Feature Selection

The purpose of feature selection (or dimensionality reduction) is to (i) avoid the curse of dimensionality, (ii) reduce the amount of time and memory required by the data mining algorithms, (iii) help eliminating irrelevant features, (iv) reduce noise, and possibly (v) allow data to be more easily visualized.

Feature selection methods can be categorized in three main classes:

- Embedded approaches
- Filter approaches
- Wrapper approaches

In **embedded approaches**, feature selection naturally occurs as part of the data mining algorithm. Lasso and Ridge regression are examples of algorithms that embed feature selection. With **filter approaches**, the features are selected using a procedure that is independent from a specific data mining algorithm. For example, Principal Component Analysis (PCA) is an example of feature selection based on a filter approach. Another example is the selection of attributes based on a correlation measure. **Wrapper approaches** apply a data mining algorithm as a black box to find best subset of attributes. For example, we might apply greedy-search or a genetic algorithm to find the best set of features for a decision tree. Alternatively, we might apply a **brute-force** approach and try all possible feature subsets to find the best one for a specific mining algorithm.

In this notebook we discuss some examples of embedded, filter, and wrapper approaches to feature selection. There are other notebooks that deal with wrapper approaches. We will use the Boston Housing dataset. For further details on the function used we refer you to the Scikit-Learn page on feature selection.

As usual, we begin by importing all the necessary libraries, we load the data, and set the random seed.

In [2]:
```
import numpy as np
import pandas as pd
import random

from sklearn import datasets
from sklearn import linear_model
from sklearn import naive_bayes
from sklearn import neighbors
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.ensemble import ExtraTreesRegressor

from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import RFE

from sklearn.decomposition import PCA

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl


plt.style.use('fivethirtyeight')
mpl.rcParams['lines.linewidth'] = 2
# mpl.rcParams['axes.labelsize'] = 14
# mpl.rcParams['xtick.labelsize'] = 12
# mpl.rcParams['ytick.labelsize'] = 12
mpl.rcParams['text.color'] = 'k'

%matplotlib inline
```

In [3]:
```
data = datasets.load_boston()

X = data["data"]
y = data["target"]

input_variables = data.feature_names
target_variable = 'MEDV'

seed = 1234

# Let's create also a pandas data frame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['MEDV'] = y
df.head()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Since we are using regression approaches that are sensitive to the variable range, we need to normalized the input variables.

In [4]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

Xs = scaler.fit_transform(data["data"])

dfs = pd.DataFrame(Xs, columns=data.feature_names)
```

```
dfs['MEDV'] = y
dfs.head()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	-0.419782	0.284830	-1.287909	-0.272599	-0.144217	0.413672	-0.120013	0.140214	-0.982843	-0.666608	-1.459000	0.441052	-1.075562	24.0
1	-0.417339	-0.487722	-0.593381	-0.272599	-0.740262	0.194274	0.367166	0.557160	-0.867883	-0.987329	-0.303094	0.441052	-0.492439	21.6
2	-0.417342	-0.487722	-0.593381	-0.272599	-0.740262	1.282714	-0.265812	0.557160	-0.867883	-0.987329	-0.303094	0.396427	-1.208727	34.7
3	-0.416750	-0.487722	-1.306878	-0.272599	-0.835284	1.016303	-0.809889	1.077737	-0.752922	-1.106115	0.113032	0.416163	-1.361517	33.4
4	-0.412482	-0.487722	-1.306878	-0.272599	-0.835284	1.228577	-0.511180	1.077737	-0.752922	-1.106115	0.113032	0.441052	-1.026501	36.2

Baseline Performance

Let's begin by computing a baseline for the dataset containing all the features. We will use this baseline performance to compare the predictive power of the feature subsets computed by all the methods discussed in this notebook.

The performance of basic linear regression using a ten-fold crossvalidation is 0.682.

In [5]:
```
kfolds = KFold(10,shuffle=True,random_state=seed)

model = linear_model.LinearRegression()
scores = cross_val_score(model, X, y, cv=kfolds)

print("R2 Mean %.3f StdDev %.3f"%(scores.mean(),scores.std()))
```

we should have used the standardized version of the data (scaled version)

```
R2 Mean 0.682 StdDev 0.125
```

1. Reduced Variance Feature Selection

This feature selection algorithm computes the variance of all the variables and eliminates the variables with a variance below a certain threshold. The rationale is that, if the variance is very low, the values of the feature are very similar to each other so we may eliminate the variable.

In the example below, the threshold is set as the variance of a Bernoulli distribution of a given probability (0.8 in this example).

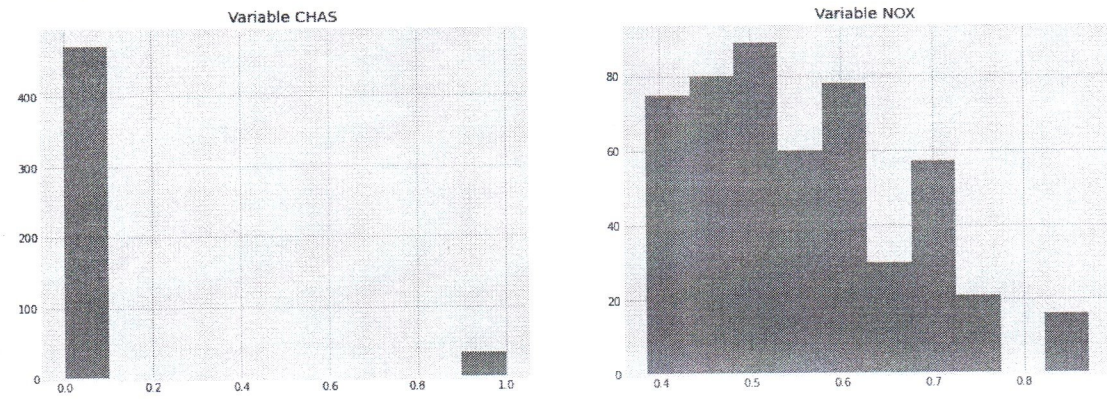
In [6]:
```
feature_selection_variance_model = VarianceThreshold(threshold=(.8 * (1 - .8)))
X_selected_features_variance = feature_selection_variance_model.fit_transform(X)


mask = feature_selection_variance_model.get_support() #list of booleans
print("Reduced data set shape = ", X_selected_features_variance.shape)
print("     Selected features = ", data.feature_names[mask])
print("      Deleted Features = ", data.feature_names[~mask])
```

Notice that we may have a variable which has very low variance but that variance is very important → the decision is on us, it is not an automatic process
Also: we set the treshold. Here we use the variance of a bernoulli distribution

