Introduction to Spark SQL in Python

Создадим таблицу SQL из dataframe (фундаментальная абстракция данных Spark). Spark DataFrame — это распределенная коллекция данных, организованных в именованные столбцы.

```
df = spark.read.csv(filename, header=True) # получаем dataframe, тип <class 'pyspark.sql.dataframe.DataFrame'>
df.createOrReplaceTempView("schedule") # тип <class 'pyspark.sql.dataframe.DataFrame'>
spark.sql("SELECT * FROM schedule WHERE station = 'San Jose'").show() # создаем таблицу Spark
```

```
# все способы получить имена столбцов
result = spark.sql("SHOW COLUMNS FROM tablename")
result = spark.sql("SELECT * FROM tablename LIMIT 0")
result = spark.sql("DESCRIBE tablename") # таблица с именами и типами
result.show()
print(result.columns)
```

Оконная функция

```
query = """SELECT train_id, station, time, diff_min,
           SUM(diff_min) OVER (PARTITION BY train_id ORDER BY time) AS running_total
           FROM schedule"""
spark.sql(query).show()
+--------+-------------+-----+--------+-------------+
|train_id|      station| time|diff_min|running_total|
+--------+-------------+-----+--------+-------------+
|     217|       Gilroy|6:06a|     9.0|          9.0|
|     217|   San Martin|6:15a|     6.0|         15.0|
|     324|San Francisco|7:59a|     4.0|          4.0|
|     324|  22nd Street|8:03a|    13.0|         17.0|
```

```
query = """SELECT train_id, station, time, LEAD(time, 1) OVER (ORDER BY time) AS time_next
           FROM sched
           WHERE train_id=324 """
spark.sql(query).show()
```

```
SELECT # тут PARTITION BY
train_id,
station,
time,
LEAD(time,1) OVER (PARTITION BY train_id ORDER BY time) AS time_next
#добавление предложения OVER определяет этот запрос как запрос оконной функции
#Функция LEAD позволяет запрашивать более одной строки в таблице за раз без необходимости присоединения таблицы к самой себе
FROM sched
```

Точечная нотация

```
spark.sql('SELECT train_id, MIN(time) AS start FROM schedule GROUP BY train_id').show()
df.groupBy('train_id').agg({'time':'min'}).withColumnRenamed('MIN(time)', 'start').show()

spark.sql('SELECT train_id, MIN(time), MAX(time) FROM schedule GROUP BY train_id').show()
result = df.groupBy('train_id').agg({'time':'min', 'time':'max'})
result.show()
print(result.columns[1]) #max(time)
```

```
from pyspark.sql.functions import min, max, col
expr = [min(col("time")).alias('start'), max(col("time")).alias('end')]
dot_df = df.groupBy("train_id").agg(*expr)
dot_df.show()
+--------+-----+-----+
|train_id|start|  end|
+--------+-----+-----+
|     217|6:06a|6:59a|
|     324|7:59a|9:05a|
+--------+-----+-----+
query = "SELECT train_id, MIN(time) AS start, MAX(time) AS end FROM schedule GROUP BY train_id"
```

```
df.columns
#['train_id', 'station', 'time']
df.show(5)
df.select('train_id', 'station').show(5) #могли бы так

df.select('train_id', 'station')
```

```
df.select(df.train_id, df.station)
from pyspark.sql.functions import col
df.select(col('train_id'), col('station')) # пример когда полезен...

df.select('train_id', 'station').withColumnRenamed('train_id', 'train').show(5)
df.select(col('train_id').alias('train'), 'station') #можно так


spark.sql('SELECT train_id AS train, station FROM schedule LIMIT 5').show()
df.select(col('train_id').alias('train'), 'station').limit(5).show()


# оконные функции тоже в точечной и sql нотации
query = """SELECT *,
           ROW_NUMBER() OVER(PARTITION BY train_id ORDER BY time) AS id
           FROM schedule"""
spark.sql(query).show(11) # или точечная нотация...

from pyspark.sql import Window,
from pyspark.sql.functions import row_number
df.withColumn("id", row_number().over(Window.partitionBy('train_id').orderBy('time')))
# пример
dot_df = df.withColumn('time_next', lead('time', 1).over(Window.partitionBy('train_id')
        .orderBy('time'))).show()
```

- ROW_NUMBER in SQL : pyspark.sql.functions.row_number
- The inside of the OVER clause : pyspark.sql.Window
- PARTITION BY : pyspark.sql.Window.partitionBy
- ORDER BY : pyspark.sql.Window.orderBy

```
window = Window.partitionBy('train_id').orderBy('time')
dfx = df.withColumn('next', lead('time', 1).over(window))


# пример
query = """SELECT *,
          (UNIX_TIMESTAMP(LEAD(time, 1) OVER(PARTITION BY train_id ORDER BY time),'H:m')
          - UNIX_TIMESTAMP(time, 'H:m'))/60 AS diff_min
          FROM schedule"""
sql_df = spark.sql(query)
sql_df.show()

window = Window.partitionBy('train_id').orderBy('time')
dot_df = df.withColumn('diff_min', (unix_timestamp(lead('time', 1).over(window),'H:m')
                    - unix_timestamp('time', 'H:m'))/60)
    +--------+-------------+-----+--------+
    |     217|       Gilroy|6:06a|     9.0|
    |     217|   San Martin|6:15a|     6.0|
    |     217|  Morgan Hill|6:21a|    15.0|
```

Использование оконной функции SQL для обработки естественного языка

```
df = spark.read.text('sherlock.txt')
print(df.first()) # получить первую строку
# Row(value='The Project Gutenberg EBook of The Adventures of Sherlock Holmes')
print(df.count())
# 5500
df1.show(15, truncate=False)
```

Для загрузки файла паркета. Parquet — это формат файлов Hadoop для хранения структур данных

```
 # df1 = spark.read.load('sherlock.parquet')
 # df1.where('id > 70').show(5, truncate=False)



df = df1.select(lower(col('value')))
print(df.first())
# Row(lower(value)= 'the project gutenberg ebook of the adventures of sherlock holmes')
df.columns
# ['lower(value)']

df = df1.select(lower(col('value')).alias('v'))
df.columns
# ['v']
```

```
df = df1.select(regexp_replace('value', 'Mr\.', 'Mr').alias('v'))
# "Mr. Holmes." ==> "Mr Holmes."
df = df1.select(regexp_replace('value', 'don\'t', 'do not').alias('v'))
# "don't know." ==> "do not know."
#Чтобы точка не интерпретировалась как специальный символ во втором аргументе, мы ставим перед ней обратную косую черту
df = df2.select(split('v', '[ ]').alias('words')) # можно просто ' '
df.show(truncate=False) # разедлит и поместит в список


punctuation = "_|.\?\!\",\'\[\]\*()"
df3 = df2.select(split('v', '[ %s]' % punctuation).alias('words'))
df3.show(truncate=False)


df4 = df3.select(explode('words').alias('word')) # берет массив и помещает каждый элемент в свою строку
df4.show()
print(df4.count())

nonblank_df = df.where(length('word') > 0) # удаление пустых строк

df2 = df.select('word', monotonically_increasing_id().alias('id'))
df2.show() #создаем столбец целых чисел увеличивающихся
# выход


df2 = df.withColumn('title', when(df.id < 25000, 'Preface').when(df.id < 50000, 'Chapter 1')
                                                           .when(df.id < 75000, 'Chapter 2')
                                                           .otherwise('Chapter 3'))
df2 = df2.withColumn('part', when(df2.id < 25000, 0).when(df2.id < 50000, 1)
                                                    .when(df2.id < 75000, 2)
                                                    .otherwise(3))
                                                    .show()
df2 = df.repartition(4, 'part') # перераспределение данных в df
print(df2.rdd.getNumPartitions()) # 4 - кол-во разделов


$ ls sherlock_parts # пусть есть папка с 14 файлами
df_parts = spark.read.text('sherlock_parts') # загрузить все текстовые файлы в папке в фрейм данных
# файлы считываются параллельно и распределяются по нескольким разделам
```

Оконные функции

```
df.select('part', 'title').distinct().sort('part').show(truncate=False)
query = """SELECT id, word AS w1,
          LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
          LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
          FROM df"""
spark.sql(query).sort('id').show()
+---+----------+----------+----------+
| id|   w1     |     w2   |        w3|
+---+----------+----------+----------+
| 0 |       the|   project| gutenberg|
| 1 |   project| gutenberg|     ebook|
| 2 | gutenberg|     ebook|        of|

#тоже самое но через LAG
lag_query = """SELECT id,
            LAG(word,2) OVER(PARTITION BY part ORDER BY id ) AS w1,
            LAG(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
            word AS w3
            FROM df
            ORDER BY id"""
spark.sql(lag_query).show()
+---+----------+----------+----------+
| id|        w1|        w2|        w3|
+---+----------+----------+----------+
| 0|       null|      null|       the|
| 1|       null|       the|   project|
| 2|        the|   project| gutenberg|
| 3|    project| gutenberg|     ebook|
# если смотрим на результат для более позднего раздела, используя WHERE part=2
lag_query = """SELECT id,
            LAG(word,2) OVER(PARTITION BY part ORDER BY id ) AS w1,
            LAG(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
            word AS w3
            FROM df
            WHERE part=2"""
spark.sql(lag_query).show()
+----+----------+----------+----------+
|  id|        w1|        w2|        w3|
+----+----------+----------+----------+
|8859|      null|      null|     part2|
```

```
|8860|      null|     part2| adventure|
|8861|     part2| adventure|        ii|
```

Какие слова встречаются вместе

```
query3 = """SELECT id,
            word AS w1,
            LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
            LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
            FROM df"""

query3agg = """SELECT w1, w2, w3, COUNT(*) as count FROM (
                      SELECT word AS w1,
                      LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
                      LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
                      FROM df)
GROUP BY w1, w2, w3
ORDER BY count DESC"""
spark.sql(query3agg).show() # считаем какие тройки вместе встречаются

query3agg = """SELECT w1, w2, w3, length(w1)+length(w2)+length(w3) as length
               FROM (SELECT word AS w1,
                LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
                LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
                FROM df
                WHERE part <> 0 and part <> 13)
            GROUP BY w1, w2, w3
            ORDER BY length DESC"""
spark.sql(query3agg).show(truncate=False)
```

Опять же, обратите внимание, что Spark может выполнять этот запрос параллельно для нескольких рабочих процессов, и нам не нужно указывать, как именно это сделать. Поскольку данные секционированы, Spark может автоматически распараллелить оконную функцию SQL.

