دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش تمرین دوم داده کاوی

Classification With Spark

پدیدآورنده:

محمد امین کیانی

4003613052

دانشجوی کارشناسی، دانشکده‌ی کامپیوتر، دانشگاه اصفهان، اصفهان،

استاد درس: جناب اقای دکتر کیانی

نیمسال دوم تحصیلی 04-1403

فهرست مطالب

مستندات

بخش‌های اصلی پروژه

یک نوع مجموعه داده چند متغیره است که از ۱۴ ویژگی زیر تشکیل شده است:

سن، جنس، نوع درد قفسه سینه، فشار خون در حالت استراحت، کلسترول سرم، قند خون ناشتا، نتایج الکتروکاردیوگرافی در حالت استراحت، حداکثر ضربان قلب به دست آمده، آنژین ناشی از ورزش، اوج ضربان قلب - افت ST ناشی از ورزش نسبت به استراحت، شیب اوج قطعه ST در ورزش، تعداد عروق اصلی و تالاسمی.

این پایگاه داده شامل ۷۶ ویژگی است، اما تمام مطالعات منتشر شده مربوط به استفاده از زیرمجموعه‌ای از ۱۴ مورد از آنها است. پایگاه داده کلیولند تنها پایگاه داده‌ای است که تا به امروز توسط محققان یادگیری ماشینی استفاده شده است. یکی از وظایف اصلی در این مجموعه داده، پیش‌بینی بر اساس ویژگی‌های داده شده یک بیمار است که آیا آن فرد خاص بیماری قلبی دارد یا خیر و وظیفه دیگر، تشخیص و یافتن بینش‌های مختلف از این مجموعه داده است.

مشخصات سخت‌افزاری:

- CPU: Intel Core i5-8th Gen
- RAM: 12GB
- GPU: RADEON (4GB)
- سیستم‌عامل : win10

اجرای لوکال(local mode) : کل کد Spark روی یک JVM و یک سیستم اجرا می‌شود.

اجرای کلاستر(cluster mode) : وظایف بین چند Worker (معمولاً سیستم‌های جداگانه یا در اینجا داخل ماشین‌های مجازی) توزیع می‌شوند.

ویژگی‌های قابل مشاهده:

- ستون‌های cp, restecg, slope, thal شامل **مقادیر متنی (categorical)** هستند.
- ستون‌های sex, fbs, exang هم به صورت متنی (TRUE/FALSE, Male/Female) هستند که باید به عدد تبدیل شوند.
- بعضی ردیف‌ها **missing value** دارند (جاهای خالی).

پیش پردازش لازم:

1 . حذف ستون‌های غیر ضروری:

- id, dataset تأثیری در مدل ندارند.
  - اما آیا حذف ستون dataset درست است؟
    - بهتر است حذف نشود. این ستون اطلاعاتی درباره‌ی منبع (مثل Cleveland یا VA Long Beach) دارد و ممکن است بعضی دیتاست‌ها ویژگی‌های خاصی داشته باشند واین اطلاعات می‌تواند برای مدل ارزشمند باشد.
    - پس باید : تبدیل آن به عدد با StringIndexer

2 . تبدیل مقادیر متنی به عددی:

- با StringIndexer برای ستون‌های:
  - sex, cp, restecg, slope, thal, fbs, exang
  - ولی ستون fbs و exang از نوع BooleanType هستند TRUE/FALSE ولی StringIndexer فقط با StringType یا NumericType کار می‌کند.

3 . حذف ردیف‌های ناقص:

- با ()dropna. یا فقط روی رکوردهایی که کامل هستند.

4 . تبدیل ستون هدف (num) به کلاس باینری یا چندکلاسه:

- اگر فقط 0 و 1 باشد: باینری
- اگر تا 4 داشته باشد: multi-class classification
- ستون num کلاس هدف (label) ماست.
- در دیتاست بیماری قلبیUCI ، مقدار num به شکل زیر تفسیر می‌شود:

مقدار	معنا
0	بدون بیماری قلبی (سالم)
1	وجود بیماری خفیف
2	بیماری متوسط
3	بیماری شدید
4	بیماری بسیار شدید

این یک مسئله‌ی classification چندکلاسه (multi-class classification) هست.