```
Reduced data set shape =  (506, 11)
     Selected features =  ['CRIM' 'ZN' 'INDUS' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO' 'B' 'LSTAT']
      Deleted Features =  ['CHAS' 'NOX']
```

In [7]:
```
plt.subplots(1,2,figsize=(16,6))

plt.subplot(1,2,1)
plt.hist(df['CHAS'],label='CHAS')
plt.title("Variable CHAS")

plt.subplot(1,2,2)
plt.hist(df['NOX'],label='NOX')
plt.title("Variable NOX")
```

Out[7]:
```
Text(0.5, 1.0, 'Variable NOX')
```



In [8]:
```
variance_model = LinearRegression()
variance_scores = cross_val_score(variance_model, X[:,mask], y, cv=kfolds)
print("Variance Model R2 Mean %.3f StdDev %.3f"%(variance_scores.mean(),variance_scores.std()))
```

```
Variance Model R2 Mean 0.664 StdDev 0.140
```

→ the performance is more or less the same (as the baseline)

As we note, eliminating two of the features we have a small decrease in the performance from 0.682 to 0.664. We should check if such difference is significant.

2. Univariate Feature Selection

apply a statistic that looks at how much information does a variable have w.r.t. the target variable (since we're considering the target, we can say that this is kind of supervised method)

Univariate feature selection works by selecting the best features based on univariate statistical tests. In the example below, we apply *mutual_info_regression* that estimates mutual information for a continuous target variable.

In [9]:
```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_regression

# apply the procedure to take the best k variables based on mutual_info_regression
```

```
feature_selection_univariate_model = SelectKBest(mutual_info_regression, k=4)

# fit the feature selection model and select the four variables
X_selected_features_univariate = feature_selection_univariate_model.fit_transform(X,y)

mask = feature_selection_univariate_model.get_support() #list of booleans
print("Reduced data set shape = ",X_selected_features_univariate.shape)
print("     Selected features = ",data.feature_names[mask])
print("      Deleted Features = ", data.feature_names[~mask])
```

```
Reduced data set shape =  (506, 4)
     Selected features =  ['INDUS' 'NOX' 'RM' 'LSTAT']
      Deleted Features =  ['CRIM' 'ZN' 'CHAS' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO' 'B']
```

In [10]:
```
univariate_model = linear_model.LinearRegression()
univariate_scores = cross_val_score(univariate_model, X[:,mask], y, cv=kfolds)
print("Univariate Model R2 Mean %.3f StdDev %.3f"%(univariate_scores.mean(),univariate_scores.std()))
```

```
Univariate Model R2 Mean 0.585 StdDev 0.147
```

This approach dramatically reduces the number of attributes but, at the same time, the reduced data set has a much lower performance. Interestingly, the feature that were eliminated are not very informative as well: in fact, if we compute the performance using the features that have been droppped we note that the performance is even lower.

In [15]:
```
mask = np.invert(mask)
univariate_model = linear_model.LinearRegression()
univariate_scores = cross_val_score(univariate_model, X[:,mask], y, cv=kfolds)
print("Univariate Model R2 Mean %.3f StdDev %.3f"%(univariate_scores.mean(),univariate_scores.std()))
```

```
Univariate Model R2 Mean 0.585 StdDev 0.147
```

3. Principal Component Analysis

Given a data set described by n features (dimensions), Principal component analysis (PCA) finds k≤n orthogonal vectors (also called dimensions or basis) that best capture the variance of the data. The orthogonal vector that captures the most variance is called the *first principal component*. The orthogonal direction that captures the second largest projected variance is called the second principal component, and so on.

We now apply principal component analysis. Since we need to decide how many component to select and for this purpose we apply PCA and plot the explained variance ratio and the cumulative explained variance. Note that to apply PCA we first need to normalize the data, which in this case we do by applying z-score normalization.

In [16]:
```
from sklearn.preprocessing import StandardScaler
full_pca_model = PCA()
X_std = StandardScaler().fit_transform(X)
full_fitted_model = full_pca_model.fit(X_std)
```

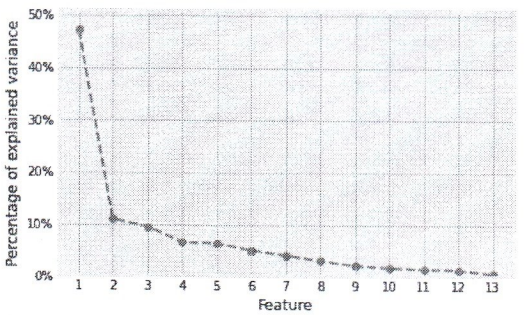
here we don't fix the number of components because we still have to decide it

We can now plot the percentage of explained variance that every feature explains. In the plot below, we note that the first feature (the first principal components) explains 47.1% of the overall (100%) variance while the second feature (the second principal component) explains the 11.0% of the variance.

In [17]:
```
full_fitted_model.explained_variance_ratio_
```

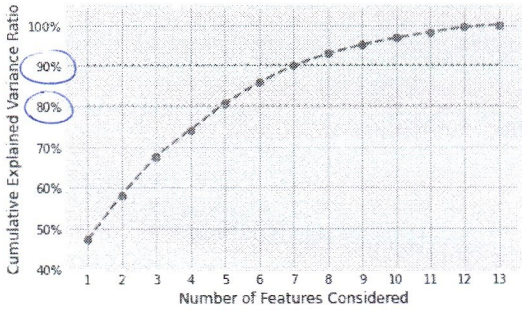
Out[17]:
```
array([0.47129606, 0.11025193, 0.0955859 , 0.06596732, 0.06421661,
       0.05056978, 0.04118124, 0.03046902, 0.02130333, 0.01694137,
       0.0143088 , 0.01302331, 0.00488533])
```

In [18]:
```
plt.plot(full_fitted_model.explained_variance_ratio_, '--o');
plt.xticks(np.arange(0,13,1),labels=np.arange(1,14,1));
plt.xlabel("Feature");
plt.ylabel("Percentage of explained variance");
plt.xticks(np.arange(0,13,1),labels=np.arange(1,14,1));
plt.yticks(np.arange(0.0,0.51,.1),labels=["%.0f%%"%(x*100) for x in np.arange(0.0,0.51,.1)]);
plt.ylim([0.0,0.51]);
```



The plot above reports how much variance each component explains. When selecting the number of components we are typically interested in selecting enough dimensions to explain a target value of variance. For instance, we might be interested in selecting enough variables to explain 90% of the variance. Accordingly, we usually check the cumulative explained variance ratio reported below. As can be noted, the first 5 dimensions explain 80% of the variance while we need 7 dimensions to explain 90% of the variance.

In [19]:
```
plt.plot(full_fitted_model.explained_variance_ratio_.cumsum(), '--o');
plt.xticks(np.arange(0,13,1),labels=np.arange(1,14,1));
plt.yticks(np.arange(0.4,1.1,.1),labels=["%.0f%%"%(x*100) for x in np.arange(0.4,1.1,.1)]);
plt.ylim([0.4,1.05]);
plt.plot([0,12],[.9,.9],':');
plt.plot([0,12],[.8,.8],':');
plt.xlabel("Number of Features Considered");
plt.ylabel("Cumulative Explained Variance Ratio");
```



Five components can actually explain 80% of the variance in the data so we apply PCA and select the first five components.

In [20]:
```
feature_selection_pca_model = PCA(n_components=5)
fitted_model = feature_selection_pca_model.fit(X_std)

