Model Development Report

I assume that a given dataset is representative, and my model will be tested on datasets sampled from the same general population.

To build a model I will follow the next steps.

1. First of all, look at the dataset, and conduct Dataset exploration.
2. Make basic Modeling using different linear models.
3. Make Feature selection for linear models:
    A. Variance estimation.
    B. Select K strategy.
    C. Feature relevance analysis.
    D. Feature correlation analysis.
    E. Backward Elimination
    F. RFE strategy
    G. Lasso strategy
    H. Conclusion
4. Make Modeling using ensemble methods.
    A. GradientBoostingRegressor
    B. RandomForestRegressor
    C. Hyperparameters' tuning
5. Construct Final Model with Data Processing Pipeline

Also, following further steps that one could be done to improve the model:

1. Analyze features more deeply, split them into numerical and categorical.
2. More wide model selection.
3. More wide hyperparameters' tuning to improve "Correct predictions" metric (predictions with an absolute error less or equal to 3).

I will use pandas to deal with dataset, scikit-learn as a library of methods, and seaborn and mathplotlib to draw diagrams.

Dataset exploration

1. Here I load the dataset to dataframe, look at it's features. Look at the target variable and it's range. I understand that it is problem for regression.
2. Also I count number of missing data. We take into account that the dataset has many columns with missing values.

In [4]:

```
import pandas as pd
data = pd.read_csv("dataset_00_with_header.csv")

print(data.head())

# See the range of target variable
target_range = sorted(list(set(data.y)))
print("Target variable values:", target_range)
print("Target variable range:",  target_range[-1] - target_range[0])

# See the number of missing values
print(data.isna().sum())
```

```
      x001  x002  x003  x004   x005  x006  x007  x008  x009  x010
...   x296  \
0  1540332   NaN   NaN   NaN    8.0     1     0     1     0     0
...       0
1   823066   4.0   3.0   3.0    4.0     0     2     2     0     0
...    5206
2  1089795   NaN   NaN   NaN   96.0     1     0     0     0     1
...       0
3  1147758  63.0  14.0  38.0  258.0     0     0     0     1     2
...       0
4  1229670  34.0  25.0  29.0   34.0     1     0     0     0     3
...       0

     x297  x298  x299  x300  x301  x302  x303  x304    y
0     NaN     0     0     0     0   NaN     0   NaN  706
1  0.9339     1     1     1     0   NaN     0   NaN  558
2     NaN     0     0     0     0   NaN     0   NaN  577
3     NaN     1     1     1     0   NaN     0   NaN  526
4     NaN     0     0     0     0   NaN     0   NaN  496

[5 rows x 305 columns]
Target variable values: [300, 301, 302, 303, 304, 305, 306, 307, 30
8, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321,
322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 33
5, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348,
349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 36
2, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375,
376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 38
9, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402,
403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 41
6, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429,
430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 44
3, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456,
457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 47
0, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483,
484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 49
7, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510,
511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 52
4, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537,
538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 55
1, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564,
565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 57
8, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591,
592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 60
5, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618,
619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 63
2, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645,
646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 65
9, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672,
673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 68
6, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699,
700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 71
3, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726,
727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 74
0, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753,
754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 76
7, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780,
781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 79
4, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807,
808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 82
1, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834,
```

```
835, 836, 837, 838, 839]
Target variable range: 539
x001        0
x002    21432
x003    21432
x004    21424
x005     6110
x006        0
x007        0
x008        0
x009        0
x010        0
x011        0
x012        0
x013        0
x014        0
x015        0
x016        0
x017        0
x018        0
x019        0
x020        0
x021        0
x022        0
x023        0
x024        0
x025        0
x026        0
x027        0
x028        0
x029        0
x030        0
       ...
x276        0
x277        0
x278        0
x279        0
x280        0
x281        0
x282        0
x283        0
x284        0
x285        0
x286        0
x287    24821
x288    49756
x289    49756
x290    49756
x291        0
x292        0
x293    51133
x294        0
x295    86533
x296        0
x297    58112
x298        0
x299        0
x300        0
x301        0
x302    73069
x303        0
```

```
x304    81875
y           0
Length: 305, dtype: int64
```

Modeling using different linear models

1. I construct basic model using linear models and choose best one to test feature selection in further steps.
2. I consider different ways to deal with missing values. I test Imputer with different strategies, and dropping columns with missing values.
3. I compute Score and RMSE to compare it with further values after feature selection.

