شخص را مشخص می کند.

## معرفی ویژگی های ورودی

1. id\_1 :

شماره شناسایی شخص اول که یک متغیر عددی و از نوع صحیح است.

1. id\_2:

شماره شناسایی شخص دوم که یک متغیر عددی و از نوع صحیح است.

این دو ویژگی در ساخت مدل شرکت نمی کنند و در پیش پردازش از مجموعه داده حذف خواهند شد.

1. cmp\_fname\_c1:

میزان مشابهت جزء اول نام دو نفر که یک متغیر عددی پیوسته و از نوع اعشاری است.

1. cmp\_fname\_c2:

میزان مشابهت جزء دوم نام دو نفر که یک متغیر عددی پیوسته و از نوع اعشاری است.

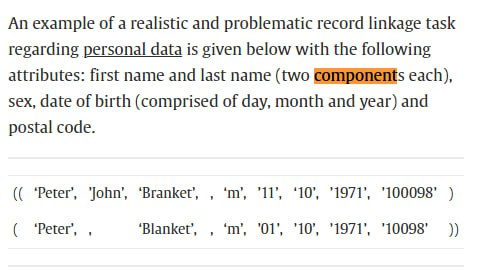
1. cmp\_lname\_c1:

میزان مشابهت جزء اول نام خانوادگی دو نفر که یک متغیر عددی پیوسته و از نوع اعشاری است.

1. cmp\_lname\_c2:

میزان مشابهت جزء دوم نام خانوادگی دو نفر که یک متغیر عددی پیوسته و از نوع اعشاری است.

(Sariyar *et al.*, 2012) در تحقیق خود اشاره کرده است که نام و نام خانوادگی در این مجموعه داده هر کدام دارای دو جزء هستند.



1. cmp\_sex:

تشابه جنسیت دو نفر که یک متغیر باینری است و مقدار آن 0 یا 1 است که 0 به معنی عدم یکسانی جنسیت و 1 به معنی یکسان بودن جنسیت دو نفر است.

1. cmp\_bd:

تشابه روز تولد دو نفر که یک متغیر باینری است و مقدار آن 0 یا 1 است که 0 به معنی عدم یکسانی روز تولد و 1 به معنی یکسان بودن روز تولد دو نفر است.

1. cmp\_bm:

تشابه ماه تولد دو نفر که یک متغیر باینری است و مقدار آن 0 یا 1 است که 0 به معنی عدم یکسانی ماه تولد و 1 به معنی یکسان بودن ماه تولد دو نفر است.

1. cmp\_by:

تشابه سال تولد دو نفر که یک متغیر باینری است و مقدار آن 0 یا 1 است که 0 به معنی عدم یکسانی سال تولد و 1 به معنی یکسان بودن سال تولد دو نفر است.

1. cmp\_plz:

تشابه کدپستی دو نفر که یک متغیر باینری است و مقدار آن 0 یا 1 است که 0 به معنی عدم یکسانی کدپستی و 1 به معنی یکسان بودن سال تولد دو نفر است.

9 ویژگی فوق، در ساخت مدل شرکت می کنند.

## ویژگی خروجی (هدف):

1. is\_match:

وضعیت تطابق دو نفر است که یک متغیر باینری است و مقدار آن True یا False است که True به معنی تطابق دو نفر و False به معنی عدم تطابق دو نفر است.

# تحلیل اکتشافی داده (**Exploratory data analysis**)

قرار هست خودم این قسمت را انجام بدهم 😊

## مجموعه داده در یک نگاه

# فراخوانی کتابخانه های مورد نیاز و انجام تنظیمات **pyspark**

ابتدا توسط کد زیر متدها و کتابخانه های مورد نیاز را فراخوانی نمودیم. همچنین تنظیمات اولیه مورد نیاز برای کتابخانه pyspark را انجام دادیم:

from pyspark.sql import SparkSession

from pyspark import SparkConf, SparkContext

from pyspark.sql.types import StructType,IntegerType,FloatType,BooleanType,StringType

from pyspark.sql.functions import rand, count, isnull, when, col

conf = SparkConf().setMaster("local[\*]").setAppName("My App")

sc = SparkContext.getOrCreate(conf = conf)

sc.\_conf.set('spark.executor.memory','15g')\

    .set('spark.driver.memory','15g')\

        .set('spark.driver.maxResultsSize','0')

spark=SparkSession.builder\

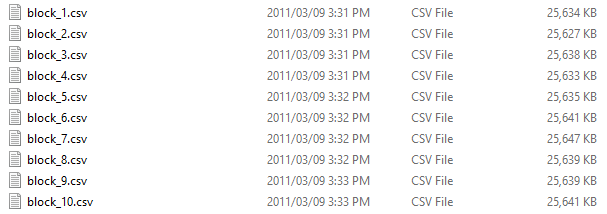
    .appName('myApp')\

        .config("spark.driver.memory", "15g")\

            .getOrCreate()

# تجمیع چانک های مجموعه داده

مجموعه داده شامل ده فایل با مشخصات زیر می باشد:



این چانک ها را توسط کد زیر تجمیع کرده و در یک متغیر دیتافریم ذخیره کردیم:

def load\_data(files,schema):

    df=spark.read.csv(files,header=True

                  ,schema=schema)

    return df

def load\_record\_linkage\_data():

    schema = StructType() \

      .add("id\_1",IntegerType(),True) \

      .add("id\_2",IntegerType(),True) \

      .add("cmp\_fname\_c1",FloatType(),True) \

      .add("cmp\_fname\_c2",FloatType(),True) \

      .add("cmp\_lname\_c1",FloatType(),True) \

      .add("cmp\_lname\_c2",FloatType(),True) \

      .add("cmp\_sex",IntegerType(),True) \

      .add("cmp\_bd",IntegerType(),True) \

      .add("cmp\_bm",IntegerType(),True) \

      .add("cmp\_by",IntegerType(),True) \

      .add("cmp\_plz",IntegerType(),True) \

      .add("is\_match",BooleanType(),False)

    files=[f'./data/block\_{id}.csv' for id in range(1,11)]

    return load\_data(files,schema=schema)

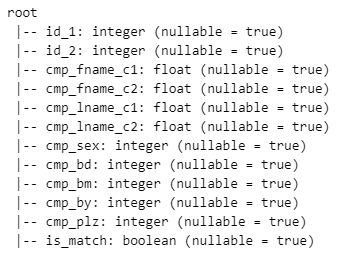
df=load\_record\_linkage\_data()

# مرحله پیش پردازش

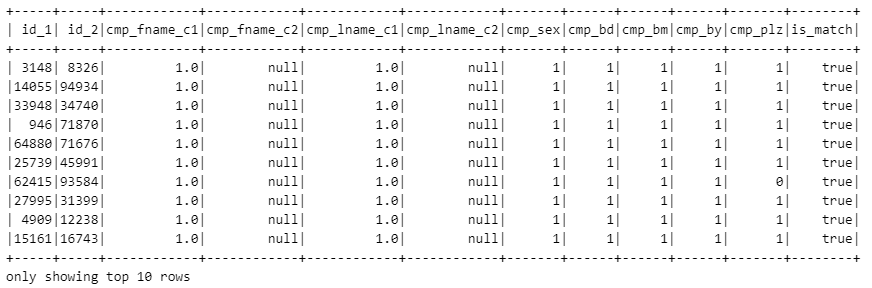
## تبدیل به داده‌های عددی

تمام ویژگی های این مجموعه داده از قبل عددی شده اند بجز خروجی که به صورت منطقی و باینری است:

df.printSchema()



df.show(10)



و آن را توسط تابع زیر عددی کردیم:

from pyspark.sql.functions import when, lit

def convert\_label\_binary(input\_df):

    temp = input\_df.withColumn('label',

                             when(input\_df['is\_match']==True,

                                  lit(1)).otherwise(0)

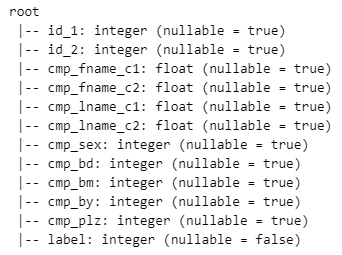
                                  )

    return temp

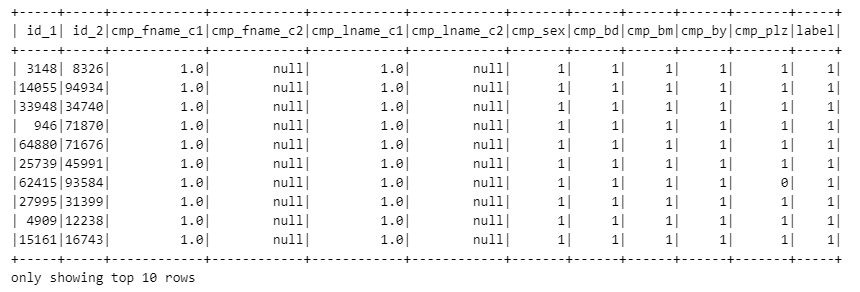
numerical\_df = convert\_label\_binary(df).drop('is\_match')

که نتیجه آن به صورت زیر می باشد:

numerical\_df.printSchema()



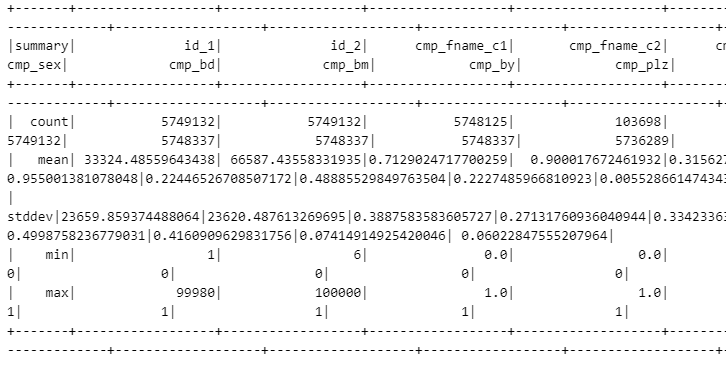
numerical\_df.show(10)



## نرمال سازی

ویژگی ها همه نرمال شده و بین 0 و 1 قرار دارند.

numerical\_df.describe().show()



## تشخیص و حذف داده های پرت

با توجه به اینکه همه ویژگی ها از جنس شباهت و بر اساس مقایسه بوده و پس از نرمال سازی در بازه معنی دار و معین 0 و 1 قرار دارند، لذا داده پرتی وجود ندارد.

