Feature Selection

<div id="author""> David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Feature selection is not always necessary for building machine learning models, but it is typically a worthwhile process to pursue. The goal is *parsimony*, to reduce the number of predictor variables (features) in the model, to keep predictive accuracy at or almost at the same level, but with a much simpler model, with fewer predictor variables.

Two reasons to pursue feature selection:

1. Data costs money. The fewer the predictors, the less data needs to be collected.
2. Understanding the underlying relationships between predictors and target variable, which indirectly often leads to the construction of better models.

Preliminaries

In [1]:
```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2023-07-17 at 16:37
```

In [2]:
```
import os
os.getcwd()
```

Out[2]:
```
'/Users/davidgerbing/Documents/000/575/0Templates'
```

In [3]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

Data

In [4]:
```
#d = pd.read_csv('data/Boston.csv')
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

In [5]:
```
d.shape
```

Out[5]:
```
(506, 15)
```

In [6]:
```
d.head()
```

Out[6]:

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	bl
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	39(
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	39(
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	39:
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	39.
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	39(

Do not need the first column, so drop.

In [7]:
```
d = d.drop(["Unnamed: 0"], axis="columns")
d.head()
```

Out[7]:

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33

Store the features, the predictor variables, in data structure X. Store the target variable in data structure y. To run multiple regression with all possible predictor variables, one possibility defines X as the entire data frame with *medv* dropped, as in

X = d.drop(['medv'], axis="columns")

Alternatively, use the procedure below that manually defines a vector of the predictor variables (features) names, and then define X as the subset of *d* that contains just these variables.

In [8]:
```
y = d['medv']
pred_vars = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad'
             'tax', 'ptratio', 'black', 'lstat']
```

```
X = d[pred_vars]
```

Not necessary, but see how many features in the model, and observe the data type of the X and y data structures. The function `len()` provides the length of a vector, that is, the number of elements of a vector.

In [9]:
```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 13
```

X and y are created as two different pandas data types: X is a data frame, y is a one-dimensional array called a `series`. A data frame can have a single column, but, somewhat confusingly (in my opinion), subsetting a data frame down to a single variable is no longer is a data frame.

Feature Selection

Features, the predictor variables, should be ...

- *relevant*: Predictors each correlate with the target
- *unique*: Predictors do not correlate much with each other

The problem of *collinearity* is the problem of correlated predictor variables, the features. Too much correlation and redundancy make estimating the slope coefficients difficult, though it does not harm predictive accuracy per se. Generally, improve model fit by adding new information in the form of a new predictor variable to the model to the extent that the new predictor is relevant and unique.

Do, however, be aware of the problem of *data leakage*. When testing the model on data previously unseen by the model, all aspects of that data must have been unseen, just as in a real-world forecasting scenario. Otherwise, the data is said to leak from training to testing data. Making decisions regarding the model based on *all* the data then by definition includes both training and testing data. Best to make decisions regarding

model estimation only from the training data.

Manual Selection

Base selection of the predictor variables on satisfying the two criterion: relevance and uniqueness. The goal here is to produce a single output, a table, that displays numerical indices for both criterion.

Uniqueness. Besides the correlation coefficient of two predictor variables, a more general indicator of collinearity is the *variance inflation factor* or *VIF*. The *VIF* assesses the linear redundancy of one predictor variable not just with one other predictor variable, but all the other predictor variables.

Relevance. Compute the correlation of each predictor with the target.

In [10]:
```
print("X is a: ", type(X))
print("X.values is a: ", type(X.values))
```

