Data Science Project

Real – Time Customer Churn Prediction in Telecom Companies

Fellipe Augusto Soares Silva

2020

Chapters

Figures

Tabelas

Summary

This project was developed with the purpose of predicting in real time the turnover of customers of a Telecom Company providing metrics for the company to act more quickly in preventing losses and also to retain satisfied consumer.

Using Pyhton programming language together with Apache Spark (cluster computing platform) large amounts of data were collected in order to implement a Supervised Machine Learning Predictive Model in historical data.

For this project, the dataset provided by the Telecom Company to Kaggle, an online community of data and machine learning scientists owned by Google LLC, was used as open data for the community.

With the data provided I developed: data loading, Feature engineering, cleaning and transformation, processing in memory ensuring speed, exploratory analyzes evidencing opportunities for the company, Machine Learning and a series of other practices in Spark.

After exploring all the information provided by the data, two predictive classification models were implemented in test data to train and analyze the model's accuracy. When selecting the best machine learning model, new and unknown data was introduced in the model to test its efficiency by presenting the results through a confusion matrix.

This project's differential will also be the presentation of Spark Jobs being executed in real time through PySparkShell, a powerful tool for interactive analysis of data behavior

.

Palavras – Chave: Data Science, Machine Learning, Python, Big Data, Exploratory Analysis, Business Intelligence, BI, Spark, Pyspark, Pyspark Shell, Spark RDD, Spark SQL, Spark Streaming, Spark MLLib, Forecasting, Accuracy, Confusion Matrix, Customer Churn, Telecom, Kaggle.

1. Introduction

This project was developed using the Spark API[1] for Python[2], the Pyspark, and Jupyter Notebook[3] as a script development and testing plataform[4], in addition to the Pyspark Shell for interactive analysis of the execution of Jobs.

Spark is a tool that belongs to the Apache Software Foundation, has a large capacity for processing Big Data and is one of the most implemented mechanisms by companies today. In addition to being open-source, Spark provides speed and ease in handling large amount of data through its high-level libraries.

This project used the concepts and practices of:

- Batch processing with Spark RDD[5];
- Query SQL with Spark SQL[6];
- Spark Session to use Pyspark Dataframes[7];
- Machine Learning with Spark MLLib[8].

Python is a programming language that has been gaining space in the job market due to its versatility and functionality when combined with more than 100,000 other libraries such as:

- NumPy for numerical computing;
- Pandas for data manipulation;
- SciPy for scientific computing;
- MatplotLib for visualization and plotting.

In this way, Pyspark is a Spark API for the Python programming language, allowing the combination of these two concepts previously presented, transforming the tool into an excellent option for use in distributed computing, Big Data and data streaming projects.

[1] https://spark.apache.org/docs/latest/api/python/index.html
[2] Python is an object-oriented programming language.
[3] https://jupyter.org/
[4] Script is a set of instructions in code written in computer language so that it performs different functions inside a program.
[5] https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=rdd#pyspark.RDD
[6] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=sql#module-pyspark.sql
[7] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=dataframe#pyspark.sql.SparkSession.createDataFrame
[8] https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html?highlight=mllib

Jupyter Notebook[9] , on the other hand, is an open-source project conceived in 2014 as a web computing environment for creating codes through different programming languages such as R[10], Scala[11], Python[12] among others. Jupyter is a tool that also allows you to document your work in different sources such as HTML, LaTeX[13], PNG, PDF, SVG, among others.

Finally, a tool available for analyzing the distribution and recovery of data within the cluster is Pyspark Shell[14]. With it, each job is presented in real time with its execution chain and the time it took to finish the job.

Like any Data Science project, before starting the parameterization of the predictive model it is necessary to perform exploratory analysis and to know the dataset. During this procedure, the Business Intelligence (BI) analysis presented several business opportunities that will be highlighted in the chapter in question. It was also possible to add some information based on the existing variables so that the predictive model was more accurate.

To conclude, I will present all script items, commented line by line and presenting graphical analysis to complement the understanding of the development of this Data Science project.

The beginning of business problem solving lies in understanding the problem itself, which will be presented in the following chapter.

During the presentation of this project I will expose the most important parts of the code. You will find in the last chapter the source code commented in full. I decided to do it this way so that the presentation of the project would be more dynamic and less heavy so that people from different business areas could understand the potential of data science.

[9] https://jupyter.org/
[10] https://www.r-project.org/
[11] https://www.scala-lang.org/
[12] https://www.python.org/
[13] https://www.latex-project.org/
[14] https://spark.apache.org/docs/latest/quick-start.html

2. Business Problem

Customer Turnover, or Customer Churn, is an important metric for companies to assess whether their business is flowing in a healthy way. This metric is defined by customers who stopped consuming products from that company, regardless of the reason that caused the disconnection of the customer from the company.

Performing the Customer Churn calculation is relatively simple:

$$Churn\ Rate = \frac{Total\ Disconnected\ Customers}{Total\ Customers} \times 100$$

For a company that has lost 5 of its 500 customers, the churn rate is equal to 1%. Each market segment will determine its ideal churn rate, which can vary a lot, but the essential thing is to understand why the company has customer disconnections because to understand is to generate opportunities for improvement.

The churn value indicates only a portion of the problem, it is essential to go deeper into the problem and raise important questions for the business to last and thrive in the competitive job market. Listening to the customer is necessary, understanding your target audience helps to direct more assertive projects and lead company employees according to the established goals, study the market and its evolution, study your product and how it is marketed, among many other sources. It is important not to let the churn rate increase.

For this project, we will study ways to predict Customer Churn in a telecom company through a large set of data and not just looking at the customer's decision to maintain the service or not. Data from international plans, number and duration of calls made, spent in periods of the day will be studied, offering the machine learning model information to predict whether a customer is likely to cancel their service plan.

With this information in hand, decision makers can act more quickly in the prevention processes and not in the containment process as the latter may be too late and the customer has already made his decision.

Aiming at this agility, Spark was chosen to process a large volume of information and deliver reliability to the company, employees and its stakeholders. With this project I sought to introduce intelligence to the business, generating opportunities and increasing profitability.

3. Dataset

For this business problem we have the following data set:

3.1 Training Dataset

This dataset is structured as follows:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail_messages	total_day_minutes	total_day_calls	total_day_charge	total_eve_minutes	total_eve_calls	total_eve_charge	total_night_minutes	total_night_calls	total_night_charge	total_intl_minutes	total_intl_calls	total_intl_charge	number_customer_service_calls	churn
1	KS	128	area_code_415	no	yes	25	265.1	110	45.07	197.4	99	16.78	244.7	91	11.01	10	3	2.7	1	no
2	OH	107	area_code_415	no	yes	26	161.6	123	27.47	195.5	103	16.62	254.4	103	11.45	13.7	3	3.7	1	no
3	NJ	137	area_code_415	no	no	0	243.4	114	41.38	121.2	110	10.3	162.6	104	7.32	12.2	5	3.29	0	no
4	OH	84	area_code_408	yes	no	0	299.4	71	50.9	61.9	88	5.26	196.9	89	8.86	6.6	7	1.78	2	no
5	OK	75	area_code_415	yes	no	0	166.7	113	28.34	148.3	122	12.61	186.9	121	8.41	10.1	3	2.73	3	no
6	AL	118	area_code_510	yes	no	0	223.4	98	37.98	220.6	101	18.75	203.9	118	9.18	6.3	6	1.7	0	no
7	MA	121	area_code_510	no	yes	24	218.2	88	37.09	348.5	108	29.62	212.6	118	9.57	7.5	7	2.03	3	no
8	MO	147	area_code_415	yes	no	0	157	79	26.69	103.1	94	8.76	211.8	96	9.53	7.1	6	1.92	0	no
9	LA	117	area_code_408	no	no	0	184.5	97	31.37	351.6	80	29.89	215.8	90	9.71	8.7	4	2.35	1	no
10	WV	141	area_code_415	yes	yes	37	258.6	84	43.96	222	111	18.87	326.4	97	14.69	11.2	5	3.02	0	no
11	IN	65	area_code_415	no	no	0	129.1	137	21.95	228.5	83	19.42	208.8	111	9.4	12.7	6	3.43	4	yes
12	RI	74	area_code_415	no	no	0	187.7	127	31.91	163.4	148	13.89	196	94	8.82	9.1	5	2.46	0	no
13	IA	168	area_code_408	no	no	0	128.8	96	21.9	104.9	71	8.92	141.1	128	6.35	11.2	2	3.02	1	no
14	MT	95	area_code_510	no	no	0	156.6	88	26.62	247.6	75	21.05	192.3	115	8.65	12.3	5	3.32	3	no
15	IA	62	area_code_415	no	no	0	120.7	70	20.52	307.2	76	26.11	203	99	9.14	13.1	6	3.54	4	no
16	NY	161	area_code_415	no	no	0	332.9	67	56.59	317.8	97	27.01	160.6	128	7.23	5.4	9	1.46	4	yes
17	ID	85	area_code_408	no	yes	27	196.4	139	33.39	280.9	90	23.88	89.3	75	4.02	13.8	4	3.73	1	no
18	VT	93	area_code_510	no	no	0	190.7	114	32.42	218.2	111	18.55	129.6	121	5.83	8.1	3	2.19	3	no
19	VA	76	area_code_510	no	yes	33	189.7	66	32.25	212.8	65	18.09	165.7	108	7.46	10	5	2.7	1	no
20	TX	73	area_code_415	no	no	0	224.4	90	38.15	159.5	88	13.56	192.8	74	8.68	13	2	3.51	1	no
21	FL	147	area_code_415	no	no	0	155.1	117	26.37	239.7	93	20.37	208.8	133	9.4	10.6	4	2.86	0	no
22	CO	77	area_code_408	no	no	0	62.4	89	10.61	169.9	121	14.44	209.6	64	9.43	5.7	6	1.54	5	yes
23	AZ	130	area_code_415	no	no	0	183	112	31.11	72.9	99	6.2	181.8	78	8.18	9.5	19	2.57	0	no
24	SC	111	area_code_415	no	no	0	110.4	103	18.77	137.3	102	11.67	189.6	105	8.53	7.7	6	2.08	2	no
25	VA	132	area_code_510	no	no	0	81.1	86	13.79	245.2	72	20.84	237	115	10.67	10.3	2	2.78	0	no
26	NE	174	area_code_415	no	no	0	124.3	76	21.13	277.1	112	23.55	250.7	115	11.28	15.5	5	4.19	3	no
27	WY	57	area_code_408	no	yes	39	213	115	36.21	191.1	112	16.24	182.7	115	8.22	9.5	3	2.57	0	no

Figure 1 - Train Dataset

3.2 Test Dataset

This dataset is structured as follows:

id	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail_messages	total_day_minutes	total_day_calls	total_day_charge	total_eve_minutes	total_eve_calls	total_eve_charge	total_night_minutes	total_night_calls	total_night_charge	total_intl_minutes	total_intl_calls	total_intl_charge	number_customer_service_calls	churn
1	HI	101	area_code_510	no	no	0	70,9	123	12,05	211,9	73	18,01	236	73	10,62	10,6	3	2,86	3	no
2	MT	137	area_code_510	no	no	0	223,6	86	38,01	244,8	139	20,81	94,2	81	4,24	9,5	7	2,57	0	no
3	OH	103	area_code_408	no	yes	29	294,7	95	50,1	237,3	105	20,17	300,3	127	13,51	13,7	6	3,7	1	no
4	NM	99	area_code_415	no	no	0	216,8	123	36,86	126,4	88	10,74	220,6	82	9,93	15,7	2	4,24	1	no
5	SC	108	area_code_415	no	no	0	197,4	78	33,56	124	101	10,54	204,5	107	9,2	7,7	4	2,08	2	no
6	IA	117	area_code_415	no	no	0	226,5	85	38,51	141,6	68	12,04	223	90	10,04	6,9	5	1,86	1	no
7	ND	63	area_code_415	no	yes	32	218,9	124	37,21	214,3	125	18,22	260,3	120	11,71	12,9	3	3,48	1	no
8	LA	94	area_code_408	no	no	0	157,5	97	26,78	224,5	112	19,08	310,8	106	13,99	11,1	6	3	0	no
9	MO	138	area_code_510	no	no	0	89,1	117	15,15	126,8	46	10,78	190,5	71	8,57	9,9	4	2,67	2	no
10	TX	128	area_code_415	no	yes	43	177,8	100	30,23	147,3	89	12,52	194,2	92	8,74	11,9	1	3,21	0	no
11	AR	113	area_code_510	no	yes	39	209,8	77	35,67	164,1	90	13,95	159,7	100	7,19	9	4	2,43	1	no
12	TX	140	area_code_415	no	no	0	93,2	109	15,84	197,6	116	16,8	219,8	94	9,89	10,5	2	2,84	1	no
13	ME	102	area_code_415	no	no	0	228,1	86	38,78	156	97	13,26	227,9	124	10,26	10,6	9	2,86	1	no
14	ND	108	area_code_415	no	no	0	112,6	86	19,14	114,9	101	9,77	177,8	119	8	7,2	6	1,94	3	no
15	DE	60	area_code_408	no	no	0	207,3	77	35,24	207,9	105	17,67	108,2	89	4,87	12,9	5	3,48	1	no
16	MN	96	area_code_408	no	no	0	208,1	93	35,38	189,2	107	16,08	279,6	90	12,58	7,4	2	2	1	no
17	KS	178	area_code_415	no	yes	22	112,8	66	19,18	232,6	100	19,77	194,8	119	8,77	14,3	3	3,86	1	no
18	MN	75	area_code_415	no	no	0	225,3	124	38,3	228	81	19,38	254,3	106	11,44	11,7	3	3,16	1	no
19	NC	106	area_code_415	no	yes	25	169,4	105	28,8	240,5	108	20,44	159,4	114	7,17	13,9	5	3,75	4	no
20	HI	158	area_code_510	no	no	0	193,3	121	32,86	208,1	97	17,69	228,1	99	10,26	7,1	9	1,92	1	no
21	NV	111	area_code_415	no	yes	35	161,2	142	27,4	159,1	104	13,52	167,9	98	7,56	14,7	5	3,97	1	no
22	CO	102	area_code_510	no	no	0	95,6	88	16,25	167,6	106	14,25	177,3	95	7,98	9,8	2	2,65	3	no
23	TN	92	area_code_510	no	yes	25	79,8	99	13,57	313,6	120	26,66	135,5	104	6,1	9,3	8	2,51	2	no
24	DE	42	area_code_415	no	yes	31	170,8	101	29,04	233,4	104	19,84	174,2	121	7,84	11	3	2,97	2	no
25	OH	69	area_code_415	no	no	0	229,2	111	38,96	165,3	104	14,05	235,1	80	10,58	5,2	5	1,4	1	no
26	OR	117	area_code_415	no	yes	38	259,3	94	44,08	245,6	71	20,88	269,3	125	12,12	9,2	1	2,48	3	no
27	NE	76	area_code_415	no	yes	41	212,6	110	36,14	172,7	97	14,68	186,3	78	8,38	10,1	5	2,73	0	no

Figure 2 - Test Dataset

4. Data Dictionary

Both datasets have 20 variables, one of which is the predictor variable, churn. Follows the Data Dictionary with the structuring of the variables.

Feature	Meaning
ID	Customer ID
state	Abbreviation for each state
account_length	Account extension
area_code	Area code for each state
international_plan	Indicative if the client has an international plan
voice_mail_plan	Indicative if the customer has a voice plan
number_vmail_messages	Number of voice messages the customer has
total_day_minutes	Number of minutes used during the day
total_day_calls	Number of calls made during the day
total_day_charge	Total cost of calls made during the day
total_eve_minutes	Number of minutes used in the afternoon
total_eve_calls	Number of calls made during the afternoon
total_eve_charge	Total cost of calls made during the afternoon
total_night_minutes	Number of minutes used during the night
total_night_calls	Number of calls made during the night
total_night_charge	Total cost of overnight calls
total_intl_minutes	Number of minutes during international calls
total_intl_calls	Number of international calls
total_intl_charge	Total cost of international calls
number_customer_service_calls	Number of calls to Customer Service
churn	This is the predictor variable, the objective of the study, indicating whether the customer canceled the service or not.

Table 1 – Data Dictionary

As we are working with Pyspark the reading mode is different from the conventional one provided by Python libraries. In this case, we have to create RDD's and load the data into them.

RDD[15] stands for Resilient Distributed Datasets and corresponds to a collection of objects partitioned in the cluster[16], distributed, immutable and with Spark structure.

Using a Spark Context - sc - (object that tells spark how to connect to the cluster) we read the dataset as follows:

```
telecomRDD = sc.textFile("projeto4_telecom_treino.csv")
```

textFile transforms a csv file into an RDD object. At this point, the telecomRDD object received the data, distributed it in the cluster and became immutable. To acquire any information, we must perform actions on the RDD's as:

```
telecomRDD.take(5)
```

This action returns the first 5 rows of the dataset:

```
['"","state","account_length","area_code","international_plan","voice_mail
_plan","number_vmail_messages","total_day_minutes","total_day_calls","tota
l_day_charge","total_eve_minutes","total_eve_calls","total_eve_charge","to
tal_night_minutes","total_night_calls","total_night_charge","total_intl_mi
nutes","total_intl_calls","total_intl_charge","number_customer_service_cal
ls","churn"',
 '"1","KS",128,"area_code_415","no","yes",25,265.1,110,45.07,197.4,99,16.7
8,244.7,91,11.01,10,3,2.7,1,"no"',
 '"2","OH",107,"area_code_415","no","yes",26,161.6,123,27.47,195.5,103,16.
62,254.4,103,11.45,13.7,3,3.7,1,"no"',
 '"3","NJ",137,"area_code_415","no","no",0,243.4,114,41.38,121.2,110,10.3,
162.6,104,7.32,12.2,5,3.29,0,"no"',
 '"4","OH",84,"area_code_408","yes","no",0,299.4,71,50.9,61.9,88,5.26,196.
9,89,8.86,6.6,7,1.78,2,"no"']
```

Note that each line has its data in single quotes (shown in red in the image above), indicating that they were recognized as a single string (text). It is necessary to transform this information so that each data is in its respective column, so, before the exploratory analysis we will carry out the feature engineering, treating and adding relevant variables.

Even before performing feature engineering, let's understand how PySpark Shell works.

[15] https://spark.apache.org/docs/latest/rdd-programming-guide.html
[16] Set of interconnected computers that work as if they were one big system.

5. PySpark Shell

As stated earlier in the introductory chapter from now on we will analyze the execution of the Jobs generated in Pyspark Shell. Not all will be presented as there is repetition in the code's actions, but the most relevant will be highlighted in the documentation of this project. Interesting fact will be when we enter the chapter of Machine Learning because it will be possible to see the large amount of Jobs generated sequentially.

The action of the previous chapter ("take") generated the following job:

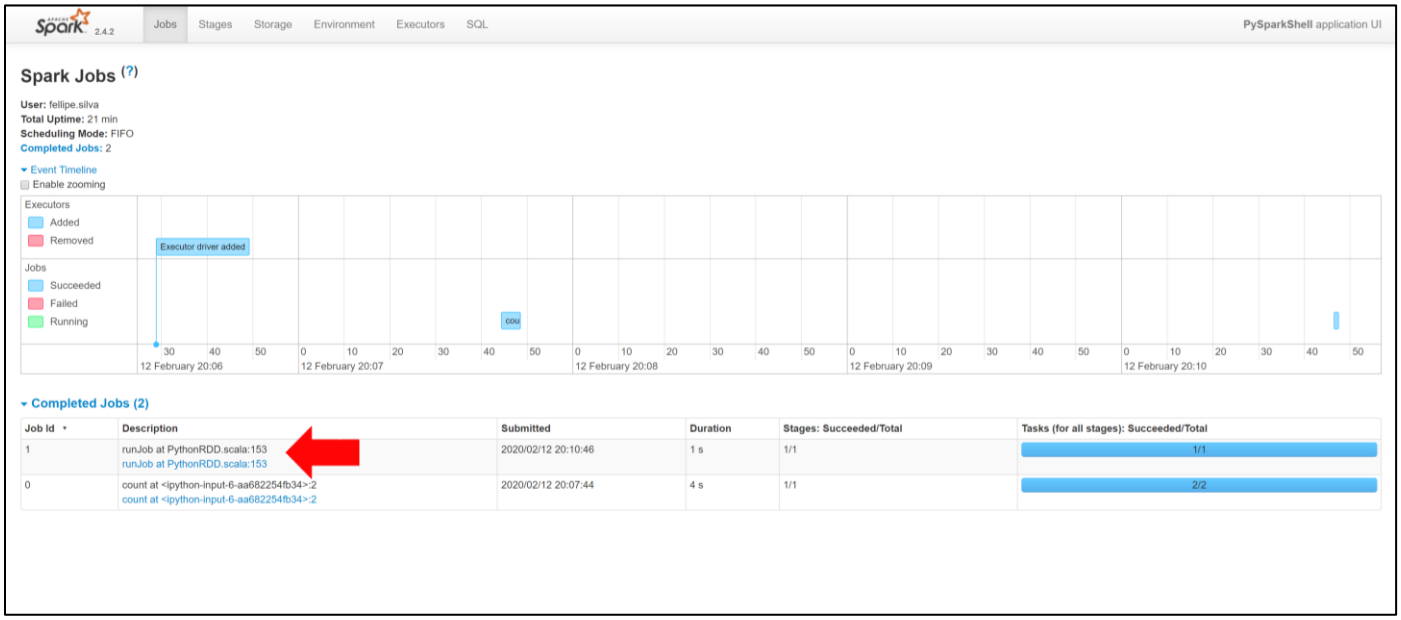

Figure 3 - Pyspark Shell I

As illustrated in the figure above, we have two Jobs executed, one from the test I performed earlier to retrieve from the cluster the number of rows we have in the data ("count") and the job with the red arrow ("take") that retrieves all the data of my telecomRDD.

With the action of Job id 0 it takes us four seconds to recover, and with Job id 1 it takes us a second, because in the action of the "take" I requested only the first 5 lines, while the action of the "count" requested everything there is in the cluster.

Note that we have a timeline where it shows in real time the execution of all Jobs. At the end of the project, we will have a series of operations filling this timeline.

Within each Job we can see the details of its execution:

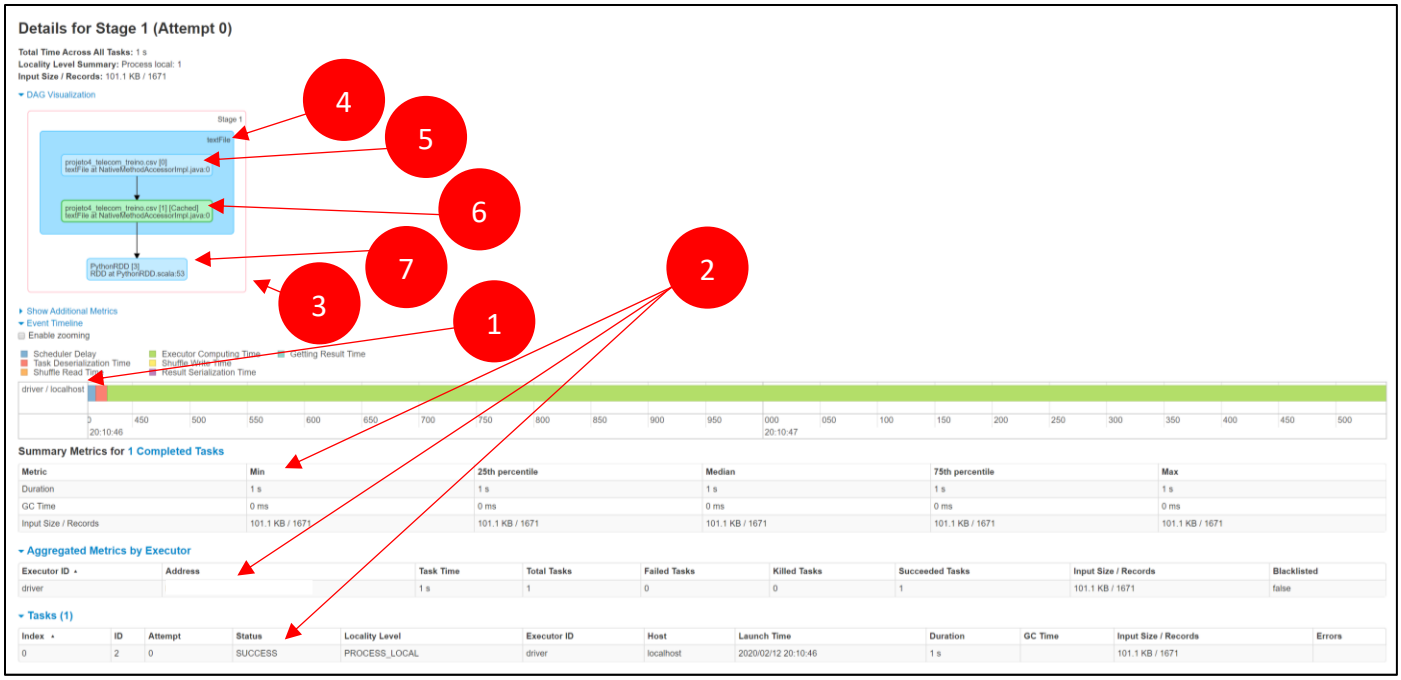

Figure 4 - Pyspark Shell II

In item 1 we can see the specific timeline of that Job within the cluster. Note that because we are not in a production environment we have only one machine in the cluster, localhost. If we had a configured cluster, Jobs' data and “tasks” would be distributed among the machines and we would see the execution on each of them.

In item 2 we have details of the execution time, metrics indicating the time spent in each quartile of execution, the address of the machine that requested the “task”, how much memory was used and if the Job was successful or failed.

Item 3 concerns the visualization of the DAG (Direct Acyclic Graph) and refers to the chain of dependencies and execution of an RDD. In this case we can see that spark executes the concept of pipeline[17] in operations, passing from instruction to instruction until the final result.

In this DAG we have blue and green boxes. The box in item 4 represents an operation that I wrote in my code, in this case a read operation with the “textfile”. This reading is from an input file on HDFS[18] (Hadoop Distributed File System). The box in item 5 represents the RDD generated by this reading operation. Each smaller box (items 5, 6, 7) is RDD’s.

Then, in green (item 6) we have the same instruction as item 5, but with the command "cache" persisting the RDD in memory for faster access later, so that I don't need to access this data in HDFS.

Finally, in item 7 we have the final RDD with the action “take” returning the result previously presented.

We move on to the next chapter with Feature Engineering.

[17] Data entry is processed and delivered for next processing following a queue.
[18] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

6. Feature Engineering

Feature Engineering is the process of treating, adding and removing variables. This process consists of finding out which columns of data create the most useful attributes to improve the accuracy of the machine learning model.

As our model does not read string, I removed the first line with the action "filter", that is, it filters everything that does not have the determined condition:

```
telecomRDD2 = telecomRDD.filter(lambda x: "state" not in x)
```

Note that I did not attribute the result to the RDD already created (telecomRDD) as we would have an error message indicating its immutability. This way we have telecomRDD2 to perform cleaning and transformation.

```
['"1","KS",128,"area_code_415","no","yes",25,265.1,110,45.07,197.4,99,16.7
8,244.7,91,11.01,10,3,2.7,1,"no"',
 '"2","OH",107,"area_code_415","no","yes",26,161.6,123,27.47,195.5,103,16.
62,254.4,103,11.45,13.7,3,3.7,1,"no"',
 '"3","NJ",137,"area_code_415","no","no",0,243.4,114,41.38,121.2,110,10.3,
162.6,104,7.32,12.2,5,3.29,0,"no"',
 '"4","OH",84,"area_code_408","yes","no",0,299.4,71,50.9,61.9,88,5.26,196.
9,89,8.86,6.6,7,1.78,2,"no"',
 '"5","OK",75,"area_code_415","yes","no",0,166.7,113,28.34,148.3,122,12.61
,186.9,121,8.41,10.1,3,2.73,3,"no"']
```

With the data without the header, we perform transformation actions, so I created a function facilitating the process of manipulating the variables where I initially divide the dataset using the comma separator (“,”) and later assigning each variable to an object converting to int, float , str according to its type.

Still in the function, after the process above, each variable will be inserted in an object "lines" through the function “Row”[19] that will prepare the lines to later create a Pyspark Dataframe assisting in the use of the "select" function of SparkSql for exploratory analysis.

