Appendix

Contents

A Software and programming

This project has been developed primarily in `Python` and, in particular, through a `Jupyter Notebook` whose most relevant cells will be shown. We used `Python` because of its versatility and popularity in data science projects. There are numerous packages which are particularly useful in data analysis and in the application of machine learning techniques. The following libraries have been used:

- `NumPy`: It provides a more efficient and versatile handling of vectors, matrices and multidimensional arrays beyond `Python`'s default lists. In addition, it contains numerous functions of all kinds which are compatible with the array format introduced by the library. They are useful in the management of all kinds of data and its handling through linear algebra methods.

  Available at `https://pypi.org/project/numpy`

- `Pandas`: Essential library in data science that offers data structures and operations that greatly facilitate data processing, searching and filtering. For this purpose, it introduces the *DataFrame* data structure that facilitates the grouping of data in indexed tables where we can apply all kinds of functions and transformations.

  Available at `https://pypi.org/project/pandas`

- `matplotlib`: Creation of plots from arrays with numerous plotting functions for histograms, scatter plots, and two- and three-dimensional representations of all kinds with a wide variety of parameters for choosing the appearance of the graphs.

  Available at `https://pypi.org/project/matplotlib`

- `traj-dist`: As a complement to the paper `https://arxiv.org/pdf/1508.04904.pdf`, it allows to evaluate several popular algorithms that measure distances between pairs of trajectories defined by arrays of not necessarily the same length. It includes SSPD, OWD, Hausdorff, Fréchet, discrete Fréchet, DTW, LCSS, ERP and EDR distances.

  Available at `https://pypi.org/project/traj-dist`

- `seaborn`: Another visualization library based on `matplotlib` that provides very illustrative graphs of statistical information. For example, in this project, it has been used to obtain matrices of plots and confusion matrices with heat map format.

  Available at `https://pypi.org/project/seaborn`

- `scikit-learn`: One of the most popular libraries in the world of machine learning that provides some functions related to the design of decision tree models, neural networks and other tools related to model selection, data preprocessing, accuracy and error measurement, model validation, etc.

  Available at `https://pypi.org/project/scikit-learn`

- `TensorFlow`: Another library that provides artificial intelligence functions. It has more functions adapted to the world of *deep learning*. It introduces a particular data format known as *tensor*.

Available at `https://pypi.org/project/tensorflow`

- `Keras`: Neural network library which runs on top of `TensorFlow` with a more user-friendly design.

  Available at `https://pypi.org/project/keras`

- `Yellowbrick`: Library specialized in graphical representations of elements of machine learning and with a particularly suitable implementation of multiclass ROC curves.

  Available at `https://pypi.org/project/yellowbrick`

- `XGBoost`: It provides algorithms based on *gradient boosting* methods. In the case of algorithms based on decision trees, they are relatively popular because they can outperform classical trees and forests in some cases, so it was worth trying their implementation.

  Available at `https://pypi.org/project/xgboost`

We show below some functions imported from the aforementioned libraries for their application throughout the project:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import traj_dist.distance as tdist
import random
import seaborn as sn
from sklearn import tree
from sklearn.datasets import load_iris, make_regression
from sklearn.cluster import KMeans
from sklearn.tree import DecisionTreeClassifier, export_graphviz,
    DecisionTreeRegressor
from sklearn.model_selection import train_test_split, validation_curve, KFold,
    GridSearchCV, cross_validate
from sklearn.metrics import confusion_matrix, accuracy_score,
    mean_absolute_error
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,
    GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.utils import resample
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from yellowbrick.classifier import ROCAUC
from xgboost import XGBClassifier, XGBRegressor
```

Likewise, the visualization and initial processing of the geographically referenced data has been done through the Geographic Information System `QGIS`. The spatial data with which this project has been carried out are in `LineString` and `MultiLineString` formats which are suitable for direct loading into `QGIS`. Therefore, we can simultaneously process hundreds of geographical trajectories, the location of each type of dumpster and other information geographically associated with each work order. The design of a proper integration between `Python` and `QGIS` for

efficient data processing has been an important part of this project. Its relevance will depend on the times when graphical representation of geographical information may be needed and the times when strictly numerical and statistical processing of the data is required.

QGIS is available at https://www.qgis.org/en/site/forusers/download.html

B About data preprocessing

B.1 Data filtering

In order to prepare the data, we must automate the process of reading data, formatting, filtering orders for which complete information (actual, theoretical and dumpsters) is available and discarding duplicate information:

```
def filtra_ordenes(csvteoricas,csvreales,csvcoordenadas):
    dfteoricas=pd.read_csv(csvteoricas,sep=";")
    dfreales=pd.read_csv(csvreales,sep=";")
    dfcoordenadas=pd.read_csv(csvcoordenadas,sep=";")

    dfcoordenadas["Latitude"] = dfcoordenadas["Latitude"].astype(str).str.
    replace(",",".")
    dfcoordenadas["Longitude"] = dfcoordenadas["Longitude"].astype(str).str.
    replace(",",".")

    todasreales = pd.Series(dfreales['WorkOrderId'])
    duplicadas = list(todasreales[todasreales.duplicated()])

    ordenesteoricas=np.unique(dfteoricas['WorkOrderId'])
    ordenesreales = np.array([x for x in dfreales['WorkOrderId'] if x not in
    duplicadas])
    ordenescoordenadas=np.unique(dfcoordenadas['WorkOrderId'])
    ordenes=[ordenesteoricas,ordenesreales,ordenescoordenadas]
    ordenescomunes=np.sort(list(set.intersection(*map(set,ordenes))))

    filtroteorico=dfteoricas[dfteoricas['WorkOrderId'].isin(ordenescomunes)]
    filtroreal=dfreales[dfreales['WorkOrderId'].isin(ordenescomunes)]
    filtrocoord=dfcoordenadas[dfcoordenadas['WorkOrderId'].isin(ordenescomunes)]

    filtrocoordfinal = pd.DataFrame(columns=filtrocoord.columns)

    for orden in ordenescomunes:
        filtrocoordfinal = filtrocoordfinal.append(filtrocoord[filtrocoord['
    WorkOrderId']==orden].drop_duplicates(subset=['ContainerId']),ignore_index=
    True)

    filtroteorico.to_csv('FiltroTeorico_'+csvteoricas,sep=';',index=False)
    filtroreal.to_csv('FiltroReal_'+csvreales,sep=';',index=False)
    filtrocoordfinal.to_csv('FiltroCoord_'+csvcoordenadas,sep=';',index=False)
```

B.2 Creation of variables

Now, we show the main function that has been used to create the variables from the reading of 3 csv files of theoretical routes, actual tracks and dumpster coordinates, the extraction of their information and all the transformations discussed in the project together with the clustering.