In [28]:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV, Ridge, Lass
o
from sklearn.metrics import mean_squared_error
from sklearn.impute import SimpleImputer
from math import sqrt

# Split the dataset into train and test
train, test = train_test_split(data, test_size=0.2)
X = train.drop("y", 1)

# Make Imputer for missing data. Different strategies were tested
# ("mean" was the best based on model Score and RMSE)
#imp = SimpleImputer(strategy='median')
#imp = SimpleImputer(strategy='most_frequent')
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y
reg = LinearRegression()
reg.fit(X, y)

# Calculate quality metrics
print('Train score:', reg.score(X, y))
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
y_actual = test.y
y_predicted = reg.predict(X_test)
rms = sqrt(mean_squared_error(y_actual, y_predicted))
print("RMSE:", rms)
print("Test score:", reg.score(X_test, y_actual))
```

```
Train score: 0.8467565067402127
RMSE: 46.5950792167957
Test score: 0.8457294099656313
```

In [29]:

```
correct_predictions = 0
for y_a, y_p in zip(y_actual, y_predicted):
    if abs(y_a - y_p) <= 3:
        correct_predictions += 1

print("Correct predictions: %.3f %%" % (correct_predictions / len(y_actual) * 10
0))
```

```
Correct predictions: 5.710 %
```

Test for dropping columns with any missing values. The result is significantly worse:

In [33]:

```
train, test = train_test_split(data.dropna(how="any", axis=1), test_size=0.2)
print("After dropping missed, features count:", train.shape[1])
X = train.drop("y", 1)

y = train.y
reg = LinearRegression()
reg.fit(X, y)

# Calculate quality metrics
print('Train score:', reg.score(X, y))
X_test = test.drop("y", 1)
y_actual = test.y
y_predicted = reg.predict(X_test)
rms = sqrt(mean_squared_error(y_actual, y_predicted))
print("RMSE:", rms)
print("Test score:", reg.score(X_test, y_actual))
```

```
After dropping missed, features count: 264
Train score: 0.7904935010717437
RMSE: 54.947608335121735
Test score: 0.7820734782416499
```

Iterate over several linear models and compare them. Especially is is interesting to test Cross-Validated versions.

In [30]:

```
models = {
    "LinearRegression(normalize=True)": LinearRegression(normalize=True),
    "Lasso()": Lasso(),
    "Ridge()": Ridge(),
    "LassoCV(cv=5)": LassoCV(cv=5),
    "RidgeCV(cv=5)": RidgeCV(cv=5),
}

train, test = train_test_split(data, test_size=0.2)
X = train.drop("y", 1)
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y

for name, model in models.items():

    reg = model
    reg.fit(X, y)

    # Calculate quality metrics
    print("Model: ", name)
    print("  Train score:", reg.score(X, y))
    X_test = test.drop("y", 1)
    X_test = imp.transform(X_test)
    y_actual = test.y
    y_predicted = reg.predict(X_test)
    rms = sqrt(mean_squared_error(y_actual, y_predicted))
    print("  RMSE:", rms)
    print("  Test score:", reg.score(X_test, y_actual))
```

```
Model:  LinearRegression(normalize=True)
  Train score: 0.846153159907327
  RMSE: 46.251156103880874
  Test score: 0.8483073158071576

Model:  Lasso()
  Train score: 0.825710573002647
  RMSE: 49.307722438912016
  Test score: 0.8275952069400246
Model:  Ridge()
  Train score: 0.8461530722943613
  RMSE: 46.251429977366904
  Test score: 0.8483055193233191

