ANALYSIS OF FRED DATA

Predicting Consumption

In this example, we analyze some data downloaded from the Federal Reserve Economic Data. In particular, we want to build a model to predict the Real Personal Consumption Expenditures (PCECC96) using four information:

- Unemployment Rate (UNRATE)
- Industrial Production: Total index (IPB50001SQ)
- Real Disposable Personal Income (DPIC96)
- Personal Saving (PSAVE)

The example is inspired to Chapter 7 of the book Forecasting: Principles and Practice by Rob J Hyndman and George Athanasopoulos. We preprocessed the original data and saved them in a csv file. We renamed the columns to provide a simplified indication of the original information.

As usual, we start by loading the libraries and the data.

In [1]:
```
import pandas as pd
import numpy as np
import math
from datetime import datetime
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

# select the style from fivethityeight website
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'

# predefined figsize
figsize=(12,9)

import warnings
warnings.filterwarnings('ignore')

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.metrics import mean_squared_error

#
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

In [2]:
```
df = pd.read_csv('fred_data.csv', index_col='observation_date', parse_dates=['observation_date'])
```

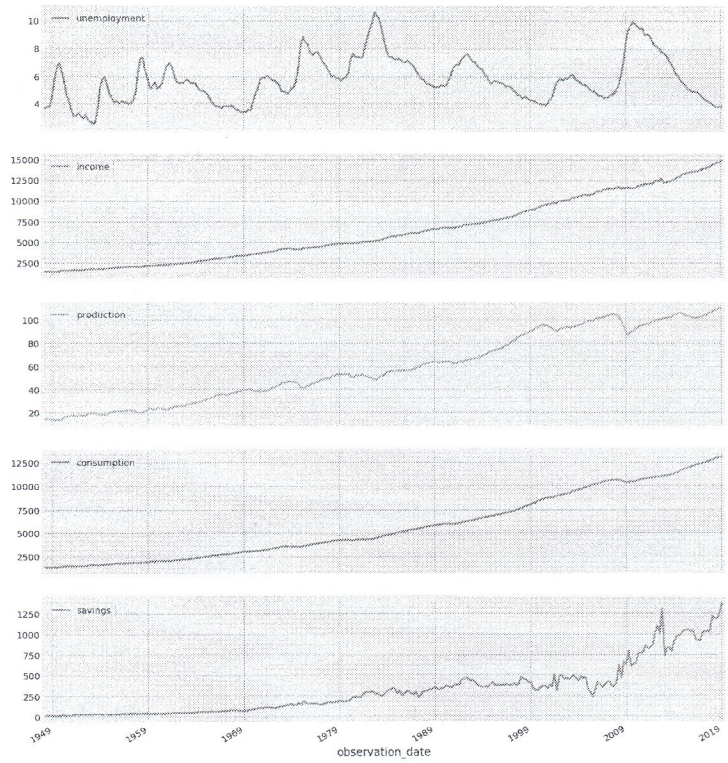
In [3]:
```
df.head()
```

Out[3]:

observation_date	unemployment	income	production	consumption	savings
1948-01-01	3.733333	1421.789	14.7233	1307.283	12.763
1948-04-01	3.666667	1465.890	14.8803	1322.494	16.672
1948-07-01	3.766667	1489.793	15.0187	1324.446	19.876
1948-10-01	3.833333	1498.173	14.8618	1335.016	19.458
1949-01-01	4.666667	1468.673	14.3818	1337.177	15.143

Let's plot the data.

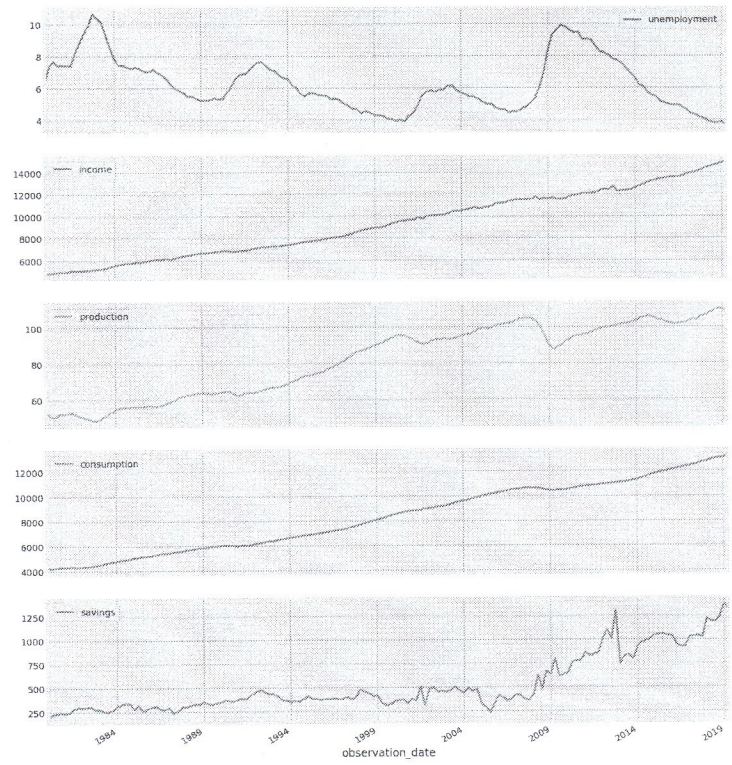
In [4]:
```
df.plot(subplots=True,figsize=(12,16));
```



We want to predict Real Personal Consumption Expenditures (PCECC96) in the recent years so it does not make sense to keep the data from 1950s. Thus, we focus on the data starting from 1980.

In [5]:
```
df=df['1980':]
```

In [6]:
```
df.plot(subplots=True,figsize=(12,16));
```



Stationarity

As already discussed, as the very first step we need to check whether our target series is stationary. For this purpose, we apply Augmented Dickey-Fuller test with a 95% (or 0.95) confidence level and check the returned p-value. If the p-value is less than 0.05 (that is 1-0.95) we can reject the null hypothesis that the data comes from a random walk; if the p-value is greater than 0.05 we cannot reject the null hypothesis that the data actually comes from a random walk, thus it is likely to be non-stationary and we need to transform it into a stationary series by applying the usual transformations.

In [7]:
```
p_value_consumption = adfuller(df['consumption'])
print("p-value: %.4f"%p_value_consumption[1])
```

```
p-value: 0.9875
```

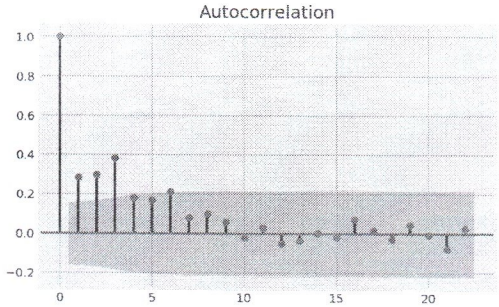
We need to apply a transformation to the target variable to make it stationary. For instance, instead of using the actual value we can forecast its percentage change.

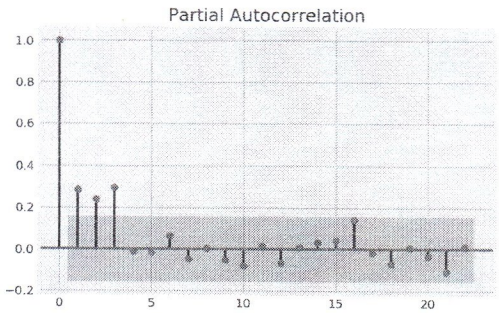
In [8]:
```
# let's copy the original data
df_change = df.copy()

# let's substitute the original consumption with *pct_change()*
df_change['consumption'] = df['consumption'].pct_change()
df_change.dropna(inplace=True)

# let' check the p-value
p_value_consumption = adfuller(df_change['consumption'])
print("p-value: %.4f"%p_value_consumption[1])
```

```
p-value: 0.0076
```

As can be noted, the simple transformation has made the series non-stationary. Since we are now tracking the percentage change of the original variable it might be a good idea also to

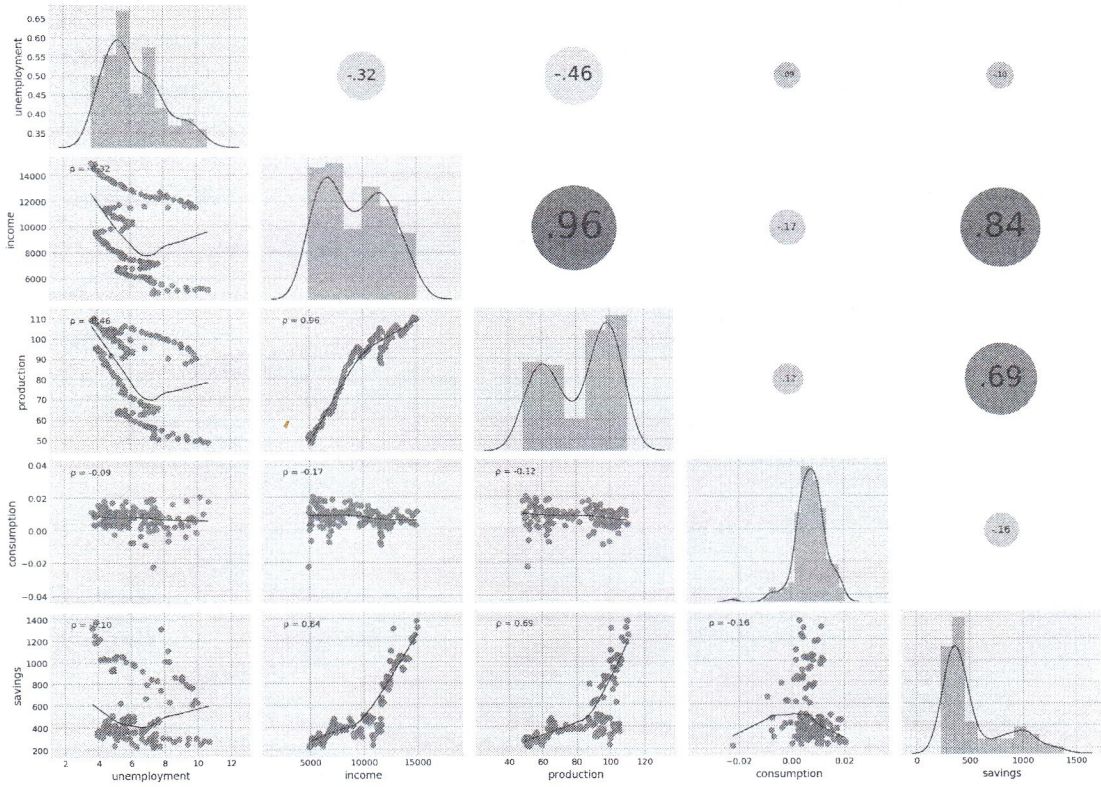
In [9]:
```
acf_consumption = plot_acf(df_change['consumption'])
acf_consumption = plot_pacf(df_change['consumption'])
```





Let's analyze the correlation among variables.

In [10]:
```
from scipy.stats import pearsonr

def corrfunc(x,y, ax=None, **kws):
    """Plot the correlation coefficient in the top left hand corner of a plot."""
    r, _ = pearsonr(x, y)
    ax = ax or plt.gca()
    # Unicode for lowercase rho (ρ)
    rho = '\u03C1'
    ax.annotate(f'{rho} = {r:.2f}', xy=(.1, .9), xycoords=ax.transAxes)

def corrdot(*args, **kwargs):
    corr_r = args[0].corr(args[1], 'pearson')
    corr_text = f"{corr_r:2.2f}".replace("0.", ".")
    ax = plt.gca()
    ax.set_axis_off()
    marker_size = abs(corr_r) * 10000
    ax.scatter([.5], [.5], marker_size, [corr_r], alpha=0.6, cmap="Blues",
               vmin=-1, vmax=1, transform=ax.transAxes)
    font_size = abs(corr_r) * 40 + 5
    ax.annotate(corr_text, [.5, .5,],  xycoords="axes fraction",
                ha='center', va='center', fontsize=font_size)

# g = sns.pairplot(stocks,palette=["Blues_d"])
g = sns.PairGrid(df_change, aspect=1.4, diag_sharey=False)
g.map_lower(corrfunc)
g.map_lower(sns.regplot, lowess=True, ci=False, line_kws={'color': 'Black','linewidth':1})
g.map_diag(sns.distplot, kde_kws={'color': 'Black','linewidth':1})
g.map_upper(corrdot)
plt.show()
```



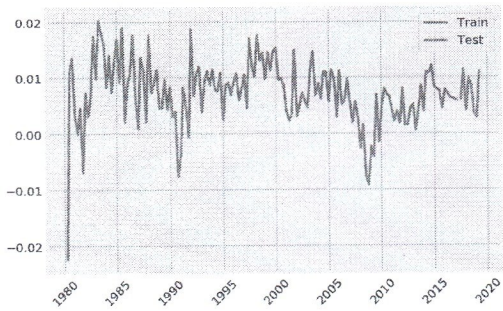
As we can note income and production are highly correlated as well as income and savings, which is reasonable to assume. Now let's define the input variables and the target variabale; then define the training and testing data; finally, we plot them two sets.

In [11]:
```
target_variable = 'consumption'
input_variables = df_change.columns[df_change.columns!=target_variable]
```

In [12]:
```
X=df_change[input_variables]
y=df_change[target_variable]
```

In [13]:
```
X_train, X_test = X[:-8], X[-8:]
y_train, y_test = y[:-8], y[-8:]
```

In [14]:
```
plt.plot(y_train,label='Train')
plt.plot(y_test,label='Test')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



Linear Regression

We start by applying a simple linear regression to the raw data.

In [15]:
```
lr_model = LinearRegression()
lr_model.fit(X_train,y_train)

yt = lr_model.predict(X_train)
yp = lr_model.predict(X_test)

print('RMSE on Test %.5f'%math.sqrt(mean_squared_error(yp,y_test)))
```

```
RMSE on Test 0.00448
```

In [16]:
```
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

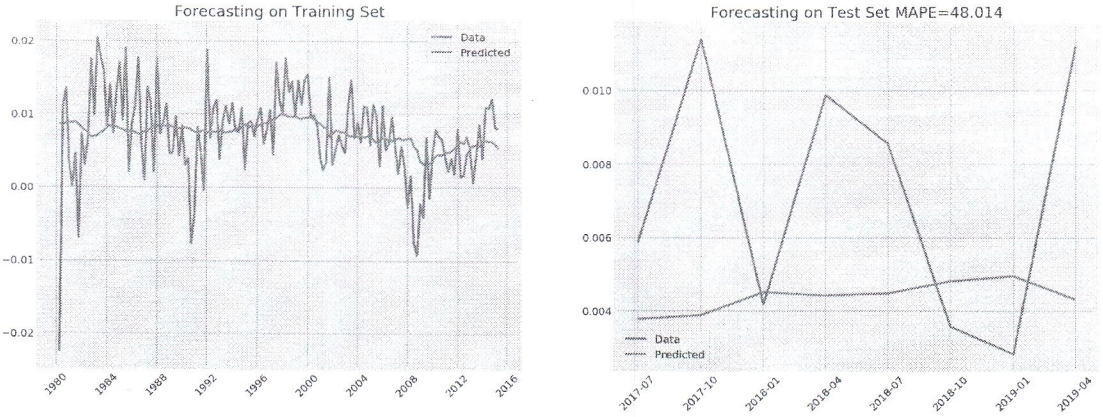
In [17]:
```
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,yp))
```

```
MAPE on Test 48.014
```

In [18]:
```
df_result_train = pd.DataFrame(y_train)
df_result_train['predicted'] = yt
df_result_test = pd.DataFrame(y_test)
df_result_test['predicted'] = yp
```

In [19]:
```
fig,axes = plt.subplots(1,2,figsize=(16,6))
```

```
plt.subplot(1,2,1)
plt.plot(df_result_train['consumption'][:-8],label='Data')
plt.plot(df_result_train['predicted'][:-8],label='Predicted')
# plt.ylim([180,300])
plt.xticks(rotation=45)
plt.title("Forecasting on Training Set")
plt.legend();

plt.subplot(1,2,2)
plt.plot(df_result_test['consumption'],label='Data')
plt.plot(df_result_test['predicted'],label='Predicted')
# plt.ylim([200,330])
plt.xticks(rotation=45)
plt.title("Forecasting on Test Set MAPE=%.3f"%mean_absolute_percentage_error(y_test,yp))
plt.legend();
```



What if we add as an input also the consumption at time t-1? Let's see if this can improve the predictive power.

In [37]:
```
df_change['consumption-01'] = df_change['consumption'].shift()
df_change.dropna(inplace=True)

target_variable = 'consumption'
input_variables = df_change.columns[df_change.columns!=target_variable]

X=df_change[input_variables]
y=df_change[target_variable]

X_train, X_test = X[:-8], X[-8:]
y_train, y_test = y[:-8], y[-8:]

lr_model = LinearRegression()
lr_model.fit(X_train,y_train)

yt = lr_model.predict(X_train)
yp = lr_model.predict(X_test)

print('RMSE on Test %.5f'%math.sqrt(mean_squared_error(yp,y_test)))
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,yp))
```

```
RMSE on Test 0.00270
MAPE on Test 50.756
```

It does not really improve the performance. So what if we add to the inputs also the percentage change of the original input variables? And for each input variable also its value at t-1?

In [38]:
```
# Let's copy the original data
df_change = df.copy()

# let's substitute the original consumption with *pct_change()*
df_change['consumption'] = df['consumption'].pct_change()

target_variable = 'consumption'
input_variables = df_change.columns[df_change.columns!=target_variable]

# add the % change also for the other variables
for var in input_variables:
    df_change[var+'_change'] = df_change[var].pct_change()

# add t-1 for all the variables
input_variables = df_change.columns[df_change.columns!=target_variable]
for var in input_variables:
    df_change[var+'-01'] = df_change[var].shift()

df_change.dropna(inplace=True)
```

In [39]:
```
df_change
```

Out[39]:

observation_date	unemployment	income	production	consumption	savings	unemployment_change	income_change	production_change	savings_
1980-07-01	7.666667	4904.532	50.3368	0.010969	234.053	4.545455e-02	0.011232	-0.016425	0.
1980-10-01	7.400000	4972.057	52.2738	0.013511	242.873	-3.478261e-02	0.013768	0.038481	0.
1981-01-01	7.433333	4965.100	52.3984	0.003350	236.564	4.504505e-03	-0.001399	0.002384	-0.
1981-04-01	7.400000	4970.894	52.5593	0.000073	241.431	-4.484305e-03	0.001167	0.003071	0.
1981-07-01	7.400000	5078.088	53.0648	0.004626	282.647	4.440892e-16	0.021564	0.009618	0.
...	...	...	...	...	...	...	...	...	
2018-04-01	3.900000	14495.935	107.9190	0.009880	1187.399	-4.098361e-02	0.006640	0.011237	-0.
2018-07-01	3.800000	14613.284	109.2838	0.008568	1186.382	-2.564103e-02	0.008095	0.012647	-0.

observation_date	unemployment	income	production	consumption	savings	unemployment_change	income_change	production_change	savings_
2018-10-01	3.800000	14715.221	110.3249	0.003571	1247.600	0.000000e+00	0.006976	0.009527	0.
2019-01-01	3.866667	14878.094	109.7876	0.002833	1375.510	1.754386e-02	0.011068	-0.004870	0.
2019-04-01	3.633333	14966.643	109.1964	0.011197	1318.083	-6.034483e-02	0.005952	-0.005385	-0.

156 rows × 17 columns

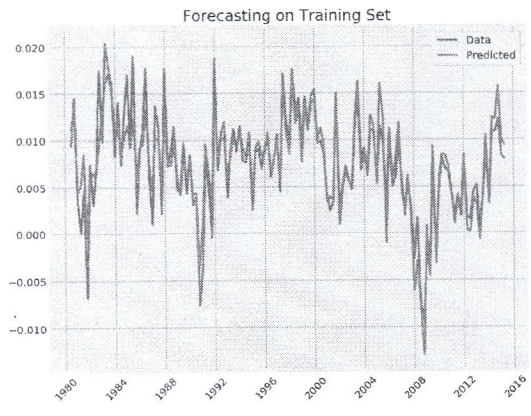
In [40]:
```
target_variable = 'consumption'
input_variables = df_change.columns[df_change.columns!=target_variable]

X=df_change[input_variables]
y=df_change[target_variable]

X_train, X_test = X[:-8], X[-8:]
y_train, y_test = y[:-8], y[-8:]

lr_model = LinearRegression()
lr_model.fit(X_train,y_train)

yt = lr_model.predict(X_train)
yp = lr_model.predict(X_test)

print('RMSE on Test %.5f'%math.sqrt(mean_squared_error(yp,y_test)))
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,yp))
```

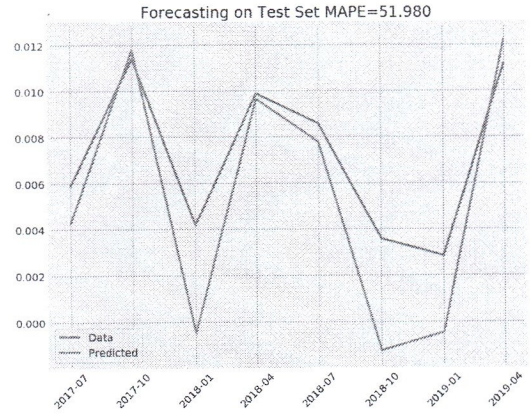
```
RMSE on Test 0.00275
MAPE on Test 51.980
```

As we can see it does not really improve performance.

In [43]:
```
df_result_train = pd.DataFrame(y_train)
df_result_train['predicted'] = yt
df_result_test = pd.DataFrame(y_test)
df_result_test['predicted'] = yp
```

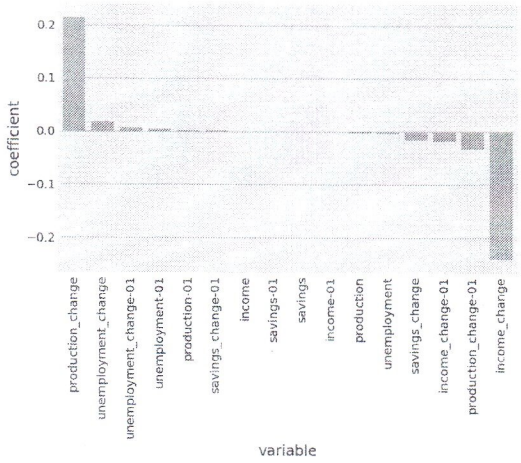
In [44]:
```
fig,axes = plt.subplots(1,2,figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(df_result_train['consumption'][:-8],label='Data')
plt.plot(df_result_train['predicted'][:-8],label='Predicted')
# plt.ylim([180,300])
plt.xticks(rotation=45)
plt.title("Forecasting on Training Set")
plt.legend();

plt.subplot(1,2,2)
plt.plot(df_result_test['consumption'],label='Data')
plt.plot(df_result_test['predicted'],label='Predicted')
# plt.ylim([200,330])
plt.xticks(rotation=45)
plt.title("Forecasting on Test Set MAPE=%.3f"%mean_absolute_percentage_error(y_test,yp))
plt.legend();
```





We can plot the model weights to analyze what variables have the most influence in the prediction. As we can see several variables contribute to the final model.

In [45]:
```
lr_coefficients = pd.DataFrame({'variable':X.columns, 'coefficient':lr_model.coef_}).sort_values(by=['coefficient'],ascending=F
sns.barplot(x='variable',y='coefficient',data=lr_coefficients);
plt.xticks(rotation=90);
```



We want a simpler model so we apply Lasso that tends to zero-out useless variables.

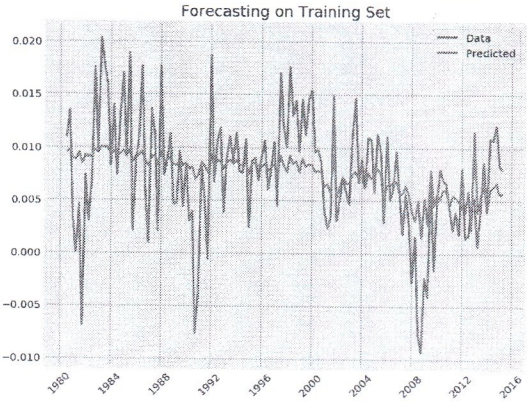
In [48]:
```
lasso_model = Lasso(alpha=0.01,max_iter=1000)
lasso_model.fit(X_train,y_train)

yt = lasso_model.predict(X_train)
yp = lasso_model.predict(X_test)

print('RMSE on Test %.5f'%math.sqrt(mean_squared_error(yp,y_test)))
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,yp))
```

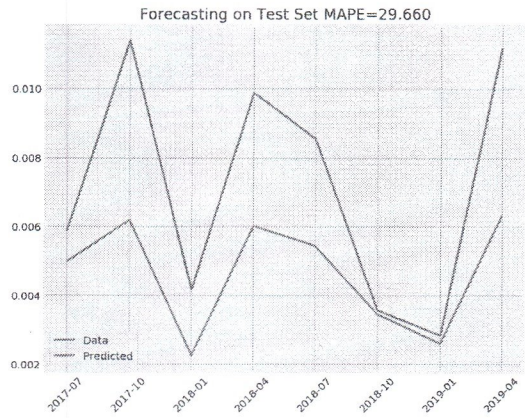
```
RMSE on Test 0.00316
MAPE on Test 29.660
```

As we note the prediction is much better and also the number of variables contributing to the model is much fewer (see the weight plot below). Note that the performance improvement is also due to the choice of an adequate value of α. If we repeat the experiment with an α of 1.0 the performance will worsen significantly.

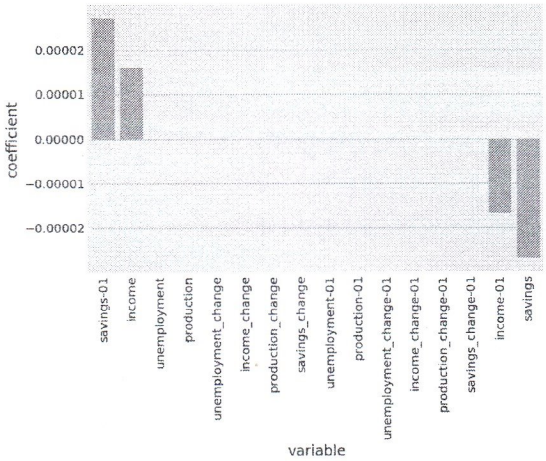
In [49]:
```
df_result_train = pd.DataFrame(y_train)
df_result_train['predicted'] = yt
df_result_test = pd.DataFrame(y_test)
df_result_test['predicted'] = yp
```

In [50]:
```
fig,axes = plt.subplots(1,2,figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(df_result_train['consumption'][:-8],label='Data')
plt.plot(df_result_train['predicted'][:-8],label='Predicted')
# plt.ylim([180,300])
plt.xticks(rotation=45)
plt.title("Forecasting on Training Set")
plt.legend();

plt.subplot(1,2,2)
plt.plot(df_result_test['consumption'],label='Data')
plt.plot(df_result_test['predicted'],label='Predicted')
# plt.ylim([200,330])
plt.xticks(rotation=45)
plt.title("Forecasting on Test Set MAPE=%.3f"%mean_absolute_percentage_error(y_test,yp))
plt.legend();
```





In [51]:
```
lasso_coefficients = pd.DataFrame({'variable':X.columns, 'coefficient':lasso_model.coef_}).sort_values(by=['coefficient'],ascer
sns.barplot(x='variable',y='coefficient',data=lasso_coefficients);
plt.xticks(rotation=90);
```



Random Forest Models

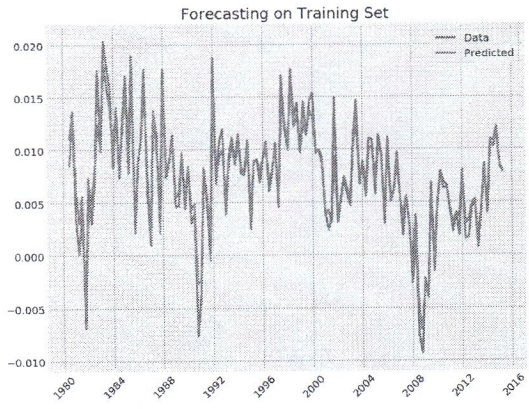
Let's apply random forest models and see whether they can improve the prediction.

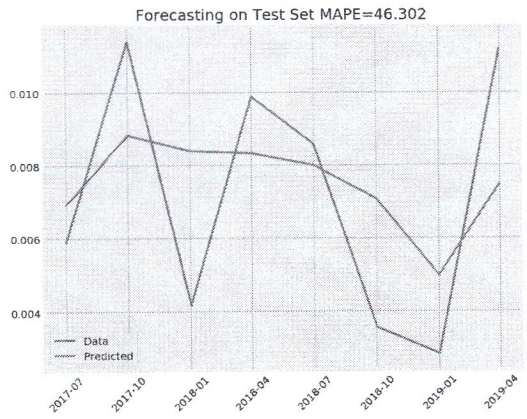
In [52]:
```
forest = RandomForestRegressor(n_estimators=100, random_state=1)
forest.fit(X_train, y_train)

forest_yt = forest.predict(X_train)
forest_yp = forest.predict(X_test)

print('RMSE on Test %.3f'%math.sqrt(mean_squared_error(y_test,forest_yp)))
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,forest_yp))
```

```
RMSE on Test 0.003
MAPE on Test 46.302
```

In [53]:
```
df_result_train = pd.DataFrame(y_train)
df_result_train['predicted'] = forest_yt
df_result_test = pd.DataFrame(y_test)
df_result_test['predicted'] = forest_yp
```

In [54]:
```
fig,axes = plt.subplots(1,2,figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(df_result_train['consumption'][:-8],label='Data')
plt.plot(df_result_train['predicted'][:-8],label='Predicted')
# plt.ylim([180,300])
plt.xticks(rotation=45)
plt.title("Forecasting on Training Set")
plt.legend();

plt.subplot(1,2,2)
plt.plot(df_result_test['consumption'],label='Data')
plt.plot(df_result_test['predicted'],label='Predicted')
# plt.ylim([200,330])
plt.xticks(rotation=45)
plt.title("Forecasting on Test Set MAPE=%.3f"%mean_absolute_percentage_error(y_test,forest_yp))
plt.legend();
```



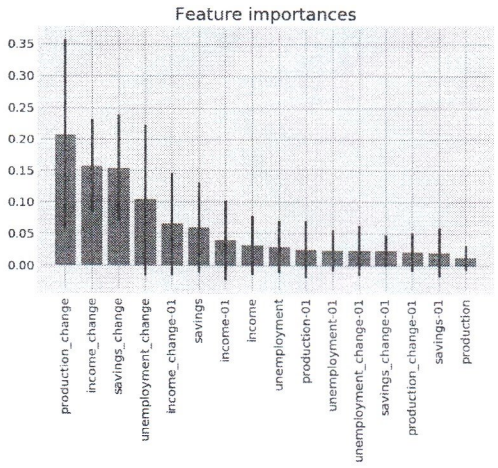


Random forests as other ensemble methods produce an evaluation of feature importance which we can visualize.

In [61]:
```
def PlotRFFeatureImportance(forest,feature_names,sort_importance=True):
    importances = forest.feature_importances_
    std = np.std([tree.feature_importances_ for tree in forest.estimators_],
                 axis=0)
    if (sort_importance):
        indices = np.argsort(importances)[::-1]
    else:
        indices = np.argsort(feature_names)[::-1]

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

#     for f in range(X.shape[1]):
#         print("%d. feature %d %s (%f)" % (f + 1, indices[f], 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]), feature_names[indices],rotation=90)
    plt.xlim([-1, X.shape[1]])
    plt.show()
```

In [62]:
```
PlotRFFeatureImportance(forest,input_variables,True)
```



In [63]:
```
etr = ExtraTreesRegressor(n_estimators=100, random_state=1)
etr.fit(X_train, y_train)

etr_yt = etr.predict(X_train)
etr_yp = etr.predict(X_test)

print('RMSE on Test %.3f'%math.sqrt(mean_squared_error(y_test,etr_yp)))
print('MAPE on Test %.3f'%mean_absolute_percentage_error(y_test,etr_yp))
```

```
RMSE on Test 0.003
MAPE on Test 54.715
```

What Next?

There are several other approaches we can try to improve our model, for example we might

- add new t-k variables
- add rolling windows
- ...

In [ ]:

Analysis of the Boston Housing Dataset

In this notebook, we are going to perform a regression analysis on the Boston Housing data set using all the available variables. The notebook has been derived from the following Kaggle notebooks:

- tolgahancepel's https://www.kaggle.com/tolgahancepel/boston-housing-regression-analysis/
- Prasad Perera's https://www.kaggle.com/prasadperera/the-boston-housing-dataset

In [1]:
```
# Basic libraries
import pandas as pd
import numpy as np
import math
from scipy import stats

# Regression models
from sklearn import linear_model
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, ElasticNet, Lasso, LassoCV

# Model evaluation
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

# Preprocessing
from sklearn.preprocessing import StandardScaler

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [2]:
```
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 24
4-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference
of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [3]:
```
df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['MEDV'] = boston.target
X = boston.data
y = boston.target
```

In [4]:
```
df.head()
```

Out[4]:

	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

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
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

In [5]:
```
df.describe()
```

Out[5]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

Note that all the variables have 506 values which is exactly the number of rows read. This means that we don't have any missing value. If you want to be sure we count the number of "null" values for each variables as follows:

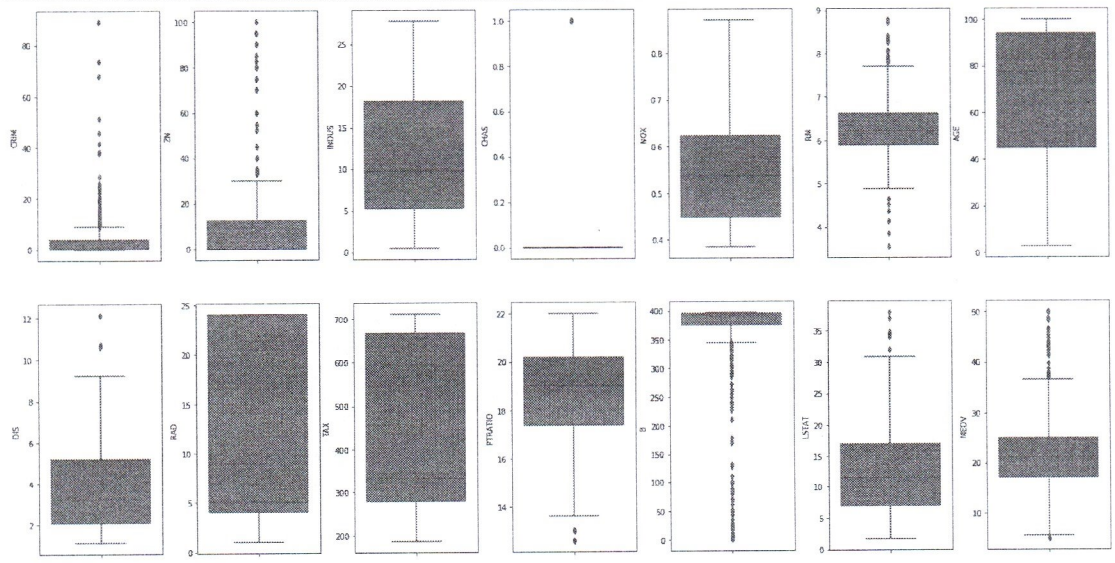
In [6]:
```
df.isnull().sum()
```

Out[6]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

Indeed, we don't have any missing value.

We can also check the value distribution of all the variables at once using boxplots.

In [7]:
```
fig, axs = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))
index = 0
axs = axs.flatten()
for k,v in df.items():
    sns.boxplot(y=k, data=df, ax=axs[index])
    index += 1
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=5.0)
```



The boxplots suggest that columns like CRIM, ZN, RM, B have outliers. Also the target variable MEDV has some outliers for values between 40 and 50 and this will come back later when we will be evaluating our linear regression models. We can check the percentage of outliers in every column by computing the first and third quartile (Q_1 and Q_3), the interquartile difference (IRQ) and check the percentage of values that are smaller than $Q_1 - 1.5*$IRQ and larger than $Q_3 + 1.5*$IRQ.

Note that attribute *CHAS* is a boolean variable so the corresponding boxplot is not informative for such type of variable and thus it should not be considered.

Note that this definition of *outlier* is specific to boxplots and does not define the broad concept of outlier which is typically task dependent and cannot be identified by one unique formula as done in boxplots.

In [8]:
```
outliers_percentage = []
variables = []
for k, v in df.items():
    Q1 = v.quantile(0.25)
    Q3 = v.quantile(0.75)
```

```
    IRQ = Q3 - Q1
    v_col = v[(v <= Q1 - 1.5 * IRQ) | (v >= Q3 + 1.5 * IRQ)]
    perc = np.shape(v_col)[0] * 100.0 / np.shape(df)[0]
    outliers_percentage.append(perc)
    variables.append(k)
#     print("Column %s outliers = %.2f%%" % (k, perc))

outliers = pd.DataFrame({'Variable':variables, '% Outliers':outliers_percentage })
outliers.sort_values(by=["% Outliers"],ascending=False)
```

Out[8]:

	Variable	% Outliers
3	CHAS	100.000000
11	B	15.217391
1	ZN	13.438735
0	CRIM	13.043478
13	MEDV	7.905138
5	RM	5.928854
10	PTRATIO	2.964427
12	LSTAT	1.383399
7	DIS	0.988142
2	INDUS	0.000000
4	NOX	0.000000
6	AGE	0.000000
8	RAD	0.000000
9	TAX	0.000000

Again, since *CHAS* is a boolean variable should not be looked at since the percentage of outliers in this case is meaningless. It is interesting to note that the target variable *MEDV* have 7.9% of outliers with values above 40 (see the boxplots) and this will come out again when analyzing the performance of linear regression models.

Some Data Exploration

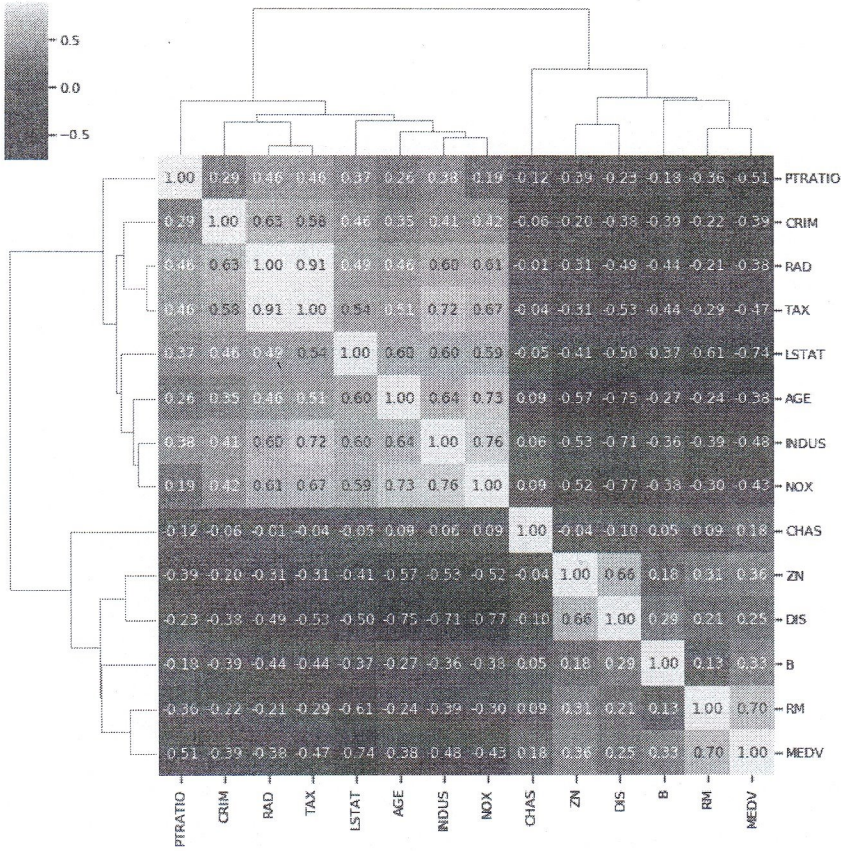
Let's explore the data a little bit. The data set is very simple but still it is good practice to do it anyway.

Correlation Analysis

We start by analyzing the correlations and plot the correlation matrix using a clustermap which applies hierarchical clustering on the rows/columns of the table.

In [9]:
```
corrmat = df.corr();
plt.figure(figsize=(12,12));

# Setting the default font size when plotting in notebooks (used to clear previous settings)
sns.set_context("notebook", font_scale=1.0, rc={"lines.linewidth": 2.5})

# Plot the clustermap
sns.clustermap(corrmat, annot=True,vmax=0.9,fmt=".2f");
```

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



The correlation matrix shows that *TAX* and *RAD* are highly positively correlated (0.91). Now, let's check on what variables are highly correlated with the target (*MEDV*) since they might be the candidate input variables for a predictive model.

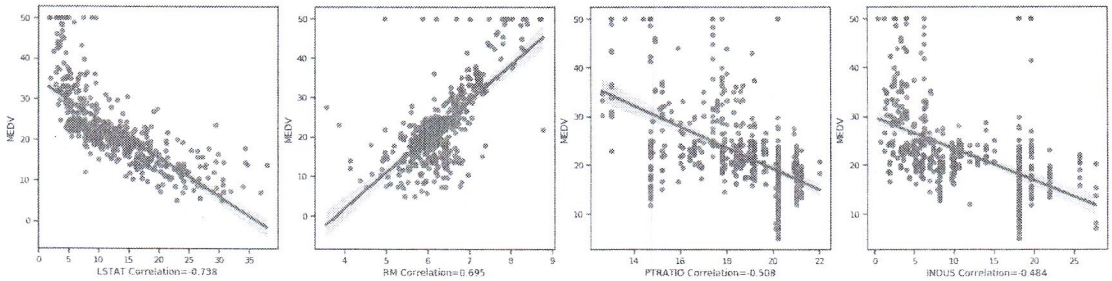
In [10]:
```
correlation_with_target = pd.DataFrame({'Variable':boston.feature_names,'Correlation':[df['MEDV'].corr(df[variable]) for variab

# we are interested both in positive and negative correlations so we are going to use their absolute value
correlation_with_target['|Correlation|'] = np.abs(correlation_with_target['Correlation'])

correlation_with_target.sort_values(by=['|Correlation|'],ascending=False)
```

Out[10]:

	Variable	Correlation	|Correlation|
12	LSTAT	-0.737663	0.737663
5	RM	0.695360	0.695360
10	PTRATIO	-0.507787	0.507787
2	INDUS	-0.483725	0.483725
9	TAX	-0.468536	0.468536
4	NOX	-0.427321	0.427321
0	CRIM	-0.388305	0.388305
8	RAD	-0.381626	0.381626
6	AGE	-0.376955	0.376955
1	ZN	0.360445	0.360445
11	B	0.333461	0.333461
7	DIS	0.249929	0.249929
3	CHAS	0.175260	0.175260

LSTAT, RM PTRAIO, INDUS have an absolute correlation score with *MEDV* that is near to 0.5 or above. This might suggest that these six variables might be good predictors. So for instance we could build our first models using one or more of these variables. Let's plot each variable against MEDV. Let's plot each variable against the target.

In [11]:
```
from sklearn.preprocessing import MinMaxScaler

# We scale the columns between 0 to 1 to have the same values on the x axis
scaler = MinMaxScaler()

high_correlated_variables = ['LSTAT', 'RM', 'PTRATIO','INDUS']
X_high_correlated_variables = df[high_correlated_variables]
y = df['MEDV']

X_normalized = scaler.fit_transform(X)

# x = pd.DataFrame(data=min_max_scaler.fit_transform(x), columns=column_sels)
fig, axs = plt.subplots(ncols=4, nrows=1, figsize=(20, 5))

axs = axs.flatten()
for i, variable in enumerate(high_correlated_variables):
    sns.regplot(y=y, x=X_high_correlated_variables[variable], ax=axs[i])
    axs[i].set_xlabel(variable+" Correlation=%.3f"%(df['MEDV'].corr(df[variable])))
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=5.0)
```



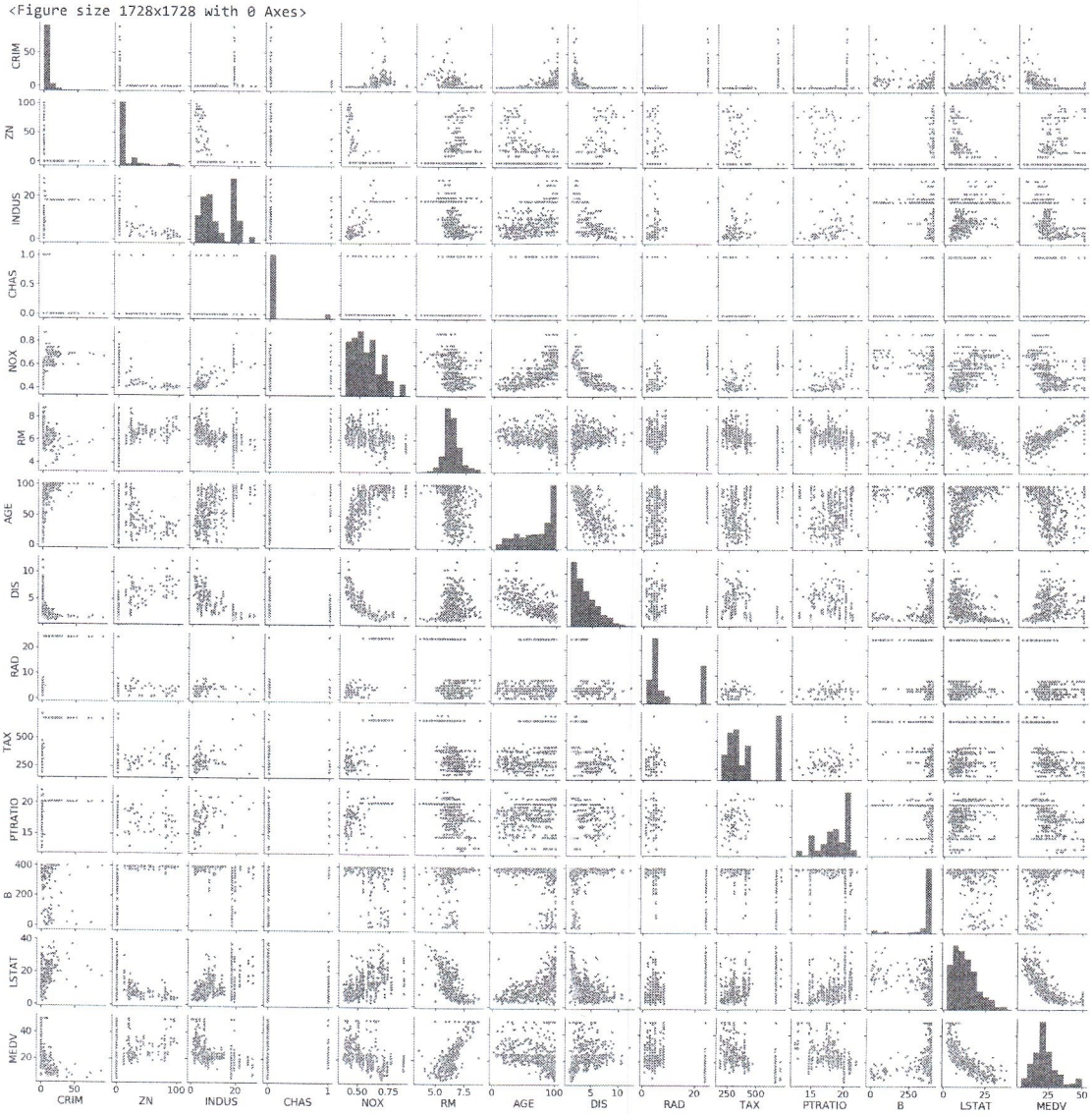
Note that, the variable with the highest correlation value (*LSTAT*) appears to be a reasonable predictor, as the linear regressor in the plot suggests. Also the linear regression using *RM* (the second mostly correlated variable) seems to capture a trend for *MEDV*. In fact, in many books, these two variables are used to demonstrate linear regression using this specific dataset.

Scatter Plots

Scatter plots are another good way to try to capture trends among variables. For example, we can have an overall view of the existing relations among variables by plotting the scatter plots between all the variable pairs.

In [12]:
```
plt.figure(figsize=(24,24));

# Increase font size when plotting in notebooks
sns.set_context("notebook", font_scale=2)

# Plot the pairplot
sns.pairplot(df);
plt.show();
```

```
<Figure size 1728x1728 with 0 Axes>
```



Linear Regression

We start with simple linear regression and build a model to predict MEDV. We want to show the difference between the evaluation using train/test sets and crossvalidation. For this purpose we will first split the data into train and test. Next we are going to evalutate linear regression using crossvalidation on the train set and then we are going to compare the performance reported using

- Crossvalidation on the train set
- Performance on the train set
- Performance on the test set

Since we are going to use crossvalidation several times and since we want to have replicable results we set the random seed and define the crossvalidation procedure once for all.

In [13]:
```
# random seed to be used when needed
random_seed=2398745

# uncomment this line to see a completely different result
# random_seed=983458690

# 10-fold crossvalidation
tenfold_xval = KFold(10, shuffle=True, random_state=random_seed)

# train and test generation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = random_seed)
```

In [14]:
```
regressor_linear = LinearRegression()
cv_linear = cross_val_score(regressor_linear, X_train, y_train, cv=tenfold_xval, scoring='r2')
regressor_linear.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_linear_train = r2_score(y_train, regressor_linear.predict(X_train))

# Computing R2 on the train set
r2_score_linear_test = r2_score(y_test, regressor_linear.predict(X_test))

# Computing the Root Mean Square Error
rmse_linear_test = (np.sqrt(mean_squared_error(y_test, regressor_linear.predict(X_test))))
```

In [15]:
```
print("             CV: %.3f +/ %.3f"%(cv_linear.mean(),cv_linear.std()))
print("R2_score (train): %.3f"%r2_score_linear_train)
print(" R2_score (test): %.3f"%r2_score_linear_test)
print("           RMSE: %.3f"%rmse_linear_test)
```

```
             CV: 0.707 +/ 0.072
R2_score (train): 0.737
 R2_score (test): 0.734
           RMSE: 4.961
```

As we can see the evaluation on the train set is *optimistic* in that it promise the higher R2 value. The performance reported using crossvalidation is much lower. The performance evaluation on the test is lower than the one measured on the train. But note that these results depend on a random process and might significantly change if we use a different random seed (try to replace the seed 2398745 with 983458690 and see what happens).

Crossvalidation & Confidence Intervals

Note that crossvalidation return 10 values and from the mean and standard deviation we can compute confidence intervals to estimate a range of plausible values for the model future performance. For example, given the ten data collected from crossvalidation, given a mean R2 of 0.707 and a standard deviation of 0.072 we have that,

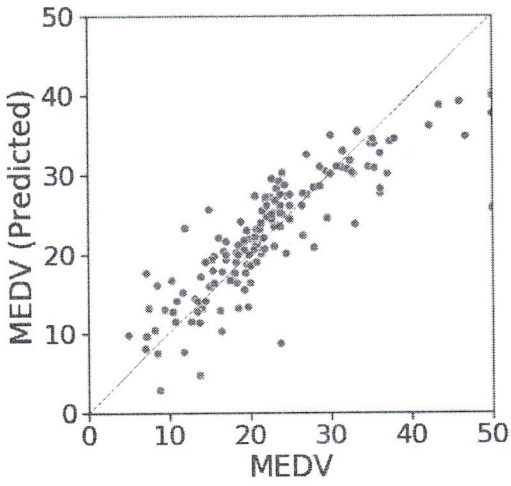
- the 95% confidence interval is (0.662; 0.752), thus with 95% probability we predict that the performance will be between 0.662 and 0.752
- the 99% confidence interval is (0.648; 0.766)
- the 99.5% confidence interval is (0.643; 0.771)

Confidence intervals give us a way to provide a range for future performance that can be very useful when we deploy the final model.

Residuals Plots

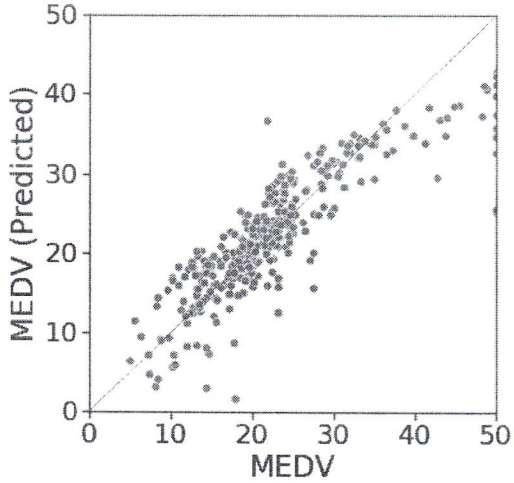
We can analyze the model errors either by plotting (i) the predicted values against the actual values of the target variable and (ii) the predicted values against the residual error.

In [16]:
```
plt.figure(figsize=(6,6))
# font = {'family' : 'sans', 'size'   : 14}
# plt.rc('font', **font)
# plt.plot([0,40],[0,0],'-',c="red")
# plt.plot([0,40],[-3,-3],'--',c="red")
# plt.plot([0,40],[3,3],'--',c="red")
plt.xlabel("MEDV (Actual)")
plt.ylabel("MEDV (Predicted)")
# plt.ylabel("Normalized Residuals")
plt.plot([0,50],[0,50],lw=.5,ls="-.",color='black')
sns.scatterplot(y_test, regressor_linear.predict(X_test),s=50)
plt.xlim([0,50])
plt.ylim([0,50])
plt.show()
```



It is interesting to check whether the model has the same behavior also on the train set. If it does not, it might suggest that we are learning a model that is not adequate for the test set we are considering. Note that, the values of *MEDV* of 50 (the same that we identified as outlier in our exploratory analysis) are predicted very badly. One possible solution might be to eliminate those few cases and continue our analysis without using the data points with a value of *MEDV* above 50.

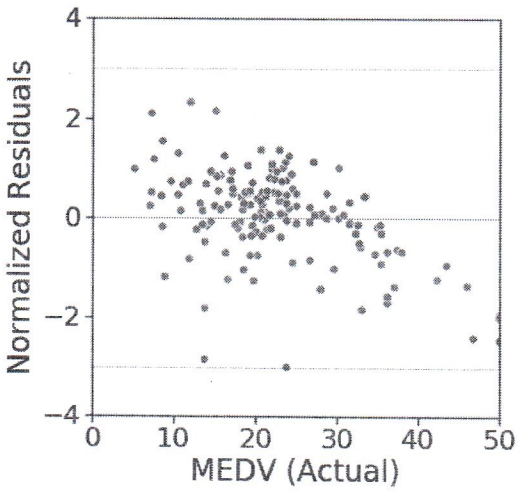
In [17]:
```
plt.figure(figsize=(6,6))
plt.xlabel("MEDV (Actual)")
plt.ylabel("MEDV (Predicted)")
plt.plot([0,50],[0,50],lw=.5,ls="-.",color='black')
sns.scatterplot(y_train, regressor_linear.predict(X_train),s=50)
plt.xlim([0,50])
plt.ylim([0,50])
plt.show()
```



As can be noted, the model underestimate the prediction for higher values of the target variable: the dots for high values of "MEDV (Actual)" lie below the line. Again the outlier values of *MEDV* (above 50) are predicted very badly and in a real scenario they might be eliminated from the analysis process.

In [18]:
```
residuals_test = regressor_linear.predict(X_test)-y_test
normalized_residuals_test = StandardScaler().fit_transform(residuals_test.values.reshape(-1,1)).reshape(-1,)
```

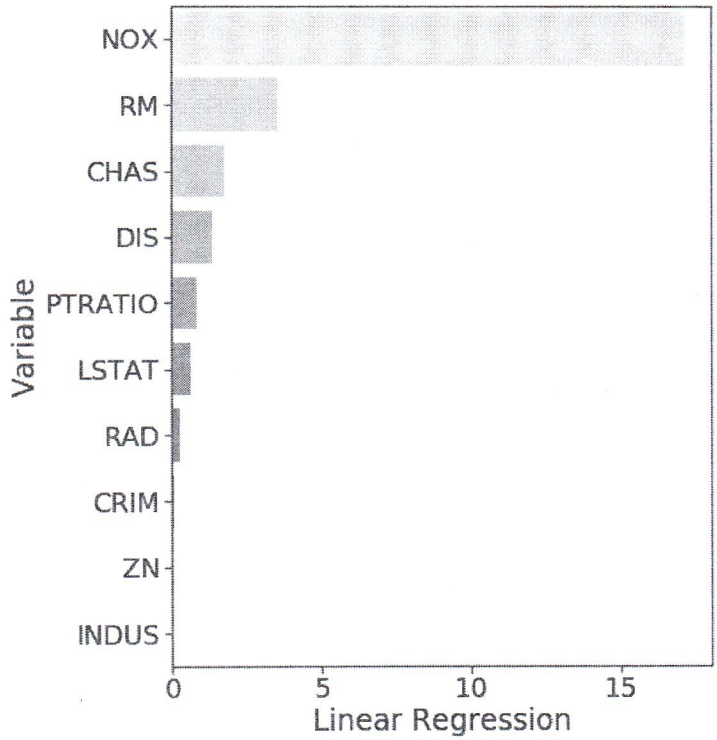
In [19]:
```
plt.figure(figsize=(6,6))
# font = {'family' : 'sans', 'size'   : 14}
# plt.rc('font', **font)
plt.plot([0,50],[0,0],'-',c="red",lw=0.5)
plt.plot([0,50],[-3,-3],'--',c="red",lw=0.5)
plt.plot([0,60],[3,3],'--',c="red",lw=0.5)
sns.scatterplot(y_test, normalized_residuals_test,s=50)
plt.xlabel("MEDV (Actual)")
plt.ylabel("Normalized Residuals")
plt.xlim([0,50])
plt.ylim([-4,4])
plt.show()
```



Feature Analysis

We can analyze what are the most influential variables for the prediction by plotting the 10 largest coefficients (using absolute values).

In [22]:
```
coefficients = pd.DataFrame({'Variable':boston.feature_names, \
                             'Linear Regression':np.abs(regressor_linear.coef_)})
sorted_linear_coefficients = coefficients.sort_values(by=['Linear Regression'], ascending=False)

plt.figure(figsize=(8,10))
sns.barplot(x='Linear Regression', y='Variable', data = sorted_linear_coefficients.head(10),palette='Blues');
```



Also this plot shows that the model underestimate the prediction for higher values of MEDV.

Polynomial Regression

We can try to improve our model by fitting a second degree polynomial. To do this, we must,

1. Extend our data by adding the polynomial features for degree 2 to the train and test sets
2. Repeat the above procedure using the new data sets

In [23]:
```
### STEP 1. Extend the data adding polynomial features