Кэширование

```
df.cache() # кэшировать dataframe
df.unpersist() # раскешировать
df.is_cached # был ли вэширован df
```

Уровень хранения кадра данных указывает 5 сведений о том, как он кэшируется:

- useDisk = True
- useMemory = True
- useOffHeap = False
- deserialized = True
- replication = 1

```
df.storageLevel
#StorageLevel(True, True, False, True, 1)
```

```
#кэширование таблицы
df.createOrReplaceTempView('df')
spark.catalog.isCached(tableName='df')

spark.catalog.cacheTable('df') # кэширование
spark.catalog.isCached(tableName='df') # была ли кэширована
spark.catalog.uncacheTable('df') # раскэшировать
spark.catalog.clearCache() # удаление кэшированных таблиц
spark.catalog.dropTempView('table1') # удаление временная таблица из каталога

spark.catalog.listTables() #[Table(name='text',database=None,description=None,tableType='TEMPORARY'
```

Ведение журнала (lagging)

```
import logging
logging.basicConfig(stream=sys.stdout, level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info("Hello %s", "world")
logging.debug("Hello, take %d", 2)
# 2019-03-14 15:92:65,359 - INFO - Hello world
```

```
import logging
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info("Hello %s", "world")
logging.debug("Hello, take %d", 2)
# 2018-03-14 12:00:00,000 - INFO - Hello world
# 2018-03-14 12:00:00,001 - DEBUG - Hello, take 2


t = timer()
t.elapsed()
# 1. elapsed: 0.0 sec
t.elapsed() # Do something that takes 2 seconds
# 2. elapsed: 2.0 sec
t.reset() # Do something else that takes time: reset
t.elapsed()
# 3. elapsed: 0.0 sec

class timer:
  start_time = time.time()
  step = 0
  def elapsed(self, reset=True):
    self.step += 1
    print("%d. elapsed: %.1f sec %s" % (self.step, time.time() - self.start_time))
  if reset:
    self.reset()
  def reset(self):
    self.start_time = time.time()


import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
# < create dataframe df here >
t = timer()
logging.info("No action here.")
t.elapsed()
logging.debug("df has %d rows.", df.count())
t.elapsed()


ENABLED = False
t = timer()
logger.info("No action here.")
t.elapsed()
if ENABLED:
    logger.info("df has %d rows.", df.count())
t.elapsed()
# 2019-03-14 12:34:56,789 - Pyspark - INFO - No action here.
# 1. elapsed: 0.0 sec
# 2. elapsed: 0.0 sec

# 2019-03-14 12:34:56,789 - INFO - No action here.
# 1. elapsed: 0.0 sec
# 2019-03-14 12:34:58,789 - INFO - df has 1107014 rows.
# 2. elapsed: 2.0 sec
```

Планы запросов - предоставит подробную информацию о плане без его фактического выполнения

```
df = spark.read.load('/temp/df.parquet')
df.registerTempTable('df')
spark.sql('EXPLAIN SELECT * FROM df').first()

#Row(plan='== Physical Plan ==\n
#*FileScan parquet [word#1928,id#1929L,title#1930,part#1931] - имена столбцов
# Batched: true,
# Format: Parquet,
# Location: InMemoryFileIndex[file:/temp/df.parquet],
# PartitionFilters: [],
# PushedFilters: [],
# ReadSchema: struct<word:string,id:bigint,title:string,part:int>') - типы столбцов


df.explain() # можно для фрейма данных (результат будет аналогичен)
df.cache() #кэширование
df.explain() # чтение снизу вверх
```

Классификация текстов

```
from pyspark.sql.functions import split, explode
df.where(length('sentence') == 0) # выдаст строки с пустыми строками
```

```
# создание пользовательских функций (выдает True если длина меньше 10)
from pyspark.sql.functions import udf
from pyspark.sql.types import BooleanType #указать Spark тип, который должен быть возвращен нашей новой пользовательской функцией

short_udf = udf(lambda x: True if not x or len(x) < 10 else False,
                          BooleanType()) #один аргумент x, указывающий, что она работает с одним столбцом
df.select(short_udf('textdata').alias("is short")).show(3)
+--------+
|is short|
+--------+
|   false|
|    true|
|   false|


from pyspark.sql.types import StringType, IntegerType, FloatType, ArrayType
df3.select('word array', in_udf('word array').alias('without endword')).show(5, truncate=30)
+----------------------------+----------------------+
|                  word array|       without endword|
+----------------------------+----------------------+
|[then, how, many, are, there]|[then, how, many, are]|
|                [how, many]|                [how]|
from pyspark.sql.types import StringType, ArrayType
in_udf = udf(lambda x:
    x[0:len(x)-1] if x and len(x) > 1
    else [],
    ArrayType(StringType()))


# разреженные vs плотные
hasattr(x, "toArray") #определить, что объект является разреженным вектором
x.numNonzeros() #определить, что вектор пуст


from pyspark.sql.functions import udf # конспект
from pyspark.sql.types import IntegerType
bad_udf = udf(lambda x: x.indices[0]
              if (x and hasattr(x, "toArray") and x.numNonzeros())
              else 0, IntegerType()) # будет ошибка
try:
    df.select(bad_udf('outvec').alias('label')).first()
except Exception as e:
    print(e.__class__)
    print(e.errmsg)
# <class 'py4j.protocol.Py4JJavaError'>
# An error occurred while calling o90.collectToPython.

first_udf = udf(lambda x: int(x.indices[0])
                if (x and hasattr(x, "toArray") and x.numNonzeros())
                else 0,
                IntegerType()) # всё ок
+-------+--------------------+-----+--------------------+-------------------+
|endword|                 doc|count|            features|             outvec|
+-------+--------------------+-----+--------------------+-------------------+
|     it|[please, do, not,...| 1149|(12847,[15,47,502...|  (12847,[7],[1.0])|

df.withColumn('label', k_udf('outvec')).drop('outvec').show(3)
+-------+--------------------+-----+--------------------+-----+
|endword|                 doc|count|            features|label|
+-------+--------------------+-----+--------------------+-----+
|     it|[please, do, not,...| 1149|(12847,[15,47,502...|    7|


from pyspark.ml.feature import CountVectorizer
cv = CountVectorizer(inputCol='words', outputCol="features")
model = cv.fit(df)
result = model.transform(df)
print(result)

DataFrame[words: array<string>, features: vector]
# Dense string array on left, dense integer vector on right
+------------------------+--------------------------------------+
|                   words |                             features |
+------------------------+--------------------------------------+
|          [Hello, world] |                 (10,[7,9],[1.0,1.0]) |
|        [How, are, you?] |           (10,[1,3,4],[1.0,1.0,1.0]) |
|[I, am, fine, thank, you]|(10,[0,2,5,6,8],[1.0,1.0,1.0,1.0,1.0])|
# первая строка -  длина разреженного вектора 10, 7 и 9 место в словаре и встречаются один раз


df_true = df.where("endword in ('she','he','hers','his','her', 'him')").withColumn('label', lit(1))
df_false = df.where("endword not in ('she','he','hers','his','her','him')").withColumn('label', lit(0))
```

```
df_examples = df_true.union(df_false)
df_train, df_eval = df_examples.randomSplit((0.60, 0.40), 42)

from pyspark.ml.classification import LogisticRegression
logistic = LogisticRegression(maxIter=50, regParam=0.6, elasticNetParam=0.3)
model = logistic.fit(df_train)
print("Training iterations: ", model.summary.totalIterations)


predicted = df_trained.transform(df_test) # добавляет столбец прогнозирования и вероятности
x = predicted.first
print("Right!" if x.label == int(x.prediction) else "Wrong") # вырный прогноз или нет


model_stats = model.evaluate(df_eval)
type(model_stats)
# pyspark.ml.classification.BinaryLogisticRegressionSummary)
print("\nPerformance: %.2f" % model_stats.areaUnderROC)
```

