

The background image shows a dark, industrial landscape. In the foreground, there are large, dark piles of what appears to be coal or industrial waste. In the background, several tall smokestacks are visible, each emitting thick plumes of dark smoke that rise into a hazy, grey sky. The overall scene conveys a sense of heavy industry and environmental impact.

Ward Clustering on CO₂-GDP (1965-2018)

Danilo Meleleo, Giovanni Pietrucci e Daniele Polidori

Index


1. Dataset

- a. Data preparation

2. Implementation

- a. Spark setting
- b. Ward Algorithm
- c. Sequential vs Parallel vs Distributed

3. Performances

- a. Local (Sequential vs Parallel)
 - b. Google Cloud Platform (Weak vs Strong scaling)
 - c. Graphs
- 



1.

Dataset



Dataset

1. CO2 Emissions _ 1960 - 2018

<https://www.kaggle.com/datasets/kkhandekar/co2-emissions-1960-2018>

2. GDP per capita (constant LCU)

https://data.worldbank.org/indicator/NY.GDP.PCAP.KN?end=2021&most_recent_year_desc=false&start=2021&view=map



Data Preparation

Data Cleaning

- Drop unused column
- Drop NaN
- Discard useless group of country

Join data on Country Name

- CO2
- GDP

Minor Adjustment

- Round double values
- Fill missing values



2.

Implementation



Implementation



Implemented in
Scala+Spark

follows the **MapReduce** paradigm.



Main **collections**:

Df_annuali : List[Dataframe]

Forest : List[Int]

Dizionario : List[List[Int]]



Main **functions**:

Ward

Number Cluster

Spark setting

```
val spark: SparkSession = SparkSession
  .builder()
  //.master("local[*]")
  .master("yarn")
  .appName("Ward on annual co2-gdp" )
  .getOrCreate()

import spark.sqlContext.implicits._
```

We use:

- `.master("local[*]")` to run **locally**.
- `.master("yarn")` to run on a **distributed** cluster.

SQLImplicits:

- A collection of implicit methods for converting common Scala native objects or RDD into DataFrame/Datasets through implicit Encoders.

Ward's minimum variance method

1. **Agglomerative hierarchical clustering** method based on minimizations of the total within-cluster variance.
2. Start from a forest of **n clusters** each containing a single point p such as $p_i = (co2, gdp)$.
3. At each steps we search for **all** the possible merge **combinations** of **two clusters** available in the forest
4. Once **all** the clusters **combinations** are obtained, calculate the **midpoint** among all the points belonging to the clusters under analysis.

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```

Ward's minimum variance method

1. **Agglomerative hierarchical clustering** method based on minimizations of the total within-cluster variance.
2. Start from a forest of **n clusters** each containing a single point p such as $p_i = (co2, gdp)$.
3. At each steps we search for **all** the possible merge **combinations** of **two clusters** available in the forest
4. Once **all** the clusters **combinations** are obtained, calculate the **midpoint** among all the points belonging to the clusters under analysis.

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```

Ward's minimum variance method

1. **Agglomerative hierarchical clustering** method based on minimizations of the total within-cluster variance.
2. Start from a forest of **n clusters** each containing a single point p such as $p_i = (co2, gdp)$.
3. At each steps we search for **all** the possible merge **combinations of two clusters** available in the forest
Number of Combinations = $n! / [(n-2)! * (2)!]$
4. Once **all** the clusters **combinations** are obtained, calculate the **midpoint** among all the points belonging to the clusters under analysis.

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni =
    forest.filter(_ != -1).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```

Ward's minimum variance method

1. **Agglomerative hierarchical clustering** method based on minimizations of the total within-cluster variance.
2. Start from a forest of **n clusters** each containing a single point p such as $p_i = (co2, gdp)$.
3. At each steps we search for **all** the possible merge **combinations** of **two clusters** available in the forest
4. Once **all** the clusters **combinations** are obtained, calculate the **midpoint** among all the points belonging to the clusters under analysis.

```
var forest: List[Int] = List.range(0, original_length)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )
}
```

DISTANCE:

```
val all_x = points_new.map(dataFrame(_)._1) //CO2
val all_y = points_new.map(dataFrame(_)._2) //GDP

val ptMedio = Point(
  all_x.sum / points_new_length,
  all_y.sum / points_new_length
)
```

$$(\bar{X}, \bar{Y}) = \left(\frac{1}{k} \sum_{i=1}^k x_i, \frac{1}{k} \sum_{i=1}^k y_i \right)$$