X_selected_features_pca = fitted_model.transform(X_std)
print("    Explained Variance = %.3f" % fitted_model.explained_variance_ratio_.cumsum()[-1])
print("Reduced data set shape =",X_selected_features_pca.shape)
```

here we fix the number of components because we already did the analysis

```
    Explained Variance = 0.807
Reduced data set shape = (506, 5)
```

In [21]:
```
pca_model = linear_model.LinearRegression()
pca_scores = cross_val_score(pca_model, X_selected_features_pca, y, cv=kfolds)
pca_scores.mean()
print("PCA Model R2 Mean %.3f StdDev %.3f"%(pca_scores.mean(),pca_scores.std()))
```

```
PCA Model R2 Mean 0.648 StdDev 0.176
```

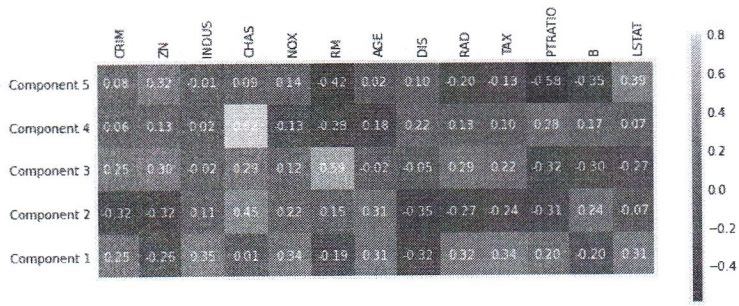
With using just the first five principal components as input variables, we reach the same performance as the one obtained with all the variables. However, the standard deviation is much higher than in the previous cases. The components are linear combinations of the original variables accordingly, when building a model using these new features we lose the connection with the original variables. For this purpose, it might interesting to check the weights that each component assigns to each original variable.

In [22]:
```
number_of_components = feature_selection_pca_model.components_.shape[0]
number_of_variables = feature_selection_pca_model.components_.shape[1]
component_ticks = np.arange(0,number_of_components,1)
component_labels = ['Component '+str(x+1) for x in component_ticks]

plt.figure(figsize=(12,12));
plt.matshow(feature_selection_pca_model.components_);

# avoid issue that cuts the first and last row
plt.ylim(-0.5, len(feature_selection_pca_model.components_)-0.5)

plt.yticks(component_ticks, labels=component_labels);
plt.colorbar();
plt.grid(False);
plt.xticks(range(len(input_variables)),input_variables,rotation=90,ha='left');
for (i, j), z in np.ndenumerate(feature_selection_pca_model.components_):
    plt.text(j, i, '{:0.2f}'.format(z), ha='center', va='center',color="white");
plt.show();
```

! **Remember**: to perform PCA we use only the input variables, we don't use the target (that's because we want to change the input space)

```
<Figure size 864x864 with 0 Axes>
```

Weights of the original variables in the new components



4. Embedded Approaches

In Lasso, feature selection naturally occurs as part of the learning process. Thus, it is an example of an **embedded approaches** to feature selection. We now apply Lasso with a regularization coefficient (alpha) of 0.1 and evaluate its performance using 10-fold crossvalidation.

In [23]:
```
lasso_model = Lasso(alpha=0.1,max_iter=300,random_state=seed)
lasso_scores = cross_val_score(lasso_model, X, y, cv=kfolds)

print("Lasso R2 Mean %.3f StdDev %.3f"%(lasso_scores.mean(),lasso_scores.std()))
```

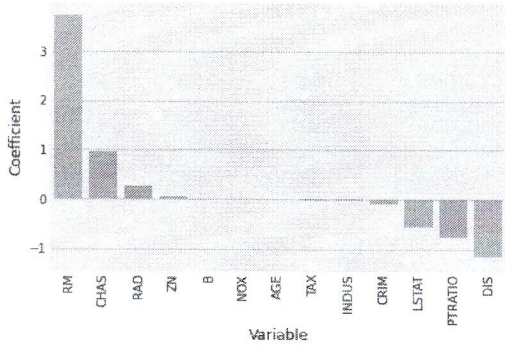
here we put $\alpha = 0.1$, however α is an hyperparameter (we should search for it)

we should have used the standardized input variables

```
Lasso R2 Mean 0.667 StdDev 0.133
```

As can be noted the performance is slightly lower than the one obtained by the previous feature sets. The standard deviations is also higher than the one we had with all the features. Lasso performs feature selection by zeroing out the weights of less important variables. We can visualize the weights to understand

In [24]:
```
lasso_model.fit(X,y)
lr_coefficients = pd.DataFrame({'Variable':input_variables, 'Coefficient':lasso_model.coef_}).sort_values(by=['Coefficient'],as
sns.barplot(x='Variable',y='Coefficient',data=lr_coefficients);
plt.xticks(rotation=90);
```



By modifying the regularization term alpha we can increase or decrease the amount of feature selection that Lasso performs.

5. Recursive Feature Elimination

we start with all the variables and we eliminate recursively one variable at the time

The previous approaches are all filter approaches in that they are not using the class information for the attribute selection. We start with a very simple one that starts from all the features and recursively consider smaller and smaller sets of features.

In [25]:
```
rfe = RFE(LinearRegression(), n_features_to_select=5,step=1)
rfe = rfe.fit(X, y)