Введение в PySpark

```
# через SparkContext как sc - подключение к кластеру
from pyspark.sql import SparkSession
my_spark = SparkSession.builder.getOrCreate()# интерфейс для подключения (интерфейс dataframe)

print(spark.catalog.listTables()) # catalog атрибут данных внутри кластера

flights10 = spark.sql("FROM flights SELECT * LIMIT 10")
# нет таблицы в качестве аргумента т.к нет ее локально

query = "SELECT origin, dest, COUNT(*) as N FROM flights GROUP BY origin, dest"
flight_counts = spark.sql(query)
pd_counts = flight_counts.toPandas() # локально через pandas
print(pd_counts.head()) #перевод из Spark DataFrame в pandas DataFrame

# из pandas в Spark но данные хранятся локально а не в каталоге SparkSession
# методы Spark DataFrame можно а вот доступ к данным в других контекстах нет
# если .sql() то выдаст ошибку (надо временную таблицу)
# регистрация как таблицу в каталоге, но доступ к ней возможен только из конкретной SparkSession
spark_temp = spark.createDataFrame(pd.DataFrame(np.random.random(10)))

spark_temp.createOrReplaceTempView('temp')
print(spark.catalog.listTables())

airports = spark.read.csv("/usr/local/share/datasets/airports.csv", header=True)
airports.show()
```

Манипулирование данными

```
flights = spark.table("flights") # чтобы создать фрейм данных, содержащий значения таблицы flights в .catalog
flights.show()
flights = flights.withColumn("duration_hrs", flights.air_time/60)

flights.filter("air_time > 120").show()
flights.filter(flights.air_time > 120).show()

selected1 = flights.select("tailnum", "origin", "dest")
temp = flights.select(flights.origin, flights.dest, flights.carrier)
filterA = flights.origin == "SEA"
filterB = flights.dest == "PDX"
selected2 = temp.filter(filterA).filter(filterB)

flights.select((flights.air_time/60).alias("duration_hrs"))
flights.selectExpr("air_time/60 as duration_hrs") # или так аналогичный результат

avg_speed = (flights.distance/(flights.air_time/60)).alias("avg_speed")
speed1 = flights.select("origin", "dest", "tailnum", avg_speed)
speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air_time/60) as avg_speed")


# min(), .max() и .count() методы GroupedData
df.groupBy().min("col").show() # создается объект GroupedData и в конце dataframe


flights.filter(flights.origin == "PDX").groupBy().min("distance").show()
flights.filter(flights.origin == "SEA").groupBy().max("air_time").show()

flights.filter(flights.carrier == "DL").filter(flights.origin == "SEA").groupBy().avg("air_time").show()
flights.withColumn("duration_hrs", flights.air_time/60).groupBy().sum("duration_hrs").show()
```

```
by_plane = flights.groupBy("tailnum")
by_plane.count().show()
by_origin = flights.groupBy("origin")
by_origin.avg("air_time").show()

import pyspark.sql.functions as F
by_month_dest = flights.groupBy('month', 'dest')
by_month_dest.avg('dep_delay').show()
# Этот метод позволяет передать агрегатное выражение столбца, в котором используется любая
# агрегатная функция из подмодуля pyspark.sql.functions
by_month_dest.agg(F.stddev('dep_delay')).show()


# объединение
airports = airports.withColumnRenamed("faa", "dest")
flights_with_airports = flights.join(airports, on = 'dest', how = "leftouter")


model_data = model_data.withColumn("plane_year", model_data.plane_year.cast('integer'))# integer или double
model_data = model_data.withColumn("plane_age", model_data.year - model_data.plane_year)
#очистка от пропущенных
model_data = model_data.filter("arr_delay is not NULL and dep_delay is not NULL and air_time is not NULL and plane_year is not NULL")


carr_indexer = StringIndexer(inputCol="carrier", outputCol="carrier_index") # число строковой категории
carr_encoder = OneHotEncoder(inputCol="carrier_index", outputCol="carrier_fact") # one-hot (конспект)
vec_assembler = VectorAssembler(inputCols=["month", "air_time", "carrier_fact", "dest_fact", "plane_age"], outputCol="features")
# выше сибираем все признаки в один вектор и метка
from pyspark.ml import Pipeline
flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler])

import pyspark.ml.evaluation as evals
evaluator = evals.BinaryClassificationEvaluator(metricName="areaUnderROC") # как печатать


import pyspark.ml.tuning as tune
grid = tune.ParamGridBuilder()
grid = grid.addGrid(lr.regParam, np.arange(0, .1, .01))
grid = grid.addGrid(lr.elasticNetParam, [0, 1])
grid = grid.build() # выводит сетку

cv = tune.CrossValidator(estimator=lr, estimatorParamMaps=grid, evaluator=evaluator)
models = cv.fit(training)
best_lr = models.bestModel
test_results = best_lr.transform(test)
print(evaluator.evaluate(test_results))
```

Big Data Fundamentals with PySpark

```
# SparkContext как sc это типо ключа от двери
sc.version # версия Spark
sc.pythonVer # версия Python
sc.master #local[*] это URL-адрес кластера

# необработанные данные в PySpark с помощью SparkContext
rdd = sc.parallelize([1,2,3,4,5]) # PythonRDD[1] at RDD at PythonRDD.scala:53
rdd2 = sc.textFile("test.txt")
#/usr/local/share/datasets/README.md MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0


# функциональные объекты и скрытые функции
items = [1, 2, 3, 4]
list(map(lambda x: x + 2 , items)) # [3, 4, 5, 6]
list(filter(lambda x: (x%2 != 0), items)) # [1, 3]


# создание RDD создание, подгрузка, из других RDD
numRDD = sc.parallelize([1,2,3,4]) # <class 'pyspark.rdd.RDD'>
helloRDD = sc.parallelize("Hello world")
type(helloRDD)

fileRDD = sc.textFile("README.md")
type(fileRDD)

numRDD = sc.parallelize(range(10), minPartitions = 6) # разделы
fileRDD = sc.textFile("README.md", minPartitions = 6)
# узнать через  getNumPartitions()

RDD = sc.parallelize([1,2,3,4])
RDD_map = RDD.map(lambda x: x * x) # [1, 4, 9, 16]
RDD_filter = RDD.filter(lambda x: x > 2) # [3, 4]
RDD = sc.parallelize(["hello world", "how are you"])
```

```
RDD_flatmap = RDD.flatMap(lambda x: x.split(" ")) # ["hello", "world", "how", "are", "you"]

inputRDD = sc.textFile("logs.txt")
errorRDD = inputRDD.filter(lambda x: "error" in x.split()) # [...,...,...] где есть error
warningsRDD = inputRDD.filter(lambda x: "warnings" in x.split())
combinedRDD = errorRDD.union(warningsRDD)

RDD_map.collect() # [1, 4, 9, 16]
RDD_map.take(2) # [1, 4]
RDD_map.first() # [1]
RDD_flatmap.count() # 5


#практика
cubedRDD = numbRDD.map(lambda x: x**3) #PythonRDD[1] at collect at <ipython-input-1-f3ba073d143f>:5
numbers_all = cubedRDD.collect() # [...]

fileRDD_filter = fileRDD.filter(lambda line: 'Spark' in line)
for line in fileRDD_filter.take(4): # выдаст первые 4 строки
  print(line)
```

Тип данных парный RDD - ключ/значение

```
# расширенные преобразования
my_tuple = [('Sam', 23), ('Mary', 34), ('Peter', 25)]
pairRDD_tuple = sc.parallelize(my_tuple) # создание либо либо

my_list = ['Sam 23', 'Mary 34', 'Peter 25']
regularRDD = sc.parallelize(my_list)
pairRDD_RDD = regularRDD.map(lambda s: (s.split(' ')[0], s.split(' ')[1]))
# операции для pair RDD
regularRDD = sc.parallelize([("Messi", 23), ("Ronaldo", 34), ("Neymar", 22), ("Messi", 24)])
pairRDD_reducebykey = regularRDD.reduceByKey(lambda x,y : x + y) # операции с одинаковыми ключами
pairRDD_reducebykey.collect()
# [('Neymar', 22), ('Ronaldo', 34), ('Messi', 47)]

pairRDD_reducebykey_rev = pairRDD_reducebykey.map(lambda x: (x[1], x[0]))
pairRDD_reducebykey_rev.sortByKey(ascending=False).collect() # сортировка по ключу
# [(47, 'Messi'), (34, 'Ronaldo'), (22, 'Neymar')]

airports = [("US", "JFK"),("UK", "LHR"),("FR", "CDG"),("US, "SFO")]
regularRDD = sc.parallelize(airports)
pairRDD_group = regularRDD.groupByKey().collect()
for cont, air in pairRDD_group:
    print(cont, list(air)) # группировка по ключу
# FR ['CDG']
# US ['JFK', 'SFO']
# UK ['LHR']

RDD1 = sc.parallelize([("Messi", 34), ("Ronaldo", 32), ("Neymar", 24)])
RDD2 = sc.parallelize([("Ronaldo", 80), ("Neymar", 120),("Messi", 100)])
RDD1.join(RDD2).collect() # объединение по ключу
# [('Neymar', (24, 120)), ('Ronaldo', (32, 80)), ('Messi', (34, 100))]


# расширенные действия
x = [1,3,4,6]
RDD = sc.parallelize(x)
RDD.reduce(lambda x, y : x + y) # 14

# сохранение RDD в каталоге
RDD.saveAsTextFile("tempFile")
RDD.coalesce(1).saveAsTextFile("tempFile") # сохранение как один файл

rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
for kee, val in rdd.countByKey().items(): # подсчет ключей
    print(kee, val)
# ('a', 2)
# ('b', 1)
sc.parallelize([(1, 2), (3, 4)]).collectAsMap() # просто в список
# {1: 2, 3: 4}
```

PySpark SQL — это библиотека Spark для структурированных данных. В отличие от RDD PySpark SQL предоставляет больше информации о структуре данных и выполняемых вычислениях.