[19] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Row

```
    # Feature Engineering I - Function
def featengI(data):

    dataList = data.split(",")

    ID = (dataList[0])
    STATE = str(dataList[1])
    ACCLGT = int(dataList[2])
    AREACODE = (dataList[3])
    #PHONENB = (dataList[4])           # Phone_No -> Phone Number (IT IS NOT IN THE test-trai
    n DATA)
    #INTPLAN = (dataList[4])           # International_Plan -> "yes" or "no" (Convert yes->1
    or no->2)
    INTPLAN = 1.0 if dataList[4] == '"yes"' else 2.0
    #VMPLAN = (dataList[5])            # Voice_Mail_Plan -> "yes" or "no" (Convert yes->1 or
    no->2) I THINK THIS FEATURE IS NOT HELPING AND CAN BE ELIMINATED BECAUSE THE NEXT VARIABLE (N
    BVMMSG) ONLY HAS VALUES WHEN THIS IS '1', IT'S LIKE DOUBLE INFORMATION ABOUT THE SAME THING
    VMPLAN = 1.0 if dataList[5] == '"yes"' else 2.0
    NBVMMSG = int(dataList[6])
    TTDAYMIN = float(dataList[7])
    TTDAYCALLS = int(dataList[8])
    TTDAYCHARGE = float(dataList[9])
    TTEVEMIN = float(dataList[10])
    TTEVECALLS = int(dataList[11])
    TTEVECHARGE = float(dataList[12])
    TTNGTMIN = float(dataList[13])
    TTNGTCALLS = int(dataList[14])
    TTNGTCHARGE = float(dataList[15])
    TTINTMIN = float(dataList[16])
    TTINTCALLS = int(dataList[17])
    TTINTCHARGE = float(dataList[18])
    NBCSCALLS = int(dataList[19])
   #TARGET = (dataList[20])          # Churn (Target) -> "yes" or "no" (Convert yes->1 or no->2)
    TARGET = 1 if dataList[20] == '"yes"' else 2

    lines = Row(ID = ID, STATE = STATE, ACCLGT = ACCLGT, AREACODE = AREACODE,
                INTPLAN = INTPLAN, VMPLAN = VMPLAN, NBVMMSG = NBVMMSG,
                TTDAYMIN = TTDAYMIN, TTDAYCALLS = TTDAYCALLS, TTDAYCHARGE = TTDAYCHARGE,
                TTEVEMIN = TTEVEMIN, TTEVECALLS = TTEVECALLS, TTEVECHARGE = TTEVECHARGE,
                TTNGTMIN = TTNGTMIN, TTNGTCALLS = TTNGTCALLS, TTNGTCHARGE = TTNGTCHARGE,
                TTINTMIN = TTINTMIN, TTINTCALLS = TTINTCALLS, TTINTCHARGE = TTINTCHARGE,
                NBCSCALLS = NBCSCALLS, TARGET = TARGET)

    return lines
```

As previously mentioned, machine learning models do not interpret texts, only numbers, so the variables INTPLAN, VMPLAN and TARGET were converted into the function where “1” corresponds to “yes” and “2” to “no” in each one.

After mapping, we can create a PySpark SQL Dataframe to use a query to obtain data and analyze it.

```
telecomDF = spSession.createDataFrame(telecomRDD3)
```

I used telecomDF object to structure the variables (source code at the end of this document) resulting in the following columns:

```
|-- ID: integer (nullable = true)
|-- STATE: string (nullable = true)
|-- ACCLGT: long (nullable = true)
|-- INTPLAN: double (nullable = true)
|-- VMPLAN: double (nullable = true)
|-- NBVMMSG: long (nullable = true)
|-- TTDAYMIN: double (nullable = true)
|-- TTDAYCALLS: long (nullable = true)
|-- TTDAYCHARGE: double (nullable = true)
|-- (TTDAYMIN / TTDAYCALLS): double (nullable = true)
|-- TTEVEMIN: double (nullable = true)
|-- TTEVECALLS: long (nullable = true)
|-- TTEVECHARGE: double (nullable = true)
|-- (TTEVEMIN / TTEVECALLS): double (nullable = true)
|-- TTINTMIN: double (nullable = true)
|-- TTINTCALLS: long (nullable = true)
|-- TTINTCHARGE: double (nullable = true)
|-- (TTINTMIN / TTINTCALLS): double (nullable = true)
|-- NBCSCALLS: long (nullable = true)
|-- TARGET: long (nullable = true)
```

In addition to the added variables, for good practices the name of the predictive variable *churn* was replaced by *target*, as indicated by the red arrow above.

With the feature engineering completed, we can start the exploratory analysis process.

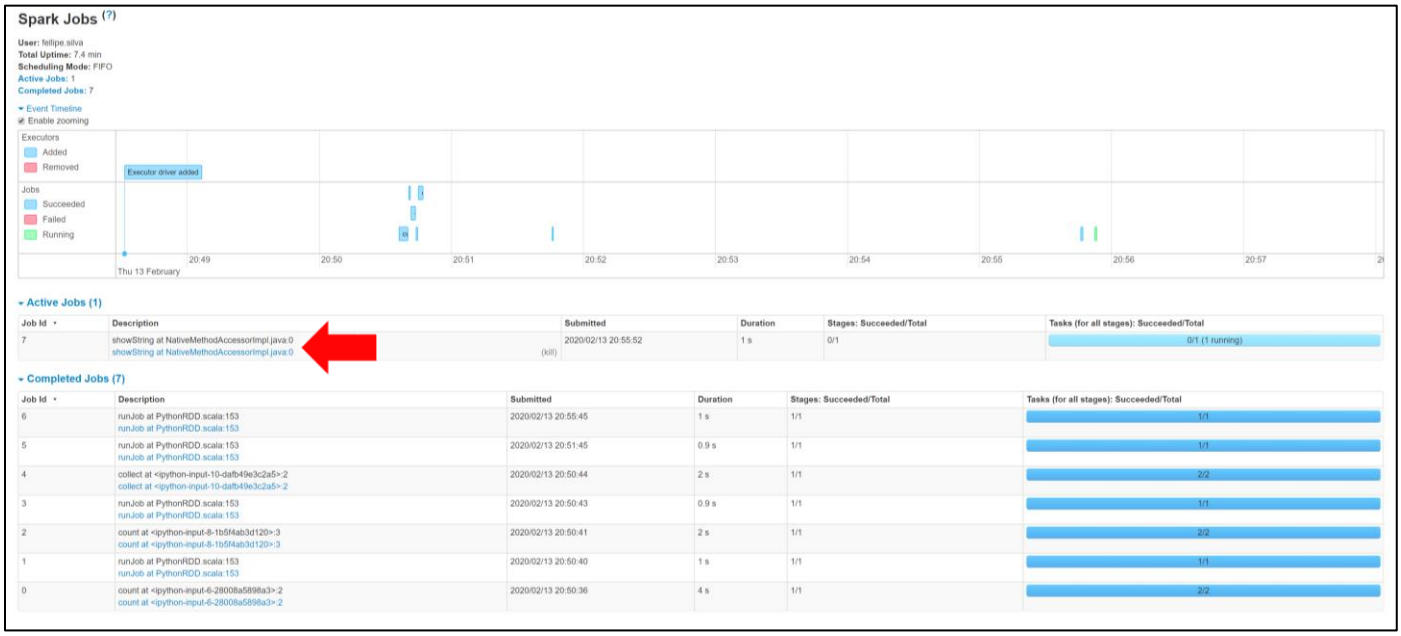

Figure 5 – Processing Jobs

Notice on the red arrow in the Pyspark Shell that Jobs running also appear while they are being processed.

7. Exploratory Analysis

Exploratory analysis is essential to understand where we have better variables, where we have problems, where we have business opportunities, and in this way, we can generate a series of important conclusions to continue the machine learning process.

To perform exploratory analysis Pyspark SQL functions were initially used, which are minor modifications in the traditional SQL language. However, as the traditional SQL language is more widely used, I made a modification in Pyspark Dataframe so that I could use standard SQL language.

This is a Pyspark SQL statement:

```
telecomDF4.cube('TARGET').agg(countDistinct('STATE').alias('Exclusive States'),
                              grouping('TARGET')).orderBy('TARGET').show()
```

Resulting in:

```
+------+----------------+----------------+
|TARGET|Exclusive States|grouping(TARGET)|
+------+----------------+----------------+
|  null|              51|               1|
|     1|              51|               0|
|     2|              51|               0|
+------+----------------+----------------+
```

Table 2 - Spark SQL

For those who have never worked with Spark it is necessary to consult its extensive documentation[20] and search for the necessary commands to perform selections, aggregations, ordering and organization of data.

Due to the ease of data manipulation with SQL language I created a temporary table ("telecomTB") in memory, speeding up data recovery and facilitating the use of SQL commands.

This is the same command used in table 2 but in SQL language:

```
spSession.sql("select TARGET, count(distinct STATE) as Exclusive_States
               from telecomTB
               group by TARGET").show()
```

That returns as a result:

[20] https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=sql#pyspark.sql.DataFrame

```
+------+----------------+
|TARGET|Exclusive_States|
+------+----------------+
|     1|              51|
|     2|              51|
+------+----------------+
```

Table 3 - SQL ANSII

Both commands return the number of States present in the dataset that have churn 1 and churn 2, that is, the occurrence of service cancellations or not, respectively. Note that we have 51 states, with cancellations in all of them, but how much does each state occupy within the cancellations made? We will analyze in detail in the following topics.

Pyspark Shell in this case was processed with more instructions within the cluster:

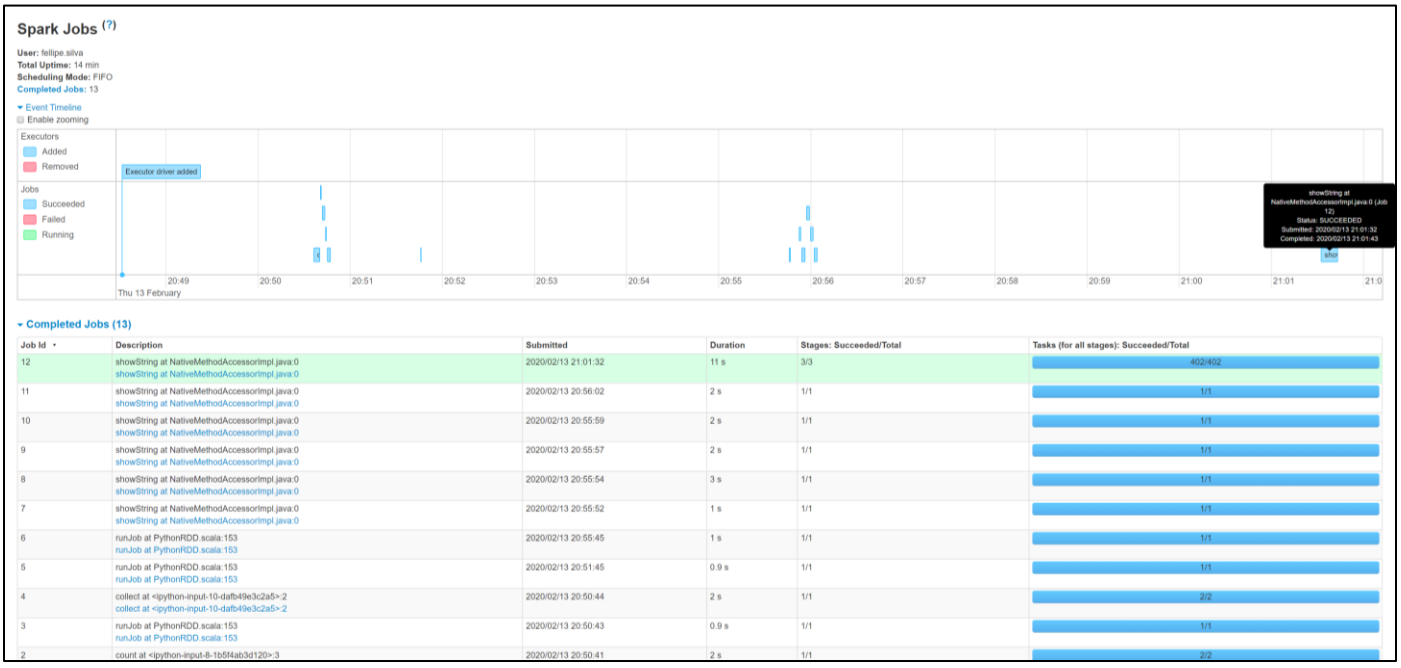

Figure 6 – Longer Jobs

Figure 6 shows the timeline and the last job executed by Spark, which in this case refers to the Spark SQL statement. Note that instead of 1 task we have 402 tasks to bring the final result. As it is a more extensive process, spark took 11 seconds to return the information, which is not much compared to the amount of data in the operation.

Within the job we can see that the DAG has more stages (stage 12, 13, 14), as shown in figure 7 below, and that they communicate, that is, the final result of one is the input of another and so on.

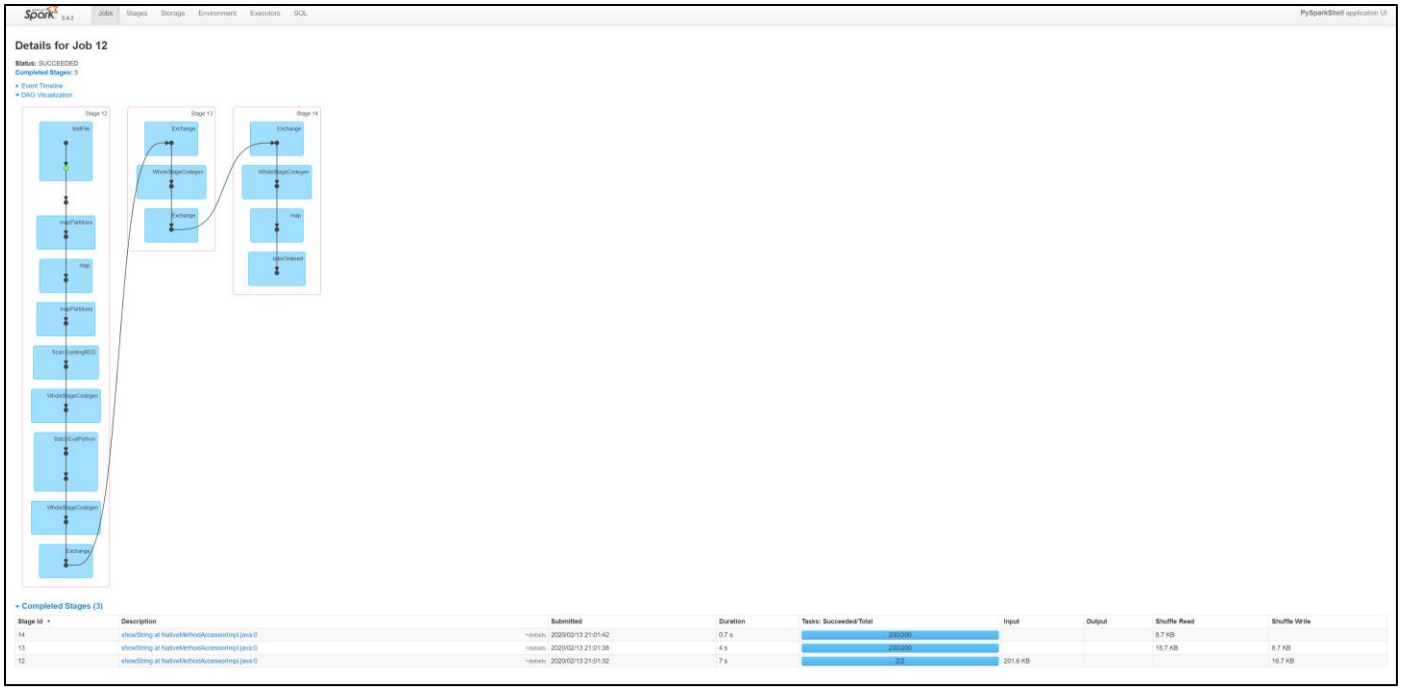

Figure 7 – DAG longer jobs

Pyspark shell provides an interactive analysis, that is, if you click on each stage it is possible to see in detail the execution of each RDD.

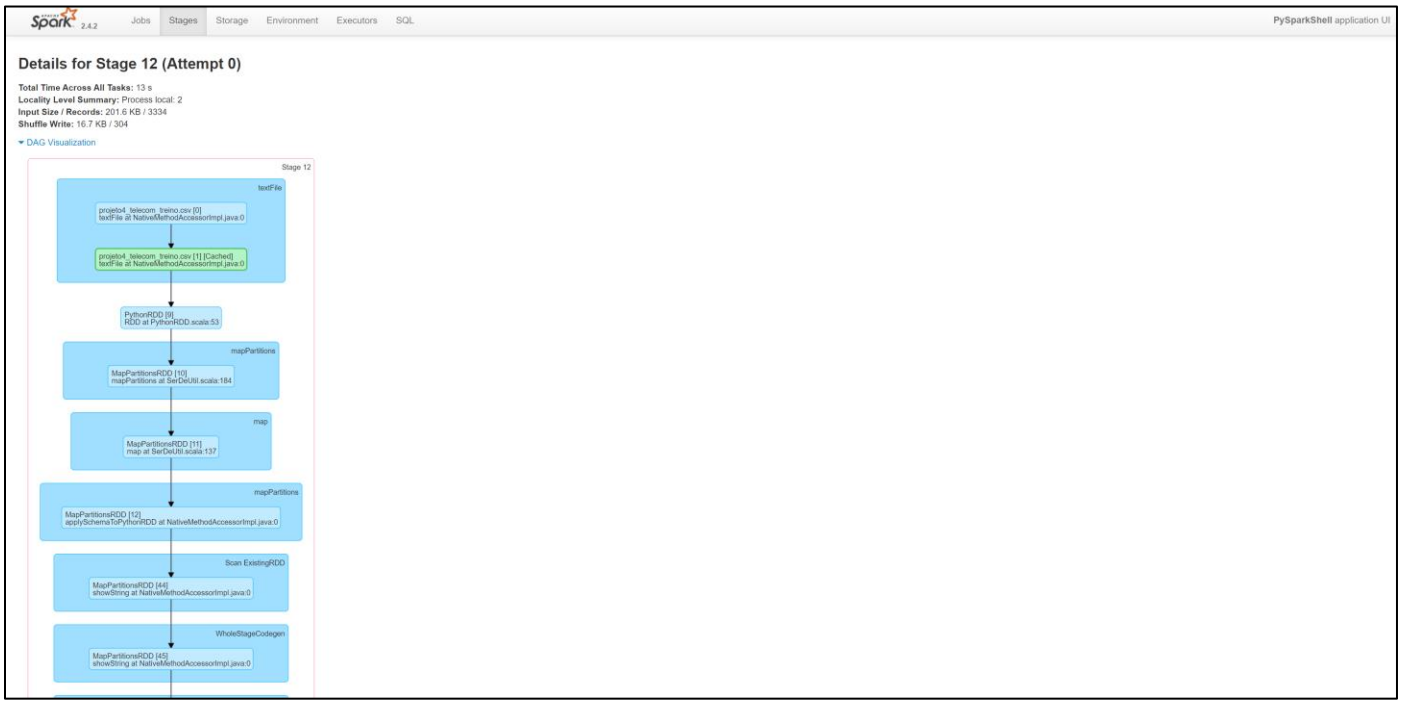

Figure 8 - Stage 12

Continued in figure 9.

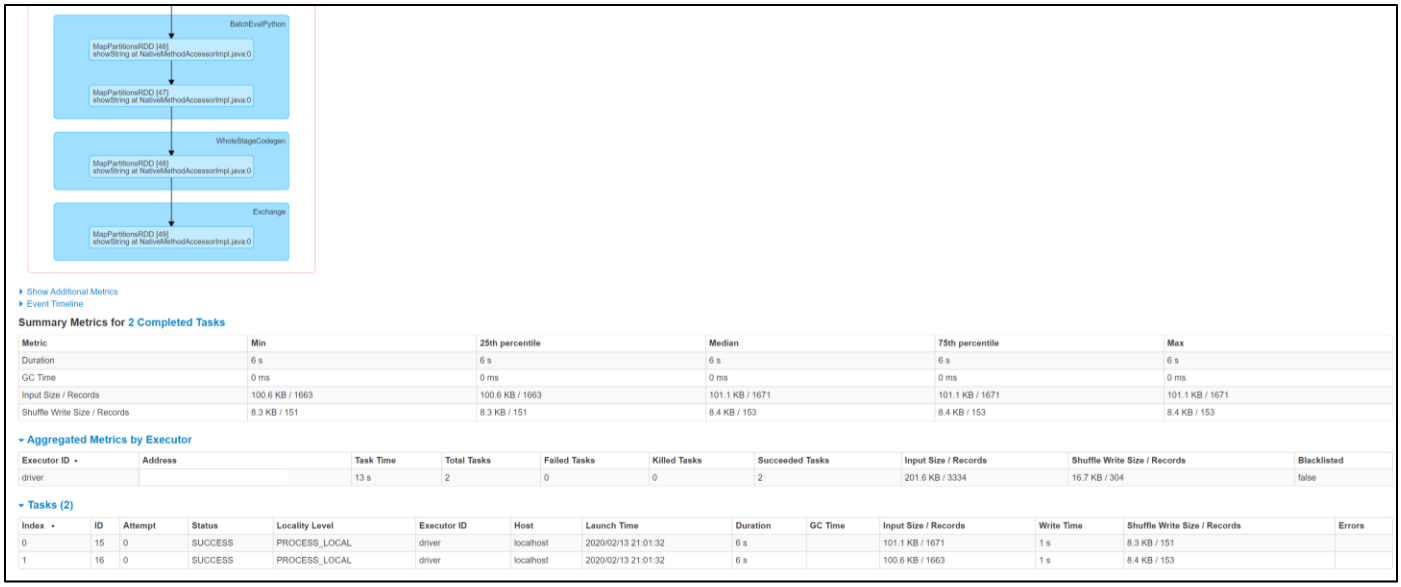

Figure 9 – Continuation Stage 12

Notice in figure 9 that the last instruction in the blue box is “takeOrdered” because what I wrote in my code was to order the final result in descending order, presenting on screen an easier way to view the result.

I believe that you have already understood the importance of Pyspark Shell for the Data Science process, with it I can carry out the execution of the code following how each instruction requested to the cluster was managed and in the event of a possible error it is faster to find where there are failures, analyze the problems and solve them to continue the project.

7.1 Global map of the States under study (according to the dataset)

Figure 10 – Global Map

As you can see, our dataset is centered on information from the United States, the map below visually presents all the states involved.

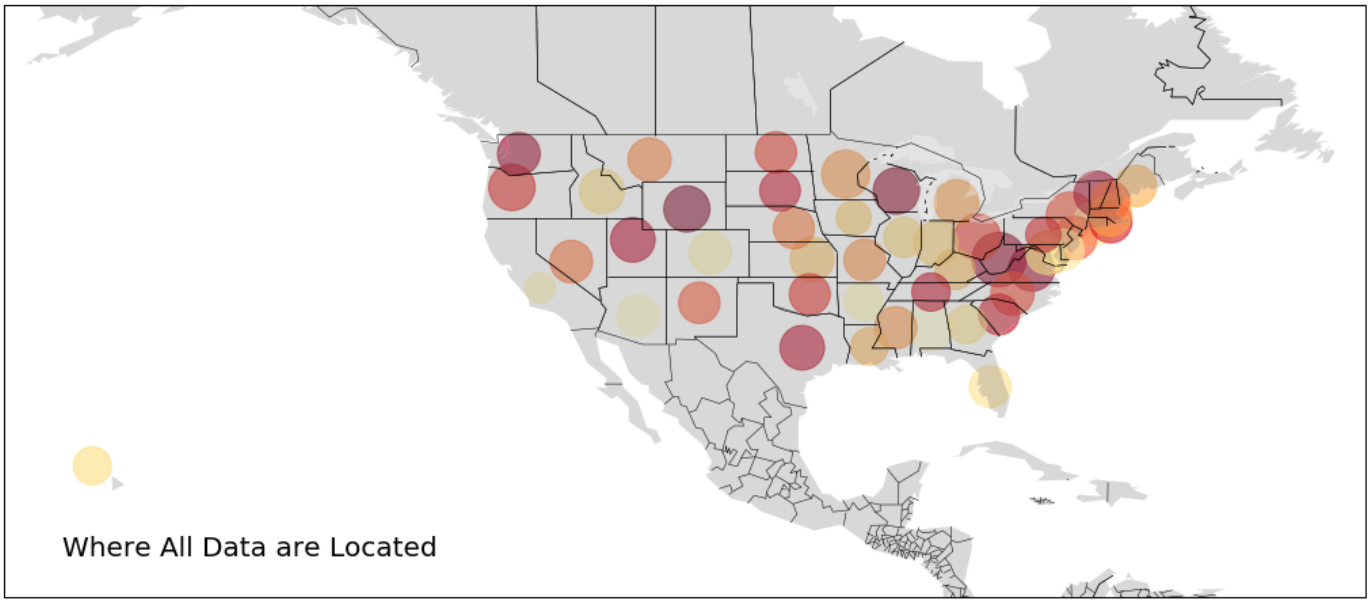

Figure 11 – Local Map

Each circle on the map has a size indicating the volume of data that each State contributed to this project, that is, the larger the circle the more information collected from that State and vice versa.

To assemble the map, it is necessary to indicate the latitude and longitude of each State, information that the dataset provided does not have. So for the construction of the map I collected latitude and longitude information for each North American state through google maps, fed a CSV spreadsheet and used join commands in my dataset provided using the acronym of each state as a key.

	STATE	latitude	longitude	label_color
48	WV	38.491226	-80.954453	48
49	WI	44.268543	-89.616508	49
50	WY	42.755966	-107.302490	50

Table 4 - Lat-Long by State

With this parameterization it was possible to analyze each region according to the volume of data and plot the result using a library package Matplotlib[21] called Basemap[22].

[21] https://matplotlib.org/
[22] https://matplotlib.org/basemap/

Unfortunately this package has been discontinued from the matplotlib library but it is possible to find the necessary API's on the internet and install manually by changing some machine variables. To do this, consult the official documentation[23] and follow the installation and configuration guide.

The matplotlib library is versatile and offers a multitude of options for presenting results graphically, from bar graphs to 3D mapping. During the execution of the project, I mixed the use of matplotlib with another excellent plot library Seaborn[24].

Seaborn is a python data visualization library based on matplotlib and offers high level graphics, making it possible to make a series of adjustments to the configuration parameters of the graphics providing more attractive shapes and colors in addition to the easy visual presentation of results.

As shown on the map of the USA, the highest concentration is in the states located in the northeast region, confirmed by the following wordcloud:

Figure 12 - WordCloud

The wordcloud indicates the acronyms of the twenty states with the highest concentration, being:

1º - WV: West Virginia

2º - MN: Minnesota

3º - NY: New York

[23] https://matplotlib.org/basemap/

[24] https://seaborn.pydata.org/index.html

To consolidate I present the 51 States with their respective participation, note that the last position is occupied by the State of California:

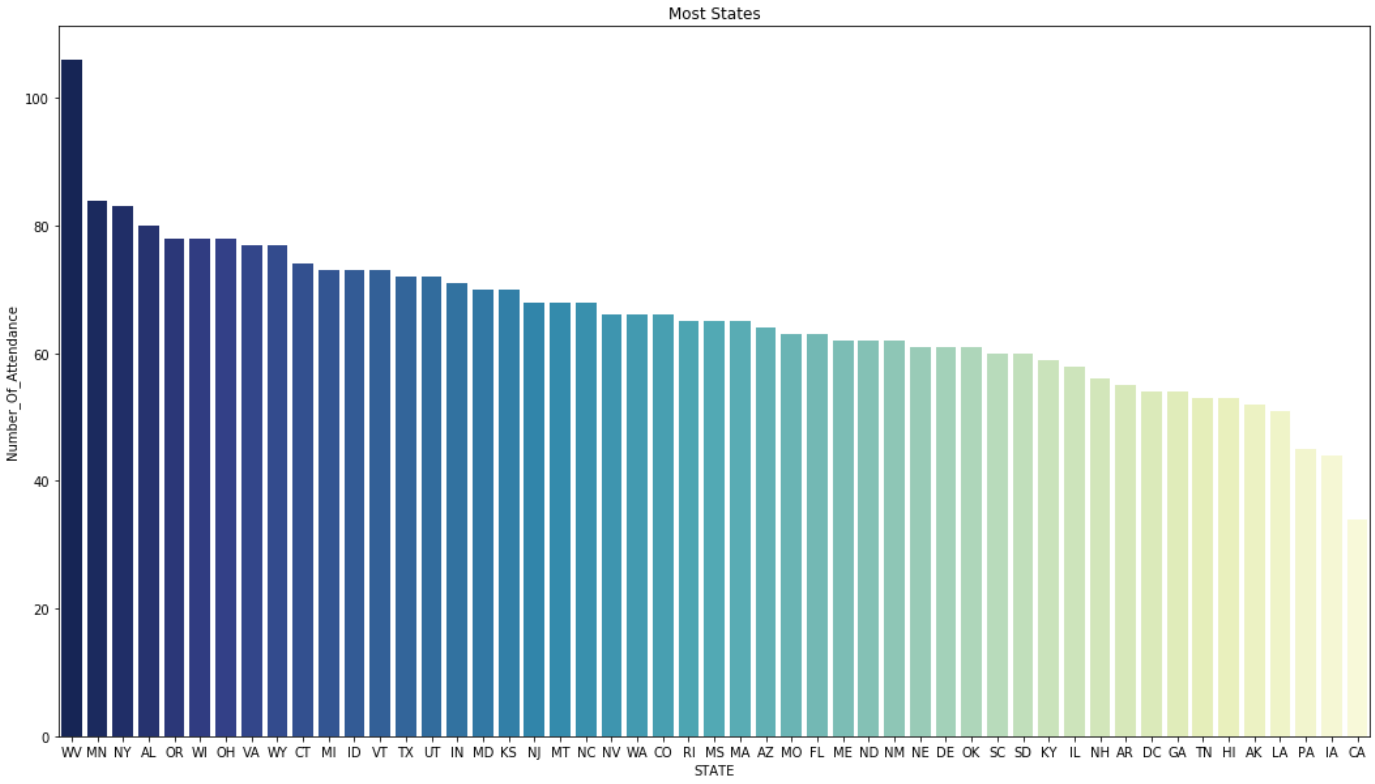

Figure 13 – States Volume

The above amounts do not necessarily indicate dissatisfaction with the services provided by the telecom company. We have to analyze each case separately and check if the States with the most records are also the States with the most disapproval (hypothesis 1). Study follows.

