```
# Import the libraries
import numpy as np
import pandas as pd
from time import time
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from sklearn.datasets import fetch_kddcup99
import pickle
import os
import seaborn as sns
from pyspark.sql import SparkSession
from pprint import pprint
import logging
import warnings

# Setup the spark warnings
warnings.filterwarnings("ignore")
logging.getLogger('py4j').setLevel(logging.ERROR)
logging.getLogger('pyspark').setLevel(logging.ERROR)
log4j_conf_path = "file:///home/quivigorelli/Distributed-K-Means-
Clustering/spark/DistributedKmeans/log4j.properties"

# Functions
def labelToInt(label):
    '''
    Map from set of labels in original dataset (`strings`) into set of
natural numbers (`int`) for easier manipulation of rdd
    '''
    uniqueLabels=list(np.unique(y))
    return uniqueLabels.index(label)


def deleteBytes(datum):
    '''
    Clean dataset from categorical attributes, leaving numerical ones
    Arguments:
    One datum of the rdd.
    Return:
    Updated datum.
    '''
    x = datum[1]["x"]
    mask = [type(i) != bytes for i in x]
    datum[1]["x"] = np.asarray(x[mask])
    print(x)
    print(mask)
    return datum


def localPlusPlusInit(points, k):
    '''
```

```
    KMeans++ initialization.
    Arguments:
    `points`: array (n, dim) of points to be clustered;
    `k`: desired number of centroids.
    Returns:
    Initial array (k, dim) of centroids, k<=n.
    '''
    # Sample one point uniformly from points array
    C=points[np.random.choice(points.shape[0])]
    C=C[np.newaxis, :]

    for _ in range(k):
        # Compute array (n,1) of probabilities associated to each
point
        probs=np.min(np.sum((points[:,:,np.newaxis]-
C.T[np.newaxis,:,:])**2, axis=1), axis=1).flatten()
        # Normalize probability distribution
        probs=probs/np.sum(probs)

        # Draw one new centroid according to distrbution
        nextCentroid=points[np.random.choice(points.shape[0],
p=probs)][np.newaxis,:]
        # Add centroid to array
        C=np.vstack((C, nextCentroid))
    return C


def weightedAverage(group):
    """
    Compute weighted average of a group from a pd.DataFrame with point
coordinates, weights, clusterId.
    Utilized in local (non-distributed) version of Lloyds algorithm,
needed also in K-Means//
    """
    weight_column='weights'
    groupby_column='clusterId'
    columns_to_average = group.columns.difference([weight_column,
groupby_column])
    weighted_averages =
group[columns_to_average].multiply(group[weight_column], axis=0).sum()
/ group[weight_column].sum()
    return weighted_averages


def localLloyds(points, k, C_init=None, weights=None,
n_iterations=100, logDict=None):
    """
    Local (non-distributed) Lloyds algorithm
    Arguments:
    `points`: array (n, dim) of points to cluster;
```

```
    `k`: number of desired clusters;
    `C_init`: optional, array (k, dim) of initial centroids
    `weights`: optional, weights for weighted average in centroid re-
computing;
    `n_iterations`: optional, number of iteration in lloyds algorithm;
    `logDict`: optional, dictionary {'CostsKmeans', 'tIterations',
'tTotal'} to store cost and time info.
    Return:
    Array of expected centroids.
    """
    t0 = time()

    # Storing cost and time info
    my_kMeansCosts = []
    tIterations = []

    df=pd.DataFrame(points)

    # If weights not given, assume uniform weights for points
    if weights is None:
        weights=np.ones(shape=len(points))
    df['weights']=weights
    df['clusterId']=np.zeros(shape=len(points))

    # If no C_init, default to K-Means++ initialization
    if C_init is None:
        C=localPlusPlusInit(points, k)
    else:
        C=C_init

    clusterId=np.argmin(np.sum((points[:,:,np.newaxis]-
C.T[np.newaxis,:,:])**2, axis=1), axis=1)
    for iteration in range(n_iterations):
        t1=time()

        # Compute centroid given cluster
        df['clusterId']=clusterId
        C_df=df.groupby('clusterId')\
            .apply(weightedAverage)\
            .reset_index()

        # Compute cluster given centroid
        C_array=C_df[C_df.columns.difference(['weights',
'clusterId'])].reset_index(drop=True).to_numpy()
        squared_distances=np.sum((points[:,:,np.newaxis]-
C_array.T[np.newaxis,:,:])**2, axis=1)
        clusterId=np.argmin(squared_distances, axis=1)

my_cost=sum(squared_distances[np.arange(len(squared_distances)),
clusterId])
```

```
        my_kMeansCosts.append(my_cost)
        t2 = time()

        tIteration = t2 - t1
        tIterations.append(tIteration)

    tEnd = time()
    tTotal = tEnd - t0

    # Store cost and time info
    if logDict is not None:
        logDict["CostsKmeans"] = my_kMeansCosts
        logDict["tIterations"] = tIterations
        logDict["tTotal"] = tTotal

    return C_array


def minmaxRescale(datum, minS, maxS):
    """
    Rescale datum in [0,1] interval for better clusterization
    Arguments:
    `datum`: see rdd format;
    `minS`: array of min coordinate value among points for each
attribute;
    `maxS`: as `minS` with max.
    Return:
    Updated datum.
    """
    mask = np.array(minS < maxS).astype(bool)
    feature = datum[1]["x"]
    feature = (feature[mask] - minS[mask])/(maxS[mask] - minS[mask])
    return (datum[0], {"x": feature, "y": datum[1]["y"], "d2":datum[1]
["d2"]})


def selectCluster(datum, C, updateDistances=True):
    """
    Associate datum to its centroid and optionally updates squared
distance between them.
    Arguments:
    `datum`: see rdd format;
    `C`: array (k, len(datum[1]["x"]));
    `updateDistances`: if True, updates `datum[1]["d2"]` with squared
distance between datum point and closest centroid in C.
    Return:
    Updated datum.
    """
    distances = np.sum((datum[1]["x"] - C)**2, axis=1)
```

```
    print('distances: ',distances)
    clusterId = np.argmin(distances)
    if updateDistances is True:
        return (clusterId, {'x':datum[1]['x'], 'y':datum[1]['y'],
'd2':distances[clusterId]})
    else:
        return (clusterId, datum[1])


def updateCentroids(Rdd):
    """
    Update centroids as spatial average of cluster points
    Argument:
    `Rdd`: see rdd format;
    Return:
    Updated array of centroids.
    """
    C=Rdd.mapValues(lambda xy: (xy['x'], 1))\
              .reduceByKey(lambda a,b : (a[0]+b[0], a[1]+b[1]))\
              .mapValues(lambda a:a[0]/a[1])\
              .values()\
              .collect()
    C=np.array(C) #check later more carefully if causes some overhead
    return C


def updateDistances(Rdd, C):
    """
    Update Rdd with square distances from centroids, given Rdd with
clusters already assigned to each point
    Arguments:
    `Rdd`: see rdd format;
    `C`: array of cluster centroids.
    Return:
    Updated rdd.
    """
    def datumUpdate(datum, C):
        '''
        Update a datum of the rdd with distance from assigned centroid
        '''
        d2=np.sum((datum[1]['x']-C[datum[0]])**2)
        #return datum
        return (datum[0], {"x": datum[1]["x"], "y": datum[1]["y"],
"d2":d2})
    Rdd=Rdd.map(lambda datum:datumUpdate(datum, C))
    return Rdd


def cost(Rdd):
    """
```

```
    Calculate global cost of clusterization, from an Rdd with
distances from centroids already updated
    """
    my_cost=Rdd.map(lambda datum : datum[1]['d2'])\
               .reduce(lambda a,b: a+b)
    return my_cost


def kMeans(Rdd, C_init, maxIterations, logParallelKmeans=None):
    """
    Distributed (parallel) Lloyds algorithm
    Arguments:
    `Rdd`: see rdd format;
    `C_init`: array (k, dim) of initial centroids;
    `maxIterations`: max number of iterations;
    `logParallelKmeans`: optional, dictionary {'CostsKmeans',
'tIterations', 'tTotal'} to store cost and time info.
    Return:
    Array of expected centroids.
    """

    t0 = time()

    # Storing cost and time info
    my_kMeansCosts = []
    tIterations = []
    C=C_init

    for t in range(maxIterations):
        t1 = time()
        RddCached = Rdd.map(lambda datum: selectCluster(datum,
C)).persist() ###

        # Now we compute the new centroids by calculating the averages
of points belonging to the same cluster.
        C=updateCentroids(RddCached)
        my_cost = cost(RddCached)

        my_kMeansCosts.append(my_cost)
        t2 = time()

        tIteration = t2 - t1
        tIterations.append(tIteration)

        #RddCached.unpersist()

        # Break loop if convergence of cost is reached
        if (len(my_kMeansCosts) > 1) and (my_kMeansCosts[-1] >
0.999*my_kMeansCosts[-2]):
            break
```

```
    tEnd = time()
    tTotal = tEnd - t0

    # Store cost and time info in argument dictionary
    if logParallelKmeans is not None:
        logParallelKmeans["CostsKmeans"] = my_kMeansCosts
        logParallelKmeans["tIterations"] = tIterations
        logParallelKmeans["tTotal"] = tTotal

    return C


def naiveInitFromSet(Rdd, k, spark_seed=12345,logNaiveInit=None):
    """
    Uniform sampling of k points from Rdd
    Arguments:
    `Rdd`: see rdd structure;
    `k`: desired number of clusters;
    `spark_seed`: optional, seed for spark random sampling;
    `logNaiveInit`: optional, dictionary {'tTotal'} to store time
info.
    Return:
    Initial array (k, dim) of centroids.
    """
    t0 = time()
    # Sampling. Replacement is set to False to avoid coinciding
centroids BUT no guarantees that in the original dataset all points
are distinct!!!
    kSubset=Rdd.takeSample(False, k, seed=spark_seed)
    C_init=np.array([datum[1]['x'] for datum in kSubset])

    tEnd = time()

    if logNaiveInit is not None:
        logNaiveInit["tTotal"] = tEnd - t0

    return C_init


def naiveInitFromSpace(k, dim):
    """
    Uniform drawing of k points from euclidean space assuming the Rdd
has been mapped into a [0,1]^dim space
    Arguments:
    `k`: desired number of clusters;
    `dim`: dimensionality of points space.
    Return:
    Initial array (k, dim) of centroids.
    """
```

```
    C_init=np.random.uniform(size=(k,dim))
    return C_init


def parallelInit(Rdd, k, l, logParallelInit=None):
    """
    Parallel initialization
    Arguments:
    `Rdd`: see rdd structure;
    `k`: desired number of clusters;
    `l`: coefficient to adjust sampling probability in order to obtain
at least k centroids;
    `logParallelInit`: optional, dictionary {'CostsKmeans',
'tIterations', 'tTotal'} to store cost and time info.
    Return:
    Initial array (k, dim) of centroids.
    """
    t0 = time()

    # initialize C as a point in the dataset
    C=naiveInitFromSet(Rdd, 1)

    # associate each datum to the only centroid (computed before) and
computed distances and cost
    Rdd=Rdd.map(lambda datum : (0, datum[1]))
    Rdd=updateDistances(Rdd, C).persist() ###

    my_cost=cost(Rdd)

    # number of iterations (log(cost))
    n_iterations=int(np.log(my_cost))
    if(n_iterations<1): n_iterations=1

    tSamples = []
    tCentroids = []
    CostInits = [my_cost]
    # iterative sampling of the centroids
    for _ in range(n_iterations):

        t1=time()
        # sample C' according to the probability
        C_prime=Rdd.filter(lambda datum :
np.random.uniform()<l*datum[1]['d2']/my_cost)\
                   .map(lambda datum : datum[1]['x'])\
                   .collect()
        C_prime=np.array(C_prime)
        t2=time()

        # stack C and C', update distances, centroids, and cost
        if (C_prime.shape[0]>0):
```

```
            C=np.vstack((C, C_prime))

            #Rdd.unpersist() ###
            Rdd=Rdd.map(lambda datum: selectCluster(datum,
C)).persist() ###

            my_cost=cost(Rdd)
        t3=time()

        tSample = t2 -t1
        tCentroid = t3 - t2
        tSamples.append(tSample)
        tCentroids.append(tCentroid)
        CostInits.append(my_cost)

    #erase centroids sampled more than once
    C=C.astype(float)
    C=np.unique(C, axis=0)
    Rdd=Rdd.map(lambda datum: selectCluster(datum, C))

    #compute weights of centroids (sizes of each cluster) and put them
in a list whose index is same centroid index as C
    wx=Rdd.countByKey()
    weights=np.zeros(len(C))
    weights[[list(wx.keys())]]=[list(wx.values())]

    #subselection of k centroids from C, using local Lloyds algorithm
with k-means++ initialization
    if C.shape[0]<=k:
        C_init=C
    else:
        C_init=localLloyds(C, k, weights=weights, n_iterations=100)
#can be set to lloydsMaxIterations for consistency TODO

    tEnd = time()

    if logParallelInit is not None:
        logParallelInit["tSamples"] = tSamples
        logParallelInit["tCentroids"] = tCentroids
        logParallelInit["CostInit"] = CostInits
        logParallelInit["tTotal"] = tEnd - t0

    #Rdd.unpersist() ###
    return C_init

def predictedCentroidsLabeler(C_expected, C_predicted):
    """
    Associate expected and predicted centroids based on distance.
    Parameters:
    `C_expected`: array (k, dim) of expected centroids;
```

```
    `C_predicted`: array (k,dim) of predicted centroids;
    Return:
    List of labels, one for each expected centroid and pointing to its
nearest predicted centroid;
    List of corresponding distances.
    """
    # Compute the distance matrix
    distMatrix=np.sum((C_expected[:,:,np.newaxis]-
C_predicted.T[np.newaxis, :,:])**2,axis=1)
    # The labeler i-th entry j, tells that i-th centroid of C_expected
is associated to j-th element of C_predicted
    labeler=np.argmin(distMatrix,axis=1)
    # Square distance of element of C_expected to nearest point in
C_predicted

distances=np.sqrt(np.array(distMatrix[np.arange(len(distMatrix)),label
er]).astype(float))
    return labeler, distances


def nearestCentroidDistances(C):
    """
    Associate each centroid to the distance of the nearest one
    Parameters:
    `C`:  array (k, dim) of centroids;
    Return:
    List of labels, one for each centroid and pointing to its nearest
centroid;
    List of corresponding distances.
    """
    # Compute the distance matrix
    distMatrix=np.sum((C[:,:,np.newaxis]-
C.T[np.newaxis, :,:])**2,axis=1)
    distMatrix+=np.diag(np.repeat(np.inf, distMatrix.shape[0]))

    # The labeler i-th entry j, tells that i-th centroid of C_expected
is associated to j-th element of C_predicted
    labeler=np.argmin(distMatrix,axis=1)

    # Square distance of element of C_expected to nearest point in
C_predicted

distances=np.sqrt(np.array(distMatrix[np.arange(distMatrix.shape[0]),l
abeler]).astype(float))
    return labeler, distances
```

