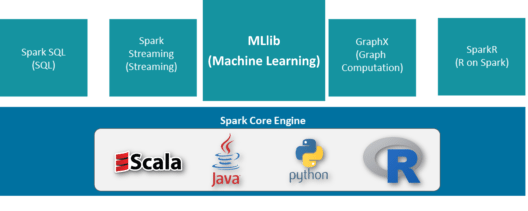
**What is PySpark MLlib?**

PySpark MLlib is a machine-learning library. It is a wrapper over PySpark Core to do data analysis using machine-learning algorithms. It works on distributed systems and is scalable. We can find implementations of classification, clustering, linear regression, and other machine-learning algorithms in PySpark MLlib.



## ****Machine Learning(Python) Industrial Use Cases****

Machine learning algorithms, applications, and platforms are helping manufacturers find new business models, fine-tune product quality, and optimize manufacturing operations to the shop floor level. So Let’s continue our PySpark MLlib Tutorial and understand how the various industries are using Machine Learning.



**Government:**

Government agencies such as public safety and utilities have a particular need for machine learning. They use it for face detection, security and fraud detection. Public sector agencies are making use of machine learning for government initiatives to gain vital insights into policy data.



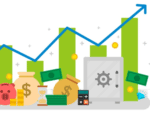
**Marketing and E-commerce:**

The number of purchases made online is steadily increasing, which allows companies to gather detailed data on the whole customer experience. Websites recommending items you might like based on previous purchases are using machine learning to analyze your buying history and promote other items you’d be interested in.



**Transportation:**

Analyzing data to identify patterns and trends is key to the transportation industry, which relies on making **routes more efficient** and predicting potential problems to increase profitability. Companies use ML to enable an efficient ride-sharing marketplace, identify suspicious or fraudulent accounts, suggest optimal pickup and drop-off points.



**Finance:**

Today, machine learning has come to play an integral role in many phases of the financial ecosystem, from approving loans to managing assets, to assessing risks. Banks and other businesses in the financial industry use machine learning technology to **prevent fraud.**



**Healthcare:**

Machine learning is a fast-growing trend in the healthcare industry, thanks to the advent of wearable devices and sensors that can use data to assess a patient’s health in real time. **Google** has developed a machine learning algorithm to help identify cancerous tumors on mammograms. **Stanford** is using a deep learning algorithm to identify skin cancer.

## ****Machine Learning Lifecycle****

A typical Machine Learning Cycle involves majorly two phases:

* Training
* Testing

## Machine learning with Spark

You will proceed as follow:

* **Step 1)** Basic operation with PySpark
* **Step 2)** Data preprocessing
* **Step 3)** Build a data processing pipeline
* **Step 4)** Build the classifier
* **Step 5)** Train and evaluate the model
* **Step 6)** Tune the hyperparameter

Spark is designed to process a considerable amount of data. Spark's performances increase relative to other machine learning libraries when the dataset processed grows larger.

### Step 1) Basic operation with PySpark

You use inferSchema set to True to tell Spark to guess automatically the type of data. By default, it is turn to False.

**df = spark.read.csv("adult\_data.csv", header=True, inferSchema= True)**

Let's have a look at the data type

df.printSchema()

root

|-- age: integer (nullable = true)

|-- workclass: string (nullable = true)

|-- fnlwgt: integer (nullable = true)

|-- education: string (nullable = true)

|-- education\_num: integer (nullable = true)

|-- marital: string (nullable = true)

|-- occupation: string (nullable = true)

|-- relationship: string (nullable = true)

|-- race: string (nullable = true)

|-- sex: string (nullable = true)

|-- capital\_gain: integer (nullable = true)

|-- capital\_loss: integer (nullable = true)

|-- hours\_week: integer (nullable = true)

|-- native\_country: string (nullable = true)

|-- label: string (nullable = true)

You can see the data with show.

df.show(5, truncate = False)

+---+----------------+------+---------+-------------+------------------+-----------------+-------------+-----+------+------------+------------+----------+--------------+-----+

|age|workclass |fnlwgt|education|education\_num|marital |occupation |relationship |race |sex |capital\_gain|capital\_loss|hours\_week|native\_country|label|

