Evaluation

May 12, 2022

1 Lab 1 Evaluation

Names, Surnames, and Group : (to complete by 2 students)

1.0.1 Djihadi Ahamdy, Malik Sedira, Wissem Ferchichi

In the following, we consider the (binarized) Compas dataset that we studied in the Lab

1) A decision tree configuration is a set of parameters that one can use to build decision trees. Propose 6 configurations that are likely to provide different topologies and caracteristics
2) Train a decision tree for each of the previous configurations on the full dataset

First we prepare the data set to build the decision tree and make all the import we need

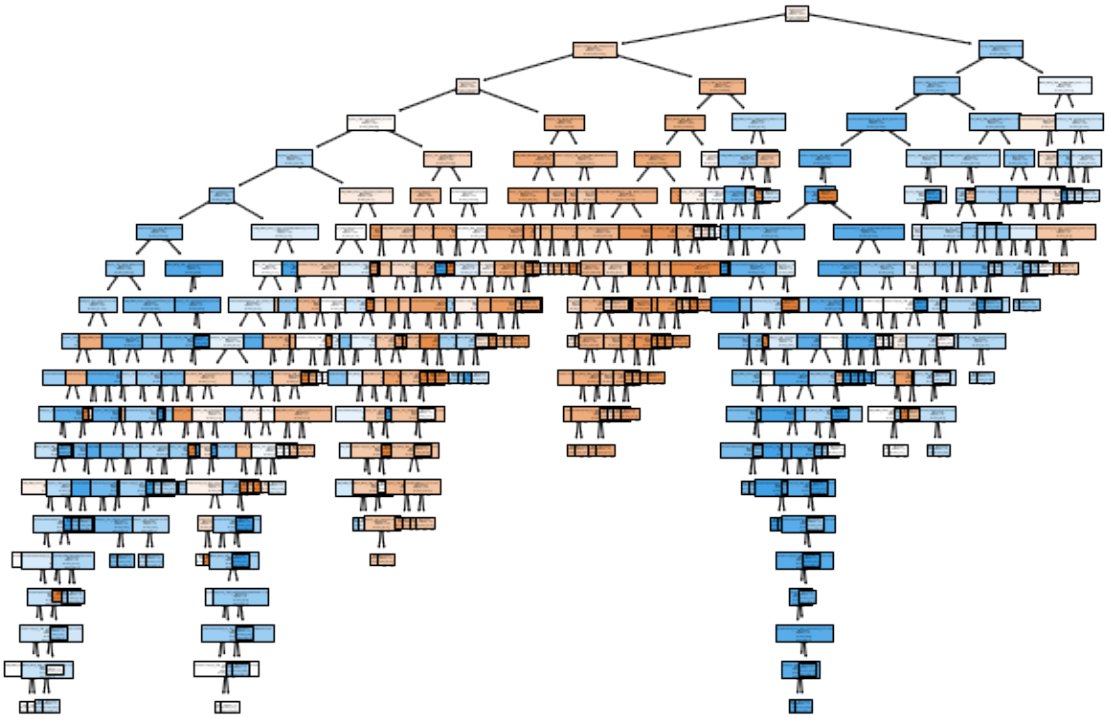
[1]:
```
from sklearn import tree
from sklearn import metrics
from sklearn import model_selection
from matplotlib import pyplot as plt # for a good visualization of the tree
import csv
import pandas as pd
import random
import numpy as np
from utils import load_from_csv


train_examples, train_labels, features, prediction = load_from_csv("./compass.
↪csv")
```

In order to have 6 differents configuration, we decide to specify the criter of "decision tree Classifier".Like the max depth of the tree, min leafes,the strategy used to choose the split at each node, the function to measure the quality of a split and the number of features to consider when looking for the best split.

Configuration A: without specification

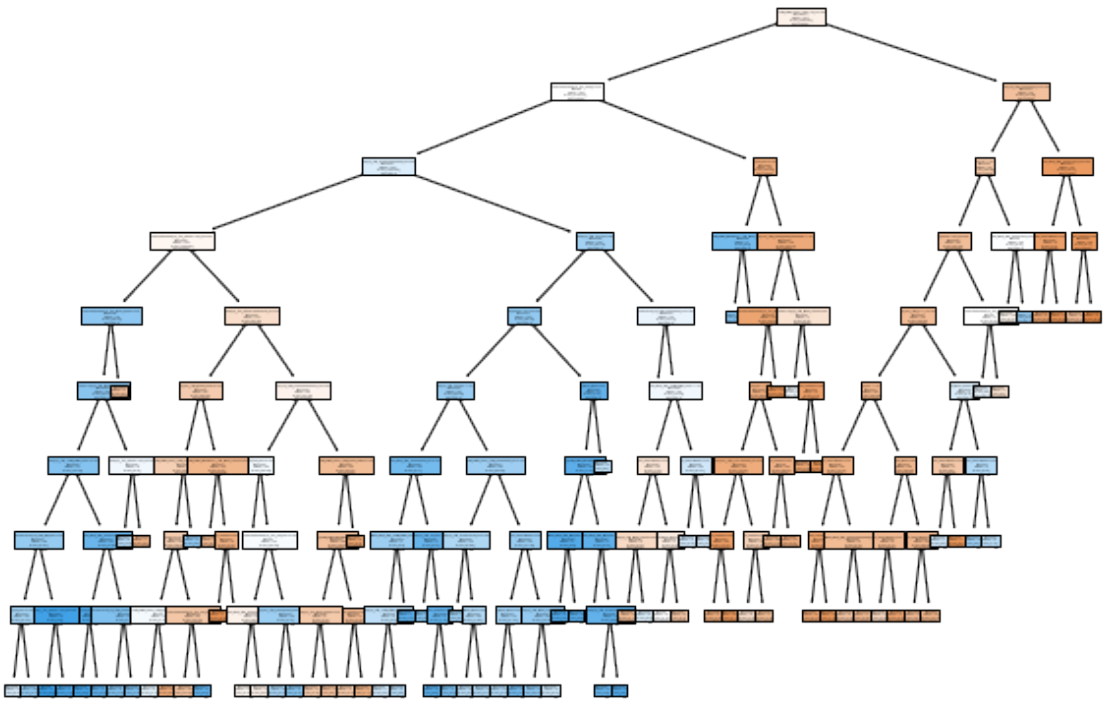
[2]:
```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf,
                   feature_names= features,
```

```
                    class_names= ("false (0)", "true (1)" ),
                    filled=True)
```