- ولی در بعضی مسائل برای بهبود نتایج آن را به 1/0 باینری تبدیل می‌کنند:

```
df = df.withColumn("label", (df["num"] > 0).cast("integer"))
```

- در این پروژه بهتراست **چندکلاسه نگه‌داریم اما هر دو حالت بررسی شده است.**

نتایج پیش‌پردازش:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("redwankarimsony/heart-disease-data")

print("Path to dataset files:", path)

df = spark.read.csv("/content/heart_disease_uci.csv", header=True,
inferSchema=True)
df.printSchema()
df.show(5)

from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml import Pipeline

df = df.dropna()
categorical_cols = []
numerical_cols = [col for col in df.columns if col != 'target']

assembler = VectorAssembler(inputCols=numerical_cols,
outputCol="features")
pipeline = Pipeline(stages=[assembler])
df_transformed = pipeline.fit(df).transform(df)

df_final = df_transformed.select("features", "target")
df_final.show(5)

from pyspark.sql.functions import col

# تبدیل ستون‌های boolean به رشته مرحله 0:
df = df.withColumn("fbs", col("fbs").cast("string"))
df = df.withColumn("exang", col("exang").cast("string"))

# حذف ردیف‌های ناقص
df = df.dropna()
```

```
# لیست ستون‌های دسته‌ای (categorical) که باید index بشن
categorical_cols = ["sex", "cp", "restecg", "slope", "thal", "fbs",
"exang", "dataset"]

# ستون‌های عددی (numeric)
numeric_cols = ["age", "trestbps", "chol", "thalch", "oldpeak", "ca"]

# کردن index مراحل
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml import Pipeline

indexers = [StringIndexer(inputCol=col, outputCol=col + "_idx") for col in
categorical_cols]

# ترکیب featureها
assembler = VectorAssembler(
    inputCols=[col + "_idx" for col in categorical_cols] + numeric_cols,
    outputCol="features"
)

# pipeline ساخت
pipeline = Pipeline(stages=indexers + [assembler])

# اجرا
df_prepared = pipeline.fit(df).transform(df)

# نهایی‌سازی دیتافریم
df_final = df_prepared.select("features", col("num").alias("label"))
df_final.show(5)
```

```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|[0.0,3.0,1.0,2.0,...|    0|
|(14,[2,6,8,9,10,1...|    2|
|[0.0,0.0,1.0,0.0,...|    1|
|(14,[1,3,8,9,10,1...|    0|
|[1.0,2.0,1.0,1.0,...|    0|
+--------------------+-----+
only showing top 5 rows
```

1 . features

- بردار ویژگی‌ها (feature vector) هست که شامل ترکیب تمام ستون‌های categorical و numeric است.
- در Spark ، این بردار به صورت DenseVector یا SparseVector ذخیره می‌شود:
  - [0.0,3.0,1.0,2.0,...] → DenseVector
  - (14,[2,6,8,9,10,11],[1.0,...]) → SparseVector با طول 14، فقط مقادیر غیر صفر رو نشان می‌دهد (برای بهینگی)

2 . label

- مقدار ستون num یا همان ستون هدف
- این ردیف متعلق به کدام کلاس از بیماری هاست:
  - 0 : سالم
  - 1تا 4 : شدت بیماری
- اما اکثر پیش‌بینی‌ها 0.0هستند یعنی مدل ممکن است**bias** داشته باشد به سمت کلاس 0 (سالم)

دلیل :

- داده‌های آموزشی به شدت **نامتوازن** هستند (خیلی بیشتر label=0 داریم نسبت به 1..4) و Random Forest هم گرایش به کلاس اکثریت را بیشتر حدس زدن دارد.

استفاده از Weighted Random Forest یا الگوریتم‌های مقاوم به داده‌های نامتوازن مثل:

- DecisionTreeClassifier (اما نتایج آن ضعیف تر بود)
- GBTClassifier (مخصوص دو کلاسه ها)

استفاده از تکنیک‌های balancing مثل:

- Undersampling کلاس 0 که سبب کاهش دیتا شد و مدل را حتی ضعیف تر کرد.
- Oversampling کلاس‌های 1 تا 4 که سبب اگمنت و افزایش دیتا شد و مدل را به شدت بهبود و تقویت بخشید.

مراحل کار با اسپارک:

```
assembler = VectorAssembler(inputCols=numerical_cols,
outputCol="features")

pipeline = Pipeline(stages=[assembler])

df_transformed = pipeline.fit(df).transform(df)
```

VectorAssembler : تمام ستون‌های عددی را به یک وکتور ویژگی واحد تبدیل می‌کند.
Pipeline : به سبکScikit-learn ، مراحل متوالی را اجرا می‌کند.
()fit : مدل را روی داده یاد می‌گیرد (در اینجا فقط ساختار)
()transform : تبدیل روی دیتافریم اجرا می‌شود

- همه‌ی این عملیات‌ها در Spark به صورت **lazy** انجام می‌شوند یعنی تا زمانی که ()show. یا ()collect. یا آموزش مدل اجرا نشود، فقط plan ذخیره می‌شود.
- Pipeline ها برای مدیریت مراحل پردازش داده‌ها ساخته شده‌اند و می‌توانند روی داده‌های خیلی بزرگ بدون نیاز به حافظه‌ی زیاد کار کنند.
- VectorAssembler و StringIndexer روی RDD زیرین اجرا می‌شوند و داده‌ها را به DenseVector یا SparseVector تبدیل می‌کنند که قابل فهم برای مدل‌های ML هستند.
- در Spark (و اصولاً در علم داده)، transform یعنی:
  - اعمال تغییرات مشخص‌شده (مثل تبدیل، ترکیب، یا فیلتر کردن) روی یک دیتافریم و تولید یک دیتافریم جدید با داده‌های تغییریافته.
  - StringIndexer با .fit() یاد می‌گیرد که مثلاً مقدار "male" باید 0 و "female" باید 1 باشد.
  - transform تغییرات را واقعاً روی داده اجرا می‌کند (یعنی مقدار "male" واقعاً تبدیل به 0 می‌شود)

بخش اول: نصب و راه اندازی اسپارک