+---+----------------+------+---------+-------------+------------------+-----------------+-------------+-----+------+------------+------------+----------+--------------+-----+

|39 |State-gov |77516 |Bachelors|13 |Never-married |Adm-clerical |Not-in-family|White|Male |2174 |0 |40 |United-States |<=50K|

|50 |Self-emp-not-inc|83311 |Bachelors|13 |Married-civ-spouse|Exec-managerial |Husband |White|Male |0 |0 |13 |United-States |<=50K|

|38 |Private |215646|HS-grad |9 |Divorced |Handlers-cleaners|Not-in-family|White|Male |0 |0 |40 |United-States |<=50K|

|53 |Private |234721|11th |7 |Married-civ-spouse|Handlers-cleaners|Husband |Black|Male |0 |0 |40 |United-States |<=50K|

|28 |Private |338409|Bachelors|13 |Married-civ-spouse|Prof-specialty |Wife |Black|Female|0 |0 |40 |Cuba |<=50K|

+---+----------------+------+---------+-------------+------------------+-----------------+-------------+-----+------+------------+------------+----------+--------------+-----+

only showing top 5 rows

If you didn't set inferShema to True, here is what is happening to the type. There are all in string.

df\_string = sqlContext.read.csv(SparkFiles.get("adult.csv"), header=True, inferSchema= False)

df\_string.printSchema()

root

|-- age: string (nullable = true)

|-- workclass: string (nullable = true)

|-- fnlwgt: string (nullable = true)

|-- education: string (nullable = true)

|-- education\_num: string (nullable = true)

|-- marital: string (nullable = true)

|-- occupation: string (nullable = true)

|-- relationship: string (nullable = true)

|-- race: string (nullable = true)

|-- sex: string (nullable = true)

|-- capital\_gain: string (nullable = true)

|-- capital\_loss: string (nullable = true)

|-- hours\_week: string (nullable = true)

|-- native\_country: string (nullable = true)

|-- label: string (nullable = true)

To convert the continuous variable in the right format, you can use recast the columns. You can use withColumn to tell Spark which column to operate the transformation.

# Import all from `sql.types`

from pyspark.sql.types import \*

# Write a custom function to convert the data type of DataFrame columns

def convertColumn(df, names, newType):

for name in names:

df = df.withColumn(name, df[name].cast(newType))

return df

# List of continuous features

CONTI\_FEATURES = ['age', 'fnlwgt','capital\_gain', 'education\_num', 'capital\_loss', 'hours\_week']

# Convert the type

df\_string = convertColumn(df\_string, CONTI\_FEATURES, FloatType())

# Check the dataset

df\_string.printSchema()

root

|-- age: float (nullable = true)

|-- workclass: string (nullable = true)

|-- fnlwgt: float (nullable = true)

|-- education: string (nullable = true)

|-- education\_num: float (nullable = true)

|-- marital: string (nullable = true)

|-- occupation: string (nullable = true)

|-- relationship: string (nullable = true)

|-- race: string (nullable = true)

|-- sex: string (nullable = true)

|-- capital\_gain: float (nullable = true)

|-- capital\_loss: float (nullable = true)

|-- hours\_week: float (nullable = true)

|-- native\_country: string (nullable = true)

|-- label: string (nullable = true)

from pyspark.ml.feature import StringIndexer

#stringIndexer = StringIndexer(inputCol="label", outputCol="newlabel")

#model = stringIndexer.fit(df)

#df = model.transform(df)

df.printSchema()

#### Select columns

You can select and show the rows with select and the names of the features. Below, age and fnlwgt are selected.

df.select('age','fnlwgt').show(5)

+---+------+

|age|fnlwgt|

+---+------+

| 39| 77516|

| 50| 83311|

| 38|215646|

| 53|234721|

| 28|338409|

+---+------+

only showing top 5 rows

#### Count by group

If you want to count the number of occurence by group, you can chain:

* groupBy()
* count()

together. In the example below, you count the number of rows by the education level.

df.groupBy("education").count().sort("count",ascending=True).show()

+------------+-----+

| education|count|

+------------+-----+

| Preschool| 51|

| 1st-4th| 168|

| 5th-6th| 333|

| Doctorate| 413|

| 12th| 433|

| 9th| 514|

| Prof-school| 576|

| 7th-8th| 646|

| 10th| 933|

| Assoc-acdm| 1067|

| 11th| 1175|

| Assoc-voc| 1382|

| Masters| 1723|

| Bachelors| 5355|

|Some-college| 7291|

| HS-grad|10501|

+------------+-----+

#### Describe the data

To get a summary statistics, of the data, you can use describe(). It will compute the :

* count
* mean
* standarddeviation
* min
* max

df.describe().show()