7.2 Higher and lower churn rate states

```
spSession.sql("select STATE, TARGET as Churn, count(ID) as Number_Of_Attendance \
               from telecomTB \
               group by STATE, Churn \
               order by STATE, Churn")
```

The SQL command above acquires the data of the States and the churn of each one, ordering according to the name of the State.

With this information, I performed the churn rate analysis to verify the hypothesis of item1. Graph below:

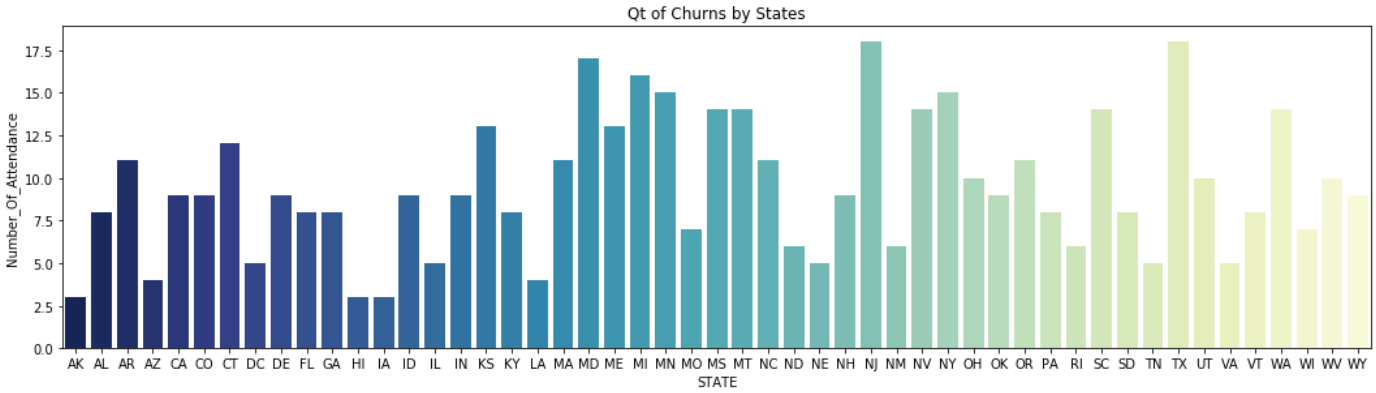

Figure 14 - Churn Rate by State

Searching for West Virginia (WV) in the chart above it is possible to state that it is not among the states with the highest Churn Rate, with Texas (TX) and New Jersey (NJ) being the states with the most cancellations of the telecom company's plans.

The previous hypothesis was refuted through individual analysis, so it is important to carefully analyze each finding. Drawing hasty conclusions without analyzing the data can lead to small errors with great impact on the final result.

Returning to the visualization of the global map, we have the following 5 states with the highest Churn Rate records and then the 5 with the lowest Churn:

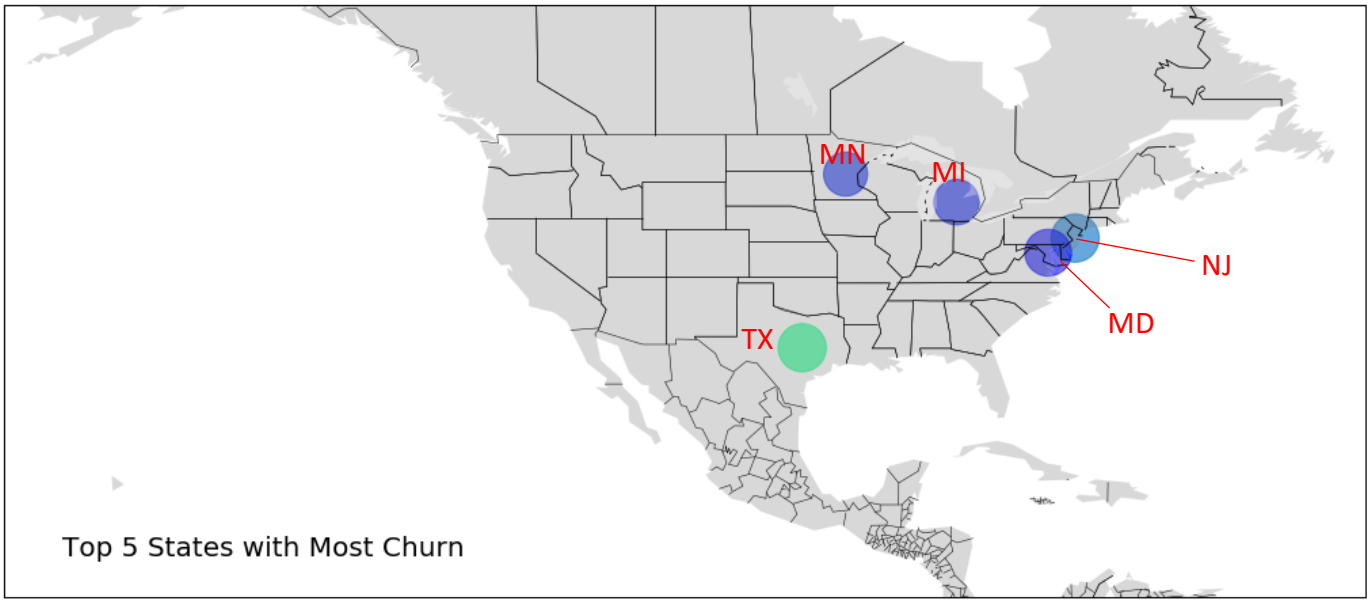

Figure 15 - Top 5 States with most Churn

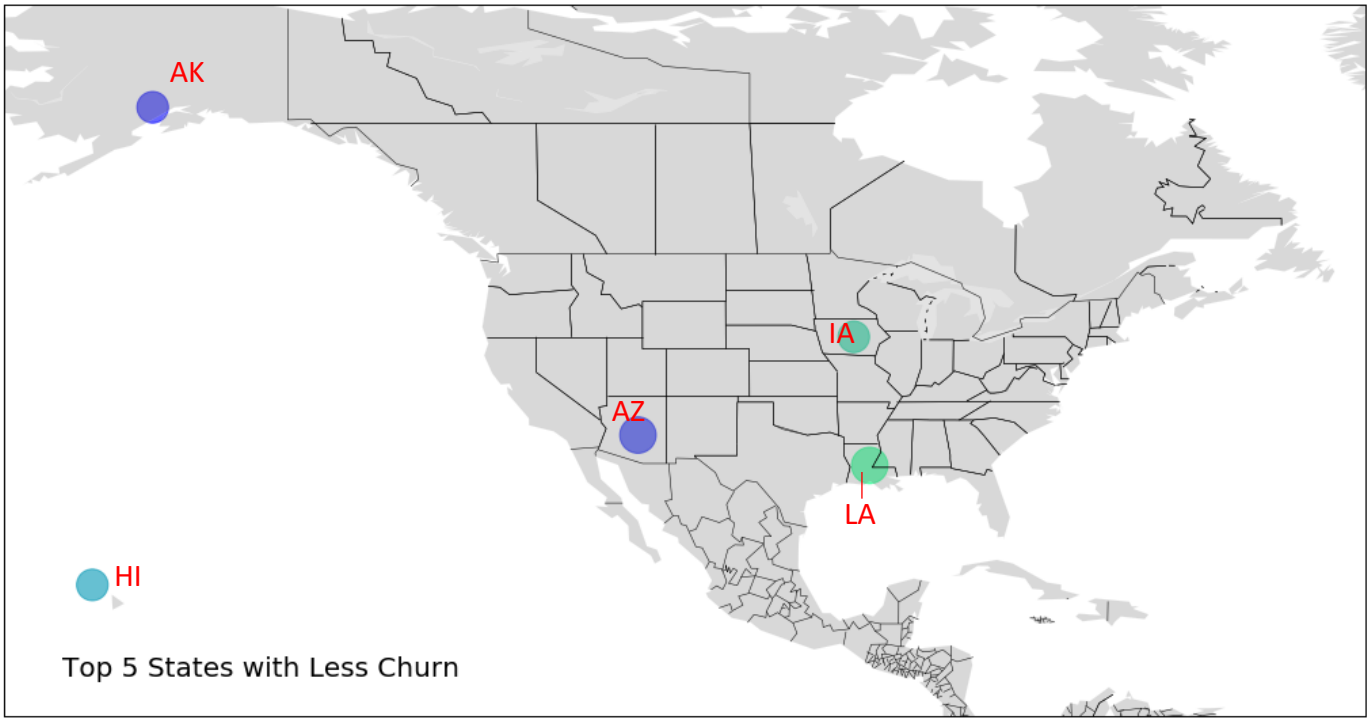

Figure 16 - Top 5 States with less Churn

Working with data means identifying and generating opportunities from hidden but valuable information.

For this business problem I found a series of opportunities hidden in the data and that can be used to offer services in the windows that are not being filled by both customers and the Telecom company and that perhaps both have not identified.

I share in the following items the insights acquired as business opportunities.

7.3 First Business Opportunity - International Plans

Initially, we will look at international plans. Through the command below we can obtain information regarding States that make international calls and do not have an international plan.

```
spSession.sql("select STATE, count(ID) as QT_INTPLAN \
               from telecomTB \
               where INTPLAN == 2 \
               group by STATE \
               order by QT_INTPLAN desc")
```

STATE	QT_INTPLAN
WV	99
MN	76
NY	75
AL	72
OR	71
WI	70
OH	69
VA	68
IN	68
ID	67

Table 5 – International Plans by State

These are the 10 States with most international calls and who do not have international plans. This is valuable information for the marketing and sales team as it is an opportunity to set up an action plan to achieve more revenue by offering plans by regions.

It is possible to go further and study individual personalized plans, that is, look at each consumer, our second business opportunity.

7.4 Second Business Opportunity - Customized International Plans

At this moment we enter specifically into the consumption of each customer.

CUSTOMER	TTINTCALLS
23	19
378	18
2957	18
983	18
3311	17
1568	16
2622	16
675	15
637	15
1890	15
2836	15
1393	15
922	14
2002	14
1180	14
3231	14
1356	14
757	14
186	13
3207	13

Table 6 – International Plan by Customer

In table 6 we have the columns:

- CUSTOMER: Customer ID;
- TTINTCALLS: Total International Calls.

Through the data collected, we can map the use of international calls by consumers and offer customized individualized plans for each customer. Although it is valuable information for the sales team, how to put together a marketing plan that can convince consumers that they need to hire an international plan? The answer is again in the data.

7.5 Third Business Opportunity - International Pricing Plans

So far we have identified the states with the most international calls and without an adequate plan, soon after we were able to determine which consumer does not have the plan and frequently calls outside the country, but delivering this information to the customer may not generate value and therefore may not convert into sale of new plans.

To add value to the marketing plan we have to analyze how the lack of an international plan is impacting the client financially and how much he could save by hiring an appropriate plan.

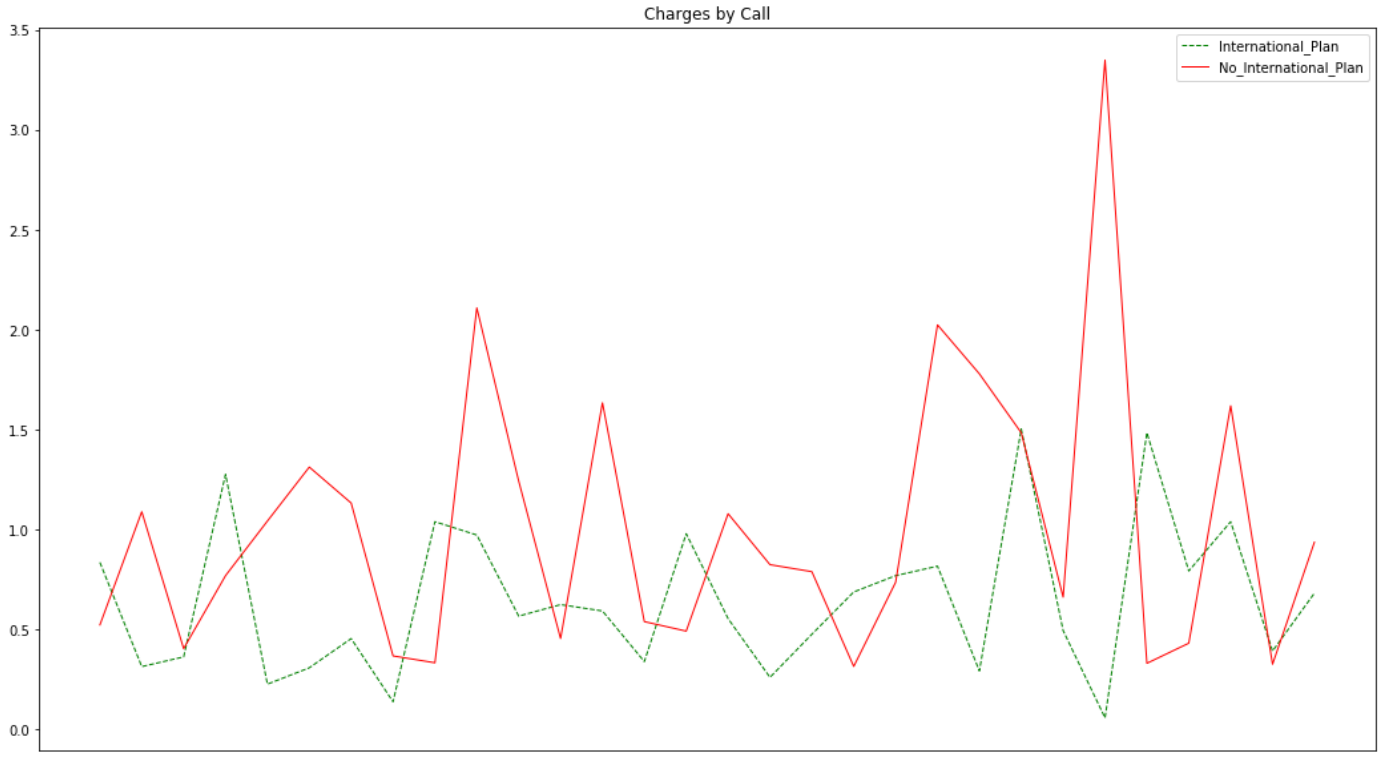

Figure 17 - Cost per International Call

Through a collection of information such as total calls and taxation it was possible to assemble the graph in image 17 illustrating how much is being paid by consumers with an international plan and consumers without an international plan.

In the image, the green line is the amounts paid to the Telecom company for international calls made by customers with an international plan. The red line indicates the amounts paid for international calls by those who do not have this plan.

Notice in the graph that the red line is almost always above the green line indicating that more fees are paid by customers without an international plan. Statistically, through the data, we can calculate and affirm that:

Customers without an international plan are paying 55.82% more fees!

With this information, we can generate value for the marketing plan because we will have enough content to prove to the client that he can save money by hiring an international plan.

These are some of the opportunities that can be offered to the customer, but what about the opportunities that the Telecom company can take advantage of? Are there windows not filled in by the company? Yes, there is hidden information in the data that can be revealed to the company's administrators in terms of earned revenue that can transform the way of managing the business, another opportunity offered by the data.

7.6 Fourth Business Opportunity - Acquisition of Revenue

Calculating the average rate applied in each period of the day we have the following:

Fees applied by the Telecom company per day period	
Day	$ 0,170
Evening	$ 0,085

Table 7 - Rates Per Minute

Knowing this, we can calculate the financial impact within the total volume of calls made per period. First let's look at the total volume:

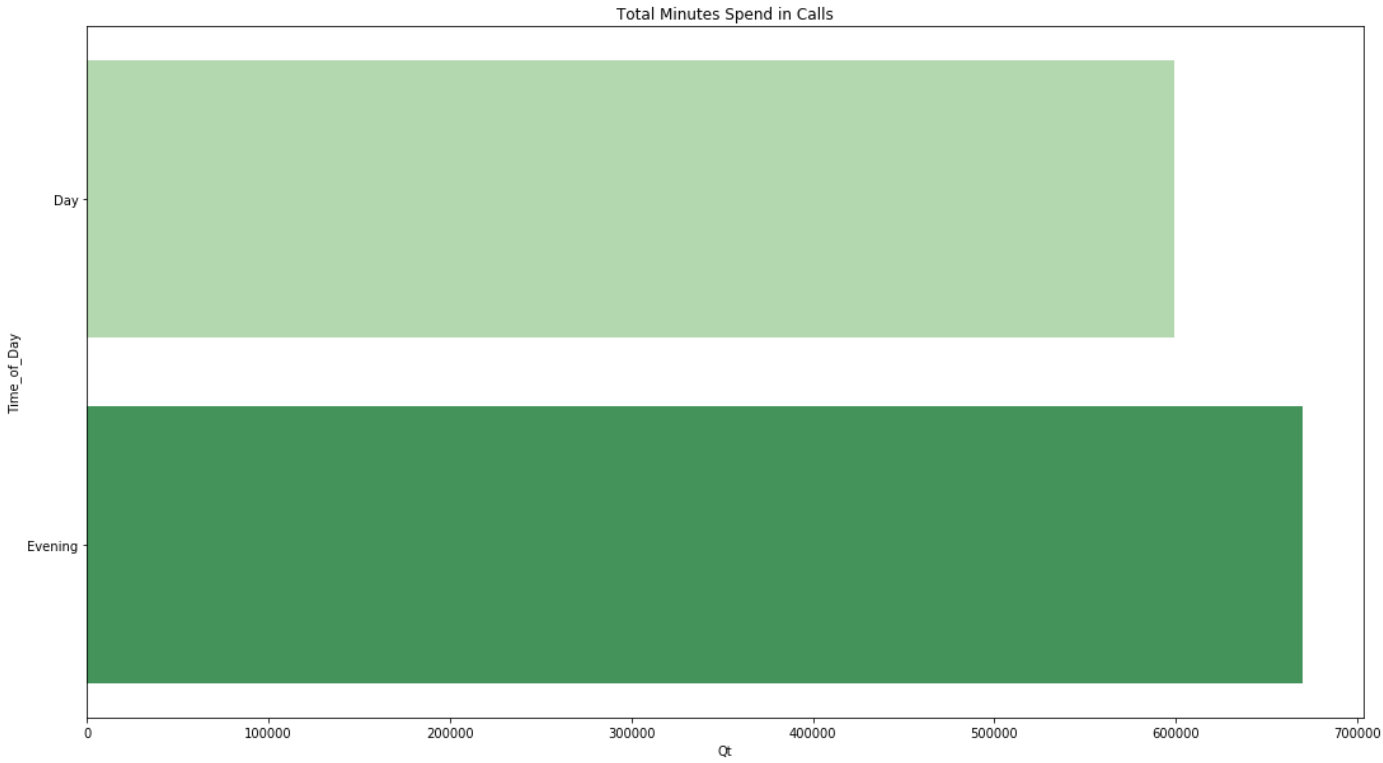

Figure 18 - Total minutes used per period

As shown in figure 18, the volume of calls made in the afternoon is higher, which does not mean that this is the period with the highest revenue as the rate is lower as follows:

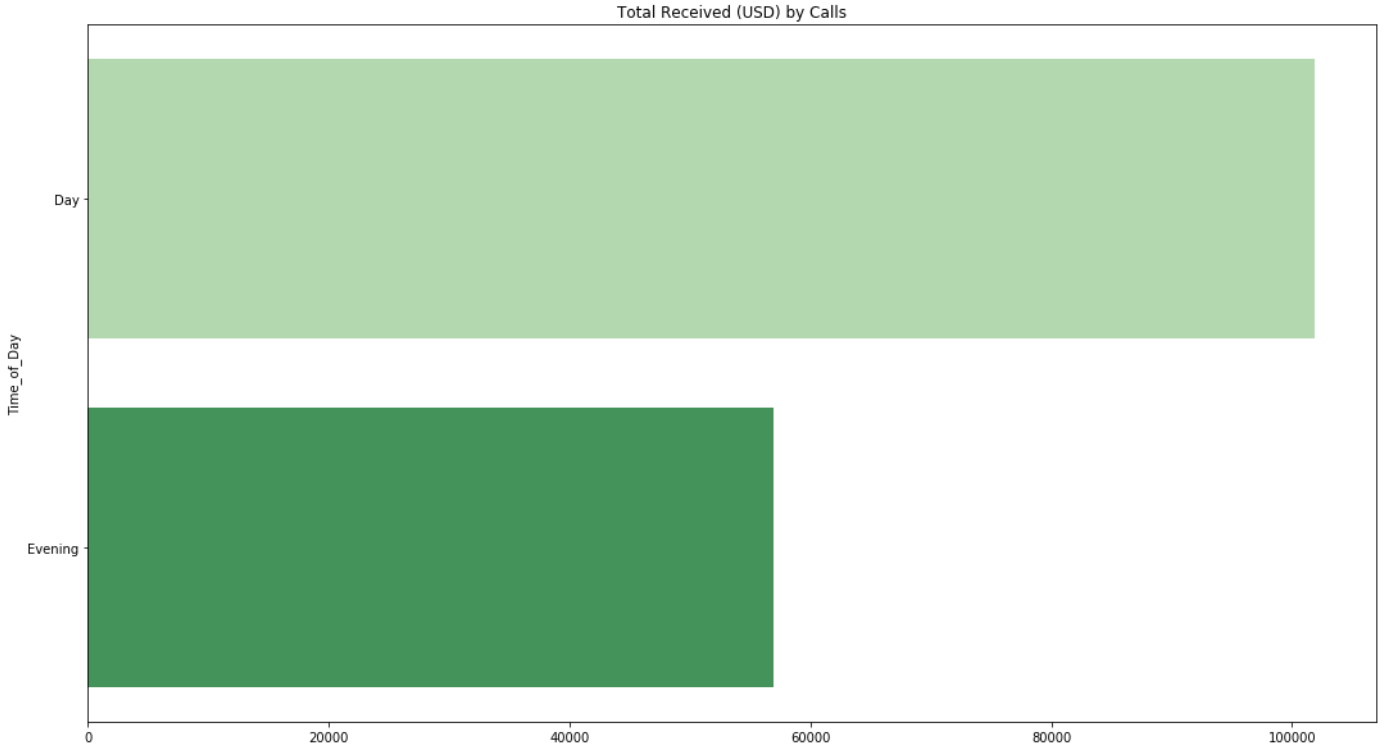

Figure 19 - Total revenue per period

For the data collection period we have revenue from:

Current Revenue: $ 158,801.11

And if 0.01 cents were changed in each rate, what effect on revenue?

Proposed Fees applied by the Telecom company per day period	
Dia	$ 0,160
Tarde	$ 0,105

Table 8 - Proposed Fees

Adjustment in revenue:

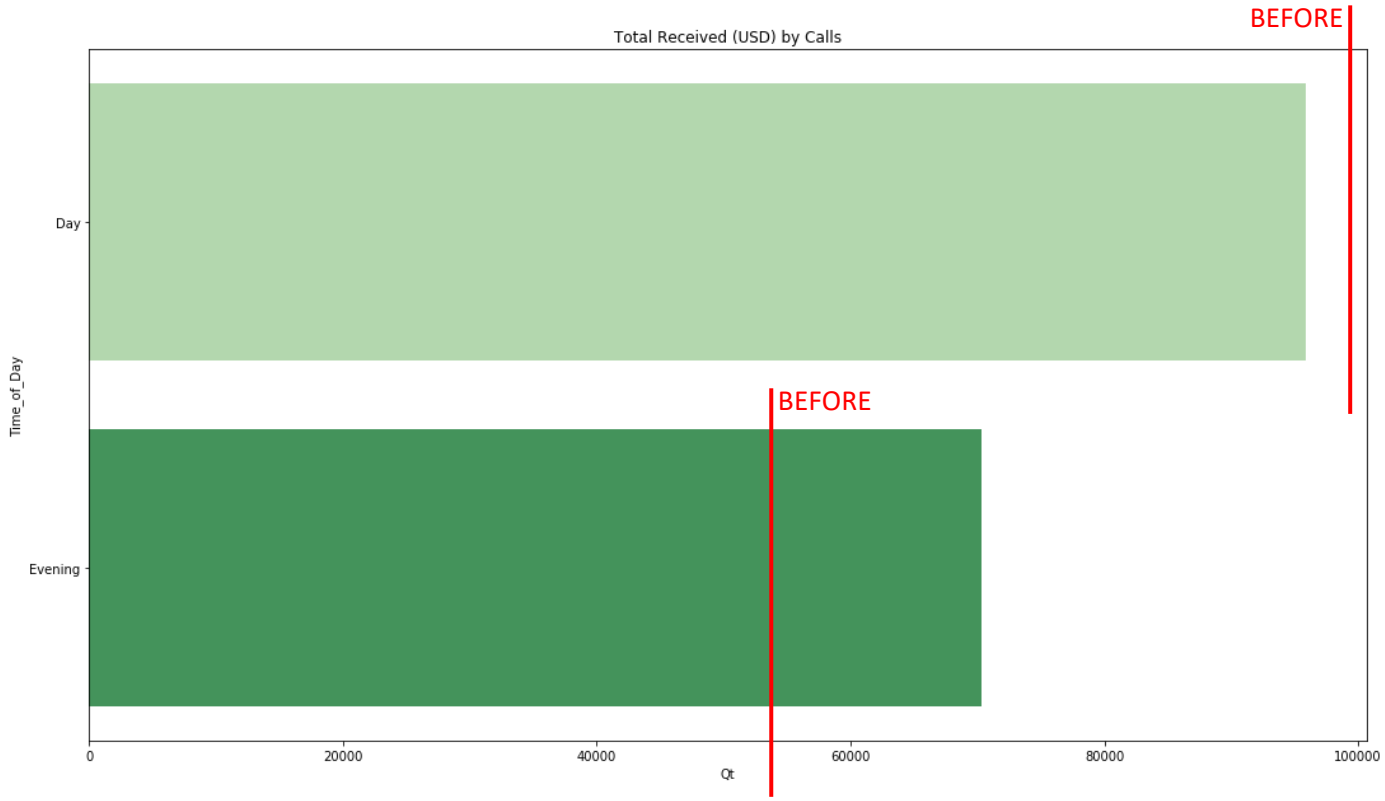

Figure 20 - Total received in USD (proposal 01)

With small rate adjustments we can go from $ 158,801.11 to:

New Revenue: $ 166,206.55

I understand that possibly the volume of calls in the afternoon is higher because the rate applied is 50% lower than the rate of the day, an attraction to increase the volume of calls in this period. So let's think differently: what if instead of changing the rate, marketing plans were made to encourage consumption during the day? Let's simulate an environment in which the marketing incentive has generated a 10% increase in consumption.