Spark setup

```
# Build a spark session
spark = SparkSession.builder \
    .master("spark://spark-master:7077")\
    .appName("ResultsApplication")\
    .config("spark.executor.memory", "512m")\
    .getOrCreate()

# Create a spark context and set the logLevel
sc = spark.sparkContext
sc.setLogLevel("ERROR")
```

```
24/07/08 18:51:54 WARN SparkSession: Using an existing Spark session;
only runtime SQL configurations will take effect.
```

We are running applications on a Spark cluster consisting of one master node and two worker nodes (slaves). Each worker node is equipped with 4 CPU cores and 6.8 GB of memory. The master node manages the cluster resources and job scheduling, while the worker nodes execute the tasks assigned by the master, enabling efficient distributed data processing.

Data exploration

The *KDD Cup 1999* is a widely dataset used for evaluating network intrusion detection systems. It contains simulated network traffic data, with each instance representing a connection and labeled as either normal or one of various types of network attacks. The full dataset includes 41 features capturing different characteristics of the network connections.

For more information, visit the official website.

```
# Fetch the data
data = fetch_kddcup99(return_X_y = True)
```

In this scenario, we loaded the entire dataset into memory to parallelize it later, which is not always feasible due to memory constraints. A more general approach would be to parallelize the dataset directly from the file using Spark. However, we chose to repeat certain operations both on the parallelized dataset and locally to observe and compare the differences in terms of time efficiency.

```
print("The number of records is ", data[0].shape[0])
print("The dimensionality of each record is ", data[0].shape[1])
```

```
The number of records is  494021
The dimensionality of each record is  41
```

```
# Divide samples (attributes) and features (targets)
x = data[0]
y = data[1]
```

```
# Optionally, shuffle the dataset
shuffling=True

if shuffling:
    shuffled_indices = np.random.permutation(len(y))
    x=x[shuffled_indices]
    y=y[shuffled_indices]

# Optionally, subselect the dataset (suggested to shuffle first)
subLen = 300_000
x = x[:subLen,]
y = y[:subLen]

# Parallallelize the dataset
nSlice = None # number of partition (if None, returns the total number
of cores of the workers)
Rdd = sc.parallelize([(None, {"x": x[i],"y": y[i], "d2":None}) for i
in range(len(y))], numSlices = nSlice)
```

We parallelized the dataset using a custom data structure where each record is represented as a tuple with two elements. The first element is an integer (defaulting to None) that references the index of the centroids array. The second element is a dictionary containing three keys: the attribute (an array of 41 entries), the target, and the squared distance from the selected centroid.

```
# Check the number of partitions
print("The number of partions used is: ", Rdd.getNumPartitions())

The number of partions used is:  8
```

In the following, we explore the dataset to understand the attributes types and the number of unique elements for each feature.

```
# Attribute names from the official documentation
attributes = [
    "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"
]

# Persist the Rdd used to work on each individual record (datum)
RddX = Rdd.map(lambda datum: datum[1]["x"]).persist()

# Collect the type of each attribute
typeElement = RddX.map(lambda x: [set([type(x[i])]) for i in
range(len(x))])\
                  .reduce(lambda a, b: [a[i].union(b[i]) for i in
range(len(a))])



# Convert the types into strings
types_str = [str(t) for t in typeElement]

# Show the results with matplotlib
fig, ax = plt.subplots()

# Hide the axes
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
```

```
ax.set_frame_on(False)

# Create the table
table_data = [[attributes[i], types_str[i]] for i in
range(len(types_str))]
column_labels = ("Index", "Type")
table = ax.table(cellText=table_data, colLabels=column_labels,
cellLoc='center', loc='center')

# Adjust the table
table.auto_set_font_size(False)
table.set_fontsize(6)
table.scale(1.2, 1.2)
```