# داده‏های از دست رفته (**Missing Values**)

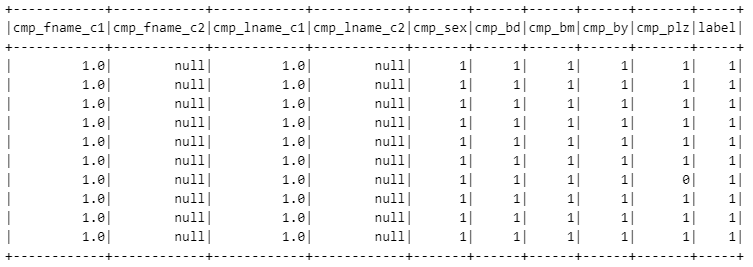
## بررسی داده‏های از دست رفته

از آنجا که ویژگی id برای هر فرد و ترکیب id\_1 و id\_2 برای هر رکورد منحصر به فرد است همانگونه که در بخش های قبلی به آن اشاره شد در ساخت مدل ارزشی ایجاد نمی کنند و مشارکت ندارند. لذا برای ایجاد مدل باید حذف شوند که این کار را با کد زیر انجام دادیم:

no\_id\_numerical\_df = numerical\_df.drop('id\_1','id\_2')

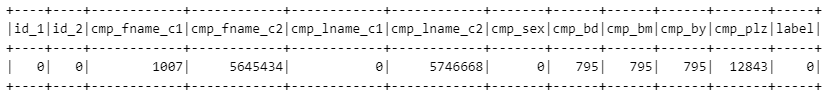
نتیجه آن به صورت زیر است:

no\_id\_numerical\_df.show(10)



توسط کد زیر وضعیت داده‏های از دست رفته را در مجموعه داده بررسی می کنیم:

numerical\_df.select([count(when(isnull(column), column)).alias(column) for column in numerical\_df.columns]).show()



اگر رکورد های شامل همه داده های از دست رفته را حذف کنیم تنها 20 رکورد باقی می ماند.

با توجه به تعداد کل رکوردها هیچ رکوردی که همه یا حداقل 2 ویژگی آن داده‏های از دست رفته داشته باشد وجود ندارد:

تعداد کل رکوردها:

no\_id\_numerical\_df.count()



تعداد رکوردهای باقیمانده در صورت حذف رکوردهایی که تمام ویژگی های آنها Null باشد:

no\_id\_numerical\_df.na.drop(how='all').count()



تعداد رکوردهای باقیمانده در صورت حذف رکوردهایی که حداقل 2 ویژگی آنها Null باشد:

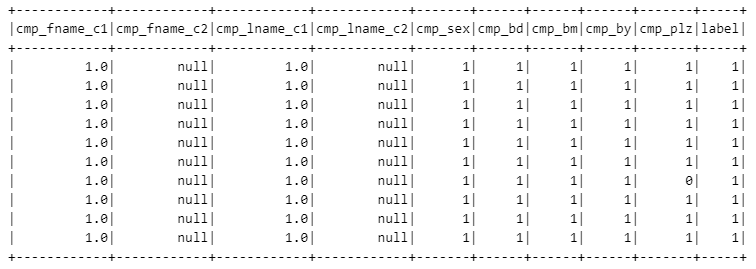
miss\_df.na.drop(how='any',thresh=2).count()



## پر کردن داده‏های از دست رفته

همانگونه که در قسمت قبل ذکر شد مجموعه داده دارای داده‏های از دست رفته زیادی مخصوصاً در ویژگی‏های cmp\_fname\_c2 و cmp\_lname\_c2 است:

no\_id\_numerical\_df.show(10)



نمونه داده قبل از پر کردن داده‏های از دست رفته

با استفاده از روش Imputer داده‏های از دست رفته را پر کردیم که کد آن به صورت زیر می باشد:

from pyspark.ml.feature import Imputer

from pyspark.ml.feature import Imputer

def fill\_missing\_values(input\_df):

    # for float variables

    miss\_df=input\_df.drop('id\_1','id\_2')

    miss\_df=miss\_df.replace('?',None)

    float\_cols=[

    'cmp\_fname\_c1',

    'cmp\_fname\_c2',

    'cmp\_lname\_c1',

    'cmp\_lname\_c2',

    ]

    float\_imputer = Imputer(

        inputCols=float\_cols,

        outputCols=[f"{col}\_imp" for col in float\_cols]

    ).setStrategy('mean')

    # for binary variables

    binary\_cols=[

        'cmp\_sex',

        'cmp\_bd',

        'cmp\_bm',

        'cmp\_by',

        'cmp\_plz',

    ]

    binary\_imputer = Imputer(

        inputCols=binary\_cols,

        outputCols=[f"{col}\_imp" for col in binary\_cols]

