Project Report
Analysis of Program (1 and 2)

DT8034: Big Data Parallel Programming
2022

Vincenzo Buono

Date

June 20, 2022

Contents

1 Introduction

The following project aims to solve a *binary classification* problem through the use of multiple Tree-based and Linear Regression *Machine Learning algorithms* on top of the **Apache Spark**[1] framework and *MLlib*[2].

1.1 Description

The classifier's objective is to establish the mushroom's edibility given a set of physical categorical properties (moreover in Section 2.1). Provided that there is no simple rule or property that directly identifies the mushroom's class (*definitely edible, definitely poisonous, unknown edibility*) [3], this problem fits to be solved through the use of *ML* means.

2 Data Preprocessing

The dataset has been identified and selected among the provided reference links, and is the *Mushroom Data Set* from the *UCI Machine Learning Repository*[4]. It contains the descriptions of the hypothetical samples of 23 species of gilled mushrooms in the *Agaricus* and *Lepiota* family mushroom [5]. It contains *8124* instances each characterized by *22* attributes defining a physical characteristic that span from the *cap-color* to the *spore-print-color* as shown by the *Pyspark schema* in *Table 3*.

2.1 Exploratory Data Analysis

By a first inspection, the dataset does not seem to contain any *NaN* or *NULL* values with all the feature being, as previously discussed, *categorical*. An extract of the *.describe()* output is depicted in *Table 1*.

Table 1: Pyspark dataframe describe (extract)

summary	class	cap-shape	cap-surface	cap-color	...	population	habitat
count	8124	8124	8124	8124	...	8124	8124
mean	None	None	None	None	...	None	None
stddev	None	None	None	None	...	None	None
min	e	b	f	b	...	a	d
max	p	x	y	y	...	y	w

2.1.1 Missing Values

To properly further prepare and process the data is necessary to correctly identify missing or invalid entries. Moreover, with all attributes being *categorical* is possible to have entries that do not contain *NaN* or *NULL* values and therefore not automatically counted, but do contain invalid or not allowed values. For this reason, is necessary to check if all the values are within the allowed entries.

```
# feature info encoded e.g. cap-shape: bell=b,conical=c,convex=x,
    flat=f, knobbed=k,sunken=s

feature_ = {
    'class': ['p', 'e'],
    'cap-shape': ['b', 'c', 'x', 'f', 'k', 's'],
    'cap-surface': ['f', 'g', 'y', 's'],
    'cap-color': ['n', 'b', 'c', 'g', 'r', 'p', 'u', 'e', 'w', 'y
        '],
    'bruises': ['t', 'f'],
    'odor': ['a', 'l', 'c', 'y', 'f', 'm', 'n', 'p', 's'],
    'gill-attachment': ['a', 'd', 'f', 'n'],
    'gill-spacing': ['c', 'w', 'd'],
    'gill-size': ['b', 'n'],
    'gill-color': ['k', 'n', 'b', 'h', 'g', 'r', 'o', 'p', 'u', '
        e', 'w', 'y'],
    'stalk-shape': ['e', 't'],
    'stalk-root': ['b', 'c', 'u', 'e', 'z', 'r'],
    'stalk-surface-above-ring': ['f', 'y', 'k', 's'],
    'stalk-surface-below-ring': ['f', 'y', 'k', 's'],
    'stalk-color-above-ring': ['n', 'b', 'c', 'g', 'o', 'p', 'e',
        'w', 'y'],
    'stalk-color-below-ring': ['n', 'b', 'c', 'g', 'o', 'p', 'e',
        'w', 'y'],
    'veil-type': ['p', 'u'],
    'veil-color': ['n', 'o', 'w', 'y'],
    'ring-number': ['n', 'o', 't'],
    'ring-type': ['c', 'e', 'f', 'l', 'n', 'p', 's', 'z'],
    'spore-print-color': ['k', 'n', 'b', 'h', 'r', 'o', 'u', 'w',
        'y'],
    'population': ['a', 'c', 'n', 's', 'v', 'y'],
    'habitat': ['g', 'l', 'm', 'p', 'u', 'w', 'd']
}
```

Listing 1: Python Dictionary containing allowed categorical values per feature

To achieve this every feature values is compared against a *Python dictionary* containing all the allowed categorical values, with each *attribute as a key* (*Listing 1*).

Inspecting the dataset not just for *NaN* or *NULL* values, but also for values that are not within the *expected categorical range*, by running the script shown in *listing*

2, is possible to observe that the attribute *stalk-root* contains **2480** values that are invalid or unexpected (*Table 2*) that were not detected before.

```
## Missing/Incorrect Values
##############################
# check feature's values
# against allowed
##############################

# look for any null or invalid/unexpected values
dataset.df.select([count( when(  sql_col(x).isNull() | ~sql_col(x
    ).isin(feature_[x]),  x)).alias(x) for x in dataset.df.columns
    ])
```

Listing 2: Check for entries with values that are not in the allowed range

Table 2: Features with invalid values (extract)

class	cap-shape	cap-surface	cap-color	stalk-root	...	population	habitat
0	0	0	0	2480	...	0	0

For convenience each and every *invalid value* has been converted to *None* by running the script in *Listing 3*.

```
## Transform Incorrect values
##############################
# Transform incorrect values
# in NaN/NULL for easier
# imputation during
# preprocessing
##############################

for c in dataset.df.columns:
    #print(c)
    dataset.df = dataset.df.withColumn(c,when(~sql_col(c).isin(
        feature_[c]), None).otherwise(sql_col(c)))

print(f'Displaying only null values (make sure value same as
    before):\n')
dataset.df.select([count( when(  sql_col(x).isNull(),  x)).alias(
    x) for x in dataset.df.columns])
```

Listing 3: Change invalid features' values to None

2.1.2 Data Distribution

The dataset is *balanced* with a *positive-to-negative class ratio*(poisonous/edible) of *0.482* and *0.518* respectively as depicted in *Figure 1*. *Figure 2.1.2* shows *feature distribution* grouped by *class* and plot as a *Pie chart*.

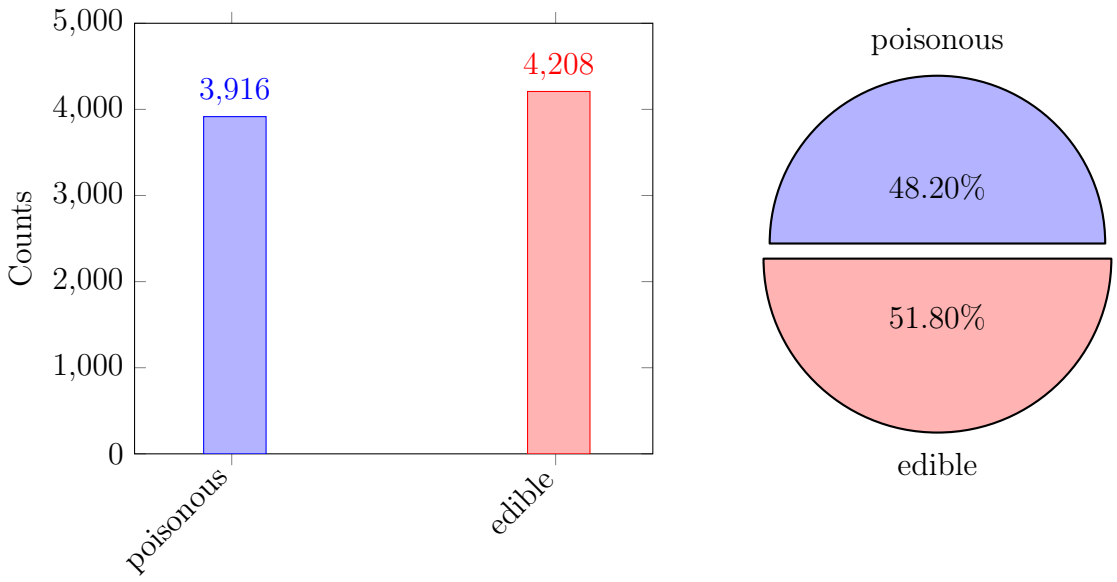

Figure 1: Count plot and Pie chart side-by-side of class distribution group-by *Class*

By inspecting *Figure 2*, is possible to observe that most of the entries (mushroom) have as *habitat'* value *'d'(woods)* with no *noticeable unrepresented class*, while *'w' (waste)* is the habits with the least entries and having the *edible class unrepresented*.

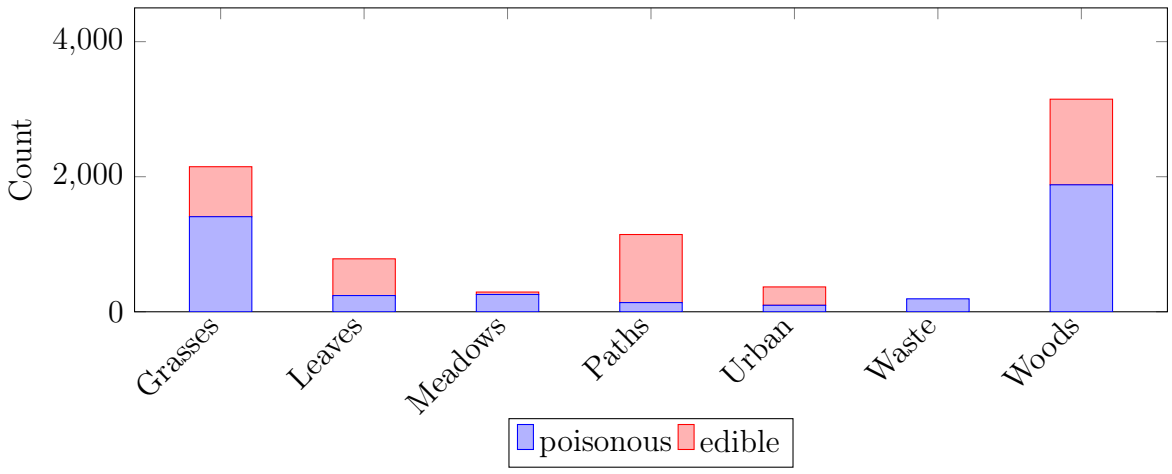

Figure 2: Stacked Bar chart each *class* per *habitat*

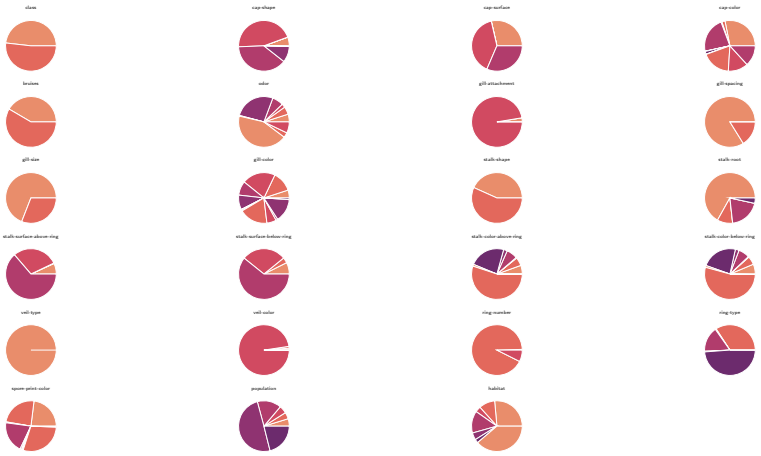

Figure 3: Pie chart of feature distribution group-by *Class*

2.1.3 Data Sparsity Analysis

To easily diagnose the *sparsity* within the analyzed dataset, the *Missingno library*[6] has been used as a *data visualiation suite*[7]. From the *sparsity matrix*, as shown in *Figure 2.1.3*, is possible to observe that the date is *lightly sparse* with only exception (More in *Section 2.1.1*) for the feature *stalk-shape* that seems to contain multiple invalid values. Moreover, confirmation can be seen in *Figure 2.1.3*.