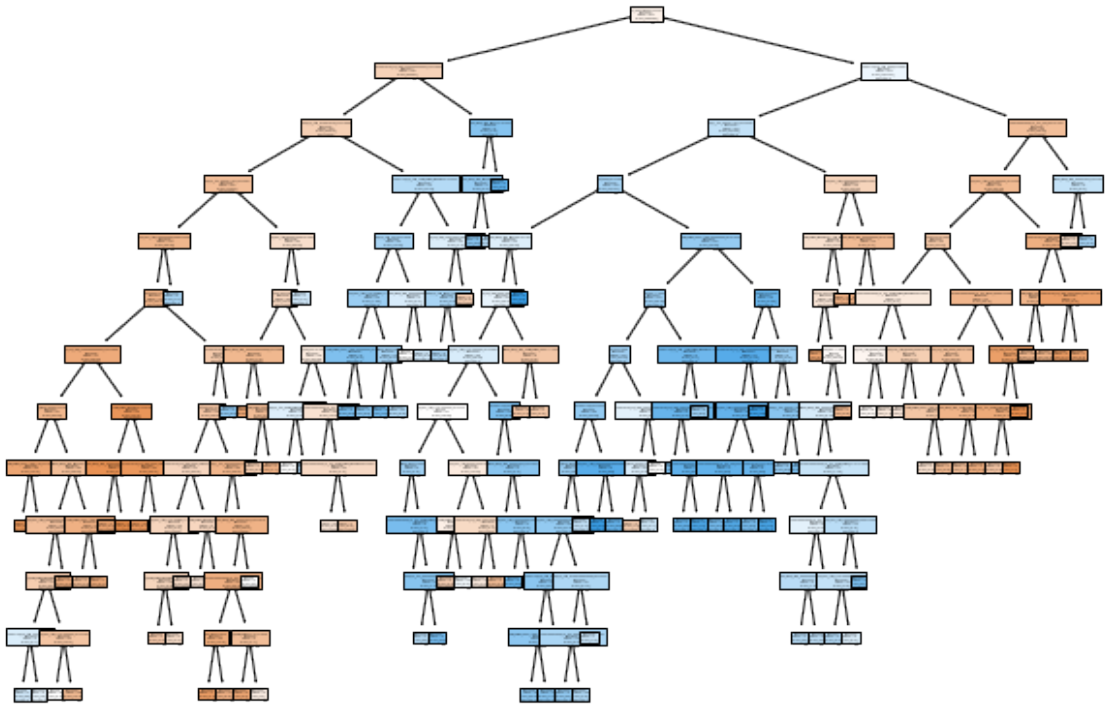
Configuration B:

[3]:
```
profendeur = 9
split = 'best'
criter = 'gini'
min_feuille = 7
max_features = 'auto'

clf = tree.DecisionTreeClassifier(criterion = criter, max_depth = profendeur,␣
 ↪splitter = split , min_samples_leaf = min_feuille,max_features=max_features)
clf_a = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf_a,
                    feature_names= features,
                    class_names= ("false (0)", "true (1)" ),
                    filled=True)
```



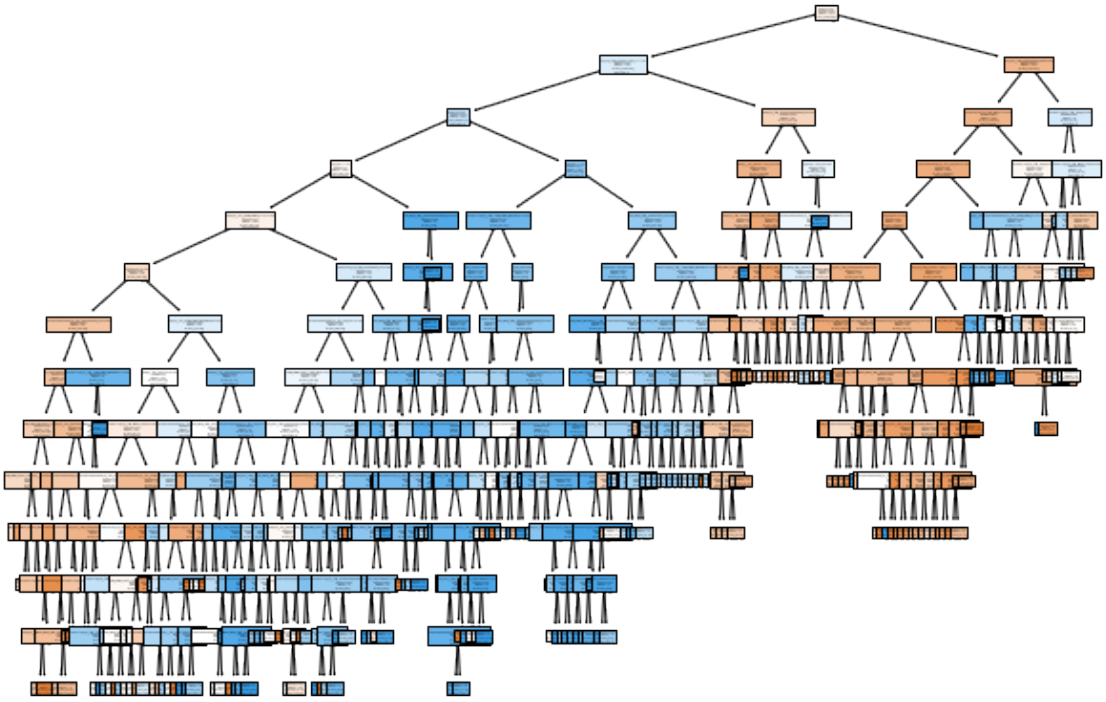
Configuration C:

[4]:
```
profendeur = 12
split = 'random'
criter = 'gini'
min_feuille = 12
max_features = 'sqrt'

clf = tree.DecisionTreeClassifier(criterion = criter, max_depth = profendeur,␣
 ↪splitter = split , min_samples_leaf = min_feuille,max_features=max_features)
clf_b = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf_b,
                   feature_names= features,
                   class_names= ("false (0)", "true (1)" ),
                   filled=True)
```



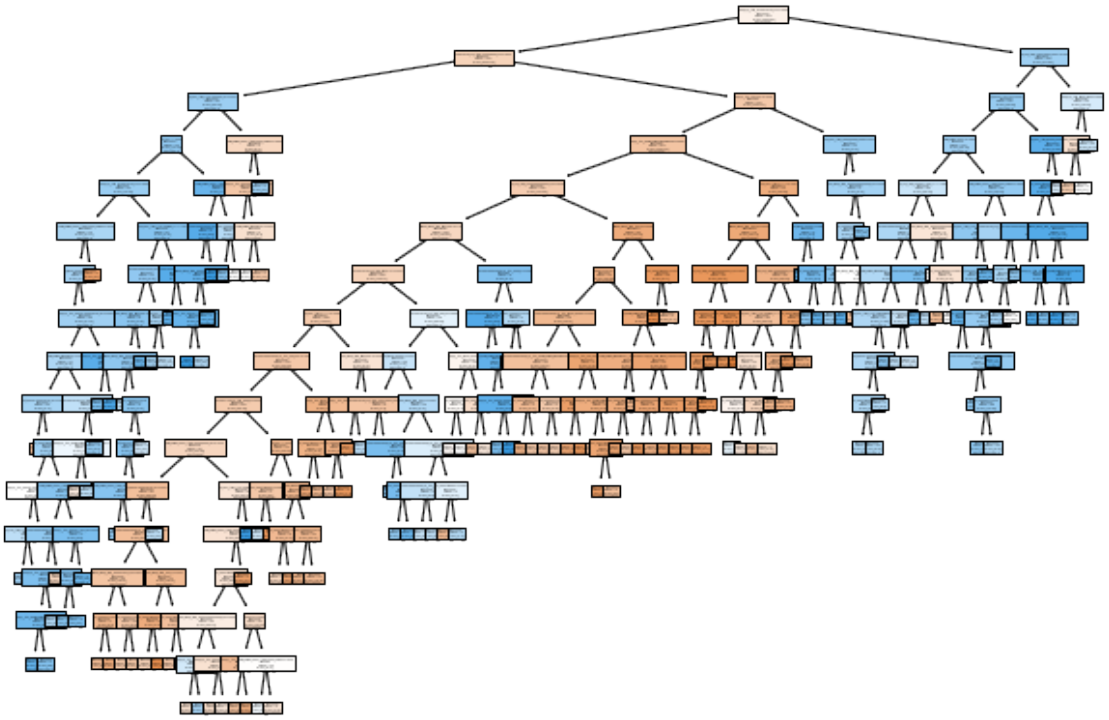
Configuration D:

[5]:
```
profendeur = 13
split = 'best'
criter = 'entropy'
min_feuille = 1
max_features = 'sqrt'

clf = tree.DecisionTreeClassifier(criterion = criter, max_depth = profendeur,␣
 ↪splitter = split , min_samples_leaf = min_feuille,max_features=max_features)
clf_c = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf_c,
                   feature_names= features,
                   class_names= ("false (0)", "true (1)" ),
                   filled=True)
```



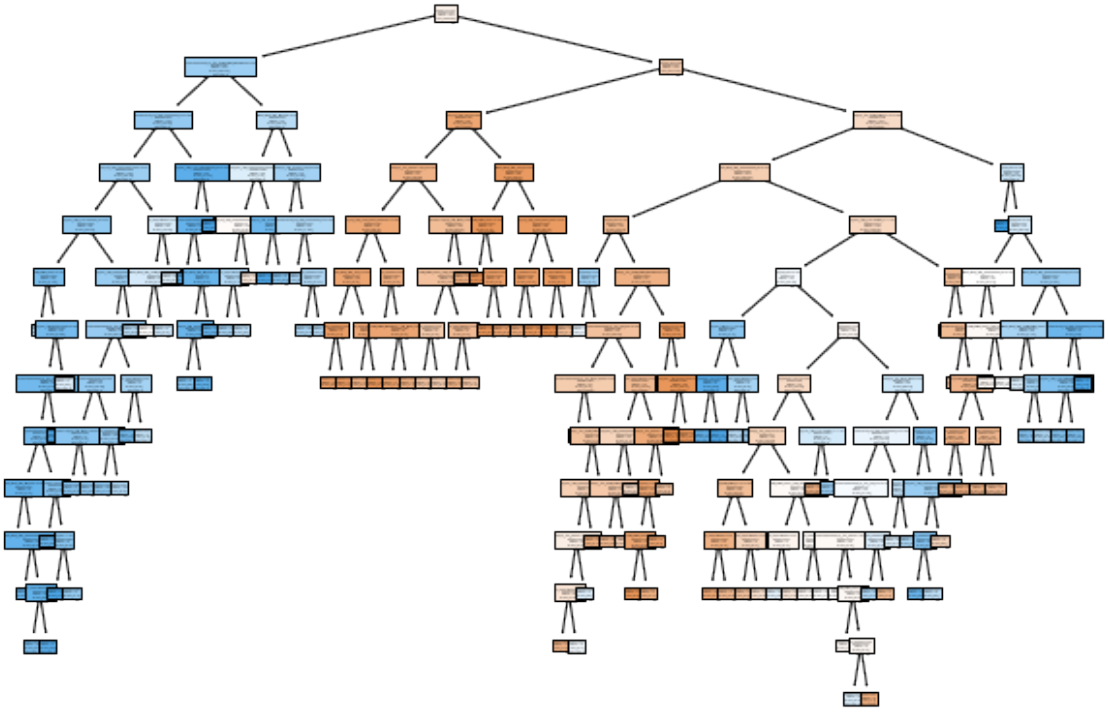
Configuration E:

[6]:
```
profendeur = None
split = 'random'
criter = 'gini'
min_feuille = 5
max_features = 'sqrt'

clf = tree.DecisionTreeClassifier(criterion = criter, max_depth = profendeur,␣
 ↪splitter = split , min_samples_leaf = min_feuille,max_features=max_features)
clf_d = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf_d,
                   feature_names= features,
                   class_names= ("false (0)", "true (1)" ),
                   filled=True)
```



Configuration F:

[7]:
```
profendeur = 20
split = 'best'
criter = 'entropy'
min_feuille = 10
max_features = 'sqrt'

clf = tree.DecisionTreeClassifier(criterion = criter, max_depth = profendeur,␣
 ↪splitter = split , min_samples_leaf = min_feuille,max_features=max_features)
clf_e = clf.fit(train_examples, train_labels)
fig = plt.figure(figsize=(10,7))
_ = tree.plot_tree(clf_e,
                   feature_names= features,
                   class_names= ("false (0)", "true (1)" ),
                   filled=True)
```



3) Propose an evaluation in terms of training and testing accuracies using 5-cross validation on two decision trees that have different typologies

[8]:
```
taille = len(train_examples)

def taill_en_poucentage(x):
    return x*taille/100
def moyen(L):
    return sum(L)/len(L)

tail_prct = int(taill_en_poucentage(80))

#DIFFERENT SPECIFICATIION TO BUILD DIFFERENT TREE DECISION
profondeur=[5,2,5,3,6,8]
min_feuille =[1,10,5,2,4,6]

kf = model_selection.KFold(n_splits=5) # Specify the split in K
kf.get_n_splits(train_examples)

score_tree1=[]
score_tree2=[]
```

```
# CROSS VALIDATION FOR FIRST TYPOLOGI

for train_index, test_index in kf.split(train_examples):
    #print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = train_examples[train_index], train_examples[test_index]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    clf = tree.DecisionTreeClassifier(max_depth=profondeur[0],min_samples_leaf␣
 ↪= min_feuille[0])
    clf = clf.fit(X_train, y_train)
    #print("SCORE : ",clf.score(X_train,y_train)*100 , "Profondeur :
 ↪",profondeur[index],"Feuille Min :",min_feuille[index])
    #print("----------------- \n")

    #ADD THE SCORE TO AN LIST
    score_tree1.append(clf.score(X_train,y_train)*100)



# CROSS VALIDATION FOR THE SECOND TYPOLOGI

for train_index, test_index in kf.split(train_examples):
    #print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = train_examples[train_index], train_examples[test_index]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    clf = tree.DecisionTreeClassifier(max_depth=profondeur[1],min_samples_leaf␣
 ↪= min_feuille[1])
    clf = clf.fit(X_train, y_train)
    #print("SCORE : ",clf.score(X_train,y_train)*100 , "Profondeur :
 ↪",profondeur[index],"Feuille Min :",min_feuille[index])
    #print("----------------- \n")

    #ADD THE SCORE TO AN LIST
    score_tree2.append(clf.score(X_train,y_train)*100)

print("SCORE TREE N°1 :",moyen(score_tree1))
print("SCORE TREE N°2 :",moyen(score_tree2))
```

```
SCORE TREE N°1 : 67.82190256523161
SCORE TREE N°2 : 63.607068477391955
```