Ранее вы узнали о SparkContext, который является основной точкой входа для создания RDD. Точно так же SparkSession предоставляет единую точку входа для взаимодействия с базовыми функциями Spark и позволяет программировать Spark с помощью DataFrame API.

SparkSession делает для DataFrames то же, что SparkContext делает для RDD

```
# создание DataFrame через RDD или через SparkSession
iphones_RDD = sc.parallelize([("XS", 2018, 5.65, 2.79, 6.24), ("XR", 2018, 5.94, 2.98, 6.84),
                              ("X10", 2017, 5.65, 2.79, 6.13), ("8Plus", 2017, 6.23, 3.07, 7.12)])
names = ['Model','Year', 'Height', 'Width', 'Weight']
iphones_df = spark.createDataFrame(iphones_RDD, schema=names)
type(iphones_df) # pyspark.sql.dataframe.DataFrame
#Схема — это структура данных в DataFrame, которая помогает Spark более эффективно оптимизировать запросы к данным

df_csv = spark.read.csv("people.csv", header=True, inferSchema=True)
df_json = spark.read.json("people.json", header=True, inferSchema=True)
df_txt = spark.read.txt("people.txt", header=True, inferSchema=True)
#nferSchema=True дать указание считывателю DataFrame вывести схему из данных и, попытаться назначить правильный тип данных для каждого ст


df_id_age = test.select('Age').show(3)
new_df_age21 = new_df.filter(new_df.Age > 21).show(3)
test_df_age_group = test_df.groupby('Age').count().orderBy('Age').show(3)
test_df_no_dup = test_df.select('User_ID', 'Gender', 'Age').dropDuplicates()
test_df_sex = test_df.withColumnRenamed('Gender', 'Sex') # переименовать было стало
test_df.printSchema()
#    |-- User_ID: integer (nullable = true)
#    |-- Product_ID: string (nullable = true)
test_df.columns #['User_ID', 'Gender', 'Age'
test_df.describe().show()
```

SQL-запросы нельзя запускать напрямую к DataFrame. Чтобы выполнить SQL-запросы к существующему фрейму данных, мы можем использовать функцию createOrReplaceTempView для создания временной таблицы, как показано в этом примере

```
df.createOrReplaceTempView("table1")
df2 = spark.sql("SELECT field1, field2 FROM table1")
df2.collect()


# визуализация либо pyspark_dist_explore library, toPandas(), HandySpark library
test_df = spark.read.csv("test.csv", header=True, inferSchema=True)
test_df_age = test_df.select('Age')
hist(test_df_age, bins=20, color="red")
# через библиотеку
test_df = spark.read.csv('test.csv', header=True, inferSchema=True)
hdf = test_df.toHandy()
hdf.cols["Age"].hist()
```

PySpark MLlib

```
from pyspark.mllib.recommendation import ALS # совместной фильтрации для рекоменательных систем
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.clustering import KMeans
#pyspark.mllib - это встроенная библиотека для API на основе RDD.


# совместная фильтрация
from pyspark.mllib.recommendation import Rating
r = Rating(user = 1, product = 2, rating = 5.0)
(r[0], r[1], r[2]) # (1, 2, 5.0) пользователь, продукт и рейтинг

data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) # создаем RDD
training, test=data.randomSplit([0.6, 0.4])
training.collect()
test.collect()
# используя алгоритм ALS ищем продукты для клиентов
r1 = Rating(1, 1, 1.0)
r2 = Rating(1, 2, 2.0)
r3 = Rating(2, 1, 2.0)
ratings = sc.parallelize([r1, r2, r3])
ratings.collect()
# [Rating(user=1, product=1, rating=1.0), Rating(user=1, product=2, rating=2.0),
# Rating(user=2, product=1, rating=2.0)]
model = ALS.train(ratings, rank=10, iterations=10)
# прогнозирвание рейтингов для пар пользователь-продукт
unrated_RDD = sc.parallelize([(1, 2), (1, 1)])
predictions = model.predictAll(unrated_RDD)
predictions.collect()
#[Rating(user=1, product=1, rating=1.0000278574351853), Rating(user=1, product=2, rating=1.9890355703778122)]
rates = ratings.map(lambda x: ((x[0], x[1]), x[2]))
rates.collect()
  #[((1, 1), 1.0), ((1, 2), 2.0), ((2, 1), 2.0)]
preds = predictions.map(lambda x: ((x[0], x[1]), x[2]))
preds.collect()
```

```
  #[((1, 1), 1.0000278574351853), ((1, 2), 1.9890355703778122)]
rates_preds = rates.join(preds)
rates_preds.collect()
  #[((1, 2), (2.0, 1.9890355703778122)), ((1, 1), (1.0, 1.0000278574351853))]
MSE = rates_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean() # ошибка MSE


# пример
data = sc.textFile(file_path) #['1,31,2.5,1260759144',
                              #'1,1029,3.0,1260759179']
ratings = data.map(lambda l: l.split(',')) #[['1', '31', '2.5', '1260759144'],
                                           # ['1', '1129', '2.0', '1260759185']]

ratings_final = ratings.map(lambda line: Rating(int(line[0]),int(line[1]),float(line[2])))#[Rating(user=1, product=31, rating=2.5),
                                                                                          #Rating(user=1, product=1129, rating=2.0)]
training_data, test_data = ratings_final.randomSplit([0.8, 0.2]) #[Rating(user=1, product=31, rating=2.5),
                                                                 #Rating(user=1, product=1029, rating=3.0)]
model = ALS.train(training_data, rank=10, iterations=10)
testdata_no_rating = test_data.map(lambda p: (p[0], p[1])) # [(1, 1129)]
predictions = model.predictAll(testdata_no_rating)
predictions.take(2) #[Rating(user=599, product=69069, rating=2.771806602194091),
```

Классификация

Типы данных PySpark MLlib:

- Vector

> Плотные
>
> ```
> denseVec = Vectors.dense([1.0, 2.0, 3.0])
> DenseVector([1.0, 2.0, 3.0])
> ```
>
> Разреженные
>
> ```
> sparseVec = Vectors.sparse(4, {1: 1.0, 3: 5.5})
> SparseVector(4, {1: 1.0, 3: 5.5})
> ```

- LabelledPoint

```
positive = LabeledPoint(1.0, [1.0, 0.0, 3.0])
negative = LabeledPoint(0.0, [2.0, 1.0, 1.0]) # метка класса и признаки
```

```
from pyspark.mllib.feature import HashingTF
sentence = "hello hello world"
words = sentence.split()
tf = HashingTF(10000)
tf.transform(words)
# SparseVector(10000, {3065: 1.0, 6861: 2.0}) #кол-во признаков, номер признака и кол-во

data = [LabeledPoint(0.0, [0.0, 1.0]), LabeledPoint(1.0, [1.0, 0.0])]
RDD = sc.parallelize(data)
lrm = LogisticRegressionWithLBFGS.train(RDD)
lrm.predict([1.0, 0.0])
lrm.predict([0.0, 1.0])


# пример
spam_rdd = sc.textFile(file_path_spam)
non_spam_rdd = sc.textFile(file_path_non_spam)

spam_words = spam_rdd.flatMap(lambda email: email.split(' ')) #['...', '...' и тд]
non_spam_words = non_spam_rdd.flatMap(lambda email: email.split(' '))

tf = HashingTF(numFeatures=200)
spam_features = tf.transform(spam_words)
non_spam_features = tf.transform(non_spam_words)
# [SparseVector(200, {103: 1.0, 111: 1.0, 119: 1.0}),
# SparseVector(200, {14: 1.0, 89: 1.0, 193: 1.0, 199: 1.0}),

spam_samples = spam_features.map(lambda features:LabeledPoint(1, features))
non_spam_samples = non_spam_features.map(lambda features:LabeledPoint(0, features))
#[LabeledPoint(1.0, (200,[103,111,119],[1.0,1.0,1.0])),
# LabeledPoint(1.0, (200,[14,89,193,199],[1.0,1.0,1.0,1.0])),

samples = spam_samples.join(non_spam_samples)
```

```
train_samples,test_samples = samples.randomSplit([0.8, 0.2])
model = LogisticRegressionWithLBFGS.train(train_samples)
predictions = model.predict(test_samples.map(lambda x: x.features)) #[0, 1, 0, 1]
labels_and_preds = test_samples.map(lambda x: x.label).zip(predictions)
# [(1.0, 0), (1.0, 1), (1.0, 0), (1.0, 0)]
accuracy = labels_and_preds.filter(lambda x: x[0] == x[1]).count() / float(test_samples.count())
print("Model accuracy : {:.2f}".format(accuracy))
```