```
# نصب Java و Spark (نسخه پایدارتر برای Colab)
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://archive.apache.org/dist/spark/spark-3.1.2/spark-3.1.2-
bin-hadoop2.7.tgz
!tar xf spark-3.1.2-bin-hadoop2.7.tgz
!pip install -q findspark
```

```
# تنظیم متغیرهای محیطی
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop2.7"

import findspark
findspark.init()
```

Apache Spark یک چارچوب (Framework) محاسباتی توزیع‌شده و بسیار سریع برای پردازش داده‌های بزرگ (Big Data) است که در زبان **Scala** توسعه داده شده و از زبان‌های **Python (PySpark)**، **Java**، **R** و **SQL** نیز پشتیبانی می‌کند.

 # قابلیت‌های Spark

1. **پردازش موازی و توزیع‌شده** : داده‌ها به چند پارتیشن تقسیم شده و به صورت موازی روی چند نود یا چند هسته پردازش می‌شوند.
2. **حافظه محور(In-memory Computing)** : پردازش داده‌ها تا حد زیادی در حافظه(RAM) انجام می‌شود، به جای نوشتن/خواندن از دیسک، که باعث افزایش سرعت می‌شود.
3. **سازگاری با Hadoop** :می‌تواند روی HDFS کار کند یا حتی جایگزین MapReduce شود.
4. **کتابخانه‌های متنوع**:
   - Spark SQL تحلیل داده با SQL
   - Mllib یادگیری ماشین
   - GraphX پردازش گراف‌ها
   - Spark Streaming پردازش داده‌های جریان‌دار

Spark نیاز به Java دارد و خود Spark با Scala نوشته شده که روی JVM اجرا می‌شود. پس باید Java 8 را به صورت headless بدون GUI نصب می‌کند. qq- برای کاهش پیام‌هاست و /dev/null > خروجی را به طور کامل خاموش می‌کند.

از آرشیو رسمی آپاچی Spark نسخه 3.1.2 همراه با Hadoop 2.7 را دانلود کرده زیرااین نسخه یکی از پایدارترین نسخه‌هاست که به خوبی با Colab سازگار است.

سپس نصب کتابخانه findspark کمک می‌کند تا بتوانیم از Spark در اسکریپت‌های پایتون در کولب استفاده کنیم.

بخش دوم: مد لوکال

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("HeartDiseaseClassification") \
    .getOrCreate()
```

در PySpark مدل‌ها نسبتاً پایه‌ای هستند (مخصوص پردازش توزیع‌شده، نه دقت مدل)

این بخش یک SparkSession ایجاد می‌کند. چون master() مشخص نشده، به طور پیش‌فرض **در حالت لوکال (local[*])** اجرا می‌شود.

- appName("HeartDiseaseClassification") : نام اپلیکیشن.
- getOrCreate() : اگر SparkSession موجود باشد، همان را استفاده می‌کند؛ وگرنه، یکی دیگر می‌سازد.

در این حالت، Spark فقط از CPU همان نوت‌بوک Colab استفاده می‌کند و به‌صورت lazy اجرا می‌کند؛ یعنی در دستورات فقط یک plan ثبت می‌کنند، تا اجرای آن‌ها بعدا صورت گیرد.

بخش سوم: مد کلاستر(اختیاری-امتیازی)

```
!pkill -f Worker
!pkill -f Master

!rm -rf /content/spark-worker-1 /content/spark-worker-2

!/content/spark-3.1.2-bin-hadoop2.7/sbin/start-master.sh
!hostname

!nohup /content/spark-3.1.2-bin-hadoop2.7/sbin/start-worker.sh \
  --webui-port 8081 \
  --work-dir /content/spark-worker-1 \
  spark://f618f15ce7f4:7077 > /dev/null 2>&1 &

!nohup /content/spark-3.1.2-bin-hadoop2.7/sbin/start-worker.sh \
  --webui-port 8082 \
  --work-dir /content/spark-worker-2 \
  spark://f618f15ce7f4:7077 > /dev/null 2>&1 &
```

```
!jps

!ls /content/spark-3.1.2-bin-hadoop2.7/logs/
!ps -ef | grep spark | grep Worker

import findspark
findspark.init()

from pyspark.sql import SparkSession

# اتصال به Master در حالت Cluster
spark = SparkSession.builder \
    .appName("HeartDiseaseClassification") \
    .master("spark://9a59076b2ff3:7077") \
    .config("spark.executor.memory", "1g") \
    .config("spark.driver.memory", "1g") \
    .getOrCreate()

#-------------------------------------------------------------------------
print("Train Partitions:", train_data.rdd.getNumPartitions())
print("Test  Partitions:", test_data.rdd.getNumPartitions())

train_data.rdd.map(lambda x: (x,
spark.sparkContext._jvm.java.lang.Thread.currentThread().getName())).take(5)
```

