Optimus Documentation

Release 2.1.3

Favio Vazquez

Nov 16, 2018

Contents

OPTIMUS

As data scientists, we care about extracting the best information out of our data. Data is the new soil, you have to get in and get your hands dirty, without cleaning and preparing it, it just useless.

Data preparation accounts for about 80% of the work of data scientists, so having a solution that connects to your database or file system, uses the most important framework for machine learning and data science at the moment (Apache Spark) and that can handle lots of information, working both in a cluster in a parallelized fashion or locally on your laptop is really important to have.

Say Hi! to Optimus and visit our web page.

Prepare, process and explore your Big Data with fastest open source library on the planet using Apache Spark and Python (PySpark).

Overview

1.1 Description

Optimus (By Iron)is the missing framework for cleaning, pre-processing and exploring data in a distributed fashion. It uses all the power of Apache Spark (optimized via Catalyst) to do so. It implements several handy tools for data wrangling and munging that will make your life much easier. The first obvious advantage over any other public data cleaning library or framework is that it will work on your laptop or your big cluster, and second, it is amazingly easy to install, use and understand.

Installation

In your terminal just type:

```
pip install optimuspyspark
```

2.1 Requirements

- Apache Spark >= 2.3.1 (installed with the package)
- Python >= 3.6

Column Operations

Here you will see a detailed overview of all the column operations available in Optimus. You can access the operation via `df.cols`

Let's create a sample dataframe to start working.

```python
# Import Optimus
from optimus import Optimus
# Create Optimus instance
op = Optimus()

from pyspark.sql.types import StructType, StructField, StringType, BooleanType,
↪IntegerType, ArrayType

df = op.create.df(
            [
                ("words", "str", True),
                ("num", "int", True),
                ("animals", "str", True),
                ("thing", StringType(), True),
                ("two strings", StringType(), True),
                ("filter", StringType(), True),
                ("num 2", "string", True),
                ("col_array",  ArrayType(StringType()), True),
                ("col_int",  ArrayType(IntegerType()), True)


            ]
,
[
                ("  I like    fish  ", 1, "dog", "housé", "cat-car", "a","1",["baby",
↪ "sorry"],[1,2,3]),
                ("    zombies", 2, "cat", "tv", "dog-tv", "b","2",["baby 1", "sorry 1
↪"],[3,4]),
                ("simpsons   cat lady", 2, "frog", "table","eagle-tv-plus","1","3", [
↪"baby 2", "sorry 2"], [5,6,7]),
                (None, 3, "eagle", "glass", "lion-pc", "c","4", ["baby 3", "sorry 3"]
↪,[7,8])
```

```
        ])
```

To see the dataframe we will use the `table()` function, a much better way to see your results, instead of the built-in `` `show() `` function.

```
df.table()
```

words	num	animals	thing	two strings	filter	num 2	col_array	col_int
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]

3.1 cols.append(col_name=None, value=None)

Appends a column to a Dataframe

```
df = df.cols.append("new_col_1", 1)
df.table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

```
from pyspark.sql.functions import *

df.cols.append([
    ("new_col_2", 2.22),
    ("new_col_3", lit(3))
    ]).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1	new_col_2	new_col_3
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1	2.22	3
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1	2.22	3
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1	2.22	3
null	3	ea-gle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1	2.22	3

```
df.cols.append([
("new_col_4", "test"),
("new_col_5", df['num']*2),
("new_col_6", [1,2,3])
]).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1	new_col_4	new_col_5	new_col_6
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1	test	2	[1, 2, 3]
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1	test	4	[1, 2, 3]
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1	test	4	[1, 2, 3]
null	3	ea-gle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1	test	6	[1, 2, 3]

3.2 cols.select(columns=None, regex=None, data_type=None)

Select columns using index, column name, regex or data type

```
columns = ["words", 1, "animals", 3]
df.cols.select(columns).table()
```

words	num	animals	thing
I like fish	1	dog	housé
zombies	2	cat	tv
simpsons cat lady	2	frog	table
null	3	eagle	glass

```
df.cols.select("n.*", regex = True).show()
```

num	num 2	new_col_1
1	1	1
2	2	1
2	3	1
3	4	1

```
df.cols.select("*", data_type = "str").table()
```

thing	words	animals	filter	two strings	num 2
housé	I like fish	dog	a	cat-car	1
tv	zombies	cat	b	dog-tv	2
table	simpsons cat lady	frog	1	eagle-tv-plus	3
glass	null	eagle	c	lion-pc	4

3.3 cols.rename(columns_old_new=None, func=None)

Changes the name of a column(s) dataFrame.

