

How To Use Spark ML and Spark Streaming

How to use SparkML to make predictions on streaming data using PySpark

Spark ML

- we will predict whether someone will get a heart attack based on their age, gender, and medical conditions. A logistic regression will be trained, and we stream in unseen data to make predictions.
- **Data Collection:** on Kaggle. **A dataset for heart attack classification**
- <https://www.kaggle.com/rashikrahmanpritom/heart-attack-analysis-prediction-dataset>
- The data consists of 303 rows and 14 columns.
- Each row represents the information of a patient.
- The features of this dataset consist of the following columns:

- age: age in years
- sex: sex (1 = male; 0 = female)
- cp: chest pain type (1 = typical angina; 2 = atypical angina; 3 = non-anginal pain; 0 = asymptomatic)
- trtbps: resting blood pressure (in mm Hg on admission to the hospital)
- chol: serum cholesterol in mg/dl
- fbs: fasting blood sugar > 120 mg/dl (1 = true; 0 = false)
- restecg: resting electrocardiographic results (1 = normal; 2 = having ST-T wave abnormality; 0 = hypertrophy)
- thalachh: maximum heart rate achieved
- exng: exercise-induced angina (1 = yes; 0 = no)
- oldpeak: ST depression induced by exercise relative to rest
- slp: the slope of the peak exercise ST segment (2 = upsloping; 1 = flat; 0 = downsloping)
- caa: number of major vessels (0-3) colored by fluoroscopy
- thall: 2 = normal; 1 = fixed defect; 3 = reversible defect
- The target column is the following:
- output: 0= less chance of heart attack 1= more change of heart attack.

Steps

- The first step is to create a schema to ensure that the data will consist of the correct data type when reading in the csv file.
- Next, we will use `spark.read.format()` function with “csv” as an argument, add the option to read in the header and assign the schema we created to the data frame.
- Last we load the data, and we also change the target column to label so that our logistic regression can identify which column the target variable is.

```

# We use the following schema
schema = StructType( \
    [StructField("age", LongType(), True), \
    StructField("sex", LongType(), True), \
    StructField("cp", LongType(), True), \
    StructField('trtbps', LongType(), True), \
    StructField("chol", LongType(), True), \
    StructField("fbs", LongType(), True), \
    StructField("restecg", LongType(), True), \
    StructField("thalachh", LongType(), True), \
    StructField("exng", LongType(), True), \
    StructField("oldpeak", DoubleType(), True), \
    \
    StructField("slp", LongType(), True), \
    StructField("caa", LongType(), True), \
    StructField("thall", LongType(), True), \
    StructField("output", LongType(), True), \
    ])

data = "dbfs:/FileStore/tables/heart.csv"
df=spark.read.format('csv').option('header', True).schema(schema).load(data)
df = df.withColumnRenamed("output", "label")
df.display()
df.printSchema()

```

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	label
1	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
2	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
3	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
4	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
5	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
6	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1

Machine Learning:

- When it comes to data preprocessing steps, we first split the data into a training (70%) and test (30%) set.

```
testDF, trainDF = df.randomSplit([0.3, 0.7])
```

created a pipeline that consists of five stages:

The first stage is a vector assembler that takes in the age, trtbps, chol, thalachh, oldpeak columns and turns them into a vector.

The second stage entails the scaling process of the features mentioned above am using the MinMaxScaler() function from the pyspark.ml.feature library.

After that, one-hot encode the sex, cp, fbs, restecg, slp, exng, caa, and thall columns, since those are nominal categorical variables.

Next, create a second vector assembler and add the one-hot encoded columns and scaled features into one vector.

Last but not least, the last stage consists of a Logistic Regression with the following parameters:

maxIter = 10

regParam = 0.01

- chose a logistic regression algorithm because our target consists of binary numbers (0 and 1).
- Once the pipeline has been created, fit and transform the training set.
- After that, select the label, probability, and prediction columns.
- The pipeline construction and the predictions when training the model.

- # We create a one hot encoder.
- `ohe = OneHotEncoder(inputCols = ['sex', 'cp', 'fbs', 'restecg', 'slp', 'exng', 'caa', 'thall'], outputCols=['sex_ohe', 'cp_ohe', 'fbs_ohe', 'restecg_ohe', 'slp_ohe', 'exng_ohe', 'caa_ohe', 'thall_ohe'])`
- # Input list for scaling
- `inputs = ['age', 'trtbps', 'chol', 'thalachh', 'oldpeak']`
- # We scale our inputs
- `assembler1 = VectorAssembler(inputCols=inputs, outputCol="features_scaled1")`
- `scaler = MinMaxScaler(inputCol="features_scaled1", outputCol="features_scaled")`
- # We create a second assembler for the encoded columns.
- `assembler2 = VectorAssembler(inputCols=['sex_ohe', 'cp_ohe', 'fbs_ohe', 'restecg_ohe', 'slp_ohe', 'exng_ohe', 'caa_ohe', 'thall_ohe', 'features_scaled'], outputCol="features")`
- # Create stages list
- `myStages = [assembler1, scaler, ohe, assembler2, lr]`
- # Set up the pipeline
- `pipeline = Pipeline(stages= myStages)`
- # We fit the model using the training data.
- `pModel = pipeline.fit(trainDF)`
- # We transform the data.
- `trainingPred = pModel.transform(trainDF)`
- ## We select the actual label, probability and predictions
- `trainingPred.select('label', 'probability', 'prediction').show()`

label	probability	prediction
1	[0.06216529861690...	1.0
1	[0.01290433721954...	1.0
1	[0.03544144443131...	1.0
1	[0.07312011957665...	1.0
1	[0.03685624359462...	1.0
1	[0.00474806978395...	1.0
1	[0.20083871134740...	1.0
1	[0.08915481644462...	1.0
1	[0.08915481644462...	1.0
0	[0.86138853542893...	0.0
1	[0.00210976458103...	1.0
0	[0.73422844544809...	0.0
0	[0.92027484241729...	0.0
0	[0.47650526226107...	1.0

the marked yellow rows show that the lower the probability, the more confident the model is that the prediction is a 1. On the other hand, the marked red row shows the higher the probability, the more certain it predicts the output to be a zero

- evaluate the performance of the model, calculating the overall accuracy score
- When training the model on the training data, the accuracy score resulted in 0.902913%, which is a satisfying result.