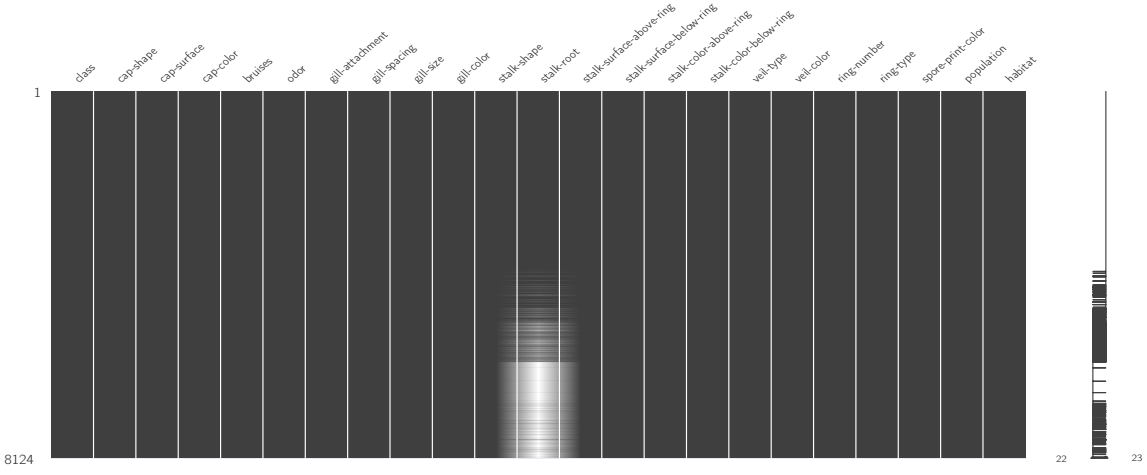

Figure 4: *Missigno Matrix* Data Sparsity

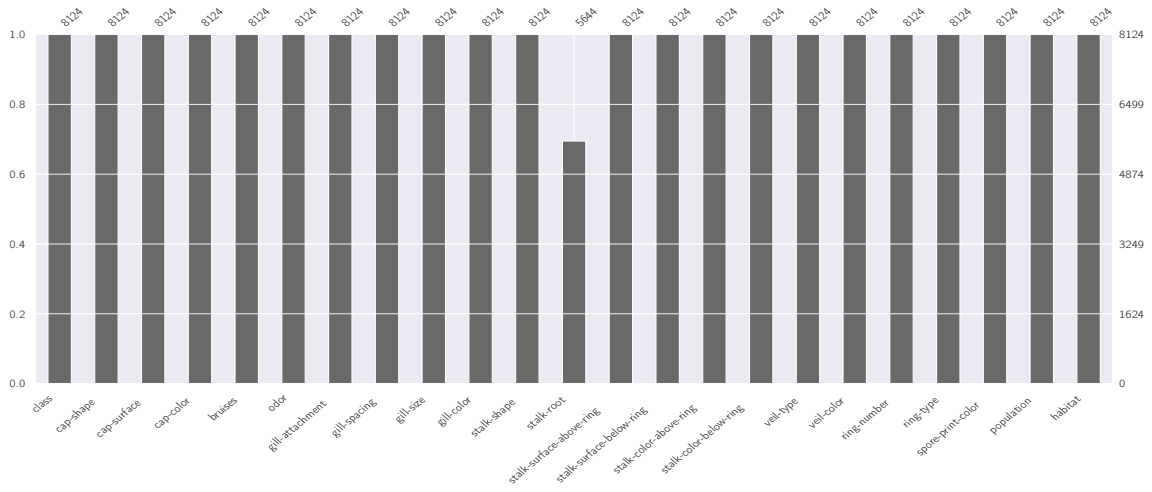

Figure 5: *Missigno Bar* Data Sparsity

2.1.4 Feature Correlation

By inspecting the *Pearson Correlation matrix* depicted in *Figure 2.1.4* is possible to detect that are not present *highly correlated features* (> 0.90). The *Correlation Matrix* has been generated using the *Listing 4*, and plotted using the *Snippet 5*.

```
1  # assembler
2  assembler = VectorAssembler(inputCols=dataset.encoded.columns,
       outputCol='enc_features')
3  dataset_e_attr = assembler.transform(dataset.encoded)
4
5  # print features
6  dataset_e_attr.select('enc_features').show(5)
7
8  dataset.X = dataset_e_attr.select('enc_features')
9
10
11 # Pearson corr-matrix
12 corr_matrix = Correlation.corr(dataset_e_attr, 'enc_features', '
       pearson').collect()[0][0]
13
14 print(f'Correlation Matrix: \n')
15 spark.createDataFrame(corr_matrix.toArray().tolist(), dataset.
       encoded.columns).toPandas()
```

Listing 4: Pearson Correlation Matrix

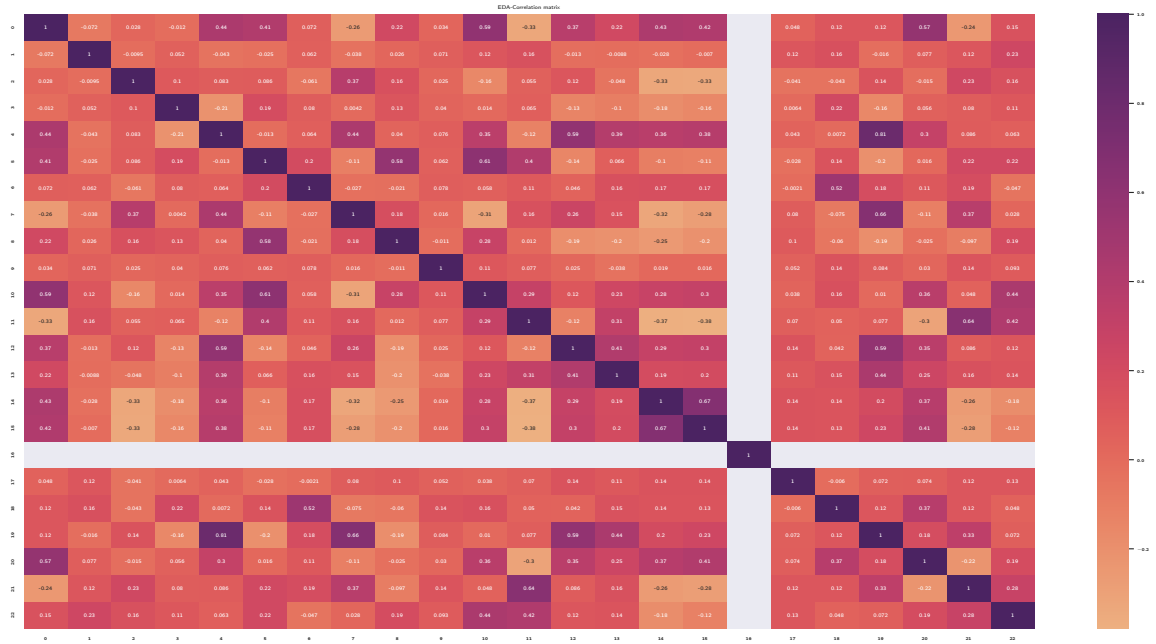

Figure 6: *Pearson* Correlation Matrix

```
1  # assembler
2  assembler = VectorAssembler(inputCols=dataset.encoded.columns,
       outputCol='enc_features')
3  dataset_e_attr = assembler.transform(dataset.encoded)
4
5  # print features
6  dataset_e_attr.select('enc_features').show(5)
7
8  dataset.X = dataset_e_attr.select('enc_features')
9
10
11 # Pearson corr-matrix
12 corr_matrix = Correlation.corr(dataset_e_attr, 'enc_features', '
       pearson').collect()[0][0]
13
14 print(f'Correlation Matrix: \n')
15 spark.createDataFrame(corr_matrix.toArray().tolist(), dataset.
       encoded.columns).toPandas()
```

Listing 5: Plot the *Correlation Matrix*

2.2 Custom Transformers

In order to later streamline the process through the use of standard *Pipelines*, two *custom transformers* have been implemented to first *clean* and then *prepare* the data to *train the machine learning models*. *Listing 6*, as discussed in *Section 2.1.1*, performs initial clean-up by transforming invalid values to *None* and subsequently dropping rows with *null values*. *Listing 7* allows to prepare the *dataframe* by processing columns' naming to fit the further *machine learning model training*.

```
1  #########
2  # Clean #
3  ###########################################################################
4  class CleanTransformer(Transformer, HasInputCol, HasOutputCol):
5      def __init__(self, inputCols, allowedValues):
6          self.inputCols = inputCols
7          self.allowedValues = allowedValues
8      # override _transform
9      def _transform(self, dataset):
10         for c in dataset.columns:
11             dataset = dataset.withColumn(c,when(~sql_col(c).isin(
                   self.allowedValues[c]), None).otherwise(sql_col(c)
                   ))
12         return dataset.na.drop()
```

Listing 6: Custom Transformer to perform initial data clean up from invalid values

```
1  ############
2  # Prepare #
3  ###########################################################################
4  class PrepareTransformer(Transformer, HasInputCol, HasOutputCol):
5      def __init__(self, extractCols, newNames):
6          self.extractCols = extractCols
7          self.newNames = newNames
8      # override _transform
9      def _transform(self, dataset):
10         data = dataset.select(self.extractCols)
11         for key in self.newNames:
12             data = data.withColumnRenamed(key, self.newNames[key])
13         return data
```

Listing 7: Custom Transformer to perform post processing and column management

2.3 Feature Encoding

Subsequently to the fact that, as discussed in *Section 2.1*, all the *23 features are categorical* included the *label feature (class)*, a *label encoder(StringIndexer)* has been used to encode *string column of labels* into *a column of label indices* to better fit the *machine learning algorithms*. For this reason a *StringIndexer* has been adopted along with a *Pipeline* as shown in *Listing 8*.

```
1  ##########################
2  # Reused in job-script #
3  ###############################################################################
4  indexers = [StringIndexer(inputCol=col, outputCol=col+'_idx') for
       col in dataset.df.columns]
5  ###############################################################################
6
7  # Cell-specifics #
8  pip_idx = Pipeline(stages=indexers) # better way is to use a
       pipeline
9  dataset_idx = dfwrapper(pip_idx.fit(dataset.df).transform(dataset
       .df))
10 dataset_idx.df = dataset_idx.df.select([col+'_idx' for col in
       dataset.df.columns])
11
12 dataset_idx.df.printSchema()
13
14 # save to main dfwrapper
15 dataset.encoded = dataset_idx.df
```

Listing 8: StringIndexer

2.4 One-hot Encoding Pre-processing

Before applying the *One-hot Encoding* is necessary to individuate and filter out columns with individual unique values; this has been accomplished by applying the code presented in *Listing 9*.

```
1  print(f'{"-"*50}')
2  for col in dataset.df.columns:
3      if (dataset.df.select(col).distinct().count()==1):
4          print(f'Column [{col}] has {dataset.df.select(col).
               distinct().count()} unique values')
5
6  print(f'{"-"*50}')
7
8  # just for code clarity
```

```
9  for col in dataset.df.columns:
10     print(f'Column [{col}] has {dataset.df.select(col).distinct()
           .count()} unique values')
11     #dataset.df.select(col).distinct().show() # show all distinc
           values per feature
12
13 print(f'{"-"*50}')
14
15 # only column found with single value
16 dataset.df.select('veil-type').distinct().show() # show all
       distinc values per feature
17 dataset.encoded.select('veil-type_idx').distinct().show() # show
       all distinc values per feature
```

