PARALLEL IMPLEMENTATION
OF MACHINE LEARNING
ALGORITHMS USING
PYCOMPSS

-MASTER THESIS-

Mentor
Dusan Jakovetic, PhD

Novi Sad, September 2018
Abstract

Due to large amounts of multidimensional data, conventional machine learning algorithms and tools are often impractical to use in many industrial applications. To address this problem, algorithms are often either simplified or run in parallel over a multiple-machines computational infrastructure, e.g., a computer cluster. Barcelona Supercomputing Center (BSC) created a library, dubbed PyCompss, with an idea to simplify the process of parallel programming and make this task easier and more available for users.

In this work, we present a parallel implementation of several clustering and classification algorithms using PyCompss. For each algorithm, we describe a methodology of how the algorithm is partitioned into smaller functions that can run in parallel at runtime. We illustrate performance of the algorithms on several real public datasets.
Contents

1 Introduction 4

2 PyCOMPSs Programming Model 5
  2.1 Example ............................................. 5
  2.2 Syntax ............................................. 6

3 K Nearest Neighbors 10
  3.1 KNN Classification .................................. 10
    3.1.1 PyCompss Implementation ...................... 11
  3.2 KNN Regression ..................................... 15
    3.2.1 PyCompss Implementation ...................... 16

4 K Means 20
  4.1 K Means Algorithm .................................. 20
  4.2 PyCompss Implementation ............................ 21

5 Decision Tree 27
  5.1 Decision Tree Algorithm .......................... 27
  5.2 PyCompss Implementation ............................ 29

6 Random Forest 36
  6.1 Random Forest Algorithm .......................... 36
  6.2 PyCompss Implementation ............................ 37

7 ADMM LASSO 42
  7.1 ADMM ............................................ 42
  7.2 LASSO ........................................... 44
  7.3 PyCompss Implementation ............................ 45

8 Simulation Experiments 51
  8.1 Simulation Setup and Benchmarks ................... 51
  8.2 Datasets ........................................... 51
  8.3 Metrics ............................................ 53
  8.4 KNN Classifier ..................................... 54
  8.5 KNN Regression .................................... 55
  8.6 K Means ............................................ 56
  8.7 Decision Tree ...................................... 57
  8.8 Random Forest ..................................... 59
  8.9 LASSO ............................................ 60
9 Conclusion 68
10 Acknowledgments 69
11 List of Figures 70
12 List of Tables 71
13 List of Listings 72
1 Introduction

There is no subject so old that something new cannot be said about it

F. M. Dostoevsky

Machine Learning is a field that quickly spreads in all industries and even everyday life. Its importance and possibilities for use are constantly growing, but there are some limitations. In large industries amounts of data acquired are so vast that conventional machine learning algorithms often turn out to be impractical for real time computations. There are two common ways to address this problem; either 1.) algorithms are simplified or 2.) they are run in parallel over a computer cluster. In this work, we will address the second idea, implementing machine learning algorithms in parallel.

Barcelona Supercomputing Center (BSC) created a programming model dubbed COMPSs and its Python language variant called PyCOMPSs [12], with an idea to simplify the process of parallel programming and make it easier and more available to users.

The thesis consists of 9 chapters organized as follows. Chapter PyCOMPSs Programming Model explains in more details the mentioned library and goes through easy examples and case of uses. Chapters 3-7 describe specific Machine Learning (ML) algorithms, each covering first the theoretical background and then implementation in PyCOMPSs. The algorithms covered in chapters 3-7 are: K Nearest Neighbors, K Means, Decision Tree, Random Forest and ADMM LASSO, respectively. Chapter Simulation Experiments covers some initial testings of each of algorithms introduced. Finally, chapter 9 concludes the work, discussing covered materials and proposing ideas and directions for future work and improvements.
2 PyCOMPSs Programming Model

Python programming language is one of the most popular languages in the field of data science, and its popularity still grows. It contains a lot of numerical, statistical and machine learning libraries such as numpy [19], scipy [20], sklearn [18] etc. Besides the ease of use and a wide scope of tools provided by these libraries another reason for general acceptance of Python is the fact that it is a free source language, and that makes it affordable and available to everyone.

There are obviously a lot of advantages in using Python, but in order to use it for large scale applications, programming language needs to be parallelizable. Its default implementation, CPython [21], cannot run multiple threads [6], and most of the other solutions are either immensely complicated or not efficient enough.

In order to address this problem, Barcelona Supercomputing Center (BSC) [16] developed PyCOMPSs, a parallel programming framework for Python applications [6]. A PyCOMPSs application is a plain sequential Python script. In the model, the user is mainly responsible for (i) identifying the functions to be executed as asynchronous parallel tasks and (ii) annotating them with a standard Python decorator. A runtime system is in charge of exploiting the inherent concurrency of the script, automatically detecting and enforcing the data dependencies between tasks and spawning these tasks to the available resources, which can be nodes in a cluster, cloud or grid [6].

2.1 Example

When programming with PyCOMPSs, a user’s first step is to define the tasks of the application. Application can be composed of the number of functions, and some of them may be computationally very demanding. Such functions are good candidates to be decorated as tasks and then executed in parallel. Figure 1 presents a simplified code for the KMeans algorithm (explained in chapter 4) in PyCOMPSs. Figure 1(a) shows definitions of tasks and functions, and Figure 1(b) shows the main program of the algorithm.

The user defines functions to be run in parallel simply by placing @task decorator [9] before a function definition. In Figure 1(a) we have 4 func-
tions defined as tasks. In functions \texttt{closest} (line 5) and \texttt{average} (line 9) we specified the output type (\texttt{dict} and \texttt{np.array} respectively) inside of \texttt{@task} decorator. In case of function \texttt{merge} \texttt{dict} (line 12) (and analogously \texttt{merge} \texttt{matrix} (line 17)) we specified the type of file as \texttt{INOUT}, meaning that the \texttt{dict1} will be modified inside the task.

When we have finished with function definitions, the code can be executed with PyCOMPSs. When running the script, PyCOMPSs creates an asynchronous task for each invocation to a decorated function, adding these tasks to a dependency graph \cite{6}. For example, if one task writes data that the subsequent task reads, then there is dependency between these two tasks. PyCOMPSs finds these dependencies using the information about the direction specified inside of task definition (e.g. \texttt{INOUT}). Data dependencies are then enforced by PyCOMPSs in order for the application to be executed correctly. Figure 2 shows task-dependency graph built from the first part of example in Figure 1(b). In the first for loop we have function \texttt{closest} executed three times, and they all depend on the matrix \texttt{centers} created above, because they receive it as an input parameter. The loop also generates three \texttt{merge} \texttt{dict} tasks, each depending on the corresponding \texttt{closest} task; in fact, each \texttt{merge} \texttt{dict} depends on the result produced by previous iteration, and they are arranged in a chainlike structure of tasks.

After exiting the first loop, we have obtained a dictionary of points, that we will use to extract matrices for each centroid. But before doing that, we need first to synchronize for the last value of \texttt{dictionary}. In order to do this, PyCOMPSs provides an API function, \texttt{compss.wait.on} \cite{6} (line 8), which steams the main control flow until the last value of \texttt{dictionary} is obtained, i.e. call to \texttt{compss.wait.on} will wait for the last \texttt{merge} \texttt{dict} task to be finished before returning the final result of \texttt{dictionary}.

### 2.2 Syntax

#### Task Definition

In PyCOMPSs, the user can define as a task functions, instance methods and class methods \cite{6}. Task is defined using \texttt{@task} decorator containing information about the parameters of function and about the task itself. The parameters supported by the decorator are shown in Table 1.

If the parameter is of the primitive type (integer, long, float, double) or string, there is no need to include parameter in the \texttt{@task} decorator. Also, there is no need for specifying the read-only object parameters; direction
from pycompss.api.task import task
from pycompss.api.parameter import *
import numpy as np

@task(returns = dict)
def closest(matrix, centers):
    dictionary = {}
    ...
    # compute the closest center for each row of matrix
    return dictionary

@task(returns = np.array)
def average(matrix):
    center = ...
    # compute columnwise average of a matrix
    return center

@task(dict1 = INOUT)
def merge_dict(dict1, dict2):
    ...
    # merge dict1 and dict2
    dict1.update(dict2)

def extract_matrix(dictionary):
    matrix_list = []
    ...
    # extract the points with the same value of center
    # and put them in one matrix
    return matrix_list

@task(matrix1 = INOUT)
def merge_matrix(matrix1, matrix2):
    ...
    # add matrix2 on to matrix1
    matrix1.update(matrix2)

def split_matrix(matrix):
    ...
    # split matrix into tree smaller matrices
    return matrix_list

Figure 1a: Task definitions in K Means algorithm example
from pycompss.api.api import compss_wait_on
import numpy as np

centers, matrix1, matrix2, matrix3 = ...  
# initialize centers, matrix1,...,matrix3

dictionary = {}
for m in [matrix1, matrix2, matrix3]:
    d = closest(m, centers)
    mergeDict(dictionary, d)

dictionary = compss_wait_on(dictionary)
matrices = extract_matrix(dictionary)

for i in range(len(matrices)):
    centers[i] = average(matrices[i])

centers = compss_wait_on(centers)

matrix = np.array()
for m in matrices:
    merge_matrix(matrix, m)

matrix = compss_wait_on(matrix)

[matrix1, matrix2, matrix3] = split_matrix(matrix)

Figure 1b: Main program

Figure 2: Dependency Graph
Table 1: Arguments for task decorator

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
</table>
| Formal parameter name | -INOUT: read-write object  
|                 | -OUT: write-only object                                              |
|                 | -FILE: read-only file                                                |
|                 | -FILE_INOUT: read-write file                                         |
|                 | -FILE_OUT: write-only file                                          |
| returns         | int (for integer and boolean), long, float, str, dict, list, tuple, user-defined classes |
| isModifier      | True (default) or False                                             |
| priority        | True or False (default)                                             |

automatically defaults to IN.

Besides that, there are three other arguments presented in Table 1. If the function or method returns a value, the type of value must be specified using the returns argument. Second, if a task corresponds to an instance method, it is by default assumed that the task modifies the object on which the method is invoked. To switch it off, the user needs to set the isModifier argument to False. The last argument is used to mark a task as high-priority task setting priority to True. If the task is free of dependencies, and this argument is set True, it will be executed before any of the regular tasks.

Main Program

The main program is a part of the Python script that contains calls to tasks. All the actual calculation and transformation performed on data is done in the main program. But in order to access processed data, it needs to be synchronized, so that the main program can work with the correct version of data. There are two API functions that may need to be invoked for this purpose: compss.wait.on or compss.open.

compss.wait.on(obj, to_write = True) synchronizes for the last version of object obj and returns the synchronized object. Parameter to_write indicates whether the returned object will be modified. compss.open(file_name, mode = ‘r’) synchronizes for the last version of file file_name and returns the file descriptor for that synchronized file. Parameter mode represents the mode in which the file will be opened [6].
3 K Nearest Neighbors

K Nearest Neighbors (KNN) is a common machine learning algorithm used both for clustering and for regression. It is widely used because it is easily understood and implemented. For this algorithm, it is necessary to have labels or values so it belongs to the category of supervised learning [1]. Let us start with classification.