```
df.cols.rename('num','number').table()
```

words	num-ber	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

```
df.cols.rename([('num','number'),("animals","gods")], str.upper).table()
```

WORDS	NUM	ANI-MALS	THING	TWO STRINGS	FIL-TER	NUM 2	COL_ARRAY	COL_INT	NEW_COL_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

```
df.cols.rename(str.lower).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

3.4 cols.cast()

Cast multiple columns to a specific datatype.

List of tuples of column names and types to be casted. This variable should have the following structure:

colsAndTypes = [('columnName1', 'integer'), ('columnName2', 'float'), ('columnName3', 'string')]

The first parameter in each tuple is the column name, the second is the final datatype of column after the transformation is made.

```
df.cols.cast([("num", "string"),("num 2", "integer")]).dtypes

 [('words', 'string'),
 ('num', 'string'),
 ('animals', 'string'),
 ('thing', 'string'),
 ('two strings', 'string'),
 ('filter', 'string'),
 ('num 2', 'int'),
 ('col_array', 'array<string>'),
 ('col_int', 'array<int>'),
 ('new_col_1', 'int')]
```

You can cast all columns to a specific type too.

```
df.cols.cast("*", "string").dtypes

[('words', 'string'),
 ('num', 'string'),
 ('animals', 'string'),
 ('thing', 'string'),
 ('two strings', 'string'),
 ('filter', 'string'),
 ('num 2', 'string'),
 ('col_array', 'string'),
 ('col_int', 'string'),
 ('new_col_1', 'string')]
```

3.5 cols.keep(columns=None, regex=None)

Only keep the columns specified.

```
df.cols.keep("num").table()
```

num
1
2
2
3

3.6 cols.move(column, position, ref_col)

Move a column to specific position

```
df.cols.move("words", "after", "thing").table()
```

num	animals	thing	words	two strings	filter	num 2	col_array	col_int	new_col_1
1	dog	housé	I like fish	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
2	cat	tv	zombies	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
2	frog	table	simpsons cat lady	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
3	eagle	glass	null	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

3.7 cols.sort(order="asc")

Sort dataframes columns asc or desc

```
df.cols.sort().table()
```

animals	col_array	col_int	filter	new_col_1	num	num 2	thing	two strings	words
dog	[baby, sorry]	[1, 2, 3]	a	1	1	1	housé	cat-car	I like fish
cat	[baby 1, sorry 1]	[3, 4]	b	1	2	2	tv	dog-tv	zombies
frog	[baby 2, sorry 2]	[5, 6, 7]	1	1	2	3	table	eagle-tv-plus	simpsons cat lady
eagle	[baby 3, sorry 3]	[7, 8]	c	1	3	4	glass	lion-pc	null

```
df.cols.sort(order = "desc").table()
```

words	two strings	thing	num 2	num	new_col_1	filter	col_int	col_array	animals
I like fish	cat-car	housé	1	1	1	a	[1, 2, 3]	[baby, sorry]	dog
zombies	dog-tv	tv	2	2	1	b	[3, 4]	[baby 1, sorry 1]	cat
simpsons cat lady	eagle-tv-plus	table	3	2	1	1	[5, 6, 7]	[baby 2, sorry 2]	frog
null	lion-pc	glass	4	3	1	c	[7, 8]	[baby 3, sorry 3]	eagle

3.8 cols.drop()

Drops a list of columns

```
df2 = df.cols.drop("num")
df2.table()
```

words	animals	thing	two strings	filter	num 2	col_array	col_int	new_col_1
I like fish	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

```
df2 = df.cols.drop(["num","words"])
df2.table()
```

animals	thing	two strings	filter	num 2	col_array	col_int	new_col_1
dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