```
X is a:  <class 'pandas.core.frame.DataFrame'>
X.values is a:  <class 'numpy.ndarray'>
```

Use the `statsmodels` function `variance_inflation_factor()` to compute the variance inflation factor for each predictor. The VIF's are a property only of the X's, so the target y is not part of this analysis. The `variance_inflation_factor()` function does not compute all the VIF's, but only one at a time. Create a data frame named *vif*, then fill each row of the data frame with the corresponding name of the predictor variable and its corresponding variance inflation factor.

To systematically calculate and retrieve the VIF's, one for each feature, traverse through the variables in X one at a time with a programming structure known as a `for` loop, from the first X variable through the last X variable, where `X.shape[1]` is the number of rows of the data frame.

Because the loop cannot traverse through the original data frame, transfer the X data frame to a more primitive data structure, a `numpy` structure of a numeric matrix, obtained with the `values` method.

1. To begin, create an empty data frame with any valid name. Here we use *vif*. Then define a variable called *Predictor* in the data frame, filled with the names of the columns of the X data structure using the `columns` method.

2. Then create a variable called *VIF*, the variance inflation factor for each predictor variable. Loop through the data matrix (not data frame) with the `values` method

for each predictor variable.

3. Calculate the correlation of each predictor (feature) with the target and store in the variable called *Relevance*. Store in the data series *cr*, then loop through *cr* for each variable to copy the value to the new *Relevance* variable.

4. Finally, display the contents of the created *vif* data frame by listing its name as the last line of code in the cell. (If we wish to display information before the last line, then need the `print()` function.)

In [11]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = pd.DataFrame()
vif['Predictor'] = X.columns

vif['VIF'] = [variance_inflation_factor(X.values, i)
              for i in range(X.shape[1])]

cr = d.corr()['medv'].round(3)
vif['Relevance'] = [cr[i]
              for i in range(X.shape[1])]
vif
```

Out[11]:

	Predictor	VIF	Relevance
0	crim	2.100373	-0.388
1	zn	2.844013	0.360
2	indus	14.485758	-0.484
3	chas	1.152952	0.175
4	nox	73.894947	-0.427
5	rm	77.948283	0.695
6	age	21.386850	-0.377
7	dis	14.699652	0.250
8	rad	15.167725	-0.382
9	tax	61.227274	-0.469
10	ptratio	85.029547	-0.508
11	black	20.104943	0.333
12	lstat	11.102025	-0.738

There is much collinearity in the data, consistent with the correlation matrix that shows many feature correlations far from 0. Many features could be deleted to yield a more parsimonious model that would be just as effective if not more so. Although *rm* has one of the highest VIF's, it is also strongly related to the target as shown by the regression coefficients' analysis and has one of the highest correlations with the target. A high VIF does not mean a feature should be deleted because perhaps a relevant feature is correlated with other, less relevant features that, when deleted, lower the VIF on the

more relevant feature.

Automated Feature Selection

The pure machine learning approach seeks to automate everything. This approach makes the most sense when there are many, tens if not hundreds, of features. Otherwise, best to perform feature selection manually, analyzing correlations, variance inflation factors, p-values from the regression analysis of all features, and all possible subset regressions. And there is always understanding the meaning of the individual features (predictor variables), favoring the most understandable and meaningful, and perhaps easiest or cheapest for which to collect the data.

Let's proceed as if we have too many features to model effectively or we wish to rely only on influential predictor variables. So we pare down our model here using automated feature selection. We begin with all 13 features.

If you have the computation time, do this after the analysis with all the features. If computation time is limited, do at least some feature selection before the model evaluation.

Automated Univariate Feature Selection

There is one simple `sklearn` feature selection module called `SelectKBest` that selects the specified number of features according to relevance, the correlation of each feature with the target. It simply selects those features with the highest correlations with the target. Specify the number of retained features with the `k` parameter.

Here the logical array we name `selected` indicates which of the `k` values in the X feature data structure are to be retained.

In [12]:
```
from sklearn.feature_selection import SelectKBest, f_regression
selector = SelectKBest(f_regression, k=5).fit(X,y)
selected = selector.get_support()
selected
```

Out[12]:
```
array([False, False,  True, False, False,  True, False, False, False,
        True,  True, False,  True])
```

Select the selected variables by subsetting the original X data structure

Select the selected variables by subsetting the original X data structure.

In [13]:
```
X2 = X.iloc[:, selected]
X2.head()
```

Out[13]:

	indus	rm	tax	ptratio	lstat
0	2.31	6.575	296	15.3	4.98
1	7.07	6.421	242	17.8	9.14
2	7.07	7.185	242	17.8	4.03
3	2.18	6.998	222	18.7	2.94
4	2.18	7.147	222	18.7	5.33

Automated Multivariate Feature Selection

A more sophisticated, though more costly in CPU time procedure, is the `sklearn` module `RFE`, for *recursive feature elimination*. First, specify the estimation procedure by which to initially assign weights to the features, such as linear regression as in this example. The `RFE` procedure then evaluates the features and identifies the single weakest feature on the basis of model fit, which is then pruned from the model. This assumes the parameter *step* is set at 1, which is the number of features pruned at each step.