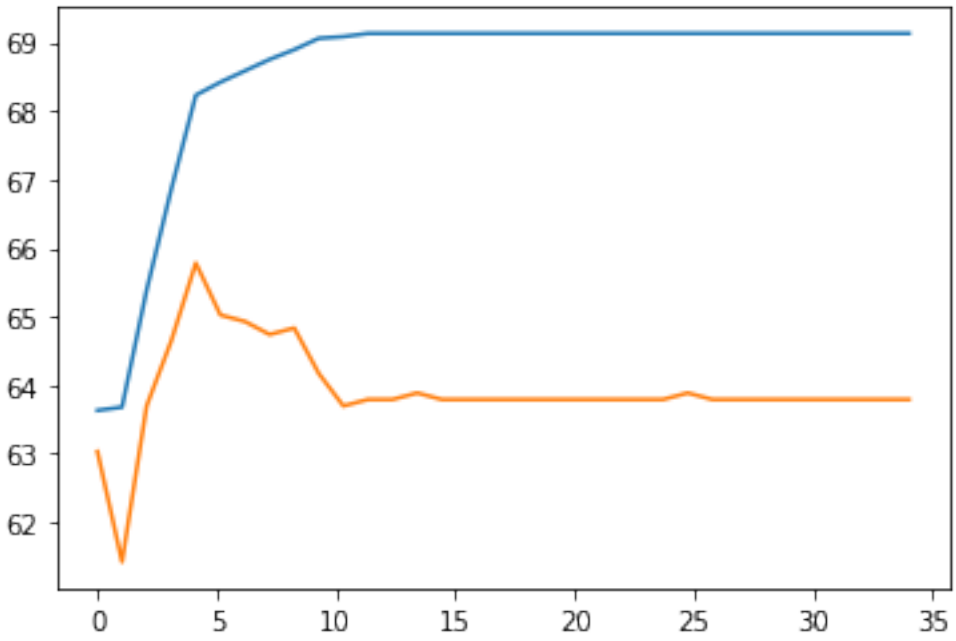
4) Propose an experimental study that shows the transition phase from underfitting to overfitting

[9]:
```
taille = 0

#print("TRUC GENERAL")

#CONFIGURATION
```

```
profendeur = 5
split = 'best'
feuille = 5

print("PREDICTION :",prediction)
taille = len(train_examples)

def taill_en_poucentage(x):
    return x*taille/100
def difference (liste1,liste2):
    somme = 0
    for i in range(len(liste1)):
        if (liste1[i] != liste2[i]):
            somme = somme + 1
    return somme
tail_prct = int(taill_en_poucentage(80))




y1=[]
y2=[]

for i in range (1,35) :
    profendeur = i
    train = train_examples[:tail_prct]
    labels = train_labels[:tail_prct]

    test = train_examples[tail_prct:]
    waitedResult = train_labels[tail_prct:]

    clf = tree.DecisionTreeClassifier(max_depth = profendeur, splitter = split␣
 ↪, min_samples_leaf = feuille)
    clf = clf.fit(train, labels)
    y1.append(clf.score(test,waitedResult)*100)
    y2.append(clf.score(train,labels)*100)

x1=np.linspace(0,34,34)
plt.plot(x1,y2)
plt.plot(x1,y1)




train = train_examples[:tail_prct]
labels = train_labels[:tail_prct]
test = train_examples[tail_prct:]
```

```
waitedResult = train_labels[tail_prct:]

clf = tree.DecisionTreeClassifier(max_depth = profendeur, splitter = split ,␣
 ↪min_samples_leaf = feuille)
clf = clf.fit(train, labels)
print ("SCORE FOR THE PREDICTION :",clf.score(test,waitedResult)*100)
predicted = clf.predict(test)
#print ("",(difference(waitedResult,predicted)/len(test))*100)
```

```
PREDICTION : two_year_recid
SCORE FOR THE PREDICTION : 63.791469194312796
```



We can see above tow curves, the orange one represent the predection score of the tree decision after the training, the blue one represent the predection score of the tree decision during the training. With a depth from 0 to 3, we can see and underfitting because the model is not efficient enough to predict a reel value. With depth equal 4, we have a good fitting and the prediction score is the highest. With a depth equal 5 or higher, we can see and overfitting becaure the score prediction of the decision tree is decreasing.

5) Construct the confusion matrix on a particular good configuration (after explaining your choice)

   - To construct or confusion matrix we decide to compare two parameter the predicted value with the waited Value. This confusion will display the number of correspondance between predict and waited Result.

[14]:
```
confusionMatrix = metrics.confusion_matrix(predicted, waitedResult)
print("Confusion Matrix : \n",confusionMatrix)
```

```
Confusion Matrix :
 [[406 229]
 [153 267]]
```

6) Provide an evaluation of the fairness of the model based on the False Positive Rate

- The confusion Matrix can be read as the number of : -True positif at confusionMatrix [0][0] -True negatif at confusionMatrix [0][1] -False positif at confusionMatrix [1][0] -False negatif at confusionMatrix [1][1]

[11]:
```
print("True negatif rate" , (confusionMatrix[0][1] /␣
 ↪(confusionMatrix[0][0]+confusionMatrix[0][1]))*100)
print("False positif rate" , (confusionMatrix[1][0] /␣
 ↪(confusionMatrix[1][0]+confusionMatrix[1][1]))*100)
```

```
True negatif rate 36.06299212598425
False positif rate 36.42857142857142
```

perceptron

May 12, 2022

0.1 Objectives of the practical work

The objective is to get hands on experience on the fundamental elements of neural networks:

- perceptron architecture (linear regression)
- loss function
- empirical loss
- gradient descent

For this we will implement from scratch the data-structure and algorithms to train a perceptron. Note that slides related to the perceptron and neural networks in general are available on moodle.

0.2 Dataset

The objective of the regression is the prediction of the hydrodynamic performance of sailing yachts from dimensions and velocity. The **inputs** are linked to dimension and hydrodynamics characteristics: 1. Longitudinal position of the center of buoyancy (*flottabilité*), adimensional. 2. Prismatic coefficient, adimensional. 3. Length-displacement ratio, adimensional. 4. Beam -draught ratio ((*tiran d'eau*), adimensional. 5. Length-beam ratio, adimensional. 6. Froude number, adimensional

Target value/predicted value (Output) = Residuary resistance per unit weight of displacement, adimensional

[1]:
```
# Import some useful libraries and functions
from matplotlib import pyplot as plt
import numpy as np
import pandas

def print_stats(dataset):
    """Print statistics of a dataset"""
    print(pandas.DataFrame(dataset).describe())
```

[2]:
```
# Download the data set and place in the current folder (works on linux only)
filename = 'yacht_hydrodynamics.data'

import os.path
import requests

if not os.path.exists(filename):
```

```
    print("Downloading dataset...")
    r = requests.get('https://arbimo.github.io/tp-supervised-learning/tp2/' +␣
 ↪filename)
    open(filename , 'wb').write(r.content)

print('Dataset available')
```

```
Dataset available
```

0.2.1 Explore the dataset

- how many examples are there in the dataset ? There are 308 examples in the data set
- how many features for each example ? here are 6 features in each example
- what is the ground truth of the 10th example ? The ground truth of the 10th example is 1.83

[3]:
```
# loads the dataset and slip between inputs (X) and ground truth (Y)
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth

# Print the first 5 examples
for i in range(0,5):
    print(f"f({X[i]}) = {Y[i]}")
```

```
f([-5.    0.6   4.78  4.24  3.15  0.35]) = 8.62
f([-5.    0.565  4.77   3.99   3.15   0.15 ]) = 0.18
f([-2.3   0.565  4.78   5.35   2.76   0.15 ]) = 0.29
f([-5.    0.6    4.78   4.24   3.15   0.325]) = 6.2
f([0.    0.53  4.78  3.75  3.15  0.175]) = 0.59
```

The following command adds a column to the inputs.