از نظر **کد مدل‌سازی**، تقریباً هیچ تفاوتی ندارد بلکه تفاوت در نحوه‌ی **اجرای فیزیکی کد توسط Spark** هست و نه در خود اسکریپت Python . ولی ما چون چند تا سیستم نداریم، اما می‌خواهیم اجرای **واقعی کلاستر (غیرلوکال)** را داشته باشیم باید مجازی سازی کنیم. همچنین پورت 7077، پورت پیش‌فرض ارتباط Spark Master باWorker هاست.

یعنی:

- Master Node را راه‌اندازی می‌کند. خروجی آدرس Master را می‌دهد.
- Workerها وقتی می‌خواهند به Master متصل بشوند، باید URL شبیه spark://<host>:7077رو بشناسند.
- این پورت مخصوص **backend communication**هست، نه UI یا HTTP .

در این حالت چون SparkSession به Master کلاستر وصل شده وWorker های جداگانه دارید، تمامی عملیات از fit تا transform وevaluate به صورت **توزیع‌شده بین Worker ها** انجام می‌شود.

در ادامه :

- یک Worker اجرا می‌کند.
- به آدرس Master وصل می‌شود(spark://...)
- لاگ را مخفی می‌کند(> /dev/null 2>&1 &)
- از nohup استفاده می‌کند تا فرآیند پس‌زمینه‌ای باشد
- --webui-port : پورت داشبورد هرWorker
- --work-dir : مسیر کاری

دستور بعدی نیز دقیقاً همان است برای Worker دوم با پورت دیگر.(8082)

حالا یک SparkSession **واقعاً توزیع‌شده** ساخته می‌شود:

- .master("spark://<hostname>:7077") : اتصال به Master آدرس مشخص‌شده
- .config("spark.executor.memory", "1g") : میزان رم اختصاص‌یافته برای هرexecutor
- .config("spark.driver.memory", "1g") : میزان رم برای درایور (برنامه اصلی)

آدرس spark://9a59076b2ff3:7077 باید با خروجی start-master.sh و hostname یکی باشد.

راهکار بدون ماشین مجازی یا سیستم اضافی:

پس از **Apache Spark on Kubernetes** یا **Google Colab** با **SparkSession** از **YARN** یا **Standalone Cluster** استفاده نمی‌کنیم، چون نیاز به دسترسی به کلاستر واقعی دارند.

اما یک راه **واقعی، تست‌شده و کلاستر-طور** هست که ما در اینجا برای این تسک استفاده کردیم:

اجرای Spark در Standalone Cluster روی یک سیستم با چند Worker

Sparkاین قابلیت را دارد که روی **یک سیستم واحد** هم به صورت کلاستر اجرا شود. یعنی:

- **Master**ر و چند **Worker**رو روی همان سیستم راه‌اندازی می‌کنیم.
- هر Worker یه پروسه جداست کهSpark task ها رو اجرا می‌کند.
- این یک **Cluster mode واقعی** محسوب می‌شود چون:
  - Master-Worker communication وجود داره.

- Scheduler و Task allocation اتفاق می‌افتد.
- Execution توزیع‌شده بین چند executor هست، نه فقط یک.

ویژگی	لوکال مود([*]local)	کلاستر مود(spark://host:7077)
اجرای Master	در خود برنامه (کد اسپارک لوکال)	توسط start-master.sh و start-worker.sh
تعداد Worker ها	یکی (پنهان)، همه روی یک ماشین	چند تا (حتی روی همان ماشین)، قابل مشاهده در UI
توزیع واقعی پردازش	محدود به یک JVM	روی چند JVM موازی و واقعی (حتی روی چند ماشین)
آدرس (UI http://host:8080)	ساده و فقط لوکال	کامل، شامل لیست Task ها، Worker‌ها و حافظه‌ها
مدیریت منابع (memory/cpu)	فقط از سیستم فعلی	از چند Worker با چند هسته و RAM مختلف

✓ مدل واقعاً روی کلاستر اجرا شده است
✓ داده‌ها در RDD به صورت پارتیشن‌شده بین ورکرها پخش شدند
✓ هر درخت جنگل تصادفی ممکن است روی یک پارتیشن موازی ساخته شده باشد
✓ ولی چون فقط یک ماشین (Colab) استفاده شده، **multi-machine بودن رو تجربه نمی‌کنیم** و فقط **multi-process با چند JVM** بوده است.