X_selected_features_rfe = X[:,rfe.support_]

mask = rfe.get_support()
print("Reduced data set shape = ", X_selected_features_rfe.shape)
print("     Selected features = ", data.feature_names[mask])
print("      Deleted Features = ", data.feature_names[~mask])
```

also here we should have used standard-X

recursive feature elimination with target linearRegression(), we want at most 5 features and we want to eliminate (step=) 1 feature at each iteration.
Notice that this is the 1st time we specify the target method (LinearRegression())
→ wrapper approach

```
Reduced data set shape =  (506, 5)
     Selected features =  ['CHAS' 'NOX' 'RM' 'DIS' 'PTRATIO']
      Deleted Features =  ['CRIM' 'ZN' 'INDUS' 'AGE' 'RAD' 'TAX' 'B' 'LSTAT']
```

In [35]:
```
rfe_model = LinearRegression()
rfe_scores = cross_val_score(rfe_model, X[:,mask], y, cv=kfolds)
print("RFE R2 Mean %.3f StdDev %.3f"%(rfe_scores.mean(),rfe_scores.std()))
```

```
RFE R2 Mean 0.574 StdDev 0.235
```

Note that the reported performance is much lower than in the previous cases and the standard deviation is much higher than in the previous cases. This is probably due to the fact that we selected 5 features (which is generically low). (in this case)

6. Random Forest

We now apply a random forest and use its scoring to select the attributes.

Random forest has the feature that we can look at the forest and obtain a score for each variable

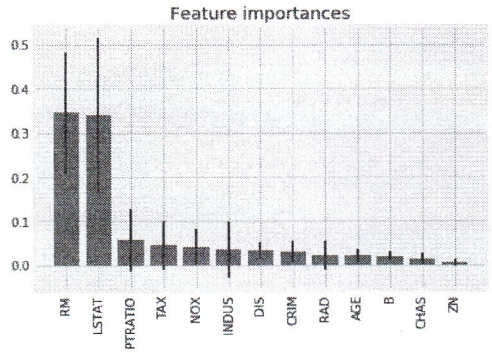
In [36]:
```
forest = ExtraTreesRegressor(n_estimators=250, random_state=0)
forest.fit(X, y)
```

Out[36]:
```
ExtraTreesRegressor(n_estimators=250, random_state=0)
```

Let's plot the feature importance.

In [37]:
```
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%2d. feature %2d %7s (%.3f)" % (f + 1, indices[f], data.feature_names[indices[f]], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), data.feature_names[indices],rotation=90)
plt.xlim([-1, X.shape[1]])
plt.show()
```

```
Feature ranking:
 1. feature  5      RM (0.344)
 2. feature 12   LSTAT (0.340)
 3. feature 10 PTRATIO (0.056)
 4. feature  9     TAX (0.044)
 5. feature  4     NOX (0.041)
 6. feature  2   INDUS (0.035)
 7. feature  7     DIS (0.032)
 8. feature  0    CRIM (0.030)
 9. feature  8     RAD (0.022)
10. feature  6     AGE (0.020)
11. feature 11       B (0.019)
12. feature  3    CHAS (0.013)
13. feature  1      ZN (0.004)
```



In [38]:
```
feature_selection_model = SelectFromModel(forest, prefit=True)
```

In [39]:
```
X_selected_features_forest = feature_selection_model.transform(X)
X_selected_features_forest.shape
```

Out[39]: (506, 2)

At this point, it would be interesting to compare the performance of our target regression algorithm on the reduced datasets. Note that, for wrapper feature selection the target algorithm and the algorithm used for selection usually coincide since wrapper approaches target selection for a specific algorithm and this should be the same later used for mining.

In [41]:
```
forest_model = linear_model.LinearRegression()
forest_scores = cross_val_score(forest_model, X_selected_features_forest, y, cv=kfolds)
print("RF Selected Features Model R2 Mean %.3f StdDev %.3f"%(forest_scores.mean(),forest_scores.std()))
```

```
RF Selected Features Model R2 Mean 0.590 StdDev 0.145
```

Also in this case the performance of the model using a subset of features is much lower than our baseling while the standard deviation is higher.

Discussion

We applied several methods for feature selection some implementing a *filter* approach, one an embedded approach (Lasso), and some implementing a wrapper approach. As we noted, sometimes feature selection can improve the performance of a predictor by getting rid of less important variables or lead a similar performance as the one obtained using the original set of variables; in other cases, the smaller set of feature corresponds to a reduced performance.

Performing feature selection is similar to tuning a hyperparameter of the algorithm accordingly we should apply the same procedures (using a train/validation/test partition) to guarantee a reliable evaluation of the performance. At the end, we should have compared the results we obtained to check whether the difference in performance was statistically significant. The quite high values of standard deviation associated with our crossvalidation suggest that probably none of the differences we recored will turn out to be significant. But this is left as an exercise to you.

In [ ]:

FEATURES SELECTION

Do we already know what method we'll use?

- Yes → wrapper approach (we have a target algorithm)
- No → filter approach

Note that features selection is a kind of hyperparameter.
We should have divided in train and validation.
Moreover, usually we apply more than one method and we look for coherence.

7. Wrapper Approach - Hill Climbing

we start from a point in space and we generate a perturbation of the point adding some random value. We create another point in the neighborhood of the original point, we compute the function (that we want to maximize, for instance) in the new point and if the function in this point is better than the function in the previous we keep the new point.

In this notebook we implement a rather simple feature selection procedure that follows a wrapper approach. The search algorithm, hill climbing in this case, is wrapped around the target classification/regression algorithm.

First we import the libraries that we will need.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

from sklearn import datasets
from sklearn import linear_model
from sklearn.neighbors import KNeighborsRegressor

from sklearn.preprocessing import StandardScaler

from deap import algorithms
from deap import base
from deap import creator
from deap import tools

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

Next we load the data and generate the k-fold evaluations.

In [2]:
```
data = datasets.load_boston()

scaler = StandardScaler()
X = scaler.fit_transform(data["data"])
y = data["target"]


number_of_variables = X.shape[1]
input_variables = data.feature_names
target_variable = 'MEDV'

seed = 1234
np.random.seed(seed)

# Let's create also a pandas data frame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['MEDV'] = y
df.head()

kfolds = KFold(10,shuffle=True,random_state=seed)
```

When applying a wrapper approach we are searching for the best subset of feature for a target model. In this case we will search for the best subset of features for plain linear regression.

In [3]:
```
def EvaluateFeatureSubsetSingleObjective(individual):
    selected_columns = []
    for i,allele in enumerate(individual):
        if (allele==1):
            selected_columns.append(df.columns[i])

    model = linear_model.LinearRegression()
    scores = cross_val_score(model, df[selected_columns], y, cv=kfolds)
    return scores.mean()
```

Hill Climbing

In [4]:
```
def HillClimbing(number_of_variables,number_of_evaluations,evaluation_function):