    ).setStrategy('mode')

    imputed\_df=float\_imputer.fit(miss\_df).transform(miss\_df)

    output\_df=binary\_imputer.fit(imputed\_df).transform(imputed\_df)

    output\_df=output\_df.select([x for x in output\_df.columns if '\_imp' in x or x=='is\_match'])

    return output\_df

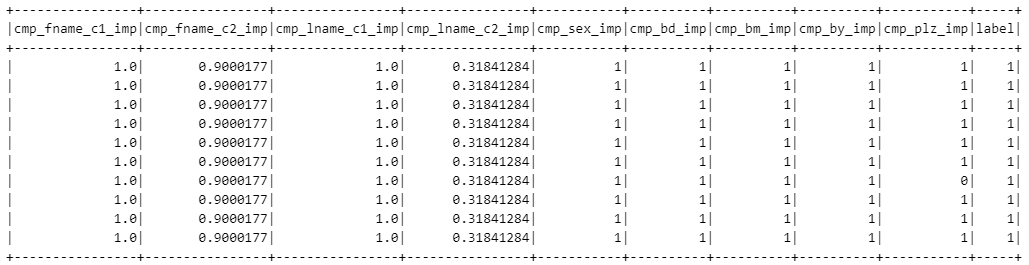
def preprocessing\_df(input\_df):

    output\_df = convert\_label\_binary(fill\_missing\_values(input\_df))

    return output\_df.drop('is\_match')

prep\_df=preprocessing\_df(df)

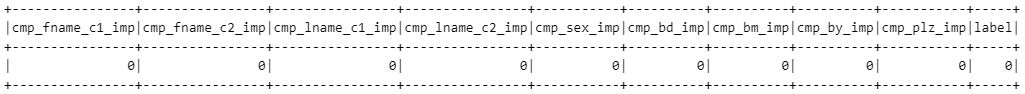
نتیجه اجرای کد فوق به صورت زیر است:



نمونه داده پس از پر کردن داده‏های از دست رفته

توسط کد زیر چک می‏کنیم که تمام داده‏های از دست رفته پر شده باشد:

prep\_df.select([count(when(isnull(column), column)).alias(column) for column in prep\_df.columns]).show()



همانطور که ملاحظه می‏شود هیچ داده از دست رفته ای برای هیچ کدام از ویژگی‏ها وجود ندارد.

# مهندسی ویژگی (**Feature Engineering**)

در این مرحله جهت آماده سازی داده برای ورود به مرحله مدل سازی لازم است تمام ویژگی های ورودی به صورت بردار در قالب یک ویژگی تجمیع شوند. در واقع پس از این مرحله مجموعه داده دارای دو ستون خواهد بود: ستون اول تمام ویژگی های ورودی که به برداری از ویژگی ها تبدیل شده اند و ستون دوم خروجی یا برچسب هر رکورد. کد این مرحله به صورت زیر می باشد:

from pyspark.ml.feature import VectorAssembler

def feature\_engineering(input\_df,feature\_list,label\_name):

    assembler = VectorAssembler(inputCols=feature\_list,

                             outputCol='features')

    assembled\_df = assembler.transform(input\_df)

    output\_df=assembled\_df.select('features', label\_name)

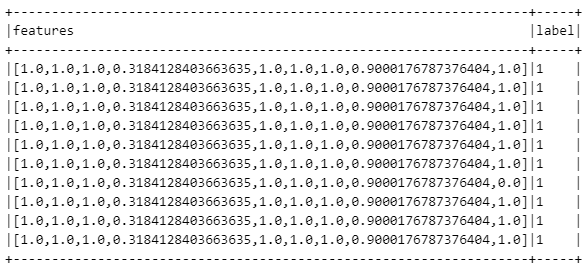
    return output\_df

input\_features=list(set(prep\_df.columns) - set(['label','is\_match']))

assembled\_df = feature\_engineering(prep\_df,input\_features,'label')

خروجی مجموعه داده پس از انجام مهندسی ویژگی به صورت زیر است:

assembled\_df.show(10, truncate=False)



# مدل‌سازی

در این پروژه سه الگوریتم زیر برای ساخت مدل یادگیری ماشینی در نظر گرفته شده است:

1. Logistic Regression
2. Decision Tree
3. Random Forest

قالب اصلی آموزش و پیش‌بینی مدل‌ها به صورت زیر است:

estim=[Estimator](featuresCol='features', labelCol='label')

pipeline = Pipeline(stages=[estim])

model = pipeline.fit(train)

result = model.transform(test)

در مراحل قبل، تمام ویژگی‌های مورد نیاز را در ستون features به صورت بردار تجمیع کرده‌ایم. همچنین خروجی مورد انتظار پس از تبدیل به باینری در ستون label نگه‌داری می‌شود. برای استفاده از حداکثر امکانات pyspark از pipeline استفاده می‌کنیم تا بعدا در صورتی که نیاز به cross\_validation یا پیش‌پردازش‌های خاصی بود، به صورت pipeline انجام شود. برای آموزش از متد fit و برای پیش‌بینی روی داده‌های آزمون از متد transform استفاده می‌کنیم.

## روش رگرسیون منطقی (**Logistic Regression**)

در این روش که برای طبقه‌بندی باینری به کار می‌رود، به دنبال پیش‌بینی طبقه براساس متغیرهای وابسته آن است. مفروض اصلی آن، خطی بودن متغیرهای مستقل است. مزیت این روش، استفاده ساده آن است و عملکرد بالایی که برای داده‌های تفکیک‌پذیر خطی دارد. برای پیاده‌سازی با pyspark از کد زیر استفاده می‌کنیم:

lr=LogisticRegression(featuresCol='features', labelCol='label')

pipeline = Pipeline(stages=[lr])

model = pipeline.fit(train)

lr\_result = model.transform(test)

