V by Vodafone Data Challenge 2018

This project has been developed by Gianmaria Carnazzi, Davide Castellini, Maria Stella Cocco, Matteo Karl Donati, Federico Maria Rosi and Noah Kuntner.

Introduction

How we did it ¶

How we did it (for real)

The dataset `dataset_challenge_v5.TRAINING.csv` provides information about 2000 customers who have shown interest in one of the four V by Vodafone products.

We are going to go through the following steps in order to profile customers interested in one of the products of concern.

The first problem we dealt with was that of preprocessing the data in order to be able to extract the right information from it. In fact, the dataset presents some redundant columns, entries formatted in such a way that they cannot be used for our purposes and some other issues that we will cover in detail in this notebook.

The second challenge we faced was the missing data throughout the dataset. We tried different approaches in order to predict them with the best accuracy possible, so as to be able to perform the final step of our project, clustering.

In order to perform appropriate clustering, we took advantage of some algorithms that we believe can give the best results, as we will show later.

We will go deeply through each point we mentioned above and we will provide explanation of our choices throughout the notebook.

Enjoy.

In [1]:
```
%matplotlib inline
```

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

In [3]:
```
pd.set_option('display.max_columns', None)
```

In [4]:
```
df = pd.read_csv('dataset_challenge_v5.TRAINING.csv')
```

In [5]:
```
df
```

Out[5]:

	Unnamed: 0	Unnamed: 0.1	Product	CustomerAge	DataAllowance	DataArpu	DeviceFlag4G
0	0	0	V-Bag	(80, 90]	0.239013	0.023735	0.0
1	1	1	V-Auto	(60, 70]	0.239013	0.118674	0.0
2	2	2	V-Auto	(30, 40]	0.095605	0.178010	1.0
3	4	4	V-Bag	(70, 80]	0.119507	0.249070	0.0
4	5	5	V-Pet	(30, 40]	0.289900	0.083072	0.0
5	7	7	V-Bag	(50, 60]	0.024672	0.076564	0.0
6	9	9	V-Pet	(40, 50]	0.239013	0.083072	0.0
7	10	10	V-Auto	(60, 70]	0.047803	0.142408	0.0
8	11	11	V-Pet	(30, 40]	0.138782	0.213468	1.0

Preprocessing

The first two columns contain exactly the same information, therefore one of the two can be deleted and the other one can be renamed in a more appropriate way.

In [6]:
```
c = list(df.columns)
c[0] = 'ID'
df.columns = c
del df['Unnamed: 0.1']
```

We change the type of the categorical 0/1 from float type to int type.

In [7]:
```
l = ['DeviceFlag4G', 'DeviceFlagSmartphone', 'CustomerGender', 'CustomerImmigrant
```

In [8]:
```
for col in l:
    df[col] = df[col].astype(int)
```

Device Operating System

In the `DeviceOperatingSystem` column two categories dominate over the others, which only combine to form a very small percentage of the total number of devices.
Thus, we group those categories together into a new one called `Other`.

In [9]:
```
df['DeviceOperatingSystem'].value_counts()
```

Out[9]:
```
Android           858
iOS               568
Windows Phone       7
Windows Mobile      6
Proprietary         5
Series 40           2
BREW                1
Firefox             1
BB10                1
Symbian^3           1
VRTXmc              1
BlackBerry OS       1
Name: DeviceOperatingSystem, dtype: int64
```

In [10]:
```
def f(x):
    if not (x =='Android' or x == 'iOS'):
        return 'Other'
    return x
```

In [11]:
```
df['DeviceOperatingSystem'] = df['DeviceOperatingSystem'].map(f, na_action='ignor
```

Geographical Data

The data provided, i.e. `Province`, `Region` and `ZipCode`, is not particularly suitable to be used in a learning algorithm. Therefore, we use that data to extrapolate information that we believe to be more fitting, such as latitude, longitude, degree of urbanization and population.

To get the data of interest we use two datasets.
The first one, `comuni_ancitel.csv`, contains all the information that we need. The only problem is that we do not have a link between our data and the data in the dataset, we therefore rely on another dataset, `comuni_cap`, which enables us to map the zipcodes into the ISTAT numbers, from which we can then obtain the information of interest.

In [12]:
```
df_comuni = pd.read_csv('comuni_ancitel.csv',sep = ';')
df_cap = pd.read_csv('comuni_cap.csv')
df_cap = df_cap.dropna(axis=0)
```

The dataset `comuni_cap.csv` is structered in the following way: there are three columns, respectively `Comune`, `Cap`, `ISTAT`.
The information under the `Cap` column are formatted in three different ways (as ranges, lists or numbers) but what we want is each entry to represent a single Cap.
The code below goes through the dictionary `d`, which maps the zipcodes into the ISTAT numbers using the `comuni_cap.csv` dataset, and creates a new one, namely `d_cap`, which presents the desired structure.

In [13]:
```
d = df_cap.loc[:,'cap':'ISTAT'].set_index('cap').to_dict()['ISTAT']
```

In [14]:
```
import re
```

In [15]:
```
d_cap = {}

for s in d:
    if len(s) == 11:
        l = s.split('-')
        l1 = [i for i in range(int(l[0]), int(l[1]) + 1)]
        for cap in l1:
            d_cap[float(cap)] = d[s]
    else:
        l = re.findall(r'\d+', s)
        for cap in l:
            d_cap[float(cap)] = d[s]
```

Now that we have a dictionary that is suitable to our needs, we can map the zipcodes of the original Vodafone dataset into the ISTAT numbers.

However, we do not map directly from the dictionary, because there are some zipcodes that cannot be found and this would lead to a `KeyError`.
In order to solve this problem we apply the mapping through the function `f` defined below.

In [16]:
```
def f(x, d):
    if np.isnan(x):
        return x

    x = np.int64(x)

    try:
        y = d[x]
    except:
        y = np.nan

    return y
```

In [17]:
```
df['ISTAT'] = df['ZipCode'].apply(f, d = d_cap)
```

In order to obtain the information we mentioned above (latitude, longitude, degree of urbanization and population), we create dictionaries that map from the ISTAT numbers into the categories of interest.
Concerning the geographic area, we create another dictionary mapping from the specific regions into the geographic areas (Nord, Centro, Sud, Isole).

In [18]:
```
d_lat = df_comuni.loc[:,'ISTAT':'Latitudine'].set_index('ISTAT').to_dict()['Latit
d_long = df_comuni.loc[:,'ISTAT':'Longitudine'].set_index('ISTAT').to_dict()['Lor
d_pop = df_comuni.loc[:,'ISTAT':'PopResidente'].set_index('ISTAT').to_dict()['Pop
d_urb = df_comuni.loc[:,'ISTAT':'GradoUrbaniz'].set_index('ISTAT').to_dict()['Gra
```

In [19]:
```
d_area = {'ABRUZZI': 'Sud',
          'BASILICATA': 'Sud',
          'CALABRIA': 'Sud',
          'CAMPANIA': 'Sud',
          'EMILIA-ROMAGNA': 'Nord',
          'FRIULI-VENEZIA GIULIA': 'Nord',
          'LAZIO': 'Centro',
          'LIGURIA': 'Nord',
          'LOMBARDIA': 'Nord',
          'MARCHE': 'Centro',
          'MOLISE': 'Sud',
          'PIEMONTE': 'Nord',
          'PUGLIA': 'Sud',
          'SARDEGNA': 'Isole',
          'SICILIA': 'Isole',
          'TOSCANA': 'Centro',
          'TRENTINO-ALTO ADIGE': 'Nord',
          'UMBRIA': 'Centro',
          "VALLE D'AOSTA": 'Nord',
          'VENETO': 'Nord'}
```

We now add the gathered information to the Vodafone datasets.
As before, we are not able to directly apply the map from the dictionaries, since there are some ISTAT numbers that cannot be found in `comuni_ancitel.csv`, so we rely again on the `f` function.

In [20]:
```
df['Lat'] = df['ISTAT'].apply(f, d = d_lat)
df['Long'] = df['ISTAT'].apply(f, d = d_long)
df['Popolazione'] = df['ISTAT'].apply(f, d = d_pop)
df['GradoUrbanizzazione'] = df['ISTAT'].apply(f, d = d_urb)
```

In [21]:
```
df['GeoArea'] = df['Region'].map(lambda x: d_area[x], na_action='ignore')
```

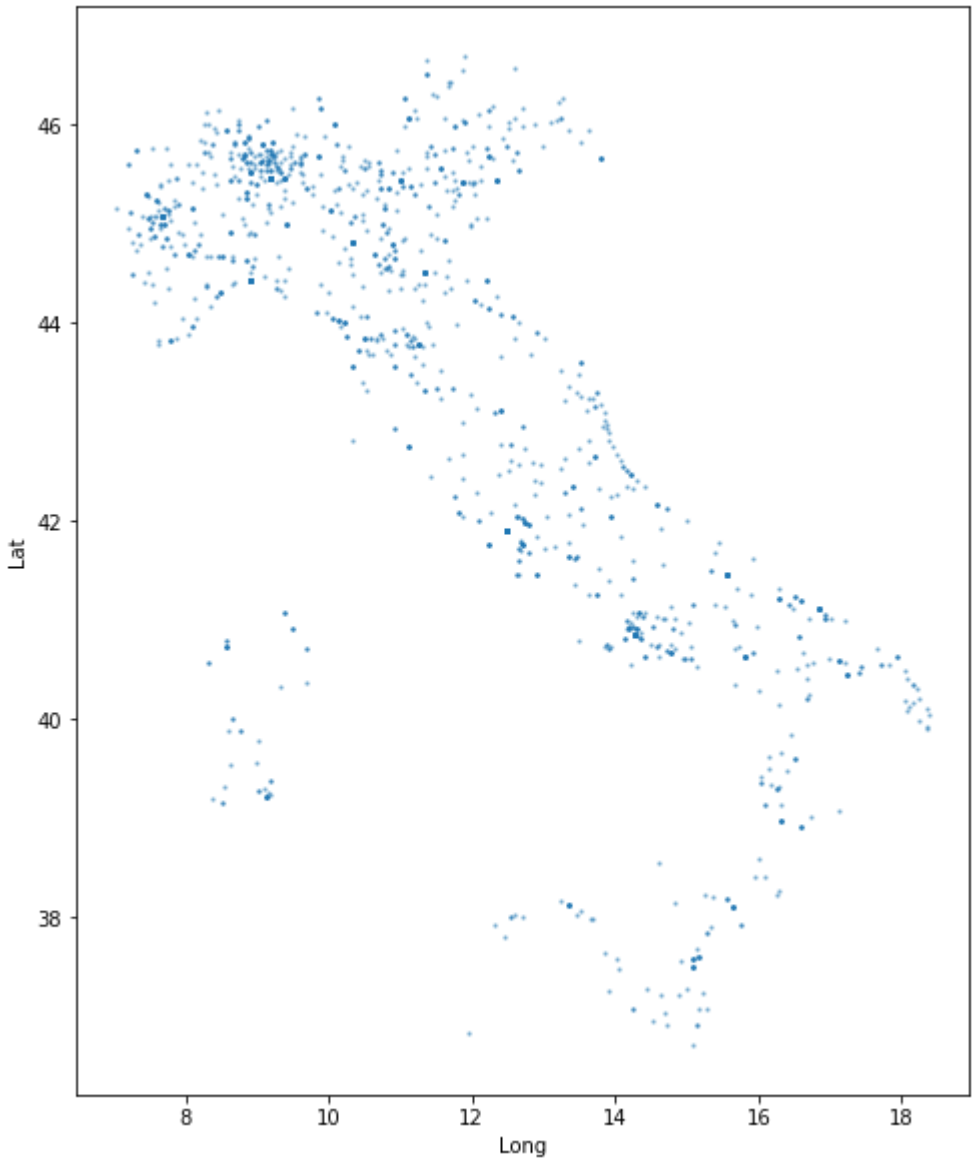
Latitude and Longitude are not recognized as floats, since they use the comma to separate the decimal values instead of the dot.
To deal with this problem we first transform them in string and then use the method `replace`.