Model:  LassoCV()
  Train score: 0.4502391185201766
  RMSE: 87.9712146822853
  Test score: 0.45121702763717375

Model:  RidgeCV()
  Train score: 0.8461480921585185
  RMSE: 46.25388272817376
  Test score: 0.8482894299342727
```

After these calculations, I decided to choose basic LinearRegression for further feature selection procedures. I move it to function.

In [7]:

```
def estimate_model(X, y, X_test, y_actual, model=LinearRegression, **params):
    reg = model(**params)
    reg.fit(X, y)
    print('Train score:', reg.score(X, y))
    y_predicted = reg.predict(X_test)
    rms = sqrt(mean_squared_error(y_actual, y_predicted))
    print("RMSE:", rms)
    print("Test score:", reg.score(X_test, y_actual))
```

Feature selection for linear models

I make feature selection.

Variance estimation

First of all, I do Variance estimation. Value 0.95 is came from experiments with Random Search and Grid Search strategies. For the sake of brevity, I do not describe here full Search code for this problem (more illustrative example will be described in further calculations)

In [108]:

```
# Variance fiiltering
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=(.95 * (1 - .95)))

train, test = train_test_split(data, test_size=0.2)
X = train.drop("y", 1)

imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
X = sel.fit_transform(X)
y = train.y
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
X_test = sel.transform(X_test)
y_actual = test.y
estimate_model(X, y, X_test, y_actual)
```

```
Train score: 0.8432715355351239
RMSE: 48.2329842769225
Test score: 0.8353551776488938
```

Select K strategy

Different values was tested, here I show example for K=50 (Also from Grid Search). The results were bad.

In [70]:

```
# select k strategy
from sklearn.feature_selection import mutual_info_regression as mir
from sklearn.feature_selection import SelectKBest

sel = SelectKBest(mir, k=50)
train, test = train_test_split(data[:30000], test_size=0.2)
X = train.drop("y", 1)
y = train.y
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
X = sel.fit_transform(X, y)
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
X_test = sel.transform(X_test)
y_actual = test.y
estimate_model(X, y, X_test, y_actual)
```

```
Train score: 0.7723087221296201
RMSE: 55.78032159061759
Test score: 0.7836384289738724
```

Feature relevance analysis

I try to find most relevant features.

In [31]:

```
cor = data.corr()
cor_target = abs(cor["y"])