Listing 9: One-hot Encoding Pre-processing

2.5 One-hot Encoding

Thereafter the application of the *StringIndexer*, a *one-hot encoding* has been applied as shown in *Listing 10*.

```
1  ############################
2  # reused in job-script #
3  ############################################################
4  # columns to encode w/o single-valued columns #
5  ##############################################################################
6  enc_cols_in = [c for c in dataset.encoded.columns if c not in ['
       veil-type', 'veil-type_idx', 'class', 'class_idx']]
7  enc_cols_out = [c+'_vec' for c in enc_cols_in]
8
9  encoder = OneHotEncoder(inputCols=enc_cols_in, outputCols=
       enc_cols_out)
10 ##############################################################################
11
12 model = encoder.fit(dataset.encoded)
13 oh_enc_dataset = model.transform(dataset.encoded)
```

Listing 10: One-hot Encoding

Table 3: Pyspark dataframe schema

Feature	Type
class	string (nullable = true)
cap-shape	string (nullable = true)
cap-surface	string (nullable = true)
cap-color	string (nullable = true)
bruises	string (nullable = true)
odor	string (nullable = true)
gill-attachment	string (nullable = true)
gill-spacing	string (nullable = true)
gill-size	string (nullable = true)
gill-color	string (nullable = true)
stalk-shape	string (nullable = true)
stalk-root	string (nullable = true)
stalk-surface-above-ring	string (nullable = true)
stalk-surface-below-ring	string (nullable = true)
stalk-color-above-ring	string (nullable = true)
stalk-color-below-ring	string (nullable = true)
veil-type	string (nullable = true)
veil-color	string (nullable = true)
ring-number	string (nullable = true)
ring-type	string (nullable = true)
spore-print-color	string (nullable = true)
population	string (nullable = true)
habitat	string (nullable = true)

3 Spark Implementation

The project has been implemented using the *Apache Spark framework*[1] leveraging *multiple executors* in different configurations (More in *Section 4.1*) to **maximize the process parallelization** and **increase the classifiers' performances**. According to the instructions, two programs have been produced with different *partition schemes* and each has been tested with different *cluster configurations* on the *Google Cloud Platform*[8]; the first one has be designed using the *default partitioning scheme*, while the latter has been implemented with the data partitioned between worker nodes. Ultimately, the performance of each *configuration*, included of every analyzed *machine learning algorithm*, has been evaluated and its performances analyzed. The aim of this project is to study the *performances' improvement*, if present, resulted from *clustering* and *parallelizing* data between multiple workers. Per specification, the following *ML* algorithms, from the *MLlib*[2], has been implemented per-configuration:

- Binomial Logistic Regression[9]
- Random Forest Classifier[10]
- Gradient-boosted Tree Classifier[11]

3.1 Cross Validation

Each *machine learning estimators' hyperparameter* has been tuned using *Cross-Validation*[12] and the specifics have been included in each *ML model* respective section as well as the best individuated parameter's value. For this study a constant value of $folds(k = 3)$ has been chosen and kept consistent across all the analyzed estimators.

3.2 Data Split

The dataset after the transformation has been *randomly split* into a *train* and *test* set with a ration of 80 and 20 percent respectively with the use of the *Pyspark* method *.randomSplit()* as shown in *Listing 11*.

```
1  ## Data Split
2  ######################
3  # 80/20 #
4  ##########
5
6  train, test = dataset.df.randomSplit([0.8, 0.2])
```

Listing 11: Tran/Test Set Split (80/20)

The *class distribution* per each set, as plot in *Figure 7*, is reasonably balanced with delta non-exceeding the 10th percentile.

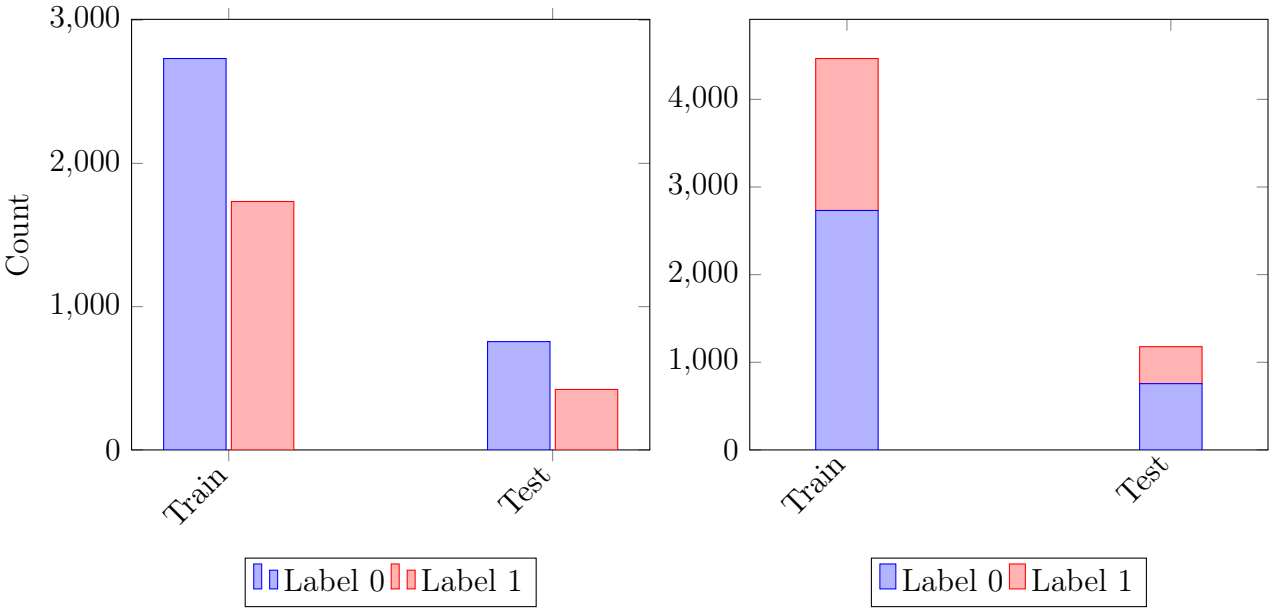

Figure 7: Train/Test Set Class Distribution

3.3 Pipeline

In order to streamline and optimally tune the *ML models*, a *Pipeline* has been adopted, with built-in the custom *clean transformer*(More in *Section 2.2*), *prepare transfomer*(*Listing 13*), feature *VectorAssembler* (*Listing 14*) and the *StringIndexer* and *One-hot Encoder*, discussed in *Section 2.3*.

```
1  ##############################
2  # Clean Custom Transform #
3  ##############################################################################
4  clean_transformer = CleanTransformer(inputCols=dataset.df.columns
       , allowedValues=feature_)
5  ##############################################################################
```

Listing 12: *Pyspark* Pipeline - *CleanTransformer*

```
1  ##############################
2  # Prepare Custom Transform #
3  ##############################################################################
4  prepare_transformer = PrepareTransformer(extractCols=['class_idx'
       , 'features'], newNames={ 'class_idx': 'label'})
5  ##############################################################################
```

Listing 13: *Pyspark* Pipeline - *PrepareTransformer*

```
1  ##############
2  # assembler #
3  ###############################################################################
4  vec_assembler = VectorAssembler(inputCols=[c+'_idx_vec' for c in
       dataset.df.columns if c not in ['veil-type', 'veil-type_idx',
       'class']] ['veil-type_idx', 'class_idx'], outputCol='features'
       )
5  ###############################################################################
```

Listing 14: *Pyspark* Pipeline - *VectorAssembler*

The full *Pipeline* with all the stages is shown in *Listing 15*. After all the *Pipeline's* stages have been applying to the dataset, the resulting dataframe is composed by two column, *'label'*, containing the class, and *'feature'* containing the 23 encoded categorical attributes as shown in *Table 4*.

```
1  ############
2  # Pipeline #
3  ###############################################################################
4  pipeline = Pipeline(stages=[clean_transformer]+indexers+[encoder
       ]+[vec_assembler]+[prepare_transformer]) #indexers-> list of
       indexer
5  dataset.df = pipeline.fit(dataset.df).transform(dataset.df)
6  ###############################################################################
```

Listing 15: *Pyspark* Pipeline - *Stages and Fit()*

Table 4: *Pyspark* Dataset prepared for *MLLib* (Only showing truncated top 5 rows)

Label	Features
1.00	(78,[0,7,9,15,20, ... ,58,62,68,74,77],[1.0,1.0,1.0,1.0,1.0, ... , 1.0,1.0,1.0,1.0,1.0])
0.00	(78,[0,7,10,15,18, ... ,56,58,61,70,72],[1.0,1.0,1.0,1.0,1.0, ... , 1.0,1.0,1.0,1.0,1.0])
0.00	(78,[2,7,11,15,19, ... ,56,58,61,70,75],[1.0,1.0,1.0,1.0,1.0, ... , 1.0,1.0,1.0,1.0,1.0])
1.00	(78,[0,5,11,15,20, ... ,58,62,68,74,77],[1.0,1.0,1.0,1.0,1.0, ... , 1.0,1.0,1.0,1.0,1.0])
0.00	(78,[0,7,8,16,22, ... ,56,60,61,69,72],[1.0,1.0,1.0,1.0,1.0, ... , 1.0,1.0,1.0,1.0,1.0])

3.4 Binomial Logistic Regression

To predict the *binary label* has been selected the *LogisticRegression* estimator with the *hyperparameter* expressed in *Table 5*. After *Cross-validation*, the best parameter have been individuated to be 0.1 for the *regularization parameter (regParam)* and 0 for the *elasticNet* parameter.

In *Apache Spark MLlib* implement linear Logistic regression methods for linear and non-linear features with L_1 or L_2 *regularization*[13] as shown in *Equation 1.* Therefore, when the *elasticNetParam* (α) is set to 1 we're implementing a *Lasso model* and when set to 0 a *Ridge Regression model* respectively. On the other hand, the *regParam* acts as λ in *Equation 1.*

$$\alpha\left(\lambda\|\mathbf{w}\|_1\right) + (1-\alpha)\left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right), \alpha \in [0,1], \lambda \geq 0 \tag{1}$$

3.4.1 Hyperparameter optimization

The parameters tuned using *Cross-validation* with 3 *k-folds*, are expressed in *Table 5* amongside the optimal values. The optimal *elasticNetParam* has been found to be 0, and, as discussed in *Section 3.4*, is equivalent to a L_2 regularization (*Ridge Regression*).

Table 5: *Logistic Regression Hyperparameter Optimization-ParamGridBuilder*

Parameter	Values							Best
regParam (C)	0.001	0.01	0.1	1	5	10	100	**0.1**
elasticNetParam	0.0	1.0	-	-	-	-	-	**0.0**

3.5 Random Forest

Random Forest are algorithms for learning ensembles of *Decision Trees*[10]. The *RandomForestClassifier* allows to train multiple trees in parallel whereas *GBTs*(More in *Section 3.6*) can only train a tree at the time, taking longer as shown in *Section 4.*

3.5.1 Hyperparameter optimization

The *hyperparameters* optimized are tabulated in *Table 6* alongside with their optimal values. It is important to note that *Random Forests*, compared to *GBTs*, the increase of number of trees reduces reduces the *variance*[11], in contrary with *GBTs* that reduce *bias.* The *Hyperparameter* is often easier as the performance improve *monotonically* with the increase in *dt*(decision tree(s)) count [11].