Index	Type
duration	{<class 'int'>}
protocol_type	{<class 'bytes'>}
service	{<class 'bytes'>}
flag	{<class 'bytes'>}
src_bytes	{<class 'int'>}
dst_bytes	{<class 'int'>}
land	{<class 'int'>}
wrong_fragment	{<class 'int'>}
urgent	{<class 'int'>}
hot	{<class 'int'>}
num_failed_logins	{<class 'int'>}
logged_in	{<class 'int'>}
num_compromised	{<class 'int'>}
root_shell	{<class 'int'>}
su_attempted	{<class 'int'>}
num_root	{<class 'int'>}
num_file_creations	{<class 'int'>}
num_shells	{<class 'int'>}
num_access_files	{<class 'int'>}
num_outbound_cmds	{<class 'int'>}
is_host_login	{<class 'int'>}
is_guest_login	{<class 'int'>}
count	{<class 'int'>}
srv_count	{<class 'int'>}
serror_rate	{<class 'float'>}
srv_serror_rate	{<class 'float'>}
rerror_rate	{<class 'float'>}
srv_rerror_rate	{<class 'float'>}
same_srv_rate	{<class 'float'>}
diff_srv_rate	{<class 'float'>}
srv_diff_host_rate	{<class 'float'>}
dst_host_count	{<class 'int'>}
dst_host_srv_count	{<class 'int'>}
dst_host_same_srv_rate	{<class 'float'>}
dst_host_diff_srv_rate	{<class 'float'>}
dst_host_same_src_port_rate	{<class 'float'>}
dst_host_srv_diff_host_rate	{<class 'float'>}
dst_host_serror_rate	{<class 'float'>}
dst_host_srv_serror_rate	{<class 'float'>}
dst_host_rerror_rate	{<class 'float'>}
dst_host_srv_rerror_rate	{<class 'float'>}

We examined the number of **unique values for each attribute dimension**. This helps us understand the diversity of inputs in the dataset and provides insight into the characteristics and significance of each feature.

We performed the task both in parallel and locally. Not surprisingly, executing it in parallel takes considerably longer than executing it locally. This is because the "count unique" query is not easily parallelizable and is more efficient when processed directly in the RAM of a single device, minimizing overhead. However, this approach assumes that the entire dataset can fit into the RAM of the device, which is not always feasible.

```
%%time
# Collect the number of unique entries for each dimension of the
attributes
uniquesParallel = []
for i in range(41):
    # Map to select the i-th attribute, then count unique occurencies
    s = RddX.map(lambda x: x[i])\
        .distinct()\
        .count()
    uniquesParallel.append(s)

# Unpersist the RddX RDD
RddX.unpersist()
```

```
CPU times: user 224 ms, sys: 87.5 ms, total: 311 ms
Wall time: 32.7 s
```

```
PythonRDD[1] at RDD at PythonRDD.scala:53
```

```
print("The number of uniques for each dimension is", uniquesParallel)
```

```
The number of uniques for each dimension is [1691, 3, 65, 11, 2773,
8635, 2, 3, 2, 20, 5, 2, 14, 2, 3, 13, 12, 3, 5, 1, 1, 2, 456, 426,
89, 43, 73, 45, 96, 71, 62, 256, 256, 101, 101, 101, 65, 96, 57, 101,
101]
```

```
%%time
# Count unique occurencies locally with numpy
uniques = []
for i in range (x.shape[1]):
    k = (len(np.unique(x[:, i])))
    uniques.append(k)

print("The uniques are:", uniques)
```

```
The uniques are: [1691, 3, 65, 11, 2773, 8635, 2, 3, 2, 20, 5, 2, 14,
2, 3, 13, 12, 3, 5, 1, 1, 2, 456, 426, 89, 43, 73, 45, 96, 71, 62,
256, 256, 101, 101, 101, 65, 96, 57, 101, 101]
```

```
CPU times: user 7.16 s, sys: 11.2 ms, total: 7.18 s
Wall time: 7.19 s
```

Now we do the same for the **unique target values**

```
%%time
# Persist the RDD for the target values
RddY = Rdd.map(lambda datum: datum[1]["y"]).persist()

# Count the number of unique occurencies
kTrue = RddY.distinct()\
        .count()
```

```
[Stage 83:====================>                                    (3
+ 5) / 8]

CPU times: user 12.2 ms, sys: 0 ns, total: 12.2 ms
Wall time: 754 ms
```

The number of unique targets represents the **true value of** k in the KMeans algorithm. In practical applications, this value is typically unknown, but currently, our focus is on assessing data processing performance rather than iterating over possible k values.

```
print("The true number of the classes is", kTrue)
```

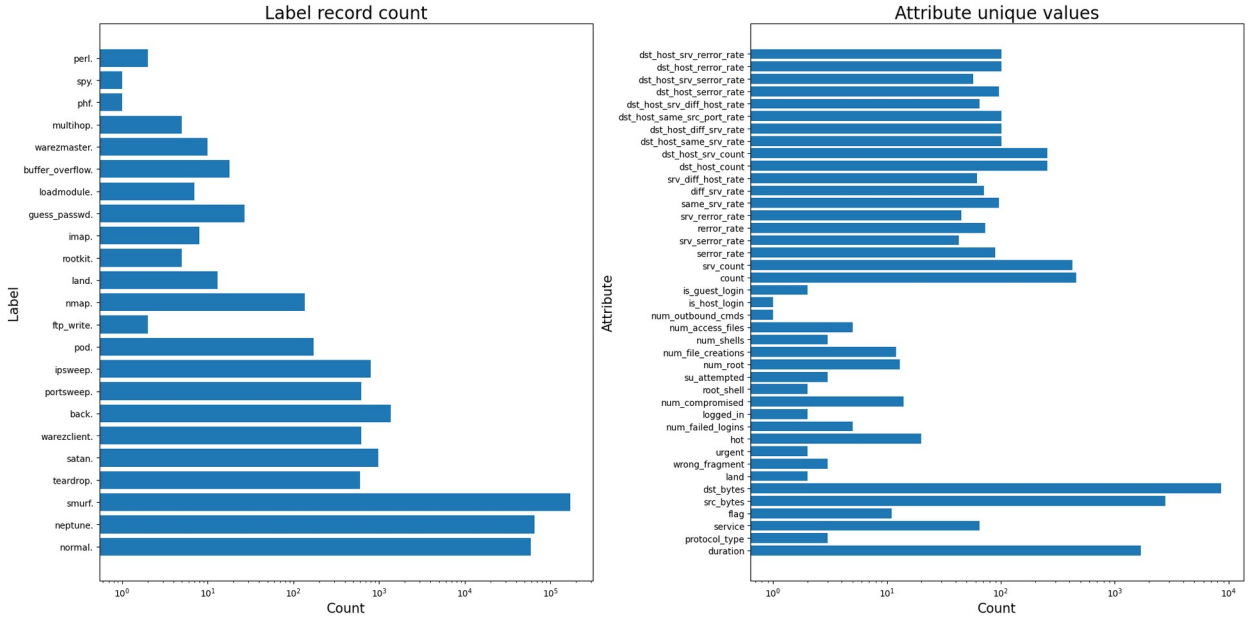
```
The true number of the classes is 23
```

```
%%time
# Count the number of records for each target label
uniquesParallely = RddY.countByValue()

# Unpersist the target RDD
RddY.unpersist()

print("The y things are:", uniquesParallely)
```

```
The y things are: defaultdict(<class 'int'>, {b'normal.': 59005,
b'neptune.': 65065, b'smurf.': 170553, b'teardrop.': 596, b'satan.':
972, b'warezclient.': 622, b'back.': 1360, b'portsweep.': 619,
b'ipsweep.': 801, b'pod.': 171, b'ftp_write.': 2, b'nmap.': 137,
b'land.': 13, b'rootkit.': 5, b'imap.': 8, b'guess_passwd.': 27,
b'loadmodule.': 7, b'buffer_overflow.': 18, b'warezmaster.': 10,
b'multihop.': 5, b'phf.': 1, b'spy.': 1, b'perl.': 2})
CPU times: user 4.05 ms, sys: 0 ns, total: 4.05 ms
Wall time: 377 ms
```

```
# Plot the occurrencies
fig, ax = plt.subplots(1, 2, figsize=(20, 10))
```

```
title_fontsize = 20
label_fontsize = 15

# Labels
yUnique = [a.decode('utf-8') for a in uniquesParallely.keys()]
xUnique = list(uniquesParallely.values())
ax[0].barh(yUnique, xUnique)
ax[0].set_xlabel("Count", fontsize=label_fontsize)
ax[0].set_ylabel("Label", fontsize=label_fontsize)
ax[0].set_xscale("log")
ax[0].set_title("Label record count", fontsize=title_fontsize)

# Attributes
ax[1].barh(attributes, uniquesParallel)
ax[1].set_xlabel("Count", fontsize=label_fontsize)
ax[1].set_ylabel("Attribute", fontsize=label_fontsize)
ax[1].set_xscale("log")
ax[1].set_title("Attribute unique values", fontsize=title_fontsize)

fig.tight_layout()
```



Now, we remove the categorical variables that we observed are linked to the Byte types. Afterward, we persist the RDD for future use.

```
# Remove the categorical attributes and persist
Rdd = Rdd.map(deleteBytes)\
         .persist()

# Print the dimensionality after the cut off
numberColumns = len([i for i in typeElement if i != set([bytes])])
```

```
print("The number of columns (dimensionality) after filtering the
bytes is", numberColumns)
```

```
The number of columns (dimensionality) after filtering the bytes is 38
```