3.1 KNN Classification

"Show me who your friends are and I’ll tell you who you are?" [13]

One may or may not agree with this saying, but in practical machine learning this approach works very well. The main idea of KNN classification is to assign the same label to a datapoint as those of its nearest neighbors.

The algorithm is very simple. Only parameter K is needed and it stands for the number of neighboring points we want to encounter when classifying a new point. In training set, we have m labeled points in n dimensions. KNN is a so called lazy algorithm [1], meaning that it does not make any calculations in training phase. Data is simply stored, and calculations are done only when classifying new points. This also means that training phase is fast, but classification is slow.

To classify a new point \( p \), we first calculate distances \( d_1, d_2, \ldots, d_m \) from this point to all points in the training set (usual metric for this is Euclidian distance). After that, we take labels of K closest points \( l_{i1}, l_{i2}, \ldots, l_{ik} \), and estimate label \( l \) of our point by voting, i.e. decide which class has the most instances in these K points, and assign our point to that class.

To classify new points in this way, we need \( m \times n \) calculations for each point, which is computationally slow process when dealing with large numbers of points and dimensions.

Choice of K is adaptable and usually depends on the application. In practice \( K = 1 \) is often used, but in this way, classification is very much influenced by noise. On the other hand, with K considerably large, too many points make a contribution, and decision boundary is in a way blurry. One of the good practices when dealing with binary classification [1] is to pick an odd K, in order to prevent ties in the voting part. Best way for deciding on value of K is experimenting using K-fold crossvalidation [4],
Figure 3: This figure shows how different choices of \( K \) lead to different predictions. Green circle in the middle is the point whose class we need to estimate. Choosing \( K = 3 \) we will assign our point to the class of red triangulars, but for \( K = 5 \) the majority of neighbors are blue rectangulars so the outcome is different.

and picking the one with best score obtained. Figure 3 shows how different choices of \( K \) lead to different decisions.

3.1.1 PyCompss Implementation

Implementation follows, in detail, the theory explained in this chapter, but the momentous idea here is parallelization. We need to break an algorithm into composite functions and decide which of them should be distributed and in what way. Considering the nature of KNN, we don’t have to work on training phase, but parallelization is included in prediction stage. The idea is to split original data set into chunks and observe each of them as stand alone problem, performing mini KNN in parallel on each and then concatenating the results. Figure 4 presents the idea of an algorithm, and a detailed explanation is bellow.
Implementation

```python
import numpy as np
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on
from scipy import stats

class KNNClassifier(object):
    """
    K-nearest neighbors classifier
    """

def __init__(self, K=3, num=10):
    """
    #param num: on how many blocks to partition our data
    self.num = num
    self.K = K
    self.train_data = None
    self.labels = None

def fit(self, data, labels):
    if len(data) < self.K:
        raise ValueError('The number of neighbors precedes the number of datapoints')
    if len(data) != len(labels):
        raise ValueError('The length of data must be the same of the length of labels')
    self.train_data = data
    self.labels = labels
    return self

def predict(self, points):
    K = self.K
    train_data = self.train_data
    labels = self.labels
    num = self.num
    size = int(len(train_data)/num)
    X = []
    L = []
    for i in range(num-1):
        X.append(train_data[i*size:(i+1)*size])
        L.append(labels[i*size:(i+1)*size])
    X.append(train_data[(num-1)*size:])
    L.append(labels[(num-1)*size:])
    partial_res = [partial_knn(points, X[i],
```
L[i], K) for i in range(num)]
dictionary = mergeReduce(reduceNeighbors,
                        partial_res)
dictionary = compss_wait_on(dictionary)

pred_labels = []
for k,v in dictionary.items():
    pred_labels.append(find_mode(v))
return pred_labels

def fit_predict(self, data, labels):
    self.fit(data, labels)
    return self.predict(data)

def mergeReduce(function, data):
    """ Apply function cumulatively to the items of 
data, from left to right in binary tree structure, 
so as to reduce the data to a single value. 
: param function: function to apply to reduce 
data 
: param data: List of items to be reduced 
: return: result of reduce the data to a single 
value """
    from collections import deque
    q = deque(range(len(data)))
    while len(q):
        x = q.popleft()
        if len(q):
            y = q.popleft()
            data[x] = function(data[x], data[y])
            q.append(x)
        else:
            return data[x]

@task(returns = dict)
def partial_knn(points, XP, labels, K):
    """ Performs knn on partial training data 
param points: Points to be classified 
param XP: Partial training data points 
param labels: labels of points from XP 
param K: numer of neighbors 
"""
    dic = {}
for x in enumerate(points):
    neighbors = sorted([[labels[i[0]],
                         np.linalg.norm(x[1] - i[1])] for i in enumerate(XP)],
                       key=lambda t: t[1])[:K]
    dic[x[0]] = neighbors
return dic

@task(returns=dict, priority=True)
def reduceNeighbors(a, b):
    
    Reduce method to merge the result of two partial_knn methods
    
    """
    K = len(a[0])
    for key in b:
        a[key] = a[key] + b[key]
        a[key] = sorted(a[key], key=lambda t: t[1])[:K]
    return a

def find_mode(list_a):
    """
    Takes list of point’s neighbors, ex. : [[0, 0.123],[0, 0.155],[1, 0.222]]
    and finds the most common label
    """
    from scipy import stats
    return stats.mode([i[0] for i in list_a])[0][0]

Listing 1: PyCOMPSs Implementation of KNN Classifier

As shown in Listing 1, KNNclassifier class consists of three methods (omitting __init__()): .fit(), .predict() and .fit_predict(). In .fit() method, besides checking for feasibility in dimensions, we only store data points and labels as a training set used later for determining the affiliations of new points.

The .predict() method is much more complex and all non-method functions bellow are involved in it. num parameter addresses the number of chunks we want to split data on when distributing computations. Data is split in chunks and stored in list X, and from there we feed it to function partial_knn. This function is a task and it will be executed in parallel, working on each chunk of data at the same time. It takes as input new
points and one of the data chunks, and performs reduced KNN over it, returning a dictionary containing for each point distances to closest K training points and corresponding labels. When all partial KNNs are calculated, output is fed to `mergeReduce` function that merges all obtained dictionaries and, using `reduceNeighbors` task, reduces them into a single dictionary containing the final label of each point.

Method `.fit_predict()` performs `.fit()` and `.predict()` methods in a row on the same data, returning labels that are not necessarily same as original labels. (.fit_predict(data) is equivalent to calling .fit(data).predict(data))

### 3.2 KNN Regression

The regression with KNN is in a major part practically the same as classification. The whole process is exactly the same up to voting, except that in case of classifications each point has a label, which is a categorical value, and in regression each point has a numerical value.

When all the distances $d_1, d_2, ..., d_m$ are measured, and closest K points are found, the value $v$ of new point $p$ is estimated not by voting but by averaging. There are two ways for doing this. **Weighted** or **unweighted** average. In the unweighted case, new point is simply assigned the sum of values of K
nearest neighbors divided by K,
\[ v = \frac{(v_{i1} + v_{i2} + \ldots + v_{ik})}{K} \]

In the weighted case, we differently treat points on different distance. The idea is that closer neighbors should carry more information so are assigned greater weight. If the coordinates of new point are exactly the same as any point in training set, we will simply assign that same value for new point \( p \). If not, we will invert the distances of KNN and scale them to sum 1.

\[ w_i = \frac{1}{d_i} \]

\[ w_i = \frac{w_i}{\sum_{n=1}^{k} w_n} \]

Then, each of these rescaled distances represents the weight of the corresponding point, and value of new point is the sum of values of KNN each multiplied with the corresponding weight.

\[ v = v_{i1} * w_1 + v_{i2} * w_2 + \ldots + v_{ik} * w_k \]

### 3.2.1 PyCompss Implementation

Implementation of KNN regression is basically completely the same as that of KNN classification in 3.1.1; with the only difference in the final stage where now, instead of looking for the mode of neighboring labels, we calculate the weighted average of values using function `regression`. Note that both in classification and regression two successive executions on the same data will produce exactly the same results. The results will not differ when we run algorithm on different number of processors or even execute it sequentially.

**Implementation**

```python
import numpy as np
from pycompss.api.task import task
from pycompss.api.api import compss
from scipy import stats

class KNNRegressor(object):
    """K-nearest neighbors regressor"""
```
def __init__(self, K=3, num = 10):
    #param num: on how many blocks to partition our data
    self.num = num
    self.K = K
    self.train_data = None
    self.labels = None

def fit(self, data, labels):
    if len(data) < self.K:
        raise ValueError('The number of neighbors preceeds the number of datapoints')
    if len(data) != len(labels):
        raise ValueError('The length of data must be the same of the length of labels')
    self.train_data = data
    self.labels = labels
    return self

def predict(self, points):
    K = self.K
    train_data = self.train_data
    labels = self.labels
    num = self.num
    size = int(len(train_data)/num)
    X = []
    L = []
    for i in range(num-1):
        X.append(train_data[i*size:(i+1)*size])
        L.append(labels[i*size:(i+1)*size])
    X.append(train_data[(num-1)*size:])
    L.append(labels[(num-1)*size:])
    partial_res = [partial_knn(points, X[i], L[i], K) for i in range(num)]
    dictionary = mergeReduce(reduceNeighbors, partial_res)
    dictionary = compss_wait_on(dictionary)
    pred_labels = []
    for k,v in dictionary.items():
        pred_labels.append(regression(v))
    return pred_labels

def fit_predict(self, data, labels):
    self.fit(data, labels)
return self.predict(data)

def mergeReduce(function, data):
    """  
    Apply function cumulatively to the items of data, from left to right in binary tree structure, so as to reduce the data to a single value.
    param function: function to apply to reduce data
    param data: List of items to be reduced
    return: result of reduce the data to a single value
    """
    from collections import deque
    q = deque(range(len(data)))
    while len(q):
        x = q.popleft()
        if len(q):
            y = q.popleft()
            data[x] = function(data[x], data[y])
            q.append(x)
        else:
            return data[x]

@task(returns=dict)
def partial_knn(points, XP, labels, K):
    """
    Performs knn on partial training data
    param points: Points to be classified
    param XP: Partial training data points
    param labels: labels of points from XP
    param K: nume of neighbors
    """
    dic = {}
    for x in enumerate(points):
        neighbors = sorted([[labels[i[0]],
            np.linalg.norm(x[1] - i[1])
        for i in enumerate(XP)],
        key=lambda t: t[1][K]]
        dic[x[0]] = neighbors
    return dic