```
def clasifica1(cen_x):
    if cen_x==centroides[0]:
        return 0
    if cen_x==centroides[1]:
        return 1
    if cen_x==centroides[2]:
        return 2
    if cen_x==centroides[3]:
        return 3

def clasifica2(Bondad):
    if Bondad>=0 and Bondad<min1:
        return 0
    if Bondad>=min1 and Bondad<min2:
        return 1
    if Bondad>=min2 and Bondad<min3:
        return 2
    if Bondad>=min3:
        return 3

def calcula_variables(csvteoricas,csvreales,csvcoordenadas):

    ordenes = pd.read_csv(csvcoordenadas,sep=';')

    ordenesag = ordenes.groupby('WorkOrderId').agg({
        'Scheduled': 'sum',
        'Collected': 'sum'
    })
    ordenesag['NContenedores']=ordenes.groupby('WorkOrderId').count().iloc[:,2]
    ordenesag['01']=ordenesag['NContenedores']-ordenesag['Scheduled']
    ordenesag['10']=ordenesag['NContenedores']-ordenesag['Collected']
    ordenesag['11']=ordenesag['NContenedores']-ordenesag['01']-ordenesag['10']
    ordenesag=ordenesag.drop(columns=['Scheduled','Collected'])
    ordenesag['Ruido'] = None
    ordenesag['BienHechos']=ordenesag['11']/(ordenesag['11']+ordenesag['10'])
    ordenesag['Similitud'] = None
    ordenesag['Adicionales']=ordenesag['01']/(ordenesag['11']+ordenesag['10']+
    ordenesag['01'])
    ordenesag['RatioLongitudes'] = None

    teoricas = pd.read_csv(csvteoricas,sep=";")
    reales = pd.read_csv(csvreales,sep=";")

    for j in range(len(reales['WorkOrderId'])):
        wktr = reales['WKT'].iloc[j].replace("LINESTRING(" , "").replace(")","")
    .replace(","," ").split()
        wktreales = []
```

```
46          for i in range(0,len(wktr),2):
47              wktreales.append([float(wktr[i]),float(wktr[i+1])])
48          longitudes_r = np.sqrt(np.sum(np.diff(np.array(wktreales), axis=0)**2,
    axis=1))
49          longitud_real = np.sum(longitudes_r)
50
51          wktt = teoricas['WKT'].iloc[j].replace("MULTILINESTRING((" , "").replace
    ("))","").replace(","," ").split()
52          wktteoricas = []
53          for i in range(0,len(wktt),2):
54              wktteoricas.append([float(wktt[i]),float(wktt[i+1])])
55          longitudes_t = np.sqrt(np.sum(np.diff(np.array(wktteoricas), axis=0)**2,
     axis=1))
56          longitud_teorica = np.sum(longitudes_t)
57
58          ratio = np.abs(longitud_real-longitud_teorica)/longitud_teorica
59          ordenesag['RatioLongitudes'].iloc[j]=ratio
60
61          distancia = tdist.sspd(np.array(wktreales),np.array(wktteoricas),'
    euclidean')
62          ordenesag['Similitud'].iloc[j]=distancia
63
64          ordenesag['Ruido'].iloc[j]=np.random.normal(0,0.7)
65
66      ordenesag['Similitud']=0.001/(0.001+ordenesag['Similitud'])
67
68      ordenesag['RatioLongitudes']=(ordenesag['RatioLongitudes']-ordenesag['
    RatioLongitudes'].min())/(ordenesag['RatioLongitudes'].max()-ordenesag['
    RatioLongitudes'].min())
69
70      ordenesag['Bondad']=10*ordenesag['BienHechos']+4*ordenesag['Similitud']+2*
    ordenesag['Adicionales']-1*ordenesag['RatioLongitudes']+ordenesag['Ruido']
71
72      ordenesag['Bondad']=10*(ordenesag['Bondad']-ordenesag['Bondad'].min())/(
    ordenesag['Bondad'].max()-ordenesag['Bondad'].min())
73
74      mezcla = ordenesag.copy()
75
76      kmeans = KMeans(n_clusters=4, random_state=0)
77      mezcla['cluster'] = kmeans.fit_predict(mezcla[['Bondad']])
78      centroids = kmeans.cluster_centers_
79      cen_x = [i[0] for i in centroids]
80      mezcla['cen_x'] = mezcla.cluster.map({0:cen_x[0], 1:cen_x[1], 2:cen_x[2], 3:
    cen_x[3]})
81
82      global centroides
83
84      centroides = np.sort(np.unique(mezcla['cen_x']))
85      colors = ['#DF2020', '#81DF20', '#2095DF', '#20DF95']
86      mezcla['c'] = mezcla.cluster.map({0:colors[0], 1:colors[1], 2:colors[2], 3:
    colors[3]})
87      plt.scatter(mezcla.Bondad,mezcla.Bondad, c=mezcla.c, alpha = 0.6, s=10)
88      ordenesag['Resultado']=mezcla['cen_x'].apply(clasifica1)
89
90      global min0
```

```
    global min1
    global min2
    global min3

    min0 = min(ordenesag[ordenesag['Resultado']==0]['Bondad'])
    min1 = min(ordenesag[ordenesag['Resultado']==1]['Bondad'])
    min2 = min(ordenesag[ordenesag['Resultado']==2]['Bondad'])
    min3 = min(ordenesag[ordenesag['Resultado']==3]['Bondad'])

    print(min0)
    print(min1)
    print(min2)
    print(min3)

    ordenesag['Bondad']=ordenesag['Bondad'].astype(float).round(0)

    ordenesag['Resultado']=ordenesag['Bondad'].apply(clasifica2)

    return ordenesag
```

C About the similarity of trajectories

C.1 Densification of trajectories

In this project, SSPD has been found to be the optimal algorithm for the calculation of the degree of similarity of the available pairs of trajectories in our dataset. Further improvement in the accuracy of the distances can be achieved by densifying each of the trajectories with more points. These trajectories consist of a sequence of points connected by straight lines, but if we want to fill these lines with more equispaced points to reconstruct the trajectories more densely, the following function can be used:

```
def densificar(coordenadas, segmentos):
    denso=coordenadas
    for i in range(len(coordenadas)-1):
        denso=np.insert(denso,i*segmentos+1,np.array([list(a) for a in zip(np.
    linspace(coordenadas[i][0],coordenadas[i+1][0],segmentos,endpoint=False)
    [1:],np.linspace(coordenadas[i][1],coordenadas[i+1][1],segmentos,endpoint=
    False)[1:])]),0)
    return denso
```

where `coordenadas` is the array of points that make up a trajectory and `segmentos` is the number of segments we want to have between two consecutive points of the original array (i.e., there will be `segmentos`-1 new points between each pair of original consecutive points).

C.2 Alternative measures of the degree of similarity

Until we observed the suitability of the SSPD, we analyzed numerous popular algorithms that would work well on other specific problems. We can classify these algorithms into two types:

- *Warping-based*: These are distances that take into account the time indexing of each point. For example, DTW, LCSS, EDR and ERP distances.
- *Shape-based*: These are distances that only take into account the geometrical shape of the trajectories. For example, Hausdorff, Fréchet and SSPD distances.

Let us start by characterizing the four aforementioned warping-based algorithms:

- ***Dynamic Time Warping*** **(DTW)**
- ***Longest Common SubSequence*** **(LCSS)**
- ***Edit Distance on Real sequence*** **(EDR)**
- ***Edit distance with Real Penalty*** **(ERP)**

In the following mathematical descriptions, we want to obtain the distance between two trajectories T^i and T^j where n^i is the number of points of T^i and n^j is the number of points of T^j. The k-th point of trajectory T^i is p_k^i and the k-th point of trajectory T^j is p_k^j. Likewise, $rest(T^i)$ (respectively, $rest(T^j)$) is the trajectory T^i (respectively, T^j), but without its first point. Also, LCSS and EDR require the specification of a spatial threshold ε_d and ERP takes a paremeter g as a reference value to penalize *gaps* (which appear when there are points that are not matched with any other point).

The definition of the 4 algorithms is summarized in the following table:

NAME	Cost function $\delta_{NAME}(p_1,p_2)$ =	Distance $NAME(T^i,T^j)$ =
DTW	$\|p_1p_2\|_2$	$= \begin{cases} 0 & \text{if } n^i = n^j = 0 \\ \infty & \text{if } n^i = 0 \text{ or } n^j = 0 \\ \delta_{DTW}(p_1^i,p_1^j) + \min \begin{Bmatrix} DTW(rest(T^i),rest(T^j)), \\ DTW(rest(T^i),T^j)), \\ DTW(T^i,rest(T^j)) \end{Bmatrix} & \text{otherwise} \end{cases}$
LCSS	$\begin{cases} 1 & \text{if } \|p_1p_2\|_2 < \varepsilon_d \\ 0 & \text{if } p_1 \text{ or } p_2 \text{ is a gap} \\ 0 & \text{otherwise} \end{cases}$	$= \begin{cases} 0 & \text{if } n^i = 0 \text{ or } n^j = 0 \\ LCSS(rest(T^i),rest(T^j)) + \delta_{LCSS}(p_1^i,p_1^j) & \text{if } \delta_{LCSS}(p_1^i,p_1^j) = 1 \\ \max \begin{Bmatrix} LCSS(rest(T^i),T^j)) + \delta_{LCSS}(p_1^i,gap), \\ LCSS(T^i,rest(T^j)) + \delta_{LCSS}(gap,p_1^j) \end{Bmatrix} & \text{otherwise} \end{cases}$
EDR	$\begin{cases} 0 & \text{if } \|p_1p_2\|_2 < \varepsilon_d \\ 1 & \text{if } p_1 \text{ or } p_2 \text{ is a gap} \\ 1 & \text{otherwise} \end{cases}$	$= \begin{cases} n^i & \text{if } n^j = 0 \\ n^j & \text{if } n^i = 0 \\ EDR(rest(T^i),rest(T^j)) & \text{if } \delta_{EDR}(p_1^i,p_1^j) = 0 \\ \min \begin{Bmatrix} EDR(rest(T^i),rest(T^j)) + \delta_{EDR}(p_1^i,p_1^j), \\ EDR(rest(T^i),T^j)) + \delta_{EDR}(p_1^i,gap), \\ EDR(T^i,rest(T^j)) + \delta_{EDR}(gap,p_1^j) \end{Bmatrix} & \text{otherwise} \end{cases}$
ERP	$\begin{cases} \|p_1p_2\|_2 & \text{if } p_1, p_2 \text{ are not gaps} \\ \|p_1g\|_2 & \text{if } p_2 \text{ is a gap} \\ \|gp_2\|_2 & \text{if } p_1 \text{ is a gap} \end{cases}$	$= \begin{cases} \sum_{k=1}^{n^i} \|p_k^i g\|_2 & \text{if } n^j = 0 \\ \sum_{l=1}^{n^j} \|p_l^j g\|_2 & \text{if } n^i = 0 \\ \min \begin{Bmatrix} ERP(rest(T^i),rest(T^j)) + \delta_{ERP}(p_1^i,p_1^j), \\ ERP(rest(T^i),T^j)) + \delta_{ERP}(p_1^i,gap), \\ ERP(T^i,rest(T^j)) + \delta_{ERP}(gap,p_1^j) \end{Bmatrix} & \text{otherwise} \end{cases}$

Table 1: Four warping-based distances: DTW, LCSS, EDR and ERP. On the left, cost functions δ that we define to use them in the definitions of the distances on the right.

For our particular problem, the warping-based algorithms would yield unreliable results because they would attempt to match the points of the two trajectories according to their ordering in time.

