Lab 1 Evaluation

Names, Surnames, and Group :

BACCAR Rostom 4IR C1

CHAUZY Célia 4IR C1

YAKHELEF Morgan 4IR C1

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

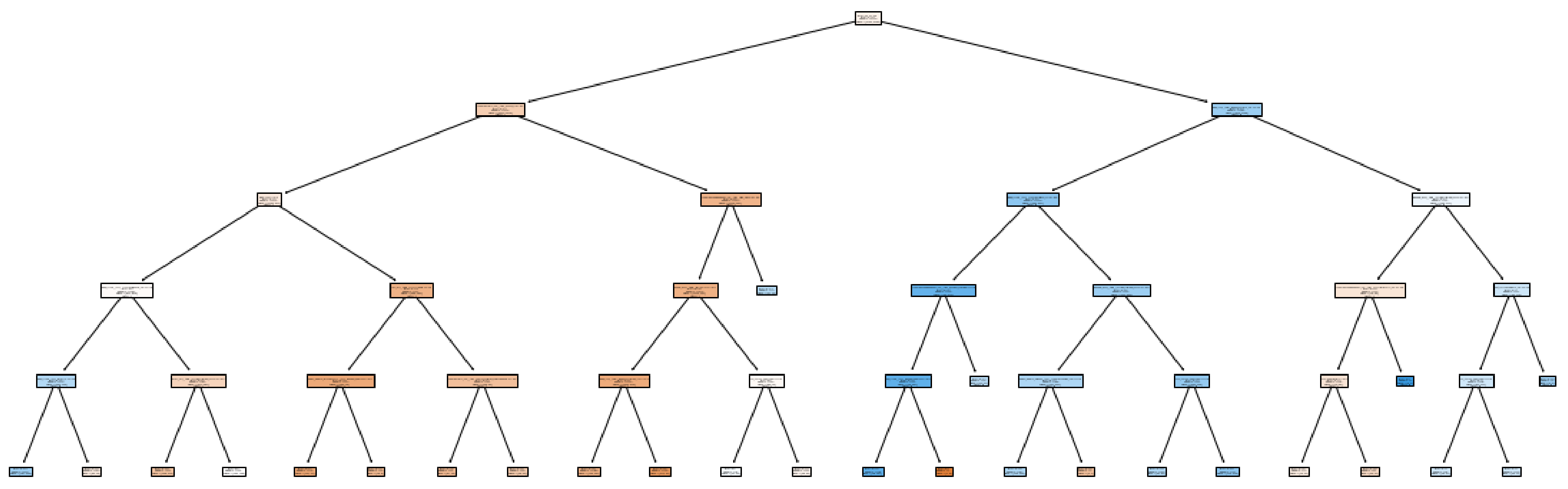
The likelyhood of providing different topologies and caracteristics depends on the parameters we use to build the trees. These parameters are splitter, max_depth and min_samples_split.

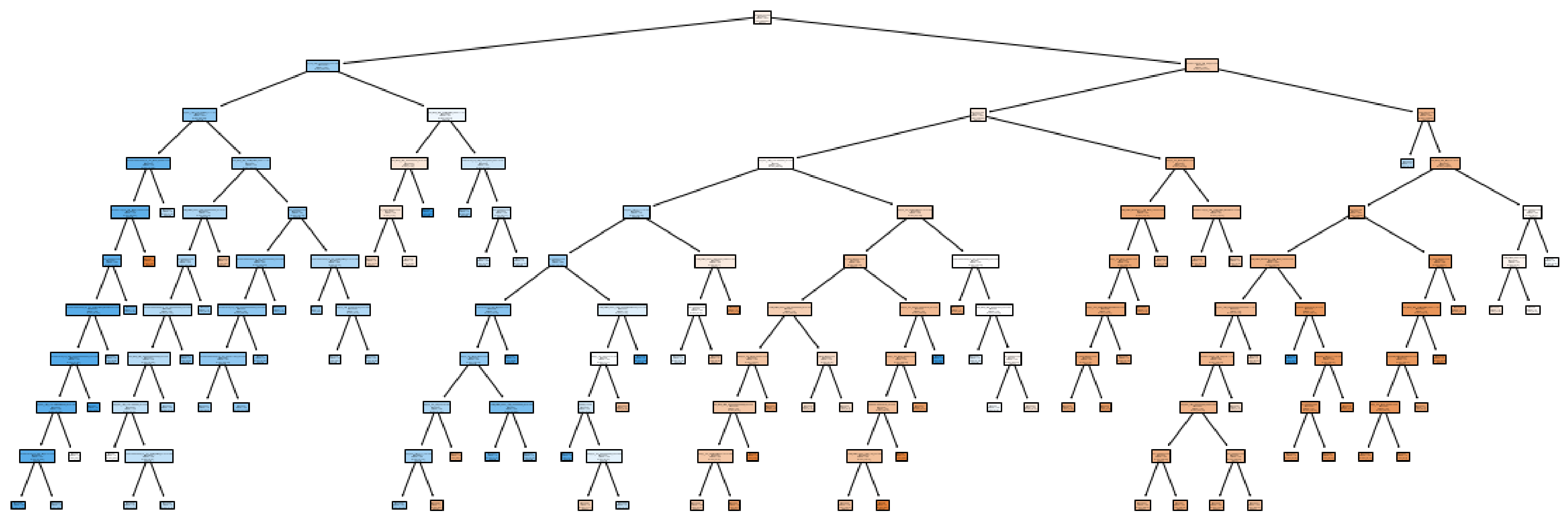
- The splitter, which takes only two values (best or random), determines how the split at each note is going to be: either it's the best split or the best random split.
- The max_depth parameter determines how deep our tree is going to be, that is its height. The deeper the tree the more complex and accurate the model but the more likely it's going to end up overfitting so there is a certain balance to hit.
- The min_samples_split parameter determines the minimum number of samples required to do a split at a node. The greater this number is, the more complex the model is but there is also a risk of overfitting

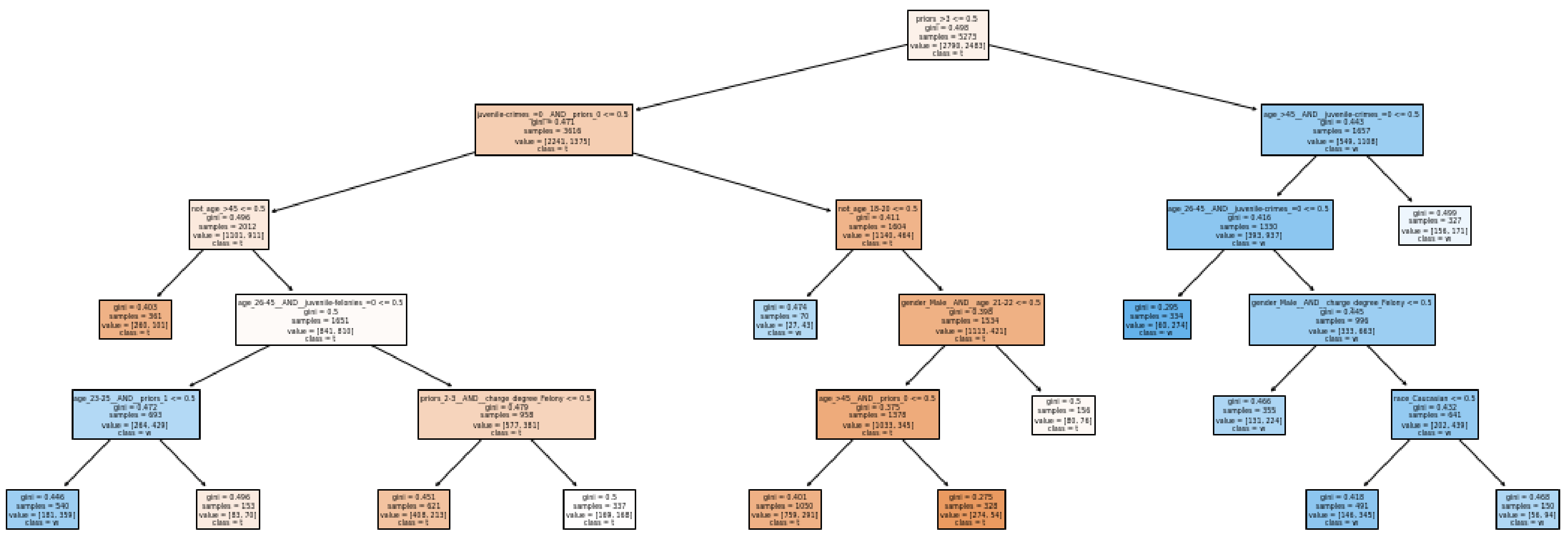
Here are the 6 configurations that we propose:
split=best, max_depth=5, min_samples_split=100
split=best, max_depth=10, min_samples_split=100
split=best, max_depth=5, min_samples_split=500
split=random, max_depth=5, min_samples_split=100
split=random, max_depth=20, min_samples_split=50
split=random, max_depth=5, min_samples_split=1000
</span>