Parallel data processing

Now, we execute a single run of the algorithm for additional one-shot analysis. The comprehensive benchmarking for the full analysis is conducted in another notebook, and the results will be presented in the next section of this report.

```
%%time
# Collect the maxiumum and minimum value for each dimension
maxS = Rdd.map(lambda datum: datum[1]["x"])\
          .reduce(lambda a, b: np.maximum(a, b))
minS = Rdd.map(lambda datum: datum[1]["x"])\
          .reduce(lambda a, b: np.minimum(a, b))

# Rescale each dimension in the [0, 1] interval
# Remove the attributes for which max = min
Rdd = Rdd.map(lambda datum: minmaxRescale(datum, minS, maxS))\
         .persist()
```

```
CPU times: user 15.5 ms, sys: 0 ns, total: 15.5 ms
Wall time: 6.82 s
```

```
%%time
# Define the log dictionary
logParallelInit = {}
logParallelKmeans = {}

# Define k and l for KMeans|| initialization
k=kTrue
l=k*2

# KMeans|| initialization
C_init = parallelInit(Rdd, k, l, logParallelInit)
# Lloyd iterations on the resulting centroids
C = kMeans(Rdd, C_init, 15, logParallelKmeans)
```

```
CPU times: user 2.43 s, sys: 85.2 ms, total: 2.51 s
Wall time: 15min 30s
```

KMeans performance

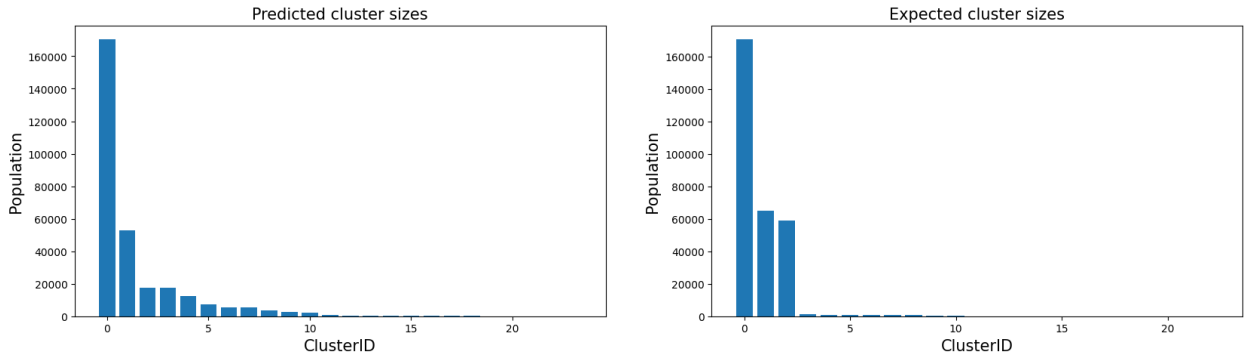
Now, we define a custom pipeline to evaluate the algorithm's performance. The test focuses on assessing the typical cluster distance using the true labels and comparing them with the predicted labels.

```
# Select again the clusters and persist the RDD
Rdd=Rdd.map(lambda datum: selectCluster(datum, C)).persist()
C_predicted=C # For clarity of notation

# Compute the predicted population of each cluster
predictedPopulationDict=Rdd.countByKey()
predictedPopulationDict = dict(sorted(predictedPopulationDict.items(),
key=lambda item: item[1], reverse=True))



# RDD based on the true target values
expectedRdd=Rdd.map(lambda datum:(labelToInt(datum[1]['y']),
datum[1])) # Use true values as ClusterID
C_expected=updateCentroids(expectedRdd) # Compute the expected
centroids
expectedRdd=updateDistances(expectedRdd, C_expected) # Update the
squared distance values

# Compute the expected population of each cluster
expectedPopulationDict=Rdd.map(lambda datum:(datum[1]['y'],
datum[1])).countByKey()
expectedPopulationDict = dict(sorted(expectedPopulationDict.items(),
key=lambda item: item[1], reverse=True))
```

As a sanity check, we compare the population of the expected and predicted clusters

```
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
label_fontsize = 15
title_fontsize = 20
ax[0].bar(np.arange(len(predictedPopulationDict)),
predictedPopulationDict.values())
ax[0].set_title("Predicted cluster sizes", fontsize=label_fontsize)
ax[0].set_xlabel("ClusterID", fontsize=label_fontsize)
ax[0].set_ylabel("Population", fontsize=label_fontsize)

ax[1].bar(np.arange(len(expectedPopulationDict)),
expectedPopulationDict.values())
ax[1].set_title("Expected cluster sizes", fontsize=label_fontsize)
```

```
ax[1].set_xlabel("ClusterID", fontsize=label_fontsize)
ax[1].set_ylabel("Population", fontsize=label_fontsize)

Text(0, 0.5, 'Population')
```



Next, we calculate the mean squared error between the expected centroids and their nearest predicted centroids after associating each expected centroid with the closest predicted one.

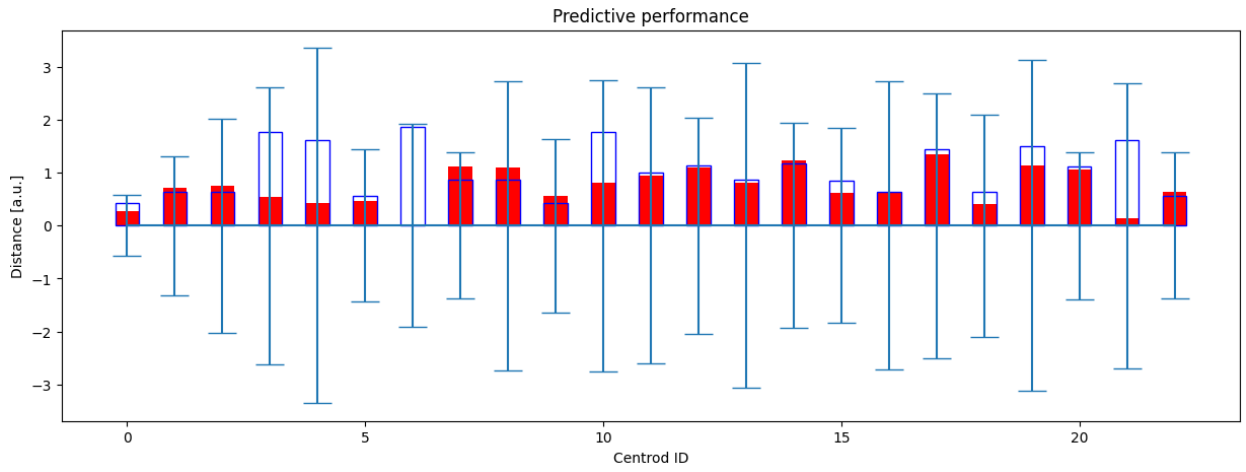
```
# Associate predicted and expected centroids
labeler,
distancesPredictedExpected=predictedCentroidsLabeler(C_expected,
C_predicted)
# Compute the MSE
meanSquaredError=np.sum(distancesPredictedExpected**2)/len(distancesPr
edictedExpected)

print('Index of nearest predicted centroid to each expected
centroid:', labeler)
print('Distances between expected centroids and nearest predicted
centroids:', distancesPredictedExpected)
print('MeanSquaredError: ', meanSquaredError)

Index of nearest predicted centroid to each expected centroid: [18 16
17 15  7 17 12  4 16 18  3  0  0 22 17 16  4 21 16 22 16  5 17]
Distances between expected centroids and nearest predicted centroids:
[2.67140854e-01 7.04271191e-01 7.60096473e-01 5.35103343e-01
 4.22126474e-01 4.69552730e-01 8.26663976e-04 1.11451630e+00
 1.10092859e+00 5.52633364e-01 8.04691149e-01 9.50619689e-01
 1.09097151e+00 8.03570042e-01 1.22689293e+00 6.16942607e-01
 6.23929935e-01 1.34857048e+00 4.02928522e-01 1.12987874e+00
 1.05705078e+00 1.34747947e-01 6.41524997e-01]
MeanSquaredError:  0.654639967564056

# Compute the typical distance of each "true" cluster (standard
deviation)
stdExpected=expectedRdd.mapValues(lambda datum1: (datum1['d2'], 1))\
                       .reduceByKey(lambda a,b : (a[0]+b[0],
a[1]+b[1]))\
```

```
                        .mapValues(lambda a:np.sqrt(a[0]/a[1]))\
                        .collect()

stdExpected=list(zip(*stdExpected))
```

In the following plot we show:

- with blue errorbars the typical distance of each expected cluster with respect to the expected centroid
- with blue boxes the distances to each expected centroid of its nearest **expected** centroid (nearest neighbour)
- with red bars the distances to each expected centroid of its nearest **predicted** centroid (error with the respect to the optimal value)

```
plt.figure(figsize=(15, 5))
xloc = np.arange(len(C_expected))
plt.errorbar(np.arange(len(C_expected)),
np.zeros(shape=(len(stdExpected[0]))), stdExpected[1], capsize=10)
plt.bar(xloc, distancesPredictedExpected, width=0.5, color='red')
plt.bar(xloc, nearestCentroidDistances(C_expected)[1], width=0.5,
edgecolor='blue', facecolor='none')

plt.title("Predictive performance")
plt.xlabel("Centrod ID")
plt.ylabel("Distance [a.u.]")

Text(0, 0.5, 'Distance [a.u.]')
```



