Data Engineering

Overview

- The Machine Learning Process
- Connecting Spark with Postgres
- Data Cleaning
- Feature Engineering
  - Variable Types
  - Summary Statistics
  - Five-number Summary and Boxplots
  - Numeric Variables
    - Identifying High Correlations
    - Handling Outliers
  - Categorical Variables and Categories' Encoding
  - Assembling Features into One Vector
  - Data Scaling
  - Data Pipelines

The Machine Learning Process

Let's look at a real-life example of machine learning. Imagine you run a website where people sell their cars. You want your system to suggest fair starting prices for sellers when they post ads. Regression analysis can help in this case by using data from past sales, looking at car features and prices, and finding patterns between them. However, your database doesn't have enough ads, so you decide to gather car prices from public sources. You find many useful car sale records online, but much of the data is in CSV files, and a lot of it is in PDF and Word documents (with car sale listings).

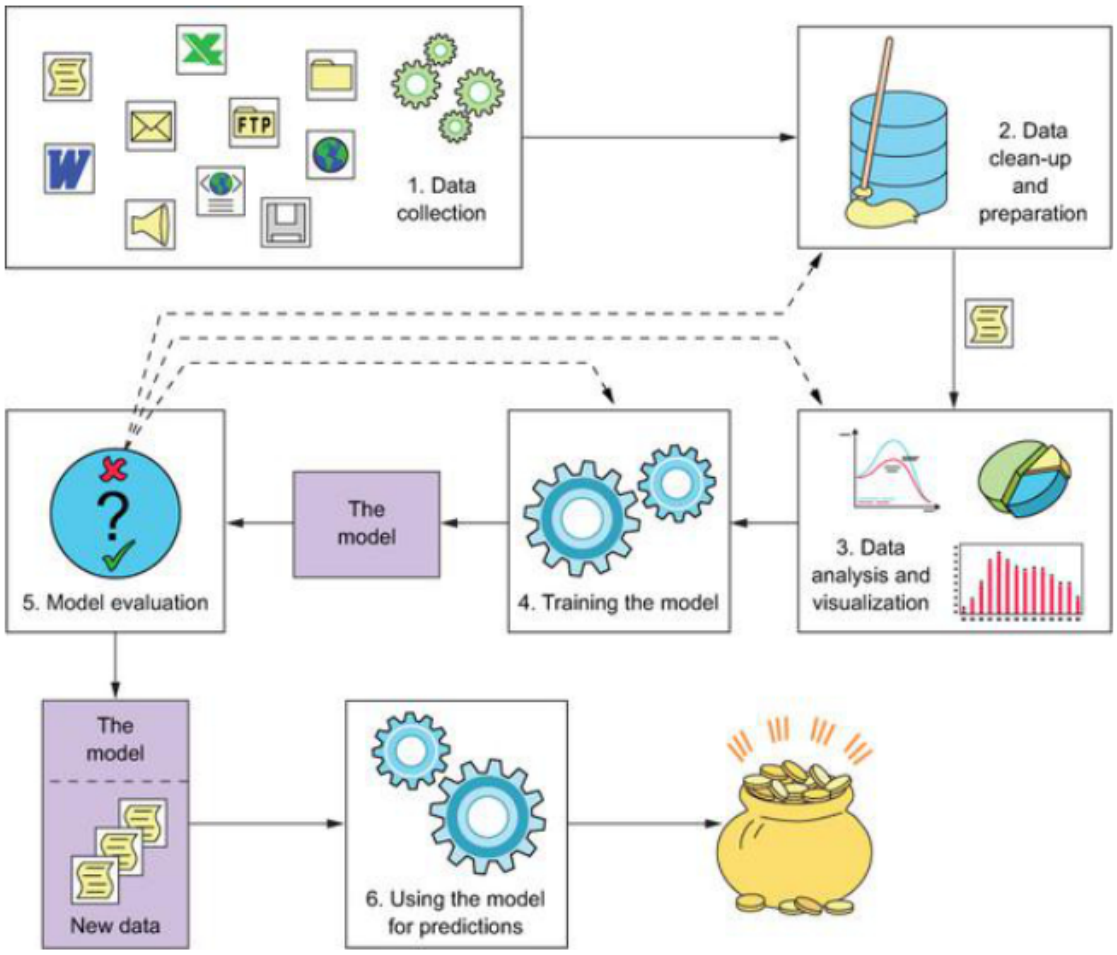
First, you go through the PDFs and Word documents to find and match similar details, like manufacturer, model, and make. Since a regression model can't work with text values (like 'automatic' or 'manual'), you create a method to turn these values into numbers. Then, you realize that some important details, like the year the car was made, are missing from certain records, so you choose to remove those incomplete records from your dataset.

Once you've cleaned and stored the data, you begin to explore different aspects of it—like how the fields relate to each other and their patterns (this helps reveal underlying connections within the data). Next, you choose the right type of regression analysis to apply

Suppose you pick linear regression because the relationships in the data seem to be linear based on your earlier analysis. Before creating the model, you normalize and scale the data (I'll explain how and why this is important later) and divide it into training and validation sets. You then train your model using the training data, using past data to adjust the model's settings to predict future prices that are unknown. After training, you end up with a functional linear regression model.

However, when you test the model on your validation dataset, the performance is disappointing. You adjust some of the training parameters, test the model again, and continue this cycle until the model performs satisfactorily. Once it's working well, you integrate the model into your web application. Soon after, you start receiving emails from clients curious about how you achieved this or from those unhappy with inaccurate predictions.

What this example illustrates is that a machine-learning project consists of multiple steps. Although typical steps are shown in the following figure:



Machine Learning Modeling

The entire process can usually be broken down into the following:

Collecting data: First the data needs to be gathered from various sources. The sources can be log files, database records, signals coming from sensors, and so on. Spark can help load the data from relational databases, CSV files, remote services, and distributed file systems like HDFS, or from real-time sources using Spark Streaming.

Cleaning and preparing data: Data isn't always available in a structured format appropriate for machine learning (text, images, sounds, binary data, and so forth), so you need to devise and carry out a method of transforming this unstructured data into numerical features. Additionally, you need to handle missing data and the different forms in which the same values can be entered (for example, VW and Volkswagen are the same carmaker). Often, data also needs to be scaled so that all dimensions are of comparable ranges.