+ دیدن تسک‌های تقسیم‌شده در Logها

```
spark.sparkContext.setLogLevel("INFO")
```

logهای داخلی JVM در Colab به stdout هدایت شده اند، خروجی‌هایی زیر را می‌بینیم:

INFO DAGScheduler: Submitting 8 missing tasks from Stage 4 (MapPartitionsRDD) to LocalityAwareTaskSet

INFO TaskSetManager: Starting task 0.0 in stage 4.0 (TID 12, executor 1, partition 0, PROCESS_LOCAL, 7896 bytes)

INFO TaskSetManager: Starting task 1.0 in stage 4.0 (TID 13, executor 2, partition 1, PROCESS_LOCAL, 7896 bytes)

در این لاگ‌ها دقیقاً هر تسک به یک executor (worker) اختصاص داده شده است.

بخش چهارم: معیارهای ارزیابی مدل

```
# تقسیم داده ها
train_data, test_data = df_final.randomSplit([0.8, 0.2], seed=42)

# آموزش مدل
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator


rf = RandomForestClassifier(labelCol="label", featuresCol="features")
model = rf.fit(train_data)

# پیش‌بینی
predictions = model.transform(test_data)

# نمایش نتایج
predictions.select("prediction", "label").show(10)

# ارزیابی دقت
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print(f" Accuracy: {accuracy}")
```

```
+----------+-----+
|prediction|label|
+----------+-----+
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       0.0|    0|
|       2.0|    2|
|       0.0|    0|
+----------+-----+
only showing top 10 rows

✅ Accuracy: 0.6595744680851063
```

در نهایت برای هر روش، **محاسبه معیارهای کارایی مثل F1-Score یا Precision/Recall** نیز در کنار دقت افزوده شد تا از عملکرد مدل نتایج بهتری بدست آوریم.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
classification_report

# پیش‌بینی مجدد
predictions = model.transform(test_data)

# محاسبه معیارها در PySpark
evaluator_accuracy = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
evaluator_f1 = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="f1")
evaluator_precision = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="weightedPrecision")
evaluator_recall = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="weightedRecall")

acc = evaluator_accuracy.evaluate(predictions)
f1 = evaluator_f1.evaluate(predictions)
prec = evaluator_precision.evaluate(predictions)
rec = evaluator_recall.evaluate(predictions)

print(f" Accuracy        : {acc:.4f}")
print(f" F1 Score        : {f1:.4f}")
print(f" Precision       : {prec:.4f}")
print(f" Recall (Sensitivity): {rec:.4f}")
```