Benchmarks

Import the log files and description

At each run of the algorithm (pre-processing, initialization, and Lloyd iterations), we saved a log file with information about time efficiency and algorithm performance. Each file, saved as a pickle, contains a dictionary with data from a scan over a predefined set of workers to compare parallel performance. Each dictionary has a nested structure of "dictionaries of dictionaries" to handle different run contexts in a unified and flexible manner. This flexibility comes with the drawback of a more complex usage.

```
# Location of the log files
filesP = os.listdir("dataP")  # Parallel initialization
filesR = os.listdir("dataR")  # Naive (random) initialization

print("filesP has ", len(filesP), "files")
print("filesR has ", len(filesR), "files")
```

```
filesP has  2 files
filesR has  2 files
```

```
# Load the pickles with the logs in two lists (parallel and random)
totalLogP = []
for file in filesP:
    pathFile = os.path.join("dataP", file)
    with open(pathFile, 'rb') as f:
        s = pickle.load(f)
    totalLogP.append(s)

totalLogR = []
for file in filesR:
    pathFile = os.path.join("dataR", file)
    with open(pathFile, 'rb') as f:
        s = pickle.load(f)
    totalLogR.append(s)
```

Here examples of nested dictionary structures. The first one refers to a "Parallel Init" run, while the second one to a "Random Init" run.

```
# Parallel init example
pprint(totalLogP[0])
```

```
{'tDurationsParallel': {'Number of partition128': 1126.6152312755585,
                        'Number of partition16': 915.2583429813385,
                        'Number of partition2': 2916.459883928299,
                        'Number of partition256': 1316.1773827075958,
                        'Number of partition32': 1161.9603824615479,
                        'Number of partition4': 1718.7826912403107,
                        'Number of partition64': 972.8995172977448,
```

```
                         'Number of partition8': 876.2816195487976},
 'tPreOperationsParallel': {'Number of partition128':
19.820308446884155,
                            'Number of partition16':
10.725661039352417,
                            'Number of partition2': 18.64969515800476,
                            'Number of partition256':
29.06898808479309,
                            'Number of partition32':
13.518644094467163,
                            'Number of partition4':
12.929564952850342,
                            'Number of partition64':
14.573980569839478,
                            'Number of partition8':
10.530800104141235},
 'totalLogParallelInit': {'Number of partition128': {'CostInit':
[843029.9190250103,

41242.46035814129,

11618.857152040846,

6515.874452333205,

4379.615086935071,

3232.9190282315076,

2665.7193403727592,

2272.009269748305,

2089.131106168153,

1877.3489408572775,

1723.008422719009,

1579.400790281332,

1447.6151884287149,

1358.6836763880476],
                                                    'tCentroids':
[17.092105865478516,

26.234716415405273,

32.86576175689697,
```

```
40.274641036987305,
48.85936450958252,
57.13522124290466,
64.82174038887024,
71.61310529708862,
79.64548707008362,
84.1491162776947,
92.58800148963928,
99.59088397026062,
108.227543592453],
                                                    'tSamples':
[1.9398612976074219,
1.731004238128662,
1.8601210117340088,
1.8389818668365479,
1.8242743015289307,
1.8414041996002197,
2.121408700942993,
2.450037956237793,
1.9219880104064941,
1.8956823348999023,
1.713805913925171,
1.796037197113037,
1.8901896476745605],
                                                    'tTotal':
978.1837546825409},
                            'Number of partition16': {'CostInit':
[891555.5554243803,
22377.653392100456,
```

```
8629.214899629782,
5436.296716977931,
4260.795307674229,
3693.4135514749714,
2972.4869914393166,
2540.8690295473752,
2254.4309990245597,
1924.4365082423058,
1804.3187924517606,
1679.928830120031,
1566.4870302624386,
1457.8910337457664],
                                                    'tCentroids':
[12.663001775741577,
19.51159381866455,
28.317543029785156,
32.34527087211609,
38.656880378723145,
46.827125549316406,
56.31013226509094,
57.457207679748535,
68.31060099601746,
71.4232063293457,
82.15160608291626,
85.98417043685913,
95.7277398109436],
                                                    'tSamples':
```

```
[0.8386671543121338,
0.8171069622039795,
0.8569900989532471,
0.7518148422241211,
0.8357596397399902,
0.7499816417694092,
0.6464004516601562,
0.8005459308624268,
0.6953597068786621,
0.8766813278198242,
0.7099354267120361,
0.7232522964477539,
0.806626558303833],                                    'tTotal':
818.8865065574646},
                    'Number of partition2': {'CostInit':
[891555.5554243359,
34577.33079382057,
10674.196787091434,
6935.988986047896,
4799.194078541578,
3851.499789400862,
3203.844055363491,
2665.9284753856473,
2356.1538628193257,
2198.94533684475,
2002.930445091094,
1847.0223907129443,
```

```
1680.3068973200234,
1567.9206822287215],
                                                    'tCentroids':
[34.37669229507446,
56.87931299209595,
83.23446941375732,
107.59471559524536,
131.97507643699646,
157.73672437667847,
185.29766201972961,
206.84126925468445,
225.1283221244812,
249.35660481452942,
273.93789505958557,
305.92539501190186,
333.4616506099701],
                                                    'tSamples':
[1.120103359222412,
1.1706554889678955,
0.9483442306518555,
1.0482196807861328,
0.9928898811340332,
1.0721609592437744,
1.239293098449707,
1.1702768802642822,
1.203402042388916,
1.0181424617767334,
```

```
1.0759682655334473,
1.0064671039581299,
1.0744972229003906],
                                                    'tTotal':
2744.77747297287},
                           'Number of partition256': {'CostInit':
[2094763.126227179,
28528.229569572974,
10953.799029023643,
6333.44264582754,
4323.101610946494,
3621.57986242778,
2953.606632096169,
2693.534742322433,
2342.109309040679,
2101.498854579611,
1828.4941429116902,
1696.2358641854344,
1565.1146909201505,
1439.257019325019,
1359.0750908170012],
                                                     'tCentroids':
[17.440978050231934,
24.83854341506958,
33.44323658943176,
41.34664058685303,
47.97041964530945,
54.671419620513916,
59.76294279098511,
```

```
66.4251868724823,
71.84071159362793,
78.86712622642517,
88.27250504493713,
93.68002557754517,
99.27846956253052,
107.76170182228088],
                                                    'tSamples':
[3.4942941665649414,
3.372929334640503,
3.438781976699829,
3.442998170852661,
3.4150946140289307,
3.712352752685547,
4.043724536895752,
3.489170789718628,
3.6657192707061768,
3.4702072143554688,
2.9539926052093506,
3.4772772789001465,
3.2379939556121826,
3.5504159927368164],
                                                    'tTotal':
1065.12073802948},
                           'Number of partition32': {'CostInit':
[1601525.631138093,
70982.98463394282,
12854.129735744746,
```

```
7203.102566750421,
5039.777206241083,
3782.8190417370442,
3008.7965252298245,
2608.265009099853,
2319.4600205730303,
2106.297739951042,
1920.2566548406478,
1718.7798349960303,
1549.2635994896345,
1445.8953639030187,
1371.7720844645594],
                                                'tCentroids':
[16.321369409561157,
22.282538175582886,
31.617116451263428,
36.274001359939575,
46.511996269226074,
55.12869882583618,
56.06206774711609,
64.05211472511292,
69.1995062828064,
74.88440203666687,
82.83336639404297,
91.18759965896606,
97.58591675758362,
105.72941970825195],
```