Since the variable we wish to obtain is the degree of similarity between two paths (the theoretical one and the actual one) based on their shapes, it seems logical to study shape-based algorithms such as those described below:

- **Hausdorff distance**:

$$D_{Haus}(T^1, T^2) = \max \left\{ \max_{\substack{i_1 \in [1,\dots,n^1] \\ j_2 \in [1,\dots,n^2-1]}} \left\{ D_{ps}\left(p^1_{i_1}, s^2_{j_2}\right) \right\}, \max_{\substack{j_1 \in [1,\dots,n^1-1] \\ i_2 \in [1,\dots,n^2]}} \left\{ D_{ps}\left(p^2_{i_2}, s^1_{j_1}\right) \right\} \right\} \quad (1)$$

  where $D_{ps}\left(p^1_{i_1}, s^2_{j_2}\right)$ is the distance from point $p^1_{i_1}$ to segment $s^2_{j_2}$ and $D_{ps}\left(p^2_{i_2}, s^1_{j_1}\right)$ is the distance from point $p^2_{i_2}$ to segment $s^1_{j_1}$.

- **Fréchet distance**:

  Given two curves A and B, we define this distance as the infimum over all reparametrizations α and β of $[0,1]$ of the maximum over all $t \in [0,1]$ of the distance between $A(\alpha(t))$ and $B(\beta(t))$:

$$D_{Frec}(A, B) = \inf_{\alpha,\beta} \max_{t \in [0,1]} \{d\left(A(\alpha(t)), B(\beta(t))\right)\} \quad (2)$$

  Alternatively, there exists a discrete version of this algorithm.

- ***Symmetrized Segment-Path Distance*** **(SSPD)**:

  The one described in the main project and the one that proved to provide the most accurate results and with theoretical arguments to believe in its effectiveness for our specific problem.

  Since in our case we are looking for normalized variables to avoid scale sensitivity problems in decision trees and neural networks, it will only be necessary to set a certain parameter that guarantees an adequate transformation. This is described in the next subsection.

C.3 Determination of the SSPD normalization parameter

It is essential to send the value of all input variables to a bounded interval to avoid the problem of scale sensitivity that appears in some machine learning models. If the SSPD distance is a number between zero and infinity, the most accurate transformation given the characteristics of the problem was found to be

$$\frac{\xi}{\xi + SSPD} \quad (3)$$

This transformation is monotonically decreasing, so it reverses the ordering of the original distances without mixing them and will assign a high degree of similarity to a low distance, and vice versa. The latter will occur only if the parameter ξ is properly chosen because a bad choice in order of magnitude causes the values of the degree of similarity to accumulate at one end of the interval $(0, 1)$. The choice of the optimal parameter can be made by studying the abundance of distance data in each order of magnitude. Using the Euclidean distance, we observe that there are distances in the orders of magnitude 10^{-5}, 10^{-4}, 10^{-3} and 10^{-2}. In particular, among the 623 work orders analyzed, we have the following distribution:

Order of magnitude of the distance	**Amount of work orders**
10^{-2}	33
10^{-3}	169
10^{-4}	363
10^{-5}	58

Table 2: Number of work orders according to the order of magnitude of their associated SSPD.

Recall that the smaller this distance is, the more similar two trajectories are. We perform a qualitative inspection of the similarity of the pairs of geometrical trajectories of work orders which have distances in each of the 4 orders of magnitude described. Finally, we may say that these 4 orders of magnitude can be linked to the 4 classes we have defined for training the classification models.

That is, a distance in the order 10^{-2} would be a ***bad*** distance, a distance in the order 10^{-3} would be a ***mediocre*** distance, a distance in the order 10^{-4} would be a ***acceptable*** distance and a distance in the order 10^{-5} would be a ***very good*** distance. Of course, as we saw in the main project, this is not enough to classify work orders because of the influence of the other variables with different weights. Nevertheless, it tells us that about 2/3 of the orders have pairs of GPS tracks with a good similarity. Let us find the parameter ξ of the transformation (3) that will appropriately send these distances to the interval (0,1). Suppose that all distances in the orders 10^{-2} and 10^{-3} (in total, 202) must have a degree of similarity below 0.5 and all distances in the orders 10^{-4} and 10^{-5} (in total, 421) must have a degree of similarity above 0.5. The parameter that allows this arrangement of the values is $\xi = 0.001$. To confirm that transformation (3) is actually realistic, we may fix manually a few reference values according to our intuition in certain work orders to see that our transformation is able to predict degrees of similarity that a person would assign without a computer. If we take a set of 10 work orders with sufficiently diverse distances, we can check that the degree of similarity that a person would assign manually to a pair of trajectories is close enough to the value predicted by the function (3) with $\xi = 0.001$.