#Select highly relevant features
relevant_features = cor_target[cor_target > 0.2]
```

In [32]:

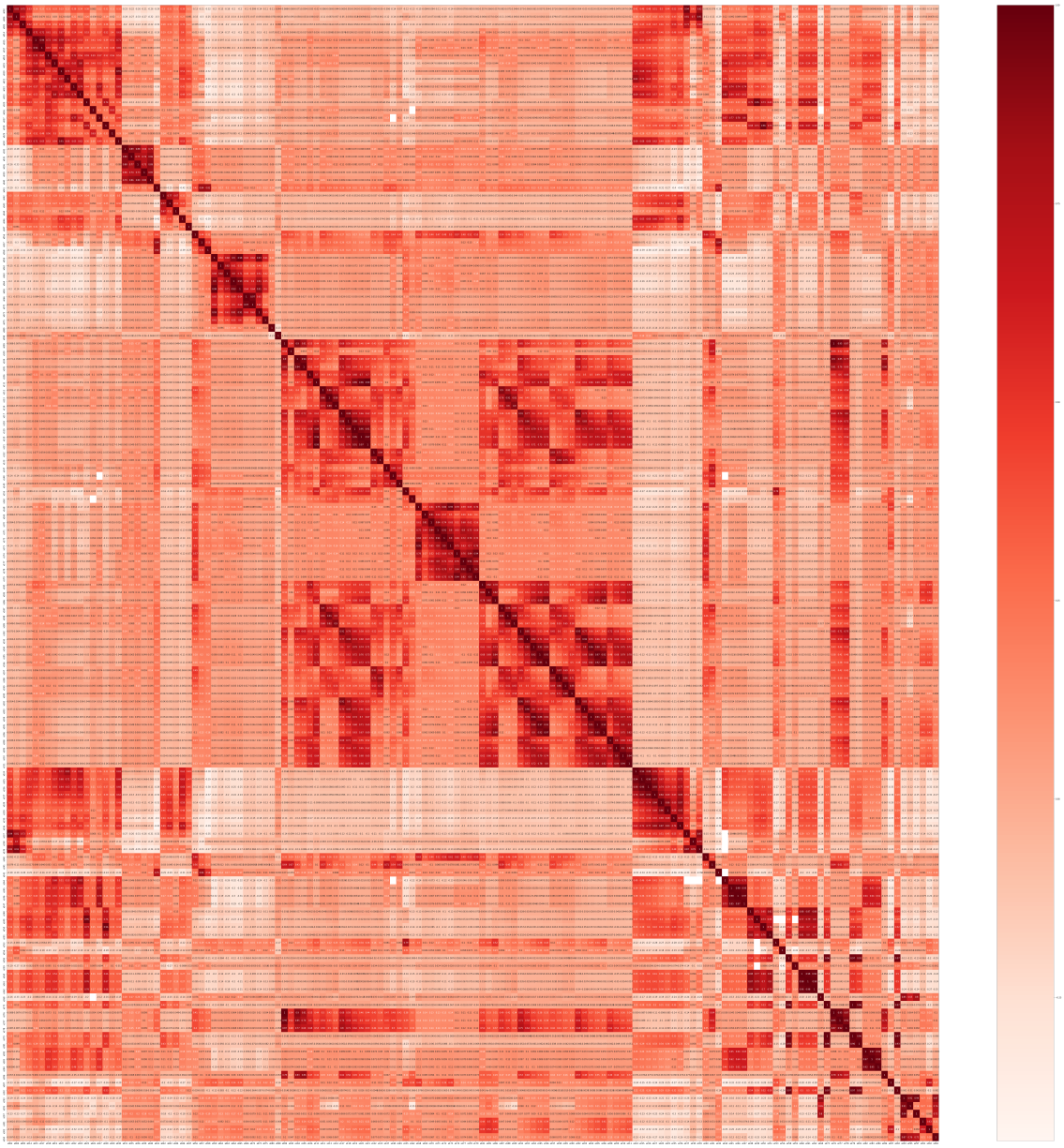
```
data_selected = data[list(relevant_features.index)]
train, test = train_test_split(data_selected, test_size=0.2)
X = train.drop("y", 1)
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
y_actual = test.y
estimate_model(X, y, X_test, y_actual)
```

```
Train score: 0.833392850252221
RMSE: 48.3941929953706
Test score: 0.8338045864769359
```

Plotting feature correlation diagram.

In [49]:

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

plt.figure(figsize=(100,100))
X_selected = data_selected.drop("y", 1)
cor = X_selected.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.show()
```



Feature correlation analysis

To try to improve model quality, find most correlated features. Drop it and estimate a model.

In [50]:

```
import numpy as np
cor = cor.abs()

upper = cor.where(np.triu(np.ones(cor.shape), k=1).astype(np.bool))
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
```

In [51]:

```
data_selected_filtered = data_selected.drop(to_drop, axis=1)
data_selected_filtered.shape
```

Out[51]:

```
(100000, 122)
```

In [71]:

```
train, test = train_test_split(data_selected_filtered, test_size=0.2)
X = train.drop("y", 1)
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
y_actual = test.y
estimate_model(X, y, X_test, y_actual)
```

```
Train score: 0.8454706660849636
RMSE: 46.44417722521158
Test score: 0.8457742277337201
```