'tSamples':
[0.9800570011138916,
0.8967080116271973,
0.8486120700836182,
0.882718563079834,
0.8753995895385742,
0.829594612121582,
0.8131394386291504,
0.949406623840332,
0.8689103126525879,
0.8476817607879639,
0.8226644992828369,
0.9605312347412109,
0.6979317665100098,
0.848773717880249],
'tTotal':
980.8802380561829},
'Number of partition4': {'CostInit':
[2223256.496517683,
58275.187236190715,
15117.338164315988,
7492.701184889861,
5491.0433568077115,
4141.556888892963,
3473.3583017584583,
2858.559858498532,
2487.5859838572815,
2162.1684975189987,

```
1919.9214738361075,
1765.4805436598567,
1616.20679054093,
1478.0450011882313,
1381.1972864673216],
                                                    'tCentroids':
[13.098650693893433,
27.931509256362915,
39.81945991516113,
53.59331917762756,
64.32080483436584,
93.69615769386292,
96.14066004753113,
108.745112657547,
122.67282629013062,
136.7574405670166,
150.79018878936768,
164.84979462623596,
179.89866662025452,
191.3278887271881],
                                                    'tSamples':
[0.7486796379089355,
0.8389637470245361,
0.7579433917999268,
0.6595571041107178,
0.6617116928100586,
0.7932686805725098,
0.7865848541259766,
```

```
0.8459608554840088,
0.7660346031188965,
0.7702269554138184,
0.8567297458648682,
0.711101770401001,
0.7883894443511963,
0.809903621673584],
                                                'tTotal':
1666.448011636734},
                           'Number of partition64': {'CostInit':
[1062852.2618626256,
56283.523745564045,
11259.154768067461,
6406.763363665761,
4258.798369865063,
3272.05091118598,
2885.6470398262263,
2349.998481071997,
2109.8747417259196,
1926.7860604089453,
1719.387112374389,
1598.2542540324723,
1443.2573331264082,
1351.768510414011],
                                                 'tCentroids':
[14.460254430770874,
23.23945426940918,
32.2976348400116,
39.66645526885986,
```

```
45.06172728538513,
50.39931869506836,
58.97108602523804,
67.47464942932129,
75.48974418640137,
78.8404586315155,
87.32318902015686,
95.3143196105957,
103.69666767120361],
                                                    'tSamples':
[1.3998363018035889,
2.333587884902954,
1.0383284091949463,
1.248624324798584,
1.284059762954712,
1.108712911605835,
1.2358744144439697,
1.1507201194763184,
1.1776132583618164,
1.3565161228179932,
1.3936538696289062,
1.330669641494751,
1.1656172275543213],
                                                    'tTotal':
907.9069976806641},
                           'Number of partition8': {'CostInit':
[891555.5554263061,
33927.19435053103,
```

```
10901.317924007788,
6135.548123954543,
4941.270404556077,
3649.814060380155,
2910.798577969736,
2545.3242604744655,
2226.1752166524716,
1975.2555548138662,
1796.6171079799242,
1681.1847633852371,
1534.9404615078295,
1445.2145687147777],
                                                'tCentroids':
[11.381458282470703,
20.180540800094604,
26.88866639137268,
31.74018430709839,
38.81391096115112,
48.69937872886658,
53.620816230773926,
58.794254779815674,
67.91973447799683,
75.13777709007263,
82.12062764167786,
87.50198769569397,
91.46046209335327],
                                                'tSamples':
[0.5604074001312256,
```

```
0.7893469333648682,

1.0149109363555908,

0.7487082481384277,

0.7158007621765137,

0.7343785762786865,

0.6166317462921143,

0.8378362655639648,

0.6786437034606934,

0.6768665313720703,

0.6516311168670654,

0.7696502208709717,

0.8361482620239258],
                                                    'tTotal':
812.2571785449982}},
 'totalLogParallelKmeans': {'Number of partition128': {'CostsKmeans':
[13256.068171158195,

13215.004392704688,

13199.35080389415,

13159.125690088671,

12995.72416396117,

12716.594692734348,

12702.29249101154,

12699.56508101606],
                                                      'tIterations':
[16.877744674682617,

16.08725643157959,

15.860199928283691,

15.853113889694214,
```

```
15.62473440170288,
16.735519886016846,
15.63037919998169,
15.941917419433594],
                                                    'tTotal':
128.61088252067566},
                              'Number of partition16': {'CostsKmeans':
[20135.80143345077,
20077.17761306065,
19954.853090147248,
19481.940068563963,
19367.454492017652,
19142.548174496766,
19083.416618553074,
19066.139849626437],
                                                   'tIterations':
[10.880249977111816,
11.704259395599365,
10.999428272247314,
11.290096044540405,
10.57009744644165,
9.916236162185669,
10.009653091430664,
10.275928258895874],
                                                    'tTotal':
85.64597034454346},
                              'Number of partition2': {'CostsKmeans':
[12473.205641309907,
12366.496092733309,
12331.738021866051,
```

```
12306.192485847434,
12291.58355890608,
12289.572628760156],
                                                    'tIterations':
[27.306171655654907,
24.897124767303467,
25.204771995544434,
25.39222741127014,
24.535747289657593,
25.696465015411377],
                                                    'tTotal':
153.03252053260803},
                               'Number of partition256': {'CostsKmeans':
[17171.126716995568,
17065.423822757337,
16970.043764839,
16931.461220474528,
16904.78187947643,
16880.295452888167,
16801.292200639666,
14895.434094154803,
12514.543307999575,
12494.443562591292,
12493.79589315728],
                                                     'tIterations':
[19.992228746414185,
20.28275966644287,
19.837072610855103,
20.77976942062378,
```

```
20.015557289123535,
19.699762105941772,
20.831369161605835,
19.40836238861084,
22.38698434829712,
19.50278639793396,
19.250699043273926],
                                                    'tTotal':
221.9873731136322},
                              'Number of partition32': {'CostsKmeans':
[14126.406907854667,
14033.887584557884,
13959.66936071383,
13834.257777667352,
13786.206960802598,
13705.417893725662,
13658.396304373387,
13605.01363918185,
13531.854086231762,
13353.599330872743,
12909.174485262909,
12788.849158969733,
12761.957357692343,
12757.83950672009],
                                                  'tIterations':
[12.061691761016846,
11.586308240890503,
11.568902015686035,
11.38906478881836,
```

```
11.316141366958618,
11.46327018737793,
11.998432636260986,
13.350913286209106,
11.131362438201904,
10.87834095954895,
17.41956663131714,
11.066790103912354,
11.18279504776001,
11.147577285766602],
                                                'tTotal':
167.56118655204773},
                                'Number of partition4': {'CostsKmeans':
[14154.331052221676,
14116.365971069381,
14112.281310880897],
                                                'tIterations':
[13.848233938217163,
12.48716115951538,
13.067476987838745],
                                                'tTotal':
39.40287971496582},
                                'Number of partition64': {'CostsKmeans':
[13002.761720190696,
12946.280623463417,
12858.600492427484,
12856.29847669335],
                                                'tIterations':
[13.89802074432373,
12.171608209609985,
12.45176911354065,
```

```
11.89674425125122],
                                                    'tTotal':
50.41814970970154},
                               'Number of partition8': {'CostsKmeans':
[14491.25799091136,

14427.780944459158,

14379.067008154318,

14340.9031047214,

14326.827877044685],
                                                    'tIterations':
[11.779228448867798,

9.766197919845581,

10.539317607879639,

10.567087888717651,

10.841463088989258],
                                                    'tTotal':
53.49331045150757}}}
```

```
# Random init example
pprint(totalLogR[0])
```

```
{'tDurationsNaive': {'Number of partition128': 246.70958375930786,
                     'Number of partition16': 86.43284273147583,
                     'Number of partition2': 235.50393104553223,
                     'Number of partition256': 251.0647897720337,
                     'Number of partition32': 69.03074526786804,
                     'Number of partition4': 113.30451202392578,
                     'Number of partition64': 163.41193318367004,
                     'Number of partition8': 97.91845202445984},
 'tPreOperationsNaive': {'Number of partition128': 14.513504028320312,
                         'Number of partition16': 9.8938889503479,
                         'Number of partition2': 15.74240779876709,
                         'Number of partition256': 20.451018571853638,
                         'Number of partition32': 9.526236057281494,
                         'Number of partition4': 10.653437614440918,
                         'Number of partition64': 10.467516660690308,
                         'Number of partition8': 10.049140214920044},
 'totalLogNaiveInit': {'Number of partition128': {'tTotal':
5.65146279335022},
                       'Number of partition16': {'tTotal':
1.9742786884307861},
                       'Number of partition2': {'tTotal':
```

```
3.5396792888641357},
                         'Number of partition256': {'tTotal':
9.955411911010742},
                         'Number of partition32': {'tTotal':
2.4714605808258057},
                         'Number of partition4': {'tTotal':
2.5076260566711426},
                         'Number of partition64': {'tTotal':
3.1421775817871094},
                         'Number of partition8': {'tTotal':
1.82527756690979}},
 'totalLogNaiveKmeans': {'Number of partition128': {'CostsKmeans':
[37588.1663885156,

26849.463446228598,

25295.796181492984,

24023.460059451285,

22624.616949744275,

22344.19222911269,

22278.733899910447,

22222.518277534593,

22182.3281202257,

22048.96907256953,

21939.825265074447,

21863.338224566698,

21551.51639223278,

21176.68219891586,

20696.410448243634,

20654.242605739484,

20646.015374805942],
                                                  'tIterations':
[14.191697835922241,

13.370175123214722,

13.394645690917969,
```

```
13.088303089141846,
12.958121538162231,
13.09688687324524,
14.682480573654175,
13.33041501045227,
13.132146835327148,
13.155274868011475,
13.28964638710022,
13.04693865776062,
13.236061811447144,
13.329109191894531,
13.32405710220337,
13.168695449829102,
12.749795913696289],
                                                'tTotal':
226.54448246955872},
                           'Number of partition16': {'CostsKmeans':
[166384.53462033137,
65895.60563070832,
65706.02173432914,
65383.54430620917,
52128.157728599246,
45756.44843686664,
43376.97766168072,
42947.12896710763,
42930.39892842657],
                                                'tIterations':
[9.774840593338013,
```

```
9.00724983215332,
7.818156719207764,
7.3633623123168945,
8.3872971534729,
8.063755750656128,
8.452018022537231,
7.990036725997925,
7.707749605178833],                                          'tTotal':
74.56449055671692},
                          'Number of partition2': {'CostsKmeans':
[42516.03346356709,
30489.899569214176,
28870.557007268813,
27552.014438794737,
23278.24060503552,
18984.3147713026,
18760.240816989506,
18693.13101953215,
18670.316806382605,
18655.67316314641],                                          'tIterations':
[23.22072458267212,
21.562345504760742,
21.96597170829773,
21.554337978363037,
21.63866353034973,
21.29712414741516,
21.602866649627686,
```

```
21.09733819961548,
21.08929681777954,
21.192963123321533],
                                                'tTotal':
216.22165393829346},
                           'Number of partition256': {'CostsKmeans':
[155888.09765600174,
63416.33422365386,
44822.84167562508,
35598.031912505205,
33770.97849253809,
33674.87549087495,
33436.71391261008,
30595.98345517677,
27523.668063972655,
27327.893091206282,
27266.89180942977,
27251.758602438957],
                                                 'tIterations':
[19.29913306236267,
18.30908989906311,
18.258602142333984,
17.94507622718811,
19.428080081939697,
18.150232553482056,
18.10883116722107,
18.008137941360474,
18.24814772605896,
```

```
17.732861518859863,
18.98153281211853,
18.188443183898926],
                                                'tTotal':
220.6581907272339},
                          'Number of partition32': {'CostsKmeans':
[49520.386358362106,
30527.58542146165,
23020.67136604504,
21911.733491508534,
21754.20134667065,
21737.064097228238],
                                                'tIterations':
[10.73610258102417,
9.135873317718506,
9.866609334945679,
8.899936199188232,
8.921936511993408,
9.472395420074463],
                                                'tTotal':
57.03286814689636},
                          'Number of partition4': {'CostsKmeans':
[155320.87780776434,
69808.04708340974,
54725.117012482755,
43558.38585393489,
42153.973898581025,
39892.612618343206,
36677.7784101257,
36601.76233647565,
36533.77835639385,
```

```
36487.0518173548,
36465.68111002703],
                                                'tIterations':
[11.667833089828491,
9.159595251083374,
8.723095893859863,
8.762112855911255,
8.907129526138306,
9.004110336303711,
8.768619537353516,
8.80795955657959,
8.801142930984497,
8.743201732635498,
8.798418521881104],
                                                'tTotal':
100.14324736595154},
                           'Number of partition64': {'CostsKmeans':
[139644.47538727487,
36535.13405139362,
33452.63488532336,
32223.7184884443,
31195.288436264193,
31096.484073064148,
31059.985512694107,
30952.398379187733,
30697.26167899626,
30519.036479392384,
30363.737355164907,
```

```
30089.8334407707,
29614.347467224787,
29549.847534218025,
29539.846943083],
                                                'tIterations':
[11.088544845581055,
9.938262939453125,
10.427467346191406,
9.227502346038818,
10.919835567474365,
9.76926326751709,
9.618249416351318,
9.622366666793823,
9.751763343811035,
9.964595556259155,
9.911336183547974,
10.541150331497192,
9.954270601272583,
9.533994913101196,
9.533462762832642],
                                                'tTotal':
149.80209684371948},
                        'Number of partition8': {'CostsKmeans':
[82567.0250963437,
33104.67062765449,
31797.19712773952,
30862.86104723941,
30695.516272739696,
30633.602145267392,
```

```
30564.992296609005,
30482.895593478763,
30440.947605577367,
30416.90273136094],
                                                'tIterations':
[9.358434200286865,
8.255417346954346,
7.617798089981079,
7.953901290893555,
8.634777307510376,
9.0293550491333,
8.520485401153564,
8.888786792755127,
8.818779230117798,
8.965948820114136],
                                                'tTotal':
86.04372000694275}}}
```