Analyzing data and extracting features: Next you analyze the data, examine its correlations, and visualize them (using various tools) if necessary. (The number of dimensions may be reduced in this step if some of them don't bring any extra information: for example, if they're redundant.) You then choose the appropriate machine-learning algorithm (or set of algorithms) and split the data into training and validation subsets—this is important because you'd like to see how the model behaves on the data not seen during the training phase. Or you decide on a different cross-validation method, where you continuously split the dataset into different training and validation datasets and average the results over the rounds.

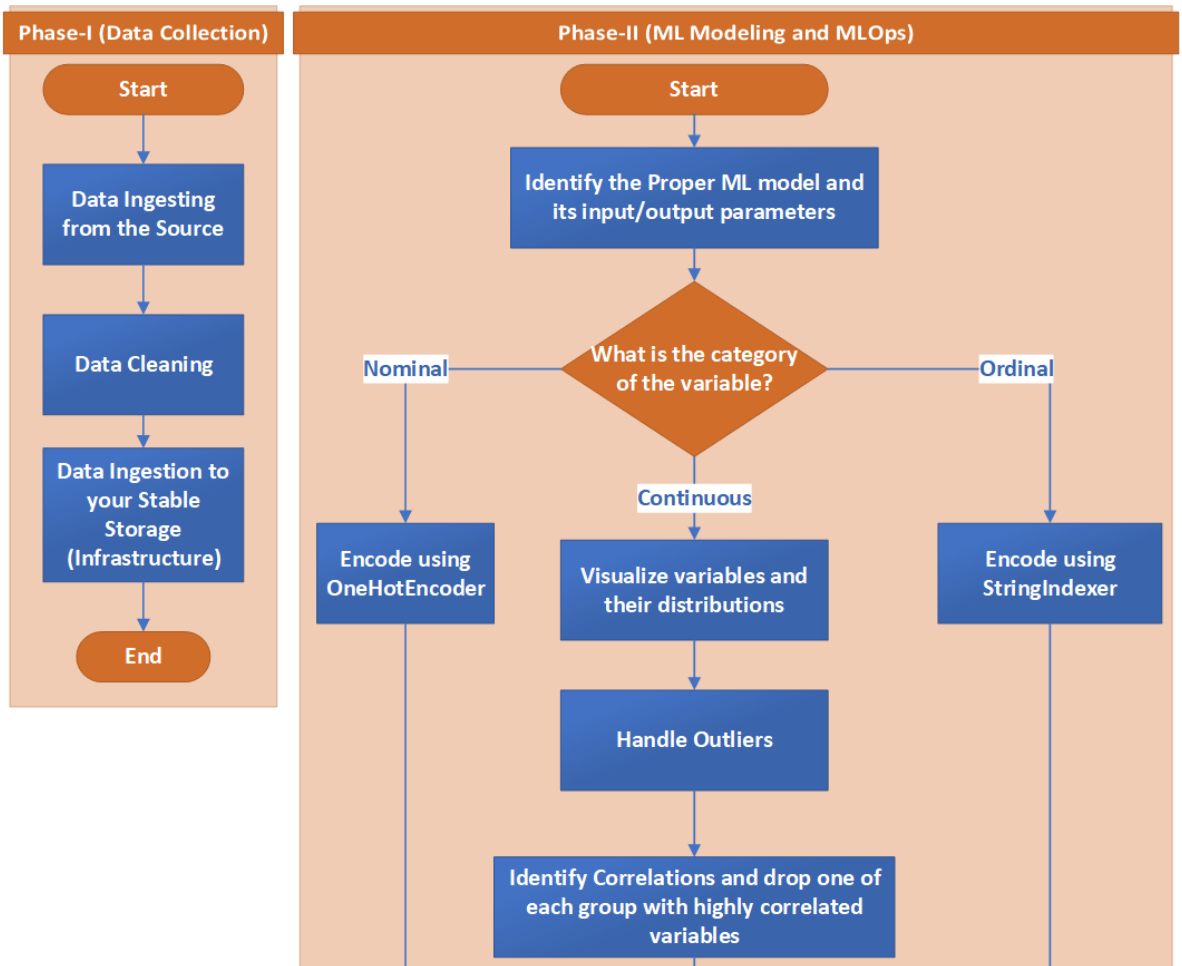
Training the model: You train a model by running an algorithm that learns a set of algorithm-specific parameters from the input data.

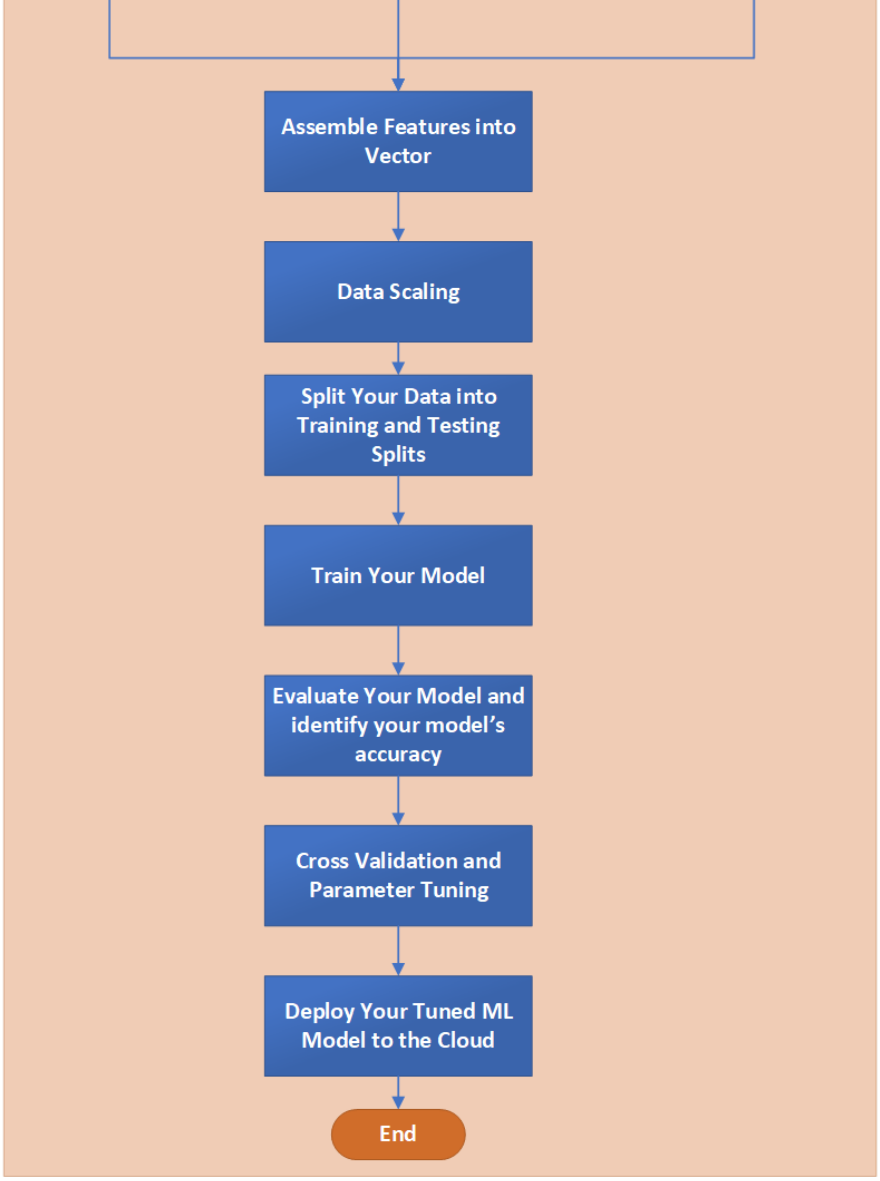
Evaluating the model: You then put the model to use on the validation dataset and evaluate its performance according to some criteria. At this point, you may decide that you need more input data or that you need to change the way features were extracted. You may also change the feature space or switch to a different model. In any of these cases, you go back to step 1 or step 2.