Кластеризация

```
RDD = sc.textFile("WineData.csv").map(lambda x: x.split(",")).\
map(lambda x: [float(x[0]), float(x[1])])
RDD.take(5) #[[14.23, 2.43], [13.2, 2.14], [13.16, 2.67], [14.37, 2.5], [13.24, 2.87]]

from pyspark.mllib.clustering import KMeans
model = KMeans.train(RDD, k = 2, maxIterations = 10)
model.clusterCenters # центр кластера
# [array([12.25573171, 2.28939024]), array([13.636875 , 2.43239583])]

from math import sqrt
def error(point):
    center = model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))
WSSSE = RDD.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE)) #  77.962

wine_data_df = spark.createDataFrame(RDD, schema=["col1", "col2"])
wine_data_df_pandas = wine_data_df.toPandas()
cluster_centers_pandas = pd.DataFrame(model.clusterCenters, columns=["col1", "col2"])
cluster_centers_pandas.head()

plt.scatter(wine_data_df_pandas["col1"], wine_data_df_pandas["col2"]);
plt.scatter(cluster_centers_pandas["col1"], cluster_centers_pandas["col2"], color="red", marker="x"
# сначала конвертируем RDD в Spark DataFrame, а затем в Pandas DataFrame
# конвертируем кластерные центры из модели KMeans в Pandas DataFrame
```

Cleaning Data with PySpark

```
# схема
import pyspark.sql.types import *# указывается имя поля, тип, и могут ли быть нулевыми
peopleSchema = StructType([StructField('name', StringType(), True),
                           StructField('age', IntegerType(), True),
                           StructField('city', StringType(), True)])
people_df = spark.read.format('csv').load(name='rawdata.csv', schema=peopleSchema)


# загрузка и чтение
df = spark.read.format('parquet').load('filename.parquet')
df = spark.read.parquet('filename.parquet')

df.write.format('parquet').save('filename.parquet')
df.write.parquet('filename.parquet')


flight_df = spark.read.parquet('flights.parquet') # создаем df
flight_df.createOrReplaceTempView('flights') #псевдоним данных Parquet в виде таблицы SQL
short_flights_df = spark.sql('SELECT * FROM flights WHERE flightduration < 100')


# df неизменяемые, всегда создается новый df
voter_df.filter(voter_df.name.like('M%')) # только те строки с M начинаются
voters = voter_df.select('name', 'position')

voter_df.filter(voter_df.date > '1/1/2019') # или voter_df.where(...)
voter_df.select(voter_df.name)
voter_df.withColumn('year', voter_df.date.year)
voter_df.drop('unused_column')


voter_df.filter(voter_df['name'].isNotNull()) # удалание пустых значений
voter_df.filter(voter_df.date.year > 1800)
voter_df.where(voter_df['_c0'].contains('VOTE'))
voter_df.where(~ voter_df._c1.isNull())

import pyspark.sql.functions as F
voter_df.withColumn('upper', F.upper('name'))

voter_df.withColumn('splits', F.split('name',' ')) # столбец списков
```

```
voter_df.withColumn('year', voter_df['_c4'].cast(IntegerType()))

# работа с ArrayType() - список
.size() # кол-во элементов
.getItem(<index>) # элемент в индексе


# пример
voter_df = voter_df.withColumn('splits', F.split(voter_df.VOTER_NAME, '\s+'))
voter_df = voter_df.withColumn('first_name', voter_df.splits.getItem(0))
voter_df = voter_df.withColumn('last_name', voter_df.splits.getItem(F.size('splits') - 1))
voter_df = voter_df.drop('splits')
voter_df.show()


df.select(df.Name, df.Age, F.when(df.Age >= 18, "Adult")) #из pyspark.sql.functions
+-----+----+------+
| Name| Age|      |
+-----+----+------+
|Alice|  14|      |
|  Bob|  18| Aduit|
+-----+----+------+
df.select(df.Name, df.Age, when(df.Age >= 18, "Adult").when(df.Age < 18, "Minor"))
# иначе
df.select(df.Name, df.Age, when(df.Age >= 18, "Adult").otherwise("Minor"))


# пользовательские функции UDF
import pyspark.sql.functions.udf
def reverseString(mystr):
    return mystr[::-1]
udfReverseString = udf(reverseString, StringType()) #обернуть функцию и сохранить ее в переменной
# имя метода и тип данных Spark для возврата
user_df = user_df.withColumn('ReverseName', udfReverseString(user_df.Name))
# функция применяется для каждой строки

def sortingCap():
    return random.choice(['G', 'H', 'R', 'S'])
udfSortingCap = udf(sortingCap, StringType())
user_df = user_df.withColumn('Class', udfSortingCap()) # случайно возвращает букву

# пример с сортировкой
voter_df = df.select(df["VOTER NAME"]).distinct()
voter_df = voter_df.withColumn('ROW_ID', F.monotonically_increasing_id())
voter_df.orderBy(voter_df.ROW_ID.desc()).show(10)


print("\nThere are %d partitions in the voter_df DataFrame.\n" % voter_df.rdd.getNumPartitions())# кол-во разделов
print("\nThere are %d partitions in the voter_df_single DataFrame.\n" % voter_df_single.rdd.getNumPartitions())

voter_df = voter_df.withColumn('ROW_ID', F.monotonically_increasing_id())
voter_df_single = voter_df_single.withColumn('ROW_ID', F.monotonically_increasing_id())

voter_df.orderBy(voter_df.ROW_ID.desc()).show(10)
voter_df_single.orderBy(voter_df_single.ROW_ID.desc()).show(10)


previous_max_ID = voter_df_march.select('ROW_ID').rdd.max()[0]

voter_df_april = voter_df_april.withColumn('ROW_ID', F.monotonically_increasing_id() + previous_max_ID)
voter_df_march.select('ROW_ID').show()
voter_df_april.select('ROW_ID').show()
```

Кэширование

```
voter_df = spark.read.csv('voter_data.txt.gz') # создание df
voter_df.cache().count() # кэш трансформер
# либо кэшировать
voter_df = voter_df.withColumn('ID', monotonically_increasing_id())
voter_df = voter_df.cache() # для кэширования
voter_df.show() # тут уже кэшируется
print(voter_df.is_cached) # кэширование
voter_df.unpersist() # отмена кэширования


start_time = time.time()
departures_df = departures_df.distinct().cache()
print("Counting %d rows took %f seconds" % (departures_df.count(), time.time() - start_time))

start_time = time.time() # сброс времени
print("Counting %d rows again took %f seconds" % (departures_df.count(), time.time() - start_time))
```

```
# Counting 139358 rows took 4.620641 seconds
# Counting 139358 rows again took 0.905145 seconds
```

Фишки импорта

```
airport_df = spark.read.csv('airports-*.txt.gz') # один оператор импорта, файлов несколько


df_csv = spark.read.csv('singlelargefile.csv')
df_csv.write.parquet('data.parquet')
df = spark.read.parquet('data.parquet')

# Пример
full_df = spark.read.csv('departures_full.txt.gz')
split_df = spark.read.csv('departures_0*.txt.gz') # разбитый

start_time_a = time.time()
print("Total rows in full DataFrame:\t%d" % full_df.count())
print("Time to run: %f" % (time.time() - start_time_a))

start_time_b = time.time()
print("Total rows in split DataFrame:\t%d" % split_df.count())
print("Time to run: %f" % (time.time() - start_time_b))


spark.conf.get(<configuration name>) # для чтения параметра конфигурации с именем
spark.conf.set(<configuration name>) # для записи параметра
# пример
app_name = spark.conf.get('spark.app.name') # Name: pyspark-shell
driver_tcp_port = spark.conf.get('spark.driver.port') #  Driver TCP port: 32931
num_partitions = spark.conf.get('spark.sql.shuffle.partitions') # Number of partitions: 200

# пример
# Store the number of partitions in variable
before = departures_df.rdd.getNumPartitions()
# Configure Spark to use 500 partitions
spark.conf.set('spark.sql.shuffle.partitions', 500)
# Recreate the DataFrame using the departures data file
departures_df = spark.read.csv('departures.txt.gz').distinct()
# Print the number of partitions for each instance
print("Partition count before change: %d" % before) # 1
print("Partition count after change: %d" % departures_df.rdd.getNumPartitions()) # 1


voter_df = df.select(df['VOTER NAME']).distinct()
voter_df.explain() # план, который будет запущен для получения результатов



# перетасовка
# Limit use of .repartition(num_partitions)
# Use .coalesce(num_partitions) instead
# Use care when calling .join()
# Use .broadcast()
# May not need to limit i

# Broadcasting
from pyspark.sql.functions import broadcast
combined_df = df_1.join(broadcast(df_2))


# пример
# Import the data to a DataFrame
departures_df = spark.read.csv('2015-departures.csv.gz', header=True)

# Remove any duration of 0
departures_df = departures_df.filter(departures_df[3] > 0)

# Add an ID column
departures_df = departures_df.withColumn('id', F.monotonically_increasing_id())

# Write the file out to JSON format
departures_df.write.json('output.json', mode='overwrite')


# продолжи
```