- what is in the value added this column?
  The column added contains only the value 1
- why are we doing this?
  So we can simulate the weight

[4]:
```
X = np.insert(X, 0, np.ones((len(X))), axis= 1)
print_stats(X)
```

```
           0           1           2           3           4           5  \
count  308.0  308.000000  308.000000  308.000000  308.000000  308.000000
mean     1.0   -2.381818    0.564136    4.788636    3.936818    3.206818
std      0.0    1.513219    0.023290    0.253057    0.548193    0.247998
min      1.0   -5.000000    0.530000    4.340000    2.810000    2.730000
25%      1.0   -2.400000    0.546000    4.770000    3.750000    3.150000
50%      1.0   -2.300000    0.565000    4.780000    3.955000    3.150000
75%      1.0   -2.300000    0.574000    5.100000    4.170000    3.510000
max      1.0    0.000000    0.600000    5.140000    5.350000    3.640000
```

```
                6
count  308.000000
mean     0.287500
std      0.100942
min      0.125000
25%      0.200000
50%      0.287500
75%      0.375000
max      0.450000
```

0.3 Creating the perceptron

We now want to define a perceptron, that is, a function of the form:

$h_w(x) = w_0 + w_1 \times x_1 + \cdots + w_n \times x_n$

- Complete the code snippet below to:
  - create the vector of weight `w`
  - implement the `h` function that evaluate an example based on the vector of weights
  - check if this works on a few examples

[5]:
```
w = None # TODO
index = 0

w = [row[0] for row in X]
ligne = X[index,1:]
def h(w, x):
    h = np.multiply(w, x)
    return sum(h)
h(w[index], ligne)


# print the ground truth and the evaluation of h_w on the first example
```

[5]:
```
8.12
```

0.4 Loss function

Complete the definiton of the loss function below such that, for a **single** example `x` with ground truth `y`, it returns the L_2 loss of h_w on `x`.

[16]:
```
def loss(w, x, y):
    return (y - h(w, x))**2

print(loss(w[index], ligne, Y[index]))
print(len(X))
```

```
0.25
308
```

0.5 Empirical loss

Complete the function below to compute the empirical loss of h_w on a **set** of examples X with associated ground truths Y.

[17]:
```
def emp_loss(w, X, Y):
    somme = 0
    for index in range (0,len(X)):
        ligne = X[index,1:]
        somme = somme + (loss(w[index], ligne, Y[index]) / len(X))
    return somme

print("Empirical loss :",emp_loss(w,X,Y))
```

```
Empirical loss : 228.81521738311693
```

0.6 Gradient update

A gradient update is of the form: $w \leftarrow w + dw$

- Complete the function below so that it computes the dw term (the 'update') based on a set of examples `(X, Y)` the step (`alpha`)

If you are not sure about the gradient computation, check out the perceptron slides on Moodle (in particular, slide 26). Make sure this computation is clear to you!

[13]:
```
def compute_update(w, X, Y, alpha):
    dw = np.zeros_like(w)
    for index in range(0,len(w)):
        ligne = X[index,1:]
        dw[index] = w[index] + alpha*loss(w[index], ligne, Y[index])*ligne[0]
    return dw
#print(w,X,Y)
#compute_update(w, X, Y, alpha = 2)
#print(w,X,Y)
```

0.7 Gradient descent

Now implement the gradient descent algorithm that will:

- repeatedly apply an update the weights
- stops when a max number of iterations is reached (do not consider early stopping for now)
- returns the final vector of weights

[14]:
```
def descent(w_init, X, Y, alpha, max_iter):
    index = 0
    wzero = 0
```

```
    dw = w_init
    while(index < max_iter ):
        dw += compute_update(w_init, X, Y, alpha)
        index+=1
    return dw

test = descent( w ,X, Y, 2, 50)

#print(test)
```

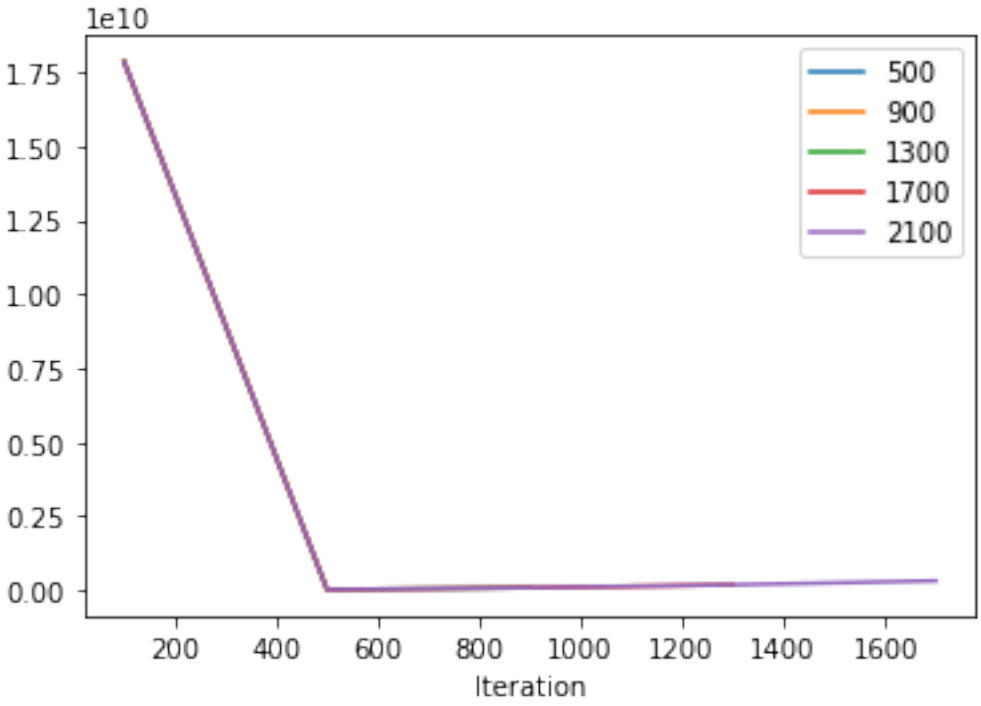
0.8 Exploitation

You gradient descent is now complete and you can exploit it to train your perceptron.

- Train your perceptron to get a model.
- Visualize the evolution of the loss on the training set. Has it converged? - Yes it converged as you can see in the plot the loss decrease thanks to the alpha and the iteration
- Try training for several choices of `alpha` and `max_iter`. What seem like a reasonable choice? - 'alpha' = 10e-5 and max iter = 500
- Is the final model the optimal one for a perceptron? - 'alpha' = 10e-5 and max iter = 500

[49]:
```
# Code sample that can be used to visualize the difference between the ground␣
↪truth and the prediction

iteration = 100
list_iter = []
list_emp_loss = []
list_alpha=[10e-2,10e-5,10e-7,10e-9,10e-10,10e-11]
alpha = 10e-2


for i in range (0,5):
    list_iter.append(iteration)
    temp_W = [row[0] for row in X]
    temp_W = descent(temp_W,X,Y,list_alpha[i],iteration)
    list_emp_loss.append(emp_loss(temp_W,X,Y))
    iteration+=400
    plt.plot(list_iter,list_emp_loss, label=str(iteration))
    plt.legend()


plt.xlabel("Iteration")
```

[49]:
```
Text(0.5, 0, 'Iteration')
```



1 Going further

The following are extensions of the work previously done. If attempting them **do not modify** the code you produced above so that it can be evaluated.