    # current evaluation
    evaluations = 0

    # start from a random set of features
    current_feature_subset = [random.randint(0,1) for x in range(number_of_variables)]

    # that will also provide an initial evaluation of the best performance
    best_performance = evaluation_function(current_feature_subset)

    print("%5d\t\t%3.2f\t%s"%(evaluations,best_performance,str(current_feature_subset)))

    # continue until all the evaluations have been performed
    while evaluations<number_of_evaluations:

        # generate a neighbor candidate using a 10% perturbation of the current subset
        perturbation = [(lambda x: 1-x if (random.random()<0.1) else x)(x) for x in current_feature_subset]

        # evaluate only if there is at least one variable
        if (sum(perturbation)>0):
            performance = evaluation_function(perturbation)

            if (performance>best_performance):
                best_performance = performance
                current_feature_subset = perturbation

        evaluations = evaluations + 1
        print("%5d\t\t%3.2f\t%s"%(evaluations,best_performance,str(current_feature_subset)))

    print("Best Feature Subset = %s "%(str(current_feature_subset)))
    print("Performance = %3.2f"%(best_performance))
```

Let's run hill-climbing for 100 evaluations.

In [5]:
```
HillClimbing(number_of_variables,100,EvaluateFeatureSubsetSingleObjective)
```

```
  0    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  1    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  2    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  3    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  4    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  5    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  6    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  7    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  8    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
  9    0.59    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
 10    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 11    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 12    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 13    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 14    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 15    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 16    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 17    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 18    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 19    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 20    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 21    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 22    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 23    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 24    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 25    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 26    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 27    0.67    [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
 28    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 29    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 30    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 31    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 32    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 33    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 34    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 35    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 36    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 37    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 38    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 39    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 40    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 41    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 42    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 43    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 44    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 45    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 46    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 47    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 48    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 49    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 50    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 51    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 52    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 53    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 54    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 55    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 56    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 57    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 58    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 59    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 60    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 61    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 62    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 63    0.68    [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 64    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 65    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 66    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 67    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 68    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 69    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 70    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 71    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 72    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 73    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 74    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 75    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 76    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 77    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 78    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 79    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 80    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 81    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 82    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 83    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 84    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 85    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 86    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 87    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 88    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 89    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 90    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 91    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 92    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 93    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 94    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 95    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 96    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 97    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 98    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
 99    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
100    0.69    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
Best Feature Subset = [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
Performance = 0.69
```

The datapoints in space are vectors of size = # variables (input size of the dataset). These vectors are of 0/1 : $1_i := \begin{cases} 1 & \text{if } i\text{-th feature is there} \\ 0 & \text{otherwise} \end{cases}$

How do we evaluate the set of features? (to decide when move from one point to another) We perform a linear regression with the included variables and we do a cross-validation score.

We can repeat the same process targeting another model like for instance a k-nearest-neighbour regressor with a k of 5.

In [6]:
```
def EvaluateFeatureSubsetKNN(individual):
    selected_columns = []
    for i,allele in enumerate(individual):
        if (allele==1):
```

```
            selected_columns.append(df.columns[i])

    model = KNeighborsRegressor(5)
    scores = cross_val_score(model, df[selected_columns], y, cv=kfolds)
    return scores.mean()
```

In [7]:
```
HillClimbing(number_of_variables,100,EvaluateFeatureSubsetKNN)
```

```
     0          0.27    [0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0]
     1          0.27    [0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0]
     2          0.27    [0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0]
     3          0.27    [0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0]
     4          0.27    [0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0]
     5          0.55    [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
     6          0.55    [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
     7          0.55    [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
     8          0.55    [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
     9          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    10          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    11          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    12          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    13          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    14          0.59    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
    15          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    16          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    17          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    18          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    19          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    20          0.64    [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
    21          0.65    [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1]
    22          0.65    [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1]
    23          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    24          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    25          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    26          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    27          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    28          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    29          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    30          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    31          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    32          0.73    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    33          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    34          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    35          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    36          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    37          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    38          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    39          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    40          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    41          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    42          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    43          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    44          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    45          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    46          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    47          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    48          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    49          0.74    [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
    50          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    51          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    52          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    53          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    54          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    55          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    56          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    57          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    58          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    59          0.75    [1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    60          0.75    [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    61          0.75    [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    62          0.75    [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    63          0.75    [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    64          0.75    [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1]
    65          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    66          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    67          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    68          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    69          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    70          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    71          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    72          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    73          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    74          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    75          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    76          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    77          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    78          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    79          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    80          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    81          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    82          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    83          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    84          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    85          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    86          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    87          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    88          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    89          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    90          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    91          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    92          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    93          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    94          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    95          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    96          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    97          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    98          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    99          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
   100          0.75    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
Best Feature Subset = [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
Performance = 0.75
```

better performance with fewer variables
→ kNN is much better than linear regression in this particular problem
We can say that similarity is a better principle to apply here than looking for linear combinations

Discussion

Note that, with k-NN, we were able to reach a better performance with much fewer features. Might we draw some insight from this result? Also note that when doing feature selection we used the entire dataset but feature selection is, as a matter of fact, similar to the search of the hyper-parameter alpha for Lasso/Ridge regression so we should perform it on the training set using the test set for the final evaluation of the feature subset.

In [ ]:

8. Feature Selection - Wrapper Approach using a Genetic Algorithm

In this notebook we implement a rather simple feature selection procedure that follows a wrapper approach. The search algorithm, Genetic algorithms in this case, is wrapped around the target classification/regression algorithm.

In [1]:
```
# install the evolutionary computation library
!pip install deap
```

```
Requirement already satisfied: deap in /Users/pierluca/opt/anaconda3/lib/python3.8/site-packages (1.3.1)
Requirement already satisfied: numpy in /Users/pierluca/opt/anaconda3/lib/python3.8/site-packages (from deap) (1.19.2)
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

from sklearn import datasets
from sklearn import linear_model
from sklearn import naive_bayes

from deap import algorithms
from deap import base
from deap import creator
from deap import tools

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

In [3]:
```
data = datasets.load_boston()

X = data["data"]
y = data["target"]

number_of_variables = X.shape[1]
input_variables = data.feature_names
target_variable = 'MEDV'

seed = 1234
np.random.seed(seed)

# Let's create also a pandas data frame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['MEDV'] = y
df.head()

kfolds = KFold(10,shuffle=True,random_state=seed)
```

In [4]:
```
def EvaluateFeatureSubsetSingleObjective(individual):
    selected_columns = []
    for i,allele in enumerate(individual):
        if (allele==1):
            selected_columns.append(df.columns[i])

    model = linear_model.LinearRegression()
    scores = cross_val_score(model, df[selected_columns], y, cv=kfolds)
    return scores.mean(),
```

Simple Genetic Algorithm

If looks for the feature subset that maximizes the overall performance.