Using the model: Finally, you deploy the built model to the production environment of your website.

The mechanics of using an API (Spark or some other machine-learning library) to train and test the models is only the last and the shortest part of the process. Equally important are collection, preparation, and analysis of data, where knowledge about the problem domain is needed.

Process Flow for Big-Data Machine Learning Process





Process Flow for Big-Data Machine Learning Modeling

Food for Thought: Why do you think a 2-phase process is necessary in big-data ML modeling?!

Phase-I Ingest Data from CSV and Insert them into PostgreSQL

1. Data Ingestion

In [ ]:
```
!python -m wget https://www.andrew.cmu.edu/user/mfarag/14763/KDDTrain+.txt
```

Initialize the Application

In [ ]:
```
# Uncomment the following lines if you are using Windows!
#import findspark
#findspark.init()
#findspark.find()

import pyspark

from pyspark.sql import SparkSession
from pyspark import SparkContext, SQLContext

appName = "Big Data Analytics"
master = "local"

# Create Configuration object for Spark.
conf = pyspark.SparkConf()\
    .set('spark.driver.host','127.0.0.1')\
    .setAppName(appName)\
    .setMaster(master)

# Create Spark Context with the new configurations rather than relying on the default on
sc = SparkContext.getOrCreate(conf=conf)

# You need to create SQL Context to conduct some database operations like what we will s
sqlContext = SQLContext(sc)

# If you have SQL context, you create the session from the Spark Context
spark = sqlContext.sparkSession.builder.getOrCreate()
```

Read-in the Dataset

In [ ]:
```
# Load data from csv to a dataframe on a local machine.
# header=False means the first row is not a header
# sep=',' means the column are seperated using ','
col_names = ["duration","protocol_type","service","flag","src_bytes",
    "dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
    "logged_in","num_compromised","root_shell","su_attempted","num_root",
    "num_file_creations","num_shells","num_access_files","num_outbound_cmds",
    "is_host_login","is_guest_login","count","srv_count","serror_rate",
    "srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
    "diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
    "dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
    "dst_host_rerror_rate","dst_host_srv_rerror_rate","classes","difficulty_level"]

df = spark.read.csv("KDDTrain+.txt",header=False, inferSchema= True).toDF(*col_names)
```

For this lecture's purposes, We will insert the data into the table using the overwrite mode

If the overwrite mode is not working for you, please populate the table by creating the SQL table and using the append mode to populate it.

In [ ]:
```
db_properties={}
#update your db username
db_properties['username']="postgres"
#update your db password
db_properties['password']="bigdata"
#make sure you got the right port number here
db_properties['url']= "jdbc:postgresql://localhost:5432/postgres"
#make sure you had the Postgres JAR file in the right location
db_properties['driver']="org.postgresql.Driver"
db_properties['table']= "NSLKDD"


df.write.format("jdbc")\
.mode("overwrite")\
.option("url", db_properties['url'])\
.option("dbtable", db_properties['table'])\
.option("user", db_properties['username'])\
.option("password", db_properties['password'])\
.option("Driver", db_properties['driver'])\
.save()
```

Now, read the data back!

In [ ]:
```
df_read = sqlContext.read.format("jdbc")\
    .option("url", db_properties['url'])\
    .option("dbtable", db_properties['table'])\
    .option("user", db_properties['username'])\
    .option("password", db_properties['password'])\
    .option("Driver", db_properties['driver'])\
    .load()

df_read.show(1, vertical=True)
```

2. Now, let's go ahead and talk about Data Cleaning and Preprocessing.

It's important to clean your data before storing them in the database tables. In some cases, it's OK to conduct data cleaning before before you conduct feature engineering. In Data cleaning, we typically need to conduct the following activities:

- Rename the column names to remove special characters and blanks - if any - with an underscore "_";
- Choose the correct data type for your columns and cast column data types as needed
- Drop the columns that are not of interest to you (duplicated, not included in research, etc.)
- Handle missing values and N/As

2.1 Let's test renaming columns, casting column data types and dropping duplicate rows

You may use these techniques in other activities (e.g., project or HW assignments)

In [ ]:
```
# Column Rename
renamed_columns_df = df_read.withColumnRenamed("rerror_rate","newly_renamed_rerror_rate"
# Casting one column type
casted_types_df = (renamed_columns_df.withColumn("new_column_srv_serror_rate", \
                    renamed_columns_df["srv_serror_rate"] \
                    .cast("integer")).drop("srv_serror_rate")
                    .distinct() # deleting duplicate rows
           )
```

In [ ]:
```
casted_types_df.printSchema()
```

Please note that these column renames are for experimental purposes only to show you how to use these functions. However, this dataset doesn't need these column renames. So, we will go back to use our df_read object.