3.9 Chaining

The past transformations were done step by step, but this can be achieved by chaining all operations into one line of code, like the cell below. This way is much more efficient and scalable because it uses all optimization issues from the lazy evaluation approach.

```
df\
.cols.rename([('num','number')])\
.cols.drop(["number","words"])\
.withColumn("new_col_2", lit("spongebob"))\
.cols.append("new_col_1", 1)\
.cols.sort(order= "desc")\
.rows.drop(df["num 2"] == 3)\
.table()
```

two strings	thing	num 2	new_col_2	new_col_1	filter	col_int	col_array	animals
cat-car	housé	1	spongebob	1	a	[1, 2, 3]	[baby, sorry]	dog
dog-tv	tv	2	spongebob	1	b	[3, 4]	[baby 1, sorry 1]	cat
lion-pc	glass	4	spongebob	1	c	[7, 8]	[baby 3, sorry 3]	eagle

3.10 cols.unnest(columns, mark=None, n=None, index=None)

Split array or string in different columns

```
df.cols.unnest("two strings","-").table()
```

words	num	animals	thing	two strings	filter	num 2	col_array	col_int	two strings_0	two strings_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	cat	car
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	dog	tv
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	eagle	tv
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	lion	pc

Only getting the first element

```
df.cols.unnest("two strings","-", index = 1).table()
```

words	num	animals	thing	two strings	filter	num 2	col_array	col_int	two strings_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	car
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	tv
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	tv
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	pc

Unnest array of string

```
df.cols.unnest(["col_array"]).table()
```

words	num	animals	thing	two strings	filter	num 2	col_array	col_int	col_array_0	col_array_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	baby	sorry
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	baby 1	sorry 1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	baby 2	sorry 2
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	baby 3	sorry 3

Split in 3 parts

```
df \
.cols.unnest(["two strings"], n= 3, mark = "-") \
.table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	two strings_0	two strings_1	two strings_2
I like fish	1	dog	house	cat-car	a	1	[baby, sorry]	[1, 2, 3]	cat	car	null
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	dog	tv	null
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	eagle	tv	plus
null	3	ea-gle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	lion	pc	null

3.11 cols.impute(input_cols, output_cols, strategy="mean")

Imputes missing data from specified columns using the mean or median.

```
# Create test dataset
df_fill = op.spark.createDataFrame([(1.0, float("nan")), (2.0, float("nan")),
                          (float("nan"), 3.0), (4.0, 4.0), (5.0, 5.0)], ["a", "b"])

df_fill.cols.impute(["a", "b"], ["out_a", "out_b"], "median").table()
```

a	b	out_a	out_b
1.0	NaN	1.0	4.0
2.0	NaN	2.0	4.0
NaN	3.0	2.0	3.0
4.0	4.0	4.0	4.0
5.0	5.0	5.0	5.0

3.12 cols.select_by_dtypes(data_type)

Returns one or multiple dataframe columns which match with the data type provided.

```
df.cols.select_by_dtypes("int").table()
```

num
1
2
2
3

3.13 cols.apply_by_dtypes(columns, func, func_return_type, args=None, func_type=None, data_type=None)

Apply a function using pandas udf or udf if apache arrow is not available.

In the next example we replace a number in a string column with "new string":

```
def func(val, attr):
    return attr

df.cols.apply_by_dtypes("filter", func, "string", "new string", data_type="integer").
↪table()
```

words	num	ani-mals	thing	two strings	filter	num 2	col_array	col_int
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]
simpsons cat lady	2	frog	table	eagle-tv-plus	new string	3	[baby 2, sorry 2]	[5, 6, 7]
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]

3.14 User Define Functions in Optimus

Now we'll create a UDF function that sum a values (32 in this case) to two columns

```
df = df.cols.append("new_col_1", 1)

def func(val, attr):
    return val + attr

df.cols.apply(["num", "new_col_1"], func, "int", 32 ,"udf").table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	33	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	33
zombies	34	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	33
simpsons cat lady	34	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	33
null	35	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	33