# import the library to generate polynomial features from the existing ones
from sklearn.preprocessing import PolynomialFeatures

# Create the polynomial features for the train and test sets. We don't want the bias (constant column added) and also
# we are only using variable interactions (e.g. x*y) rather than squared values (x^2)
polynomial = PolynomialFeatures(degree = 2,include_bias=False,interaction_only=True)
X2_train = polynomial.fit_transform(X_train)
X2_test = polynomial.transform(X_test)
```

In [24]:
```
### STEP 2. Repeat the evaluation
regressor_polynomial = LinearRegression()

cv_polynomial = cross_val_score(regressor_polynomial, X2_train, y_train, cv=tenfold_xval, scoring='r2')
regressor_polynomial.fit(X2_train,y_train)

# Computing R2 on the train set
r2_score_polynomial_train = r2_score(y_train, regressor_polynomial.predict(X2_train))

# Computing R2 on the train set
r2_score_polynomial_test = r2_score(y_test, regressor_polynomial.predict(X2_test))

# Computing the Root Mean Square Error
rmse_polynomial_test = (np.sqrt(mean_squared_error(y_test, regressor_polynomial.predict(X2_test))))
```

In [25]:
```
### STEP 3. Print the results
print("                CV: %.3f +/ %.3f"%(cv_polynomial.mean(),cv_polynomial.std()))
print("R2_score (train): %.3f"%r2_score_polynomial_train)
print(" R2_score (test): %.3f"%r2_score_polynomial_test)
print("              RMSE: %.3f"%rmse_polynomial_test)
```

```
                CV: 0.821 +/ 0.057
R2_score (train): 0.931
 R2_score (test): 0.819
              RMSE: 4.097
```

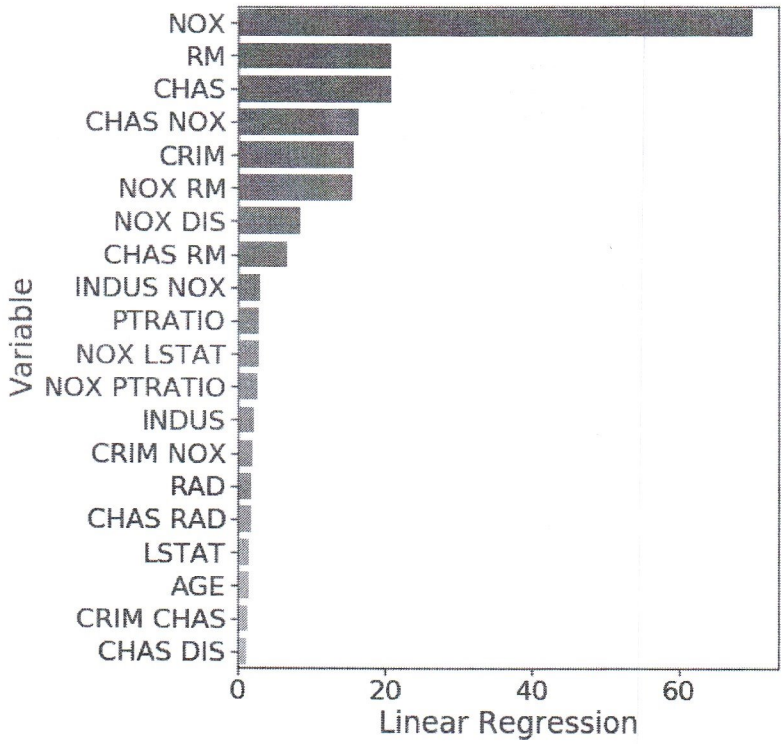
The prediction is higher than the model using only the original variables. This suggests that there might be some interesting interactions among the problem features.

Feature Analysis

We can analyze what are the most influential variables for the prediction by plotting the 20 largest coefficients. As we can note several variables that are listed represent interactions of the original variables.

In [26]:
```
coefficients = pd.DataFrame({'Variable':polynomial.get_feature_names(boston.feature_names), \
                             'Linear Regression':np.abs(regressor_polynomial.coef_)})
sorted_coefficients = coefficients.sort_values(by=['Linear Regression'], ascending=False)
```

In [27]:
```
plt.figure(figsize=(8,10))
sns.barplot(x='Linear Regression', y='Variable', data = sorted_coefficients.head(20),palette='Reds_d');
```



Ridge and Lasso Regression

We now apply Ridge (L_2) and Lasso (L_1) regression. First, we need to apply normalization to the data set since there are variables that have very different ranges. For example, variable *B* and *Tax* have values measured in hundreds while *LSTAT* has much lower values. Next, we apply Ridge and Lasso regression using the same approach as before.

Normalization

When we have variables of different scales, although the algorithm processes each variable separately, the optimization process will tend to be dominated by the variables with higher values. This has also an impact on the number of iterations required to convergence and sometimes, as we have seen in some examples, we might not reach convergence or have numerical stability issues depending on the method. The solution is to normalize the data so that all the features are on the same scale. We can apply basic range normalization which rescale the values in the interval between 0 and 1. We can apply z-score normalization (also called standardization) that rescale a feature to have mean zero and variance one.

In [28]:
```
# STEP #1 Standardize the data

# we apply standardization
scaler = StandardScaler()
Xs_train = scaler.fit_transform(X_train)
Xs_test = scaler.transform(X_test)

# next we generate the polynomial features

polynomial = PolynomialFeatures(degree = 2,include_bias=False,interaction_only=True)
X2s_train = polynomial.fit_transform(Xs_train)
X2s_test = polynomial.transform(Xs_test)
```

In [29]:
```
### STEP 2. Repeat the evaluation
ridge_polynomial = Ridge()

cv_ridge = cross_val_score(ridge_polynomial, X2s_train, y_train, cv=tenfold_xval, scoring='r2')
ridge_polynomial.fit(X2s_train,y_train)

# Computing R2 on the train set
r2_score_ridge_train = r2_score(y_train, ridge_polynomial.predict(X2s_train))

# Computing R2 on the train set
r2_score_ridge_test = r2_score(y_test, ridge_polynomial.predict(X2s_test))

# Computing the Root Mean Square Error
rmse_ridge_test = (np.sqrt(mean_squared_error(y_test, ridge_polynomial.predict(X2s_test))))
```

In [30]:
```
### STEP 3. Print the results
print("             CV: %.3f +/ %.3f"%(cv_ridge.mean(),cv_ridge.std()))
print("R2_score (train): %.3f"%r2_score_ridge_train)
print(" R2_score (test): %.3f"%r2_score_ridge_test)
print("           RMSE: %.3f"%rmse_ridge_test)
```

```
             CV: 0.846 +/ 0.055
R2_score (train): 0.928
 R2_score (test): 0.847
           RMSE: 3.757
```

Scikit-Learn Pipelines

Instead of repeating all the steps over and over, we can create a pipeline that encode the sequence of polynomial feature creation, normalization, and modeling. We now apply Lasso. We use a value of α of 0.01 but we could as well applied a procedure to compute the best possible value of α .

In [31]:
```
from sklearn.pipeline import Pipeline

lasso_steps = [
```

```
    ('scaler', StandardScaler()),
    ('polynomial', PolynomialFeatures(degree = 2,include_bias=False,interaction_only=True)),
    ('model', Lasso(alpha=0.01,max_iter=3000))
]

lasso_pipeline = Pipeline(lasso_steps)
```

In [32]:
```
# STEP 1. Repeat the whole process but this time, when I invoke fit, it will run the entire sequence of functions
#         In fact, I am starting from the raw train and test data
cv_lasso = cross_val_score(lasso_pipeline, X_train, y_train, cv=tenfold_xval, scoring='r2')
lasso_pipeline.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_lasso_train = r2_score(y_train, lasso_pipeline.predict(X_train))

# Computing R2 on the train set
r2_score_lasso_test = r2_score(y_test, lasso_pipeline.predict(X_test))

# Computing the Root Mean Square Error
rmse_lasso_test = (np.sqrt(mean_squared_error(y_test, lasso_pipeline.predict(X_test))))
```

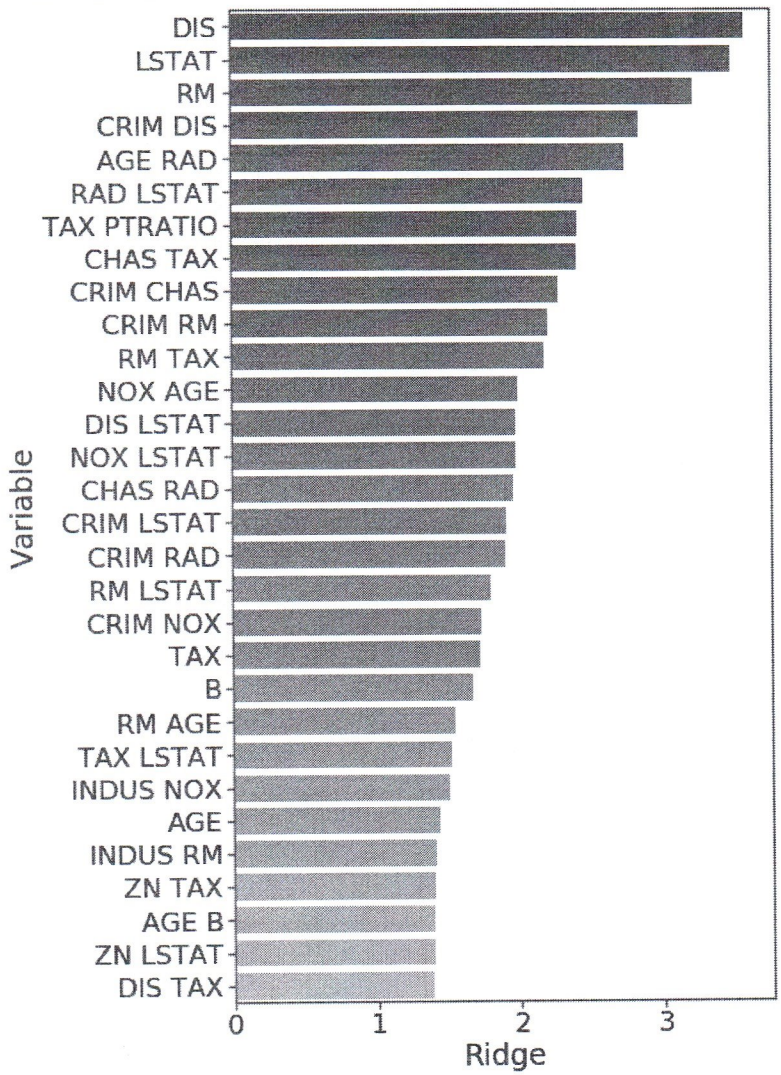
In [33]:
```
### STEP 2. Print the results
print("             CV: %.3f +/ %.3f"%(cv_lasso.mean(),cv_lasso.std()))
print("R2_score (train): %.3f"%r2_score_lasso_train)
print(" R2_score (test): %.3f"%r2_score_lasso_test)
print("            RMSE: %.3f"%rmse_lasso_test)
```

```
             CV: 0.845 +/ 0.070
R2_score (train): 0.925
 R2_score (test): 0.859
            RMSE: 3.607
```

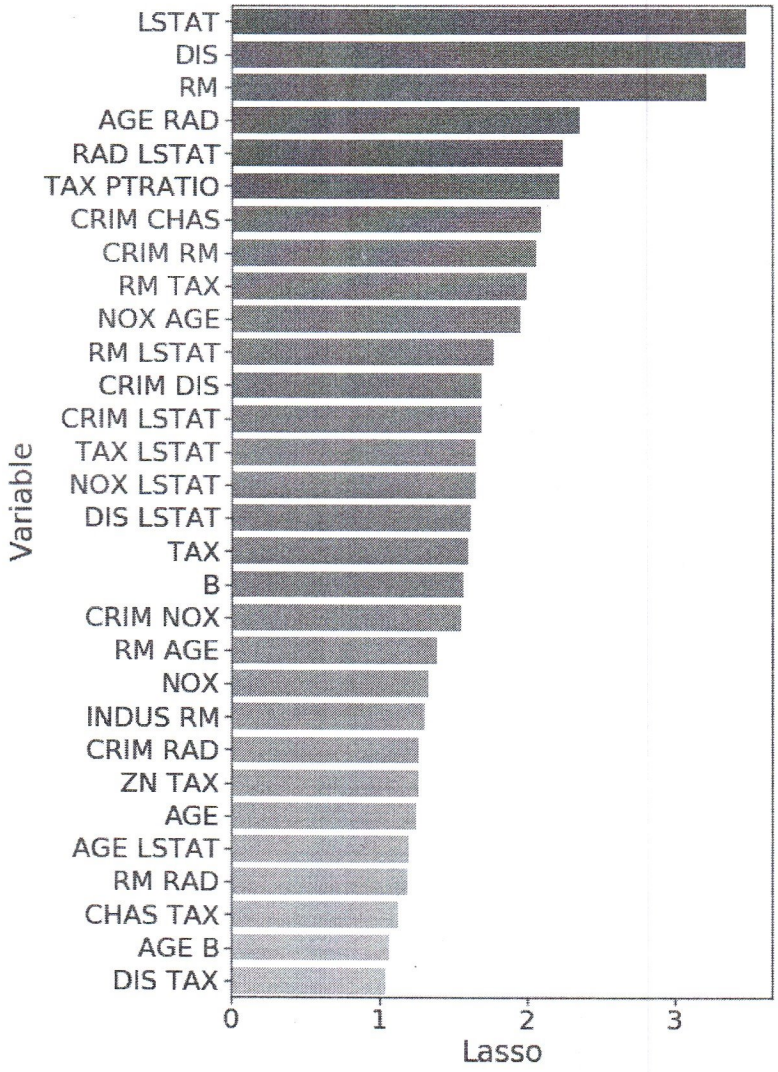
Analysis of Feature Importance

We can repeat the analysis of feature importance for the Ridge and Lasso regression. Note that we should not compare these values with the ones computed for simple linear regression since the first model was fitted using the original data, whereas these two models are based on the standardized data.

In [34]:
```
coefficients['Ridge'] = np.abs(ridge_polynomial.coef_)
coefficients['Lasso'] = np.abs(lasso_pipeline[2].coef_)
plt.figure(figsize=(8,15))
sns.barplot(x='Ridge', y='Variable', data = coefficients.sort_values(by=['Ridge'],ascending=False).head(30),palette='Greens_d')
```



In [35]:
```
plt.figure(figsize=(8,15))
sns.barplot(x='Lasso', y='Variable', data = coefficients.sort_values(by=['Lasso'],ascending=False).head(30),palette='Blues_d');
```



Note that the weights produced by Linear Regression, Ridge and Lasso have much different scales. The first plot shows weight absolute values ranges between zero and 20 whereas for Ridge and Lasso have values between 0 and 3.5.

The α parameter influence the weight penalization. If we increase α to 1 in Lasso for example, we will increase weight penalization so that we will get smaller weight values and for several variables the weights will go to zero. However, this will also lead to a decrease of the model performance.

k-Nearest Neighbor Regression

Now, let's go further and apply other regression methods. We start from k-nearest neighbor regression.

In [36]:
```
from sklearn.neighbors import KNeighborsRegressor

knn_steps = [
    ('scaler', StandardScaler()),
    ('polynomial', PolynomialFeatures(degree = 2,include_bias=False,interaction_only=True)),
    ('model', KNeighborsRegressor(n_neighbors= 5, weights='uniform', algorithm='kd_tree', leaf_size=30))
]

knn_pipeline = Pipeline(knn_steps)
```

In [37]:
```
# STEP 1. Repeat the whole process but this time, when I invoke fit, it will run the entire sequence of functions
#         In fact, I am starting from the raw train and test data
cv_knn = cross_val_score(knn_pipeline, X_train, y_train, cv=tenfold_xval, scoring='r2')
knn_pipeline.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_knn_train = r2_score(y_train, knn_pipeline.predict(X_train))

# Computing R2 on the train set
r2_score_knn_test = r2_score(y_test, knn_pipeline.predict(X_test))

# Computing the Root Mean Square Error
rmse_knn_test = (np.sqrt(mean_squared_error(y_test, knn_pipeline.predict(X_test))))
```

In [38]:
```
### STEP 2. Print the results
print("             CV: %.3f +/ %.3f"%(cv_knn.mean(),cv_knn.std()))
print("R2_score (train): %.3f"%r2_score_knn_train)
print(" R2_score (test): %.3f"%r2_score_knn_test)
print("            RMSE: %.3f"%rmse_knn_test)
```

```
             CV: 0.725 +/ 0.142
R2_score (train): 0.825
 R2_score (test): 0.776
            RMSE: 4.551
```

We applied k-NN with with a value of k equal to 5. If we use a smaller k (like for example 1), the average performance is lower but, most importantly, the corresponding standard deviation is likely to increase because the predicted value is based only on one data point and might be

noisy. If we use a larger k (like for example 10 or 20) the prediction the associated standard deviation is likely to decrease as k increases. Overall, a value of 5 for k might be a good trade-off although its predicted performance is much worse than previous linear models.

Regression using Decision Trees (or Regression Trees)

We can apply decision trees to compute a regression model. Note that decision trees don't use the input variables for computation but work by partitioning the input space. Accordingly, they don't need any normalization.

In [39]:
```
### STEP 1. Evaluate the model
from sklearn.tree import DecisionTreeRegressor

regressor_tree = DecisionTreeRegressor(max_depth=3, random_state=random_seed)
cv_tree = cross_val_score(regressor_tree, X_train, y_train, cv=tenfold_xval, scoring='r2')
regressor_tree.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_tree_train = r2_score(y_train, regressor_tree.predict(X_train))

# Computing R2 on the train set
r2_score_tree_test = r2_score(y_test, regressor_tree.predict(X_test))

# Computing the Root Mean Square Error
rmse_tree_test = (np.sqrt(mean_squared_error(y_test, regressor_tree.predict(X_test))))
```

In [40]:
```
### STEP 2. Print the results
print("              CV: %.3f +/ %.3f"%(cv_tree.mean(),cv_tree.std()))
print("R2_score (train): %.3f"%r2_score_tree_train)
print(" R2_score (test): %.3f"%r2_score_tree_test)
print("            RMSE: %.3f"%rmse_tree_test)
```

```
              CV: 0.700 +/ 0.160
R2_score (train): 0.831
 R2_score (test): 0.631
            RMSE: 5.841
```

Random Forests

One tree performs rather poorly compared to linear regressors. We can try to use an entire forest of trees and check what happens.

In [41]:
```
### STEP 1. Evaluate the model
from sklearn.ensemble import RandomForestRegressor

regressor_rf = RandomForestRegressor(n_estimators = 500, random_state = random_seed)

cv_rf = cross_val_score(regressor_rf, X_train, y_train, cv=tenfold_xval, scoring='r2')
regressor_rf.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_rf_train = r2_score(y_train, regressor_rf.predict(X_train))

# Computing R2 on the train set
r2_score_rf_test = r2_score(y_test, regressor_rf.predict(X_test))

# Computing the Root Mean Square Error
rmse_rf_test = (np.sqrt(mean_squared_error(y_test, regressor_rf.predict(X_test))))
```

In [42]:
```
### STEP 2. Print the results
print("              CV: %.3f +/ %.3f"%(cv_rf.mean(),cv_rf.std()))
print("R2_score (train): %.3f"%r2_score_rf_train)
print(" R2_score (test): %.3f"%r2_score_rf_test)
print("            RMSE: %.3f"%rmse_rf_test)
```

```
              CV: 0.840 +/ 0.112
R2_score (train): 0.978
 R2_score (test): 0.871
            RMSE: 3.458
```

Gradient Boosting Regressor

Finally, let's apply gradient boosting. If you want to try the very powerful xgboost you have to install the library separately from https://xgboost.readthedocs.io/

In [43]:
```
from sklearn.ensemble import GradientBoostingRegressor

regressor_gbr = GradientBoostingRegressor(alpha=0.9,learning_rate=0.05, max_depth=2, min_samples_leaf=5, min_samples_split=2, r

cv_gbr = cross_val_score(regressor_gbr, X_train, y_train, cv=tenfold_xval, scoring='r2')
regressor_gbr.fit(X_train,y_train)

# Computing R2 on the train set
r2_score_gbr_train = r2_score(y_train, regressor_gbr.predict(X_train))

# Computing R2 on the train set
r2_score_gbr_test = r2_score(y_test, regressor_rf.predict(X_test))

# Computing the Root Mean Square Error
rmse_gbr_test = (np.sqrt(mean_squared_error(y_test, regressor_gbr.predict(X_test))))
```

In [44]:
```
### STEP 2. Print the results
print("              CV: %.3f +/ %.3f"%(cv_gbr.mean(),cv_gbr.std()))
print("R2_score (train): %.3f"%r2_score_gbr_train)
print(" R2_score (test): %.3f"%r2_score_gbr_test)
print("            RMSE: %.3f"%rmse_gbr_test)
```

```
              CV: 0.842 +/ 0.105
R2_score (train): 0.975
 R2_score (test): 0.871
            RMSE: 3.710
```

Comparing Perfomance

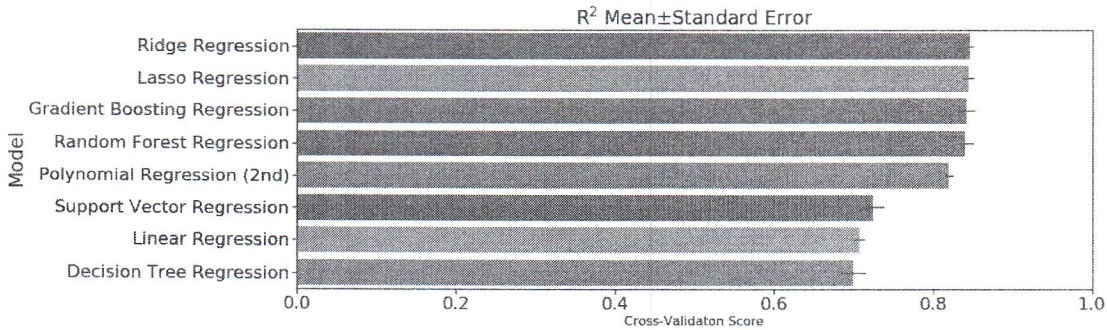
We can now collect all the data about the performance of all the models and discuss our findings. The bar plot below reports for each model the average R^2 with error bars showing the standard error σ/k, with k is the number of crossvalidation folds.

In [45]:
```
models = [('Linear Regression', rmse_linear_test, r2_score_linear_train, r2_score_linear_test, cv_linear.mean(), cv_linear.std(
          ('Polynomial Regression (2nd)', rmse_polynomial_test, r2_score_polynomial_train, r2_score_polynomial_test, cv_polynon
          ('Ridge Regression', rmse_ridge_test, r2_score_ridge_train, r2_score_ridge_test, cv_ridge.mean(), cv_ridge.std()),
          ('Lasso Regression', rmse_lasso_test, r2_score_lasso_train, r2_score_lasso_test, cv_lasso.mean(), cv_lasso.std()),
          ('Support Vector Regression', rmse_knn_test, r2_score_knn_train, r2_score_knn_test, cv_knn.mean(), cv_knn.std()),
          ('Decision Tree Regression', rmse_tree_test, r2_score_tree_train, r2_score_tree_test, cv_tree.mean(), cv_tree.std()),
          ('Random Forest Regression', rmse_rf_test, r2_score_rf_train, r2_score_rf_test, cv_rf.mean(), cv_rf.std()),
          ('Gradient Boosting Regression', rmse_gbr_test, r2_score_gbr_train, r2_score_gbr_test, cv_gbr.mean(), cv_gbr.std())
         ]
predict = pd.DataFrame(data = models, columns=['Model', 'RMSE', 'R2_Score(training)', 'R2_Score(test)', 'CV Mean', 'CV Std'])
predict.sort_values(by=['CV Mean'],ascending=False)
```

Out[45]:

	Model	RMSE	R2_Score(training)	R2_Score(test)	CV Mean	CV Std
2	Ridge Regression	3.757497	0.927628	0.847445	0.846371	0.054659
3	Lasso Regression	3.606973	0.924641	0.859423	0.845193	0.070051
7	Gradient Boosting Regression	3.709825	0.974818	0.870805	0.842499	0.104721
6	Random Forest Regression	3.457868	0.977877	0.870805	0.839796	0.111578
1	Polynomial Regression (2nd)	4.096625	0.931109	0.818665	0.820551	0.056508
4	Support Vector Regression	4.550668	0.825449	0.776242	0.724833	0.141843
0	Linear Regression	4.961402	0.737261	0.734027	0.707284	0.071732
5	Decision Tree Regression	5.841354	0.830514	0.631315	0.700018	0.160289

In [46]:
```
f, axe = plt.subplots(1,1, figsize=(18,6))

predict.sort_values(by=['CV Mean'], ascending=False, inplace=True)

sns.barplot(x='CV Mean', y='Model', data = predict, xerr=predict['CV Std']/10, ax = axe)
#axes[0].set(xlabel='Region', ylabel='Charges')
axe.set_xlabel('Cross-Validaton Score', size=16)
axe.set_ylabel('Model')
axe.set_title("R$^2$ Mean$\pm$Standard Error")
axe.set_xlim(0,1.0)
plt.show()
```



Model Selection

We now need to decide what model to select. We evaluated all the models using crossvalidation so we can apply t-test to check whether the difference in performance is statistically significant. Next, we can decide what model we should deploy. It is rather clear that the performance of SVR, Linear Regression and Decision Trees will turn out to be statistically significant. It is interesting to check whether the difference in the top four-five models is statistically significant.

First, we generate a data frame containing all the raw crossvalidation performance. Next, we apply t-test to all the possible pairwise combination. Note that, since the all the crossvalidation were computed using the same folds, we can apply a paired t-test.

In [47]:
```
df_crossvalidation = pd.DataFrame({
    'Ridge Regression':cv_ridge,
    'Lasso Regression':cv_lasso,
    'Gradient Boosting Regression':cv_gbr,
    'Random Forest Regression':cv_rf,
    'Polynomial Regression (2nd)':cv_polynomial,
    'Support Vector Regression':cv_knn,
    'Linear Regression':cv_linear,
    'Decision Tree Regression':cv_tree,
})
```

In [48]:
```
confidence_level = 0.95

no_variables = len(df_crossvalidation.columns)

p_value = np.zeros((no_variables,no_variables))

for first,first_model in enumerate(df_crossvalidation.columns):

    p_value[first,first] = 1.0

    for second in range(first+1,(len(df_crossvalidation.columns))):

        second_model = df_crossvalidation.columns[second]

        paired_test = stats.ttest_rel(df_crossvalidation[first_model], df_crossvalidation[second_model])

        p_value[first,second] = paired_test[1]
        p_value[second,first] = paired_test[1]
```

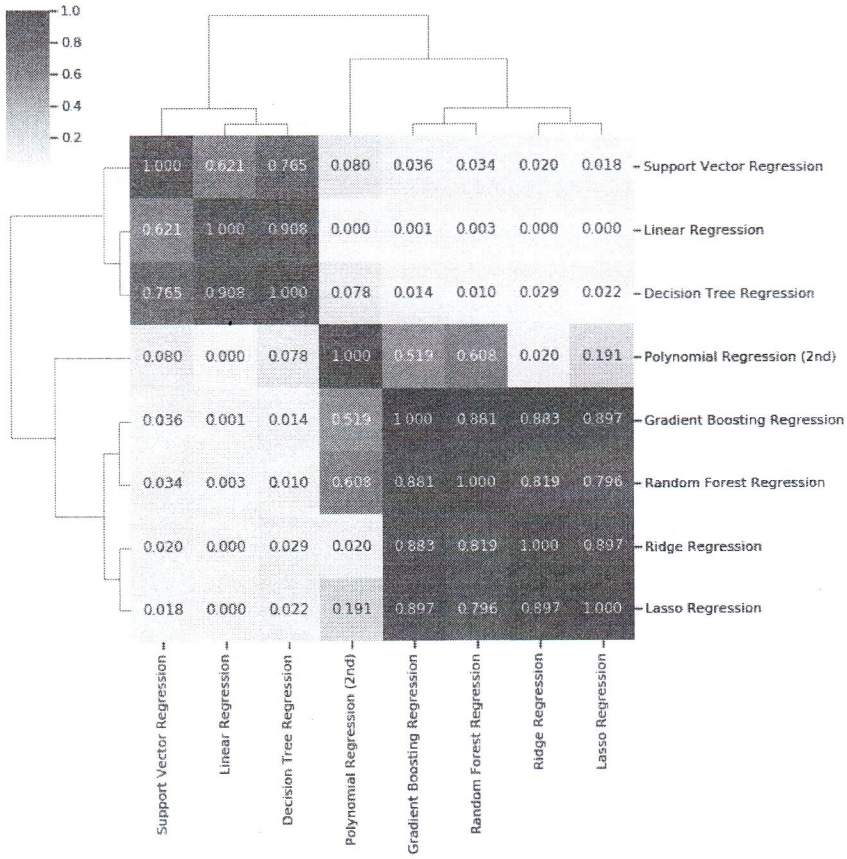
```
        if (paired_test[1]<(1-confidence_level)):
            print("%15s vs %15s => Difference is statistically significant (cf %3.2f p-value=%.4f)"%(first_model,second_model,c
```

```
Ridge Regression vs Polynomial Regression (2nd) => Difference is statistically significant (cf 95.00 p-value=0.0200)
Ridge Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0199)
Ridge Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0000)
Ridge Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0287)
Lasso Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0176)
Lasso Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0000)
Lasso Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0223)
Gradient Boosting Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0357)
Gradient Boosting Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0008)
Gradient Boosting Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0144)
Random Forest Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0341)
Random Forest Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0028)
Random Forest Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0104)
Polynomial Regression (2nd) vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0004)
```

In [49]:
```
p_value_mat = pd.DataFrame(data=p_value,index=df_crossvalidation.columns,columns=df_crossvalidation.columns)
```

In [50]:
```
sns.set_context("notebook", font_scale=1.0, rc={"lines.linewidth": 2.5});
# Plot the clustermap
# sns.clustermap(corrmat, annot=True,vmax=0.9,fmt=".2f");
plt.figure(figsize=(16,16));
sns.clustermap(p_value_mat,annot=True,fmt=".3f",cmap="Blues");
```

```
<Figure size 1152x1152 with 0 Axes>
```



As can be noted, the difference all the top four approaches are not statistically significant since their p-value is almost one and thus must higher than 0.05 (computed as 1-confidence level). Also the difference with polynomial regression is not statistically significant but has a lower p-value. And its difference with the top scoring method (Ridge Regression) is actually statistically significant.

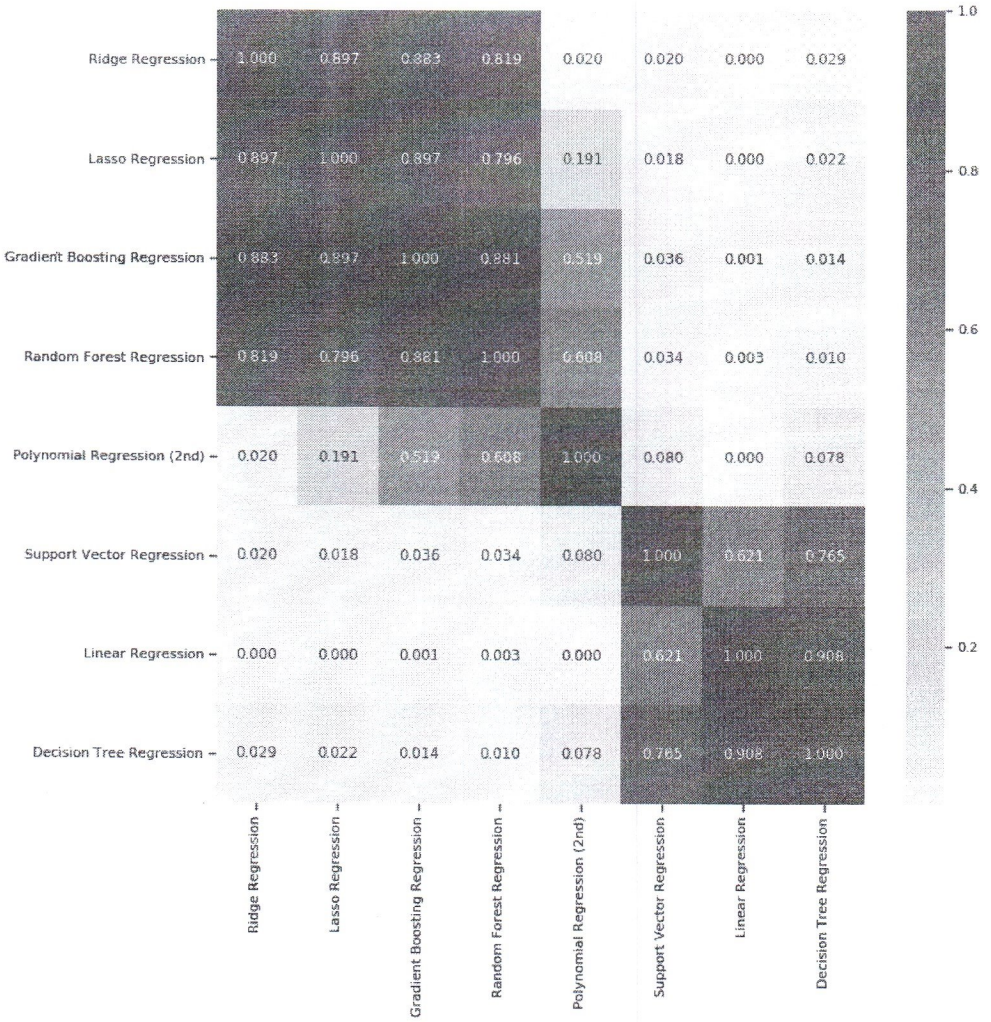
I can repeat the same analysis using a non-parametric test, that does not assume that the performance is normally distributed.

In [155...
```
confidence_level = 0.95

no_variables = len(df_crossvalidation.columns)

wilcoxon_p_value = np.zeros((no_variables,no_variables))

for first,first_model in enumerate(df_crossvalidation.columns):

    wilcoxon_p_value[first,first] = 1.0

    for second in range(first+1,(len(df_crossvalidation.columns))):

        second_model = df_crossvalidation.columns[second]

        wilcoxon_test = stats.wilcoxon(df_crossvalidation[first_model], df_crossvalidation[second_model])

        wilcoxon_p_value[first,second] = wilcoxon_test[1]
        wilcoxon_p_value[second,first] = wilcoxon_test[1]


        if (wilcoxon_test[1]<(1-confidence_level)):
            print("%15s vs %15s => Difference is statistically significant (cf %3.2f p-value=%.4f)"%(first_model,second_model,c
```

```
Ridge Regression vs Polynomial Regression (2nd) => Difference is statistically significant (cf 95.00 p-value=0.0284)
Ridge Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0166)
Ridge Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0051)
Ridge Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0367)
Lasso Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0166)
Lasso Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0051)
Lasso Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0284)
```

```
Gradient Boosting Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0367)
Gradient Boosting Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0069)
Gradient Boosting Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0218)
Random Forest Regression vs Support Vector Regression => Difference is statistically significant (cf 95.00 p-value=0.0367)
Random Forest Regression vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0125)
Random Forest Regression vs Decision Tree Regression => Difference is statistically significant (cf 95.00 p-value=0.0166)
Polynomial Regression (2nd) vs Linear Regression => Difference is statistically significant (cf 95.00 p-value=0.0093)
```

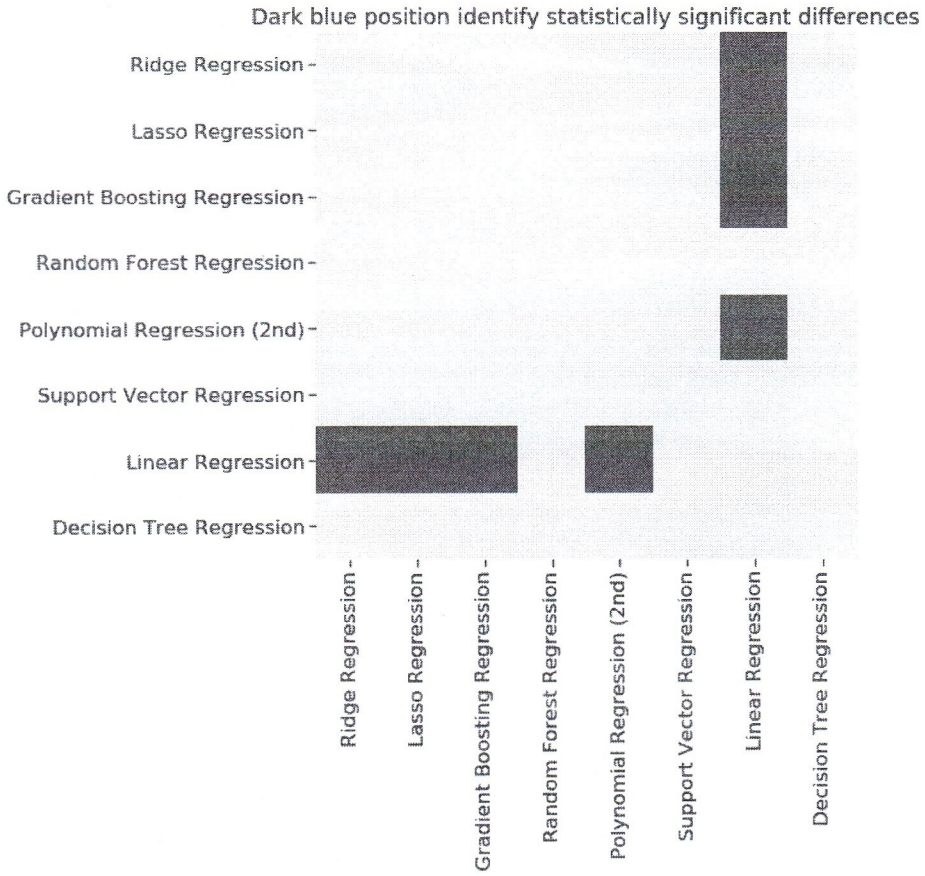
In [156...]:
```
sns.set_context("notebook", font_scale=1.0, rc={"lines.linewidth": 2.5});
# Plot the clustermap
# sns.clustermap(corrmat, annot=True,vmax=0.9,fmt=".2f");
plt.figure(figsize=(12,12));
sns.heatmap(p_value_mat,annot=True,fmt=".3f",cmap="Blues");
```



Note that we performed 28 comparisons since we have 8 classifiers and thus we perfomed 7x8/2 (or "8 choose 2") comparisons we might want to applyBonferroni's correction. Our current α for a 95% confidence level is 0.05 the adjusted version would be 0.05/28 that is, 0.0018. If we use this new adjusted threshold then every difference becomes not statistically significant except the ones against linear regression. It would be a very robust approach but not really useful to select a model.

In [64]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
ax = plt.figure(figsize=(8,8))
sns.heatmap(p_value_mat<(1-confidence_level)/28,fmt=".3f",cmap="Blues",cbar=False)
plt.title("Dark blue position identify statistically significant differences")
```

Out[64]:
```
Text(0.5, 1, 'Dark blue position identify statistically significant differences')
```



Discussion

We built several models for the Boston Housing dataset. Ridge and Lasso perform slightly better than Gradient Boosting and Random Forests which perform better than basic linear regression using 2nd degree polynomials. The question is now whether the difference we measure is statistically significant. If it is, then we can select the model with the best performance. If the difference is not statistically significant, it means that the difference we see is likely to be due by chance so we might base our decision on other factors like for instance the model complexity. In this case, we might decide to choose the simple model and/or the less computationally expensive model.

For example, we note that, although Gradient Boosting and Random Forests use the same number of predictors (500), gradient boosting is using tree that are limited in depth (max_depth=2) whereas the trees in the random forest are not limited. Thus, the random forest model is more complex than the one built by gradient boosting.

In [ ]:

Anomaly Detection

Anomaly (our outlier) detection deals with the finding of data points (behaviors) that are "different" from what "expected". The terms anomaly and outlier are often used interchangeably. Anomaly/outlier detection is typically an unsupervised task and thus is highly related to cluster analysis. Clustering looks for patterns in data whereas anomaly detection aims at identifying the data points that deviate from such patterns. Accordingly, anomaly detection and clustering analysis serve different, almost orthogonal, purposes.

In this notebook, we will check some of the most common anomaly detection techniques. For this purpose we first generate a working example.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

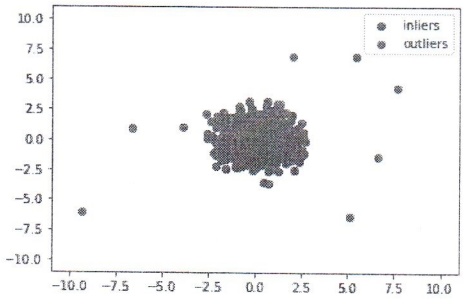
In [2]:
```
# num_outliers = 0
```

In [3]:
```
def GenerateAnomalyDetectionExample(n=1000,d=2,r=0.01,or_=10,random_state=1234):
    """n number of samples
       d number of dimensions
       r outlier ratio
       or_ outlier range
    """

    num_samples = n
    num_dimensions = d
    outlier_ratio = r

    # number of "usual" data points
    num_inliers = int(num_samples * (1-outlier_ratio))

    # number of outliers
    num_outliers = num_samples - num_inliers

    np.random.seed(random_state)
    # Generate the normally distributed inliers (mean 0, sigma 1)
    X_standard = np.random.randn(num_inliers, num_dimensions)

    # Add outliers sampled from a random uniform distribution
    X_outliers = np.random.uniform(low=-or_, high=or_, size=(num_outliers, num_dimensions))

    X = np.r_[X_standard, X_outliers]

    # Generate labels, 1 for inliers and -1 for outliers

    labels = np.ones(num_samples, dtype=int)
    labels[-num_outliers:]= (-1)

    return X, labels
```

In [4]:
```
X,labels = GenerateAnomalyDetectionExample()
```

Plotting the data

In [5]:
```
plt.scatter(X[labels==1,0], X[labels==1,1], c='b', label='inliers')
plt.scatter(X[labels==-1,0], X[labels==-1,1], c='r', label='outliers')
plt.xlim(-11,11)
plt.ylim(-11,11)
plt.legend(numpoints=1)
plt.show()
```



Single Variable Statistical Methods

The simplest way to identify anomalies is to analyze each variable separately using simple statistics like for example standard deviation. If we assume that the variables are normally distributed we can label as outlier the points lying outside 2σ or 3σ

In [6]:
```
for c in range(X.shape[1]):
    print("Variable X%d mean=%.3f sigma=%.3f"%(c,X[:,c].mean(),X[:,c].std()))
```

```
Variable X0 mean=0.037 sigma=1.101
Variable X1 mean=0.033 sigma=1.101
```

In [7]:
```
def PrintStdOutliers(X):
    for c in range(X.shape[1]):
        mean = X[:,c].mean()
        std = X[:,c].std()
        lower = mean - 3*std
        upper = mean + 3*std
        outliers = ((X[:,c]>upper) | (X[:,c]<lower))

        if (outliers.sum()>0):
            print("Variable X%d (%.3f,%.3f) has %d outliers"%(c,mean,std,outliers.sum()))
            print("\t==============================")
            for i,v in enumerate(X[outliers==True,c]):
```

```
            print("\t%d\t%.3f"%(i,v))
        print("\t================================")

    else:
        print("Variable X%d has no outliers")
```

In [8]:
```
PrintStdOutliers(X)
```

```
Variable X0 (0.037,1.101) has 7 outliers
        ================================
        0       -3.881
        1       5.072
        2       5.438
        3       6.610
        4       -6.687
        5       7.687
        6       -9.346
        ================================
Variable X1 (0.033,1.101) has 7 outliers
        ================================
        0       -3.564
        1       6.881
        2       -3.647
        3       -6.413
        4       6.916
        5       4.236
        6       -6.063
        ================================
```

Mean Absolute Deviation

The median absolute deviation (MAD) is an alternative to the standard deviation for finding outliers in one-dimensional data. MAD is defined as the median of the absolute deviations from the series median.

In [9]:
```
from scipy import stats

stats.median_abs_deviation(X, scale='normal')
```

Out[9]:
```
array([0.9727663 , 0.98924766])
```

In [10]:
```
def MAD(X,scale_factor = 1.4826):
    """This shows the actual implementation of MAD. The scale factor
       is an approximation and thus the result is slightly different."""
    mad = np.zeros(X.shape[1])
    for c in range(X.shape[1]):
        mad[c] = np.median(np.abs(X[:,c] - np.median(X[:,c])))
    return scale_factor*mad
```

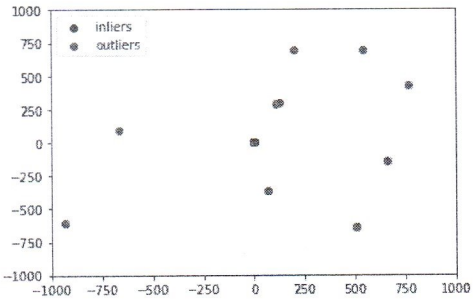
In [11]:
```
MAD(X)
```

Out[11]:
```
array([0.97276484, 0.98924618])
```

Instead of using the standard deviation values, we can use MAD values that are more robust than standard deviation with respect to the range of the outliers. Typically the lower and upper limit to define anomalies are taken by subtracting/adding 2xMAD multiplied by an adjustment coefficient to the median. For example, let's generate another dataset in which the range of the outliers is 1000.

In [12]:
```
X1,labels1 = GenerateAnomalyDetectionExample(n=1000, d=2, r=0.01, or_=1000)
```

In [13]:
```
plt.scatter(X1[labels==1,0], X1[labels==1,1], c='b', label='inliers')
plt.scatter(X1[labels==-1,0], X1[labels==-1,1], c='r', label='outliers')
plt.xlim(-1001,1001)
plt.ylim(-1001,1001)
plt.legend(numpoints=1)
plt.show()
```



In [14]:
```
def PrintAnomalyLimits(X):
    mad = stats.median_abs_deviation(X, scale='normal')
    for c in range(X.shape[1]):
        mean = X[:,c].mean()
        std = X[:,c].std()
        median = np.median(X[:,c])


        lower = mean - 3*std
        upper = mean + 3*std
        mad_lower = median - 3*mad[c]
        mad_upper = median + 3*mad[c]

        print("Variable X%d (%.3f,%.3f)"%(c,mean,std))
        print("\tStandard Deviation Limits (%.3f,%.3f)"%(lower,upper))
        print("\tMAD Limits (%.3f,%.3f)"%(mad_lower,mad_upper))
```

In [15]:
```
PrintAnomalyLimits(X)
```

```
Variable X0 (0.037,1.101)
        Standard Deviation Limits (-3.267,3.341)
        MAD Limits (-2.895,2.942)
Variable X1 (0.033,1.101)
```

```
        Standard Deviation Limits (-3.270,3.337)
        MAD Limits (-2.948,2.988)
```

In [16]:
```
PrintAnomalyLimits(X1)
```

```
Variable X0 (1.409,54.537)
        Standard Deviation Limits (-162.201,165.020)
        MAD Limits (-2.895,2.942)
Variable X1 (0.749,47.371)
        Standard Deviation Limits (-141.362,142.861)
        MAD Limits (-2.948,2.988)
```

Thus, the limits for determining outliers using MAD are less influenced by the range of outliers as it happens when using the standard deviation.

In [17]:
```
PrintStdOutliers(X1)
```

```
Variable X0 (1.409,54.537) has 7 outliers
        ================================
        0       200.117
        1       507.237
        2       543.814
        3       660.987
        4       -668.750
        5       768.742
        6       -934.630
        ================================
Variable X1 (0.749,47.371) has 9 outliers
        ================================
        0       688.116
        1       -364.721
        2       -641.337
        3       295.514
        4       691.640
        5       -143.349
        6       423.611
        7       290.468
        8       -606.319
        ================================
```

In [18]:
```
def PrintMADOutliers(X, tolerance=3):
    mad = stats.median_abs_deviation(X, scale='normal')

    # the coefficient is usually twice the mad multiplied by a tolerance factor
    for c in range(X.shape[1]):
        mean = np.mean(X[:,c])
        std = np.std(X[:,c])
        median = np.median(X[:,c])
        lower = median - tolerance*mad[c]
        upper = median + tolerance*mad[c]
        outliers = ((X[:,c]>upper) | (X[:,c]<lower))

        if (outliers.sum()>0):
            print("Variable X%d (%.3f,%.3f) has %d outliers"%(c,mean,std,outliers.sum()))
            print("\t================================")
            for i,v in enumerate(X[outliers==True,c]):
                print("\t%d\t%.3f"%(i,v))
            print("\t================================")

        else:
            print("Variable X%d has no outliers")
```

In [19]:
```
PrintMADOutliers(X1)
```

```
Variable X0 (1.409,54.537) has 11 outliers
        ================================
        0       -3.881
        1       200.117
        2       69.231
        3       507.237
        4       126.998
        5       543.814
        6       660.987
        7       -668.750
        8       768.742
        9       112.741
        10      -934.630
        ================================
Variable X1 (0.749,47.371) has 13 outliers
        ================================
        0       -3.564
        1       3.126
        2       3.110
        3       688.116
        4       -364.721
        5       -641.337
        6       295.514
        7       691.640
        8       -143.349
        9       89.549
        10      423.611
        11      290.468
        12      -606.319
        ================================
```

Multivariate Approaches - Elliptic Envelop

We can extend the previous approach to multivariate distributions. Elliptic envelope fitting can be a simple and elegant way to perform anomaly detection for normally distributed datasets. Anomalies are data points that do not conform to the expected distribution, these approaches exclude such outliers in the training data. Thus, this method is only minimally affected by the presence of anomalies in the dataset.

In [20]:
```
from sklearn.covariance import EllipticEnvelope

for outlier_ratio in [0.2, 0.1, 0.05, 0.01]:
    classifier = EllipticEnvelope(contamination=outlier_ratio)
    classifier.fit(X)
    y_pred = classifier.predict(X)
    num_errors = sum(y_pred != labels)
    print('Outlier Ratio %.3f\tNumber of errors: %d'%(outlier_ratio,num_errors))
```

```
Outlier Ratio 0.200     Number of errors: 190
```

```
Outlier Ratio 0.100     Number of errors: 90
Outlier Ratio 0.050     Number of errors: 40
Outlier Ratio 0.010     Number of errors: 4
```

One-class Support Vector Machines

One-class SVMs detect anomalies by fitting an SVM using one single target class. The data, which is assumed to contain no anomalies, is used for training a model. This comprises decision boundaries that can be used to classify future incoming data points. Note that, SVMs are sensitive to outliers thus the training data should be cleaned and contain no anomalies. This approach is generally more suitable for novelty discovery than for anomaly detection.

In [21]:
```
from sklearn import svm

for outlier_ratio in [0.2, 0.1, 0.05, 0.01]:
    classifier = svm.OneClassSVM(nu=0.99 * outlier_ratio + 0.01,kernel="rbf",gamma=0.1)
    classifier.fit(X)
    y_pred = classifier.predict(X)
    print('Outlier Ratio %.3f\tNumber of errors: %d'%(outlier_ratio,num_errors))
```

```
Outlier Ratio 0.200     Number of errors: 4
Outlier Ratio 0.100     Number of errors: 4
Outlier Ratio 0.050     Number of errors: 4
Outlier Ratio 0.010     Number of errors: 4
```

Isolation Forests

This approach identifies anomalies by computing the number of splits required to isolate a single data points; that is, how many times we need to perform splits on features in the dataset before we end up with a region that contains only the single target sample. The intuition outliers are easier to isolate with fewer splits because they usually have some feature value differences that distinguish them from the more typical data points (the inliers).

In [22]:
```
from sklearn.ensemble import IsolationForest

num_samples=1000
for outlier_ratio in [0.2, 0.1, 0.05, 0.01]:
    rng = np.random.RandomState(99)

    classifier = IsolationForest(max_samples=num_samples, contamination=outlier_ratio, random_state=rng)
    classifier.fit(X)
    y_pred = classifier.predict(X)
    num_errors = sum(y_pred != labels)
    print('Outlier Ratio %.3f\tNumber of errors: %d'%(outlier_ratio,num_errors))
```

```
Outlier Ratio 0.200     Number of errors: 190
Outlier Ratio 0.100     Number of errors: 90
Outlier Ratio 0.050     Number of errors: 40
Outlier Ratio 0.010     Number of errors: 4
```

Local Outlier Factor

This is an anomaly score that classifies anomalies based on the local density around a sample, that. is, the concentration of other points in the immediate surrounding neighborhood. The size of the neighborhood region is defined either by a fixed distance threshold or by the closest n neighboring points. LOF evaluates the isolation of a single data point with respect to its closest n neighbors. The data points with a significantly lower local density than that of their closest n neighbors are considered to be anomalies.

In [23]:
```
from sklearn.neighbors import LocalOutlierFactor

for outlier_ratio in [0.2, 0.1, 0.05, 0.01, 0.005]:
    classifier = LocalOutlierFactor(n_neighbors=100, contamination=outlier_ratio)
    y_pred = classifier.fit_predict(X)
    num_errors = sum(y_pred != labels)
    print('Outlier Ratio %.3f\tNumber of errors: %d'%(outlier_ratio,num_errors))
```

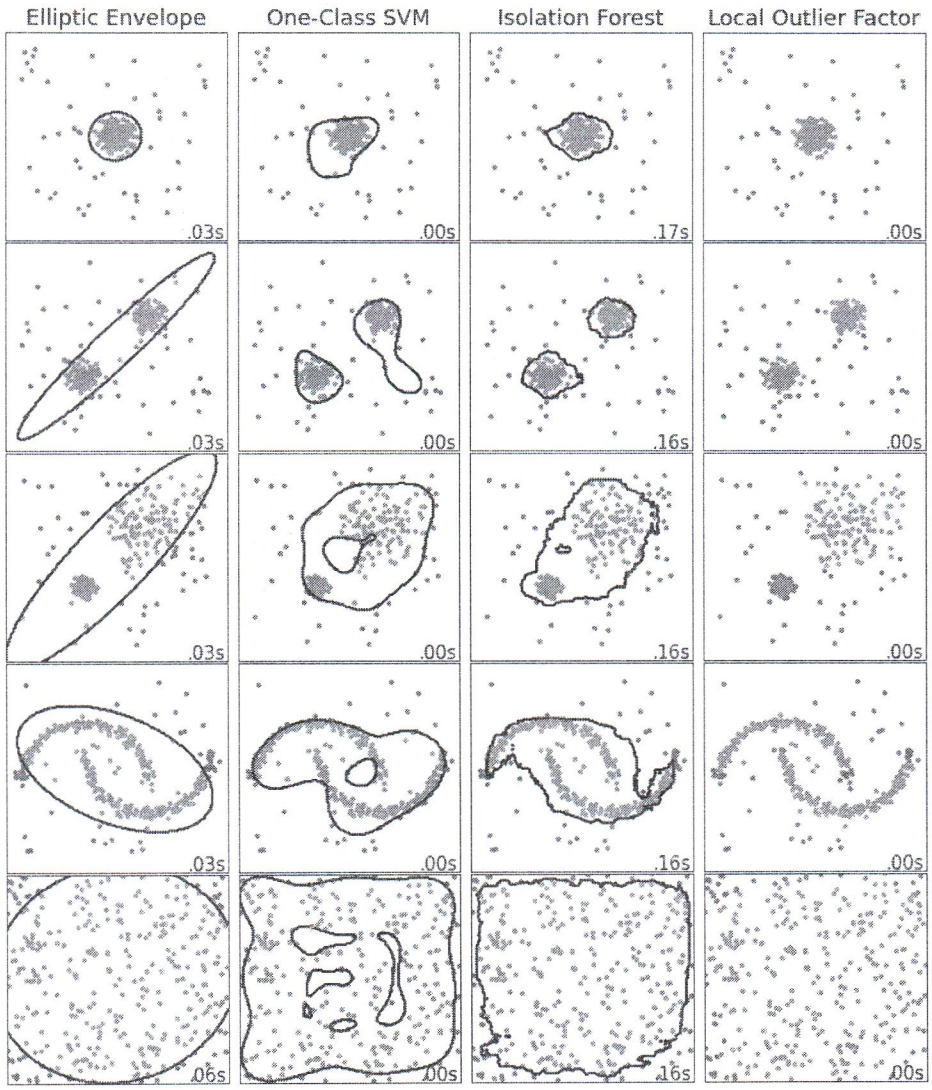
```
Outlier Ratio 0.200     Number of errors: 190
Outlier Ratio 0.100     Number of errors: 90
Outlier Ratio 0.050     Number of errors: 40
Outlier Ratio 0.010     Number of errors: 4
Outlier Ratio 0.005     Number of errors: 5
```

Comparison on Toy Datasets

In [24]:
```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Albert Thomas <albert.thomas@telecom-paristech.fr>
# License: BSD 3 clause

import time

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn import svm
from sklearn.datasets import make_moons, make_blobs
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

matplotlib.rcParams['contour.negative_linestyle'] = 'solid'

# Example settings
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(outliers_fraction * n_samples)
n_inliers = n_samples - n_outliers

# define outlier/anomaly detection methods to be compared
anomaly_algorithms = [
    ("Elliptic Envelope", EllipticEnvelope(contamination=outliers_fraction)),
```

```
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel="rbf",
                                      gamma=0.1)),
    ("Isolation Forest", IsolationForest(contamination=outliers_fraction,
                                         random_state=42)),
    ("Local Outlier Factor", LocalOutlierFactor(
        n_neighbors=35, contamination=outliers_fraction))]

# Define datasets
blobs_params = dict(random_state=0, n_samples=n_inliers, n_features=2)
datasets = [
    make_blobs(centers=[[0, 0], [0, 0]], cluster_std=0.5,
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[0.5, 0.5],
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[1.5, .3],
               **blobs_params)[0],
    4. * (make_moons(n_samples=n_samples, noise=.05, random_state=0)[0] -
          np.array([0.5, 0.25])),
    14. * (np.random.RandomState(42).rand(n_samples, 2) - 0.5)]

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 150),
                     np.linspace(-7, 7, 150))

plt.figure(figsize=(len(anomaly_algorithms) * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1
rng = np.random.RandomState(42)

for i_dataset, X in enumerate(datasets):
    # Add outliers
    X = np.concatenate([X, rng.uniform(low=-6, high=6,
                       size=(n_outliers, 2))], axis=0)

    for name, algorithm in anomaly_algorithms:
        t0 = time.time()
        algorithm.fit(X)
        t1 = time.time()
        plt.subplot(len(datasets), len(anomaly_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        # fit the data and tag outliers
        if name == "Local Outlier Factor":
            y_pred = algorithm.fit_predict(X)
        else:
            y_pred = algorithm.fit(X).predict(X)

        # plot the levels lines and the points
        if name != "Local Outlier Factor":  # LOF does not implement predict
            Z = algorithm.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='black')

        colors = np.array(['#377eb8', '#ff7f00'])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[(y_pred + 1) // 2])

        plt.xlim(-7, 7)
        plt.ylim(-7, 7)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show();
```

```
Automatically created module for IPython interactive environment
```


Elliptic Envelope
One-Class SVM
Isolation Forest
Local Outlier Factor
.03s
.00s
.17s
.00s
.03s
.00s
.16s
.00s
.03s
.00s
.16s
.00s
.03s
.00s
.16s
.00s
.06s
.00s
.16s
.00s

In [ ]:

Anscombe's Quartet

This is a classic example to demonstrate how summary statistics are now enough to have an overview of a dataset. They are four datasets which share the same summary statistics but when plotted show a completely different picture.