Ward's minimum variance method

5. **Variance(SE)** calculation between the cluster's point and the midpoint calculated previously
6. Search for the cluster u with the **minimum variance**
7. **Delete** from the forest the cluster composing u
8. **Adding** u to the forest
9. **Repeat** until there's only one cluster in the forest (**root of the tree**)

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList
```

```
  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )
```

DISTANCE:

```
val error_square = (all_x zip all_y).map(
  punto =>
  ptMedio.error_square_fun( Point(punto._1, punto._2) )
).sum
```

```
def error_square_fun(other: Point): Double =
  pow(other.x - x, 2) + pow(other.y - y, 2)
}
```

$$SE = \sum_{i=1}^k \left((x_i - \bar{X})^2 + (y_i - \bar{Y})^2 \right)$$

Ward's minimum variance method

5. **Variance(SE)** calculation between the cluster's point and the midpoint calculated previously
6. Search for the cluster u with the **minimum variance**
7. **Delete** from the forest the cluster composing u
8. **Adding** u to the forest
9. **Repeat** until there's only one cluster in the forest (**root of the tree**)

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```

Ward's minimum variance method

5. **Variance(SE)** calculation between the cluster's point and the midpoint calculated previously
6. Search for the cluster u with the **minimum variance**
7. **Delete** from the forest the cluster composing u
8. **Adding** u to the forest
9. **Repeat** until there's only one cluster in the forest (**root of the tree**)

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```

Ward's minimum variance method

5. **Variance(SE)** calculation between the cluster's point and the midpoint calculated previously
6. Search for the cluster u with the **minimum variance**
7. **Delete** from the forest the cluster composing u
8. **Adding** u to the forest
9. **Repeat** until there's only one cluster in the forest (**root of the tree**)

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

  val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

  val error_list = combinazioni.par.map(
    distance(xy_zip, _, dizionario)
  )

  val coppia =
    combinazioni(error_list.indexOf(error_list.min))

  forest = forest.updated(coppia(0), -1)
  forest = forest.updated(coppia(1), -1)

  forest = forest :+ forest.length

  dizionario = dizionario :+ coppia
}
```


Ward's minimum variance method

5. **Variance(SE)** calculation between the cluster's point and the midpoint calculated previously
6. Search for the cluster u with the **minimum variance**
7. **Delete** from the forest the cluster composing u
8. **Adding** u to the forest
9. **Repeat** until there's only one cluster in the forest (root of the tree)

```
var forest: List[Int] = List.range(0, original_lenght)
var dizionario: List[List[Int]] = forest.map(List(_))

while(forest.count(_ > -1) > 1) {

    val combinazioni = forest.filter(_ != (-1)).combinations(2).toList

    val error_list = combinazioni.par.map(
        distance(xy_zip, _, dizionario)
    )

    val coppia =
        combinazioni(error_list.indexOf(error_list.min))

    forest = forest.updated(coppia(0), -1)
    forest = forest.updated(coppia(1), -1)

    forest = forest :+ forest.length

    dizionario = dizionario :+ coppia
}
```

Number Cluster function

It returns the **roots** of the clusters created.

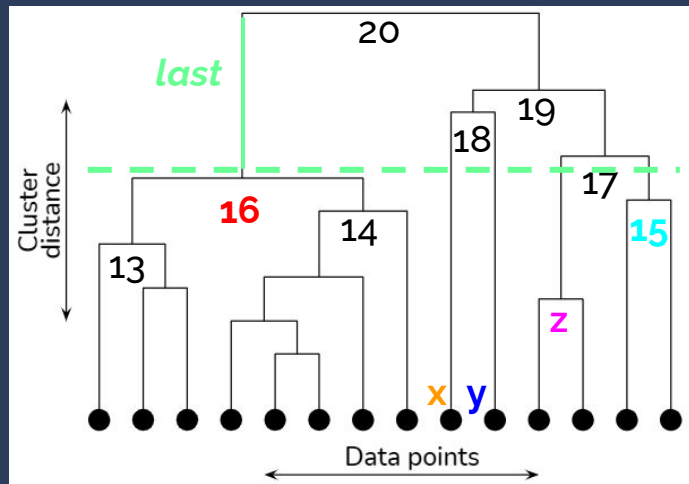
The **dendrogram**'s root is the couple (*dizionario.last*).