Streaming

- To incorporate Spark Streaming, I repartitioned the test data set into ten different files to replicate the streaming simulation
- # We now repartition the test data and break them down into 10 different files and write it to a csv file.
- `testData = testDF.repartition(10)`
- #Remove directory in case we rerun it multiple times.
- `dbutils.fs.rm("FileStore/tables/HeartTest/",True)`
- #Create a directory
- `testData.write.format("CSV").option("header",True).save("FileStore/tables/HeartTest/")`

- After that, we first created a source, which consisted of the following lines of code.
- # Source

```
sourceStream=spark.readStream.format("csv").option("header",True).  
schema(schema).option("ignoreLeadingWhiteSpace",True).option("m  
ode","dropMalformed").option("maxFilesPerTrigger",1).load("dbfs:/Fi  
leStore/tables/HeartTest").withColumnRenamed("output","label")
```


- we use `spark.readStream` and read in a file with the format “csv”. In addition to that, we add the schema that we created at the beginning when reading in the file, followed by multiple options such as:
- `ignoreLeadingWhiteSpace: True` → Removes the leading whitespace.
- `mode: DropMalformed` → When set to `DropMalformed` it ignores the whole corrupted records.
- `maxFilesPerTrigger: 1` → Maximum number of new files to be considered in every trigger.
- After that, we load in the data from the directory where we repartitioned the test data to replicate the simulation of streaming. Last but not least, we change the output column to label for the implementation of the logistic regression.
- The last step is to set up the streaming of the test data. we use the pipeline that has been fit to the training set (`pModel`) and use the transform functionality with the argument “`sourceStream`”, which is the source we created before. Lastly, we select the label, probability, and prediction columns.
- Please see next for the code snippet.

```
# Stream test data to the ML model.
streamingHeart = pModel.transform(sourceStream).select('label','probability','prediction')

display(streamingHeart)
```

Cancel

▶ (1) Spark Jobs

▶  display_query_4 (id: 6758f961-017a-423d-9486-d0f76c6cc9fb) Last updated: 5 seconds ago

- As we can see, the green light is on, which indicates that we are streaming in unseen data, which comes from the test data that has been repartitioned to replicate the simulation for streaming purposes.
- Below is a sample output of our streaming, which shows us the actual label of the test data, the probability, and the model's prediction on the unseen data.

	label	probability	prediction
1	1	▸ {"vectorType": "dense", "length": 2, "values": [0.06936682704327157, 0.9306331729567284]}	1
2	1	▸ {"vectorType": "dense", "length": 2, "values": [0.08839288702241423, 0.9116071129775858]}	1
3	0	▸ {"vectorType": "dense", "length": 2, "values": [0.9763057239991667, 0.023694276000833292]}	0
4	0	▸ {"vectorType": "dense", "length": 2, "values": [0.2865262085446083, 0.7134737914553917]}	1
5	1	▸ {"vectorType": "dense", "length": 2, "values": [0.5400215235999122, 0.4599784764000878]}	0
6	0	▸ {"vectorType": "dense", "length": 2, "values": [0.7409623237987002, 0.2590376762012998]}	0
7	0	▸ {"vectorType": "dense", "length": 2, "values": [0.9870852296380591, 0.012914770361940886]}	0
8	1	▸ {"vectorType": "dense", "length": 2, "values": [0.014947087920467996, 0.985052912079532]}	1
9	1	▸ {"vectorType": "dense", "length": 2, "values": [0.06708020307258718, 0.9329197969274128]}	1
10	1	▸ {"vectorType": "dense", "length": 2, "values": [0.023880705452510025, 0.97611929454749]}	1
11	0	▸ {"vectorType": "dense", "length": 2, "values": [0.9818049053680156, 0.018195094631984432]}	0
12	0	▸ {"vectorType": "dense", "length": 2, "values": [0.9856062791486043, 0.014393720851395675]}	0
13	1	▸ {"vectorType": "dense", "length": 2, "values": [0.019830028132041745, 0.9801699718679583]}	1
14	1	▸ {"vectorType": "dense", "length": 2, "values": [0.08410446750719074, 0.9158955324928093]}	1
15	1	▸ {"vectorType": "dense", "length": 2, "values": [0.6021188967122059, 0.39788110328779414]}	0
16	1	▸ {"vectorType": "dense", "length": 2, "values": [0.029355106178307347, 0.9706448938216926]}	1

- To assess the predictions on the test data, we can see the probabilities of which class the streamed data is part of.
- For example, when we look at row 1, we can see the vector in the probability column, which consists of [0.06936682704327157, 0.9306331729567284].
- The first element in the vector represents the probability of class 0 (no heart attack), and the second element the probability of class 1 (heart attack).
- The model picks the higher probability value and assigns the streamed data to the class with the higher probability.
- In the first example, the model predicts 1, which is correct when comparing it with the actual label.

- wasn't about the performance of the model
- how we can use unseen streamed data in our machine learning model