Backward Elimination

In [55]:

```
import statsmodels.api as sm

#I will try Backward Elimination on reduced dataset (to save a time)
X = data[:20000].drop("y", 1)
y = data.y[:20000]
cols = list(X.columns)

# For convenience - use Imputing analogue from Pandas
X.fillna(X.mean(), inplace=True)
pmax = 1
while len(cols) > 0:
    p = []
    X_1 = X[cols]
    X_1 = sm.add_constant(X_1)
    model = sm.OLS(y, X_1).fit()
    p = pd.Series(model.pvalues.values[1:],index = cols)
    pmax = max(p)
    feature_with_p_max = p.idxmax()
    if(pmax > 0.05):
        cols.remove(feature_with_p_max)
    else:
        break

selected_features_BE = cols
```

In [57]:

```
data_selected_filtered = data[selected_features_BE + ['y']]
data_selected_filtered.shape
```

Out[57]:

```
(100000, 128)
```

Build and test model:

In [72]:

```
train, test = train_test_split(data_selected_filtered, test_size=0.2)
X = train.drop("y", 1)
imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
y_actual = test.y

estimate_model(X, y, X_test, y_actual)
```

```
Train score: 0.8451513803492834
RMSE: 46.43233963639462
Test score: 0.8471812098239396
```

RFE strategy

RFE strategy to choose optimal number of features (Recursive feature elimination)

In [102]:

```
from sklearn.feature_selection import RFE

nof_list = np.arange(304)
high_score = 0
nof = 1
score_list = []
LIMIT = 10000

# Use a smaller dataset - to save a time
# NOTE:
# I use slice here just for the sake of simplisity.
# Of course, it will be better to choose random subset using .sample() (not slic
e).
# Or, shuffle the initial dataset and slice after that.
X = data[:LIMIT].drop("y", 1)
X.fillna(X.mean(), inplace=True)
y = data[:LIMIT].y

# I will test different ranges of n. Here is one of them.
for n in range(215, 217):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ran
dom_state=1)
    model = LinearRegression()
    rfecv = RFE(model, nof_list[n])
    fitted = rfe.fit(X_train, y_train)
    X_train_rfe = rfe.fit_transform(X_train, y_train)
    X_test_rfe = rfe.transform(X_test)

    print(X_test.shape, X_train.shape)
    print(X_test_rfe.shape, X_train_rfe.shape)

    model.fit(X_train_rfe, y_train)
    score = model.score(X_test_rfe, y_test)
    score_list.append(score)
    if(score > high_score):
        high_score = score
        nof = nof_list[n]

print("Optimum number of features: %d" % nof)
print("Score of model with %d features: %f" % (nof, high_score))
```

```
(3000, 304) (7000, 304)
(3000, 157) (7000, 157)
(3000, 304) (7000, 304)
(3000, 157) (7000, 157)
Optimum number of features: 215
Score of model with 215 features: 0.804221
```

Check the model, and see features:

In [35]:

```
train, test = train_test_split(data, test_size=0.2)
X = train.drop("y", 1)
X.fillna(X.mean(), inplace=True)
y = train.y
cols = list(X.columns)
rfe = RFE(model, nof)
X_rfe = rfe.fit_transform(X, y)
X_test = test.drop("y", 1)
X_test.fillna(X.mean(), inplace=True)
X_test_rfe = rfe.transform(X_test)
y_actual = test.y

estimate_model(X_rfe, y, X_test_rfe, y_actual)

temp = pd.Series(rfe.support_,index = cols)
selected_features_rfe = temp[temp==True].index
print(selected_features_rfe)
```

```
Train score: 0.8264593497930401
RMSE: 49.15052334390021
Test score: 0.8286571592489188
Index(['x006', 'x007', 'x008', 'x010', 'x011', 'x013', 'x014', 'x01
5', 'x016',
       'x017',
       ...
       'x287', 'x293', 'x295', 'x297', 'x298', 'x299', 'x300', 'x30
1', 'x302',
       'x304'],
      dtype='object', length=215)
```