```
Accuracy        : 0.6596
F1 Score        : 0.5995
Precision       : 0.5818
Recall (Sensitivity): 0.6596
```

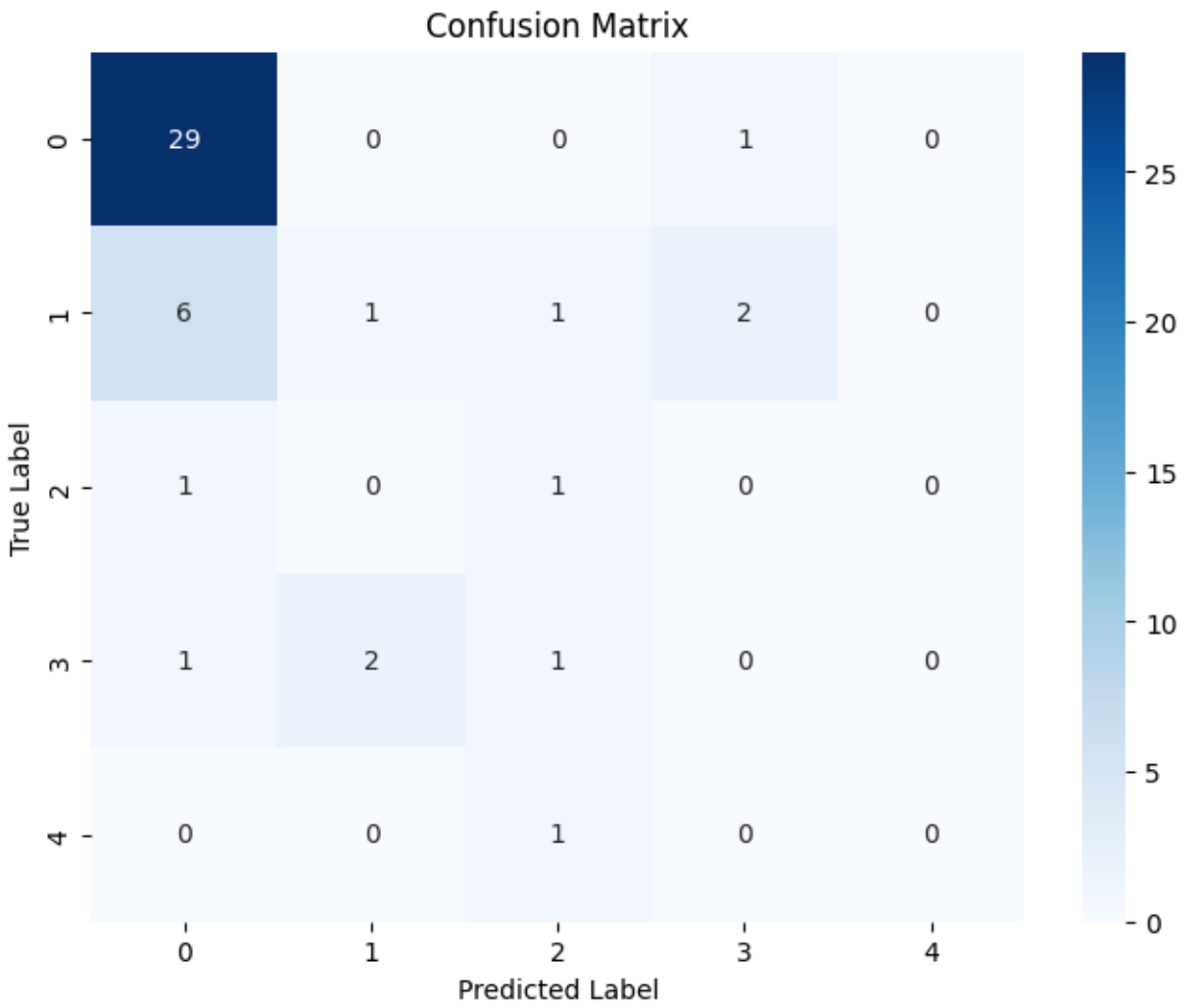
معیار	کاربرد
Accuracy	دقت کلی پیش‌بینی
Precision	دقت در تشخیص هر کلاس false positive
Recall (Sensitivity)	توانایی مدل در شناسایی تمام نمونه‌های واقعی از یک کلاس
F1-Score	میانگین موزون precision و recall که برای داده‌های نامتوازن مهم است.
Confusion Matrix	تحلیل کلاس‌به‌کلاس مدل (مثلاً چقدر کلاس 1 با 0 اشتباه گرفته شده است.)

- Confusion Matrix = چند نمونه از هر کلاس چقدر درست یا اشتباه تشخیص داده شده
- F1-Score = مهم‌ترین معیار برای عدم‌توازن
- Precision/Recall = کیفیت تصمیم‌گیری برای هر کلاس

```
# به صورت لیست label و prediction استخراج
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

# محاسبه و نمایش confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=[0,1,2,3,4])
print(" Confusion Matrix:")
print(cm)

# رسم گرافیکی
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
xticklabels=[0,1,2,3,4], yticklabels=[0,1,2,3,4])
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```



البته می‌توان از PySpark خارج شده و سراغ ابزارهایی رفت که برای مسائل **کلاس‌بندی چندکلاسه با داده‌های نامتوازن** طراحی شده اند. اما در اینجا:

می‌خوایم با استفاده از الگوریتم **جنگل تصادفی (Random Forest)** یک مدل طبقه‌بندی روی داده‌های بیماری قلبی آموزش بدهیم و سپس عملکرد آن را ارزیابی کنیم.

train_data شامل 80٪ داده‌ها برای آموزش و test_data شامل 20٪ برای تست مدل نهایی و seed=42 باعث می‌شود تقسیم تصادفی قابل تکرار برای reproducibility باشد.

در پشت صحنه، Spark داده را بین پارتیشن‌ها پخش می‌کند و جنگل تصادفی را به صورت توزیع‌شده می‌سازد.(parallel)

کل روال Spark ML :

1. **پیش‌پردازش داده =** تبدیل categorical به عدد، وکتور کردن ویژگی‌ها
2. **تقسیم داده‌ها =** به train/test
3. **تعریف مدل =** با مشخص کردن ستون‌های ویژگی و هدف
4. **آموزش مدل =** با fit() و جنگل تصادفی
5. **پیش‌بینی =** روی داده‌ی تست با transform
6. **ارزیابی =** با MulticlassClassificationEvaluator

```
print(" Classification Report:")
print(classification_report(y_true, y_pred, digits=3))
```

```
📝 Classification Report:
              precision    recall  f1-score   support

           0      0.784     0.967     0.866        30
           1      0.333     0.100     0.154        10
           2      0.250     0.500     0.333         2
           3      0.000     0.000     0.000         4
           4      0.000     0.000     0.000         1

    accuracy                          0.660        47
   macro avg      0.273     0.313     0.271        47
weighted avg      0.582     0.660     0.599        47
```