First, we import all the libraries we need.

In [1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
# from tabulate import tabulate

import matplotlib.pyplot as plt
%matplotlib inline
```

Let's get some information about the dataset, the variables, their values, etc.

In [2]:
```
df = sns.load_dataset('anscombe')
df.columns
```

Out[2]:
```
Index(['dataset', 'x', 'y'], dtype='object')
```

In [3]:
```
# stats about numerical variables
df.describe()
```

Out[3]:

	x	y
count	44.000000	44.000000
mean	9.000000	7.500682
std	3.198837	1.958925
min	4.000000	3.100000
25%	7.000000	6.117500
50%	8.000000	7.520000
75%	11.000000	8.747500
max	19.000000	12.740000

In [4]:
```
# stats about nominal variables
df.describe(include=['O'])
```

Out[4]:

	dataset
count	44
unique	4
top	I
freq	11

In [5]:
```
# group the data according to the dataset and then plot mean, variance
summary = df.groupby('dataset').agg([np.mean, np.var]).transpose()
summary
```

Out[5]:

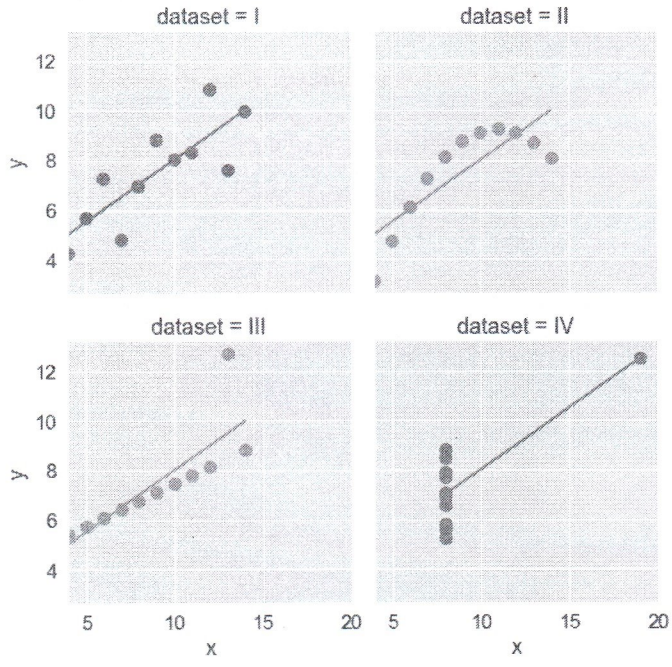
	dataset	I	II	III	IV
x	mean	9.000000	9.000000	9.00000	9.000000
	var	11.000000	11.000000	11.00000	11.000000
y	mean	7.500909	7.500909	7.50000	7.500909
	var	4.127269	4.127629	4.12262	4.123249

In [6]:
```
# and correlation
groups = df.groupby('dataset')
corr = [g.corr()['x'][1] for _, g in groups]
corr
```

Out[6]:
```
[0.81642051634484, 0.8162365060002428, 0.8162867394895981, 0.8165214368885028]
```

The four datasets are practically identical from the viewpoint of basic summary statistics, but if we check for instance the median values something start to change and if we plot them we get a completely different picture.

In [7]:
```
sns.set(font_scale=1.5)
g = sns.lmplot(x="x", y="y", col="dataset",
               hue="dataset", data=df,
               col_wrap=2, ci=None, palette="muted", height=4,
               scatter_kws={"s": 100, "alpha": 1})
g.set(xticks=np.arange(5,21,5));
```


dataset = I
dataset = II
dataset = III
dataset = IV
y
x
12
10
8
6
4
5
10
15
20

In [ ]:

Association Rule Mining

This notebooks ...

http://r-statistics.co/Association-Mining-With-R.html

First we load the R library for association rule mining. Note that there is no such thing in Python and only basic implementations are available for Python.

In [2]:
```
library(arules)
```

Load an example data set about groceries

In [3]:
```
data(Groceries)
inspect(head(Groceries, 10))
```

```
     items
[1]  {citrus fruit,
      semi-finished bread,
      margarine,
      ready soups}
[2]  {tropical fruit,
      yogurt,
      coffee}
[3]  {whole milk}
[4]  {pip fruit,
      yogurt,
      cream cheese ,
      meat spreads}
[5]  {other vegetables,
      whole milk,
      condensed milk,
      long life bakery product}
[6]  {whole milk,
      butter,
      yogurt,
      rice,
      abrasive cleaner}
[7]  {rolls/buns}
[8]  {other vegetables,
      UHT-milk,
      rolls/buns,
      bottled beer,
      liquor (appetizer)}
[9]  {pot plants}
[10] {whole milk,
      cereals}
```

Check the number of items in the top five transactions.

In [5]:
```
size(head(Groceries,10))
```

1. 4
2. 3
3. 1
4. 4
5. 4
6. 5
7. 1
8. 5
9. 1
10. 2

In [6]:
```
ap_frequent_itemsets <- apriori(Groceries, parameter = list (target="frequent itemsets", supp = 0.001, conf = 0.5, maxlen=3)) #
```

```
Apriori

Parameter specification:
 confidence minval smax arem  aval originalSupport maxtime support minlen
         NA    0.1    1 none FALSE            TRUE       5   0.001      1
 maxlen            target   ext
      3 frequent itemsets FALSE

Algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

Absolute minimum support count: 9

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [157 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3
Warning message in apriori(Groceries, parameter = list(target = "frequent itemsets", :
"Mining stopped (maxlen reached). Only patterns up to a length of 3 returned!"
 done [0.00s].
writing ... [9969 set(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

In [7]:
```
inspect(head(ap_frequent_itemsets,10))
```

```
     items                     support     count
[1]  {hair spray}              0.001118454 11
[2]  {flower soil/fertilizer}  0.001931876 19
[3]  {rubbing alcohol}         0.001016777 10
[4]  {frozen fruits}           0.001220132 12
[5]  {prosecco}                0.002033554 20
[6]  {honey}                   0.001525165 15
[7]  {cream}                   0.001321810 13
[8]  {decalcifier}             0.001525165 15
[9]  {organic products}        0.001626843 16
[10] {soap}                    0.002643620 26
```

In [9]:
```
# maxlen = 3 limits the elements in a rule to 3
rules <- apriori(Groceries, parameter = list (supp = 0.001, conf = 0.5, maxlen=3))
```

```
Apriori

Parameter specification:
 confidence minval smax arem  aval originalSupport maxtime support minlen
        0.5    0.1    1 none FALSE            TRUE       5   0.001      1
 maxlen target   ext
      3  rules FALSE

Algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

Absolute minimum support count: 9

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [157 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3
Warning message in apriori(Groceries, parameter = list(supp = 0.001, conf = 0.5, :
"Mining stopped (maxlen reached). Only patterns up to a length of 3 returned!"
 done [0.00s].
writing ... [1472 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

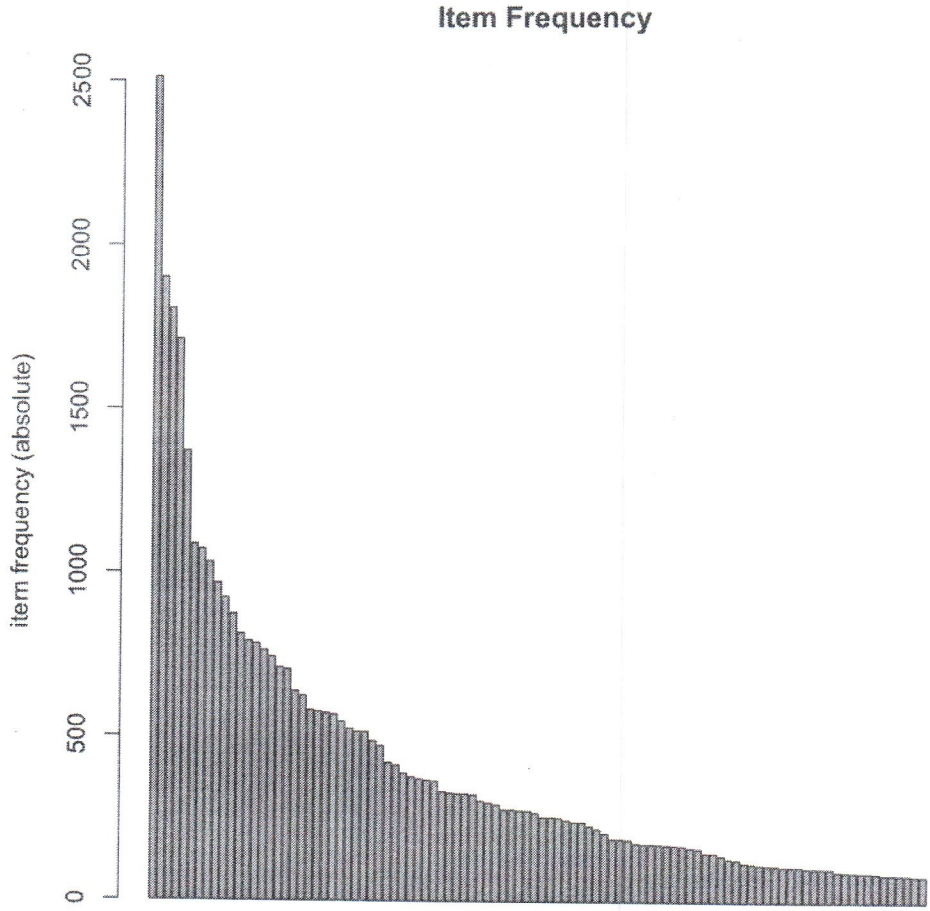
In [10]:
```
frequentItems <- eclat (Groceries, parameter = list(supp = 0.07, maxlen = 15)) # calculates support for frequent items inspect(
```

```
Eclat

parameter specification:
 tidLists support minlen maxlen            target   ext
    FALSE    0.07      1     15 frequent itemsets FALSE

algorithmic control:
 sparse sort verbose
      7   -2    TRUE

Absolute minimum support count: 688

create itemset ...
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [18 item(s)] done [0.00s].
creating sparse bit matrix ... [18 row(s), 9835 column(s)] done [0.00s].
writing  ... [19 set(s)] done [0.00s].
Creating S4 object  ... done [0.00s].
```

In [11]:
```
LIST(head(Groceries, 3))
```

1. A. 'citrus fruit'
   B. 'semi-finished bread'
   C. 'margarine'
   D. 'ready soups'
2. A. 'tropical fruit'
   B. 'yogurt'
   C. 'coffee'
3. 'whole milk'

In [12]:
```
# plot frequent items
itemFrequencyPlot(Groceries, names = FALSE, topN=100, type="absolute", main="Item Frequency")
```



In [13]:
```
# Min Support as 0.001, confidence as 0.8.
rules <- apriori (Groceries, parameter = list(supp = 0.001, conf = 0.5))

# 'high-confidence' rules.
rules_conf <- sort (rules, by="confidence", decreasing=TRUE)
inspect(head(rules_conf))
```

```
Apriori

Parameter specification:
 confidence minval smax arem  aval originalSupport maxtime support minlen
        0.5    0.1    1 none FALSE            TRUE       5   0.001      1
 maxlen target   ext
     10  rules FALSE

Algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

Absolute minimum support count: 9

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [157 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 4 5 6 done [0.01s].
writing ... [5668 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
    lhs                     rhs                   support confidence     lift count
[1] {rice,
     sugar}              => {whole milk}       0.001220132          1 3.913649    12
[2] {canned fish,
     hygiene articles}   => {whole milk}       0.001118454          1 3.913649    11
[3] {root vegetables,
     butter,
     rice}               => {whole milk}       0.001016777          1 3.913649    10
[4] {root vegetables,
     whipped/sour cream,
     flour}              => {whole milk}       0.001728521          1 3.913649    17
[5] {butter,
     soft cheese,
     domestic eggs}      => {whole milk}       0.001016777          1 3.913649    10
[6] {citrus fruit,
     root vegetables,
     soft cheese}        => {other vegetables} 0.001016777          1 5.168156    10
```

In [14]:
```
rules_lift <- sort (rules, by="lift", decreasing=TRUE) # 'high-lift' rules.
inspect(head(rules_lift)) # show the support, lift and confidence for all rules
```

```
    lhs                     rhs                   support confidence     lift count
[1] {Instant food products,
     soda}               => {hamburger meat} 0.001220132  0.6315789 18.99565    12
[2] {soda,
     popcorn}            => {salty snack}    0.001220132  0.6315789 16.69779    12
[3] {flour,
     baking powder}      => {sugar}          0.001016777  0.5555556 16.40807    10
[4] {ham,
```

```
     processed cheese}      => {white bread}    0.001931876  0.6333333 15.04549    19
[5] {whole milk,
     Instant food products} => {hamburger meat} 0.001525165  0.5000000 15.03823    15
[6] {other vegetables,
     curd,
     yogurt,
     whipped/sour cream}    => {cream cheese }  0.001016777  0.5882353 14.83409    10
```

In [15]:
```
subsetRules <- which(colSums(is.subset(rules, rules)) > 1) # get subset rules in vector
length(subsetRules)  #> 3913
rules <- rules[-subsetRules] # remove subset rules.
```

3913

This can be achieved by modifying the appearance parameter in the apriori() function. For example,

To find what factors influenced purchase of product X

To find out what customers had purchased before buying 'Whole Milk'. This will help you understand the patterns that led to the purchase of 'whole milk'.

In [16]:
```
rules <- apriori (data=Groceries, parameter=list (supp=0.001,conf = 0.08), appearance = list (default="lhs",rhs="whole milk"),
rules_conf <- sort (rules, by="confidence", decreasing=TRUE) # 'high-confidence' rules.
inspect(head(rules_conf))
```

```
    lhs                     rhs              support confidence     lift count
[1] {rice,
     sugar}              => {whole milk} 0.001220132          1 3.913649    12
[2] {canned fish,
     hygiene articles}   => {whole milk} 0.001118454          1 3.913649    11
[3] {root vegetables,
     butter,
     rice}               => {whole milk} 0.001016777          1 3.913649    10
[4] {root vegetables,
     whipped/sour cream,
     flour}              => {whole milk} 0.001728521          1 3.913649    17
[5] {butter,
     soft cheese,
     domestic eggs}      => {whole milk} 0.001016777          1 3.913649    10
[6] {pip fruit,
     butter,
     hygiene articles}   => {whole milk} 0.001016777          1 3.913649    10
```

In [17]:
```
rules <- apriori (data=Groceries, parameter=list (supp=0.001,conf = 0.15,minlen=2), appearance = list(default="rhs",lhs="whole
rules_conf <- sort (rules, by="confidence", decreasing=TRUE) # 'high-confidence' rules.
inspect(head(rules_conf))
```

```
    lhs             rhs                support    confidence lift      count
[1] {whole milk} => {other vegetables} 0.07483477 0.2928770  1.5136341 736
[2] {whole milk} => {rolls/buns}       0.05663447 0.2216474  1.2050318 557
[3] {whole milk} => {yogurt}           0.05602440 0.2192598  1.5717351 551
[4] {whole milk} => {root vegetables}  0.04890696 0.1914047  1.7560310 481
[5] {whole milk} => {tropical fruit}   0.04229792 0.1655392  1.5775950 416
[6] {whole milk} => {soda}             0.04006101 0.1567847  0.8991124 394
```

Bagging Regressor

This is a very simple implementation of bagging for regression that can be easily modified for classification. We test it using the Boston Housing dataset.

In [102]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree
from sklearn.metrics import r2_score
from sklearn.ensemble import BaggingRegressor
import sklearn
```

In [2]:
```
# set the font for plotting and the figure size
font = {'size': 14};
plt.rc('font', **font);
plt.figure(figsize=(8,6));
```

```
<Figure size 576x432 with 0 Axes>
```

In [3]:
```
from sklearn.datasets import load_boston
boston = load_boston()
X = boston.data
y = boston.target
```

In [4]:
```
def BaggingFit(X,y,base_model,no_models):
    """Fits a bagging regressor using X and y.

        base_model: is the model used
        no_models: is the number of models in the ensemble

    """

    # number of examples and number of variables
    m,n = X.shape

    # array of models, input and output datasets
    models = [ None ] * no_models
    inputs = [ None ] * no_models
    targets = [ None ] * no_models

    for i in range(no_models):

        # generate the array of bootstrapped examples
        ind = np.floor( m * np.random.rand(m) ).astype(int)

        # generate the datasets
        Xi, yi = X[ind,:] , y[ind]

        inputs[i] = Xi
        targets[i] = yi

        # create a copy of the base model
        model = sklearn.base.clone(base_model)

        # fit the classifier
        model.fit(Xi, yi)

        # memorize the model for later prediction
        models[i] = model

    # return the array of models
    return models, inputs, targets
```

The code above is written to be readable and also to return the single dataset used to build every model which might be useful if we wish to do an out-of-bag evaluation.

In [95]:
```
def BaggingFitSimplified(X,y,base_model,no_models):
    """Fits a bagging regressor using X and y.

        base_model: is the model used
        no_models: is the number of models in the ensemble

        Simplified version that does not store the datasets used
    """

    # number of examples and number of variables
    m,n = X.shape

    # array of models, input and output datasets
    models = [ None ] * no_models

    for i in range(no_models):

        # generate the array of bootstrapped examples
        ind = np.floor( m * np.random.rand(m) ).astype(int)

        # create a copy of the base model
        model = sklearn.base.clone(base_model)

        # fit the classifier
        model.fit(X[ind,:] , y[ind])

        # memorize the model for later prediction
        models[i] = model

    # return the array of models
    return models
```

In [96]:
```
def BaggingPredict(models,X,use_average=False):
```

```
    no_models = len(models)
    m = X.shape[0]
    predict = np.zeros( (m, no_models) )
    for i in range(no_models):
        predict[:,i] = models[i].predict(X)

    if use_average:
        predict = np.mean(predict, axis=1)
    else:
        # Make overall prediction by majority vote
        predict = np.mean(predict, axis=1) > 0 # if +1 vs -1

    return predict
```

First, let's evaluate our implementation. We use the more compact one shown in the slides.

In [97]:
```
models = BaggingFitSimplified(X,y,DecisionTreeRegressor(max_depth=2),10)
```

In [98]:
```
yp = BaggingPredict(models,X,use_average=True)
```

In [100...
```
print("Performance of Our Bagging Regressor Mean=%.3f"%r2_score(boston.target,yp))
```

```
Performance of Our Bagging Regressor Mean=0.753
```

Next, let's evaluate a single regressor of unlimited depth.

In [101...
```
xval_score = cross_val_score(DecisionTreeRegressor(), X, y,cv=KFold(10,shuffle=True,random_state=1234))
print("Single model performance mean=%.3f std=%.3f"%(np.mean(xval_score),np.std(xval_score)))
```

```
Single model performance mean=0.670 std=0.176
```

Finally, let's use the bagging regressor available in Scikit-learn

In [103...
```
bagging = BaggingRegressor(DecisionTreeRegressor(max_depth=2),n_estimators=10,random_state=1234)
```

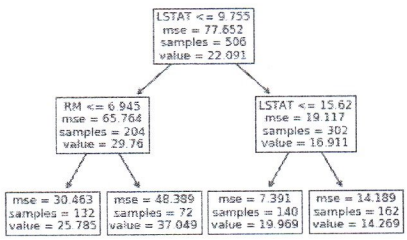
In [105...
```
xval_score = cross_val_score(bagging, X, y,cv=KFold(10,shuffle=True,random_state=1234))
print("Bagging regressor performance mean=%.3f std=%.3f"%(np.mean(xval_score),np.std(xval_score)))
```

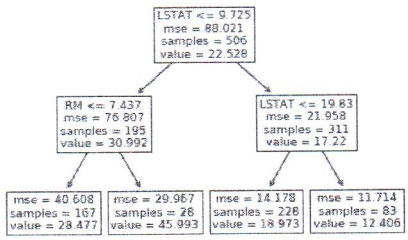
```
Bagging regressor performance mean=0.717 std=0.082
```

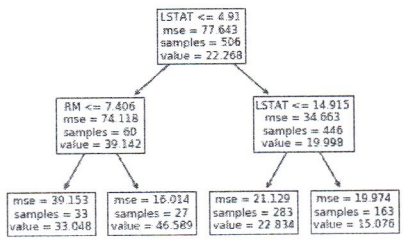
Note that we did not use crossvalidation with our implementation since it does not follow the fit-predict pattern required by Scikit-learn and thus we would have to implement the procedure ourselves. Also note that bagging results in a much lower standard deviation when compared to a single classifier.

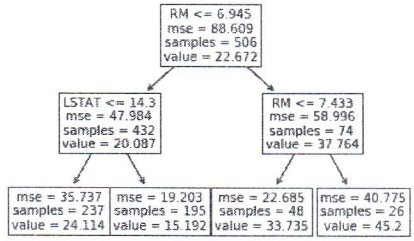
As a curiosity we can plot all the models computed by our implementation.

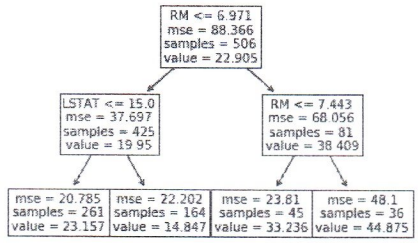
In [106...
```
for i in range(len(models)):
    plot_tree(models[i],feature_names=boston.feature_names);
    plt.show()
```

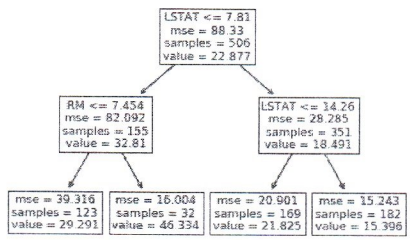


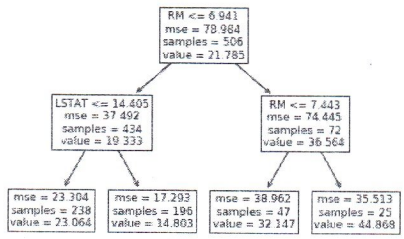


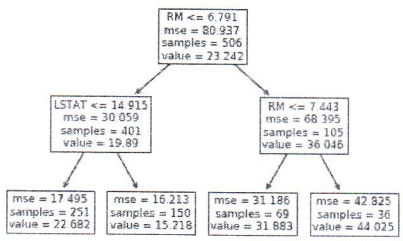


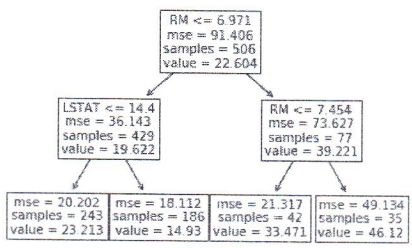

RM <= 6.945
mse = 88.609
samples = 506
value = 22.672
LSTAT <= 14.3
mse = 47.984
samples = 432
value = 20.087
RM <= 7.433
mse = 58.996
samples = 74
value = 37.764
mse = 35.737
samples = 237
value = 24.114
mse = 19.203
samples = 195
value = 15.192
mse = 22.685
samples = 48
value = 33.735
mse = 40.775
samples = 26
value = 45.2

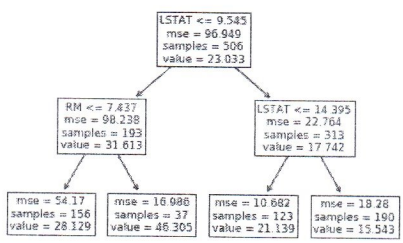

RM <= 6.971
mse = 88.366
samples = 506
value = 22.905
LSTAT <= 15.0
mse = 37.697
samples = 425
value = 19.95
RM <= 7.443
mse = 68.056
samples = 81
value = 38.409
mse = 20.785
samples = 261
value = 23.157
mse = 22.202
samples = 164
value = 14.847
mse = 23.81
samples = 45
value = 33.236
mse = 48.1
samples = 36
value = 44.875


LSTAT <= 7.81
mse = 88.33
samples = 506
value = 22.877
RM <= 7.454
mse = 82.092
samples = 155
value = 32.81
LSTAT <= 14.26
mse = 28.285
samples = 351
value = 18.491
mse = 39.316
samples = 123
value = 29.291
mse = 16.004
samples = 32
value = 46.334
mse = 20.901
samples = 169
value = 21.825
mse = 15.243
samples = 182
value = 15.396


RM <= 6.941
mse = 78.984
samples = 506
value = 21.785
LSTAT <= 14.405
mse = 37.492
samples = 434
value = 19.333
RM <= 7.443
mse = 74.445
samples = 72
value = 36.564
mse = 23.304
samples = 238
value = 23.064
mse = 17.293
samples = 196
value = 14.803
mse = 38.962
samples = 47
value = 32.147
mse = 35.513
samples = 25
value = 44.868


RM <= 6.791
mse = 80.937
samples = 506
value = 23.242
LSTAT <= 14.915
mse = 30.059
samples = 401
value = 19.89
RM <= 7.443
mse = 68.395
samples = 105
value = 36.046
mse = 17.495
samples = 251
value = 22.682
mse = 16.213
samples = 150
value = 15.218
mse = 31.186
samples = 69
value = 31.883
mse = 42.825
samples = 36
value = 44.025


RM <= 6.971
mse = 91.406
samples = 506
value = 22.604
LSTAT <= 14.4
mse = 36.143
samples = 429
value = 19.622
RM <= 7.454
mse = 73.627
samples = 77
value = 39.221
mse = 20.202
samples = 243
value = 23.213
mse = 18.112
samples = 186
value = 14.93
mse = 21.317
samples = 42
value = 33.471
mse = 49.134
samples = 35
value = 46.12


LSTAT <= 9.545
mse = 96.949
samples = 506
value = 23.033
RM <= 7.437
mse = 98.238
samples = 193
value = 31.613
LSTAT <= 14.395
mse = 22.764
samples = 313
value = 17.742
mse = 54.17
samples = 156
value = 28.129
mse = 16.986
samples = 37
value = 46.305
mse = 10.682
samples = 123
value = 21.139
mse = 18.28
samples = 190
value = 15.543

Bootstrap Resampling

Bootstrap resamples a dataset of N instances with replacement to generate a new dataset of the same size. An instance has a probability of $1-1/N$ of not being picked. Thus its probability of ending up in the test data is:

$$(1 - 1/N)^N \approx e^{-1} = 0.368$$

This means the training data will contain approximately 63.2% of the instances.

Let's define a function to generate a dataset applying resampling with replacement from an original dataset.

In [1]:
```
import numpy as np
import pandas as pd
import random
from sklearn.datasets import load_boston
```

In [2]:
```
def bootstrap(dataset, ratio=1.0):

    # compute the number of rows of the generated dataset
    n_rows = int(dataset.shape[0]*ratio)

    # compute the number of columns
    n_cols = dataset.shape[1]

    # create the output dataset with all zero
    sampled_dataset = np.zeros((n_rows,n_cols))

    # randomly select a row from the original dataset and then copy it to the output dataset
    for s in range(n_rows):
        sample_index = int(random.random()*n_rows)
        sampled_dataset[s,:] = dataset[sample_index,:]

    return sampled_dataset
```

Now let's apply bootstrap to the boston housing dataset.

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

We can apply bootstrapping and generate a new dataset,

In [6]:
```
np.random.seed(328489)
bootstrap_dataset = bootstrap(data)
print("Bootstrap dataset contains %d (%.1f) unique examples"%
      (np.unique(bootstrap_dataset,axis=0).shape[0],
       100*np.unique(bootstrap_dataset,axis=0).shape[0]/data.shape[0]))
```

```
Bootstrap dataset contains 319 (63.0) unique examples
```

We can repeat the process several times and compute on average how many unique examples a data set generated with bootstrap resampling contains.

In [7]:
```
sum = 0.0
i = 0
for i in range(500):
    sum = sum + np.unique(bootstrap(data),axis=0).shape[0]/506
    i = i+1
print("The percentage of unique data points is %.1f"%(100*(sum/i)))
```

```
The percentage of unique data points is 63.3
```

This is the value we expected from the theory.

BOOTSTRAP RESAMPLING EVALUATION

Classifier Evaluation using Bootstrap

Bootstrap can also be used to evaluate a classifier. Given the original data D, k datasets D_i are generated from D and used to train a model M_i that is evaluated using the entire dataset D returning the evaluation θ_i. The overall evaluation is computed as the mean and standard deviation of the k values of θ_i. Note however that the estimates will be somewhat optimistic becvause of the overlap between the training (generated by bootstrap resampling) and the testing performed on the entire dataset (63.2%). Cross-validation does not suffer from this limitation since it keeps the training and testing sets disjoint.

In [4]:
```
import numpy as np
import pandas as pd
import random

np.random.seed(238476293)
```

In [5]:
```
def bootstrap(X, y, ratio=1.0):

    # compute the number of rows of the generated dataset
    n_rows = int(X.shape[0]*ratio)

    # compute the number of columns
    n_cols = X.shape[1]

    # create the output dataset with all zero
    sampled_X = np.zeros((n_rows,n_cols))
    sampled_y = np.zeros(n_rows)

    # randomly select a row from the original dataset and then copy it to the output dataset
    for s in range(n_rows):
        sample_index = int(random.random()*n_rows)
        sampled_X[s,:] = X[sample_index,:]
        sampled_y[s] = y[sample_index]

    return sampled_X, sampled_y
```

In [7]:
```
def bootstrap_evaluation(classifier, X, y, k, metrics):
    evaluation = []
    for i in range(k):
        bX,by = bootstrap(X,y)
        classifier.fit(bX,by)
        yp = classifier.predict(X)
        evaluation.append(metrics(y,yp))
    return evaluation
```

Boston Housing Data

First, let apply bootstrap evaluation using a regression problem, the boston housing dataset.

In [8]:
```
from sklearn.datasets import load_boston
boston = load_boston()
X = boston.data
y = boston.target
```

In [9]:
```
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression

evaluation = bootstrap_evaluation(LinearRegression(), X, y, 100, r2_score)
```

In [10]:
```
print("Bootstrap Evaluation %.3f +/ %.3f"%(np.array(evaluation).mean(),np.array(evaluation).std()))
```

```
Bootstrap Evaluation 0.731 +/ 0.005
```

In [11]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
xval_evaluation = cross_val_score(LinearRegression(), X, y, cv=KFold(n_splits=10, random_state=1234, shuffle=True))
```

In [12]:
```
print("Crossvalidation Evaluation %.3f +/ %.3f"%(np.array(xval_evaluation).mean(),np.array(xval_evaluation).std()))
```

```
Crossvalidation Evaluation 0.682 +/ 0.125
```

Iris Data

Next, we can apply the same approach to a classification problem.

In [13]:
```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

iris = load_iris()
X = iris.data
y = iris.target

classifier = LogisticRegression(max_iter=1000)

evaluation = bootstrap_evaluation(classifier, X, y, 100, accuracy_score)
print("Bootstrap Evaluation %.3f +/ %.3f"%(np.array(evaluation).mean(),np.array(evaluation).std()))

xval_evaluation = cross_val_score(classifier, X, y, cv=KFold(n_splits=10, random_state=1234, shuffle=True))
print("Crossvalidation Evaluation %.3f +/ %.3f"%(np.array(xval_evaluation).mean(),np.array(xval_evaluation).std()))
```

```
Bootstrap Evaluation 0.971 +/ 0.010
Crossvalidation Evaluation 0.967 +/ 0.045
```

In [ ]:

Boston Housing

We are going to

In [3]:
```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, ElasticNet, Lasso, LassoCV

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

%matplotlib inline
```

In [2]:
```
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 24
4-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference
of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [3]:
```
df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['MEDV'] = boston.target
```

In [4]:
```
df.describe()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

In [1]:
```
random_state=1234
```

In [4]:
```
crossvalidation = KFold(10, shuffle=True, random_state=random_state)
```

In [ ]:
```
def r2_cv(model, X_train, y, random_state=12345678):
    score = cross_val_score(model, X_train, y, scoring="r2", cv =KFold(10, shuffle=True, random_state=random_state))
```

```
    return(r2)
```

Linear Regresssion

In [ ]:
```
lr = LinearRegression(normalize=False,fit_intercept=True)
score = cross_val_score(lr, )
```

In [ ]:
```
alphas = [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 50, 75]
cv_r2_ridge = [r2_cv(Ridge(alpha = alpha), X_train, y).mean() for alpha in alphas]
```

Given a dataset, the name of a column and a degree it generates all the additional columns needed to perform a polynomial regression

We define a helper function to plot the original data against the data generated by a model

In [5]:
```
def plot_scatter(x,y,xp,yp,title=""):
    """Plots the original data (x,y) and a set of point (xp,yp) showing the model approximation"""
    font = {'family' : 'sans',
        'size'   : 14}
    plt.rc('font', **font)

    plt.scatter(x, y,  color='blue')
    plt.plot(xp, yp, color='red', linewidth=3)
    plt.xlabel("LSTAT")
    plt.ylabel("MEDV")

    if (title!=""):
        plt.title(title)

    plt.xlim([0,40])
    plt.ylim([0,60])
    plt.show()
```

Simple Linear Regression

We start with a very basic model using one input variable x and fits the data using the model $y = w_0 + w_1 x$. For example, we can use variable LSTAT (percentage of lower status of the population) as input to predict the target variable MEDV (median value of owner occupied homes in $1000s) as target.

First, we create the matrix X of inputs and the target vector y.

In [6]:
```
X = df['LSTAT'].values.reshape(-1,1)
y = df['MEDV']
```

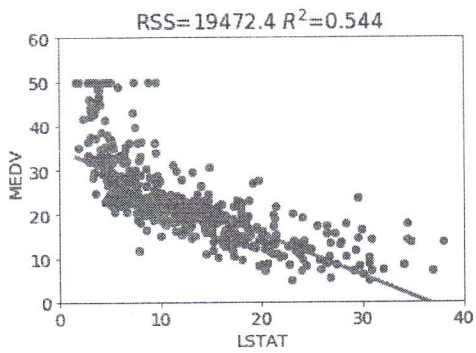
Next, we create a linear regressor and fit it with the input/output data.

In [7]:
```
linear_regressor = linear_model.LinearRegression()
X.reshape(-1,1)
linear_regressor.fit(X, y);
```

We can now compute the R^2 score and RSS by first computing the predicted output yp.

In [8]:
```
yp = linear_regressor.predict(X)

r2 = r2_score(y,yp)

rss = sum((yp-y)*(yp-y))
```

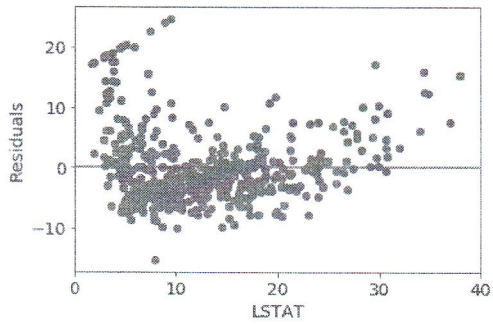
In [10]:
```
title = "RSS=%.1f $R^2$=%.3f"%(rss,r2)
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
yplot = linear_regressor.predict(Xplot)
plot_scatter(X,y,Xplot,yplot,title)
```



RSS and R^2 provide an overall evaluation of our model. It is more interesting to explore how our model makes mistakes by analyzing the residuals, that is the difference between the target values and the fitted values. First we compute the residuals and then we plot them.

In [11]:
```
residuals = y-yp
```

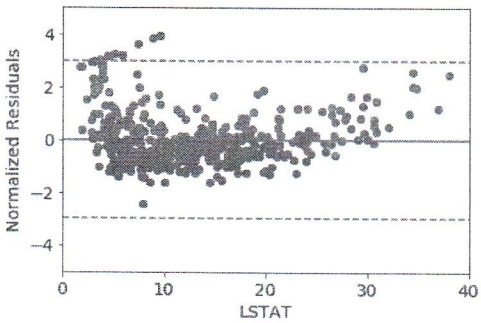
In [12]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.scatter(X, residuals,  color='blue')
plt.plot([0,40],[0,0],'-',c="red")
plt.xlabel("LSTAT")
plt.ylabel("Residuals")
plt.xlim([0,40])
plt.show()
```



The points are not all randomly scattered, showing different variances at different values of the input variable. We can also spot some patterns (for examples, points in a straight line). The plot using z-score normalized residuals is more informative since the values -3 and 3 represents the range containing the 99.7% of the values

In [13]:
```
from sklearn.preprocessing import StandardScaler
normalized_residuals = StandardScaler().fit_transform(residuals.values.reshape(-1,1))
```

In [14]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.scatter(X, normalized_residuals,  color='blue')
plt.plot([0,40],[0,0],'-',c="red")
plt.plot([0,40],[-3,-3],'--',c="red")
plt.plot([0,40],[3,3],'--',c="red")
plt.xlabel("LSTAT")
plt.ylabel("Normalized Residuals")
plt.xlim([0,40])
plt.ylim([-5,5])
plt.show()
```



Multiple Linear Regression

We can build a model to predict MEDV using more input variables from the ones available in the data. However, in this example to be able to plot the model using more input variables, we consider polynomial models based on the same input variable LSTAT.

In [16]:
```
from sklearn.preprocessing import PolynomialFeatures
```

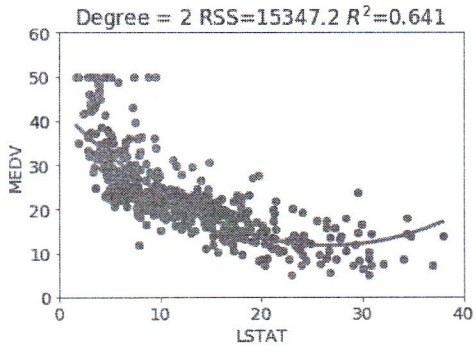
Let' start by computing a model using a second degree polynomial. Next, we use linear regression to fit the new input data and the target variable.

In [17]:
```
polynomial2 = PolynomialFeatures(degree=2, include_bias=False)
X2 = polynomial2.fit_transform(X)
```

In [18]:
```
linear_regressor2 = linear_model.LinearRegression()
X2.reshape(-1,2)
linear_regressor2.fit(X2, y);
yp2 = linear_regressor2.predict(X2)
```

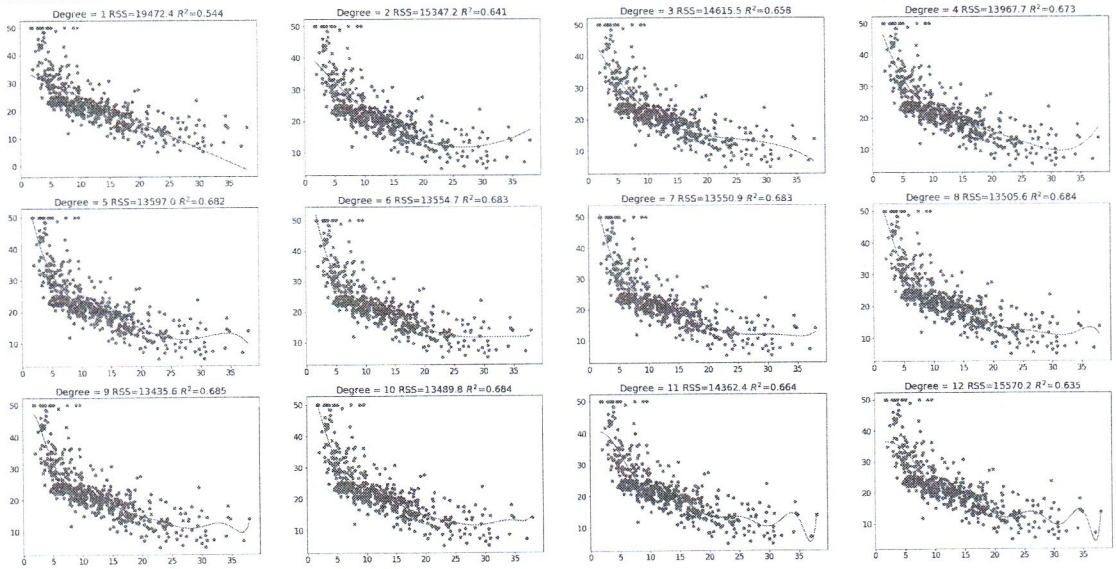
Let's evaluate the model and plot it.

In [20]:
```
r2_p2 = r2_score(y,yp2)
rss_p2 = sum((yp2-y)*(yp2-y))

# let's create the input data to plot the model
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
Xplot2 = polynomial2.fit_transform(Xplot)

# compute the model on the plot data
yplot2 = linear_regressor2.predict(Xplot2)

plot_scatter(X[:,0],y,Xplot2[:,0],yplot2,"Degree = 2 RSS=%.1f $R^2$=%.3f"%(rss_p2,r2_p2))
```



We can increase the degree of the polynomial and check whether and how much the model improves.

In [22]:
```
max_polynomial = 12
f, axarr = plt.subplots(3, 4)

plt.rcParams['figure.figsize'] = (40.0, 20.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
        X_polynomial = X


    linear_regressor = linear_model.LinearRegression()
    X_polynomial.reshape(-1,degree)
    linear_regressor.fit(X_polynomial, y);

    yp = linear_regressor.predict(X_polynomial)
    yplot = linear_regressor.predict(Xplot_polynomial)

    r2 = r2_score(y,yp)
    rss = sum((yp-y)*(yp-y))

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d RSS=%.1f $R^2$=%.3f"%(degree,rss,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



As can be noted the RSS decreases as the degree of the polynomial increases. Only for a degree of 12 we note an increase of RSS and a decrease of R^2. However, this is not a robust evaluation since we are scoring the models on the same data we used to build the model. To get a robust evaluation we need to either split the data into train set and test set, build the model on the train set, and evaluate the model on the test set. Or we can apply crossvalidation.

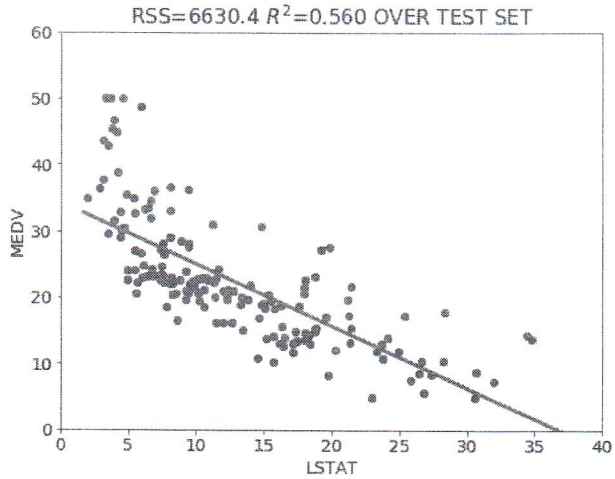
Model Evaluation Using a Test Set

To evaluate the model we now split the data between train and test using 2/3 of the data for training and 1/3 for testing. We start with the simple linear regression we done at the beginning. Note that to be able to replicate the results we are setting a random seed.

In [23]:
```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.33, random_state=1234)
```

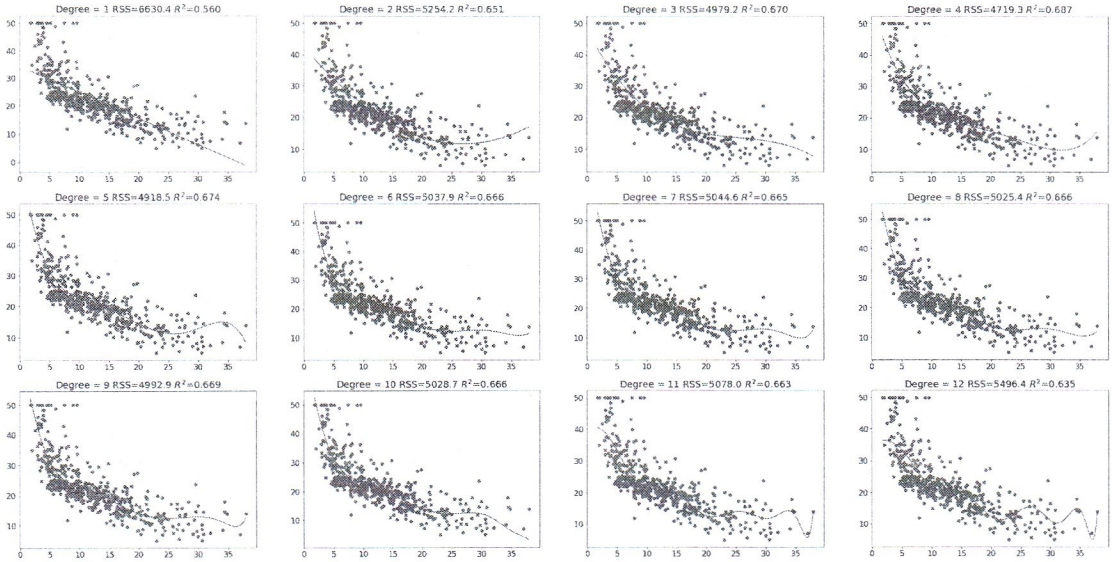
In [25]:
```
# build the model on the train
linear_regressor.fit(X_train, y_train);
```

```
# evaluate the model on the test
yp = linear_regressor.predict(X_test)

r2 = r2_score(y_test,yp)

rss = sum((yp-y_test)*(yp-y_test))

title = "RSS=%.1f $R^2$=%.3f OVER TEST SET"%(rss,r2)
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
yplot = linear_regressor.predict(Xplot)
plt.rcParams['figure.figsize'] = (8.0, 6.0)
plot_scatter(X_test,y_test,Xplot,yplot,title)
```



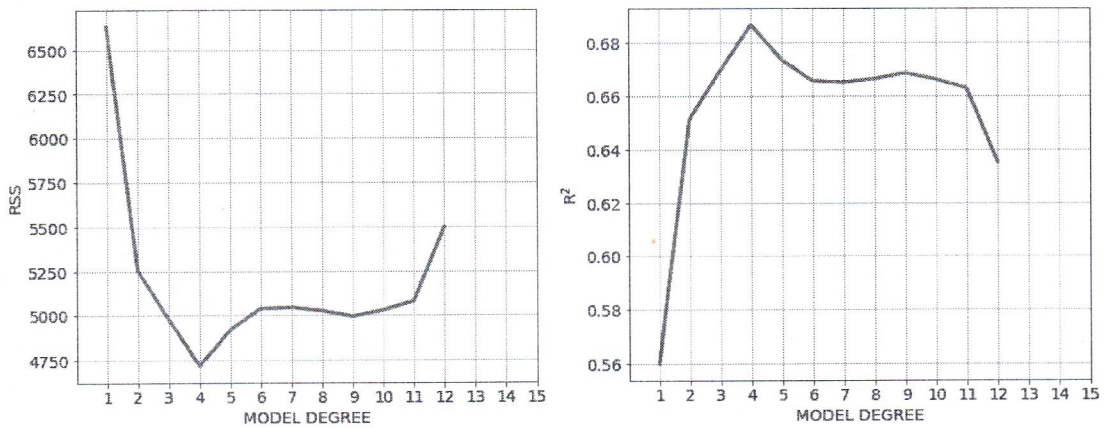
Note that RSS is lower since it is computed over a smaller data set. R^2 is slightly better. Let's see what happens with higher polynomials.

In [27]:
```
max_polynomial = 12
f, axarr = plt.subplots(3, 4)

plt.rcParams['figure.figsize'] = (40.0, 20.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)

rss_train_values = []
r2_train_values = []


rss_values = []
r2_values = []

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
        X_polynomial = X


    X_train, X_test, y_train, y_test = model_selection.train_test_split(X_polynomial, y, test_size=0.33, random_state=1234)

    linear_regressor = linear_model.LinearRegression()
    X_polynomial.reshape(-1,degree)
    linear_regressor.fit(X_train, y_train);

    yp_train = linear_regressor.predict(X_train)
    r2_train = r2_score(y_train,yp_train)
    rss_train = ((yp_train-y_train)**2).sum()
    rss_train_values.append(rss_train)
    r2_train_values.append(r2_train)

    yp = linear_regressor.predict(X_test)
    yplot = linear_regressor.predict(Xplot_polynomial)

    r2 = r2_score(y_test,yp)
    rss = ((yp-y_test)**2).sum()

    rss_values.append(rss)
    r2_values.append(r2)

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d RSS=%.1f $R^2$=%.3f"%(degree,rss,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



We can plot how RSS and R^2 changes based on the degree of the polynomial approximation.

In [28]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("RSS")
axarr[0].set_xlim([0,13])
axarr[0].set_xticks(range(1,16))
axarr[0].grid()
axarr[0].plot(range(1,max_polynomial+1), rss_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("R$^2$")
axarr[1].set_xlim([0,13])
axarr[1].set_xticks(range(1,16))
axarr[1].grid()
axarr[1].plot(range(1,max_polynomial+1), r2_values, color="blue",linewidth=3);
```



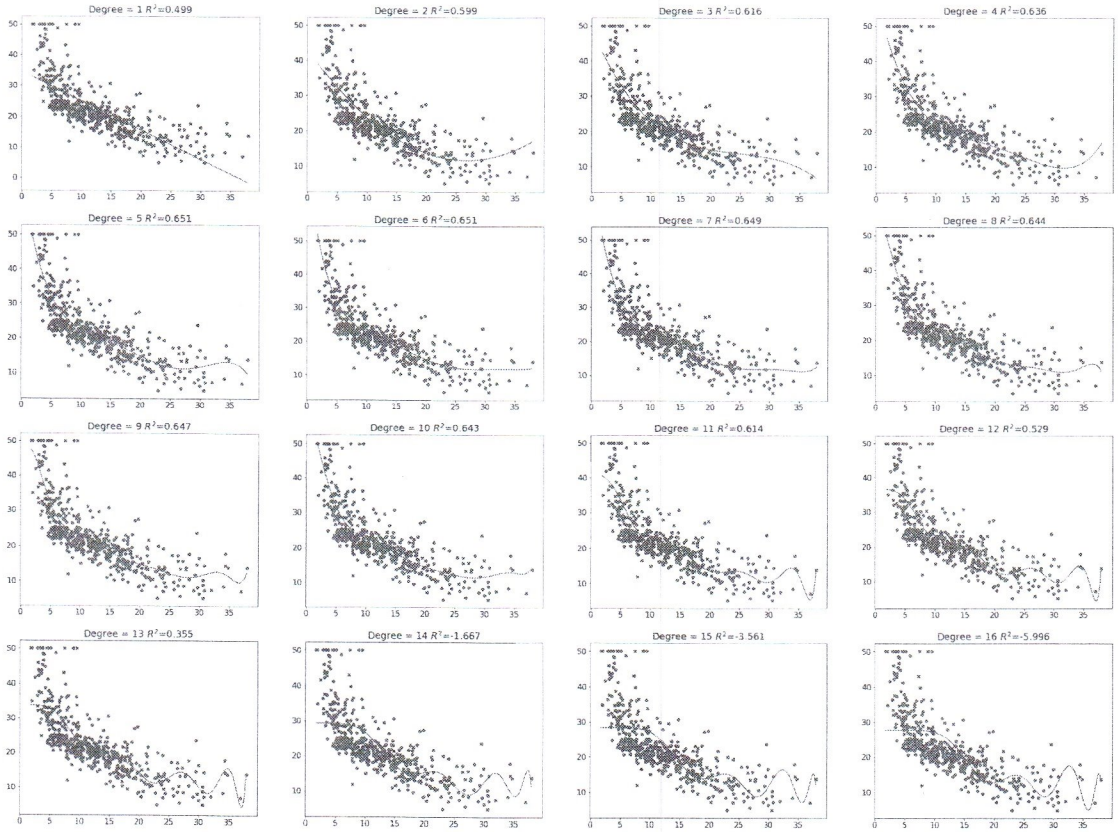
Note that as the degree of the polynomial increases until a value of 4, the RSS decreases and similarly R^2 inscreases. Then the RSS starts increasing again, the model is starting to overfit the data. The results appear to suggest that a polynomial of degree 4 is the best choice, but are we sure? Our evaluation is based on one run performed over a specific train/test partition. To have a more robust evaluation we can either repeat the procedure with different train/test partitions or otherwise use crossvalidation.

Model Evaluation using Crossvalidation

We now repeat the procedure but evaluate the model using crossvalidation.

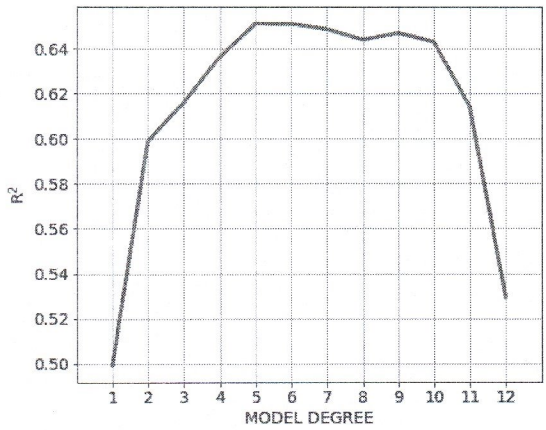
In [32]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

max_polynomial = 16
f, axarr = plt.subplots(4, 4)

plt.rcParams['figure.figsize'] = (40.0, 30.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)

rss_values = []
r2_values = []

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
```

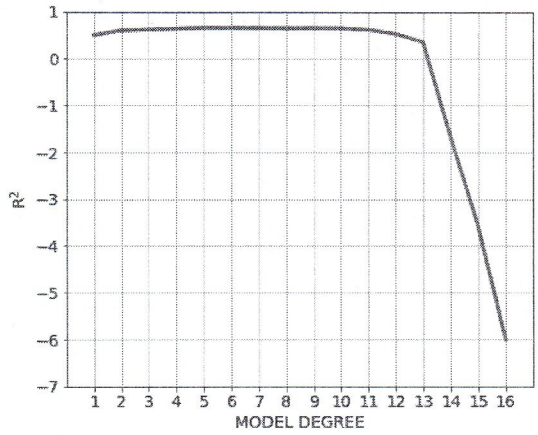
```
        X_polynomial = X


    linear_regressor = linear_model.LinearRegression()

    X_polynomial.reshape(-1,degree)

    # score returns the scores for each single run
    score = cross_val_score(linear_regressor, X_polynomial, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

    r2 = score.mean()
    r2_values.append(r2)

    # let's build the final model (using **all the data**)
    linear_regressor.fit(X_polynomial,y)
    yplot = linear_regressor.predict(Xplot_polynomial)

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d $R^2$=%.3f"%(degree,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



As you can note the evaluation given by crossvalidation is less "optimistic" about the performance of our models, in fact the values of R^2 are generally lower. Also if we plot R^2 as a function of the degree (below) we have a much different curve that suggests a degree of 5 or 6 as the best choice --- a quite different result with respect to the previous basic holdout evaluation. Note that, for higher degree polynomials R^2 becomes negative, this means that the model performs worst that the basic mean. The model tries very hard to fit the training data so to become useless to predict unseen cases.