Feature Engineering with PySpark

```
spark.version # версия спарк тут version 2.3.1
import sys # версия питона
```

```
sys.version_info

spark.read.json('example.json') # всё в df
spark.read.csv('example.csv')
spark.read.parquet('example.parq')
df = spark.read.parquet('example.parq')

df.count()
df.columns
len(df.columns)
df.dtypes # [(название столбца, тип)...]

df.describe(['LISTPRICE']).show() # можно для одного, всех или списка
df.agg({'SALESCLOSEPRICE': 'mean'}).collect()#[Row(avg(SALESCLOSEPRICE)=262804.4668)]
# так как mean агрегатная функция
# пример
from pyspark.sql.functions import skewness
print(df.agg({'LISTPRICE': 'skewness'}).collect())

df.cov('SALESCLOSEPRICE', 'YEARBUILT') # 1281910.384

# для использования seaborn надо преобразовать df PySpark в pandas PySpark но могут быть сбои
# берем подвыборку
df.sample(False, 0.5, 42).count() # без повторения, объем, начальное зерно
# тип pyspark.sql.dataframe.DataFrame
import seaborn as sns
sample_df = df.select(['SALESCLOSEPRICE']).sample(False, 0.5, 42)
pandas_df = sample_df.toPandas()
sns.distplot(pandas_df)

import seaborn as sns
s_df = df.select(['SALESCLOSEPRICE', 'SQFTABOVEGROUND'])
s_df = s_df.sample(False, 0.5, 42)
pandas_df = s_df.toPandas()
sns.lmplot(x='SQFTABOVEGROUND', y='SALESCLOSEPRICE', data=pandas_df)


cols_to_drop = ['NO', 'UNITNUMBER', 'CLASS']
df = df.drop(*cols_to_drop)
df = df.where(~df['POTENTIALSHORTSALE'].like('Not Disclosed'))

# удаление выбросов
std_val = df.agg({'SALESCLOSEPRICE': 'stddev'}).collect()[0][0]
mean_val = df.agg({'SALESCLOSEPRICE': 'mean'}).collect()[0][0]
hi_bound = mean_val + (3 * std_val)
low_bound = mean_val - (3 * std_val)
df = df.where((df['LISTPRICE'] < hi_bound) & (df['LISTPRICE'] > low_bound))
# удаление пропущенных
df = df.dropna() # либо хотя бы где-то есть
df = df.dropna(how='all', subset['LISTPRICE', 'SALESCLOSEPRICE']) # по столбцам
df = df.dropna(thresh=2) # как минимум в двух столбцах

# удаление дубликатов
df.dropDuplicates()
df.dropDuplicates(['streetaddress'])


# практика
df.select(['ASSUMABLEMORTGAGE']).distinct().show()
yes_values = ['Yes w/ Qualifying', 'Yes w/No Qualifying']
text_filter = ~df['ASSUMABLEMORTGAGE'].isin(yes_values) | df['ASSUMABLEMORTGAGE'].isNull()
df = df.where(text_filter)
print(df.count())
```

```
# min-max scaling
# define min and max values and collect them
max_days = df.agg({'DAYSONMARKET': 'max'}).collect()[0][0]
min_days = df.agg({'DAYSONMARKET': 'min'}).collect()[0][0]
# create a new column based off the scaled data
df = df.withColumn("scaled_days", (df['DAYSONMARKET']-min_days)/(max_days-min_days))
df[['scaled_days']].show(5)

# стандартизация
mean_days = df.agg({'DAYSONMARKET': 'mean'}).collect()[0][0]
stddev_days = df.agg({'DAYSONMARKET': 'stddev'}).collect()[0][0]
df = df.withColumn("ztrans_days", (df['DAYSONMARKET']-mean_days)/stddev_days)
df.agg({'ztrans_days': 'mean'}).collect()
df.agg({'ztrans_days': 'stddev'}).collect()

#log-scaling
from pyspark.sql.functions import log
df = df.withColumn('log_SalesClosePrice', log(df['SALESCLOSEPRICE']))


df.where(df['ROOF'].isNull()).count() # подсчет пропущенных
# построение тепловой карты
import seaborn as sns
sub_df = df.select(['ROOMAREA1'])
sample_df = sub_df.sample(False, .5, 4)
pandas_df = sample_df.toPandas()
sns.heatmap(data=pandas_df.isnull()) # для True False

# заполенение пропущенных
df.fillna(0, subset=['DAYSONMARKET'])

col_mean = df.agg({'DAYSONMARKET': 'mean'}).collect()[0][0]
df.fillna(col_mean, subset=['DAYSONMARKET'])


# присоединение
DataFrame.join(other, on=None, how=None)

cond = [df['OFFMARKETDATE'] == hdf['dt']]
df = df.join(hdf, on=cond, 'left')
df.where(~df['nm'].isNull()).count()
# через SQL запрос
df.createOrReplaceTempView("df")
hdf.createOrReplaceTempView("hdf")
sql_df = spark.sql("""SELECT*
                      FROM df
                      LEFT JOIN hdf ON df.OFFMARKETDATE = hdf.dt""")


# пример
def min_max_scaler(df, cols_to_scale):
  for col in cols_to_scale:
    max_days = df.agg({col: 'max'}).collect()[0][0]
    min_days = df.agg({col: 'min'}).collect()[0][0]
    new_column_name = 'scaled_' + col
    df = df.withColumn(new_column_name,(df[col]-min_days)/(max_days-min_days))
  return df

df = min_max_scaler(df, cols_to_scale)
df[['DAYSONMARKET', 'scaled_DAYSONMARKET']].show()

# пример
walk_df = walk_df.withColumn('longitude', walk_df['longitude'].cast('double'))
walk_df = walk_df.withColumn('latitude', walk_df['latitude'].cast('double'))

df = df.withColumn('longitude', round(df['longitude'], 5))
df = df.withColumn('latitude', round(df['latitude'], 5))

condition = [(df['longitude'] == walk_df['longitude']), (df['latitude'] == walk_df['latitude'])]

join_df = df.join(walk_df, on=condition, how='left')
print(join_df.where(~join_df['walkscore'].isNull()).count())
```

Генериование признаков

```
df = df.withColumn('TSQFT', (df['WIDTH'] * df['LENGTH']))
df = df.withColumn('TSQFT', (df['SQFTBELOWGROUND'] + df['SQFTABOVEGROUND']))
df = df.withColumn('PRICEPERTSQFT', (df['LISTPRICE'] / df['TSQFT']))
df = df.withColumn('DAYSONMARKET', datediff('OFFMARKETDATE', 'LISTDATE'))
```

```
from pyspark.sql.functions import to_date # либо to_timestamp для даты Spark
df = df.withColumn('LISTDATE', to_date('LISTDATE'))
df[['LISTDATE']].show(2)

from pyspark.sql.functions import year, month
df = df.withColumn('LIST_YEAR', year('LISTDATE'))
df = df.withColumn('LIST_MONTH', month('LISTDATE'))

from pyspark.sql.functions import dayofmonth, weekofyear
df = df.withColumn('LIST_DAYOFMONTH', dayofmonth('LISTDATE'))
df = df.withColumn('LIST_WEEKOFYEAR', weekofyear('LISTDATE'))

from pyspark.sql.functions import datediff
df.withColumn('DAYSONMARKET', datediff('OFFMARKETDATE', 'LISTDATE'))

from pyspark.sql.functions import lag
from pyspark.sql.window import Window
w = Window().orderBy(m_df['DATE'])
m_df = m_df.withColumn('MORTGAGE-1wk', lag('MORTGAGE', count=1).over(w))
m_df.show(3)
+----------+------------+----------------+
|      DATE|    MORTGAGE|    MORTGAGE-1wk|
+----------+------------+----------------+
|2013-10-10|        4.23|            null|
|2013-10-17|        4.28|            4.23|
|2013-10-24|        4.13|            4.28|
+----------+------------+----------------+


from pyspark.sql.functions import when
find_under_8 = df['ROOF'].like('%Age 8 Years or Less%')
find_over_8 = df['ROOF'].like('%Age Over 8 Years%')

df = df.withColumn('old_roof', (when(find_over_8, 1).when(find_under_8, 0).otherwise(None)))
df[['ROOF', 'old_roof']].show(3, truncate=100)

from pyspark.sql.functions import split
split_col = split(df['ROOF'], ',')

df = df.withColumn('Roof_Material', split _col.getItem(0))
df[['ROOF', 'Roof_Material']].show(5, truncate=100)


# из этого
+---+-----------------------------------------------+
|NO |                                      roof_list|
+---+-----------------------------------------------+
| 2 |  Asphalt Shingles, Pitched, Age 8 Years or Less|
# в это (пивот)
+---+--------------------+
|NO |        ex_roof_list|
+---+--------------------+
|  2|    Asphalt Shingles|
|  2|             Pitched|
|  2| Age 8 Years or Less|

from pyspark.sql.functions import split, explode, lit, coalesce, first
df = df.withColumn('roof_list', split(df['ROOF'], ',')) #в столбец списка
ex_df = df.withColumn('ex_roof_list', explode(df['roof_list'])) # список в один столбец
# Create a dummy column of constant value
ex_df = ex_df.withColumn('constant_val', lit(1))
# Pivot the values into boolean columns
piv_df = ex_df.groupBy('NO').pivot('ex_roof_list').agg(coalesce(first('constant_val')))

#пример
from pyspark.sql.functions import coalesce, first
# Pivot
piv_df = ex_df.groupBy('NO').pivot('ex_garage_list').agg(coalesce(first('constant_val')))
joined_df = df.join(piv_df, on='NO', how='left')
zfill_cols = piv_df.columns
zfilled_df = joined_df.fillna(0, subset=zfill_cols) # таблица + one-hot для слов в списке


# бинаризация
from pyspark.ml.feature import Binarizer
# важно чтобы тип был double
df = df.withColumn('FIREPLACES', df['FIREPLACES'].cast('double'))
# Create binarizing transformer
bin = Binarizer(threshold=0.0, inputCol='FIREPLACES', outputCol='FireplaceT')
df = bin.transform(df) # все что выше 0.0 -> 1
+----------+-------------+
|FIREPLACES|   FireplaceT|
+----------+-------------+
```

```
|       0.0|         0.0|
|       1.0|         1.0|
|       2.0|         1.0|
```

```
#Bucketing (сегментирование, биннинг, порядковые переменные)
from pyspark.ml.feature import Bucketizer
splits = [0, 1, 2, 3, 4, float('Inf')]
# Create bucketing transformer
buck = Bucketizer(splits=splits, inputCol='BATHSTOTAL', outputCol='baths')
df = buck.transform(df)
df[['BATHSTOTAL', 'baths']].show(4)
# от 0 до 1 -> 1, от 1 до 2 -> 2, от 2 до 3 -> 3
```

```
#one-hot кодирование
from pyspark.ml.feature import OneHotEncoder, StringIndexer
# Create indexer transformer (сопоставление слова с числом)
stringIndexer = StringIndexer(inputCol='CITY', outputCol='City_Index')
# Fit transformer
model = stringIndexer.fit(df)
# Apply transformer
indexed = model.transform(df)
```

```
encoder = OneHotEncoder(inputCol='City_Index', outputCol='City_Vec')
# Apply the encoder transformer
encoded_df = encoder.transform(indexed)
encoded_df[['City_Vec']].show(4) # без последней категории
```

```
+-------------+
|     City_Vec|
+-------------+
|    (4,[],[])|
|    (4,[],[])|
|(4,[2],[1.0])|
|(4,[2],[1.0])|
+-------------+
```