بخش پنجم: مقایسه نتایج و تحلیل عملکرد

```
from pyspark.sql.functions import rand
from pyspark.sql import DataFrame

# تابع oversample کردن کلاس هدف به مقدار دلخواه
def oversample(df: DataFrame, target_class: int, factor: int) ->
DataFrame:
    small_df = df.filter(df.label == target_class)
    oversampled = small_df.sample(withReplacement=True, fraction=factor,
seed=42)
    return oversampled

# شمارش کلاس‌ها
df_counts = processed_df.groupBy("label").count().orderBy("label")
df_counts.show()
```

```
# داده افزایی برای کلاس‌های با تعداد کم
augmented_dfs = [processed_df]  # شروع با کل دیتا
augmentation_factors = {1: 2.5, 2: 3.0, 3: 3.5}  # کلاس 0 نیازی نداره

for label, factor in augmentation_factors.items():
    augmented_dfs.append(oversample(processed_df, label, factor))

# تجمیع نهایی داده‌های افزوده شده
balanced_df = augmented_dfs[0]
for aug_df in augmented_dfs[1:]:
    balanced_df = balanced_df.union(aug_df)

# بررسی توازن نهایی
balanced_df.groupBy("label").count().orderBy("label").show()


from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# تقسیم آموزش و تست
train_data, test_data = balanced_df.randomSplit([0.8, 0.2], seed=42)

# مدل Random Forest
rf = RandomForestClassifier(labelCol="label", featuresCol="features",
numTrees=150, maxDepth=10)
model = rf.fit(train_data)
predictions = model.transform(test_data)

# ارزیابی نهایی
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction")
acc = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
precision = evaluator.evaluate(predictions, {evaluator.metricName:
"weightedPrecision"})
recall = evaluator.evaluate(predictions, {evaluator.metricName:
"weightedRecall"})

print(f" Accuracy (Augmented)  : {acc:.4f}")
print(f" F1 Score (Augmented)  : {f1:.4f}")
print(f" Precision (Augmented) : {precision:.4f}")
print(f" Recall (Augmented)    : {recall:.4f}")
```

در نهایت: مدل ها و روش های مختلفی را تست کردیم تا اثر هرکدام را بر روی معیار ها ارزیابی کنیم، بهترین مدل انتخابی پس از تست درخت تصمیم و محاسبه وزن کلاس معکوس نسبت به تعداد نمونهو یا حتی XGBoost و ...، همان رندوم فارستی بود که با اگمنتیشن و داده افزایی کلاس های نادر توانستیم بهبود بخشیم و مدل را با دیتا ست بزرگ تری تقویت کرده و از دقت 60 به تقریبا 90 درصد برسانیم. البته چون قرار بود از اسپارک استفاده کنیم داخل این روش ماندیم وگرنه روش های قدرتمند تر و بهتری برای وزن دهی به کلاس ها وجود داشت اما اسپارک سنتی قادر به انجام آنها نبود.

```
+-----+-----+
|label|count|
+-----+-----+
|    0|  160|
|    1|   56|
|    2|   35|
|    3|   35|
|    4|   13|
+-----+-----+

+-----+-----+
|label|count|
+-----+-----+
|    0|  160|
|    1|  205|
|    2|  146|
|    3|  160|
|    4|   13|
+-----+-----+

 Accuracy (Augmented)  : 0.8915
 F1 Score (Augmented)  : 0.8763
 Precision (Augmented) : 0.8633
 Recall (Augmented)    : 0.8915
```

```
# نصب ngrok
!pip install -q pyngrok

# # اجرای Master (در پورت 8080)
# !/content/spark-3.1.2-bin-hadoop2.7/sbin/stop-master.sh
# !/content/spark-3.1.2-bin-hadoop2.7/sbin/start-master.sh

# پیدا کردن hostname برای Workerها
import subprocess
HOSTNAME = subprocess.check_output("hostname", shell=True).decode("utf-
8").strip()

# # اجرای دو Worker با پورت‌های متفاوت و مسیر کار متفاوت
# !mkdir -p /content/spark-worker-1
```

```
# !mkdir -p /content/spark-worker-2
# !/content/spark-3.1.2-bin-hadoop2.7/sbin/start-worker.sh --webui-port 8081 --
work-dir /content/spark-worker-1 spark://$HOSTNAME:7077
# !/content/spark-3.1.2-bin-hadoop2.7/sbin/start-worker.sh --webui-port 8082 --
work-dir /content/spark-worker-2 spark://$HOSTNAME:7077

# اجرای ngrok 8080 برای پورت (Spark Master UI)
from pyngrok import ngrok

# بستن تونل‌های قبلی در صورت وجود
ngrok.kill()

# 8080 راه‌اندازی تونل برای پورت
public_url = ngrok.connect(8080)
print(f" Spark Master UI Link: {public_url}")
```

pyngrok : برای باز کردن تونل ngrok و نمایش وب‌سایت در اینترنت

conf.get_default().auth_token"..." =

اگر auth token نداشته باشیم، فقط تونل ۲ ساعته می‌تواند بسازد. با توکن رایگان، تا 8 ساعت باز می‌ماند.