1.0.1 Improvements to gradient descent

Consider improving the gradient descent with:

- Stochastic Gradient Descent (SGD), which means selecting a subset of the examples for training
- Detection of convergence to halt the algorithm before the maximum number of iterations

1.0.2 Data normalization

Different input features can have different units, and very different ranges. Within the perceptron computation, these values will be summed together. While gradient descent is normally able to deal with this (by adapting the weights of the perceptron for each input feature), standardizing the input features usually eases the perceptron training, and can sometimes improve accuracy.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler(copy=True)
X_normalized = sc.fit_transform(X)
```

mlp

May 12, 2022

1 Introduction

The objective of this lab is to dive into particular kind of neural network: the *Multi-Layer Perceptron* (MLP).

To start, let us take the dataset from the previous lab (hydrodynamics of sailing boats) and use scikit-learn to train a MLP instead of our hand-made single perceptron. The code below is already complete and is meant to give you an idea of how to construct an MLP with scikit-learn. You can execute it, taking the time to understand the idea behind each cell.

[1]:
```
# Importing the dataset
import numpy as np
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1]
Y = dataset[:, -1]

#to visualize the data you can uncomment the two following line
#print(X)
#print(Y)
```

[2]:
```
# Preprocessing: scale input data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
print(X)
```

```
[[-1.73302265  1.5423783  -0.03418367  0.55395671 -0.2294799   0.62017367]
 [-1.73302265  0.03714219 -0.07376475  0.09717082 -0.2294799  -1.36438208]
 [ 0.05415696  0.03714219 -0.03418367  2.58208609 -1.80462994 -1.36438208]
 …
 [ 0.05415696 -1.46809393 -0.11334584 -0.46924369 -0.18909144 -0.86824314]
 [-1.73302265  1.5423783  -0.03418367  0.55395671 -0.2294799  -0.12403473]
 [ 0.05415696 -1.46809393  1.27199221 -0.45097226  1.22450475  0.62017367]]
```

[3]:
```
# Split dataset into training and test set
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y,random_state=1,␣
 ↪test_size = 0.20)
```

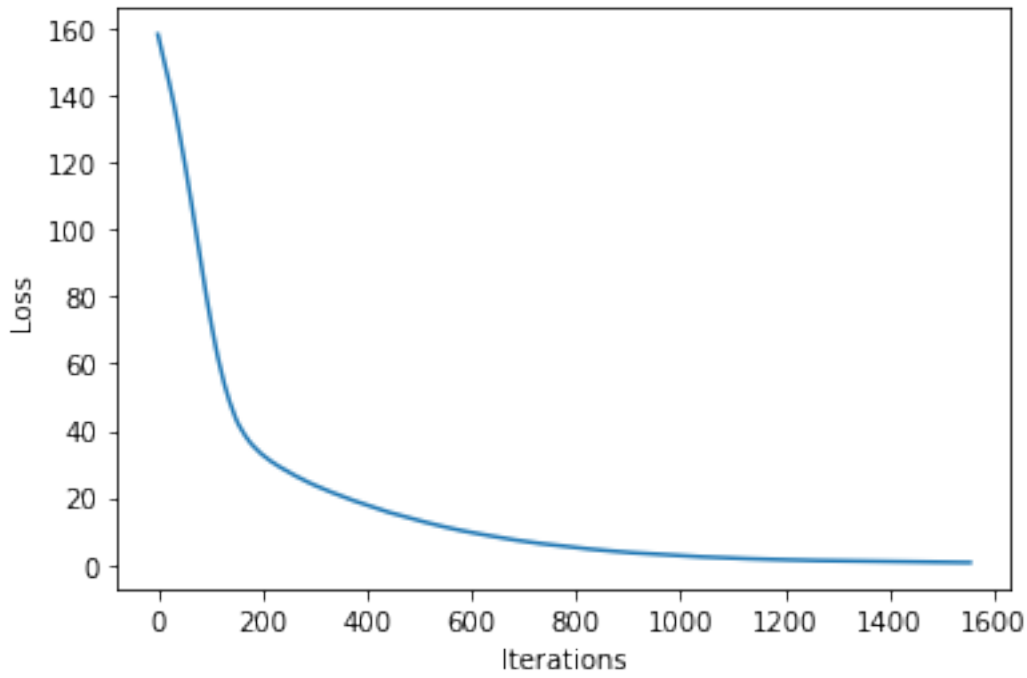
[4]:
```
# Define a multi-layer perceptron (MLP) network for regression
from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(max_iter=3000, random_state=1) # define the model, with␣
 ↪default params
mlp.fit(x_train, y_train) # train the MLP
```

[4]:
```
MLPRegressor(max_iter=3000, random_state=1)
```

[5]:
```
# Evaluate the model
from matplotlib import pyplot as plt

print('Train score: ', mlp.score(x_train, y_train))
print('Test score:  ', mlp.score(x_test, y_test))
plt.plot(mlp.loss_curve_)
plt.xlabel("Iterations")
plt.ylabel("Loss")
```

```
Train score:  0.9940765369322633
Test score:   0.9899773031580283
```

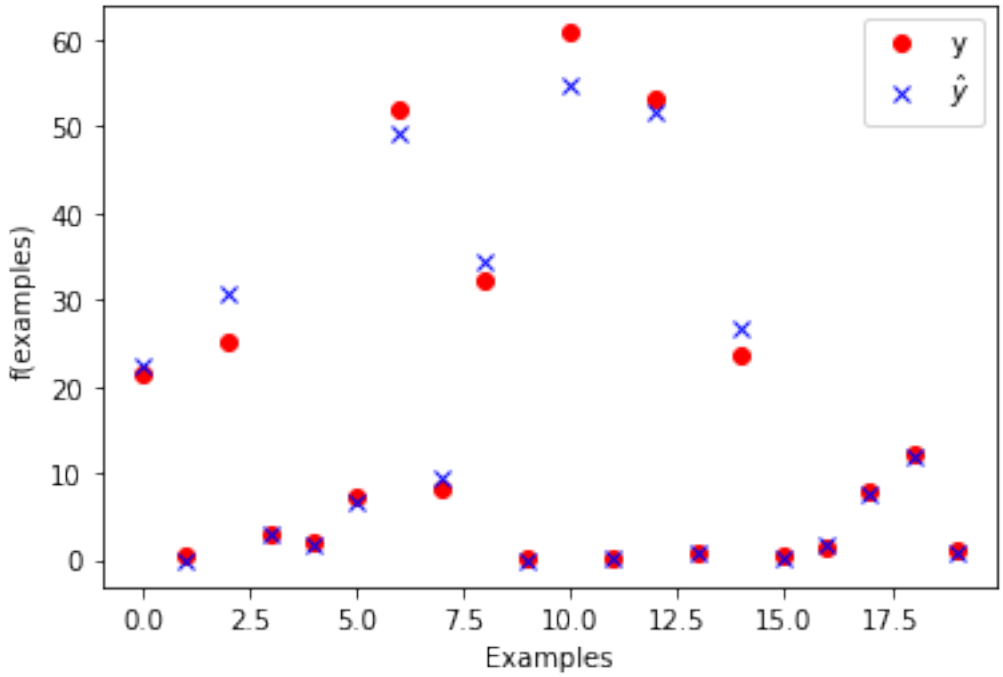
[5]:
```
Text(0, 0.5, 'Loss')
```



[6]:
```
# Plot the results
num_samples_to_plot = 20
plt.plot(y_test[0:num_samples_to_plot], 'ro', label='y')
```