In [ ]:
```
# We are assigning the df_read to casted_types_df because no data type changes were need
casted_types_df = df_read
```

In [ ]:
```
casted_types_df.show(1,vertical=True)
```

2.2.1 Types of Missing Values

There are several ways to handle missing values in your dataset. You need to stay vigilent in selectin the approach since some of these methods may impact the quality of your data.

- Missing Completely at Random (MCAR): The probability of missing values is the same across all the variables.
- Missing at Random (MAR): Like MCAR but it is possible to predict the missing value based on some other variables.
- Not Missing at Random (NMAR): This can be handled by studying the root cause of missing

2.2.2 Reasons for Missing Values

- Data may be missing for some of the time period of the analysis.
- Events not happening, such as a student's exam score missing because they didn't take the test.
- Responses omitted for certain survey questions.
- Questions not applicable in some contexts.
- Random gaps in data.

2.2.3 Handling Missing Values

There are several ways to handle missing values in your dataset. You need to stay vigilent in selectin the approach since some of these methods may impact the quality of your data.

- **Drop Columns:** Removing a column from your dataset is always possible, but you should carefully consider whether to clean it using a different method or to remove it entirely. Your decision should be guided by several factors, such as the importance of the data in the column and the proportion of missing values. As a general guideline, avoid dropping a column if it contains meaningful data in at least 50% of its entries.
- **Drop Rows having Nulls:** This is always an option as well. However, depending on the size of your dataset and the number of rows to be dropped, this "may/may not" be a good option. For rows, you can use a similar threshold (50%) and drop rows with missing values only if it doesn't significantly reduce your sample size
- **Fill the missing values:** This approach is "usually" applicable for numeric columns. This is done using an operation that is called **imputation**. By creating imputed columns, we will create columns which will consist of values that fill the missing value by taking a statistical method such as mean/median of the original columns to fill the missing value.

2.2.3.1 Check NA Values

In some cases, your dataframe is able to detect if your cells/fields are empty or holding a missing value. In this case, missing values can be spotted using isNull or is NaN

In [ ]:
```
from pyspark.sql.functions import *

null_counts_plays_df = df_read.select([count(when(isnan(c) | col(c).isNull(), c)).alias(
                        for c in df_read.columns])

null_counts_plays_df.show(truncate=False, vertical=True)
```

So, our NSL-KDD dataset is clean but let's assume some hypothetical scenarios here.

2.2.3.2 Let's play with some null value handling. To drop all rows with null value(s):

In [ ]:
```
casted_types_df_with_na_dropped_rows = casted_types_df.na.drop()
casted_types_df_with_na_dropped_rows.show(1,vertical=True)
```

2.2.3.3 To drop rows based on number of NAs in the row:

In [ ]:
```
casted_types_df_with_na_dropped_rows = casted_types_df.na.drop(how="any", thresh=2)
casted_types_df_with_na_dropped_rows.count()
```

2.2.3.4 What if the number of missing values is large or we don't want to drop them due to a potential significant impact to our sample size? Imputation can be an option. Let's go ahead and assume we have null values in **src_bytes** column and we would like to **impute** them.

In [ ]:
```
#In some cases, your missing values are recoded with a string value (and not recognized
# The below code can be partially used outside of imputation context
# to replace a string value with NA using regex_replace
missing_value = "NA"
df_with_substituted_na = (casted_types_df_with_na_dropped_rows\
    .withColumn('src_bytes', \
                when(casted_types_df_with_na_dropped_rows.src_bytes==missing_value,\
                    regexp_replace(casted_types_df_with_na_dropped_rows.src_bytes,missin
                .otherwise(casted_types_df_with_na_dropped_rows.src_bytes))\
                )

df_with_substituted_na.show(1, vertical=True)
```

In [ ]:
```
from pyspark.ml.feature import Imputer

columns_to_be_imputed = ["src_bytes"]
value_not_in_dataset = -200

# Replace None/Missing Value with a value that can't be present in the dataset.
df_with_filled_na = casted_types_df_with_na_dropped_rows.fillna(-200, columns_to_be_impu

#Create new columns with imputed values. New columns will be suffixed with "_imputed"
imputer = Imputer (
            inputCols=columns_to_be_imputed,
            outputCols=["{}_imputed".format(c) for c in columns_to_be_imputed])\
            .setStrategy("median").setMissingValue(value_not_in_dataset)

df_imputed = imputer.fit(df_with_filled_na).transform(df_with_filled_na)
# we will drop the old column without imputation. We have only one column to be imputed
df_imputed_enhanced = df_imputed.drop(columns_to_be_imputed[0])

# We will rename our newly imputed column with the correct name
df_fully_imputed = df_imputed_enhanced.withColumnRenamed("src_bytes_imputed","src_bytes"
```

In [ ]:
```
df_fully_imputed.printSchema()
```

In [ ]:
```
df_fully_imputed.show(1,vertical=True)
```

Phase-II: ML Modeling and MLOps

Now, let's go ahead and talk about Feature Engineering.

The basic process here is:

- Begin by classifying your variables into numerical and categorical
- Then, look at numerical summaries of numerical variables.
- Follow this with an examination to the distribution of each variable individually.
- Then move on to study the relationships among the variables.
- Try to visualize all what you can!

Exploring data for machine learning is a lot similar to exploring data when performing a transformation in the sense that we manipulate the data to uncover some inconsistencies, patterns, or gaps

1. Classify Your variables

- The information is organized in variables.
- A variable is any characteristic of an individual or a case.
- In Spark, each attribute (column) in your dataframe represent a variable
- A variable can take different values for different individuals.

Reminder: Types of Variables

- **Continuous Variables: or quantitative variables.**
- **Categorical Variables:** can be divided into:
  - **Binary Variables:** when you have only two choices (0/1, True/False)
  - **Ordinal Variables:** when the categories have a certain ordering (like low/medium/high)
  - **Nominal Variables:** when the categories have no specific ordering (like the color of an item).

Note the following:

- Identifying your variables as categorical (with the proper sub-type) or continuous has a direct impact on the data preparation and, down the road, the performance of your machine learning model.
- Proper identification is dependent on the context (what does the column mean?) and how you want to encode its meaning.
- Don't worry if you don't get it right the first time. You can always come back and touch up your feature types.