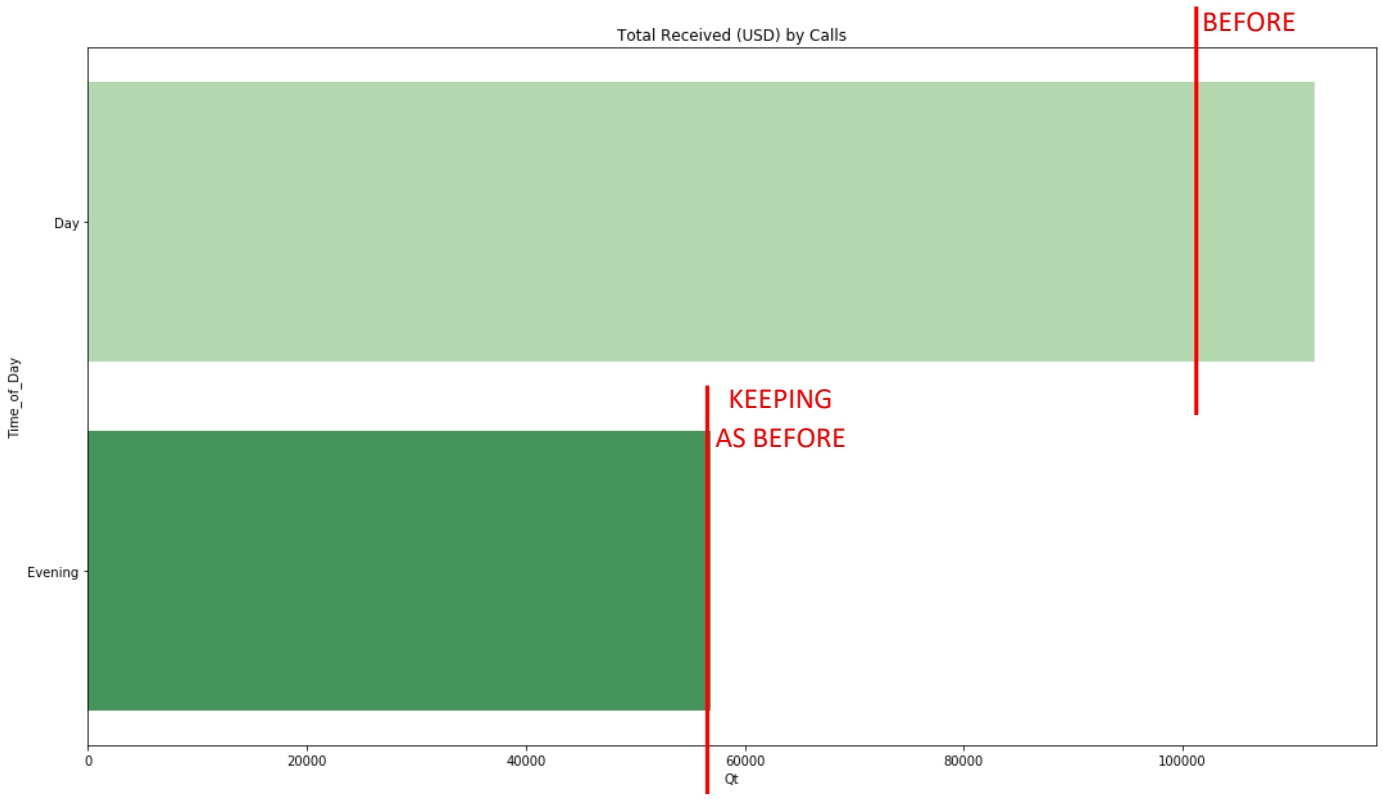

Figure 21 - Total received in USD (proposal 02)

With this production environment we would have a revenue of:

New Revenue: $ 168,987.34

Consolidating revenues according to each proposed environment:

Proposed revenues for the Telecom company		Difference
Original Revenue	$ 158.801,110	$ -
Proposed Revenue 01	$ 166.206,550	$ 7.405,440
Proposed Revenue 02	$ 168.987,340	$ 10.186,230

Table 9 - Revenue Differences

7.7 Fifth Business Opportunity - Customer Service

Let's now look at another source of information, how the customer interacts with the company's support. This is usually a time when the customer encounters a problem and needs help, a solution, it is the moment to retain customer loyalty but it is not always possible.

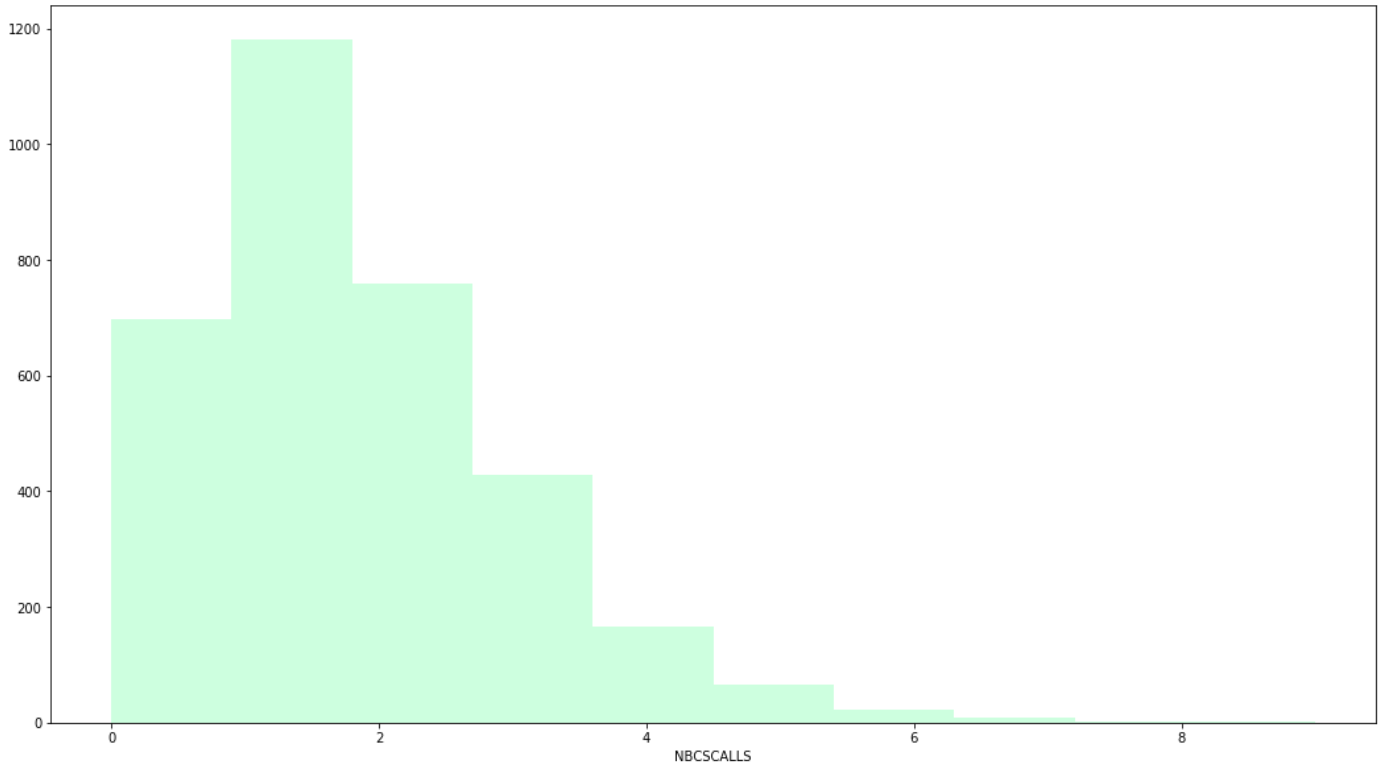

Figure 22 - Number of Customer Service Calls

The histogram above shows the number of calls that customers make to the Company's support. Note that we have a higher volume of calls between 1 and 3 calls and soon afterwards it drops considerably. This chart does not indicate cancellation, only the volume of calls.

Let's understand after how many calls customers decide to cancel the service.

The following chart compares two important pieces of information, the number of calls made to the Customer Service (x-axis) and the choice between canceling or not the service (y-axis).

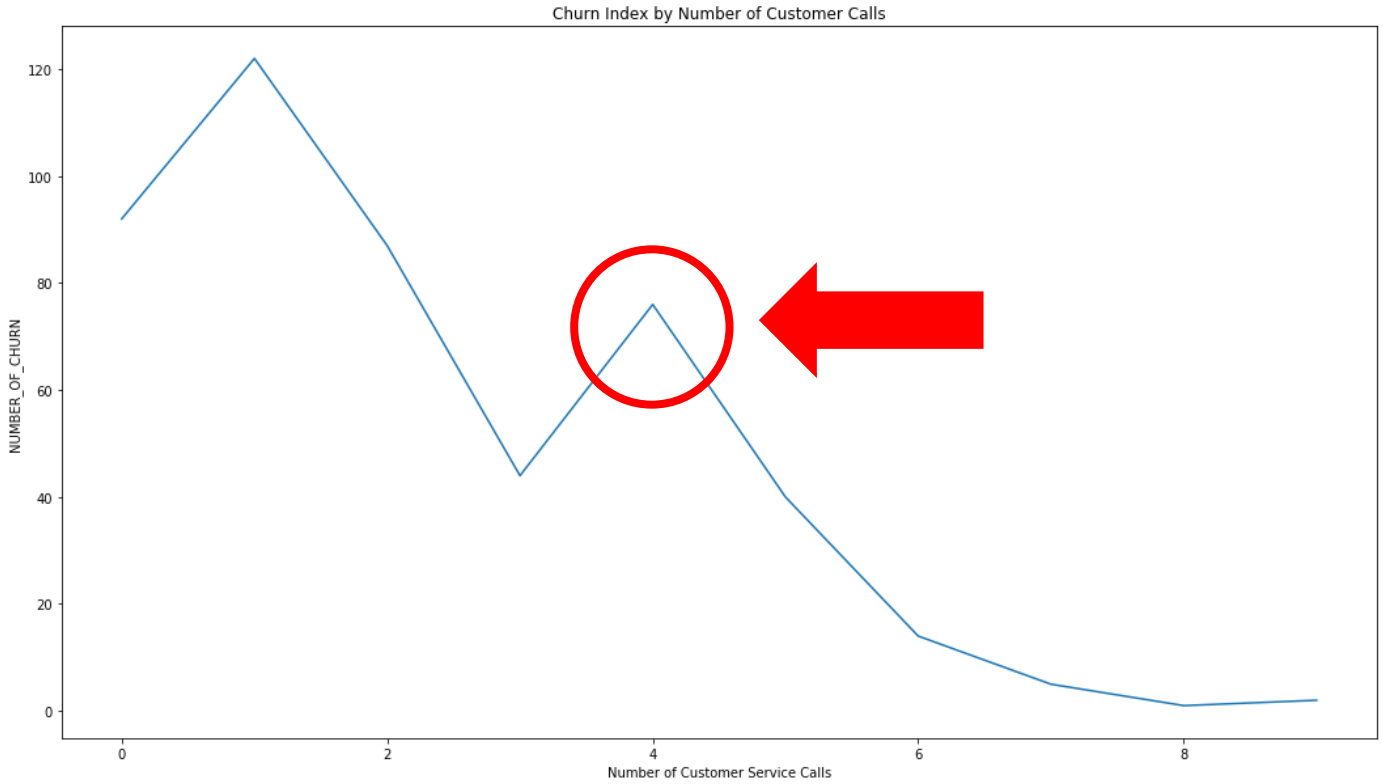

Figure 23 - Churn Index by Customer Service

Note that dissatisfied consumers cancel the service when they make the first call to the Customer Service and the tendency is for cancellations to decrease, however, a point of concern is that indicated in the red circle in figure 23, that is, four calls to the Customer Service because the chart was indicating a decrease in the churn rate and suddenly a peak accused cancellations.

Unfortunately we don't have enough information to ascertain the reason, I recommend to collect different data so that we can properly state the cause of the problem and try to mitigate the churn rate.

This is only part of the data analysis, an in-depth study can provide many other opportunities for improvement and identification of business opportunities.

Understanding how data behaves makes us more confident in building the predictive model, the subject of the next chapter.

During exploratory analysis we work more with SQL statements than with Spark functions, but we will now enter an important library responsible for machine learning models, Spark MLlib[25].

[25] https://spark.apache.org/mllib/

8. Building the Machine Learning Model

In this chapter, we will build two different machine learning models in order to analyze which one we get the most accuracy from and then present new and unknown data and then evaluate if we have a reliable model

Machine Learning is an area of artificial intelligence that studies learning techniques by applying programming and computing to build mathematical models with the ability to get knowledge automatically based on the data provided.

Through this data, the trained algorithm is able to make decisions when new data is presented to it, thus providing complex problem solving solutions that in many cases would be difficult for a person to perform.

Within this concept we have 2 types of learning:

- Supervised Learning
  In this type of learning, we have a set of input data and possible output data that should be used to train the model. In other words, by testing this model we can compare the predicted output data with that provided and evaluate the accuracy of the trained model. If unsatisfactory, it is possible to go back to the beginning of model creation and improve the predictor variables to perform new tests.

- Unsupervised Learning
  In this other type of learning, we have an input data set but the outputs are unknown, so I can't compare the predicted data with the expected outputs. For this learning modality other techniques are applied, for example, the data will be grouped and the results will change according to the variables.

Machine Learning can be used to solve a range of problems from internet search suggestions, spam message tracking, to digital marketing usage.

For our business problem, Real – Time Customer Churn Prediction in Telecom Companies, we will use supervised learning, but before we go into creating the predictive model, let's see how data is balanced.

8.1 Data Balancing

During the loading of the variables, I performed an analysis on the data set and obtained an interesting result: the data distribution of the predictor variable (churn) is totally unbalanced. This can be a big problem because it generates a biased model prioritizing one variable over another.

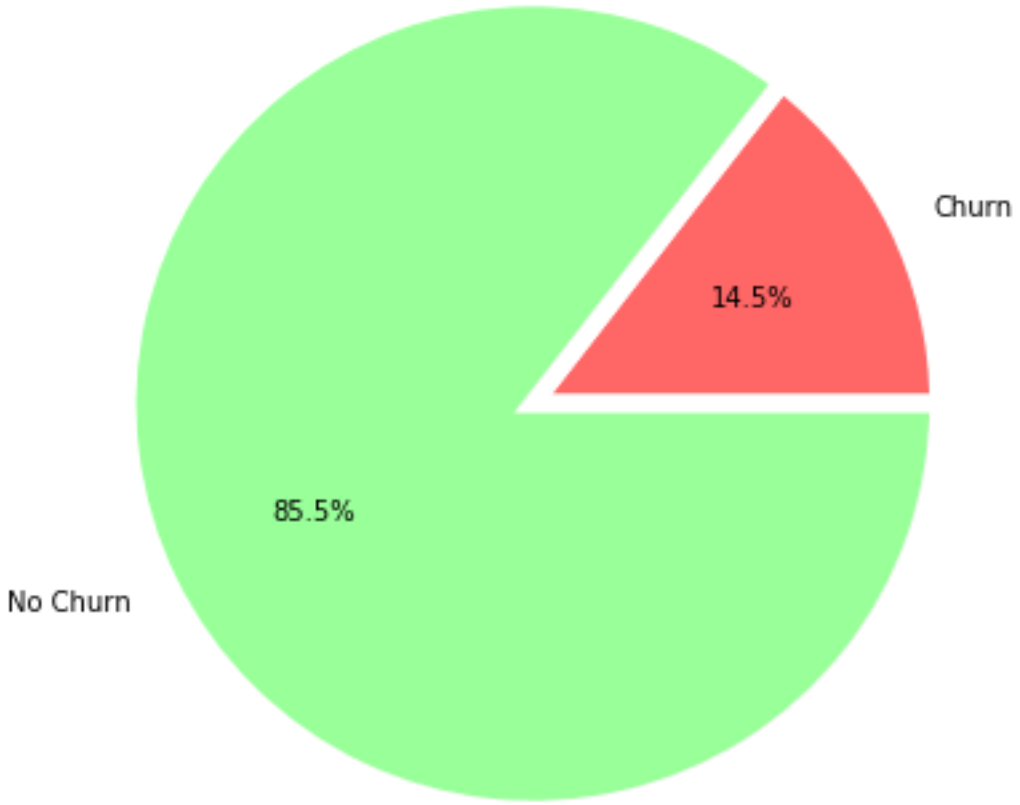

Figure 24 - Amount of Data (%)

We will have no problem creating the predictive model but we will have to observe the behavior of the machine learning model when presenting new and unknown data. In case of failure it will be necessary to adjust this balance and retrain the model.

8.2 Correlation and Variables of Importance

Studying the correlation between variables is an important source for understanding a problem and finding possible solutions. Finding the relevant variables can help improve the predictive model and bring valuable sources of information to the analysis process.

The correlation coefficient varies from -1 to 1 indicating that two variables have a strong negative or positive correlation, respectively.

With the aid of a "loop for" I ran through my dataset comparing each variable to the Churn variable using the following command:

```
for i in telecomDF4.columns:
    # for each one I'll 'select(i)', and use the 'take' action from [0][0] combination as if it was
      a matrix
    if not(isinstance(telecomDF4.select(i).take(1)[0][0], str)):
        print("Correlation between Target(Churn) with: ", i, telecomDF4.stat.corr('TARGET', i))
```

Resulting in:

```
Correlation between Target(Churn) with:  ID -0.0402316843047844
Correlation between Target(Churn) with:  ACCLGT -0.016540742243674304
Correlation between Target(Churn) with:  INTPLAN 0.25985184734548217
Correlation between Target(Churn) with:  VMPLAN -0.10214814067014684
Correlation between Target(Churn) with:  NBVMMSG 0.0897279698350642
Correlation between Target(Churn) with:  TTDAYMIN -0.2051508292613897
Correlation between Target(Churn) with:  TTDAYCALLS -0.01845931160857706
Correlation between Target(Churn) with:  TTDAYCHARGE -0.20515074317015394
Correlation between Target(Churn) with:  (TTDAYMIN / TTDAYCALLS) -0.152041247
Correlation between Target(Churn) with:  TTEVEMIN -0.09279579031259169
Correlation between Target(Churn) with:  TTEVECALLS -0.00923313191307793
Correlation between Target(Churn) with:  TTEVECHARGE -0.09278603942871404
Correlation between Target(Churn) with:  (TTEVEMIN / TTEVECALLS) -0.050302182
Correlation between Target(Churn) with:  TTINTMIN -0.06823877562717734
Correlation between Target(Churn) with:  TTINTCALLS 0.05284433577413784
Correlation between Target(Churn) with:  TTINTCHARGE -0.06825863150391472
Correlation between Target(Churn) with:  (TTINTMIN / TTINTCALLS) -0.085472967
Correlation between Target(Churn) with:  NBCSCALLS -0.20874999878379397
Correlation between Target(Churn) with:  TARGET 1.0
```

Table 10 – Correlation and Variables of Importance

The most important variables are:

- INTPLAN: Indicative if the client has an international plan
- NBVMMSG: Number of voice messages the customer has
- TTDAYMIN: Number of minutes used during the day
- TTDAYCHARGE: Total cost of calls made during the day
- (TTDAYMIN / TTDAYCALLS): Minutes per call
- TTEVEMIN: Number of minutes used in the afternoon
- TTEVECHARGE: Total cost of calls made during the afternoon
- NBCSCALLS: Number of calls to Customer Service

These correlation operations generated 39 different jobs for the cluster, each costing an average of 2 seconds to be processed, a job from the “take” action where we acquire the data for delivery to the second “corr” job that makes the correlation between the variables. Everything spelled out in Pyspark Shell.

8.3 Pré – Processing of the Dataset

To load these most important variables into the machine learning model, it is necessary to first understand an important concept of data delivery to Spark MLlib (MLlib is the abbreviation for Machine Learning Library), the concept of Dense Vector and Sparse Vector.

The need for some algorithms (mainly regression) for apache spark machine learning is that the data must be in a specific format to be able to train the algorithm.

For that, I need to pass the data in vector format, specifically using a dense or sparse vector, as shown in the image below, because Apache spark works with a cluster and then it will distribute the data between the machines.

Dense Vector:	9	4	0	0	0	2	0	0	9

Sparse Vector:	size:	9			
	indices:	0	1	5	8
	values:	9	4	2	9

Table 11 – Dense Vector x Sparse Vector

Conceptually they are the same object, just a vector. However sparse vectors are vectors that have many values like zero. While a dense vector is when most of the values in the vector are nonzero.

To build the vector I need to import the following library:

```
from pyspark.ml.linalg import Vectors
```

With the function "Vectors" I created the following function using the importance variables previously selected and the dense vector:

```
def transformFeatures(row):
    obj = (row['TARGET'], Vectors.dense(row['INTPLAN'], row['NBVMMSG'], row['TTDAY
      MIN'], row['TTDAYCHARGE'], row['(TTDAYMIN / TTDAYCALLS)'], row['TTEVEMIN'],
      row['TTEVECHARGE'], row['NBCSCALLS']) )
    return obj
```

Resulting in the following structure, the first column being the Target value:

```
[(2, DenseVector([2.0, 25.0, 265.1, 45.07, 2.41, 197.4, 16.78, 1.0])),
 (2, DenseVector([2.0, 26.0, 161.6, 27.47, 1.3138, 195.5, 16.62, 1.0])),
 (2, DenseVector([2.0, 0.0, 243.4, 41.38, 2.1351, 121.2, 10.3, 0.0])),
 (2, DenseVector([1.0, 0.0, 299.4, 50.9, 4.2169, 61.9, 5.26, 2.0])),
 (2, DenseVector([1.0, 0.0, 166.7, 28.34, 1.4752, 148.3, 12.61, 3.0]))]
```

Best viewed in table format:

TARGET	FEATURES
2	[2.0,25.0,265.1,4...]
2	[2.0,26.0,161.6,2...]
2	[2.0,0.0,243.4,41...]
2	[1.0,0.0,299.4,50...]
2	[1.0,0.0,166.7,28...]
2	[1.0,0.0,223.4,37...]
2	[2.0,24.0,218.2,3...]
2	[1.0,0.0,157.0,26...]
2	[2.0,0.0,184.5,31...]
2	[1.0,37.0,258.6,4...]

Table 12 - Target x Features

Now we are ready for Spark Machine Learning processes.

8.4 Machine Learning

8.5 Train/Test Split

Although the data is ready for delivery to the machine learning model, I will separate the dataset into training and test data following the proportion of 70% and 30% respectively.

Initially I will use the training data so that the model performs its mathematical transformation by learning patterns and delivering a final model, after this process I analyze with the test data if it has acceptable accuracy. Remembering that the test data is different and unknown to the model.

TARGET	Qt of Data
null	2358
1	348
2	2010

TARGET	Qt of Data
null	975
1	135
2	840

Table 13 - train x test

8.6 Chosen Machine Learning Models

Two classification models were chosen for training and evaluation:

- DecisionTreeClassifier;
- RandomForestClassifier.

Let's study how these machine learning models work.

8.6.1 DecisionTreeClassifier X RandomForestClassifier

With the selected variables we can train the predictive model, but a key point is to try to identify when the model enters the underfitting zone, when it encounters the smallest error (ideal value) and when it arrives in the overfitting zone. However, first let's understand some concepts about Decision Tree and randomForest.

The machine learning model chosen for our classification problem will be defined by the best accuracy provided by the model after presented test data. As the name suggests, randomForest means Random Forest, which in Data Science we can make analogy to Decision Trees where each tree has a depth and decides between its 'leaves' which is the best path to travel.

Imagine an inverted tree:

Figure 25 - DecisionTree

End nodes (or leaves) are at the bottom of the decision tree. This means that the decision trees are drawn upside down. Thus, the leaves are the bottom and the roots are the tops (figure above).

A Decision Tree works with both categorical and continuous variables and works by dividing the population (or sample) into subpopulations (two or more sets) based on the most significant divisors of the input variables. For this and many other reasons, decision trees are used in classification and regression problems where the supervised learning algorithm has a predefined target variable.

8.6.2 RandomForest Predictive Model x Underfitting x Overfitting

In randomForest, or Random Forest, we grow multiple trees instead of a single tree. But how does the classification process work? Initially for classifying a new attribute-based object, a tree generates a classification for that object (which is as if the tree gives votes for this class). This process goes on for each tree in the forest and finally, the forest chooses the classification with the most votes (from all trees in the forest). In case of regression, the average of the exits by different trees is considered.

To illustrate the process performed by randomForest, follow figure below:

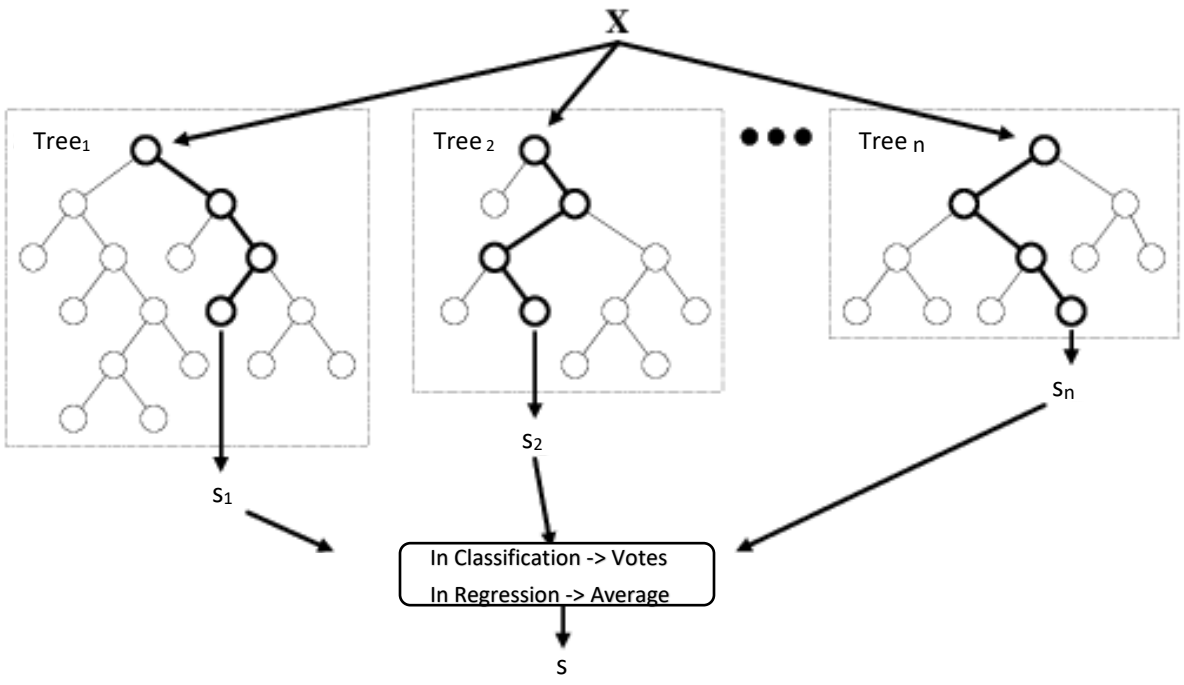

Figure 26 - RandomForest

As shown in figure 26, we can have n trees and each tree can have as many leaves as it wants. This is where we have a problem, because a shallow tree that has been trained to classify an object may not be accurate because it has learned little, or in other words underfitting. At the other extreme we have overfitting, that is, if no limit is set the model will give 100% accuracy in the training set because it ends up making a leaf for each observation.

I imagine the question now would be, “But isn't offering 100% accuracy good? " The answer is yes and no, because it is good to have accuracy, but here the accuracy is only in the training data and my goal is to generate an unbiased machine learning model where any data can be predicted. In case of overfitting when I present new data (which is the test data) the model will fail and return poor accuracy.