To apply the estimator, invoke the `fit()` function on the specified feature and target data structures, X and y. The process is recursively repeated until the requested number of features, *n_features_to_select*, is obtained. In this example, retain the top 5 features.

This method generally produces a better model than `SelectKBest`, but the issue is computation time. If the CPU time is available, `RFE` is preferred.

In [14]:
```
from sklearn.linear_model import LinearRegression
estimator = LinearRegression()
from sklearn.feature_selection import RFE
selector = RFE(estimator, n_features_to_select=5, step=1).fit(X,y)
```

The features are selected, but now pare down the X data frame of feature data to just include the selected features. Rely upon two variables that `RFE()` created. The output vector *support_* indicates by `True` or `False` the selected variables. The output *ranking_* vector ranks the features, with all the selected variables ranked at 1.

In [15]:
```
print(selector.support_)
print(selector.ranking_)
```

```
[False False False  True  True  True False  True False False  True False
```

```
 False]
[4 6 5 1 1 1 9 1 3 7 1 8 2]
```

Use the *support_* output structure from `RFE()`. Subset the data with `iloc()` to redefine the feature data frame.

In [16]:
```
X2 = X.iloc[:, selector.support_]
X2.head()
```

Out[16]:

	chas	nox	rm	dis	ptratio
0	0	0.538	6.575	4.0900	15.3
1	0	0.469	6.421	4.9671	17.8
2	0	0.469	7.185	4.9671	17.8
3	0	0.458	6.998	6.0622	18.7
4	0	0.458	7.147	6.0622	18.7

We see that the five feature variables selected by the more sophisticated `RFE()` differ from the five chosen features by `SelectKBest()`.

To view the rankings of all the features, to show the order of the variables that did not make the final 5, access the output *ranking_* variable. Note that one of the two features most highly correlated with the target, *lstat*, did not make the cut.

The crucial information not shown here is how much higher is R^2, or how much lower is MSE, for a five-feature model. No answer from this analysis. To test, the model would need to be re-run.

In [17]:
```
rnk = pd.DataFrame()
rnk['Feature'] = X.columns
rnk['Rank']= selector.ranking_
rnk.sort_values('Rank').transpose()
```

Out[17]:

	3	4	5	7	10	12	8	0	2	1	9	11	6
Feature	chas	nox	rm	dis	ptratio	lstat	rad	crim	indus	zn	tax	black	age
Rank	1	1	1	1	1	2	3	4	5	6	7	8	9

Postscript

The model should also be analyzed with standardized variables to put everything on a common scale. Further, at least one outlier should be removed. Given the high degree of collinearity, the model can likely be reduced to about 3 or 4 features with little if any lose in predictive power.

Also, the model should be developed on one set of data, the training data, and then evaluated on testing data, apart from the training data. If no new data is available, then split the original data up into 75%/25% samples and then estimate (learn) on the 75% sample and test on the 25% sample.

The most useful statistical information, from my experience, for feature selection is what is called *all subset regression*, which evaluates R^2 for all (or many) possible subsets of feature combinations (actually, the adjusted version). Then it becomes straightforward to see which core set of predictors are best combined to achieve one of the best models among the available alternatives.

On a bit of a tangent here, but in terms of the most general advice, my R `Regression()` function provides this subset regression analysis automatically. I prefer that program in my `lessR` R package to anything I have seen in Python when doing regression analysis. Very straightforward to use and does cross-validation as well with the parameter `kfold` set to some value larger than 1 to specify the number of folds. Read the data and run the function. Not part of this course per se, but helpful to apply in real-world contexts. Here is the R code that gives all of the above, plus all subsets regressions.

```
library(lessR)
```

```
d = Read("http://web.pdx.edu/~gerbing/data/Boston.csv")
```

```
Regression(medv ~ crim + zn + indus + chas + nox + rm + age + dis +
rad + tax + ptratio + black + lstat)
```

As a bonus, add the parameter `Rmd="house"` (or named whatever), and you will generate a complete written report.