Now a we'll create a Pandas UDF function that sum a values (10 in this case) to two columns

```
def func(val, attr):
    return val + attr

df.cols.apply(["num", "new_col_1"], func, "int", 10).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	11	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	11
zombies	12	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	11
simpsons cat lady	12	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	11
null	13	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	11

Create an abstract udf to filter a rows where the value of column "num"> 1

```
from optimus.functions import abstract_udf as audf

def func(val, attr):
    return val>1

df.rows.select(audf("num", func, "boolean")).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

Create an abstract udf (Pandas UDF) to pass two arguments to a function a apply a sum operation

```
from optimus.functions import abstract_udf as audf

def func(val, attr):
    return val+attr[0]+ attr[1]

df.withColumn("num_sum", audf ("num", func, "int", [10,20])).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1	num_sum
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1	31
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1	32
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1	32
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1	33

3.15 cols.apply_expr(columns, func=None, args=None, filter_col_by_dtypes=None, verbose=True)

Apply a expression to column.

Here we'll apply a column expression to when the value of "num" or "num 2" is grater than 2:

```python
from pyspark.sql import functions as F
def func(col_name, attr):
    return F.when(F.col(col_name)>2 ,10).otherwise(1)

df.cols.apply_expr(["num","num 2"], func).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	1	cat	tv	dog-tv	b	1	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	1	frog	table	eagle-tv-plus	1	10	[baby 2, sorry 2]	[5, 6, 7]	1
null	10	eagle	glass	lion-pc	c	10	[baby 3, sorry 3]	[7, 8]	1

Convert to uppercase:

```python
from pyspark.sql import functions as F
def func(col_name, attr):
    return F.upper(F.col(col_name))

df.cols.apply_expr(["two strings","animals"], func).table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	DOG	housé	CAT-CAR	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	CAT	tv	DOG-TV	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	FROG	table	EAGLE-TV-PLUS	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	EA-GLE	glass	LION-PC	c	4	[baby 3, sorry 3]	[7, 8]	1

3.16 cols.count_na(columns)

Returns the NAN and Null count in a Column.

```python
import numpy as np

df_null = op.spark.createDataFrame(
```

```
    [(1, 1, None), (1, 2, float(5)), (1, 3, np.nan), (1, 4, None), (1, 5, float(10)),
→(1, 6, float('nan')), (1, 6, float('nan'))],
    ('session', "timestamp1", "id2"))

df_null.cols.count_na("*")

Out -> {'session': 0, 'timestamp1': 0, 'id2': 5}
```

3.17 cols.count_uniques(columns, estimate=True)

Returns how many unique items exist in a columns

```
df.cols.count_uniques("*")
```

And you'll get:

```
{'words': {'approx_count_distinct': 3},
 'num': {'approx_count_distinct': 3},
 'animals': {'approx_count_distinct': 4},
 'thing': {'approx_count_distinct': 4},
 'two strings': {'approx_count_distinct': 4},
 'filter': {'approx_count_distinct': 4},
 'num 2': {'approx_count_distinct': 4},
 'col_array': {'approx_count_distinct': 3},
 'col_int': {'approx_count_distinct': 4},
 'new_col_1': {'approx_count_distinct': 1}}
```

3.18 cols.replace(columns, search_and_replace=None, value=None, regex=None)

Replace a value or a list of values by a specified string

```
df.cols.replace("animals",["dog","cat"],"animals").table()
```

Replace "dog","cat" in column "animals" by the word "animals":

words	num	animals	thing	two strings	filter	num 2	col_array	col_int	new_col_1
I like fish	1	animals	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	animals	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

Replace "dog-tv", "cat", "eagle", "fish" in columns "two strings","animals" by "animals":

```
df.cols.replace(["two strings","animals"], ["dog-tv", "cat", "eagle", "fish"],
↪"animals").table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1
zombies	2	ani-mals	tv	animals	b	2	[baby 1, sorry 1]	[3, 4]	1
simpsons cat lady	2	frog	table	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1
null	3	ani-mals	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1

3.19 cols.nest(input_cols, output_col, shape=None, separator=" ")

Concat multiple columns to one with the format specified

```
df.cols.nest(["num", "new_col_1"], output_col = "col_nested", shape ="vector").table()
```