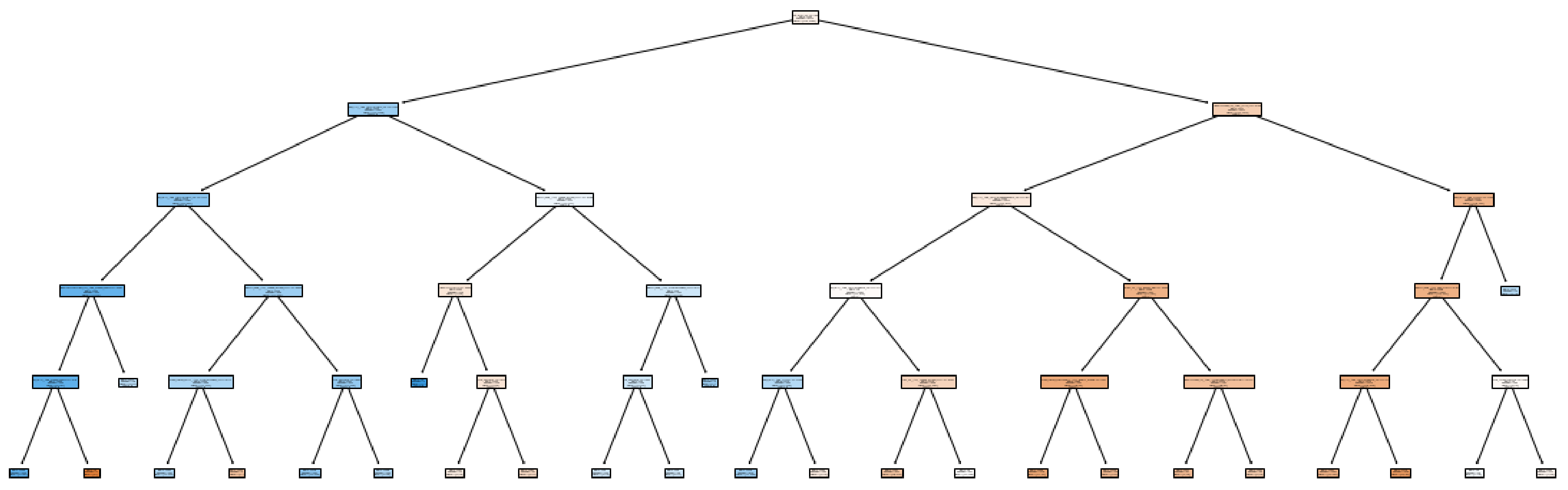
Train a decision tree for each of the previous configurations on the full dataset

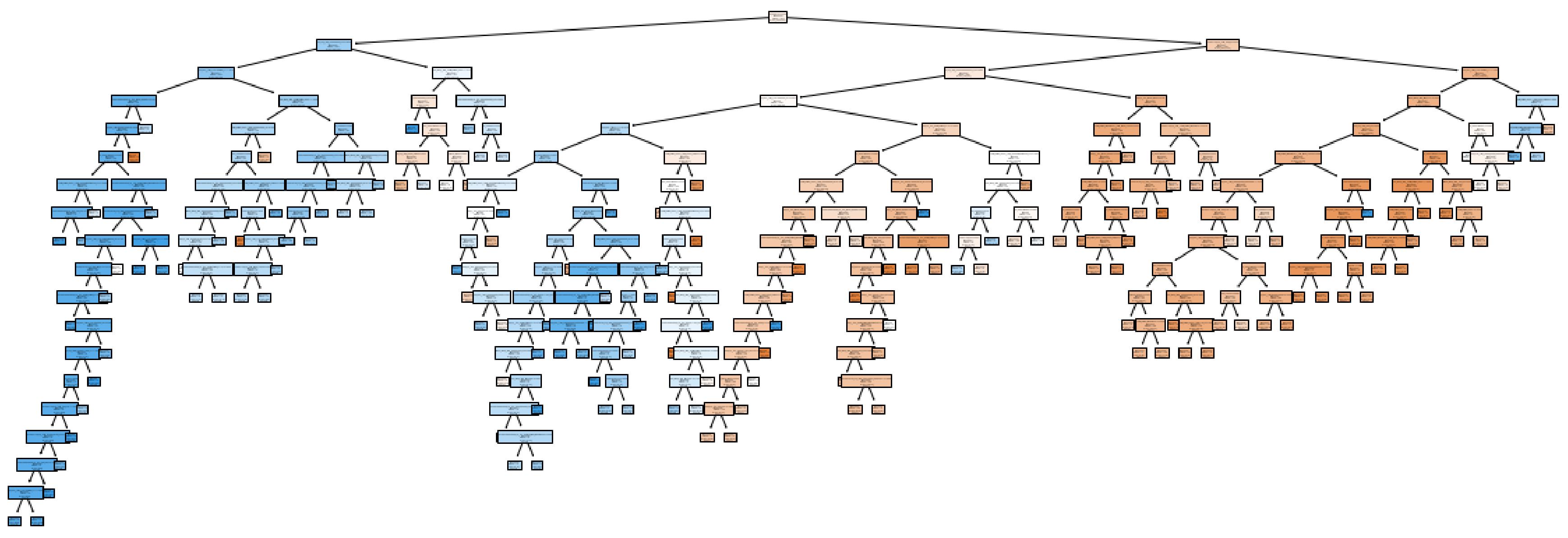
In [2]:
```
#Imports
from sklearn import tree
from matplotlib import pyplot as plt
import csv
import numpy as np
from utils import load_from_csv
#Trees construction
train_examples, train_labels, features, prediction = load_from_csv("./compass.csv")
trees_config=[["best",5,100],["best",10,100],["best",5,500],["random",5,100],["random",20,50],["random",5,1000]]
for t in trees_config:
    clf = tree.DecisionTreeClassifier(splitter=t[0],max_depth=t[1],min_samples_split=t[2])
    clf = clf.fit(train_examples, train_labels)
    fig = plt.figure(figsize=(20,7))
    _ = tree.plot_tree(clf,
                       feature_names= (features),
                       class_names= (prediction),
                       filled=True)
    plt.show()
```

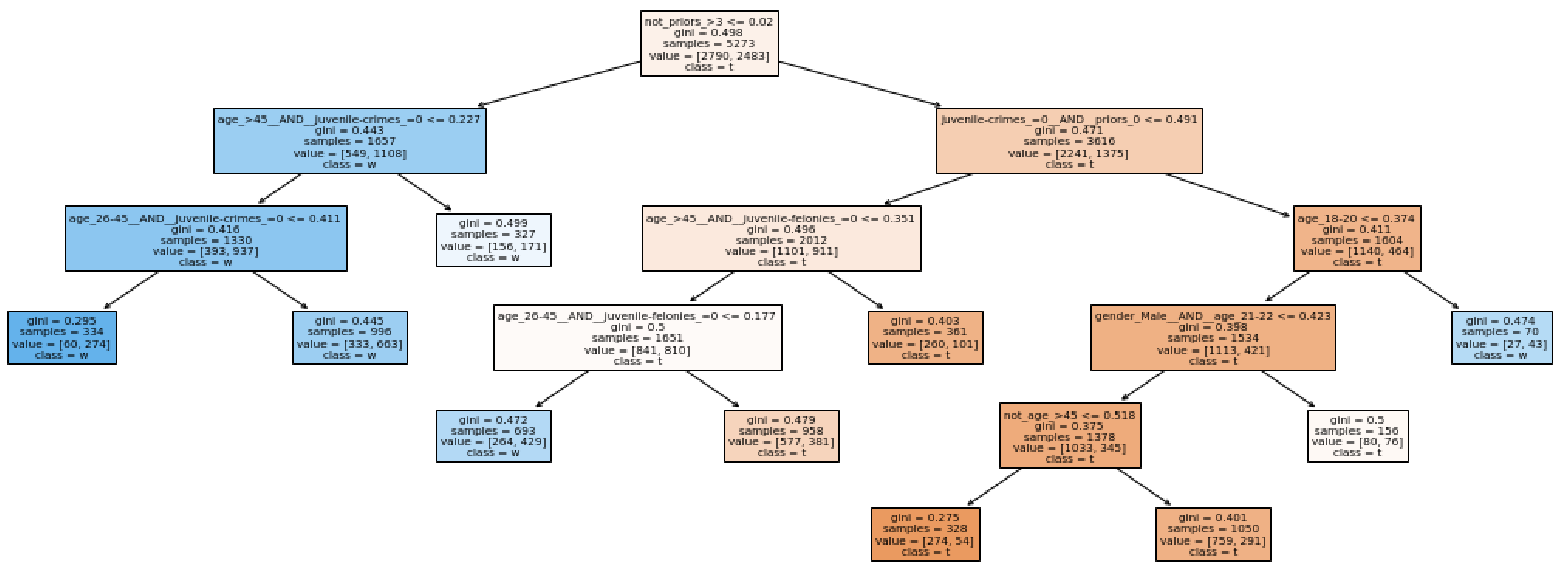





priors_>3 <= 0.5
gini = 0.498
samples = 5273
value = [2790, 2483]
class = t
juvenile-crimes_=0__AND__priors_0 <= 0.5
gini = 0.471
samples = 3616
value = [2241, 1375]
class = t
age_>45__AND__juvenile-crimes_=0 <= 0.5
gini = 0.443
samples = 1657
value = [549, 1108]
class = w
not_age_>45 <= 0.5
gini = 0.496
samples = 2012
value = [1101, 911]
class = t
not_age_18-20 <= 0.5
gini = 0.411
samples = 1604
value = [1140, 464]
class = t
age_26-45__AND__juvenile-crimes_=0 <= 0.5
gini = 0.416
samples = 1330
value = [393, 937]
class = w
gini = 0.499
samples = 327
value = [156, 171]
class = w
gini = 0.403
samples = 361
value = [260, 101]
class = t
age_26-45__AND__juvenile-felonies_=0 <= 0.5
gini = 0.5
samples = 1651
value = [841, 810]
class = t
gini = 0.474
samples = 70
value = [27, 43]
class = w
gender_Male__AND__age_21-22 <= 0.5
gini = 0.398
samples = 1534
value = [1113, 421]
class = t
gini = 0.295
samples = 334
value = [60, 274]
class = w
gender_Male__AND__charge degree_Felony <= 0.5
gini = 0.445
samples = 996
value = [333, 663]
class = w
age_23-25__AND__priors_1 <= 0.5
gini = 0.472
samples = 693
value = [264, 429]
class = w
priors_2-3__AND__charge degree_Felony <= 0.5
gini = 0.479
samples = 958
value = [577, 381]
class = t
age_>45__AND__priors_0 <= 0.5
gini = 0.375
samples = 1378
value = [1033, 345]
class = t
gini = 0.5
samples = 156
value = [80, 76]
class = t
gini = 0.466
samples = 355
value = [131, 224]
class = w
race_Caucasian <= 0.5
gini = 0.432
samples = 641
value = [202, 439]
class = w
gini = 0.446
samples = 540
value = [181, 359]
class = w
gini = 0.496
samples = 153
value = [83, 70]
class = t
gini = 0.451
samples = 621
value = [408, 213]
class = t
gini = 0.5
samples = 337
value = [169, 168]
class = t
gini = 0.401
samples = 1050
value = [759, 291]
class = t
gini = 0.275
samples = 328
value = [274, 54]
class = t
gini = 0.418
samples = 491
value = [146, 345]
class = w
gini = 0.468
samples = 150
value = [56, 94]
class = w