Lasso strategy

Select features and see the result

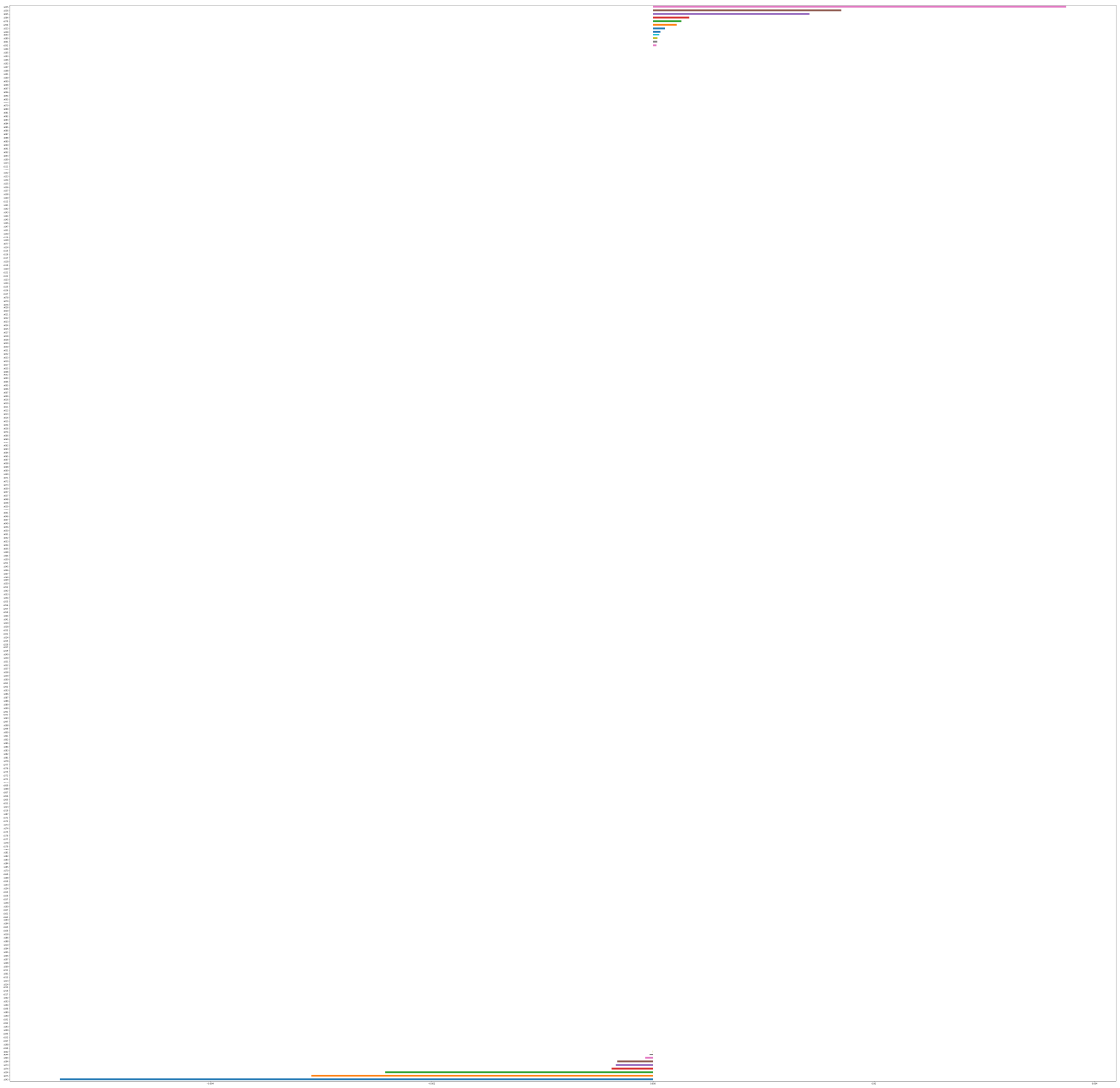
In [20]:

```
reg = LassoCV()
reg.fit(X, y)
print("Best alpha using built-in LassoCV: %f" % reg.alpha_)
print("Best score using built-in LassoCV: %f" % reg.score(X, y))
coef = pd.Series(reg.coef_, index = X.columns)

print("Lasso picked " + str(sum(coef != 0)) + " variables and eliminated the oth
er " \
      + str(sum(coef == 0)) + " variables")

imp_coef = coef.sort_values()
matplotlib.rcParams['figure.figsize'] = (100, 100)
imp_coef.plot(kind = "barh")
```

```
Best alpha using built-in LassoCV: 5481.851816
Best score using built-in LassoCV: 0.450751
Lasso picked 21 variables and eliminated the other 283 variables
```