+-------+------------------+-----------+------------------+------------+-----------------+--------+----------------+------------+------------------+------+------------------+----------------+------------------+--------------+-----+

|summary| age| workclass| fnlwgt| education| education\_num| marital| occupation|relationship| race| sex| capital\_gain| capital\_loss| hours\_week|native\_country|label|

+-------+------------------+-----------+------------------+------------+-----------------+--------+----------------+------------+------------------+------+------------------+----------------+------------------+--------------+-----+

| count| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561| 32561|32561|

| mean| 38.58164675532078| null|189778.36651208502| null| 10.0806793403151| null| null| null| null| null|1077.6488437087312| 87.303829734959|40.437455852092995| null| null|

| stddev|13.640432553581356| null|105549.97769702227| null|2.572720332067397| null| null| null| null| null| 7385.292084840354|402.960218649002|12.347428681731838| null| null|

| min| 17| ?| 12285| 10th| 1|Divorced| ?| Husband|Amer-Indian-Eskimo|Female| 0| 0| 1| ?|<=50K|

| max| 90|Without-pay| 1484705|Some-college| 16| Widowed|Transport-moving| Wife| White| Male| 99999| 4356| 99| Yugoslavia| >50K|

+-------+------------------+-----------+------------------+------------+-----------------+--------+----------------+------------+------------------+------+------------------+----------------+------------------+--------------+-----+

If you want the summary statistic of only one column, add the name of the column inside describe()

df.describe('capital\_gain').show()

+-------+------------------+

|summary| capital\_gain|

+-------+------------------+

| count| 32561|

| mean|1077.6488437087312|

| stddev| 7385.292084840354|

| min| 0|

| max| 99999|

+-------+------------------+

#### Drop column

There are two intuitive API to drop columns:

* drop(): Drop a column
* dropna(): Drop NA's

Below you drop the column education\_num

df.drop('education\_num').columns

['age',

'workclass',

'fnlwgt',

'education',

'marital',

'occupation',

'relationship',

'race',

'sex',

'capital\_gain',

'capital\_loss',

'hours\_week',

'native\_country',

'label']

#### Filter data

You can use filter() to apply descriptive statistics in a subset of data. For instance, you can count the number of people above 40 year old

df.filter(df.age > 40).count()

13443

#### Descriptive statistics by group

Finally, you can group data by group and compute statistical operations like the mean.

df.groupby('marital').agg({'capital\_gain': 'mean'}).show()

+--------------------+------------------+

| marital| avg(capital\_gain)|

+--------------------+------------------+

| Separated| 535.5687804878049|

| Never-married|376.58831788823363|

|Married-spouse-ab...| 653.9832535885167|

| Divorced| 728.4148098131893|

| Widowed| 571.0715005035247|

| Married-AF-spouse| 432.6521739130435|

| Married-civ-spouse|1764.8595085470085|

+--------------------+------------------+

### Step 2) Data preprocessing

Data processing is a critical step in machine learning. After you remove garbage data, you get some important insights. For instance, you know that age is not a linear function with the income. When people are young, their income is usually lower than mid-age. After retirement, a household uses their saving, meaning a decrease in income. To capture this pattern, you can add a square to the age feature

**Add age square**

To add a new feature, you need to:

1. Select the column
2. Apply the transformation and add it to the DataFrame

from pyspark.sql.functions import \*

# 1 Select the column

age\_square = df.select(col("age")\*\*2)

# 2 Apply the transformation and add it to the DataFrame

df = df.withColumn("age\_square", col("age")\*\*2)

df.printSchema()

root

|-- age: integer (nullable = true)

|-- workclass: string (nullable = true)

|-- fnlwgt: integer (nullable = true)

|-- education: string (nullable = true)

|-- education\_num: integer (nullable = true)

|-- marital: string (nullable = true)

|-- occupation: string (nullable = true)

|-- relationship: string (nullable = true)

|-- race: string (nullable = true)

|-- sex: string (nullable = true)

|-- capital\_gain: integer (nullable = true)

|-- capital\_loss: integer (nullable = true)

|-- hours\_week: integer (nullable = true)

|-- native\_country: string (nullable = true)

|-- label: string (nullable = true)

|-- age\_square: double (nullable = true)

You can see that age\_square has been successfully added to the data frame. You can change the order of the variables with select. Below, you bring age\_square right after age.