Merge two columns in a column vector:

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1	col_nested
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1	[1.0,1.0]
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1	[2.0,1.0]
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1	[2.0,1.0]
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1	[3.0,1.0]

Merge two columns in a string columns:

```
df.cols.nest(["animals", "two strings"], output_col= "col_nested", shape = "string").
↪table()
```

words	num	ani-mals	thing	two strings	fil-ter	num 2	col_array	col_int	new_col_1	col_nested
I like fish	1	dog	housé	cat-car	a	1	[baby, sorry]	[1, 2, 3]	1	dog cat-car
zombies	2	cat	tv	dog-tv	b	2	[baby 1, sorry 1]	[3, 4]	1	cat dog-tv
simpsons cat lady	2	frog	ta-ble	eagle-tv-plus	1	3	[baby 2, sorry 2]	[5, 6, 7]	1	frog eagle-tv-plus
null	3	eagle	glass	lion-pc	c	4	[baby 3, sorry 3]	[7, 8]	1	eagle lion-pc

Row Operations

Here you will see a detailed overview of all the row operations available in Optimus. You can access the operations via `df.rows`

Let's create a sample dataframe to start working.

words	num	animals	thing	second	filter
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b
simpsons cat lady	2	frog	table	7	1
null	3	eagle	glass	8	c

4.1 rows.append(row)

Append a row at the end of a dataframe

```
df.rows.append(["this is a word",2, "this is an animal", "this is a thing", 64, "this
→is a filter"]).table()
```

words	num	animals	thing	second	filter
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b
simpsons cat lady	2	frog	table	7	1
null	3	eagle	glass	8	c
this is a word	2	this is an animal	this is a thing	64	this is a filter

4.2 rows.sort()

Sort the columns by rows or multiple conditions.

```
df.rows.sort("animals").table()
```

words	num	animals	thing	second	filter
simpsons cat lady	2	frog	table	7	1
null	3	eagle	glass	8	c
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b

```
df.rows.sort("animals", "desc").table()
```

words	num	animals	thing	second	filter
simpsons cat lady	2	frog	table	7	1
null	3	eagle	glass	8	c
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b

```
df.rows.sort([("animals","desc"),("thing","asc")]).table()
```

words	num	animals	thing	second	filter
simpsons cat lady	2	frog	table	7	1
null	3	eagle	glass	8	c
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b

4.3 rows.select(*args, **kwargs)

Alias of Spark filter function. Return rows that match a expression.

```
df.rows.select(df["num"]==1).table()
```

words	num	animals	thing	second	filter
I like fish	1	dog dog	housé	5	a

4.4 rows.select_by_dtypes(col_name, data_type=None)

This function has built in order to filter some type of row depending of the var type detected by python

words	num	animals	thing	second	filter
simpsons cat lady	2	frog	table	7	1

4.5 rows.drop(where=None)

Drop a row depending on a dataframe expression

```
df.rows.drop((df["num"]==2) | (df["second"]==5)).table()
```

words	num	animals	thing	second	filter
null	3	eagle	glass	8	c

4.6 rows.drop_by_dtypes(col_name, data_type=None)

Drop rows by cell data type

```
df.rows.drop_by_dtypes("filter", "int").table()
```

words	num	animals	thing	second	filter
I like fish	1	dog dog	housé	5	a
zombies	2	cat	tv	6	b
null	3	eagle	glass	8	c

4.7 Drop using an abstract UDF

```
from optimus.functions import abstract_udf as audf

def func_data_type(value, attr):
    return value >1


df.rows.drop(audf("num", func_data_type, "boolean")).table()
```

words	num	animals	thing	second	filter
I like fish	1	dog dog	housé	5	a

Feature Engineering with Optimus

Now with Optimus we have made easy the process of Feature Engineering.

When we talk about Feature Engineering we refer to creating new features from your existing ones to improve model performance. Sometimes this is the case, or sometimes you need to do it because a certain model doesn't recognize the data as you have it, so these transformations let you run most of Machine and Deep Learning algorithms.

These methods are part of the DataFrameTransformer, and they are a high level of abstraction for Spark Feature Engineering methods. You'll see how easy it is to prepare your data with Optimus for Machine Learning.