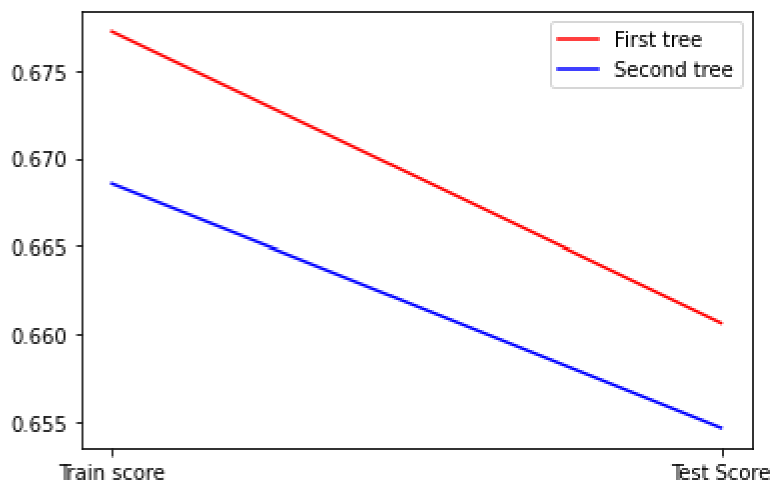

not_priors_>3 <= 0.02
gini = 0.498
samples = 5273
value = [2790, 2483]
class = t
age_>45__AND__juvenile-crimes_=0 <= 0.227
gini = 0.443
samples = 1657
value = [549, 1108]
class = w
juvenile-crimes_=0__AND__priors_0 <= 0.491
gini = 0.471
samples = 3616
value = [2241, 1375]
class = t
age_26-45__AND__juvenile-crimes_=0 <= 0.411
gini = 0.416
samples = 1330
value = [393, 937]
class = w
gini = 0.499
samples = 327
value = [156, 171]
class = w
age_>45__AND__juvenile-felonies_=0 <= 0.351
gini = 0.496
samples = 2012
value = [1101, 911]
class = t
age_18-20 <= 0.374
gini = 0.411
samples = 1604
value = [1140, 464]
class = t
gini = 0.295
samples = 334
value = [60, 274]
class = w
gini = 0.445
samples = 996
value = [333, 663]
class = w
age_26-45__AND__juvenile-felonies_=0 <= 0.177
gini = 0.5
samples = 1651
value = [841, 810]
class = t
gini = 0.403
samples = 361
value = [260, 101]
class = t
gender_Male__AND__age_21-22 <= 0.423
gini = 0.398
samples = 1534
value = [1113, 421]
class = t
gini = 0.474
samples = 70
value = [27, 43]
class = w
gini = 0.472
samples = 693
value = [264, 429]
class = w
gini = 0.479
samples = 958
value = [577, 381]
class = t
not_age_>45 <= 0.518
gini = 0.375
samples = 1378
value = [1033, 345]
class = t
gini = 0.5
samples = 156
value = [80, 76]
class = t
gini = 0.275
samples = 328
value = [274, 54]
class = t
gini = 0.401
samples = 1050
value = [759, 291]
class = t

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

The k-fold cross validation alrorithm is a great way to train the tree. It makes sure that the training is not biased towards a certain split and it helps us make the best out of our data in terms of training. The idea is to perform the training k times (in this case 5 times) by splitting the data into k folds each time.
We will choose two different trees in terms of topologies and characteristics. For each tree, we will use the k-fold cross validation algorithm to train it. We will then evaluate the training of each tree by measuring the score of its training set and its testing set.
Note: the training is done a certain number of times for each tree to make sure the final score is relevant
Tree model for the following bits code:
color:blue"> Tree=[splitter, max_depth, min_samples_split, nbr_kfold_splits, nbr_trainings]

In [3]:
```python
#Function taking a tree + parameters as input, returns train & test score of the tree
def evaluation (t):
    train_sum=0
    test_sum=0
    num_splits=t[3]
    trainings=t[4]
    kf = KFold(n_splits=num_splits)
    for i in range(0,trainings):
        clf = tree.DecisionTreeClassifier(splitter=t[0],max_depth=t[1],min_samples_split=t[2])
        for train_index, test_index in kf.split(train_examples):
            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 = clf.fit(x_train, y_train)
            train_sum+=clf.score(x_train, y_train)
            test_sum+=clf.score(x_test, y_test)
    train_score=train_sum/(trainings*num_splits)
    test_score=test_sum/(trainings*num_splits)
    return [train_score,test_score]
```

In [4]:
```python
#Training & Evaluation
from sklearn.model_selection import KFold
trees_config=[["best",5,100,5,100],["random",5,1000,5,100]]
scores=[]
for t in trees_config:
    scores.append(evaluation(t))
#Score visualisation
score_types=["Train score","Test Score"]
plt.plot(score_types,scores[0],'red',label='First tree')
plt.plot(score_types,scores[1],'blue',label='Second tree')
plt.legend()
plt.show()
```



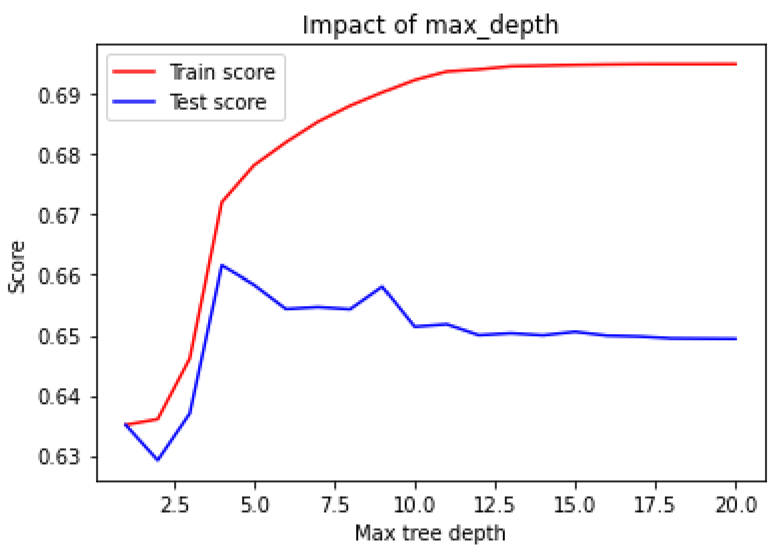
Propose an experimental study that shows the transition phase from underfitting to overfitting

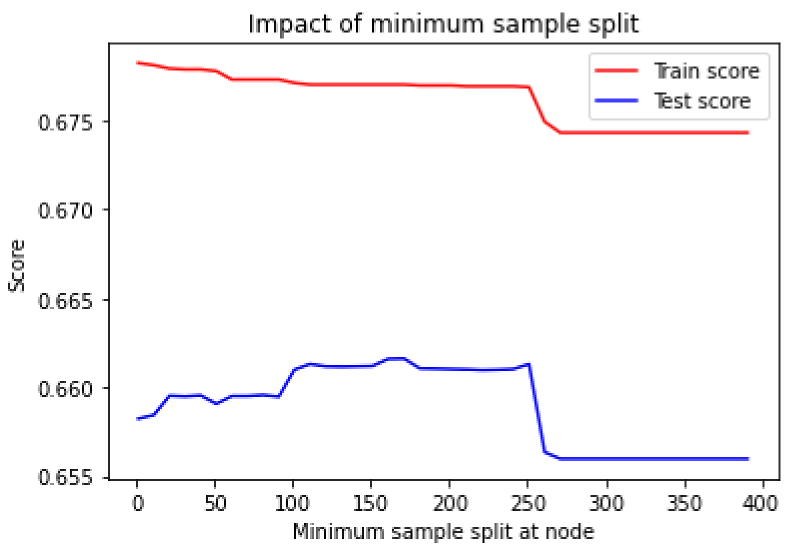
One way to visualize the transition phase from underfitting to overfitting is to see the influence of different parameters on trees.
We studied 3 parameters in the beginning: splitter, max_depth and min_samples_split. We now have a 4th parameter which is the number of splits when using the k-fold cross validation algorithm to train the trees.
The idea is to plot the train and test score of the tree according to those parameters to see how the score is impacted.