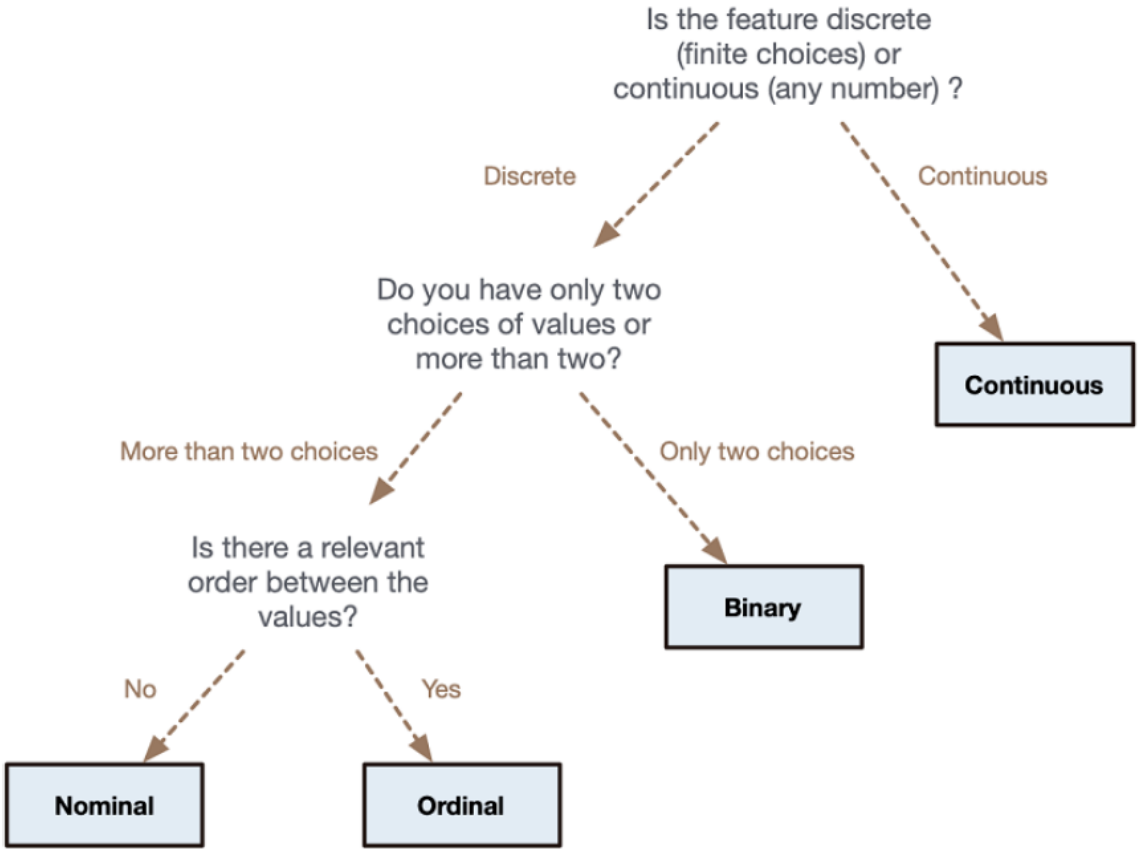


Image taken from https://livebook.manning.com/#!/book/data-analysis-with-python-and-pyspark/discussion

2. Creating a Summary Table

- Checking the summary statistics for our dataframe helps us validate our numerical columns. This is helpful for several reasons such as understanding the variability in your data and the scale of data for each variable.
- Remember, your ML model is mathematical model after all. If you have variable that includes values between 0 and 100, it will have higher influence than a column that has values between 0 and 1.
- You may include the Summary columns for some categorical columns but you would have to ignore them during your analysis

In [ ]:
```
df_fully_imputed.summary().show(truncate=False, vertical=True)
```

Wouldn't it be better to use graphs to look at these numbers rather than looking at it yourself?

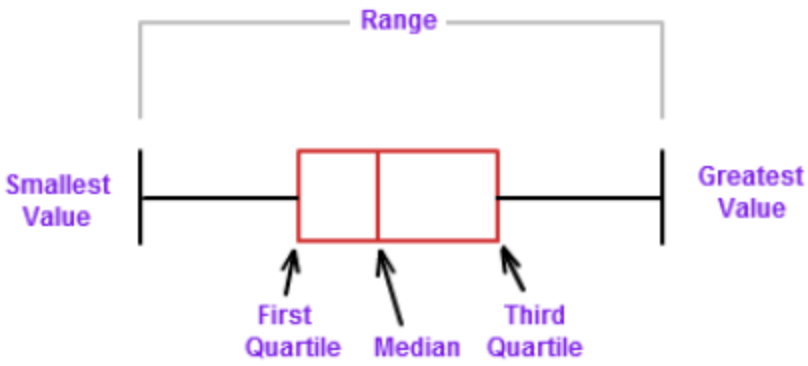
The five-number summary

The five-number summary of a set of observations consists of the smallest observation, the first quartile, the median, the third quartile, and the largest observation, written in order from smallest to largest. In symbols, the five-number summary is

- Minimum
- 25% of your data (Q1)
- Median of your data
- 75% of your data (Q3)
- Maximum

These five numbers give a reasonably complete description of both the center and the spread of the distribution.

Boxplots



Boxplot Structure

A boxplot is a graph of the five-number summary.

- A central box spans the quartiles $Q1$ and $Q3$.
- A line in the box marks the median M.
- Lines extend from the box out to the smallest and largest observations

Boxplots are

- Good for comparing data sets by showing them side-by-side on the same graph.
- Good at showing amount of variability.
- Weak at showing the shape of the distribution (e.g. does not show how many modes there are).
- Location of median indicates symmetry or asymmetry.

Now, let's Draw Boxplots!!

In [ ]:
```
# We will draw boxplots for the numerical features
# Now let's check the datatypes of the dataframe
df_fully_imputed.dtypes
```

In [ ]:
```
numeric_features = [feature[0] for feature in df_fully_imputed.dtypes if feature[1] in (
numeric_features
```

In [ ]:
```
import matplotlib.pyplot as plt

#Extract data and convert them into Pandas for visualization
converted_data = df_fully_imputed[numeric_features].toPandas()

figure = plt.boxplot(converted_data)
```

That looks ugly!!! Let's get a subset of the variables and display them.

In [ ]:
```
spotted_data = df_fully_imputed[numeric_features[36:38]].toPandas()
figure_subset = plt.boxplot(spotted_data)
```

Did you think what the circles in the previous graph represent?