The following image illustrates the problem:

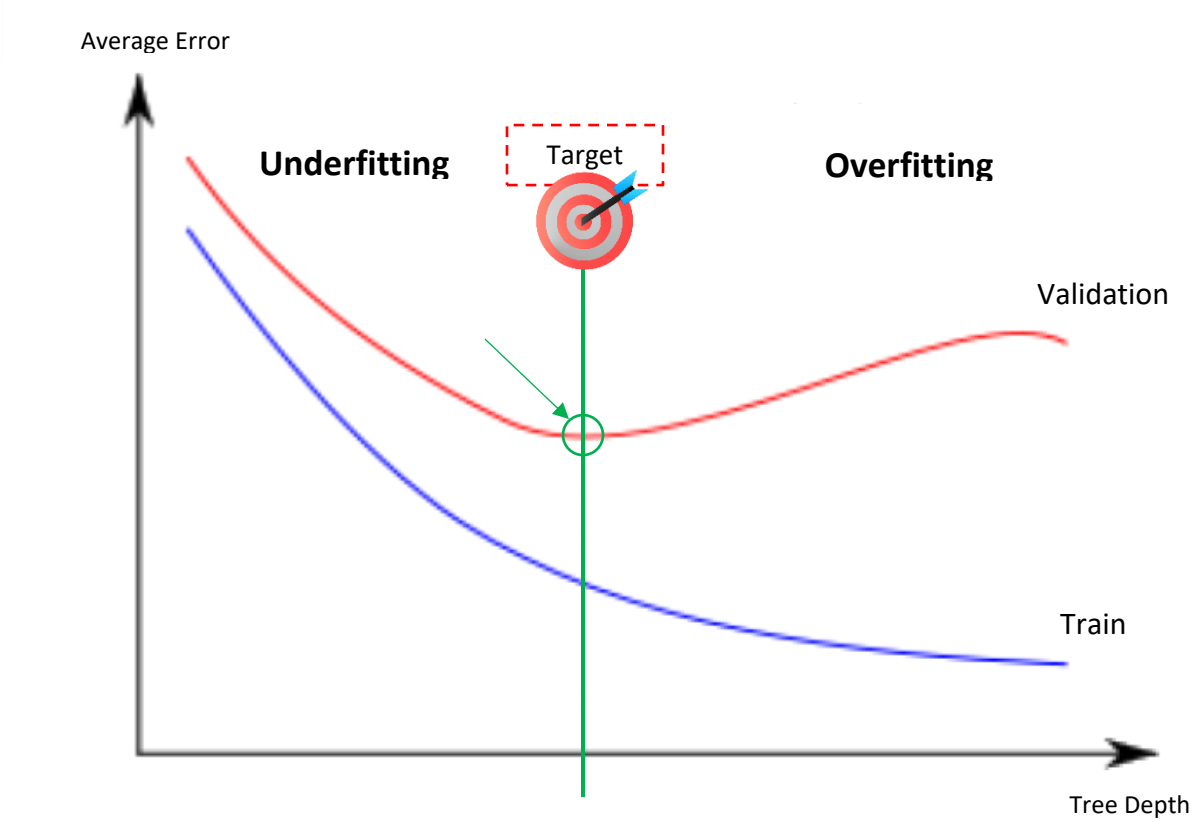

Figure 27 - Underfitting x Overfitting

In figure 27 we have 3 lines with different colors and each one represents important information:

- Blue Line: represents the average error that training data gives according to tree depth. The deeper the tree, the smaller the error as the training data was learned almost entirely.
- Red Line: represents the error in the test data (validation). Note that the error starts high, decreases, and then increases again. This is one of the key challenges faced when modeling decision trees, finding the optimal point where the error is as small as possible.
- Green Line: As you can see, the green line crosses at the ideal point, where the error in the test data (validation) is as little as possible, giving the model better accuracy.

Consolidating in the following figure the goal in performing predictive modeling controlling underfitting and overfitting:

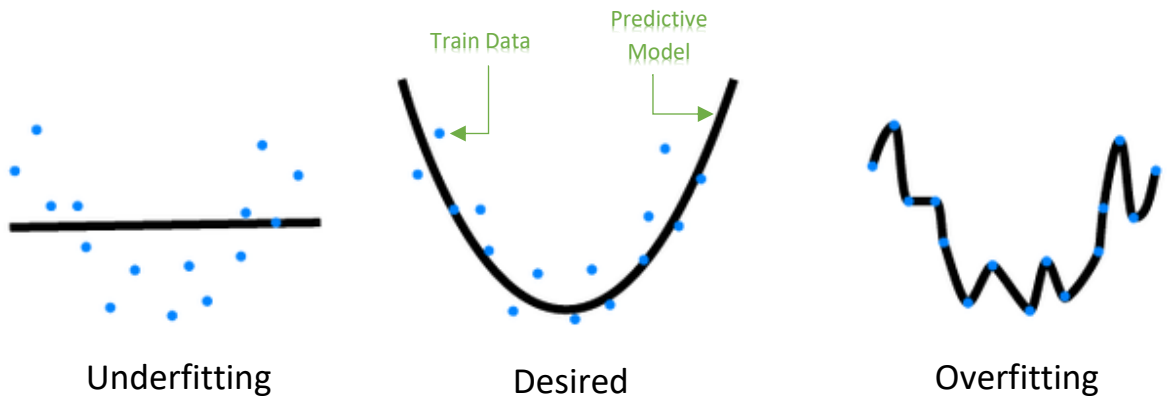

Figure 28 - Underfitting x Ideal x Overfitting

As shown above, what we want is that the model has an ideal curve avoiding poor accuracy, but also does not memorize 100% of the training data failing with new and unknown data.

8.7 Predictive model (DecisionTree) and the business problem

Back to the business problem of this project, so that the model presented does not suffer from underfitting or overfitting, a loop was created to train the different depths of the tree.

The line of code below will carry out the complete training, create the model, carry out the forecast and finally return the precision at each depth. This process will be repeated n times, where n indicates the depth range of the tree (1 - 15).

Follow code:

```
# Avoiding Underfitting and Overfitting in Decision Trees
for depth in range(1,15):
    model_v1 = DecisionTreeClassifier(maxDepth=depth, labelCol= 'TARGET', features
    Col= 'FEATURES')
    model_v1_train = model_v1.fit(train_data)
    model_v1_prediction = model_v1_train.transform(test_data)
    evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                                  labelCol = 'TARGET',
                                                  metricName = 'accuracy')
    acc = evaluator.evaluate(model_v1_prediction)
    print('In Depth %.0f this model is %.2f%% Accurate.' %(depth, acc*100))
```

Resulting in:

```
In Depth  1 this model is 86.97% Accurate.
In Depth  2 this model is 88.21% Accurate.
In Depth  3 this model is 90.46% Accurate.
In Depth  4 this model is 89.74% Accurate.
In Depth  5 this model is 91.49% Accurate.
In Depth  6 this model is 92.10% Accurate.
In Depth  7 this model is 90.67% Accurate.
In Depth  8 this model is 90.77% Accurate.
In Depth  9 this model is 89.85% Accurate.
In Depth 10 this model is 89.13% Accurate.
In Depth 11 this model is 88.92% Accurate.
In Depth 12 this model is 88.92% Accurate.
In Depth 13 this model is 88.82% Accurate.
In Depth 14 this model is 87.90% Accurate.
```

According to the model's response, depth equal to 6 offers better accuracy and avoids underfitting and overfitting.

Model 01 (DecisionTree) -> Precision of 92.10%

We will use this depth to continue carrying out the forecasts and evaluation.

8.7.1 Forecasting and Evaluating

With the model parameterized to offer the best accuracy, I presented unknown data to predict whether a customer is likely to cancel the service or not. Here is a part of this prediction made by the machine learning model:

```
[Row(prediction=2.0, TARGET=1),
 Row(prediction=1.0, TARGET=1),
 Row(prediction=2.0, TARGET=1),
 Row(prediction=1.0, TARGET=1),
 Row(prediction=2.0, TARGET=1),
 Row(prediction=2.0, TARGET=1),
 Row(prediction=2.0, TARGET=1),
 Row(prediction=1.0, TARGET=1),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=1.0, TARGET=2)]
```

Note that at times the model should predict "1" and predicted "2" indicating that the customer would not cancel the service when in fact he would. Let's build a Confusion Matrix and analyze all predictions.

In this case, Pyspark Shell presented more than 190 operations that were very fast in execution, lasting around 50 ms each. Here we can see how advantageous it is to bring spark to a big data project, as it performed machine learning processes in less than a second, that is, it created the model, delivered the trained model, performed tests with new data and presented the evaluation metric so that neither underfitting nor overfitting occurs. All in a few milliseconds of execution on all data in the cluster.

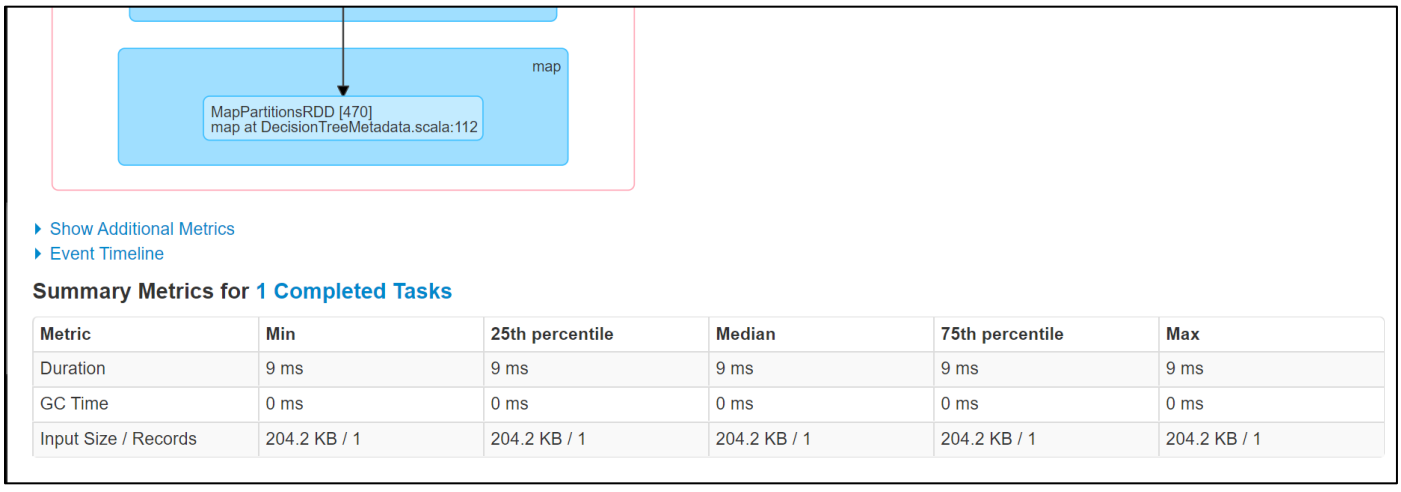

Figure 29 - DecisionTree Job

In figure 29 we can see that the last action of this job is the mapping in the machine learning model DecisionTree. The job was started and then ended 9 milliseconds later.

8.7.2 Confusion Matrix

Confusion Matrix is one of many performance gauges for evaluating data predicted by a machine learning model. We already know that the accuracy is close to 92% but what about the data that was not classified correctly? A Confusion Matrix can tell us what happened to all classified data, following image for a better understanding of its operation:

	1 (Predicted Data)	2 (Predicted Data)
1 (Original Data)	TruePositive	FalseNegative
2 (Original Data)	FalsePositive	TrueNegative

Figure 30 - Confusion Matrix Concept

Let's interpret what the Confusion Matrix is telling us:

- TruePositive(TP): It means that my model predicted from the data provided that the class would be 1 and it really is correct, it was supposed to be 1.
- FalseNegative(FN): It means that my model predicted from the data provided that the class would be 2 but was supposed to be 1.
- FalsePositive(FP): It means that my model predicted from the data provided that the class would be 1 but also missed because it was supposed to be 2.
- TrueNegative(TN): It means that my model predicted from the data provided that the class would be 2 and this time it was right because it was supposed to be 2.

Our goal is to predict and succeed so we seek to maximize the values present in TP and TN, and on the other hand mitigate values of FN and FP.

Our machine learning model returned the following results:

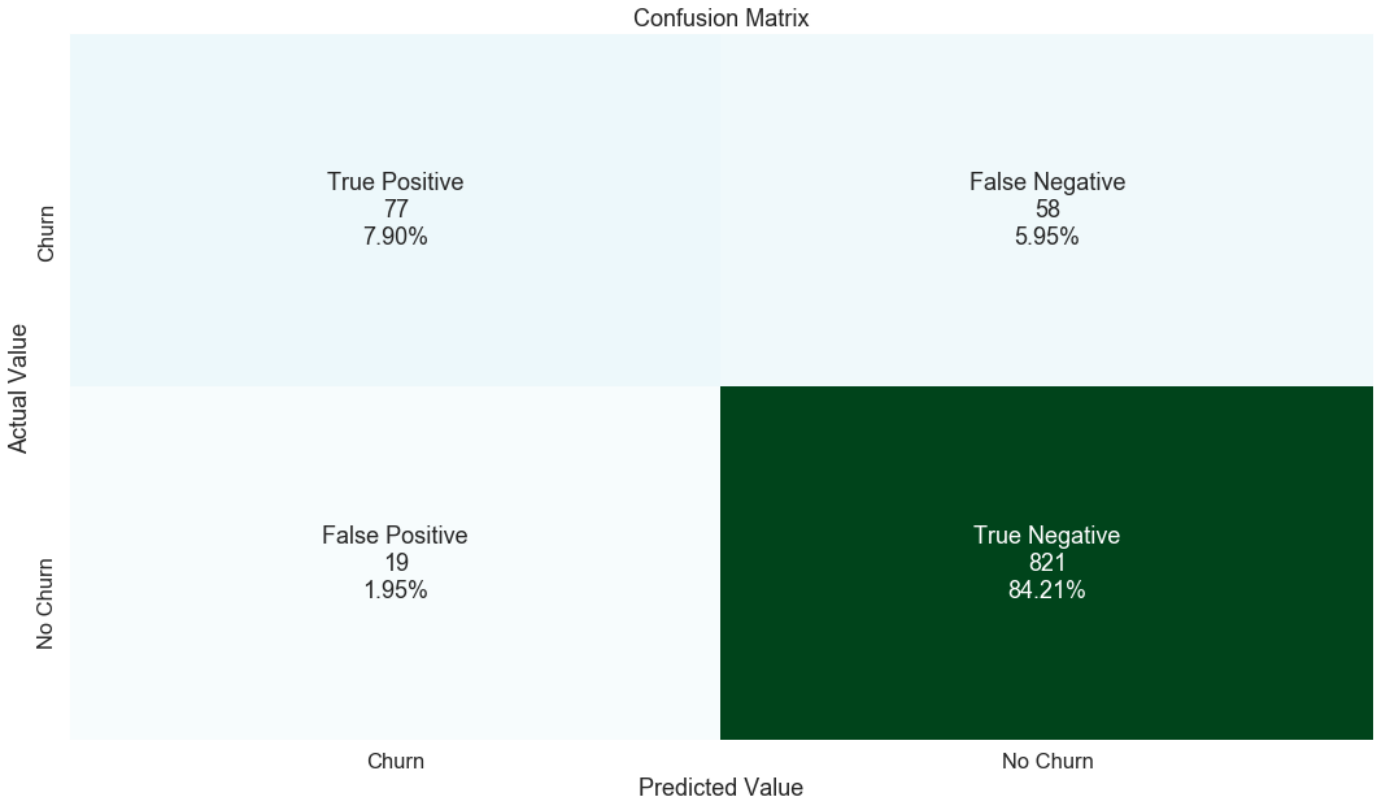

Figure 31 - Confusion Matrix I

Great results because TP and TN are the highest values among all the predicted data. It is worth mentioning that TN is much larger than the whole set foreseen, as explained at the beginning of chapter 7, we have unbalanced and prone to "No Churn" data. Later in this project we will see the reflection of having balanced data.

8.8 Predictive model (RandomForest) and the business problem

Using the same concept and processes as the previous model, that is, complete training, create the model, perform the forecast and finally return the precision we obtain:

Modelo 02 (RandomForest) -> Precisão de 90.56%

8.8.1 Forecasting and Evaluating

```
[Row(prediction=1.0, TARGET=1),
 Row(prediction=1.0, TARGET=1),
 Row(prediction=1.0, TARGET=1),
 Row(prediction=2.0, TARGET=1),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2),
 Row(prediction=2.0, TARGET=2)]
```

8.8.2 Confusion Matrix

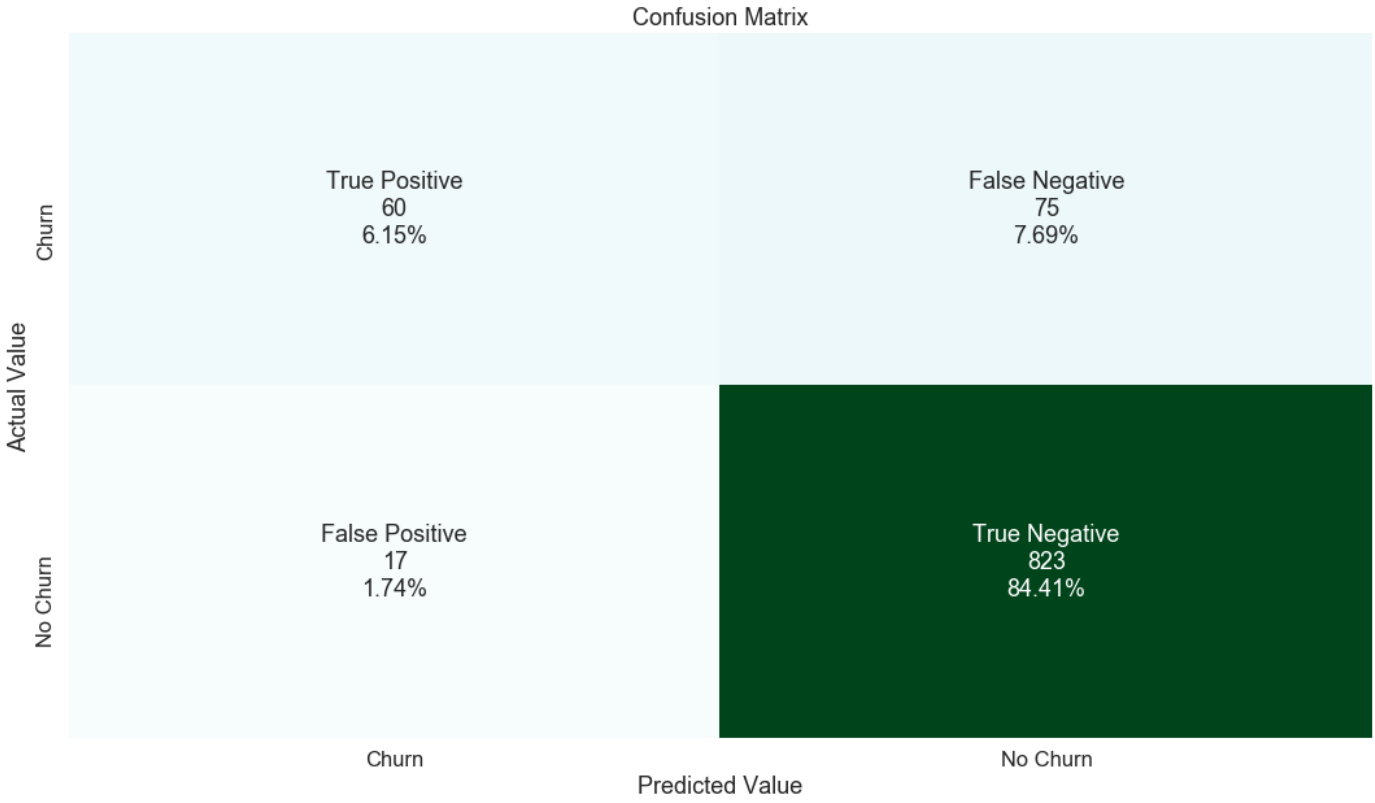

Figure 32 - Confusion Matrix II

Based on the results of the two models, we can choose the one that presented the best performance. The first model had better accuracy and delivered better True Positive and True Negative, while the second was better with True Negative but unfortunately it was precarious with False Negative surpassing even True Positive.

Our choice would be the first model, the Decision Tree mode.

As from now on the tasks will be repeated only by changing concepts and testing hypotheses, Spark Jobs will be the same too so I will not load this project with more information from Pyspark Shell. If you want to track each execution of the Jobs enter the address of the localhost cluster and view the processing details. I will return with Pyspark Shell at the end of the project presenting the consolidated results.

9. Optimizing the Result

In a production environment, that is, a real environment, to present a better result we could find ways to reduce the False Negative since it is a result where the customer is prone to cancel the service but our predictive model delivered that it is not a potential customer who is going to cancel. In other words, it predicted "No Churn" when it was "Churn".

Considering this problem, we will see the effect of having balanced data in a simulated environment with approximately 50% “Churn” and 50% “No Churn”, a machine learning model will be created, testing this model, assessing accuracy, presenting new data and finally plotting the confusion matrix evaluating the hypothesis that balanced data offers a better machine learning model.

9.1 New Data Balancing

Considering a real environment I recommend to acquire balanced data to avoid bias when creating the machine learning model, however it is not always possible to have balanced data.

So far we have:

- 85.5% of “No Churn” data;
- 14.5% of “Churn” data.

For the simulated environment I recovered the data and separated 50/50 according to the pie chart below:

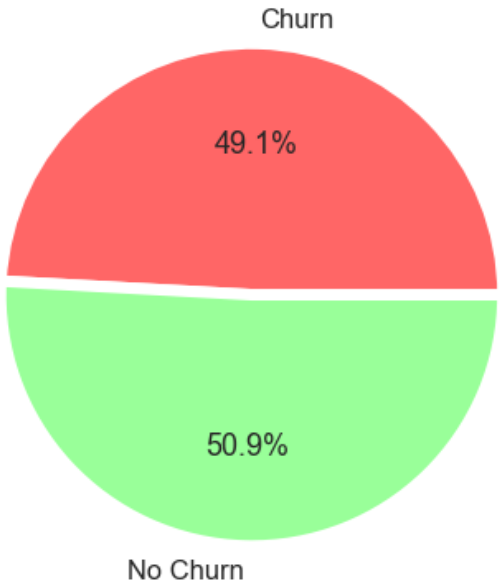

Figure 33 - Data Balancing

As the processes are the same in the following chapters, I will focus on the results.

9.2 Machine Learning

With the chosen machine learning model, DecisionTreeClassifier, I presented new balanced data to train and generate the best accuracy without underfitting or overfitting:

```
In Depth  1 this model is 62.45% Accurate.
In Depth  2 this model is 76.17% Accurate.
In Depth  3 this model is 81.59% Accurate.
In Depth  4 this model is 83.39% Accurate.
In Depth  5 this model is 84.84% Accurate.
In Depth  6 this model is 83.03% Accurate.
In Depth  7 this model is 80.51% Accurate.
In Depth  8 this model is 80.87% Accurate.
In Depth  9 this model is 81.23% Accurate.
In Depth 10 this model is 80.87% Accurate.
In Depth 11 this model is 80.87% Accurate.
In Depth 12 this model is 79.78% Accurate.
In Depth 13 this model is 79.06% Accurate.
In Depth 14 this model is 79.42% Accurate.
```

Unlike the environment with unbalanced data, here the overall accuracy is lower:

Balanced Model -> Accuracy of 84.84%

Despite this, we will continue and present new data aiming to reduce the False Negative.

9.3 New and Unknown Data

We have not yet used dataset 2.2 presented in chapter 2, data for testing. This dataset has the following balance:

- 86.6% of “No Churn” data;
- 13.4% of “Churn” data.

Note that unbalanced data will be presented for the model to forecast, but the model was trained in balanced data which generates an advantage in the forecast because the model knows more about “Churn” and “No Churn” and not just one of the variables.

Therefore, using the dataset presented in chapter 2, data for testing, I fed the balanced model and obtained general accuracy of:

Balanced Model -> Accuracy of 87.28%

9.4 Confusion Matrix

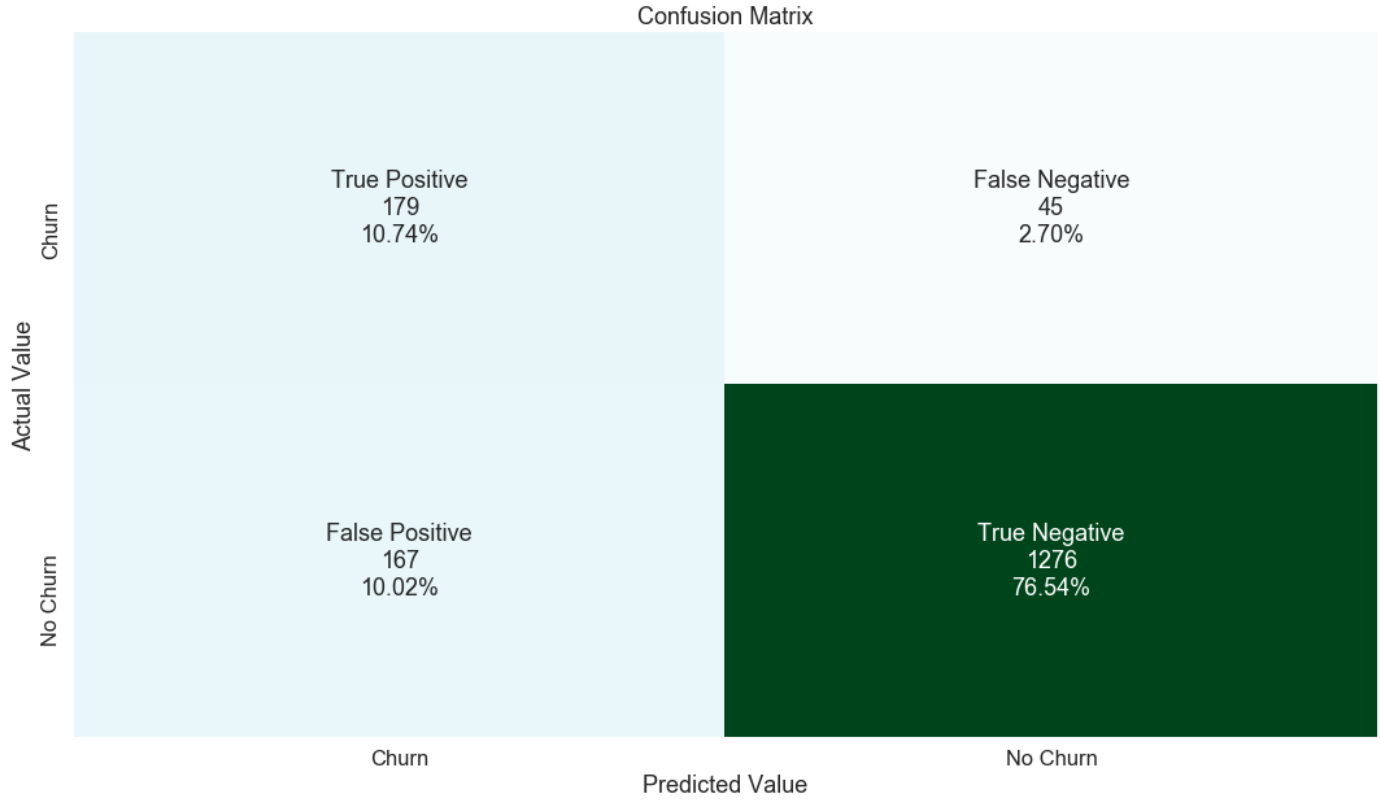

Figure 34 - Confusion Matrix III

This is our final result. Despite a lower overall accuracy, we managed to mitigate the incidence of False Negative by dropping to 2.70%.

It is worth noting that False Positive has also increased, but it is not identified as a problem since they are customers with no propensity to cancel but identified as possible cancellations. These customers will be contacted by the Telecom team to make them even more loyal.

Also note that the accuracy of True Positive increased to more than 10%, that is, only 45 out of 224 unknown values were wrong.

In this way, the hypothesis that balanced data are more viable for predictions is proven in practice because the machine learning model will be trained without partiality. In a real environment, considering balance can bring more reliability to the action process of the teams involved.