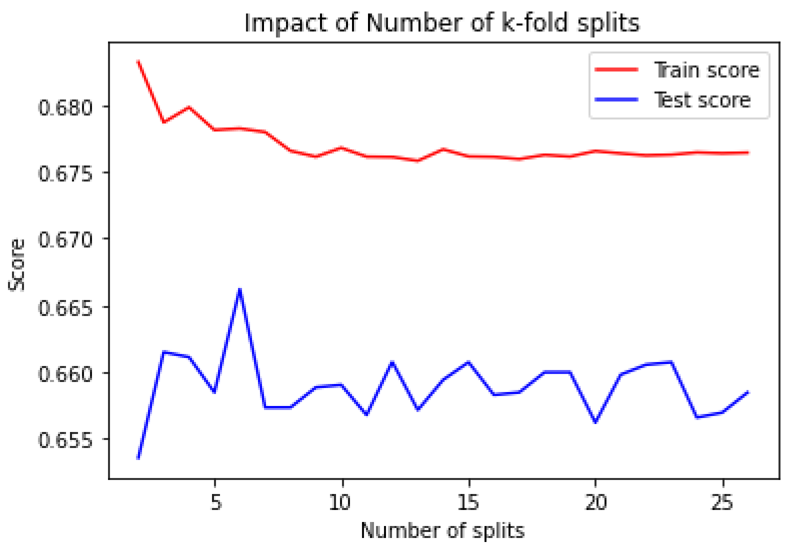
Note: the splitter parameter won't be part of the study as it is not very impactful score-wise.

In [5]:
```python
#Function that helps visualize tree scores according to a certain parameter
def score_plot(x_axis,scores,x_axis_label,title):
    plt.title(title)
    plt.plot(x_axis,extract0(scores),'red',label='Train score')
    plt.plot(x_axis,extract1(scores),'blue',label='Test score')
    plt.xlabel(x_axis_label)
    plt.ylabel('Score')
    plt.legend()
    plt.show()
#Function extracting all first/second elements of a list of lists
def extract0(lst):
    return [item[0] for item in lst]
def extract1(lst):
    return [item[1] for item in lst]
```

In [6]:
```python
#Studying impact of max_depth
max_depth=20
scores=[]
for i in range (1,max_depth+1):
    t=["best",i,10,5,10]
    scores.append(evaluation(t))
x_depth=np.arange(1,len(scores)+1)
score_plot(x_depth,scores,'Max tree depth',"Impact of max_depth")
#Studying impact of min_samples_split
max_samples=400
scores=[]
for i in range (2,max_samples+1,10):
    t=["best",5,i,5,10]
    scores.append(evaluation(t))
x_samples=np.arange(1,max_samples,10)
score_plot(x_samples,scores,'Minimum sample split at node',"Impact of minimum sample split")
#Studying impact of number of k-fold splits
max_splits=25
scores=[]
for i in range (2,max_splits+2):
    t=["best",5,5,i,1]
    scores.append(evaluation(t))
    x_splits=np.arange(2,max_splits+2)
score_plot(x_splits,scores,'Number of splits',"Impact of Number of k-fold splits")
```







As the graphs crearly show, there is a certain balance to hot when it comes to dealing with the three parameters that were studied.
The underfitting phase is recognizable by its low score and corresponds to the first part of the graph.
The best fit phase is recognizable by its peak of score (middle of graph)
The overfitting phase is recognizable by its "lower than the peak" score.
At this point, the testing score continues to decrease as the value of the parameter increases as the model takes into account even more noise than before.
The training score either increases or stays constant. Both score eventually stabilize.

Note: The phases of the "Impact of minimum sample split" graph are reversed since the less samples we have the more complex the model is

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

A good confusion matrix gives us a more detailed evaluation compared to the scores we studied earlier. In our case, we have a binarized dataset. So the confusion matrix is going to be a 2*2 matrix as we have two kinds of outputs: true or false.
The purpose is to see where the prediction failed or succeeded the most: either the "true" cases are better predicted than the "false" cases or the other way around. That way, instead of having an average score of the prediction, we know where the potential problem is if we

want to improve the model.

The key to choosing a good configuration to create a confusion matrix is to create a bias that makes the tree predict one outcome better than the other. Unfortunately, the dataset is too big to alter the data and create the bias manually.

However, we can create this bias by altering certain parameters.

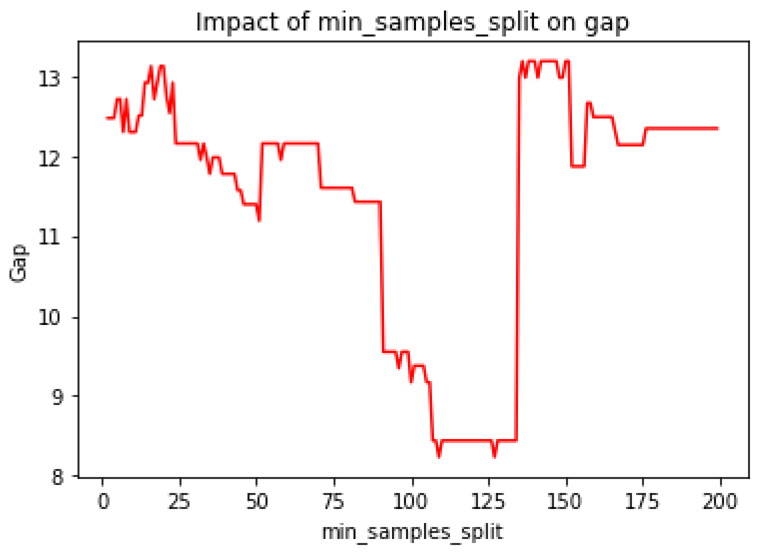
The following bits of code show the impact of these different parameters on the creation of the bias, the bias being a significant different between the true positives and the true negatives.

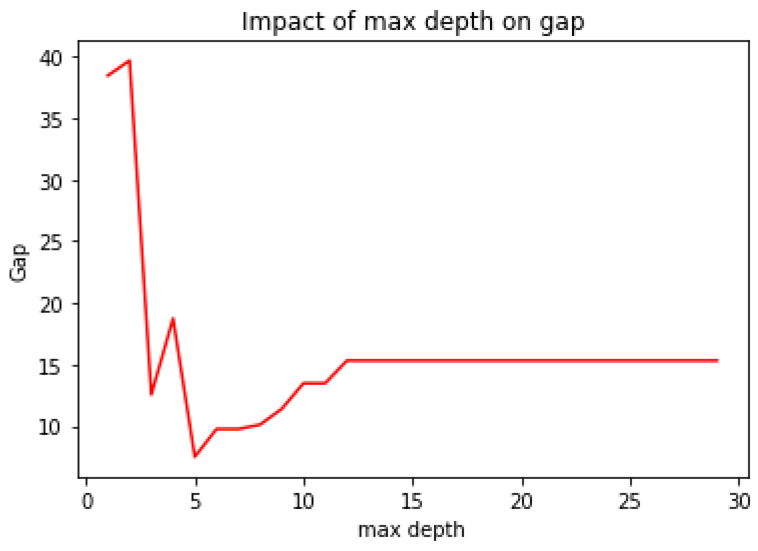
Tree model for the following bits code:

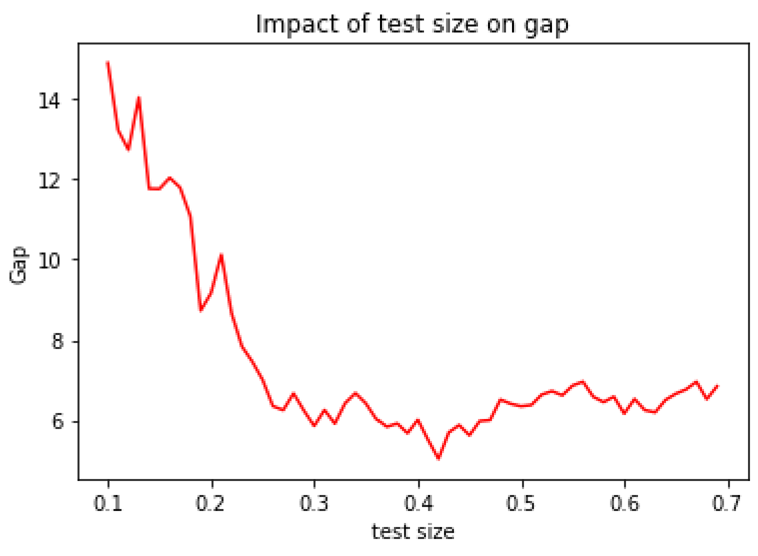
Tree = [max_depth, min_samples_split, test_size, random_state]