The answer is **Outliers**.

Based on these results, we MAY need to **handle outliers**!!

3. Outliers

An outlier is an observation that is usually large or small relative to the other values in a data set. Identifying outliers is crucial for the selection of the ML model. Some models are more sensitive to outliers than others.

Outliers are typically attributable to one of the following causes:

- The observation is observed, recorded, or entered incorrectly.
- The observation comes from a different population.
- The observation is correct but represents a rare event

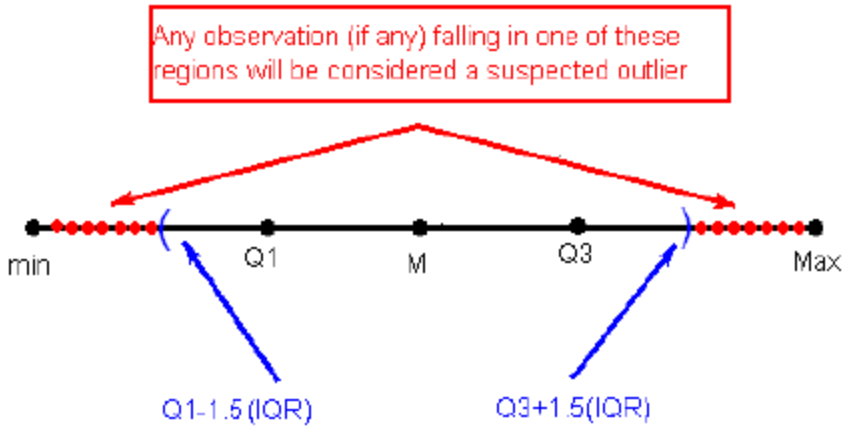
How to spot an Outlier in Numerical Variable?

There are several ways to find outliers. The most important way is the Inter-Quartile Range (IQR). The IQR is a measure of spread that is less sensitive to the influence of extreme values.
The IQR is defined as Q3 - Q1 from your boxplot.

How does the IQR is used in spotting Outliers?

The rule of **1.5×IQR Criterion for outliers**. Call an observation a suspected outlier if it falls:

- more than 1.5×IQR above the 3rd quartile or
- more than 1.5×IQR below the 1st quartile



In [ ]:
```
figure_subset.keys()
```

How to Print Q1 and Q3 Ranges?

In [ ]:
```
# Values printed as comma separated Q1, Q3 for each object.
[item.get_ydata()[1] for item in figure_subset['whiskers']]
```

How to Print the Outliers?

In [ ]:
```
# how to print outliers?!
[item.get_ydata() for item in figure_subset['fliers']]
```

Look through the following code and understand the find_outliers() function

In [ ]:
```
from functools import reduce

def column_add(a,b):
     return  a.__add__(b)

def find_outliers(df):
    # Identifying the numerical columns in a spark dataframe
    numeric_columns = [column[0] for column in df.dtypes if column[1]=='int']

    # Using the `for` loop to create new columns by identifying the outliers for each fe
    for column in numeric_columns:

        less_Q1 = 'less_Q1_{}'.format(column)
        more_Q3 = 'more_Q3_{}'.format(column)
        Q1 = 'Q1_{}'.format(column)
        Q3 = 'Q3_{}'.format(column)

        # Q1 : First Quartile ., Q3 : Third Quartile
        Q1 = df.approxQuantile(column,[0.25],relativeError=0)
        Q3 = df.approxQuantile(column,[0.75],relativeError=0)

        # IQR : Inter Quantile Range
        # We need to define the index [0], as Q1 & Q3 are a set of lists., to perform a
        # Q1 & Q3 are defined seperately so as to have a clear indication on First Quant
        IQR = Q3[0] - Q1[0]

        #selecting the data, with -1.5*IQR to + 1.5*IQR., where param = 1.5 default valu
        less_Q1 =  Q1[0] - 1.5*IQR
        more_Q3 =  Q3[0] + 1.5*IQR

        isOutlierCol = 'is_outlier_{}'.format(column)

        df = df.withColumn(isOutlierCol,when((df[column] > more_Q3) | (df[column] < less


    # Selecting the specific columns which we have added above, to check if there are an
    selected_columns = [column for column in df.columns if column.startswith("is_outlier
    # Adding all the outlier columns into a new colum "total_outliers", to see the total
    df = df.withColumn('total_outliers',reduce(column_add, ( df[col] for col in  selecte

    # Dropping the extra columns created above, just to create nice dataframe., without
    df = df.drop(*[column for column in df.columns if column.startswith("is_outlier")])

    return df
```

In [ ]:
```
# As a reminder, we don't have any null values for the outliers to be handled
numeric_columns = [column[0] for column in df_fully_imputed.dtypes if column[1] in ('int
df_fully_imputed.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in nu
```

In [ ]:
```
df_with_outlier_handling = find_outliers(df_fully_imputed)
df_with_outlier_handling.show(1, vertical=True)
```

In [ ]:
```
df_with_outlier_handling.groupby("total_outliers").count().show()
```

We don't want to lose lots of records for outliers. We will drop the rows with more than 4 outliers.

In [ ]:
```
df_with_substituted_na_and_outliers = df_with_outlier_handling.\
        filter(df_with_outlier_handling['total_Outliers']<=4)
print(df_with_substituted_na_and_outliers.count())
```

4. Variable Distribution

- The distribution of a variable tells us what values it takes and how often it takes these values.
- In any graph of data, look for the overall pattern and for striking deviations from that pattern.
- Overall pattern of a distribution can be described by its shape centre and spread
- An important kind of deviation is **an outlier** an individual value that falls outside the overall pattern.

Try to answer the following questions when looking at the distribution.

- Does the distribution have one or several major peaks, usually called modes A distribution with one major peak is called unimodal
- Is it approximately symmetric or skewed in one direction?

We can use **Seaborn** to visualize the variable distribution along with its **correlation** with other variables.

In [ ]:
```
import seaborn as sb

sb.pairplot(spotted_data)
```

- Distributions show visual correlation of variables and we identified two possible columns to be dropped. Let's take a look at the correlation between all the variables and find the variables with strong correlation.
- **Strong correlation** is identified with **>= 0.8 correlation coefficient.**
- **In order for correlation to work, null values should be handled.**

5. Correlations

Correlation means a mutual relationship between two or more things. Consider data points (x_i , y_i), i = 1,2,...n in a dataset. **The objective of correlation** is to see if large values of "x" are paired with large values of "y" and small values of "x" are paired with small values of "y". If not, check if small values of "x" are paired with large values of "y" and vice versa.