Table 6: *Random Forest Hyperparameter Optimization-ParamGridBuilder*

Parameter	Values											Best
numTrees	100	200	300	-	-	-	-	-	-	-	-	**100**
maxDepth	5	6	7	8	9	10	11	12	13	14	15	**8**

3.6 Gradient-boosted Tree Classifier

Gradient-Boosted Trees (GBTs), like *Random Forests*, are represented by an ensemble of *Decision Tree* that iteratively train *dt* with the end goal of **minimizing a loss function**[9]. The *GBTClassifier* within the *Pyspark MLlib* uses the *logistic loss function* or *twice binomial log* as shown in *Equation 2*[9].

$$2\sum_{i=1}^{N}\log\left(1+\exp\left(-2y_iF\left(x_i\right)\right)\right) \tag{2}$$

3.6.1 Hyperparameter optimization

The *hyperparameters* evaluated are shown in *Table 7* as well as their optimal values.

Table 7: *GBT Hyperparameter Optimization- ParamGridBuilder* params

Parameter	Values										Best
maxDepth	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	**5.0**

4 Results

4.1 Default Partition (Part1)

For the first part of the comparative analysis, as discussed in *Section 3*, the *Pyspark MLlib machine learning classifiers* have been deployed on the *Google Cloud* with a *standard configuration* comprising **1 master and 2 worker nodes** each with 4 *vCPU* and 15GB of dedicated memory. A detailed list of the *GCP cluster's configuration* is shown in *Table 8*.

Table 8: Google Cloud CE Cluster Configuration

Cluster Type	N-Workers	Series	Machine Type	vCPU	Memory	SSD Interface
Starndard	2	N1	n1-standard-4	4	15GB	SCSI

For the *Pyspark/RDD configuration*, has been adopted, for this configuration, a *default partition count* with *cache* disabled and default memory constraints (particularly important during cross-validation where high heap usage is necessary) as shown in *Table 9*.

Table 9: RDD Partition Scheme/Pyspark Configuration

P-Count	Cache	Driver Mem.	Executor Mem.	Max Result Size	Mem. Fraction
Default	No	1GB	1GB	1GB	0.6

4.1.1 Benchmarks

The execution of *time benchmark*, upon 10 iterations, revealed a mean *data loading time* of 5.25s when performed on *GCP Cluster SCSI SSD* with a *sample standard deviation* of 27.38. The *StringIndexer* completed its execution on average in 8.38s given a *standard deviation* of 22.31. The entire *Pipeline*(More in *Section 3.3*) only took a mean *execution time* of 14.81s with a *high-dimensionality* dataset comprising 24 categorical features.

Table 10: Time Benchmarks - Preprocessing

	Operations			
	Data Load	StringIndexer	One-hot Encoding	Pipeline
Mean ($\overline{x}$)	5.25s	8.38s	0.23s	14.81s
Std (s)	27.38	22.31	53.9	47.9

The three analyzed *ML classifiers* showed high *accuracy and precision* regardless of the high dimensionality of the dataset, with $f1$ scores approaching 1. The *RandomForestClassifier* exhibited the highest performance metrics alongside with the *GBT* with the least training and prediction time. The *GBT* showed the highest training time, but preserved the same high accuracy showed by the *Random Forest*, this is possible a direct cause, as explained in *Section 3.5*, of the inability of training multiple trees in parallel, bottle necking the gain obtained from increasing the operation parallelization. The *Logistic Regression* displayed the fastest training time but presented the worst performance metrics across the board. The performance metrics have been obtained using built-in *Pyspark BinaryClassificationEvaluator()* and *MulticlassClassificationEvaluator*. Other metrics including F_1 score, *Accuracy (Equation 4)*, *Sensitivity (Equation 5)*, *Precision (Equation 6)*, *Negative Predictive Value (Equation 7)*, *False Positive Rate (Equation 8)*, *False Negative Rate (Equation 9)* and *Matthews Correlation Coefficient (Equation 10)* have been obtained through the *Confusion Matrix* and are shown in *Table 11*.

$$F_1 = \frac{tp}{tp + \frac{1}{2}(fp + fn)} \tag{3}$$

$$\text{Acc} = \frac{\text{tp} + \text{tn}}{\text{p} + \text{n}} = \frac{\text{tp} + \text{tn}}{\text{tp} + \text{tn} + \text{fp} + \text{fn}} \tag{4}$$

$$\text{tpr} = \frac{\text{tp}}{\text{p}} = \frac{\text{tp}}{\text{tp} + \text{fn}} = 1 - \text{fnr} \tag{5}$$

$$\text{ppv} = \frac{\text{tp}}{\text{tp} + \text{fp}} = 1 - \text{fdr} \tag{6}$$

$$\text{npv} = \frac{\text{tn}}{\text{tn} + \text{fn}} = 1 - \text{for} \tag{7}$$

$$\text{fpr} = \frac{\text{fp}}{\text{n}} = \frac{\text{fp}}{\text{fp} + \text{tn}} = 1 - \text{tnr} \tag{8}$$

$$\text{fnr} = \frac{\text{fn}}{\text{p}} = \frac{\text{fn}}{\text{fn} + \text{tp}} = 1 - \text{tpr} \tag{9}$$

$$\text{MCC} = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}} \tag{10}$$

Table 11: Time Benchmarks - ML Models

	Classifier		
	Logistic Regression	Random Forest	GBTree
Mean RT ($\overline{x}$)	28.03s	34.80s	37.11s
AreaUnderROC	1.0000	-	
F_1 Score	0.9986	1.0000	1.0000
Accuracy	0.9983	1.0000	1.0000
Sensitivity (tpr)	0.9972	1.0000	1.0000
Precision (ppv)	1.0000	1.0000	1.0000
Negative Predictive Value (npv)	0.9955	1.0000	1.0000
False Positive Rate (fpr)	0.0000	0.0000	0.0000
False Negative Rate (fnr)	0.0028	0.0000	0.0000
Matthews Correlation Coefficient	0.9963	1.0000	1.0000

4.2 Optimizations (Part 2)

4.2.1 RDD Partitioning

The following section provides a comprehensive analysis on the performances improvements and caveats provided by the implementation and optimization of *data repartitioning*. The empirical testing has been conducted on the *GCP Cluster* composed of *1 master and 2 N-Workers* equipped each with *8 vCPU* each, and their specifications have been outlined in *Table 12*.

Table 12: Google Cloud CE Cluster Configuration - Data Partitioning Testing Setup

Cluster Type	N-Workers	Series	Machine Type	vCPU	Memory	SSD Interface
Standard	2	N1	Custom	8	15GB	NVMe

The testing has been conducted by iteratively measuring the *execution times* of each *Machine Learning algorithm* when provided with a different *data partition counts*. The data has been *repartitioned* with the *PySpark built-in* method (*Listing 16*). The *partition counts* under considerations are in the range of 2^n, *where* $0 \leq n \leq 8$ (*algorithm 1*), for evenly distributed data splits, while $9, 15, 31$ have been selected as a counterargument for improper partitioning between *worker nodes* and *vCPUs*. Each test has been performed 3 times, and the *mean execution times* have been proposed in *Figure 8*. The *execution time* has been observed to be at its *minimum* with 16 *data partition counts* and *maximal* at the value of 256, when considering the entirety of the *pipeline* (i.e. *fit() and predict()* combined). Across the full range of *partition counts*, the *Random Forest classifier* achieved **the lowest overall execution times** of $3.73s$ at $PC_{rf} = 16$, followed by *Logistic Regression* ($4.09s$, at $PC_{lr} = 16$) and, lastly, *Gradient-boosting Tree classifier* ($4.51s$, at $PC_{gbt} = 16$) as depicted in *Figure 8*. Moreover, all the *ML algorithms* under analysis showed a similar behavior, providing lowest execution times with *data partitions* count of 16.

Algorithm 1 RDD Data Repartition

Ensure: $0 \leq N \leq 8$

1: **for** $i = 0$ **to** N **do**
2: $\quad n \leftarrow 2^i$
3: $\quad dataset.df \leftarrow dataset.df.repartition(n)$
4: **end for**

```
1  dataset._df = dataset._df.repartition(256)
2  print('partition count:', dataset._df.rdd.getNumPartitions())
```

Listing 16: DataFrame Repartitioning

Analyzing the *execution time* of the *fit-phase*, the *Random Forest* estimator has been individuated to possess the **fastest** *running time* (1.65s, at $PC_{rf} = 16$), with the *Logistic Regression* model completing the *fit process* in 1.72s, at $RDD_{lr} = 16$ and the *GBT classifier*, showing the slowest with an *execution time* of 2.86s, at $RDD_{lr} = 16$. By inspecting *Figure 8*, and subsequently *Figure 9*, is possible to observe that within the *execution time* of the *Grandient-boosted tree* classifier, on average, 62.70% of the whole duration has been occupied by *fitting the model* and only 37.3% of the interval dedicated to compute the *predictions*; computing the *mean ratio in percentile* (*Table 13*) revealed the *Gradient-boosted Tree, Random Forest, Logistic Regression*, dedicate, in *descending order*, *highest to lowest* amount of time for the *fit-phase* respectively. Therefore, depending on the problem statement and prediction cost, *Gradient-boosted Tree* provides **faster predictions**, given that the *model fitting* is executed ahead of time and its costs negligible, while *Logistic Regression* exhibited the **highest prediction times** but provided the **lowest** *model-fit* duration, making it preferable whenever frequent *re-fitting* are necessary. In addition, is necessary to highlight that *Logistic Regression* showed on average the **lowest overall execution time**, but when *the data has been partitioned correctly* (More in *Section 4.2.1.1*), while the *Random Forest* classifier showed the **lowest execution times**.

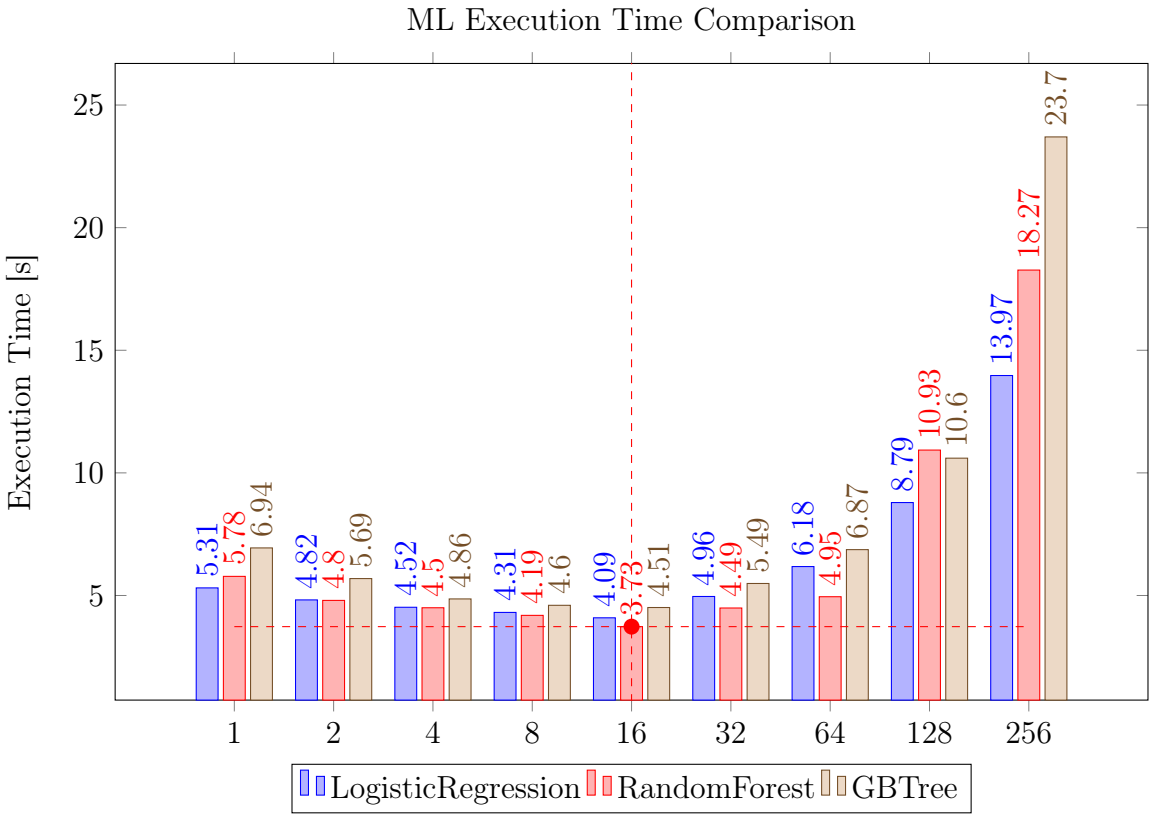

Figure 8: *Logistic Regression, Random Forest, GBTClassifier* Execution Times with Different RDD Partition Counts