5.1 Methods for Feature Engineering

5.1.1 fe.string_to_index(input_cols)

This method maps a string column of labels to an ML column of label indices. If the input column is numeric, we cast it to string and index the string values.

df Data frame to transform `input_cols` argument receives a list of columns to be indexed.

Let's start by creating a DataFrame with Optimus.

```python
from pyspark.sql import Row, types
from pyspark.ml import feature, classification

from optimus import Optimus

from optimus.ml.models import ML
import optimus.ml.feature as fe

op = Optimus()
ml = ML()
spark = op.spark
sc = op.sc
```

```
# Creating sample DF
data = [('Japan', 'Tokyo', 37800000),('USA', 'New York', 19795791),('France', 'Paris',
↪ 12341418),
        ('Spain','Madrid',6489162)]
df = op.spark.createDataFrame(data, ["country", "city", "population"])

df.table()
```

country	city	population
Japan	Tokyo	37800000
USA	New York	19795791
France	Paris	12341418
Spain	Madrid	6489162

```
# Indexing columns 'city" and 'country'
df_sti = fe.string_to_index(df, input_cols=["city", "country"])

# Show indexed DF
df_sti.table()
```

country	city	population	city_index	country_index
Japan	Tokyo	37800000	1.0	1.0
USA	New York	19795791	2.0	3.0
France	Paris	12341418	3.0	2.0
Spain	Madrid	6489162	0.0	0.0

5.1.2 fe.index_to_string(input_cols)

This method maps a column of indices back to a new column of corresponding string values. The index-string mapping is either from the ML (Spark) attributes of the input column, or from user-supplied labels (which take precedence over ML attributes).

df Data frame to transform `input_cols` argument receives a list of columns to be indexed.

Let's go back to strings with the DataFrame we created in the last step.

```
# Indexing columns 'city" and 'country'
df_sti = fe.string_to_index(df, input_cols=["city", "country"])

# Show indexed DF
df_sti.table()
```

country	city	population	city_index	country_index
Japan	Tokyo	37800000	1.0	1.0
USA	New York	19795791	2.0	3.0
France	Paris	12341418	3.0	2.0
Spain	Madrid	6489162	0.0	0.0

```
# Going back to strings from index
df_its = fe.string_to_index(df_sti, input_cols=["country_index"])
```

```
# Show DF with column "county_index" back to string
df_its.table()
```

country	city	population	country_index	city_index	country_index_string
Japan	Tokyo	37800000	1.0	1.0	Japan
USA	New York	19795791	3.0	2.0	USA
France	Paris	12341418	2.0	3.0	France
Spain	Madrid	6489162	0.0	0.0	Spain

5.1.3 fe.one_hot_encoder(input_cols)

This method maps a column of label indices to a column of binary vectors, with at most a single one-value.

df Data frame to transform `input_cols` argument receives a list of columns to be encoded.

Let's create a sample dataframe to see what OHE does:

```
# Creating DataFrame
data = [
(0, "a"),
(1, "b"),
(2, "c"),
(3, "a"),
(4, "a"),
(5, "c")
]
df = op.spark.createDataFrame(data,["id", "category"])

# One Hot Encoding
df_ohe = fe.one_hot_encoder(df, input_cols=["id"])

# Show encoded dataframe
df_ohe.table()
```

id	category	id_encoded
0	a	(5,[0],[1.0])
1	b	(5,[1],[1.0])
2	c	(5,[2],[1.0])
3	a	(5,[3],[1.0])
4	a	(5,[4],[1.0])
5	c	(5,[],[])

5.1.4 Transformer.vector_assembler(input_cols)

This method combines a given list of columns into a single vector column.

`input_cols` argument receives a list of columns to be encoded.

This is very important because lots of Machine Learning algorithms in Spark need this format to work.