## روش درخت تصمیم (**Decision Tree**)

درخت تصمیم نیز مانند رگرسیون منطقی برای داده‌های با تفکیک‌پذیر خطی مناسب است. یکی از مفروضات اصلی درخت تصمیم، در نظر گرفتن ویژگی‌های طبقه‌ای است و در صورتی که همه یا بعضی از ویژگی‌ها پیوسته باشند به طبقه‌ای تبدیل می‌کند. ایده این روش، تولید دستورالعمل‌های شرطی برای تقسیم داده‌ها براساس خلوص است تا به جایی برسیم که خلوص برگ‌ها بیشینه شود؛ یعنی هر برگ این درخت نماینده یک طبقه باشد. کد زیر برای پیاده‌سازی درخت تصمیم به کار رفته است:

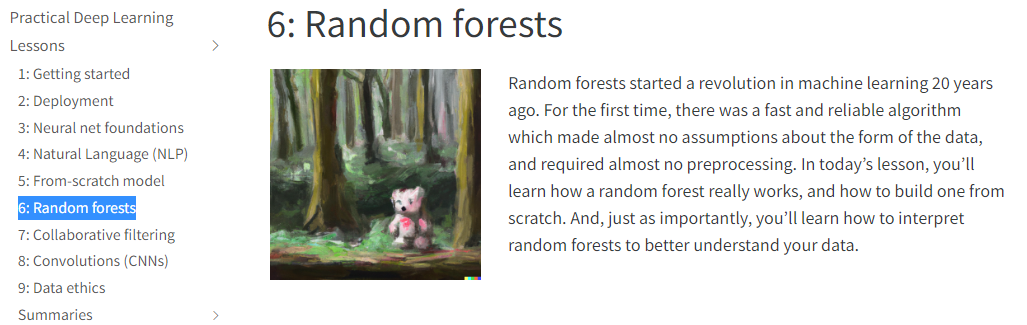
tree = DecisionTreeClassifier()

tree\_pipeline = Pipeline(stages=[tree])

tree\_model = tree\_pipeline.fit(train)

tree\_result = tree\_model.transform(test)

## روش جنگل تصادفی (**Random Forest**)

در این روش، از چندین درخت تصمیم استفاده می‌کنیم و جواب آن‌ها را با هم تجمیع می‌کنیم. در این روش برخلاف روش‌های قبلی هیچ مفروضی بر روی داده نداریم. اهمیت و عملکرد این روش از دیرباز قابل تحسین بوده و دسته‌بندی آن به عنوان روش آماری یا هوشمند محل بحث است؛ به طوری که در دوره آموزش یادگیری عمیق آقای جرمی هاوارد، روش جنگل تصادفی نیز تدریس می‌شود.

کد پیاده‌سازی جنگل تصادفی به صورت زیر است:

from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(labelCol="label",

                            featuresCol="features")

rf\_pipeline = Pipeline(stages=[ rf])

rf\_model=rf\_pipeline.fit(train)

rf\_result=rf\_model.transform(test)

## روش طبقه‌بند ماشین بردار پشتیبان خطی (Linear SVC)

در این روش، به دنبال تفکیک داده‌های هرکلاس با خط یا ابرصفحه هستیم. یعنی در نهایت، معادله خط یا ابرصفحه بدست می‌آید و فاصله برداری نقاط نسبت به آن‌ها محاسبه می‌شود.

با کد زیر می‌توانیم از پیاده‌سازی LinearSVC در pyspark استفاده کنیم:

from pyspark.ml.classification import LinearSVC

svc = LinearSVC(labelCol="label", featuresCol="features")

svc\_pipeline = Pipeline(stages=[ svc])

svc\_model=svc\_pipeline.fit(train)

svc\_result=svc\_model.transform(test)

evaluate(svc\_result,

         model\_name='SVC',

         rawPredictionCol='rawPrediction',

         require\_auc=False)

# ارزیابی

اولین قدم در ارزیابی یک مدل تعیین سنجه مناسب بر اساس اهمیت داده های FP و FN است. اگر شرایط مسأله طوری باشد که تشخیص درست داده های کلاس مثبت اهمیت بیشتری نسبت به داده های کلاس منفی داشته باشد معنی آن این است که ارزیابی مدل باید بر اساس کم شدن مقدار FP انجام شود که با توجه به نحوه محاسبه سنجه های مختلف، سنجه دقت (precision) اهمیت بیشتری پیدا می کند زیرا مقدار FP در کسر قرار دارد که کمتر شدن مقدار آن باعث بیشتر شدن مقدار این سنجه می شود. مشابه همین تحلیل برای FN وجود دارد که منجر به اهمیت سنجه یادآوری (Recall) می شود.

اینکه در یک مجموعه داده کدام یک از مقادیر FP یا FN اهمیت بیشتری دارد بستگی به شرایط و نوع مجموعه داده دارد. در مورد مجموعه داده این مسأله FP یعنی داده هایی که توسط مدل به اشتباه در کلاس مثبت قرار گرفته اند و FN یعنی داده هایی که توسط مدل به اشتباه در کلاس منفی قرار گرفته اند. داده FP یعنی دو شخص متفاوت به عنوان یک شخص توسط مدل شناسایی شده و برچسب match خورده است و داده FN یک شخص است که توسط مدل به عنوان دو شخص متفاوت شناسایی شده و برچسب non-match گرفته است. در مورد FP از دست دادن داده (data loss) اتفاق می افتد و در مورد FN تکرار داده (data duplication) اتفاق می افتد. واضح است که از دست دادن داده بحرانی تر است و باید سعی کنیم حتی الامکان از وقوع این اتفاق جلوگیری کرده و FP را تا جای ممکن کاهش دهیم. لذا با توجه به مطالب فوق سنجه اصلی برای ارزیابی مدل باید دقت (precision) باشد.