In [34]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,13])
axarr[1].set_ylim([0,1])
axarr[0].set_xticks(range(1,13))
axarr[0].grid()
axarr[0].plot(range(1,13), r2_values[0:12], color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("R$^2$")
axarr[1].set_xlim([0,17])
axarr[1].set_xticks(range(1,17))
axarr[1].set_ylim([-7,1])
axarr[1].grid()
axarr[1].plot(range(1,max_polynomial+1), r2_values, color="blue",linewidth=3);
```





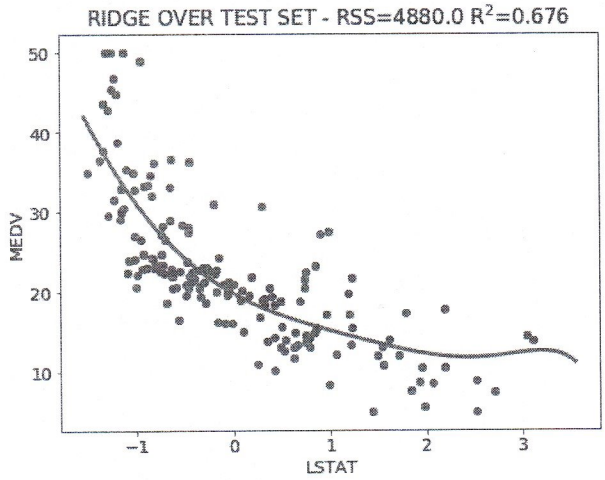
Regularization

To avoid overfitting, we can introduce Lasso (L_1) and Ridge (L_2) regularization which will take care of variables that might cause overfit. Let's consider the version of the dataset extended by adding powers of LSTAT up to the 10th degree. Also, so far we never normalized the variables which we should do since the

In [91]:
```
polynomial = PolynomialFeatures(degree=10, include_bias=False)
X_polynomial = polynomial.fit_transform(X)

scaler = StandardScaler()

scaler.fit(X_polynomial)

X_normalized = scaler.transform(X_polynomial)

Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
Xplot_polynomial = polynomial.fit_transform(Xplot)

Xplot_normalized = scaler.transform(Xplot_polynomial)

# X_normalized = StandardScaler().fit_transform(X_polynomial)
# Xplot = np.arange(np.min(X_normalized[:,0]),np.max(X_normalized[:,0]),0.1).reshape(-1,1)
```

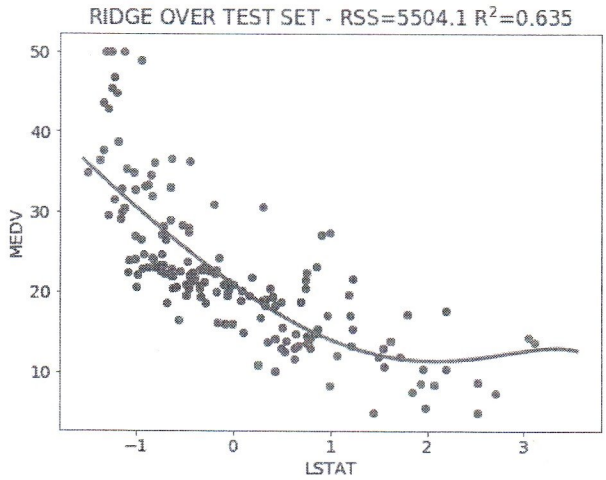
First, we apply the process using Ridge regression that define cost as, RSS($\vec{w}$) + $\alpha||\vec{w}||_2^2$ with an α of 0.1 (or λ)

In [92]:
```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.1,max_iter=1000,random_state=1234)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X_normalized, y, test_size=0.33, random_state=1234)

ridge_model = ridge.fit(X_train,y_train)

yp = ridge.predict(X_test)
yp_plot = ridge.predict(Xplot_normalized)

r2 = r2_score(y_test,yp)
rss = ((yp-y_test)**2).sum()

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_test[:,0], y_test,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("RIDGE OVER TEST SET - RSS=%.1f R$^2$=%.3f"%(rss,r2))
plt.show()
```

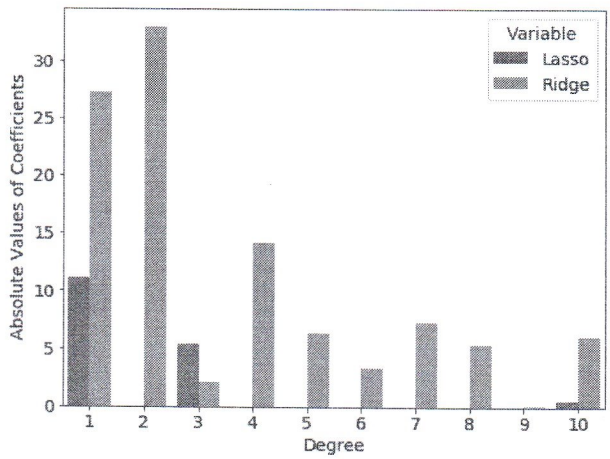


In [84]:
```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1,max_iter=1000,random_state=1234)
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X_normalized, y, test_size=0.33, random_state=1234)

lasso_model = lasso.fit(X_train,y_train)

yp = lasso.predict(X_test)
yp_plot = lasso.predict(Xplot_normalized)

r2 = r2_score(y_test,yp)
rss = ((yp-y_test)**2).sum()

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_test[:,0], y_test,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("LASSO OVER TEST SET - RSS=%.1f R$^2$=%.3f"%(rss,r2))
plt.show()
```



In [108...
```
import seaborn
coefficients = pd.DataFrame({'Degree':[x for x in range(1,len(lasso_model.coef_)+1)], 'Lasso':np.abs(lasso_model.coef_), 'Ridge
tidy = coefficients.melt(id_vars='Degree').rename(columns=str.title)
seaborn.barplot(x='Degree', y='Value', hue='Variable', data=tidy)
plt.ylabel("Absolute Values of Coefficients");
```

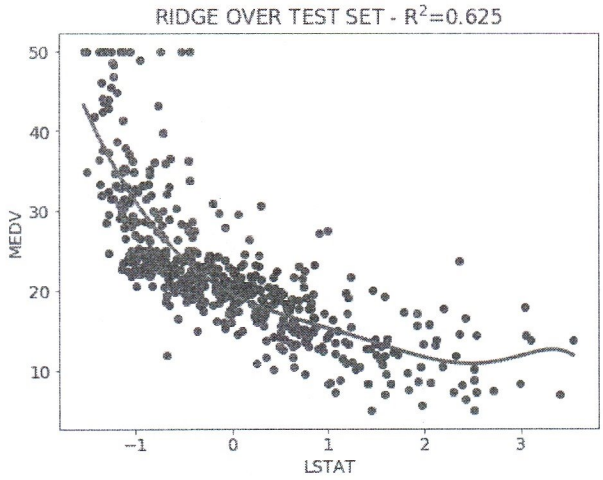


Note that Lasso creates a model with a lower R^2 score but only three variables with non zero weights.

Regularization Evaluation using Crossvalidation

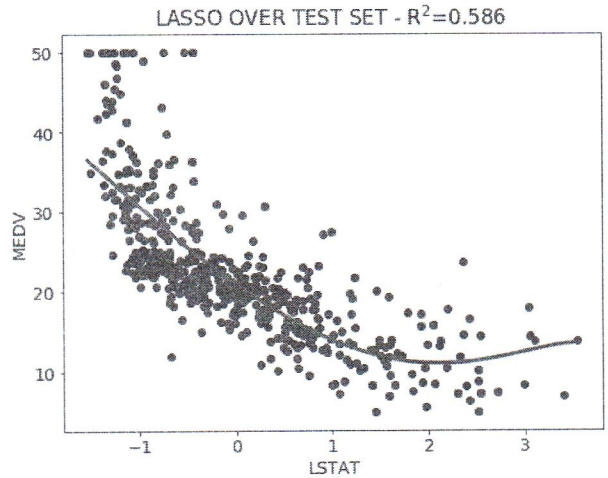
We can repeat the analysis using cross-validation

In [113...
```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.1,max_iter=1000,random_state=1234)

score = cross_val_score(ridge, X_normalized, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

r2 = score.mean()

yp = ridge.fit(X_normalized,y)
yp_plot = ridge.predict(Xplot_normalized)

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_normalized[:,0], y,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("RIDGE OVER TEST SET - R$^2$=%.3f"%(r2))
plt.show()
```



In [114...

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1,max_iter=1000,random_state=1234)

score = cross_val_score(lasso, X_normalized, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

r2 = score.mean()

yp = lasso.fit(X_normalized,y)
yp_plot = lasso.predict(Xplot_normalized)

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_normalized[:,0], y,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("LASSO OVER TEST SET - R$^2$=%.3f"%(r2))
plt.show()
```



As before, the results from the cross-validation are less optimistic. Note that, when using cross-validation the final model is produced using the entire dataset.

CLASSIFIER ENSAMBLE ERROR

Why Do Ensembles Work?

Suppose there are 25 base classifiers and each classifier has error rate, ε = 0.35. Assume classifiers are independent. The probability that the ensemble makes a wrong prediction is

$$\sum_{i=13}^{25} \binom{25}{i} \epsilon^i (1-\epsilon)^{25-i} = 0.06$$

In [1]:
```
import math
```

In [2]:
```
def nCr(n,k):
    """Compute n choose k"""
    return math.factorial(n) / math.factorial(k) / math.factorial(n-k)
```

In [3]:
```
def EnsembleError(n,e):
    """Compute the overall error of an ensemble with n classifiers with error e"""
    majority = int(n/2)+1
    error = 0
    for i in range(majority,n+1):
        error = error + nCr(n,i)*(e**i)*(1-e)**(n-i)
    return error
```

In [4]:
```
EnsembleError(25,0.35)
```

Out[4]:
```
0.06044491356702048
```

In [5]:
```
EnsembleError(40,0.6)
```

Out[5]:
```
0.8702342941780972
```

In [6]:
```
EnsembleError(5,0.45)
```

Out[6]:
```
0.4068731250000001
```

In [7]:
```
EnsembleError(5,0.55)
```

Out[7]:
```
0.593126875
```

In [ ]:
```

```

DBSCAN CHAMALEON

Density-Based Clustering

We apply density-based cluestering to known datasets used for clustering, Chamaleon and Birch3, available at https://cs.joensuu.fi/sipu/datasets/

First, we load the libraries we need.

In [2]:
```
import numpy as np
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
from sklearn.datasets import make_circles
from sklearn.datasets import make_moons

import matplotlib.pyplot as plt
%matplotlib inline
```

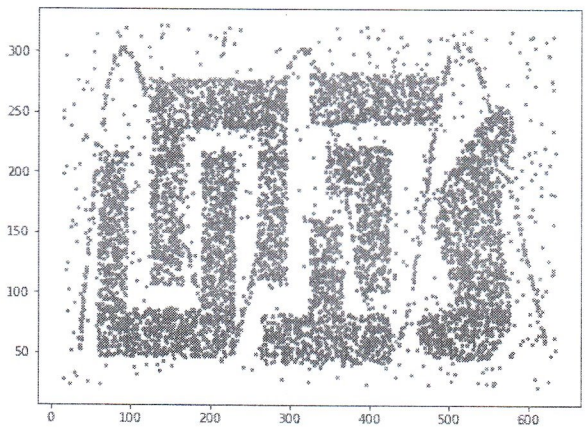
Next we define some palettes.

In [3]:
```
from matplotlib.colors import ListedColormap
dots_cmap = ListedColormap(['#e20c32','#1b80e8']) #,,])'#599062'
plt.register_cmap(cmap=dots_cmap)
colors = ['#1b80e8','#599062','#e20c32']
```

We read the Chamaleon dataset and plot it.

In [4]:
```
df = pd.read_csv('Chamaleon.txt',sep=" ")
```

In [5]:
```
plt.figure(figsize=(8,6))
plt.scatter(df['x'],df['y'],s=5.0,c=colors[0])
plt.savefig("Chamaleon.png")
```



We apply DBSCAN

In [6]:
```
db = DBSCAN(eps=10, min_samples=10).fit(df.values)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
```

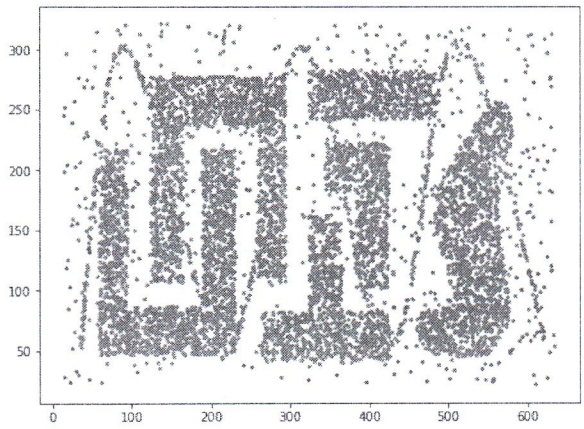
The function labels as -1 the noisy points all the other values are cluster indexes. So let's print some statistics.

In [7]:
```
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
noisy_points = db.labels_==-1
cluster_points = ~noisy_points
```

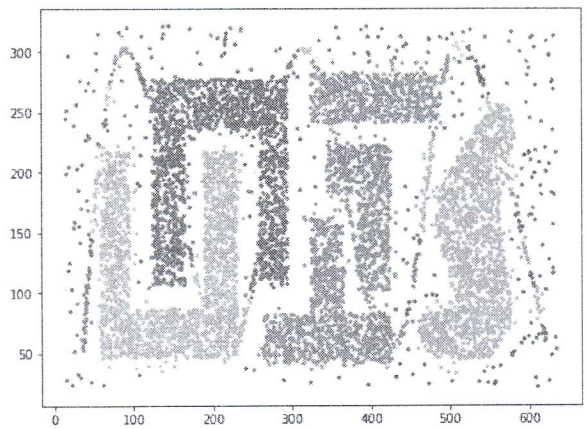
In [8]:
```
print("Number of clusters = %d"%n_clusters_)
print("Number of cluster points = %d"%sum(cluster_points))
print("Number of noisy points = %d"%sum(noisy_points))
```

```
Number of clusters = 15
Number of cluster points = 7722
Number of noisy points = 278
```

In [9]:
```
plt.figure(figsize=(8,6))
plt.scatter(df['x'],df['y'],s=5.0,c=cluster_points,cmap=dots_cmap)
plt.savefig("ChamaleonClusteredPoints.png")
```



In [10]:
```
plt.figure(figsize=(8,6))
plt.scatter(df['x'],df['y'],s=5.0,c=labels,cmap=plt.get_cmap('tab20'))
plt.savefig("ChamaleonClusters.png")
```



With an epsilon of 10 there are seveal clusters with few points. What would happen if we increase the epsilon? Or the number of minimum number points that define core objects?

DECISION TREE - BOUNDARIES

Plotting the Decision Boundaries of Decision Trees

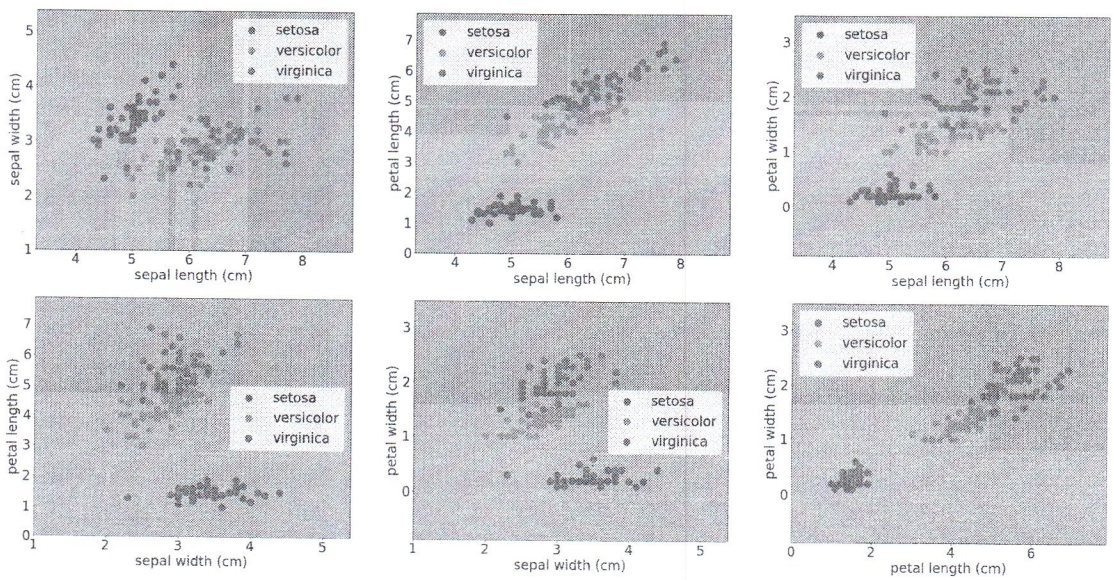
Each internal node of a decision tree test the value of an attribute and thus split the problem space into separate areas. Leaves assign a label to areas of the problem space. Let's explore this concept using the Iris data set using only two features at a time so that we can plot the decision boundaries on a two dimensional plane.

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

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
%matplotlib inline
```

Now for every pair of variables, we generate a decision tree and then we plot the decision surface.

In [2]:
```
# number of classes in the dataset
n_classes = 3
plot_step = 0.02

### Color maps for plotting
data_points_cm = matplotlib.colors.ListedColormap([plt.cm.Paired.colors[1],plt.cm.Paired.colors[3],plt.cm.Paired.colors[5]], na
ds_points_cm = matplotlib.colors.ListedColormap([plt.cm.Paired.colors[0],plt.cm.Paired.colors[2],plt.cm.Paired.colors[4]], name

# Load data
iris = load_iris()

plot_colors = "bry"

plt.rcParams['figure.figsize'] = (40.0, 20.0)
font = {'family' : 'sans', 'size'   : 28}
plt.rc('font', **font)

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
                                [1, 2], [1, 3], [2, 3]]):
    # We only take the two corresponding features
    X = iris.data[:, pair]
    y = iris.target

    # Train
    clf = DecisionTreeClassifier().fit(X, y)

    # Plot the decision boundary
    plt.subplot(2, 3, pairidx + 1)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=ds_points_cm)

    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])
    plt.axis("tight")

    # Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], label=iris.target_names[i], s=200, cmap=data_points_cm)

    plt.axis("tight")
    plt.legend()


plt.suptitle("Decision surface of a decision tree using paired features")
plt.show()
```

Decision surface of a decision tree using paired features



Decision Tree for the Weather Dataset

We are going to compute the decision tree for the weather dataset. The Scikit-learn cannot deal with categorical attributes thus we need to transform the weather dataset into a numerical representation. Previously, we used the One-Hot-Encoding. In this session we are going to compare the result produced by the one-hot-encoding with the basic numerical transformation of categorical attributs into numerical attributes.

First, we import all the libraries we need.

In [1]:
```
# general-purpose libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Scikit-Learn
import sklearn
from sklearn import linear_model
from sklearn import model_selection
from sklearn.model_selection import KFold
from sklearn import preprocessing
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_curve,roc_auc_score,precision_score,recall_score,auc


# libraries needed to plot decision trees
import io
import pydot
from scipy import misc
%matplotlib inline
```

We also define a function we will use to export decision trees as images

In [2]:
```
def visualize_decision_tree(decision_tree, feature_names, image):
    """Create tree png using graphviz.

    Args
    ----
    decision_tree -- scikit-learn DecsisionTree.
    feature_names -- list of feature names.
    image -- file containing the final image
    """

    dotfile = io.StringIO()
    sklearn.tree.export_graphviz(decision_tree, feature_names=feature_names, out_file=dotfile, filled=True, rounded=True, speci
    pydot.graph_from_dot_data(dotfile.getvalue())[0].write_png(image)
```

Next, we load the dataset.

In [3]:
```
weather = pd.read_csv('Weather.csv')
weather.head(5)
```

Out[3]:

	Outlook	Temperature	Humidity	Windy	Play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes

In [4]:
```
target = 'Play'
variables = weather.columns[weather.columns!=target]
```

We define the class attribute by encoding the Play attribute as a +1/-1 attribute

In [5]:
```
weather['Class'] = weather[target].apply(lambda x: 1 if x=='yes' else -1);
weather = weather.drop(target,axis=1);
numerical_target = 'Class'
```

We now map the categorical/nominal variables into 0/1 variables using one-hot-encoding approach

In [6]:
```
weather_binary = pd.get_dummies(weather, columns=variables)
weather_binary.head()
```

Out[6]:

	Class	Outlook_overcast	Outlook_rainy	Outlook_sunny	Temperature_cool	Temperature_hot	Temperature_mild	Humidity_high	Humidity_normal	W
0	-1	0	0	1	0	1	0	1	0	
1	-1	0	0	1	0	1	0	1	0	
2	1	1	0	0	0	1	0	1	0	
3	1	0	1	0	0	0	1	1	0	
4	1	0	1	0	1	0	0	0	1	

In [7]:
```
binary_variables = weather_binary.columns[weather_binary.columns!=numerical_target]
```

We repeat the same process but this time we are going to simply map categorical values into numbers using the tools provided by the Scikit-learn library.

In [8]:
```
weather_dict = {}
numerical_variables = weather.columns[weather.columns!=target]

# keep all the label encoders used
label_encoders = {}
```

```
for v in numerical_variables:
    label_encoders[v] = preprocessing.LabelEncoder()
    label_encoders[v].fit(weather[v])
    weather_dict[v] = label_encoders[v].transform(weather[v])

weather_numerical = pd.DataFrame(weather_dict)
weather_numerical['Class'] = weather['Class']

weather_numerical.head(5)
```

Out[8]:

	Class	Humidity	Outlook	Temperature	Windy
0	-1	0	2	1	0
1	-1	0	2	1	1
2	1	0	0	1	0
3	1	0	1	2	0
4	1	1	1	0	0

In [9]:
```
x_bin = weather_binary[binary_variables]
x_num = weather_numerical[numerical_variables]
```

Now we can apply a decision tree classifier to the two versions of the dataset using entropy as the splitting criteria.

In [10]:
```
y = weather_binary['Class']

dt_bin = dt = tree.DecisionTreeClassifier('entropy')
dt_bin = dt_bin.fit(x_bin, y)
xval_bin = model_selection.cross_val_score(dt_bin, x_bin, y, cv=StratifiedKFold(n_splits=10, shuffle=True, random_state=1234))

dt_num = tree.DecisionTreeClassifier('entropy')
dt_num = dt_num.fit(x_num, y)
xval_num = model_selection.cross_val_score(dt_num, x_num, y, cv=StratifiedKFold(n_splits=10, shuffle=True, random_state=1234))

print("Entropy - One-Hot-Encoding   Avg Accuract=%3.2f +/- %3.2f"%(np.average(xval_bin),np.std(xval_bin)))
print("Entropy - Numerical Encoding Avg Accuract=%3.2f +/- %3.2f"%(np.average(xval_num),np.std(xval_num)))
```

```
Entropy - One-Hot-Encoding   Avg Accuract=0.55 +/- 0.42
Entropy - Numerical Encoding Avg Accuract=1.00 +/- 0.00
```

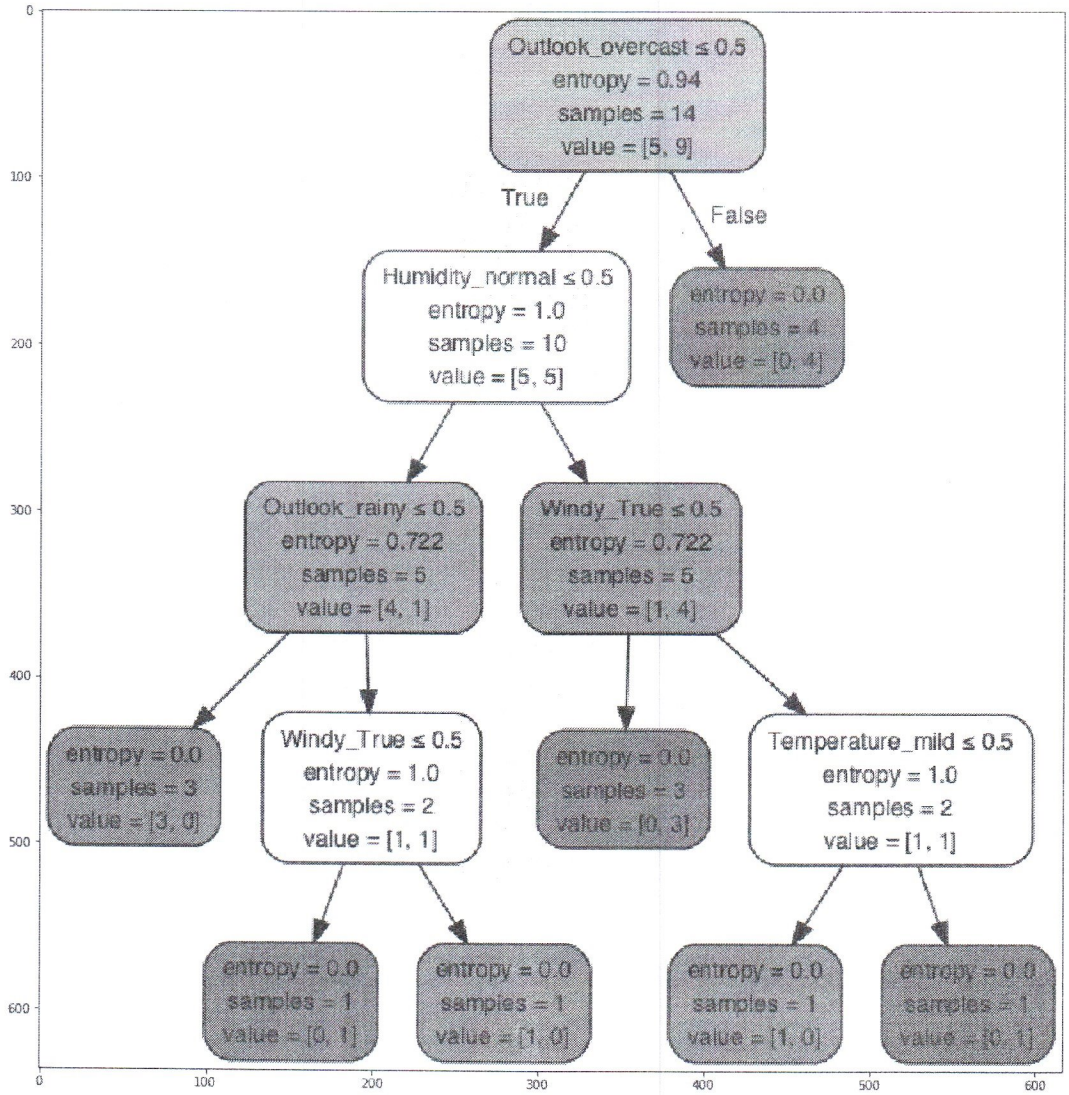
Question

- Are these performances satisfactory?
- What do the high values of standard deviation suggest?
- What might cause them?

Let's plot the decision tree for the one-hot-encoding and for the numerical encoding.

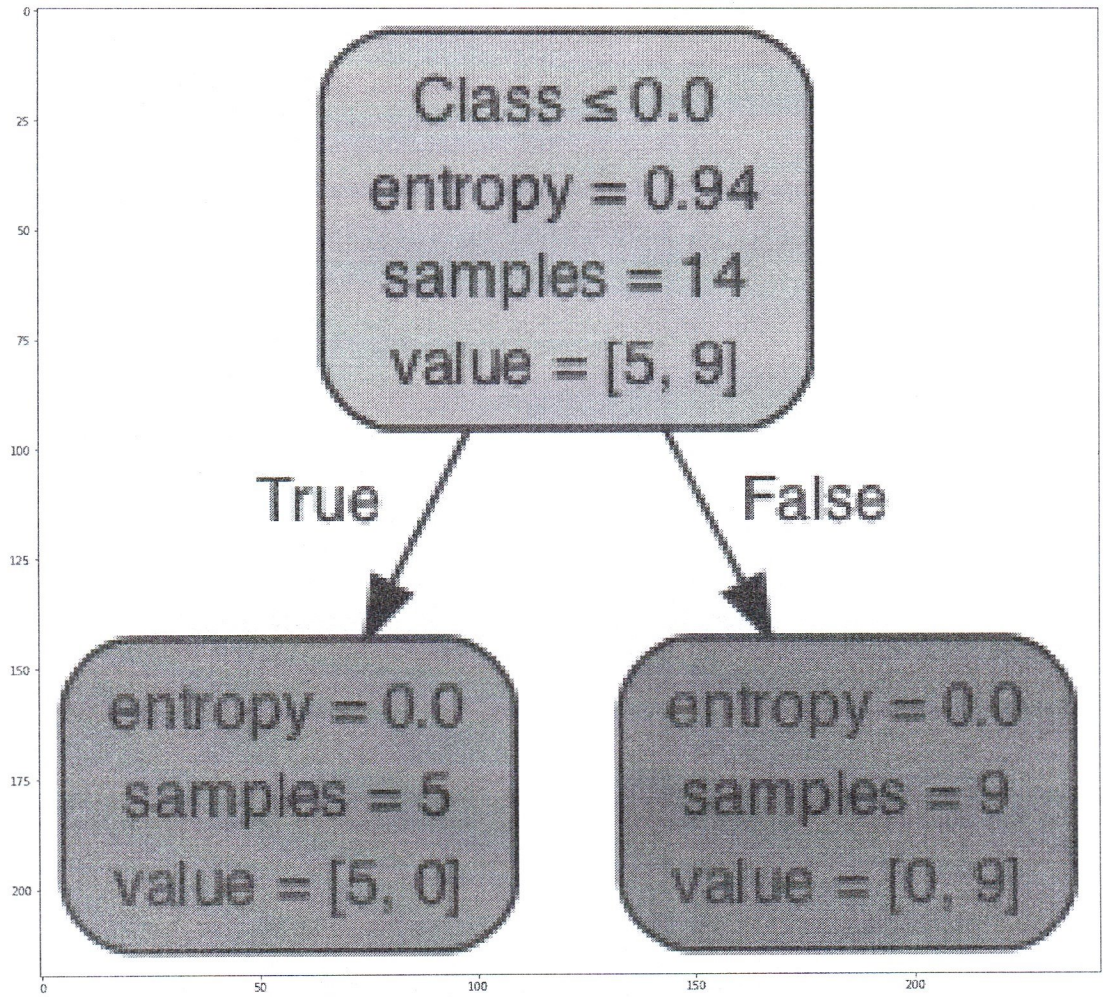
In [11]:
```
visualize_decision_tree(dt_bin, x_bin.columns, 'weather_num_gini.png')
i = misc.imread('weather_num_gini.png')
plt.figure(1, figsize=(24, 16))
plt.imshow(i);
```

```
/Users/pierluca/anaconda/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.py:2: DeprecationWarning: `imread` is depr
ecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```



In [12]:
```
visualize_decision_tree(dt_num, x_num.columns, 'weather_num_gini.png')
i = misc.imread('weather_num_gini.png')
plt.figure(1, figsize=(24, 16))
plt.imshow(i);
```

```
/Users/pierluca/anaconda/envs/python3/lib/python3.6/site-packages/ipykernel_launcher.py:2: DeprecationWarning: `imread` is depr
ecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```



Finally, let's compute the decision trees using the same datasets and the gini index (it is actually the default option).

In [13]:
```
x_bin = weather_binary[binary_variables]
x_num = weather_numerical[numerical_variables]

y = weather_binary[numerical_target]

dt_bin_gini = tree.DecisionTreeClassifier('gini')
dt_bin_gini = dt_bin_gini.fit(x_bin, y)
xval_bin_gini = model_selection.cross_val_score(dt_bin_gini, x_bin, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))


dt_num_gini = tree.DecisionTreeClassifier('gini')
dt_num_gini = dt_num_gini.fit(x_num, y)
xval_num_gini = model_selection.cross_val_score(dt_num_gini, x_num, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

print("Gini - One-Hot-Encoding   Avg Accuract=%3.2f +/- %3.2f"%(np.average(xval_bin_gini),np.std(xval_bin_gini)))
print("Gini - Numerical Encoding Avg Accuract=%3.2f +/- %3.2f"%(np.average(xval_num_gini),np.std(xval_num_gini)))
```

```
Gini - One-Hot-Encoding   Avg Accuract=0.60 +/- 0.37
Gini - Numerical Encoding Avg Accuract=1.00 +/- 0.00
```

Ensembles classifiers using trees on the iris dataset

In this notebook we apply several ensemble methods to the Iris dataset using tree classifiers and plot the resulting decision surfaces. Note that this notebook has been created using the material from http://scikit-learn.org/stable/modules/ensemble.html

First we load all the required libraries.

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

from sklearn import clone
from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                              AdaBoostClassifier,BaggingClassifier)
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.externals.six.moves import xrange
from sklearn.tree import DecisionTreeClassifier
%matplotlib inline
```

Next we define some of the parameters required to run the experiments like the number of estimators used in each ensembles and the random seed to be able to reproduce the results.

In [2]:
```
# Number of estimators used in each ensemble
n_estimators = 30

# set the random seed to be able to repeat the experiment
random_seed = 1234
```

Load the dataset

In [3]:
```
iris = load_iris()
```

Set the models to be compared

- simple decision tree
- bagging
- random forest
- extra tree classifiers
- adaboost

In [4]:
```
models = {'Decision Tree':DecisionTreeClassifier(max_depth=None),
          'Bagging':BaggingClassifier(DecisionTreeClassifier(max_depth=3),n_estimators=n_estimators),
          'Random Forest':RandomForestClassifier(n_estimators=n_estimators),
          'Extremely Randomized Trees':ExtraTreesClassifier(n_estimators=n_estimators),
          'Ada Boost':AdaBoostClassifier(DecisionTreeClassifier(max_depth=3),
                                         n_estimators=n_estimators)}
```

For each model, we apply 10-fold stratified crossvalidation and compute the average accuracy and the corresponding standard deviation

In [5]:
```
scores = {}
for pair in ([0, 1], [0, 2], [2, 3]):
    for model_name in models:
        # We only take the two corresponding features
        X = iris.data[:, pair]
        y = iris.target

        clf = models[model_name];
        score = cross_val_score(clf,X,y,cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=random_seed))
        scores[(model_name,str(pair))]=(np.average(score),np.std(score))
```

Then, we print for every variable pair the performance of all the models

In [6]:
```
for pair in ([0, 1], [0, 2], [2, 3]):
    print('Iris - Variables: ',iris.feature_names[pair[0]],' & ',iris.feature_names[pair[1]])
    for model_name in models:
        print('\t%26s\t%.3f +/- %.3f'%(model_name,scores[(model_name,str(pair))][0],scores[(model_name,str(pair))][1]))
    print('\n')
```

```
Iris - Variables:  sepal length (cm)  &  sepal width (cm)
                   Decision Tree      0.667 +/- 0.089
                         Bagging      0.787 +/- 0.088
                   Random Forest      0.713 +/- 0.095
       Extremely Randomized Trees     0.747 +/- 0.111
                        Ada Boost     0.753 +/- 0.079


Iris - Variables:  sepal length (cm)  &  petal length (cm)
                   Decision Tree      0.927 +/- 0.055
                         Bagging      0.940 +/- 0.070
                   Random Forest      0.940 +/- 0.047
       Extremely Randomized Trees     0.927 +/- 0.055
                        Ada Boost     0.933 +/- 0.060


Iris - Variables:  petal length (cm)  &  petal width (cm)
                   Decision Tree      0.947 +/- 0.050
                         Bagging      0.967 +/- 0.045
                   Random Forest      0.940 +/- 0.047
       Extremely Randomized Trees     0.947 +/- 0.050
                        Ada Boost     0.947 +/- 0.050
```

Finally, we plot the decision surfaces for every model and every attribute combination.

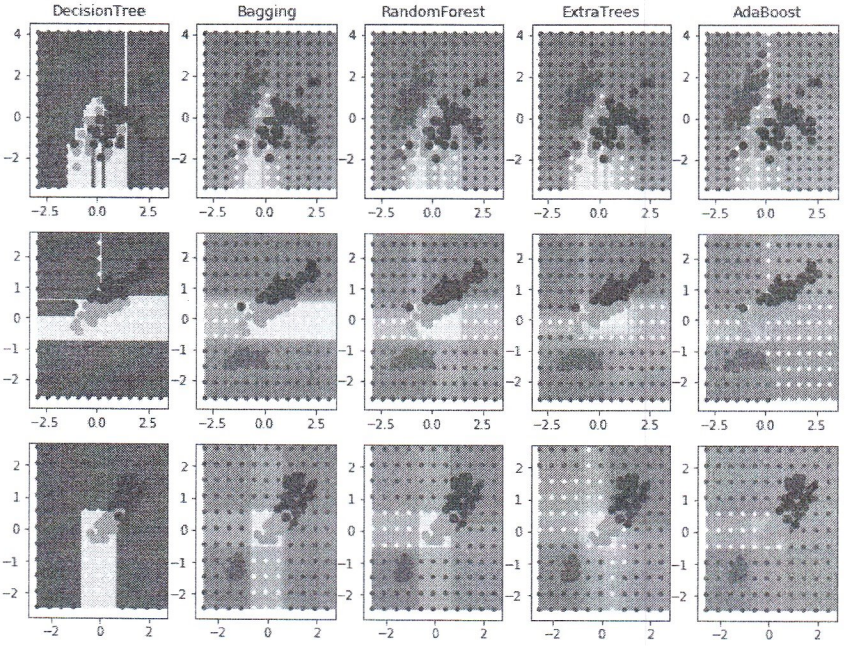
- Decision tree classifier (first column)
- Bagging (second column)
- Random forest (third column)
- Extra-trees (fourth column)
- AdaBoost (fifth column)

In [7]:
```
n_classes = 3
plot_colors = "ryb"
cmap = plt.cm.RdYlBu
plot_step = 0.02  # fine step width for decision surface contours
plot_step_coarser = 0.5  # step widths for coarse classifier guesses

plot_idx = 1

plt.figure(figsize=(12,9))
for pair in ([0, 1], [0, 2], [2, 3]):
    print('\nIris - Variables: ',iris.feature_names[pair[0]],' & ',iris.feature_names[pair[1]])


    for model_name in models:
        model = models[model_name]

        # We only take the two corresponding features
        X = iris.data[:, pair]
        y = iris.target

        # Shuffle
        idx = np.arange(X.shape[0])
        np.random.seed(random_seed)
        np.random.shuffle(idx)
        X = X[idx]
        y = y[idx]

        # Standardize
        mean = X.mean(axis=0)
        std = X.std(axis=0)
        X = (X - mean) / std

        # Train
        clf = clone(model)
        clf = model.fit(X, y)

        scores = clf.score(X, y)
        # Create a title for each column and the console by using str() and
        # slicing away useless parts of the string
        model_title = str(type(model)).split(".")[-1][:-2][:-len("Classifier")]
        model_details = model_title
#         if hasattr(model, "estimators_"):
#             model_details += " with {} estimators".format(len(model.estimators_))
#         print( model_details + " with features", pair, "has a score of", scores )

        print('\t%26s\t%.3f'%(model_name,scores))



        plt.subplot(3, 5, plot_idx)
        if plot_idx <= len(models):
            # Add a title at the top of each column
            plt.title(model_title)

        # Now plot the decision boundary using a fine mesh as input to a
        # filled contour plot
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                             np.arange(y_min, y_max, plot_step))

        # Plot either a single DecisionTreeClassifier or alpha blend the
        # decision surfaces of the ensemble of classifiers
        if isinstance(model, DecisionTreeClassifier):
            Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            cs = plt.contourf(xx, yy, Z, cmap=cmap)
        else:
            # Choose alpha blend level with respect to the number of estimators
            # that are in use (noting that AdaBoost can use fewer estimators
            # than its maximum if it achieves a good enough fit early on)
            estimator_alpha = 1.0 / len(model.estimators_)
            for tree in model.estimators_:
                Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
                Z = Z.reshape(xx.shape)
                cs = plt.contourf(xx, yy, Z, alpha=estimator_alpha, cmap=cmap)

        # Build a coarser grid to plot a set of ensemble classifications
        # to show how these are different to what we see in the decision
        # surfaces. These points are regularly space and do not have a black outline
        xx_coarser, yy_coarser = np.meshgrid(np.arange(x_min, x_max, plot_step_coarser),
                                             np.arange(y_min, y_max, plot_step_coarser))
        Z_points_coarser = model.predict(np.c_[xx_coarser.ravel(), yy_coarser.ravel()]).reshape(xx_coarser.shape)
        cs_points = plt.scatter(xx_coarser, yy_coarser, s=15, c=Z_points_coarser, cmap=cmap, edgecolors="none")

        # Plot the training points, these are clustered together and have a
        # black outline
        for i, c in zip(xrange(n_classes), plot_colors):
            idx = np.where(y == i)
            plt.scatter(X[idx, 0], X[idx, 1], c=c, label=iris.target_names[i],
                        cmap=cmap)

        plot_idx += 1  # move on to the next plot in sequence

plt.suptitle("Classifiers on feature subsets of the Iris dataset")
plt.axis("tight")

plt.show()
```

```
Iris - Variables:  sepal length (cm)  &  sepal width (cm)
                    Decision Tree      0.927
                          Bagging      0.827
                    Random Forest      0.927
        Extremely Randomized Trees     0.927
                        Ada Boost      0.840

Iris - Variables:  sepal length (cm)  &  petal length (cm)
```

```
                    Decision Tree      0.993
                          Bagging      0.953
                    Random Forest      0.993
        Extremely Randomized Trees      0.993
                        Ada Boost      0.993

Iris - Variables:  petal length (cm)  &  petal width (cm)
                    Decision Tree      0.993
                          Bagging      0.973
                    Random Forest      0.993
        Extremely Randomized Trees      0.993
                        Ada Boost      0.993
```

Classifiers on feature subsets of the Iris dataset



Increasing `max_depth` for AdaBoost lowers the standard deviation of the scores (but the average score does not improve).

See the console's output for further details about each model.

In this example you might try to:

- vary the `max_depth` for the `DecisionTreeClassifier`, `BaggingClassifier` and `AdaBoostClassifier`, perhaps try `max_depth=3` for the `DecisionTreeClassifier` or `max_depth=None` for `AdaBoostClassifier`
- vary `n_estimators`

It is worth noting that RandomForests and ExtraTrees can be fitted in parallel on many cores as each tree is built independently of the others. AdaBoost's samples are built sequentially and so do not use multiple cores.

Ensembles classifiers using trees on the loans dataset

In this notebook we apply several ensemble methods to the Iris dataset using tree classifiers and plot the resulting decision surfaces. Note that this notebook has been created using the material from http://scikit-learn.org/stable/modules/ensemble.html

First we load all the required libraries.

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

from sklearn import clone
from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                              AdaBoostClassifier,BaggingClassifier)
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
# from sklearn.externals.six.moves import xrange
from sklearn.tree import DecisionTreeClassifier
%matplotlib inline
```

Next we define some of the parameters required to run the experiments like the number of estimators used in each ensembles and the random seed to be able to reproduce the results.

In [3]:
```
# Number of estimators used in each ensemble
n_estimators = 30

# set the random seed to be able to repeat the experiment
random_seed = 1234
```

Load the dataset

In [4]:
```
df = pd.read_csv("LoansNumerical.csv")
```

In [5]:
```
target = 'safe_loans'
variables = df.columns[df.columns!=target]

X = df[variables].values
y = df['safe_loans'].values
```

Set the models to be compared

- simple decision tree
- bagging
- random forest
- extra tree classifiers
- adaboost

In [6]:
```
models = {'Decision Tree':DecisionTreeClassifier(max_depth=None),
          'Bagging':BaggingClassifier(DecisionTreeClassifier(max_depth=3),n_estimators=n_estimators),
          'Random Forest':RandomForestClassifier(n_estimators=n_estimators),
          'Extremely Randomized Trees':ExtraTreesClassifier(n_estimators=n_estimators),
          'Ada Boost':AdaBoostClassifier(DecisionTreeClassifier(max_depth=3),
                                         n_estimators=n_estimators)}
```

For each model, we apply 10-fold stratified crossvalidation and compute the average accuracy and the corresponding standard deviation

In [7]:
```
scores = {}
for model_name in models:
    clf = models[model_name];
    score = cross_val_score(clf, X, y, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=random_seed))
    scores[model_name]=(np.average(score),np.std(score))
    print('%26s %3.1f %3.1f'%(model_name,100.0*np.average(score),100.0*np.std(score)))
```

```
            Decision Tree 73.1 0.2
                  Bagging 81.5 0.1
            Random Forest 81.7 0.2
Extremely Randomized Trees 81.2 0.2
                Ada Boost 82.0 0.2
```

Then, we print for every variable pair the performance of all the models

In [8]:
```
for model_name in models:
    print('\t%26s\t%3.1f +/- %3.1f'%(model_name,100.0*scores[model_name][0],100.0*scores[model_name][1]))
```

```
                    Decision Tree      73.1 +/- 0.2
                          Bagging      81.5 +/- 0.1
                    Random Forest      81.7 +/- 0.2
        Extremely Randomized Trees      81.2 +/- 0.2
                        Ada Boost      82.0 +/- 0.2
```

In [ ]:

In [ ]:

EVALUATION USING PRECISION, RECALL & ROC

Classifier Evaluation using Precision, Recall, and ROC

We are going to apply logistic regression using the data available from the Lending Club Corporation. The company provides files containing complete loan data for all loans issued through a certain time period, including the current loan status (Current, Late, Fully Paid, etc.) and latest payment information.

We are going to use a small subset of the available data which were previously preprocessed.

The goal of this notebook is to explore the different ways we can use to evaluate the result of classification.

As the very first step, we load all the relevant libraries.

In [2]:
```
# Usual libraries for tools
import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn import model_selection
import matplotlib.pyplot as plt

# Libraries for the evaluation
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import StratifiedKFold

%matplotlib inline
```

Then, we load the data.

In [3]:
```
# loans = pd.read_csv('loansData01.csv')
loans = pd.read_csv('LoansNumerical.csv')
```

In [4]:
```
target = 'safe_loans'
features = loans.columns[loans.columns!=target]

features
```

Out[4]:
```
Index(['sub_grade_num', 'short_emp', 'emp_length_num', 'dti',
       'payment_inc_ratio', 'delinq_2yrs', 'delinq_2yrs_zero',
       'inq_last_6mths', 'last_delinq_none', 'last_major_derog_none',
       'open_acc', 'pub_rec', 'pub_rec_zero', 'revol_util',
       'total_rec_late_fee', 'int_rate', 'total_rec_int', 'annual_inc',
       'funded_amnt', 'funded_amnt_inv', 'installment', 'num_term',
       'grade_num', 'loan_amnt'],
      dtype='object')
```

From the data dictionary available on the Lending Club Corporation website, we have the description of the variables:

- **sub_grade_num**, the sub-grade of the loan as a number from 0 to 1
- **short_emp**, one year or less of employment
- emp_length_num, number of years of employment
- **dti**, debt to income ratio
- **payment_inc_ratio**, ratio of the monthly payment to income
- **delinq_2yrs**, number of delinquincies
- **delinq_2yrs_zero**, no delinquincies in last 2 years
- **inq_last_6mths**, number of creditor inquiries in last 6 months
- **last_delinq_none**, has borrower had a delinquincy
- **last_major_derog_none**, has borrower had 90 day or worse rating
- **open_acc**, number of open credit accounts
- **pub_rec**, number of derogatory public records
- **pub_rec_zero**, no derogatory public records
- **revol_util**, percent of available credit being used
- **total_rec_late_fee**, total late fees received to day
- **int_rate**, interest rate of the loan
- **total_rec_int**, interest received to date
- **annual_inc**, annual income of borrower
- **funded_amnt**, amount committed to the loan
- **funded_amnt_inv**, amount committed by investors for the loan
- **installment**, monthly payment owed by the borrower
- **num_term**, number of payments on the loan. Values are in months and can be either 36 or 60
- **grade_num**, LC assigned loan grade as a number
- **loan_amnt**, the listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.

The target variable (the class) is **safe_loans** that is +1 if the loan is safe -1 if it is risky

The input variables x are the columns corresponding to the features, the output variable y is the column corresponding to the target variable.

In [5]:
```
X = loans[features]
y = loans[target]
```

First we apply plain logistic regression without regularization (thus α is zero). The Scikit-learn function does not allow to specify α but it uses a parameter $C=1/\alpha$. Accordingly, to have no regularization we need to specify a huge value of C. And we evaluate the model using plain crossvalidation.

In [7]:
```
simple_logistic = linear_model.LogisticRegression(C=10e10)
```

```
simple_logistic.fit(X,y);
simple_eval = model_selection.cross_val_score(simple_logistic, X, y, cv=StratifiedKFold(10,shuffle=True,random_state=1234))
print("Simple Logistic Regression\t%3.2f\t%3.2f" % (np.average(simple_eval), np.std(simple_eval)))
```

```
Simple Logistic Regression      0.81    0.00
```

Evaluating Classification - Accuracy

Accuracy is the most known and most widely used measure of classification performance. It measure the percentage of correct classification achieved by the model.

$$accuracy = \frac{numberofcorrectlyclassifiedexamples}{numberofexamples}$$

Confusion Matrix

Accuracy evaluates the percentage of correct predictions but not the errors are equal. To have a better sense of how a classifier model performs the confusion matrix is usually examined. The confusion matrix includes

- **the true positives,** the number of examples labeled as positives and prediceted as positives
- **the true negatives,** the number of examples labeled as negatives and prediceted as negatives
- **the false positives,** the number of examples labeled as negatives and prediceted as positives
- **the false negatives,** the number of examples labeled as positives and prediceted as negatives

The confusion matrix is typically shown as,

	Predicted Positive	Predicted Negative
Labeled Positive	TP	FN
Labeled Negative	FP	TN

We can compute the confusion matrix for the three models we developed.

In [9]:
```
yp = simple_logistic.predict(X);

def PrintConfusionMatrix(model, true_y, predicted_y, positive=1, negative=-1):
    cm = confusion_matrix(true_y,predicted_y)
    print("\t"+str(model.classes_[0])+"\t"+str(model.classes_[1]))
    print(str(model.classes_[0]) + "\t",cm[0][0],"\t",cm[0][1])
    print(str(model.classes_[1]) + "\t",cm[1][0],"\t",cm[1][1])

print("Confusion Matrix - Simple Logistic")
PrintConfusionMatrix(simple_logistic, y, yp)
```

```
Confusion Matrix - Simple Logistic
        -1      1
-1       937     22185
1        705     98635
```

Precision and Recall

Precision and recall are alternative measures to plain accuracy introduced in the area of information retrieval and search engine. Precision focuses on the percentage of correctly classified positive examples or in the information retrieval context represents the percentage of actually good documents that have been shown as a result. Recall focuses on the percentage of positively classified examples with respect to the number of existing good documents or in the information retrieval context, recall represents the percentage of good documents shown with respect to the existing ones.

$$precision = \frac{TP}{TP+FP}$$

$$recall = \frac{TP}{TP+FN}$$

In [10]:
```
print("Precision %3.2f" % precision_score(y,yp))
print("Recall    %3.2f" % recall_score(y,yp))
```

```
Precision 0.82
Recall    0.99
```

Probabilistic Models & Classification Thresholds

Up to now we used logistic regression to predict classifier labels, however, logistic regression typically returns a probability that an example should be label using a certain class. For example, the first example, is associated to two probabilities one corresponding to label -1 and one corresponding to label +1.

In [12]:
```
X.head(1)
```

Out[12]:

	sub_grade_num	short_emp	emp_length_num	dti	payment_inc_ratio	delinq_2yrs	delinq_2yrs_zero	inq_last_6mths	last_delinq_none	last_major_d
0	0.4	0	11	27.65	8.1435	0.0	1.0	1.0	1	

1 rows × 24 columns

In [13]:
```
yp = simple_logistic.predict(X)
yprob = simple_logistic.predict_proba(X)
print("Example %d:\n\tP(-1|x[%d])=%3.2f\n\tP(+1|x[%d])=%3.2f\n\t=> Labeled as %d" %(0 , 0, yprob[0,0], 0, yprob[0,1], yp[0]))
```

```
Example 0:
        P(-1|x[0])=0.23
        P(+1|x[0])=0.77
        => Labeled as 1
```

The class assignment is based on the label with the largest probability so it is equivalent to using a threshold of 0.5 to decide which class to assign to an example. However, we might decide to use a different threshold and for instance label as positive only examples with a $P(+1|x)>0.75$ this would label as positive only cases for which we are more confident that should be labeled as positive.

In [14]:
```
def ClassifyWithThreshold(probabilities, threshold):
    return [+1 if x>=threshold else -1 for x in probabilities]

yp_confident = ClassifyWithThreshold(yprob[:,1],0.75)

print("Confusion Matrix - Simple Logistic (0.75)")
PrintConfusionMatrix(simple_logistic, y, yp_confident)

print("\n")
print("Precision %3.2f" % precision_score(y,yp_confident))
print("Recall    %3.2f" % recall_score(y,yp_confident))
```

```
Confusion Matrix - Simple Logistic (0.75)
       -1      1
-1      9212    13910
1       16578   82762


Precision 0.86
Recall    0.83
```

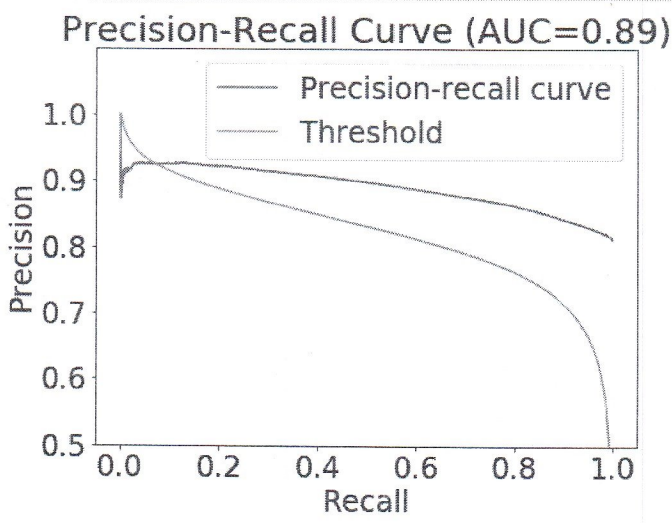
By increasing the threshold we increased the precision and thus we decreased the number of false positives while we increased the number of false negatives. Thus by modifying the threshold we can change the way our classifier makes mistakes.

Precision-Recall Curve

By modifying the threshold we can optimize our classifier to focus on one of the two metrics (precision or recall). To analyze precision varies depending on the threshold we can use the precision-recall curve which is computed by computing precision and recall from the threshold of 1 to the 0 threshold and plotting the result.

In [17]:
```
y_true = y
yp = simple_logistic.predict(X)
yprob = simple_logistic.predict_proba(X)
precision, recall, thresholds = precision_recall_curve(y_true=y, probas_pred=yprob[:,1])
auc = average_precision_score(y, yprob[:,1])
```

In [18]:
```
plt.figure(1, figsize=(8, 6));
font = {'family':'sans', 'size':24};
plt.rc('font', **font);
plt.plot(recall, precision, label="Precision-recall curve");
plt.xlabel('Recall');
plt.ylabel('Precision');
plt.ylim([0.5,1.1])
plt.yticks(np.arange(0.5,1.01,.1))
plt.title('Precision-Recall Curve (AUC=%3.2f)'%auc);
plt.plot(recall[:-1],thresholds, label="Threshold");
plt.legend()
plt.show()
```



Receiver Operating Characteristic (ROC) Curves

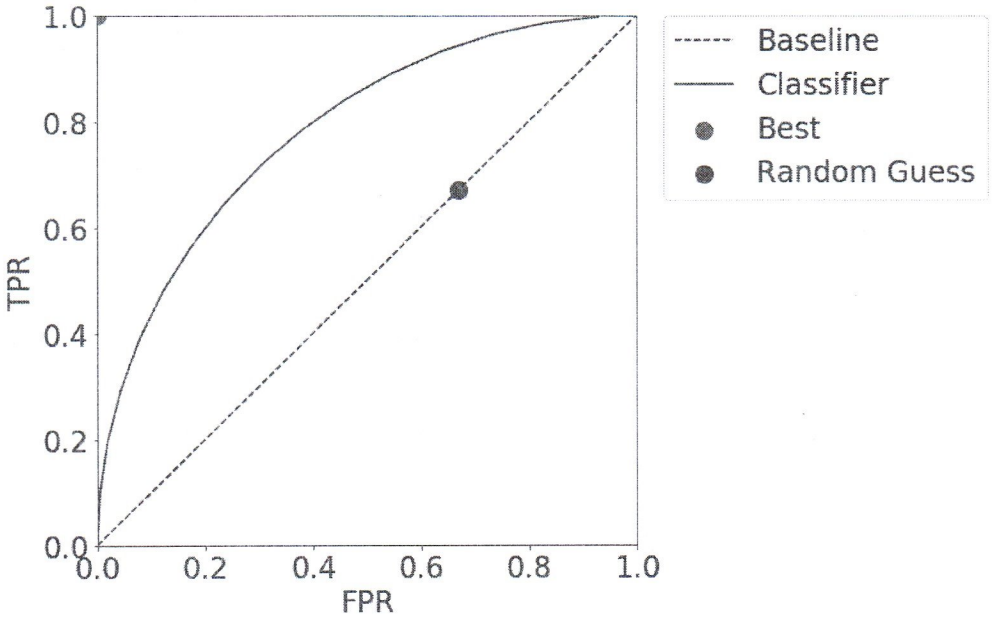
Similar to precision-recall curves, they plot the True Positive Rate (TPR) against the False Positive Rate (FPR)

$TPR = \frac{TP}{TP+FN}$

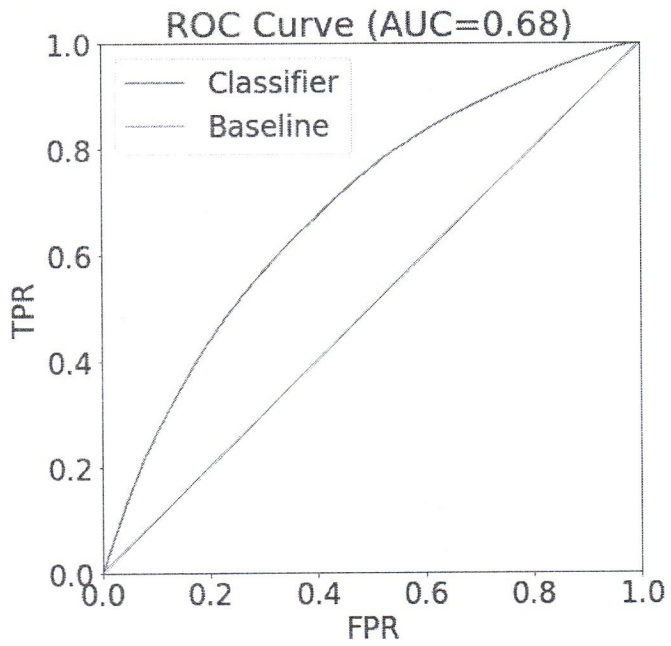
$FPR = \frac{FP}{TN+FP}$

In [22]:
```
import math

plt.figure(1, figsize=(8, 8));
font = {'family':'sans', 'size':24};
plt.rc('font', **font);
plt.xlabel('FPR');
plt.ylabel('TPR');
plt.ylim([0.0,1.0])
plt.xlim([0.0,1.0])
plt.plot([0.0,1.0],[0.0,1.0],color='black',ls='--',label='Baseline')
angle=np.arange(math.pi, math.pi/2,-0.1)
fpr = np.cos(angle)+1.
tpr = np.sin(angle)
plt.plot(fpr,tpr,label='Classifier',color='black')
plt.yticks(np.arange(0.0,1.01,.2))
plt.scatter([0.0],[1.0], s=200, color='red', label='Best')
plt.scatter([0.67],[0.67], s=200, color='blue', label='Random Guess')
```

```
# plt.legend()
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



In [28]:
```
fpr, tpr, thresholds = roc_curve(y_true=y, y_score = yprob[:,1], pos_label=1)
roc_auc = roc_auc_score(y_true=y, y_score = yprob[:,1])
plt.figure(1, figsize=(8, 8));
font = {'family':'sans', 'size':24};
plt.rc('font', **font);
plt.xlabel('FPR');
plt.ylabel('TPR');
plt.plot(fpr,tpr,label='Classifier')
# plt.plot(fpr,thresholds,label='Thresholds')
plt.plot([0.0,1.0],[0.0,1.0],label='Baseline')
plt.yticks(np.arange(0.0,1.01,.2))
plt.title('ROC Curve (AUC=%3.2f)'%roc_auc)
plt.ylim([0.0,1.0])
plt.xlim([0.0,1.0])
plt.legend()
plt.show();
```



As for the precision-recall curves, we can use the threshold value to search for the best TPR/FPR tradeoff.