Let's create a sample dataframe to see what vector assembler does:

```python
# Import Vectors
from pyspark.ml.linalg import Vectors

# Creating DataFrame
data = [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)]

df = op.spark.createDataFrame(data,["id", "hour", "mobile", "user_features", "clicked
↪"])

# Assemble features
df_va = fe.vector_assembler(df, input_cols=["hour", "mobile", "user_features"])

# Show assembled df
print("Assembled columns 'hour', 'mobile', 'user_features' to vector column 'features'
↪")
df_va.select("features", "clicked").table()
```

features	clicked
[18.0,1.0,0.0,10.0,0.5]	1.0

5.1.5 fe.normalizer(input_cols,p=2.0)

This method transforms a dataset of Vector rows, normalizing each Vector to have unit norm. It takes parameter p, which specifies the p-norm used for normalization. (p=2) by default.

`input_cols` argument receives a list of columns to be normalized.

`p` argument is the p-norm used for normalization.

Let's create a sample dataframe to see what normalizer does:

id	features	features_normalized
0	[1.0,0.5,-1.0]	[0.6666666666666666,0.3333333333333333,-0.6666666666666666]
1	[2.0,1.0,1.0]	[0.8164965809277261,0.4082482904638631,0.4082482904638631]
2	[4.0,10.0,2.0]	[0.3651483716701107,0.9128709291752769,0.18257418583505536]

Machine Learning with Optimus

Machine Learning is one of the last steps, and the goal for most Data Science WorkFlows.

Apache Spark created a library called MLlib where they coded great algorithms for Machine Learning. Now with the ML library we can take advantage of the Dataframe API and its optimization to create easily Machine Learning Pipelines.

Even though this task is not extremely hard, is not easy. The way most Machine Learning models work on Spark are not straightforward, and they need lots feature engineering to work. That's why we created the feature engineering section inside the Transformer.

To import the Machine Learning Library you just need to say to import Optimus and the ML API:

```python
from pyspark.sql import Row, types
from pyspark.ml import feature, classification
from optimus import Optimus
from optimus.ml.models import ML
from optimus.ml.functions import *

op = Optimus()
ml = ML()
spark = op.spark
sc = op.sc
```

Let's take a look of what Optimus can do for you:

6.1 Methods for Machine Learning

6.1.1 ml.logistic_regression_text(df, input_col)

This method runs a logistic regression for input (text) DataFrame.

Let's create a sample dataframe to see how it works.

```
# Import Row from pyspark
from pyspark.sql import Row
# Importing Optimus
import optimus as op

df = op.sc. \
    parallelize([Row(sentence='this is a test', label=0.),
                 Row(sentence='this is another test', label=1.)]). \
    toDF()

df.table()
```

label	sentence
0.0	this is a test
1.0	this is another test

```
df_predict, ml_model = ml.logistic_regression_text(df, "sentence")
```

This instruction will return two things, first the DataFrame with predictions and steps to build it with a pipeline and a Spark machine learning model where the third step will be the logistic regression.

The columns of df_predict are:

```
df_predict.columns

['label',
 'sentence',
 'Tokenizer_4df79504b43d7aca6c0b__output',
 'CountVectorizer_421c9454cfd127d9deff__output',
 'LogisticRegression_406a8cef8029cfbbfeda__rawPrediction',
 'LogisticRegression_406a8cef8029cfbbfeda__probability',
 'LogisticRegression_406a8cef8029cfbbfeda__prediction']
```

The names are long because those are the uid for each step in the pipeline. So lets see the prediction compared with the actual labels:

```
df_predict.cols.select([[0,6]]).table()
```

label	LogisticRegression_406a8cef8029cfbbfeda__prediction
0.0	0.0
1.0	1.0

So we just did ML with a single line in Optimus. The model is also exposed in the *ml_model* variable so you can save it and evaluate it.

6.1.2 ml.n_gram(df, input_col, n=2)

This method converts the input array of strings inside of a Spark DF into an array of n-grams. The default n is 2 so it will produce bi-grams.