In [7]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
def confusion_matrix_gap(max_depth,min_samples_split,test_size,random_state):
    clf = tree.DecisionTreeClassifier(splitter='best',max_depth=max_depth,min_samples_split=min_samples_split)
    clf = clf.fit(train_examples, train_labels)
    x_train, x_test, y_train, y_test = train_test_split(train_examples, train_labels, test_size=test_size, random_state=r
    y_pred = clf.predict(x_test)
    cf_matrix = confusion_matrix(y_test, y_pred)
    true_positives=100*cf_matrix[0][0]/(cf_matrix[0][0]+cf_matrix[0][1])
    true_negatives=100*cf_matrix[1][1]/(cf_matrix[1][0]+cf_matrix[1][1])
    gap=abs(true_positives-true_negatives)
    return gap
```

In [8]:
```
#Function that helps visualize gap according to a certain parameter
def gap_plot(title,x_axis,xlabel,gap):
    plt.title(title)
    plt.plot(x_axis,gap,'red')
    plt.xlabel(xlabel)
    plt.ylabel('Gap')
    plt.show()
#Impact of min_samples_split on gap
gap=[]
x_samples=[]
for i in range(2,200):
    gap.append(confusion_matrix_gap(10,i,0.2,30))
    x_samples.append(i)
gap_plot('Impact of min_samples_split on gap',x_samples,'min_samples_split',gap)
#Impact of max depth on gap
gap=[]
x_depth=[]
for i in range(1,30):
    gap.append(confusion_matrix_gap(i+1,50,0.2,30))
    x_depth.append(i)
gap_plot('Impact of max depth on gap',x_depth,'max depth',gap)
#Impact of test size on gap
gap=[]
x_size=[]
test_sizes=np.arange(0.1,0.7,0.01)
for i in test_sizes:
    gap.append(confusion_matrix_gap(10,100,i,30))
    x_size.append(i)
gap_plot('Impact of test size on gap',x_size,'test size',gap)
```







The three first graphs give us conclusive results: the gap is significant for a certain value of the parameter. Now that we know how each parameter impacts the overall gap, we can create a decision tree with a nice gap.
We will also print its classification report in order to have even more information about the confusion matrix

Note: The "random state" parameter was not taken into account due to its non-significant impact on the gap

In [9]:
```
from sklearn.metrics import classification_report
clf = tree.DecisionTreeClassifier(splitter='best',max_depth=10,min_samples_split=100)
clf = clf.fit(train_examples, train_labels)
x_train, x_test, y_train, y_test = train_test_split(train_examples, train_labels, test_size=0.1, random_state=30)
y_pred = clf.predict(x_test)
cf_matrix = confusion_matrix(y_test, y_pred)
print("- Confusion matrix:")
print(cf_matrix)
true_positives=100*cf_matrix[0][0]/(cf_matrix[0][0]+cf_matrix[0][1])
true_negatives=100*cf_matrix[1][1]/(cf_matrix[1][0]+cf_matrix[1][1])
print("- True positives: ",true_positives,'%')
print("- True negatives: ",true_negatives,'%')
gap=abs(true_positives-true_negatives)
print("- Gap: ",gap)
target_names=['True','False']
print("- Classification report: ")
print(classification_report(y_test, y_pred,target_names=target_names))
```

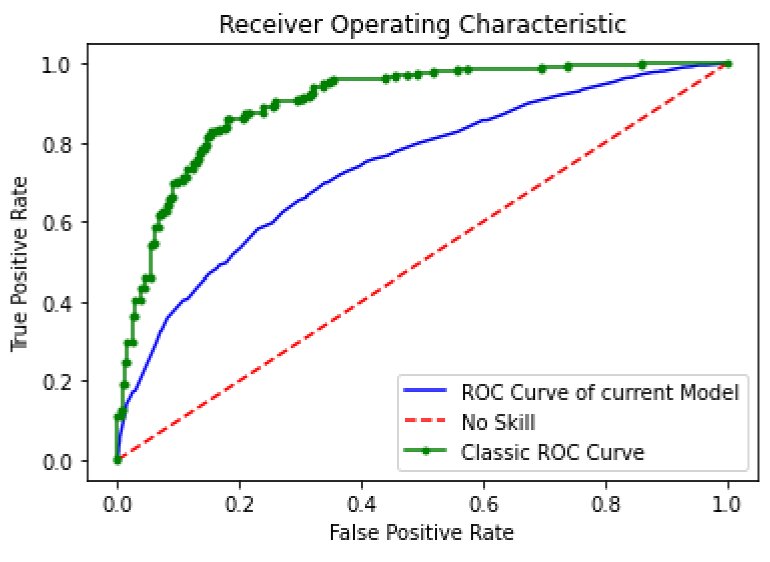
```
- Confusion matrix:
[[214  77]
 [ 98 139]]
- True positives:  73.53951890034364 %
- True negatives:  58.64978902953587 %
- Gap:  14.88972987080777
- Classification report:
              precision    recall  f1-score   support

        True       0.69      0.74      0.71       291
       False       0.64      0.59      0.61       237

    accuracy                           0.67       528
   macro avg       0.66      0.66      0.66       528
weighted avg       0.67      0.67      0.67       528
```

Precision tells us how many of the correctly predicted cases actually turned out to be positive.
Recall tells us how many of the actual positive cases we were able to predict correctly with our model.

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

The False Positive Rate (FP rate) determines the likelihood that a positive example is wrongly classified. This rate could be voluntarily increased by lowering the threshold (the classification "limiter") thus creating a bias. Depending on what our dataset represents, it could be important to increase the false positive rate, which means decreasing the true positive rate, or the other way around.
Let's say that in our case, it is important to incarcerate the people that are truly dangerous for society. This means that we want to minimize the number of incarcerated people to lower the chance of incarcerating innocent people. In this case, it is important to minizime the false positives to increase the true positives thus increasing the overall fearness of the model towards the convicted.
A good evaluation of the false positive rate is through a Receiver Operating Characteristic (ROC) curve.

In [44]:
```
from sklearn import metrics
import numpy as np
import matplotlib.pyplot as plt
train_examples, train_labels, features, prediction = load_from_csv("./compass.csv")
clf = tree.DecisionTreeClassifier(splitter='best',max_depth=10,min_samples_split=100)
clf = clf.fit(train_examples, train_labels)
x_train, x_test, y_train, y_test = train_test_split(train_examples, train_labels, test_size=0.2, random_state=0)
y_pred = clf.predict_proba(x_test)
y_pred=y_pred[:,1]
fpr, tpr, thresholds = metrics.roc_curve(np.array(y_test), np.array(y_pred))
# Print ROC curve
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label='ROC Curve of current Model')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")

#Classic ROC Curves
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
ns_probs = [0 for _ in range(len(testy))]
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
lr_probs = model.predict_proba(testX)
lr_probs = lr_probs[:, 1]
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs)
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs)
plt.plot(ns_fpr, ns_tpr, 'r', linestyle='--', label='No Skill')
plt.plot(lr_fpr, lr_tpr, 'g', marker='.', label='Classic ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```



The red diagonal line shows where True Positive Rate = False Positive Rate
Any point on this line represents a confusion matrix where the proportion of correctly classified samples is equal to the proportion of incorrectly classified samples. Our goal is to either make it so that the proportion of the correctly classified samples is greater that the proportion of incorrectly classified samples, or to simply minimize the false positives to to increase the overall fearness of the model.
Each point represents a new threshold, which translates to a new confusion matrix.
Let's go back to our original goal which is to minimize the proportion of false positives in order to incarcerate the least people. We can clearly see that the best tradeoff between decreasing the false positives and increasing the true positives is the middle portion of the graph at around FPR = 0.3 and TPR = 0.7.
This tradeoff is not very good. Classic ROC graphs (green curve) offer much better tradeoffs than this. We can conclude that the overall fearness of the model that we studied is not so great.

Lab 2 Evaluation

In [1]:
```
# Import some useful libraries and functions
import numpy as np
import pandas
def print_stats(dataset):
    """Print statistics of a dataset"""
    print(pandas.DataFrame(dataset).describe())
```

In [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
```

Explore the dataset

- How many examples are there in the dataset? There are 308 examples in the dataset (308 rows)
- How many features for each example? There are 6 features (7 columns), the right-most column being the result of each example
- What is the ground truth of the 10th example? The ground truth of this example is 1.83

In [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 only contains ones.
- Why are we doing this? This value represents the "w0" in the formula of the affine function: y = w0 + x1w1 +x2w2 + ... + xnwn. This value needs to be initialized and will be changed afterwards (with the evolution of the algorithm). It avoids having only linear functions, as we want affine functions.

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

Creating the perceptron

In [5]:
```
w = np.zeros(7)

def h(w, x):
    sum = 0
    for i in range(0,len(x)-1):
        sum+=x[i]*w[i]
    return sum
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth

# print the ground truth and the evaluation of h_w on the first example
print("--First example-- \nground truth:", Y[0], "\nevaluation of h_w:", h(w, X[0]))
```

```
--First example--
ground truth: 8.62
evaluation of h_w: 0.0
```

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`.

In [6]:
```
def loss(w, x, y):
    return (y-h(w,x))**2 #L2(a,b)=(a-b)^2
```

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.

In [7]:
```
def emp_loss(w, X, Y):
    sum=0
    for i in range(len(X)):
        sum+=(loss(w,X[i],Y[i]))
    return sum/len(X)
```

Gradient update

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

In [8]:
```
def compute_update(w, X, Y, alpha):
    dw = np.zeros(len(w))
    for i in range(0,len(w)-1):
        sum=0
        for j in range(0,len(X)-1):
            sum+=(Y[j]-h(w,X[j]))*X[j][i]
        dw[i]=sum*alpha
    return dw

compute_update(w, X, Y, alpha = 10e-7)
```

Out[8]:
```
array([-0.00754539,  0.00181636,  0.01543604,  0.01266542,  0.01033756,
        0.00130721,  0.        ])
```

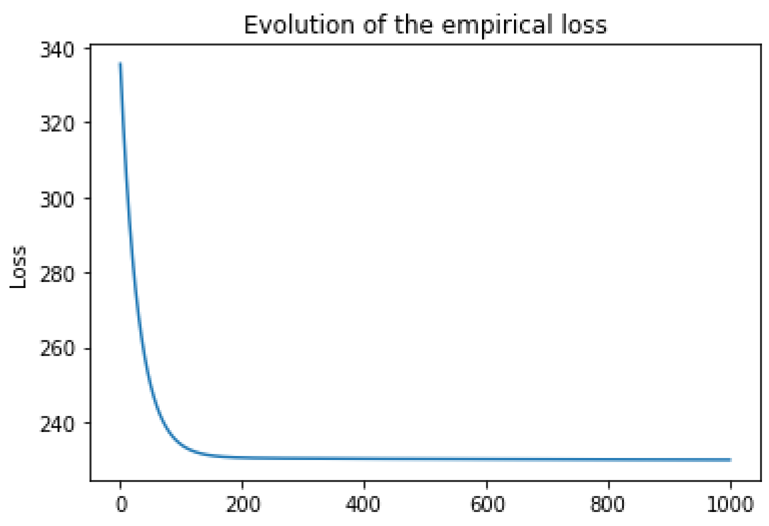
Gradient descent

In [9]:
```
def descent(w_init, X, Y, alpha, max_iter):
    w=np.copy(w_init)
    for i in range(max_iter):
        for j in range (len(w)):
            w[j]=w[j]+compute_update(w, X, Y, alpha)[j]
    return w
```

Exploitation

- Train your perceptron to get a model.
- Visualize the evolution of the loss on the training set. Has it converged? Yes, we can see on the graph below that the loss converged.

In [30]:
```
from matplotlib import pyplot as plt
import statistics