@task(returns=dict, priority=True)
def reduceNeighbors(a, b):
    """
Reduce method to merge the result of two partial_knn methods

```python
K = len(a[0])
for key in b:
a[key] = a[key] + b[key]
a[key] = sorted(a[key], key = lambda t: t[1])[:K]
return a
```

def regression(lista):
    """Takes list of point's neighbors, ex.: [[0, 0.123],[0, 0.155],[1, 0.222]] and finds the average""
    lbl = []
dist = []
for i in lista:
    if i[1] == 0:
        return i[0]
else:
    lbl.append(i[0])
dist.append(i[1])
dist = [i/sum(dist) for i in dist]
return sum([lbl[i]*dist[i] for i in range(len(lbl))])
```

Listing 2: PyCOMPSs Implementation of KNN Regression Algorithm
4 K Means

4.1 K Means Algorithm

K Means is a common machine learning algorithm used for data clustering [2]. It falls within the category of unsupervised learning [1] because it works with unlabeled data. The goal of K Means algorithm is to partition data into K distinct clusters according to their spacial distribution.

The main steps of the algorithm are assignment step and update step. Let us assume that we have a dataset consisting of n points $p_1, p_2, ..., p_n$ in m dimensional space. In the beginning we choose K initial centroids $c_1, ..., c_K$. Centroids are the points that represent the centers of the clusters. Then, we perform the first assignment step: for each point $p$ in our dataset we calculate the distance to each of the centroids, $d_1 = distance(p, c_1), ..., d_K = distance(p, c_K)$ (the usual metrics is Euclidian distance) and then assign the point $p$ to the cluster with the closest centroid (minimal distance). When this is done and each point is assigned to corresponding cluster, we perform the update step. The update step is in fact choosing new centroids of clusters. For each cluster $c$ the coordinate wise average of points from that cluster are calculated,

$$c_{new}^l = \frac{(p_{1c}^l + p_{2c}^l + ... + p_{lc}^l)}{l}$$

$$\vdots$$

$$c_{new}^m = \frac{(p_{1c}^m + p_{2c}^m + ... + p_{lc}^m)}{l}$$

and the obtained value $c_{new}$ represents new centroid of cluster $c$. In that way, we get K new centroids, and iterate again to new assignment step continuing process until convergence.

Convergence

After each successive iteration there might be a difference in the positions of new and old centroids. We calculate the coordinate wise distance between each two matching centroids (old and new centroids of the same cluster)
and sum all of these distances. If succeeding clusters are radically different, matching centroids will be further away and this sum will be bigger. But when distances get smaller and smaller, the succeeding clusters are more similar and we expect less deviation. In the moment when two consecutive iterations return identical set of clusters, this sum will be zero and clusters will not change any more. Due to imprecision of machine calculations in practice we do not wait for this sum to be zero for determining convergence, but to fall under a specified threshold [14].

**Initialization of Centroids**

There are numerous ways for choosing an initial set of centroids. Different initializations could lead to radically different outcome of clustering. Usually, the initial centroids are just K randomly chosen points from a data set, but it is a common problem to get stuck in local minima due to bad initial choice of centroids. To avoid this problem it is sometimes advised to run the algorithm for a number of times and choose the solution with best clustering score [1]. Even though the algorithm is very fast, this approach is still inefficient when working with large amounts of data. There are some other alternative ideas. One way is to randomly pick only a small percentage of data points, and perform the previous concept on them; when clustering of this reduced set is finished, the final centroids are taken as initial centroids for the whole set. One more idea is to randomly assign each point to one of the clusters, and then initial centroids are calculated in the manner of the update step. The approach we will use in this work is to pick the first centroid as a random point from data set and then for the next centroid we choose the data point that is as far away as possible from all the previously chosen.

**4.2 PyCompss Implementation**

Contrary to KNN implementation (3.1.1), here brunt is not on making predictions but on fitting problem. In order to classify new point we only need positions of centroids so we will focus more on training phase. In every iteration original dataset is broken into smaller chunks and for each of these we perform functions in parallel to find closest centroids for points. Then, for cluster in each chunk we take sum and count of points contributing and then merge all the outputs into one obtaining new centroids. It is repeated until convergence. Figure 5 visually presents the idea of the algorithm, and more detailed explanation is below.
import numpy as np
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on
from scipy import stats

class Kmeans(object):
    ""
    Kmeans
    ""
    def __init__(self, K=3, num=10, max_iter=300,
                 tol=1e-10):
        # param num: on how many blocks to
        # partition our data
        self.num = num
        self.K = K
        self.tol = tol
        self.max_iter = max_iter

    def fit(self, data):
        K = self.K
        tol = self.tol
        max_iter = self.max_iter
        num = self.num
        if len(data) < K:
            raise ValueError('The number of clusters
preceeds the number of datapoints')
        size = int(len(data) / num)
        X = [data[i*size:(i+1)*size] for i in range(num-1)]
        X.append(data[num*size:]
        mu = initialization(data, K)
        for i in range(max_iter):
            oldmu = mu.copy()
            partialResult = []
            clusters = []
            for f in range(num):
                cluster = partial_clustering(X[f],
                                              mu, f * size)
                clusters.append(cluster)
                partialResult.append(partial_sum(X[f]
                                               , cluster, f * size))

22
mu = mergeReduce(reduceCenters, partialResult)
mu = compss_wait_on(mu)
mu = [v[1] / v[0] for k, v in mu.items()]
if has_converged(mu, oldmu, tol, i, max_iter):
    break
self.centers = np.array(mu)
return self

def predict(self, new_data):
    mu = self.centers
    self.labels = closest_mu(new_data, mu)
    return self

def fit_predict(self, data):
    self.fit(data)
    return self.predict(data)

def initialization(data, K):
    """
    Function that initializes centroids
    """
    init = [data[np.random.randint(0, len(data))]]
    for i in range(K-1):
        init.append(furthest(init, data))
    return init

def furthest(init, data):
    distances = []
    for point in data:
        distances.append(min([np.linalg.norm(point - i) for i in init]))
    return data[np.argmax(distances)]

def mergeReduce(function, data):
    """
    """
    from collections import deque
    q = deque(range(len(data)))
    while len(q):
        x = q.popleft()
        if len(q):
            y = q.popleft()
data[x] = function(data[x], data[y])
q.append(x)
elif:
    return data[x]

@task(returns=dict)
def partial_clustering(XP, mu, ind):
    ""
    For each point computes the nearest center.
    ""
    dic = {}
    for x in enumerate(XP):
        bestmukey = min([(i[0], np.linalg.norm(x[1] - i[1])) for i in enumerate(mu)],
                         key=lambda t: t[1])[0]
        if bestmukey not in dic:
            dic[bestmukey] = [x[0] + ind]
        else:
            dic[bestmukey].append(x[0] + ind)
    return dic

@task(returns=dict)
def partial_sum(XP, clusters, ind):
    ""
    For each cluster returns the number of points and the sum of all the points that belong to the cluster.
    param XP: points
    param clusters: partial cluster {mu_ind: [pointInd_1, ..., pointInd_n]}
    param ind: point first ind
    return: {cluster_ind: (#points, sum(points))}
    ""
    dic = {}
    for k, v in clusters.items():
        p_idx = np.array(v) - ind
        dic[k] = (len(p_idx), np.sum(XP[p_idx], axis=0))
    return dic

@task(returns=dict, priority=True)
def reduceCenters(a, b):
    ""
    Reduce method to sum the result of two partial_sum methods
Listing 3: PyCOMPSs Implementation of K Means Algorithm

Let us go briefly through all three methods. In .fit() method, besides again checking for feasibility of data, several functions are invoked. Initialization of clustroids is performed in manner explained earlier in this chapter, using function initialization to obtain centers as far from each other as possible. After that, data is split into smaller sets and on each we first invoke function partial_clustering that returns dictionary where to each cluster, belonging points are assigned. After that function partial_sum takes its
output as an input and returns sums of points belonging to each cluster along-
side with number of points participating. Both these functions are tasks and are performed in parallel on each data chunk. When all data is processed, results are merged with function `mergeReduce` calling on `reduceCenters`, which is also a task; and new cluster centers are calculated. We continue with these iterations until convergence is achieved.

The `.predict()` method is much simpler in this case. It only performs one search, using cluster centers obtained by `.fit()` and assigns to each new point the label of the closest centroid.

Method `.fit_predict()` performs `.fit()` and `.predict()` method in a row on the same data. Labels obtained this way could differ a bit from the original labels. (.fit_predict(data) is equivalent to calling .fit(data).predict(data))

Due to random initialization of centroids, two different executions of K Means over the same data could yield different results, producing suboptimal solutions. But as long as the initial centroids are the same, the outcome of clustering will be the same no matter if we run algorithm sequentially or in parallel.
5 Decision Tree

5.1 Decision Tree Algorithm

Decision Tree is a common machine learning algorithm that uses tree-like graph structure combined with simple conditional statements to make a decision [1]. Simply explained, it is like a detective fiction novel. In the beginning everyone is under suspicion for the crime, but as the story proceeds, one by one proof (or attribute) eliminates a lot of candidates. (If it is obvious that the murderer is male, it halves the number of suspects and so on.) Following this way of reasoning, close to the end of a novel, there is little or no dilemma who has committed the crime.

In more detail, Decision Tree algorithm takes a set of labeled data points, chooses an attribute \( f \) and value \( v \) in this attribute. Then, if attribute \( f \) is numerical, data is split in a manner that all data points with value of \( f \) lower than \( v \) are separated in one set, and all the rest in the other. After that, this step is performed on each of two obtained sets, splitting it further until convergence.

Data Splitting

There are two types of attributes in dataset. Numerical and categorical [2]. Splitting numerical attribute goes as described above. We choose some value \( v \) in that attribute, and on one side put all the data with the value greater than \( v \) and to the other all the rest. If attribute is categorical, it is a little different. In this case, it isn’t possible to compare values in sense of greater or less. We split data so that on the one side are all the points with the value equal to \( v \) and to the other all the rest.

Value \( v \) is not picked at random. For specific attribute \( f \) all the possible values \( v_1, v_2, v_3, ... \) are tried as a splitting criteria, and the one with the highest obtained Information Gain after splitting is elected to be terminator [11]. The same goes for the attributes. In each node all the attributes are tested, and we take one with the best results for Information Gain after splitting.

Entropy and Information Gain

Entropy is a measure of system disorder [10]. It is telling us how messy our data is. If entropy of (unempty) set is close to zero it means that one class
makes a great majority and the bigger the Entropy, the closer are fractions of classes making the set. Consider the case of binary classification of \( N \) points, \( n \) of which in first class and \( m = N - n \) in the second. Let \( S \) denote our training set. Entropy \( E(S) \) of our set is

\[
p = \frac{n}{N} \quad \text{and} \quad q = \frac{m}{N} = 1 - p
\]

\[
E(S) = -p \log_2(p) - q \log_2(q)
\]

If set consists of only one class, e.g. class one, then \( q = 0 \) and Entropy will be zero too. If both classes have equal number of instances, both \( p \) and \( q \) will be \( \frac{1}{2} \) and the Entropy is

\[
E(S) = 2 \left(\frac{1}{2}\right) \log_2\left(\frac{1}{2}\right) = 1
\]

Similar goes for multiclass classification.

After splitting set \( S \) to sets \( S_1 \) and \( S_2 \) for each set we calculate new Entropy \( E(S_i), i = 1, 2 \); and Entropy of split is

\[
E(S, f) = \frac{s_1}{N} \cdot E(S_2) + \frac{s_2}{N} \cdot E(S_2)
\]

where \( s_i \) is number of points in set \( S_i \), and \( f \) is an attribute according to which the division is performed.

Information Gain tells us how much new information we gained with specific split. It is calculated as

\[
G(S, f) = E(S) - E(S, f)
\]

It is clear that the bigger the loss in Entropy after splitting, the bigger the increase in Information Gain; and in case Entropy is not changed, the Gain is zero.

**Convergence**

After every performed split there is one more node and one more dataset. At some moment this splitting has to end. The criteria for terminating a branching of a node are the following:

1. If Information Gain is equal to zero.
2. If number of points in dataset are less than specified threshold (minimal sample).
3. If distance of the node from the root have achieved specified limit (maximal depth).
4. If Entropy of set is less than specified threshold (impurity tolerance).
5.2 PyCompss Implementation

Unlike two previous algorithms, we will not split data across examples in order to perform parallelization. It would be extremely difficult to achieve if even possible. But the idea here is so intuitive that it imposes itself. Process will be distributed column wise, simultaneously calculating results for entropy and information gain for splitting on each attribute. This is even further distributed because for each attribute we will simulate split on each of its values in parallel. After best column and best value in that column are chosen, we split data accordingly and go to the next iteration. The idea of the algorithm is visually presented in Figure 6, and more detailed explanation is below.

Implementation