COLUMNS = ['age', 'age\_square', 'workclass', 'fnlwgt', 'education', 'education\_num', 'marital',

'occupation', 'relationship', 'race', 'sex', 'capital\_gain', 'capital\_loss',

'hours\_week', 'native\_country', 'label']

df = df.select(COLUMNS)

df.first()

Row(age=39, age\_square=1521.0, workclass='State-gov', fnlwgt=77516, education='Bachelors', education\_num=13, marital='Never-married', occupation='Adm-clerical', relationship='Not-in-family', race='White', sex='Male', capital\_gain=2174, capital\_loss=0, hours\_week=40, native\_country='United-States', label='<=50K')

**Exclude Holand-Netherlands**

When a group within a feature has only one observation, it brings no information to the model. On the contrary, it can lead to an error during the cross-validation.

Let's check the origin of the household

df.filter(df.native\_country == 'Holand-Netherlands').count()

df.groupby('native\_country').agg({'native\_country': 'count'}).sort(asc("count(native\_country)")).show()

+--------------------+---------------------+

| native\_country|count(native\_country)|

+--------------------+---------------------+

| Holand-Netherlands| 1|

| Scotland| 12|

| Hungary| 13|

| Honduras| 13|

|Outlying-US(Guam-...| 14|

| Yugoslavia| 16|

| Thailand| 18|

| Laos| 18|

| Cambodia| 19|

| Trinadad&Tobago| 19|

| Hong| 20|

| Ireland| 24|

| Ecuador| 28|

| Greece| 29|

| France| 29|

| Peru| 31|

| Nicaragua| 34|

| Portugal| 37|

| Iran| 43|

| Haiti| 44|

+--------------------+---------------------+

only showing top 20 rows

The feature native\_country has only one household coming from Netherland. You exclude it.

df\_remove = df.filter(df.native\_country != 'Holand-Netherlands')

### Step 3) Build a data processing pipeline

Similar to scikit-learn, Pyspark has a pipeline API. A pipeline is very convenient to maintain the structure of the data. You push the data into the pipeline. Inside the pipeline, various operations are done, the output is used to feed the algorithm.

For instance, one universal transformation in machine learning consists of converting a string to one hot encoder, i.e., one column by a group. One hot encoder is usually a matrix full of zeroes.

The steps to transform the data are very similar to scikit-learn. You need to:

* Index the string to numeric
* Create the one hot encoder
* Transform the data

Two APIs do the job: StringIndexer, OneHotEncoder

#### Build the pipeline

You will build a pipeline to convert all the precise features and add them to the final dataset. The pipeline will have four operations, but feel free to add as many operations as you want.

1. Encode the categorical data
2. Index the label feature
3. Add continuous variable
4. Assemble the steps.

Each step is stored in a list named stages. This list will tell the VectorAssembler what operation to perform inside the pipeline.

**1. Encode the categorical data**

This step is exaclty the same as the above example, except that you loop over all the categorical features.

from pyspark.ml import Pipeline

from pyspark.ml.feature import OneHotEncoder

CATE\_FEATURES = ['workclass', 'education', 'marital', 'occupation', 'relationship', 'race', 'sex', 'native\_country']

stages = [] # stages in our Pipeline

for categoricalCol in CATE\_FEATURES:

stringIndexer = **StringIndexer(**inputCol=categoricalCol, outputCol=categoricalCol + "Index")

encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()],

outputCols=[categoricalCol + "classVec"])

stages += [stringIndexer, encoder]

**2. Index the label feature**

Spark, like many other libraries, does not accept string values for the label. You convert the label feature with StringIndexer and add it to the list stages

# Convert label into label indices using the StringIndexer

label\_stringIdx = StringIndexer(inputCol="label", outputCol="newlabel")

stages += [label\_stringIdx]

**3. Add continuous variable**

The inputCols of the VectorAssembler is a list of columns. You can create a new list containing all the new columns. The code below popluate the list with encoded categorical features and the continuous features.

assemblerInputs = [c + "classVec" for c in CATE\_FEATURES] + CONTI\_FEATURES

**4. Assemble the steps.**

Finally, you pass all the steps in the VectorAssembler

assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")stages += [assembler]

Now that all the steps are ready, you push the data to the pipeline.