The ideal value of AUC for a ROC curve is one however, it never happens. According, we look for classifiers with an AUC as large as possible and at least greater than Ideally, the AUC for ROC ion-recall curves, thus when selecting with two models we compare their AUC.

Gradient Boosting Example

We implement a basic version of gradient boosting for regression. First we import the libraries we need and define some helper functions we are going to use to plot the model prediction and the residuals.

In [4]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import tree

# set the font for plotting and the figure size
font = {'size': 14}
plt.rc('font', **font)
plt.figure(figsize=(8,6))
```

Out[4]:
```
<matplotlib.figure.Figure at 0x10e4b00f0>
```

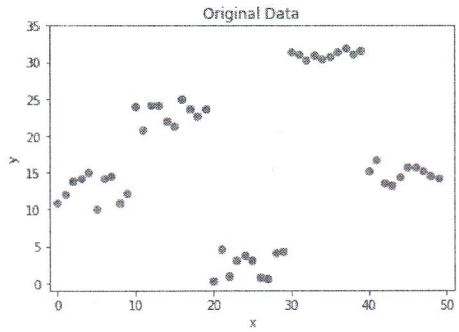
Helper Functions

In [5]:
```
def GenerateData():
    X = np.arange(0,50)
    # just random uniform distributions in differnt range

    y1 = np.random.uniform(10,15,10)
    y2 = np.random.uniform(20,25,10)
    y3 = np.random.uniform(0,5,10)
    y4 = np.random.uniform(30,32,10)
    y5 = np.random.uniform(13,17,10)

    y = np.concatenate((y1,y2,y3,y4,y5))
    y = y[:,None]

    X = X.reshape(-1, 1)
    y = y.reshape(-1, 1)
    return X, y

def PlotData(X,y,yp=[],label='',title='',fn=None):
    plt.scatter(X,y)
    plt.xlim([-1,51])
    plt.ylim([-1,35])
    plt.xlabel('x')
    plt.ylabel('y')
    if (title!=''):
        plt.title(title)
    if (fn!=None):
        plt.savefig(fn)
    #
    # if (yp!=[]):
    #     plt.plot(X,yp,color = 'red')
    #
    plt.show()

def PlotModel(X,y,yp,title='',fn=None):
    plt.scatter(X,y, color = 'blue')
    plt.plot(X,yp,color = 'red')
    plt.xlim([-1,51])
    plt.ylim([-1,35])
    plt.xlabel('x')
    plt.ylabel('y')
    if (title!=''):
        plt.title(title)
    if (fn!=None):
        plt.savefig(fn)
    plt.show()

def PlotResidual(X,y,label='',title='',fn=None):
    plt.scatter(X,y,color = 'green')
    plt.xlim([-1,51])
    plt.ylim([-25,25])
    plt.xlabel('x')
    plt.ylabel('y')
    if (title!=''):
        plt.title(title)
    if (fn!=None):
        plt.savefig(fn)
    plt.show()
```

Data Generation

Let's generate the example data and plot them

In [6]:
```
# generate the data
X, y = GenerateData()

PlotData(X,y,title='Original Data', fn='GradientBoosting-000-data.png')
```



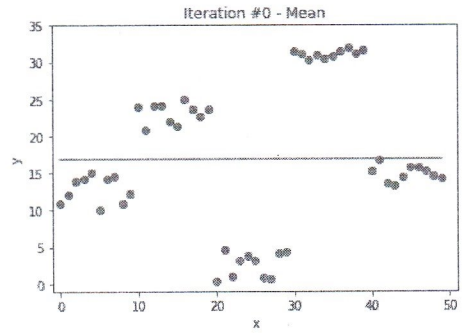
First Two Steps

We implement the first two basic steps and check the result. First, we initialize a vector **prediction** that we will use to store the sum of all the predictions.

In [8]:
```
# initially the prediction is filled with zeros
prediction = np.zeros((50,1))
```

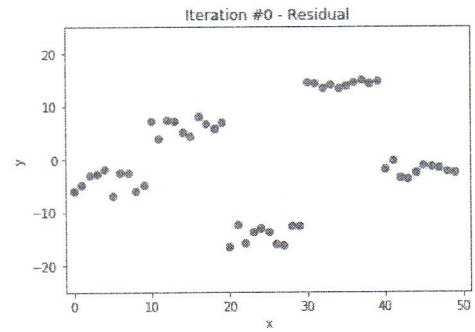
The first predictor is the mean of the target variable. We init the vector **yp** for the current prediction with the mean, add it to the vector **prediction** and plot the result.

In [9]:
```
# initially the model is just the mean
yp = np.ones((50,1))*y.mean()

# update the prediction
prediction = np.add(prediction, yp)

PlotModel(X,y,yp=prediction,title='Iteration #1 - Mean',fn='GradientBoosting-001-model.png')
```



Given the current prediction we compute the residual **y_residual** as the difference between the target vector **y** and the current prediction vector **prediction**

In [10]:
```
# update the residual
y_residual = np.subtract(y, prediction)
PlotResidual(X,y_residual,title='Iteration #0 - Residual',fn='GradientBoosting-001-residual.png')
```

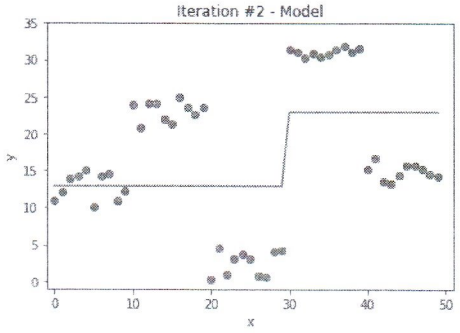


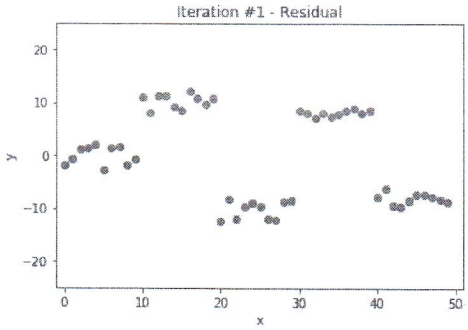
The residual are quite similar to the original points but centered around zero since we subtracted the mean. Now it is time to use the first real model training a decision stump (a tree of depth one) to predict the current residual.

In [11]:
```
clf = tree.DecisionTreeRegressor(max_depth=1)
clf = clf.fit(X,y_residual)
yp = clf.predict(X)
```

The prediction for the residual is added to the overall prediction vector **prediction** and the residual is computed.

In [12]:
```
# update the prediction
prediction = np.add(prediction, yp.reshape(-1,1))
PlotModel(X,y,yp=prediction,title='Iteration #2 - Model',fn='GradientBoosting-03-model.png')

y_residual = np.subtract(y, prediction)

PlotResidual(X,y_residual,title='Iteration #1 - Residual',fn='GradientBoosting-04-residual.png')
```



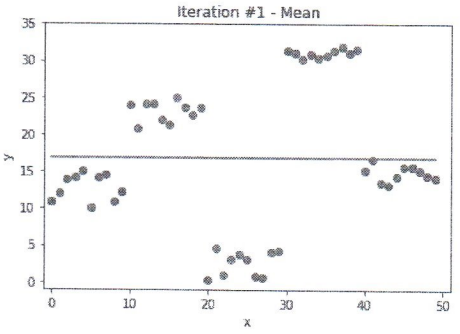


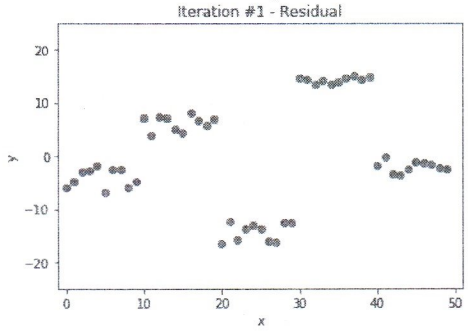
This step should be repeated until convergence.

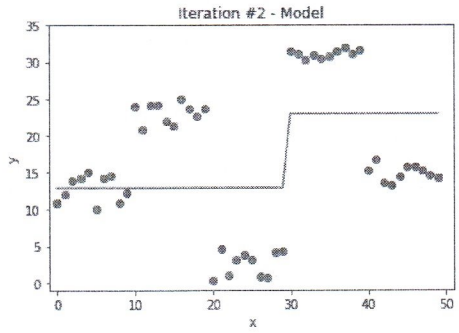
Complete Algorithm

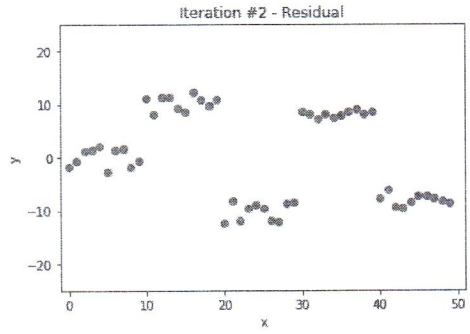
Let's use the previous steps to build a basic algorithm for gradient boosting using regression trees. Note that this only implements training. For testing we would need to store all the models to be able to implement the testing on unseen cases.

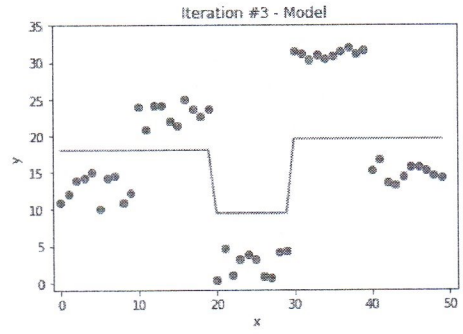
In [13]:
```
# initially the prediction is filled with zeros
prediction = np.zeros((50,1))

# initially the model is just the mean
yp = np.ones((50,1))*y.mean()
PlotModel(X,y,yp=yp,title='Iteration #1 - Mean',fn='GradientBoosting-001-model.png')

# update the prediction
prediction = np.add(prediction, yp)

# update the residual
y_residual = np.subtract(y, prediction)
PlotResidual(X,y_residual,title='Iteration #1 - Residual',fn='GradientBoosting-001-residual.png')

no_boosting_runs = 30

for i in range(no_boosting_runs):
    clf = tree.DecisionTreeRegressor(max_depth=1)
    clf = clf.fit(X, y_residual)
    yp = clf.predict(X)

    # update the prediction
    prediction = np.add(prediction, yp.reshape(-1, 1))
    PlotModel(X, y, yp=prediction, title=('Iteration #%d - Model'%(i+2)), fn=('GradientBoosting-%03d-model.png'%(i+2)))

    y_residual = np.subtract(y, prediction)

    PlotResidual(X, y_residual, title=('Iteration #%d - Residual'%(i+2)), fn='GradientBoosting-%03d-residual.png'%(i+2))
```

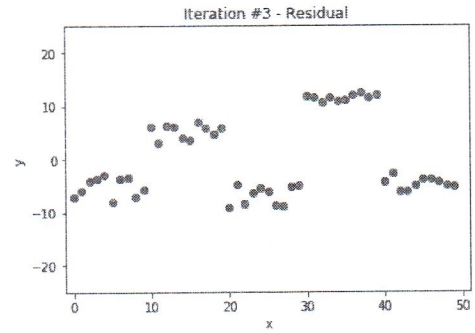


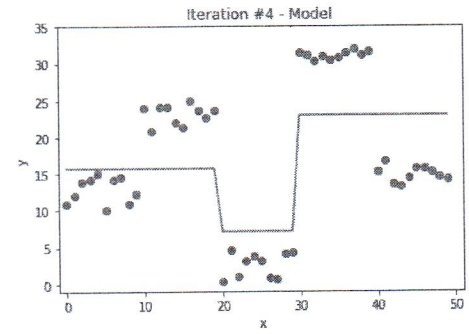

Iteration #1 - Residual
x
y

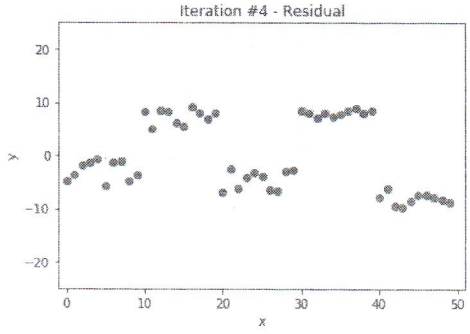

Iteration #2 - Model
x
y

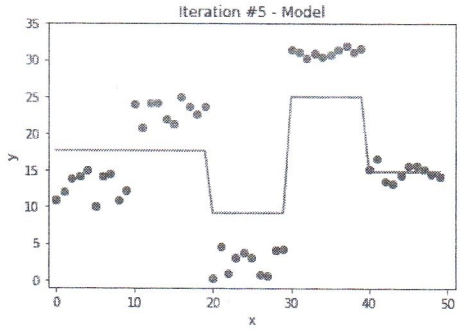

Iteration #2 - Residual
x
y

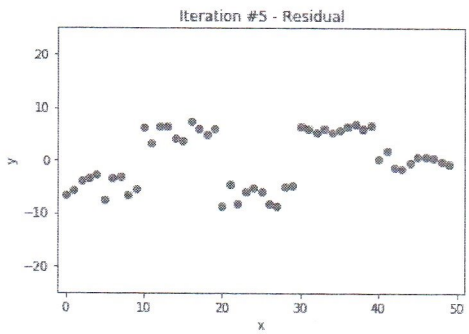

Iteration #3 - Model
x
y

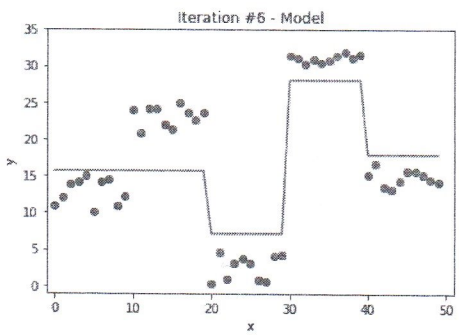

Iteration #3 - Residual
x
y

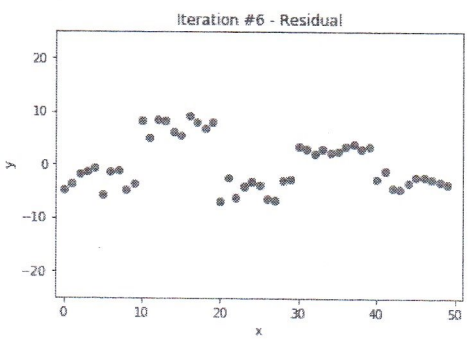

Iteration #4 - Model
x
y

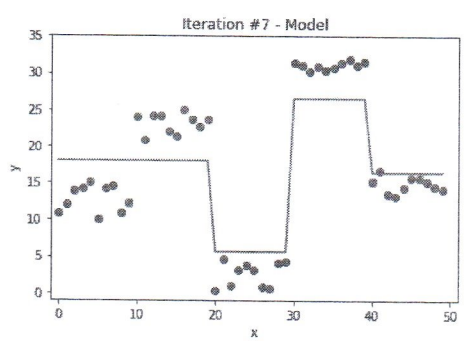

Iteration #4 - Residual
x
y

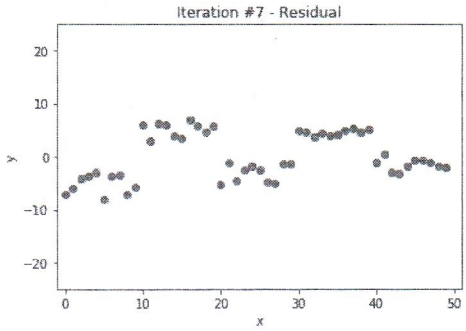

Iteration #5 - Model
x
y

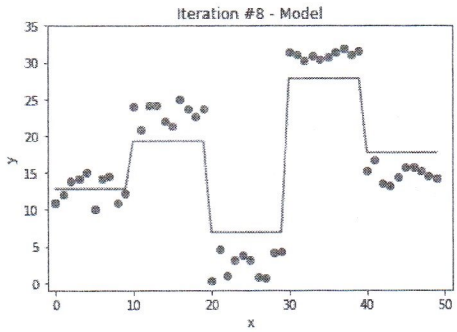

Iteration #5 - Residual
x
y

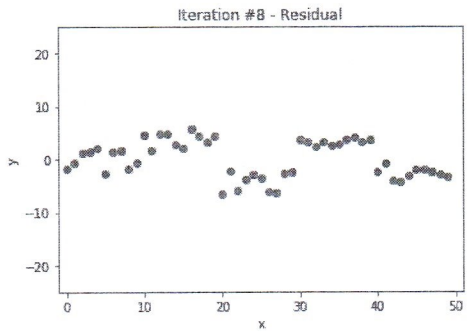

Iteration #6 - Model
x
y

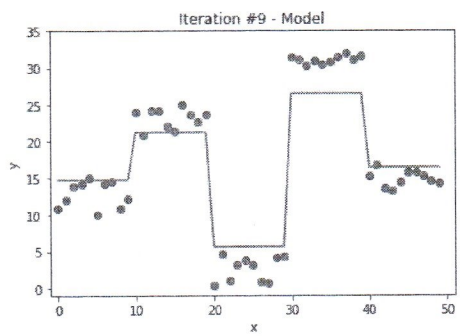

Iteration #6 - Residual
x
y

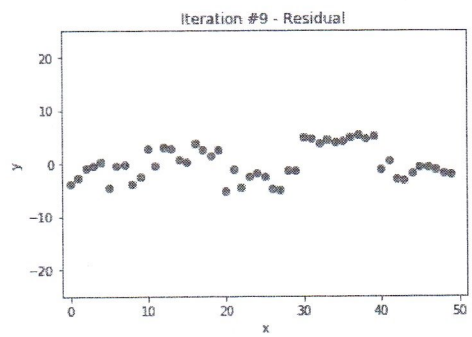

Iteration #7 - Model
x
y

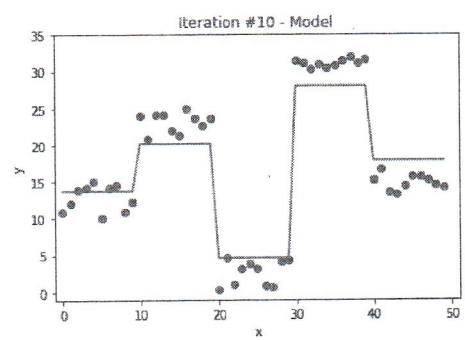

Iteration #7 - Residual
x
y

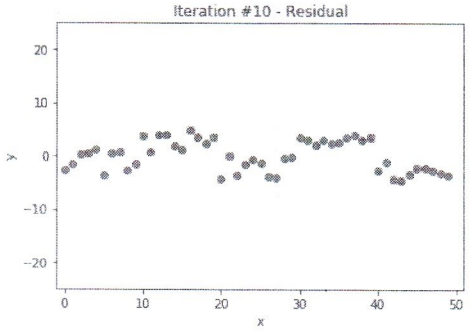

Iteration #8 - Model
x
y

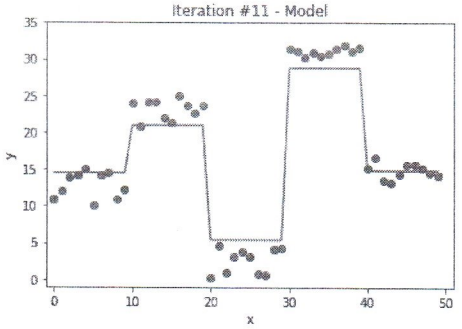

Iteration #8 - Residual
x
y

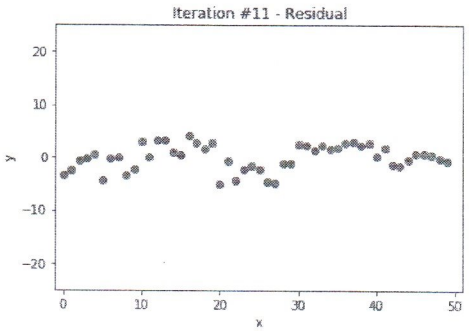

Iteration #9 - Model
x
y

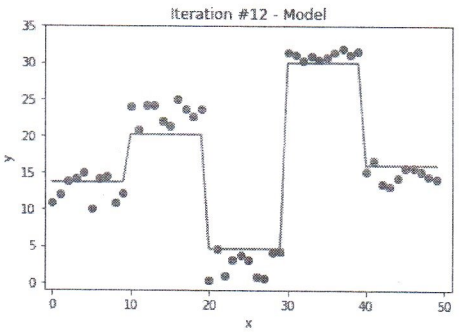

Iteration #9 - Residual
x
y

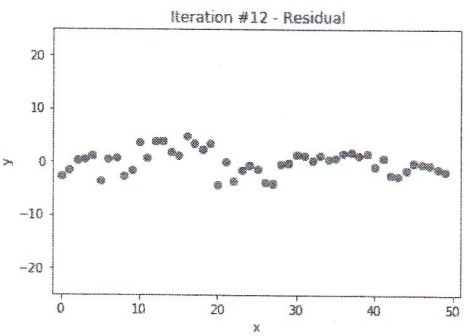

Iteration #10 - Model
x
y

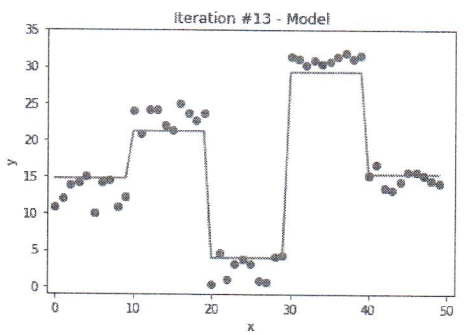

Iteration #10 - Residual
y
x

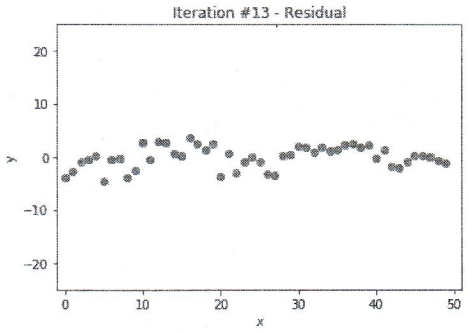

Iteration #11 - Model
y
x

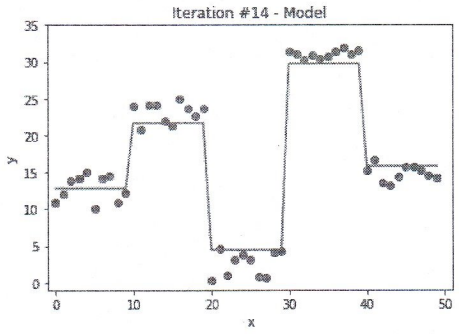

Iteration #11 - Residual
y
x

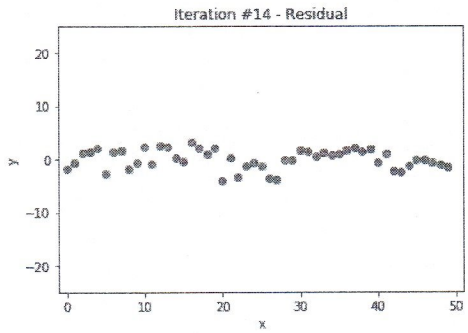

Iteration #12 - Model
y
x

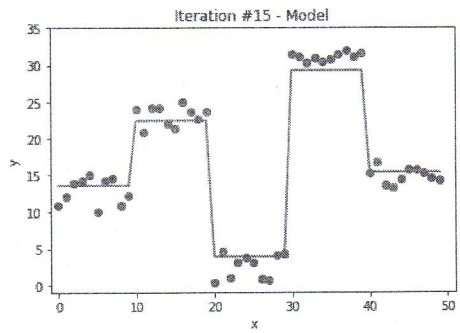

Iteration #12 - Residual
y
x

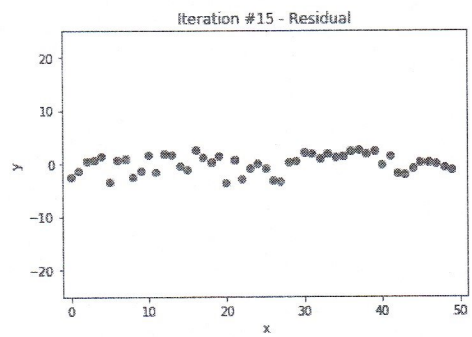

Iteration #13 - Model
y
x

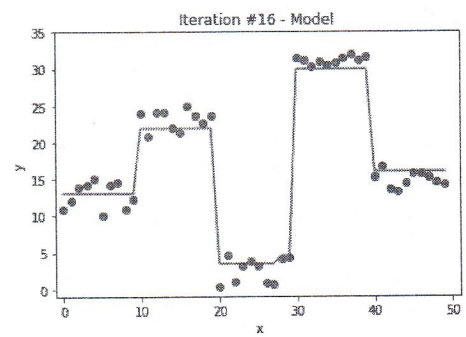

Iteration #13 - Residual
y
x

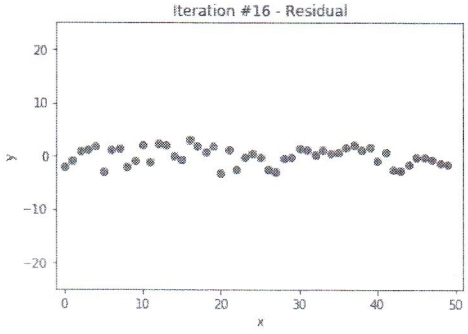

Iteration #14 - Model
y
x

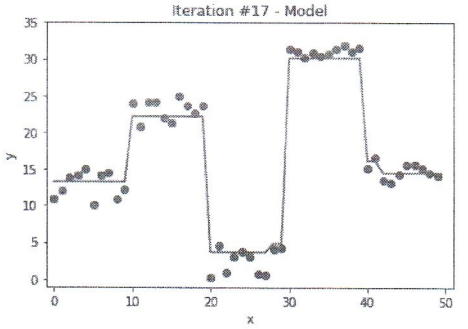

Iteration #14 - Residual
y
x

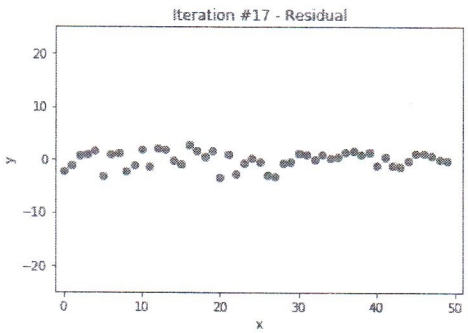

Iteration #15 - Model
y
x

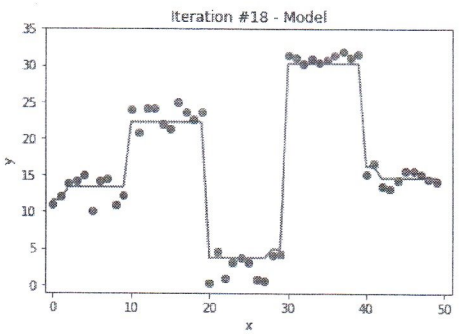

Iteration #15 - Residual
y
x

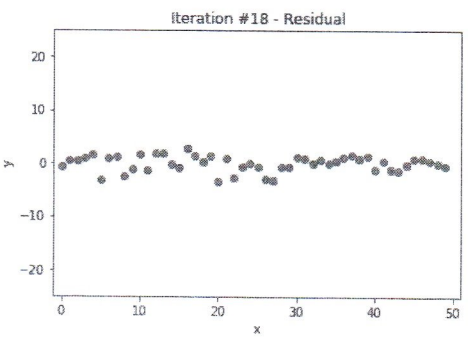

Iteration #16 - Model
y
x

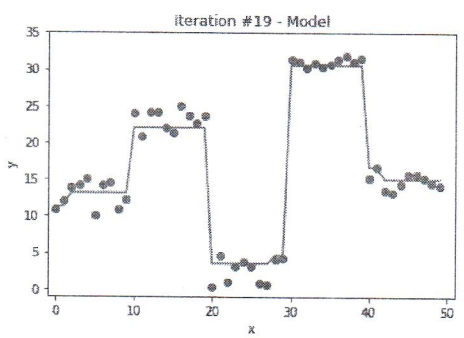

Iteration #16 - Residual
y
x

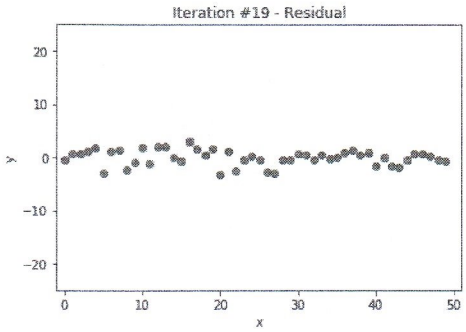

Iteration #17 - Model
y
x

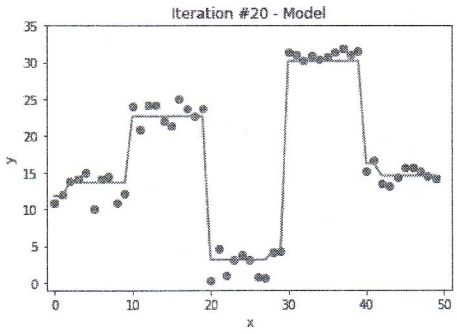

Iteration #17 - Residual
y
x

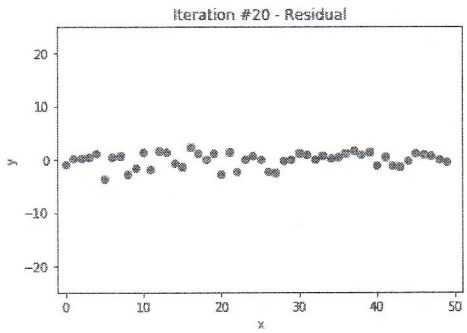

Iteration #18 - Model
y
x

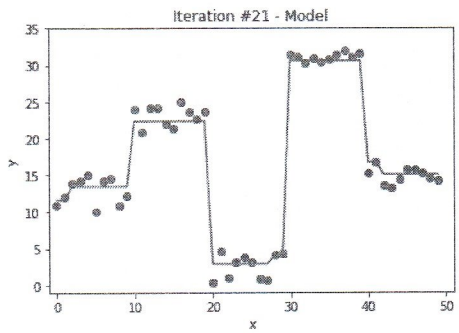

Iteration #18 - Residual
y
x

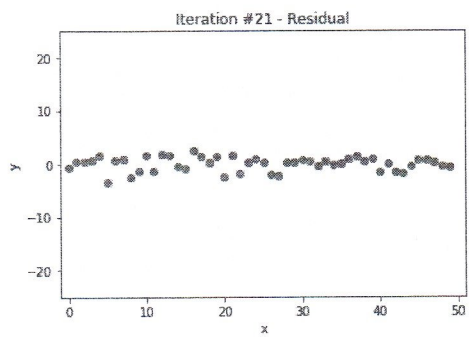

Iteration #19 - Model
y
x

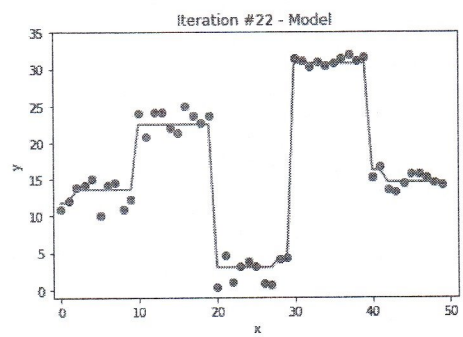

Iteration #19 - Residual
y
x

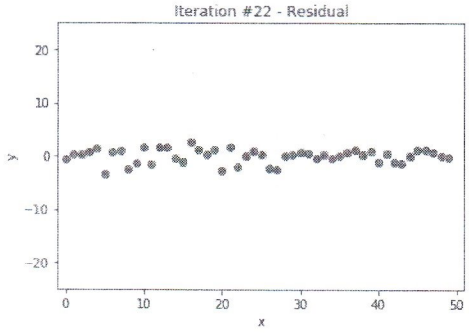

Iteration #20 - Model
y
x

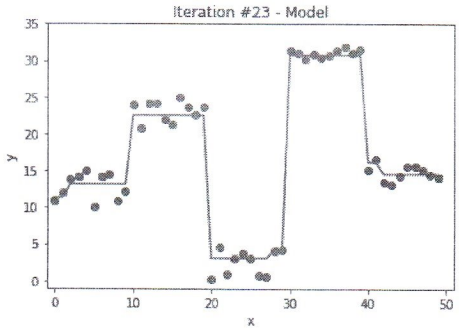

Iteration #20 - Residual
y
x

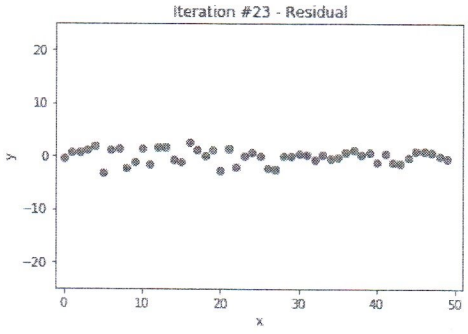

Iteration #21 - Model
y
x

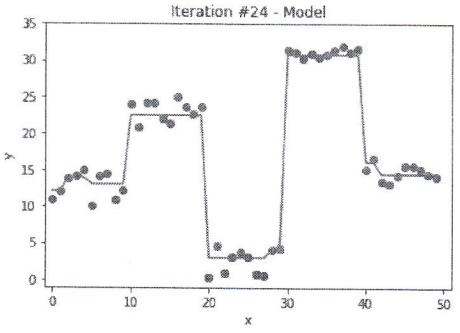

Iteration #21 - Residual
y
x

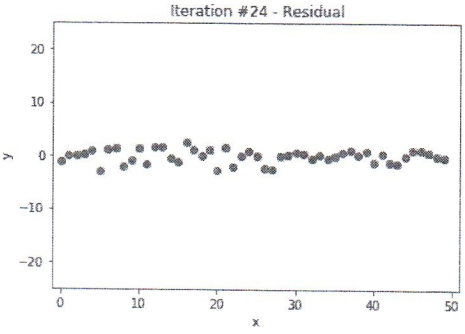

Iteration #22 - Model
y
x

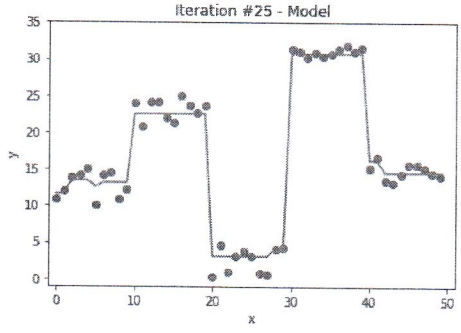

Iteration #22 - Residual
y
x

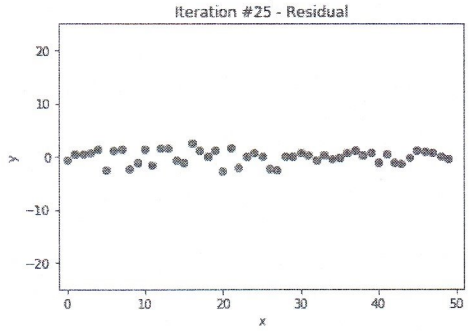

Iteration #23 - Model
y
x

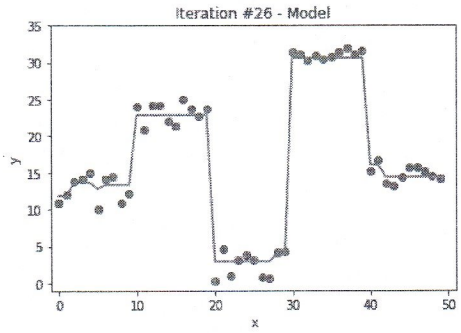

Iteration #23 - Residual
y
x

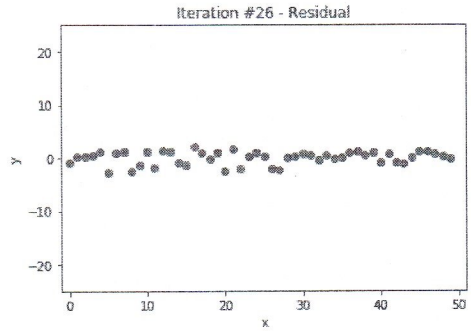

Iteration #24 - Model
y
x

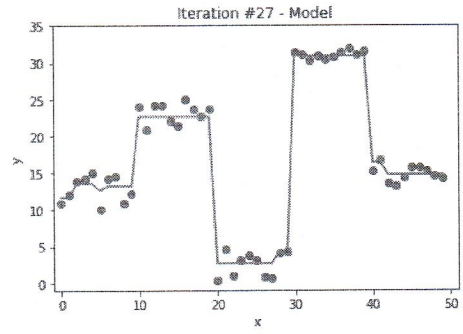

Iteration #24 - Residual
y
x

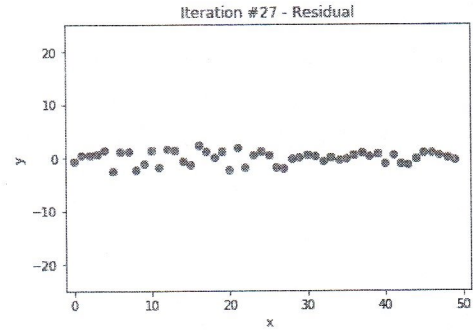

Iteration #25 - Model
y
x

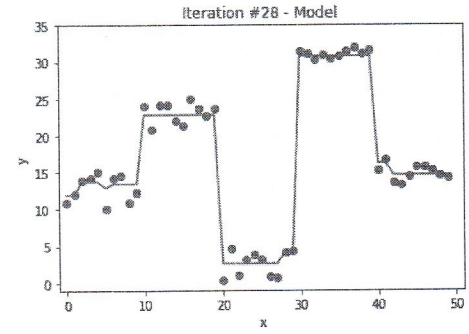

Iteration #25 - Residual
y
x

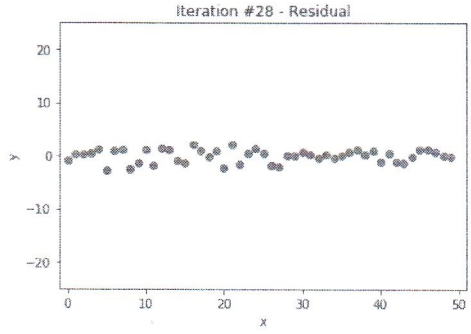

Iteration #26 - Model
y
x

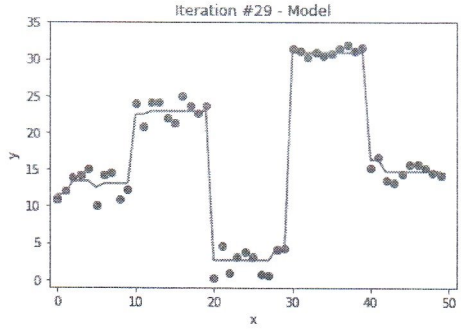

Iteration #26 - Residual
y
x

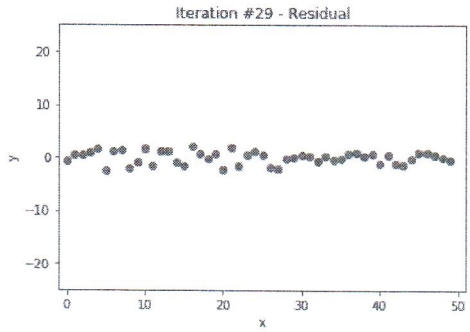

Iteration #27 - Model
y
x

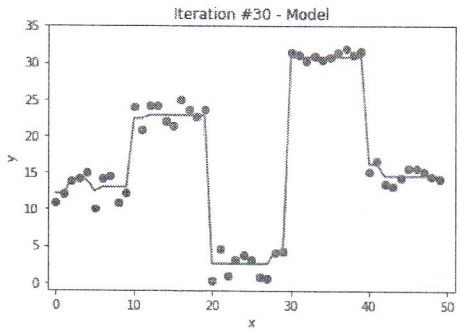

Iteration #27 - Residual
y
x

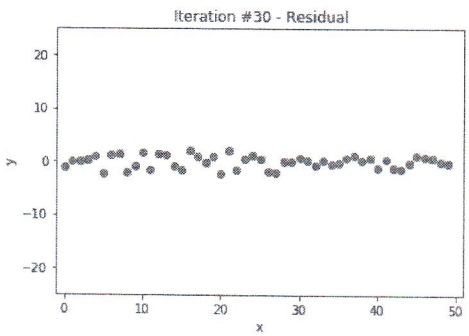

Iteration #28 - Model
y
x

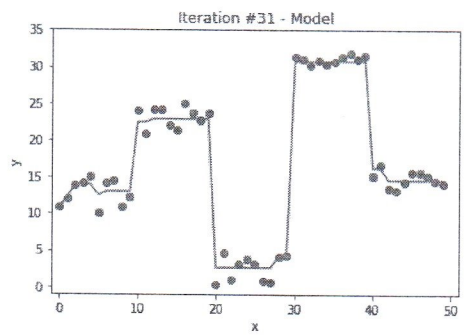

Iteration #28 - Residual
y
x

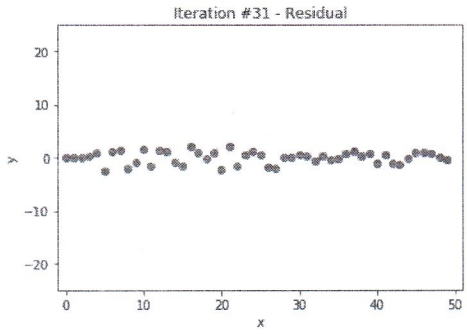

Iteration #29 - Model
y
x


Iteration #29 - Residual
y
x


Iteration #30 - Model
y
x


Iteration #30 - Residual
y
x


Iteration #31 - Model
y
x



In [ ]:

House Prices: Advanced Regression Techniques (Part 0 - Exploration)

In this notebook we apply linear regression to some data from a Kaggle. The notebook is divided into two sections. First we perform some in-depth data exploration and pre-processing, next we build the actual models.

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

This notebook is derived from other existing notebooks by other authors, like,

- https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard

First we import some of the libraries we will need:

In [2]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import skew
from scipy.stats import norm
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

In [3]:
```
all_data = pd.read_csv("HousePricesInputVariables.csv")
```

Missing Values

Let's check how many missing values are in the data set and how can we deal with them.

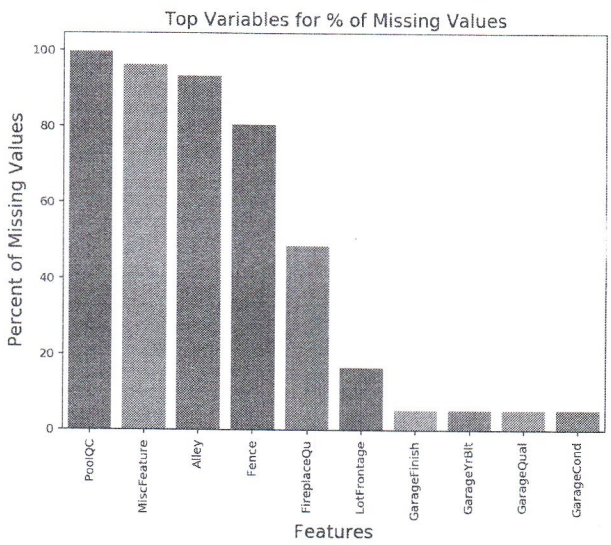
In [4]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Out[4]:

	Missing Ratio
PoolQC	99.657417
MiscFeature	96.402878
Alley	93.216855
Fence	80.438506
FireplaceQu	48.646797
LotFrontage	16.649538
GarageFinish	5.447071
GarageYrBlt	5.447071
GarageQual	5.447071
GarageCond	5.447071

In [5]:
```
f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index[:10], y=all_data_na[:10])
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of Missing Values', fontsize=15)
plt.title('Top Variables for % of Missing Values', fontsize=15)
```

Out[5]:
```
Text(0.5, 1.0, 'Top Variables for % of Missing Values')
```



We note that some of the attributes have the vast majority of the values set to unknown. If we apply imputation without any knowledge about the meaning of what a missing value represent, we would end up with attributes set almost completely to the same value. So they would be almost useless.

How can we deal with all these missing values? First, we need to ask the domain expert whether some missing values have a special meaning. We actually don't have a domain expert but we have the data description in which we find out that

- Alley: Type of alley access to property, NA means "No alley access"
- BsmtCond: Evaluates the general condition of the basement, NA means "No Basement"
- BsmtExposure: Refers to walkout or garden level walls, NA means "No Basement"
- BsmtFinType1: Rating of basement finished area, NA means "No Basement"
- BsmtFinType2: Rating of basement finished area (if multiple types), NA means "No Basement"
- FireplaceQu: Fireplace quality, NA means "No Fireplace"
- Functional: data description says NA means typical
- GarageType: Garage location, NA means "No Garage"
- GarageFinish: Interior finish of the garage, NA means "No Garage"
- GarageQual: Garage quality, NA means "No Garage"
- GarageCond: Garage condition, NA means "No Garage"
- PoolQC: Pool quality, NA means "No Pool"
- Fence: Fence quality, NA means "No Fence"
- MiscFeature: Miscellaneous feature not covered in other categories, NA means "None"

Accordingly, we need to keep this information into account when dealing with the missing values of these variables.

Impute Categorical Values with Known Meaning

Let's impute the missing values for these attributes

In [6]:
```
all_data["Alley"] = all_data["Alley"].fillna("None")

# for all basement features a missing value means that there is no basement
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')

all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")

all_data["Functional"] = all_data["Functional"].fillna("Typ")

for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

all_data["PoolQC"] = all_data["PoolQC"].fillna("None")

all_data["Fence"] = all_data["Fence"].fillna("None")

all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
```

We can also deal with other numerical variables and use some heuristic to impute them. For instance, for **LotFrontage**, since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can fill in missing values by the median LotFrontage of the neighborhood.

In [7]:
```
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))
```

We can also set the garage year, area and number of cars to zero for missing values since this means that there is no garage.

In [8]:
```
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
```

And we can do the same for basement measures.

In [9]:
```
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)
```

When **MasVnrArea** and **MasVnrType** are missing, it most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

In [10]:
```
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

For **MSZoning** (the general zoning classification), 'RL' is by far the most common value. So we can fill in missing values with 'RL'

In [11]:
```
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
```

Utilities has all records are "AllPub", except for one "NoSeWa" and 2 missing values. Since the house with 'NoSewa' is in the training set, this feature won't help to predict labels in the test set. We can then safely remove it.

In [12]:
```
all_data = all_data.drop(['Utilities'], axis=1)
```

KitchenQual has only one missing value, and same as Electrical, we set it to the most frequent values (that is 'TA')

In [13]:
```
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])
```

Exterior1st and **Exterior2nd** have only one missing value. We will use the mode (the most common value)

In [14]:
```
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])
```

For **SaleType** we can use the most frequent value (the mode) which correspond to "WD"

In [15]:
```
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
```

For **MSSubClass** a missing value most likely means No building class. We can replace missing values with None

In [16]:
```
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

Electrical has one missing value. Since this feature has mostly 'SBrkr', we can set that for the missing value.

In [17]:
```
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])
```

Anymore missing?

In [18]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Out[18]:

	Missing Ratio

No more missing values! What would have happened if we did not use or did not have the data description?

Distribution of Numerical Variables

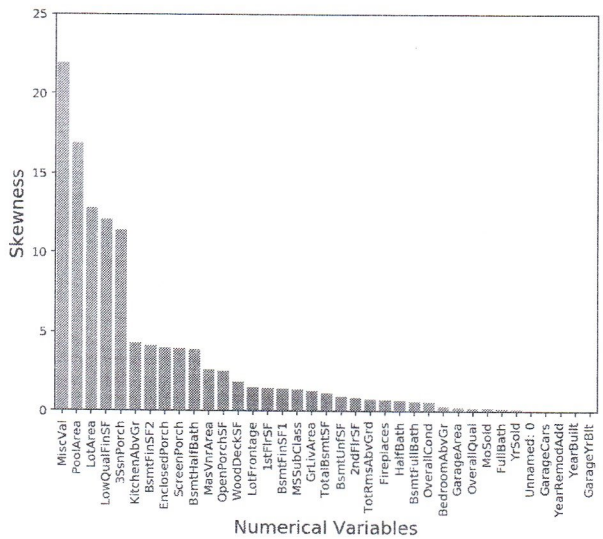
We now explore the distribution of numerical variables. As we did for the class, we will apply the log1p function to all the skewed numerical variables.

In [20]:
```
# take the numerical features
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))

skewness = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.data})
# select the variables with a skewness above a certain threshold
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:7: FutureWarning: Series.data is deprecated and will be removed in
a future version
  import sys
```

In [21]:
```
skewness = skewness.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness['Variable'], y=skewness['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

Out[21]:
```
Text(0.5, 1.0, '')
```



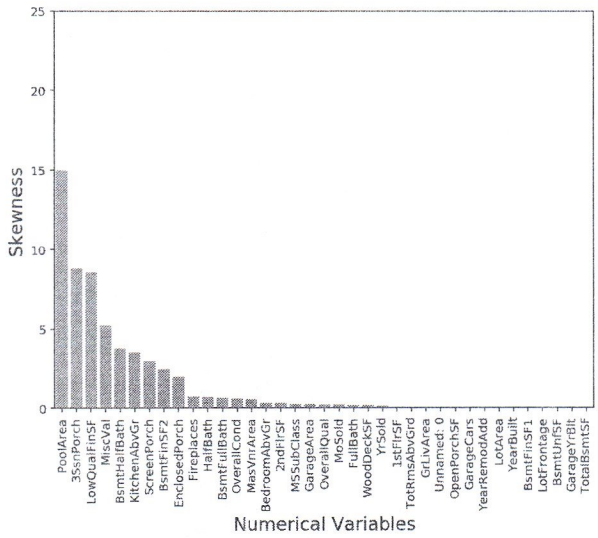
Let's apply the logarithmic transformation to all the variables with a skewness above a certain threshold (0.75). Then, replot the skewness of attributes. Note that to have a fair comparison the two plots should have the same scale.

In [22]:
```
skewed_feats = skewed_feats[skewed_feats > 0.75]
all_data[skewed_feats.index] = np.log1p(all_data[skewed_feats.index])
```

In [23]:
```
# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))
skewness_new = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.data})
# select the variables with a skewness above a certain threshold

skewness_new = skewness_new.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness_new['Variable'], y=skewness_new['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3: FutureWarning: Series.data is deprecated and will be removed in
a future version
  This is separate from the ipykernel package so we can avoid doing imports until
```

Out[23]:
```
Text(0.5, 1.0, '')
```

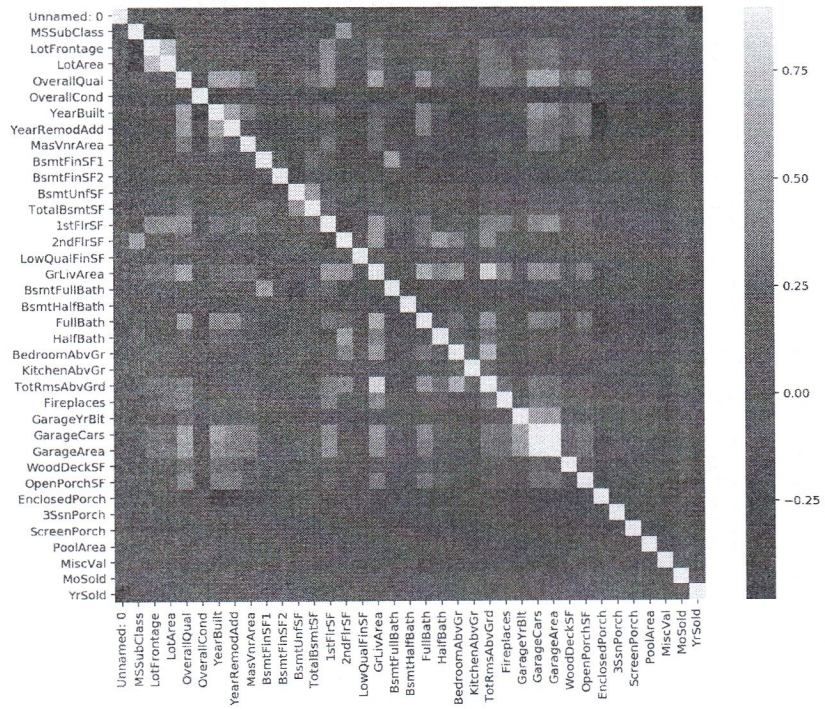


Correlation Analysis

We can finally perform some correlation analysis.

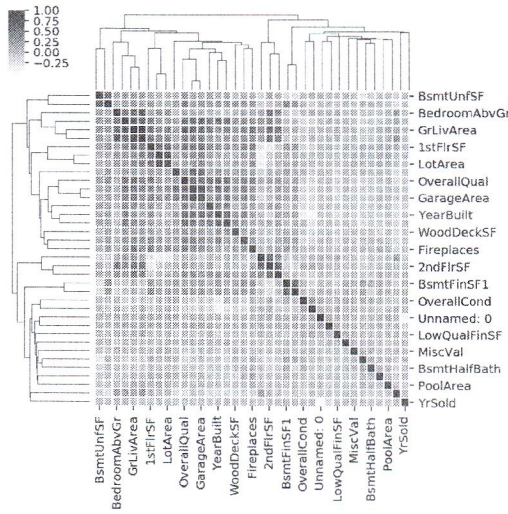
In [25]:
```
corrmat = all_data.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x1a181b0128>



In [26]:
```
import seaborn as sns
sns.clustermap(all_data.corr(), square=True, annot=False, cmap="Blues",
               linewidths=.75, figsize=(6, 6))
```

Out[26]: <seaborn.matrix.ClusterGrid at 0x1a1863bbe0>



Or we can further analyze the distribution of some variables.

One Hot Encoding

We now generate the one hot encoding for all the categorical variables. Pandas has the function **get_dummies** that generates the binary variables for all the categorical variables

In [28]:
```
print("Number of Variables before OHE: "+str(all_data.shape[1]))
```

```
Number of Variables before OHE: 79
```

In [29]:
```
all_data = pd.get_dummies(all_data)
```

In [30]:
```
print("Number of Variables after OHE: "+str(all_data.shape[1]))
```

```
Number of Variables after OHE: 301
```

Saving the Preprocessed Data

We create the matrices to be used for computing the models and also save the cleaned data so that we can avoid repeating the process.

In [41]:
```
all_data.to_csv('HousePricesInputVariablesCleaned.csv')
```

House Prices: Advanced Regression Techniques (Part 1 - Exploration)

In this notebook we apply linear regression to some data from a Kaggle. The notebook is divided into two sections. First we perform some in-depth data exploration and pre-processing, next we build the actual models.

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

This notebook is derived from other existing notebooks by other authors, like,

- https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard

First we import some of the libraries we will need:

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import skew
from scipy.stats import norm
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

In [2]:
```
all_data = pd.read_csv("HousePricesInputVariables.csv")
```

In [3]:
```
all_data.head()
```

Out[3]:

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	...	ScreenPorch	PoolArea	PoolQC	Fer
0	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	...	0	0	NaN	N
1	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	...	0	0	NaN	N
2	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	0	NaN	N
3	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corner	...	0	0	NaN	N
4	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FR2	...	0	0	NaN	N

5 rows × 79 columns

Missing Values

Let's check how many missing values are in the data set and how can we deal with them.

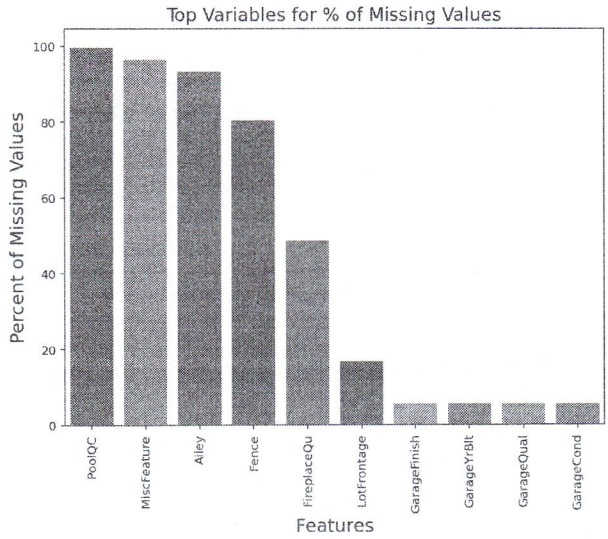
In [4]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Out[4]:

	Missing Ratio
PoolQC	99.657417
MiscFeature	96.402878
Alley	93.216855
Fence	80.438506
FireplaceQu	48.646797
LotFrontage	16.649538
GarageFinish	5.447071
GarageYrBlt	5.447071
GarageQual	5.447071
GarageCond	5.447071

In [5]:
```
f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index[:10], y=all_data_na[:10])
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of Missing Values', fontsize=15)
plt.title('Top Variables for % of Missing Values', fontsize=15)
```

Out[5]:
```
Text(0.5, 1.0, 'Top Variables for % of Missing Values')
```



We note that some of the attributes have the vast majority of the values set to unknown. If we apply imputation without any knowledge about the meaning of what a missing value represent, we would end up with attributes set almost completely to the same value. So they would be almost useless.

How can we deal with all these missing values? First, we need to ask the domain expert whether some missing values have a special meaning. We actually don't have a domain expert but we have the data description in which we find out that

- Alley: Type of alley access to property, NA means "No alley access"
- BsmtCond: Evaluates the general condition of the basement, NA means "No Basement"
- BsmtExposure: Refers to walkout or garden level walls, NA means "No Basement"
- BsmtFinType1: Rating of basement finished area, NA means "No Basement"
- BsmtFinType2: Rating of basement finished area (if multiple types), NA means "No Basement"
- FireplaceQu: Fireplace quality, NA means "No Fireplace"
- Functional: data description says NA means typical
- GarageType: Garage location, NA means "No Garage"
- GarageFinish: Interior finish of the garage, NA means "No Garage"
- GarageQual: Garage quality, NA means "No Garage"
- GarageCond: Garage condition, NA means "No Garage"
- PoolQC: Pool quality, NA means "No Pool"
- Fence: Fence quality, NA means "No Fence"
- MiscFeature: Miscellaneous feature not covered in other categories, NA means "None"

Accordingly, we need to keep this information into account when dealing with the missing values of these variables.

Impute Categorical Values with Known Meaning

Let's impute the missing values for these attributes

In [6]:
```
all_data["Alley"] = all_data["Alley"].fillna("None")

# for all basement features a missing value means that there is no basement
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')

all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")

all_data["Functional"] = all_data["Functional"].fillna("Typ")

for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

all_data["PoolQC"] = all_data["PoolQC"].fillna("None")

all_data["Fence"] = all_data["Fence"].fillna("None")

all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
```

We can also deal with other numerical variables and use some heuristic to impute them. For instance, for **LotFrontage**, since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can fill in missing values by the median LotFrontage of the neighborhood.

In [7]:
```
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))
```

We can also set the garage year, area and number of cars to zero for missing values since this means that there is no garage.

In [8]:
```
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
```

And we can do the same for basement measures.

In [9]:
```
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)
```

When **MasVnrArea** and **MasVnrType** are missing, it most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

In [10]:
```
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

For **MSZoning** (the general zoning classification), 'RL' is by far the most common value. So we can fill in missing values with 'RL'

In [11]:
```
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
```

Utilities has all records are "AllPub", except for one "NoSeWa" and 2 missing values. Since the house with 'NoSewa' is in the training set, this feature won't help to predict labels in the test set. We can then safely remove it.