In [22]:
```
df['Lat'] = df['Lat'].map(lambda x: float(str(x).replace(",", ".")), na_action='i
df['Long'] = df['Long'].map(lambda x: float(str(x).replace(",", ".")), na_action=
```

Now that `Latitude` and `Longitude` are correctly recognized we can plot them using a scatter plot.

In [23]:
```
df.plot(kind='scatter', x='Long', y='Lat', alpha=100/256, s=2, figsize=(8,10))
```

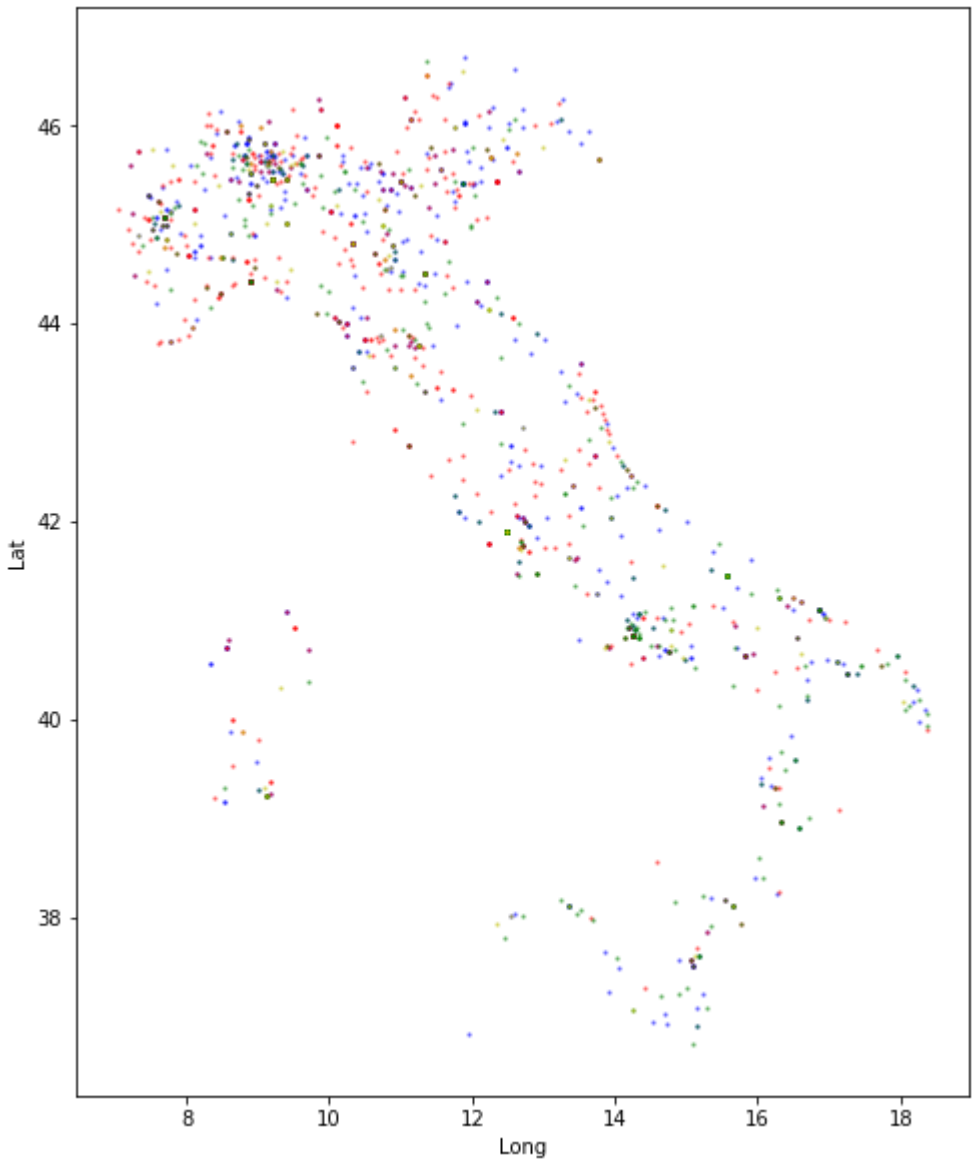
Out[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10b727a90>
```



To see if there is any relevant pattern, we plot the users again, but this time we assign to each specific product a different color.

In [24]:
```
v_bag = df['Product'] == 'V-Bag'
v_pet = df['Product'] == 'V-Pet'
v_auto = df['Product'] == 'V-Auto'
v_camera = df['Product'] == 'V-Camera'

ax = df[v_bag].plot(kind='scatter', x='Long', y='Lat', alpha=100/256, s=2, figsiz
ax = df[v_pet].plot(kind='scatter', x='Long', y='Lat', alpha=100/256, s=2, figsiz
ax = df[v_auto].plot(kind='scatter', x='Long', y='Lat', alpha=100/256, s=2, figsi
ax = df[v_camera].plot(kind='scatter', x='Long', y='Lat', alpha=100/256, s=2, fig
```



We can now delete the columns that we believe to be not particulary useful for our purpose.

In [25]:
```
del df['Province']
del df['Region']
del df['ZipCode']
del df['ISTAT']
```

We now deal with the missing values in the dataset.

Missing Data

These are the main steps covered in this part of the notebook:

- Visualization of the missing data to gain a better understanding of the problem;
- Imputation of the missing data through different methods such as Supervised Learning, Median Imputation, sampling from the distribution, etc.

Visualizing the Missing Data

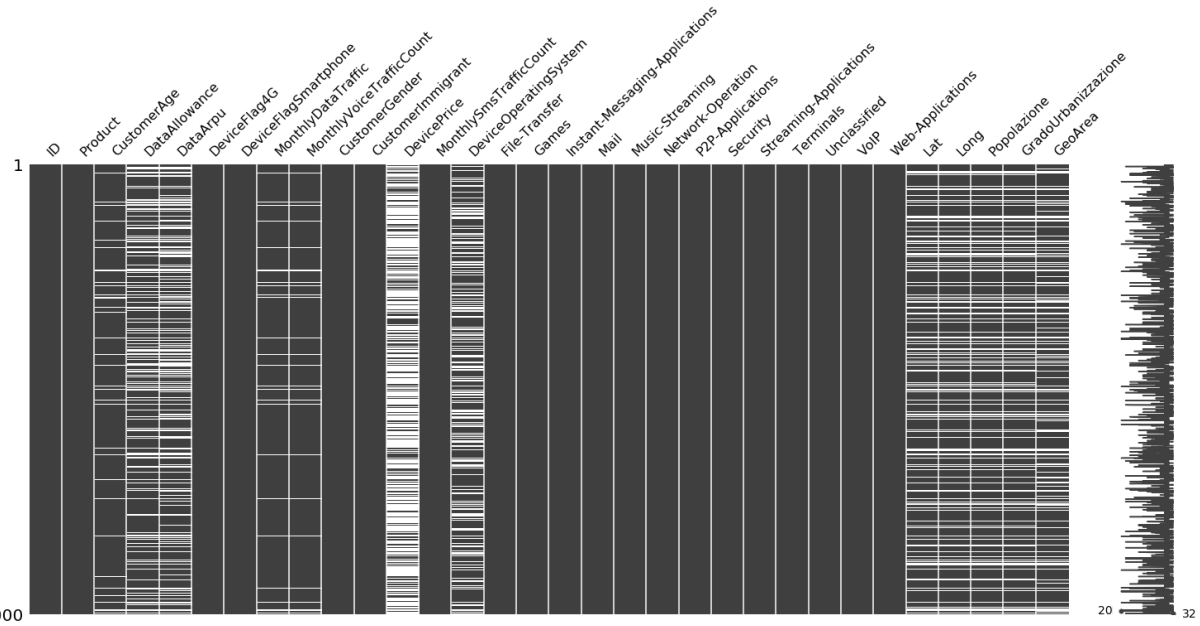
Let's now use the module missingno to get an overview of the missing values by looking for possible correlations and patterns that can help understand relations between categories.

In [26]:
```
import missingno as miss
```

The graph pictures a matrix with the missing data represented by white lines, while the sparkline at the end summarizes the general shape of the data completeness and points out the maximum and minimum rows.

In [27]:
```
miss.matrix(df)
```

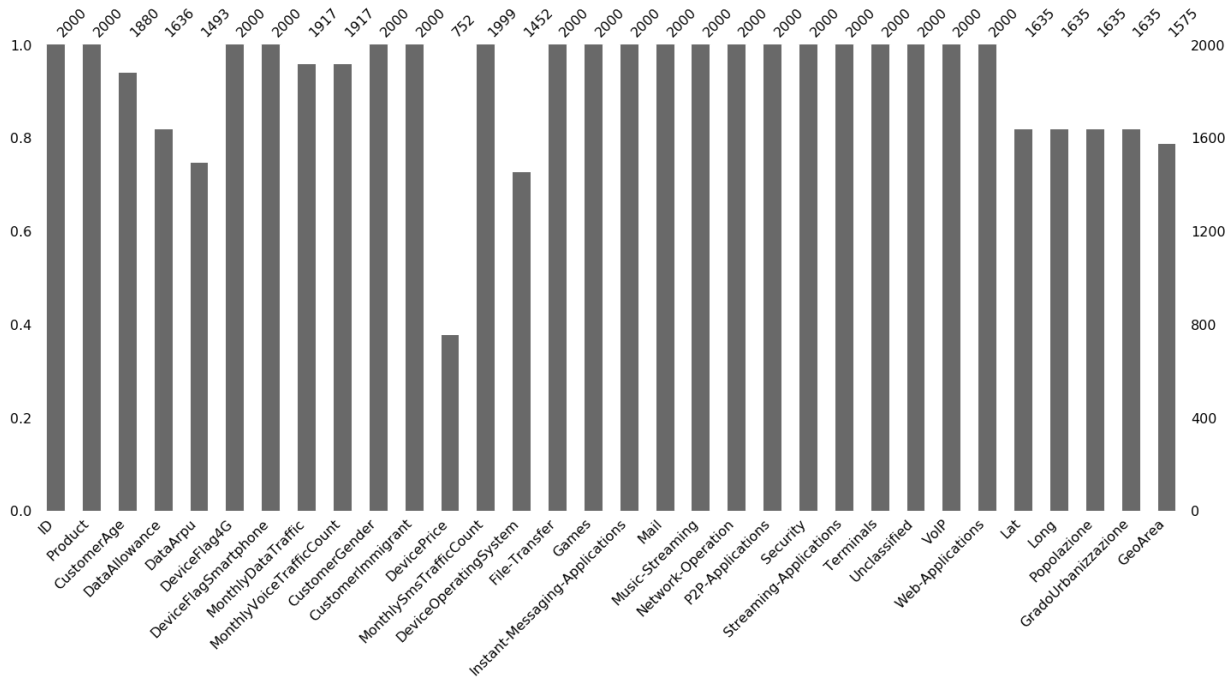
Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1122ea4e0>
```



The histogram shows the amount of data available for each category.