In general, it is recommended to avoid having correlated features in your dataset. Indeed, a group of highly correlated features will not bring additional information (or just very few), but will increase the complexity of the algorithm, thus increasing the risk of errors. Depending on the features and the model, correlated features might not always harm the performance of the model but that is a real risk.

Why would you drop highly correlated features?

In feature engineering, we remove highly correlated features for **storage and speed concerns.**

Now that we handled null values, we can calculate the correlation using the **corr()** function in Pandas dataframe. PySpark has built-in correlation support as well using the Correlation class. However, it doesn't print the values as nicely as corr() function. So, for academic purposes, we will use the corr() function.

In [ ]:
```
correlation_matrix = df_with_substituted_na_and_outliers.toPandas().corr()
print(correlation_matrix)
```

In the previous results, look for strong correlations among variables. Notice the following:

Strong Correlations among:

- serror_rate
- dst_host_serror_rate
- srv_serror_rate
- dst_host_srv_serror_rate

Strong Correlations among:

- rerror_rate
- srv_rerror_rate
- dst_host_rerror_rate
- dst_host_srv_rerror_rate

So, we should keep one variable from each group

In [ ]:
```
df_with_handled_correlations = df_with_substituted_na_and_outliers\
                .drop("dst_host_serror_rate","srv_serror_rate","dst_host_srv_serror_rate
                     "srv_rerror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate"
```

At this point, you should have handled continuous variables. **Next, let's look at categorical variables: Binary, Nominal and Ordinal variables**

6. Handle Binary Variables (by casting them)

If you have boolean variables (i.e. True/False variables), you should convert them to 0's and 1's.

- 0 belongs to False and 1 reflects True values.
- In this dataset,there are no False and True values. They are already converted to 0's and 1's (e.g. is_guest_login variable)
- However, you will have the opportunity to cast binary variables in the homework.

Here is an example on how to handle binary variables

In [ ]:
```
df_with_handled_binary = (df_with_handled_correlations
              .withColumn("is_guest_login_encoded", \
                          df_with_handled_correlations["is_guest_login"].cast("integer")
df_with_handled_binary.select("is_guest_login","is_guest_login_encoded").distinct().show
```

Again, we don't need to handle binary variables in this example and therefore, we don't need is_guest_login_encoded column. So, I'll assign the df_with_handled_binary to its old value. You don't need this step in the HW.

In [ ]:
```
# You don't need this if you are handling binary variables
df_with_handled_binary = df_with_handled_correlations
```

7. Handling Ordinal and Nominal Variables

Many machine learning algorithms cannot work with categorical data directly. The categories must be converted into numbers. This is required for both input and output variables that are categorical.

Converting strings to numbers can be achieved using one of the following **options:**

- StringIndexer from pyspark.ml.feature. This option assigns a number to each level. This may work for problems where there is a natural ordinal relationship between the categories, and in turn the integer values, such as labels for temperature 'cold', 'warm', and 'hot'. In StringIndexers, the most frequent value would get index 0, followed by the next most frequent and so on. However, there may be problems when there is no ordinal relationship and allowing the representation to lean on any such relationship might be damaging to learning to solve the problem. An example might be the labels 'dog' and 'cat'.
  In these cases, we would like to give the network more expressive power to learn a probability-like number for each possible label value. This can help in both making the problem easier for the network to model. When a one hot encoding is used for the output variable, it may offer a more nuanced set of predictions than a single label.

- OneHotEncoder (for Spark >= 3.0) or OneHotEncoderEstimator (for Spark >= 2.3) from pyspark.ml.feature library. A one hot encoding allows the representation of categorical data to be more expressive. This option will create dummy variables for each category, thereby increasing the number of features we work with. In our selected dataset, the special teams play type and offensive information variables need to be encoded.

String Indexer vs. One-Hot Encoder

String Indexing assigns a unique integer value to each category. 0 is assigned to the most frequent category, 1 to the next most frequent value, and so on. We have to define the input column name that we want to index and the output column name in which we want the results.

However, **String Indexers impose an order or rank on your data and therefore, they only fit ordinal variables.** To represent **nominal variables,** we should use **One-Hot encoders.** In Spark, encoding is done as series of tasks that are executed via **pipelines**.

Data Processing Pipelines

It's popular to use several machine learning models together. Therefore, machine learning workflows could get complex and hard to maintain. Data processing pipelines can help reducing the complexity of the machine learning workflow by allowing the reuse of workflow components across several models. A pipeline is a linear sequence of data processing and transformation stages, where each stage can be either a data transformer (for data preparation and feature engineering) or a machine learning estimator (for model training). Pipelines are designed to make it easier to construct, evaluate, and deploy machine learning workflows.

Pipeline Components

The main pipeline components are:

- Transformers
- Estimators

Transformers

Transformers typically refer to components or stages in a machine learning pipeline that are used to transform and preprocess data. These transformers are part of the Spark ML library and they are used to perform various operations on data. These operations include feature extraction, transformation, and scaling, before feeding it into machine learning algorithms. Technically, a Transformer implements a method transform(), which converts one DataFrame into another, generally by appending one or more columns.

Examples of built-in transformers include: Tokenizer, StandardScaler, VectorAssembler, OneHotEncoder, and StringIndexer.

We can also create custom Transformers by subclassing the `Transformer` class (in `pyspark.ml`). The basic syntax for creating a transformer is

In [ ]:
```
from pyspark.ml import Transformer

class myCustomTransformer(Transformer):
    def __init__(self):
        super().__init__()

    def _transform(self, input_df):
        # do some processing steps here
        return output_df

transformer1 =  myCustomTransformer()
transformer2 =  myCustomTransformer()
```

In [ ]:
```
from pyspark.ml import Pipeline
# You can use different types of transformers, estimatorabss as well.
mypipeline = Pipeline(stages=[transformer1, transformer2])
```

Estimators

An estimator is an algorithm or model-building tool that takes a dataset and returns a model. In other words, it's an abstraction of a learning algorithm.

Examples of built-in estimators include

- LogisticRegression: An estimator used for binary classification.
- RandomForestClassifier: An estimator for classification tasks using random forests.
- LinearRegression: An estimator for linear regression tasks.
- KMeans: An estimator for clustering using the K-means algorithm.