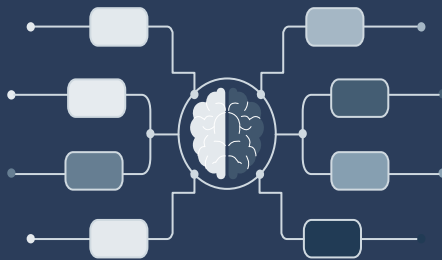
dizionario.last(o) has the longest branch of the couple.

Discard the values lower than **last** because I'm interested only in the ones included between the two in the **root**.

Flat and discard the values higher than **last**.

```
def number_cluster (dizionario: List[List[ Int]]): List[Int] = {  
  val last = dizionario.last(0)  
  val out = last :: dizionario.drop(last + 1).flatten.filter(_ < last)  
  out  
}
```



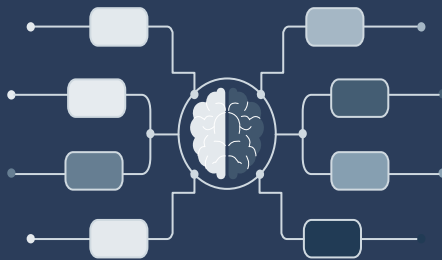


Implementation - Parallel

- **Data** parallelism:

```
// Mapping della lista di combinazioni con l'errore quadratico associato  
val error_list = combinazioni.par.map( distance(xy_zip, _, dizionario))
```

```
// Applicazione dell'algoritmo ward sui dataframe annuali  
val label_annuali = input_ward_annuali.par.map(t => ward(t._1, t._2, t._3, t._4))
```



Implementation - Distributed

- **RDD** usage:

We **convert** into RDD *input_ward_annuali*, a structure that contains the input for the annual Ward function.

We apply the Ward function on this RDD with a **map**, taking as input these values.

```
val RDD_inputWardAnnuali = spark.sparkContext.parallelize(input_ward_annuali)

// Applicazione dell'algoritmo ward sui dataframe annuali
val RDD_label_annuali = RDD_inputWardAnnuali.map(t => ward(t._1, t._2, t._3, t._4))
```



3.

Performances



Performances - Local

With `.master("local[*]")`

`data_gdp_co2.csv` (1965-2018)

	Sequential	Parallel	Distributed
Execution Time (s)	258	68	67

Scalability Test

Strong

We run the application on a different number of workers (with the same configuration) **without changing** the dataset.

Weak

We run the application on a different number of workers (with the same configuration) **scaling** the dataset.



Performances - Google Cloud Platform

Strong scaling

data_gdp_co2.csv (1965-2018)

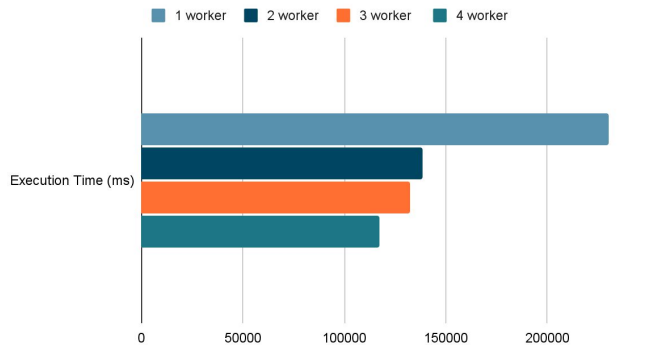
Master: **n2-standard-8** (8 vCPU, 32 GB di memoria)

Worker: **n1-custom** (6 vCPU, 22,5 GB di memoria)