In [28]:
```
miss.bar(df)
```

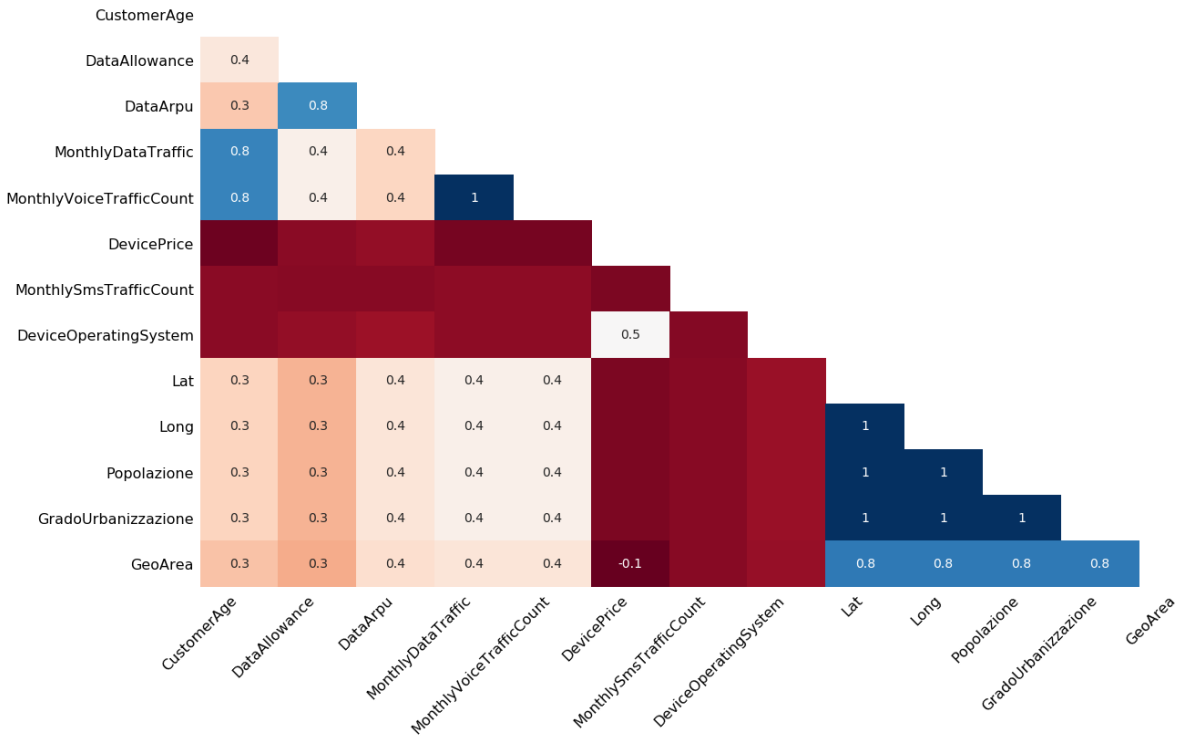
Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10c52b7b8>
```



The graph below provides another point of view of the missing values.
Nullity correlation ranges from -1 (if one variable appears and the other does not) to 0 (variables appearing or not appearing have no effect on one another) to 1 (if one variable appears the other does too).
Variables that are always full or always empty have no meaningful correlation, and so are removed from the visualization. In our case all the traffic columns are removed.
Entries marked < 1 or > -1 have a correlation that is close to be exactly positive or negative, but is still not perfect. If the nullity correlation is very close to zero (-0.05 < R < 0.05), no value will be displayed. Red cells mean negative correlation.
Some correlations are not surprising, like the positive one between `Province` , `Region` and `GeoArea` , or `ZipCode` with `Province` and `Region` , others are quite interesting like `MonthlyDataTraffic` and `MonthlyVoiceTrafficCount` with `CustomerAge` .
The heatmap works great for picking out data completeness relationships between variable pairs, but its explanatory power is limited when it comes to larger relationships and it has no particular support for extremely large datasets.

In [29]:
```
miss.heatmap(df)
```

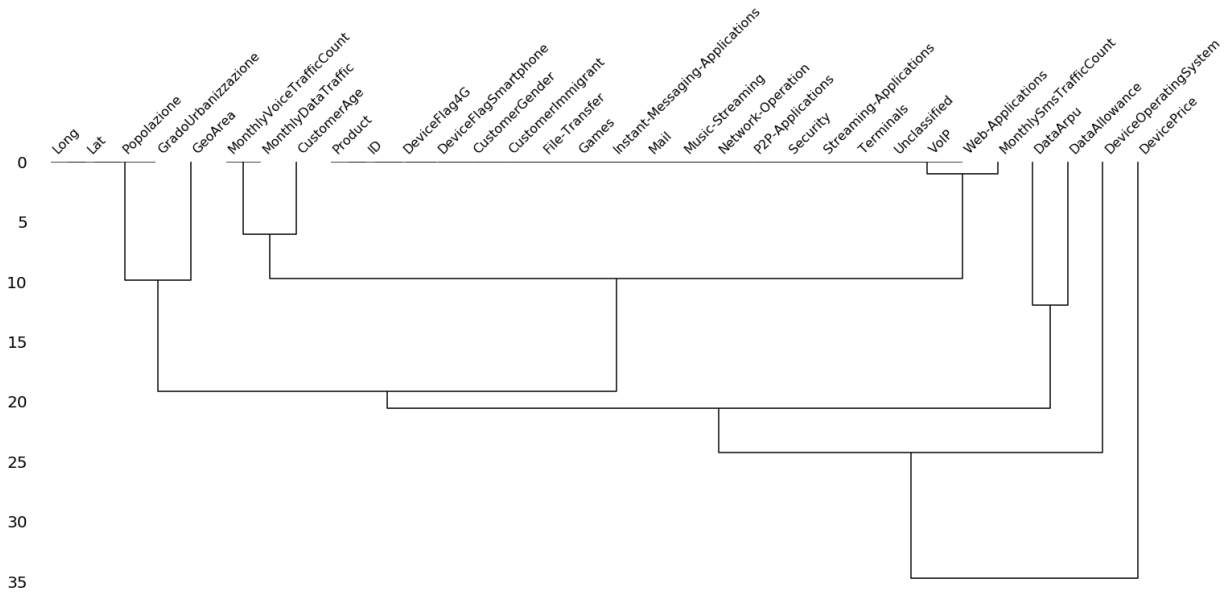
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1a19be0710>



The dendrogram reveals correlations deeper than the pairwise ones of the heatmap. It uses a hierarchical clustering algorithm (from scipy.cluster.hierarchy) to bin variables against one another by their nullity correlation, measured in terms of binary distance. At each step of the tree, the variables are split up based on which combination minimizes the distance of the remaining clusters. The more monotone the set of variables, the closer their total distance is to zero, and the closer their average distance (the y-axis) is to zero. Cluster leaves, linked together at a distance of zero, fully predict one another presence: one variable might always be empty when another is filled, or they might always both be filled or both empty, and so on. Cluster leaves which split close to zero, but not at it, predict one another very well, but still imperfectly.

In [30]:
```
miss.dendrogram(df)
```

Out[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a19bf5c50>
```



Imputation

Let's now initialize some functions and methods that will be useful in the predictions of the missing data.

This function predicts the missing data of a column by sampling randomly from the distribution of that column.

In [31]:
```
def replace_nan_from_distr(column):
    n = 2000 - df[column].count()

    s = df[column].dropna().sample(n)
    s.index = np.where(df[column].isnull())[0]

    return df[column].fillna(s)
```

The function below takes as arguments a predictive model (used to fill the missing data), the columns (used for the prediction) and the label (the column to be predicted).

In [32]:
```
def fill_nan(predictor, columns, label):
    train = df[columns + label].dropna()
    to_predict = df[columns + label][~df[columns + label].index.isin(train.index)
    X_train, X, y_train = train[columns], to_predict[columns], train[label[0]]

    predictor.fit(X_train, y_train)

    label_missing_data = pd.DataFrame(predictor.predict(X))

    label_missing_data.index = np.where(df[label].isnull())[0]

    return df[label[0]].fillna(label_missing_data[0])
```

The following class is used to split the data into train set and validation set and it offers the possibility to perform normalization or PCA.

In [33]:
```
from sklearn.model_selection import train_test_split
```

In [34]:
```
class Data:
    def __init__(self, columns, label, df = df, seed=42):
        data = df[columns + label].dropna()
        X = data[columns]
        y = data[label[0]]

        x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

        self.X, self.y = x_train, y_train
        self.vX, self.vy = x_test, y_test

    def train(self):
        return self.X, self.y

    def valid(self):
        return self.vX, self.vy
```

Now we will import some supervised learning algorithms that we will use to predict the missing data.

In particular, we used the following methods:

- Multi Layer Perceptron
- Random Forest
- Regression
- Ridge
- Bayesian Ridge

In [35]:
```
from sklearn.linear_model import SGDRegressor
```

In [36]:
```
def reg(columns, labels, **reg_params):

    data = Data(columns, labels)

    reg = SGDRegressor(**reg_params)
    reg.fit(*data.train())

    tscore = reg.score(*data.train())
    vscore = reg.score(*data.valid())

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return reg, data, tscore, vscore
```

In [37]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [38]:
```
def rfc(columns, labels, **rfc_params):

    data = Data(columns, labels)

    rfc = RandomForestClassifier(**rfc_params)
    rfc.fit(*data.train())

    tscore = rfc.score(*data.train())
    vscore = rfc.score(*data.valid())

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return rfc, data, tscore, vscore
```

In [39]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [40]:
```
def rfr(columns, labels, n_iter = 10, **rfr_params):

    data = Data(columns, labels)

    score = -1

    for j in range(n_iter):
        rfr = RandomForestRegressor(**rfr_params)
        rfr.fit(*data.train())

        tscore = rfr.score(*data.train())
        vscore = rfr.score(*data.valid())

        if vscore > score:
            rfr_, tscore_, vscore_ = rfr, tscore, vscore
            score = vscore

    print("final tscore=%g vscore=%g" % (tscore_, vscore_))

    return rfr_, data, tscore_, vscore_
```

In [41]:
```
from sklearn.neural_network import MLPClassifier
```

In [42]:
```
def mlpc(columns, labels, **mlp_params):

    data = Data(columns, labels)

    mlperc = MLPClassifier(**mlp_params)
    max_iter = mlp_params.pop('max_iter')

    tscores = []
    vscores = []

    for epoch in range(max_iter):

        mlperc.set_params(max_iter=epoch+1)
        mlperc.fit(*data.train())

        tscore = mlperc.score(*data.train())
        vscore = mlperc.score(*data.valid())

        loss = mlperc.loss_
        tscores.append(tscore)
        vscores.append(vscore)

        mlperc.set_params(warm_start=True)

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return mlperc, data, tscores, vscores
```

In [43]:
```
from sklearn.neural_network import MLPRegressor
```

In [44]:
```
def mlpr(columns, labels, **mlp_params):

    data = Data(columns, labels)

    mlperc = MLPRegressor(**mlp_params)
    max_iter = mlp_params.pop('max_iter')

    tscores = []
    vscores = []

    for epoch in range(max_iter):

        mlperc.set_params(max_iter=epoch+1)
        mlperc.fit(*data.train())

        tscore = mlperc.score(*data.train())
        vscore = mlperc.score(*data.valid())

        loss = mlperc.loss_
        tscores.append(tscore)
        vscores.append(vscore)

        mlperc.set_params(warm_start=True)

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return mlperc, data, tscores, vscores
```

In [45]:
```
from sklearn.linear_model import LinearRegression
```

In [46]:
```
def regression(columns, labels, **reg_params):

    data = Data(columns, labels)

    lm = LinearRegression(**reg_params)

    lm.fit(*data.train())

    tscore = lm.score(*data.train())
    vscore = lm.score(*data.valid())

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return lm, data, tscore, vscore
```

In [47]:
```
from sklearn.linear_model import Ridge
```

In [48]:
```
def ridge(columns, labels, **ridge_params):

    data = Data(columns, labels)

    lm = Ridge(**ridge_params)
    lm.fit(*data.train())

    tscore = lm.score(*data.train())
    vscore = lm.score(*data.valid())

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return lm, data, tscore, vscore
```

In [49]:
```
from sklearn.linear_model import BayesianRidge
```

In [50]:
```
def b_ridge(columns, labels, **b_ridge_params):

    data = Data(columns, labels)

    lm = BayesianRidge(**b_ridge_params)
    lm.fit(*data.train())

    tscore = lm.score(*data.train())
    vscore = lm.score(*data.valid())

    print("final tscore=%g vscore=%g" % (tscore, vscore))

    return lm, data, tscore, vscore
```

In [51]:
```
traffic_columns = ['File-Transfer', 'Games',
                   'Instant-Messaging-Applications', 'Mail', 'Music-Streaming',
                   'Network-Operation', 'P2P-Applications', 'Security',
                   'Streaming-Applications', 'Terminals', 'Unclassified', 'VoIP',
                   'Web-Applications']
columns = traffic_columns + ['DeviceFlag4G', 'DeviceFlagSmartphone', 'CustomerGer
```

Now, we will try to find the best method to fill the missing data of each category. To evaluate the reliability of our model, we take into account the accuracy (for the categorical columns) and the R^2 (for the numerical columns). For now, we consider satisfactory only the predictions with an accuracy greater than 0.5 and an R^2 greater than 0.1.
We then look at the distributions before and after the imputation and look at how they are affected by the predictions.
As a final step, we decide whether to use the predictive model, to use imputation by median or to perform random sampling from the distribution.