```
yw = mlp.predict(x_test)
plt.plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')
plt.legend()
plt.xlabel("Examples")
plt.ylabel("f(examples)")
```

[6]: Text(0, 0.5, 'f(examples)')



1.0.1 Analyzing the network

Many details of the network are currently hidden as default parameters.

Using the documentation of the MLPRegressor, answer the following questions.

- What is the structure of the network?
  - The network is composed by a multiLayer Perceptron regressor
- What it is the algorithm used for training? Is there algorithm available that we mentioned during the courses?
  - MLPRegressor proposed different algorithm : like 'LGBS' for quasi-Newton methods 'SGD' for stochastic gradient descent 'ADAM' for stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba
    - We already seen quasi-Newton and Gradient descent during our courses
- How does the training algorithm decides to stop the training? - He decide to stop the training when the score is not improving

2 Onto a more challenging dataset: house prices

For the rest of this lab, we will use the (more challenging) California Housing Prices dataset.

[7]:
```
# clean all previously defined variables for the sailing boats
%reset -f
```

[8]:
```
"""Import the required modules"""
from sklearn.datasets import fetch_california_housing
import pandas as pd

num_samples = 2000 # only use the first N samples to limit training time

cal_housing = fetch_california_housing()
X = pd.DataFrame(cal_housing.data,columns=cal_housing.feature_names)[:
 ↪num_samples]
Y = cal_housing.target[:num_samples]

X.head(10) # print the first 10 values
print(Y)
```

```
[4.526 3.585 3.521 … 0.526 0.469 0.939]
```

Note that each row of the dataset represents a **group of houses** (one district). The `target` variable denotes the average house value in units of 100.000 USD. Median Income is per 10.000 USD.

2.0.1 Extracting a subpart of the dataset for testing

- Split the dataset between a training set (75%) and a test set (25%)

Please use the conventional names `X_train`, `X_test`, `y_train` and `y_test`.

[9]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y,random_state=10,␣
 ↪test_size = 0.25)
```

2.0.2 Scaling the input data

A step of **scaling** of the data is often useful to ensure that all input data centered on 0 and with a fixed variance.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance). The function `StandardScaler` from `sklearn.preprocessing` computes the standard score of a sample as:

```
z = (x - u) / s
```

where `u` is the mean of the training samples, and `s` is the standard deviation of the training samples.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using transform.

- Apply the standard scaler to both the training dataset (`X_train`) and the test dataset (`X_test`).
- Make sure that **exactly the same transformation** is applied to both datasets.

Documentation of standard scaler in scikit learn

[10]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.fit_transform(x_test)
```

2.1 Overfitting

In this part, we are only interested in maximizing the **train score**, i.e., having the network memorize the training examples as well as possible.

- Propose a parameterization of the network (shape and learning parameters) that will maximize the train score (without considering the test score).

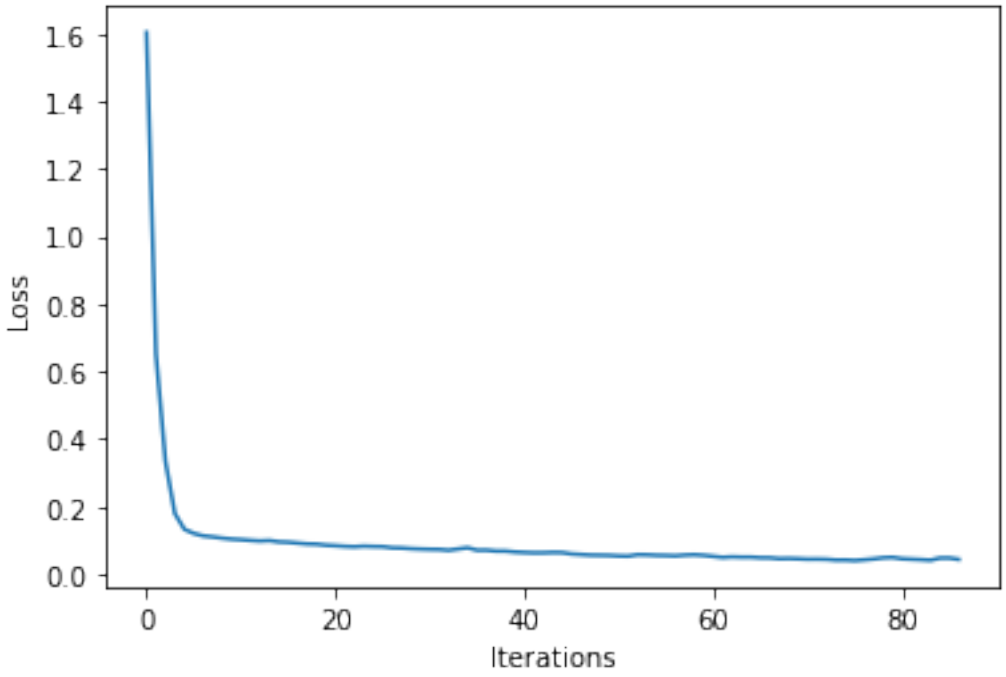
While doing this, you should (1) remain within two minutes of training time, and (2) obtain a score that is greater than 0.90.

- Is the **test** score substantially smaller than the **train** score (indicator of overfitting) ?
- Explain how the parameters you chose allow the learned model to overfit.

[11]:
```
# Define a multi-layer perceptron (MLP) network for regression
from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(max_iter=500 ,random_state=5, alpha = 3e-7,␣
 ↪hidden_layer_sizes= (100,100,100,100,100,100) )# define the model, with␣
 ↪default params
mlp.fit(x_train, y_train) # train the ML

# Evaluate the model
from matplotlib import pyplot as plt

print('Train score: ', mlp.score(x_train, y_train))
print('Test score:  ', mlp.score(x_test, y_test))
plt.plot(mlp.loss_curve_)
plt.xlabel("Iterations")
plt.ylabel("Loss")
```