Машинное обучение с PySpark

ml.regression

- GeneralizedLinearRegression
- IsotonicRegression
- LinearRegression
- DecisionTreeRegression
- GBTRegression (деревья с повышением градиента)
- RandomForestRegression

```
max_date = df.agg({'OFFMKTDATE': 'max'}).collect()[0][0]
min_date = df.agg({'OFFMKTDATE': 'min'}).collect()[0][0]
from pyspark.sql.functions import datediff
range_in_days = datediff(max_date, min_date)
# Find the date to split the dataset on
from pyspark.sql.functions import date_add
split_in_days = round(range_in_days * 0.8)
split_date = date_add(min_date, split_in_days)
# Split the data into 80% train, 20% test
train_df = df.where(df['OFFMKTDATE'] < split_date)
test_df = df.where(df['OFFMKTDATE'] >= split_date).where(df['LISTDATE'] >= split_date)
```

```
# пример с lit (откуда lit)
from pyspark.sql.functions import datediff, to_date, lit
split_date = to_date(lit('2017-12-10'))
test_df = df.where(df['OFFMKTDATE'] >= split_date).where(df['LISTDATE'] <= split_date)
test_df = test_df.withColumn('DAYSONMARKET_Original', test_df['DAYSONMARKET'])
test_df = test_df.withColumn('DAYSONMARKET', datediff(split_date, 'LISTDATE'))
```

```
print((df.count(), len(df.columns))) # (5000, 126)
from pyspark.ml.feature import VectorAssembler
# все признаки в одном столбце, но VectorAssembler не умеет работать с NaN
df = df.fillna(-1)
features_cols = list(df.columns)
features_cols.remove('SALESCLOSEPRICE')

vec = VectorAssembler(inputCols=features_cols, outputCol='features')
df = vec.transform(df)
ml_ready_df = df.select(['SALESCLOSEPRICE', 'features'])
ml_ready_df.show(5)
```

```
+----------------+--------------------+
```

```
| SALESCLOSEPRICE|            features|
+----------------+--------------------+
|         143000 |(125,[0,1,2,3,5,6...|
|         190000 |(125,[0,1,2,3,5,6...|
|         225000 |(125,[0,1,2,3,5,6...|
```

```
# пример
df = df.fillna(-1, subset=['WALKSCORE', 'BIKESCORE'])
indexers = [StringIndexer(inputCol=col, outputCol=col+"_IDX").setHandleInvalid("keep") for col in categorical_cols]
indexer_pipeline = Pipeline(stages=indexers)
df_indexed = indexer_pipeline.fit(df).transform(df)

df_indexed = df_indexed.drop(*categorical_cols)
print(df_indexed.dtypes)
# категории в числа в своих столбцах
```

Отсутствующие значения обрабатываются случайными лесами внутри системы, где они разделяются на отсутствующие значения. До тех пор, пока вы заменяете их чем-то, выходящим за пределы диапазона нормальных значений, они будут обрабатываться правильно. Аналогично, категориальные объекты нужно сопоставлять только с числами, все они могут оставаться в одном столбце с помощью StringIndexer, как мы видели в главе 3. Кодировка OneHot, которая преобразует каждое возможное значение в его собственную логическую функцию, не требуется.

```
from pyspark.ml.regression import RandomForestRegressor
rf = RandomForestRegressor(featuresCol="features", labelCol="SALESCLOSEPRICE",
                           predictionCol="Prediction_Price", seed=42)
# столбец признаков, лейблов, выхода и зерно
model = rf.fit(train_df)
predictions = model.transform(test_df)
predictions.select("Prediction_Price", "SALESCLOSEPRICE").show(5)

from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(labelCol="SALESCLOSEPRICE", predictionCol="Prediction_Price")
rmse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})
print('RMSE: ' + str(rmse)) # есть необъяснимая дисперсия
print('R^2: ' + str(r2))
```

```
# важность признака
import pandas as pd
fi_df = pd.DataFrame(model.featureImportances.toArray(), columns=['importance'])
fi_df['feature'] = pd.Series(feature_cols)
fi_df.sort_values(by=['importance'], ascending=False, inplace=True)
model_df.head(9)
|                 feature |importance|
|-------------------------|----------|
|               LISTPRICE | 0.312101 |
|       ORIGINALLISTPRICE | 0.202142 |
|              LIVINGAREA | 0.124239 |

model.save('rfr_real_estate_model') # сохранение модели, тут целый каталог файлов
from pyspark.ml.regression import RandomForestRegressionModel
# для загрузки
model2 = RandomForestRegressionModel.load('rfr_real_estate_model')
```

Machine Learning with PySpark

```
import pyspark #делает функциональность Spark доступной в интерпретаторе Python
pyspark.__version__ # '2.4.1'
# подмодули
pyspark.sql #Structured Data
pyspark.streaming #Streaming Data
pyspark.mllib #Machine Learning на RDD (устаревшая)
pyspark.ml # текущее устаревшее на DataFrame
```

Подключение к Spark - указать Spark где находится кластер через **spark://<IP address | DNS name>:** Пример: spark://13.59.151.161:7077 т.е указать URL-адрес Spark, который указывает сетевое расположение главного узла кластера. URL-адрес состоит из IP-адреса или DNS-имени и номера порта. Порт по умолчанию для Spark — 7077, но его все равно необходимо указать явно.

Но мы создадим локальный кластер с указанием количества ядер: local - одно, local[4] - 4, local[*] - все доступные

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local[*]').appName('first_spark_application').getOrCreate()
# подключение к Spark создавая объект SparkSession, указывая расположение и имя сессии
spark.stop() # остановка SparkSession
```

```
cars = spark.read.csv('cars.csv', header=True) # параметры header, sep, schema и inferSchema
# nullValue заполнитель отсутсвующих данных
cars.printSchema()
# root
# |-- mfr: string (nullable = true)
# |-- mod: string (nullable = true)
# |-- org: string (nullable = true)
cars = spark.read.csv("cars.csv", header=True, inferSchema=True)
cars.dtypes # даем автоматически определить тип

cars = spark.read.csv("cars.csv", header=True, inferSchema=True, nullValue='NA')
# 'NA' будет как NaN
schema = StructType([StructField("maker", StringType()), StructField("model", StringType()),
                     StructField("origin", StringType()), StructField("type", StringType()),
                     StructField("cyl", IntegerType()), StructField("size", DoubleType()),
                     StructField("weight", IntegerType()), StructField("length", DoubleType()),
                     StructField("rpm", IntegerType()), StructField("consumption", DoubleType())])
cars = spark.read.csv("cars.csv", header=True, schema=schema, nullValue='NA')
# Это также позволяет выбирать альтернативные имена столбцов
```

Классификация (дерево и логистическая)

```
# практика
from pyspark.sql.functions import round
flights_km = flights.withColumn('km', round(flights.mile * 1.60934, 0)).drop('mile')
flights_km = flights_km.withColumn('label', (flights_km.delay >= 15).cast('integer'))
# из True False в 1 и 0


cars = cars.drop('maker', 'model') # либо удалить либо выбрать
cars = cars.select('origin', 'type', 'cyl', 'size', 'weight', 'length', 'rpm', 'consumption')

cars.filter('cyl IS NULL').count() # подсчет пустых
cars = cars.filter('cyl IS NOT NULL') # либо прям удаление
cars = cars.dropna()

from pyspark.sql.functions import round
cars = cars.withColumn('mass', round(cars.weight / 2.205, 0))
cars = cars.withColumn('length', round(cars.length * 0.0254, 3))