We only show the two best results that we found for each category.

Categorical Columns

We now introduce a function , `plot_comparison_categorical` , which will help us notice how the distribution is influenced by filling the NaN through the use of the chosen method. Random Forest and random sampling from the distribution are the methods that have given the best results. Therefore, we print the original distribution on the left, the distribution after predicting with Random Forest in the centre and the distribution after sampling from the column on the right.

In [52]:
```
def plot_comparison_categorical(label):

    fig = plt.figure(figsize = (16, 5))

    ax = fig.add_subplot(1, 3, 1)
    df[label].hist()
    plt.title('True Distribution of %s' %label)

    ax = fig.add_subplot(1, 3, 2)
    rf_nan.hist(color='green')
    plt.title('After Predicting with Random Forest')

    ax = fig.add_subplot(1, 3, 3)
    distr_nan.hist(color='red')
    plt.title('After Sampling from Distribution')
```

DeviceOperatingSystem

In [53]:
```
label = ['DeviceOperatingSystem']
```

In [54]:
```
df['DeviceOperatingSystem'].value_counts()
```

Out[54]:
```
Android    858
iOS        568
Other       26
Name: DeviceOperatingSystem, dtype: int64
```

Random Forest Classifier:

In [55]:
```
rfc_, data, tscore, vscore = rfc(columns, label, n_estimators=117, criterion='gir
                                 min_samples_split=5, min_samples_leaf=1, mir
                                 max_features='auto', max_leaf_nodes=None, bc
                                 n_jobs=1, random_state=None, verbose=0, warn
```

```
final tscore=0.988803 vscore=0.941581
```

Multilayer Perceptron:

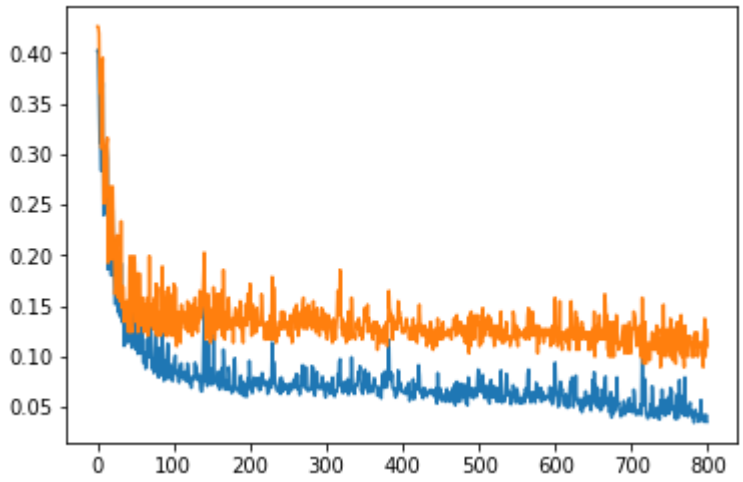
In [56]:
```
mlp, data, tscores, vscores = mlpc(columns, label, max_iter=800, hidden_layer_siz
                                   batch_size = 110, learning_rate_init=1e-1,
                                   learning_rate = 'constant', momentum = 0.0
                                   verbose = False, alpha=0.0, tol = -1)
```

```
/Users/mariastellacocco/anaconda3/lib/python3.6/site-packages/sklearn/neural_ne
twork/multilayer_perceptron.py:564: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (1) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

final tscore=0.964686 vscore=0.890034
```

From the plot we can see that the score improves greatly in the first 100 iterations, and then it stabilizes (although a lot of variability remains).

In [57]:
```
plt.plot(range(len(tscores)), 1-np.array(tscores), range(len(vscores)), 1-np.arra
```



Both methods give satisfactory results, in particular Random Forest, in which we get a validation score of 0.96. Thus, we proceed by using the Random Forest Classifier to predict the `DeviceOperatingSystem` column.

In [58]:
```
df['DeviceOperatingSystem'] = fill_nan(rfc_, columns, label)
```

CustomerAge

In [59]:
```
label = ['CustomerAge']
```

In [60]:
```
df['CustomerAge'].value_counts()
```

Out[60]:
```
(40, 50]    557
(50, 60]    409
(30, 40]    380
(20, 30]    269
(60, 70]    182
(70, 80]     62
(10, 20]     12
(80, 90]      9
Name: CustomerAge, dtype: int64
```

Random Forest Classifier:

In [61]:
```
rfc_, data, tscore, vscore = rfc(columns, label, n_estimators=117, criterion='gi
                                 min_samples_split=5, min_samples_leaf=1, mi
                                 max_features='auto', max_leaf_nodes=None, b
                                 n_jobs=1, random_state=None, verbose=0, war
```

```
final tscore=0.999335 vscore=0.324468
```

MLPClassifier:

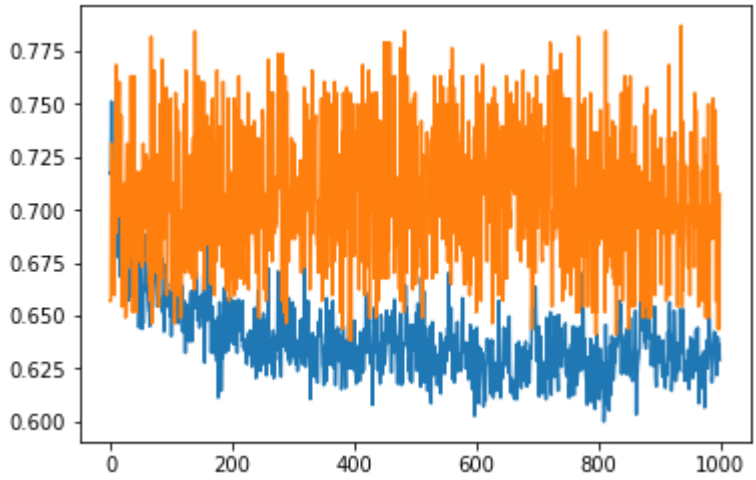
In [62]:
```
mlp, data, tscores, vscores = mlpc(columns, label, max_iter=1000, hidden_layer_si
                                   batch_size = 120, learning_rate_init=1e-1, s
                                   learning_rate = 'constant', momentum = 0.0,
                                   verbose = False, alpha=0.0, tol = -1)
```

```
/Users/mariastellacocco/anaconda3/lib/python3.6/site-packages/sklearn/neural_ne
twork/multilayer_perceptron.py:564: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (1) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

final tscore=0.371011 vscore=0.292553
```

It is clear from the plot that the algorithm struggles to find an optima.

In [63]:
```
plt.plot(range(len(tscores)), 1-np.array(tscores), range(len(vscores)), 1-np.arra
```

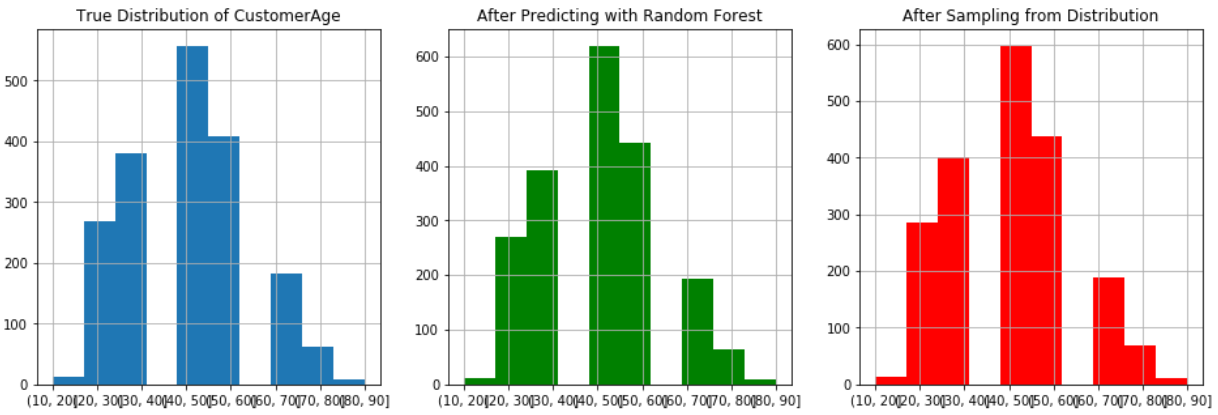


In this case we cannot obtain applicable results with any learning models.
We see what would happen to the distribution if we filled the missing values by sampling at random from the column.

In [64]:
```
rf_nan =  fill_nan(rfc_, columns, label)
```

In [65]:
```
distr_nan = replace_nan_from_distr('CustomerAge')
```

In [66]:
```
plot_comparison_categorical('CustomerAge')
```



We decide to predict the missing values using the Random Forest algorithm.

In [67]:
```
df['CustomerAge'] = fill_nan(rfc_, columns, label)
```

GradoUrbanizzazione

In [68]:
```
label = ['GradoUrbanizzazione']
```

In [69]:
```
df['GradoUrbanizzazione'].value_counts()
```

Out[69]:
```
Elevato    763
Medio      609
Basso      263
Name: GradoUrbanizzazione, dtype: int64
```

Random Forest Classifier:

In [70]:
```
rfc_, data, tscore, vscore = rfc(columns, label, n_estimators=117, criterion='gi
                                 min_samples_split=5, min_samples_leaf=1, mi
                                 max_features='auto', max_leaf_nodes=None, b
                                 n_jobs=1, random_state=None, verbose=0, war
```

```
final tscore=1 vscore=0.443425
```

MLPClassifier:

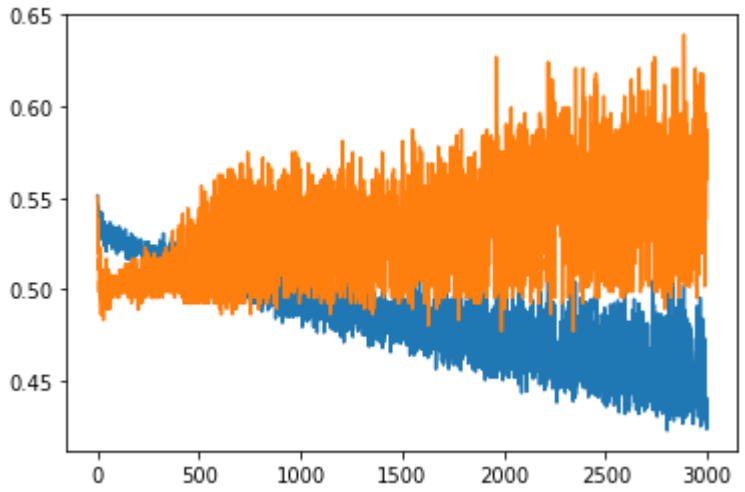
In [71]:
```
mlp, data, tscores, vscores = mlpc(columns, label, max_iter=3000, hidden_layer_si
                                   batch_size = 120, learning_rate_init=1e-1, s
                                   learning_rate = 'constant', momentum = 0.0,
                                   verbose = False, alpha=0.0, tol = -1)
```

```
/Users/mariastellacocco/anaconda3/lib/python3.6/site-packages/sklearn/neural_ne
twork/multilayer_perceptron.py:564: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (1) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

final tscore=0.559633 vscore=0.434251
```

Also in this case, we notice that the Multi Layer Perceptron does not converge as the number of iterations grows.