Table 13: *Logistic Regression, Random Forest, GBTree Classifier, Fit() and Execution Time(ET)* ratio comparison

	Classifier					
	Logistic Regression		**Random Forest**		**GBTree**	
	Fit	ET	Fit	ET	Fit	ET
Mean ($\overline{x}$) [s]	2.60	6.33	3.64	6.85	5.30	8.14
Min [s]	1.72	4.09	1.65	3.73	2.86	4.51
Max [s]	6.09	8.79	10.79	18.27	17.37	23.7
Mean Ratio %		40.48		50.23		62.70
Min Ratio %		34.65		40.67		58.93
Max Ratio %		43.59		66.67		73.29

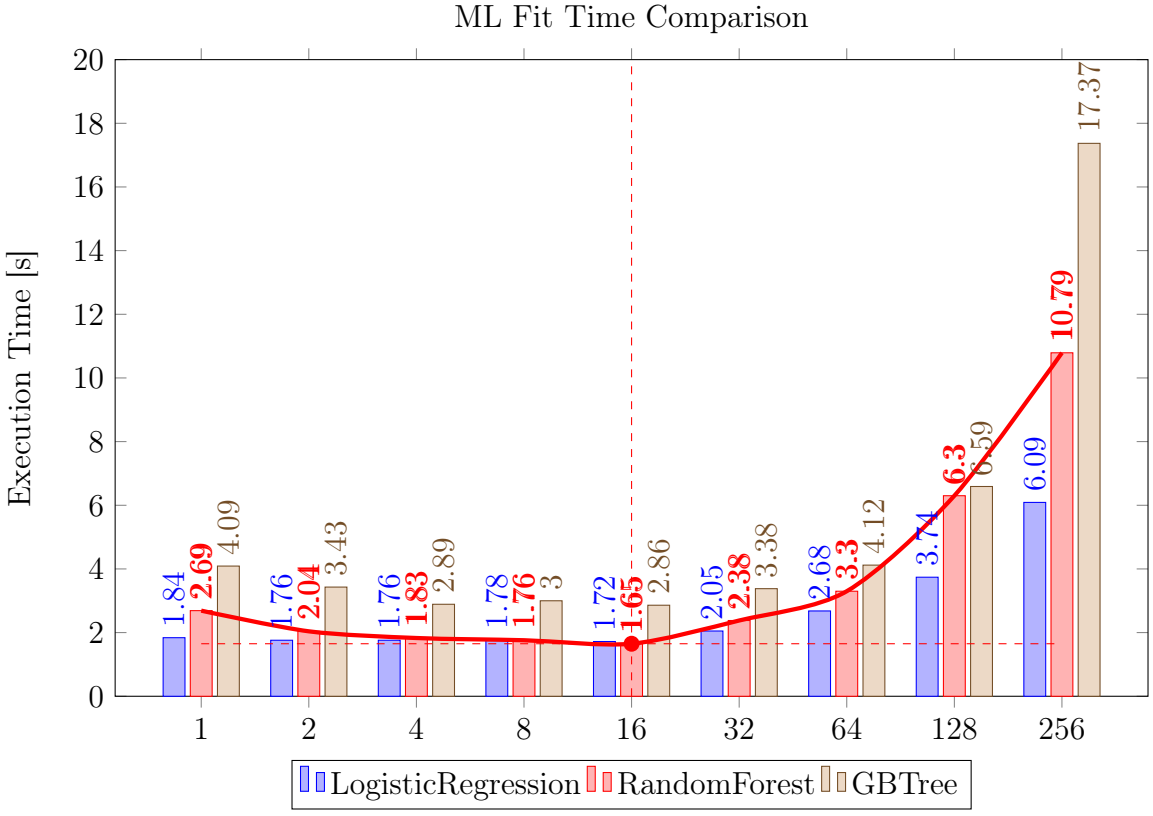

Figure 9: *Logistic Regression, Random Forest, GBTClassifier Fit-Phase* Execution Times with Different RDD Partition Counts

4.2.1.1 Partition Size and *vCPU*

By studying the correlation between *partition size* and *vCPU*, has been observed that **lowest execution times**, regardless of the *ML algorithm* implied, has been obtained when the *partition counts* exactly matched the number of *vCPU per working node*. Moreover, by examining the *trend line* plotted in *Figure 9*, is evident that the *execution times* decrease and find their **minimum** when $PC_a = N_{vCPU}$ and hereafter rapidly increase. Unambiguous representation of this behavior is shown in *Figure 10*. Furthermore, as briefly previously discussed, the *partition size* should be chosen to **maximize parallelization**, which, when the selected **partition count is too small**, it results in *low concurrency*, with *few cores* occupied for *long intervals* while others *idling*, causing **sub-optimal execution times**. On the other hand, when *repartitioning in the data* in too many segments, *high computational overhead* is introduced both during the *execution phase* (i.e. having cores' processing being halted by the *O.S scheduler* and introducing *segment load and unload overhead*, relative to **context-switching**) as well as during the partition *reducing and reassemble*, producing ultimately, **sub-optimal execution times**.

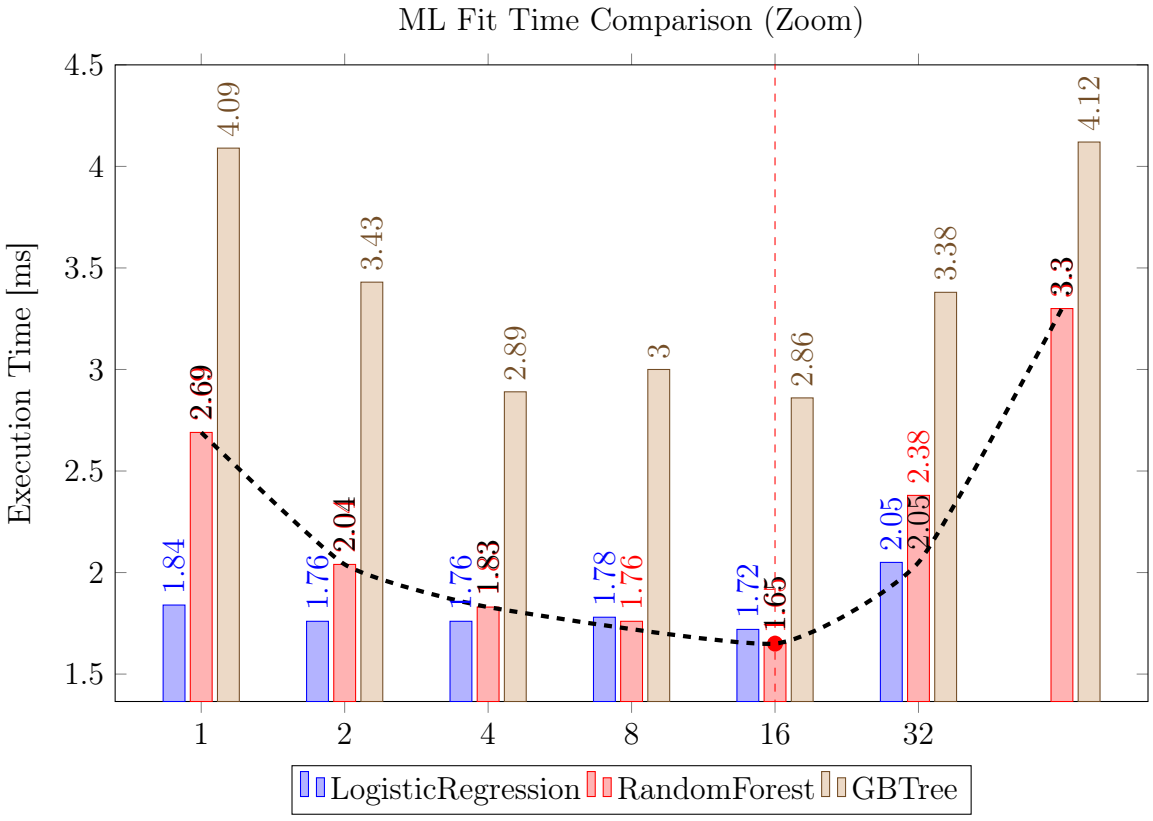

Figure 10: *Logistic Regression, Random Forest, GBTClassifier Fit-Phase* Execution Times with Different RDD Partition Counts (Zoom)

4.2.1.2 Partition Size Distribution

To further characterized the *behavior* of each *ML algorithm* in relation to its *partition size*, and ultimately, measure its performances, *non-multiple* base n (i.e. 2^N+1, where $N \in \{3,4,5\}$) *partition counts* have been selected and the *running times* have been plot in *Figure 11*. When selecting a *static value* from unevenly distributed data splits, as shown in *Figure 11* and subsequently in *Figure 12*, the *execution times* slightly raise on the PC_{nm} points breaking the pattern observed in *Section 4.2.1.1*; from the observed data, in point $PC_{nm} = 9$ the *execution time* is higher than its predecessor and successor while having a higher value than the estimation (i.e. *lower value* than its predecessor) (*Equation 11*). PC_{nm} values selected after the optimal *repartition size* showed a similar behavior. The data extracted from the *fitting-phase*, and plotted in *Figure 12* exhibited a similar behavior.

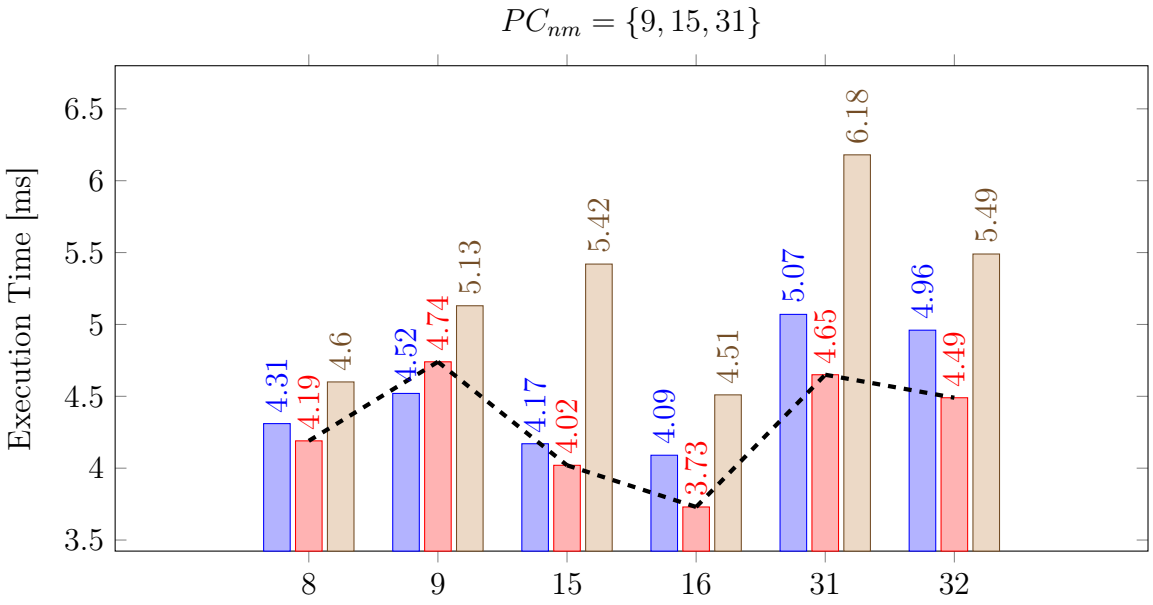

Figure 11: *Logistic Regression, Random Forest, GBTClassifier with non-multiple vCore RDD Partitions (Execution Time)*

$$\hat{E}[PC_{i+1}] = \begin{cases} PC_i \geq PC_{i+1} \geq PC_{i+2} & \text{for } i \leq N_{vCPU} \\ PC_i \leq PC_{i+1} \leq PC_{i+2} & \text{for } i \geq N_{vCPU} \end{cases} \tag{11}$$

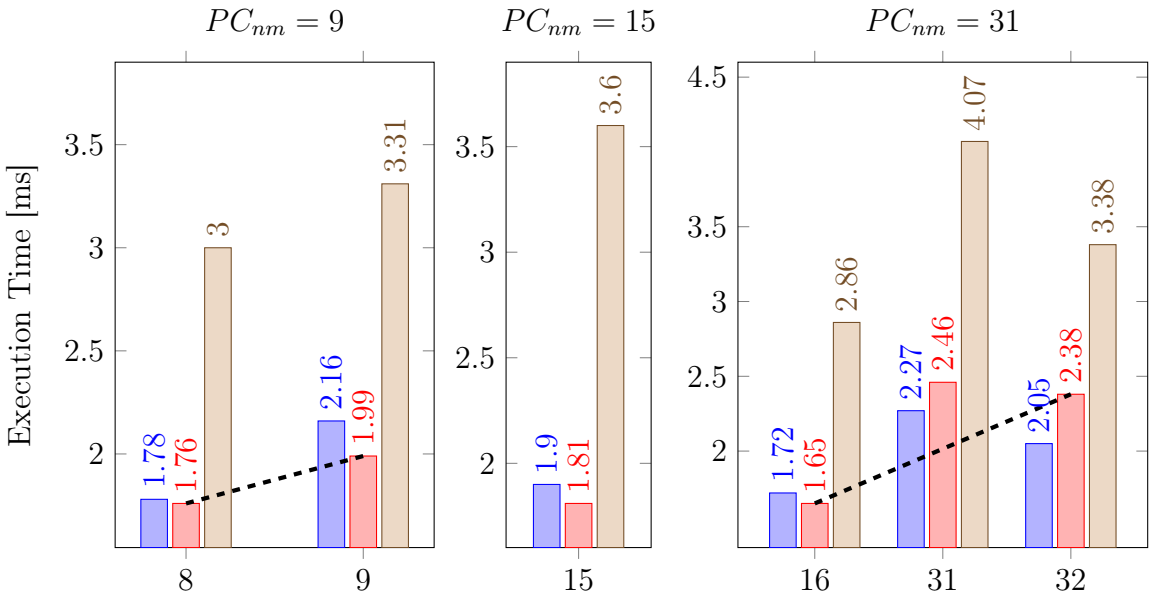

Figure 12: *Logistic Regression, Random Forest, GBTClassifier with non-multiple vCore RDD Partitions (Fit Execution Time)*

4.2.1.3 Adaptive Query Execution and Partition Configurations

Spark provides two main configuration to control the number of partition that the *shuffle operation* creates: *spark.sql.shuffle.partitions, and, spark.default.parallelism*. The *.parallelism* configuration is available under the *RDD* APIs, and by default is set to **the number of all available cores in the cluster**, providing by default, when a *shuffle* is triggered to the same performance discussed in *Section 4.2.1.1* for *optimal number of partitions*. On the contrary, the *.parallelism* configuration is available and used for the newer *DataFrame* API and its default value is set to 200; this allows *Adaptive Query Execution* (available for *Apache Spark* $\geq$ 3.2.0) to iteratively reduce the *partition size* and statistically choose the *most efficient query execution plan at runtime*[14]. Each *reduction* is performed using the built-in *coalesce()* method that minimizes data movement across partitions. Despite this paper does not focus on *data skewness*, is important to emphasize that performances of the *ML algorithms*, but more generally of *distributed predictive systems*, and consequently their models' accuracies, heavily rely on the distribution of the data. To provide an even distribution of the predictive features (*Listing 17*), is possible to utilize, including but not limited to, different *Spark Partitioners* or even implement a *custom partitioning function*.

```
1  dataframe._df = dataframe._df.repartition(20, "class")
```

Listing 17: DataFrame Repartitioning

4.2.1.4 Pipeline

The *execution time* of the implemented pipeline (More in *Section 3.3*) has been studied to identify any recurrent behavior in relation to the *partition counts* and the results have been plot in *Figure 13*. By examining the *trend line*, is possible to note a 12.15% *execution time* increase from 1 till the number of *vCPUs* (N_{vCPU}) and a significant increase henceforth of 74.21%. The PC_{nm}, discussed in *Section 4.2.1.1*, are marked in red and despite showing a similar behavior observed in *Section 4.2.1.2*, within margin of error follow the *trend line*. Therefore, the *pipeline execution time* increase *nonlinearly* with the increase of the *partition size* with a steep rise henceforth the *number of vCPU* available.

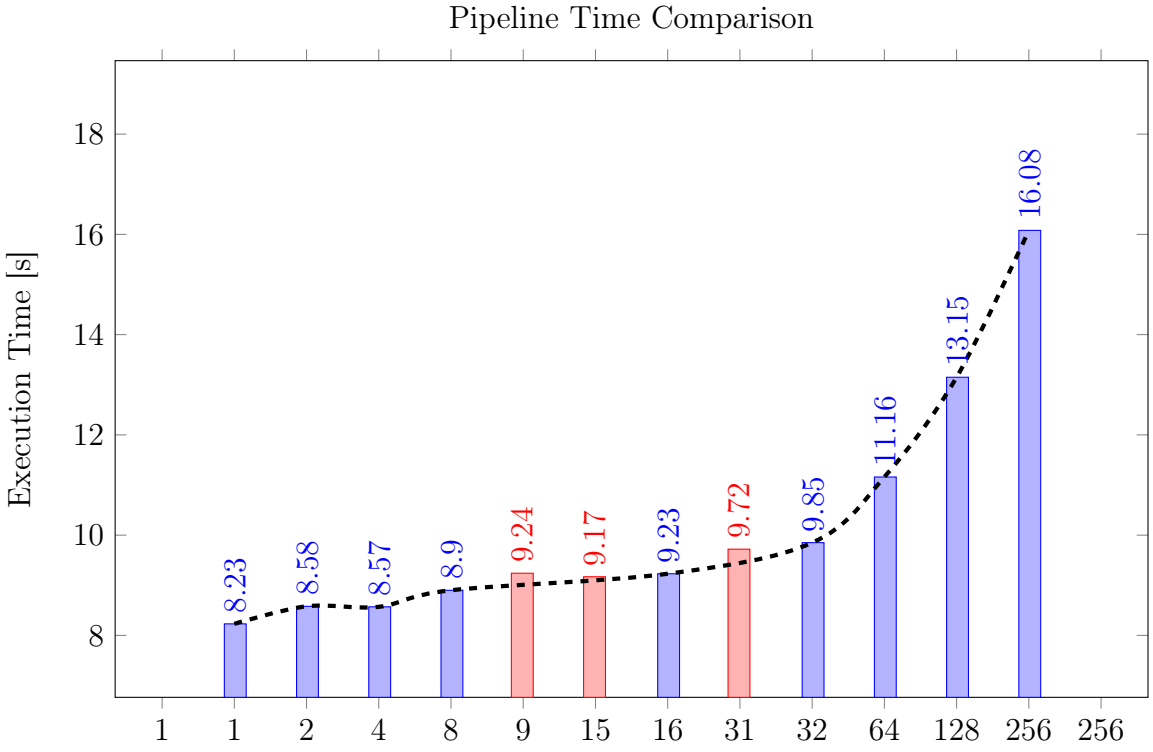

Figure 13: *Pipeline Time Comparison with different RDD Partition Counts*

4.2.1.5 Caching

Apache Spark provides built-in methods to *cache intermediate* computations essential for fast data lookup in *ML algorithms iterative loops* [15]. The *caching policy* evaluated in the following testing is *persist(StorageLevel.MEMORY_ONLY)* (i.e. cache()) allowing the cache to be store directly in *main memory*; other options are also available, including but not limited to, *StorageLevel.MEMORY_AND_DISK_ONLY* for

datasets not fitting entirely in *main memory*. Note that *persist()* only marks the relative DataFrame as *cacheable*, as *caching is a lazy operation*, therefore the cache will not be store until an action is triggered. For the conducted testing, a simple *.count()* is executed to directly materialize the cache. By inspecting *Figure 14* is displayed a similar behavior discussed in *Section 4.2.1.1*, but with the *minimum execution time* pivoting at **half the number of allocable *vCPUs*(N_{vCPU})**. Results are consistent when compared with the measurements obtained from the *Logistic Regression*(*Figure 15*) and *Gradient-boosted Tree*(*Figure 16*) classifiers. Comparing the *mean values* with the *non-cached* execution times (More in *Section 4.2.1*) has been observed a significant 59.39% reduction in the overall times when running the *Logistic Regression* algorithm, a 45.35% improvement on the *Random Forest* classifier and a 32.43% cutback in *overall times* when executing *Gradient-boosted Tree* (*Table 14*). When testing *non-multiple* base *n partition counts*, a similar behavior observed and discussed in *Section 4.2.1.2*, is discovered, with *higher values* in the PC_{nm} points when compared to their *predecessor* (for $i \leq N_{vCPU}/2$), or *lower values* (for $i \geq N_{vCPU}/2$) (*Figure 17*).

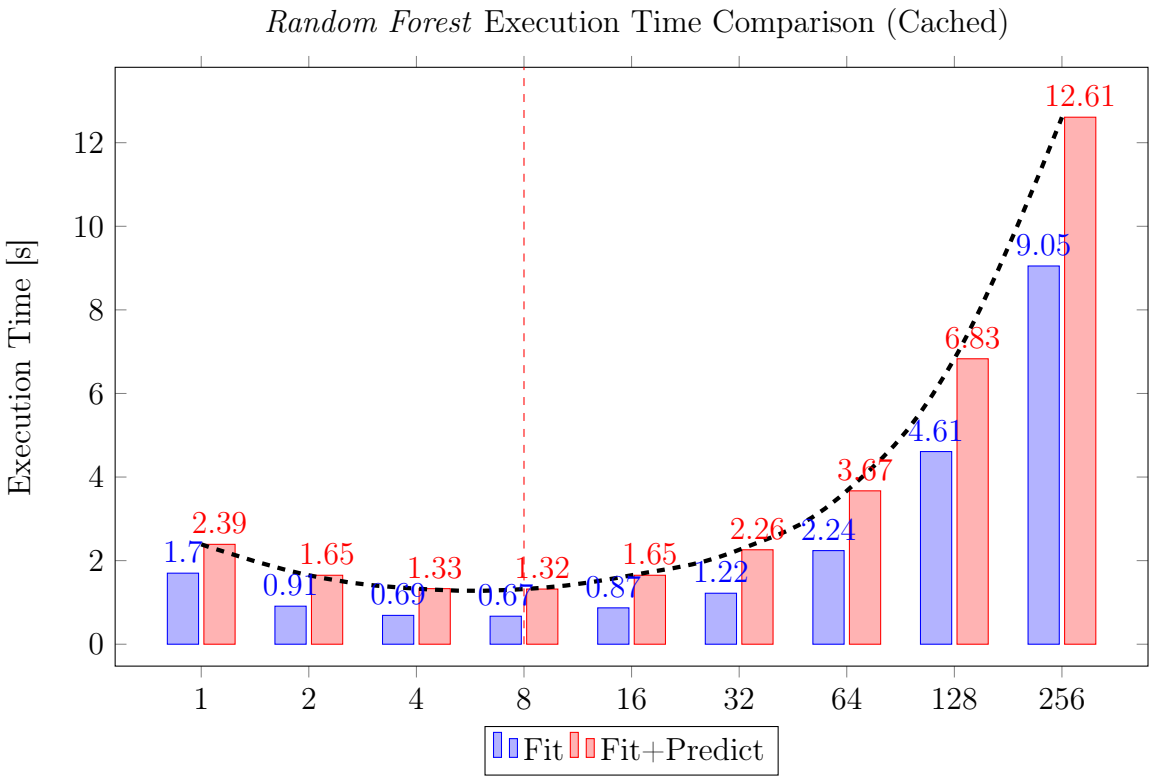

Figure 14: *Random Forest* classifier (Cache) Execution Times with different *partition sizes*