```python
import numpy as np
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on

class DecisionTree(object):
    
    # Decision Tree Algorithm
    # param max_depth: maximum allowed depth of the tree
    # param min_sample: minimum number of points in dataset in order to split it any further
    # param impurity_tol: threshold for entropy of datapoints below which we don't want to split data any further

    def __init__(self, max_depth = 4, min_sample = 5, impurity_tol = 0, tree = None, columns = {}):
        self.max_depth = max_depth
        self.min_sample = min_sample
        self.impurity_tol = impurity_tol
        self.tree = tree
        self.columns = columns

    def fit(self, data):
        att_type = []
        for i in range(data.shape[1] - 1):
            if len(np.unique(data[:, i])) > 20:
                att_type.append('num')
            else:
```
att_type.append('cat')
tree = build_tree(data, self.max_depth, self.min_sample,
                  self.impurity_tol, np.array(att_type))
self.tree = tree
return tree

def predict_proba(self, data):
tree = self.tree
if len(data.shape) == 1:
    return proba2(proba1(data, tree))
if len(data.shape) == 2:
    res = [proba2(proba1(data[i], tree)) for i in range(len(data))]
    return res

def predict(self, data):
tree = self.tree
if len(data.shape) == 1:
    return pred1(data, tree)
if len(data.shape) == 2:
    res = [pred1(data[i], tree) for i in range(len(data))]
    return res

class Question:
    
    A Question is used to partition a dataset.

    def __init__(self, column, value, att_type):
        self.column = column
        self.value = value
        self.att_type = att_type

    def match(self, data):
        if len(data.shape) == 1:
            val = data[self.column]
        if len(data.shape) == 2:
            val = data[:, self.column]
        if self.att_type[self.column] == 'num':
            return val >= self.value
        else:
            return val == self.value

    def split(self, data, column, value, att_type):
        question = Question(column, value, att_type)
        return data[question.match(data)],
data [np.invert(question.match(data))]

def Entropy(X):
    """
    Calculates entropy of array X
    """
    E = 0
    n = len(X)
    for i in np.unique(X):
        m = len(X[X == i])
        if m != 0:
            E += -1*(m/n)*np.log((1e-300+m/n)
    return E

@task(returns=np.float)
def GainValue(Data, Att, value, H_S, att_type):
    l = len(Data)
    X, Y = split(Data, Att, value, att_type)
    if len(X) == 0 or len(Y) == 0:
        return 0.0
    H_S_Att = (len(X)/l)*Entropy(X[:,-1]) + (len(Y)/l)*Entropy(Y[:,-1])
    return H_S - H_S_Att

@task(returns=list)
def Gain(Data, Att, H_S, att_type):
    V = np.unique(Data[:,Att])
    if att_type[Att] == 'cat':
        g = [(value, GainValue(Data, Att, value, H_S, att_type))
             for value in V]
    else:
        g = [(value, GainValue(Data, Att, value, H_S, att_type))
             for value in V[1:]]
    g = compss_wait_on(g)
    val, gain = max(g, key=lambda t: t[1])
    return val, gain

def class_count(a):
    unique, count = np.unique(a[:, -1], return_counts=True)
    return dict(zip(unique, count))
def max_class(a):
    dic = class_count(a)
    return max([(k, v) for k, v in dic.items()])
@task(returns = list)
def Best_split(data, att_type):
    H_S = Entropy(data[:, :-1])
    a = [(None, (None, 0))]
    for Att in range(data.shape[1]-1):
        a.append((Att, Gain(data, Att, H_S, att_type)))
    a = compss_wait_on(a)
    att, (val, gain) = max(a, key=lambda t: t[1][1])
    return att, val, gain

class Leaf:
    def __init__(self, rows):
        self.predictions = class_count(rows)
        self.category = max_class(rows)

class Decision_Node:
    def __init__(self, left_branch, right_branch, att, val, gain, itter, question):
        self.left_branch = left_branch
        self.right_branch = right_branch
        self.att = att
        self.val = val
        self.gain = gain
        self.itter = itter
        self.question = question

def build_tree(data, max_depth, min_sample, impurity_tol,
                att_type, itter=0):
    att_type = att_type
    if len(data) <= min_sample or itter == max_depth
        or Entropy(data[:, :-1]) <= impurity_tol:
        return Leaf(data)
        # stop further branching if dataset is too small or
        # tree is too deep or we used all possible attributes or
        # entropy is small enough
    att, val, gain = compss_wait_on(Best_split(data, att_type))
    question = Question(att, val, att_type)
    if gain == 0:
        return Leaf(data)  # stop further branching if
        # information gain does not improve after splitting

    # recursive call
    left_branch = build_tree(data[:, [att]], max_depth, min_sample, impurity_tol, att_type, itter+1)
    right_branch = build_tree(data[:, [att]], max_depth, min_sample, impurity_tol, att_type, itter+1)
    return Decision_Node(left_branch, right_branch, att, val, gain, itter, question)
left_data, right_data = split(data, att, val, att_type)
iter += 1
left_branch = build_tree(left_data, max_depth, min_sample,
                        impurity_tol, att_type, iter)
right_branch = build_tree(right_data, max_depth, min_sample,
                        impurity_tol, att_type, iter)
return Decision_Node(left_branch, right_branch, att, val,
gain, iter, question)

def prob1(row, node):
    # counts occurrence of classes in each leaf
    # Base case: we’ve reached a leaf
    if isinstance(node, Leaf):
        return node.predictions
    # Decide whether to follow the left or right branch.
    if node.question.match(row):
        return prob1(row, node.left_branch)
    else:
        return prob1(row, node.right_branch)

def prob2(counts):
    # presents counts in percents
    total = sum(counts.values()) * 1.0
    probs = {}
    for lbl in counts.keys():
        probs[lbl] = round(counts[lbl] / total, 2)
    return probs

def pred1(row, node):
    # picks majority class in each leaf
    if isinstance(node, Leaf):
        return node.category
    if node.question.match(row):
        return pred1(row, node.left_branch)
    else:
        return pred1(row, node.right_branch)

Listing 4: PyCOMPSs Implementation of Decision Tree Algorithm
Class `DecisionTree()` has three methods. These are `fit()`, `predict_proba()` and `predict()`. Let’s start from the first one. Method `fit()` builds a decision tree used later for decision making, and does that with function `build_tree`. Function `build_tree` is recursive and in each of iterations checks whether to stop further branching or to continue. Inside it, we invoke two functions: `Best_split` and `split`, and use instances of another two separately defined classes: `Question()` and `DecisionNode()`. Function `Best_split` returns parameters of the best split on current dataset, i.e. attribute, value and information gain. It calls on task function `Gain`, that is performed several times in parallel calculating information gain on each attribute, and it is even further distributed. Function `Gain` makes call on function `GainValue` which is also a task, and is run in parallel to check information gain for each value in current column (calling further on function `Entropy` to achieve this). And that is all concerning parallelization. Results are synchronized and fed to instance of class `Question()` in order to determine rule for splitting. Class `Leaf()` audits if we have reached leaf node, and if not, we split dataset and recursively call a `build_tree` function on each half.

Method `predict_proba()` yields the probabilities of point belonging to each class. When the Tree is fitted and we have final model for decision making, each leaf node contains certain number of data points. Depending on specific leaf creation, it can have only instances of one class or they can be mixed together in various proportions. When we call our Tree on new point, it goes through questioning until it ends up in one of the leafs. Then, label of that point is estimated by the ratio of labels in that specific leaf. If leaf contains only instances of one class, we claim that our point belongs to that class with probability 1. If ratio is different, we claim that it belongs to majority class with probability equal to ratio of that class against all the others.

Method `predict()` returns labels of points according to decision making model gained by `fit()` method. For that purpose, it does calculations as method `predict_proba()`, but it only returns class label omitting the probability.

The results obtained by two successive executions of Decision Tree over the same data set will match perfectly no matter if we run it sequentially or in parallel (on any given number of nodes).
Figure 6: Simplified scheme of Decision Tree implementation
6 Random Forest

In previous chapter we described Decision Tree algorithm. It works well and is reliable for wide range of problems, but when we need some really fine predictions it is not that trustworthy. Deviations could be very big and it is often needed to use other techniques for predictions. One of the ways to overcome this problem is ensemble method [4].

The rationale behind random forest algorithms can be illustrated through a well-known story about Seven Blind Men and an Elephant. Once upon a time seven blind men decided to 'see' the mighty beast the others were talking about so much. Each of them have lay down a hand on an elephant and got its own idea of the beast. One touching elephant’s leg imagined it as a tall tree; the other touching a trunk thought of it as a great snake and so on. Only when they combined all the impressions obtained could they understand what kind of animal it was.

The same way single Tree will probably make a wrong decision, but in combination with a lot of other trees chances for correct predictions are much better. There are two ways for ensembling Decision Trees: Bagging Trees or Random Forest [1].

6.1 Random Forest Algorithm

Random Forest is very powerful machine learning algorithm based on Decision Trees. It takes all the arguments passed to Decision Tree algorithm and additional two: number of features and number of estimators. When building Random Forest model for classification, we in fact build multiple Decision Tree models and perform the voting choosing the class that was predicted by the majority of Trees [2].

Parameter number of estimators tells us how many Trees to ensemble in our model, and number of features decides how many features each single Tree will encounter. If we have a dataset with m features and number of features is m₁, each of the Trees in model will get a ’new’ dataset, obtained by picking at random (that is where the word ’Random’ in the name comes from) only m₁ of the features from the original dataset (and of course corresponding labels) and make a decision making model on this ’new’ set.

Because of the ensembling, Random Forest is less prone to great deviations
and overfitting [2] than Decision Trees. It makes it much more reliable so it is widely used for all kinds of machine learning problems.

6.2 PyCompss Implementation

Random Forest implementation leans a lot on already discussed implementation of Decision Tree, which simplifies this problem a lot and makes code more readable. Key idea for distributing computations is to simultaneously call and fit several different trees, and later ensemble results. The idea of the algorithm is visually presented in Figure 7, and more detailed explanation is below.

Implementation

```python
import numpy as np
import random
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on

import final_DT
#this is script where the implementation of DecisionTree is

class RandomForest(object):
    def __init__(self, n_estimators = 10, n_features = 4,
                 max_depth = 4, min_sample = 5, impurity_tol = 0,
                 tree = None, lista=[]):
        self.n_estimators = n_estimators
        self.n_features = n_features
        self.max_depth = max_depth
        self.min_sample = min_sample
        self.impurity_tol = impurity_tol
        self.tree = tree
        self.lista = lista

    def fit(self, data):
        lista = [ single_tree(data, self.n_features) for
                  i in range(self.n_estimators)]
        lista = compss_wait_on(lista)
        self.lista=lista
        return lista
```
```python
def predict_proba(self, data):
    lista = self.lista
    res = []
    for i in lista:
        res.append(i[0].predict_proba(data[: , i[1]]))
    probs = []
    n = len(res)
    m = len(res[0])
    for i in range(m):
        a = [res[j][i] for j in range(n)]
        mr = mergeReduce(reduce, a)
        mr = compss_wait_on(mr)
        for k in mr.keys():
            mr[k] /= self.n_estimators
        probs.append(mr)
    return probs

def predict(self, data):
    probs = self.predict_proba(data)
    res = []
    for dic in probs:
        m = max([[k,v] for [k,v] in dic.items()],
                key=lambda t: t[1][0])
        res.append(m)
    return res

def sum_dict(list_d):
    d = {}
    for dictionary in list_d:
        for k in dictionary.keys():
            if k not in d.keys():
                d[k] = dictionary[k]
            else:
                d[k] += dictionary[k]
    return d

@task(returns=list)
def single_tree(data, n_features):
    features=random.sample(range(data.shape[1]-1), n_features)
    # pick columns to work with
    X = data[:, features + [data.shape[1]-1]]
    tree = final_DT.DecisionTree()
```

tree.fit(X)
return [tree, features]

def mergeReduce(function, data):
    """Apply function cumulatively to the items of data, from left to right in binary tree structure, so as to reduce the data to a single value.
    :param function: function to apply to reduce data
    :param data: List of items to be reduced
    :return: result of reduce the data to a single value
    """
    from collections import deque
    q = deque(range(len(data)))
    while len(q):
        x = q.popleft()
        if len(q):
            y = q.popleft()
            data[x] = function(data[x], data[y])
            q.append(x)
        else:
            return data[x]

#@task(returns=dict, priority=True)
def reduce(a, b):
    """Reduce method to sum the result of two partial_sum methods
    :param a: partial_sum {cluster_ind:
        (#points_a, sum(points_a))}
    :param b: partial_sum {cluster_ind:
        (#points_b, sum(points_b))}
    :return: {cluster_ind:
        (#points_a+#points_b, sum(points_a+points_b))}
    """
    for key in b:
        if key not in a:
            a[key] = b[key]
        else:
            a[key] = a[key] + b[key]
    return a

Listing 5: PyCOMPSs Implementation of Random Forest Algorithm
Similar to DecisionTree(), RandomForest() class has three methods: .fit(), .predict_proba() and .predict(). Method .fit() leans on function single_tree. This function is a task, and it is performed in parallel populating a list with n_estimators invocations. Each invocation of function fits unique decision tree built on n_features columns, returning model and features that are used for building it. In this way, .fit() creates multiple decision tree models taking note on features used for each. (Parallel implementation of Decision Tree algorithm is reviewed in section 5.2)

Method .predict_proba() has the same role as in Decision Tree implementation (5.2). It makes call to homonymous method on each fitted tree, and returns average of all probabilities yielded giving the ensembled probability of point belonging to certain class.

Method .predict() returns predicted class of input points, using for this purpose above method .predict_proba() and selecting only the label with greatest probability for each point.

The attributes for each Tree that builds ensemble model of Random Forest are chosen randomly, so the results obtained by consecutively executing the algorithm on the same dataset could yield slightly different results. But considering that running Decision Tree over same data gives the same results, as long as initialization of attributes is the same, executing Random Forest over same data yield the same results. That means if we fit single RF model, performing it over same dataset will always yield the same results no matter if we run it in parallel or sequentially.
Figure 7: *Simplified scheme of Random Forest implementation*
7 ADMM LASSO

7.1 ADMM

Alternating Direction Method of Multipliers (ADMM) is a general purpose algorithm for solving a wide class of optimization problems, and that has favorable properties in terms of parallelization [5]. The problem that the algorithm tries to solve has the form

\[
\begin{align*}
\text{minimize} & \quad f(x) + g(z) \\
\text{s.t.} & \quad Ax + Bz = c
\end{align*}
\]

where \( x \in \mathbb{R}^n \), \( z \in \mathbb{R}^m \), \( A \in \mathbb{R}^{p \times n} \), \( B \in \mathbb{R}^{p \times m} \) and \( c \in \mathbb{R}^p \); assuming also that \( f \) and \( g \) are convex. The optimal value for the problem above is

\[
p^* = \inf \{ f(x) + g(z) \mid Ax + Bz = c \}
\]

Lagrangian of the problem is

\[
L_\rho(x, z, y) = f(x) + g(z) + y^T (Ax + Bz - c) + \left( \frac{\rho}{2} \right) \| Ax + Bz - c \|_2^2
\]

ADMM algorithm consists of the iterations

\[
\begin{align*}
x^{k+1} & := \arg \min_x L_\rho(x, z^k, y^k) \\
z^{k+1} & := \arg \min_z L_\rho(x^{k+1}, z, y^k) \\
y^{k+1} & := y^k + \rho (Ax^{k+1} + Bz^{k+1} - c)
\end{align*}
\]

where \( \rho > 0 \).

Convergence

Let’s make the following two assumptions [5]

**Assumption 1**: The (extended-real-valued) functions \( f : \mathbb{R}^n \to \mathbb{R} \cup \{ +\infty \} \) and \( g : \mathbb{R}^m \to \mathbb{R} \cup \{ +\infty \} \) are closed, proper and convex.

**Assumption 2**: The unaugmented Lagrangian \( L_0 \) has a saddle point.

Under these assumptions, the ADMM will imply:
1. Residual convergence. \( r^k \to 0 \) \( \text{for} \ k \to \infty \), where \( r^k \) is a residual value at \( k^{th} \) iteration.

2. Objective convergence. \( f(x^k) + g(z^k) \to p^* \) \( \text{for} \ k \to \infty \).

3. Dual variable convergence. \( y^k \to y^* \) \( \text{for} \ k \to \infty \), where \( y^* \) is a dual optimal point.

Examples show that ADMM is very slow to converge to high accurate solution, but still, usually it achieves a modest accuracy convergence in only few tens of iterations. Fortunately, in practice this is often good enough. The difference in accuracy in applications is not worth the cost of running another hundreds or thousands of iterations. That makes ADMM very useful in cases where modest accuracy is sufficient [5].

**Stopping Criteria**

The necessary and sufficient conditions for the ADMM problem are:

**primal feasibility:**
\[
Ax^* + Bz^* - c = 0,
\]

**and dual feasibility:**
\[
0 \in \partial f(x^*) + A^T y^* \\
0 \in \partial g(z^*) + B^T y^*
\]

Here, \( \partial \) denotes the subdifferential operator [15].

The last condition always holds for \((x^{k+1}, z^{k+1}, y^{k+1})\); the residuals for the other two are the primal and dual residuals \( r^{k+1} \) and \( s^{k+1} \), respectively (where \( s^{k+1} = \rho A^T B (z^{k+1} - z^k) \) and \( r^{k+1} = Ax^{k+1} + Bz^{k+1} - c \), and both converge to zero as ADMM proceeds [5].

As proven in [5], we have
\[
f(x^k) + g(z^k) - p^* \leq -(y^k)^T r^k + (x^k - x^*)^T s^k
\]

It shows that small residuals will yield small objective suboptimality. We don’t know \( x^* \), so we cannot use this inequality directly; but if we estimate that \( \|x^k - x^*\|_2 \leq d \) we get
\[
f(x^k) + g(z^k) - p^* \leq \|(y^k)\|_2 \|r^k\|_2 + d \|s^k\|_2
\]

and we can use the right-hand side as an approximate bound. The reasonable termination criteria is
\[
\|r^k\|_2 \leq \epsilon_{\text{pri}} \quad \text{and} \quad \|s^k\|_2 \leq \epsilon_{\text{dual}}
\]

where \( \epsilon_{\text{pri}} > 0 \) and \( \epsilon_{\text{dual}} > 0 \) are feasibility tolerances [5].
7.2 LASSO

**Least Absolute Shrinkage and Selection Operator (LASSO)** is a $l_1$ regularized linear regression problem defined as follows

$$\text{minimize } \frac{1}{2} \|Ax - b\|_2^2 + \lambda \|x\|_1$$

where $\lambda > 0$ is a scalar regularization parameter. In ADMM form, the LASSO problem can be written as

$$\text{minimize } f(x) + g(z)$$

$$\text{s.t. } x - z = 0$$

where $f(x) = \frac{1}{2} \|Ax - b\|_2^2$ and $g(z) = \lambda \|x\|_1$. In that case we obtain ADMM iterations for LASSO as follows:

$$x^{k+1} := (A^T A + \rho I)^{-1} (A^T b + \rho (z^k - u^k))$$

$$z^{k+1} := S_{\lambda/\rho} (x^{k+1} + u^k)$$

$$u^{k+1} := u^k + x^{k+1} - z^{k+1}$$

where $S_\kappa(a) = \begin{cases} 
    a - \kappa & a > \kappa \\
    0 & |a| \leq \kappa \\
    a + \kappa & a < -\kappa 
\end{cases}$ is soft thresholding operator [5].

**Splitting across Examples**

We want to solve LASSO model fitting in distributed way splitting dataset across examples, having large number of examples with a modest number of features. It is a very common problem suitable in case where amount of datapoints is very large or when the data is already collected in distributed fashion.

Having general convex model fitting problem

$$\text{minimize } l(Ax - b) + r(x)$$

where $l : \mathbb{R}^m \to \mathbb{R}$ is a convex loss function and $r$ is a convex regularization function; we partition $A$ and $b$ by rows

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_N \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$$
where $A_i \in \mathbb{R}^{m_i \times n}$, $b_i \in \mathbb{R}^{m_i}$ and $\sum_{i=1}^{N} m_i = m$. The algorithm for solving LASSO problem with data splitted in this manner is stated as follows

$$x_i^{k+1} := (A_i^T A_i + \rho I)^{-1} (A_i^T b_i + \rho (z^{k} - u_i^{k}))$$

$$z^{k+1} := S_{\lambda/\rho} (\pi^{k+1} + \bar{\pi}^{k})$$

$$u_i^{k+1} := u_i^{k} + x_i^{k+1} - z_i^{k+1}$$

Here, $\pi^{k+1}$ is an average value of all $x$ vectors obtained in the $k+1^{st}$ iteration, and $\bar{\pi}^{k}$ is an average of all $u$ vectors in the $k^{th}$ iteration.

### 7.3 PyCompss Implementation

This algorithm slightly differs from others. First of all, it is more theoretical, and it is not used as a single algorithm, but rather as an operation used in a wider context to optimize results (for example in neural networks). For this reason it is not implemented as a class like the previous algorithms, but as a function. Another difference, and one that we are concerned about more, is that the algorithm parallelization is already defined in [5]. It means that it is not on us to decide on distributing strategy but only to follow previously explained principles and pack them in accordance with PyCOMPSs. Hence we will not again explain the details of parallelization of the code below. The idea of the algorithm is visually presented in Figure 8.

By construction, the results obtained with partitioning data on smaller blocks will yield the same results as if we work on data without partitioning. That implies that the solution will not differ if we run algorithm sequentially and in parallel (on any given number of nodes).

**Implementation**

```python
from scipy.linalg import solve_triangular
import numpy as np
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on

def lasso(A, b, mm, lmbd, rho=1, itter=300,
          reltol=1e-2, abstol=1e-4):
    """
    Least Absolute Shrinkage and Selection Operator
    (LASSO) regression analysis.
    :param A: Matrix
    :param b: Vector
    """
```
:param num: Number on how many blocks to split matrix
:param lmbd: lambda
:param rho: rho
:param iter: Iterations
:param reltol: Relative tolerance
:param abstol: Absolute tolerance
:return:

(m, n) = A.shape
part = int(m/num)  # length of each block
A_list=[A[i*part:(i+1)*part] for i in range(num)]
b_list=[b[i*part:(i+1)*part] for i in range(num)]
if part < m/num:
    A_list.append(A[num*part:])  # adding last block (it is shorter than the rest)
b_list.append(b[num*part:])

x_old = [np.zeros(n) for i in range(num)]
u_old = [np.zeros(n) for i in range(num)]
z_old = np.zeros(n)

F = step0(A_list, part, n, m, num)
L = cholesky(F)
ATb = atb(A_list, b_list, num)
ATb = compss_wait_on(ATb)
k = lmbd/(rho*num)
# These stay the same through iterations, so
# we calculate them here

nxstack = 0
nystack = 0
for i in range(itter):
    y=triang_y(L, ATb, rho, z_old, u_old, num)
    x_new = triang_x(L, y, num)

    avg_x = average(x_new)
    avg_u = average(u_old)
    avg = avg_summing(avg_x, avg_u)
    z_new = sft_trsh(k, avg)

    u_new = summing(u_old, x_old, z_old, num)

dualres=dual_resolution(num, rho, z_new, z_old)
```python
primes = pri_resolution(x_old, z_old)

x_old = x_new.copy()
z_old = z_new.copy()
u_old = u_new.copy()

nxstack = add_to_stack(nxstack, x_new)
nystack = add_to_stack(nystack, u_new)

eps_pri, eps_dual = eps(num, n, abstol, reltol, nxstack, z_new, nystack)

converged = has_converged(primes, eps_pri, dualres, eps_dual)
if compss_wait_on(converged):
    break

x_new = compss_wait_on(x_new)
z_new = compss_wait_on(z_new)
return x_new, z_new

def step0(array, rho, n, m, num):
    """computes left hand side of equation""
    return [func_0(array[i], rho, n, m) for i in range(num)]

@task(returns=np.array)
def func_0(a, rho, n, m):
    F = np.dot(a.T, a) + rho * np.eye(n)
    return F

def cholesky(list_a):
    return [func_col(f) for f in list_a]

@task(returns=np.array)
def func_col(f):
    return np.linalg.cholesky(f)

def atb(A, b, num):  # returns products Ai.T * b
    # (first part of right hand side)
    return [func_atb(A[i], b[i]) for i in range(num)]