In [12]:
```
all_data = all_data.drop(['Utilities'], axis=1)
```

KitchenQual has only one missing value, and same as Electrical, we set it to the most frequent values (that is 'TA')

In [13]:
```
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])
```

Exterior1st and **Exterior2nd** have only one missing value. We will use the mode (the most common value)

In [14]:
```
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])
```

For **SaleType** we can use the most frequent value (the mode) which correspond to "WD"

In [15]:
```
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
```

For **MSSubClass** a missing value most likely means No building class. We can replace missing values with None

In [16]:
```
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

Electrical has one missing value. Since this feature has mostly 'SBrkr', we can set that for the missing value.

In [17]:
```
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])
```

Anymore missing?

In [18]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Out[18]:

	Missing Ratio

No more missing values! What would have happened if we did not use or did not have the data description?

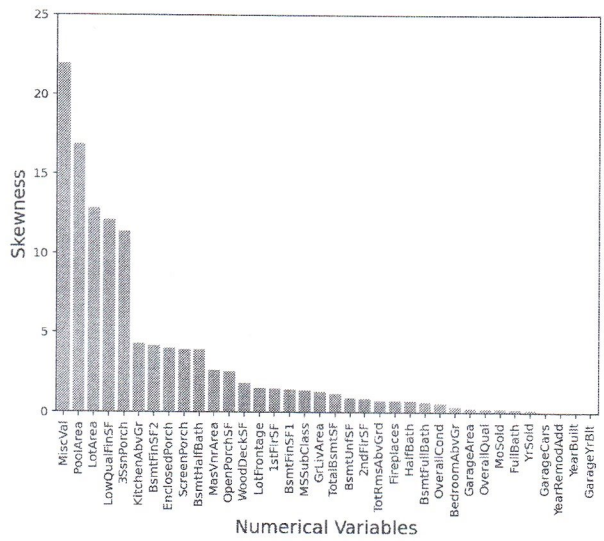
Distribution of Numerical Variables

We now explore the distribution of numerical variables. As we did for the class, we will apply the log1p function to all the skewed numerical variables.

In [20]:
```
# take the numerical features
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))

skewness = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.values})
# select the variables with a skewness above a certain threshold
```

In [21]:
```
skewness = skewness.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness['Variable'], y=skewness['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

Out[21]: Text(0.5, 1.0, '')

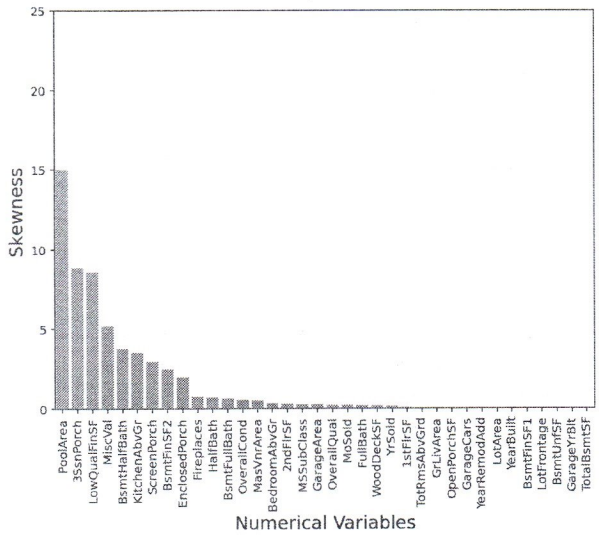


Let's apply the logarithmic transformation to all the variables with a skewness above a certain threshold (0.75). Then, replot the skewness of attributes. Note that to have a fair comparison the two plots should have the same scale.

In [22]:
```
skewed_feats = skewed_feats[skewed_feats > 0.75]
all_data[skewed_feats.index] = np.log1p(all_data[skewed_feats.index])
```

In [24]:
```
# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))
skewness_new = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.values})
# select the variables with a skewness above a certain threshold

skewness_new = skewness_new.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness_new['Variable'], y=skewness_new['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

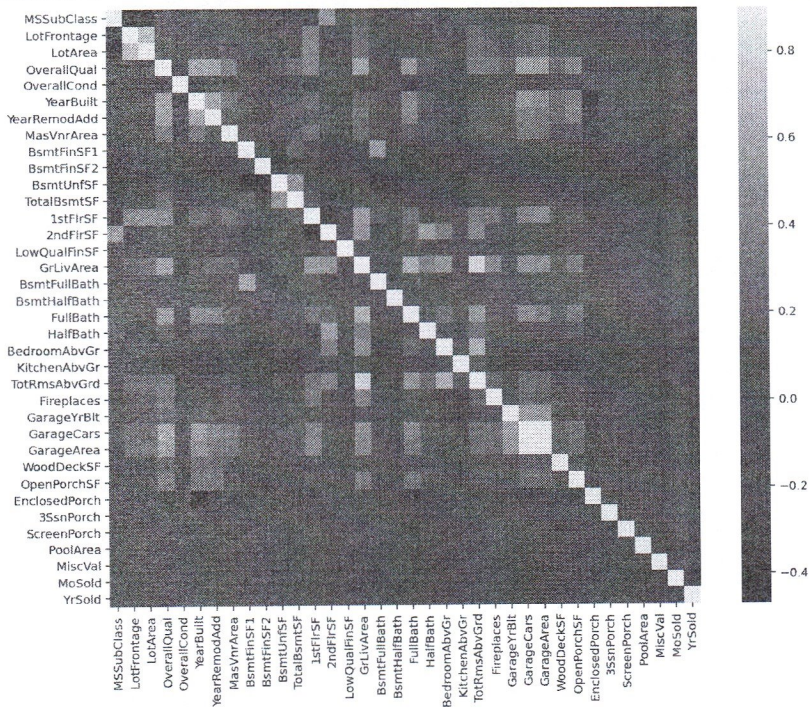
Out[24]:
```
Text(0.5, 1.0, '')
```



Correlation Analysis

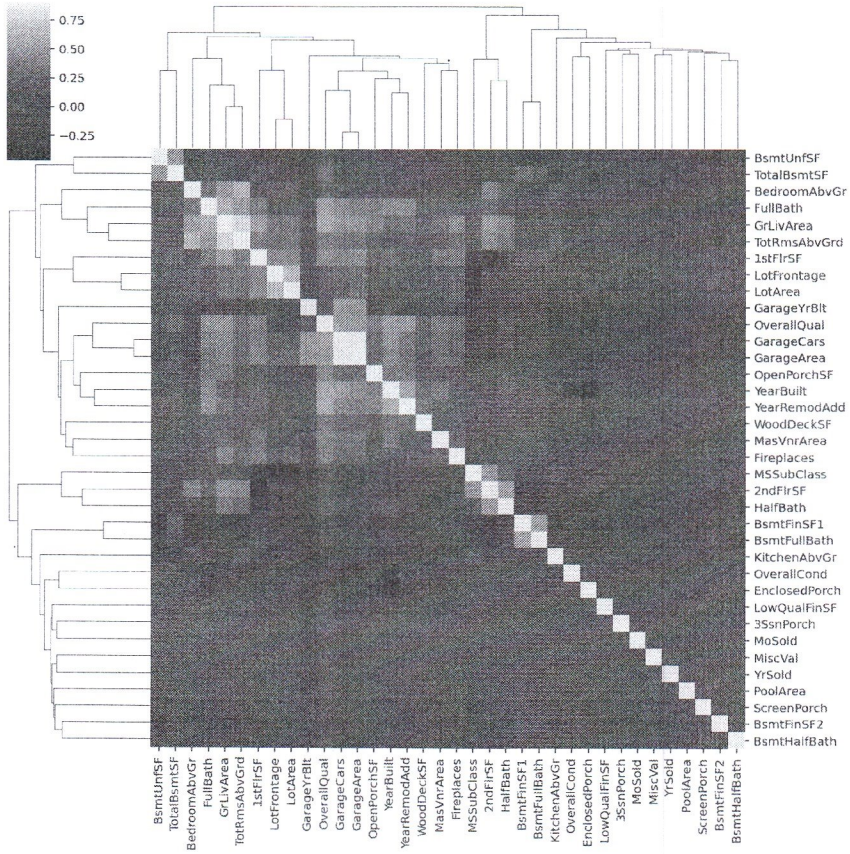
We can finally perform some correlation analysis.

In [25]:
```
corrmat = all_data.corr()
plt.figure(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True);
```



In [26]:
```
import seaborn as sns
plt.figure(figsize=(12,9))
sns.clustermap(corrmat, vmax=0.9, square=True);
```

```
/Users/pierlucalanzi/opt/anaconda3/lib/python3.8/site-packages/seaborn/matrix.py:1201: UserWarning: ``square=True`` ignored in clustermap
  warnings.warn(msg)
<Figure size 864x648 with 0 Axes>
```



Or we can further analyze the distribution of some variables.

One Hot Encoding

We now generate the one hot encoding for all the categorical variables. Pandas has the function **get_dummies** that generates the binary variables for all the categorical variables

In [ ]:
```
print("Number of Variables before OHE: "+str(all_data.shape[1]))
```

In [ ]:
```
all_data = pd.get_dummies(all_data)
```

In [ ]:
```
print("Number of Variables after OHE: "+str(all_data.shape[1]))
```

Saving the Preprocessed Data

We create the matrices to be used for computing the models and also save the cleaned data so that we can avoid repeating the process.

In [ ]:
```
all_data.to_csv('HousePricesInputVariablesCleaned.csv')
```

In [ ]:
```
```

House Prices: Advanced Regression Techniques (Part 2)

We now move to the actual model building part. As usual, we first import some of the libraries we will be needing.

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import skew
from scipy.stats import norm
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

Read the Cleaned Data

First we load the data we cleaned in the previous part.

In [2]:
```
X_train_complete = pd.read_csv("HousePricesTrainClean.csv")
X_test = pd.read_csv("HousePricesTestClean.csv")
X_train_complete.head()
```

Out[2]:

	Unnamed: 0	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	...	SaleType_New
0	0	4.110874	4.189655	9.042040	7	5	2003	2003	5.283204	6.561031	...	0
1	1	3.044522	4.394449	9.169623	6	8	1976	1976	0.000000	6.886532	...	0
2	2	4.110874	4.234107	9.328212	7	5	2001	2002	5.093750	6.188264	...	0
3	3	4.262680	4.110874	9.164401	7	5	1915	1970	0.000000	5.379897	...	0
4	4	4.110874	4.442651	9.565284	8	5	2000	2000	5.860786	6.486161	...	0

5 rows × 302 columns

In [3]:
```
X_train = X_train_complete.loc[:,'MSSubClass':'SaleCondition_Partial']
y = X_train_complete['SalePrice']
```

Linear Regression Models

Now we are going to use plain linear regressio and regularized models (both l_1 Lasso and l_2 Ridge) from the scikit learn module. I'll also define a function that returns the cross-validation rmse error so we can evaluate our models and pick the best tuning par

In [4]:
```
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, ElasticNet, Lasso, LassoCV
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error


def r2_cv(model, X_train, y, random_state=12345678):
    r2= cross_val_score(model, X_train, y, scoring="r2", cv =KFold(10, shuffle=True, random_state=random_state))
    return(r2)

def rmse_cv(model, X_train, y, random_state=12345678):
    rmse= np.sqrt(-cross_val_score(model, X_train, y, scoring="neg_mean_squared_error", cv =KFold(10, shuffle=True, random_stat
    return(rmse)
```

In [5]:
```
model_simple = LinearRegression()
model_simple.fit(X_train, y)
yp = model_simple.predict(X_train)

# compute R2 for train and using crossvalidation
r2_simple_train = r2_score(y,yp)
r2_xval_simple =  r2_cv(model_simple, X_train, y)

# compute RMSE for train and using crossvalidation
rmse_simple_train = mean_squared_error(y,yp,multioutput='raw_values')
rmse_xval_simple =  rmse_cv(model_simple, X_train, y)

print("Linear Regression")
print("=================================================")
print("\t                  Train R2=%.3f"%(r2_simple_train))
print("\t10-fold Crossvalidation R2=%.3f"%(r2_xval_simple.mean()))
print("\t                  Train RMSE=%.3f"%(rmse_simple_train))
print("\t10-fold Crossvalidation RMSE=%.3f"%(rmse_xval_simple.mean()))
```

```
Linear Regression
=================================================
                  Train R2=0.947
        10-fold Crossvalidation R2=0.720
                  Train RMSE=0.008
        10-fold Crossvalidation RMSE=0.171
```

Ridge Regression

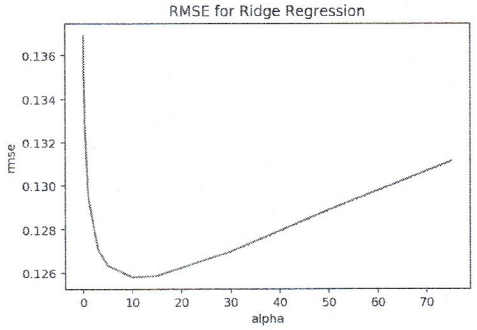
We now try ridge regression (l_2) and for this purpose we need to select a value of \alpha. The higher the \alpha, the higher the penalization of weights with a large absolute value. We select a possible range of values of \alpha and apply ridge for each value.

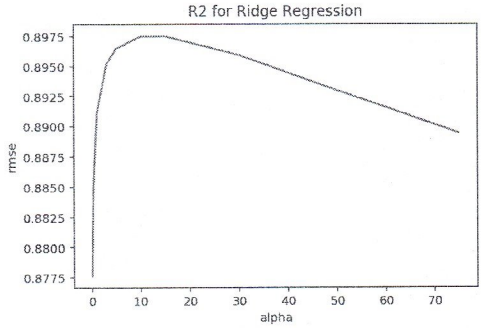
In [6]:
```
alphas = [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 50, 75]
cv_ridge = [rmse_cv(Ridge(alpha = alpha), X_train, y).mean() for alpha in alphas]
```

```
cv_r2_ridge = [r2_cv(Ridge(alpha = alpha), X_train, y).mean() for alpha in alphas]
```

Put the values into a table and plot them

In [7]:
```
cv_ridge = pd.Series(cv_ridge, index = alphas)
cv_r2_ridge = pd.Series(cv_r2_ridge, index = alphas)

cv_ridge.plot(title = "RMSE for Ridge Regression")
plt.xlabel("alpha")
plt.ylabel("rmse")
plt.show()

cv_r2_ridge.plot(title = "R2 for Ridge Regression")
plt.xlabel("alpha")
plt.ylabel("rmse")
plt.show()
```





Note the U-ish shaped curve above. When alpha is too large the regularization is too strong and the model cannot capture all the complexities in the data. If however we let the model be too flexible (alpha small) the model begins to overfit. A value of alpha = 10 is about right based on the plot above.

In [8]:
```
print("Best RMSE %.3f for alpha %.3f"%(cv_ridge.min(),cv_ridge.idxmin()))
print("Best R2 %.3f for alpha %.3f"%(cv_r2_ridge.max(),cv_r2_ridge.idxmax()))
print("Why the difference? R2[10]=%.6f R2[15]=%.6f"%(cv_r2_ridge[cv_ridge.idxmin()],cv_r2_ridge[cv_r2_ridge.idxmax()]))
```

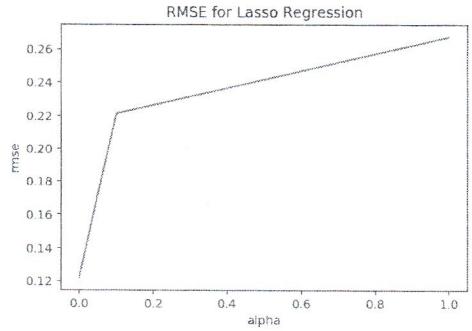
```
Best RMSE 0.126 for alpha 10.000
Best R2 0.898 for alpha 15.000
Why the difference? R2[10]=0.897505 R2[15]=0.897513
```

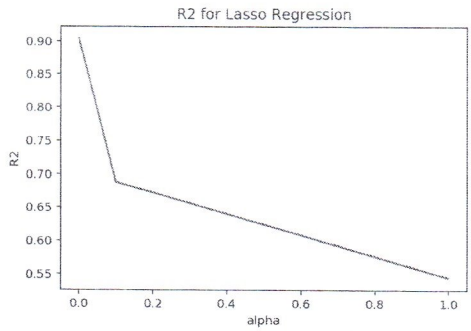
So for the Ridge regression we get a rmse of about 0.126

Lasso Regression (L_1)

We now test Lasso regression. As before we need to test different values of \alpha. Let' try out the Lasso model. For some reason the alphas in Lasso CV are really the inverse or the alphas in Ridge.

In [9]:
```
alphas = [1, 0.1, 0.001, 0.0005]
cv_lasso = [rmse_cv(Lasso(alpha = alpha), X_train, y).mean() for alpha in alphas]
cv_r2_lasso = [r2_cv(Lasso(alpha = alpha), X_train, y).mean() for alpha in alphas]

cv_lasso = pd.Series(cv_lasso, index = alphas)
cv_r2_lasso = pd.Series(cv_r2_lasso, index = alphas)

plt.figure(figsize=(6,4))
cv_lasso.plot(title = "RMSE for Lasso Regression")
plt.xlabel("alpha")
plt.ylabel("rmse")
plt.show()

plt.figure(figsize=(6,4))
cv_r2_lasso.plot(title = "R2 for Lasso Regression")
plt.xlabel("alpha")
plt.ylabel("R2")
plt.show()
```





In [10]:
```
print("Best RMSE %.3f for alpha %.3f"%(cv_lasso.min(),cv_lasso.idxmin()))
print("Best R2 %.3f for alpha %.3f"%(cv_r2_lasso.max(),cv_r2_lasso.idxmax()))
```

```
Best RMSE 0.122 for alpha 0.001
Best R2 0.904 for alpha 0.001
```

Lasso & Ridge with Built-in Crossvalidation

We performed the cross validation explicitly, but sklearn also provides two functions that include crossvalidation as part of the process, namely, LassoCV and RidgeCV.

In [11]:
```
model_ridge = RidgeCV(alphas = [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 50, 75], cv=KFold(10, shuffle=True, random_state=12345678))
model_lasso = LassoCV(alphas = [1, 0.1, 0.001, 0.0005],cv=KFold(10, shuffle=True, random_state=12345678)).fit(X_train, y)
```

We evaluate the resulting model using the same procedure we applied before.

In [12]:
```
rmse_lasso2 = rmse_cv(model_lasso, X_train, y)
rmse_ridge2 = rmse_cv(model_ridge, X_train, y)

r2_lasso2 = r2_cv(model_lasso, X_train, y)
r2_ridge2 = r2_cv(model_ridge, X_train, y)
```

Finally, we compare the results achieved with this procedure with the ones compared with the explicit cross validation.

In [13]:
```
print("Ridge Regression (10-fold crossvalidation)")
print("\tRMSE=%.3f R2=%.3f for Alpha=%.3f"%(rmse_ridge2.mean(), r2_ridge2.mean(), model_ridge.alpha_))
print("\n")
print("Lasso Regression (10-fold crossvalidation)")
print("\tRMSE=%.3f R2=%.3f for Alpha=%.3f"%(rmse_lasso2.mean(), r2_lasso2.mean(), model_lasso.alpha_))
```

```
Ridge Regression (10-fold crossvalidation)
        RMSE=0.126 R2=0.896 for Alpha=15.000


Lasso Regression (10-fold crossvalidation)
        RMSE=0.122 R2=0.904 for Alpha=0.001
```

The results are similar to the ones we obtained with the explicit search for α.

Feature Selection

One nice feature of Lasso regression is that by zeroing out the coefficients of variables that it deems unimportant, it actually performs feature selection. In practise, it simplifies the data while building the model.

In [14]:
```
coef = pd.Series(model_lasso.coef_, index = X_train.columns)
```

In [15]:
```
print("Lasso picked " + str(sum(coef != 0)) + " variables and eliminated the other " +  str(sum(coef == 0)) + " variables")
```

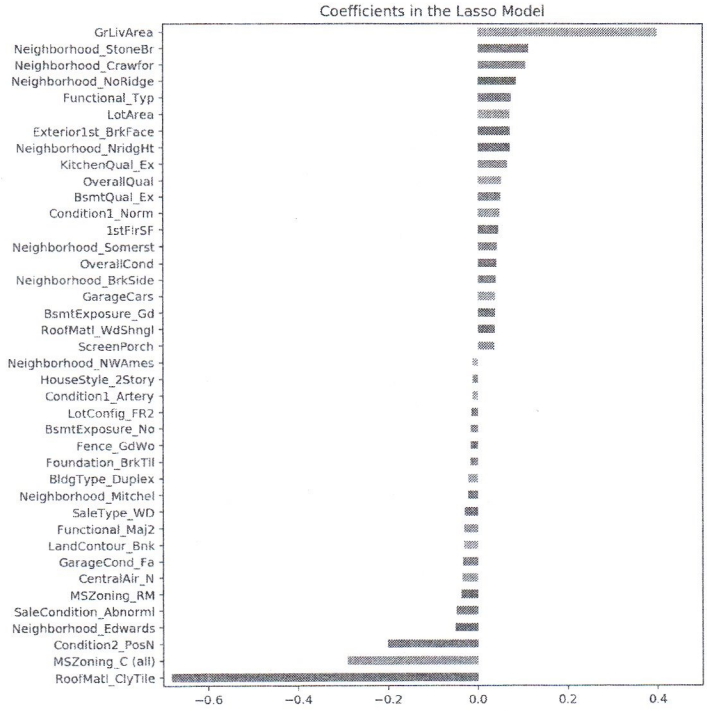
```
Lasso picked 105 variables and eliminated the other 195 variables
```

Note that, the process is stochastic and thus we cannot be sure that the selected variables are exactly the good ones so one approach is to run the procedure several times (using bootstrap which we will discuss later in the course) to check how robust the feature selection is. For instance, we check if there are variables whose weight is always zeroed out. As an example, let's plot the largest and the smallest coefficients.

In [24]:
```
imp_coef = pd.concat([coef.sort_values().head(20),
                      coef.sort_values().tail(20)])
```

In [25]:
```
matplotlib.rcParams['figure.figsize'] = (8.0, 10.0)
imp_coef.plot(kind = "barh")
plt.title("Coefficients in the Lasso Model")
plt.xlim(-0.7,0.5)
```

Out[25]:
```
(-0.7, 0.5)
```

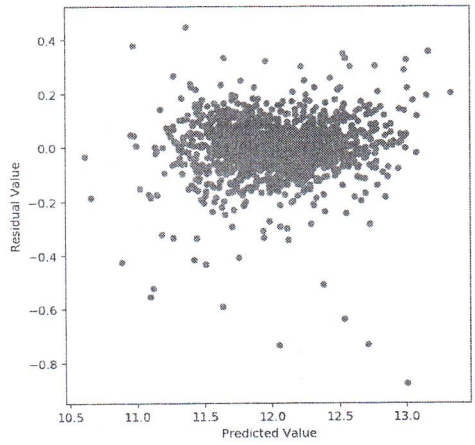


GrLivArea appears to be the most important positive feature (it identifies the above ground area by area square feet so it makes sense). Then a few other location and quality features contributed positively. Some of the negative features make less sense and would be worth looking into more.

Let's also check the residuals, that is the difference between the predicted and the actual value plotted against the predicted value.

In [26]:
```
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

preds = pd.DataFrame({"preds":model_lasso.predict(X_train), "true":y})
preds["residuals"] = preds["true"] - preds["preds"]
preds.plot(x = "preds", y = "residuals",kind = "scatter")
plt.xlabel("Predicted Value")
plt.ylabel("Residual Value")
```

Out[26]:
```
Text(0,0.5,'Residual Value')
```

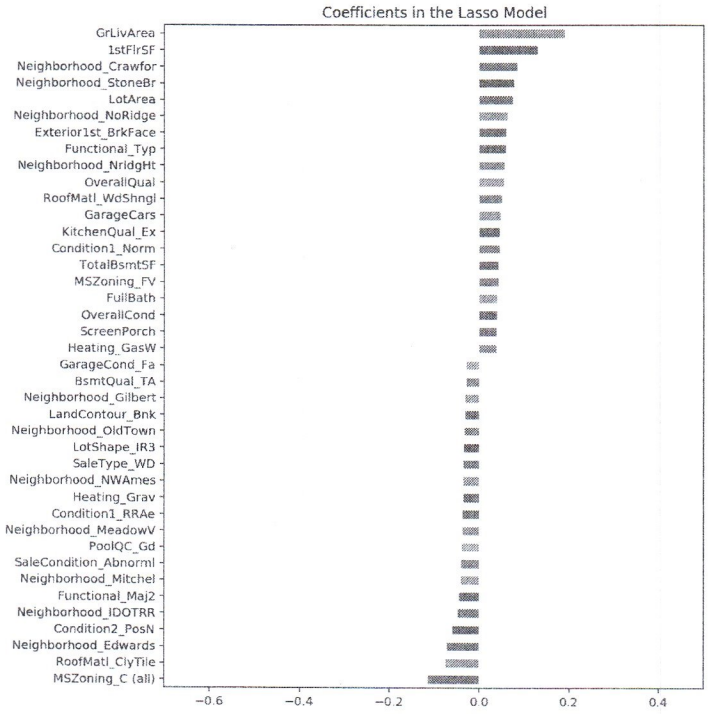


In [27]:
```
ridge_coef = pd.Series(model_ridge.coef_, index = X_train.columns)
print("Ridge picked " + str(sum(ridge_coef != 0)) + " variables and eliminated the other " +  str(sum(ridge_coef == 0)) + " var
```

```
Ridge picked 300 variables and eliminated the other 0 variables
```

In [28]:
```
ridge_imp_coef = pd.concat([ridge_coef.sort_values().head(20),
                     ridge_coef.sort_values().tail(20)])
matplotlib.rcParams['figure.figsize'] = (8.0, 10.0)
ridge_imp_coef.plot(kind = "barh")
plt.title("Coefficients in the Ridge Model")
plt.xlim(-0.7,0.5)
```

Out[28]:
```
(-0.7, 0.5)
```


Coefficients in the Lasso Model
GrLivArea
1stFlrSF
Neighborhood_Crawfor
Neighborhood_StoneBr
LotArea
Neighborhood_NoRidge
Exterior1st_BrkFace
Functional_Typ
Neighborhood_NridgHt
OverallQual
RoofMatl_WdShngl
GarageCars
KitchenQual_Ex
Condition1_Norm
TotalBsmtSF
MSZoning_FV
FullBath
OverallCond
ScreenPorch
Heating_GasW
GarageCond_Fa
BsmtQual_TA
Neighborhood_Gilbert
LandContour_Bnk
Neighborhood_OldTown
LotShape_IR3
SaleType_WD
Neighborhood_NWAmes
Heating_Grav
Condition1_RRAe
Neighborhood_MeadowV
PoolQC_Gd
SaleCondition_Abnorml
Neighborhood_Mitchel
Functional_Maj2
Neighborhood_IDOTRR
Condition2_PosN
Neighborhood_Edwards
RoofMatl_ClyTile
MSZoning_C (all)
−0.6
−0.4
−0.2
0.0
0.2
0.4

Information Gain, Information Gain Ratio, & Gini Index

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

In [2]:
```
df = pd.read_csv('Weather.csv')
```

In [3]:
```
df.describe()
```

Out[3]:

	Outlook	Temperature	Humidity	Windy	Play
count	14	14	14	14	14
unique	3	3	2	2	2
top	rainy	mild	high	False	yes
freq	5	6	7	8	9

In [4]:
```
df.head()
```

Out[4]:

	Outlook	Temperature	Humidity	Windy	Play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes

In [5]:
```
target_variable = 'Play'
input_variables = df.columns[df.columns!=target_variable]
```

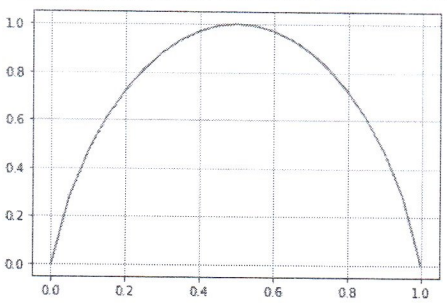
In [6]:
```
def DistributionEntropy(fc):
    """Computes the entropy given a frequency count fc
       represented using an numpy array

        [10, 0] corresponds to a distribution of [1.0, 0.0]
        which corresponds to an entropy of 0.0

        distribution [0.5 0.5] has entropy 1.0
    """

    d = fc/sum(fc)

    return sum([ - x * math.log(x,2) for x in d[d!=0] ])
```

In [7]:
```
entropy = [DistributionEntropy(np.array(fc)) for fc in zip(np.arange(0,1.05,0.05), 1- np.arange(0,1.05,0.05))]
```

In [8]:
```
entropy
```

Out[8]:
```
[0.0,
 0.28639695711595625,
 0.4689955935892812,
 0.6098403047164005,
 0.7219280948873623,
 0.8112781244591328,
 0.8812908992306927,
 0.9340680553754912,
 0.9709505944546686,
 0.9927744539878084,
 1.0,
 0.9927744539878084,
 0.9709505944546687,
 0.934068055375491,
 0.8812908992306925,
 0.8112781244591328,
 0.7219280948873623,
 0.6098403047164002,
 0.46899559358928117,
 0.2863969571159558,
 0.0]
```

In [9]:
```
x = np.arange(0,1.05,0.05)
plt.plot(x,entropy)
plt.grid()
```



In [10]:
```
DistributionEntropy(df['Play'].value_counts())
```

Out[10]:
```
0.9402859586706309
```

In [11]:
```
[x for x in range(5)]
```

Out[11]:
```
[0, 1, 2, 3, 4]
```

In [12]:
```
def AttributeEntropy(attribute,target):
    # distribution corresponding to each attribute value
    # weighted sum

    weights = attribute.value_counts()/sum(attribute.value_counts())
    attribute_values = attribute.value_counts().index

    entropy = 0

    for i,value in enumerate(attribute_values):
        mask = attribute==value
        entropy = entropy + weights[i] * DistributionEntropy(target[mask].value_counts())

    return entropy
```

In [13]:
```
def InformationGain(attribute,target):
    # original entropy
    original_entropy = DistributionEntropy(target.value_counts())

    # attribute entropy
    attribute_entropy = AttributeEntropy(attribute,target)

    # return the difference
    return original_entropy - attribute_entropy
```

In [14]:
```
InformationGain(df['Outlook'],df['Play'])
```

Out[14]:
```
0.2467498197744391
```

In [15]:
```
def IntrinsicInfo(attribute):
    return DistributionEntropy(attribute.value_counts())
```

In [16]:
```
def InformationGainRatio(attribute, target):
    return InformationGain(attribute, target)/IntrinsicInfo(attribute)
```

In [17]:
```
for attribute in input_variables:
    print(attribute + " " + str(InformationGainRatio(df[attribute], df[target_variable])))
```

```
Outlook 0.15642756242117517
Temperature 0.01877264622241867
Humidity 0.15183550136234136
Windy 0.048848615511520595
```

In [18]:
```
IntrinsicInfo(df['Outlook'])
```

Out[18]:
```
1.577406282852345
```

In [19]:
```
InformationGain(df['Outlook'], df['Play'])
```

Out[19]:
```
0.2467498197744391
```

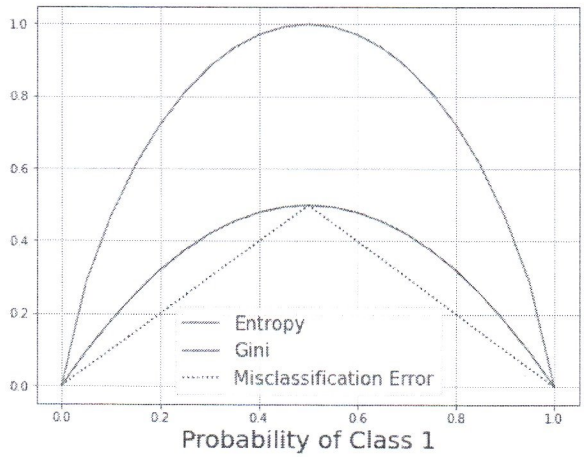
In [20]:
```
def GiniIndex(fc):
    d = np.array(fc/sum(fc))
    return 1 - np.dot(d,d)
```

In [21]:
```
def AttributeGiniIndex(attribute,target):
    # distribution corresponding to each attribute value
    # weighted sum

    weights = attribute.value_counts()/sum(attribute.value_counts())
    attribute_values = attribute.value_counts().index

    gini = 0

    for i,value in enumerate(attribute_values):
        mask = attribute==value
        gini = gini + weights[i] * GiniIndex(target[mask].value_counts())

    return gini
```

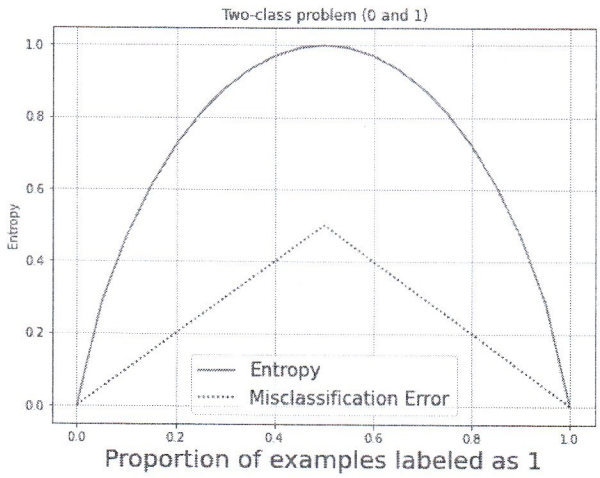
In [22]:
```
gini = [GiniIndex(np.array(fc)) for fc in zip(np.arange(0,1.05,0.05), 1- np.arange(0,1.05,0.05))]
```

In [23]:
```
error = [min(x) for x in zip(np.arange(0,1.05,0.05), 1- np.arange(0,1.05,0.05))]
```

In [24]:
```
x = np.arange(0,1.05,0.05)
plt.figure(figsize=(8,6))
plt.plot(x,entropy,label='Entropy')
plt.plot(x,gini,label='Gini',c='Green')
plt.plot(x,error,label='Misclassification Error',c='Black',ls=':')
plt.legend(fontsize=15)
plt.xlabel('Probability of Class 1',fontsize=20)
plt.grid()
plt.savefig("EntropyGiniComparison.png")
```



In [25]:
```
x = np.arange(0,1.05,0.05)
plt.figure(figsize=(8,6))
plt.plot(x,entropy,label='Entropy')
# plt.plot(x,gini,label='Gini',c='Green')
plt.plot(x,error,label='Misclassification Error',c='Black',ls=':')
plt.legend(fontsize=15)
plt.xlabel('Proportion of examples labeled as 1',fontsize=20)
plt.ylabel('Entropy')
plt.title("Two-class problem (0 and 1)")
plt.grid()
plt.savefig("EntropyComparison.png")
```



In [26]:
```
label = ['+','-','-','-','+','+','-','-','+','+']
a = ['0','0','0','0','0','1','1','1','1','1']
b = ['0','0','1','1','0','0','0','0','1','0']
c = ['0','1','1','1','1','1','1','1','1','1']

df_abc = pd.DataFrame({'a':a,'b':b,'c':c,'class':label})

df_abc.to_csv('abc_dataset.csv',index=False)
```

In [27]:
```
InformationGain(df_abc['c'],df_abc['class'])
```

Out[27]: 0.10803154614559995

In [28]:
```
DistributionEntropy(df_abc['class'].value_counts())
```

Out[28]: 1.0

In [29]:
```
AttributeEntropy(df_abc['a'],df_abc['class'])
```

Out[29]: 0.9709505944546686

In [30]:
```
InformationGain(df_abc['a'],df_abc['class'])
```

Out[30]: 0.02904940554533142

In [31]:
```
InformationGain(df_abc['b'],df_abc['class'])
```

Out[31]: 0.034851554559677034

In [32]:
```
InformationGain(df_abc['c'],df_abc['class'])
```

Out[32]: 0.10803154614559995

In [33]:
```
AttributeGiniIndex(df['Windy'],df['Play'])
```

Out[33]: 0.42857142857142855

In [34]:
```
AttributeGiniIndex(df_abc['a'],df_abc['class'])
```

Out[34]: 0.48

In [35]:
```
AttributeGiniIndex(df_abc['b'],df_abc['class'])
```

Out[35]: 0.4761904761904763

In [36]:
```
AttributeGiniIndex(df_abc['c'],df_abc['class'])
```

Out[36]: 0.4444444444444444

In [37]:
```
a=[0.22,0.31,0.31,0.31,0.33,0.41,0.43]
label=['0','0','1','0','1','1','1']

single_continuous_dataset = pd.DataFrame({'A':a,'Class':label})
```

In [38]:
```
np.sort(single_continuous_dataset['A'].unique())
```

Out[38]: array([0.22, 0.31, 0.33, 0.41, 0.43])

In [41]:
```
def ContinuousAttributeGini(attribute,target):
    # distribution corresponding to each attribute value
    # weighted sum

    original_gini = GiniIndex(target.value_counts())


    values = np.sort(attribute.unique())

    print(values)
    # number of rows
    n = attribute.shape[0]

    # number of values
    nv = values.shape[0]


    gini_value = np.zeros(nv)
    gini_split = np.zeros(nv)


    for i,value in enumerate(values):
        left_mask = attribute<=value
        right_mask = ~left_mask

        print("<<",target[left_mask].value_counts(),">>")
        print("<<",target[right_mask].value_counts(),">>")


        left_gini = GiniIndex(target[left_mask].value_counts())
        right_gini = GiniIndex(target[right_mask].value_counts())

        left_weight = sum(left_mask)/n
        right_weight = sum(right_mask)/n

        gini_index = 1 - left_weight*left_gini - right_weight*right_gini

        gini_value[i] = original_gini - gini_index
        if (i==(nv-1)):
            gini_split[i] = value
        else:
            gini_split[i] = (value+values[i+1])/2.0

        print("%d split %.3f gini %.3f"%(i,gini_split[i],gini_value[i]))
    best = np.argmax(gini_value)

    return gini_split[best],gini_value[best]
```

In [42]:
```
ContinuousAttributeGini(single_continuous_dataset['A'],single_continuous_dataset['Class'])
```

```
[0.22 0.31 0.33 0.41 0.43]
<< 0    1
Name: Class, dtype: int64 >>
<< 1    4
0    2
Name: Class, dtype: int64 >>
0 split 0.265 gini -0.129
<< 0    3
1    1
Name: Class, dtype: int64 >>
<< 1    3
Name: Class, dtype: int64 >>
1 split 0.320 gini -0.296
<< 0    3
1    2
Name: Class, dtype: int64 >>
<< 1    2
Name: Class, dtype: int64 >>
2 split 0.370 gini -0.167
<< 0    3
1    3
Name: Class, dtype: int64 >>
<< 1    1
Name: Class, dtype: int64 >>
3 split 0.420 gini -0.082
<< 1    4
0    3
Name: Class, dtype: int64 >>
<< Series([], Name: Class, dtype: int64) >>
4 split 0.430 gini -0.020
```

Out[42]: (0.43, -0.020408163265305923)

In [43]:
```
test_a=np.arange(0,1,0.1)
test_label=['0','0','0','0','0','1','1','1','1','1']

test = pd.DataFrame({'A':test_a,'Class':test_label})
```

In [ ]:
```
test
```

In [ ]:
```
ContinuousAttributeGini(test['A'],test['Class'])
```

In [ ]:
```
print(test_label)
```

In [43]:
```
GiniIndex(np.array([6,3]))
```

Out[43]:
```
0.4444444444444444
```

In [45]:
```
9/14*GiniIndex(np.array([6,3]))+5/14*GiniIndex(np.array([3,2]))
```

Out[45]:
```
0.45714285714285713
```

In [ ]:

K-Nearest Neighbor - Iris Dataset

k-Nearest neighbors classifier assigns the class of an example using the majority vote of the k most examples in the data. In this example, we apply k-nearest neighbor using the well-known Iris dataset.

First we load all the needed libraries.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, datasets, neighbors
from sklearn import model_selection
from sklearn.model_selection import StratifiedKFold
from matplotlib.colors import ListedColormap
%matplotlib inline
```

Next, we load the dataset that is included in the Scikit-Learn dataset module.

In [7]:
```
# import some data to play with
iris = datasets.load_iris()
target = np.array(iris.target)

print("Number of examples: ", iris.data.shape[0])
print("Number of variables:", iris.data.shape[0])
print("Variable names:     ", iris.feature_names)
print("Target values:      ", iris.target_names)
print("Class Distribution  ", [(x,sum(target==x)) for x in np.unique(target)])
```

```
Number of examples:  150
Number of variables: 150
Variable names:      ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target values:       ['setosa' 'versicolor' 'virginica']
Class Distribution   [(0, 50), (1, 50), (2, 50)]
```

We will use only the first two variables (sepal length and sepal width)

In [8]:
```
x = iris.data[:, :2]
y = iris.target
```

First we just evaluate the performance for k=15

In [9]:
```
k = 15
knn = neighbors.KNeighborsClassifier(n_neighbors=k)
knn_eval = model_selection.cross_val_score(knn, x, y, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=1234))

print("%d-nearest-neighbor   Accuracy=%.3f Std=%.3f"%(k,np.average(knn_eval),np.std(knn_eval)))
```

```
15-nearest-neighbor   Accuracy=0.773 Std=0.095
```

Next, we perform an experiment to select the best k. For this purpose, we use the typical train-validation-test setup in which train-validation part is performed using cross validation and the final evaluation is done with the test set that was never used for selecting k.

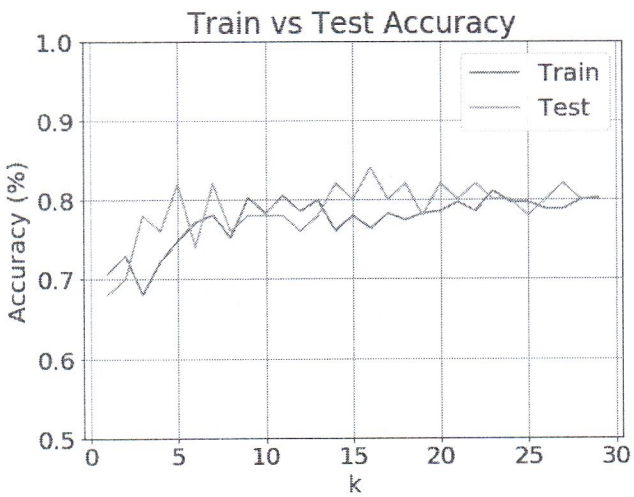
In [10]:
```
x_train, x_test, y_train, y_test = model_selection.train_test_split(x, y, test_size=0.33, random_state=1234, stratify=y)

knn_accuracy = {}
knn_std = {}

best_k = -1
best_accuracy = 0.0

best_test_k = -1
best_test_accuracy = 0.0

plt_x_label = []
plt_y_train = []
plt_y_bar_train = []
plt_y_test = []

for k in np.arange(1,30,1):
    knn = neighbors.KNeighborsClassifier(n_neighbors=k)
    knn_eval = model_selection.cross_val_score(knn, x_train, y_train, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=
    knn_accuracy[k] = np.average(knn_eval)
    knn_std[k] = np.std(knn_eval)

    knn_model = neighbors.KNeighborsClassifier(n_neighbors=k)
    knn_model = knn_model.fit(x_train, y_train)
    knn_model_eval = knn_model.score(x_test, y_test)

    print("k-nn k=%-3d Train %.3f +/- %.3f\tTest %.3f"%(k,np.average(knn_eval),np.std(knn_eval),np.average(knn_model_eval)))


    if np.average(knn_eval)>best_accuracy:
        best_accuracy = np.average(knn_eval)
        best_k = k

    if np.average(knn_model_eval)>best_test_accuracy:
        best_test_accuracy = np.average(knn_model_eval)
        best_test_k = k

    plt_x_label = plt_x_label + [k]
    plt_y_train = plt_y_train + [np.average(knn_eval)]
    plt_y_test = plt_y_test + [np.average(knn_model_eval)]
    plt_y_bar_train = plt_y_bar_train + [np.std(knn_eval)]

print("\n\nBest k=%d Accuracy on Train %.3f"%(best_k,best_accuracy))
print("Best k=%d Accuracy on Test  %.3f"%(best_test_k,best_test_accuracy))
```

```
k-nn k=1   Train 0.706 +/- 0.076        Test 0.680
k-nn k=2   Train 0.729 +/- 0.077        Test 0.700
k-nn k=3   Train 0.681 +/- 0.076        Test 0.780
k-nn k=4   Train 0.721 +/- 0.079        Test 0.760
k-nn k=5   Train 0.748 +/- 0.063        Test 0.820
k-nn k=6   Train 0.771 +/- 0.124        Test 0.740
k-nn k=7   Train 0.781 +/- 0.102        Test 0.820
k-nn k=8   Train 0.752 +/- 0.104        Test 0.760
k-nn k=9   Train 0.802 +/- 0.079        Test 0.780
```

```
k-nn k=10  Train 0.783 +/- 0.073        Test 0.780
k-nn k=11  Train 0.805 +/- 0.084        Test 0.780
k-nn k=12  Train 0.786 +/- 0.112        Test 0.760
k-nn k=13  Train 0.799 +/- 0.094        Test 0.780
k-nn k=14  Train 0.761 +/- 0.100        Test 0.820
k-nn k=15  Train 0.780 +/- 0.096        Test 0.800
k-nn k=16  Train 0.763 +/- 0.100        Test 0.840
k-nn k=17  Train 0.783 +/- 0.101        Test 0.800
k-nn k=18  Train 0.774 +/- 0.113        Test 0.820
k-nn k=19  Train 0.783 +/- 0.101        Test 0.780
k-nn k=20  Train 0.786 +/- 0.118        Test 0.820
k-nn k=21  Train 0.797 +/- 0.100        Test 0.800
k-nn k=22  Train 0.786 +/- 0.130        Test 0.820
k-nn k=23  Train 0.811 +/- 0.086        Test 0.800
k-nn k=24  Train 0.797 +/- 0.110        Test 0.800
k-nn k=25  Train 0.797 +/- 0.097        Test 0.780
k-nn k=26  Train 0.788 +/- 0.104        Test 0.800
k-nn k=27  Train 0.788 +/- 0.104        Test 0.820
k-nn k=28  Train 0.799 +/- 0.108        Test 0.800
k-nn k=29  Train 0.802 +/- 0.106        Test 0.800


Best k=23 Accuracy on Train 0.811
Best k=16 Accuracy on Test  0.840
```

In [12]:
```
plt.figure(1, figsize=(8, 6))
font = {'family' : 'sans', 'size'   : 20}
plt.rc('font', **font)
plt.title('Train vs Test Accuracy')
plt.plot(plt_x_label,plt_y_train,label='Train')
plt.plot(plt_x_label,plt_y_test,label='Test')
plt.ylim([0.5,1.0])
plt.legend()
plt.grid(color='grey')
plt.xlabel('k')
plt.ylabel('Accuracy (%)')
```

Out[12]: Text(0,0.5,'Accuracy (%)')



We repeat the same procedure using distance as the weight function for prediction, thus the class of more similar examples will weight more.

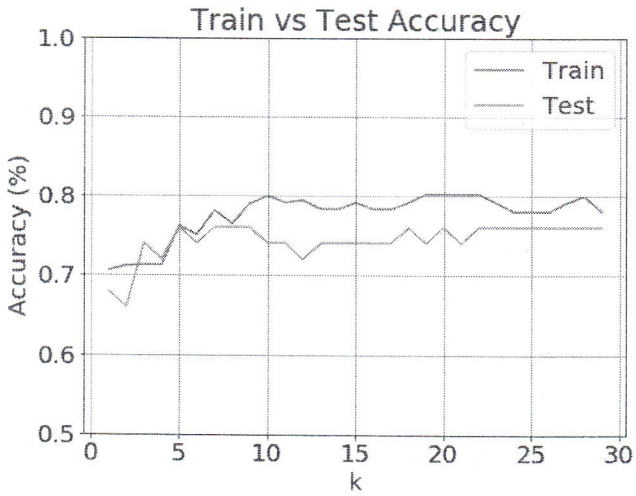
In [13]:
```
best_k = -1
best_accuracy = 0.0

best_test_k = -1
best_test_accuracy = 0.0

plt_x_label = []
plt_y_train = []
plt_y_bar_train = []
plt_y_test = []

for k in np.arange(1,30,1):
    knn = neighbors.KNeighborsClassifier(n_neighbors=k,weights='distance')
    knn_eval = model_selection.cross_val_score(knn, x_train, y_train, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=
    #print("k-nn k=%3d Accuracy=%.3f Std=%.3f"%(k,np.average(knn_eval),np.std(knn_eval)))
    knn_accuracy[k] = np.average(knn_eval)
    knn_std[k] = np.std(knn_eval)

    knn_model = neighbors.KNeighborsClassifier(n_neighbors=k,weights='distance')
    knn_model = knn_model.fit(x_train, y_train)
    knn_model_eval = knn_model.score(x_test, y_test)

    print("k-nn k=%3d Train %.3f +/- %.3f\tTest %.3f"%(k,np.average(knn_eval),np.std(knn_eval),np.average(knn_model_eval)))


    if np.average(knn_eval)>best_accuracy:
        best_accuracy = np.average(knn_eval)
        best_k = k

    if np.average(knn_model_eval)>best_test_accuracy:
        best_test_accuracy = np.average(knn_model_eval)
        best_test_k = k

    plt_x_label = plt_x_label + [k]
    plt_y_train = plt_y_train + [np.average(knn_eval)]
    plt_y_test = plt_y_test + [np.average(knn_model_eval)]
    plt_y_bar_train = plt_y_bar_train + [np.std(knn_eval)]

print("\n\nBest k=%d Accuracy on Train %.3f"%(best_k,best_accuracy))
print("Best k=%d Accuracy on Test  %.3f"%(best_test_k,best_test_accuracy))
```

```
k-nn k=  1 Train 0.706 +/- 0.076        Test 0.680
```

```
k-nn k=  2 Train 0.712 +/- 0.050        Test 0.660
k-nn k=  3 Train 0.713 +/- 0.063        Test 0.740
k-nn k=  4 Train 0.713 +/- 0.063        Test 0.720
k-nn k=  5 Train 0.762 +/- 0.067        Test 0.760
k-nn k=  6 Train 0.751 +/- 0.072        Test 0.740
k-nn k=  7 Train 0.781 +/- 0.078        Test 0.760
k-nn k=  8 Train 0.764 +/- 0.091        Test 0.760
k-nn k=  9 Train 0.789 +/- 0.087        Test 0.760
k-nn k= 10 Train 0.799 +/- 0.063        Test 0.740
k-nn k= 11 Train 0.791 +/- 0.074        Test 0.740
k-nn k= 12 Train 0.794 +/- 0.080        Test 0.720
k-nn k= 13 Train 0.783 +/- 0.073        Test 0.740
k-nn k= 14 Train 0.783 +/- 0.073        Test 0.740
k-nn k= 15 Train 0.791 +/- 0.074        Test 0.740
k-nn k= 16 Train 0.783 +/- 0.073        Test 0.740
k-nn k= 17 Train 0.783 +/- 0.073        Test 0.740
k-nn k= 18 Train 0.791 +/- 0.074        Test 0.760
k-nn k= 19 Train 0.802 +/- 0.079        Test 0.740
k-nn k= 20 Train 0.802 +/- 0.079        Test 0.760
k-nn k= 21 Train 0.802 +/- 0.079        Test 0.740
k-nn k= 22 Train 0.802 +/- 0.079        Test 0.760
k-nn k= 23 Train 0.791 +/- 0.074        Test 0.760
k-nn k= 24 Train 0.780 +/- 0.066        Test 0.760
k-nn k= 25 Train 0.780 +/- 0.066        Test 0.760
k-nn k= 26 Train 0.780 +/- 0.066        Test 0.760
k-nn k= 27 Train 0.791 +/- 0.074        Test 0.760
k-nn k= 28 Train 0.799 +/- 0.082        Test 0.760
k-nn k= 29 Train 0.780 +/- 0.096        Test 0.760


Best k=19 Accuracy on Train 0.802
Best k=5 Accuracy on Test  0.760
```

In [14]:
```
plt.figure(1, figsize=(8, 6))
font = {'family' : 'sans', 'size'   : 20}
plt.rc('font', **font)
plt.title('Train vs Test Accuracy')
plt.plot(plt_x_label,plt_y_train,label='Train')
plt.plot(plt_x_label,plt_y_test,label='Test')
plt.ylim([0.5,1.0])
plt.legend()
plt.grid(color='grey')
plt.xlabel('k')
plt.ylabel('Accuracy (%)')
```

Out[14]:
```
Text(0,0.5,'Accuracy (%)')
```

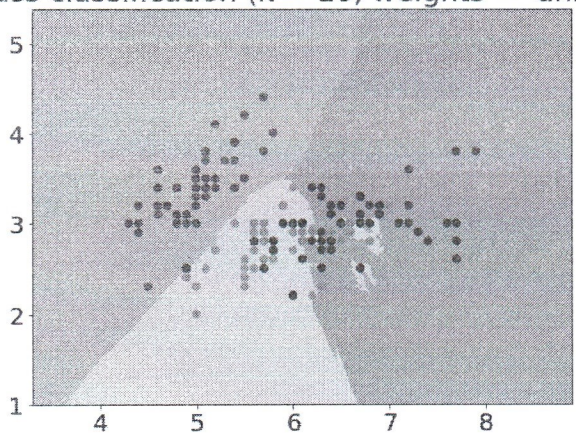


Now, we want to plot the decision boundaries for Define two color maps used to show the decision boundaries generated by the classifier.

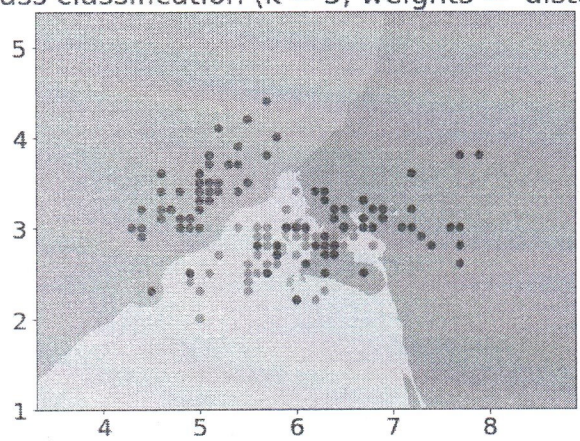
In [15]:
```
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
```

In [16]:
```
n_neighbors = {'uniform':16, 'distance':5}

h = 0.01
for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors[weights], weights=weights)
    clf.fit(x, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8,6))
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(x[:, 0], x[:, 1], c=y, cmap=cmap_bold)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("3-Class classification (k = %i, weights = '%s')"
              % (n_neighbors[weights], weights))
plt.show()
```

3-Class classification (k = 16, weights = 'uniform')



3-Class classification (k = 5, weights = 'distance')



K-Nearest Neighbor - KD-Trees — LOANS

We are now going to apply k-nearest neighbor to the loans dataset with and without kd-trees.