```
Train score:  0.9138260193968096
Test score:   0.734799767372307
```

[11]:
```
Text(0, 0.5, 'Loss')
```



2.2 Hyperparameter tuning

In this section, we are now interested in maximizing the ability of the network to predict the value of unseen examples, i.e., maximizing the **test** score. You should experiment with the possible parameters of the network in order to obtain a good test score, ideally with a small learning time.

Parameters to vary:

- number and size of the hidden layers
- activation function
- stopping conditions
- maximum number of iterations
- initial learning rate value

Results to present for the tested configurations:

- Train/test score
- training time

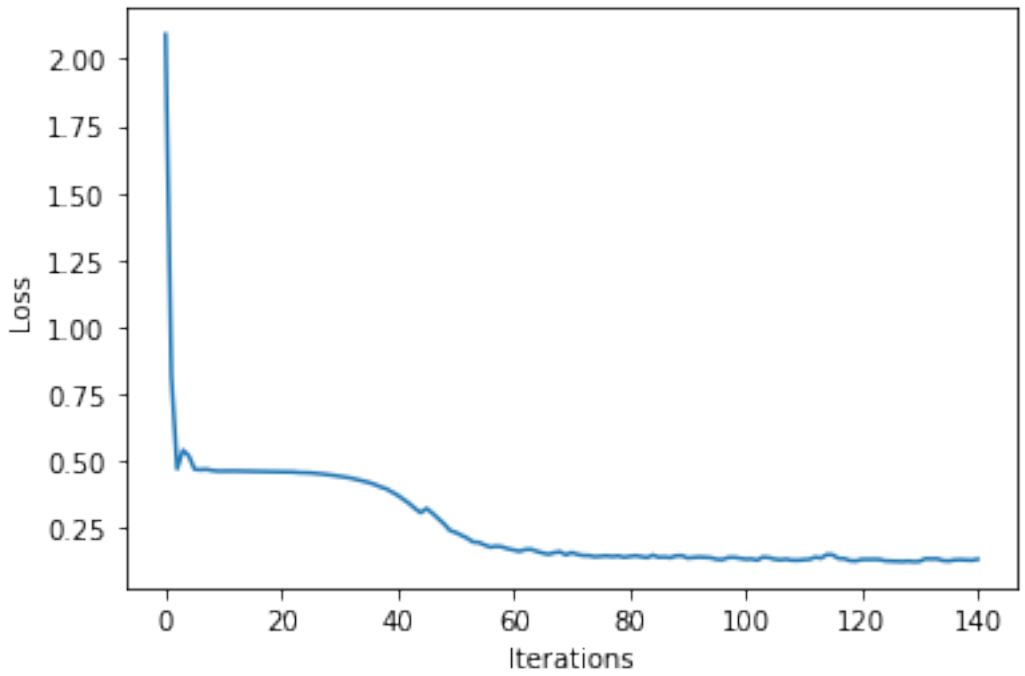
Present in a table the various parameters tested and the associated results. You can find in the last cell of the notebook a code snippet that will allow you to plot tables from python structure. Be methodical in the way your run your experiments and collect data. For each run, you should record the parameters and results into an external data structure.

(Note that, while we encourage you to explore the solution space manually, there are existing methods in scikit-learn and other learning framework to automate this step as well, e.g., GridSearchCV)

[17]:
```
# Define a multi-layer perceptron (MLP) network for regression
from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(learning_rate_init = 0.001,activation = "logistic",␣
 ↪max_iter=500 ,random_state=5, alpha = 3e-7, hidden_layer_sizes=␣
 ↪(100,100,100,100))# define the model, with default params
mlp.fit(x_train, y_train) # train the ML

# Evaluate the model
from matplotlib import pyplot as plt

print('Train score: ', mlp.score(x_train, y_train))
print('Test score:  ', mlp.score(x_test, y_test))
plt.plot(mlp.loss_curve_)
plt.xlabel("Iterations")
plt.ylabel("Loss")
```

```
Train score:  0.7355856630980302
Test score:   0.7009775458325185
```

[17]:
```
Text(0, 0.5, 'Loss')
```



[18]:
```
# Code snippet to display a nice table in jupyter notebooks  (remove from␣
 ↪report)
import pandas as pd
import numpy as np
```

```
data = []
data.append({'activation': 'relu', 'max_iter': '500','early_stopping': False,␣
↪'test_score': 0.87})
data.append({'activation': 'tanh', 'max_iter': '500', 'early_stopping': False,␣
↪'test_score': 0.91})
data.append({'activation': 'logistic', 'max_iter': '500', 'early_stopping':␣
↪False, 'test_score': 0.91})
data.append({'activation': 'identity', 'max_iter': '500', 'early_stopping':␣
↪False, 'test_score': 0.91})

table = pd.DataFrame.from_dict(data)
table = table.replace(np.nan, '-')
table = table.sort_values(by='test_score', ascending=False)
table
```

[18]:

	activation	max_iter	early_stopping	test_score
1	tanh	500	False	0.91
2	logistic	500	False	0.91
3	identity	500	False	0.91
0	relu	500	False	0.87

2.3 Evaluation

- From your experiments, what seems to be the best model (i.e. set of parameters) for predicting the value of a house?
  - Thanks our experiments, we can say the best model can be decide by : relu with a huge amount of layer and logistic with a small layer

Unless you used cross-validation, you have probably used the "test" set to select the best model among the ones you experimented with. Since your model is the one that worked best on the "test" set, your selection is *biased*.

In all rigor the original dataset should be split in three:

- the **training set**, on which each model is trained
- the **validation set**, that is used to pick the best parameters of the model
- the **test set**, on which we evaluate the final model

Evaluate the score of your algorithm on a test set that was not used for training nor for model selection.

[62]:
```
# the **training set**, on which each model is trained
# the **validation set**, that is used to pick the best parameters of the model
# the **test set**, on which we evaluate the final model

import time
from sklearn import metrics
from time import gmtime, strftime
```

```
x_train, x_test, y_train, y_test = train_test_split(X, Y,random_state=10,␣
↪test_size = 0.25)
x_train, x_validation, y_train, y_validation = train_test_split(x_train,␣
↪y_train,random_state=10, test_size = 0.45)

data =[]

nbr_layer␣
↪=[(100),(100,100),(100,100,100),(100,100,100,100),(100,100,100,100,100,100)]
stop = True
maxIter = 500
active = ["relu","identity","logistic","tanh"]
learnRate = 0.001

index = 0

for layer in nbr_layer:
    for actv in active:
        temps= time.time()
        x_train, x_test, y_train, y_test = train_test_split(X,␣
↪Y,random_state=10, test_size = 0.25)
        x_train, x_validation, y_train, y_validation =␣
↪train_test_split(x_train, y_train,random_state=10, test_size = 0.15)

        mlp = MLPRegressor(learning_rate_init = learnRate,activation␣
↪=actv,early_stopping=stop,max_iter=maxIter,random_state=5, alpha = 3e-7,␣
↪hidden_layer_sizes= layer)# define the model, with default params
        mlp.fit(x_train, y_train) # train the ML
        temps = time.time() - temps

        data.append({'index':index,'activation':actv,'layer':layer,'max_iter':
↪maxIter,'early_stopping':stop,'train_score':mlp.score(x_train,␣
↪y_train),'time':temps})
        index+=1


table = pd.DataFrame.from_dict(data)
table = table.replace(np.nan, '-')
table = table.sort_values(by='train_score', ascending=False)
table
```

```
/usr/local/insa/anaconda/lib/python3.8/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:614:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (500) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

[62]:

	index	activation	layer	max_iter	early_stopping	train_score	time
10	10	logistic	(100, 100, 100)	500	True	0.722274	1.841020
8	8	relu	(100, 100, 100)	500	True	0.718311	1.864872
14	14	logistic	(100, 100, 100, 100)	500	True	0.709680	3.166287
18	18	logistic	(100, 100, 100, 100, 100, 100)	500	True	0.700325	4.686566
2	2	logistic	100	500	True	0.698407	0.960957
0	0	relu	100	500	True	0.696545	0.717048
1	1	identity	100	500	True	0.687407	0.972398
4	4	relu	(100, 100)	500	True	0.683163	1.594779
7	7	tanh	(100, 100)	500	True	0.681408	1.252844
11	11	tanh	(100, 100, 100)	500	True	0.676987	1.261975
12	12	relu	(100, 100, 100, 100)	500	True	0.673400	2.784359
19	19	tanh	(100, 100, 100, 100, 100, 100)	500	True	0.641668	3.622228
15	15	tanh	(100, 100, 100, 100)	500	True	0.638442	2.049415
17	17	identity	(100, 100, 100, 100, 100, 100)	500	True	0.460923	10.380034
13	13	identity	(100, 100, 100, 100)	500	True	0.409403	4.141897
5	5	identity	(100, 100)	500	True	0.401997	0.842226
9	9	identity	(100, 100, 100)	500	True	0.281989	1.633010
16	16	relu	(100, 100, 100, 100, 100, 100)	500	True	0.043660	0.810340
3	3	tanh	100	500	True	0.039858	0.124614
6	6	logistic	(100, 100)	500	True	0.009941	0.311909