Time efficiency

Now, we unpack the data to compare the mean total execution time. We group our data by multiple runs, the number of partitions, and the three steps of each run: pre-processing, initialization (either parallel or random), and Lloyd iterations.

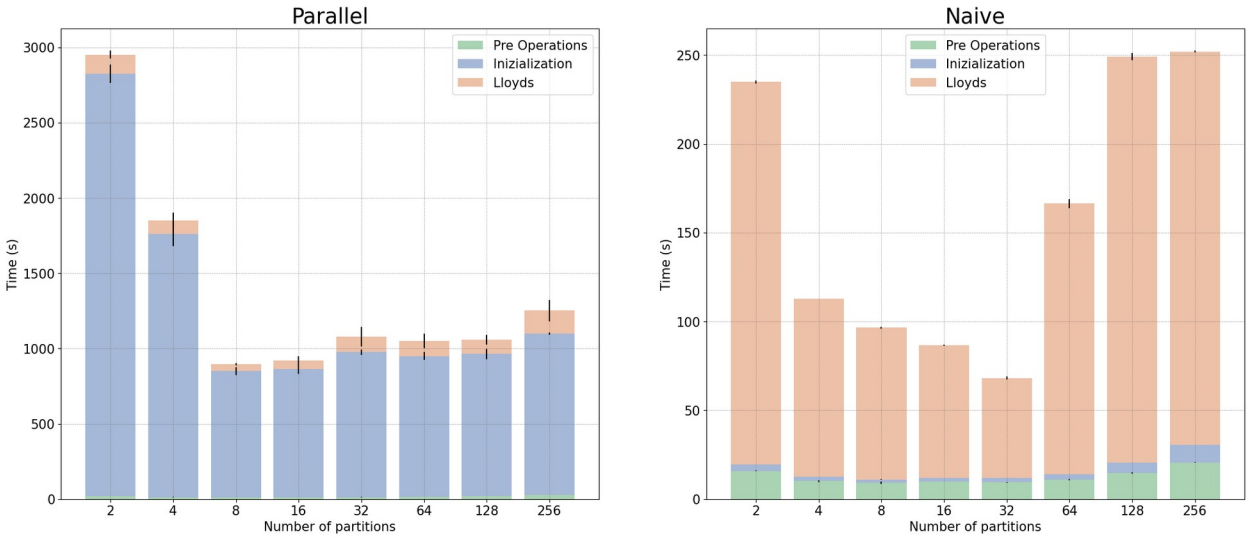
```
# Metadata about the run
int_partitions = [2, 4, 8, 16, 32, 64, 128, 256]
name_partitions = ["Number of partition" + str(i) for i in
int_partitions]

# Unpack the data for the parallel
n_files = len(totalLogP) # Number of independent runs over all the
partitions
n_partitions = len(name_partitions) # Number of partitions used for
each run
tTotalP = np.zeros((n_files, n_partitions, 3)) # Numpy array to fill

for i in range(n_files):
```

```
    log = totalLogP[i]
    for p in range(n_partitions):
        partition = name_partitions[p]
        tTotalP[i, p, 0] = log["tPreOperationsParallel"][partition]
        tTotalP[i, p, 1] = log["totalLogParallelInit"][partition]
["tTotal"]
        tTotalP[i, p, 2] = log["totalLogParallelKmeans"][partition]
["tTotal"]

# Unpack the data for the naive
n_files = len(totalLogR)
tTotalR = np.zeros((n_files, n_partitions,3))

for i in range(n_files):
    log = totalLogR[i]
    for p in range(n_partitions):
        partition = name_partitions[p]
        tTotalR[i, p, 0] = log["tPreOperationsNaive"][partition]
        tTotalR[i, p, 1] = log["totalLogNaiveInit"][partition]
["tTotal"]
        tTotalR[i, p, 2] = log["totalLogNaiveKmeans"][partition]
["tTotal"]

# Compute the mean time and standard deviation of each steps (parallel
init)
meansP = np.mean(tTotalP, axis=0)
stdDevsP = np.std(tTotalP, axis=0)

# Compute the mean time and standard deviation of each steps (random
init)
meansN = np.mean(tTotalR, axis=0)
stdDevsN = np.std(tTotalR, axis=0)
```

Comparison of the total execution time for the full pipeline of the algorithm between parallel and naive initialization methods. We divided the total time into three stages to a more detailed comparison.

```
# Barplot of the stacked times for the parallel
fig, ax = plt.subplots(1, 2, figsize=(25, 10), sharey=False)
colors = sns.color_palette("deep", n_colors=3)
title_fontsize = 25
label_fontsize = 15

ax[0].bar(np.arange(n_partitions), meansP[:, 0], bottom = 0,
yerr=stdDevsP[:, 0], alpha=0.5, label="Pre Operations",
color=colors[2])
ax[0].bar(np.arange(n_partitions), meansP[:, 1], bottom = meansP[:,
0], yerr=stdDevsP[:, 1], alpha=0.5, label="Inizialization",
color=colors[0])
```

```
ax[0].bar(np.arange(n_partitions), meansP[:, 2], bottom = (meansP[:,
0]+meansP[:, 1]), yerr=stdDevsP[:, 2], alpha=0.5, label="Lloyds",
color=colors[1])
ax[0].set_xticks(np.arange(n_partitions))
ax[0].set_xticklabels(int_partitions)
ax[0].tick_params(axis='x', labelsize=label_fontsize)
ax[0].tick_params(axis='y', labelsize=label_fontsize)
ax[0].set_xlabel("Number of partitions", fontsize=label_fontsize)
ax[0].set_ylabel("Time (s)", fontsize=label_fontsize)
ax[0].set_title("Parallel", fontsize=title_fontsize)
ax[0].legend(fontsize=label_fontsize)
ax[0].grid(True, linestyle='--', linewidth=0.5, alpha=0.7,
color='gray')


ax[1].bar(np.arange(n_partitions), meansN[:, 0], bottom = 0,
yerr=stdDevsN[:, 0], alpha=0.5, label="Pre Operations",
color=colors[2])
ax[1].bar(np.arange(n_partitions), meansN[:, 1], bottom = meansN[:,
0], yerr=stdDevsN[:, 1], alpha=0.5, label="Inizialization",
color=colors[0])
ax[1].bar(np.arange(n_partitions), meansN[:, 2], bottom = (meansN[:,
0]+meansN[:, 1]), yerr=stdDevsN[:, 2], alpha=0.5, label="Lloyds",
color=colors[1])
ax[1].set_xticks(np.arange(n_partitions))
ax[1].set_xticklabels(int_partitions)
ax[1].tick_params(axis='x', labelsize=label_fontsize)
ax[1].tick_params(axis='y', labelsize=label_fontsize)
ax[1].set_xlabel("Number of partitions", fontsize=label_fontsize)
ax[1].set_ylabel("Time (s)", fontsize=label_fontsize)
ax[1].set_title("Naive", fontsize=title_fontsize)
ax[1].legend(fontsize=label_fontsize)
ax[1].grid(True, linestyle='--', linewidth=0.5, alpha=0.7,
color='gray')
```