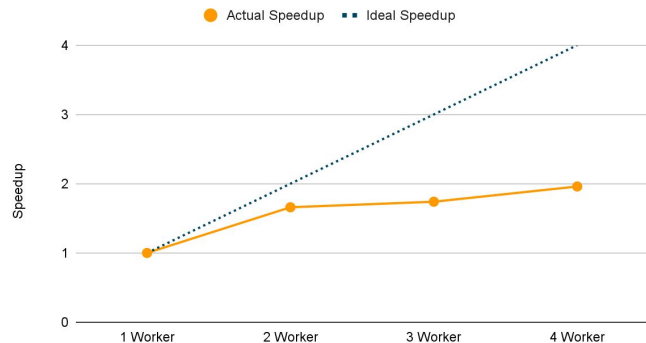
Worker	Time (ms)	Speedup
1	230'236	1
2	138'447	1,66
3	132'149	1,74
4	117'461	1,96

$$\text{Speedup} = t_1 / t_i$$

Strong: Execution Time



Speedup Test



Performances - Google Cloud Platform

Weak scaling

data_gdp_co2.csv (1967-2018)

Master: **n2-standard-8** (8 vCPU, 32 GB di memoria)

Worker: **n1-custom** (6 vCPU, 22.5 GB di memoria)

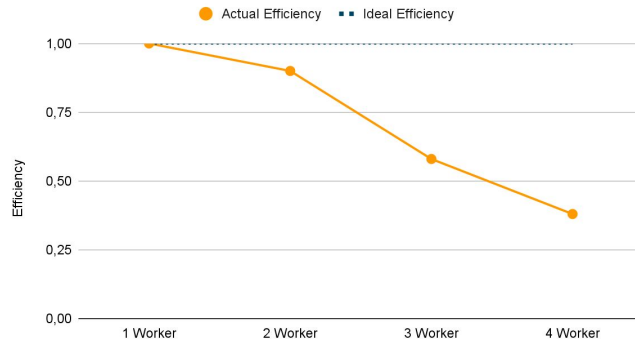
Dataset Size	Worker	Time (ms)	Efficiency
25%	1	55'417	1
50%	2	60'296	0,9
75%	3	95'255	0,58
100%	4	114'125	0,38

$$\text{Efficiency} = t_1 / t_i$$

Weak: Execution Time



Efficiency Test



Increasing Complexity

Change Dataset:

- **Old Dataset** = **214** Country per year
 - Number of Starting Combination = **22791**
- **New Dataset** = **264** Country per year
 - Number of Starting Combination = **34716**
 - Grouping Country
 - Cannot be showed on the map

Number of Combinations = $n! / [(n-2)! * (2)!]$

Old: $214! / [(214-2)! * (2)!] = 22791$

New: $264! / [(264-2)! * (2)!] = 34716$

Performances - Google Cloud Platform

Strong scaling

data_gdp_co2_with_groups.csv (1990-2013)

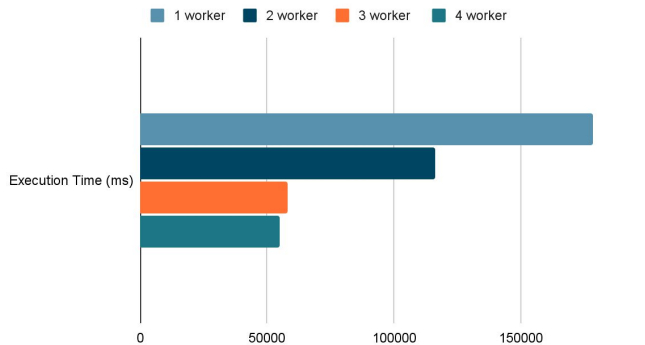
Master: **n2-standard-8** (8 vCPU, 32 GB di memoria)

Worker: **n1-custom** (6 vCPU, 22,5 GB di memoria)

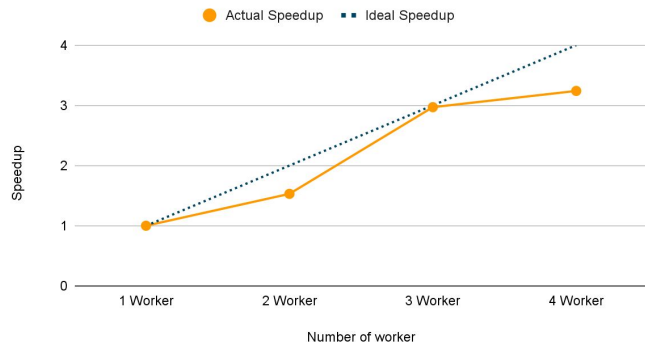
Worker	Time (ms)	Speedup
1	178'291	1
2	116'127	1,53
3	59'982	2,97
4	55'018	3,24

$$\text{Speedup} = t_1 / t_i$$

Strong: Execution Time



Speedup Test



Performances - Google Cloud Platform

Weak scaling

data_gdp_co2_with_groups.csv (1990-2013)