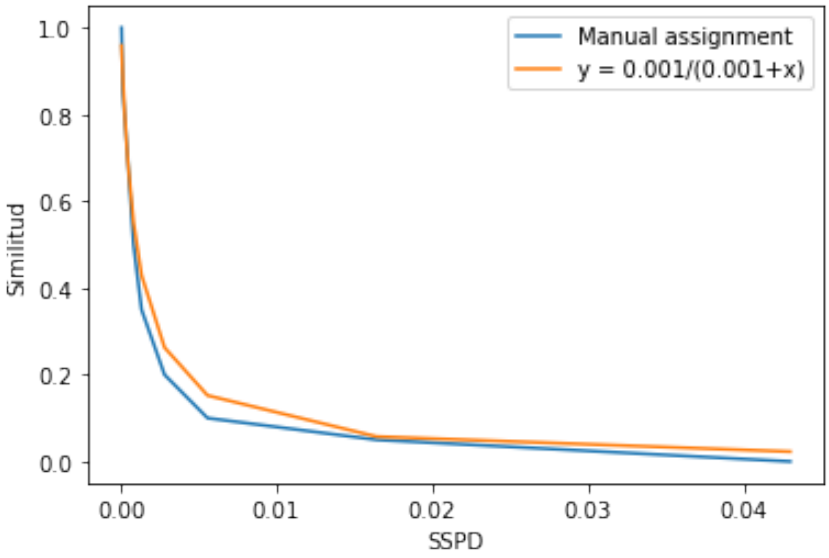

Figure 1: Degree of similarity versus SSPD distance.

Therefore, we have managed to fit a magnitude that seemed totally subjective or very difficult to calculate with enough precision and now it can give objective results.

D About decision trees

The following subsections are a brief extension of the analysis related to decision trees that was carried out in the project.

D.1 Depth of a tree

The project has emphasized that one of the great advantages of decision trees is their explainability compared to other models with more sophisticated algorithms that are less interpretable. A quick glance at a decision tree gives us a very visual idea of how to classify datasets. However, this is only possible when the tree is properly pruned. In the project, we show the graph of a tree that has grown up to the third generation (i.e., three rows plus the root node). This provides an easy interpretation in terms of two of the variables in the model and the accuracy of the model can be considered moderately good. However, if we allowed the growth of the tree until all leaf nodes had zero Gini impurity, we would obtain the following tree:

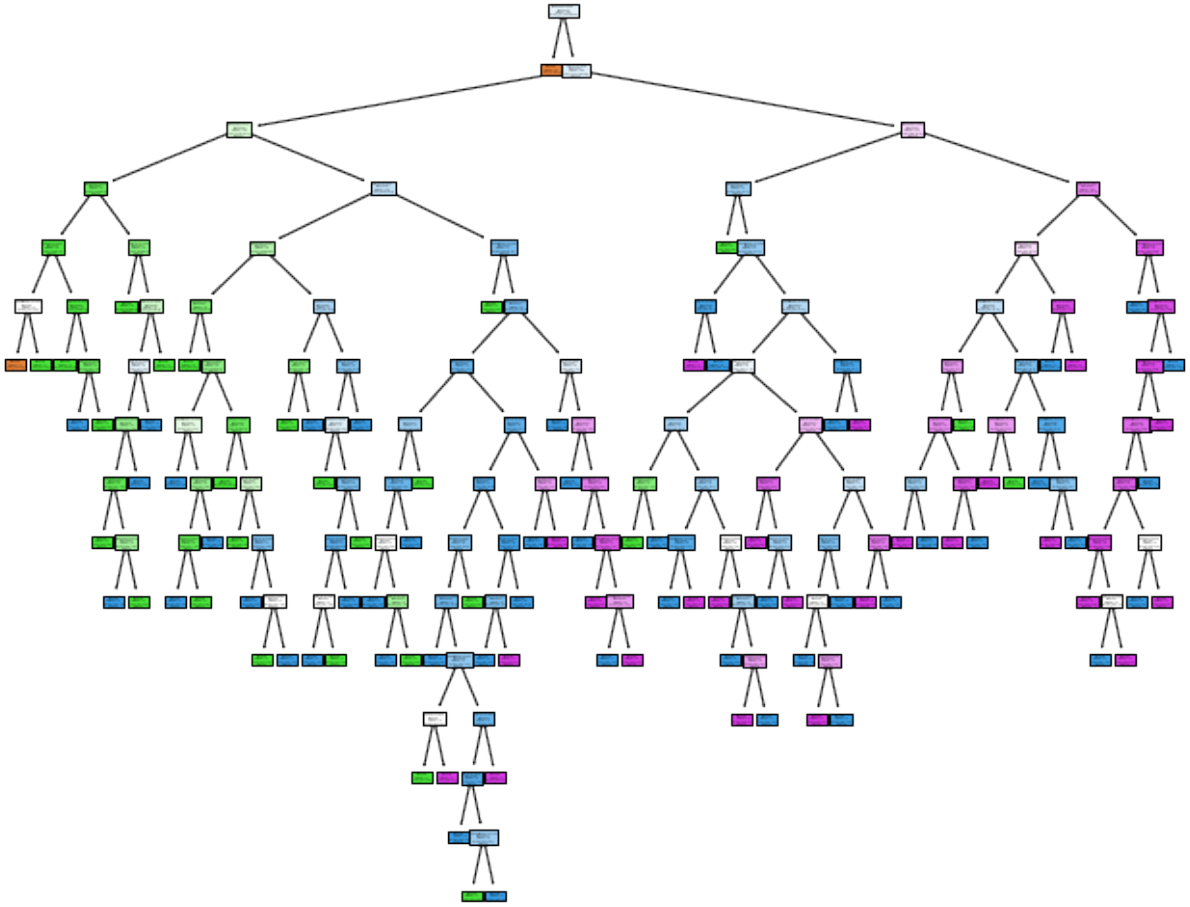

Figure 2: Decision tree of the main project if it had not been pruned.

It is not intended to make each node readable. The point is that the readability is totally lost. It would be very costly to extract all the conditions that are deduced through the logical propositions associated with such a large number of nodes. Moreover, we would not necessarily obtain a more accurate model because a tree that is too large may overfit the data. That is, it would learn the patterns of the training dataset too well, but would generalize poorly and would fail to make right predictions in any new dataset. Thus, this tree would be of no practical use later on.

D.2 Variance in trees

Decision trees are not unique for each dataset. Depending on the specific piece of data that we choose to train and to test, we can generate slightly different trees. In general, for a sufficiently large dataset we should not expect extremely high variance, but there may be specific cases that give unexpected results.

For example, for our dataset, the form of a typical decision tree is the one presented in the main project, which depends on the variables `BienHechos` and `Similitud`:

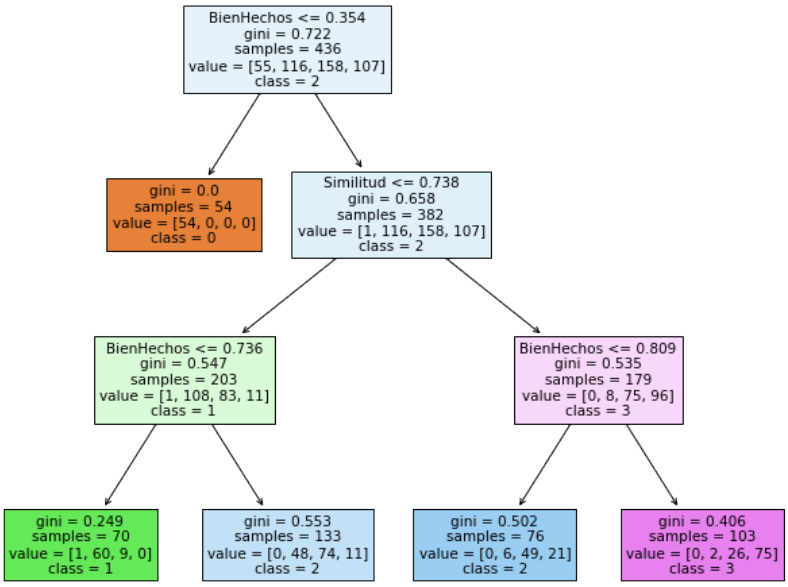

Figure 3: Main decision tree.

But in some specific cases, we may get a decision tree like the following:

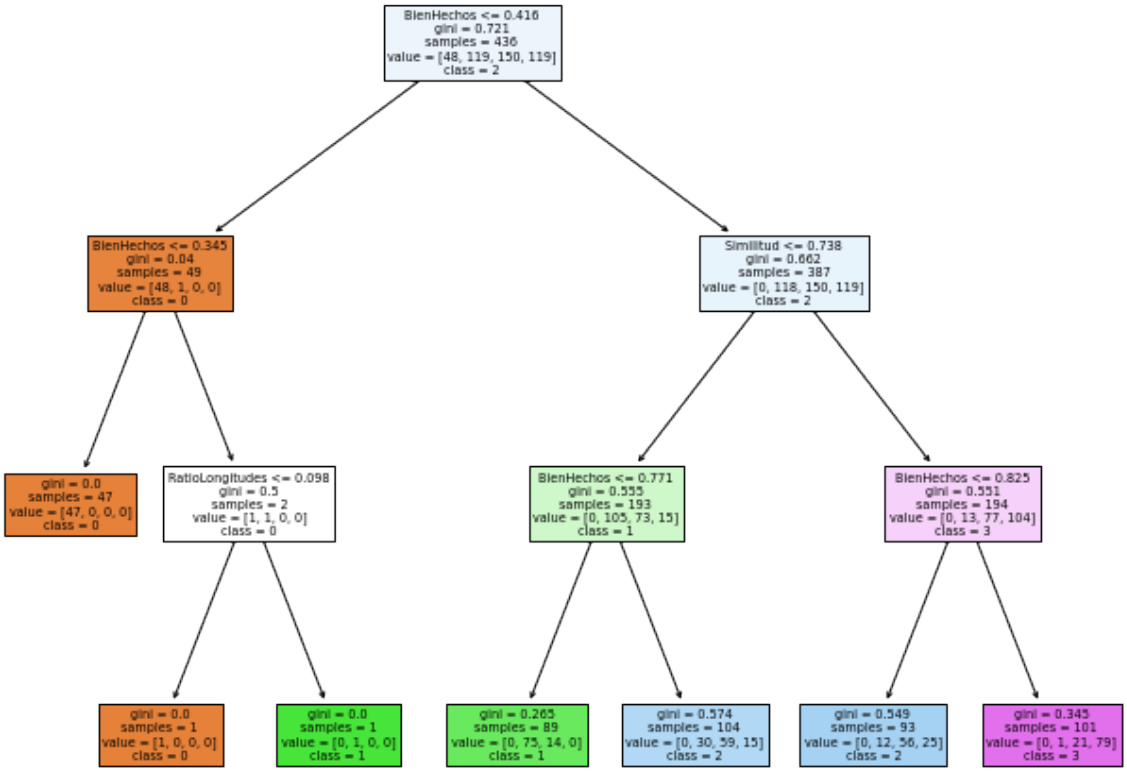

Figure 4: Alternative decision tree.

where the algorithm considered that it was relevant to include a third variable: `RatioLongitudes`. In this case, we see that it has used it to differentiate between two specific samples, and not even correctly because the model must assign a higher class to pairs of theoretical routes and actual tracks with a lower value of `RatioLongitudes`. This problem can happen in specific situations, as well as small numerical variations in the decision thresholds of the nodes depending on the tree, even if they have the same shape. Therefore, the study of decision trees is important if it can be shown that the shape of the trees is reasonably stable for a dataset and if we want to have a visual idea of what a typical classification would look like. However, if we want to achieve a higher accuracy, we need to try more elaborate models.

D.3 Selection of random forest parameters

A random forest takes into account the predictions of multiple decision trees. There is no standard formula to determine in advance the optimal number of trees, but we can evaluate the performance of the model depending on the number of trees:

```
scores = []
for k in range(1, 100):
    rfc = RandomForestClassifier(n_estimators=k,max_depth=3,n_jobs=-1,
    random_state=42)
    rfc.fit(X_train, y_train)
    y_pred = rfc.predict(X_test)
    scores.append(accuracy_score(y_test, y_pred))
```

In our case, we obtain the following result:

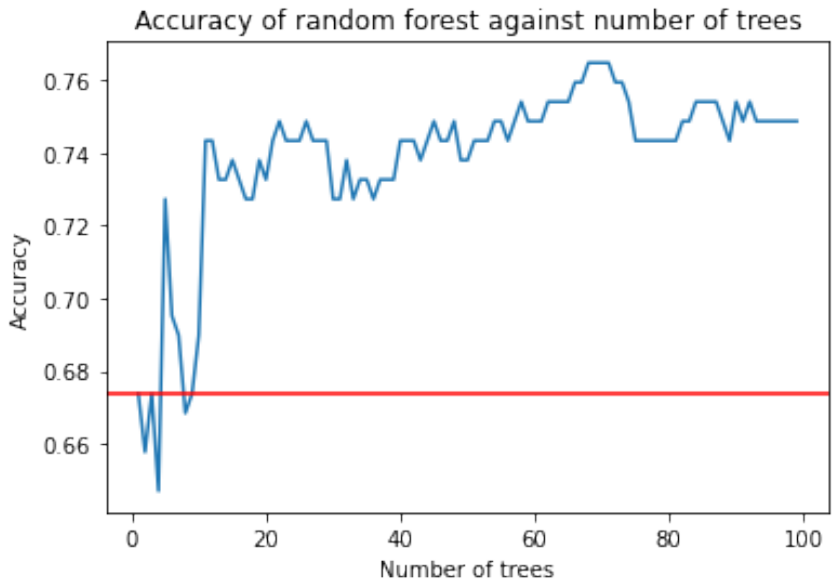

Figure 5: In blue, accuracy of the random forest versus the number of decision trees (estimators). In red, accuracy of the individual decision tree shown in the project.

We see that there are high variations in accuracy when the number of trees is low. As the number increases, the accuracy begins to stabilize in a range of values higher than the accuracy that we had achieved with a single decision tree. As we can see, the accuracy is maximized when we take 70 trees, which is the number chosen in the project. However, as we saw later with bootstrapping, the actual capacity of the model is less than the one achieved with 70 trees. Another metric commonly used to determine the number of trees and evaluate performance is the *out-of-bag* (OOB) error which should decrease as the number of trees increases.

With a larger dataset, we could further narrow the range of accuracies in which the model oscillates for a large number of trees. We could probably notice, in general, a more monotonic increase except for small ups and downs due to randomness.

D.4 Decision surfaces of the random forest

It is not useful to represent 70 individual decision trees, so we say that a random forest is less interpretable than a decision tree. However, we can take the variables of any model in pairs and represent decision surfaces. This is a type of representation that allows us to visualize how a classification model divides the space of variables according to its predictions and superficially analyze its strengths and weaknesses. It consists of training a random forest using pairs of variables and, for each pair, finding the boundaries that separate the 4 classes according to the thresholds decided by the forest.

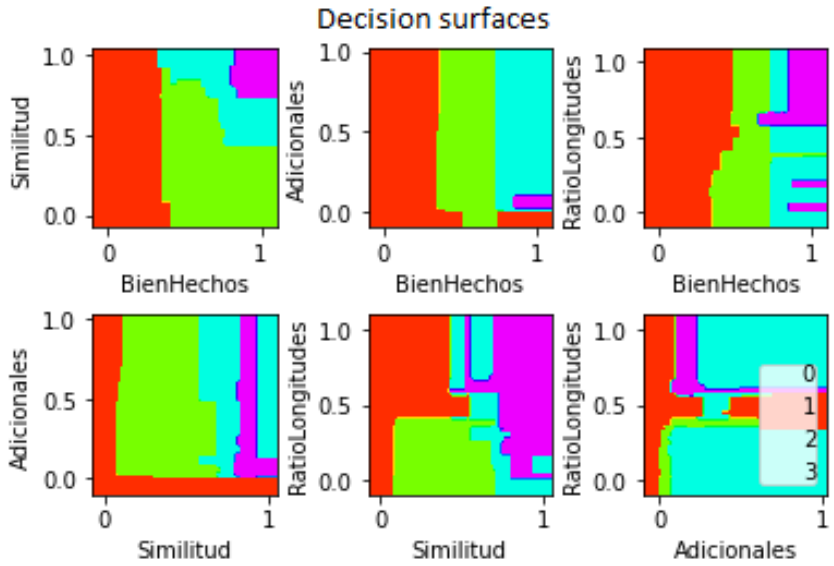

Figure 6: Decision surfaces learned by training random forests with two variables. In orange, class 0. In light green, class 1. In cyan, class 2. In fuchsia, class 3.

After that, we plot the data that we used for training and we observe the correspondences.

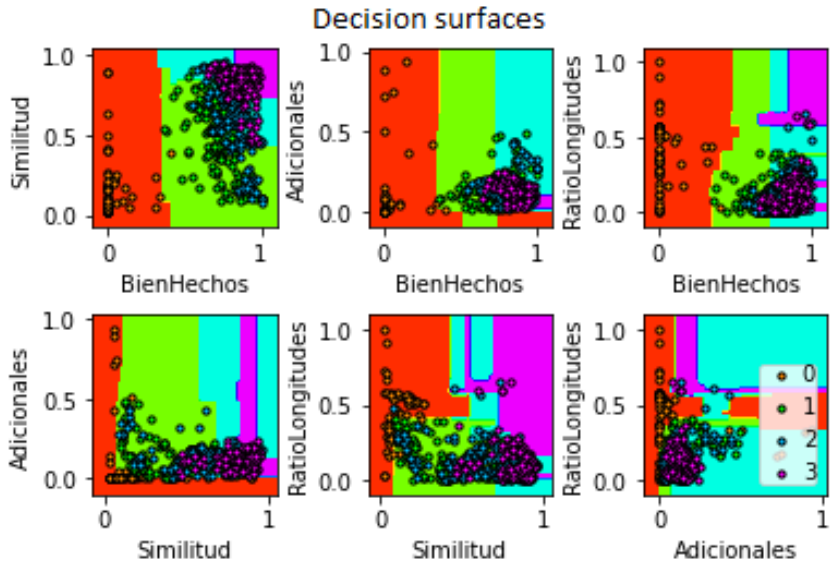

Figure 7: Same surfaces, but with the points of the dataset with the appropriate color according to the class to which we know they belong.

Surfaces are an attempt to distinguish continuous areas that have the same class, with varying degrees of success. Given the complexity of the dataset and the fact that points of the same class are not completely separated from those of another class, some of the surfaces take a complex form.

First of all, it should be noted that the surfaces are drawn in $[0,1] \times [0,1]$ squares since all variables lie in the interval $[0,1]$, but we do not necessarily have data in all areas of a square. In fact, in many cases, we see that the vast majority of the data is clustered in very specific areas (corners and edges). The algorithm makes an attempt to extrapolate the surfaces to areas where there are no points, without these predictions necessarily being correct according to our intuition.

In the square on the top left we see 4 continuous surfaces which are quite well defined. It is expected that, the higher the similarity and the higher the proportion of well-collected dumpsters, the higher the goodness of the order and the order will belong to a higher class, saving class 3 (***very good*** orders) to the upper right corner of the square. Similarly, we would expect ***bad*** orders to be in the lower left corner or, in this case, in the entire left side due to the dominance that the variable `BienHechos` has over the variable `Similitud`. Not all points are located on the surface that should correspond to them, but the main ideas have been captured.

Let us look at the upper center square. Again, we see the dominance of the variable `BienHechos` because of the verticality of the surfaces. However, we see that class 3 is saved for a small piece of the lower right corner of the square. This means that the model has captured the idea that, even if goodness increases with a high value of the variable `Adicionales`, class 3 orders will actually have a low value of this variable. This is because, in general, work orders with a high value of the variable `BienHechos` due to strict order compliance tend to divert little from the planning and do not collect many additional dumpsters.

In the other cases, the information is more chaotic and unreliable. It is clear in some cases that training with two variables is insufficient. Especially, when dealing with variables with a

lower contribution to the total goodness as in the case of the bottom right square, the information on the surfaces is much more imprecise and the points are not located where they should. This analysis simply gives us a first superficial idea of the nature of the variables.

D.5 Extreme Gradient Boosting

Within the *ensemble* methods, we find two groups: parallel learning techniques and sequential learning techniques. A clear example of the first type is random forests that take a group of independent trees and reduce the error by taking into account the performance of multiple trees. However, techniques of the second type are also worth studying. In the case of working with trees, the trained trees would be generated in a sequence with some dependence on each other so that the new generated trees learn from the errors of the previous trees.

Gradient boosting algorithms belong to the second type and they are a very popular classification and regression technique. In principle, *extreme gradient boosting* algorithms from the `XGBoost` library are better optimized and regularized for the prevention of model overfitting.

For the purposes of this project, the implementation of the classifier `XGBClassifier` and the regressor `XGBRegressor` has been tested. However, it has been observed that, for this dataset, the results are essentially equal to those of the random forest without providing information of greater relevance, so they have not been developed further in the project. Perhaps, for larger and more complex datasets, one would expect noticeable improvements in performance.