Table 14: *Logistic Regression, Random Forest, GBTree Classifier, Fit() and Execution Time(ET)* Delta changes (cache)

	Classifier					
	Logistic Regression		**Random Forest**		**GBTree**	
	Fit	ET	Fit	ET	Fit	ET
Min $[s]$	0.37	0.93	0.67	1.32	2.28	2.78
Max $[s]$	4.26	7.22	9.05	12.61	15.38	17.83
Mean Delta[1] $(\overline{x})$ %	−47.87	−59.39	−32.93	−45.35	−13.16	−32.43

Note[1]*:* a negative value implies a *reduction* from the original *execution time*, while a positive value implies an *increment*.

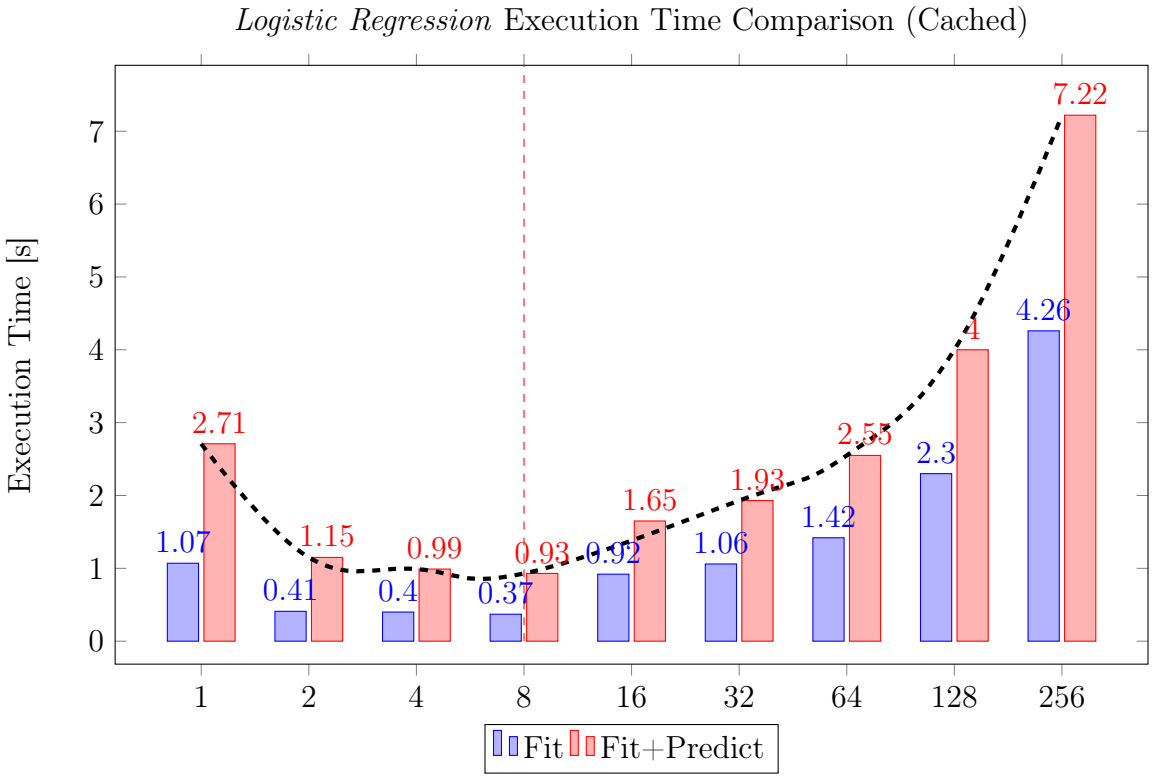

Figure 15: *Logistic Regression* classifier (Cache) Execution Times with different *partition sizes*

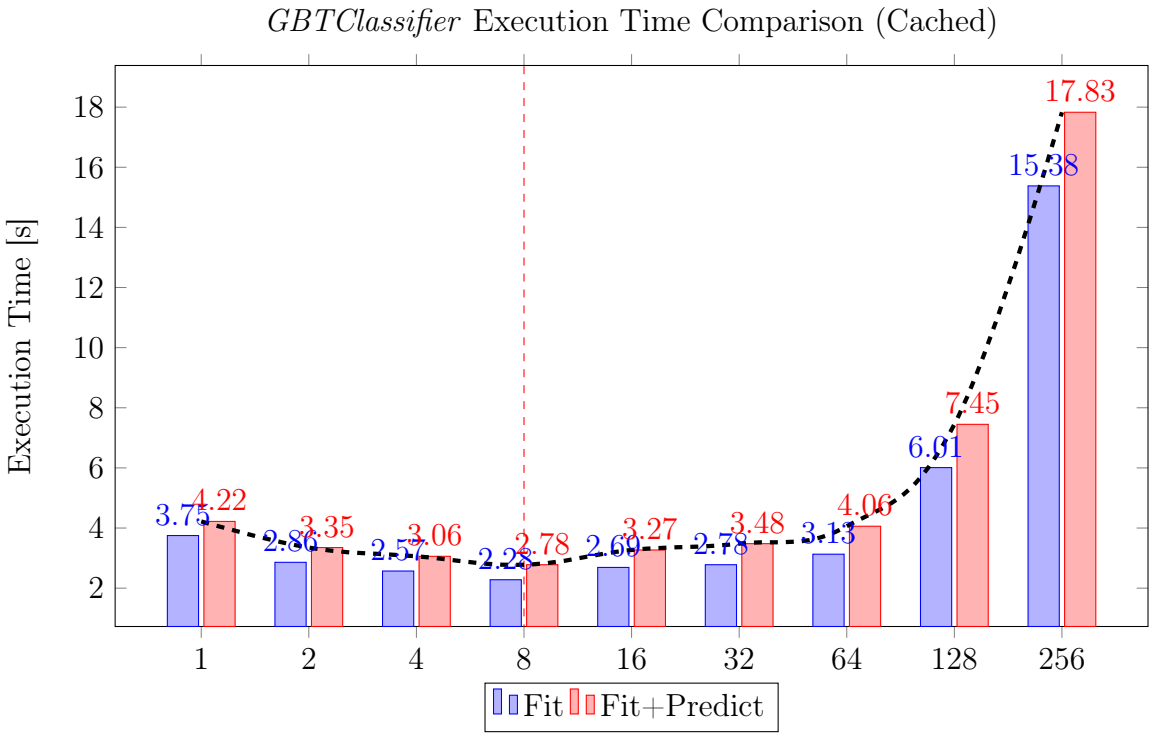

Figure 16: *Grandient-boosted Tree* classifier (Cache) Execution Times with different *partition sizes*

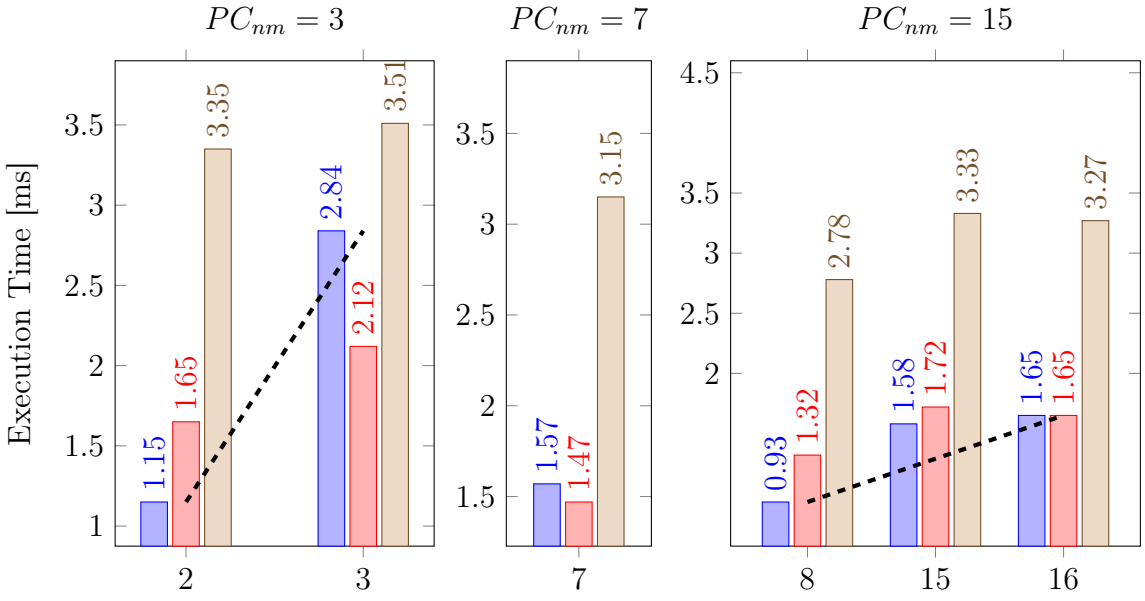

Figure 17: *Logistic Regression, Random Forest, GBTClassifier with non-multiple vCore RDD Partitions (Cache) Execution Times*

4.2.2 Horizontal Scaling: N-Workers

To study and analyze the *parallelism capabilities* of the system, as well as study the behavior and effect that, increasing the *physical system resources* has, on the performance of the *analyzed machine Learning algorithms*, different tests have been conducted with an increasingly higher count of *Worker Nodes*. Due to limitation on the available *Google Cloud Platform cluster configurations*, the number of *working nodes* (1-Master, N-Workers) has been selected in the set of $N_{workers} \in \{2, 4, 6, 7\}$ each with the specification outlined in *Table 15*. By inspecting the *execution times* as the *number of worker nodes increases*, despite the low available testing samples (i.e. $\max_{N_{workers}} = 7$), is possible to detect a **notable decline** in *execution times* as the number of *worker nodes* rises until $N_{workers} = 6$, henceforth the change levels off (*Figure 18*). Furthermore, the *Logistic Regression* exhibits the **highest reduction** in *overall times*, by dropping 40.02%, from 2 to 4 *worker nodes* and, a supplementary 10.55% from 4 to 6 and ultimately stabilizing at 4.37*s* with an observed change of 0.68%. *Random Forest* and *Gradient-boosted Tree* showed a similar behavior.