Out[20]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa78ef1c7f0>
```



Test of these 21 variables do not lead to better result.

Conclusion

Feature selection strategies do not lead to significantly better results in comparison with basic model.

Now I have a basic model to compare it with more complex methods below.

Modeling using ensemble methods

GradientBoostingRegressor

GBR was trying, but it's quality is significantly worse than the method below - RandomForesRegressor.

In [74]:

```
from sklearn.ensemble import GradientBoostingRegressor

train, test = train_test_split(data, test_size=0.2)
X = train.drop("y", 1)

imp = SimpleImputer(strategy='mean')
X = imp.fit_transform(X)
y = train.y
X_test = test.drop("y", 1)
X_test = imp.transform(X_test)
y_actual = test.y
estimate_model(X, y, X_test, y_actual, model=GradientBoostingRegressor)
```

```
Train score: 0.9286998742239324
RMSE: 31.719624409375143
Test score: 0.9283548235811027
```

RandomForestRegressor

Test RandomForestRegressor with full feature set and reduced one (obtained abobe with a help of RFE). I do not care about system warnings - it's just rough approaches.

In [36]:

```
from sklearn.ensemble import RandomForestRegressor
estimate_model(X, y, X_test, y_actual, model=RandomForestRegressor, n_estimators
=10)
estimate_model(X_rfe, y, X_test_rfe, y_actual, model=RandomForestRegressor, n_es
timators=10)
```

```
Train score: 0.9879394125316882
RMSE: 30.10595866269666
Test score: 0.9357142682154765
Train score: 0.9870950821996467
RMSE: 30.960200403711955
Test score: 0.9320143590394989
```

I will find best parameters of RFR with a help of search strategies. Here is a just approximation example: I decided to increase the number of estimators in RFE. And model shows better quality.

In [19]:

```
estimate_model(X, y, X_test, y_actual, model=RandomForestRegressor, n_estimators
=120)
```

```
Train score: 0.9920168604638694
RMSE: 28.78914989826173
Test score: 0.9406975595286273
```

Hyperparameters tuning

I will use the following strategy:

1. Random search for looking for good model hyperparameters.
2. Grid search based on the previous result.

Let's do random search:

In [6]:

```
from sklearn.model_selection import RandomizedSearchCV
import numpy as np
# For dict printing:
from pprint import pprint

rf = RandomForestRegressor(random_state=42)
print('Parameters currently in use:\n')
pprint(rf.get_params())

n_estimators = [int(x) for x in np.linspace(start=600, stop=1800, num=3)]
max_features = ['auto', 'sqrt']
max_depth = [int(x) for x in np.linspace(10, 200, num=5)]
max_depth.append(None)
min_samples_split = [2, 5, 10]
min_samples_leaf = [1, 2, 4]
bootstrap = [True, False]

random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

rf = RandomForestRegressor()
rf_random = RandomizedSearchCV(estimator=rf, param_distributions=random_grid, n_
iter=100,
                               cv=3, verbose=2, random_state=42, n_jobs=-1)

rf_random.fit(X, y)
```

```
Parameters currently in use:

{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 'warn',
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
Fitting 3 folds for each of 10 candidates, totalling 30 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:  2.3min finis
hed
```

Out[6]:

```
RandomizedSearchCV(cv=3, error_score='raise-deprecating',
          estimator=RandomForestRegressor(bootstrap=True, criterion
='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators='warn', n_job
s=None,
           oob_score=False, random_state=None, verbose=0, warm_star
t=False),
          fit_params=None, iid='warn', n_iter=10, n_jobs=-1,
          param_distributions={'n_estimators': [600, 1200, 1800],
'max_features': ['auto', 'sqrt'], 'max_depth': [10, 57, 105, 152, 2
00, None], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1,
2, 4], 'bootstrap': [True, False]},
          pre_dispatch='2*n_jobs', random_state=42, refit=True,
          return_train_score='warn', scoring=None, verbose=2)
```

In [7]:

```
best_random_params = rf_random.best_params_
print(best_random_params)
```

```
{'n_estimators': 1800, 'min_samples_split': 5, 'min_samples_leaf':
2, 'max_features': 'sqrt', 'max_depth': None, 'bootstrap': False}
```

Make one more, verbose function for results evaluation. Compare with basic model.

In [37]:

```
def evaluate(model, test_features, test_labels):
    predictions = model.predict(test_features)
    errors = abs(predictions - test_labels)
    mape = 100 * np.mean(errors / test_labels)
    accuracy = 100 - mape
    print('Model Performance:')
    print('  Average Error: %.3f' % np.mean(errors))
    print('  Accuracy: %.3f' % accuracy)
    return accuracy
```

In [ ]:

```
base_model = RandomForestRegressor(n_estimators=120, random_state=42)
base_model.fit(X, y)
y_test = y_actual
base_accuracy = evaluate(base_model, X_test, y_test)
best_random = RandomForestRegressor(**best_random_params)
best_random.fit(X, y)
random_accuracy = evaluate(best_random, X_test, y_test)

print('Improvement of %.3f %%' % (100 * (random_accuracy - base_accuracy) / base
_accuracy))
```

```
Model Performance:
  Average Error: 19.526
  Accuracy: 96.645
```

After that, make Grid search.

In [9]:

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'bootstrap': [False],
    'max_depth': [None],
    'max_features': ['sqrt'],
    'min_samples_leaf': [1, 2, 3],
    'min_samples_split': [4, 5],
    'n_estimators': [1800]
}

rf = RandomForestRegressor()
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, pre_dispatch=2,
                           cv=3, n_jobs=-1, verbose=2)
```

In [ ]:

```
# Fit the grid search to the data
grid_search.fit(X, y)
print(grid_search.best_params_)

best_grid = grid_search.best_estimator_
grid_accuracy = evaluate(best_grid, X_test, y_test)

print('Improvement of %.3f %%' % (100 * (grid_accuracy - base_accuracy) / base_a
ccuracy))
```

Final Model with Data Processing Pipeline

I chose RFR regressor with optimized hyperparameters.

In [16]:

```
from sklearn.pipeline import Pipeline
from time import time

best_grid_params = grid_search.best_params_

imputer = SimpleImputer(strategy='mean')
regressor = RandomForestRegressor(**best_grid_params, n_jobs=-1)

final_model = Pipeline([('imputing', imputer),
                        ('regression', regressor)])

t = time()
final_model.fit(X, y)
print("Overall fitting time: %.3f sec." % (time() - t))
```

```
Overall fitting time: 910.699 sec.
```

It is significanlty better than the basic model (LinearRegression) and provides following quality metrics:

In [27]:

```
print('Train score:', final_model.score(X, y))
y_predicted = final_model.predict(X_test)
rms = sqrt(mean_squared_error(y_actual, y_predicted))
print("RMSE:", rms)
print("Test score:", final_model.score(X_test, y_actual))

correct_predictions = 0
for y_a, y_p in zip(y_actual, y_predicted):
    if abs(y_a - y_p) <= 3:
        correct_predictions += 1

print("Correct predictions: %.3f %%" % (correct_predictions / len(y_actual) * 10
0))
```

```
Train score: 0.9556876171021234
RMSE: 24.743776056051956
Test score: 0.9566348892753525
Correct predictions: 27.220 %
```