#Same function as descent function but prints the empiral loss for each iteration
def descent_printing_loss(w_init, X, Y, alpha, max_iter):
    l=[]
    w=np.copy(w_init)
    for i in range(max_iter):
        for j in range (len(w)):
            w[j]=w[j]+compute_update(w, X, Y, alpha)[j]
        l.append(emp_loss(w,X,Y))
    plt.title("Evolution of the empirical loss")
    plt.plot(l)
    plt.ylabel('Loss')
    plt.show()
    return w

w_init = np.zeros(7)
descent_printing_loss(w_init, X, Y, 10e-7, 1000)
```



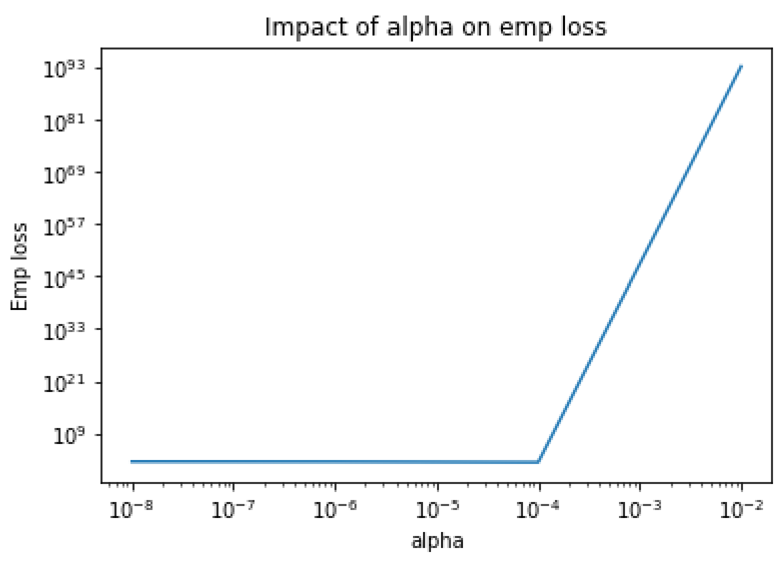
Out[30]:
```
array([-0.18285906,  0.11883795,  1.00261101,  0.74904522,  0.67785455,
        0.4424448 ,  0.        ])
```

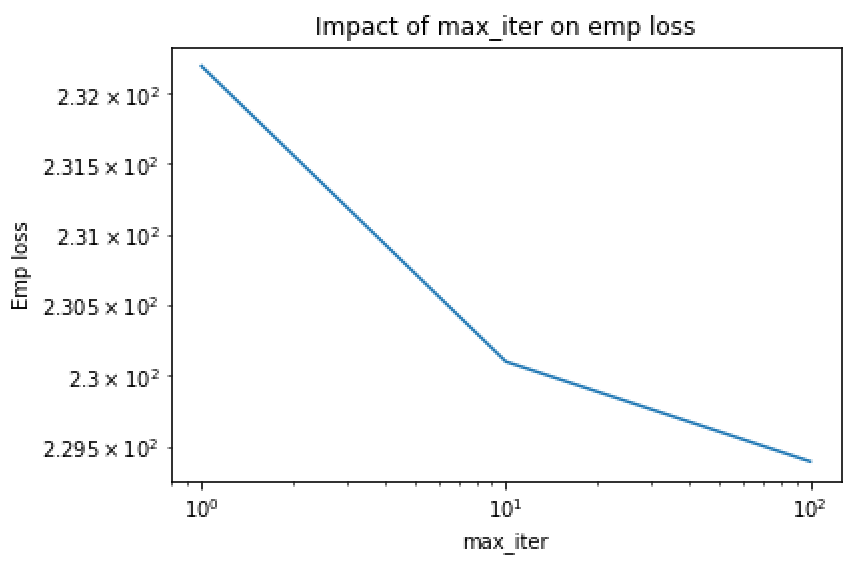
- Try training for several choices of `alpha` and `max_iter` . What seem like a reasonable choice? As we can see on the graphs below, increasing the number of max_iter enables to obtain a better result, but it is time consuming. We decided that max_iter = 100 is a reasonable choice to have a precise model but that does not need too much time to execute. Concerning alpha, we chose 10e-5 to have a compromise between execution time and precision.

In [10]:
```
#Function that constructs plots based on a parameter and the impact of that parameter on weights
from matplotlib import pyplot as plt
def construct_plt(X,Y,parameter,x_axis):
    w_init = np.zeros(7)
    emp_loss_list=[]
    if x_axis=='alpha':
        for i in parameter:
            w = descent(w_init,X,Y,i,10)
            emp_loss_list.append(emp_loss(w,X,Y))
    else:
        for i in parameter:
            w = descent(w_init,X,Y,10e-5,i)
            emp_loss_list.append(emp_loss(w,X,Y))
    plt.xscale('log')
    plt.yscale('log')
    plt.plot(parameter,emp_loss_list)
    plt.xlabel(x_axis)
    plt.ylabel("Emp loss")
    plt.title("Impact of "+x_axis+" on emp loss")
    plt.show()
```

In [11]:
```
def data_analysis(X,Y):
    #Impact of alpha on the weights (the Y axis represents an average on the values in w)
    alphas=[10e-3,10e-5,10e-7,10e-9]
    construct_plt(X,Y,alphas,'alpha')

    #Impact of max_iter on the weights (the Y axis represents an average on the values in w)
    max_iters=[1,10,100]
    construct_plt(X,Y,max_iters,'max_iter')
```

In [12]:
```
%%time
data_analysis(X,Y)
```



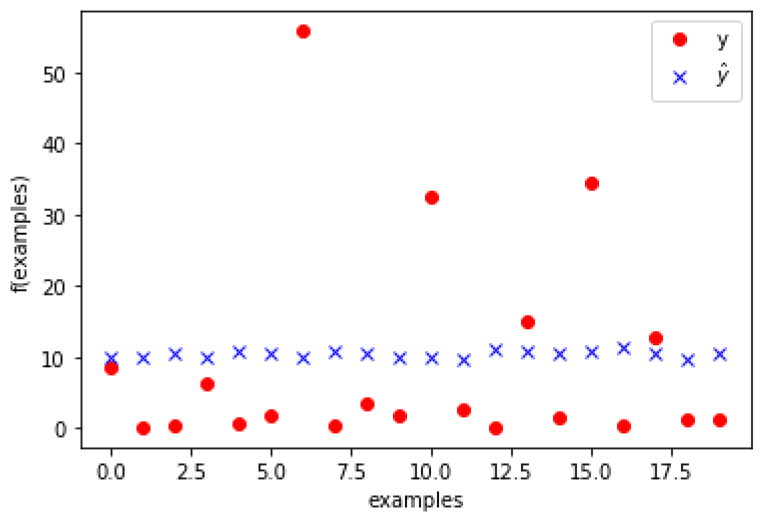


```
Wall time: 5.42 s
```

- What is the loss associated with the final model? The loss associated with our final model is 230.
- Is the final model the optimal one for a perceptron? This final model is not optimal as the loss associated with it is not 0. It means that not all our predictions are accurate; our model is not perfect.

In [13]:
```
# Code sample that can be used to visualize the difference between the ground truth and the prediction
import matplotlib.pyplot as plt

num_samples_to_plot = 20

w_init = np.zeros(7)
w_init = descent(w_init, X, Y, 10e-5, 100)
yw = [h(w_init,x) for x in X]

plt.plot(Y[0:num_samples_to_plot], 'ro', label='y')
plt.plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')

plt.legend()
plt.xlabel("examples")
plt.ylabel("f(examples)")