In [5]:
```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
# creator.create("Individual", list, typecode='b', fitness=creator.FitnessMax)
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

# Attribute generator
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, number_of_variables)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

In [6]:
```
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, number_of_variables)
toolbox.register("evaluate", EvaluateFeatureSubsetSingleObjective)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
```

In [7]:
```
pop = toolbox.population(n=100)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)

pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40, stats=stats, halloffame=hof, verbose=True)
```

gen	nevals	avg	std	min	max
0	100	0.526028	0.125451	0.150693	0.682264
1	58	0.615364	0.0491616	0.311389	0.682264
2	59	0.653358	0.0216479	0.601715	0.682615
3	69	0.669885	0.0141574	0.598556	0.686291
4	54	0.677538	0.011446	0.600359	0.686291
5	53	0.681419	0.00997024	0.61548	0.686291
6	64	0.682171	0.0141366	0.601882	0.686291
7	70	0.683856	0.0100186	0.61548	0.686291
8	65	0.681178	0.0159122	0.593379	0.686291
9	69	0.684172	0.0105652	0.61548	0.686291

10	61	0.683945	0.0132883	0.573584	0.686291
11	69	0.683625	0.0112329	0.592809	0.686291
12	65	0.685271	0.00629057	0.638633	0.686291
13	55	0.684116	0.00966022	0.61548	0.686291
14	66	0.681732	0.0145022	0.612147	0.686291
15	58	0.683427	0.012757	0.591904	0.686291
16	51	0.683609	0.0120664	0.591904	0.686291
17	49	0.684001	0.00982046	0.61548	0.686291
18	67	0.68422	0.00989542	0.61548	0.686291
19	46	0.684149	0.0136831	0.564204	0.686291
20	54	0.683715	0.0105467	0.61548	0.686291
21	57	0.683905	0.0110301	0.61548	0.686291
22	66	0.683399	0.00918831	0.634053	0.686291
23	61	0.68238	0.0139522	0.584699	0.686291
24	62	0.684472	0.0088513	0.61548	0.686291
25	57	0.684561	0.0084257	0.61548	0.686291
26	64	0.685142	0.00537025	0.650362	0.686291
27	61	0.68391	0.0128186	0.573584	0.686291
28	55	0.684438	0.00728628	0.645059	0.686291
29	54	0.684698	0.00773885	0.622291	0.686291
30	52	0.682714	0.028156	0.409405	0.686291
31	66	0.682396	0.0229235	0.464671	0.686291
32	70	0.68297	0.0144149	0.559706	0.686291
33	56	0.683015	0.0148023	0.573584	0.686291
34	56	0.682155	0.0274753	0.424096	0.686291
35	60	0.685097	0.0053679	0.650362	0.686291
36	58	0.68434	0.00906916	0.61548	0.686291
37	61	0.685245	0.00470481	0.651054	0.686291
38	49	0.68444	0.0109816	0.60158	0.686291
39	65	0.683983	0.00939502	0.61548	0.686291
40	57	0.684866	0.00789333	0.625764	0.686291

Multi-objective Version

— we try to find the best performance with the minimum set of features

It applies a multi-objective genetic algorithm that tries to maximize the performance while minimizing the number of features involved.

In [8]:
```
def EvaluateFeatureSubsetMultipleObjective(individual):
    '''returns the average performance and the number of features involved'''
    selected_columns = []
    for i,allele in enumerate(individual):
        if (allele==1):
            selected_columns.append(df.columns[i])

    if (len(selected_columns)>0):
        model = linear_model.LinearRegression()
        scores = cross_val_score(model, df[selected_columns], y, cv=kfolds)
        return scores.mean(),sum(individual)/float(len(individual))
    else:
        return 0,len(individual)
```

In [9]:
```
creator.create("FitnessMulti", base.Fitness, weights=(1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMulti)

toolbox = base.Toolbox()
# Attribute generator
toolbox.register("attr_bool", random.randint, 0, 1)
# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, number_of_variables)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Operator registering
toolbox.register("evaluate", EvaluateFeatureSubsetMultipleObjective)
toolbox.register("mate", tools.cxUniform, indpb=0.1)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selNSGA2)
```

```
/Users/pierluca/opt/anaconda3/lib/python3.8/site-packages/deap/creator.py:138: RuntimeWarning: A class named 'Individual' has a
lready been created and it will be overwritten. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
```

In [10]:
```
# random.seed(64)
MU, LAMBDA = 100, 200
pop = toolbox.population(n=MU)
hof = tools.ParetoFront()
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean, axis=0)
stats.register("std", np.std, axis=0)
stats.register("min", np.min, axis=0)
stats.register("max", np.max, axis=0)

pop, logbook = algorithms.eaMuPlusLambda(pop, toolbox, mu=MU, lambda_=LAMBDA,
                                         cxpb=0.7, mutpb=0.3, ngen=40,
                                         stats=stats, halloffame=hof)