E About neural networks

A schematic representation of the form of one of these networks could be the following:

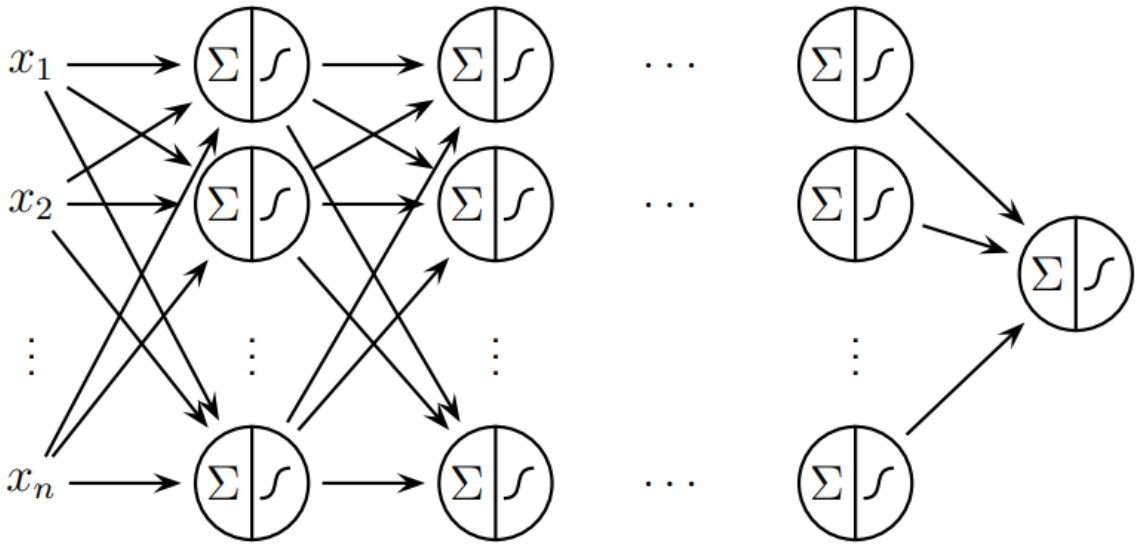

Figure 8: Multilayer perceptron.

where we see the two steps that occur in each neuron: performing a linear combination and applying a nonlinear function.

We now decribe some fundamental algorithms that determine how neural networks learn.

E.1 Gradient descent

Gradient descent is an iterative optimization algorithm for finding a local minimum of a function. This is done by going in the direction of maximum decrease which is given by the negative gradient at each point. Starting at a point, we calculate the gradient at that point, change its sign and, with a certain step, go to the next point where we repeat the process. If a suitable step has been chosen for the function, we will eventually get close enough to a local minimum of the function. The step size depends on a parameter known as *learning rate* (α) which multiplies the gradient. It must be chosen carefully because a really large value may cause the algorithm to exceed the local minimum without reaching it, whereas a really low value may require an extremely long computation time.

As usual, we define a cost function J where we will apply the gradients (vectors of derivatives with respect to the network parameters) since we want to minimize it. Thus, at each iteration, the value of a set of parameters θ as a function of the value of those of the previous iteration is given by

$$\theta_{i+1} = \theta_i - \alpha \nabla J(\theta_i) \tag{4}$$

In machine learning, we use this algorithm to update the parameters of a model. For example, the weights of a neural network.

E.2 Backpropagation

The calculation of the gradients described in the previous method is not trivial due to the high number of parameters that can exist in a neural network and their distribution in multiple layers. Therefore, we use a method known as *backpropagation*. We calculate the partial derivatives of the cost function with respect to the parameters of the last layer by applying the chain rule. Then, we apply the same method to the previous layers one by one until we reach the beginning of the network.

To simplify the explanation of the concept, suppose we have a network with only one neuron in each layer. As described in the project, there are two operations in each neuron:

$$z^{(n)} = w^{(n)} a^{(n-1)} + b^{(n)} \tag{5}$$

$$a^{(n)} = \phi^{(n)} \left(z^{(n)} \right) \tag{6}$$

Suppose that, in the gradient descent method, we need to calculate the derivative of the cost function with respect to the weight $w^{(n)}$. Then, we simply apply the chain rule:

$$\frac{\partial J_k}{\partial w^{(n)}} = \frac{\partial z^{(n)}}{\partial w^{(n)}} \cdot \frac{\partial a^{(n)}}{\partial z^{(n)}} \cdot \frac{\partial J_k}{\partial a^{(n)}} \tag{7}$$

Now, we could calculate each of these three derivatives. Assume that the cost function associated with a given sample of the training set is given by the quadratic difference between the activation value a in the n-th layer which will be just before the output layer and the desired value in the output neuron for that specific sample of the dataset, y_k. That is,

$$J_k = \left(a^{(n)} - y_k\right)^2 \tag{8}$$

Then, from the expressions (5), (6) and (8), we calculate the three derivatives, which are the following:

$$\frac{\partial z^{(n)}}{\partial w^{(n)}} = a^{(n-1)} \tag{9}$$

$$\frac{\partial a^{(n)}}{\partial z^{(n)}} = \left(\phi^{(n)}\right)' \left(z^{(n)}\right) \tag{10}$$

$$\frac{\partial J_k}{\partial a^{(n)}} = 2\left(a^{(n)} - y_k\right) \tag{11}$$

Once this process is performed for a certain sample of the dataset, the partial derivative of the total cost function would be the average of all the partial derivatives obtained for all the samples of the dataset.

Equivalently, the partial derivative with respect to the bias $b^{(n)}$ is

$$\frac{\partial J_k}{\partial b^{(n)}} = \frac{\partial z^{(n)}}{\partial b^{(n)}} \cdot \frac{\partial a^{(n)}}{\partial z^{(n)}} \cdot \frac{\partial J_k}{\partial a^{(n)}} \tag{12}$$

where

$$\frac{\partial z^{(n)}}{\partial b^{(n)}} = 1 \tag{13}$$

Again, once this process is performed for a certain sample of the dataset, the partial derivative of the total cost function would be the average of all the partial derivatives obtained for all the samples of the dataset.

For networks with more neurons in each layer, the process is analogous taking into account the multiple extra weights that appear and the different activations that appear in the same layer.

E.3 L-BFGS

`L-BFGS` (*Limited-memory Broyden–Fletcher–Goldfarb–Shanno*) is an optimization algorithm used in neural networks. It is a second order method. It belongs to the family of quasi-Newton methods. This means that we make approximations of the inverse of the Hessian matrix instead of its explicit computation. It is an improvement of the `BFGS` algorithm as it requires less information

to be stored in memory at each iteration, which is relevant in terms of efficiency. In principle, for small datasets, we can expect a better performance with this solver than with others typically used in the training of neural networks such as stochastic gradient descent or `Adam`.

E.4 Adam

This is a modification of classical stochastic gradient descent that combines the advantages of other popular stochastic gradient descent extensions such as `AdaGrad` or `RMSprop`. Adaptive *learning rates* are computed for the different parameters from estimates of the first and second moments of the gradients (assuming that the gradients are random variables). Essentially, it has 4 hyperparameters of interest:

- α: Step size.
- β_1: Exponential decay rate for the first moment estimates.
- β_2: Exponential decay rate for the second moment estimates.
- ε: Tolerance to prevent divisions by zero.

It is one of the most efficient methods and it is especially suitable for large problems in terms of amount of data and number of parameters.

E.5 Implementation of the neural networks

To create a neural network with the `L-BFGS` solver while comparing performance for different parameter combinations, the following code can be used:

```
X = variables[['BienHechos','Similitud','Adicionales','RatioLongitudes']]
y = variables['Resultado']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

mlp = MLPClassifier()
parameter_space = {
    'hidden_layer_sizes': [(4,),(5,),(6,)],
    'activation': ['tanh','logistic','relu'],
    'solver': ['lbfgs'],
    'alpha': [0.01,0.001,0.0001,0.00001],
    'max_iter': [1000],
    'learning_rate': ['constant'],
}
clf = GridSearchCV(mlp, parameter_space, n_jobs=-1, cv=3)
clf.fit(X_train, y_train)

print('Mejores parametros:\n', clf.best_params_)
```

where we can perform a parameter search with `GridSearchCV` which already applies cross validation using `K-Fold`. This method performs K distinct partitions of the total dataset into training and test subsets to evaluate the accuracy according to each of the partitions. Then, the results are averaged.

Likewise, the implementation of `Adam` in `TensorFlow` is carried out through a proper handling of the library's tensor class:

```
def get_batch(x_data, y_data, batch_size):
    idxs = np.random.randint(0, len(y_data), batch_size)
    return x_data[idxs,:], y_data[idxs]

def nn_model(x_input, W1, b1, W2, b2):
    x_input = tf.reshape(x_input, (x_input.shape[0], -1))
    x = tf.add(tf.matmul(tf.cast(x_input, tf.float32), W1), b1)
    x = tf.nn.relu(x)
    logits = tf.add(tf.matmul(x, W2), b2)
    return logits

def loss_fn(logits, labels):
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    labels=labels,

    logits=logits))
    return cross_entropy

epochs = 80
batch_size = 50

X = variables[['BienHechos','Similitud','Adicionales','RatioLongitudes']]
y = variables['Resultado']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

X_train = tf.Variable(np.array(X_train).astype('float'))
X_test = tf.Variable(np.array(X_test).astype('float'))

W1 = tf.Variable(tf.random.normal([4, 4], stddev=0.03), name='W1')

b1 = tf.Variable(tf.random.normal([4]), name='b1')

W2 = tf.Variable(tf.random.normal([4, 4], stddev=0.03), name='W2')

b2 = tf.Variable(tf.random.normal([4]), name='b2')

optimizer = tf.keras.optimizers.Adam(learning_rate=0.07,beta_1=0.9,beta_2=0.999,
    epsilon=0.0000001)

total_batch = int(len(y_train) / batch_size)
arrayloss=[]
arrayacc=[]
for epoch in range(epochs):
```

```
    avg_loss = 0
    for i in range(total_batch):
        batch_x, batch_y = get_batch(np.array(X_train), np.array(y_train),
batch_size=batch_size)

        batch_x = tf.Variable(np.array(batch_x).astype('float'))
        batch_y = tf.Variable(np.array(batch_y).astype('float'))

        batch_y = tf.one_hot(np.array(batch_y).astype('int'),4)
        with tf.GradientTape() as tape:
            logits = nn_model(batch_x, W1, b1, W2, b2)
            loss = loss_fn(logits, batch_y)
        gradients = tape.gradient(loss, [W1, b1, W2, b2])
        optimizer.apply_gradients(zip(gradients, [W1, b1, W2, b2]))
        avg_loss += loss / total_batch
    arrayloss.append(avg_loss)
    test_logits = nn_model(X_test, W1, b1, W2, b2)
    max_idxs = tf.argmax(test_logits, axis=1)
    test_acc = np.sum(max_idxs.numpy() == y_test) / len(y_test)
    arrayacc.append(test_acc)
    print(f"Epoca: {epoch + 1}, perdida={avg_loss:.3f}, precision={test_acc
*100:.3f}%")
```

Additional references

The main bibliography has been presented in the main project, but here we have some references of interest about the concepts in this appendix.

[1] Bonde, O. and Karlsson, L. (2020). *A Comparison of Selected Optimization Methods for Neural Networks.*
https://www.diva-portal.org/smash/get/diva2:1438308/FULLTEXT01.pdf

[2] Brownlee, J. (2017). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.*
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning

[3] Chen, T. and Guestrin, C. (2016). *XGBoost: A Scalable Tree Boosting System.*
https://arxiv.org/pdf/1603.02754.pdf

[4] Pandey, P. (2019). *Understanding the Mathematics behind Gradient Descent.*
https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

[5] Sanderson, G. (2017). *Backpropagation calculus — Deep learning, chapter 4.*
https://www.youtube.com/watch?v=tIeHLnjs5U8