print("Loss associated: ", emp_loss(w_init, X, Y))
```

```
Loss associated:  229.39723048470043
```

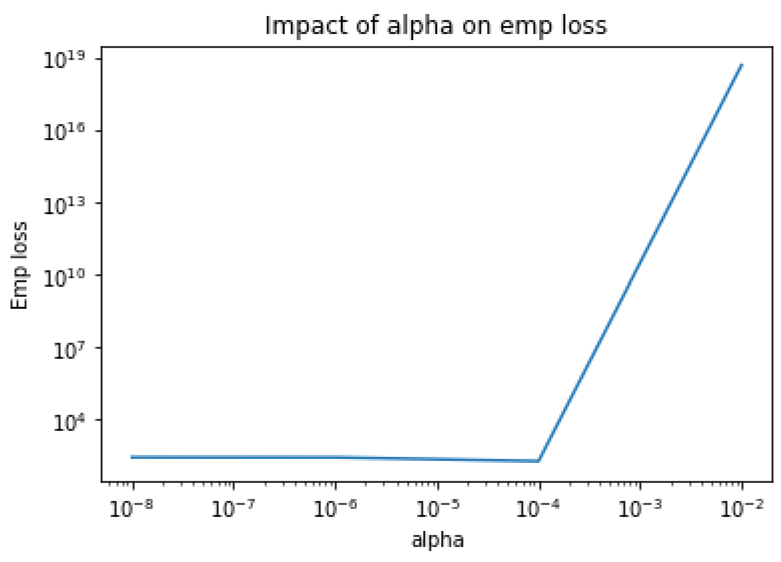


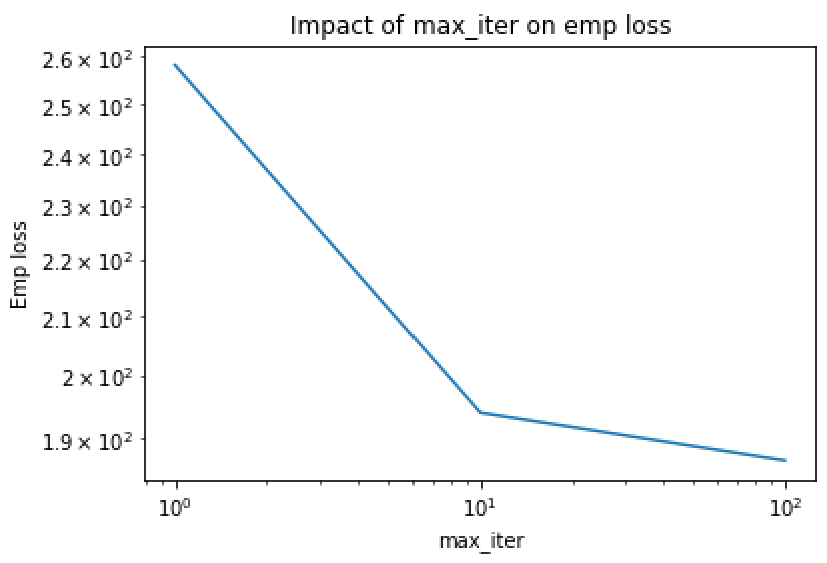
Going further

The Stochastic Gradient Descent algorithm is an improvement over the basic Gradient Descent algorithm in terms of computation time. Indeed, by selecting only a subset of the examples studied, the calculations could be significantly reduced. The subset is selected randomly so it is supposed to represent the dataset well.
Note: Stochastic means Random

There are two simple ways to proceed. Either we select random points of the dataset every time we calculate the descent (each time the w vector is updated) with less iterations than the original algorithm, or we could make a smaller randomized copy of the dataset and work with that instead. The second solution being the simplet, that's how we're going to proceed

In [14]:
```
%%time

#We have 308 examples in X. Let's make it 25 random examples
import random
alphas=[10,10e-3,10e-5,10e-7,10e-9]
max_iters=[1,10,100]
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth
randX=[]
randY=[]

for i in range (0,25):
    rand=random.choice(dataset)
    randX.append(rand[:-1])
    randY.append(rand[-1])
#Analysis
data_analysis(randX,randY)
```





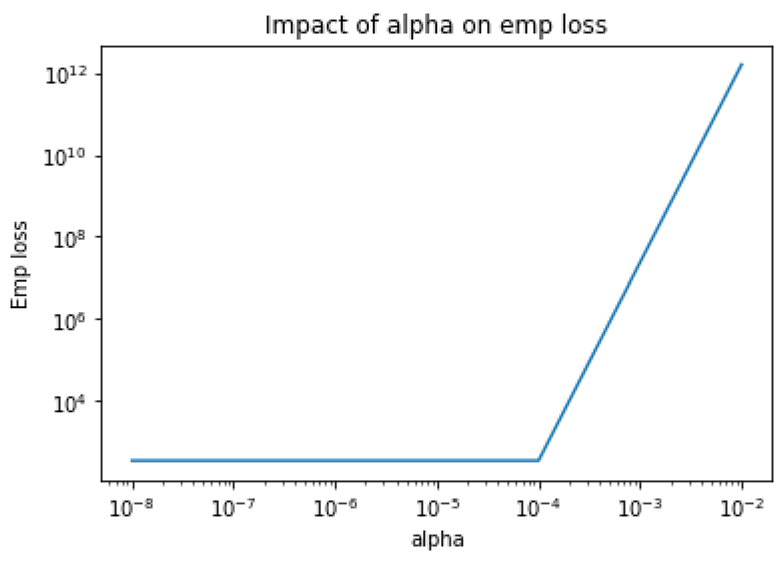
```
Wall time: 1.02 s
```

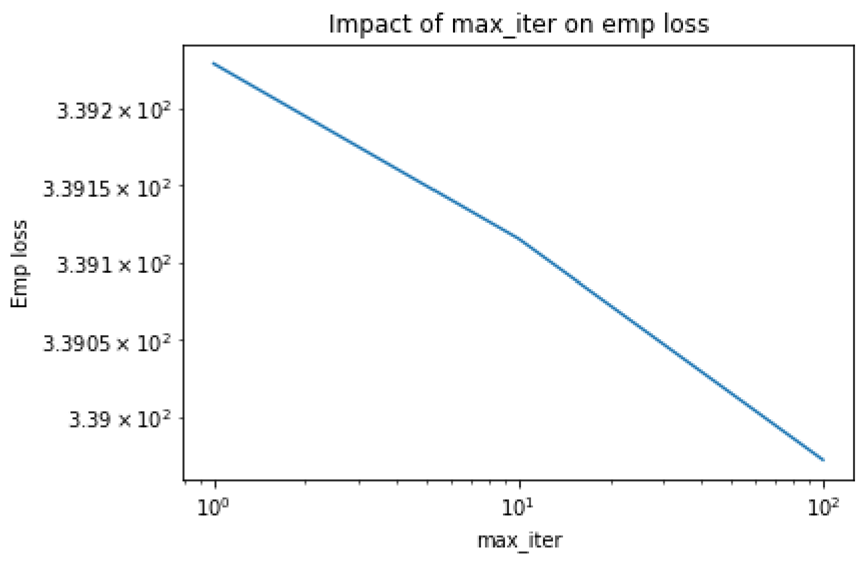
The plots seem to be fairly similar. Also, we can see that we went from an execution time of 29.1 s to 3.56 s

Data normalization

In [15]:
```
%%time

from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler(copy=True)
X_normalized = sc.fit_transform(X)
#Analysis
data_analysis(X_normalized,Y)
```





```
Wall time: 6.04 s
```

By comparing the execution time of the two cells, the Data Normalization on X seems to have no impact on the speed on the convergence

```
In [ ]:
```

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.

In [4]:
```
# Importing the dataset
import numpy as np
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1]
Y = dataset[:, -1]
```

In [5]:
```
# Preprocessing: scale input data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)

#Explanation
#These functions are applied to normalize X by removing the mean value and dividing by the variance
#First, the fit function calculates the mean and the variance and then the standardscaler applies them
```

In [6]:
```
# 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)
```

In [7]:
```
# 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
```

Out[7]:
```
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
             beta_2=0.999, early_stopping=False, epsilon=1e-08,
             hidden_layer_sizes=(100,), learning_rate='constant',
             learning_rate_init=0.001, max_iter=3000, momentum=0.9,
             n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
             random_state=1, shuffle=True, solver='adam', tol=0.0001,
             validation_fraction=0.1, verbose=False, warm_start=False)
```

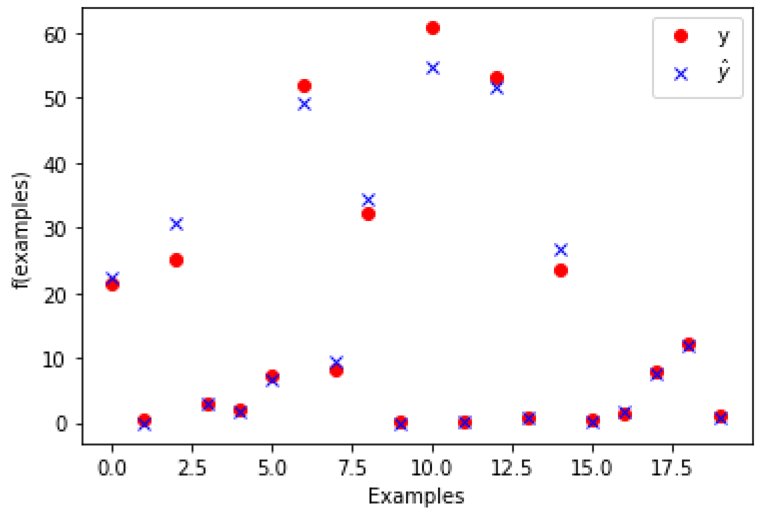
In [8]:
```
# Evaluate the model
from matplotlib import pyplot as plt

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:  0.9940765369322632
Test score:   0.9899773031580282
```

Out[8]:
```
Text(0, 0.5, 'Loss')
```

In [9]:
```
# 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)")

#Explanation :
#The red circles are the "real values"
#The blue crosses are the ones predicted by the MLP
```

Out[9]:
```
Text(0, 0.5, 'f(examples)')
```



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?
  It is a neural network.
- What it is the algorithm used for training? Is there algorithm available that we mentioned during the courses?
  The default algorithm is 'adam'. The stochastic gradient descent (SGD) algorithm that we studied during the courses is also available.
- How does the training algorithm decides to stop the training?
  It stops when the maximum number of iterations is reached. If the early stoping is activated, it can also stop when no improvment is done to the validation score.

Onto a more challenging dataset: house prices

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

In [10]:
```
# clean all previously defined variables for the sailing boats
%reset -f
```

In [11]:
```
"""Import the required modules"""
from sklearn.datasets import fetch_california_housing
import pandas as pd
import numpy as np

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
```

Out[11]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25
7	3.1200	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25
8	2.0804	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26
9	3.6912	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25

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.

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`.

In [12]:
```
from sklearn import model_selection
# We will split the data after scaling them
# X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.25, random_state=42)
```

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

In [13]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.25, random_state=42)
X_test, X_validation, y_test, y_validation = model_selection.train_test_split(X_test, y_test, test_size=0.25, random_stat

#We did the standardization before splitting the data to ensure that exactly the same transformation is applied to both a
```

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.