@task(returns=np.array)
def func_atb(a, b):
    return np.dot(a.T, b)
```
def triang_y(L, b, rho, z, u, num):  # calculating first triangular equation
    return [triang(L[i], b[i] + rho * (z[i] - u[i])) for i in range(num)]

def triang_x(L, y, num):
    return [triang(L[i].T, y[i]) for i in range(num)]

@task(returns=np.array)
def triang(a, b):
    return solve_triangular(a, b)

def sft_trsh(k, a):
    return [func_sft(k, a[i]) for i in range(len(a))]

@task(returns=np.float32)
def func_sft(k, element):
    if element > k:
        return element - k
    elif np.abs(element) <= k:
        return 0
    else:
        return element + k

def average(vectors):
    vectors = np.array(vectors)
    n, m = vectors.shape
    return list(func_sum(vectors) / n)

def avg_summing(avg_x, avg_u):
    vectors = np.array([avg_x, avg_u])
    return func_sum(vectors)

def summing(u_old, x_old, z_old, num):
    vectors = np.array([u_old, x_old, list(np.array(z_old) * -1) for i in range(num)])
    return func_sum(vectors)

@task(returns=np.array)
def func_sum(array):
    return np.sum(array, axis=0)

def dual_resolution(num, rho, z_new, z_old):
    return np.sqrt(num) * rho * np.linalg.norm(np.array(z_new) - np.array(z_old))
def pri_resolution(x_old, z_old):
a = [func_pri(x_old[i], z_old) for i in range(len(x_old))]
return np.average(a)

@task(returns = np.array)
def func_pri(x, z):
    return np.linalg.norm(np.array(x) - np.array(z))

def add_to_stack(stack, new):
    stack += np.linalg.norm(new)
    return stack

def eps(num, n, abstol, reltol, nxstack, z_new, nystack):
    eps_pri = (np.sqrt(num * n)) * abstol + reltol*
    (max(np.sqrt(nxstack), np.sqrt(num)*
         np.linalg.norm(z_new)))
    eps_dual=np.sqrt(num*n)*abstol+reltol*nystack
    return eps_pri, eps_dual

def has_converged(prires, eps_pri, dualres, eps_dual):
    return prires <= eps_pri and dualres <= eps_dual

Listing 6: PyCOMPSs Implementation of LASSO Algorithm
Figure 8: Simplified scheme of LASSO implementation
8 Simulation Experiments

We now test the developed algorithms in Chapters 3-7 on real data. For each algorithm, we adopt a standard benchmark against which we compare the algorithms’ performance.

8.1 Simulation Setup and Benchmarks

Scikit Learn is a Python library for machine learning [18]. It includes great number of common ML algorithms and it is one of the most popular tools for data science and machine learning. As benchmark algorithms for the first five algorithms we will use corresponding Scikit Learn's implementations. Precisely:

- for KNN Classification - KNeighborsClassifier,
- for KNN Regression - KNeighborsRegressor,
- for K Means - KMeans,
- for Decision Tree - DecisionTreeClassifier and
- for Random Forest RandomForestClassifier.

As a benchmark for LASSO we will use MPI implementation available in [17].

Each algorithm is tested on three different datasets and results are presented in tables and figures below. Considering that algorithms have calls of random function in some parts of implementations we can get significantly different results in two executions of the same algorithm on the same data. For this reason we will execute each algorithm 100 times and show the mean, min, max and percentiles of metric used for evaluation.

All tests are performed on a single laptop computer with following specifications:

- Processor: Intel(R) Celeron(R) CPU 1000M @ 1.80 GHz 1.80 GHz
- Installed memory (RAM): 4.00 GB (3.59 GB usable)
- System type: 64-bit Operating System, x64-based processor

In this thesis tests are run sequentially on single computer, but running in parallel on computer cluster will be part of future work.

8.2 Datasets

Let us describe all the datasets used for evaluation performances.
Iris dataset [18]. This is one of the most famous datasets in machine learning community. It is often used as a first level testing set for different implementations, so following that tradition we will use it here too. Set consists of 150 instances of three subspecies of Iris: Iris Setosa, Iris Virginica and Iris Versicolors (50 instances of each); and 4 attributes containing data about sepal and petal dimensions. Set is suitable both for classification and clustering and we will use it for testing each of the algorithms except for KNN Regressor and LASSO.

Wine dataset [18]. Again, this is a commonly used multi-class set consisting of three different sorts of wine. Set contains 178 instances (59, 71, 48 of each class respectively) and 13 attributes. This set is mainly used for classification and we will use it for testing KNN Classifier and Decision Tree.

Breast Cancer dataset [18]. This is another widely used dataset suited for classification algorithms. It is a binary classification set, with labels: 1 if cancer is diagnosed and 0 if it is not. There are 569 considered patients in total and 30 real positive attributes. This set will be used for testing performances of KNN Classifier.

Student Performance dataset [22]. This dataset is publicly available at UCI Machine Learning repository. It consists of 649 instances (students) with 33 attributes. Target values here are Math grades ranging 0-20, so it can be considered both as a set for classification (with 20 classes) and a set for regression. With regard to this, we will use this set for testing KNN Regression and Random Forest Classification.

Carbon Nanotubes dataset [22]. This is largest dataset used in this work. It is common practice to use lighter sets for initial testing in order to perform tests faster, but sets suitable for regression are often not that easy to find or not publicly available, so we won’t bother about its size. Dataset consists of 10721 points in 8 dimensions, and it will be used for testing KNN Regression.

Dataset 1. This is first artificial dataset introduced here. Scikit Learn offers simple tools for creating these kinds of sets, and we used its tool make_regression() to create $250 \times 4$ dataset suitable for regression algorithms. We use this set to test KNN Regression algorithm.

Pima Indians dataset [23]. This set is publicly available at Kaggle platform, but in order to download it you need to create an account (if you don’t have one already). Data is collected from one Indian tribe with a goal to detect the causes of diabetes within the group. It consists of 9 different medical properties (blood pressure, insulin,...) acquired form 768 persons. Data is labeled with 1 if person has diabetes and 0 if not. We will use this set for testing K Means and Decision Tree algorithms.
Dataset 2. This is another artificial dataset, but this time created suitable for clustering. As known, K Means performs best on data that is scattered in clusters with Gaussian distribution. Set consists of 800 points in 16 dimensions; 200 points taken from each of 4 randomly chosen Gaussian distributions. Naturally, it will be used to test performances of K Means algorithm.

Wine Quality dataset [23]. This dataset is similar to one mentioned above (Wine dataset), but as the first one is considering red wine this one is about white wine. Set is publicly available both on Kaggle and UCI repository. It consists of 4868 instances of white wine with 12 physicochemical attributes and target class is representing quality of wine ranging from 0 to 10. This dataset can be used both as a regression and classification set, but we will use it only for testing Random Forest classifier.

Dataset 3 [5]. Our last dataset is also artificially created, but it is available in [5], so that you don’t have to work with a new dataset in every run. This set consists of a matrix A and a vector b, each of them broken down into 4 blocks of length 40 and matrix A having 500 columns. Matrix is populated with entries from Gaussian (0,1) distribution and vector from Uniform (0,1) distribution. This set we will use to test performances of LASSO method.

8.3 Metrics
Testing different problems demands different metrics of success. Let us discuss the metrics used in this work.

Accuracy score. For classification problems, we will use Scikit Learn function accuracy_score that takes predicted values and ground truth values and calculates the ratio of correctly predicted instances. It ranges from 0.0 if each predicted value is wrong to 1.0 if all are correct.

R² score. For regression problems another metric is needed. We will use Scikit Learn’s function r2_score that calculates R² regression score. Maximum value for this metric is 1.0 but it can also be negative. Constant model, that always predict the same value disregarding the input has the score of 0.0.

Silhouette score. Considering that clustering is unsupervised method of learning we can’t use labels of classes to determine the success of our performance. Instead, we will use Scikit Learn’s function silhouette_score that computes the mean Silhouette Coefficient of all samples. Simply explained, for each point ratio of closeness to the centroid of corresponding and to the closest other cluster is calculated, and then all this results are averaged. Score ranges form -1.0 in worst case to 1.0 in the best, where 0.0 indicates
overlapping clusters.

8.4 KNN Classifier

For testing performances of KNN Classifier we will use Iris dataset, Wine dataset and Breast Cancer dataset, and the metric for evaluation is accuracy score. Each set is divided into training and test subsets with ratio 80:20. Results are also graphically presented in Figure 12 where circles represent training points, colored with respect to class they belong to, and stars are test points accordingly colored in the color of predicted class. Let’s discuss the results obtained.

Iris Dataset
Classification of Iris dataset gave identical results for both implementations with mean accuracy of 0.961. This matching should not surprise us because set is very simple and relatively easy for classification. Detailed results are presented in Table 2. Figure 9 shows the results obtained.

<table>
<thead>
<tr>
<th>Iris Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.961</td>
<td>0.8666</td>
<td>1.0</td>
<td>0.9333</td>
<td>0.9666</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.961</td>
<td>0.8666</td>
<td>1.0</td>
<td>0.9333</td>
<td>0.9666</td>
</tr>
</tbody>
</table>

Table 2: KNN Classifier performances on Iris dataset

Wine Dataset
Again, we have acquired completely the same results for both implementations, this time with an average accuracy of 0.7066. Detailed results are presented in Table 3. Figure 10 shows the results obtained.

<table>
<thead>
<tr>
<th>Wine Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.7066</td>
<td>0.5555</td>
<td>0.8888</td>
<td>0.6666</td>
<td>0.75</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.7066</td>
<td>0.5555</td>
<td>0.8888</td>
<td>0.6666</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 3: KNN Classifier performances on Wine dataset

Breast Cancer Dataset
For the third time results are matching perfectly, and this tells us that our implementation is compitable with that of Scikit Learn. Mean accuracy for Breast Cancer dataset is 0.9254 and detailed results are in the Table 4. Figure 11 shows the results obtained.
### Breast Cancer Dataset

<table>
<thead>
<tr>
<th>Mean acc.</th>
<th>Min acc.</th>
<th>Max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.9254</td>
<td>0.8684</td>
<td>0.9649</td>
<td>0.9122</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.9254</td>
<td>0.8684</td>
<td>0.9649</td>
<td>0.9122</td>
</tr>
</tbody>
</table>

Table 4: KNN Classifier performances on Breast Cancer dataset

### 8.5 KNN Regression

Sets used in testing KNN Regression are Student Performance dataset, Carbon Nanotubes dataset and Dataset 1. Evaluation metric here is $R^2$ score for regression. We split original datasets in training and test subsets with ratio 95:5 for Student Performance and Dataset1, and 99:1 for Carbon Nanotubes dataset. We need smaller test sets here than in classification problems just for the sake of visualization. Results are visualized and shown in Figure 16; training points are represented as circles and colored by the value they bear (according the colorbars shown). Test points are in shape of star and also colored according to the predicted value. Hopefully, you will struggle to see them, because in case of good prediction they should fit nicely with neighboring points. Considering this, we have annotated each test point with value predicted.

**Student Performance Dataset**

Results obtained for this set are presented in Table 5. Mean score gained by PyCOMPSs implementation is 0.7704 and by Scikit Learn is 0.7663. Hence, our implementation gives higher score in this example; it can be concluded that the deviation is also larger considering that maximal score is higher and minimal is lower in our implementation, but percentiles are better measure for that (because min and max are often outliers), and it tells us that our prediction is in general more accurate. Figure 13 shows the results obtained.

<table>
<thead>
<tr>
<th>Mean score</th>
<th>Min score</th>
<th>Max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.7704</td>
<td>0.312</td>
<td>0.952</td>
<td>0.7163</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.7663</td>
<td>0.3649</td>
<td>0.9418</td>
<td>0.7071</td>
</tr>
</tbody>
</table>

Table 5: KNN Regression performances on Student Performance dataset
Carbon Nanotubes
Detailed results are presented in Table 6. Mean score obtained by PyCOMPSs implementation is 0.9636 and by Scikit Learn is 0.9708. Looking at the rest of the Table, we see that Scikit Learn implementation indeed has better performances in this case, but only by the small margin. Figure 14 illustrates the obtained results.

<table>
<thead>
<tr>
<th>Carbon Nanotubes Dataset</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.9636</td>
<td>0.9497</td>
<td>0.9732</td>
<td>0.9605</td>
<td>0.967</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.9708</td>
<td>0.9597</td>
<td>0.9786</td>
<td>0.968</td>
<td>0.974</td>
</tr>
</tbody>
</table>

Table 6: KNN Regression performances on Carbon Nanotubes dataset

Dataset 1
Detailed results are presented in Table 7. Mean score obtained by PyCOMPSs implementation is 0.7958 and by Scikit Learn is 0.8071. Again, Scikit Learn is better by small margin; but looking at the percentiles we can tell that it in general also varies a little less, although minimal and maximal scores are voting in favor of PyCOMPSs. Figure 15 shows the results obtained.

<table>
<thead>
<tr>
<th>Dataset 1</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.7958</td>
<td>0.0911</td>
<td>0.9505</td>
<td>0.7235</td>
<td>0.8847</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.8071</td>
<td>-0.3589</td>
<td>0.9389</td>
<td>0.7754</td>
<td>0.8829</td>
</tr>
</tbody>
</table>

Table 7: KNN Regression performances on Dataset 1

8.6 K Means
Sets used for testing K Means performances are Iris dataset, Pima Indians dataset and Dataset 2. Since this is unsupervised algorithm, we don’t need labels nor do we split data on train and test sets; metric used for evaluation here is silhouette score. Results will be discussed below and also presented visually in Figure 20; datapoints are presented as circles colored according to cluster they are assigned to, and red stars present clustroids of corresponding clusters.
Iris Dataset
Results obtained for Iris dataset are presented in Table 8. Both implementations show fixed result; for PyCOMPSs freezed at 0.5789 and for Scikit Learn at 0.5816. There is no doubt that Scikit Learn performs better in this example, but again results of two implementations differ very little. Figure 17 shows the results obtained.

<table>
<thead>
<tr>
<th>Iris Dataset</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.5789</td>
<td>0.5789</td>
<td>0.5789</td>
<td>0.5789</td>
<td>0.5789</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.5816</td>
<td>0.5816</td>
<td>0.5816</td>
<td>0.5816</td>
<td>0.5816</td>
</tr>
</tbody>
</table>

Table 8: K Means performances on Iris dataset

Pima Indians Dataset
Results obtained on Pima Indians dataset are shown in Table 9. Again we have the same case as in the previous example, now with values 0.6955 for PyCOMPSs and 0.6429 for Scikit Learn implementation. Figure 18 shows the results obtained.

<table>
<thead>
<tr>
<th>Pima Indians Dataset</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.6355</td>
<td>0.6355</td>
<td>0.6355</td>
<td>0.6355</td>
<td>0.6355</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.6429</td>