# Create a Pipeline.

pipeline = Pipeline(stages=stages)

pipelineModel = pipeline.fit(df\_remove)

model = pipelineModel.transform(df\_remove)

If you check the new dataset, you can see that it contains all the features, transformed and not transformed. You are only interested by the newlabel and features. The features includes all the transformed features and the continuous variables.

model.take(1)

[Row(age=39, age\_square=1521.0, workclass='State-gov', fnlwgt=77516, education='Bachelors', education\_num=13, marital='Never-married', occupation='Adm-clerical', relationship='Not-in-family', race='White', sex='Male', capital\_gain=2174, capital\_loss=0, hours\_week=40, native\_country='United-States', label='<=50K', workclassIndex=4.0, workclassclassVec=SparseVector(8, {4: 1.0}), educationIndex=2.0, educationclassVec=SparseVector(15, {2: 1.0}), maritalIndex=1.0, maritalclassVec=SparseVector(6, {1: 1.0}), occupationIndex=3.0, occupationclassVec=SparseVector(14, {3: 1.0}), relationshipIndex=1.0, relationshipclassVec=SparseVector(5, {1: 1.0}), raceIndex=0.0, raceclassVec=SparseVector(4, {0: 1.0}), sexIndex=0.0, sexclassVec=SparseVector(1, {0: 1.0}), native\_countryIndex=0.0, native\_countryclassVec=SparseVector(40, {0: 1.0}), newlabel=0.0, features=SparseVector(99, {4: 1.0, 10: 1.0, 24: 1.0, 32: 1.0, 44: 1.0, 48: 1.0, 52: 1.0, 53: 1.0, 93: 39.0, 94: 77516.0, 95: 2174.0, 96: 13.0, 98: 40.0}))]

### Step 4) Build the classifier: logistic

To make the computation faster, you convert model to a DataFrame. You need to select newlabel and features from model using map.

from pyspark.ml.linalg import DenseVector

input\_data = model.rdd.map(lambda x: (x["newlabel"], DenseVector(x["features"])))

You are ready to create the train data as a DataFrame. You use the sqlContext

df\_train = sqlContext.createDataFrame(input\_data, ["label", "features"])

Check the second row

df\_train.show(2)

+-----+--------------------+

|label| features|

+-----+--------------------+

| 0.0|[0.0,0.0,0.0,0.0,...|

| 0.0|[0.0,1.0,0.0,0.0,...|

+-----+--------------------+

only showing top 2 rows

**Create a train/test set**

You split the dataset 80/20 with randomSplit.

# Split the data into train and test sets

train\_data, test\_data = df\_train.randomSplit([.8,.2],seed=1234)

Let's count how many people with income below/above 50k in both training and test set

train\_data.groupby('label').agg({'label': 'count'}).show()

+-----+------------+

|label|count(label)|

+-----+------------+

| 0.0| 19698|

| 1.0| 6263|

+-----+------------+

test\_data.groupby('label').agg({'label': 'count'}).show()

+-----+------------+

|label|count(label)|

+-----+------------+

| 0.0| 5021|

| 1.0| 1578|

+-----+------------+

#### Build the logistic regressor

Last but not least, you can build the classifier. Pyspark has an API called LogisticRegression to perform logistic regression.

You initialize lr by indicating the label column and feature columns. You set a maximum of 10 iterations and add a regularization parameter with a value of 0.3. Note that in the next section, you will use cross-validation with a parameter grid to tune the model

# Import `LinearRegression`

from pyspark.ml.classification import LogisticRegression

# Initialize `lr`

lr = LogisticRegression(labelCol="label",

featuresCol="features",

maxIter=10,

regParam=0.3)

# Fit the data to the model

linearModel = lr.fit(train\_data)

#You can see the coefficients from the regression

# Print the coefficients and intercept for logistic regression

print("Coefficients: " + str(linearModel.coefficients))

print("Intercept: " + str(linearModel.intercept))

Coefficients: [-0.0678914665262,-0.153425526813,-0.0706009536407,-0.164057586562,-0.120655298528,0.162922330862,0.149176870438,-0.626836362611,-0.193483661541,-0.0782269980838,0.222667203836,0.399571096381,-0.0222024341804,-0.311925857859,-0.0434497788688,-0.306007744328,-0.41318209688,0.547937504247,-0.395837350854,-0.23166535958,0.618743906733,-0.344088614546,-0.385266881369,0.317324463006,-0.350518889186,-0.201335923138,-0.232878560088,-0.13349278865,-0.119760542498,0.17500602491,-0.0480968101118,0.288484253943,-0.116314616745,0.0524163478063,-0.300952624551,-0.22046421474,-0.16557996579,-0.114676231939,-0.311966431453,-0.344226119233,0.105530129507,0.152243047814,-0.292774545497,0.263628334433,-0.199951374076,-0.30329422583,-0.231087515178,0.418918551,-0.0565930184279,-0.177818073048,-0.0733236680663,-0.267972912252,0.168491215697,-0.12181255723,-0.385648075442,-0.202101794517,0.0469791640782,-0.00842850210625,-0.00373211448629,-0.259296141281,-0.309896554133,-0.168434409756,-0.11048086026,0.0280647963877,-0.204187030092,-0.414392623536,-0.252806580669,0.143366465705,-0.516359222663,-0.435627370849,-0.301949286524,0.0878249035894,-0.210951740965,-0.621417928742,-0.099445190784,-0.232671473401,-0.1077745606,-0.360429419703,-0.420362959052,-0.379729467809,-0.395186242741,0.0826401853838,-0.280251589972,0.187313505214,-0.20295228799,-0.431177064626,0.149759018379,-0.107114299614,-0.319314858424,0.0028450133235,-0.651220387649,-0.327918792207,-0.143659581445,0.00691075160413,8.38517628783e-08,2.18856717378e-05,0.0266701216268,0.000231075966823,0.00893832698698]