print("BEST "+str(hof[0]))
```

gen	nevals	avg	std	min	max
0	100	[0.53352882 0.52153846]	[0.11126337 0.13785337]	[0.22216453 0.15384615]	[0.67384659 0.84615385]
1	200	[0.5985008 0.48846154]	[0.07881769 0.16583124]	[0.21474826 0.15384615]	[0.67384659 0.92307692]
2	200	[0.61645502 0.48]	[0.07371555 0.19095524]	[0.22539109 0.15384615]	[0.67384659 0.92307692]
3	200	[0.62589405 0.49230769]	[0.07872359 0.23204774]	[0.21082521 0.07692308]	[0.67596593 0.92307692]
4	200	[0.59025659 0.36692308]	[0.10562454 0.22789362]	[0.21082521 0.07692308]	[0.68031402 0.92307692]
5	200	[0.61461188 0.38692308]	[0.09938336 0.21330174]	[0.12079571 0.07692308]	[0.68201157 1.]
6	200	[0.60246301 0.35538462]	[0.1354925 0.19844961]	[0.19012883 0.07692308]	[0.68331854 1.]
7	200	[0.63422572 0.34153846]	[0.05473387 0.17004002]	[0.4993403 0.07692308]	[0.68331854 0.92307692]
8	200	[0.59140027 0.23692308]	[0.0659096 0.19205522]	[0.4993403 0.07692308]	[0.68331854 0.92307692]
9	200	[0.54339957 0.18692308]	[0.06948684 0.22538724]	[0.4993403 0.07692308]	[0.68331854 0.92307692]
10	200	[0.52979121 0.15384615]	[0.0640749 0.18365133]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
11	200	[0.52979121 0.15384615]	[0.0640749 0.18365133]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
12	200	[0.53166072 0.16153846]	[0.06586159 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
13	200	[0.53166072 0.16153846]	[0.06586159 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
14	200	[0.52987616 0.15538462]	[0.0642764 0.18872979]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
15	200	[0.53166814 0.16153846]	[0.06587812 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
16	200	[0.53166814 0.16153846]	[0.06587812 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
17	200	[0.52984242 0.15461538]	[0.06419745 0.18636828]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
18	200	[0.53167517 0.16153846]	[0.06589419 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
19	200	[0.53167517 0.16153846]	[0.06589419 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
20	200	[0.53167517 0.16153846]	[0.06589419 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
21	200	[0.53167517 0.16153846]	[0.06589419 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]
22	200	[0.53167517 0.16153846]	[0.06589419 0.19596522]	[0.4993403 0.07692308]	[0.68629149 0.84615385]

```
23      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
24      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
25      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
26      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
27      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
28      200     [0.53167517 0.16153846] [0.06589419 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
29      200     [0.52991718 0.15615385] [0.06437102 0.19074907] [0.4993403  0.07692308] [0.68629149 0.84615385]
30      200     [0.53170244 0.16153846] [0.06595438 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
31      200     [0.53170244 0.16153846] [0.06595438 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
32      200     [0.53170244 0.16153846] [0.06595438 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
33      200     [0.53170244 0.16153846] [0.06595438 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
34      200     [0.52991967 0.15538462] [0.06437686 0.18872979] [0.4993403  0.07692308] [0.68629149 0.84615385]
35      200     [0.52991967 0.15538462] [0.06437686 0.18872979] [0.4993403  0.07692308] [0.68629149 0.84615385]
36      200     [0.52991967 0.15538462] [0.06437686 0.18872979] [0.4993403  0.07692308] [0.68629149 0.84615385]
37      200     [0.52991967 0.15538462] [0.06437686 0.18872979] [0.4993403  0.07692308] [0.68629149 0.84615385]
38      200     [0.53172087 0.16153846] [0.06599553 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
39      200     [0.53172087 0.16153846] [0.06599553 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
40      200     [0.53172087 0.16153846] [0.06599553 0.19596522] [0.4993403  0.07692308] [0.68629149 0.84615385]
BEST [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
```

Discussion

Note that we applied genetic algorithms using the entire dataset for the evaluation of the feature subset. In a real scenario we should have initially split the data as train and test and then applied the genetic algorithm only using the training dataset.

In [ ]:

Discretization

Discretization transforms a continuous variable into a nominal (ordinal) one. It returns a set of contiguous intervals (or bins) that are used to map the original variable range into a sequence of labels corresponding to the intervals containing such values. Discretization is applied both to simplify the problem space by reducing the number of values both to apply methods that can only be applied to nominal attributes to problems described by continuous variables.

Discretization algorithms can be either **unsupervised** or **supervised**.

Unsupervised discretization algorithms do not use the information about an available target variable to set partition boundaries. Accordingly, they might loose information about the target variable because they might combine values that are associated with different classes into the same interval,

Supervised discretization algorithms use the information about a target variable to generate intervals that try to preserve relevant information about the class.

We now compare different discretization algorithms using the Boston Housing dataset.

In [1]:
```
import numpy as np
import pandas as pd
import random

from sklearn import datasets
from sklearn import linear_model
from sklearn import naive_bayes
from sklearn import neighbors
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.ensemble import ExtraTreesRegressor

from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import RFE

from sklearn.decomposition import PCA

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split


from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.preprocessing import StandardScaler

from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl


plt.style.use('fivethirtyeight')
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['axes.labelsize'] = 14
mpl.rcParams['xtick.labelsize'] = 12
mpl.rcParams['ytick.labelsize'] = 12
mpl.rcParams['text.color'] = 'k'

%matplotlib inline
```

In [2]:
```
random_state = 1234
```

In [3]:
```
data = datasets.load_boston()

input_variables = data.feature_names
target_variable = 'MEDV'

seed = 1234

# let's create also a pandas data frame
df = pd.DataFrame(data.data, columns=data.feature_names)
df['MEDV'] = data.target
df.head()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

We are focusing on the variable **LSTAT** so that we can easily analyze what the discretization produces.

In [4]:
```
X = df[["LSTAT"]]
y = data["target"]
```

Let's define the cross-validation we are going to apply to compare all the discretization approaches:

In [5]:
```
cv = KFold(n_splits=10, shuffle=True, random_state=random_state)
```

Also Let's set the number of intervals to 5 so that we can compare the various approaches using the same number of intervals.

we should put it as an hyperparameter (perform changes and do an knee/elbow analysis)

In [6]:
```
no_intervals = 5
```

Since linear regression is sentitive to the range of attribute values, let's normalize the input variables and generate the train/test partition that we use for evaluation.

In [7]:
```
X = StandardScaler().fit_transform(X)
```

Note that we are going to use the entire dataset for the sake of simplicity. In an actual scenario we should first extract a training dataset and a test dataset and perform the discretization on the training set and the evaluation on the test set using something like:

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.5, random_state=random_state)

And then doing all the analysis using X_train and X_test.

Unsupervised Discretization Methods

The simplest way to discretize a continuous attribute is to partition the range of values into k bins of equal size. This approach, called equal-width discretization, is vulnerable to outlie values that might skew the range. Another approach, called equal-frequency discretization, divides the variable (with n values) into k intervals containing the same number n/k values. Variables can also be discretized by applying a clustering algorithms like k-means.