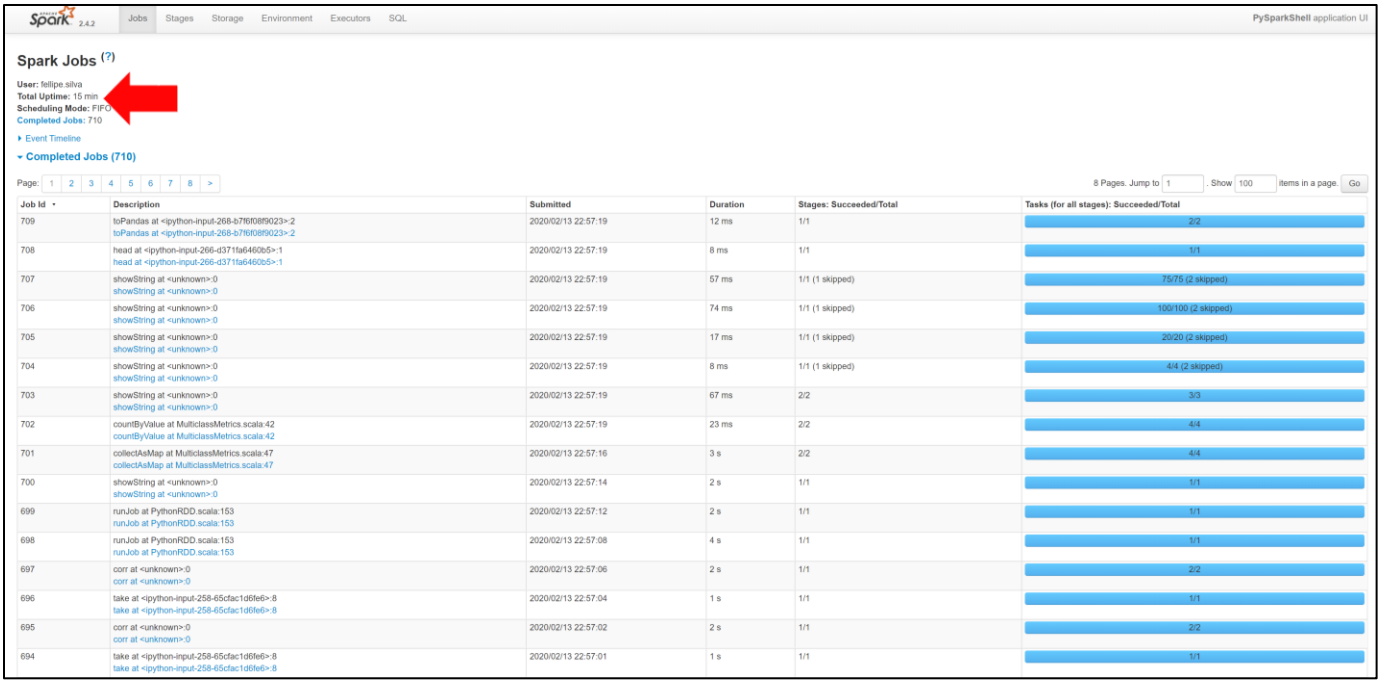

Figure 35 - PySpark Shell Final

Figure 35 illustrates the end of Jobs running on Spark. Note that 709 Jobs were generated and the total time to run the entire script is 15 minutes.

On a production scale, that is, in a real environment this time would certainly decrease because:

1. Not all the script would be executed as it is at the moment because only the chosen predictive model would be used;
2. Balancing data would already be acquired in the initial collection, even before entering the script, in other words, the ETL process (Extract - Transform - Load) could feed a datalake with the standards defined in the project;
3. Third and most importantly, we would have an entire cluster with more machines at our disposal for distributed data storage with HDFS and consequently distributed processing with spark increasing our computational capacity.

10. Final Considerations

The use of Spark in this project was essential for the distributed processing of large amount of data. It was possible to identify that the agility and flexibility in data manipulation with Spark SQL, Spark MLlib, Pyspark libraries, among others presented, brings a competitive advantage to the project of Real – Time Customer Churn Prediction in Telecom Companies.

With Pyspark Shell we observed several Jobs in operation, another advantage for the Data Science process because with it, if a possible error in the execution of the code occurs, we can find more quickly what the failure was and the causes of the problem, in addition to following each instruction in detail.

Fundamental concepts were also presented in the data science process, such as feature engineering, cleaning and transforming data, generating better opportunities for machine learning models; exploratory analysis delivering a little more value with the information implicit in the data; identification of the most relevant variables for the predictive model; building the model and assessing its accuracy.

In the exploratory analysis phase, we observed a series of business opportunities, all generated from the information contained in the data. A set of unordered numbers does not indicate much, but a set of structured data generates content and consequently value for the company.

The statistical analysis of the data balance brought an interesting question: do balanced data generate better results or not? Nothing better than using the data to prove this hypothesis in practice.

Initially, unbalanced data were used to train and test the machine learning model, where we obtained excellent accuracy, 92.10%. With the same unbalanced data, we trained and tested another model reaching an accuracy of 90.56%. These are great results, but the Confusion Matrix brought a worrying fact: False Negative portion has a considerably high value for this business problem.

Knowing this and in order to test our balanced data hypothesis, we entered a simulated environment where 50% of the data was “Churn” and 50% of the data was “No Churn”. Again, we created the predictive model, trained, tested and obtained an accuracy of 87.28%. It is not a better general accuracy than with unbalanced data, however, by analyzing the accuracy individually through the Confusion Matrix we can see that we mitigate the False Negative, dropping to 2.70% error. In a real environment we would have to analyze the best way to select the best model together with decision makers as they are responsible for leading teams and generating results for the company.

To further improve this result, it would be possible to focus more intensively on improving the model parameters, or also add new variables at the time of feature engineering, or even collect more data focusing on the analyzed resources to train the model.

Source code

```
## *** Attention: ***
## Use Java JDK 11 and Apache Spark 2.4.2

## If an error message "name 'sc' is not defined" appears, stop pyspark and delete the folder
metastore_db at the same folder where Jupyter notebook is set.

## Access http://localhost:4040 whenever you want to follow the jobs excecution

## Predicting Customer Churn in Operations from Telecom -----

## Directory ----------------------------------------------------------------------------------
# SET WORKING DIRECTORY
#"C:/FCD/BigDataRealTimeAnalyticPythonSpark/Cap12/Telecom"

## Kaggle -------------------------------------------------------------------------------------
# https://www.kaggle.com/c/churn-analytics-bda/overview

## Data Description ---------------------------------------------------------------------------
# - train.csv - the training set
# - test.csv - the test set

## Evaluation ---------------------------------------------------------------------------------
#The evaluation is based on misclassification error matrix
#The file should contain a header and have the following format:
# Id,Churn
#  1,True
#  2,False
#  3,True
#  4,False
# etc.

## Data Dictionary ----------------------------------------------------------------------------
# 1.State -> North American Abbreviated States (HI - Hawaii / MT - Montana / OH - Ohio)
# 2.Account_Length ->
# 3.Area_Code -> Phone Number Area Code
# ?.Phone_No -> Phone Number (IT IS NOT IN THE test NOR IN THE train DATA)
# 4.International_Plan -> "yes" or "no"
# 5.Voice_Mail_Plan -> "yes" or "no"
# 6.No_Vmail_Messages -> Number of Voice Mail Messages
# 7.Total_Day_minutes ->
# 8.Total_Day_Calls ->
```

```
# 9.Total_Day_charge ->
# 10.Total_Eve_Minutes ->
# 11.Total_Eve_Calls ->
# 12.Total_Eve_Charge ->
# 13.Total_Night_Minutes ->
# 14.Total_Night_Calls ->
# 15.Total_Night_Charge ->
# 16.Total_Intl_Minutes ->
# 17.Total_Intl_Calls ->
# 18.Total_Intl_Charge ->
# 19.No_CS_Calls -> Number_Customer_Service_Calls
# 20.Churn -> Target ("yes" or "no")

# Note that most information are encoded.
```

Important Libraries

```
## Imports - Libraries ----------------------------------------------------------------------------------
# IMPORTING NECESSARY LIBRARIES
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql.types import *
```

Creating a Spark Session to work with Dataframes

```
# Spark Session - used to work with Dataframes on Spark
spSession = SparkSession.builder.master("local").appName("Fellipe-Silva-DS").getOrCreate()
```

Loading Data

```
## Datasets ---------------------------------------------------------------------------------------
# Loading data and getting an RDD
# textFile CREATES AN RDD
telecomRDD = sc.textFile("projeto4_telecom_treino.csv")

# Optimizing performance using cache on RDD
telecomRDD.cache()

# Making an Action
telecomRDD.count()

# Making another Action
telecomRDD.take(5)
# Note that the data is between simple aspects '' (each line), indicating they are strings, so I'll convert it
further using a function I'll create
```

Feature Engineering

```
# Cleaning data - Removing first line
telecomRDD2 = telecomRDD.filter(lambda x: "state" not in x)
telecomRDD2.count()

# Looking Again at my data
telecomRDD2.take(5)

# Looking to ALL data and trying to figure out if there is missing values (?, NaN, "", ...)
telecomRDD2.collect()
# No Missing Values
```

Preparing Data to Dataframe

```
# Feature Engineering I - Function
def featengI(data):

    # Dividing data into columns, separating by "," character
    dataList = data.split(",")

    ID = (dataList[0])              # ID
    STATE = str(dataList[1])        # State -> North American Abbreviated States (HI - Hawaii / MT -
Montana / OH - Ohio)
    ACCLGT = int(dataList[2])       # Account_Length
    AREACODE = (dataList[3])        # Phone Number Area Code
    #PHONENB = (dataList[4])        # Phone_No -> Phone Number (IT IS NOT IN THE test-train DATA)
    #INTPLAN = (dataList[4])        # International_Plan -> "yes" or "no" (Convert yes->1 or no->2)
    INTPLAN = 1.0 if dataList[4] == '"yes"' else 2.0
    #VMPLAN = (dataList[5])         # Voice_Mail_Plan -> "yes" or "no" (Convert yes->1 or no->2) I THINK
THIS FEATURE IS NOT HELPING AND CAN BE ELIMINATED BECAUSE THE NEXT VARIABLE (NBVMMSG)
ONLY HAS VALUES WHEN THIS IS '1', IT'S LIKE DOUBLE INFORMATION ABOUT THE SAME THING
    VMPLAN = 1.0 if dataList[5] == '"yes"' else 2.0
    NBVMMSG = int(dataList[6])      # Number_Voice_Mail_Messages
    TTDAYMIN = float(dataList[7])   # Total_Day_minutes
    TTDAYCALLS = int(dataList[8])   # Total_Day_Calls
    TTDAYCHARGE = float(dataList[9])  # Total_Day_Charge
    TTEVEMIN = float(dataList[10])   # Total_Eve_Minutes
    TTEVECALLS = int(dataList[11])   # Total_Eve_Calls
    TTEVECHARGE = float(dataList[12]) # Total_Eve_Charge
    TTNGTMIN = float(dataList[13])   # Total_Night_Minutes
    TTNGTCALLS = int(dataList[14])   # Total_Night_Calls
    TTNGTCHARGE = float(dataList[15]) # Total_Night_Charge
    TTINTMIN = float(dataList[16])   # Total_Intl_Minutes
    TTINTCALLS = int(dataList[17])   # Total_Intl_Calls
```

```
    TTINTCHARGE = float(dataList[18]) # Total_Intl_Charge
    NBCSCALLS = int(dataList[19])    # Number_Customer_Service_Calls
    #TARGET = (dataList[20])        # Churn (Target) -> "yes" or "no" (Convert yes->1 or no->2)
    TARGET = 1 if dataList[20] == '"yes"' else 2

    # Creating 'lines' using the 'Row' function, preparing to the dataframe analysis, cleaning and
converting the data from string to float
    lines = Row(ID = ID, STATE = STATE, ACCLGT = ACCLGT, AREACODE = AREACODE,
                INTPLAN = INTPLAN, VMPLAN = VMPLAN, NBVMMSG = NBVMMSG,
                TTDAYMIN = TTDAYMIN, TTDAYCALLS = TTDAYCALLS, TTDAYCHARGE = TTDAYCHARGE,
                TTEVEMIN = TTEVEMIN, TTEVECALLS = TTEVECALLS, TTEVECHARGE = TTEVECHARGE,
                TTNGTMIN = TTNGTMIN, TTNGTCALLS = TTNGTCALLS, TTNGTCHARGE = TTNGTCHARGE,
                TTINTMIN = TTINTMIN, TTINTCALLS = TTINTCALLS, TTINTCHARGE = TTINTCHARGE,
                NBCSCALLS = NBCSCALLS, TARGET = TARGET)

    return lines

# Applying function above to RDD without headers
telecomRDD3 = telecomRDD2.map(featengl)

# Looking at the result
telecomRDD3.cache
telecomRDD3.take(1)

# Some data are still in String format, I'll convert to Integer or Float as needed
```

Creating a Pyspark Dataframe

```
# Creating a Dataframe SO I'M ABLE TO USE THE select FUNCTION FROM SparkSQL
telecomDF = spSession.createDataFrame(telecomRDD3)

type(telecomDF)

# Printing telecomDF Object Type and each Row Type
print(telecomDF)
# As noted SOME data (ID, STATE) are in String format, I'll convert to Integer or Float as needed

telecomDF.printSchema()

telecomDF.show(5)
```

Removing Double Quotes in Strings

```
# Function to Remove DoubleQuote (DQ) so I'll transform from String to Integer
def removingDQ(stringData):
    return stringData.replace('"', "")

# Using udf -> User Defined Function (Turning Python Functions into PySpark Functions)
```

```
# Look at this -> https://changhsinlee.com/pyspark-udf/
from pyspark.sql.functions import udf
udf_removingDQ = udf(removingDQ, StringType())

telecomDF2 = telecomDF.withColumn('ID', udf_removingDQ(telecomDF['ID'])).withColumn('STATE',
udf_removingDQ(telecomDF['STATE']))
telecomDF2.show(30)

# Now that necessary data doesn't have DQ I'm able to perform casting to transform from one type to
another
# As follows there are 2 ways to make this transformation
```

Transforming Features (Two Possibilities)

First

```
# 01 - First Possibility

# Listing all possible types
from pyspark.sql import types
for tp in ['BinaryType', 'BooleanType', 'ByteType', 'DateType',
        'DecimalType', 'DoubleType', 'FloatType', 'IntegerType',
        'LongType', 'ShortType', 'StringType', 'TimestampType']:
   print(f"{tp}: {getattr(types, tp)().simpleString()}")

# Now casting to IntegerType
from pyspark.sql.functions import col , column
telecomDF3 = telecomDF2.withColumn("ID", col("ID").cast("int"))
# The First "ID" refers to the column I'm picking in the telecomDF2
# The Second 'ID' in col("ID") refers to the column I'm picking in the telecomDF2 to cast
# cast("int") is the type transformation I want

telecomDF3.select("ID").show()

print(telecomDF3)

telecomDF3.printSchema()
# Transformation Done!
```

Second

```
# 02 - Second Possibility
telecomDF3 = telecomDF2.select( telecomDF2["ID"],
                        telecomDF2["ID"].cast(IntegerType()).alias("HI"))
telecomDF3.show()
# First I choose the column in telecomDF2 that I want to cast (telecomDF2["ID"])
# Second I transform to the type I want (cast(IntegerType()))
# Third I'm able to rename this column (alias("HI")). Further I chose to use the same names.
```

```
print(telecomDF3)

telecomDF3.printSchema()
# Transformation Done!
```

My Choice

```
# I'll choose the second one and add some features based in others
telecomDF4 = telecomDF2.select( telecomDF2['ID'].cast(IntegerType()).alias('ID'),
                        telecomDF2['STATE'],
                        telecomDF2['ACCLGT'],
                        telecomDF2['INTPLAN'],
                        telecomDF2['VMPLAN'],
                        telecomDF2['NBVMMSG'],
                        telecomDF2['TTDAYMIN'],
                        telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTDAYCHARGE'],
                        telecomDF2['TTDAYMIN']/telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTEVEMIN'],
                        telecomDF2['TTEVECALLS'],
                        telecomDF2['TTEVECHARGE'],
                        telecomDF2['TTEVEMIN']/telecomDF2['TTEVECALLS'],
                        telecomDF2['TTINTMIN'],
                        telecomDF2['TTINTCALLS'],
                        telecomDF2['TTINTCHARGE'],
                        telecomDF2['TTINTMIN']/telecomDF2['TTINTCALLS'],
                        telecomDF2['NBCSCALLS'],
                        telecomDF2['TARGET']
                        )
telecomDF4.show(3)

telecomDF4.printSchema()
```

Exploratory Analysis

```
# Exploratory Analysis
# Here I'll compare in some cases both Pyspark SQL Functions and SQL Functions

# 01.Pyspark SQL Functions -> https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html
# 02.SQL Functions -> http://www-
db.deis.unibo.it/courses/TW/DOCS/w3schools/sql/sql_func_count.asp.html

# Attention: standard queries SQL ANSI just work on JAVA 8

!java -version
```

States with more Data

```
# STATES WITH MORE DATA

# 01.Pyspark SQL Functions
from pyspark.sql.functions import *

telecomDF4.cube('TARGET').agg(countDistinct('STATE').alias('Exclusive States'),
                    grouping('TARGET')).orderBy('TARGET').show()

# 02.SQL Functions
# Registering the dataframe as a Temp Table (so i'll be able to use queries SQL ANSI)
# HERE I'M CREATING A REAL TABLE, IT WILL EXIST IN MEMORY, NOT ONLY THE DATAFRAME
telecomDF4.createOrReplaceTempView("telecomTB")

# ATTENTIOOOOON -> THIS INSTRUCTION ONLY WORKS WITH JAVA 8, AND NOT WITH JAVA 11
# Running queries SQL ANSI
spSession.sql("select TARGET, count(distinct STATE) as Exclusive_States from telecomTB group by
TARGET").show()
```

Attendance per State

```
# Attendance per State

# 02.SQL Functions
pysparkSqlDF = spSession.sql("select STATE, count(ID) as Number_Of_Attendance \
                    from telecomTB \
                    group by STATE \
                    order by Number_Of_Attendance DESC")
pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = pysparkSqlDF.toPandas()

pandasDF.head(6)

# What is the difference?
print(type(pysparkSqlDF))
print(type(pandasDF))

# Both are DataFrames but 'pandasDF' enables me to perform Pandas Functions

# Plotting some analysis using Seaborn Library
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# Setting width and height of the plot-figure
```

```
plt.figure(figsize=(18,10))

# Barplot showing amount of States being analyzed
sns.barplot(x=pandasDF['STATE'], y=pandasDF.Number_Of_Attendance, palette='YlGnBu_r')
plt.title("Most States")

# Trying to plot maps
# https://pythonprogramming.net/basemap-python-plotting-tutorial-part-5/

# As observed in the plot above:
#  Most Attendance: WV->West Virginia
#  Less Attendance:CA->Califonia

#!pip install wordcloud

from wordcloud import WordCloud
text = pandasDF['STATE']
wc = WordCloud(width=500, height=500, max_font_size=500, min_font_size=50, max_words=20,
background_color='#ffffff', colormap='winter').generate(text.to_string())
plt.imshow(wc, interpolation='bilinear')
plt.axis('off')
plt.margins(x=0, y=0)
plt.show()
```

Global Map of Attendance per State

```
# Necessary ALL the time to work with basemap package
import os

# On Office PC
#os.environ['PROJ_LIB'] = 'C:/Users/fellipe.silva/AppData/Local/Continuum/anaconda3/Lib/site-
packages/mpl_toolkits/basemap/data'
# On Home PC
os.environ['PROJ_LIB'] = 'C:/Users/fellipe.silva/Anaconda3/Lib/site-
packages/mpl_toolkits/basemap/data'

# Reading Data I Built with Latitude and Longitude values of States into a Dataframe

import pandas as pd
# read the data
USA = pd.read_csv('12B-States.csv', sep=",")
USA.head(5)

# Collecting only the columns needed
data = USA[['abv', 'latitude', 'longitude']]
```

```
# Factorizing the states abv to use as colors inside the global map plot. This makes another column
data['label_color'] = pd.factorize(data['abv'])[0]

# Renaming abv to STATE so I'll merge with my pandasDF by this column
data.columns=['STATE', 'latitude', 'longitude', 'label_color']

# Looking at the results
data.tail(3)

# Lokking at the Dataframe I want to merge
pandasDF.head(3)

# Merging (left) pandasDF and the USA long/lat data
mergedDF = pd.merge(left= pandasDF, right= data, how= 'left', left_on= 'STATE', right_on= 'STATE')
mergedDF.head(5)

from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(18,10))

#llcrnrlon-> longitude of lower left hand corner of the desired map domain (degrees).
#llcrnrlat-> latitude of lower left hand corner of the desired map domain (degrees).
#urcrnrlon-> longitude of upper right hand corner of the desired map domain (degrees).
#urcrnrlat-> latitude of upper right hand corner of the desired map domain (degrees).

# Starting with the basemap function to initialize a map
maps=Basemap(llcrnrlon=-180, llcrnrlat=-65, urcrnrlon=180, urcrnrlat=80)

# Adding elements
maps.drawmapboundary(fill_color='w')
maps.fillcontinents(color='grey', alpha=0.3)
maps.drawcoastlines(linewidth=0.1, color="white")
#maps.drawcountries()

# Add a point per position
maps.scatter(mergedDF['longitude'], mergedDF['latitude'], s=mergedDF['Number_Of_Attendance'],
alpha=0.4, c=mergedDF['label_color'], cmap="Reds")

# As we can see, all data are located only in the USA. Let's look closer.

# Focusing in the USA
from mpl_toolkits.basemap import Basemap
import numpy as np
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(18,10))

#llcrnrlon-> longitude of lower left hand corner of the desired map domain (degrees).
#llcrnrlat-> latitude of lower left hand corner of the desired map domain (degrees).
#urcrnrlon-> longitude of upper right hand corner of the desired map domain (degrees).
#urcrnrlat-> latitude of upper right hand corner of the desired map domain (degrees).

# Always start with the basemap function to initialize a map
maps=Basemap(llcrnrlon=-165, llcrnrlat=10, urcrnrlon=-50, urcrnrlat=60)

# Then add element: draw coast line, map boundary, and fill continents:
maps.drawmapboundary(fill_color='w')
maps.fillcontinents(color='grey', alpha=0.3)
maps.drawcoastlines(linewidth=0.1, color="white")
maps.drawcountries()
maps.drawstates()

# Add a point per position
maps.scatter(mergedDF['longitude'], mergedDF['latitude'], s=mergedDF['Number_Of_Attendance']*15,
alpha=0.6, c=mergedDF['label_color'], cmap="YlOrRd")

# Plot Info
plt.text( -160, 13,'Where All Data are Located', ha='left', va='bottom', size=20, color='#000000' )

# But among all, which State has the most Churn?
States with most and less Churn

# States with most Churn
# 02.SQL Functions
pysparkSqlDF = spSession.sql("select STATE, TARGET as Churn, count(ID) as Number_Of_Attendance \
                    from telecomTB \
                    group by STATE, Churn \
                    order by STATE, Churn")
pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = pysparkSqlDF.toPandas()

#pandasDF.filter(['Churn', 'Number_Of_Attendance'])
#pandasDF[['Churn', 'Number_Of_Attendance']]
Chrn = pandasDF[pandasDF.Churn  == 1].sort_values(by=['STATE'], ascending = True)
Chrn.head(5)
```

```
NotChrn = pandasDF[pandasDF.Churn  == 2].sort_values(by=['STATE'], ascending = True)
NotChrn.head(5)

# Setting width and height of the plot-figure
fig, axarr = plt.subplots(2, 1, figsize=(18,10))

sns.barplot(x=Chrn['STATE'], y=Chrn.Number_Of_Attendance, palette='YlGnBu_r',
ax=axarr[0]).set_title('Qt of Churns by States')
sns.barplot(x=NotChrn['STATE'], y=NotChrn.Number_Of_Attendance, palette='YlGnBu_r',
ax=axarr[1]).set_title('Qt of No Churns by States')

# As observed above:
#  State with most Churn: Texas(TX) and New Jersey(NJ)

# Merging (left) pandasDF and the USA long/lat data
mergedDF = pd.merge(left= Chrn, right= data, how= 'left', left_on= 'STATE', right_on= 'STATE')
mergedDF = mergedDF.sort_values(by=['Number_Of_Attendance'], ascending = False)
mergedDF = mergedDF.head(5)

# Top 5 States with Most Churn
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(18,10))

#llcrnrlon-> longitude of lower left hand corner of the desired map domain (degrees).
#llcrnrlat-> latitude of lower left hand corner of the desired map domain (degrees).
#urcrnrlon-> longitude of upper right hand corner of the desired map domain (degrees).
#urcrnrlat-> latitude of upper right hand corner of the desired map domain (degrees).

# Always start with the basemap function to initialize a map
maps=Basemap(llcrnrlon=-165, llcrnrlat=10, urcrnrlon=-50, urcrnrlat=60)

# Then add element: draw coast line, map boundary, and fill continents:
maps.drawmapboundary(fill_color='w')
maps.fillcontinents(color='grey', alpha=0.3)
maps.drawcoastlines(linewidth=0.1, color="white")
maps.drawcountries()
maps.drawstates()

# Add a point per position
maps.scatter(mergedDF['longitude'], mergedDF['latitude'], s=mergedDF['Number_Of_Attendance']*70,
alpha=0.6, c=mergedDF['label_color'], cmap="winter")
```

```
# Plot Info
plt.text( -160, 13,'Top 5 States with Most Churn', ha='left', va='bottom', size=20, color='#000000' )

# Merging (left) pandasDF and the USA long/lat data
mergedDF = pd.merge(left= Chrn, right= data, how= 'left', left_on= 'STATE', right_on= 'STATE')
mergedDF = mergedDF.sort_values(by=['Number_Of_Attendance'], ascending = False)
mergedDF = mergedDF.tail(5)

# Top 5 States with Less Churn
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(18,10))

#llcrnrlon-> longitude of lower left hand corner of the desired map domain (degrees).
#llcrnrlat-> latitude of lower left hand corner of the desired map domain (degrees).
#urcrnrlon-> longitude of upper right hand corner of the desired map domain (degrees).
#urcrnrlat-> latitude of upper right hand corner of the desired map domain (degrees).

# Always start with the basemap function to initialize a map
maps=Basemap(llcrnrlon=-165, llcrnrlat=10, urcrnrlon=-50, urcrnrlat=70)

# Then add element: draw coast line, map boundary, and fill continents:
maps.drawmapboundary(fill_color='w')
maps.fillcontinents(color='grey', alpha=0.3)
maps.drawcoastlines(linewidth=0.1, color="white")
maps.drawcountries()
maps.drawstates()

# Add a point per position
maps.scatter(mergedDF['longitude'], mergedDF['latitude'], s=mergedDF['Number_Of_Attendance']*180,
alpha=0.6, c=mergedDF['label_color'], cmap="winter")

# Plot Info
plt.text( -160, 13,'Top 5 States with Less Churn', ha='left', va='bottom', size=20, color='#000000' )

# A business opportunity is to find the gaps where services can be offered to cover a customer need that
he/she doesn't even know
# Let's see what this dataset is telling us about these opportunities
```
First Opportunity

```
# First - States with with most international calls and without an international plan
pysparkSqlDF = spSession.sql("select STATE, count(ID) as QT_INTPLAN \
                    from telecomTB \
```

```
                    where INTPLAN == 2 \
                    group by STATE \
                    order by QT_INTPLAN desc")
pysparkSqlDF.show(10)
# These are the top 10 States that makes international calls but doesn't have an International Plan,
# maybe a marketing plan could be introduced in these places to achieve more sales.

# It's possible to go even further looking at individuals customers, as follows:
```
Second Opportunity
```
# Second - more specifically: ID's with most international calls and without an international plan
pysparkSqlDF = spSession.sql("select ID as CUSTOMER, TTINTCALLS \
                    from telecomTB \
                    where INTPLAN == 2 \
                    order by TTINTCALLS desc")
pysparkSqlDF.show()
# In this way it's possible to map a customer usage of international calls and offer a personalised plan
individually.
# But how my marketing plan could convince that customers to buy it? Let's see through the data again:

# But how my marketing plan could convince that customers to buy it? Let's see through the data again:
```
Third Opportunity
```
# Third - Analysing if customers with an international plan are calling more and paying less charges
# and if who hasn't the international plan are paying higuer taxes

## With International Plan
pysparkSqlDF = spSession.sql("select ID as CUSTOMER, TTINTCALLS, TTINTCHARGE \
                    from telecomTB \
                    where INTPLAN == 1")
#pysparkSqlDF.show()
int_plan_DF = pysparkSqlDF.toPandas()
int_plan_DF['CHARGEBYCALL'] = int_plan_DF['TTINTCHARGE'] / int_plan_DF['TTINTCALLS']
int_plan_DF['CUSTOMER'] = int_plan_DF.index
int_plan_DF = int_plan_DF[50:80]

## Without International Plan
pysparkSqlDF = spSession.sql("select ID as CUSTOMER, TTINTCALLS, TTINTCHARGE \
                    from telecomTB \
                    where INTPLAN == 2")
#pysparkSqlDF.show()
no_int_plan_DF = pysparkSqlDF.toPandas()
no_int_plan_DF['CHARGEBYCALL'] = no_int_plan_DF['TTINTCHARGE'] / no_int_plan_DF['TTINTCALLS']
no_int_plan_DF['CUSTOMER'] = no_int_plan_DF.index
no_int_plan_DF = no_int_plan_DF[50:80]
```

```
# Let's take a look at some graphs
# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

plt.plot('CUSTOMER', 'CHARGEBYCALL', data=int_plan_DF, marker='', markerfacecolor='blue',
markersize=12, color='green', linestyle = 'dashed', linewidth=1, label='International_Plan')
plt.plot('CUSTOMER', 'CHARGEBYCALL', data=no_int_plan_DF, marker='', markerfacecolor='red',
markersize=12, color='red', linewidth=1, label='No_International_Plan')
plt.title('Charges by Call')
plt.xticks([])
plt.legend()

# As noted above in the red line, people without an international plan are paying more charges by call,
# the difference:
#print(int_plan_DF.CHARGEBYCALL.mean())
#print(no_int_plan_DF.CHARGEBYCALL.mean())
print("Customers without an international plan are paying %.2f%% more taxes."
%(no_int_plan_DF.CHARGEBYCALL.mean()*100/int_plan_DF.CHARGEBYCALL.mean()-100))

# As we saw some opportunities to the customers, let's take a look at the opportunities to the telecom
company
```