First we load all the needed libraries.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import linear_model, datasets, neighbors
from sklearn import model_selection
from sklearn.model_selection import StratifiedKFold
from matplotlib.colors import ListedColormap
import time
%matplotlib inline
```

```
/home/xcs/anaconda3/lib/python3.6/site-packages/matplotlib/font_manager.py:280: UserWarning: Matplotlib is building the font ca
che using fc-list. This may take a moment.
  'Matplotlib is building the font cache using fc-list. '
```

Next, we load the dataset that is included in the Scikit-Learn dataset module.

In [2]:
```
loans = pd.read_csv('LoansNumerical.csv')
target = 'safe_loans'
features = loans.columns[loans.columns!=target]

x = loans[features]
y = loans[target]
```

Now we apply plain k-nearest neighbor with a k of 15 and evaluate it using 10 fold crossvalidation

In [3]:
```
k = 15
knn = neighbors.KNeighborsClassifier(n_neighbors=k, algorithm='brute')
knn_kdtree = neighbors.KNeighborsClassifier(n_neighbors=k, algorithm='kd_tree')
```

In [4]:
```
%%time
start_kdtree = time.process_time()
knn_eval_kd = model_selection.cross_val_score(knn_kdtree, x, y, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=1234))
end_kdtree = time.process_time()
```

```
CPU times: user 11.6 s, sys: 308 ms, total: 11.9 s
Wall time: 11.9 s
```

In [5]:
```
time_taken_kdtree = end_kdtree-start_kdtree
print("%d-nearest-neighbor\tt=%.3f\tAccuracy=%.3f\tStd=%.3f"%(k,time_taken_kdtree,np.average(knn_eval_kd),np.std(knn_eval_kd)))
```

```
15-nearest-neighbor	t=11.932	Accuracy=0.806	Std=0.001
```

In [6]:
```
%%time
start_vanilla = time.process_time()
knn_eval = model_selection.cross_val_score(knn, x, y, cv=StratifiedKFold(n_splits=10,shuffle=True,random_state=1234))
end_vanilla = time.process_time()
```

```
CPU times: user 8min 23s, sys: 1min 33s, total: 9min 57s
Wall time: 7min 5s
```

In [7]:
```
time_taken_vanilla = end_vanilla-start_vanilla
print("%d-nearest-neighbor\tt=%.3f\tAccuracy=%.3f\tStd=%.3f"%(k,time_taken_vanilla,np.average(knn_eval),np.std(knn_eval)))
```

```
15-nearest-neighbor	t=597.443	Accuracy=0.806	Std=0.001
```

k-Nearest Neighbors Regression

We now apply k-nearest neighbor to regression problems. First, we apply it to some simple generated data and then to the housing data that we used in the earlier experiments.

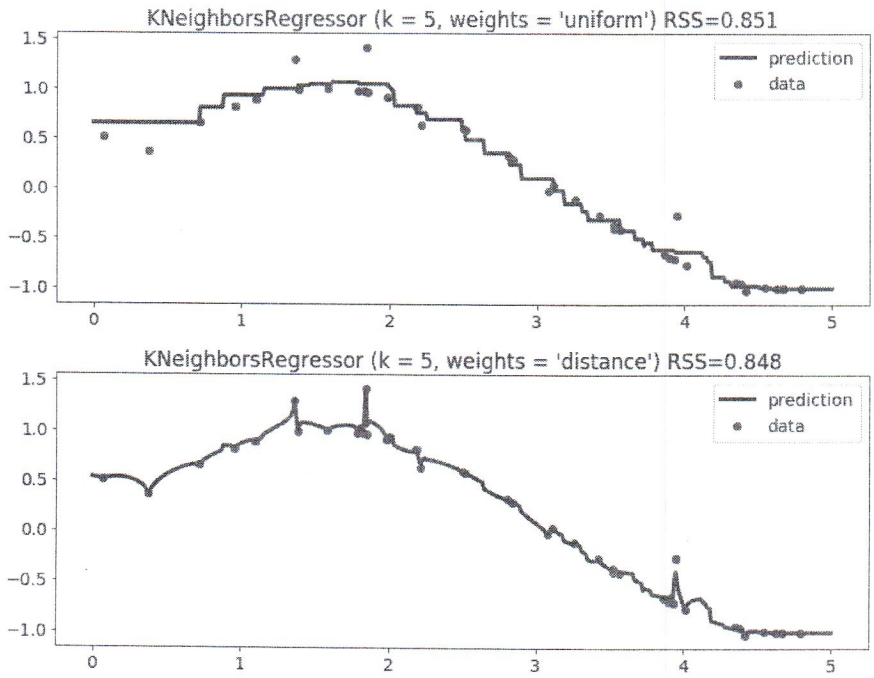
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

import pandas as pd
%matplotlib inline
```

Let's generate some sample data using a fixed random seed to be able to reproduce the analysis later.

In [2]:
```
np.random.seed(1234)
x = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(x).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))
```

Fit regression model with k of 5

In [3]:
```
n_neighbors = 5

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors, weights=weights)
    y_ = knn.fit(x, y).predict(T)

    scores = cross_val_score(knn, x, y, cv=KFold(n_splits=10, random_state=1234, shuffle=True))

    plt.figure(figsize=(12,9))
    font = {'family' : 'sans', 'size'   : 14}
    plt.rc('font', **font)
    plt.subplot(2, 1, i + 1)
    plt.scatter(x, y, c='red', label='data');
    plt.plot(T, y_, c='blue', label='prediction', linewidth=3);
    plt.axis('tight');
    plt.legend();
    plt.title("KNeighborsRegressor (k = %i, weights = '%s') RSS=%.3f" % (n_neighbors, weights, np.average(scores)));

plt.show();
```



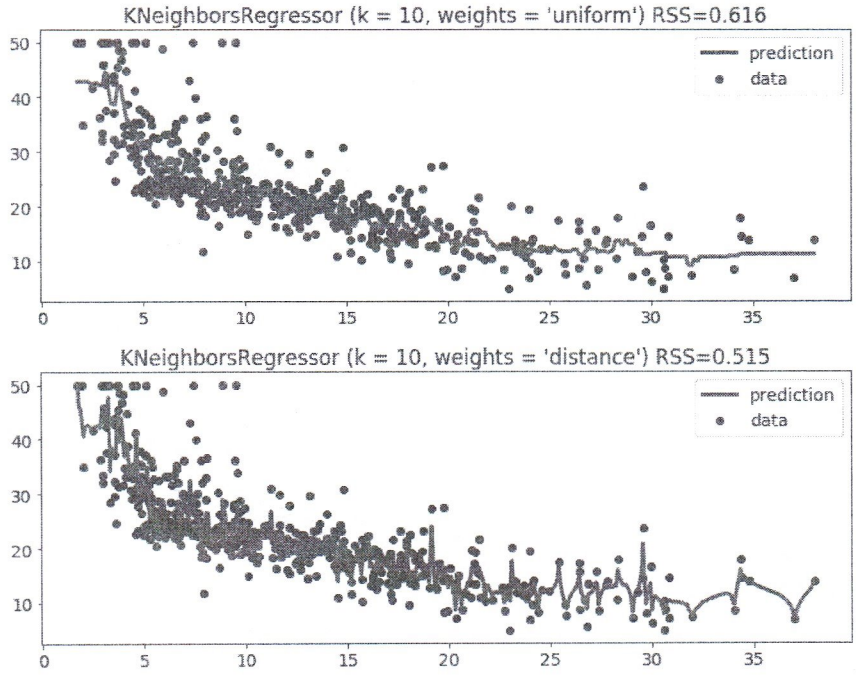
Let's repeat the experiment using the housing data.

In [9]:
```
from sklearn.datasets import load_boston
boston = load_boston()

df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['MEDV'] = boston.target
```

In [10]:
```
X = dataset['LSTAT'].values.reshape(506, 1)
y = dataset['MEDV'].values.reshape(506, 1)
T = np.linspace(np.min(x), np.max(x), 500)[:, np.newaxis]
```

In [11]:
```
n_neighbors = 10

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors, weights=weights)
    y_ = knn.fit(X, y).predict(T)

    scores = cross_val_score(knn, x, y, cv=KFold(n_splits=10, random_state=1234, shuffle=True))

    plt.figure(figsize=(12,9))
```

```
    font = {'family' : 'sans', 'size'   : 14}
    plt.rc('font', **font)
    plt.subplot(2, 1, i + 1)
    plt.scatter(x, y, c='blue', label='data');
    plt.plot(T, y_, c='red', linewidth=3, label='prediction');
    plt.axis('tight');
    plt.legend();
    plt.title("KNeighborsRegressor (k = %i, weights = '%s') RSS=%.3f" % (n_neighbors, weights, np.average(scores)));

plt.show();
```



Note that this is still an evaluation on training data. How does the prediction change when the value of k is decreased or increased?

Laplace Estimator

Simple notebook to check whether the Laplace Smoothing (or Laplace Estimator) is also applied to the class

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

from sklearn.naive_bayes import CategoricalNB
from sklearn import preprocessing
```

In [18]:
```
df = pd.DataFrame({'x':[1,1,0],'y':[0,1,1]})
```

In [19]:
```
df
```

Out[19]:

	x	y
0	1	0
1	1	1
2	0	1

In [20]:
```
input_variables = ['x']
target_variable = 'y'
```

In [21]:
```
nb = CategoricalNB(alpha=1.0)
```

In [22]:
```
nb.fit(df[input_variables],df[target_variable])
```

Out[22]:
```
CategoricalNB(alpha=1.0, class_prior=None, fit_prior=True)
```

In [23]:
```
np.exp(nb.class_log_prior_)
```

Out[23]:
```
array([0.33333333, 0.66666667])
```

In [24]:
```
nb.predict_proba(df[input_variables])
```

Out[24]:
```
array([[0.4 , 0.6 ],
       [0.4 , 0.6 ],
       [0.25, 0.75]])
```

if you apply the theory and compute the probabilities for each one of the three cases, you will get the same results when using the class priors computed without adding the smoothing.

In [ ]:
```
```

Linear Regression

We will be using a small dataset containing the data for the Boston housing market and will try to predict the value of houses (represented by the MEDV value) using only the percentage of people of lower status in the area (represented by the LSTAT variable).

In [1]:
```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
%matplotlib inline
```

In [2]:
```
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 24
4-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference
of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [3]:
```
df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['MEDV'] = boston.target
```

In [4]:
```
df.describe()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

Given a dataset, the name of a column and a degree it generates all the additional columns needed to perform a polynomial regression

We define a helper function to plot the original data against the data generated by a model

In [5]:
```
def plot_scatter(x,y,xp,yp,title=""):
    """Plots the original data (x,y) and a set of point (xp,yp) showing the model approximation"""
    font = {'family' : 'sans',
        'size'   : 14}
    plt.rc('font', **font)

    plt.scatter(x, y,  color='blue')
    plt.plot(xp, yp, color='red', linewidth=3)
```

```
    plt.xlabel("LSTAT")
    plt.ylabel("MEDV")

    if (title!=""):
        plt.title(title)

    plt.xlim([0,40])
    plt.ylim([0,60])
    plt.show()
```

Simple Linear Regression

We start with a very basic model using one input variable x and fits the data using the model $y = w_0 + w_1 x$. For example, we can use variable LSTAT (percentage of lower status of the population) as input to predict the target variable MEDV (median value of owner occupied homes in $1000s) as target.

First, we create the matrix X of inputs and the target vector y.

In [6]:
```
X = df['LSTAT'].values.reshape(-1,1)
y = df['MEDV']
```

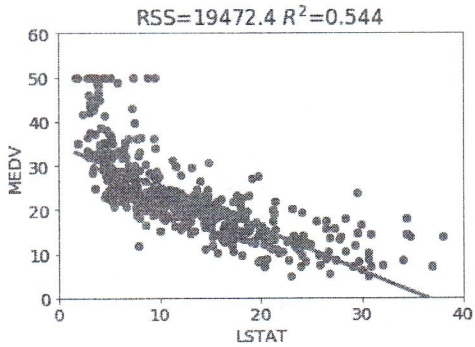
Next, we create a linear regressor and fit it with the input/output data.

In [7]:
```
linear_regressor = linear_model.LinearRegression()
X.reshape(-1,1)
linear_regressor.fit(X, y);
```

We can now compute the R^2 score and RSS by first computing the predicted output yp.

In [8]:
```
yp = linear_regressor.predict(X)

r2 = r2_score(y,yp)

rss = sum((yp-y)*(yp-y))
```

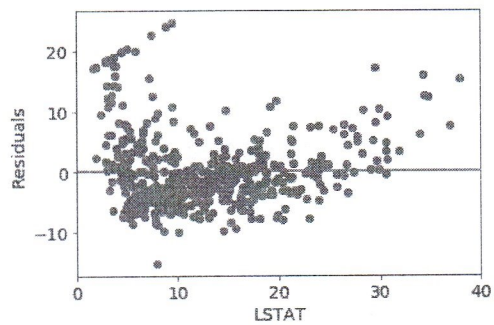
In [10]:
```
title = "RSS=%.1f $R^2$=%.3f"%(rss,r2)
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
yplot = linear_regressor.predict(Xplot)
plot_scatter(X,y,Xplot,yplot,title)
```



RSS and R^2 provide an overall evaluation of our model. It is more interesting to explore how our model makes mistakes by analyzing the residuals, that is the difference between the target values and the fitted values. First we compute the residuals and then we plot them.

In [11]:
```
residuals = y-yp
```

In [12]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.scatter(X, residuals,  color='blue')
plt.plot([0,40],[0,0],'-',c="red")
plt.xlabel("LSTAT")
plt.ylabel("Residuals")
plt.xlim([0,40])
plt.show()
```

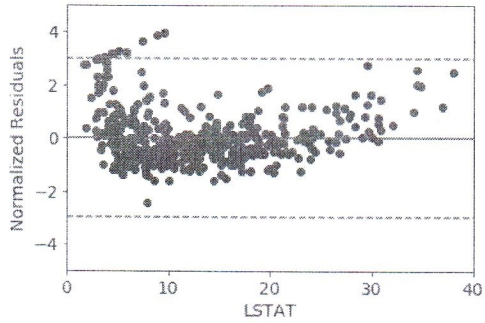


The points are not all randomly scattered, showing different variances at different values of the input variable. We can also spot some patterns (for examples, points in a straight line). The plot using z-score normalized residuals is more informative since the values -3 and 3 represents the range containing the 99.7% of the values

In [13]:
```
from sklearn.preprocessing import StandardScaler
normalized_residuals = StandardScaler().fit_transform(residuals.values.reshape(-1,1))
```

In [14]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.scatter(X, normalized_residuals,  color='blue')
plt.plot([0,40],[0,0],'-',c="red")
```

```
plt.plot([0,40],[-3,-3],'--',c="red")
plt.plot([0,40],[3,3],'--',c="red")
plt.xlabel("LSTAT")
plt.ylabel("Normalized Residuals")
plt.xlim([0,40])
plt.ylim([-5,5])
plt.show()
```



Multiple Linear Regression

We can build a model to predict MEDV using more input variables from the ones available in the data. However, in this example to be able to plot the model using more input variables, we consider polynomial models based on the same input variable LSTAT.

In [16]:
```
from sklearn.preprocessing import PolynomialFeatures
```

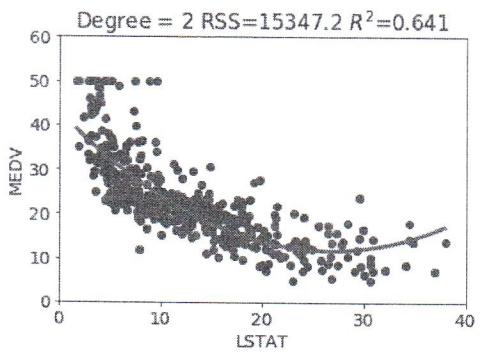
Let' start by computing a model using a second degree polynomial. Next, we use linear regression to fit the new input data and the target variable.

In [17]:
```
polynomial2 = PolynomialFeatures(degree=2, include_bias=False)
X2 = polynomial2.fit_transform(X)
```

In [18]:
```
linear_regressor2 = linear_model.LinearRegression()
X2.reshape(-1,2)
linear_regressor2.fit(X2, y);
yp2 = linear_regressor2.predict(X2)
```

Let's evaluate the model and plot it.

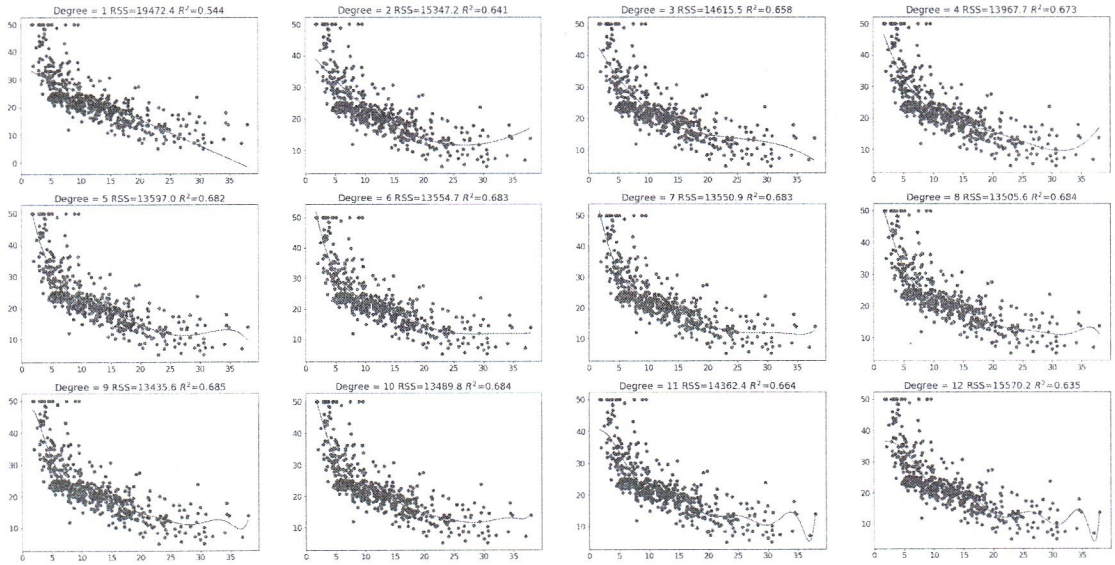
In [20]:
```
r2_p2 = r2_score(y,yp2)
rss_p2 = sum((yp2-y)*(yp2-y))

# Let's create the input data to plot the model
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
Xplot2 = polynomial2.fit_transform(Xplot)

# compute the model on the plot data
yplot2 = linear_regressor2.predict(Xplot2)

plot_scatter(X[:,0],y,Xplot2[:,0],yplot2,"Degree = 2 RSS=%.1f $R^2$=%.3f"%(rss_p2,r2_p2))
```



We can increase the degree of the polynomial and check whether and how much the model improves.

In [22]:
```
max_polynomial = 12
f, axarr = plt.subplots(3, 4)

plt.rcParams['figure.figsize'] = (40.0, 20.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
        X_polynomial = X


    linear_regressor = linear_model.LinearRegression()
    X_polynomial.reshape(-1,degree)
    linear_regressor.fit(X_polynomial, y);

    yp = linear_regressor.predict(X_polynomial)
    yplot = linear_regressor.predict(Xplot_polynomial)
```

```
    r2 = r2_score(y,yp)
    rss = sum((yp-y)*(yp-y))

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d RSS=%.1f $R^2$=%.3f"%(degree,rss,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



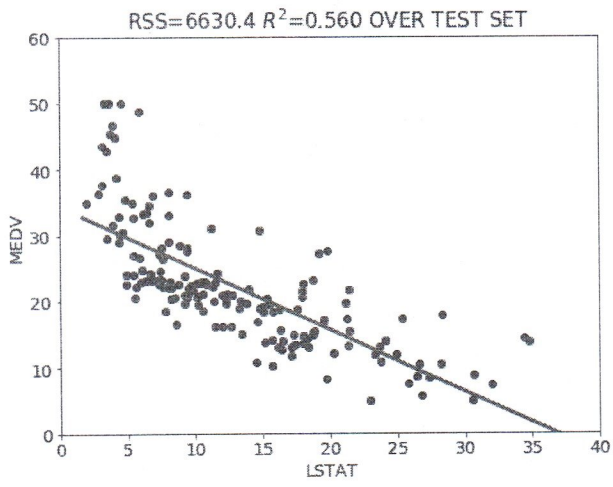
As can be noted the RSS decreases as the degree of the polynomial increases. Only for a degree of 12 we note an increase of RSS and a decrease of R^2. However, this is not a robust evaluation since we are scoring the models on the same data we used to build the model. To get a robust evaluation we need to either split the data into train set and test set, build the model on the train set, and evaluate the model on the test set. Or we can apply crossvalidation.

Model Evaluation Using a Test Set

To evaluate the model we now split the data between train and test using 2/3 of the data for training and 1/3 for testing. We start with the simple linear regression we done at the beginning. Note that to be able to replicate the results we are setting a random seed.

In [23]:
```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.33, random_state=1234)
```

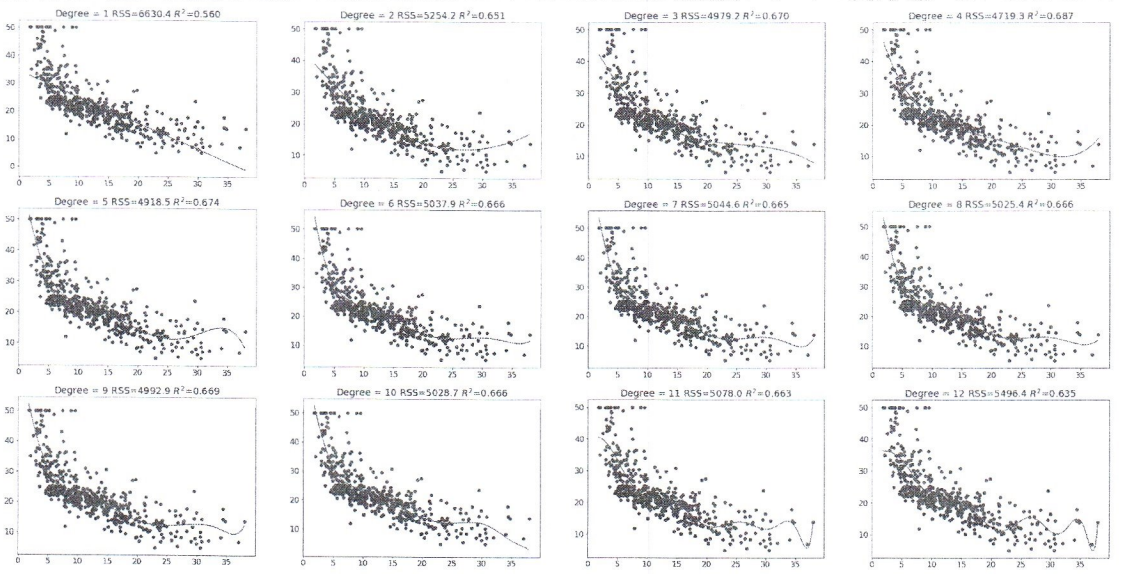
In [25]:
```
# build the model on the train
linear_regressor.fit(X_train, y_train);

# evaluate the model on the test
yp = linear_regressor.predict(X_test)

r2 = r2_score(y_test,yp)

rss = sum((yp-y_test)*(yp-y_test))

title = "RSS=%.1f $R^2$=%.3f OVER TEST SET"%(rss,r2)
Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
yplot = linear_regressor.predict(Xplot)
plt.rcParams['figure.figsize'] = (8.0, 6.0)
plot_scatter(X_test,y_test,Xplot,yplot,title)
```



Note that RSS is lower since it is computed over a smaller data set. R^2 is slightly better. Let's see what happens with higher polynomials.

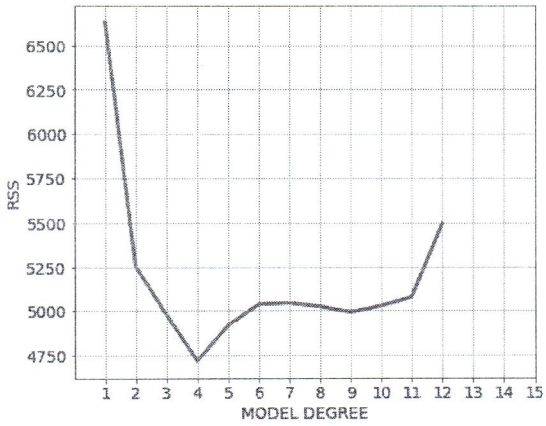
In [27]:
```
max_polynomial = 12
f, axarr = plt.subplots(3, 4)

plt.rcParams['figure.figsize'] = (40.0, 20.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)
```

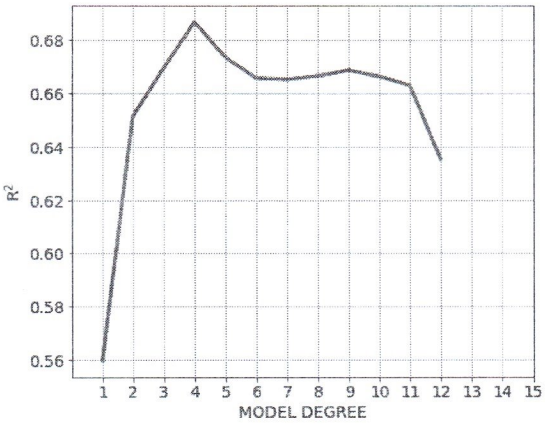
```
rss_train_values = []
r2_train_values = []


rss_values = []
r2_values = []

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
        X_polynomial = X


    X_train, X_test, y_train, y_test = model_selection.train_test_split(X_polynomial, y, test_size=0.33, random_state=1234)

    linear_regressor = linear_model.LinearRegression()
    X_polynomial.reshape(-1,degree)
    linear_regressor.fit(X_train, y_train);

    yp_train = linear_regressor.predict(X_train)
    r2_train = r2_score(y_train,yp_train)
    rss_train = ((yp_train-y_train)**2).sum()
    rss_train_values.append(rss_train)
    r2_train_values.append(r2_train)

    yp = linear_regressor.predict(X_test)
    yplot = linear_regressor.predict(Xplot_polynomial)

    r2 = r2_score(y_test,yp)
    rss = ((yp-y_test)**2).sum()

    rss_values.append(rss)
    r2_values.append(r2)

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d RSS=%.1f $R^2$=%.3f"%(degree,rss,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



We can plot how RSS and R^2 changes based on the degree of the polynomial approximation.

In [28]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("RSS")
axarr[0].set_xlim([0,13])
axarr[0].set_xticks(range(1,16))
axarr[0].grid()
axarr[0].plot(range(1,max_polynomial+1), rss_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("R$^2$")
axarr[1].set_xlim([0,13])
axarr[1].set_xticks(range(1,16))
axarr[1].grid()
axarr[1].plot(range(1,max_polynomial+1), r2_values, color="blue",linewidth=3);
```



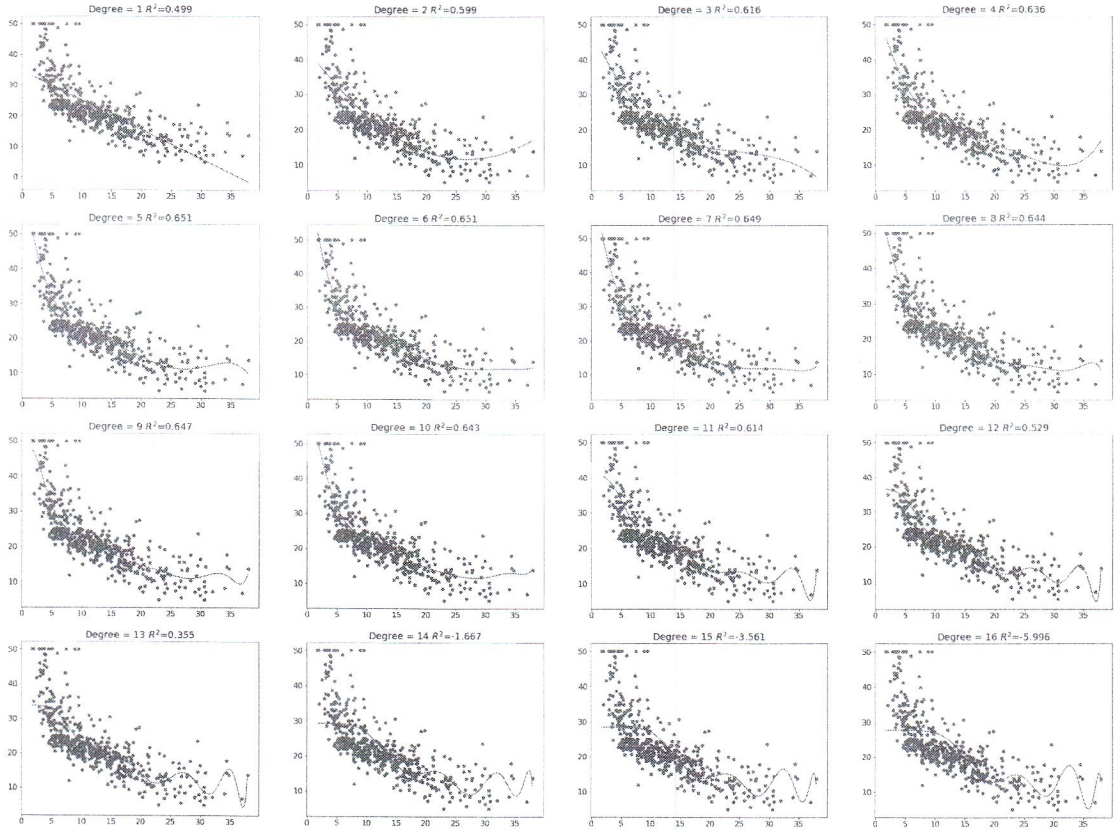


Note that as the degree of the polynomial increases until a value of 4, the RSS decreases and similarly R^2 inscreases. Then the RSS starts increasing again, the model is starting to overfit the data. The results appear to suggest that a polynomial of degree 4 is the best choice, but are we sure? Our evaluation is based on one run performed over a specific train/test partition. To have a more robust evaluation we can either repeat the procedure with different train/test partitions or otherwise use crossvalidation.

Model Evaluation using Crossvalidation

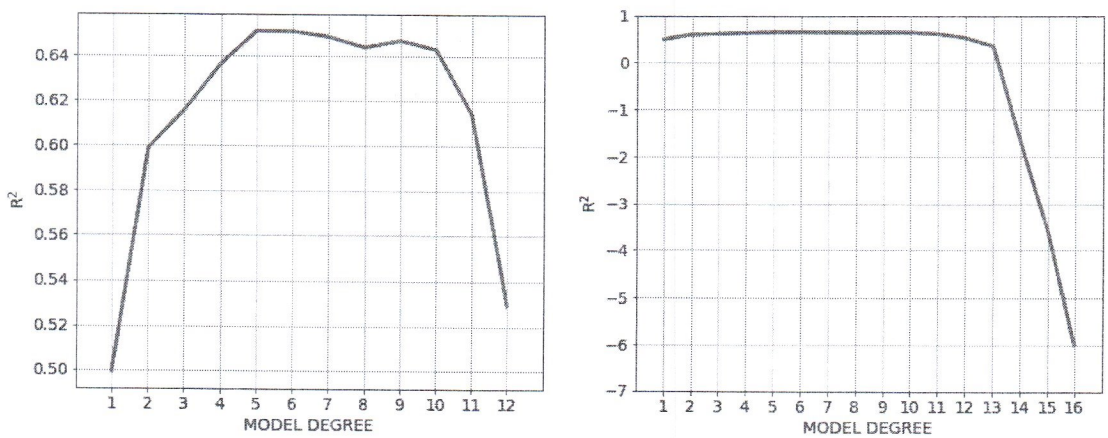
We now repeat the procedure but evaluate the model using crossvalidation.

In [32]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

max_polynomial = 16
f, axarr = plt.subplots(4, 4)

plt.rcParams['figure.figsize'] = (40.0, 30.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)

rss_values = []
r2_values = []

for degree in range(1,max_polynomial+1):
    if (degree!=1):
        polynomial = PolynomialFeatures(degree=degree, include_bias=False)
        X_polynomial = polynomial.fit_transform(X)
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = polynomial.fit_transform(Xplot)
    else:
        Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
        Xplot_polynomial = Xplot
        X_polynomial = X


    linear_regressor = linear_model.LinearRegression()

    X_polynomial.reshape(-1,degree)

    # score returns the scores for each single run
    score = cross_val_score(linear_regressor, X_polynomial, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

    r2 = score.mean()
    r2_values.append(r2)

    # let's build the final model (using **all the data**)
    linear_regressor.fit(X_polynomial,y)
    yplot = linear_regressor.predict(Xplot_polynomial)

    title = "DEGREE "+str(degree)+" RSS=" + str(round(rss,3))+" R="+str(round(r2,3))

    target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
    target_plot.scatter(X[:,0],y, color="blue")
    target_plot.set_title("Degree = %d $R^2$=%.3f"%(degree,r2))
    target_plot.plot(Xplot2[:,0],yplot, color="red")
```



As you can note the evaluation given by crossvalidation is less "optimistic" about the performance of our models, in fact the values of R^2 are generally lower. Also if we plot R^2 as a function of the degree (below) we have a much different curve that suggests a degree of 5 or 6 as the best choice --- a quite different result with respect to the previous basic holdout evaluation. Note that, for higher degree polynomials R^2 becomes negative, this means that the model performs worst that the basic mean. The model tries very hard to fit the training data so to become useless to predict unseen cases.

In [34]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,13])
axarr[1].set_ylim([0,1])
axarr[0].set_xticks(range(1,13))
axarr[0].grid()
axarr[0].plot(range(1,13), r2_values[0:12], color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("R$^2$")
axarr[1].set_xlim([0,17])
axarr[1].set_xticks(range(1,17))
axarr[1].set_ylim([-7,1])
axarr[1].grid()
axarr[1].plot(range(1,max_polynomial+1), r2_values, color="blue",linewidth=3);
```



Regularization

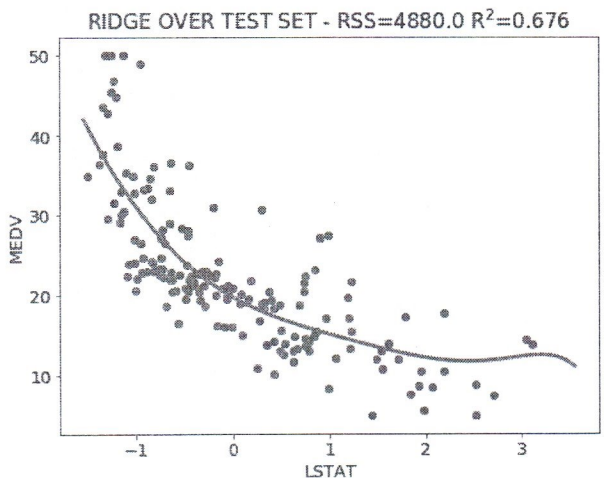
To avoid overfitting, we can introduce Lasso (L_1) and Ridge (L_2) regularization which will take care of variables that might cause overfit. Let's consider the version of the dataset extended by adding powers of LSTAT up to the 10th degree. Also, so far we never normalized the variables which we should do since the

In [91]:
```
polynomial = PolynomialFeatures(degree=10, include_bias=False)
X_polynomial = polynomial.fit_transform(X)
```

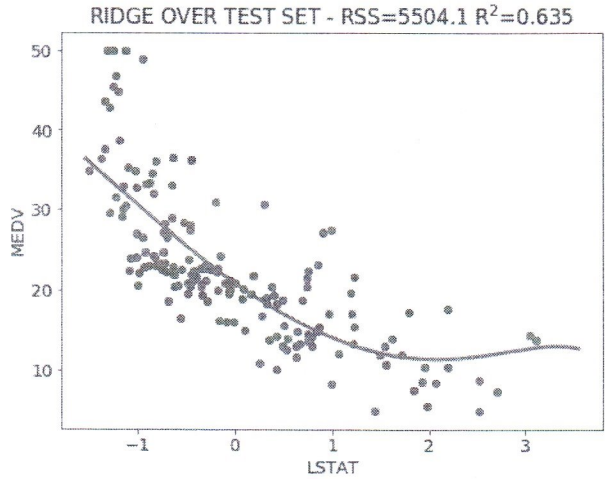
```
scaler = StandardScaler()

scaler.fit(X_polynomial)

X_normalized = scaler.transform(X_polynomial)

Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
Xplot_polynomial = polynomial.fit_transform(Xplot)

Xplot_normalized = scaler.transform(Xplot_polynomial)

# X_normalized = StandardScaler().fit_transform(X_polynomial)
# Xplot = np.arange(np.min(X_normalized[:,0]),np.max(X_normalized[:,0]),0.1).reshape(-1,1)
```

First, we apply the process using Ridge regression that define cost as, RSS($\vec{w}$) + $\alpha||\vec{w}||_2^2$ with an α of 0.1 (or λ)

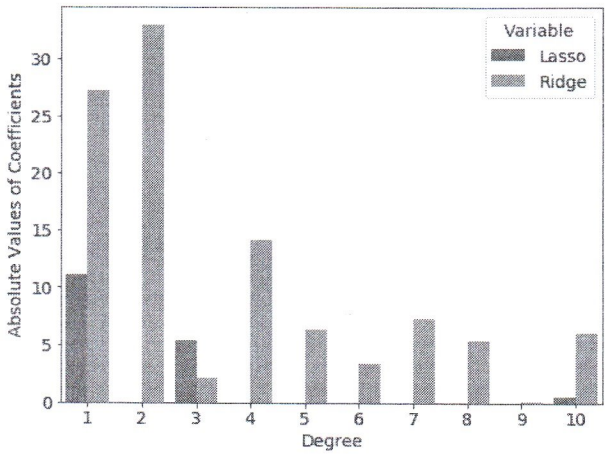
In [92]:
```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.1,max_iter=1000,random_state=1234)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X_normalized, y, test_size=0.33, random_state=1234)

ridge_model = ridge.fit(X_train,y_train)

yp = ridge.predict(X_test)
yp_plot = ridge.predict(Xplot_normalized)

r2 = r2_score(y_test,yp)
rss = ((yp-y_test)**2).sum()

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_test[:,0], y_test,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("RIDGE OVER TEST SET - RSS=%.1f R$^2$=%.3f"%(rss,r2))
plt.show()
```



In [84]:
```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1,max_iter=1000,random_state=1234)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X_normalized, y, test_size=0.33, random_state=1234)

lasso_model = lasso.fit(X_train,y_train)

yp = lasso.predict(X_test)
yp_plot = lasso.predict(Xplot_normalized)

r2 = r2_score(y_test,yp)
rss = ((yp-y_test)**2).sum()

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_test[:,0], y_test,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("LASSO OVER TEST SET - RSS=%.1f R$^2$=%.3f"%(rss,r2))
plt.show()
```



In [108...
```
import seaborn
coefficients = pd.DataFrame({'Degree':[x for x in range(1,len(lasso_model.coef_)+1)], 'Lasso':np.abs(lasso_model.coef_), 'Ridge
tidy = coefficients.melt(id_vars='Degree').rename(columns=str.title)
seaborn.barplot(x='Degree', y='Value', hue='Variable', data=tidy)
plt.ylabel("Absolute Values of Coefficients");
```

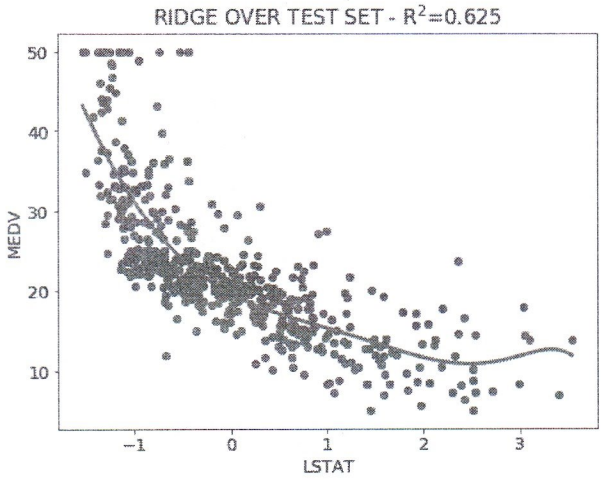


Note that Lasso creates a model with a lower R^2 score but only three variables with non zero weights.

Regularization Evaluation using Crossvalidation

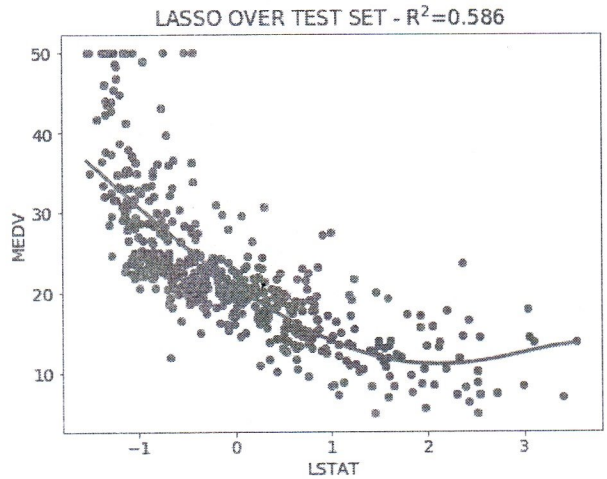
We can repeat the analysis using cross-validation

In [113...
```
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.1,max_iter=1000,random_state=1234)

score = cross_val_score(ridge, X_normalized, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

r2 = score.mean()

yp = ridge.fit(X_normalized,y)
yp_plot = ridge.predict(Xplot_normalized)

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_normalized[:,0], y,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("RIDGE OVER TEST SET - R$^2$=%.3f"%(r2))
plt.show()
```



In [114...

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1,max_iter=1000,random_state=1234)

score = cross_val_score(lasso, X_normalized, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

r2 = score.mean()

yp = lasso.fit(X_normalized,y)
yp_plot = lasso.predict(Xplot_normalized)

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)

plt.scatter(X_normalized[:,0], y,  color='blue')
plt.plot(Xplot_normalized[:,0], yp_plot, color='red', linewidth=3)
plt.xlabel("LSTAT")
plt.ylabel("MEDV")
plt.title("LASSO OVER TEST SET - R$^2$=%.3f"%(r2))
plt.show()
```



As before, the results from the cross-validation are less optimistic. Note that, when using cross-validation the final model is produced using the entire dataset.

Logistic Regression - Multiclass Classification

Logistic regression assumes that the class attribute has only two values (e.g., good/bad, 1/0, 1/-1). When facing a problem with more class values, there are two options for applying a two value classifier. One can build one classifier for each class value and train it against all the other classes. Otherwise, one can minimize the loss using on the multinomial loss fit across the entire probability distribution.

In this example, we apply logistic regression using the **one versus the rest** evaluation using a well-known dataset called Iris

First we load all the needed libraries.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, datasets
from sklearn import model_selection
%matplotlib inline
```

Let's define the colormaps that we will use for plotting

In [2]:
```
from matplotlib.colors import ListedColormap
background_cmap = ListedColormap(['#68abf0','#b2d0b7','#f65d79'])
background_cmap = ListedColormap(['#a6cdf6','#b2d0b7','#f98ea1'])
dots_cmap = ListedColormap(['#1b80e8','#599062','#e20c32'])
plt.register_cmap(cmap=background_cmap)
plt.register_cmap(cmap=dots_cmap)
colors = ['#1b80e8','#599062','#e20c32']
```

Next, we load the dataset that is included in the Scikit-Learn dataset module.

In [3]:
```
# import some data to play with
iris = datasets.load_iris()
target = np.array(iris.target)

print("Number of examples: ", iris.data.shape[0])
print("Number of variables:", iris.data.shape[0])
print("Variable names:     ", iris.feature_names)
print("Target values:      ", iris.target_names)
print("Class Distribution  ", [(x,sum(target==x)) for x in np.unique(target)])
```

```
Number of examples:  150
Number of variables: 150
Variable names:      ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target values:       ['setosa' 'versicolor' 'virginica']
Class Distribution   [(0, 50), (1, 50), (2, 50)]
```

We will use only the first two variables (sepal length and sepal width)

In [4]:
```
X = iris.data[:, :2]
y = iris.target
```

And build a logistic regression model without regularization (thus, $\alpha = 0$ which means that C must be very large) and using **one versus rest** for multiclass classification (this is the default option so **multi_class='ovr'** can be avoided.

In [5]:
```
logistic = linear_model.LogisticRegression(C=10e10, multi_class='ovr',random_state=1234)
logistic.fit(X,y)
xval = model_selection.cross_val_score(logistic, X, y)

print ("Average accuracy = %3.2f +/- %3.2f" %(np.average(xval),np.std(xval)))
```

```
Average accuracy = 0.81 +/- 0.04
```

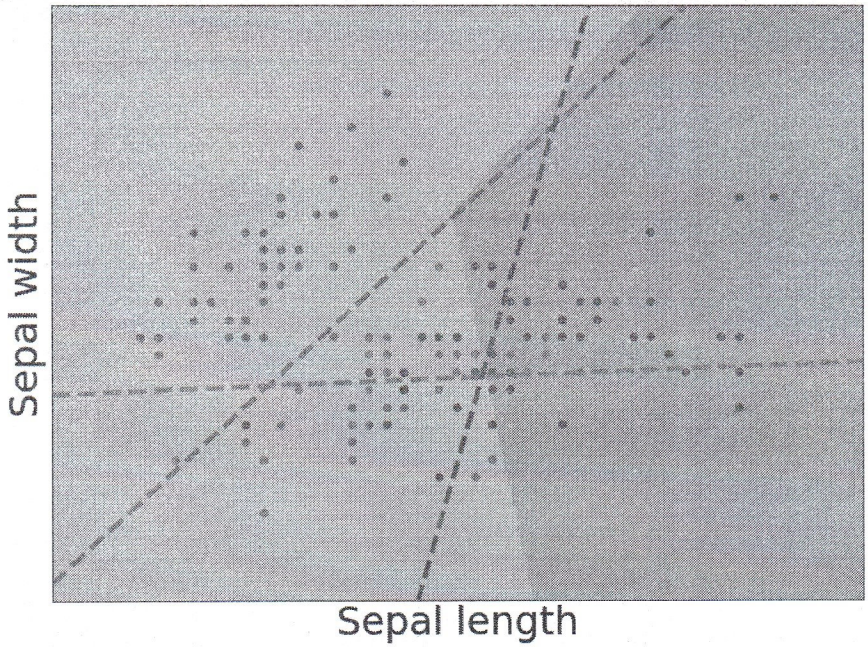
Let's plot the decision boundaries. First we compute the boundaries for the two input variables (x0_min & x0_max; x1_min & x1_max). Next we build a grid and compute the classification for each position so as to paint the regions according to an assigned class.

In [6]:
```
x0_min, x0_max = X[:, 0].min() - .5, X[:, 0].max() + .5
x1_min, x1_max = X[:, 1].min() - .5, X[:, 1].max() + .5

h = .01
xx0, xx1 = np.meshgrid(np.arange(x0_min, x0_max, h), np.arange(x1_min, x1_max, h))

z = logistic.predict(np.c_[xx0.ravel(), xx1.ravel()])
```

Now let's plot the points and the boundaries.

In [10]:
```
z = z.reshape(xx0.shape)
plt.figure(1, figsize=(12, 9))
plt.pcolormesh(xx0, xx1, z, cmap=background_cmap)

font = {'family' : 'sans', 'size'   : 32}
plt.rc('font', **font)

for i, color in zip(logistic.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color) #, cmap=plt.cm.Pastel2)

plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(xx0.min(), xx0.max())
plt.ylim(xx1.min(), xx1.max())
plt.xticks(())
plt.yticks(())


### plot also the planes
coef = logistic.coef_
intercept = logistic.intercept_

def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]
    plt.plot([x0_min, x0_max], [line(x0_min), line(x0_max)],
             ls="--", lw=4, color=color)
```

```
# colors = "rgb"
for i, color in zip(logistic.classes_, colors):
    plot_hyperplane(i, color)
plt.savefig("fig_iris_ovr.png");
plt.show()
```

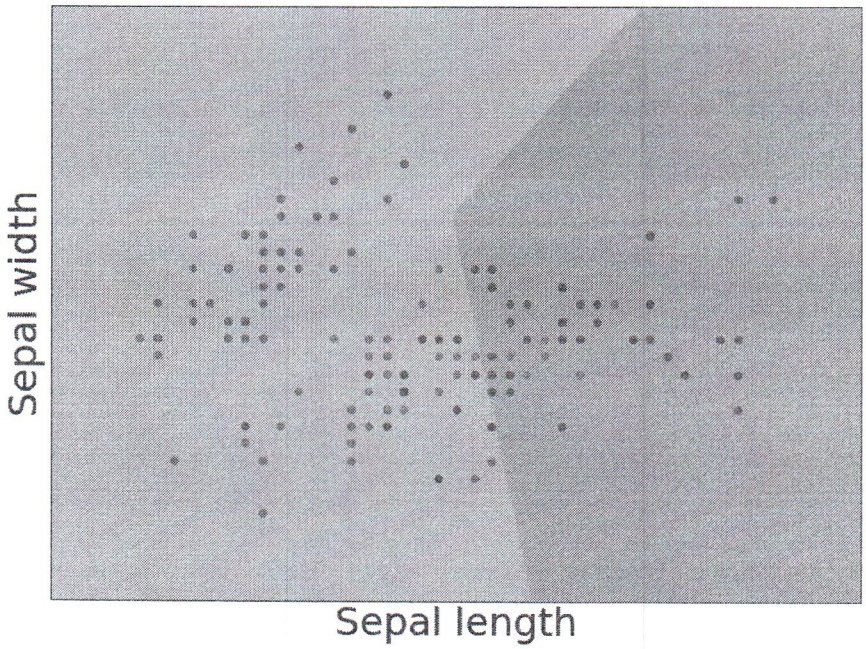


We now repeat the same procedure using the multinomial approach.

In [8]:
```
logistic_mn = linear_model.LogisticRegression(C=10e10, solver='sag', multi_class='multinomial', random_state=1234, max_iter=10
logistic_mn.fit(X,y)
xval = model_selection.cross_val_score(logistic_mn, X, y)

print ("Average accuracy = %3.2f +/- %3.2f" %(np.average(xval),np.std(xval)))
```

```
Average accuracy = 0.82 +/- 0.06
```

In [11]:
```
x0_min, x0_max = X[:, 0].min() - .5, X[:, 0].max() + .5
x1_min, x1_max = X[:, 1].min() - .5, X[:, 1].max() + .5

h = .01
xx0, xx1 = np.meshgrid(np.arange(x0_min, x0_max, h), np.arange(x1_min, x1_max, h))

z = logistic_mn.predict(np.c_[xx0.ravel(), xx1.ravel()])

z = z.reshape(xx0.shape)
plt.figure(1, figsize=(12, 9))
plt.pcolormesh(xx0, xx1, z, cmap=background_cmap)

font = {'family' : 'sans', 'size'   : 32}
plt.rc('font', **font)

for i, color in zip(logistic_mn.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color) #, cmap=plt.cm.Pastel2)

plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(xx0.min(), xx0.max())
plt.ylim(xx1.min(), xx1.max())
plt.xticks(())
plt.yticks(())
plt.savefig("fig_iris_multinomial.png");
plt.show()
```



Questions

- Why are we using the random start?
- What happens if we modify the value of C and decrease it significantly?
- We applied 10-fold crossvalidation to evaluate each algorithm. But 10-fold crossvalidation generates ten models with different performances, which one should we deploy at the end?