اولین چالش در ارزیابی مدل، نامتوازن بودن مجموعه داده است. برای مواجهه به داده‌های نامتوازن روش‌های متداول زیر توصیه می‌شود

1. استفاده از سنجه‌هایی مثل Macro F1, Weighted F1، AUC score
2. نمونه‌گیری متوازن و استفاده از متریک‌های معمولی مثل accuracy
3. وزن دهی متناسب به loss که در هنگام ساخت مدل وزن‌دهی را تعریف کنیم

## سنجه **Macro F1**

در حالت کلی سنجه F1 میانگین هارمونیک دو سنجه دقت (precision) و یادآوری (recall) است. برای رفع ابهام، نام‌های دیگر این سنجه همراه محاسبه آن آورده شده است:

## سنجه دقت (precision) یا positive predictive value (PPV)

## سنجه یادآوری (Recall) یا sensitivity یا hit rateیا true positive rate (TPR)

به زبان ساده، سنجه دقت نشان می‌دهد که از بین پیش‌بینی‌های مدل برای کلاس True، چه نسبتی صحیح پیش‌بینی شده است. سنجه یادآوری هم نشان می‌دهد از بین تمام داده‌های کلاس True، چه نسبتی درست پیش‌بینی شده است. با توجه به اینکه مقایسه دو سنجه از یک مدل با دو سنجه از یک مدل دیگر کار دشواری است، یک سنجه F1 که میانگین هارمونیک این دو است معرفی شده است که به صورت زیر است:

تمام این سنجه‌ها در حالت عالی نمایانگر قدرت پیش‌بینی مدل است. اما وقتی داده نامتوازن باشد، باید از حالت خاصی از F1 یعنی macro F1 استفاده کرد. سنجه Macro F1 میانگین غیر وزن‌دار F1 برای تمام کلاس‌ها را اندازه می‌گیرد(N تعداد کلاس‌ها است).

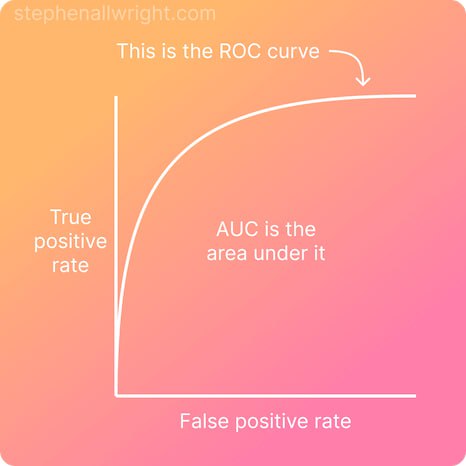
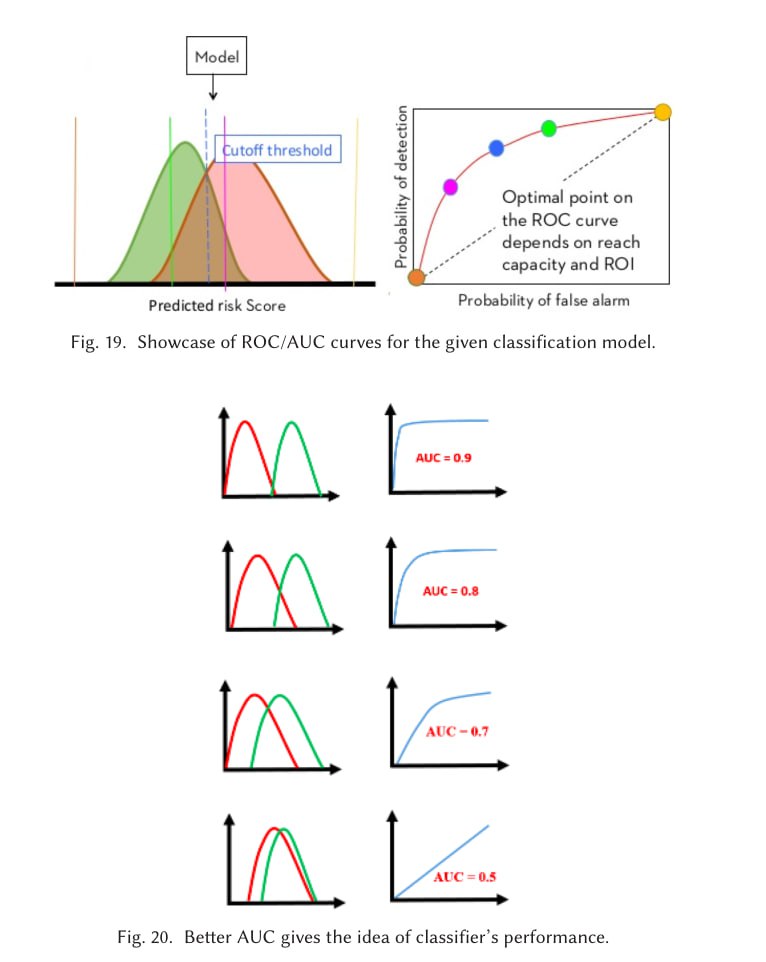
حالت خاصی از این سنجه به نام Weighted F1 وجود دارد که معیار support هر طبقه را نیز به f1 آن طبقه ضرب کرده و سپس میانگین را حساب می‌کند.