Fourth Opportunity

```
# Fourth - Now looking how the company is acquiring resources analysing the Charges of each time of
day
# Get a ScatterPlot of the Charges of each time of day (Maybe in one Graph)

pysparkSqlDF = spSession.sql("select TTDAYCHARGE/TTDAYMIN as DAY_CHARGE_MIN,
TTEVECHARGE/TTEVEMIN as EVENING_CHARGE_MIN \
                from telecomTB")
#pysparkSqlDF.show()
pandasDF = pysparkSqlDF.toPandas()
pandasDF[0:1]

print('The Charges Applied by the Telecom Company are:')
print('Day Charge/Min:    $%.3f' %(pandasDF.DAY_CHARGE_MIN[0]))
print('Evening Charge/Min: $%.3f' %(pandasDF.EVENING_CHARGE_MIN[0]))

pysparkSqlDF = spSession.sql("select TTDAYMIN, TTEVEMIN \
                from telecomTB")
#pysparkSqlDF.show()
pandasDF = pysparkSqlDF.toPandas()
import pandas as pd
plotingDF = pd.DataFrame({'Qt': [pandasDF.TTDAYMIN.sum(), pandasDF.TTEVEMIN.sum()],
```

```
                'Time_of_Day': ['Day', 'Evening']})

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

sns.barplot(x=plotingDF['Qt'], y=plotingDF.Time_of_Day, palette='Greens')
plt.title("Total Minutes Spend in Calls")

# If we take a close look at the plot there is not much difference between day and evening quantities of calls
# The Day Charge is USD 0.170 and the Evening Charge is USD 0.085

dayCharge = 0.170
eveCharge = 0.085
plotingDF = pd.DataFrame({'Qt': [pandasDF.TTDAYMIN.sum()*dayCharge,
pandasDF.TTEVEMIN.sum()*eveCharge],
                'Time_of_Day': ['Day', 'Evening']})

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

sns.barplot(x=plotingDF['Qt'], y=plotingDF.Time_of_Day, palette='Greens')
plt.title("Total Received (USD) by Calls")
print("Total Received: $%.2f" %(pandasDF.TTDAYMIN.sum()*dayCharge +
pandasDF.TTEVEMIN.sum()*eveCharge))

# Total Received: $158,801.11

# Knowing this volume of calls per time of day it is possible to manage the Day or/and Evening Charge and see what happens to earnings
dayCharge = 0.16   # Lowering 0.01 cents
eveCharge = 0.105  # Increasing 0.02 cents
plotingDF = pd.DataFrame({'Qt': [pandasDF.TTDAYMIN.sum()*dayCharge,
pandasDF.TTEVEMIN.sum()*eveCharge],
                'Time_of_Day': ['Day', 'Evening']})

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

sns.barplot(x=plotingDF['Qt'], y=plotingDF.Time_of_Day, palette='Greens')
plt.title("Total Received (USD) by Calls")
print("Total Received: USD %.2f" %(pandasDF.TTDAYMIN.sum()*dayCharge +
pandasDF.TTEVEMIN.sum()*eveCharge))

# By this way we leave from USD 158,801.11 to USD 166,206.55 only changing some cents
```

```
# On the other hand, it's possible not to change the charge but make some incentive marketing for the
daytime consumption.

# What if the volume of calls increased in 10% by day, how much the company will receive?
# Without changing the charges but making marketing
dayCharge = 0.170
eveCharge = 0.085
plotingDF = pd.DataFrame({'Qt': [pandasDF.TTDAYMIN.sum()*1.1*dayCharge,
pandasDF.TTEVEMIN.sum()*eveCharge],
                          'Time_of_Day': ['Day', 'Evening']})

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

sns.barplot(x=plotingDF['Qt'], y=plotingDF.Time_of_Day, palette='Greens')
plt.title("Total Received (USD) by Calls")
print("Total Received: $%.2f" %(pandasDF.TTDAYMIN.sum()*1.1*dayCharge +
pandasDF.TTEVEMIN.sum()*eveCharge))

# By this way we leave from USD 158,801.11 to USD 168,987.34 by acquiring 10% more customers
```
Fifth Opportunity
```
# Fifth - The last one, Understand How Much NBCSCALLS are converted in Churn(TARGET), i.e., the
correlation between both features
pysparkSqlDF = spSession.sql("select NBCSCALLS, TARGET as CHURN \
                          from telecomTB")
#pysparkSqlDF.show()
pandasDF = pysparkSqlDF.toPandas()

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))
sns.distplot(pandasDF['NBCSCALLS'], bins=10, hist=True, kde=False, rug=False, color='#82ffb0')
# Through the graph that follows we get to assume that most customers calls from 1 to 4 times more
often
# But how much of these are converted in Positive Churn?

pysparkSqlDF = spSession.sql("select NBCSCALLS, TARGET as CHURN, count(ID) as NUMBER_OF_CHURN
\
                          from telecomTB \
                          group by NBCSCALLS, CHURN \
                          order by NBCSCALLS asc")
pysparkSqlDF.show()
pandasDF = pysparkSqlDF.toPandas()
# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))
```

```
sns.lineplot(data=pandasDF[pandasDF.CHURN  == 1], x='NBCSCALLS', y='NUMBER_OF_CHURN',
palette='Red', legend=False).set_title('Churn Index by Number of Customer Calls')
plt.xlabel('Number of Customer Service Calls')

# As the graph above shows, my recommendation is to pay attention in number of Customer Service '4'
and understand why
# this sudden high on number of churn is happening once the trend was comming down.

# Analysis Performed
# States with with most international calls and without an international plan
# or more specifically: ID's with most international calls and without an international plan

# Analyse if who has the international plan calls more and pay less
# and if who hasn't the intenational plan is paying high taxes
# Using Line plots

# Get a ScatterPlot of the Charges of each time of day (Maybe in one Graph)
# Get the Highest, Mid and Lowest Charge per Time of Day
# Get the Charge per time of Day (and maybe include in the ScatterPlot individually)

# Get a line graph with minutes together with charge by ID

# Get min/calls each time of day

# Get How Much NBCSCALLS are converted in Churn(TARGET), i.e., the correlation between both features
```

Machine Learning

```
# Features Available to Modeling
telecomDF4.printSchema()
```

Data Balance

```
# Before entering in the Machine Learning Model let's see if we have balanced data so my model will be
fair in predicting both Churn and No Churn
pysparkSqlDF = spSession.sql("select TARGET, count(ID) as INSTANCES \
                    from telecomTB \
                    group by TARGET")
pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = pysparkSqlDF.toPandas()

plt.figure(figsize=(7,7))
```

```
plt.pie(pandasDF['INSTANCES'], labels=['Churn', 'No Churn'], colors=['#ff6666', '#99ff99'],
wedgeprops={'linewidth': 7, 'edgecolor': 'white'}, autopct='%1.1f%%')

# The ideal is to generate a machine learning model that can predict both one attribute and another, but
unfortunatelly I don't have enough data from Churn.
# This project will continue knowing that my model could possibly fail in predicting Churn.
# After training I'll acquire new, random and unknown data to test if the model was successful in this
way.
```

Correlation

```
# Correlation between features
# Getting all my data into a dataframe:
#telecomDF4.take(5)

# First I'll get each column of telecomDF4
for i in telecomDF4.columns:
    # for each one I'll 'select(i)', and use the 'take' action from [0][0] combination as if it was a matrix
    if not(isinstance(telecomDF4.select(i).take(1)[0][0], str)):
        print("Correlation between Target(Churn) with: ", i, telecomDF4.stat.corr('TARGET', i))
```

Pre-Processing the Dataset

```
# Pre-Processing my Dataset

# Creating an LabeledPoint (target, Vector[features])
# Removing not relevant columns or with low correlation to the model, this way I choose the columns I
want the model to have (just observe the selection done in Vectors.dense(....))

# ATTENTION TO THIS VERY IMPORTANT CONCEPT!!!!!!!!!
# SOME APACHE SPARK MACHINE LEARNING ALGORITHMS (MAINLY THE REGRESSION TYPE) NEED DATA
TO BE
# IN A SPECIFIC FORMAT TO PERFORM THE TRAINING(FIT).
# TO THIS HAPPEN I NEED TO DELIVER TO THE MODEL THE DATA IN VECTOR FORMAT, SPECIFICALLY
USING A
# DENSE OR SPARSE VECTOR. THIS BECAUSE APACHE SPARK WORKS WITH CLUSTER AND THEN IT WILL
# DISTRIBUTE THESE DATA BETWEEN MACHINES.
from pyspark.ml.linalg import Vectors

def transformFeatures(row):
    obj = (row['TARGET'], Vectors.dense(row['INTPLAN'], row['NBVMMSG'], row['TTDAYMIN'],
row['TTDAYCHARGE'], row['(TTDAYMIN / TTDAYCALLS)'], row['TTEVEMIN'], row['TTEVECHARGE'],
row['NBCSCALLS']) )
    return obj

# HERE telecomDF4 IS A DATAFRAME, BUT IS BEING CONVERTED TO AN rdd AND THEN I'M ABLE TO USE
THE map FUNCTION
```

```
telecomRDD4 = telecomDF4.rdd.map(transformFeatures)

telecomRDD4.take(5)

# CONVERTING TO DATAFRAME AGAIN
telecomDF5 = spSession.createDataFrame(telecomRDD4, ['TARGET', 'FEATURES'])
telecomDF5.show(10)
telecomDF5.cache()
```

Machine Learning Modeling

Train/Test Split

```
# Now entering the Machine Learning Process
import random

random.seed(69)
(train_data, test_data) = telecomDF5.randomSplit([0.7, 0.3])
print(train_data.count(), "data to train my model")
print(test_data.count(), "data to test my model")

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have
#train_data.select('TARGET').distinct().show()

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have in the train_data
train_data.cube('TARGET').agg(count('TARGET').alias('Qt of Data')).orderBy('TARGET').show()

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have in the test_data
test_data.cube('TARGET').agg(count('TARGET').alias('Qt of Data')).orderBy('TARGET').show()
```

Creating and Training Model - I

```
# Decision Tree ML Model
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Avoiding Underfitting and Overfitting in Decision Trees
for depth in range(1,15):
    model_v1 = DecisionTreeClassifier(maxDepth=depth, labelCol= 'TARGET', featuresCol= 'FEATURES')
    model_v1_train = model_v1.fit(train_data)
    model_v1_prediction = model_v1_train.transform(test_data)
    evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                    labelCol = 'TARGET',
                                    metricName = 'accuracy')
    acc = evaluator.evaluate(model_v1_prediction)
    print('In Depth %.0f this model is %.2f%% Accurate.' %(depth, acc*100))

# Better Depth is 6
```

```
# Building the model
model_v1 = DecisionTreeClassifier(maxDepth=6, labelCol= 'TARGET', featuresCol= 'FEATURES')

# Training my Model
model_v1_train = model_v1.fit(train_data)

print(model_v1_train.numNodes, "Nodes")
print(model_v1_train.depth, "Depth")
```

Predictions and Evaluation

```
# Making Predictions with test_data
model_v1_prediction = model_v1_train.transform(test_data)
model_v1_prediction.select('prediction', 'TARGET').collect()

# Evaluating my model
evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                labelCol = 'TARGET',
                                metricName = 'accuracy')
acc = evaluator.evaluate(model_v1_prediction)
print('This model is %.2f%% Accurate.' %(acc*100))

# Taking the Confusion Matrix
model_v1_prediction.groupBy('TARGET', 'prediction').count().show()
```

Plotting Confusion Matrix

```
model_v1_prediction.head(5)

type(model_v1_prediction)

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = model_v1_prediction.toPandas()

# Getting Necessarry Data from model_v1_prediction
Actual = pandasDF.ix[:, 'TARGET']
Predic = pandasDF.ix[:, 'prediction']

data = {'Actual': Actual,
      'Predicted': Predic
      }

confMatDF = pd.DataFrame(data, columns=['Actual', 'Predicted'])
confMatDF.head(5)

# Renaming Target (1-> Churn, 2-> No Churn)
def targetTransformation(lst):
   if lst == 1:
```

```
    return 'Churn'
  else:
    return 'No Churn'

auxList1 = list(confMatDF['Actual'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Actual'] = auxList2

auxList1 = list(confMatDF['Predicted'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Predicted'] = auxList2

confusionMatrix = pd.crosstab(confMatDF.Actual, confMatDF.Predicted, rownames=['Actual Value'],
colnames=['Predicted Value'] )
confusionMatrix

confusionMatrix.values.sum()

# Looking as Percentage of Churn and No Churn
import numpy as np

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

status = ['True Positive', 'False Negative', 'False Positive', 'True Negative']
val    = [x for x in confusionMatrix.values.flatten()]
percent= ['{:.2%}'.format(x) for x in (confusionMatrix.values.flatten()/confusionMatrix.values.sum())]

labels = [f'{q1}\n{q2}\n{q3}' for q1, q2, q3 in zip(status, val, percent)]
labels = np.asarray(labels).reshape(2,2)

sns.heatmap((confusionMatrix/confusionMatrix.values.sum()), annot=labels, fmt='', cmap='BuGn',
cbar=False)
sns.set(font_scale=1.5)
plt.title('Confusion Matrix')

# We get pretty good results but the True Negative is clearly high because we don't have a balanced
dataset
# Let's see the reflex of this unbalanced data in an unseen data later
```

Creating and Training Model - II

```
# Random Forest ML Model
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```
# Building new model
model_v2 = RandomForestClassifier(labelCol = "TARGET", featuresCol = "FEATURES")
model_v2_train = model_v2.fit(train_data)

# With the trained model I show it new data to make predictions with test_data
model_v2_prediction = model_v2_train.transform(test_data)
model_v2_prediction.select('prediction', 'TARGET').collect()

# Evaluating my model
evaluator = MulticlassClassificationEvaluator(predictionCol = "prediction",
                                labelCol = "TARGET",
                                metricName = "accuracy")
acc = evaluator.evaluate(model_v2_prediction)
print('This model is %.2f%% Accurate.' %(acc*100))
```

Plotting Confusion Matrix

```
model_v2_prediction.head(5)

type(model_v2_prediction)

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = model_v2_prediction.toPandas()

# Getting Necessarry Data from model_v1_prediction
Actual = pandasDF.ix[:, 'TARGET']
Predic = pandasDF.ix[:, 'prediction']

data = {'Actual': Actual,
      'Predicted': Predic
      }

confMatDF = pd.DataFrame(data, columns=['Actual', 'Predicted'])
confMatDF.head(5)

# Renaming Target (1-> Churn, 2-> No Churn)
def targetTransformation(lst):
  if lst == 1:
     return 'Churn'
  else:
     return 'No Churn'

auxList1 = list(confMatDF['Actual'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Actual'] = auxList2
```

```
auxList1 = list(confMatDF['Predicted'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Predicted'] = auxList2

confusionMatrix = pd.crosstab(confMatDF.Actual, confMatDF.Predicted, rownames=['Actual Value'],
colnames=['Predicted Value'] )
confusionMatrix

confusionMatrix.values.sum()

# Looking as Percentage of Churn and No Churn
import numpy as np

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

status = ['True Positive', 'False Negative', 'False Positive', 'True Negative']
val    = [x for x in confusionMatrix.values.flatten()]
percent= ['{:.2%}'.format(x) for x in (confusionMatrix.values.flatten()/confusionMatrix.values.sum())]

labels = [f'{q1}\n{q2}\n{q3}' for q1, q2, q3 in zip(status, val, percent)]
labels = np.asarray(labels).reshape(2,2)

sns.heatmap((confusionMatrix/confusionMatrix.values.sum()), annot=labels, fmt='', cmap='BuGn',
cbar=False)
sns.set(font_scale=1.5)
plt.title('Confusion Matrix')

# The first model leads to a better Accuracy and TP and TN higher.
# The second one is giving a better TN prediction, but the FN is higher than the TP.
# My choice is the first model.
```

New and Unseen Data

Loading Data

```
## Datasets ------------------------------------------------------------------------------------
# Loading data and getting an RDD
# textFile CREATES AN RDD
telecomRDD = sc.textFile("projeto4_telecom_teste.csv")

# Optimizing performance using cache on RDD
telecomRDD.cache()

# Making an Action
telecomRDD.count()
```

```
# Making another Action
telecomRDD.take(5)
# Note that the data is between simple aspects '' (each line), indicating they are strings, so I'll convert it further using a function I'll create
```

Feature Engineering

```
# Cleaning data - Removing first line
telecomRDD2 = telecomRDD.filter(lambda x: "state" not in x)
telecomRDD2.count()

# Looking Again at my data
telecomRDD2.take(5)

# Looking to ALL data and trying to figure out if there is missing values (?, NaN, "", ...)
telecomRDD2.collect()
# No Missing Values
```

Preparing Data to Dataframe

```
# Feature Engineering I - Function
def featengI(data):

    # Dividing data into columns, separating by "," character
    dataList = data.split(",")

    ID = (dataList[0])              # ID
    STATE = str(dataList[1])        # State -> North American Abbreviated States (HI - Hawaii / MT - Montana / OH - Ohio)
    ACCLGT = int(dataList[2])       # Account_Length
    AREACODE = (dataList[3])        # Phone Number Area Code
    #PHONENB = (dataList[4])        # Phone_No -> Phone Number (IT IS NOT IN THE test-train DATA)
    #INTPLAN = (dataList[4])        # International_Plan -> "yes" or "no" (Convert yes->1 or no->2)
    INTPLAN = 1.0 if dataList[4] == '"yes"' else 2.0
    #VMPLAN = (dataList[5])         # Voice_Mail_Plan -> "yes" or "no" (Convert yes->1 or no->2) I THINK THIS FEATURE IS NOT HELPING AND CAN BE ELIMINATED BECAUSE THE NEXT VARIABLE (NBVMMSG) ONLY HAS VALUES WHEN THIS IS '1', IT'S LIKE DOUBLE INFORMATION ABOUT THE SAME THING
    VMPLAN = 1.0 if dataList[5] == '"yes"' else 2.0
    NBVMMSG = int(dataList[6])      # Number_Voice_Mail_Messages
    TTDAYMIN = float(dataList[7])   # Total_Day_minutes
    TTDAYCALLS = int(dataList[8])   # Total_Day_Calls
    TTDAYCHARGE = float(dataList[9])  # Total_Day_Charge
    TTEVEMIN = float(dataList[10])   # Total_Eve_Minutes
    TTEVECALLS = int(dataList[11])   # Total_Eve_Calls
    TTEVECHARGE = float(dataList[12]) # Total_Eve_Charge
```

```
    TTNGTMIN = float(dataList[13])    # Total_Night_Minutes
    TTNGTCALLS = int(dataList[14])    # Total_Night_Calls
    TTNGTCHARGE = float(dataList[15]) # Total_Night_Charge
    TTINTMIN = float(dataList[16])    # Total_Intl_Minutes
    TTINTCALLS = int(dataList[17])    # Total_Intl_Calls
    TTINTCHARGE = float(dataList[18]) # Total_Intl_Charge
    NBCSCALLS = int(dataList[19])     # Number_Customer_Service_Calls
    #TARGET = (dataList[20])         # Churn (Target) -> "yes" or "no" (Convert yes->1 or no->2)
    TARGET = 1 if dataList[20] == '"yes"' else 2

    # Creating 'lines' using the 'Row' function, preparing to the dataframe analysis, cleaning and
converting the data from string to float
    lines = Row(ID = ID, STATE = STATE, ACCLGT = ACCLGT, AREACODE = AREACODE,
                INTPLAN = INTPLAN, VMPLAN = VMPLAN, NBVMMSG = NBVMMSG,
                TTDAYMIN = TTDAYMIN, TTDAYCALLS = TTDAYCALLS, TTDAYCHARGE = TTDAYCHARGE,
                TTEVEMIN = TTEVEMIN, TTEVECALLS = TTEVECALLS, TTEVECHARGE = TTEVECHARGE,
                TTNGTMIN = TTNGTMIN, TTNGTCALLS = TTNGTCALLS, TTNGTCHARGE = TTNGTCHARGE,
                TTINTMIN = TTINTMIN, TTINTCALLS = TTINTCALLS, TTINTCHARGE = TTINTCHARGE,
                NBCSCALLS = NBCSCALLS, TARGET = TARGET)

    return lines

# Applying function above to RDD without headers
telecomRDD3 = telecomRDD2.map(featengl)

# Looking at the result
telecomRDD3.cache
telecomRDD3.take(1)

# All data are in String format, I'll convert to Integer or Float as needed
#schema = StructType([
#    StructField('ID', StringType(), True)
#])
```

Creating a Pyspark Dataframe

```
# Creating a Dataframe SO I'M ABLE TO USE THE select FUNCTION FROM SparkSQL
telecomDF = spSession.createDataFrame(telecomRDD3)

type(telecomDF)

# Printing telecomDF Object Type and each Row Type
print(telecomDF)
# As noted SOME data (ID, STATE) are in String format, I'll convert to Integer or Float as needed

telecomDF.printSchema()
```

```
telecomDF.show(5)
```
Removing Double Quotes in Strings

```
# Function to Remove DoubleQuote (DQ) so I'll transform from String to Integer
def removingDQ(stringData):
   return stringData.replace('"', "")

# Using udf -> User Defined Function (Turning Python Functions into PySpark Functions)
# Look at this -> https://changhsinlee.com/pyspark-udf/
from pyspark.sql.functions import udf
udf_removingDQ = udf(removingDQ, StringType())

telecomDF2 = telecomDF.withColumn('ID', udf_removingDQ(telecomDF['ID'])).withColumn('STATE',
udf_removingDQ(telecomDF['STATE']))
telecomDF2.show(30)

# Now that necessary data doesn't have DQ I'm able to perform casting to transform from one type to
another
# As follows there are 2 ways to make this transformation
```
Transforming Features (Two Possibilities)
My Choice

```
# I'll choose the second one
telecomDF4 = telecomDF2.select( telecomDF2['ID'].cast(IntegerType()).alias('ID'),
                        telecomDF2['STATE'],
                        telecomDF2['ACCLGT'],
                        telecomDF2['INTPLAN'],
                        telecomDF2['VMPLAN'],
                        telecomDF2['NBVMMSG'],
                        telecomDF2['TTDAYMIN'],
                        telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTDAYCHARGE'],
                        telecomDF2['TTDAYMIN']/telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTEVEMIN'],
                        telecomDF2['TTEVECALLS'],
                        telecomDF2['TTEVECHARGE'],
                        telecomDF2['TTEVEMIN']/telecomDF2['TTEVECALLS'],
                        telecomDF2['TTINTMIN'],
                        telecomDF2['TTINTCALLS'],
                        telecomDF2['TTINTCHARGE'],
                        telecomDF2['TTINTMIN']/telecomDF2['TTINTCALLS'],
                        telecomDF2['NBCSCALLS'],
                        telecomDF2['TARGET']
                        )
```

```
telecomDF4.show(3)

# Registering the dataframe as a Temp Table (so i'll be able to use queries SQL ANSI)
# HERE I'M CREATING A REAL TABLE, IT WILL EXIST IN MEMORY, NOT ONLY THE DATAFRAME
telecomDF4.createOrReplaceTempView("telecomTB")
```

Data Balance

```
# Before entering in the Machine Learning Model let's see if we have balanced data so my model will be fair in predicting both Churn and No Churn
pysparkSqlDF = spSession.sql("select TARGET, count(ID) as INSTANCES \
                    from telecomTB \
                    group by TARGET")
pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = pysparkSqlDF.toPandas()

plt.figure(figsize=(7,7))

plt.pie(pandasDF['INSTANCES'], labels=['Churn', 'No Churn'], colors=['#ff6666', '#99ff99'],
wedgeprops={'linewidth': 7, 'edgecolor': 'white'}, autopct='%1.1f%%')

# The test data provided also has few Churn Data, It's more likeble that the model will have a great acurracy.
# But it's important to notice that in a real environment I highly recommend to acquire more Churn data.
```

Correlation

```
# Correlation between features
# Getting all my data into a dataframe:
#telecomDF4.take(5)

# First I'll get each column of telecomDF4
for i in telecomDF4.columns:
  # for each one I'll 'select(i)', and use the 'take' action from [0][0] combination as if it was a matrix
  if not(isinstance(telecomDF4.select(i).take(1)[0][0], str)):
    print("Correlation between Target(Churn) with: ", i, telecomDF4.stat.corr('TARGET', i))
```

Pre-Processing the Dataset

```
# Pre-Processing my Dataset

# Criando um LabeledPoint (target, Vector[features])
# Removing not relevant columns or with low correlation to the model, this way I choose the columns I want the model to have (just observe the selection done in Vectors.dense(....))

# ATTENTION TO THIS VERY IMPORTANT CONCEPT!!!!!!!!!
```

```
# SOME APACHE SPARK MACHINE LEARNING ALGORITHMS (MAINLY THE REGRESSION TYPE) NEED DATA TO BE
# IN A SPECIFIC FORMAT TO PERFORM THE TRAINING(FIT).
# TO THIS HAPPEN I NEED TO DELIVER TO THE MODEL THE DATA IN VECTOR FORMAT, SPECIFICALLY USING A
# DENSE OR SPARSE VECTOR. THIS BECAUSE APACHE SPARK WORKS WITH CLUSTER AND THEN IT WILL
# DISTRIBUTE THESE DATA BETWEEN MACHINES.
from pyspark.ml.linalg import Vectors

def transformFeatures(row):
   obj = (row['TARGET'], Vectors.dense(row['INTPLAN'], row['NBVMMSG'], row['TTDAYMIN'],
row['TTDAYCHARGE'], row['(TTDAYMIN / TTDAYCALLS)'], row['TTEVEMIN'], row['TTEVECHARGE'],
row['NBCSCALLS']) )
   return obj

# HERE telecomDF4 IS A DATAFRAME, BUT IS BEING CONVERTED TO AN rdd AND THEN I'M ABLE TO USE THE map FUNCTION
telecomRDD4 = telecomDF4.rdd.map(transformFeatures)

telecomRDD4.take(5)

# CONVERTING TO DATAFRAME AGAIN
telecomDF5 = spSession.createDataFrame(telecomRDD4, ['TARGET', 'FEATURES'])
telecomDF5.show(10)
telecomDF5.cache()
```

Machine Learning Modeling

```
# Making Predictions with test_data
new_data_prediction = model_v1_train.transform(telecomDF5)
# Evaluating my model
evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                labelCol = 'TARGET',
                                metricName = 'accuracy')
acc = evaluator.evaluate(new_data_prediction)
print('This model is %.2f%% Accurate.' %(acc*100))

# Taking the Confusion Matrix
new_data_prediction.groupBy('TARGET', 'prediction').count().show()
```

Plotting Confusion Matrix

```
new_data_prediction.head(5)

type(new_data_prediction)

# Transforming to Pandas so I'll plot some analysis using Seaborn
```

```
pandasDF = new_data_prediction.toPandas()

# Getting Necessarry Data from model_v1_prediction
Actual = pandasDF.ix[:, 'TARGET']
Predic = pandasDF.ix[:, 'prediction']

data = {'Actual': Actual,
        'Predicted': Predic
        }

confMatDF = pd.DataFrame(data, columns=['Actual', 'Predicted'])
confMatDF.head(5)

# Renaming Target (1-> Churn, 2-> No Churn)
def targetTransformation(lst):
    if lst == 1:
        return 'Churn'
    else:
        return 'No Churn'

auxList1 = list(confMatDF['Actual'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Actual'] = auxList2

auxList1 = list(confMatDF['Predicted'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Predicted'] = auxList2

confusionMatrix = pd.crosstab(confMatDF.Actual, confMatDF.Predicted, rownames=['Actual Value'],
colnames=['Predicted Value'] )
confusionMatrix

confusionMatrix.values.sum()

# Looking as Percentage of Churn and No Churn
import numpy as np

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

status = ['True Positive', 'False Negative', 'False Positive', 'True Negative']
val    = [x for x in confusionMatrix.values.flatten()]
percent= ['{:.2%}'.format(x) for x in (confusionMatrix.values.flatten()/confusionMatrix.values.sum())]

labels = [f'{q1}\n{q2}\n{q3}' for q1, q2, q3 in zip(status, val, percent)]
```

```
labels = np.asarray(labels).reshape(2,2)

sns.heatmap((confusionMatrix/confusionMatrix.values.sum()), annot=labels, fmt='', cmap='BuGn',
cbar=False)
sns.set(font_scale=1.5)
plt.title('Confusion Matrix')

# We get pretty good results but the True Negative is clearly high because we don't have a balanced
dataset
# Like I told before it's important to notice that in a real environment I highly recommend to acquire
more Churn data and try to minimize the False Negative where Churn is considered No Churn.

# What is the reflex of having balanced Churn and No Churn Data?
# Let's simulate an environment where we have 50/50 data and analyse the Confusion Matrix
```