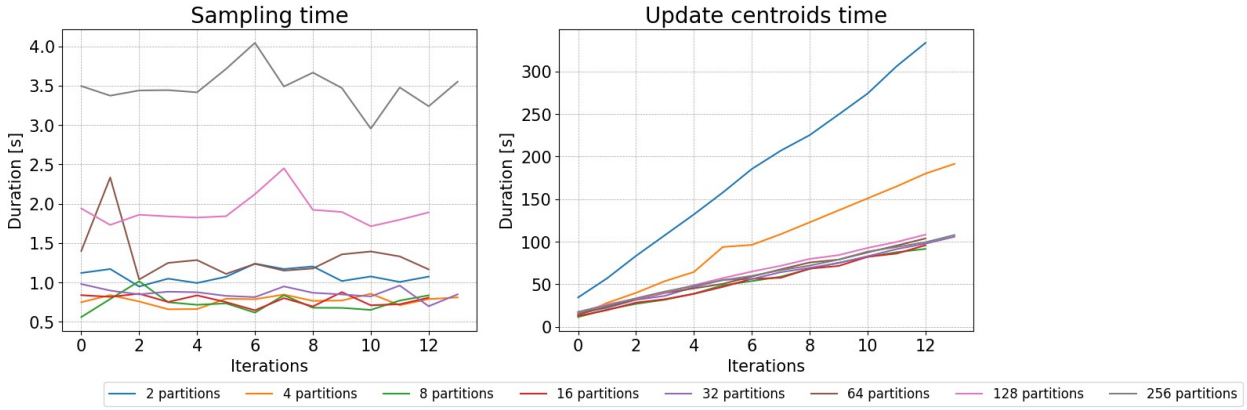
Each subplot of the above figure shows a bar plot of the algorithm's performance for the three steps, divided by the number of partitions used. Please note that the bars are stacked, so the total time taken to complete a run is represented by the total height of each bar.

Now we visually compare the time spent based on the number of iterations and partitions used for sampling and updating centroids, specifically focusing on parallel initialization.

```
# Unpack tSamples and tCentroids for each number of partitions
PIDA_tSamples=parallelInitDataArray=[[sample['totalLogParallelInit']
[n_partition]['tSamples'] for n_partition in name_partitions] for
sample in totalLogP]
PIDA_tCentroids=parallelInitDataArray=[[sample['totalLogParallelInit']
[n_partition]['tCentroids'] for n_partition in name_partitions] for
sample in totalLogP]

# Choose a sample (independent runs of the algorithm for a given set
of possible partitions)
sample=0

# Plot the results
label_names = [str(int_partitions[i])+" partitions" for i in
range(len(int_partitions))]
title_fontsize = 20
label_fontsize = 15

fig, (ax0, ax1)=plt.subplots(ncols=2, nrows=1, figsize=(15, 5))
# tSamples
for i in range(len(name_partitions)):
    ax0.plot(PIDA_tSamples[sample][i], label=label_names[i])
ax0.set_xlabel('Iterations', fontsize=label_fontsize)
ax0.set_ylabel('Duration [s]', fontsize=label_fontsize)
ax0.set_title('Sampling time', fontsize=title_fontsize)
ax0.tick_params(axis='x', labelsize=label_fontsize)
ax0.tick_params(axis='y', labelsize=label_fontsize)
```

```
ax0.xaxis.set_major_locator(MaxNLocator(integer=True))
ax0.grid(True, linestyle='--', linewidth=0.5, alpha=0.7, color='gray')


# tCentroids
for i in range(len(name_partitions)):
    ax1.plot(PIDA_tCentroids[sample][i], label=label_names[i])
ax1.set_xlabel('Iterations', fontsize=label_fontsize)
ax1.set_ylabel('Duration [s]', fontsize=label_fontsize)
ax1.set_title('Update centroids time', fontsize=title_fontsize)
ax1.tick_params(axis='x', labelsize=label_fontsize)
ax1.tick_params(axis='y', labelsize=label_fontsize)
ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
ax1.grid(True, linestyle='--', linewidth=0.5, alpha=0.7, color='gray')
ax1.legend(loc=(-1.1, -0.25), fontsize=12, ncols=len(int_partitions))
```

```
<matplotlib.legend.Legend at 0x21058bf3f80>
```



KMeans cost

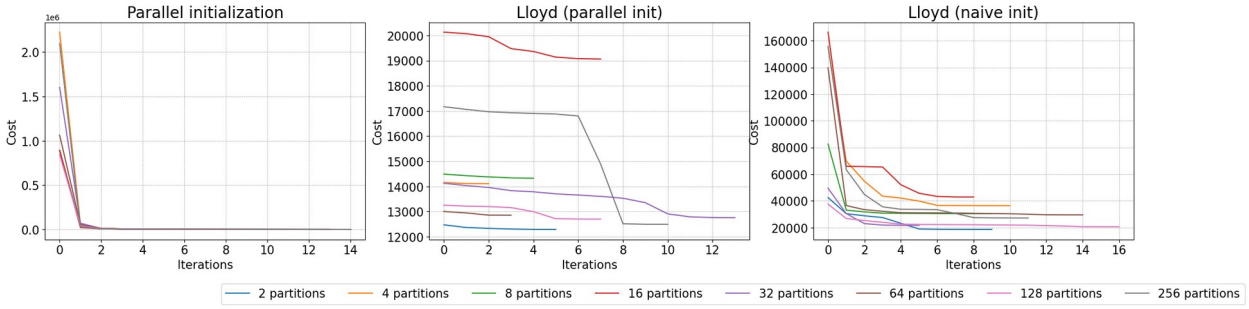
We also analyze the dependency of the cost function on the number of iterations and partitions for parallel, Lloyd's, and naive initializations. The cost function is defined as sum of the squared distances between each points and the nearest centroid. In formula,
$\text{Cost} = \sum_{x} min_{i=1,\dots,k} || x - c_i ||^2$ where c_i is an element of the set of centroids and x a point in space.

```
# Figure to compare the cost during different phases of the algorithm
fig, (axInit, ax0, ax1)=plt.subplots(ncols=3, nrows=1, sharey=False,
figsize=(25, 5))
label_names = [str(int_partitions[i])+" partitions" for i in
range(len(int_partitions))]
title_fontsize = 20
label_fontsize = 15

# Choose a sample
```

```
sample=0

# Unpack the data about the cost during the parallel initialization
initFold=totalLogP
NESTEDLIST=[[sample['totalLogParallelInit'][n_partition]['CostInit']
for n_partition in name_partitions] for sample in initFold]

# Plot the data
for i in range(len(name_partitions)):
    axInit.plot(NESTEDLIST[sample][i], label=label_names[i])
axInit.set_xlabel('Iterations', fontsize=label_fontsize)
axInit.set_ylabel('Cost', fontsize=label_fontsize)
axInit.set_title('Parallel initialization', fontsize=title_fontsize)
axInit.tick_params(axis='x', labelsize=label_fontsize)
axInit.tick_params(axis='y', labelsize=label_fontsize)
axInit.xaxis.set_major_locator(MaxNLocator(integer=True))
axInit.grid(True, linestyle='--', linewidth=0.5, alpha=0.7,
color='gray')

# Unpack the data about the cost during the Lloys iterations after
parallel initialization
initFold=totalLogP
NESTEDLIST=[[sample['totalLogParallelKmeans'][n_partition]
['CostsKmeans'] for n_partition in name_partitions] for sample in
initFold]

# Plot the data
for i in range(len(name_partitions)):
    ax0.plot(NESTEDLIST[sample][i], label=label_names[i])
ax0.set_xlabel('Iterations', fontsize=label_fontsize)
ax0.set_ylabel('Cost', fontsize=label_fontsize)
ax0.set_title('Lloyd (parallel init)', fontsize=title_fontsize)
ax0.tick_params(axis='x', labelsize=label_fontsize)
ax0.tick_params(axis='y', labelsize=label_fontsize)
ax0.xaxis.set_major_locator(MaxNLocator(integer=True))
ax0.grid(True, linestyle='--', linewidth=0.5, alpha=0.7, color='gray')

# Unpack the data about the cost during the Lloyds iterations after
random initialization
initFold=totalLogR
NESTEDLIST=[[sample['totalLogNaiveKmeans'][n_partition]['CostsKmeans']
for n_partition in name_partitions] for sample in initFold]

# Plot the data
for i in range(len(name_partitions)):
    ax1.plot(NESTEDLIST[sample][i], label=label_names[i])
ax1.set_xlabel('Iterations', fontsize=label_fontsize)
ax1.set_ylabel('Cost', fontsize=label_fontsize)
ax1.set_title('Lloyd (naive init)', fontsize=title_fontsize)
ax1.tick_params(axis='x', labelsize=label_fontsize)
```

```
ax1.tick_params(axis='y', labelsize=label_fontsize)
ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
ax1.grid(True, linestyle='--', linewidth=0.5, alpha=0.7, color='gray')
ax1.legend(loc=(-1.85, -0.3), fontsize=15, ncols=len(int_partitions))
```

```
<matplotlib.legend.Legend at 0x210596d5cd0>
```




```
# Stop the context
sc.stop()
```