In [ ]:

REGRESSION EXAMPLE CLEAN

In [5]:
```
# import all the required libraries and put matplotlib in inline mode to plot on the notebook
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline

# linear regression
from sklearn import linear_model

# nearest neighbor
from sklearn import neighbors

# regression trees (simple and ensemble)
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

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

In [6]:
```
# define the random seed if needed
random_seed = 1234

# define the figure size and the font size
fig_width = 12
fig_height = 9
fig_font_size = 16
```

In [7]:
```
df = pd.read_csv('housing.csv')
df.describe(include='all')
```

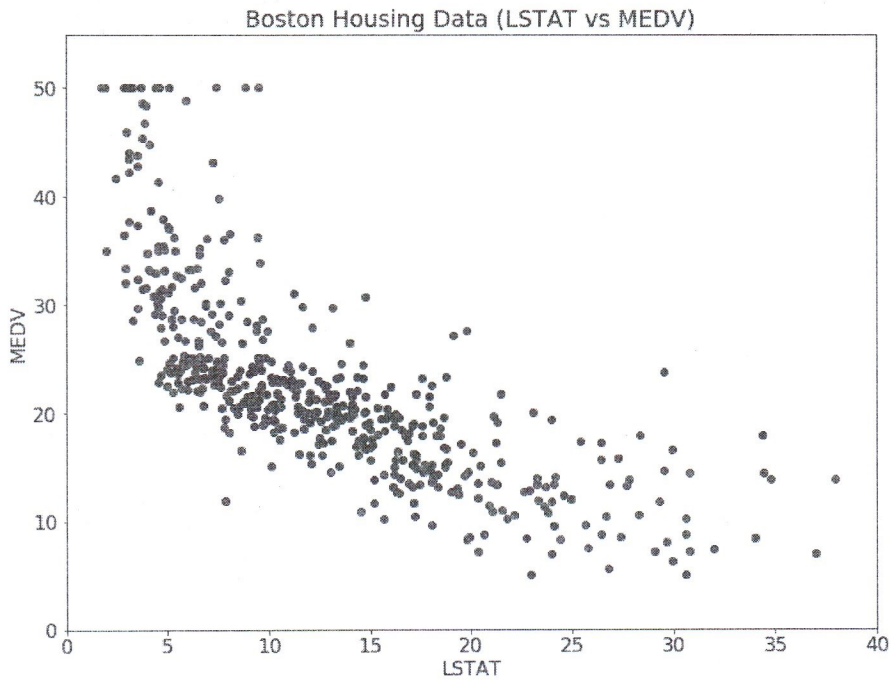
Out[7]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

In [17]:
```
def plot_prediction(x,y,xp,yp,title="",filename=""):
    """Plots the original data (x,y) and a set of point (xp,yp) showing the model approximation"""

    plt.figure(figsize=(fig_width,fig_height))

    font = {'family' : 'sans', 'size'   : fig_font_size}
    plt.rc('font', **font)

    plt.scatter(x, y,  color='blue')

    plt.plot(xp, yp, color='red', linewidth=3)

    plt.xlabel("LSTAT")
    plt.ylabel("MEDV")

    plt.title(title)

    plt.xlim([0,40])
    plt.ylim(0,55)


    if (filename!=""):
        plt.savefig(filename)

    plt.show()
```

In [18]:
```
# compute the data inputs
X = df['LSTAT'].values.reshape(-1,1)
y = df['MEDV'].values.ravel()

# input values used to plot the predicted model
X_test = np.linspace(np.min(X), np.max(X), 500)[:, np.newaxis]
```

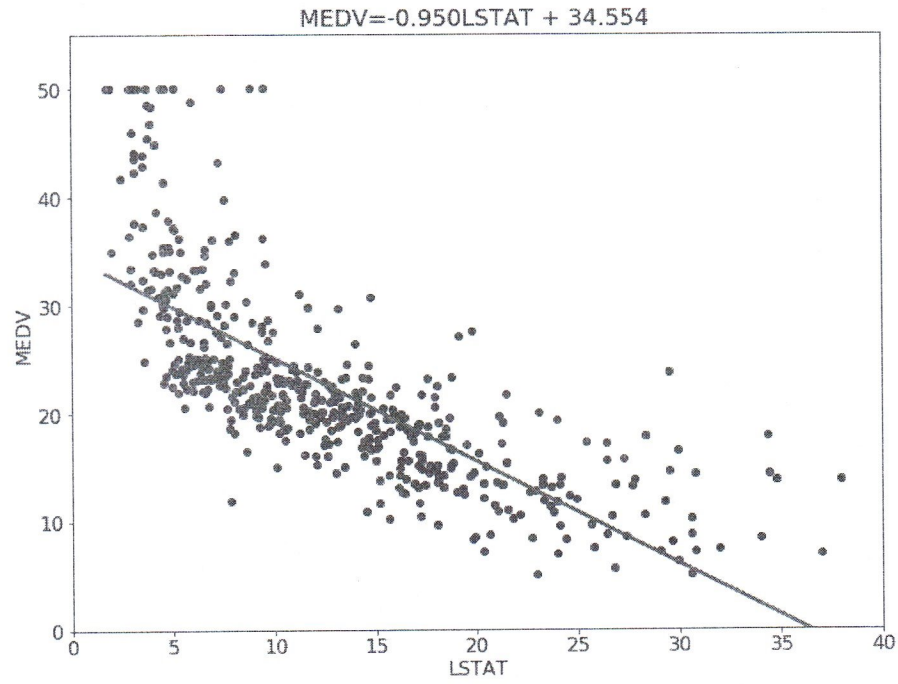
In [19]:
```
plt.figure(figsize=(fig_width,fig_height))

font = {'family' : 'sans', 'size'   : fig_font_size}
plt.rc('font', **font)

plt.scatter(X, y,  color='blue')

plt.xlabel("LSTAT")
plt.ylabel("MEDV")

plt.title("Boston Housing Data (LSTAT vs MEDV)")

plt.xlim([0,40])
plt.ylim(0,55)

plt.show()
```



In [20]:
```
plot_prediction(X,y);
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-df2291097a60> in <module>
----> 1 plot_prediction(X,y);

TypeError: plot_prediction() missing 2 required positional arguments: 'xp' and 'yp'
```

In [21]:
```
# simple linear regression
regr = linear_model.LinearRegression()
regr.fit(X, y)

# model output for the input data
y_test = regr.predict(X_test)

# plot the result
plot_prediction(X, y, X_test, y_test, "MEDV=%.3fLSTAT + %.3f" % (regr.coef_[0],regr.intercept_),'Housing-LinearRegression.png')
```



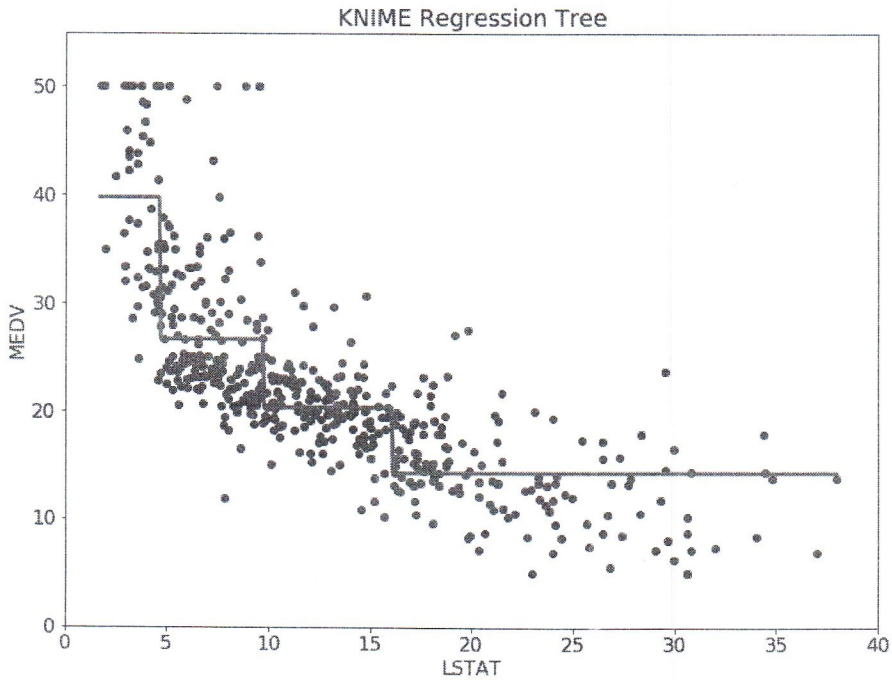
In [ ]:
```
knn = neighbors.KNeighborsRegressor(5, weights='uniform')

knn.fit(X, y)

y_test = knn.predict(X_test)

plot_prediction(X,y,X_test,y_test, "KNeighborsRegressor (k = 5, weights = 'uniform')");
```

In [ ]:
```
knn = neighbors.KNeighborsRegressor(10, weights='distance')

knn.fit(X, y)

y_test = knn.predict(X_test)

plot_prediction(X,y,X_test,y_test, "KNeighborsRegressor (k = 10, weights = 'distance')");
```

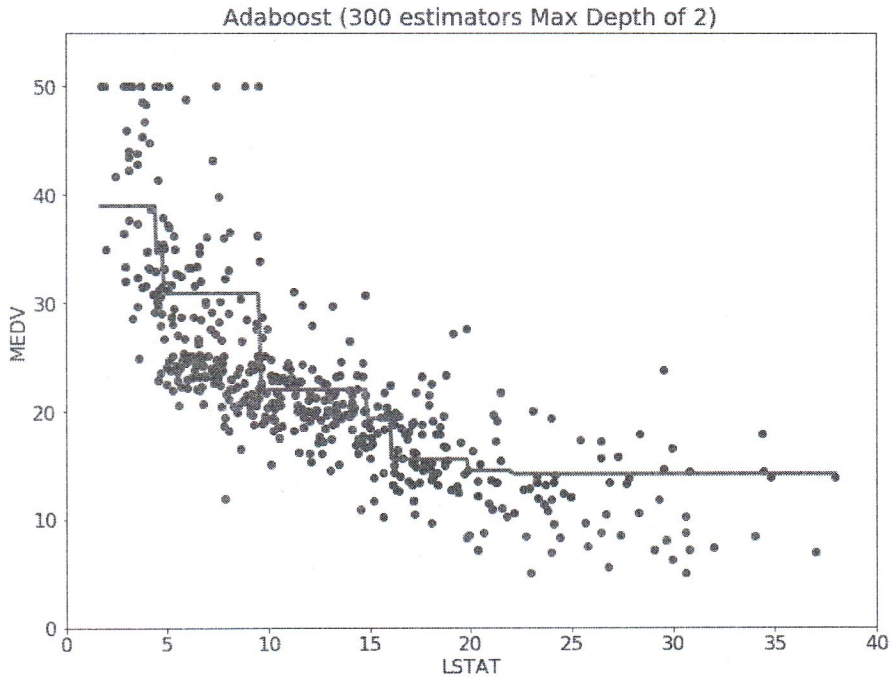
In [4]:
```
regression_tree = DecisionTreeRegressor(max_depth=2)
regression_tree.fit(X,y.ravel())
y_test = regression_tree.predict(X_test)

plot_prediction(X,y,X_test,y_test, "Regression Tree (Max Depth=2)");
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-065e1226a1f0> in <module>
      1 regression_tree = DecisionTreeRegressor(max_depth=2)
----> 2 regression_tree.fit(X,y.ravel())
      3 y_test = regression_tree.predict(X_test)
      4
      5 plot_prediction(X,y,X_test,y_test, "Regression Tree (Max Depth=2)");

NameError: name 'X' is not defined
```

In [48]:
```
# implementation of the model developed using KNIME

def TreeModel(x):
    if (x<=9.725):
        if (x<=4.65):
            return 39.718
        else:
            return 26.6463
    else:
        if (x<=16.08):
            return 20.302
        else:
            return 14.2618


y_test = np.vectorize(TreeModel)(X_test)

plot_prediction(X,y,X_test,y_test, "KNIME Regression Tree");
```



In [49]:
```
adaboost_model = AdaBoostRegressor(DecisionTreeRegressor(max_depth=2), n_estimators=300, random_state=random_seed)
adaboost_model.fit(X, y.ravel())

y_test = adaboost_model.predict(X_test)

plot_prediction(X, y, X_test, y_test, "Adaboost (300 estimators Max Depth of 2)");
```


Adaboost (300 estimators Max Depth of 2)
MEDV
LSTAT
0
10
20
30
40
50
0
5
10
15
20
25
30
35
40

REGRESSION EXAMPLE

In [2]:
```
# import all the required libraries and put matplotlib in inline mode to plot on the notebook
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline

# linear regression
from sklearn import linear_model

# nearest neighbor
from sklearn import neighbors

# regression trees (simple and ensemble)
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

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

In [10]:
```
# define the random seed if needed
random_seed = 1234
```

In [3]:
```
dataset = pd.read_csv('housing.csv')
dataset.columns
dataset.describe()
```

Out[3]:

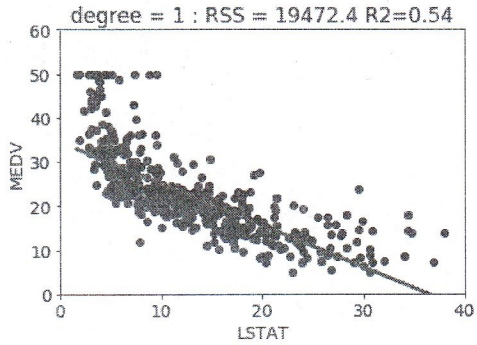
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

In [4]:
```
def plot_approximator(x,y,xp,yp,xlim,ylim,title=""):
    """Plots the original data (x,y) and a set of point (xp,yp) showing the model approximation"""
    font = {'family' : 'sans',
        'size'   : 14}
    plt.rc('font', **font)

    plt.scatter(x, y,  color='blue')
    plt.plot(xp, yp, color='red', linewidth=3)
    plt.xlabel("LSTAT")
    plt.ylabel("MEDV")

    if (title!=""):
        plt.title(title)

    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.figure(figsize=(12,9))
    plt.show()
```

In [5]:
```
def compute_polynomial_model(x, coef, intercept):
    """Compute the polynomial given the input x, the intercept and the coefficients"""
    min_x = min(x)
    max_x = max(x)
    xp = np.arange(min_x, max_x, (max_x-min_x)/100.0)

    x = xp
    yp = intercept

    for w in coef:
        yp = yp + w * x
        x = x * xp
    return xp,yp
```

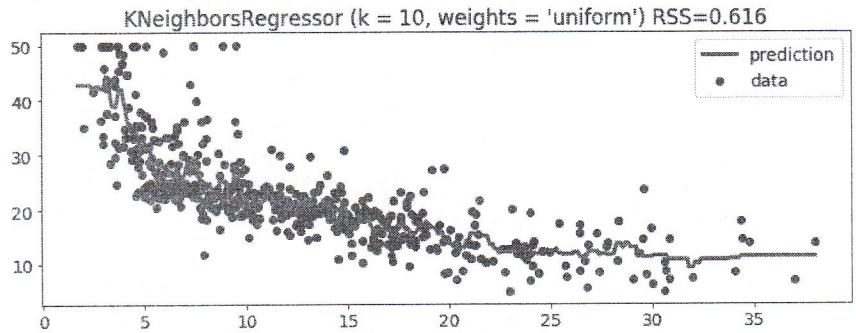
In [6]:
```
# compute the data inputs
dataset_train_x = dataset['LSTAT'].values
x = dataset_train_x.reshape(506, 1)

# compute the data output
dataset_train_y = dataset.MEDV.values
y = dataset_train_y.reshape(506, 1)

# apply simple linear regression to fit the data
regr = linear_model.LinearRegression()
regr.fit(x, y)

# model output for the input data
yp = regr.predict(x)

# compute the model output as a line
xm,ym = compute_polynomial_model(x[:,0],regr.coef_, regr.intercept_)

# compute rss cost
rss = sum((yp-y)*(yp-y))

# the cost as R^2
r2 = regr.score(x,y)

title = "degree = 1 : RSS = "+str(round(rss[0],1)) + " R2="+str(round(r2,2))
```

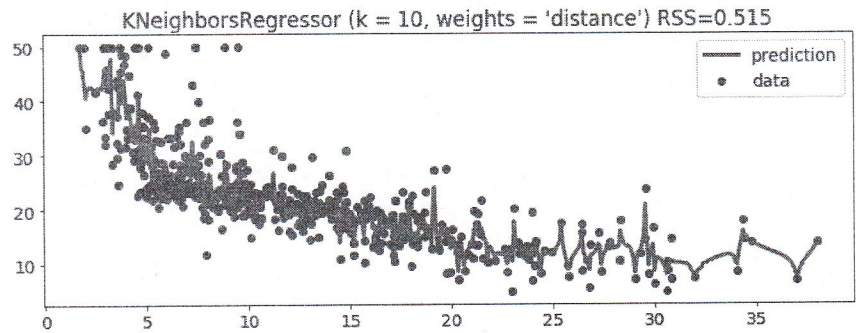
```
# plot the result
plot_approximator(x[:,0],y,xm,ym,title)
```



In [7]:
```
x = dataset['LSTAT'].values.reshape(506, 1)
y = dataset['MEDV'].values.reshape(506, 1)
T = np.linspace(np.min(x), np.max(x), 500)[:, np.newaxis]
```

In [8]:
```
n_neighbors = 10

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors, weights=weights)
    y_ = knn.fit(x, y).predict(T)

    scores = cross_val_score(knn, x, y, cv=KFold(n_splits=10, random_state=1234, shuffle=True))

    plt.figure(figsize=(12,9))
    font = {'family' : 'sans', 'size'   : 14}
    plt.rc('font', **font)
    plt.subplot(2, 1, i + 1)
    plt.scatter(x, y, c='blue', label='data');
    plt.plot(T, y_, c='red', linewidth=3, label='prediction');
    plt.axis('tight');
    plt.legend();
    plt.title("KNeighborsRegressor (k = %i, weights = '%s') RSS=%.3f" % (n_neighbors, weights, np.average(scores)));

plt.show();
```





In [45]:
```
# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=2), n_estimators=300, random_state=random_seed)
```

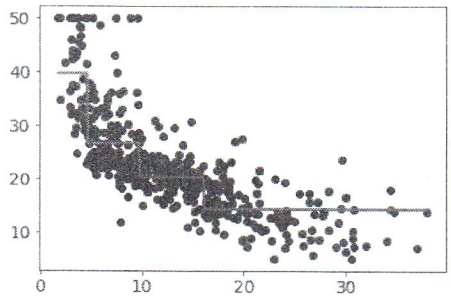
In [46]:
```
X = dataset['LSTAT'].values.reshape(-1,1)

regr_1.fit(X, y.ravel())
regr_2.fit(X, y.ravel())
```

Out[46]:
```
AdaBoostRegressor(base_estimator=DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best'),
         learning_rate=1.0, loss='linear', n_estimators=300,
         random_state=1234)
```

In [47]:
```
# Predict
Ty_1 = regr_1.predict(T)
Ty_2 = regr_2.predict(T)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(T, Ty_1, c="g", label="n_estimators=1", linewidth=2)
```

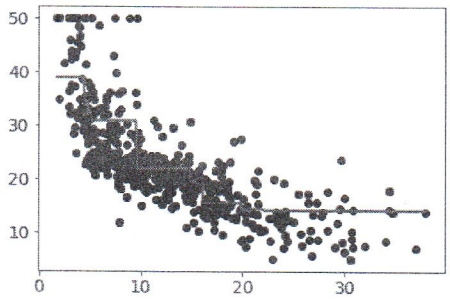
```
# plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
# plt.xlabel("data")
# plt.ylabel("target")
# plt.title("Boosted Decision Tree Regression")
# plt.legend()
# plt.show()
```

Out[47]: [<matplotlib.lines.Line2D at 0x1a18fb9f98>]



In [48]:
```
# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(T, Ty_2, c="g", label="n_estimators=1", linewidth=2)
# plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
# plt.xlabel("data")
# plt.ylabel("target")
# plt.title("Boosted Decision Tree Regression")
# plt.legend()
# plt.show()
```

Out[48]: [<matplotlib.lines.Line2D at 0x1a191a7128>]



In [49]:
```
def TreeModel(x):
    if (x<=9.725):
        if (x<=4.65):
            return 39.718
        else:
            return 26.6463
    else:
        if (x<=16.08):
            return 20.302
        else:
            return 14.2618
```

In [51]:
```
TM_y = X.apply(TreeModel)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-51-b4af8e2eec1a> in <module>
----> 1 TM_y = X.apply(TreeModel)

AttributeError: 'numpy.ndarray' object has no attribute 'apply'
```

In [52]:
```
X.shape
```

Out[52]: (506, 1)

In [53]:
```
X.values.shape
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-53-bd5b9b37fd09> in <module>
----> 1 X.values.shape

AttributeError: 'numpy.ndarray' object has no attribute 'values'
```

In [54]:
```
X.values
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-54-7d01be6f817a> in <module>
----> 1 X.values

AttributeError: 'numpy.ndarray' object has no attribute 'values'
```

In [57]:
```
X.ravel().apply(TreeModel)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-57-83f5b2fa9183> in <module>
----> 1 X.ravel().apply(TreeModel)

AttributeError: 'numpy.ndarray' object has no attribute 'apply'
```

In [58]:
```
np.array(map(TreeModel, x))
```

Out[58]:
```
array(<map object at 0x1a19258f60>, dtype=object)
```

In [59]:
```
TM_y = np.array(map(TreeModel, x))
```

In [60]:
```
TM_y
```

Out[60]:
```
array(<map object at 0x1a19258198>, dtype=object)
```

In [61]:
```
TM_y.ravel()
```

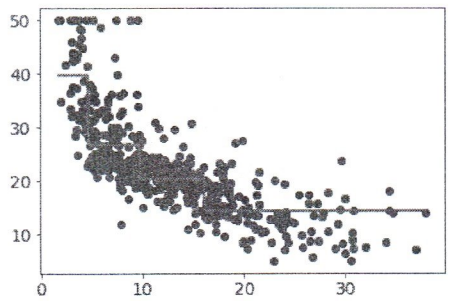
Out[62]:
```
array([<map object at 0x1a19258198>], dtype=object)
```

In [63]:
```
vTreeModel = np.vectorize(TreeModel)
```

In [69]:
```
TM_y = np.vectorize(TreeModel)(T)
```

In [70]:
```
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(T, TM_y.ravel(), c="g", label="n_estimators=1", linewidth=2)
```

Out[70]:
```
[<matplotlib.lines.Line2D at 0x1a19594a90>]
```



In [ ]:

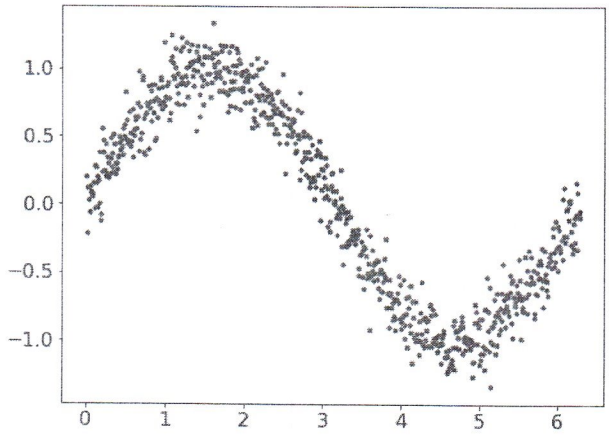
Ridge & Lasso Regression

We will be using a small dataset to predict the target variable using basic linear regression, ridge regression and Lasso. We import the library functions we need and disactivate the warnings.

In [1]:
```
# import all the required libraries and put matplotlib in inline mode to plot on the notebook
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import model_selection
from sklearn.metrics import r2_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV

from sklearn.preprocessing import PolynomialFeatures
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')

saved_figsize = plt.rcParams['figure.figsize']
```

We generate the datapoints using a simple sinus and plot the data.

In [2]:
```
X = np.array([i*np.pi/180 for i in np.arange(0,360,.5)]).reshape(-1,1)

np.random.seed(10)  #Setting seed for reproducability

y = np.sin(X)+np.random.normal(0,0.15,len(X)).reshape(-1,1)

# dataset = pd.DataFrame({'x':X, 'y':y})

##
## Split train and test
##

plt.rcParams['figure.figsize'] = (8.0, 6.0)
font = {'family' : 'sans', 'size'   : 16}
plt.rc('font', **font)
plt.plot(X,y,'.', color='blue');

## We are going to use polynomials of at most degree 12
max_polynomial=12
```



We define the usual utility functions to plot the resulting approximation.

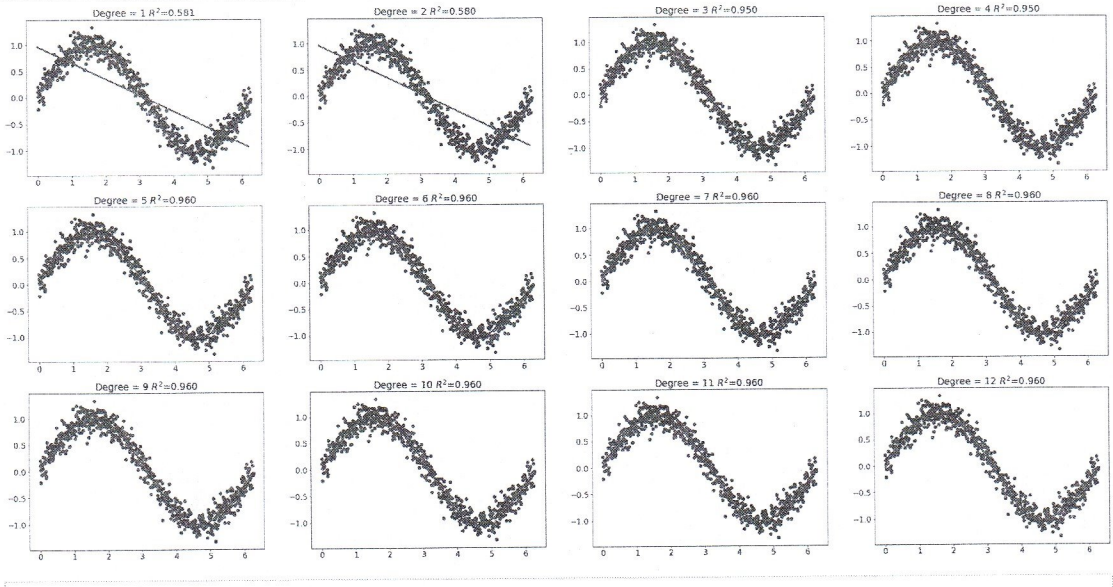
In [3]:
```
def plot_scatter(x,y,xp,yp,title=""):
    """Plots the original data (x,y) and a set of point (xp,yp) showing the model approximation"""
    font = {'family' : 'sans',
        'size'   : 14}
    plt.rc('font', **font)

    plt.scatter(x, y,  color='blue')
    plt.plot(xp, yp, color='red', linewidth=3)
    plt.xlabel("x")
    plt.ylabel("y")

    if (title!=""):
        plt.title(title)

    plt.show()
```

In [4]:
```
def ApplyRegression(regressor,evaluation='crossvalidation',do_plot=True):
    max_polynomial = 12

    if (do_plot):
        f, axarr = plt.subplots(3, 4)
        plt.rcParams['figure.figsize'] = (40.0, 20.0)
        font = {'family' : 'sans', 'size'   : 16}
        plt.rc('font', **font)

    r2_values = []
    max_coefficients = []
```

```
    for degree in range(1,max_polynomial+1):
        if (degree!=1):
            polynomial = PolynomialFeatures(degree=degree, include_bias=False)
            X_polynomial = polynomial.fit_transform(X)
            Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
            Xplot_polynomial = polynomial.fit_transform(Xplot)
        else:
            Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
            Xplot_polynomial = Xplot
            X_polynomial = X

        if (evaluation=='crossvalidation'):

            # apply crossvalidation
            score = cross_val_score(regressor, X_polynomial, y, cv=KFold(n_splits=10, shuffle=True, random_state=1234))

            # score the model
            r2 = score.mean()
            r2_values.append(np.mean(score))

            # create the model on the entire data
            regressor.fit(X_polynomial, y);

            max_coefficients.append(np.amax(np.abs(regressor.coef_)))

            # compute the model on the plotted poins
            yplot = regressor.predict(Xplot_polynomial)
        elif (evaluation=='holdout'):

            X_train, X_test, y_train, y_test = model_selection.train_test_split(X_polynomial, y, test_size=0.33, random_state=1

            regressor.fit(X_train, y_train);

            max_coefficients.append(np.amax(np.abs(regressor.coef_)))

            # score the model
            yp = regressor.predict(X_test)
            r2 = r2_score(y_test, yp)
            r2_values.append(r2)

        elif (evaluation=='train'):

            # create the model on the entire data
            regressor.fit(X_polynomial, y);

            #evaluation on the train
            yp = regressor.predict(X_polynomial)
            r2 = r2_score(y, yp)
            r2_values.append(r2)

            max_coefficients.append(np.amax(np.abs(regressor.coef_)))
        else:
            raise Exception('evaluation valid values:\n- crossvalidation\n- holdout\n- train')

        # compute the model on the plotted poins
        yplot = regressor.predict(Xplot_polynomial)

        if (do_plot):
            target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
            target_plot.scatter(X[:,0],y, color="blue")
            target_plot.set_title("Degree = %d $R^2$=%.3f"%(degree,r2))
            target_plot.plot(Xplot[:,0],yplot, color="red",linewidth=3)

    return r2_values, max_coefficients, regressor.coef_
```

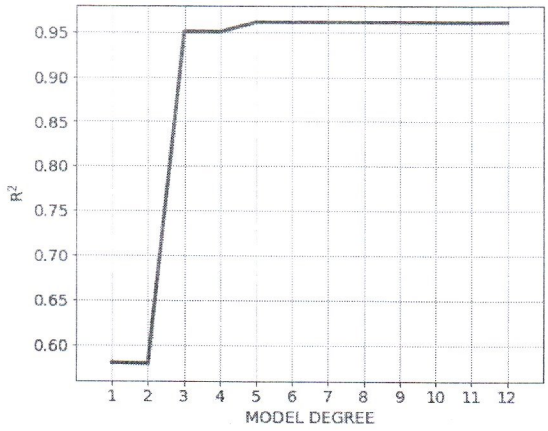
Let's plot various approximations using different polynomials, different regression methods and crossvalidation.

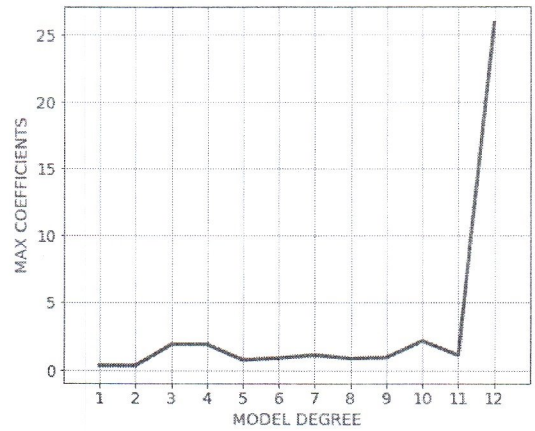
In [6]:
```
r2_values, max_coefficients, weights = ApplyRegression(LinearRegression(),'crossvalidation')
```



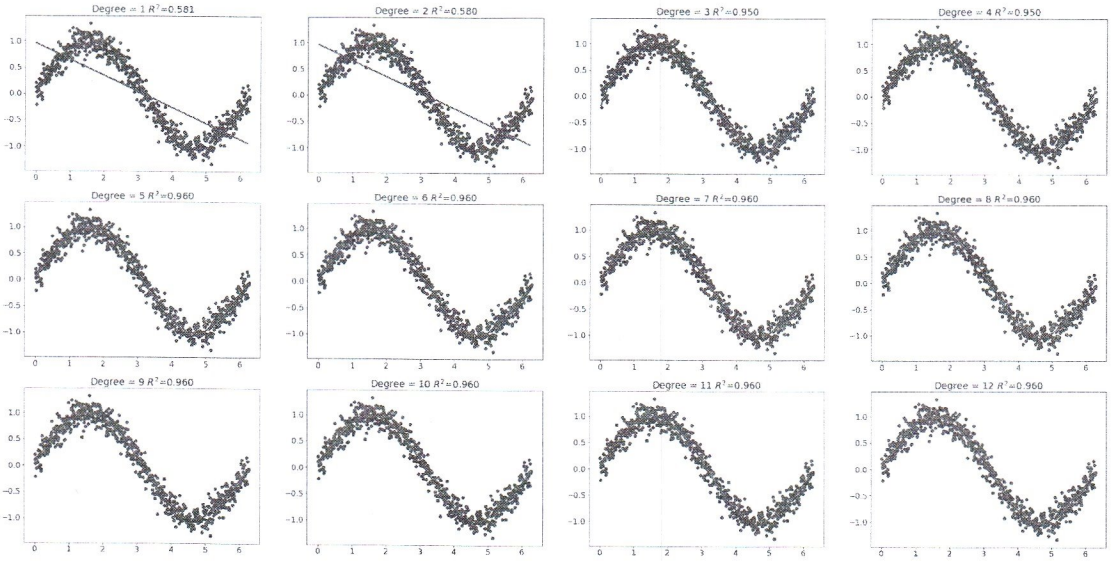
In [10]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,13])
```

```
axarr[0].set_xticks(range(1,13))
axarr[0].grid()
axarr[0].plot(range(1,13), r2_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("MAX COEFFICIENTS")
axarr[1].set_xlim([0,13])
axarr[1].set_xticks(range(1,13))
axarr[1].grid()
axarr[1].plot(range(1,13), max_coefficients, color="blue",linewidth=3);
```



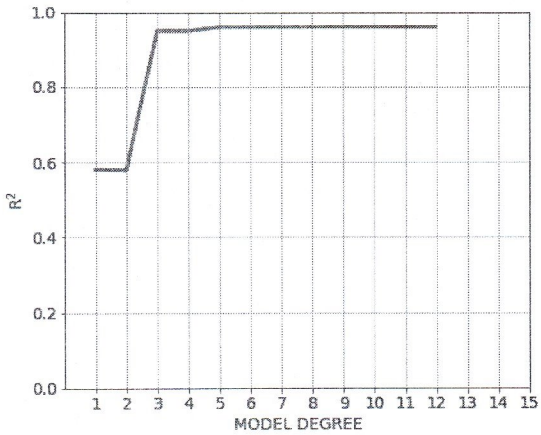


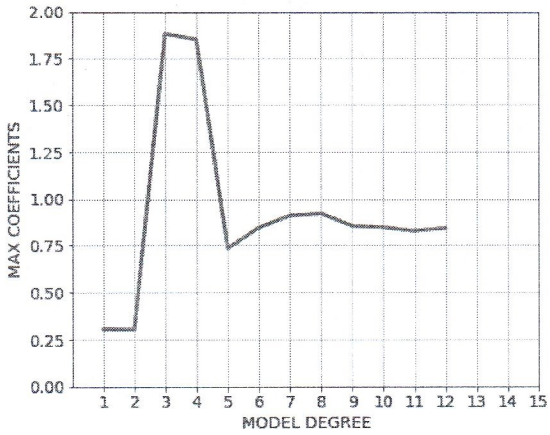
In [21]:
```
r2_values, max_coefficients, weights = ApplyRegression(Ridge(alpha=0.1,random_state=1234),'crossvalidation')
```



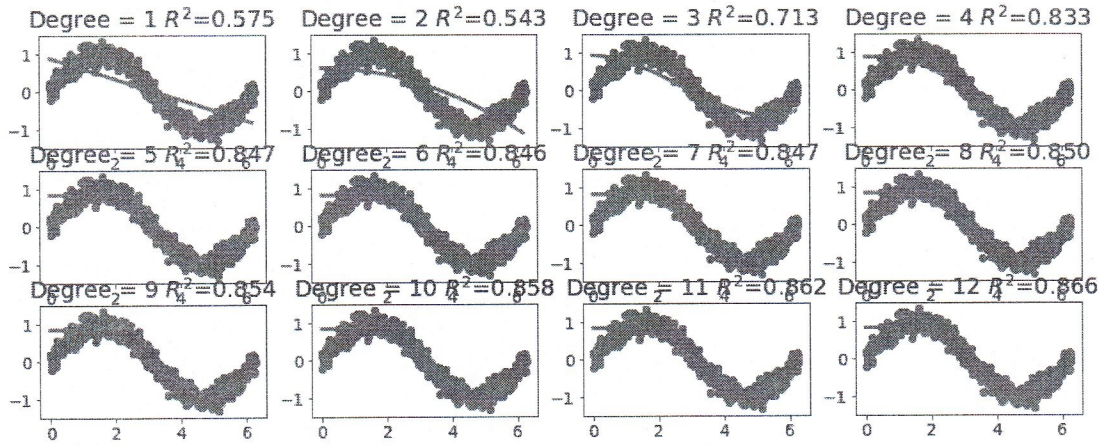
Let's plot the largest absolute weight value in each run.

In [22]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,13])
axarr[0].set_ylim([0,1])
axarr[0].set_xticks(range(1,16))
axarr[0].grid()
axarr[0].plot(range(1,13), r2_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("MAX COEFFICIENTS")
axarr[1].set_xlim([0,13])
axarr[1].set_ylim([0,2])
axarr[1].set_xticks(range(1,16))
axarr[1].grid()
axarr[1].plot(range(1,13), max_coefficients, color="blue",linewidth=3);
```

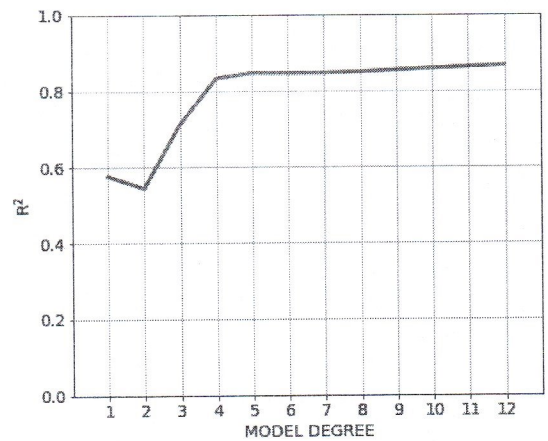


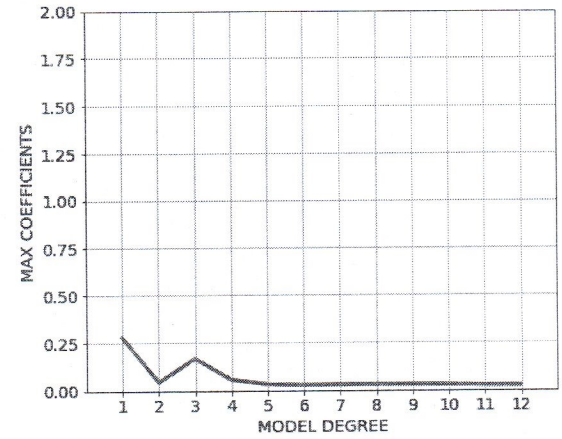


In [23]:
```
r2_values, max_coefficients, weights = ApplyRegression(Lasso(alpha=0.1, random_state=1234),'crossvalidation')
```



In [24]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 2)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,13])
axarr[0].set_ylim([0,1])
axarr[0].set_xticks(range(1,13))
axarr[0].grid()
axarr[0].plot(range(1,13), r2_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("MAX COEFFICIENTS")
axarr[1].set_xlim([0,13])
axarr[1].set_ylim([0,2])
axarr[1].set_xticks(range(1,13))
axarr[1].grid()
axarr[1].plot(range(1,13), max_coefficients, color="blue",linewidth=3);
```





The Effect of α on the Weights

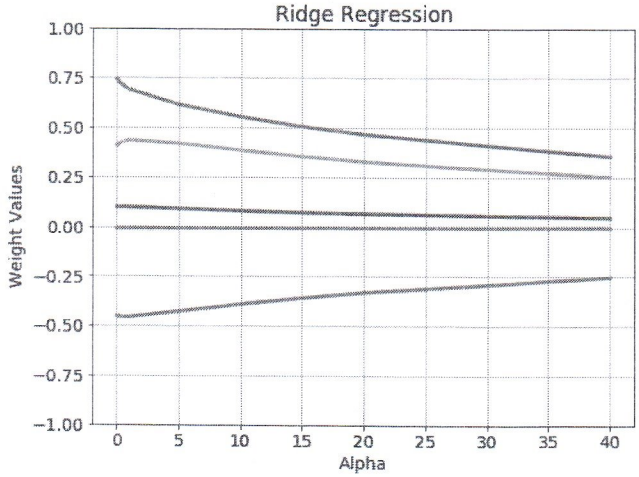
Let's see how the final weights vary based on the value of α.

In [41]:
```
set_of_alphas = [0.01, 0.05, 0.1, 0.5, 1, 5, 10, 15, 20, 40]
degree = 5

ridge_weights = []
lasso_weights = []

for alpha in set_of_alphas:
```

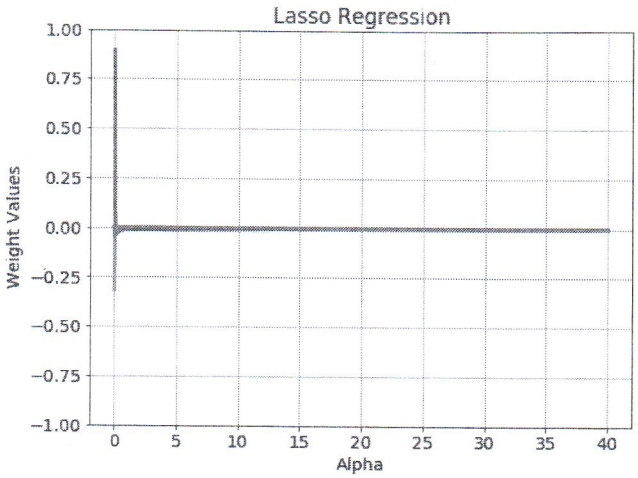
```
#     xm, ym, yp, rss, r2, regr = compute_polynomial_ridge(dataset,'x', 'y', 10, a=alpha)
    # create the model on the entire data
    polynomial = PolynomialFeatures(degree=degree, include_bias=False)
    X_polynomial = polynomial.fit_transform(X)

    ridge = Ridge(alpha=alpha, random_state=1234, max_iter=10000)
    ridge.fit(X_polynomial, y)
    ridge_weights.append(ridge.coef_[0])

    lasso = Lasso(alpha=alpha, random_state=1234)
    lasso.fit(X_polynomial, y)
    lasso_weights.append(lasso.coef_)
```

In [42]:
```
ridge_matrix = pd.DataFrame(data=ridge_weights)
ridge_matrix.columns = ['w'+str(i) for i in range(degree)]

lasso_matrix = pd.DataFrame(data=lasso_weights)
lasso_matrix.columns = ['w'+str(i) for i in range(degree)]
```

In [43]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.rcParams['figure.figsize'] = (8.0, 6.0)

for i in range(degree):
    plt.plot(set_of_alphas, ridge_matrix['w'+str(i)], linewidth=3)

plt.ylim([-1,1])
plt.xlabel('Alpha')
plt.ylabel('Weight Values')
plt.title("Ridge Regression")
plt.grid()
```



In [44]:
```
font = {'family' : 'sans', 'size'   : 14}
plt.rc('font', **font)
plt.rcParams['figure.figsize'] = (8.0, 6.0)

for i in range(degree):
    plt.plot(set_of_alphas, lasso_matrix['w'+str(i)], linewidth=3)

plt.ylim([-1,1])
plt.xlabel('Alpha')
plt.ylabel('Weight Values')
plt.title("Lasso Regression")
plt.grid()
```



Selecting the Regularization Coefficient α

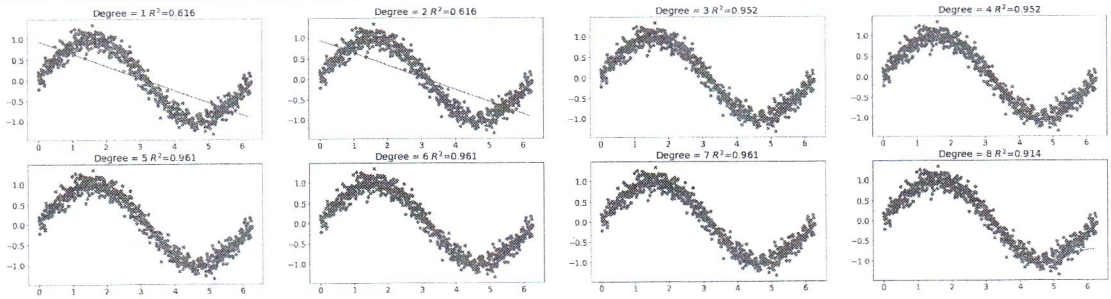
When applying regularization we need to select the value of α which works as a hyperparameter. For this purpose, we cannot just run different regression using the entire data set with different values of α but we need to evaluate each α by applying either holdout evaluation or crossover evaluation on a training set; then, once we selected the alpha, we can evaluate the value over the test set. Note that, by applying holdout on the

training set we generate two sets, the training set and the validation set (sometimes called the dev or development set). In the following example, we apply RidgeCV and LassoCV, two procedures available in Scikit-Learn that perform crossvalidation to compute the best value of α.

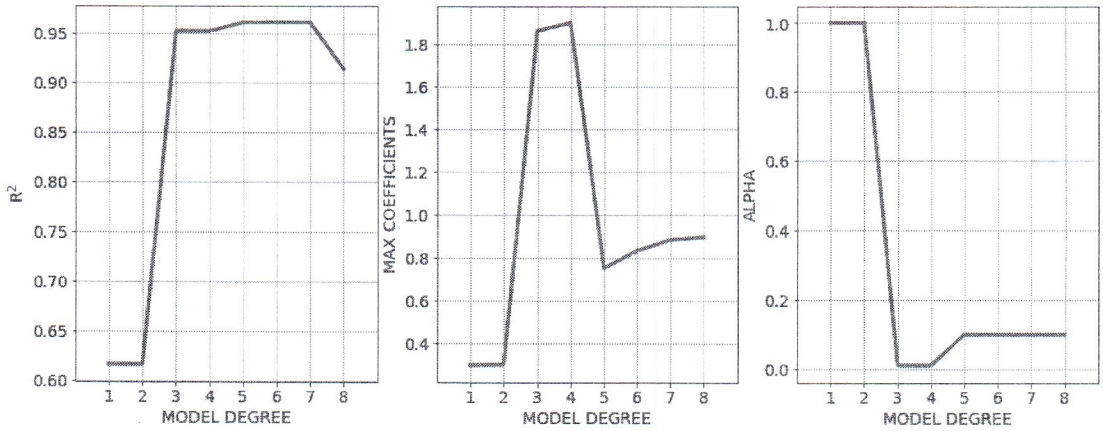
In [45]:
```
def ApplyRegularizedCVRegression(regressor, do_plot=True):
    max_polynomial = 8

    if not isinstance(regressor, RidgeCV) and not isinstance(regressor, LassoCV):
        raise Exception('Works with RidgeCV or LassoCV')

    if (do_plot):
        f, axarr = plt.subplots(2, 4)
        plt.rcParams['figure.figsize'] = (40.0, 10.0)
        font = {'family' : 'sans', 'size'   : 16}
        plt.rc('font', **font)

    r2_values = []
    max_coefficients = []
    computed_alphas = []

    for degree in range(1,max_polynomial+1):
        if (degree!=1):
            polynomial = PolynomialFeatures(degree=degree, include_bias=False)
            X_polynomial = polynomial.fit_transform(X)
            Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
            Xplot_polynomial = polynomial.fit_transform(Xplot)
        else:
            Xplot = np.arange(np.min(X),np.max(X),0.1).reshape(-1,1)
            Xplot_polynomial = Xplot
            X_polynomial = X

        X_train, X_test, y_train, y_test = model_selection.train_test_split(X_polynomial, y, test_size=0.33, random_state=1234)

        clf = regressor.fit(X_train, y_train)

        max_coefficients.append(np.amax(np.abs(regressor.coef_)))

        yp = regressor.predict(X_test)
        r2 = r2_score(y_test, yp)
        r2_values.append(r2)

        computed_alphas.append(regressor.alpha_)

        # compute the model on the plotted poins
        yplot = regressor.predict(Xplot_polynomial)

        if (do_plot):
            target_plot = axarr[int((degree-1)/4),int((degree-1)%4)]
            target_plot.scatter(X[:,0],y, color="blue")
            target_plot.set_title("Degree = %d $R^2$=%.3f"%(degree,r2))
            target_plot.plot(Xplot[:,0],yplot, color="red")

    return r2_values, max_coefficients, computed_alphas
```

First, let's try Ridge regression using crossvalidation to compute, for each polynomial, the best alpha among the ones provided as a parameter.

In [48]:
```
r2_values, max_coefficients, computed_alphas = ApplyRegularizedCVRegression(RidgeCV(alphas=[1e-10,1e-5,1e-4,1e-3, 1e-2, 1e-1, 1
```

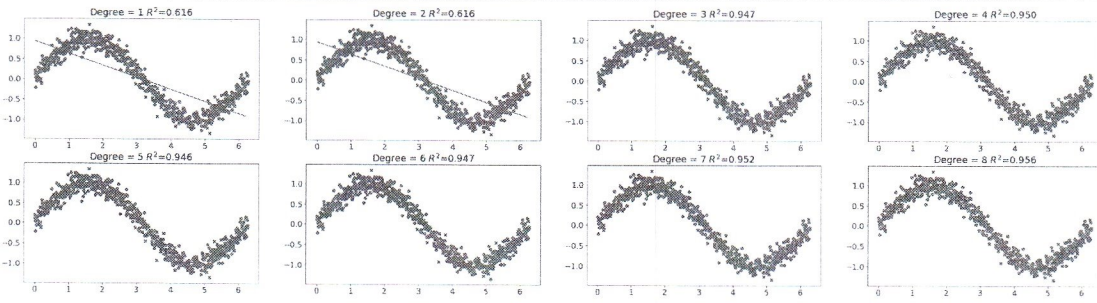


In [49]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 3)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,len(r2_values)+1])
axarr[0].set_xticks(range(1,len(r2_values)+1))
axarr[0].grid()
axarr[0].plot(range(1,len(r2_values)+1), r2_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("MAX COEFFICIENTS")
axarr[1].set_xlim([0,len(r2_values)+1])
axarr[1].set_xticks(range(1,len(r2_values)+1))
axarr[1].grid()
axarr[1].plot(range(1,len(r2_values)+1), max_coefficients, color="blue",linewidth=3);

axarr[2].set_xlabel("MODEL DEGREE")
axarr[2].set_ylabel("ALPHA")
axarr[2].set_xlim([0,len(r2_values)+1])
axarr[2].set_xticks(range(1,len(r2_values)+1))
axarr[2].grid()
axarr[2].plot(range(1,len(r2_values)+1), computed_alphas, color="blue",linewidth=3);
```

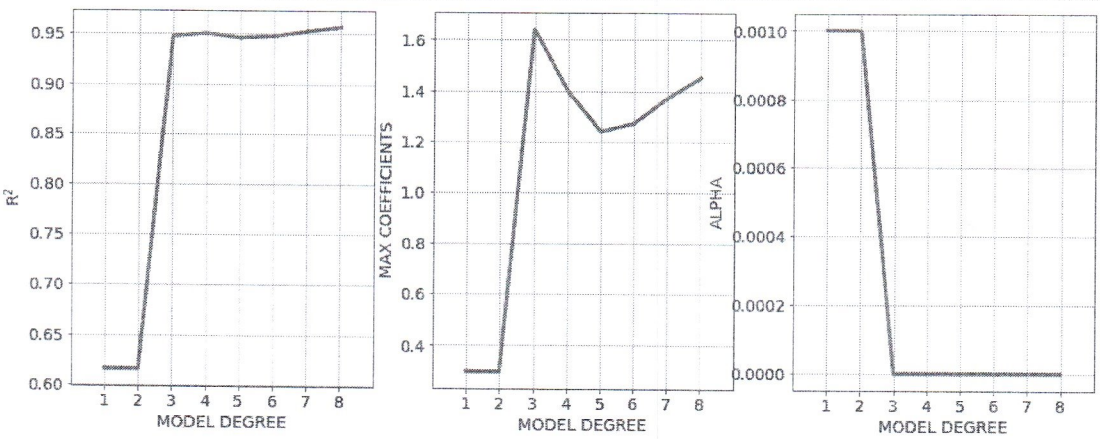


Let's repeat everything using Lasso regression.

In [51]:
```
r2_values, max_coefficients, computed_alphas = ApplyRegularizedCVRegression(LassoCV(alphas=[1e-10,1e-5,1e-4,1e-3, 1e-2, 1e-1, 1
```



In [52]:
```
plt.rcParams['figure.figsize'] = (16.0, 6.0)
font = {'family' : 'sans', 'size':14}
plt.rc('font', **font)
f, axarr = plt.subplots(1, 3)

axarr[0].set_xlabel("MODEL DEGREE")
axarr[0].set_ylabel("R$^2$")
axarr[0].set_xlim([0,len(r2_values)+1])
axarr[0].set_xticks(range(1,len(r2_values)+1))
axarr[0].grid()
axarr[0].plot(range(1,len(r2_values)+1), r2_values, color="blue",linewidth=3);

axarr[1].set_xlabel("MODEL DEGREE")
axarr[1].set_ylabel("MAX COEFFICIENTS")
axarr[1].set_xlim([0,len(r2_values)+1])
axarr[1].set_xticks(range(1,len(r2_values)+1))
axarr[1].grid()
axarr[1].plot(range(1,len(r2_values)+1), max_coefficients, color="blue",linewidth=3);

axarr[2].set_xlabel("MODEL DEGREE")
axarr[2].set_ylabel("ALPHA")
axarr[2].set_xlim([0,len(r2_values)+1])
axarr[2].set_xticks(range(1,len(r2_values)+1))
axarr[2].grid()
axarr[2].plot(range(1,len(r2_values)+1), computed_alphas, color="blue",linewidth=3);
```



TF-IDF Computation Example

This is a very simple example of TF-IDF computation. It is divided in two section. The first part discusses TF-IDF as presented during the last edition of the course (2020/2021) and should be used to prepare for the exam. The second part shows how scikit-learn computes TF-IDF and discusses the difference between our basic version discussed during the course and the actual implementation available in scikit-learn. This second version won't be a part of any written exam and it is included for completeness.

References

- https://scikit-learn.org/stable/modules/feature_extraction.html#tfidf-term-weighting
- https://towardsdatascience.com/natural-language-processing-feature-engineering-using-tf-idf-e8b9d00e7e76

First, we import the libraries we will need.

In [54]:
```
import pandas as pd
import numpy as np
import math
import nltk

nltk.download('stopwords')

from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/pierlucalanzi/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Next we define a small corpus of three documents (the same used in the exam problem). This is the same corpus used in the problem from June 28 2019 exam that has been discussed at length in the forum.

In [55]:
```
corpus = [
    'A time to plant and a time to reap',
    'Time for you and time for me',
    'Fly Time'
]
```

From the corpus we have to extract the "bag of word" representation of the document which does not consider the order of the words but represents the documents simply as a set of words. Note that to extract the representation we also eliminate all the stopwords (those words that are not really interesting since they don't provide much information).

In [56]:
```
def compute_bag_of_words(corpus):
    bag_of_words = []
    for text in corpus:
        bag_of_words.append([x.lower() for x in text.split(' ') if not (x.lower() in stopwords.words('english'))])
    return bag_of_words
```

In [57]:
```
bag_of_words = compute_bag_of_words(corpus)
```

In [58]:
```
bag_of_words
```

Out[58]:
```
[['time', 'plant', 'time', 'reap'], ['time', 'time'], ['fly', 'time']]
```

Given a corpus we also need to compute the dictionary that will be used to create the table that represents the documents. The dictionary is simply the set of all the words contained in the corpuse (without any stopword that has been already eliminated)

In [59]:
```
def compute_dictionary(corpus):
    dictionary = set(bag_of_words[0])

    for bow in bag_of_words[1:]:
        dictionary = dictionary.union(set(bow))
    return dictionary
```

In [60]:
```
dictionary = compute_dictionary(corpus)
```

In [61]:
```
dictionary
```

Out[61]:
```
{'fly', 'plant', 'reap', 'time'}
```

As the very next step we can compute the "term frequency", that is, how frequent every word in the dictionary appears in each document.

In [62]:
```
def compute_tf_count(bag_of_words,dictionary):

    word_count = []

    for document in bag_of_words:
        document_word_count = dict.fromkeys(dictionary, 0)

        for w in document:
            document_word_count[w] = document_word_count[w]+1

        word_count.append(document_word_count)

    return word_count
```

In [63]:
```
word_count = compute_tf_count(bag_of_words,dictionary)
```

In [64]:
```
word_count
```

Out[64]:
```
[{'plant': 1, 'reap': 1, 'time': 2, 'fly': 0},
 {'plant': 0, 'reap': 0, 'time': 2, 'fly': 0},
 {'plant': 0, 'reap': 0, 'time': 1, 'fly': 1}]
```

We can now normalize these values and compute the actual frequency of each word. Note that we did not do this in the example done in class since we did not have the whole dictionary.

In [65]:
```
def compute_tf(document_word_count):
```

```
    document_tf = {}

    number_of_words = sum(document_word_count.values())

    for word, count in document_word_count.items():
        document_tf[word] = count / float(number_of_words)

    return document_tf
```

In [66]:
```
tf = []
for document_word_count in word_count:
    tf.append(compute_tf(document_word_count))
```

In [67]:
```
tf
```

Out[67]:
```
[{'plant': 0.25, 'reap': 0.25, 'time': 0.5, 'fly': 0.0},
 {'plant': 0.0, 'reap': 0.0, 'time': 1.0, 'fly': 0.0},
 {'plant': 0.0, 'reap': 0.0, 'time': 0.5, 'fly': 0.5}]
```

Let's now create and actual table for the counts and frequencies.

In [68]:
```
def create_df(tf,dictionary):
    data = {}
    for word in dictionary:

        word_tf = []

        for document_tf in tf:
            word_tf.append(document_tf[word])

        data[word] = word_tf

    df = pd.DataFrame(data)

    return df
```

In [69]:
```
df_count = create_df(word_count,dictionary)
df_tf = create_df(tf,dictionary)
```

In [70]:
```
df_count
```

Out[70]:

	plant	reap	time	fly
0	1	1	2	0
1	0	0	2	0
2	0	0	1	1

In [71]:
```
df_tf
```

Out[71]:

	plant	reap	time	fly
0	0.25	0.25	0.5	0.0
1	0.00	0.00	1.0	0.0
2	0.00	0.00	0.5	0.5

We can now compute the IDF for every word in the dictionary. The formula presented in class for IDF is,

$$idf(w) = log(M/k)$$

where w is a word in the corpus dictionary, M is the number of documents in the corpus, and k is the number of documents in which word w appears. As usual we are going to use logarithm base 2 for the computations. Note that in previous editions of the course we used a slightly different formula. Note also that any type of logarithm might be used, we decide to use base 2 so that we can compare our results without worrying what base one has used.

In [72]:
```
def compute_idf(df):
    idf = {}

    # number of documents
    M = len(df)

    for word in df.columns:

        # number of documents in which the word appears

        k = sum(df[word]>0.0)

        idf[word] = math.log(M/k,2)

    return idf
```

In [73]:
```
idf = compute_idf(df_tf)
```

In [74]:
```
idf
```

Out[74]:
```
{'plant': 1.5849625007211563,
 'reap': 1.5849625007211563,
 'time': 0.0,
 'fly': 1.5849625007211563}
```

Note that since the word "time" appears in all the documents its IDF value is zero. We can finally can compute the TF-IDF representation of the corpus as a table that we can use for instance to cluster the documents.

In [75]:
```
tf_idf = df_tf.copy()
for word in tf_idf.columns:
    tf_idf[word] = tf_idf[word]*idf[word]
```

In [76]:
```
tf_idf
```

Out[76]:

	plant	reap	time	fly
0	0.396241	0.396241	0.0	0.000000
1	0.000000	0.000000	0.0	0.000000
2	0.000000	0.000000	0.0	0.792481

The results are similar to those available in the solution of the June 28 2019 exams. The difference in the values is due to the different formula that in 2018/2019 was used which contained a smoothing factor.

TF-IDF using Scikit-Learn

Scikit-learn has its own set of functions to preprocess a corpus to generate TF-IDF representation so that we can avoid doing everything by hand. The following few lines replicate the entire process we just performed.

In [78]:
```
vectorizer = TfidfVectorizer(stop_words=stopwords.words('english'))
vectors = vectorizer.fit_transform(corpus)
sklearn_idf_values = vectorizer.idf_
sklearn_feature_names = vectorizer.get_feature_names()
dense = vectors.todense()
denselist = dense.tolist()

sklearn_tfidf = pd.DataFrame(denselist, columns=feature_names)
```

If we print the table produced by scikit-learn we will note that the values are quite different. This because scikit-learn normalizes the term frequencies using the formula,

$$\mathrm{tf}(w) = \frac{n_w}{\sqrt{\sum n_i}}$$

It applies a smoothed version of IDF, that is,

$$\mathrm{idf}(w) = \log(M + 1/k + 1) + 1$$

where w is a word in the corpus dictionary, M is the number of documents in the corpus, k is the number of documents in which word w appears, and log is computed using the natural base. So for instance if we check the term frequency computed using TfidfVectorizer,

In [79]:
```
sklearn_tfidf
```

Out[79]:

	fly	plant	reap	time
0	0.000000	0.542701	0.542701	0.641055
1	0.000000	0.000000	0.000000	1.000000
2	0.861037	0.000000	0.000000	0.508542

So to check the computation performed by scikit-learn we first repeat the same process without the IDF computation and produce the plain TF representation.

In [83]:
```
vectorizer = TfidfVectorizer(stop_words=stopwords.words('english'),use_idf=False)
vectors = vectorizer.fit_transform(corpus)
feature_names = vectorizer.get_feature_names()
dense = vectors.todense()
denselist = dense.tolist()

sklearn_tf = pd.DataFrame(denselist, columns=feature_names)
sklearn_tf
```

Out[83]:

	fly	plant	reap	time
0	0.000000	0.408248	0.408248	0.816497
1	0.000000	0.000000	0.000000	1.000000
2	0.707107	0.000000	0.000000	0.707107

We note that we can compute the same values from our original code by applying the Euclidean norm to normalize the word counts:

In [81]:
```
def compute_tf_euclidean_norm(document_word_count):

    document_tf = {}

    denominator = math.sqrt(sum(np.power(list(document_word_count.values()),2)))

    for word, count in document_word_count.items():
        document_tf[word] = count / float(denominator)

    return document_tf
```

In [82]:
```
tf_euclidean_norm = []
for document_word_count in word_count:
    tf_euclidean_norm.append(compute_tf_euclidean_norm(document_word_count))
df_euclidean_norm = create_df(tf_euclidean_norm,dictionary)

# use the same column order
df_euclidean_norm[['fly','plant','reap','time']]
```

Out[82]:

	fly	plant	reap	time
0	0.000000	0.408248	0.408248	0.816497
1	0.000000	0.000000	0.000000	1.000000
2	0.707107	0.000000	0.000000	0.707107

As note, applying the Euclidean norm to our word counts produces the same values for term frequency values generated by scikit-learn. We can apply the smoothed idf used by TfidfVectorizer to generate the TF-IDF representation,

In [48]:
```
def compute_smoothed_idf(df):
```

```
    idf = {}

    # number of documents
    M = len(df)

    for word in df.columns:

        # number of documents in which the word appears

        k = sum(df[word]>0.0)

        idf[word] = math.log((M+1)/(k+1))+1

    return idf
```

In [48]:
```
smoothed_idf = compute_smoothed_idf(df_tf)
smoothed_idf
```

Out[48]:
```
{'plant': 1.6931471805599454,
 'reap': 1.6931471805599454,
 'time': 1.0,
 'fly': 1.6931471805599454}
```

Producing the same idf values computed by scikit-learn.

In [50]:
```
for i,word in enumerate(sklearn_feature_names):
    print(word+"\t"+str(sklearn_idf_values[i]))
```

```
fly     1.6931471805599454
plant   1.6931471805599454
reap    1.6931471805599454
time    1.0
```

We can now combine everything and generate the same TF-IDF representation produced by scikit-learn by

1. multiplying the normalized TF by the smoothed IDF
2. reapplying the l2 normalization to the TF-IDF values computed

So, first step multiply the L2 normalized Tf with the IDF values

In [51]:
```
smoothed_tf_idf = df_euclidean_norm.copy()
for word in smoothed_tf_idf.columns:
    smoothed_tf_idf[word] = smoothed_tf_idf[word]*smoothed_idf[word]
smoothed_tf_idf[['fly','plant','reap','time']]
```

Out[51]:

	fly	plant	reap	time
0	0.000000	0.691224	0.691224	0.816497
1	0.000000	0.000000	0.000000	1.000000
2	1.197236	0.000000	0.000000	0.707107

Now apply L2 normalization to each row obtaining the same table produced using scikit-learn.

In [130...
```
l2_norm_smoothed_tf_idf = smoothed_tf_idf.copy()
for row in range(len(l2_norm_smoothed_tf_idf)):
    denominator = math.sqrt(sum(np.power(l2_norm_smoothed_tf_idf.iloc[row].values,2)))
    l2_norm_smoothed_tf_idf.iloc[row] = l2_norm_smoothed_tf_idf.iloc[row]/denominator

l2_norm_smoothed_tf_idf[['fly','plant','reap','time']]
```

Out[130...

	fly	plant	reap	time
0	0.000000	0.542701	0.542701	0.641055
1	0.000000	0.000000	0.000000	1.000000
2	0.861037	0.000000	0.000000	0.508542

Conclusions

Overall, what we have discussed during the lecture is a simplification of the process that uses simpler formulas but the process is the same used in sci-kit learn. If you want to prepare for the exams try to reproduce the computations shown in the first part of this notebook using TF as plain count (as used in the lecture slides) or as normalized percentage. The second part on scikit-learn implementation is just for your curiosity :)

In [ ]: