Data Science Project

1 Introduction

The "Communities and Crime" dataset belongs to the UCI machine learning repository and can be found at this address: https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime. The aim of the project is to predict the number of violent crimes per 100k population in american cities, given socio-economic indicators of that cities. We first note an important thing about this dataset: as described on the website, each continuous variable has already been normalized using an equal-interval binning method - including the outcome variable.
Therefore, instead of dealing with real continuous variables, we are dealing with categorical variables with a huge amount of modalities. Hence, we have to make a choice: either we handle the problem as a classification task, either we handle it as a regression task. As we will explain later, due to the large number of modalities - about one hundred - for ex-continuous variables, we choose to handle it as a regression problem and to see these ex-continuous variables as continuous variables. Moreover, during the binning process, all the values more than 3 standard deviation above the normalized mean were normalized to 1 and all the values less than 3 standard deviation under the normalized mean were normalized to 0.
Because of this transformation, the relationships between variables no longer hold. The ratio within variables is preserved -except for the 1/0 values. This makes the dataset tricky to handle.

To sum-up, this dataset is composed of 126 features -continuous as well as categorical -, 1994 observations and one variable to predict: the number of violent crimes per 100 000 citizens. We will consider that we face a regression problem.

In [200]:

```
import pandas as pd
import numpy as np
import sklearn
import pylab
from sklearn.svm import LinearSVR
from sklearn.grid_search import GridSearchCV
from sklearn.linear_model import Ridge, LinearRegression, RidgeCV
from sklearn.preprocessing import normalize
from sklearn.feature_selection import RFECV
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_extraction import DictVectorizer
from pandas import unique
from sklearn import cross_validation
import matplotlib.pyplot as plt
from pylab import pcolor, colorbar
from sklearn.feature_selection import SelectFromModel
from pandas.tools.plotting import scatter_matrix
% pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
WARNING: pylab import has clobbered these variables: ['unique', 'pylab', 'f']
`%matplotlib` prevents importing * from pylab and numpy
```

2 The dataset

First, we load the dataset with the names of the variables and take a look at the missing values.

In [275]:

```
crime = pd.read_csv("crime.txt", header = None)
titles = pd.read_csv("names.txt", sep = None, header = None)
crime.columns = titles.ix[:, 1]
```

```
C:\Python\python\lib\site-packages\ipykernel\__main__.py:2: ParserWarning: Falling back to the 'python' eng
ine because the 'c' engine does not support sep=None with delim_whitespace=False; you can avoid this warning
by specifying engine='python'.
  from ipykernel import kernelapp as app
```

In [92]:

```
crime = crime.replace("?", np.nan)
crime.isnull().sum()/crime.shape[0]
```

```
C:\Python\python\lib\site-packages\pandas\core\common.py:449: FutureWarning: elementwise comparison failed; returning scalar instead, but in the future will perform elementwise comparison
  mask = arr == x
```

Out[92]:

```
1
state                    0.000000
county                   0.588766
community                0.590271
communityname            0.000000
fold                     0.000000
population               0.000000
householdsize            0.000000
racepctblack             0.000000
racePctWhite             0.000000
racePctAsian             0.000000
racePctHisp              0.000000
agePct12t21              0.000000
agePct12t29              0.000000
agePct16t24              0.000000
agePct65up               0.000000
numbUrban                0.000000
pctUrban                 0.000000
medIncome                0.000000
pctWWage                 0.000000
pctWFarmSelf             0.000000
pctWInvInc               0.000000
pctWSocSec               0.000000
pctWPubAsst              0.000000
pctWRetire               0.000000
medFamInc                0.000000
perCapInc                0.000000
whitePerCap              0.000000
blackPerCap              0.000000
indianPerCap             0.000000
AsianPerCap              0.000000
                           ...
PctSameHouse85           0.000000
PctSameCity85            0.000000
PctSameState85           0.000000
LemasSwornFT             0.840020
LemasSwFTPerPop          0.840020
LemasSwFTFieldOps        0.840020
LemasSwFTFieldPerPop     0.840020
LemasTotalReq            0.840020
LemasTotReqPerPop        0.840020
PolicReqPerOffic         0.840020
PolicPerPop              0.840020
RacialMatchCommPol       0.840020
PctPolicWhite            0.840020
PctPolicBlack            0.840020
PctPolicHisp             0.840020
PctPolicAsian            0.840020
PctPolicMinor            0.840020
OfficAssgnDrugUnits      0.840020
NumKindsDrugsSeiz        0.840020
PolicAveOTWorked         0.840020
LandArea                 0.000000
PopDens                  0.000000
PctUsePubTrans           0.000000
PolicCars                0.840020
PolicOperBudg            0.840020
LemasPctPolicOnPatr      0.840020
LemasGangUnitDeploy      0.840020
LemasPctOfficDrugUn      0.000000
PolicBudgPerPop          0.840020
ViolentCrimesPerPop      0.000000
dtype: float64
```

Obviously, we have a high percentage of missing values for some features - about 60% and 84%. We choose to remove the variables with a rate of missing values higher than 20%. We also drop the "folder" column, which is not a feature, as described on the website. We also check the number of different values for the "state" and the "communityname" features. We choose to drop the "communityname": beacause of the high number of different values - almost as much as the number of observation - it will not be informative. Moreover, as "state" is the only categorical variable of the dataset, we drop it for the moment. We will use it later. Finally, we drop the rows with missing values - losing 1 row - and change the type of the "OtherPerCap" feature - from Object to float.

In [93]:

```
bol = crime.isnull().sum()/crime.shape[0] > 0.20
crime = crime.drop(crime.columns[bol], axis = 1)
crime = crime.drop("fold", axis = 1)
```

In [94]:

```
print(len(crime["communityname"].unique())/crime.shape[0])
print(len(crime["state"].unique())/crime.shape[0])
```

```
0.9167502507522568
0.023069207622868605
```

In [95]:

```
crime_1 = crime.drop(["communityname", "state"], axis = 1)
crime_1 = crime_1.dropna()
crime_1["OtherPerCap"] = crime["OtherPerCap"].astype(float)
```

In [96]:

```
print(crime.shape)
print(crime_1.shape)
```

```
(1994, 103)
(1993, 101)
```

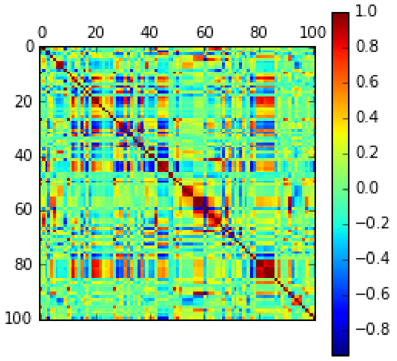
Now, it is time to check the correlation between the variables.
From the plot correlation matrix below, we clearly see that a lot of variables are highly correlated.

In [97]:

```
plt.matshow(crime_1.corr())
colorbar()
```

Out[97]:

```
<matplotlib.colorbar.Colorbar at 0xbe6c898>
```



After a deeper look at the correlations, we decide to remove the less relevant feature - according to varying criteria, e.g does a feature seems to contain more information than the other one ? - in the couples of variable with a correlation higher than 80%. After these step, we have removed one half of the features.

In [98]:

```
crime_1 = crime_1.drop(['agePct16t24', 'agePct12t21',"population",
                                  'pctWInvInc', 'perCapInc',"agePct65up", "medFamInc", "whitePerCap",
                                  "NumUnderPov", "PctPopUnderPov","pctWPubAsst","PctLess9thGrade","PctEmploy
",
                                  "PctBSorMore", "MalePctDivorce", "FemalePctDiv", "householdsize", 'PctYoun
gKids2Par',
                                  'PctTeen2Par','PctKids2Par', 'PctWorkMomYoungKids', 'numbUrban', 'NumIlleg
',
                                  "PctImmigRecent", "PctImmigRec5", "PctImmigRec8","PctRecentImmig", "PctRec
Immig5",
                                  "PctRecImmig8", "PctNotSpeakEnglWell","PctLargHouseOccup", "PersPerOccupH
ous",
                                  "PersPerOwnOccHous","racePctHisp", "PctLargHouseFam", "PctHousOwnOcc", "O
wnOccHiQuart",
                                  "OwnOccLowQuart","RentHighQ", "RentLowQ", "racepctblack", "racePctWhite",
                                  "PctOccupMgmtProf", "PctForeignBorn", "PctSpeakEnglOnly", "PctSameHouse85
","MedRent",
                                  "RentMedian","pctWSocSec", "OwnOccMedVal", "PctFam2Par"], axis = 1)
crime_1.shape
```

Out[98]:

```
(1993, 50)
```

Finally, before we run the first algorithm, we divide the data set in two parts: the training set and the test set. We divide it randomly, using a third of the observations as the test set, and the remaining points as the training set.
Moreover, for a question of reproducibility and to make the comparison between models easier, we set the "random_value" parameter to 43.

In [99]:

```
TrainX, TestX, TrainY, TestY = cross_validation.train_test_split(crime_1.ix[:, 0:49], crime_1.ix[:, 49], tes
t_size = 0.33,random_state = 43)
```

3 First algorithms.

We choose to run a SVM and a Random Forest algorithms on the dataset.
We print the Mean Absolute Error and plot the predicted values against the true values of ViolentCrimesPerPop.

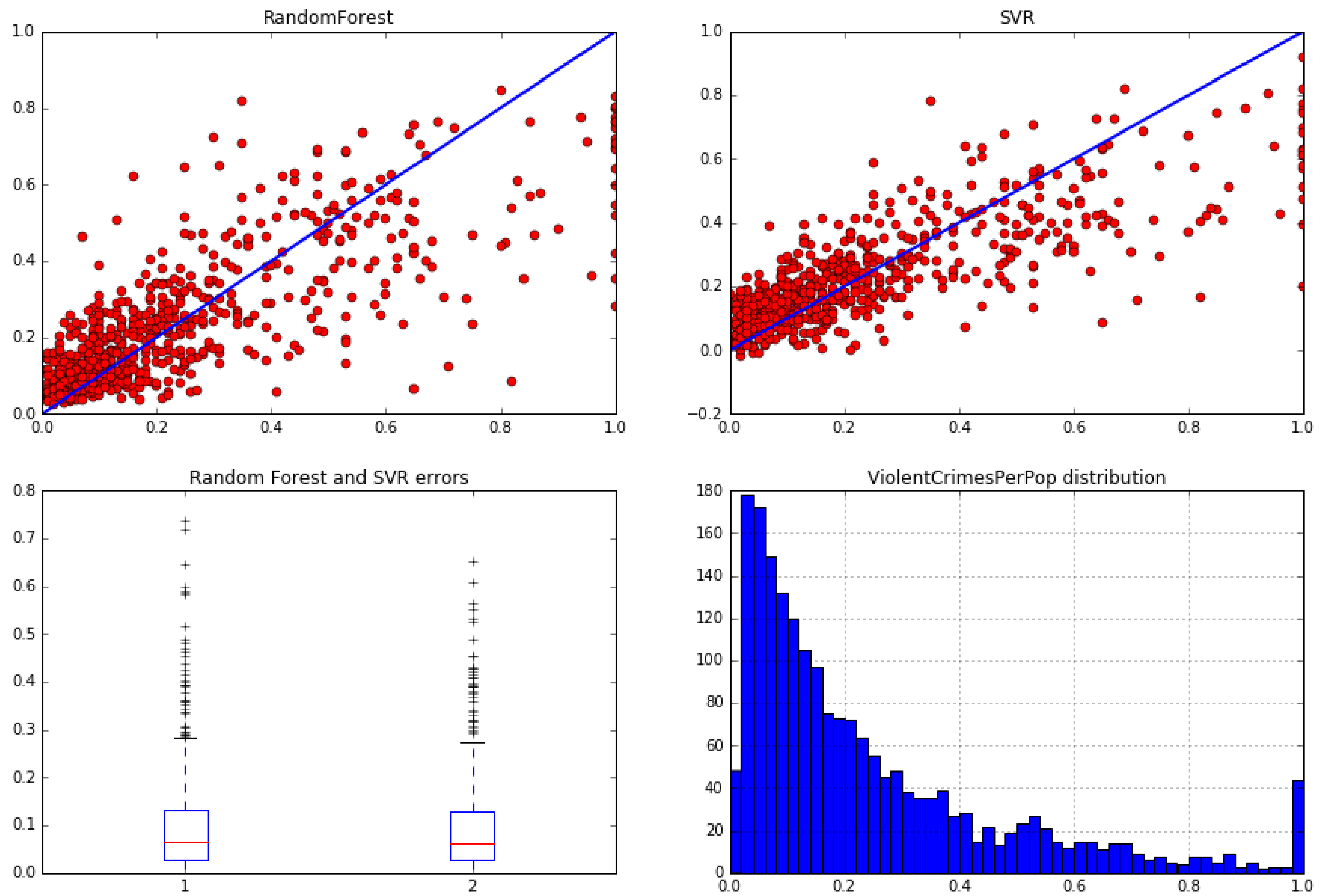
In [252]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX, TrainY)
SV.fit(TrainX, TrainY)
yForest = Forest.predict(TestX)
ySV = SV.predict(TestX)
errorsForest = np.abs(yForest -  TestY)
errorsSV = np.abs(ySV -  TestY)
errorForest = np.mean(errorsForest)
errorSV = np.mean(errorsSV)
print(errorForest)
print(errorSV)
```

```
0.10165197568389053
0.09714662728634305
```

In [101]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(221)
ax1.set_title("RandomForest")
plt.plot(TestY, yForest, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(222)
ax2.set_title("SVR")
plt.plot(TestY, ySV, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = subplot(223)
ax3.set_title("Random Forest and SVR errors")
data_to_plot = [errorsForest, errorsSV]
ax3.boxplot(data_to_plot)
ax4 = plt.subplot(224)
ax4.set_title("ViolentCrimesPerPop distribution")
crime_1["ViolentCrimesPerPop"].hist(bins = 50)
plt.show()
```



Obviously, the higher are the true value, the less precise are the algorithms. We also note that the points on the Random Forest graphic are more spread in general than the points on SVM graphics. Hence, the SVM is globally more precise than the Random Forest. Moreover, some values predicted by the SVM are negative.
As explained in the introduction, because of the normalization of the dataset, we note an excess of 1 values in the outcome. All variables 3 times the standard deviation above the mean were set to one. It may disturb the algorithms. So, we try to remove all the rows with an outcome value equal to 1. We will examine these examples latter.

In [102]:

```
crime_2 = crime_1.ix[crime_1["ViolentCrimesPerPop"] !=1, :]
```

In [103]:

```
TrainX_2, TestX_2, TrainY_2, TestY_2 = cross_validation.train_test_split(crime_2.ix[:, 0:49], crime_2.ix[:, 49], test_size = 0.33,random_state = 43)
```

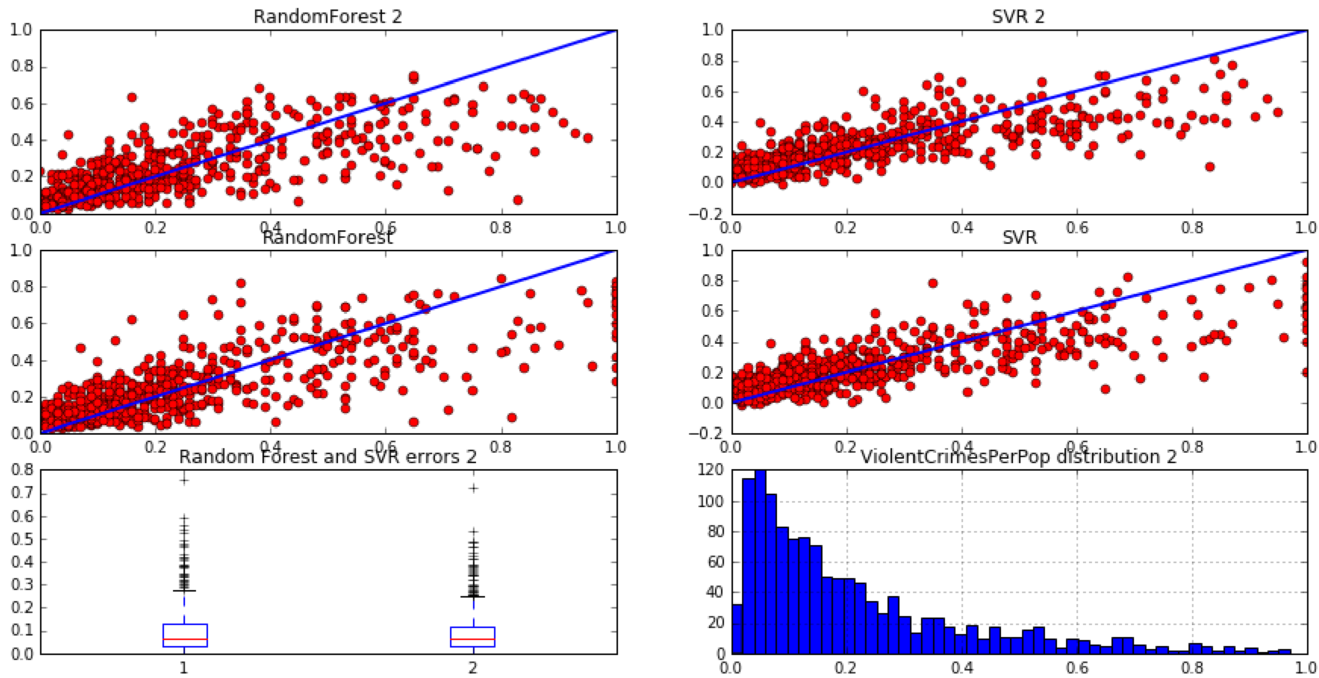
In [104]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX_2, TrainY_2)
SV.fit(TrainX_2, TrainY_2)
yForest_2 = Forest.predict(TestX_2)
ySV_2 = SV.predict(TestX_2)
errorsForest_2 = np.abs(yForest_2 -  TestY_2)
errorsSV_2 = np.abs(ySV_2 -  TestY_2)
errorForest_2 = np.mean(errorsForest_2)
errorSV_2 = np.mean(errorsSV_2)
print(errorForest_2)
print(errorSV_2)
```

```
0.09674689440993799
0.09317910862105523
```

In [105]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest 2")
plt.plot(TestY_2, yForest_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR 2")
plt.plot(TestY_2, ySV_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest")
plt.plot(TestY, yForest, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR")
plt.plot(TestY, ySV, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax5 = subplot(425)
ax5.set_title("Random Forest and SVR errors 2")
data_to_plot = [errorsForest_2, errorsSV_2]
ax5.boxplot(data_to_plot)
ax6 = plt.subplot(426)
ax6.set_title("ViolentCrimesPerPop distribution 2")
TrainY_2.hist(bins = 50)
plt.show()
```



Unsurprisingly, as we droped few points, the MAE decreases for the two models. But the models look a bit more precise: the points are a little less scattered - it is more obvious with the Random Forest than with the SVM.
The normalization to 0 can explain the negative predictions. Let's confirm that.

In [106]:

```
crime_3 = crime_2.ix[crime_2["ViolentCrimesPerPop"] !=0, :]
```

In [107]:

```
TrainX_3, TestX_3, TrainY_3, TestY_3 = cross_validation.train_test_split(crime_3.ix[:, 0:49], crime_3.ix[:, 49], test_size = 0.33,random_state = 43)
```

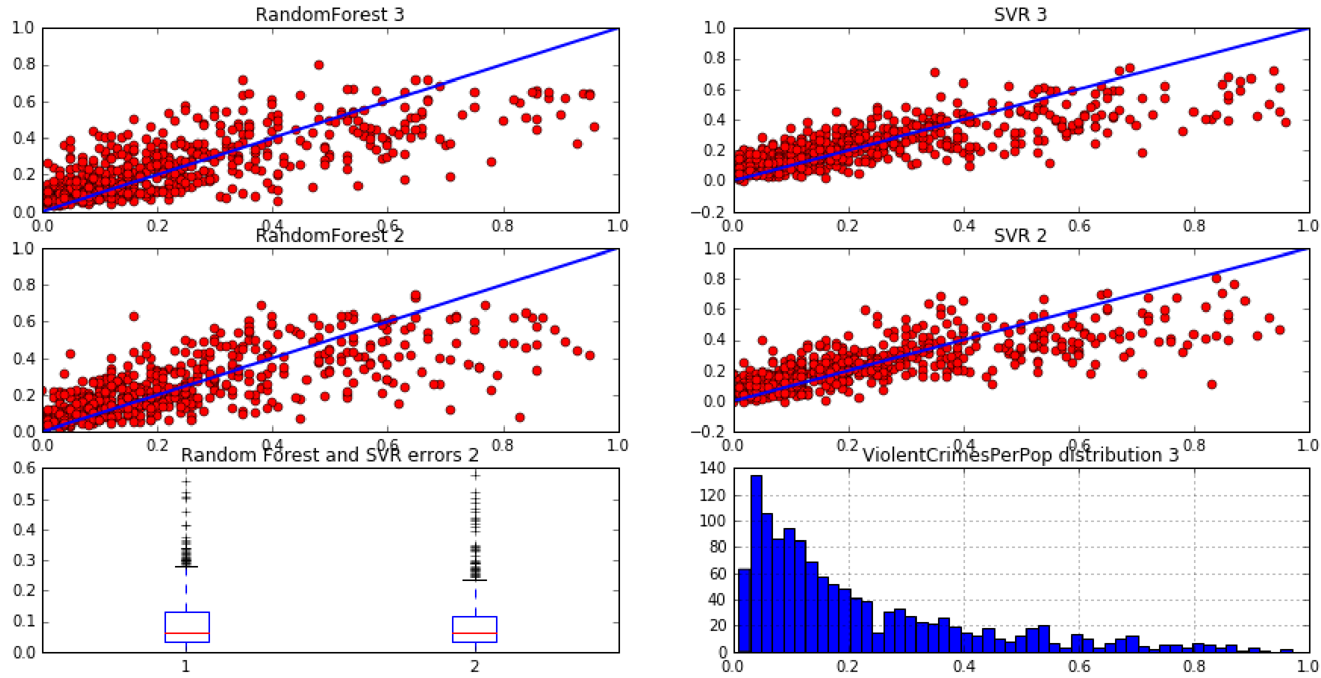
In [108]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX_3, TrainY_3)
SV.fit(TrainX_3, TrainY_3)
yForest_3 = Forest.predict(TestX_3)
ySV_3 = SV.predict(TestX_3)
errorsForest_3 = np.abs(yForest_3 -  TestY_3)
errorsSV_3 = np.abs(ySV_3 -  TestY_3)
errorForest_3 = np.mean(errorsForest_3)
errorSV_3 = np.mean(errorsSV_3)
print(errorForest_3)
print(errorSV_3)
```

```
0.09745156249999995
0.09041721126749094
```

In [109]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest 3")
plt.plot(TestY_3, yForest_3, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR 3")
plt.plot(TestY_3, ySV_3, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest 2")
plt.plot(TestY_2, yForest_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR 2")
plt.plot(TestY_2, ySV_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax5 = subplot(425)
ax5.set_title("Random Forest and SVR errors 2")
data_to_plot = [errorsForest_3, errorsSV_3]
ax5.boxplot(data_to_plot)
ax6 = plt.subplot(426)
ax6.set_title("ViolentCrimesPerPop distribution 3")
TrainY_3.hist(bins = 50)
plt.show()
```



We still have negative values predicted by the SVM. Moreover, it does not make the algorithms more precise. We keep the 0 values.
As we noted earlier, the higher is the outcome, the less precise are the algorithms. Hence, we will try to apply a logarithm to the outcome.

In [110]:

```
crime_log = crime_2.copy()
crime_log["ViolentCrimesPerPop"] = np.log(crime_log["ViolentCrimesPerPop"]+1)
```

In [111]:

```
TrainX_log, TestX_log, TrainY_log, TestY_log = cross_validation.train_test_split(crime_log.ix[:, 0:49], crim
e_log.ix[:, 49], test_size = 0.33,random_state = 43)
```

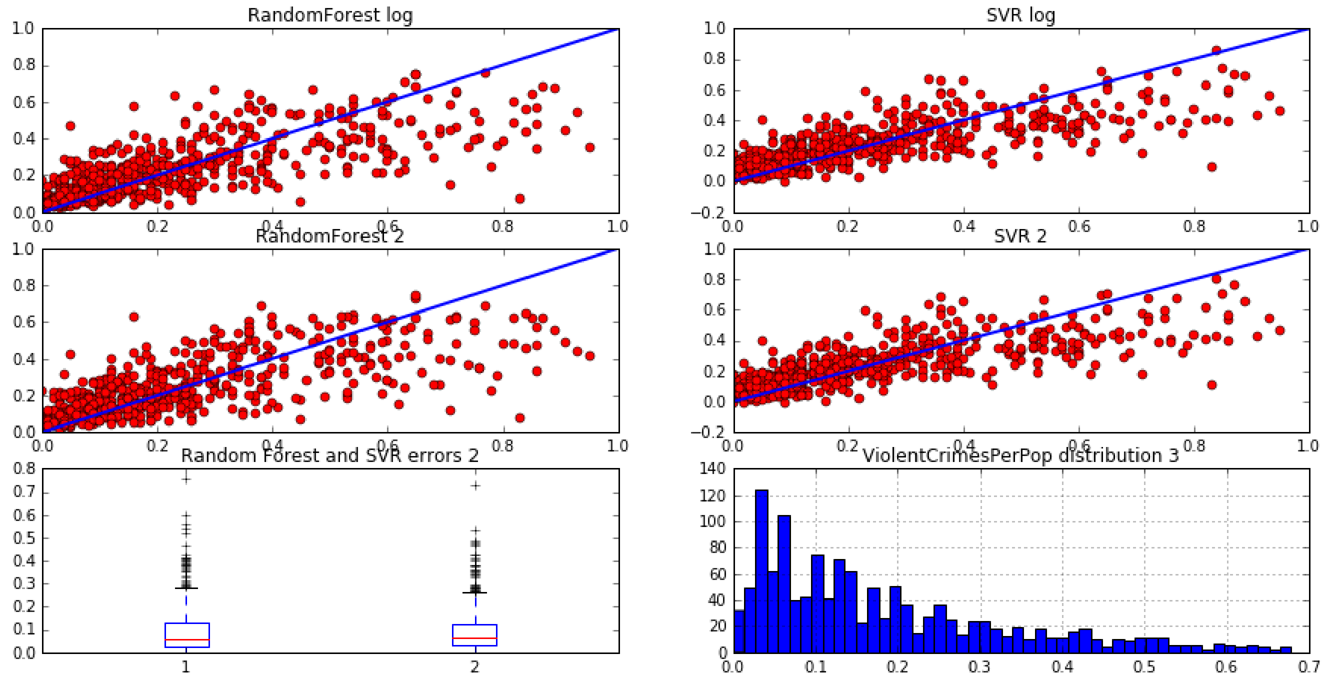
In [112]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX_log, TrainY_log)
SV.fit(TrainX_log, TrainY_log)
yForest_log = Forest.predict(TestX_log)
ySV_log = SV.predict(TestX_log)
errorsForest_log = np.abs((np.exp(yForest_log)-1) -  (np.exp(TestY_log)-1))
errorsSV_log = np.abs((np.exp(ySV_log)-1) -  (np.exp(TestY_log)-1))
errorForest_log = np.mean(errorsForest_log)
errorSV_log = np.mean(errorsSV_log)
print(errorForest_log)
print(errorSV_log)
```

```
0.09339716490340338
0.09432072250066363
```

In [113]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest log")
plt.plot((np.exp(TestY_log)-1), (np.exp(yForest_log)-1), "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR log")
plt.plot((np.exp(TestY_log)-1), (np.exp(ySV_log)-1), "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest 2")
plt.plot(TestY_2, yForest_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR 2")
plt.plot(TestY_2, ySV_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax5 = subplot(425)
ax5.set_title("Random Forest and SVR errors 2")
data_to_plot = [errorsForest_log, errorsSV_log]
ax5.boxplot(data_to_plot)
ax6 = plt.subplot(426)
ax6.set_title("ViolentCrimesPerPop distribution 3")
TrainY_log.hist(bins = 50)
plt.show()
```



The logarithm does not seem to improve the quality of the predictions. So, we choose to keep the crime_2 dataset.
It is time to take a closer look at the features.

4 Features distributions.

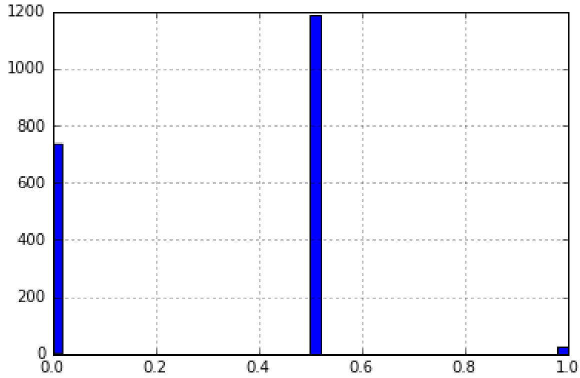
We looked at every feature's distribution, and kept in mind several features we would like to manipulate.
The first one is "MedNumBR". Here is its distribution.

In [114]:

```
crime_2["MedNumBR"].hist(bins = 50)
```

Out[114]:

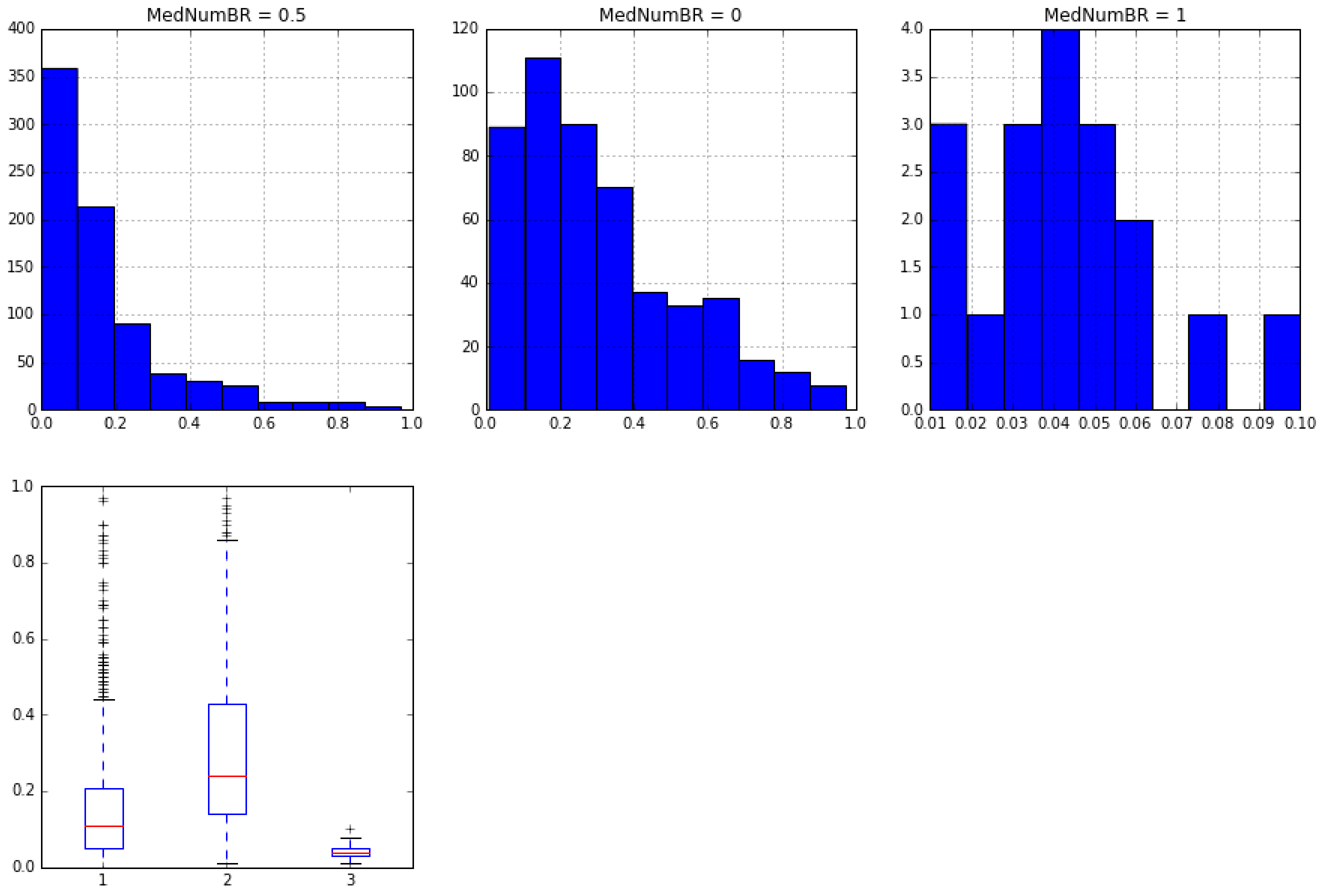
```
<matplotlib.axes._subplots.AxesSubplot at 0xbd9f0b8>
```



This variable takes only three different values. Let's check the distribution of the outcome given the value of "MedNumBR".

In [115]:

```
v1 = TrainY_2[TrainX_2["MedNumBR"] == 0.5]
v2 = TrainY_2[TrainX_2["MedNumBR"] == 0]
v3 = TrainY_2[TrainX_2["MedNumBR"] == 1]
ax1 = plt.subplot(2, 3, 1)
ax1.set_title("MedNumBR = 0.5")
v1.hist(figsize=(15, 10))
ax2 = plt.subplot(2, 3, 2)
ax2.set_title("MedNumBR = 0")
v2.hist()
ax3 = plt.subplot(2, 3, 3)
ax3.set_title("MedNumBR = 1")
v3.hist()
data_to_plot = [v1, v2, v3]
fig = plt.figure(1, figsize=(9, 6))
ax4 = plt.subplot(2, 3, 4)
bp = ax4.boxplot(data_to_plot)
plt.show()
```



Depending on the value of "MedNumBR", the distribution of the outcome is not the same. Indeed, if MedNumBR is equal to 0, ViolentCrimesPerPop is more likely to be high. On the contrary, if MedNumBR is equal to 1, ViolentCrimesPerPop is more likely to be very small.
We decide to transform this variable into a categorical one, with three modalities.

In [116]:

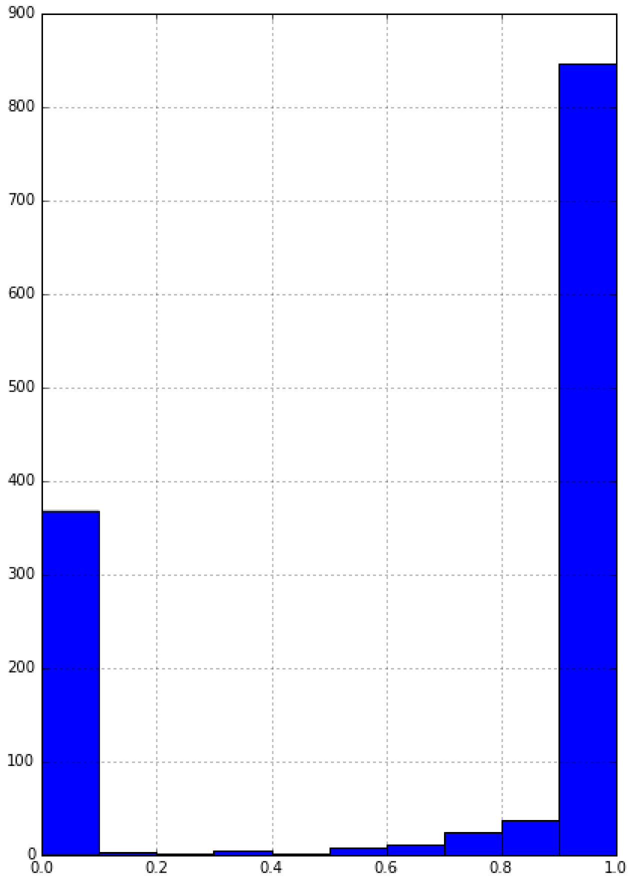
```
crime_2["MedNumBR"] = crime_2["MedNumBR"].replace(0, "a")
crime_2["MedNumBR"] = crime_2["MedNumBR"].replace(1,"c" )
crime_2["MedNumBR"] = crime_2["MedNumBR"].replace(0.5, "b")
crime_2["MedNumBR"] = crime_2["MedNumBR"].astype("category")
```

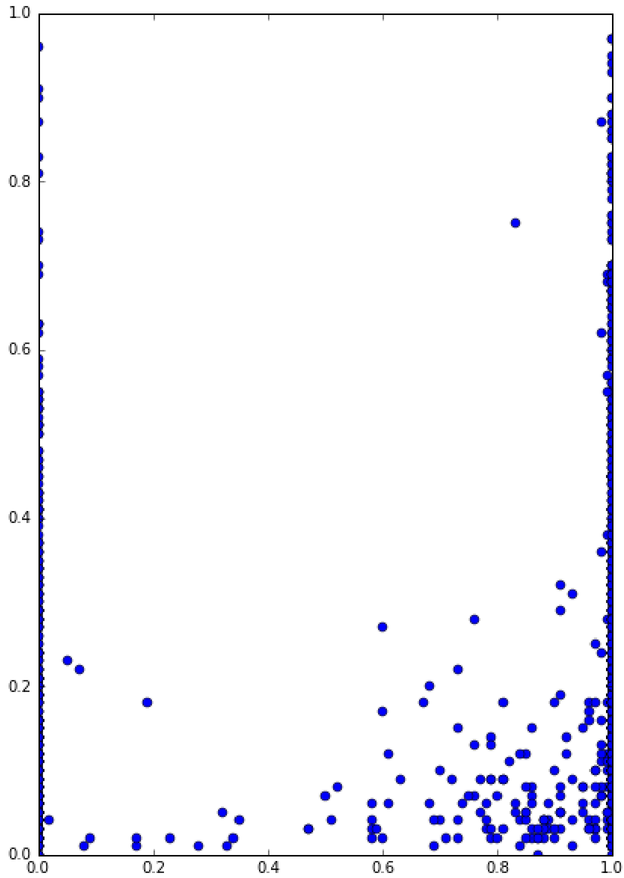
```
C:\Python\python\lib\site-packages\ipykernel\__main__.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vie
w-versus-copy
  if __name__ == '__main__':
C:\Python\python\lib\site-packages\ipykernel\__main__.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vie
w-versus-copy
  from ipykernel import kernelapp as app
C:\Python\python\lib\site-packages\ipykernel\__main__.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vie
w-versus-copy
  app.launch_new_instance()
C:\Python\python\lib\site-packages\ipykernel\__main__.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-vie
w-versus-copy
```

In [276]:

```
dummies = pd.get_dummies(crime_2["MedNumBR"])
dummies = dummies.drop("c", axis = 1)
dummies.columns = ["MedNumBR = 0", "MedNumBR = 0.5"]
crime_dummies = crime_2.drop("MedNumBR", axis = 1)
crime_dummies = pd.concat([crime_dummies.ix[:, 0:48], dummies, crime_dummies.ix[:, 48]], axis = 1)
```

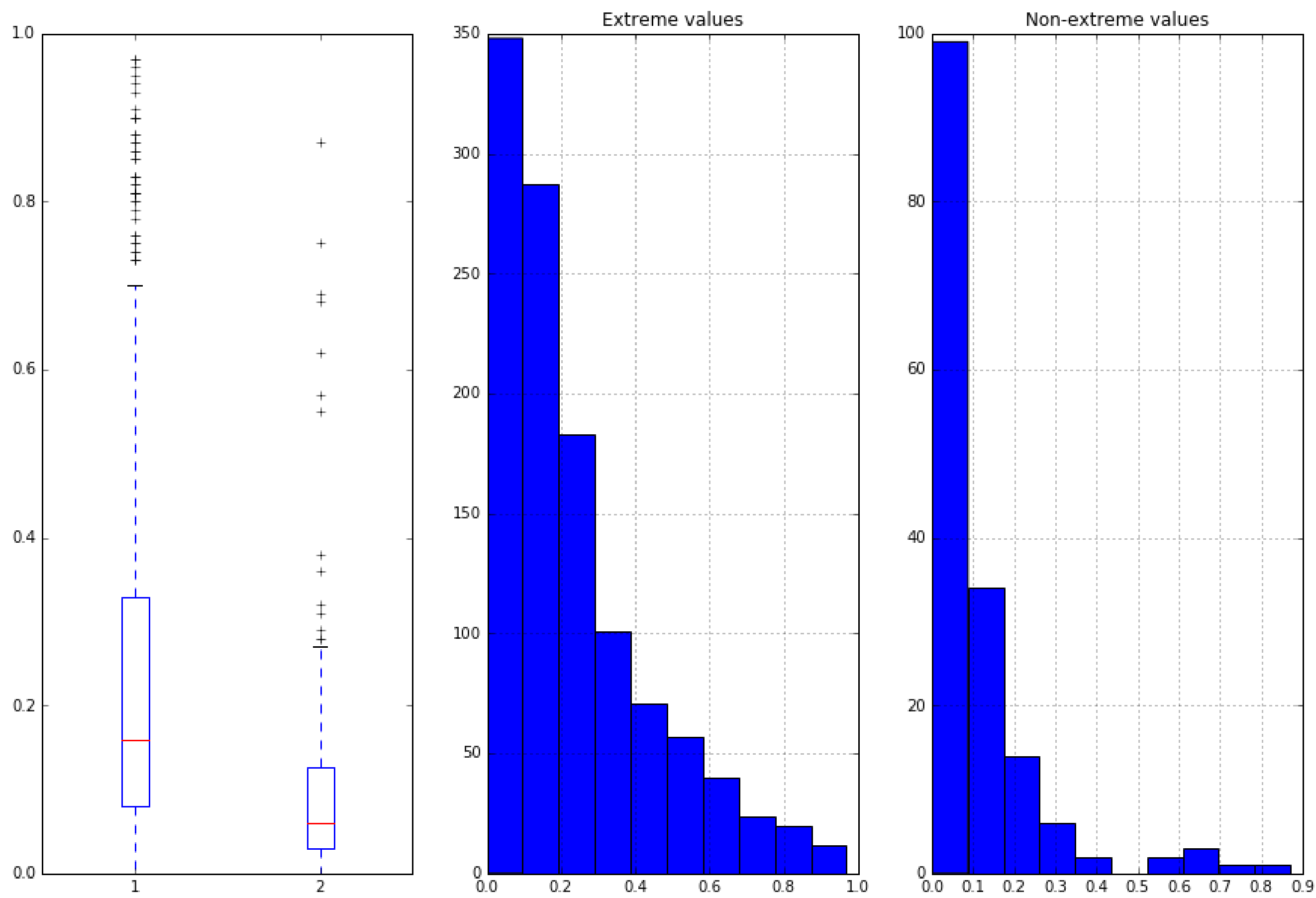
The second variable we want to change is the "pctUrban" variable: the percentage of people living in an area classified as urban. When this variable takes one of the two extremes values, the number of violent crime is much more likely to be high. So we create a categorical variable: one if pctUrban is equal to 0 or 1, else 0:





In [119]:

```
v = np.logical_or(TrainX_2.pctUrban == 1, TrainX_2.pctUrban == 0)
TrainYTRUE = TrainY_2[v]
v = v !=True
TrainYFALSE = TrainY_2[v]
data_to_plot = [TrainYTRUE, TrainYFALSE]
ax1 = plt.subplot(131)
ax1.boxplot(data_to_plot)
ax2 = plt.subplot(132)
ax2.set_title("Extreme values")
TrainYTRUE.hist(figsize=(15, 10))
ax3 = plt.subplot(133)
ax3.set_title("Non-extreme values")
TrainYFALSE.hist()
plt.show()
```



In [120]:

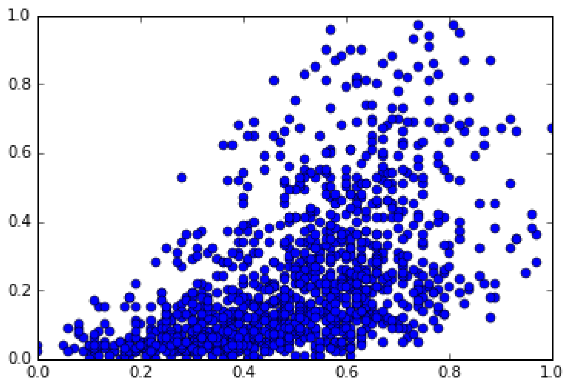
```
crime_dummies["pctUrban"] = np.logical_or(crime_dummies["pctUrban"] == 0, crime_dummies["pctUrban"] == 1).as
type(int)
crime_dummies["pctUrban"] = crime_dummies["pctUrban"].astype("category")
```

We also transform the TotalPctDiv variable. When we plot the outcome against it, we see a kind of parabola correlated with the outcome at 54%.

In [123]:

```
plt.plot(TrainX_2["TotalPctDiv"], TrainY_2, "o")
print("Correlation with the outcome:" ,np.corrcoef(TrainX_2["TotalPctDiv"], TrainY_2)[0, 1])
```

```
Correlation with the outcome: 0.543103077386
```

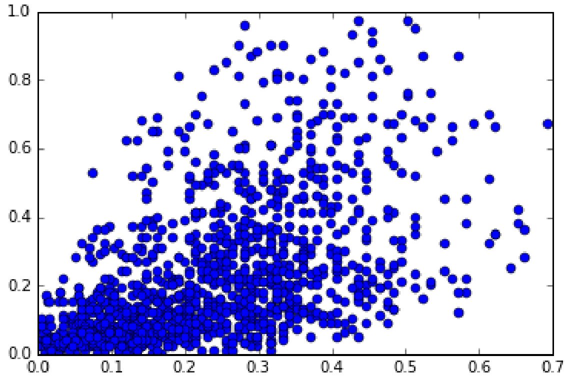


We think that the square of this variable would be more correlated with ViolentCrimesPerPop. So we try:

In [124]:

```
plt.plot(np.log(TrainX_2["TotalPctDiv"]**2+1), TrainY_2, "o")
print("Correlation with the outcome:" ,np.corrcoef(np.log(TrainX_2["TotalPctDiv"]**2+1), TrainY_2)[0, 1])
```

```
Correlation with the outcome: 0.546585321858
```



The correlation does not increase that much. We don't keep this transformation.
Now it is time to test the algorithms with the changes we have done:

In [125]:

```
TrainX_dummies, TestX_dummies, TrainY_dummies, TestY_dummies = cross_validation.train_test_split(crime_dummi
es.ix[:, 0:50], crime_dummies.ix[:, 50], test_size = 0.33,random_state = 43)
```

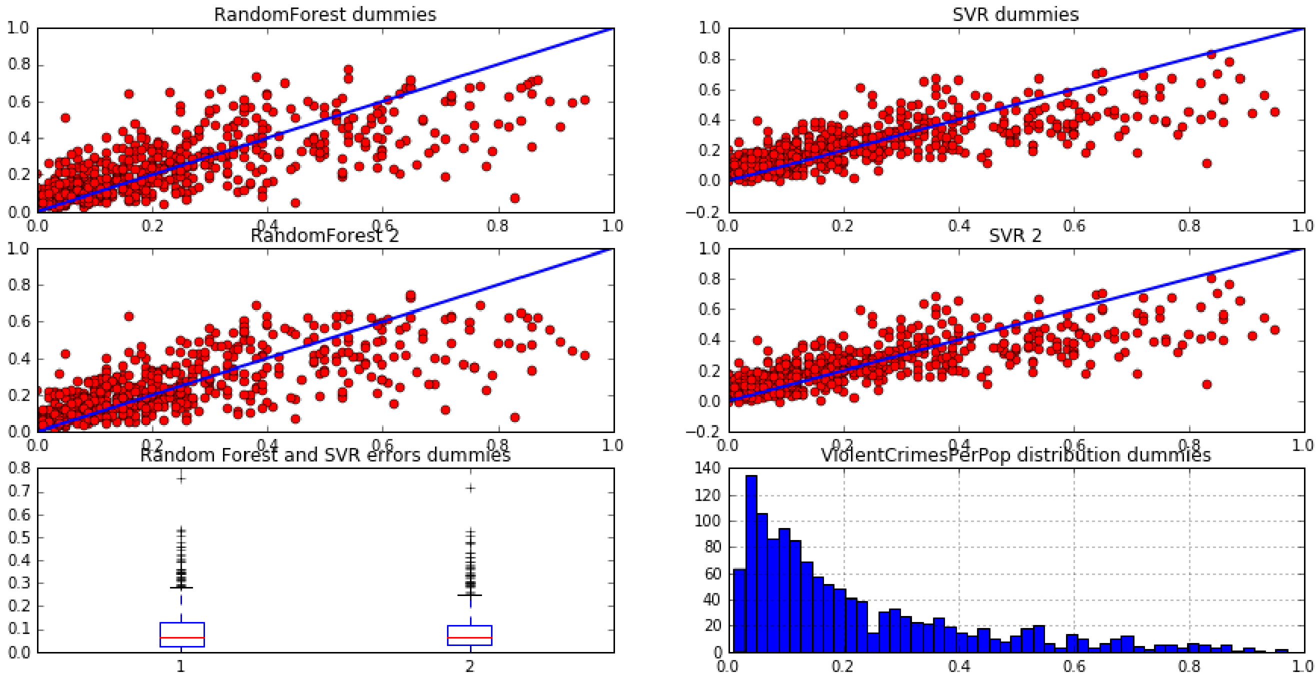
In [126]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX_dummies, TrainY_dummies)
SV.fit(TrainX_dummies, TrainY_dummies)
yForest_dummies = Forest.predict(TestX_dummies)
ySV_dummies = SV.predict(TestX_dummies)
errorsForest_dummies = np.abs(yForest_dummies -  TestY_dummies)
errorsSV_dummies = np.abs(ySV_dummies -  TestY_dummies)
errorForest_dummies = np.mean(errorsForest_dummies)
errorSV_dummies = np.mean(errorsSV_dummies)
print(errorForest_dummies)
print(errorSV_dummies)
```

```
0.09718633540372672
0.09395074313633432
```

In [127]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest dummies")
plt.plot(TestY_dummies, yForest_dummies, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR dummies")
plt.plot(TestY_dummies, ySV_dummies, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest 2")
plt.plot(TestY_2, yForest_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR 2")
plt.plot(TestY_2, ySV_2, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax5 = subplot(425)
ax5.set_title("Random Forest and SVR errors dummies")
data_to_plot = [errorsForest_dummies, errorsSV_dummies]
ax5.boxplot(data_to_plot)
ax6 = plt.subplot(426)
ax6.set_title("ViolentCrimesPerPop distribution dummies")
TrainY_3.hist(bins = 50)
plt.show()
```



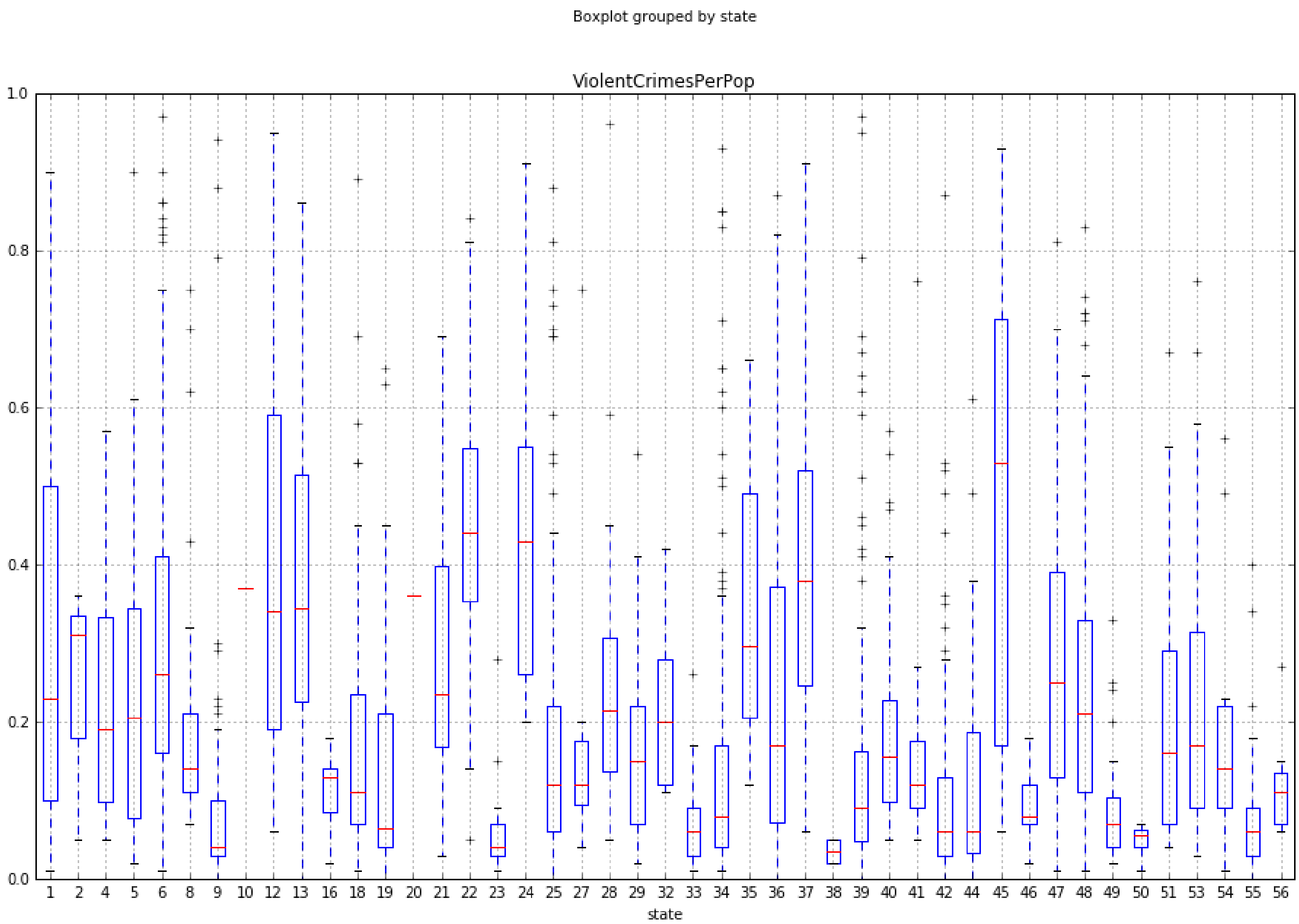
The transformations we have done seem to change almost nothing.
Earlier in this notebook, we droped the "state" variable, which was the only real categorical variable. We should take a closer look at it:

In [128]:

```
state = crime["state"]
state = state[crime["ViolentCrimesPerPop"] != 1]
crime_dummies = pd.concat([state, crime_dummies], axis = 1)
crime_dummies.boxplot("ViolentCrimesPerPop", by = "state", figsize = (15, 10))
```

Out[128]:

```
<matplotlib.axes._subplots.AxesSubplot at 0xc842d68>
```



It is clear, from the boxplot above, that the state variable may be informative. For example, the number of crimes in the state 45 is more likely to be high than than in the state 38. We integrate it in the dataset.

In [129]:

```
df = pd.get_dummies(crime_dummies.state)
df.columns = range(df.shape[1])
df = pd.concat([df.ix[:, 0:43], crime_dummies], axis = 1)
df = df.drop("state", axis = 1)
```

In [130]:

```
df = df.dropna()
```

We now run our algorithms on this new dataset:

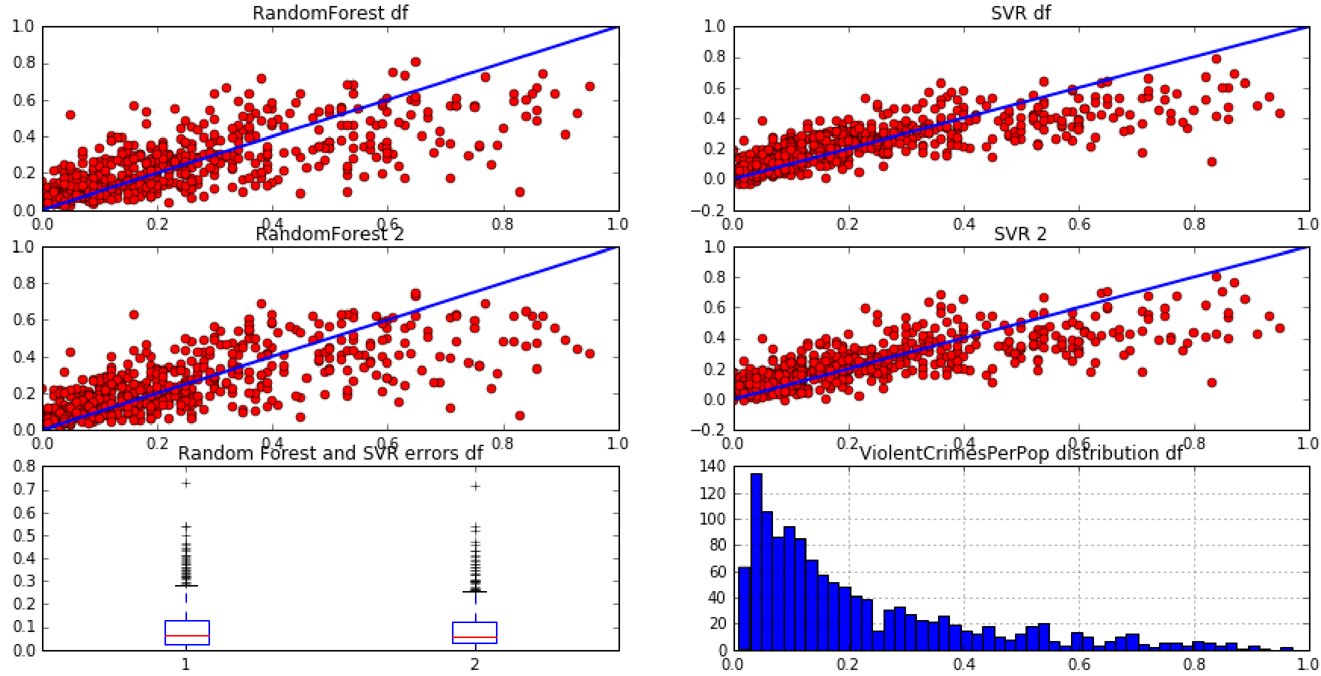
In [131]:

```
TrainX_df, TestX_df, TrainY_df, TestY_df = cross_validation.train_test_split(df.ix[:, 0:94], df.ix[:, 94], t
est_size = 0.33,random_state = 43)
```

In [132]:

```
Forest = RandomForestRegressor()
SV = SVR()
Forest.fit(TrainX_df, TrainY_df)
SV.fit(TrainX_df, TrainY_df)
yForest_df = Forest.predict(TestX_df)
ySV_df = SV.predict(TestX_df)
errorsForest_df = np.abs(yForest_df -  TestY_df)
errorsSV_df = np.abs(ySV_df -  TestY_df)
errorForest_df = np.mean(errorsForest_df)
errorSV_df = np.mean(errorsSV_df)
print(errorForest_df)
print(errorSV_df)
```

```
0.09634937888198755
0.08944043933987765
```



The MAE of the SVM decreased while the MAE of the Random Forest is almost the same.

Unfonrtunately, after a closer look at the farest points, we did not succed to identify what makes the algorithms fail on them.

4 Feature selection

In [134]:

```
Forest = RandomForestRegressor()
SV = SVR(kernel = "linear")
rfecvForest = RFECV(Forest)
rfecvForest.fit(TrainX_df, TrainY_df)
rfecvSV = RFECV(SV)
rfecvSV.fit(TrainX_df, TrainY_df)
```

Out[134]:

```
RFECV(cv=None,
   estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
  kernel='linear', max_iter=-1, shrinking=True, tol=0.001, verbose=False),
   estimator_params=None, scoring=None, step=1, verbose=0)
```

In [135]:

```
TrainX_Forest = rfecvForest.transform(TrainX_df)
TrainX_SV = rfecvSV.transform(TrainX_df)
TestX_Forest = rfecvForest.transform(TestX_df)
TestX_SV = rfecvSV.transform(TestX_df)
Forest = rfecvForest.estimator_
SV = rfecvSV.estimator_
```

In [140]:

```
SV = SVR()
SV.fit(TrainX_SV, TrainY_df)
ySV = SV.predict(TestX_SV)
```

In [136]:

```
yForest = Forest.predict(TestX_Forest)
```

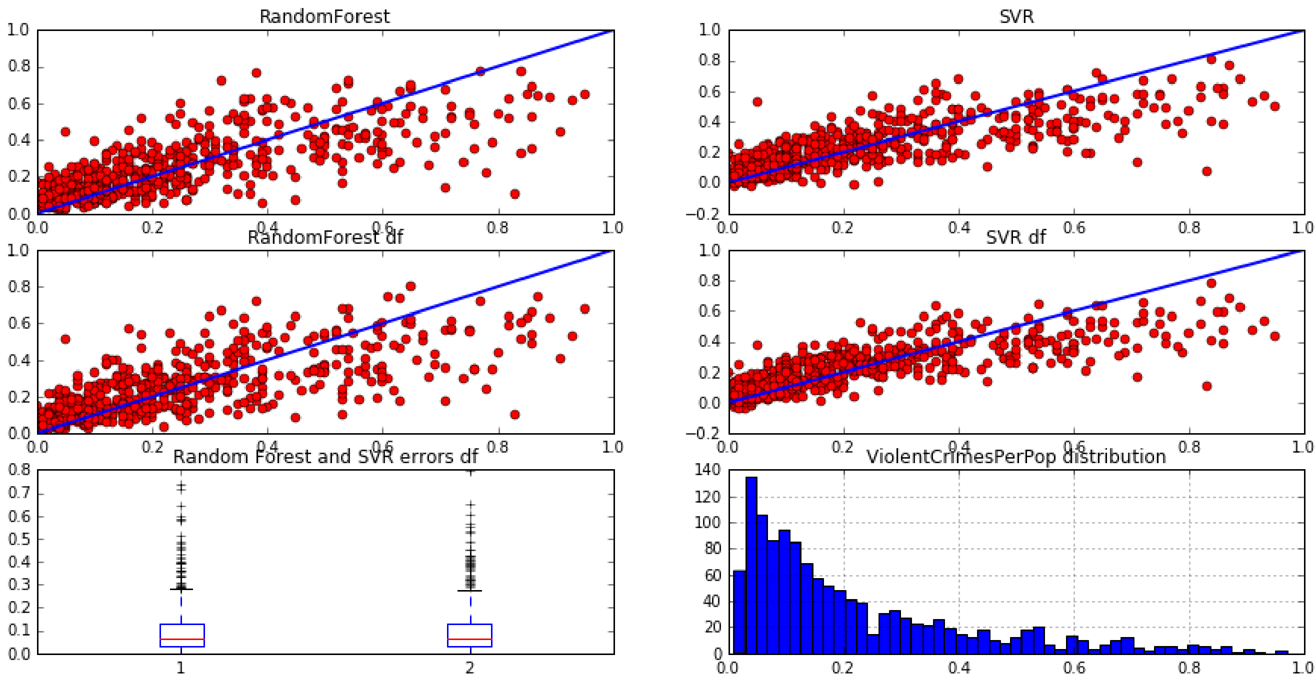
In [141]:

```
print(np.mean(np.abs(yForest - TestY_df)))
print(np.mean(np.abs(ySV - TestY_df)))
```

```
0.09373757763975155
0.08960452784434854
```

In [142]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest")
plt.plot(TestY_df, yForest, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR")
plt.plot(TestY_df, ySV, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest df")
plt.plot(TestY_df, yForest_df, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR df")
plt.plot(TestY_df, ySV_df, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax5 = subplot(425)
ax5.set_title("Random Forest and SVR errors df")
data_to_plot = [errorsForest, errorsSV]
ax5.boxplot(data_to_plot)
ax6 = plt.subplot(426)
ax6.set_title("ViolentCrimesPerPop distribution")
TrainY_3.hist(bins = 50)
plt.show()
```



While the MAE of the Random Forest decreases, the overall precision does not seem to be better. Moreover, the MAE of the SVM increases very slightly and the algorithm seems a less precise. Hence, we choose to keep the result of the feature selection for the Random Forest, but not for the SVM. We keep the following features:

In [149]:

```
list(TrainX_df.ix[:, rfecvForest.support_].columns)
```

Out[149]:

```
[0,
 4,
 8,
 9,
 11,
 18,
 20,
 21,
 26,
 37,
 'racePctAsian',
 'agePct12t29',
 'pctUrban',
 'medIncome',
 'pctWWage',
 'pctWFarmSelf',
 'pctWRetire',
 'blackPerCap',
 'indianPerCap',
 'AsianPerCap',
 'OtherPerCap',
 'HispPerCap',
 'PctNotHSGrad',
 'PctUnemployed',
 'PctEmplManu',
 'PctEmplProfServ',
 'PctOccupManu',
 'MalePctNevMarr',
 'TotalPctDiv',
 'PersPerFam',
 'PctWorkMom',
 'PctIlleg',
 'NumImmig',
 'PctImmigRec10',
 'PctRecImmig10',
 'PersPerRentOccHous',
 'PctPersOwnOccup',
 'PctPersDenseHous',
 'PctHousLess3BR',
 'HousVacant',
 'PctHousOccup',
 'PctVacantBoarded',
 'PctVacMore6Mos',
 'MedYrHousBuilt',
 'PctHousNoPhone',
 'PctWOFullPlumb',
 'MedRentPctHousInc',
 'MedOwnCostPctInc',
 'MedOwnCostPctIncNoMtg',
 'NumInShelters',
 'NumStreet',
 'PctBornSameState',
 'PctSameCity85',
 'PctSameState85',
 'LandArea',
 'PopDens',
 'PctUsePubTrans',
 'LemasPctOfficDrugUn',
 'MedNumBR = 0',
 'MedNumBR = 0.5']
```

In [146]:

```
rfecvForest.n_features_
```

Out[146]:

```
60
```

The algorithm droped 34 variables.

5 Finding the best parameters

In [246]:

```
SV = SVR()
parameters = {'gamma': numpy.arange(0.0001, 0.001, 0.0001), 'C':np.arange(1, 10)}
searchSV= GridSearchCV(SV, parameters, scoring = "mean_absolute_error")
searchSV.fit(TrainX_df, TrainY_df)
```

Out[246]:

```
GridSearchCV(cv=None, error_score='raise',
       estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'C': array([1, 2, 3, 4, 5, 6, 7, 8, 9]), 'gamma': array([ 0.0001,  0.0002,  0.0003,  0.00
04,  0.0005,  0.0006,  0.0007,
        0.0008,  0.0009])},
       pre_dispatch='2*n_jobs', refit=True, scoring='mean_absolute_error',
       verbose=0)
```

In [247]:

```
SV_Tuned = searchSV.best_estimator_
```

In [249]:

```
ySV_Tuned = SV_Tuned.predict(TestX_df)
np.mean(np.abs(ySV_Tuned - TestY_df))
```

Out[249]:

```
0.09051951081987497
```

In [266]:

```
RF = RandomForestRegressor()
parameters = {'max_depth': numpy.arange(25, 200, 1), "min_samples_leaf":np.arange(25, 100, 5)}
searchRF= GridSearchCV(RF, parameters, scoring = "mean_absolute_error")
searchRF.fit(TrainX_df.ix[:, rfecvForest.support_], TrainY_df)
```

Out[266]:

```
GridSearchCV(cv=None, error_score='raise',
       estimator=RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'max_depth': array([ 25,  26, ..., 198, 199]), 'min_samples_leaf': array([25, 30, 35, 40,
45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95])},
       pre_dispatch='2*n_jobs', refit=True, scoring='mean_absolute_error',
       verbose=0)
```

In [267]:

```
RF_tuned = searchRF.best_estimator_
```

In [268]:

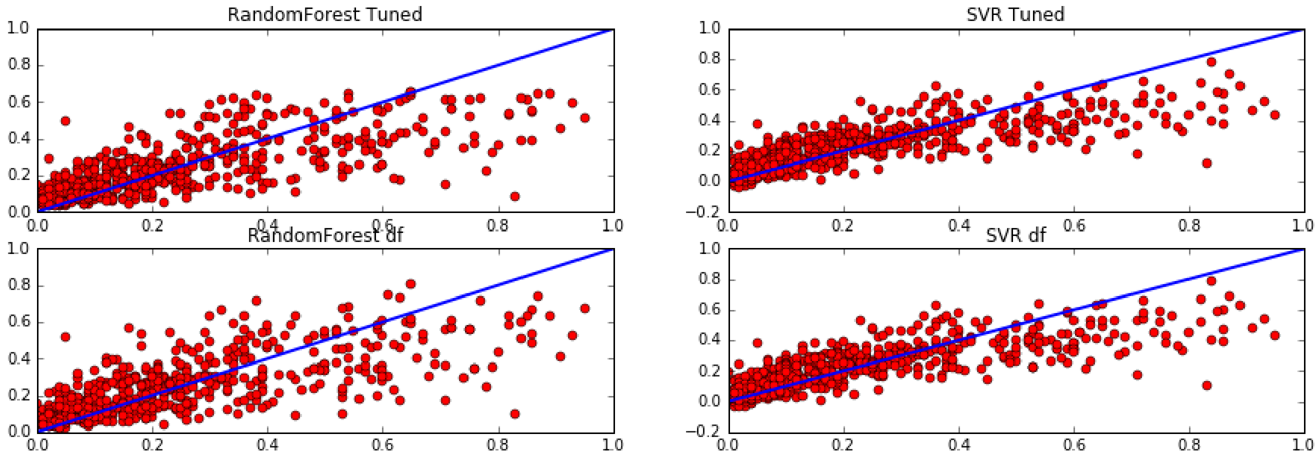
```
yRF_Tuned = RF_tuned.predict(TestX_df.ix[:, rfecvForest.support_])
np.mean(np.abs(yRF_Tuned - TestY_df))
```

Out[268]:

```
0.09322182742607506
```

In [270]:

```
fig = plt.figure(figsize = (15, 10))
ax1 = plt.subplot(421)
ax1.set_title("RandomForest Tuned")
plt.plot(TestY_df, yRF_Tuned, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax2 = plt.subplot(422)
ax2.set_title("SVR Tuned")
plt.plot(TestY_df, ySV_Tuned, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax3 = plt.subplot(423)
ax3.set_title("RandomForest df")
plt.plot(TestY_df, yForest_df, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
ax4 = plt.subplot(424)
ax4.set_title("SVR df")
plt.plot(TestY_df, ySV_df, "ro")
plt.plot([0, 1], [0, 1], linestyle = "-", linewidth = 2)
plt.show()
```



We did not found any parameters that really improve the accuracy of the two models.

Finally, the two models give us approximately the same results. They both understimate the number of Violent crimes when the outcome is "high". We also note that the SVM is more precise than the Random Forest: indeed, on the above plots, the points are less scattered for the SVM than for the Random Forest. However, the SVM predict negatives values, which is not the case of the Random Forest. Moreover, we did not succed to perform a good feature selection for the SVM, while we removed 34 features for the Random Forest, without loosing a lot of precision.

6 Conclusion

We do not think that the model is usable. The mean absolute error is about 0.09 for both models while the outcome is of the order of 0.1, which seems to big. Furthermore, as we said before, both models present a bias: they underestimate the high values of the outcome. Despite the fact that we succeded to drop a lot of features - almost the half - without losing much precision, the accuracy is to low.
We tried to remedy to that situation by looing closer to the points where the models failed the most, but we did not manage to identify their particular caracteristics, what makes them hard to handle for the models. We also tried to stack a linear regression over the two models to correct the bias, with no success. This dataset was tricky because of the binning of the features, which made the relationships between variables impossible to exploit. So, lost good opportunities to create new features - for example, we had the first, second and forth quartile of some variable. We could have use its dispersion e.g 4th quartile minus first quartile, as a variable. It would probably be much easier to use the original data.

For now, the models are not exploitable. The MAE seems to high and they present a bias. This could lead to mobilise less resources than needed in the most violent areas.
Nevertheless, the models have several strengths. The first one is that we managed to significantly reduce the number of features with only a tiny loss of precision. Consequently, the acquistion cost of the features is reduced. The second one is that the models -at least the SVM, can easily be run on larger data sets. However the performances of the models can change over time. Even if many of the socio-economic indicators will probably be always correlated with the number of violent crimes, the importance of some features can change. For example, a city which has a high number of crimes can change to a peaceful one over time. But the models integrate that feature. So, the model may have to be refreshed on a regular basis. Fortunately, the refreshing will not be costly: all the socio-economic indicators as well as the outcome are gathered every year for different purposes.