Intercept: -1.9884177974805692

### Step 5) Train and evaluate the model

To generate prediction for your test set, you can use linearModel with transform() on test\_data

# Make predictions on test data using the transform() method.

predictions = linearModel.transform(test\_data)

You can print the elements in predictions

predictions.printSchema()

root

|-- label: double (nullable = true)

|-- features: vector (nullable = true)

|-- rawPrediction: vector (nullable = true)

|-- probability: vector (nullable = true)

|-- prediction: double (nullable = false)

You are interested by the label, prediction and the probability

selected = predictions.select("label", "prediction", "probability")

selected.show(20)

+-----+----------+--------------------+

|label|prediction| probability|

+-----+----------+--------------------+

| 0.0| 0.0|[0.91560704124179...|

| 0.0| 0.0|[0.92812140213994...|

| 0.0| 0.0|[0.92161406774159...|

| 0.0| 0.0|[0.96222760777142...|

| 0.0| 0.0|[0.66363283056957...|

| 0.0| 0.0|[0.65571324475477...|

| 0.0| 0.0|[0.73053376932829...|

| 0.0| 1.0|[0.31265053873570...|

| 0.0| 0.0|[0.80005907577390...|

| 0.0| 0.0|[0.76482251301640...|

| 0.0| 0.0|[0.84447301189069...|

| 0.0| 0.0|[0.75691912026619...|

| 0.0| 0.0|[0.60902504096722...|

| 0.0| 0.0|[0.80799228385509...|

| 0.0| 0.0|[0.87704364852567...|

| 0.0| 0.0|[0.83817652582377...|

| 0.0| 0.0|[0.79655423248500...|

| 0.0| 0.0|[0.82712311232246...|

| 0.0| 0.0|[0.81372823882016...|

| 0.0| 0.0|[0.59687710752201...|

+-----+----------+--------------------+

only showing top 20 rows

### Evaluate the model

You need to look at the accuracy metric to see how well (or bad) the model performs. Currently, there is no API to compute the accuracy measure in Spark. The default value is the ROC, receiver operating characteristic curve. It is a different metrics that take into account the false positive rate.

Before you look at the ROC, let's construct the accuracy measure. You are more familiar with this metric. The accuracy measure is the sum of the correct prediction over the total number of observations.

You create a DataFrame with the label and the `prediction.

cm = predictions.select("label", "prediction")

You can check the number of class in the label and the prediction

cm.groupby('label').agg({'label': 'count'}).show()

+-----+------------+

|label|count(label)|

+-----+------------+

| 0.0| 5021|

| 1.0| 1578|

+-----+------------+

cm.groupby('prediction').agg({'prediction': 'count'}).show()

+----------+-----------------+

|prediction|count(prediction)|

+----------+-----------------+

| 0.0| 5982|

| 1.0| 617|

+----------+-----------------+

For instance, in the test set, there is 1578 household with an income above 50k and 5021 below. The classifier, however, predicted 617 households with income above 50k.

You can compute the accuracy by computing the count when the label are correctly classified over the total number of rows.

cm.filter(cm.label == cm.prediction).count() / cm.count()

0.8237611759357478

You can wrap everything together and write a function to compute the accuracy.

def accuracy\_m(model):

predictions = model.transform(test\_data)

cm = predictions.select("label", "prediction")

acc = cm.filter(cm.label == cm.prediction).count() / cm.count()

print("Model accuracy: %.3f%%" % (acc \* 100))

accuracy\_m(model = linearModel)

Model accuracy: 82.376%