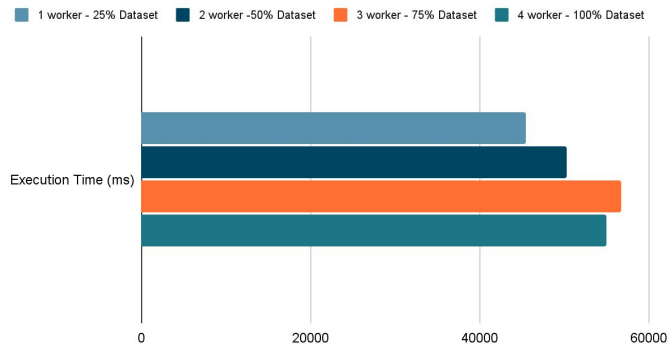
Master: **n2-standard-8** (8 vCPU, 32 GB di memoria)

Worker: **n1-custom** (6 vCPU, 22.5 GB di memoria)

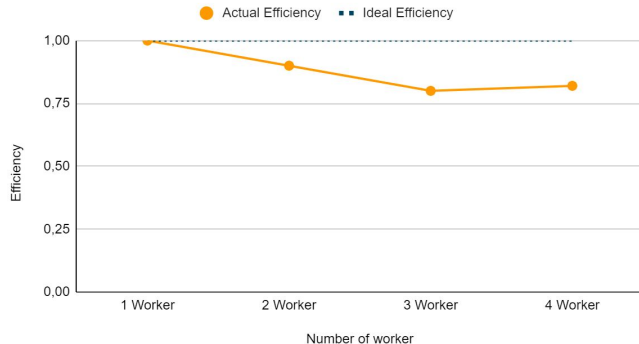
Dataset Size	Worker	Time (ms)	Efficiency
25%	1	45'480	1
50%	2	50'334	0,9
75%	3	56'775	0,80
100%	4	55'018	0,82

$$\text{Efficiency} = t_1 / t_i$$

Weak: Execution Time

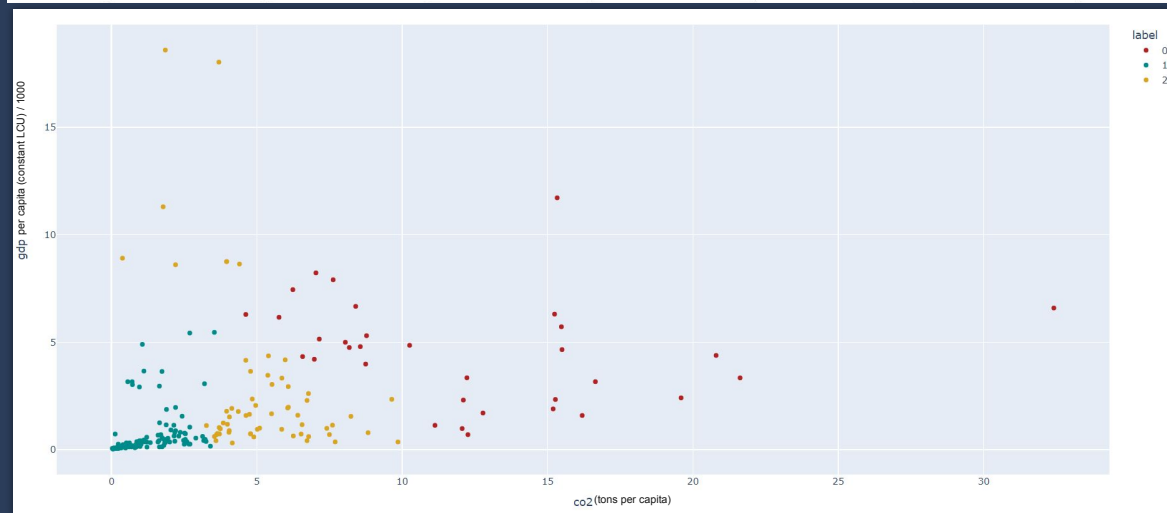


Efficiency Test



Graphs

Year 2018





DEMO