## سنجه **AUC score**

قبل از اینکه به امتیاز AUC بپردازیم باید با منحنی ROC آشنا شویم. منحنی ROC تغییرات نرخ مثبت‌های واقعی به نرخ مثبت‌های کاذب را در طول آستانه‌های مختلف نشان می‌دهد.

نمودار ROC براساس تغییرات نرخ‌ها می‌تواند حالت‌های مختلفی داشته باشد. هرچقدر نمودار بالاتر باشد نشان‌دهنده عملکرد بهتر طبقه‌بند است که داده‌های طبقات را بهتر تفکیک کرده است.

برای محاسبه نمودار بهتر از معیار امتیاز AUC یا مساحت زیر منحنی استفاده می‌کنیم تا شهودی که مطرح شد را به صورت معیار کمیتی تبدیل کنیم.



برای ارزیابی دو رویکرد مختلف پیاده‌سازی شده است. در رویکرد اول، ابتدا به صورت دستی سنجه‌ها را از پیش‌بینی‌ها محاسبه می‌کنیم. تابع evaluate\_from\_scratch برای محاسبه دستی سنجه‌های ارزیابی استفاده می‌شود:

def evaluate\_from\_scratch(pred,

                          model\_name='Logistic Regression'):

    pred.groupBy('label', 'prediction').count().show()

    # Calculate the elements of the confusion matrix

    TN = pred.filter('prediction = 0 AND label = prediction').count()

    TP = pred.filter('prediction = 1 AND label = prediction').count()

    FN = pred.filter('prediction = 0 AND label = 1').count()

    FP = pred.filter('prediction = 1 AND label = 0').count()

    # Accuracy measures the proportion of correct predictions

    accuracy = (TN + TP) / (TN + TP + FN + FP)

    recall = (TP) / (TP+FN)

    precision= (TP) / (TP+FP)

    f1=2\*(precision\*recall)/(precision+recall)

    print(f'EVALUATION SUMMARY for {model\_name}:')

    print(f" accuracy:{accuracy}")

    print(f" precision:{precision}")

    print(f" recall:{recall}")

    print(f" f1-score:{f1}")

با استفاده کد بالا، سنجه‌های مورد نیاز را به صورت دستی با استفاده از ماتریس آشفتگی بدست می‌آوریم.

برای رویکرد دوم، از متدهای pyspark استفاده می‌کنیم. تابع evaluate\_from\_spark ، تمام سنجه‌ها را با استفاده از متدهای pyspark محاسبه می‌کند و منحنی ROC را نیز نمایش می‌دهد.

def evaluate\_from\_spark(predictions,

                        model\_name='Logistic Regression',

                        rawPredictionCol="probability",

                        labelCol="label",

                        predictionCol="prediction",

                        require\_auc=True

                        ):

    print(rawPredictionCol)

    eval2= MulticlassClassificationEvaluator(predictionCol=predictionCol,

                                              labelCol=labelCol)

    ACC  = eval2.evaluate(predictions,

                          {eval2.metricName:"accuracy"})

    PREC  = eval2.evaluate(predictions,

                           {eval2.metricName:"weightedPrecision"})

    REC  = eval2.evaluate(predictions,

                          {eval2.metricName:"weightedRecall"})

    F1  = eval2.evaluate(predictions,

                         {eval2.metricName:"f1"})

    WeightedFMeasure=eval2.evaluate(predictions,

                                    {eval2.metricName:"weightedFMeasure"})

    print(f"{model\_name} Performance Measure")

    print(" Accuracy = %0.8f" % ACC)

    print(" Weighted Precision = %0.8f" % PREC)

    print(" Weighted Recall = %0.8f" % REC)

    print(" F1 = %0.8f" % F1)

    print(" Weighted F Measure = %0.8f" % WeightedFMeasure)

    if require\_auc:

        eval = BinaryClassificationEvaluator(rawPredictionCol=rawPredictionCol,

                                             labelCol=labelCol)

        AUC  = eval.evaluate(predictions)

        print(" AUC = %.8f" % AUC)

        print(" ROC curve:")

        PredAndLabels           = predictions.select(rawPredictionCol,

                                                     labelCol)

        PredAndLabels\_collect   = PredAndLabels.collect()

        PredAndLabels\_list      = [(float(i[0][0]), 1.0-float(i[1])) for i in PredAndLabels\_collect]

        PredAndLabels           = sc.parallelize(PredAndLabels\_list)

        fpr = dict()

# FPR: False Positive Rate

        tpr = dict()

# TPR: True Positive Rate

        roc\_auc = dict()

        y\_test = [i[1] for i in PredAndLabels\_list]

        y\_score = [i[0] for i in PredAndLabels\_list]

        fpr, tpr, \_ = roc\_curve(y\_test, y\_score)

        roc\_auc = auc(fpr, tpr)

        plt.figure(figsize=(5,4))

        plt.plot(fpr, tpr, label='ROC curve (area = %0.8f)' % roc\_auc)

        plt.plot([0, 1], [0, 1], 'r--')

        # plt.xlim([0.0, 1.0])

        # plt.ylim([0.0, 1.05])

        plt.yticks(np.arange(0,1.03,0.1))

        plt.xlabel('False Positive Rate')

        plt.ylabel('True Positive Rate')

        plt.title(f'ROC Curve - {model\_name}')

        plt.legend(loc="lower right")

        plt.show()

برای تجمیع توابع بالا از تابع evaluate استفاده می‌کنیم تا خروجی به صورت یکپارچه چاپ شود.

def evaluate(predictions,

             model\_name=None,

             rawPredictionCol="probability",

             labelCol="label",

             predictionCol="prediction",

             require\_auc=True

             ):

    print('Evaluate From Scratch:')

    evaluate\_from\_scratch(predictions,model\_name)

    print('\nEvaluate From Spark Library:')

    evaluate\_from\_spark(predictions,

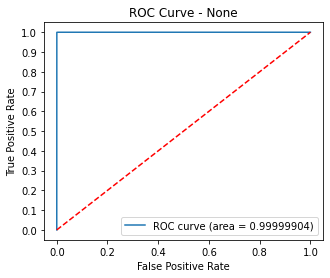
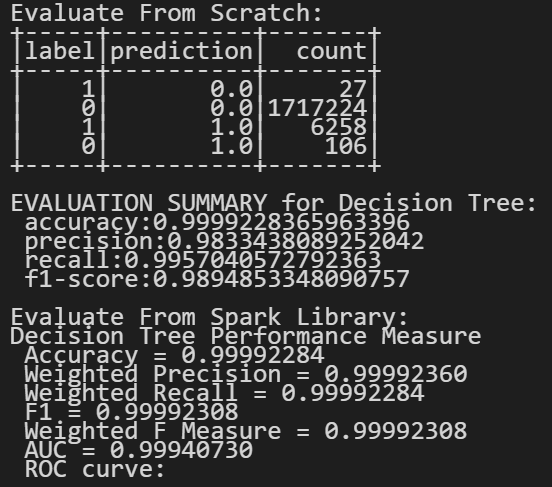
                        model\_name,

                        rawPredictionCol=rawPredictionCol,

                        labelCol=labelCol,

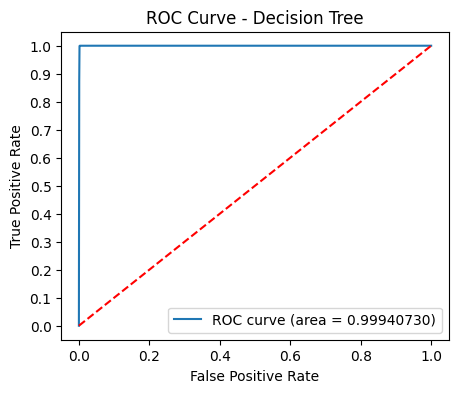
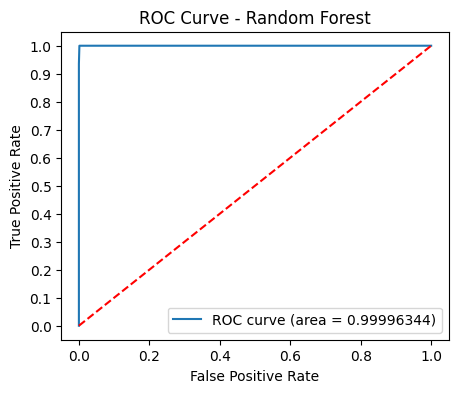
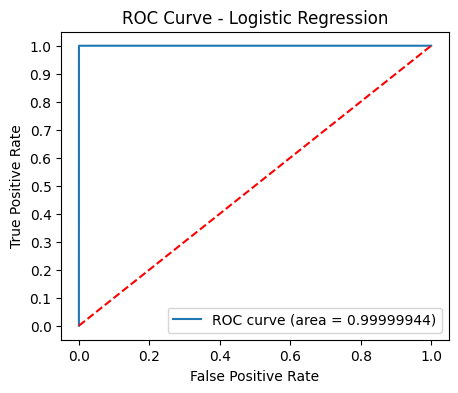
                        predictionCol=predictionCol,

                        require\_auc=require\_auc)

نمونه خروجی ارزیابی یک مدل به صورت زیر است:

در جدول زیر سنجه‌های ارزیابی برای مدل‌های مختلف آورده شده است:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **سنجه** | **رگرسیون منطقی** | **درخت تصمیم** | **جنگل تصادفی** | **SVC خطی** |
| **Accuracy** | 0.99998376 | 0.99992284 | 0.99982305 | 0.99985322 |
| **Precision** | 0.99840688 | 0.98334381 | 0.99983283 | 0.99848841 |
| **Recall** | 0.99713604 | 0.99570406 | 0.95163087 | 0.96057521 |
| **Weighted F1** | 0.99998375 | 0.99992308 | 0.99982087 | 0.99985180 |
| **AUC score** | 0.99999943 | 0.99940730 | 0.99996344 |  |

همچنین منحنی ROC به تفکیک مدل‌ها را نیز به صورت زیر رسم شده است:

# مقایسه عملکرد مدل‌ها

برای نشان دادن تفاوت عملکرد مدل‌ها به صورت ملموس‌تر، نمودار میله‌ای عملکرد آن‌ها را به صورت زیر رسم می‌کنیم. مشاهده می‌شود که مدل رگرسیون منطقی از صحت و f1 بالاتری برخوردار است. همچنین امتیاز AUC آن نیز بیشتر بوده که به معنی طبقه‌بندی قابل اعتماد‌تر است.