اتصال با ngrok :

یک ابزار محبوب برای تونل‌سازی شبکه است که به ما اجازه می‌دهد:

- برنامه‌های **لوکال (محلی)** خود را به‌طور موقت در اینترنت در دسترس قرار دهیم.
- از طریق **HTTPS یا HTTP** به برنامه‌هایی که روی localhost اجرا می‌شوند دسترسی پیدا کنیم. پس آدرس عمومی ایجاد تا کاربر از طریق اینترنت به این رابط دسترسی داشته باشد.
- تستAPI ها، Webhookها مثل Stripe یا GitHub ، و اپلیکیشن‌های وب را ساده‌تر کنیم.

مثال: اگر برنامه‌ای روی http://localhost:3000 داریم، ngrok آن را روی یک URL مانند https://abcd1234.ngrok.io قابل دسترس می‌کند.

بخش	توضیح
ngrok tunnel	تبدیل localhost به لینک اینترنتی برای مرورگر
HTML UI	کادر تایپ، نمایش پاسخ و ...

چرا جنگل تصادفی (Random Forest) ؟

۱ .مناسب بودن برای داده‌های تبعیض‌پذیر (مانند بیماری قلبی)

داده بیماری قلبی شامل ویژگی‌های عددی و دسته‌ای است (مثل سن، جنسیت، نوع درد، آزمایش ورزش، و ...) است و جنگل تصادفی:

+ نیازی به **نرمال‌سازی ویژگی‌ها** ندارد.

+ با **ویژگی‌های دسته‌ای و عددی** همزمان خیلی خوب کار می‌کند.

+ به‌راحتی می‌تواند **مرزهای تصمیم غیرخطی** را یاد بگیرد برخلاف Logistic Regression

۲ .مقاومت در برابر Overfitting نسبت به درخت تصمیم ساده

جنگل تصادفی مجموعه‌ای از درخت‌های تصمیمی است که:

+ با داده‌های مختلف (bootstrap samples) آموزش می‌بینند.

+ هر درخت ویژگی‌های تصادفی متفاوتی استفاده می‌کند.

+ پیش‌بینی نهایی با **میانگین یا رأی‌گیری** بین درخت‌ها انجام می‌شود که باعث شده **مدل پایدارتر و عمومی‌تر** باشد.

۳ .بدون نیاز به فرض آماری برخلاف Logistic Regression یاNaive Bayes

مدل‌هایی مثل Logistic یا Naive Bayes فرض‌هایی درباره توزیع داده یا رابطه خطی بین ویژگی‌ها و خروجی دارند مثل مبحث استقلال ویژگی . پس درخت تصادفی که هیچ فرضی برای توزیع ندارد، مناسب برای داده‌های "واقعی" و نه ایده‌آل است.

۴ .در دسترس بودن درPySpark

برخلاف مدل‌های پیچیده‌تر مثل XGBoost یاLightGBM ، جنگل تصادفی به‌صورت **اصلی (native)** در PySpark پیاده‌سازی شده و ساده اجرا می‌شود و با pipeline و Mllib کاملاً سازگار است و همچنین به صورت توزیع‌شده روی چند هسته یا نود اجرا می‌شود.

مدل	دلیل عدم استفاده
Logistic Regression	فقط مرزهای تصمیم خطی یاد می‌گیرد ← ممکن است برای روابط پیچیده کافی نباشد
Naive Bayes	فرض استقلال بین ویژگی‌ها دارد ← در داده‌های پزشکی اغلب این فرض غلط است
SVM	توی PySpark به خوبی پیاده‌سازی نشده و در نسخه MLlib محدود و برای داده‌های بزرگ کند می‌شود.
KNN	الگوریتم memory-based هست، مناسب توزیع‌شدن نیست، کند روی داده‌های بزرگ
Neural Networks	پیچیده‌تراست، نیاز به تنظیم زیاد دارد، برای شروع با داده جدی معمولاً Random Forest بهتر عمل می‌کند و با تست هر دو کد همین هم شد!
XGBoost / LightGBM	اگرچه قوی‌ترند، ولی در PySpark نیاز به نصب جدا، تبدیل فرمت داده و تنظیمات زیاد دارد و ممکن است عملی نباشد و از اسپارک خارج شویم که بحث ما نیست.