In [72]:
```
plt.plot(range(len(tscores)), 1-np.array(tscores), range(len(vscores)), 1-np.arra
```



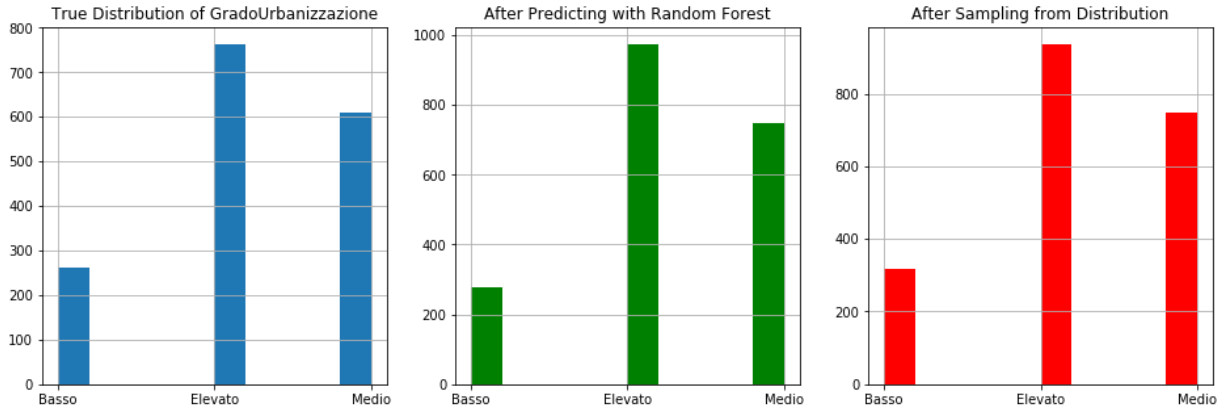
In both cases we obtain results around 0.5. They are not completely satisfactory but we do not a

priori refuse to use them.
We try to predict the results by random sampling from the distribution and compare how this method behaves with respect to the prediction.

In [73]:
```
rf_nan =  fill_nan(rfc_, columns, label)
```

In [74]:
```
distr_nan = replace_nan_from_distr('GradoUrbanizzazione')
```

In [75]:
```
plot_comparison_categorical('GradoUrbanizzazione')
```



We decide to use the prediction with Random Forest, as the score is acceptable and it behaves quite well with respect to the original distribution.

In [76]:
```
df['GradoUrbanizzazione'] = fill_nan(rfc_, columns, label)
```

GeoArea

In [77]:
```
label = ['GeoArea']
```

In [78]:
```
df['GeoArea'].value_counts()
```

Out[78]:
```
Nord      808
Centro    337
Sud       316
Isole     114
Name: GeoArea, dtype: int64
```

Random Forest Classifier:

In [79]:
```
rfc_, data, tscore, vscore = rfc(columns, label, n_estimators=117, criterion='gi
                                 min_samples_split=5, min_samples_leaf=1, min_
                                 max_features='auto', max_leaf_nodes=None, boo
                                 n_jobs=1, random_state=None, verbose=0, warm_
```

```
final tscore=0.998413 vscore=0.463492
```

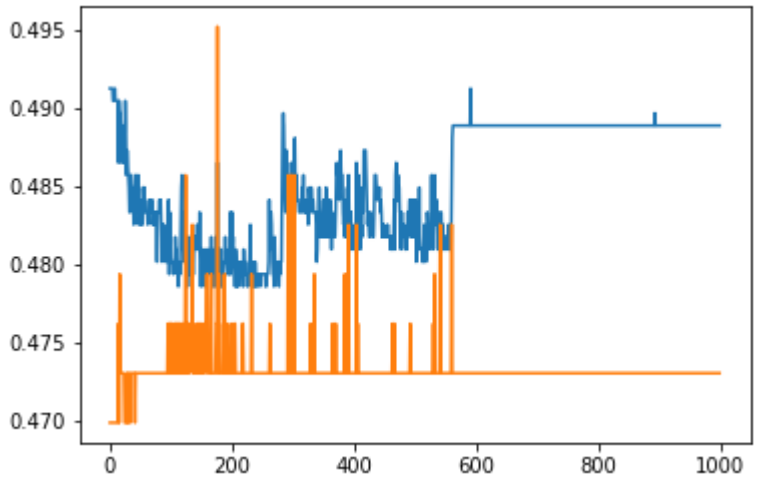
MLPClassifier:

In [80]:
```
mlp_, data, tscores, vscores = mlpc(columns, label, max_iter=1000, hidden_layer_s
                                    batch_size = 100, learning_rate_init=1e-1, s
                                    learning_rate = 'constant', momentum = 0.0,
                                    verbose = False, alpha=0.0, tol = -1)
```

```
/Users/mariastellacocco/anaconda3/lib/python3.6/site-packages/sklearn/neural_ne
twork/multilayer_perceptron.py:564: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (1) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

final tscore=0.511111 vscore=0.526984
```

In [81]:
```
plt.plot(range(len(tscores)), 1-np.array(tscores), range(len(vscores)), 1-np.arra
```

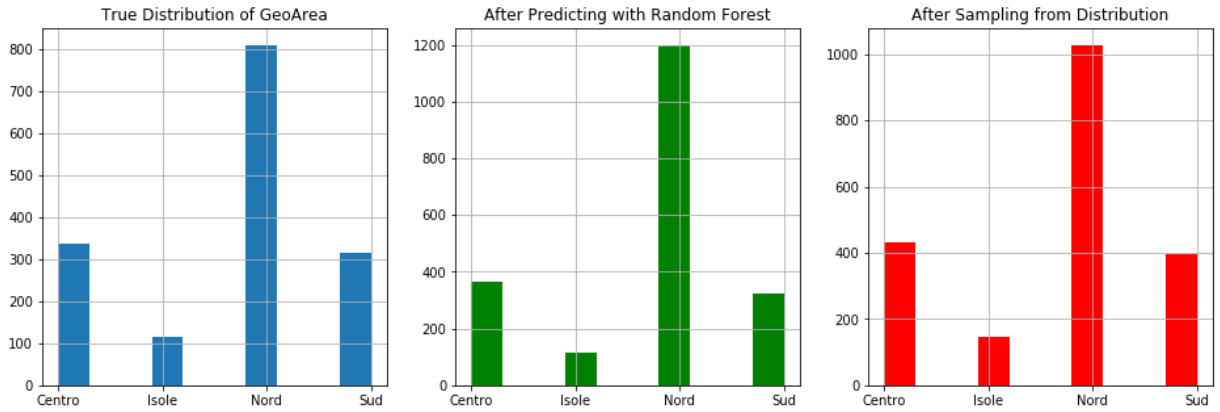


The scores are very similar to what we have encountered in `GradoUrbanizzazione`. Therefore, we proceed in the same way.

In [82]:
```
rf_nan = fill_nan(rfc_, columns, label)
```

In [83]:
```
distr_nan = replace_nan_from_distr('GeoArea')
```

In [84]:
```
plot_comparison_categorical('GeoArea')
```



In [85]:
```
df['GeoArea'] = fill_nan(rfc_, columns, label)
```

Numerical Columns

Regarding the numerical columns, we proceed by looking at the score obtained with Random Forest or other regression methods, plotting the density that we would obtain using this method and compare it to what we would get if we used the median to fill the missing values.

In [86]:
```
def plot_comparison_numerical(label, pred_model, figsize = (16, 5)):

    fig = plt.figure(figsize = figsize)

    ax = fig.add_subplot(1, 3, 1)
    df[label].plot.density()
    plt.title('True Distribution of %s' %label)

    ax = fig.add_subplot(1, 3, 2)
    pred_model.plot.density(color='green')
    plt.title('After Predicting with the Learning Model')

    ax = fig.add_subplot(1, 3, 3)
    median_nan.plot.density(color='red')
    plt.title('After Replacing with the Median')
```

DataAllowance

In [87]:
```
label = ['DataAllowance']
```

Random Forest Regressor:

In [88]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.813654 vscore=0.00381922
```

Ridge:

In [89]:
```
ri, data, tscores, vscores = ridge(columns, label)
```

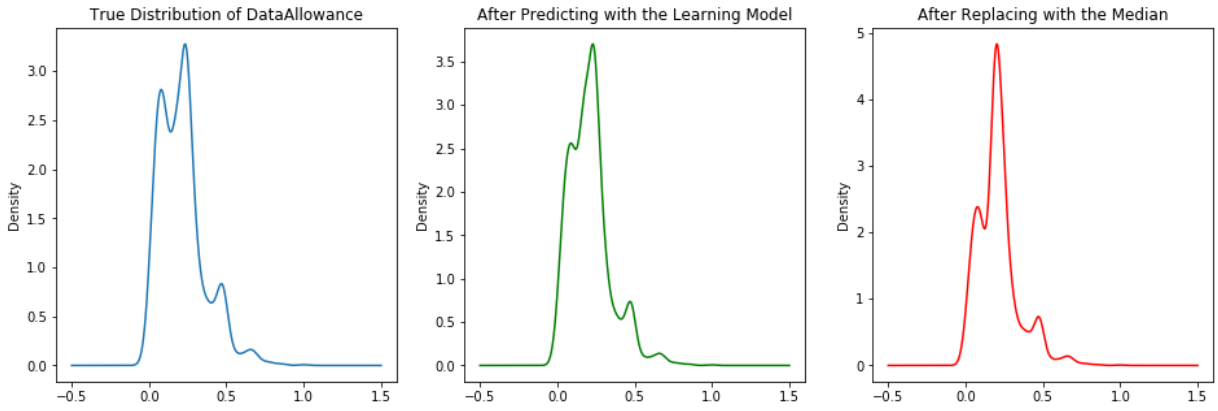
```
final tscore=0.0324718 vscore=0.0309073
```

The results obtained with Random Forest and with Ridge are quite bad. Therefore, we make a comparison between using the prediction and using the median to fill the NaN in order to verify if using the median would be a better choice.

In [90]:
```
median_nan = df['DataAllowance'].fillna((df['DataAllowance'].median()), inplace=T
```

In [91]:
```
rf_nan =  fill_nan(rfr_, columns, label)
```

In [92]:
```
plot_comparison_numerical('DataAllowance', rf_nan)
```



As we can see from the plots, Random Forest is the method that less alters the distribution. Thus, we proceed by using the latter algorithm.

In [93]:
```
df['DataAllowance'] = fill_nan(rfr_, columns, label)
```

DataArpu

In [94]:
```
label = ['DataArpu']
```

Random Forest Regressor:

In [95]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.809965 vscore=-0.069409
```

Bayesian Ridge:

In [96]:
```
br, data, tscores, vscores = b_ridge(columns, label)
```

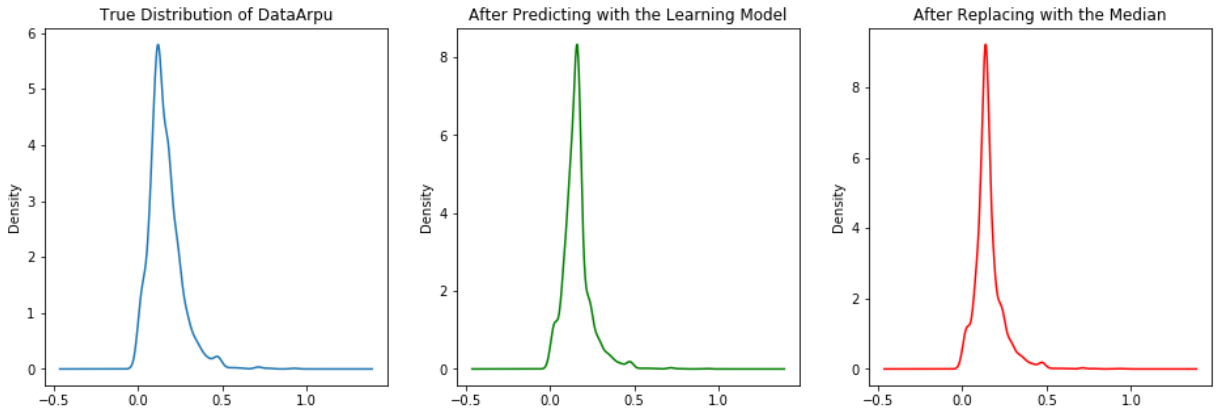
```
final tscore=0.0232459 vscore=-0.0572624
```

As with the DataAllowance column, the results obtained with the prediction are not good at all. Hence, we see if filling the NaN with the median would be preferable.

In [97]:
```
median_nan = df['DataArpu'].fillna((df['DataArpu'].median()), inplace=False)
```

In [98]:
```
br_nan =  fill_nan(br, columns, label)
```

In [99]:
```
plot_comparison_numerical('DataArpu', br_nan)
```



We proceed by using the Bayesian Ridge algorithm as it seems to resemble the original distribution slightly better than the distribution obtained after replacing with the median.

In [100]:
```
df['DataArpu'] = fill_nan(br, columns, label)
```

MontlyVoiceTrafficCount

Now, in the next two columns, `MonthlyVoiceTrafficCount` and `MonthlyDataTraffic` , there are only 83 values missing. Therefore, as shown in the plots, predicting or replacing them with the median does not make any relevant difference.

In [101]:
```
label = ['MonthlyVoiceTrafficCount']
```

In [102]:
```
df['MonthlyVoiceTrafficCount'].count()
```

Out[102]: 1917

Random Forest Regressor:

In [103]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.821544 vscore=0.0674098
```

Bayesian Ridge:

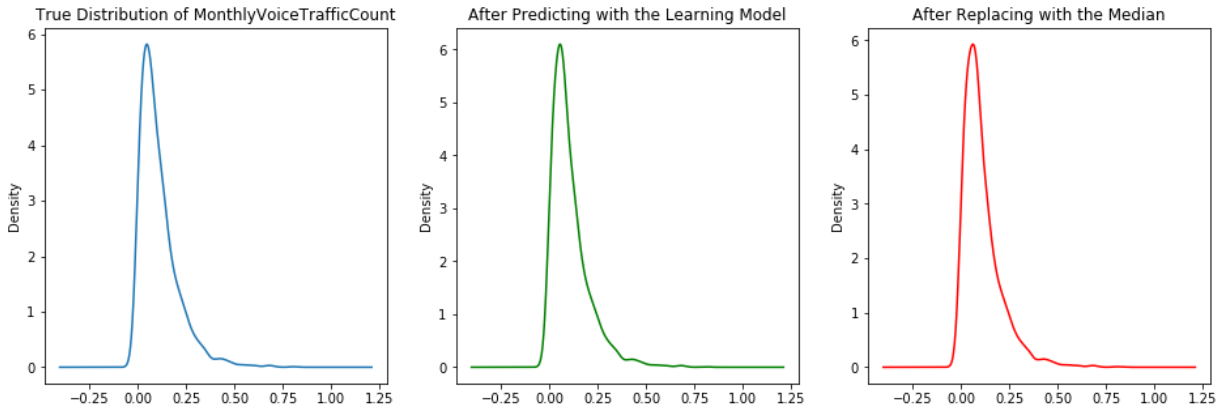
In [104]:
```
br, data, tscores, vscores = b_ridge(columns, label)
```

```
final tscore=0.0339367 vscore=0.0278568
```

In [105]:
```
median_nan = df['MonthlyVoiceTrafficCount'].fillna((df['MonthlyVoiceTrafficCount'
```

In [106]:
```
br_nan =  fill_nan(br, columns, label)
```

In [107]:
```
plot_comparison_numerical('MonthlyVoiceTrafficCount', br_nan)
```



In [108]:
```
df['MonthlyVoiceTrafficCount'] = fill_nan(br, columns, label)
```

MonthlyDataTraffic

In [109]:
```
label = ['MonthlyDataTraffic']
```

In [110]:
```
df['MonthlyDataTraffic'].count()
```

Out[110]: 1917

Random Forest Regressor:

In [111]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.85804 vscore=0.220464
```

Bayesian Ridge:

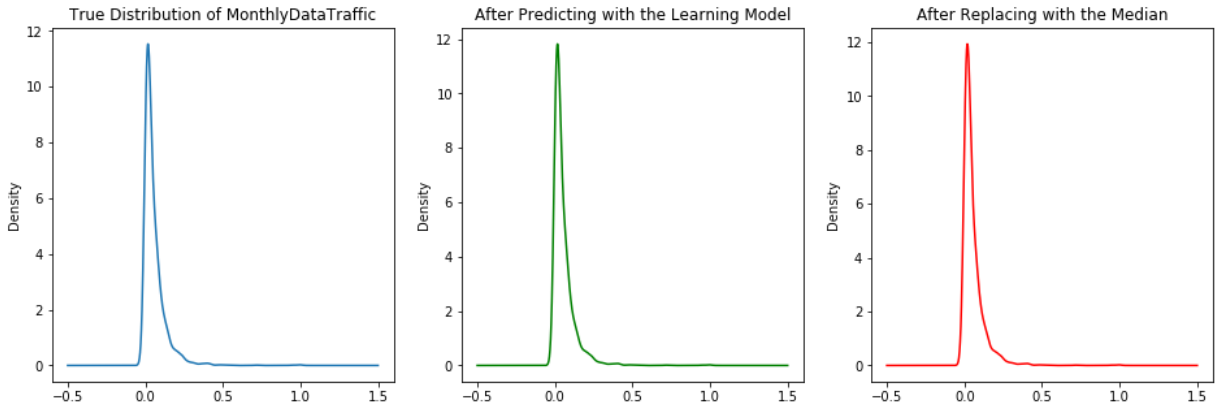
In [112]:
```
lm, data, tscore, vscore = b_ridge(columns, label)
```

```
final tscore=0.143405 vscore=0.0707411
```

In [113]:
```
median_nan = df['MonthlyDataTraffic'].fillna((df['MonthlyDataTraffic'].median()),
```

In [114]:
```
rf_nan =  fill_nan(rfr_, columns, label)
```

In [115]:
```
plot_comparison_numerical('MonthlyDataTraffic', rf_nan)
```



In [116]:
```
df['MonthlyDataTraffic'] = fill_nan(rfr_, columns, label)
```

DevicePrice

In [117]:
```
label = ['DevicePrice']
```

Random Forest Regressor:

In [118]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.881972 vscore=0.44006
```

MLPRegressor:

In [119]:
```
lr, data, tscore, vscore = regression(columns, label)
```

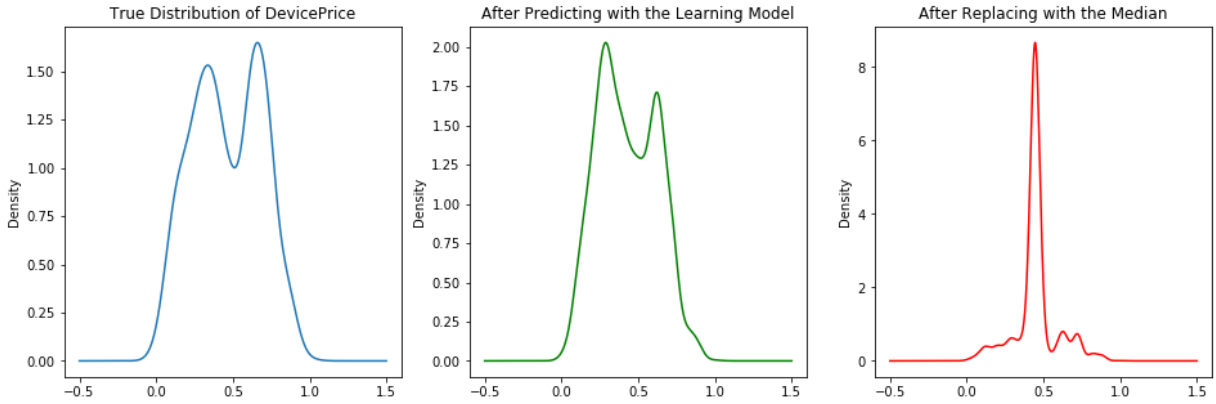
```
final tscore=0.222073 vscore=0.0227307
```

We get a quite acceptable R^2 score using Random Forest, thus, also on the basis of what we would get if we replaced the missing data with the median, we proceed by using prediction.

In [120]:
```
median_nan = df['DevicePrice'].fillna((df['DevicePrice'].median()), inplace=False
```

In [121]:
```
rf_nan =  fill_nan(rfr_, columns, label)
```

In [122]:
```
plot_comparison_numerical('DevicePrice', rf_nan)
```



In [123]:
```
df['DevicePrice'] = fill_nan(rfr_, columns, label)
```

MonthlySmsTrafficCount

In this case, there is only one missing data. Hence we replace it with the median.

In [124]:
```
label = ['MonthlySmsTrafficCount']
```

In [125]:
```
df['MonthlySmsTrafficCount'].count()
```

Out[125]: 1999

In [126]:
```
df['MonthlySmsTrafficCount'] = df['MonthlySmsTrafficCount'].fillna((df['MonthlySm
```

Popolazione

In [127]:
```
label = ['Popolazione']
```

Random Forest Regressor

In [128]:
```
rfr_, data, tscore, vscore = rfr(columns, label, n_iter = 100)
```