<td>0.6429</td>
<td>0.6429</td>
<td>0.6429</td>
<td>0.6429</td>
</tr>
</tbody>
</table>

Table 9: K Means performances on Pima Indians dataset

Dataset2
Detailed results are presented in Table 10. Mean score obtained by PyCOMPSs implementation is 0.2669 and by Scikit Learn is 0.3433. In this example Scikit Learn performs significantly better than PyCOMPSs, which is also confirmed by percentile values (both implementation show low deviation, even though maximal scores are equal). Figure 19 shows the results obtained.

<table>
<thead>
<tr>
<th>Dataset2</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.2669</td>
<td>0.2669</td>
<td>0.2669</td>
<td>0.2669</td>
<td>0.2669</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
</tr>
</tbody>
</table>

Table 10: K Means performances on Dataset2

8.7 Decision Tree
Sets used for testing performances of Decision Tree are Iris dataset, Pima Indians dataset and Wine dataset; and the metric for evaluation is accuracy score. Sets are divided into training and test subsets with ratio 80:20. Re-
<table>
<thead>
<tr>
<th>Dataset 2</th>
<th>mean score</th>
<th>min score</th>
<th>max score</th>
<th>25 percentile score</th>
<th>75 percentile score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.2669</td>
<td>0.2224</td>
<td>0.3433</td>
<td>0.2633</td>
<td>0.2676</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
<td>0.3433</td>
</tr>
</tbody>
</table>

**Table 10: K Means performances on Dataset 2**

Results are discussed below and presented in Figure 24 in form of scatter plots; training points are presented as circles colored according to classes they belong, and test points are shown as stars colored according to predicted class while edge of star has color of ground truth label.

**Iris Dataset**

Detailed results are presented in Table 11. Mean accuracy obtained by PyCOMPSs is 0.9396 and by Scikit Learn is 0.945. Difference is small and performances seem even closer when we take a look at the rest of the table. Figure 21 shows the results obtained.

<table>
<thead>
<tr>
<th>Iris Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.9396</td>
<td>0.7666</td>
<td>1.0</td>
<td>0.9</td>
<td>0.9666</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.945</td>
<td>0.7666</td>
<td>1.0</td>
<td>0.9333</td>
<td>0.9666</td>
</tr>
</tbody>
</table>

**Table 11: Decision Tree performances on Iris dataset**

**Pima Indians Dataset**

Results obtained at Pima Indians dataset are shown in Table 12. PyCOMPSs implementation yields mean accuracy of 0.7516 and Scikit Learn 0.7484. In this example PyCOMPSs performs better showing also lower variance in results. Figure 22 illustrates the obtained results.

<table>
<thead>
<tr>
<th>Pima Indians Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.7516</td>
<td>0.6075</td>
<td>0.8481</td>
<td>0.7215</td>
<td>0.7721</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.7484</td>
<td>0.6329</td>
<td>0.8481</td>
<td>0.7088</td>
<td>0.7848</td>
</tr>
</tbody>
</table>

**Table 12: Decision Tree performances on Pima Indians dataset**

**Wine Dataset**

Detailed results obtained are presented in Table 13. In this example PyCOMPSs implementation yields mean accuracy of 0.9102 and Scikit Learn
0.918. Results differ very little, and looking at the rest of the table, only difference is seen in 75\textsuperscript{th} percentile, indicating higher skewness towards 1 for the Scikit Learn results. Figure 23 illustrates the obtained results.

<table>
<thead>
<tr>
<th>Wine Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.9102</td>
<td>0.6944</td>
<td>1.0</td>
<td>0.8888</td>
<td>0.9444</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.918</td>
<td>0.6944</td>
<td>1.0</td>
<td>0.8888</td>
<td>0.9722</td>
</tr>
</tbody>
</table>

Table 13: Decision Tree performances on Wine dataset

8.8 Random Forest

For inspecting performances of Random Forest we used Iris dataset, Student Performance dataset and Wine Quality dataset. Metric used is accuracy score and each set is being split into training and test sets with ratio 80:20. Results are discussed below and also presented in a form of a scatter plot in Figure 28. Training points are represented by circles, colored according to belonging class and stars are test points, colored as class predicted.

Iris Dataset

Results obtained are shown in detail in Table 14. Mean accuracy yielded by PyCOMPSs implementation is 0.9473 and by Scikit Learn it is 0.9556. Difference is not significant especially considering that percentile values are matching. Figure 25 shows the results obtained.

<table>
<thead>
<tr>
<th>Iris Dataset</th>
<th>mean acc.</th>
<th>min acc.</th>
<th>max acc.</th>
<th>25 percentile acc.</th>
<th>75 percentile acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.9473</td>
<td>0.8333</td>
<td>1.0</td>
<td>0.9333</td>
<td>0.9666</td>
</tr>
<tr>
<td>Scikit Learn</td>
<td>0.9556</td>
<td>0.8666</td>
<td>1.0</td>
<td>0.9333</td>
<td>0.9666</td>
</tr>
</tbody>
</table>

Table 14: Random Forest performances on Iris dataset

Student Performance Dataset

Detailed results are presented in Table 15. Mean accuracy obtained by PyCOMPSs implementation is 0.2397 and by Scikit Learn is 0.3098. This rather large difference is also confirmed by the rest of the values in the Table. Scikit Learn implementation has shown much better performances in this example maybe pointing out that PyCOMPSs implementation has problems with datasets like this one. Figure 26 illustrates the obtained results.
### Table 15: Random Forest performances on Student Performance dataset

![Table 15](image)

### Wine Quality Dataset

Results obtained are shown in Table 16. In this example PyCOMPSs yields average accuracy of 0.5205 and Scikit Learn 0.5376. Again, Scikit Learn shows to perform better but only by the small margin on this dataset. The rest of the values also confirm this ratio of success of two algorithms. Figure 27 shows the results obtained.

![Table 16](image)

### 8.9 LASSO

**Dataset 3**

For testing LASSO performances we used Dataset 3 as mentioned above. Tests are run on both implementations with parameters $\lambda = 0.5$ and $\rho = 1$. We have calculated primal and dual elements described in the implementation of LASSO, and also values of objective function. Objective function is

$$
\text{minimize} \quad \frac{1}{2} \|Ax - b\|_2^2 + \lambda \|x\|_1
$$

as mentioned above, but it is performed over each of 4 blocks of data and expressed as an average value.

Both implementations achieve the convergence in only 30 iterations with roughly similar values. Values obtained in last iteration are presented in Table 17, and in Figure 29 we can see how each value changes over iterations.
Looking at the primal and dual elements in Figure 29 we can see that they match almost perfectly. Behavior of an Objective function is similar in both cases, but its value is visibly larger in PyCOMPSs implementation. Magnitude of the difference is rather small, but still it might be good to find out the reasons for different convergence points. Considering that both implementations are derived from the same paper, most probable cause could be precision of libraries used for linear algebra since they differ a little between C and Python.

<table>
<thead>
<tr>
<th>Dataset 3</th>
<th>primal residual</th>
<th>dual residual</th>
<th>primal epsilon</th>
<th>dual epsilon</th>
<th>objective function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCOMPSs</td>
<td>0.0290</td>
<td>0.0477</td>
<td>0.0973</td>
<td>0.0785</td>
<td>18.8159</td>
</tr>
<tr>
<td>MPI</td>
<td>0.0355</td>
<td>0.0667</td>
<td>0.0985</td>
<td>0.0696</td>
<td>17.1890</td>
</tr>
</tbody>
</table>

*Table 17: LASSO performances on Dataset 3*
Figure 9: Iris dataset

Figure 10: Wine dataset

Figure 11: Breast Cancer dataset

Figure 12: Performances of KNN Classification
Figure 13: Student Performance dataset

Figure 14: Carbon Nanotubes dataset

Figure 15: Dataset 1

Figure 16: Performances of KNN Regression
Figure 17: Iris dataset

Figure 18: Pima Indians dataset

Figure 19: Dataset 2

Figure 20: Performances of K Means clustering
Figure 21: Iris dataset

Figure 22: Pima Indians dataset

Figure 23: Wine dataset

Figure 24: Performances of Decision Tree classification
Figure 25: *Iris dataset*

Figure 26: *Student Performance dataset*

Figure 27: *Wine Quality dataset*

Figure 28: *Performances of Random Forest classification*
Figure 29: Performances of LASSO
9 Conclusion

In this work, we have presented a parallel implementation of several clustering and classification algorithms using PyCOMPSs. For each algorithm, we have described a methodology of how the algorithm is partitioned into smaller functions that can run in parallel at runtime. We illustrated performance of the algorithms on several real public datasets. Looking at the overall scores obtained, we can tell that implementations in Scikit Learn often perform better than those introduced here, but difference is rather small and we can be satisfied by the success achieved, especially in performances of the KNN. Since this thesis is just a part of currently active project, the work started here will continue expanding beyond the scope of this thesis. Initial pool of algorithms can easily be extended as these are only a few of numerous popular algorithms. Even so, there is always room for improvements. We can and should investigate in more details some ideas for optimizing calculations and adjusting parameters and settings. Of course, another important item to cover, as soon as it becomes possible, is to try and test all algorithms on a computer cluster. This work has been done in the context of the Horizon 2020 European Commission-funded project I-BiDaaS, Grant Agreement No. 780787.
This work wouldn’t even be started, let alone finished, without a great deal of support and guidance. I will use this occasion to thank those that helped me in shaping my master thesis in last couple months. 

First of all, I want to thank my mentor Dušan Jakovetić. It’s him who offered me the possibility to take part in this project and proposed it to be an idea for thesis. He is dedicated both as a professor and as a mentor, very interested in matters discussed and willing to help and answer every question.

I am very grateful to the other colleagues on the project too, especially for the support in technical settings for this work. Particular appreciation to the group from the BSC for supervising the implementation details delivered here.

I am also more than thankful to my closest friends - Cki, Bane, Djile, Hornjak, Jela, Kristina and Papara; for help and support not only in working hours but in the resting periods as well and everything that I’ve been through in these past months.

Finally, very great deal of gratitude I owe to my family - my parents, brother and sister. They are the pillars of my world and nothing would be possible without them.
11 List of Figures

1a Task definitions
1b Main program
2 Dependency graph
3 Example how different choices of K lead to different classification
4 Graphical representation of KNN algorithm implementation
5 Graphical representation of K Means algorithm implementation
6 Graphical representation of Decision Tree algorithm implementation
7 Graphical representation of Random Forest algorithm implementation
8 Graphical representation of LASSO implementation
9 Performances of KNN Classification on Iris dataset
10 Performances of KNN Classification on Wine dataset
11 Performances of KNN Classification on Breast Cancer dataset
12 Performances of KNN Classification
13 Performances of KNN Regression on Student Performance dataset
14 Performances of KNN Regression on Carbon Nanotubes dataset
15 Performances of KNN Regression on Dataset 1
16 Performances of KNN Regression
17 Performances of K Means Clustering on Iris dataset
18 Performances of K Means Clustering on Pima Indians dataset
19 Performances of K Means Clustering on Dataset 2
20 Performances of K Means Clustering
21 Performances of Decision Tree Classification on Iris dataset
22 Performances of Decision Tree Classification on Pima Indians dataset
23 Performances of Decision Tree Classification on Wine dataset
24 Performances of Decision Tree Classification
25 Performances of Random Forest Classification on Iris dataset
26 Performances of Random Forest Classification on Student dataset
27 Performances of Random Forest Classification on Wine Quality dataset
28 Performances of Random Forest Classification
29 Performances of LASSO method on Dataset 3
12 List of Tables

1 Available arguments for task decorator
2 Performances of KNN Classifier on Iris dataset
3 Performances of KNN Classifier on Wine dataset
4 Performances of KNN Classifier on Breast Cancer dataset
5 Performances of KNN Regression on Student Performance dataset
6 Performances of KNN Regression on Carbon Nanotubes dataset
7 Performances of KNN Regression on Dataset 1
8 Performances of K Means Clustering on Iris dataset
9 Performances of K Means Clustering on Pima Indians dataset
10 Performances of K Means Clustering on Dataset 2
11 Performances of Decision Tree Classifier on Iris dataset
12 Performances of Decision Tree Classifier on Pima Indians dataset
13 Performances of Decision Tree Classifier on Wine dataset
14 Performances of Random Forest Classifier on Iris dataset
15 Performances of Random Forest Classifier on Student Performance dataset
16 Performances of Random Forest Classifier on Wine Quality dataset
17 Performances of LASSO method on Dataset 3
### List of Listings

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PyCOMPSs implementation of KNN Classification algorithm</td>
</tr>
<tr>
<td>2</td>
<td>PyCOMPSs implementation of KNN Regression algorithm</td>
</tr>
<tr>
<td>3</td>
<td>PyCOMPSs implementation of K Means Clustering algorithm</td>
</tr>
<tr>
<td>4</td>
<td>PyCOMPSs implementation of Decision Tree Classification algorithm</td>
</tr>
<tr>
<td>5</td>
<td>PyCOMPSs implementation of Random Forest Classification algorithm</td>
</tr>
<tr>
<td>6</td>
<td>PyCOMPSs implementation of LASSO algorithm</td>
</tr>
</tbody>
</table>
References


[16] https://www.bsc.es/


[21] https://www.python.org/


[23] https://www.kaggle.com/
Paralelna implementacija algoritama mašinskog učenja koristeći PyCompss

Jezik publikacije: engleski

Jezik izvoda: engleski

Zemlja publikovanja: Republika Srbija

Uže geografsko područje: Bačka


Izdavač: autorski reprint

Mesto i adresa: Novi Sad, Trg Dositeja Obradovića 4

Fizički opis rada: 9 poglavlja, 78 strana, 23 lit. citata, 17 tabela, 29 grafika, 6 listinga

Naučna oblast: matematika
Naučna disciplina: primenjena matematika

ND
Ključne reči: Mašinsko učenje, Algoritmi, Implementacija, Paralelizacija, Klasifikacija, Regresija, Klasterizacija, KNN, K-srednjih vrednosti, Stablo odlučivanja, Random Forest, LASSO

UDK
Čuva se: u biblioteci Departmana za matematiku i informatiku, Prirodno-matematičkog fakulteta, u Novom Sadu

CU
Važna napomena:

VN
Izvod: U ovom radu prikazujemo paralelnu implementaciju nekoliko algoritama za klasifikaciju i klasterizaciju koristeći PajKompas. Za svaki algoritam, opisujemo metodologiju rastavljanja algoritma u manje funkcije koje mogu da se pokrenu i izvršavaju paralelno. Prikazujemo performanse algoritama na nekoliko stvarnih javno dostupnih skupova podataka.

IZ
Datum prihvatanja teme od strane NN veca: 03.09.2018.

DP
Datum odbrane:

DO
Članovi komisije:

KO
Predsednik: dr Srdjan Škrbić, vanredni profesor
Mentor: dr Dušan Jakovetić, docent
Član: dr Dragana Bajović, docent
Accession number:
ANO
Identification number:
INO
Document type: monograph type
DT
Type of record: printed text
TR
Contents code: master thesis
CC
Author: Stevo Racković
AU
Mentor: PhD Dušan Jakovetić
MN
Title: Parallel Implementation of Machine Learning Algorithms Using Py-Compss
XI
Language of text: English
LT
Language of abstract: e
LA
Country of publication: Republic of Serbia
CP
Locality of publication: Backa
LP
Publication year: 2018.
PY
Publisher: author’s reprint
PU
Publ. place: Novi Sad, Trg Dositeja Obradovica 4
PP
Physical description: 9 sections, 78 pages, 23 references, 17 tables, 29 figures,
6 listings
PD
Scientific field: mathematics
SF
Abstract: In this work, we present a parallel implementation of several clustering and classification algorithms using PyCompss. For each algorithm, we describe a methodology of how the algorithm is partitioned into smaller functions that can run in parallel at runtime. We illustrate performance of the algorithms on several real public datasets.