Balancing Churn Data

Loading Data

```
## Datasets ----------------------------------------------------------------------------------
# Loading data and getting an RDD
# textFile CREATES AN RDD
telecomRDD = sc.textFile("projeto4_telecom_treino.csv")

# Optimizing performance using cache on RDD
telecomRDD.cache()

# Making an Action
telecomRDD.count()

# Making another Action
telecomRDD.take(5)
# Note that the data is between simple aspects '' (each line), indicating they are strings, so I'll convert it
further using a function I'll create
```

Feature Engineering

```
# Cleaning data - Removing first line
telecomRDD2 = telecomRDD.filter(lambda x: "state" not in x)
telecomRDD2.count()

# Looking Again at my data
telecomRDD2.take(5)

# Looking to ALL data and trying to figure out if there is missing values (?, NaN, "", ...)
telecomRDD2.collect()
```

```
# No Missing Values
```
Preparing Data to Dataframe

```
# Feature Engineering I - Function
def featengl(data):

    # Dividing data into columns, separating by "," character
    dataList = data.split(",")

    ID = (dataList[0])            # ID
    STATE = str(dataList[1])        # State -> North American Abbreviated States (HI - Hawaii / MT -
Montana / OH - Ohio)
    ACCLGT = int(dataList[2])       # Account_Length
    AREACODE = (dataList[3])         # Phone Number Area Code
    #PHONENB = (dataList[4])         # Phone_No -> Phone Number (IT IS NOT IN THE test-train DATA)
    #INTPLAN = (dataList[4])         # International_Plan -> "yes" or "no" (Convert yes->1 or no->2)
    INTPLAN = 1.0 if dataList[4] == '"yes"' else 2.0
    #VMPLAN = (dataList[5])          # Voice_Mail_Plan -> "yes" or "no" (Convert yes->1 or no->2) I THINK
THIS FEATURE IS NOT HELPING AND CAN BE ELIMINATED BECAUSE THE NEXT VARIABLE (NBVMMSG)
ONLY HAS VALUES WHEN THIS IS '1', IT'S LIKE DOUBLE INFORMATION ABOUT THE SAME THING
    VMPLAN = 1.0 if dataList[5] == '"yes"' else 2.0
    NBVMMSG = int(dataList[6])       # Number_Voice_Mail_Messages
    TTDAYMIN = float(dataList[7])    # Total_Day_minutes
    TTDAYCALLS = int(dataList[8])    # Total_Day_Calls
    TTDAYCHARGE = float(dataList[9])  # Total_Day_Charge
    TTEVEMIN = float(dataList[10])    # Total_Eve_Minutes
    TTEVECALLS = int(dataList[11])    # Total_Eve_Calls
    TTEVECHARGE = float(dataList[12]) # Total_Eve_Charge
    TTNGTMIN = float(dataList[13])    # Total_Night_Minutes
    TTNGTCALLS = int(dataList[14])    # Total_Night_Calls
    TTNGTCHARGE = float(dataList[15]) # Total_Night_Charge
    TTINTMIN = float(dataList[16])    # Total_Intl_Minutes
    TTINTCALLS = int(dataList[17])    # Total_Intl_Calls
    TTINTCHARGE = float(dataList[18]) # Total_Intl_Charge
    NBCSCALLS = int(dataList[19])     # Number_Customer_Service_Calls
    #TARGET = (dataList[20])         # Churn (Target) -> "yes" or "no" (Convert yes->1 or no->2)
    TARGET = 1 if dataList[20] == '"yes"' else 2

    # Creating 'lines' using the 'Row' function, preparing to the dataframe analysis, cleaning and
converting the data from string to float
    lines = Row(ID = ID, STATE = STATE, ACCLGT = ACCLGT, AREACODE = AREACODE,
            INTPLAN = INTPLAN, VMPLAN = VMPLAN, NBVMMSG = NBVMMSG,
            TTDAYMIN = TTDAYMIN, TTDAYCALLS = TTDAYCALLS, TTDAYCHARGE = TTDAYCHARGE,
            TTEVEMIN = TTEVEMIN, TTEVECALLS = TTEVECALLS, TTEVECHARGE = TTEVECHARGE,
```

```
        TTNGTMIN = TTNGTMIN, TTNGTCALLS = TTNGTCALLS, TTNGTCHARGE = TTNGTCHARGE,
        TTINTMIN = TTINTMIN, TTINTCALLS = TTINTCALLS, TTINTCHARGE = TTINTCHARGE,
        NBCSCALLS = NBCSCALLS, TARGET = TARGET)

    return lines

# Applying function above to RDD without headers
telecomRDD3 = telecomRDD2.map(featengl)

# Looking at the result
telecomRDD3.cache
telecomRDD3.take(1)

# Some data are still in String format, I'll convert to Integer or Float as needed
```
Creating a Pyspark Dataframe
```
# Creating a Dataframe SO I'M ABLE TO USE THE select FUNCTION FROM SparkSQL
telecomDF = spSession.createDataFrame(telecomRDD3)

type(telecomDF)

# Printing telecomDF Object Type and each Row Type
print(telecomDF)
# As noted SOME data (ID, STATE) are in String format, I'll convert to Integer or Float as needed

telecomDF.printSchema()

telecomDF.show(5)
```
Removing Double Quotes in Strings
```
# Function to Remove DoubleQuote (DQ) so I'll transform from String to Integer
def removingDQ(stringData):
    return stringData.replace('"', "")

# Using udf -> User Defined Function (Turning Python Functions into PySpark Functions)
# Look at this -> https://changhsinlee.com/pyspark-udf/
from pyspark.sql.functions import udf
udf_removingDQ = udf(removingDQ, StringType())

telecomDF2 = telecomDF.withColumn('ID', udf_removingDQ(telecomDF['ID'])).withColumn('STATE',
udf_removingDQ(telecomDF['STATE']))
telecomDF2.show(30)

# Now that necessary data doesn't have DQ I'm able to perform casting to transform from one type to
another
```
My Choice

```
telecomDF4 = telecomDF2.select( telecomDF2['ID'].cast(IntegerType()).alias('ID'),
                        telecomDF2['STATE'],
                        telecomDF2['ACCLGT'],
                        telecomDF2['INTPLAN'],
                        telecomDF2['VMPLAN'],
                        telecomDF2['NBVMMSG'],
                        telecomDF2['TTDAYMIN'],
                        telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTDAYCHARGE'],
                        telecomDF2['TTDAYMIN']/telecomDF2['TTDAYCALLS'],
                        telecomDF2['TTEVEMIN'],
                        telecomDF2['TTEVECALLS'],
                        telecomDF2['TTEVECHARGE'],
                        telecomDF2['TTEVEMIN']/telecomDF2['TTEVECALLS'],
                        telecomDF2['TTINTMIN'],
                        telecomDF2['TTINTCALLS'],
                        telecomDF2['TTINTCHARGE'],
                        telecomDF2['TTINTMIN']/telecomDF2['TTINTCALLS'],
                        telecomDF2['NBCSCALLS'],
                        telecomDF2['TARGET']
                        )
telecomDF4.show(3)

# Registering the dataframe as a Temp Table (so i'll be able to use queries SQL ANSI)
# HERE I'M CREATING A REAL TABLE, IT WILL EXIST IN MEMORY, NOT ONLY THE DATAFRAME
telecomDF4.createOrReplaceTempView("telecomTB")
```

New Data Balance

```
type(telecomDF4)

# Before entering in the Machine Learning Model again let's get 50/50 of the data
pysparkSqlDF = spSession.sql("select * \
                        from telecomTB \
                        where TARGET == 2")
#pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF2 = pysparkSqlDF.toPandas()
len(pandasDF2)

# Sampling almost the same quantity of Churn(483)
pandasDF2 = pandasDF2.sample(500)
len(pandasDF2)
```

```
# Getting Churn Data
pysparkSqlDF = spSession.sql("select * \
                    from telecomTB \
                    where TARGET == 1")
pandasDF1 = pysparkSqlDF.toPandas()
len(pandasDF1)

# concat both data
pandasDF = pd.concat([pandasDF1, pandasDF2])
pandasDF.head(3)

pandasDF['TARGET'].value_counts().sort_values()

plt.figure(figsize=(7,7))

plt.pie(pandasDF['TARGET'].value_counts().sort_values(), labels=['Churn', 'No Churn'], colors=['#ff6666',
'#99ff99'], wedgeprops={'linewidth': 7, 'edgecolor': 'white'}, autopct='%1.1f%%')

# Transforming the pandas Dataframe to a pyspark Dataframe
pysparkSqlDF = spSession.createDataFrame(pandasDF)
```

Correlation

```
# Correlation between features
# Getting all my data into a dataframe:
#telecomDF4.take(5)

# First I'll get each column of telecomDF4
for i in pysparkSqlDF.columns:
   # for each one I'll 'select(i)', and use the 'take' action from [0][0] combination as if it was a matrix
   if not(isinstance(pysparkSqlDF.select(i).take(1)[0][0], str)):
      print("Correlation between Target(Churn) with: ", i, pysparkSqlDF.stat.corr('TARGET', i))
```

Pre-Processing the Dataset

```
# Pre-Processing my Dataset

# Criando um LabeledPoint (target, Vector[features])
# Removing not relevant columns or with low correlation to the model, this way I choose the columns I
want the model to have (just observe the selection done in Vectors.dense(....))

# ATTENTION TO THIS VERY IMPORTANT CONCEPT!!!!!!!!!
# SOME APACHE SPARK MACHINE LEARNING ALGORITHMS (MAINLY THE REGRESSION TYPE) NEED DATA
TO BE
# IN A SPECIFIC FORMAT TO PERFORM THE TRAINING(FIT).
# TO THIS HAPPEN I NEED TO DELIVER TO THE MODEL THE DATA IN VECTOR FORMAT, SPECIFICALLY
USING A
# DENSE OR SPARSE VECTOR. THIS BECAUSE APACHE SPARK WORKS WITH CLUSTER AND THEN IT WILL
```

```
# DISTRIBUTE THESE DATA BETWEEN MACHINES.
from pyspark.ml.linalg import Vectors

def transformFeatures(row):
   obj = (row['TARGET'], Vectors.dense(row['INTPLAN'], row['NBVMMSG'], row['TTDAYMIN'],
row['TTDAYCHARGE'], row['(TTDAYMIN / TTDAYCALLS)'], row['TTEVEMIN'], row['TTEVECHARGE'],
row['NBCSCALLS']) )
   return obj

# HERE telecomDF4 IS A DATAFRAME, BUT IS BEING CONVERTED TO AN rdd AND THEN I'M ABLE TO USE
THE map FUNCTION
telecomRDD4 = pysparkSqlDF.rdd.map(transformFeatures)

telecomRDD4.take(5)

# CONVERTING TO DATAFRAME AGAIN
telecomDF5 = spSession.createDataFrame(telecomRDD4, ['TARGET', 'FEATURES'])
telecomDF5.show(10)
telecomDF5.cache()
```

Machine Learning Modeling

Train/Test Split

```
# Now entering the Machine Learning Process
import random

random.seed(9)
(train_data, test_data) = telecomDF5.randomSplit([0.7, 0.3])
print(train_data.count(), "data to train my model")
print(test_data.count(), "data to test my model")

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have
#train_data.select('TARGET').distinct().show()

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have in the train_data
train_data.cube('TARGET').agg(count('TARGET').alias('Qt of Data')).orderBy('TARGET').show()

# Looking for how much data of Churn 'yes'(1) and 'no'(2) we have in the test_data
test_data.cube('TARGET').agg(count('TARGET').alias('Qt of Data')).orderBy('TARGET').show()
```

Creating and Training Model

```
# Decision Tree ML Model
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
import random
```

```
# Avoiding Underfitting and Overfitting in Decision Trees
for depth in range(1,15):
    model_v1 = DecisionTreeClassifier(maxDepth=depth, labelCol= 'TARGET', featuresCol= 'FEATURES')
    model_v1_train = model_v1.fit(train_data)
    model_v1_prediction = model_v1_train.transform(test_data)
    evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                    labelCol = 'TARGET',
                                    metricName = 'accuracy')
    acc = evaluator.evaluate(model_v1_prediction)
    print('In Depth %.0f this model is %.2f%% Accurate.' %(depth, acc*100))

# Better Depth is 5
# Building the model
model_v1 = DecisionTreeClassifier(maxDepth=5, labelCol= 'TARGET', featuresCol= 'FEATURES')

# Training my Model
model_v1_train = model_v1.fit(train_data)

print(model_v1_train.numNodes, "Nodes")
print(model_v1_train.depth, "Depth")
```

Predictions and Evaluation

```
# Making Predictions with test_data
model_v1_prediction = model_v1_train.transform(test_data)
model_v1_prediction.select('prediction', 'TARGET').collect()

# Evaluating my model
evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                labelCol = 'TARGET',
                                metricName = 'accuracy')
acc = evaluator.evaluate(model_v1_prediction)
print('This model is %.2f%% Accurate.' %(acc*100))

# Taking the Confusion Matrix
model_v1_prediction.groupBy('TARGET', 'prediction').count().show()
```

Plotting Confusion Matrix

```
model_v1_prediction.head(5)

type(model_v1_prediction)

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = model_v1_prediction.toPandas()

# Getting Necessarry Data from model_v1_prediction
Actual = pandasDF.ix[:, 'TARGET']
```

```
Predic = pandasDF.ix[:, 'prediction']

data = {'Actual': Actual,
        'Predicted': Predic
        }

confMatDF = pd.DataFrame(data, columns=['Actual', 'Predicted'])
confMatDF.head(5)

# Renaming Target (1-> Churn, 2-> No Churn)
def targetTransformation(lst):
    if lst == 1:
        return 'Churn'
    else:
        return 'No Churn'

auxList1 = list(confMatDF['Actual'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Actual'] = auxList2

auxList1 = list(confMatDF['Predicted'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Predicted'] = auxList2

confusionMatrix = pd.crosstab(confMatDF.Actual, confMatDF.Predicted, rownames=['Actual Value'],
colnames=['Predicted Value'] )
confusionMatrix

confusionMatrix.values.sum()

# Looking as Percentage of Churn and No Churn
import numpy as np

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

status = ['True Positive', 'False Negative', 'False Positive', 'True Negative']
val    = [x for x in confusionMatrix.values.flatten()]
percent= ['{:.2%}'.format(x) for x in (confusionMatrix.values.flatten()/confusionMatrix.values.sum())]

labels = [f'{q1}\n{q2}\n{q3}' for q1, q2, q3 in zip(status, val, percent)]
labels = np.asarray(labels).reshape(2,2)

sns.heatmap((confusionMatrix/confusionMatrix.values.sum()), annot=labels, fmt='', cmap='BuGn',
cbar=False)
```

```
sns.set(font_scale=1.5)
plt.title('Confusion Matrix')
```

New and Unseen Data

Loading Data

```
## Datasets ------------------------------------------------------------------------------
# Loading data and getting an RDD
# textFile CREATES AN RDD
telecomRDD = sc.textFile("projeto4_telecom_teste.csv")

# Optimizing performance using cache on RDD
telecomRDD.cache()

# Making an Action
telecomRDD.count()

# Making another Action
telecomRDD.take(5)
# Note that the data is between simple aspects '' (each line), indicating they are strings, so I'll convert it
further using a function I'll create
```

Feature Engineering

```
# Cleaning data - Removing first line
telecomRDD2 = telecomRDD.filter(lambda x: "state" not in x)
telecomRDD2.count()

# Looking Again at my data
telecomRDD2.take(5)

# Looking to ALL data and trying to figure out if there is missing values (?, NaN, "", ...)
telecomRDD2.collect()
# No Missing Values
```

Preparing Data to Dataframe

```
# Feature Engineering I - Function
def featengI(data):

    # Dividing data into columns, separating by "," character
    dataList = data.split(",")

    ID = (dataList[0])              # ID
    STATE = str(dataList[1])        # State -> North American Abbreviated States (HI - Hawaii / MT -
Montana / OH - Ohio)
```

```
    ACCLGT = int(dataList[2])       # Account_Length
    AREACODE = (dataList[3])        # Phone Number Area Code
    #PHONENB = (dataList[4])        # Phone_No -> Phone Number (IT IS NOT IN THE test-train DATA)
    #INTPLAN = (dataList[4])        # International_Plan -> "yes" or "no" (Convert yes->1 or no->2)
    INTPLAN = 1.0 if dataList[4] == '"yes"' else 2.0
    #VMPLAN = (dataList[5])         # Voice_Mail_Plan -> "yes" or "no" (Convert yes->1 or no->2) I THINK
THIS FEATURE IS NOT HELPING AND CAN BE ELIMINATED BECAUSE THE NEXT VARIABLE (NBVMMSG)
ONLY HAS VALUES WHEN THIS IS '1', IT'S LIKE DOUBLE INFORMATION ABOUT THE SAME THING
    VMPLAN = 1.0 if dataList[5] == '"yes"' else 2.0
    NBVMMSG = int(dataList[6])      # Number_Voice_Mail_Messages
    TTDAYMIN = float(dataList[7])   # Total_Day_minutes
    TTDAYCALLS = int(dataList[8])   # Total_Day_Calls
    TTDAYCHARGE = float(dataList[9])  # Total_Day_Charge
    TTEVEMIN = float(dataList[10])   # Total_Eve_Minutes
    TTEVECALLS = int(dataList[11])   # Total_Eve_Calls
    TTEVECHARGE = float(dataList[12]) # Total_Eve_Charge
    TTNGTMIN = float(dataList[13])   # Total_Night_Minutes
    TTNGTCALLS = int(dataList[14])   # Total_Night_Calls
    TTNGTCHARGE = float(dataList[15]) # Total_Night_Charge
    TTINTMIN = float(dataList[16])   # Total_Intl_Minutes
    TTINTCALLS = int(dataList[17])   # Total_Intl_Calls
    TTINTCHARGE = float(dataList[18]) # Total_Intl_Charge
    NBCSCALLS = int(dataList[19])    # Number_Customer_Service_Calls
    #TARGET = (dataList[20])        # Churn (Target) -> "yes" or "no" (Convert yes->1 or no->2)
    TARGET = 1 if dataList[20] == '"yes"' else 2

    # Creating 'lines' using the 'Row' function, preparing to the dataframe analysis, cleaning and
converting the data from string to float
    lines = Row(ID = ID, STATE = STATE, ACCLGT = ACCLGT, AREACODE = AREACODE,
                INTPLAN = INTPLAN, VMPLAN = VMPLAN, NBVMMSG = NBVMMSG,
                TTDAYMIN = TTDAYMIN, TTDAYCALLS = TTDAYCALLS, TTDAYCHARGE = TTDAYCHARGE,
                TTEVEMIN = TTEVEMIN, TTEVECALLS = TTEVECALLS, TTEVECHARGE = TTEVECHARGE,
                TTNGTMIN = TTNGTMIN, TTNGTCALLS = TTNGTCALLS, TTNGTCHARGE = TTNGTCHARGE,
                TTINTMIN = TTINTMIN, TTINTCALLS = TTINTCALLS, TTINTCHARGE = TTINTCHARGE,
                NBCSCALLS = NBCSCALLS, TARGET = TARGET)

    return lines

# Applying function above to RDD without headers
telecomRDD3 = telecomRDD2.map(featengl)

# Looking at the result
telecomRDD3.cache
telecomRDD3.take(1)
```

```
# All data are in String format, I'll convert to Integer or Float as needed
#schema = StructType([
#    StructField('ID', StringType(), True)
#])
```

Creating a Pyspark Dataframe

```
# Creating a Dataframe SO I'M ABLE TO USE THE select FUNCTION FROM SparkSQL
telecomDF = spSession.createDataFrame(telecomRDD3)

type(telecomDF)

# Printing telecomDF Object Type and each Row Type
print(telecomDF)
# As noted SOME data (ID, STATE) are in String format, I'll convert to Integer or Float as needed

telecomDF.printSchema()

telecomDF.show(5)
```

Removing Double Quotes in Strings

```
# Function to Remove DoubleQuote (DQ) so I'll transform from String to Integer
def removingDQ(stringData):
    return stringData.replace('"', "")

# Using udf -> User Defined Function (Turning Python Functions into PySpark Functions)
# Look at this -> https://changhsinlee.com/pyspark-udf/
from pyspark.sql.functions import udf
udf_removingDQ = udf(removingDQ, StringType())

telecomDF2 = telecomDF.withColumn('ID', udf_removingDQ(telecomDF['ID'])).withColumn('STATE',
udf_removingDQ(telecomDF['STATE']))
telecomDF2.show(30)

# Now that necessary data doesn't have DQ I'm able to perform casting to transform from one type to
another
# As follows there are 2 ways to make this transformation
```

Transforming Features (Two Possibilities)

My Choice

```
# I'll choose the second one
telecomDF4 = telecomDF2.select( telecomDF2['ID'].cast(IntegerType()).alias('ID'),
                        telecomDF2['STATE'],
                        telecomDF2['ACCLGT'],
                        telecomDF2['INTPLAN'],
                        telecomDF2['VMPLAN'],
                        telecomDF2['NBVMMSG'],
                        telecomDF2['TTDAYMIN'],
```

```
                    telecomDF2['TTDAYCALLS'],
                    telecomDF2['TTDAYCHARGE'],
                    telecomDF2['TTDAYMIN']/telecomDF2['TTDAYCALLS'],
                    telecomDF2['TTEVEMIN'],
                    telecomDF2['TTEVECALLS'],
                    telecomDF2['TTEVECHARGE'],
                    telecomDF2['TTEVEMIN']/telecomDF2['TTEVECALLS'],
                    telecomDF2['TTINTMIN'],
                    telecomDF2['TTINTCALLS'],
                    telecomDF2['TTINTCHARGE'],
                    telecomDF2['TTINTMIN']/telecomDF2['TTINTCALLS'],
                    telecomDF2['NBCSCALLS'],
                    telecomDF2['TARGET']
                    )
telecomDF4.show(3)

# Registering the dataframe as a Temp Table (so i'll be able to use queries SQL ANSI)
# HERE I'M CREATING A REAL TABLE, IT WILL EXIST IN MEMORY, NOT ONLY THE DATAFRAME
telecomDF4.createOrReplaceTempView("telecomTB")
```

Data Balance

```
# Before entering in the Machine Learning Model let's see if we have balanced data so my model will be
fair in predicting both Churn and No Churn
pysparkSqlDF = spSession.sql("select TARGET, count(ID) as INSTANCES \
                    from telecomTB \
                    group by TARGET")
pysparkSqlDF.show()

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = pysparkSqlDF.toPandas()

plt.figure(figsize=(7,7))

plt.pie(pandasDF['INSTANCES'], labels=['Churn', 'No Churn'], colors=['#ff6666', '#99ff99'],
wedgeprops={'linewidth': 7, 'edgecolor': 'white'}, autopct='%1.1f%%')

# The test data provided also has few Churn Data, It's more likeble that the model will have a great
acurracy.
# But it's important to notice that in a real environment I highly recommend to acquire more Churn data.
```

Correlation

```
# Correlation between features
# Getting all my data into a dataframe:
#telecomDF4.take(5)

# First I'll get each column of telecomDF4
```

```
for i in telecomDF4.columns:
   # for each one I'll 'select(i)', and use the 'take' action from [0][0] combination as if it was a matrix
   if not(isinstance(telecomDF4.select(i).take(1)[0][0], str)):
      print("Correlation between Target(Churn) with: ", i, telecomDF4.stat.corr('TARGET', i))
```

Pre-Processing the Dataset

```
# Pre-Processing my Dataset

# Criando um LabeledPoint (target, Vector[features])
# Removing not relevant columns or with low correlation to the model, this way I choose the columns I
want the model to have (just observe the selection done in Vectors.dense(....))

# ATTENTION TO THIS VERY IMPORTANT CONCEPT!!!!!!!!!
# SOME APACHE SPARK MACHINE LEARNING ALGORITHMS (MAINLY THE REGRESSION TYPE) NEED DATA
TO BE
# IN A SPECIFIC FORMAT TO PERFORM THE TRAINING(FIT).
# TO THIS HAPPEN I NEED TO DELIVER TO THE MODEL THE DATA IN VECTOR FORMAT, SPECIFICALLY
USING A
# DENSE OR SPARSE VECTOR. THIS BECAUSE APACHE SPARK WORKS WITH CLUSTER AND THEN IT WILL
# DISTRIBUTE THESE DATA BETWEEN MACHINES.
from pyspark.ml.linalg import Vectors

def transformFeatures(row):
   obj = (row['TARGET'], Vectors.dense(row['INTPLAN'], row['NBVMMSG'], row['TTDAYMIN'],
row['TTDAYCHARGE'], row['(TTDAYMIN / TTDAYCALLS)'], row['TTEVEMIN'], row['TTEVECHARGE'],
row['NBCSCALLS']) )
   return obj

# HERE telecomDF4 IS A DATAFRAME, BUT IS BEING CONVERTED TO AN rdd AND THEN I'M ABLE TO USE
THE map FUNCTION
telecomRDD4 = telecomDF4.rdd.map(transformFeatures)

telecomRDD4.take(5)

# CONVERTING TO DATAFRAME AGAIN
telecomDF5 = spSession.createDataFrame(telecomRDD4, ['TARGET', 'FEATURES'])
telecomDF5.show(10)
telecomDF5.cache()
```

Machine Learning Modeling

```
# Making Predictions with test_data
new_data_prediction = model_v1_train.transform(telecomDF5)
# Evaluating my model
evaluator = MulticlassClassificationEvaluator(predictionCol= 'prediction',
                                labelCol = 'TARGET',
                                metricName = 'accuracy')
```

```
acc = evaluator.evaluate(new_data_prediction)
print('This model is %.2f%% Accurate.' %(acc*100))

# Taking the Confusion Matrix
new_data_prediction.groupBy('TARGET', 'prediction').count().show()
Plotting Confusion Matrix
new_data_prediction.head(5)

type(new_data_prediction)

# Transforming to Pandas so I'll plot some analysis using Seaborn
pandasDF = new_data_prediction.toPandas()

# Getting Necessarry Data from model_v1_prediction
Actual = pandasDF.ix[:, 'TARGET']
Predic = pandasDF.ix[:, 'prediction']

data = {'Actual': Actual,
        'Predicted': Predic
        }

confMatDF = pd.DataFrame(data, columns=['Actual', 'Predicted'])
confMatDF.head(5)

# Renaming Target (1-> Churn, 2-> No Churn)
def targetTransformation(lst):
   if lst == 1:
      return 'Churn'
   else:
      return 'No Churn'

auxList1 = list(confMatDF['Actual'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Actual'] = auxList2

auxList1 = list(confMatDF['Predicted'])
auxList2 = list(map(targetTransformation, auxList1))
confMatDF['Predicted'] = auxList2

confusionMatrix = pd.crosstab(confMatDF.Actual, confMatDF.Predicted, rownames=['Actual Value'],
colnames=['Predicted Value'] )
confusionMatrix

confusionMatrix.values.sum()
```

```
# Looking as Percentage of Churn and No Churn
import numpy as np

# Setting width and height of the plot-figure
plt.figure(figsize=(18,10))

status = ['True Positive', 'False Negative', 'False Positive', 'True Negative']
val    = [x for x in confusionMatrix.values.flatten()]
percent= ['{:.2%}'.format(x) for x in (confusionMatrix.values.flatten()/confusionMatrix.values.sum())]

labels = [f'{q1}\n{q2}\n{q3}' for q1, q2, q3 in zip(status, val, percent)]
labels = np.asarray(labels).reshape(2,2)

sns.heatmap((confusionMatrix/confusionMatrix.values.sum()), annot=labels, fmt='', cmap='BuGn',
cbar=False)
sns.set(font_scale=1.5)
plt.title('Confusion Matrix')
# As notice, a balanced dataset is better in predicting Churn and lowering False Negative preditions
# because the machine learning model will be trained to give better results.
```