```
final tscore=0.773363 vscore=0.00960071
```

In [129]:
```
lm, data, tscore, vscore = regression(columns, label)
```

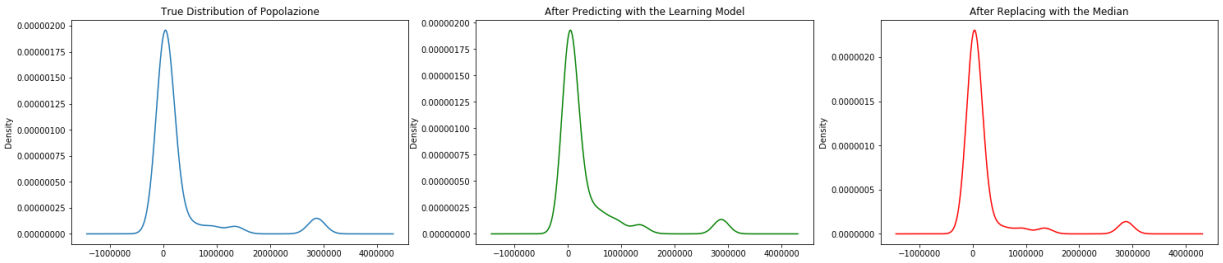
```
final tscore=0.00997363 vscore=-0.015946
```

The results obtained with the predictions are very unsatisfactory.
Let's see if we obtain something better by using the median.

In [130]:
```
median_nan = df['Popolazione'].fillna((df['Popolazione'].median()), inplace=False
```

In [131]:
```
rf_nan =  fill_nan(rfr_, columns, label)
```

In [132]:
```
plot_comparison_numerical('Popolazione', rf_nan, figsize = (25, 5))
```



As we can see from the plots, replacing the missing values with the median seems to be a slightly better choice.

In [133]:
```
df['Popolazione'] = df['Popolazione'].fillna((df['Popolazione'].median()), inplac
```

We now transform CustomerAge into a numerical variable.

In [134]:
```
d_age = {'(10, 20]':0.15, '(20, 30]':0.25, '(30, 40]':0.35, '(40, 50]':0.45, '(50
           '(70, 80]':0.75, '(80, 90]':0.85}

df['CustomerAge'] = df['CustomerAge'].map(lambda z: d_age[z], na_action='ignore')
```

We Create a new dataframe with one hot encoding applied to the categorical variable.

In [135]:
```
df_ohe = pd.get_dummies(df, columns = ['DeviceOperatingSystem', 'GeoArea', 'Grado
```

Now that we have finally completed the missing data, we can pass to the main point of the challenge: clustering.

Clustering

In [136]:
```
from sklearn.decomposition import PCA
```

Since we have the labels, in order to estimate the goodness of our cluster, we define a function as an external criterion of clustering quality: purity. To compute it, each cluster is assigned to the label which is most frequent in the cluster, and then the accuracy of this assignment is measured by counting the number of correctly assigned objects and dividing by the total number of data points in the dataset.

$$purity(W, \mathrm{C}) = \frac{1}{N}\sum_{k} \max_{j} \; |w_k \cap c_j|$$

where N stands for the total number of data points in the dataset (2000 in our case); W is the set of clusters; C is the set of labels; k is the number of clusters; j is the number of labels.

In [137]:
```
def purity(df, labels, verbose = False):
    cluster_map = pd.DataFrame()
    cluster_map['ID'] = df.index.values
    cluster_map['Cluster'] = labels
    cluster_map = cluster_map.join(df['Product'])

    x = 0

    for i in np.unique(labels):
        c = cluster_map[cluster_map['Cluster'] == i]['Product'].value_counts()
        x += max(c)

        if verbose:
            print('cluster:', i)
            print(c)

    return x / df.shape[0]
```

In [138]:
```
data_columns = ['CustomerAge', 'DataAllowance', 'DataArpu',
                'DeviceFlag4G', 'DeviceFlagSmartphone', 'MonthlyDataTraffic',
                'MonthlyVoiceTrafficCount', 'CustomerGender', 'CustomerImmigrant',
                'DevicePrice', 'MonthlySmsTrafficCount', 'DeviceOperatingSystem',
                'File-Transfer', 'Games', 'Instant-Messaging-Applications', 'Mail'
                'Music-Streaming', 'Network-Operation', 'P2P-Applications', 'Secur
                'Streaming-Applications', 'Terminals', 'Unclassified', 'VoIP',
                'Web-Applications', 'Popolazione', 'GradoUrbanizzazione','GeoArea'

numerical_columns = ['CustomerAge', 'DataAllowance', 'DataArpu',
                     'MonthlyDataTraffic', 'MonthlyVoiceTrafficCount',
                     'DevicePrice', 'MonthlySmsTrafficCount',
                     'File-Transfer', 'Games', 'Instant-Messaging-Applications', 'M
                     'Music-Streaming', 'Network-Operation', 'P2P-Applications', 'S
                     'Streaming-Applications', 'Terminals', 'Unclassified', 'VoIP',
                     'Web-Applications']

data_ohe = ['CustomerAge', 'DataAllowance', 'DataArpu',
            'DeviceFlag4G', 'DeviceFlagSmartphone', 'MonthlyDataTraffic',
            'MonthlyVoiceTrafficCount', 'CustomerGender', 'CustomerImmigrant',
            'DevicePrice', 'MonthlySmsTrafficCount', 'File-Transfer', 'Games',
            'Instant-Messaging-Applications', 'Mail', 'Music-Streaming',
            'Network-Operation', 'P2P-Applications', 'Security',
            'Streaming-Applications', 'Terminals', 'Unclassified', 'VoIP',
            'Web-Applications', 'Popolazione',
            'DeviceOperatingSystem_Android', 'DeviceOperatingSystem_Other',
            'DeviceOperatingSystem_iOS', 'GeoArea_Centro', 'GeoArea_Isole',
            'GeoArea_Nord', 'GeoArea_Sud', 'GradoUrbanizzazione_Basso',
            'GradoUrbanizzazione_Elevato', 'GradoUrbanizzazione_Medio']
```

Since our data is very heterogeneous, applying traditional distances does not lead to satisfactory results. Therefore, we define a new distance:

$$d(x,y) = \sqrt{\sum_i w_i^2 (x_i - y_i)^2}$$

In order to use this distance, we need to find the weights.

In order to find these weights, we apply Simulated Annealing to the K-Nearest-Neighbors algorithm, trying to minimize the cost and then use the weights to transform our data.

In [139]:
```
X = df[numerical_columns]
X_ohe = df_ohe[data_ohe]
labels = df['Product']
```

K-nearest neighbors

In [140]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
```

Before applying Simulated Annealing we find the optimal parameters for K-Nearest-Neighbors.

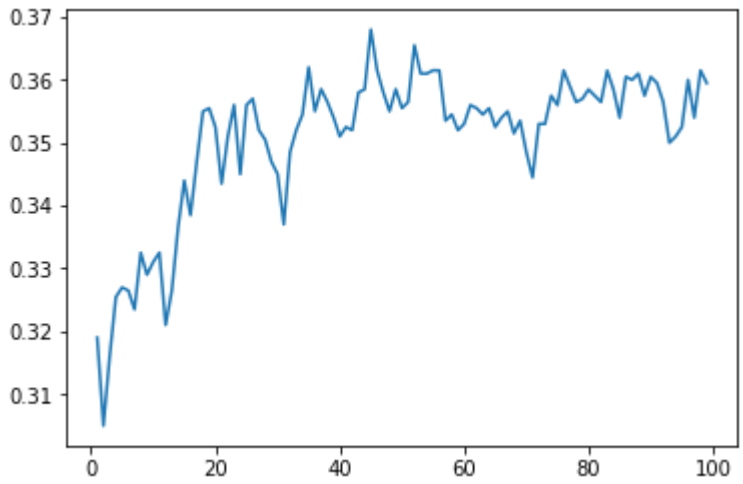
In [141]:
```
def knn(X, labels, n_neighbors = 100, **knn_params):
    scores = np.zeros(n_neighbors)

    for i in range(1, n_neighbors):
        neigh = KNeighborsClassifier(i, **knn_params)
        scores[i] = np.mean(cross_val_score(neigh, X, labels))

    plt.plot([i for i in range(1, n_neighbors)], scores[1:])
    print('n_neighbors:', np.argmax(scores))
    print('accuracy:', np.max(scores))
```

In [142]:
```
knn(X, labels)
```

```
n_neighbors: 45
accuracy: 0.36799428389
```



Let's now apply Simulated Annealing but, as we can see, the improvement is not particularly relevant since we only gain a couple of percentage points.

In [143]:
```
import SimAnn
import Weights_Finder_knn
```

In [144]:
```
labels = df['Product']
X = X.as_matrix()
```

In [145]:
```
probl = Weights_Finder_knn.Weights_Finder(X, labels, n_neighbors = 38, step = 0.1

probl, best = SimAnn.simann(probl, iters=5, seed=None,
                            beta0=25.0, beta1=175.0, beta_steps=5,
                            debug_delta_cost= True)
```

```
T= 0.04
  costs: current= 0.638988744788 best= 0.638988744788
  acceptance rate= 1.0
T= 0.016
  costs: current= 0.645490760315 best= 0.638988744788
  acceptance rate= 1.0
T= 0.01
  costs: current= 0.644991758319 best= 0.638988744788
  acceptance rate= 0.8
T= 0.00727272727273
  costs: current= 0.645987497674 best= 0.638988744788
  acceptance rate= 1.0
T= 0.00571428571429
  costs: current= 0.646999763366 best= 0.638988744788
  acceptance rate= 0.6
T= 0.0
  costs: current= 0.635506224871 best= 0.635506224871
  acceptance rate= 0.8
```

In [146]:
```
w = best.w
```

We start the unsupervised clustering with K-Means, being one of the most simple and popular algorithms in this field.

K-Means

In [147]:
```
from sklearn.cluster import KMeans
```

In [148]:
```
from sklearn.metrics import silhouette_score
```

In [149]:
```
def k_means(X, n_components = 0, scale = False, verbose = False, df = df, **km_pa
    if scale:
        scaler = StandardScaler()
        X = scaler.fit_transform(X)

    dopca = n_components is None or n_components > 0

    if dopca:
            pca = PCA(n_components = n_components)
            X = pca.fit_transform(X)

    km = KMeans(**km_params)
    km.fit(X)
    score = purity(df, km.labels_, verbose)

    return km, score
```

We first try to find the right number of clusters.
Using the silhouette metric, we notice that the quality of our clusters quickly decreases as we increase the number of clusters.

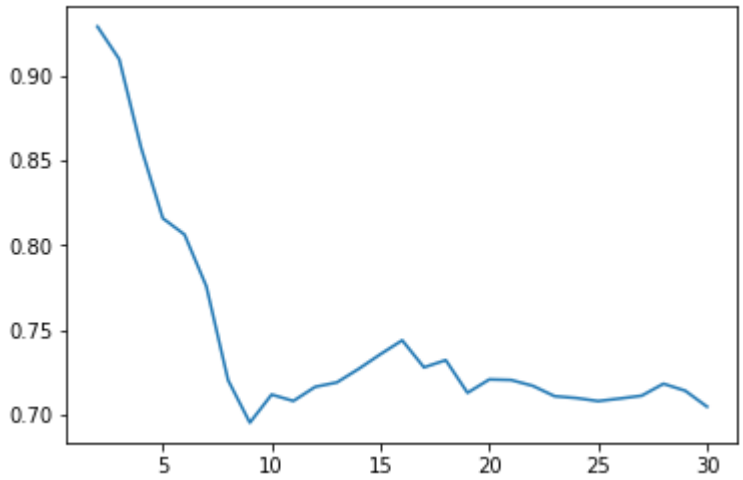
In [150]:
```
scores = []

for i in range(2, 31):

    km, puriry = k_means(X_ohe, n_clusters = i)
    scores.append(silhouette_score(X_ohe, km.labels_))

plt.plot(np.arange(2, 31), scores)
```

Out[150]:
```
[<matplotlib.lines.Line2D at 0x112b0d668>]
```



disclaimer
The functions `check_distr()`, `prod_in_clust()`, `plot_hists()` have been developed jointly with Group 8.