# категории в числа
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol='type', outputCol='type_idx')
# чем чаще тем ближе к 0, контроль через stringOrderType
indexer = indexer.fit(cars)
cars = indexer.transform(cars)

from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler(inputCols=['cyl', 'size'], outputCol='features')
assembler.transform(cars)
+---+----+---------+
|cyl|size| features|
+---+----+---------+
|  3| 1.0|[3.0,1.0]|
|  4| 1.3|[4.0,1.3]|
# разеделение на обучение и тест
cars_train, cars_test = cars.randomSplit([0.8, 0.2], seed=23)


from pyspark.ml.classification import DecisionTreeClassifier
tree = DecisionTreeClassifier()
tree_model = tree.fit(cars_train)
prediction = tree_model.transform(cars_test) # добавляет новые столбцы
+-----+----------+----------------------------------------+
|label|prediction|                             probability |
+-----+----------+----------------------------------------+
| 1.0 |      0.0 |[0.9615384615384616,0.0384615384615385]|
# 0 с вероятностью 0.961
prediction.groupBy("label", "prediction").count().show()
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|  1.0|       1.0|    8| <- True positive (TP)
|  0.0|       1.0|    2| <- False positive (FP)
|  1.0|       0.0|    3| <- False negative (FN)
|  0.0|       0.0|    6| <- True negative (TN)


from pyspark.ml.classification import LogisticRegression
logistic = LogisticRegression()
```

```
logistic = logistic.fit(cars_train)
prediction = logistic.transform(cars_test)
+-----+----------+--------------------------------------+
|label|prediction|                           probability |
+-----+----------+--------------------------------------+
| 0.0 |      0.0 |[0.8683802216422138,0.1316197783577862]|
| 0.0 |      1.0 |[0.1343792056399585,0.8656207943600416]|

from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator()
evaluator.evaluate(prediction, {evaluator.metricName: 'weightedPrecision'})
# другие метрики weightedRecall, accuracy, f1


from pyspark.sql.functions import regexp_replace
#Дефис экранируется обратной косой чертой, потому что он имеет другое значение в контексте регулярных выражений.
# Убегая от него, вы указываете Spark буквально интерпретировать дефис
REGEX = '[,\\-]'
books = books.withColumn('text', regexp_replace(books.text, REGEX, ' '))
# из   Forever, or a Long, Long Time   в   Forever or a Long Long Time
from pyspark.ml.feature import Tokenizer # список токенов и дополнительно слова в нижнем регистре
books = Tokenizer(inputCol="text", outputCol="tokens").transform(books)

from pyspark.ml.feature import StopWordsRemover # удаление стоп слов
stopwords = StopWordsRemover()
stopwords.getStopWords() # получаем список стоп-слов
stopwords = stopwords.setInputCol('tokens').setOutputCol('words') # можно было и обычно
books = stopwords.transform(books)


from pyspark.ml.feature import HashingTF
hasher = HashingTF(inputCol="words", outputCol="hash", numFeatures=32)
books = hasher.transform(books)
#из [forever, long, long, time]  в  (32,[8,13,14],[2.0,1.0,1.0]) кол-во признаков, хэши, кол-во раз
# после можно получить tf-idf представление
from pyspark.ml.feature import IDF
books = IDF(inputCol="hash", outputCol="features").fit(books).transform(books)
#[forever, long, long, time] (32,[8,13,14],[2.598,1.299,1.704])
```

Регрессия

```
#необходимо преобразовать значения индекса в формат, в котором можете выполнять значимые математические операции
from pyspark.ml.feature import OneHotEncoder # one-hot
onehot = OneHotEncoder(inputCols=['type_idx'], outputCols=['type_dummy'])
onehot = onehot.fit(cars)
onehot.categorySizes # [6]
cars = onehot.transform(cars)
cars.select('type', 'type_idx', 'type_dummy').distinct().sort('type_idx').show()
+-------+--------+-------------+
|   type|type_idx|   type_dummy|
+-------+--------+-------------+
|Midsize|     0.0|(5,[0],[1.0])| # разреженный формат и всего 5 категорий
|  Small|     1.0|(5,[1],[1.0])| # последняя не входит

from pyspark.mllib.linalg import DenseVector, SparseVector
DenseVector([1, 0, 0, 0, 0, 7, 0, 0]) # плотное представление
# DenseVector([1.0, 0.0, 0.0, 0.0, 0.0, 7.0, 0.0, 0.0])
SparseVector(8, [0, 5], [1, 7]) # кол-во, позиция, и значения
# SparseVector(8, {0: 1.0, 5: 7.0})


# Bucketing
#Результирующая категориальная переменная часто является более мощным предиктором, чем исходная непрерывная переменная
from pyspark.ml.feature import Bucketizer
bucketizer = Bucketizer(splits=[3500, 4500, 6000, 6500], inputCol="rpm", outputCol="rpm_bin")
bucketed = bucketizer.transform(cars)
bucketed.select('rpm', 'rpm_bin').show(5)
bucketed.groupBy('rpm_bin').count().show()


assembler = VectorAssembler(inputCols=['mass','cyl','type_dummy','density_line','density_quad','density_cube'], outputCol='features')
cars = assembler.transform(cars)
+--------------------------------------------------------------------------------+-----------+
|features                                                                        |consumption|
+--------------------------------------------------------------------------------+-----------+
|[1451.0,6.0,1.0,0.0,0.0,0.0,0.0,303.8743455497,63.63860639785,13.32745683724]| 9.05 |

regression = LinearRegression(labelCol='consumption').fit(cars_train)
regression.coefficients # DenseVector([-0.012, 0.174,-0.897,-1.445,-0.985,-1.071,-1.335, 0.189,-0.780, 1.160])
# когда данных мало а признаков много
# alpha = 0 | lambda = 0.1 -> Ridge
ridge = LinearRegression(labelCol='consumption', elasticNetParam=0, regParam=0.1)
```

```
ridge.fit(cars_train)

# alpha = 1 | lambda = 0.1 -> Lasso
lasso = LinearRegression(labelCol='consumption', elasticNetParam=1, regParam=0.1)
lasso.fit(cars_train)
```

Ensembles & Pipelines

```
indexer = StringIndexer(inputCol='type', outputCol='type_idx')
onehot = OneHotEncoder(inputCols=['type_idx'], outputCols=['type_dummy'])
assemble = VectorAssembler(inputCols=['mass', 'cyl', 'type_dummy'], outputCol='features')
regression = LinearRegression(labelCol='consumption')


from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[indexer, onehot, assemble, regression])
pipeline = pipeline.fit(cars_train)
predictions = pipeline.transform(cars_test)
# Конвейерный метод transform() будет вызывать метод transform() только для каждого из этапов конвейера
pipeline.stages[3]
print(pipeline.stages[3].intercept) # 4.1943
print(pipeline.stages[3].coefficients) #DenseVector([0.0028,0.2705,-1.1813,-1.3696,-1.1751,-1.1553,-1.8894])


# для кросс-валидации  (?) тут круто данные перетасовать
regression = LinearRegression(labelCol='consumption')
evaluator = RegressionEvaluator(labelCol='consumption')
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
params = ParamGridBuilder().build()
cv = CrossValidator(estimator=regression, estimatorParamMaps=params,
                    evaluator=evaluator, numFolds=10, seed=13)
# нужна модель, оценщик качества, сетка (пустая), объект CrossValidatior
cv = cv.fit(cars_train)
cv.avgMetrics # [0.800663722151572] RMSE
evaluator.evaluate(cv.transform(cars_test)) # на тесте
# обучение (бьем на фолды) - тест (один раз)


# Grid-Search
# сравниваем модель с константой и без
regression = LinearRegression(labelCol='consumption', fitIntercept=True)
regression = regression.fit(cars_train)
evaluator.evaluate(regression.transform(cars_test))

regression = LinearRegression(labelCol='consumption', fitIntercept=False)
regression = regression.fit(cars_train) # без константы


from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder()
params = params.addGrid(regression.fitIntercept, [True, False])
params = params.build()
print('Number of models to be tested: ', len(params)) # кол-во параметров

cv = CrossValidator(estimator=regression, estimatorParamMaps=params, evaluator=evaluator)
cv = cv.setNumFolds(10).setSeed(13).fit(cars_train) # утсановка параметров
cv.avgMetrics # [0.800663722151, 0.907977823182]
cv.bestModel # лучшая модель
predictions = cv.transform(cars_test) # ведет себя как лучшая модель
cv.bestModel.explainParam('fitIntercept')


#сложная сетка
params = ParamGridBuilder().addGrid(regression.fitIntercept, [True, False]) \
                           .addGrid(regression.regParam, [0.001, 0.01, 0.1, 1, 10]) \
                           .addGrid(regression.elasticNetParam, [0, 0.25, 0.5, 0.75, 1]).build()
print ('Number of models to be tested: ', len(params)) # 50
```

Ансамбль

```
# случайный лес
from pyspark.ml.classification import RandomForestClassifier
forest = RandomForestClassifier(numTrees=5)
forest = forest.fit(cars_train)
forest.trees # отдельные деревья
# после трансформ получается столбец вероятности и лейблов
+-----+----------------------+
|label|probability |prediction|
+-----+----------------------+
| 0.0 |  [0.8,0.2] |      0.0 |
```

```
forest.featureImportances # важность признаков
# SparseVector(6, {0: 0.0205, 1: 0.2701, 2: 0.108, 3: 0.1895, 4: 0.2939, 5: 0.1181})
```