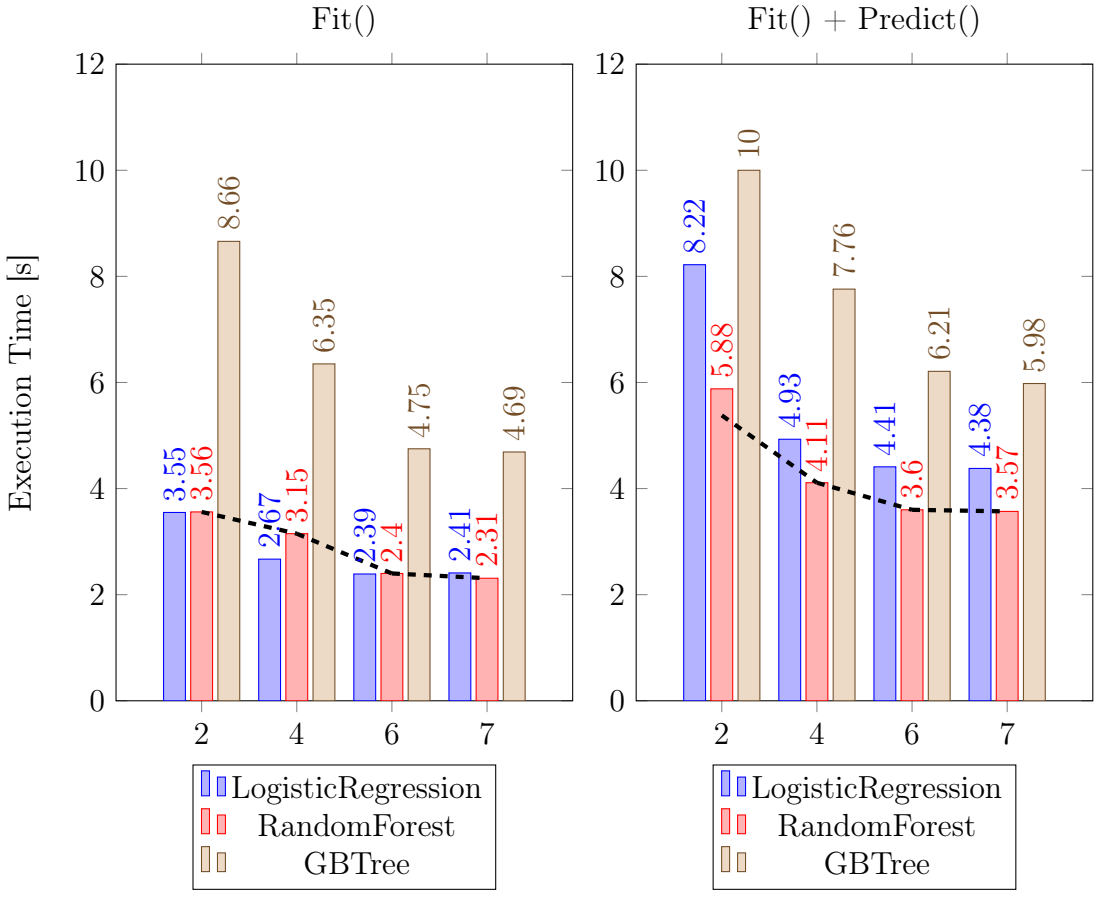

Figure 18: *Logistic Regression, Random Forest, GBTClassifier* Execution Times with Different N-Workers Count

According to the obtained results, is possible to infer that *all the analyzed algorithms* benefit from additional *physical resources*(i.e. worker nodes) that can handle parallel work, however, seems evident that there exist a *diminishing return*, in terms of performances, that is approached as the *number of worker nodes* approaches 7. Therefore, is possible to state that the parallelism of the implemented programs, when defined in function of the number of *logical working units* (worker nodes), is limited, or *bottleneck*, by the **serial portion** of the program. This limit is driven, and found at $N_{workers} = 7$, due to the limited amount of *vCPU* per worker node (due to *GPC limitations*, as previously described). Raising the *physical limitation* of each worker, subsequently increasing the *throughput*, will yield *higher parallelism* by reducing the bottleneck. By measuring *single thread*, *single-core execution time* is possible to estimate the theoretical speed up using *Amdahl's law*[16](*Equation 12*). The *rate of changes* ($\Delta N_{a \to b}$) in percentile for each *machine learning algorithm analyzed* is proposed in *Table 16*.

Table 15: Google Cloud CE Cluster Configuration

Node	Series	Machine Type	vCPU	Memory	SSD Interface
Master	N1	n1-standard-4	4	15GB	SCSI
Worker(s)	N1	n1-standard-2	2	7.5GB	SCSI

$$
\begin{aligned}
S_{\text{latency}}(s) &= \frac{1}{(1-p) + \frac{p}{s}} \\
\text{s.t.} \quad &\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s \to \infty} S_{\text{latency}}(s) = \frac{1}{1-p} \end{cases}
\end{aligned}
\tag{12}
$$

Table 16: *Logistic Regression, Random Forest, GBTree Classifier, Fit() and Execution Time(ET)* delta changes

	Classifier								
	Logistic Regression			**Random Forest**			**GBTree**		
	$N_{2 \to 4}$	$N_{4 \to 6}$	$N_{6 \to 7}$	$N_{2 \to 4}$	$N_{4 \to 6}$	$N_{6 \to 7}$	$N_{2 \to 4}$	$N_{4 \to 6}$	$N_{6 \to 7}$
Δ Fit()[1] [%]	-24.79	-10.49	-0.84	-11.52	-23.81	-3.75	-26.67	-25.20	-1.26
Δ ET()[1] [%]	-40.02	-10.55	-0.68	-26.34	-12.41	-0.83	-22.4	-19.97	-4.99

Note[1]*:* a negative value implies a *reduction* from the original *execution time*, while a positive value implies an *increment*.

References

[1] Apache Spark. Apache spark, . URL https://spark.apache.org.

[2] Apache Spark. Mllib | apache spark, . URL https://spark.apache.org/mllib/.

[3] Kaggle. Mushroom classification | kaggle. URL https://www.kaggle.com/datasets/uciml/mushroom-classification.

[4] UCI. Uci machine learning repository, . URL https://archive.ics.uci.edu/ml/index.php.

[5] UCI. Uci machine learning repository, . URL https://archive-beta.ics.uci.edu/ml/datasets/mushroom.

[6] Residentmario/missingno: Missing data visualization module for python. URL https://github.com/ResidentMario/missingno.

[7] Aleksey Bilogur. Missingno: a missing data visualization suite. *The Journal of Open Source Software*, 3:547, 2 2018. doi: 10.21105/JOSS.00547.

[8] Google Inc. Cloud computing services | google cloud. URL https://cloud.google.com/.

[9] Apache Spark. Binomial logistic regression - spark 3.2.1 documentation, . URL https://spark.apache.org/docs/latest/ml-classification-regression.html#binomial-logistic-regression.

[10] Apache Spark. Random forest classifier - spark 3.2.1 documentation, . URL https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier.

[11] Apache Spark. Gradient boosted tree classifier - spark 3.2.1 documentation, . URL https://spark.apache.org/docs/latest/ml-classification-regression.html#gradient-boosted-tree-classifier.

[12] Apache Spark. Ml tuning - spark 3.2.1 documentation, . URL https://spark.apache.org/docs/latest/ml-tuning.html.

[13] Apache Spark. Linear methods - ml - documentation, . URL https://spark.apache.org/docs/1.5.0/ml-linear-methods.html.

[14] Apache Spark. Performance tuning - spark 3.3.0 documentation, . URL https://spark.apache.org/docs/latest/sql-performance-tuning.html#:~:text=Adaptive%20Query%20Execution%20(AQE)%20is,sql.

[15] Apache Spark. pyspark.pandas.dataframe.spark.cache, . URL https://spark.apache.org/docs/3.2.0/api/python/reference/pyspark.pandas/api/pyspark.pandas.DataFrame.spark.cache.html.

[16] Behrooz Parhami. Amdahl's reliability law: A simple quantification of the weakest-link phenomenon. *Computer*, 48(7):55–58, 2015. doi: 10.1109/MC.2015.202.

A Appendix

A.1 Supplementary plots

Figure 19 depicts a *Count plot* of each feature group by *class* column.

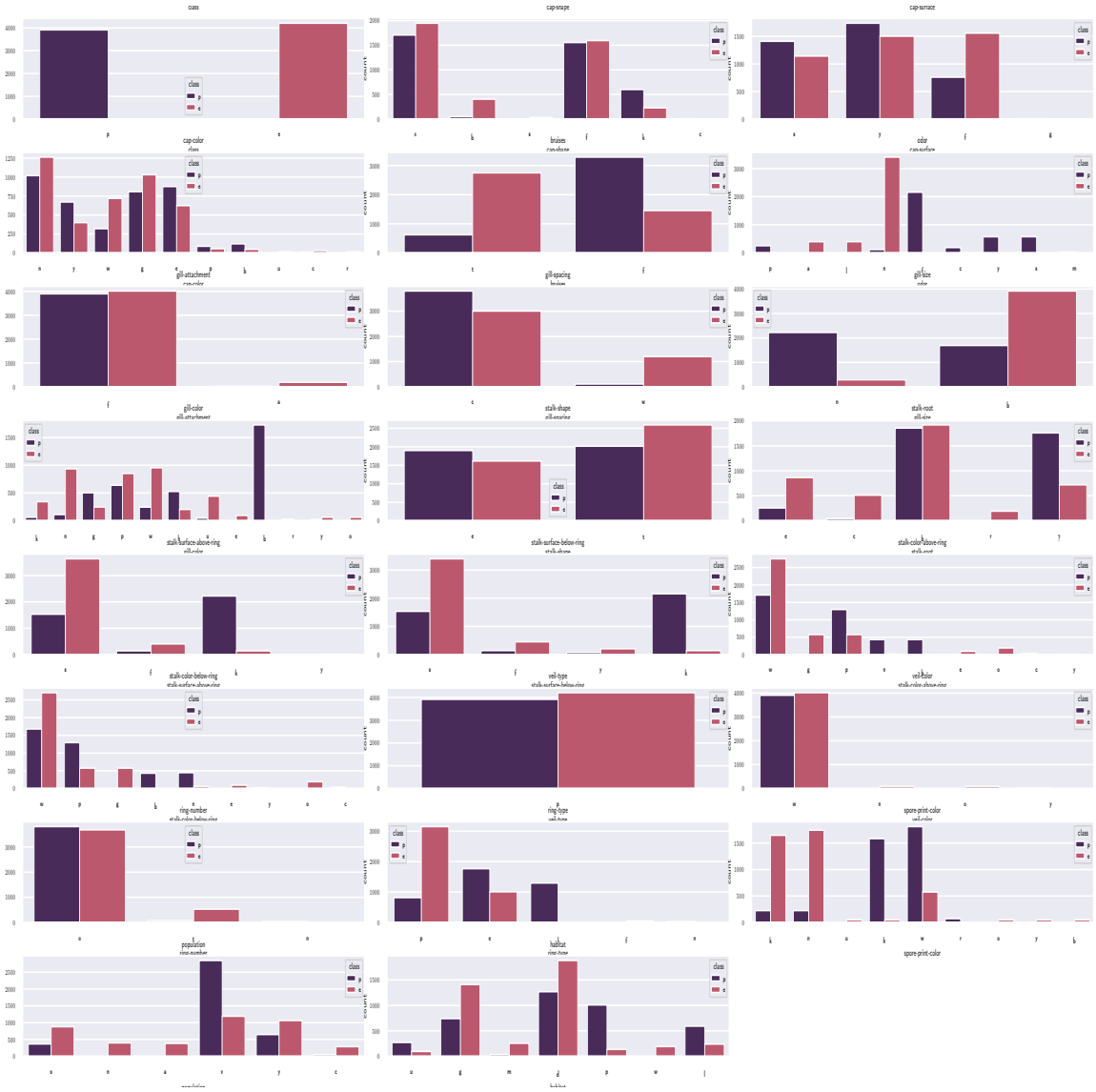

Figure 19: Dataset Features gruop-by Class