Let's create a sample dataframe to see how it works.

```
df = op.sc. \
    parallelize([['this is the best sentence ever'],
                 ['this is however the worst sentence available']]). \
    toDF(schema=types.StructType().add('sentence', types.StringType()))

df_model, tfidf_model = n_gram(df, input_col="sentence", n=2)
```

The columns of df_predict are:

```
['sentence',
 'Tokenizer_4a0eb7921c3a33b0bec5__output',
 'StopWordsRemover_4c5b9a5473e194516f3f__output',
 'CountVectorizer_41638674bb4c4a8d454c__output',
 'NGram_4e1d89fc70917c522134__output',
 'CountVectorizer_4513a7ba6ce22e617be7__output',
 'VectorAssembler_42719455dc1bde0c2a24__output',
 'features']
```

So lets see the bi-grams (we can change n as we want) for the sentences:

```
df_model.cols.select([[0,4]).table()
```

sentence	NGram_4e1d89fc70917c522134__output
this is the best sentence ever	[best sentence, sentence ever]
this is however the worst sentence available	[however worst, worst sentence, sentence available]

And that's it. N-grams with only one line of code.

Above we've been using the Pyspark Pipes definitions of Daniel Acuña, that he merged with Optimus, and because we use multiple pipelines we need those big names for the resulting columns, so we can know which uid correspond to each step.

Tree models with Optimus

Yes the rumor is true, now you can build Decision Trees, Random Forest models and also Gradient Boosted Trees with just one line of code in Optimus. Let's download some sample data for analysis.

We got this dataset from Kaggle. The features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. n the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].

Let's download it with Optimus and save it into a DF:

```
# Downloading and creating Spark DF
df = op.load.url("https://raw.githubusercontent.com/ironmussa/Optimus/master/tests/
↪data_cancer.csv")
```

6.1.3 ml.random_forest(df, columns, input_col)

One of the best "tree" models for machine learning is Random Forest. What about creating a RF model with just one line? With Optimus is really easy.

Let's download some sample data for analysis.

```
df_predict, rf_model = ml.random_forest(df_cancer, columns, "diagnosis")
```

This will create a DataFrame with the predictions of the Random Forest model.

Let's see df_predict:

```
['label',
 'diagnosis',
 'radius_mean',
 'texture_mean',
 'perimeter_mean',
 'area_mean',
 'smoothness_mean',
 'compactness_mean',
 'concavity_mean',
 'concave points_mean',
 'symmetry_mean',
 'fractal_dimension_mean',
 'features',
 'rawPrediction',
 'probability',
 'prediction']
```

So lets see the prediction compared with the actual label:

```
df_predict.select([0,15]).table()
```

label	prediction
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	0.0
1.0	1.0
1.0	1.0
0.0	0.0

only showing top 20 rows

The rf_model variable contains the Random Forest model for analysis.

It will be the same for Decision Trees and Gradient Boosted Trees, let's check it out.

6.1.4 ml.decision_tree(df, columns, input_col)

```
df_predict, dt_model = ml.random_forest(df_cancer, columns, "diagnosis")
```

This will create a DataFrame with the predictions of the Decision Tree model.

Let's see df_predict:

```
['label',
 'diagnosis',
 'radius_mean',
 'texture_mean',
 'perimeter_mean',
 'area_mean',
 'smoothness_mean',
 'compactness_mean',
 'concavity_mean',
 'concave points_mean',
 'symmetry_mean',
 'fractal_dimension_mean',
 'features',
 'rawPrediction',
 'probability',
 'prediction']
```

So lets see the prediction compared with the actual label:

```
df_predict.select([0,15]).table()
```

label	prediction
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
0.0	0.0

only showing top 20 rows

6.1.5 ml.gbt(df, columns, input_col)

```
df_predict, gbt_model = ml.gbt(df_cancer, columns, "diagnosis")
```

This will create a DataFrame with the predictions of the Gradient Boosted Trees model.

Let's see df_predict:

```
['label',
 'diagnosis',
 'radius_mean',
 'texture_mean',
 'perimeter_mean',
 'area_mean',
 'smoothness_mean',
 'compactness_mean',
 'concavity_mean',
 'concave points_mean',
 'symmetry_mean',
 'fractal_dimension_mean',
 'features',
 'rawPrediction',
 'probability',
 'prediction']
```

So lets see the prediction compared with the actual label:

```
df_predict.select([0,15]).show()
```

label	prediction
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
1.0	1.0
0.0	0.0

only showing top 20 rows

CHAPTER 7

Library maintained by Favio Vazquez and Argenis Leon