In [8]:
```
discretizations = {
    'Onehot-Equal-Width':KBinsDiscretizer(n_bins=no_intervals, encode='onehot', strategy='uniform'),
    'Onehot-Equal-Frequency':KBinsDiscretizer(n_bins=no_intervals, encode='onehot', strategy='quantile'),
    'Onehot-Equal-Clustering':KBinsDiscretizer(n_bins=no_intervals, encode='onehot', strategy='kmeans'),
    'Ordinal-Equal-Width':KBinsDiscretizer(n_bins=no_intervals, encode='ordinal', strategy='uniform'),
    'Ordinal-Equal-Frequency':KBinsDiscretizer(n_bins=no_intervals, encode='ordinal', strategy='quantile'),
    'Ordinal-Equal-kMeans':KBinsDiscretizer(n_bins=no_intervals, encode='ordinal', strategy='kmeans')
}
```

very sentitive to outliers

it creates many intervals where we have many values and few intervals where we have fewer datapoints

The intervals can be sorted and so considered ordinals (Sklearn gives us 2 options: either pure categorical var (one-hot-encoding) or ordinal variables)

In [9]:
```
performance = {}

p = cross_val_score(LinearRegression(),X,y,cv=cv)
performance['No Discretization'] = (np.average(p),np.std(p))

for discretization_name in discretizations.keys():

    Xd = discretizations[discretization_name].fit_transform(X)

    # since the discretization producing ordinals will generate integer values,
    # we might also want to normalize these values depending on the output range
    # note that in this simple case it does not make much difference in performance.

    if (discretization_name[:7]=="Ordinal"):
        Xd = StandardScaler().fit_transform(Xd)

    p = cross_val_score(LinearRegression(),Xd,y,cv=cv)
    performance[discretization_name] = (np.average(p),np.std(p))
```

In [10]:
```
for method in performance.keys():
    print("%12s\tMean=%3.2f Std=%3.2f"%(method,performance[method][0],performance[method][1]))
```

```
No Discretization       Mean=0.50 Std=0.11
Onehot-Equal-Width      Mean=0.47 Std=0.09
Onehot-Equal-Frequency  Mean=0.53 Std=0.13
Onehot-Equal-Clustering Mean=0.49 Std=0.13
Ordinal-Equal-Width     Mean=0.44 Std=0.09
Ordinal-Equal-Frequency Mean=0.52 Std=0.10
Ordinal-Equal-kMeans    Mean=0.46 Std=0.10
```

To analyse whether the reported difference is statistically significant we should apply a paired t-test with an adequate Bonferroni's adjustment.

Supervised Discretization using Decision Tree

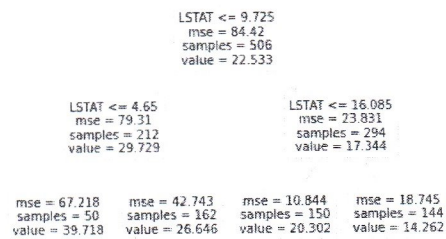
Supervised discretization methods exploit the information about the target variable to try to minimize the loss of information about the target value that discretization may cause. We are going to show a method that uses decision tree to discover the intervals to partition a continuous variable.

We can use a very simple decision tree of fixed depth to generate intervals that try to maintain the information about the target variable as much as possible. Note that since decision trees are not sensitive to the variable range we are using the raw version of the attribute, not the normalized one.

In [11]:
```
tree_model = DecisionTreeRegressor(max_depth=2)
tree_model.fit(df[['LSTAT']], y)
```

Out[11]:
```
DecisionTreeRegressor(max_depth=2)
```

In [12]:
```
plot_tree(tree_model,feature_names=['LSTAT']);
plt.show()
```



The tree represents four intervals each one identified by each one of the leaves. Accordingly, if I use the tree to predict the output for the original input variable, I will obtain a column containing four values each one identifying one of the intervals. The intervals are identified by the values used in the decision tree for the splitting decisions, that is, 4.65, 9.725, and 16.085.

Note that, if we use the tree to predict the target value using the input variable (LSTAT) this will produce a vector containing the 4 values corresponding to the four leaves that actually identify the four intervals.

In [13]:
```
df['Predicted MEDV'] = np.round(tree_model.predict(df[['LSTAT']]),2)
df['Predicted MEDV'].unique()
```

Out[13]:
```
array([26.65, 39.72, 20.3 , 14.26])
```

If we check the tree, value 39.72 identifies the first interval, 26.65 identifies the second interval, 20.30 the third one, and 14.26 the fourth one. So if we want to generate an ordinal attribute for instance with 4 values we might decide to map these four values in four integers like for instance 1,2,3,4.

In [14]:
```
ordinal_mapping = {39.72:1,26.65:2,20.30:3,14.26:4}
```

In [15]:
```
df['Discrete LSTAT (Ordinal)'] = df['Predicted MEDV'].apply(lambda x: ordinal_mapping[x])
```

If we want to check how the variable LSTAT was discretized we can group by the original data using the new attribute,

In [16]:
```
df[['LSTAT','Discrete LSTAT (Ordinal)']].groupby(by=['Discrete LSTAT (Ordinal)']).agg({'LSTAT':['min','max']})
```

Out[16]:

	LSTAT	
	min	max
Discrete LSTAT (Ordinal)		
1	1.73	4.63
2	4.67	9.71
3	9.74	16.03
4	16.14	37.97

The values for the intervals correspond to the ones we can find in the splits inside the regression tree.

In [17]:
```
p = cross_val_score(LinearRegression(),df[['Discrete LSTAT (Ordinal)']],y,cv=cv)
```

In [18]:
```
print("%12s\tMean=%3.2f Std=%3.2f"%("Supervised", np.mean(p), np.std(p)))
```

```
  Supervised    Mean=0.58 Std=0.09
```

As can be noted the performance is slightly higher although given the high values of standard deviation the difference is probably not statistically significant.

└ because we're creating intervals that facilitate the prediction of the target variable

Discussion

There are several supervised discretization methods that have been developed. Most of them are based on information-based metrics to generate partitions that try to minimize the loss of information that discretization is likely to introduce. There are no official implementations available for scikit-learn. Some of the methods are discussed in,

- Fayyad, Usama M.; Irani, Keki B. (1993) "Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning". https://www.ijcai.org/Proceedings/93-2/Papers/022.pdf
- Dougherty, J.; Kohavi, R. ; Sahami, M. (1995). "Supervised and Unsupervised Discretization of Continuous Features". https://ai.stanford.edu/~ronnyk/disc.pdf

There are a couple of implementations that are not integrated in scikit-learn but follow its fit/transform pattern,

- https://github.com/navicto/Discretization-MDLPC
- https://github.com/hlin117/mdlp-discretization

In [ ]:
```
```