Estimator Components

Estimators typically have two main methods: fit and transform.

- The **fit** method is used to train the estimator on a given dataset. It takes a DataFrame as input and returns a model.
- The **transform** method is used to apply the trained model to new data, transforming the input data into predictions or other outputs.

Let's take an example where we try to one-hot encode activity classes in our NSL-KDD dataset

In [ ]:
```
from pyspark.ml.feature import StringIndexer, OneHotEncoder
from pyspark.ml import Pipeline


# We don't have any ordinal variables. Only nominal variables
# first part : transform the columns to numeric
stage_1 = StringIndexer(inputCol= 'classes', outputCol= 'classes_index')
stage_2 = StringIndexer(inputCol= 'protocol_type', outputCol= 'protocol_type_index')
stage_3 = StringIndexer(inputCol= 'service', outputCol= 'service_index')
stage_4 = StringIndexer(inputCol= 'flag', outputCol= 'flag_index')

# second part : one-hot encode the numeric columns
stage_5= OneHotEncoder(inputCols=["classes_index","protocol_type_index",
                                  "service_index","flag_index"],
                       outputCols=['classes_encoded','protocol_type_encoded',
                                   'service_encoded','flag_encoded'])

# setup the pipeline
pipeline = Pipeline(stages=[stage_1, stage_2, stage_3, stage_4, stage_5])

# fit the pipeline model and transform the data as defined
pipeline_model = pipeline.fit(df_with_handled_binary)
df_encoded = pipeline_model.transform(df_with_handled_binary)
```

In order to view/read the output of One-Hot encoding, it's best to convert your mini-dataframe to Pandas

In [ ]:
```
# Notice, the length of our One-Hot Encoding array
print(df_encoded.select("classes").distinct().count())
# In our NSL-KDD dataset, the length is 3. Now, let's view some examples
df_encoded.select("classes","classes_index","classes_encoded")\
                .distinct().toPandas()
```

Read more about One-Hot Encoding and String Indexing from this article:
https://medium.com/@nutanbhogendrasharma/role-of-onehotencoder-and-pipelines-in-pyspark-ml-feature-part-2-3275767e74f0

8. Combining Features into Single Vector

Typically, machine learning models accept the ML input as a vector of features. So, to combine your features into a single vector in Spark, you may use VectorAssembler.

In [ ]:
```
feature_list = df_encoded.drop("classes","classes_index",
                                        "protocol_type","protocol_type_index",
                                        "service","service_index",
                                        "flag","flag_index").columns
print(feature_list)
```

In [ ]:
```
from pyspark.ml.feature import VectorAssembler


vector_assembler = VectorAssembler(
    inputCols=feature_list,
    outputCol="vectorized_features")

df_with_assembled_features = vector_assembler.transform(df_encoded)
```

In [ ]:
```
df_with_assembled_features.select("vectorized_features").show(1, truncate=False)
```

Important Note: Your Machine Learning Target/Outcome Variable MUST NOT be in the Feature List that is Assembled

In [ ]:
```
# Let's take an example if we are trying to predict the activity/attack classes.
# In this case, classes should not be in the feature list.
# Notice we are dropping classes_encoded variable here
feature_list = df_encoded.drop("classes","classes_index",
                                        "protocol_type","protocol_type_index",
                                        "service","service_index",
                                        "flag","flag_index",
                                "classes_encoded").columns

vector_assembler = VectorAssembler(
    inputCols=feature_list,
    outputCol="vectorized_features")

df_with_assembled_features = vector_assembler.transform(df_encoded)
```

In [ ]:
```
df_with_assembled_features.select("vectorized_features","classes_encoded","classes")\
    .distinct().toPandas()
```

Now that we are done with assembling all the features, **let's scale them so they all have standard domains and no feature will have significantly higher impact than another one before we develop our machine learning model.**

9. Data Scaling

In most cases, the numerical features of the dataset do not have a certain range and they differ from each other. In order for a symmetric dataset, scaling is required.

Normalization

Normalization (or min-max normalization) scales all values in a fixed range between 0 and 1. This transformation does not change the distribution of the feature and due to the decreased standard deviations, the effects of the outliers increases. Therefore, before normalization, it is recommended to handle the outliers

Standardization

Standardization (or z-score normalization) scales the values while taking into account standard deviation. If the standard deviation of features is different, their range also would differ from each other. This reduces the effect of the outliers in the features.

$$x_{i,j}^{*} = \frac{x_{i,j} - x_j^{min}}{x_j^{max} - x_j^{min}}$$

Normalization Formulae (Min-Max Scaler)

$$z = \frac{x_i - \mu}{\sigma}$$

Standardization Formulae

Scaling Formulas

In this class, we will focus on standard scalers. StandardScalers standardize features by removing the mean and scaling to unit variance using column summary statistics on the samples in the training set.

The “unit std” is computed using the corrected sample standard deviation, which is computed as the square root of the unbiased sample variance.

To scale data to predict Activity/Attack Classes

The code is shown below.

In [ ]:
```
from pyspark.ml.feature import StandardScaler
standard_scaler = StandardScaler(inputCol= 'vectorized_features', outputCol= 'features')
scaled_model = standard_scaler.fit(df_with_assembled_features)
df_with_scaled_features = scaled_model.transform(df_with_assembled_features)

df_with_scaled_features.select("classes","classes_encoded","features").distinct().toPand
```

And at this point, your data should be ready for your ML model.

Summary of Data Engineering

- 1. Data Ingestion
- 2. Data Cleaning. This includes handling missing values (via dropping or imputation), renaming columns and casting column data types.
- 3. Variable Classification
- 4. For continuous variables,
  - use boxplots to identify if outliers exist and handle outliers.
  - use variable distributions and handle high correlations
- 5. For Binary variables, cast them from boolean to integer data types - if needed-
- 6. For ordinal variables, encode them using StringIndexers
- 7. For nominal variables, encode them using One-Hot Encoding
- 8. Assemble your input features as into single vector
- 9. Scale your feature vector

Reading

- Types of Missing Values: https://stefvanbuuren.name/fimd/sec-MCAR.html
- Read more about One-Hot Encoding and String Indexing from this article: https://medium.com/@nutanbhogendrasharma/role-of-onehotencoder-and-pipelines-in-pyspark-ml-feature-part-2-3275767e74f0