In [14]:
```
from sklearn.neural_network import MLPRegressor

#Default parameters
mlp = MLPRegressor(max_iter=5000, random_state=1)
mlp.fit(X_train, y_train)
print("default parameters")
print('Train score: ', mlp.score(X_train, y_train))
print('Test score:  ', mlp.score(X_test, y_test), "\n")


#Increasing the max iterations number
# -> still the same score as the algorithm stops before reaching this max number of iterations
mlp = MLPRegressor(max_iter=10000, random_state=1)
mlp.fit(X_train, y_train)
print("increasing max_iter")
print('Train score: ', mlp.score(X_train, y_train))
print('Test score:  ', mlp.score(X_test, y_test), "\n")


#Increasing the number of iterations with no change (from default 10 to 100)
# -> The score increases too
mlp = MLPRegressor(max_iter=5000, random_state=1, n_iter_no_change=100)
mlp.fit(X_train, y_train)
print("increasing n_iter_no_change")
print('Train score: ', mlp.score(X_train, y_train))
print('Test score:  ', mlp.score(X_test, y_test), "\n")


#Adding layers and neurons in each layer
mlp = MLPRegressor(max_iter=5000, random_state=1, hidden_layer_sizes=(500,800))
mlp.fit(X_train, y_train)
print("icreasing hidden_layer_sizes")
print('Train score: ', mlp.score(X_train, y_train))
print('Test score:  ', mlp.score(X_test, y_test), "\n")
```

```
default parameters
Train score:  0.8452851538383562
Test score:   0.7375328102591283

increasing max_iter
Train score:  0.8452851538383562
Test score:   0.7375328102591283

increasing n_iter_no_change
Train score:  0.9076256338554253
Test score:   0.7485330155057166

icreasing hidden_layer_sizes
Train score:  0.9054794103484964
Test score:   0.7658520507413025
```

Conclusion

We can see that increasing n_iter_no_change or hidden_layer_sizes allow us to have a train score > 0.9. But when we increase the hidden_layer_sizes it takes more times to execute.

Definitions : Overfitting means that "*our model doesn't generalize well from our training data to unseen data*".

Answer to the questions :

- Is the **test** score substantially smaller than the **train** score (indicator of overfitting) ?
  The test score is way smaller than the train score. The test score is still between 0.75 and 0.78 even when we maximize the train score.
- Explain how the parameters you chose allow the learned model to overfit.
  Here, the parameters we choose allow the learned model to overfit because we just change when the algorithm will stop or the size of hidden layers. So we allow our model to learn the nose too. For exemple when we add layers and neurons (when we increase the hidden_layer_sizes) we permits to the model to learn more, but in that way the model learn also the nose. But when a model do not distinct clearly the nose from the signal, it can provide good result on the test score. But, we think that a test score of 0.77 is not that bad for the begining.

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)

Definitions

from sklearn.neural_network.MLPRegressor documentation

- number and size of the hidden layers
  hidden_layer_sizes : The ith element represents the number of neurons in the ith hidden layer.

- activation function
  activation : Activation function for the hidden layer.

- stopping conditions
  early_stopping : Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as **validation** and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs. Only effective when solver='sgd' or 'adam'.
  validation_fraction : The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.
  n_iter_no_change : Maximum number of epochs to not meet tol improvement. Only effective when solver='sgd' or 'adam'.

- maximum number of iterations
  max_iter : Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

- initial learning rate value
  learning_rate_init : The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

In [16]:
```python
# TODO

import time
import random

hidden_layer_sizes=[(100,),(250,400),(500,800),(750,1200)]
activation=['identity', 'logistic', 'tanh', 'relu']
early_stopping =[False,True]
validation_fonction=[0.1,0.2,0.3,0.4,0.5]
n_iter_no_change=[10,40,70,100]
max_iter=[1000,2500,5000]
learning_rate_init=[0.001,0.005,0.01,0.1]
data = []

#TOO MUCH TIME for all the possibilities !!!!
#for i in range(len(hidden_layer_sizes)):
#    for j in range(len(activation)):
#        for k in range(len(early_stopping)):
#            for l in range(len(validation_fonction)):
#                for m in range(len(n_iter_no_change)):
#                    for n in range(len(max_iter)):
#                        for o in range(len(learning_rate_init)):

for test in range(25):
    i=random.choice(hidden_layer_sizes)
    j=random.choice(activation)
    k=random.choice(early_stopping)
    l=random.choice(validation_fonction)
    m=random.choice(n_iter_no_change)
    n=random.choice(max_iter)
    o=random.choice(learning_rate_init)
    mlp = MLPRegressor(hidden_layer_sizes=i,
                       activation= j,
                        early_stopping = k,
                        validation_fraction=l,
                        n_iter_no_change=m,
                        max_iter= n,
                        learning_rate_init=o
                        )

    start = time.time()
    mlp.fit(X_train, y_train)
    stop = time.time()

    data.append({'hidden_layer_sizes':i,
                 'activation':j,
                 'early_stopping': k,
                 'validation_fraction':l,
                 'n_iter_no_change':m,
                 'max_iter': n,
                 'learning_rate_init':o,
                 'validation_score': mlp.score(X_validation, y_validation),
                 'training time': stop - start
                })
    print("Test n°"+str(test)+" ok !")

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

```
Test n°0 ok !
Test n°1 ok !
Test n°2 ok !
Test n°3 ok !
Test n°4 ok !
Test n°5 ok !
Test n°6 ok !
Test n°7 ok !
Test n°8 ok !
Test n°9 ok !
Test n°10 ok !
Test n°11 ok !
Test n°12 ok !
Test n°13 ok !
Test n°14 ok !
Test n°15 ok !
C:\Users\morga\Anaconda3\lib\site-packages\sklearn\neural_network\multilayer_perceptron.py:566: ConvergenceWarning: Stoch
astic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
Test n°16 ok !
Test n°17 ok !
C:\Users\morga\Anaconda3\lib\site-packages\sklearn\neural_network\multilayer_perceptron.py:566: ConvergenceWarning: Stoch
astic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
Test n°18 ok !
Test n°19 ok !
Test n°20 ok !
Test n°21 ok !
Test n°22 ok !
Test n°23 ok !
Test n°24 ok !
```

Out[16]:

	hidden_layer_sizes	activation	early_stopping	validation_fraction	n_iter_no_change	max_iter	learning_rate_init	validation_score	training time
6	(250, 400)	relu	True	0.4	100	5000	0.001	0.816452	7.117100
12	(500, 800)	relu	False	0.5	40	1000	0.005	0.807638	80.077527
3	(250, 400)	relu	True	0.3	10	5000	0.010	0.799975	2.288840
5	(100,)	tanh	False	0.4	70	5000	0.100	0.798727	1.032391
13	(500, 800)	logistic	False	0.5	70	5000	0.010	0.790377	433.347729
21	(250, 400)	tanh	True	0.4	100	5000	0.005	0.788263	8.188252
9	(500, 800)	logistic	False	0.4	100	5000	0.010	0.785270	449.787747
18	(100,)	logistic	False	0.1	40	1000	0.001	0.785041	6.272222
15	(250, 400)	relu	False	0.3	10	2500	0.010	0.780960	2.741719
16	(100,)	logistic	False	0.2	40	1000	0.001	0.779398	6.475277
22	(100,)	logistic	True	0.4	70	5000	0.010	0.776780	1.480161
11	(100,)	identity	True	0.1	100	2500	0.010	0.746611	0.521628
7	(100,)	identity	True	0.5	100	5000	0.010	0.745558	0.509456
20	(750, 1200)	identity	False	0.5	40	1000	0.005	0.737936	36.210609
4	(500, 800)	identity	True	0.5	10	2500	0.010	0.737561	5.239119
14	(100,)	identity	True	0.1	100	1000	0.001	0.737559	0.702634
1	(250, 400)	logistic	False	0.1	40	2500	0.100	0.733053	23.587156
19	(250, 400)	identity	True	0.2	40	1000	0.005	0.724370	2.623107

	hidden_layer_sizes	activation	early_stopping	validation_fraction	n_iter_no_change	max_iter	learning_rate_init	validation_score	training time
17	(100,)	identity	True	0.1	70	2500	0.100	0.723209	0.627318
2	(250, 400)	identity	False	0.5	10	1000	0.005	0.720608	1.379661
0	(250, 400)	relu	True	0.3	100	1000	0.100	0.708832	8.519260
10	(750, 1200)	identity	False	0.3	70	5000	0.010	0.706537	75.664499
23	(750, 1200)	tanh	True	0.5	40	2500	0.010	0.649939	40.392035
24	(500, 800)	relu	False	0.5	10	5000	0.100	0.513370	14.941002
8	(500, 800)	logistic	False	0.4	70	2500	0.100	0.389158	82.111200

Evaluation

- From your experiments, what seems to be the best model (i.e. set of parameters) for predicting the value of a house?

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.

Response

We already split our data set on 3 to have a training set, a validation set and a test set.
25% of our initial dataset constitute the test set. And 25% of our train dataset constitute the validation test (0.1875 of the initial dataset). So the train dataset represent 0.5625 of the initial dataset.

From our experiments the best model seems to be

```
MLPRegressor(hidden_layer_sizes=(250, 400),
                         activation= relu,
                          early_stopping = True,
                          validation_fraction=0.4,
                          n_iter_no_change=100,
                          max_iter=5000 ,
                          learning_rate_init=0.001
                          )
```

With a validation score = 0.816452.
We will now evalutate our model with the test set.

In [18]:
```
mlp = MLPRegressor(hidden_layer_sizes=(250, 400),
                        activation= 'relu',
                        early_stopping = True,
                        validation_fraction=0.4,
                        n_iter_no_change=100,
                        max_iter=5000 ,
                        learning_rate_init=0.001
                        )

start = time.time()
mlp.fit(X_train, y_train)
stop = time.time()

print("Result")
print('Test score:  ', mlp.score(X_test, y_test), "\n")
```

```
Result
Test score:   0.7408860206482837
```

Conclusions

We have a final test score at 0.740 that is predictably less than previously but it still a very good score !