In [151]:
```
df['Numeric_Product'] = df['Product'].astype('category')

product_legend = [df['Numeric_Product'].cat.categories]

df['Numeric_Product'].cat.categories = np.arange(4)
df['Numeric_Product'] = df['Numeric_Product'].astype(np.int64)
```

In [152]:
```
product_distr = (df['Numeric_Product'].value_counts()[np.arange(4)] / 2000).as_ma
product_distr
```

Out[152]:
```
array([ 0.225,  0.37 ,  0.083,  0.322])
```

In [153]:
```
def cluster_distribution(km, df = df):
    products = df['Numeric_Product']
    labels = km.labels_

    clusters = np.unique(km.labels_)
    distr = np.zeros((km.n_clusters, 4))

    for cluster in clusters:
        mask = km.labels_ == cluster
        distr[cluster] = products[mask].value_counts().sort_index().reindex([i fo

    distr = np.nan_to_num(distr)

    return distr / np.sum(distr, axis = 1).reshape(km.n_clusters, 1)
```

In [154]:
```
def cluster_composition(km, cluster, df = df):
    products = df['Numeric_Product']

    clusters = np.unique(km.labels_)

    distr = np.zeros((km.n_clusters, 4))

    mask = km.labels_ == cluster

    return products[mask]
```

In [155]:
```
def clusters_hist(km, df = df):
    k = km.n_clusters

    all_distr = cluster_distribution(km)

    fig = plt.figure(figsize = (15,10))

    for cluster in range(k):
        ax = fig.add_subplot(2, 3, cluster + 1)
        obs = cluster_composition(km, cluster, df)

        n = len(obs)

        tr_obs = n * product_distr

        decimal = tr_obs - tr_obs.astype(int)

        # we round up the element with the highest decimal part and round down th

        up = decimal == decimal[np.argmax(decimal)]
        decimal[up] = np.ceil(decimal[up])
        decimal[~up] = np.floor(decimal[~up])

        tr_obs = (tr_obs.astype(int) + decimal).astype(int)
        tr_vals = np.hstack([np.ones(tr_obs[i], dtype = int)*i for i in range(4)]


        counts, bins, patches = plt.hist(obs, alpha = 0.5, stacked = True, bins =
        ax.set_xticks(bins)
        ax.set_xticklabels(*list(product_legend))

        plt.hist(tr_vals, alpha = 0.5, bins = 4, color = 'r')
```

In [156]:
```
km_, purity_ = k_means(X, n_clusters = 6)
```

By plotting the distribution of the products for each cluster against the original distribution of the clusters in the entire dataset, we notice that the distributions of the clusters closely resembles the distribution of the products in the dataset.
This shows that the clustering fails to classify the users based on the products.

In [157]:
```
km_, purity_ = k_means(X_ohe, n_clusters = 6)
```

If we try to find the number of clusters by looking at the purity score, we see that the score improves as we increase the clusters and obviously it reaches 1.0 when each data point has its own cluster.

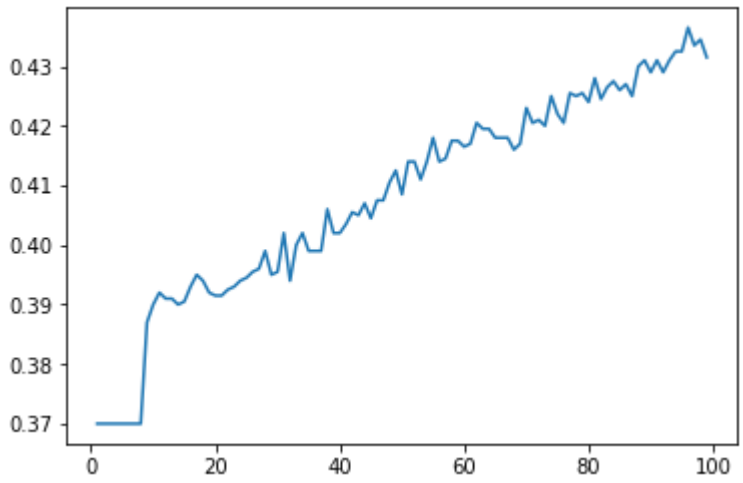
In [158]:
```
scores = []

for i in range(1, 100):

    km, purity_ = k_means(X_ohe, n_clusters = i)
    scores.append(purity_)

plt.plot(np.arange(1, 100), scores)
```

Out[158]:
```
[<matplotlib.lines.Line2D at 0x112951c88>]
```



We apply PCA to our data and we see that we do get an improvement even though it is a very small one

In [159]:
```
km, purity_ = k_means(X_ohe, n_components = 0.99, n_clusters =12)
purity_
```

Out[159]:
```
0.39100000000000001
```

We now try to find a set of weights that improves our results through Simulated Annealing. At the beginning, we got very positive results applying Simulated Annealing on K-Means. However, we tried to use the weights obtained outside Simulated Annealing noting no improvement. Looking for errors, we realized that it was due to a wrong delta cost. This is due to the fact that K-Means initializes the centroids at random every time you run it, thus making the delta cost wrong. To overcome this issue, we need to change the Simulated Annealing code: we do not update the cost through the `delta_cost` but we calculate the cost of the current instance every time.
After fixing the issue, however, we do not get an enhancement in the score.

In [160]:
```
import SimAnn_kmeans
import Weights_Finder_kmeans
```

In [161]:
```
probl = Weights_Finder_kmeans.Weights_Finder(X, df, n_clusters = 17, step = 0.1)

probl, best = SimAnn_kmeans.simann(probl, iters=5, seed=None,
                                   beta0=25.0, beta1=175.0, beta_steps=5,
                                   debug_delta_cost= False)
```

```
T= 0.04
  costs: current= 0.6195 best= 0.6115
  acceptance rate= 1.0
T= 0.016
  costs: current= 0.6215 best= 0.6115
  acceptance rate= 0.8
T= 0.01
  costs: current= 0.6175 best= 0.6115
  acceptance rate= 0.8
T= 0.00727272727273
  costs: current= 0.6215 best= 0.6115
  acceptance rate= 1.0
T= 0.00571428571429
  costs: current= 0.6195 best= 0.6115
  acceptance rate= 1.0
T= 0.0
  costs: current= 0.6165 best= 0.6115
  acceptance rate= 0.6
```

In [162]:
```
w = best.w
```

We now turn to K-Prototypes which should be more appropriate given our data, since it can handle both categorical and numerical variables.

K-Prototypes

In [163]:
```
from kmodes.kprototypes import KPrototypes
```

In [164]:
```
X_kp = df[data_columns].as_matrix()
```

We define a function that returns the indices of the categorical data, which we will turn useful in the application of the K-Prototypes algorithm.

In [165]:
```
def get_cat_indices(df):
    df_cat = df.select_dtypes(include=['object', 'int'])
    return [df.columns.get_loc(c) for c in df_cat.columns]
```

In [166]:
```
categorical_variables = get_cat_indices(df[data_columns])
categorical_variables
```

Out[166]:
```
[3, 4, 7, 8, 11, 26, 27]
```

In [167]:
```
kp = KPrototypes(n_clusters=4)
```

In [168]:
```
labels = kp.fit_predict(X_kp, categorical = categorical_variables)
```

As before, we obtain a score around 0.37. This is due to the fact that 0.37 is the percentage of V-Bag in the entire dataset, which is the same problem that had arisen using K-Means.

In [169]:
```
purity(df, labels)
```

Out[169]:
```
0.37
```

In [170]:
```
df['Product'].value_counts()
```

Out[170]:
```
V-Bag       740
V-Pet       644
V-Auto      450
V-Camera    166
Name: Product, dtype: int64
```

In [171]:
```
740 / 2000
```

Out[171]:
```
0.37
```

DBSCAN

At last, we decide to apply the DBSCAN algorithm to the dataset in order to see if we could get better results with respect to those obtained with the other algorithms we used.

Moreover, the DBSCAN algorithm has the advantage of being a non-parametric model, that is, it does not require the user to pass the desired number of clusters as input.

In [172]:
```
from sklearn.cluster import DBSCAN
```

In [173]:
```
def dbscan_clustering(X, n_components = None, scale = False, df = df, **dbscan_pa
    if scale:
        scaler = StandardScaler()
        X = scaler.fit_transform(X)

    dopca = n_components is None or n_components > 0

    if dopca:
            pca = PCA(n_components= None, copy=True, whiten=False, svd_solver='au
                      tol=0.0, iterated_power='auto', random_state=None)
            X = pca.fit_transform(X)

    db = DBSCAN(**dbscan_params)
    db.fit(X)
    score = purity(df, db.labels_)


    return db, score
```

In order to perform the best possible clustering, we plot the parameters that most influence the output of the algorithm, namely `eps` and `min_samples`, to see which values of those parameters maximize our score. Then we apply the results that can best fit our purpose.

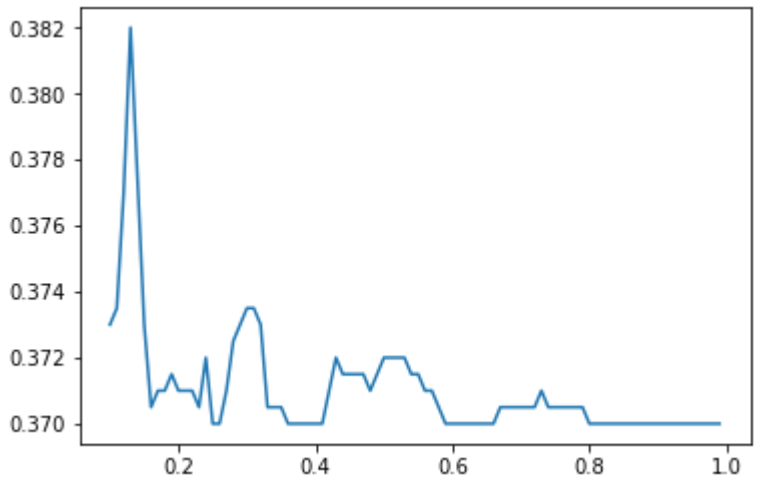
In [174]:
```
scores = []

for i in np.arange(0.1, 1.0, 0.01):

    ap, purity_ = dbscan_clustering(X, eps = i)
    scores.append(purity_)

plt.plot(np.arange(0.1, 1.0, 0.01), scores)
```

Out[174]:
```
[<matplotlib.lines.Line2D at 0x10c044630>]
```



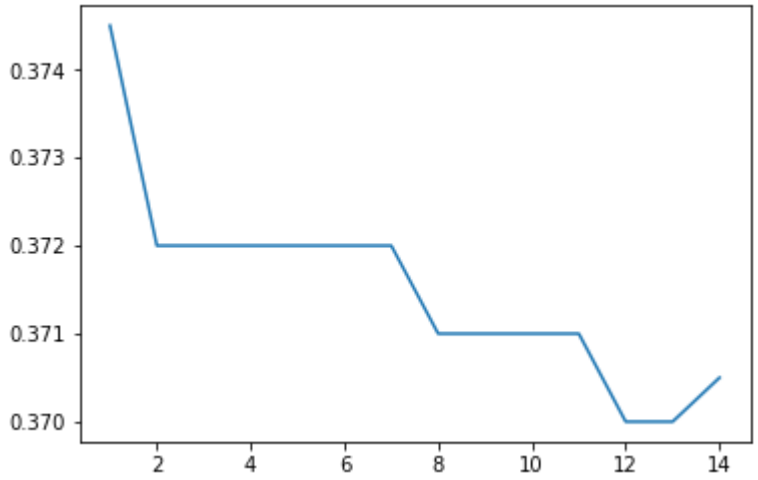
In [175]:
```
scores = []

for i in range (1, 15):

    ap, purity_ = dbscan_clustering(X, min_samples = i)
    scores.append(purity_)

plt.plot(np.arange(1, 15), scores)
```

Out[175]:
```
[<matplotlib.lines.Line2D at 0x1a17901240>]
```



In [176]:
```
db, score = dbscan_clustering(X, n_components=10, eps=0.15, min_samples=4)
n_clusters = len(np.unique(db.labels_))
print('n_clusters:', n_clusters)
print('accuracy:', score)
```

```
n_clusters: 16
accuracy: 0.376
```

Conclusions

After trying different learning algorithms, both supervised (K-Nearest-Neighbors) and unsupervised (K-Means, K-Prototypes and DBSCAN), we found results that never went beyond the 43% (relying on a number of clusters that does not exceed 20). Also, the composition of the clusters does not reflect the one we hoped for.
This could be due to the fact that the data we have is not very informative for profiling users' preferences on the four V by Vodafone products.