Partial Least Square Regression

A Discussion, Implementation and Testing
Final Project of STA663

Juncheng Dong Xiaoqiao Xing

Spring 2020

Abstrct

This paper is about a follow-up research on PARTIAL LEAST-SQUARES REGRESSION: A TUTORIAL[1] and An EXAMPLE OF 2-BLOCK PREDICTIVE PARTIAL LEAST-SQUARES REGRESSION WITH SIMULATED DATA[2]. In first and second section, we discuss about background of linear regression and its solution. Then in third section, we identify the problem of ordinary solution to linear regression and why we need partial least squares(PLS) regression followed by details of principal component analysis(PCA) which is closely related to PLS and details of PLS itself. We break down both steps and equations and conceptual understanding of algorithms to provide a deeper understanding of the algorithms instead of simply remembering the steps to perform PLS. Then we implement PLS in python and test our program with randomly generated data as well as real-life data (an automobile data set). In the end, We conduct a comparative analysis between PLS and its competing algorithms.

1 Background

Linear regression is one of the most frequently used models in data analysis as well as regression problems. For each observation I, we have m observed properties $x_i (i = 1...m)$. These properties can be considered as a vector $\mathbf{x}$, and each observation has an associated dependent variable y that is correlated with its properties, hence the properties $\mathbf{x}$ are often called as independent variables while y is called as dependent variable.

Linear regression is often the first model students learnt in data analysis and machine learning classes because of its simple form. If we assume there exists a linear relation between dependent variable and independent variables, then the model will look like

$$y = \beta_0 + \beta_1 * x_1 + ... + \beta_m * x_m$$

If we have n observation, then we can write observations' properties as matrix $\mathbf{X}$ where each row represents one observation and each column represents one property of n observations. Hence shape of $\mathbf{X}$ is n by m. We can write observations' dependent variable as vector $\mathbf{y}$. Hence the size of $\mathbf{y}$ is n by 1. Our goal is to find a vector $\boldsymbol{\beta}$ such that

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$$

2 Ordinary Least Square

$\boldsymbol{\beta}$ is the solution to linear system $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$. In order to find such $\boldsymbol{\beta}$, we need to first look at the structure of it. .

Structure of $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$

As can simply proved by linear algebra

1. If $m > n$, when number of variables is larger than number of observations, infinite solutions exist
2. if $m = n$, when number of variables is the same as number of observations and the properties are independent to each other, then a unique solution exists.
3. if $m < n$, when number of variables is less than number of observations, there does not exist solution

In most of applications and scenarios, we have more observations than variables. However, as listed above, when m<n, there exist no solution. But we can find $\boldsymbol{\beta}$ such that $\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|$. This is the well-known ordinary least square (OLS). The solution to OLS has a nice, clean and closed form.

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

3 Problems with OLS

When $\mathbf{X}$ is not full-rank , then $(\mathbf{X}^T\mathbf{X})^{-1}$ does not exist, OLS solution is not feasible anymore. To be specific, $\mathbf{X}$ is not full-rank means one of the two following situations

Problems to be solved

1. some independent variables are correlated with each other
2. number of variables exceeds number of observations.

Conceptually it is easy to understand why these two situations will cause problem. Highly correlated variables means the information we can have from the data is duplicated thus not enough to inference for $\boldsymbol{\beta}$. The same problem exists when we do not have enough observations.

Insufficient amount of data and correlated independent variables are not uncommon in many research areas where the observations are difficult to collect and datas are closely related. One such area is chemical analysis[2]. In order to overcome these problems, some solutions have been proposed. The most naive ways is to collect more observations or deleting correlated columns. However, these naive solutions may not be able to perform easily in real datasets. In real life, collecting more data can be very difficult in many areas, such as rare cases in medical research. It may appear that deleting correlated columns is a easier way – we can find independent columns of $\mathbf{X}$ by finding its pivot columns, which is a the standard linear algebra method. However, many possible independent column combination may exist and we cannot know which column to delete because we do not

know which column is MORE related to dependent variable. Fortunately, there are couple algorithms that can solve this problem, and Partial least square regression (PLS) is one of them. The other famous ones include principal component regression (PCR), which we will discuss in next section, and ridge regression, which has very different concepts to PLS and PCR but can also solves this problem.

4 Principal Component Analysis/PCR

One problem with deleting existing properties is that we do not know which property is closely related to $\mathbf{y}$. We can solve this problem by generating new properties with existing properties such that not two of the new properties are uncorrelated with each other[3]. By combining existing properties to new properties, we do not "lose" any information. If $\mathbf{T}$ is our new property matrix and $\mathbf{t_i}, \mathbf{t_j}$ are its columns, that is

$$cov(\mathbf{t_i}, \mathbf{t_j}) = 0 \, if \, i \neq j$$

Such new properties can be found by a technique called principal component analysis (PCA). PCA will find a matrix $\mathbf{P}$ such that $\mathbf{T} = \mathbf{XP}$. Each column of $\mathbf{P}$ gives us one linear combination of existing properties and the new properties maintains the most variance of original property matrix $\mathbf{X}$. These linear combinations (columns of $\mathbf{P}$) are called principal components. PCA give us m principal components, the same amount as the amount of original properties. We do not have to use all of the principal components to generate new properties $\mathbf{T}$. One good thing about PCA is that the columns of $\mathbf{P}$ are ordered by the amount of variance they can explain, which means $\mathbf{p_1}$ explains more variance than $\mathbf{p_2}$, $\mathbf{p_2}$ explains more variance than $\mathbf{p_3}$... Under the important assumption that principal components that can only explain very little variance are likely to be noise, we can only use the first k principal component to generate new properties. So our new property matrix is

$$\mathbf{T}_{n\times k} = \mathbf{X}_{n\times m}\mathbf{P}_{m\times k}$$

.

With PCA, we can have uncorrelated properties from original properties. And principal component regression(PCR) is just linear regression with the new uncorrelated properties as independent variables and original dependent variable $\mathbf{y}$ as dependent variable. Since there are no correlated variables, OLS will not have trouble to work anymore.

In the paper[2], $\mathbf{T}$ and $\mathbf{P}$ are calculated by an iterative algorithm called the Nonlinear Iterative Partial Least Squares (NIPALS).

NIPALS:

1) Randomly pick any column $\mathbf{x_i}$ from $\mathbf{X}$ and let $\mathbf{t} = \mathbf{x_i}$
2)

$$\mathbf{p} = \frac{\mathbf{X}^T\mathbf{t}}{\mathbf{t}^T\mathbf{t}}$$

3)

$$\mathbf{p} = \frac{\mathbf{p}}{\|\mathbf{p}\|}(Normalize\ \mathbf{p})$$

4)

$$\mathbf{t} = \frac{\mathbf{Xp}}{\mathbf{p}^T\mathbf{p}}$$

5) If $\mathbf{t}$ converges, then $\mathbf{t}$ and $\mathbf{p}$ is one new property and one principal component. If $\mathbf{t}$ does not converge, repeat from step2. 6) $\mathbf{X} = \mathbf{X} - \mathbf{tp}^T$ for next run of NIPALS

Note: NIPALS gives only 1 principal component, so if we want to find all n principal components, we need to run NIPALS for n times. Another way to find $\mathbf{P}$ is by Singular Value Decomposition(SVD). The details will be skipped. The reason for elaborating on NIPALS is that it is very important in PLS.

5 Partial Least Square Regression

In this section, we will first provide a conceptual description of PLS, and then we will go through algorithm step by step. While PCA can find principal components that best explains the variance of independent variables $\mathbf{X}$, this variance is not related to dependent variable $\mathbf{y}$. There can be very important information hidden in dependent variable especially when there are more than 1 dependent variable. So conceptually, PLS decomposes $\mathbf{X}$ into components that not only explains variance of $\mathbf{X}$ but also explains variance of $\mathbf{Y}$. Note here y is written as $\mathbf{Y}$ instead of $\mathbf{y}$ because it represents multiple dependent variables $\mathbf{y_1}$, $\mathbf{y_2}$ ...

PLS is similar to PCA that $\mathbf{X}$ and $\mathbf{Y}$ are also decomposed into

$$\mathbf{X} = \mathbf{TP}^T$$
$$\mathbf{Y} = \mathbf{UQ}^T$$

Then a linear regression is applied onto $\mathbf{T}$ and $\mathbf{U}$. In order to decompose $\mathbf{X}$ /$\mathbf{Y}$ to get $\mathbf{T}$ /$\mathbf{U}$, we can perform NIPALS separately on each of $\mathbf{X}$ and $\mathbf{Y}$. However, separate decomposition means we lose covariance between $\mathbf{X}$ and $\mathbf{Y}$ which can be crucial to prediction. So the most important part of PLS is that it runs NIPALS in an altering way between $\mathbf{X}$ and $\mathbf{Y}$. Therefore, $\mathbf{T}$ and $\mathbf{U}$ generated by PLS can explain not only most variance of $\mathbf{X}$ and $\mathbf{Y}$ but also the most covariance between $\mathbf{X}$ and $\mathbf{Y}$.

Detail of PLS is as following (Many versions of PLS exist, we choose the one proposed by Prof.Herve Abdi [4])

Partial Least Square Regression

0) Normalize $\mathbf{X}$ and $\mathbf{Y}$

1) Let $\mathbf{E} = \mathbf{X}$, $\mathbf{F} = \mathbf{Y}$

2) let $\mathbf{u}$ be any random column in $\mathbf{F}$

3)

$$\mathbf{w} = \frac{\mathbf{E}^T\mathbf{u}}{\|\mathbf{E}^T\mathbf{u}\|}$$

4)

$$\mathbf{t} = \frac{\mathbf{E}\mathbf{w}}{\|\mathbf{E}\mathbf{w}\|}$$

5)

$$\mathbf{q} = \frac{\mathbf{F}^T\mathbf{t}}{\|\mathbf{F}^T\mathbf{t}\|}$$

6)

$$\mathbf{u} = \mathbf{F}\mathbf{q}$$

7) if $\mathbf{t}$ has not converged, repeat from step3. If $\mathbf{t}$ has converged

$$\mathbf{p} = \mathbf{E}^T\mathbf{t}$$
$$b = \mathbf{t}^T\mathbf{u}$$

8) $\mathbf{E} = \mathbf{E} - \mathbf{t}\mathbf{p}^T$, $\mathbf{F} = \mathbf{F} - b\mathbf{t}\mathbf{q}^T$ for next component. To find next component, first record $\mathbf{p}$ $\mathbf{t}$ $\mathbf{u}$ $\mathbf{q}$ $\mathbf{w}$ b, then repeat from step2

In order to make predictions, we need to use the $\mathbf{p}$, $\mathbf{q}$ and b generated by PLS. Let $\mathbf{P}$ be the matrix whose columns are $\mathbf{p}$, $\mathbf{Q}$ be the matrix whose columns are $\mathbf{q}$ and $\mathbf{B}$ be the diagonal matrix whose diagonals are b. Then prediction $\overline{\mathbf{Y}}$ is calculated as:

$$\overline{\mathbf{Y}} = \left(\mathbf{P}^T\right)^{-1}\mathbf{B}\mathbf{Q}^T$$

6 Optimization

We have used Python to implement PLS and uploaded to a Github repository at https://github.com/Juncheng-Dong/Partial-Least-Square (Click URL to go to the repository). After implementing the algorithm in vanilla Python, we tried to improve its performance.

To install our package, just

```
pip install -i https://test.pypi.org/simple/ pls==1.0.2
```

6.1 Numpy/Vectorization

Since most the algorithms is about calculation between matrix and vectors, we can use Numpy, one of the most optimized libraries, to help us. Performance benchmark is measured on a randomly generated 30×20 matrix $\mathbf{X}$ and 20×5 matrix $\mathbf{Y}$ being fully decomposed. Profiling and benchmarking are done by Jupyter notebook.

Benchmark

[150]:
```
%timeit -r3 -n3 PLS(X,Y,30)
```

```
5.07 ms ± 182 µs per loop (mean ± std. dev. of 3 runs, 3 loops each)
```

Profiling

[157]:
```
profile = %prun -r -q PLS(X,Y,30)
profile.sort_stats('cumtime').print_stats(10)
```

```
         892 function calls in 0.006 seconds

   Ordered by: cumulative time
   List reduced from 25 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.006    0.006 {built-in method builtins.
 ↪exec}
        1    0.000    0.000    0.006    0.006 <string>:1(<module>)
        1    0.002    0.002    0.006    0.006 <ipython-
input-148-abf33df6dec3>:1(PLS)
      290    0.002    0.000    0.002    0.000 <ipython-
input-6-359c7a80eeb8>:1(norm)
       30    0.001    0.000    0.001    0.000 {method 'choice' of
'numpy.random.mtrand.RandomState' objects}
        2    0.000    0.000    0.000    0.000 <ipython-
input-3-b74fda64ce2b>:1(Normalize)
       30    0.000    0.000    0.000    0.000 _dtype.py:319(_name_get)
```

```
       30    0.000    0.000    0.000    0.000 numerictypes.py:
↪365(issubdtype)
        2    0.000    0.000    0.000    0.000 {method 'std' of 'numpy.
↪ndarray'
objects}
        2    0.000    0.000    0.000    0.000 _methods.py:215(_std)
```

6.2 JIT

We have tried to compile it into native codes by using JIT as well as provide function prototype for pre-compilation and it has improved our implementation from 5.07 ms to 3.91 ms, more than 20 percent faster.

[342]:
```
%timeit PLS_numba(X,Y,30)
```

```
3.91 ms ± 313 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

6.3 Bottleneck

We can see from profiling that function **norm** was called 290 times. We can further improve our algorithm by using Numpy's optimized **norm** function. After replacing **norm**, we further improve the performance from 3.9 ms to 3.43ms, about 10 percent faster

[350]:
```
%timeit PLS(X,Y,30)
```

```
3.43 ms ± 79.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

6.4 Parallelism

Since this is an iterative algorithm, PLS has loops where parallelism may apply, but each loop is all dependent on results from last loop and computationally demanding , parallelism may cause troubles and provide little help. So we did not use multi-threads or multi-process.

6.5 Pre-compiling and Encapsulation

We encapsulated our code in object-oriented programming as Python Class PLS. And important method is already pre-compiled during importing time with numba function signature. Related tests are also done in the form of jupyter notebook to prevent problems like predicting before training/ unmatched data dimension/ components number over the limit etc.

7 Application

7.1 Application on Simulated Data

Partial Least Squared Regression (PLS) is a variant of Ordinary Least Squared Regression (OLS), and it is an alternative method when solution is not feasible by OLS (eg. Multicolinearity and multiple dependent variables). However, when the number of sample equals the number of features and there is no multicolinearity among features, PLS should perform identically as OLS and give one exact solution.

To test this property, we randomly generate a 5-by-5 feature matrix **X**, and a 5-by-1 response matrix **Y**.

[10]:
```
# OLS vs PLS
X_sim = np.random.randn(5, 5)
X_sim
```

[10]:
```
array([[ 0.54866045,  0.44999304, -0.1671458 , -1.52615899,  1.64624995],
       [-1.49519606, -0.19913497,  0.61932406, -0.67909876,  0.0185365 ],
       [ 0.22316393,  0.67706721,  0.02651495,  1.16929049,  1.58098924],
       [-0.0773167 , -0.09873618,  0.63116646, -0.34468435,  0.1556186 ],
       [-0.68651197,  0.04042758, -0.17329471,  1.880685  , -0.0833826 ]])
```

[11]:
```
Y_sim = np.random.randn(5,1)
Y_sim
```

[11]:
```
array([[ 1.58777908],
       [-0.56625565],
       [ 0.75361078],
       [-2.06295635],
       [ 0.49627838]])
```

Then, we normalize both X and Y, and get B_P by using our PLS function and B_O by using OLS function in Sklearn Package.

[12]:
```
X_sim = Normalize(X_sim)
Y_sim = Normalize(Y_sim)
```

[17]:
```
from sklearn.linear_model import LinearRegression

OLS = LinearRegression()
B_O = OLS.fit(X_sim, Y_sim).coef_.T
B_O
```

[17]:
```
array([[-0.63015429],
       [ 0.28514484],
       [-0.91384279],
       [-0.28843675],
       [ 0.3701709 ]])
```

[18]:
```
[T, P, W, Q, U, B] = PLS(X_sim,Y_sim,5)

P = P.T
Q = Q.T
B_P = la.pinv(P.T)@B@Q.T
B_P
```

[18]:
```
array([[-0.63015429],
       [ 0.28514484],
       [-0.91384279],
       [-0.28843675],
       [ 0.3701709 ]])
```

[19]:
```
np.allclose(B_O, B_P)
```

[19]:
```
True
```

The result shows that when there exists an exact solution, solutions generated by OLS and PLS are identical. Thus, this simulated data case proves the fundamental property of PLS and also shows that our PLS function is correct.

7.2 Application on Real Data Sets & Comparative analysis

In this section, we use a real car dataset, which is a sample drawn from the Automobile Datan Set available on the UCI Machine Learning Repository Server[5]. We want to explain the costs indicators of cars (price, consumption, symboling) from their characteristics (engine size, fuel type, etc.)

We first use PLS to predict coefficients using leave-one-out method, and test its performance using mean squared error. Then, we compare our PLS result with results generated by other competing algorithms, such as OLS, ridge regression, and PCR. OLS is the basic form of all these algoritms; Ridge regression is also a variant of OLS, while Ridge regression can eases the problem of multicolinearity by its shrinkage parameter λ; Principal Component Regression(PCR) can also solve the problem of multicolinearity by decomposing original X into principal components, but it is not able to capture the covariance between X and Y as PLS does.

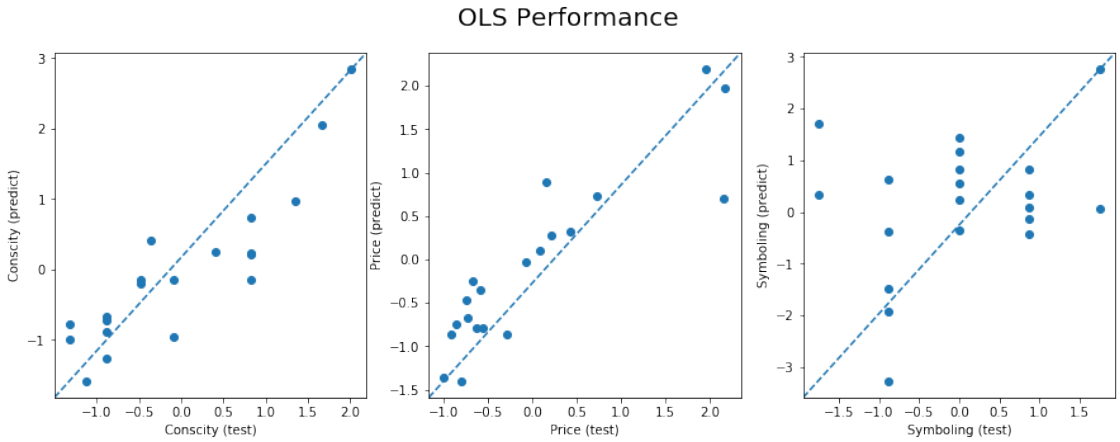
The result shows in the table below. PLS outperformed all three of other algorithms. It is not surprising to see that OLS performs the worst, as it has no mechanism to solve multicolinearity. Since OLS only allows one dependent variable in one regression, we

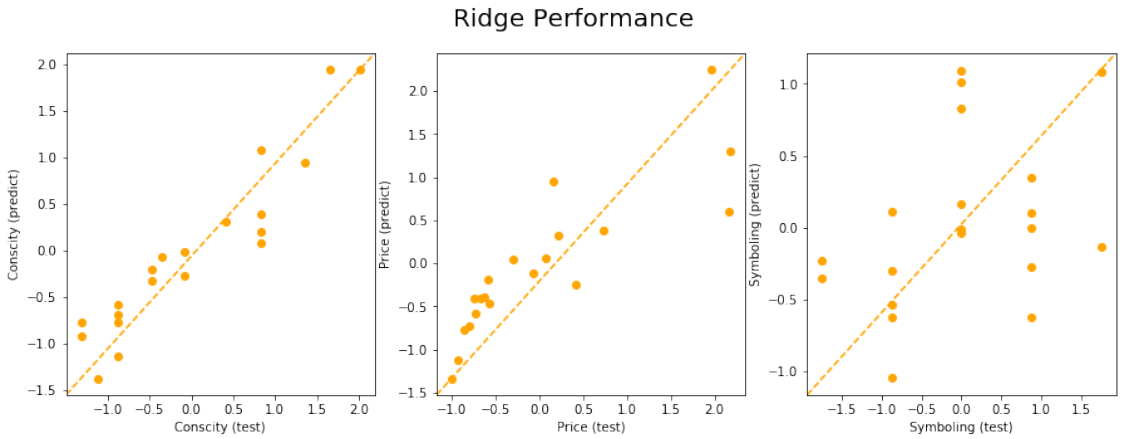
estimate coefficients for price, consumption and symboling separately by three OLS regressions on the same set of features (X). Thus, OLS is not able to capture any covariance among dependent variables as well. The result of Ridge Regression is slightly better than PCR. Although both methods try to reduce variance by using the largest singular values of the design matrix X, Ridge Regression smoothly shrinks the singular vectors whereas PCR simply throws out the worst. Thus, PCR may lost some information by completely leaving less important components out.

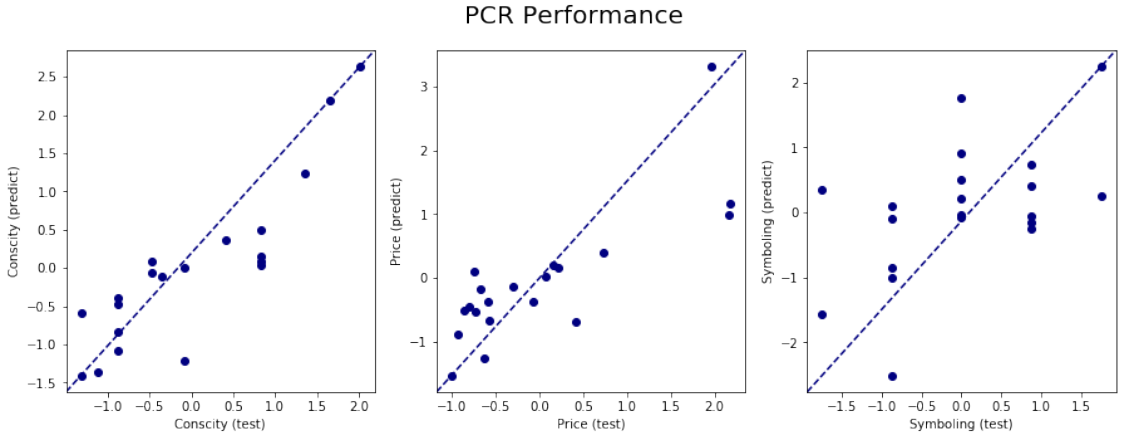
	OLS	**Ridge**	**PCR**	**PLS**
MSE	0.788	0.426	0.529	0.380

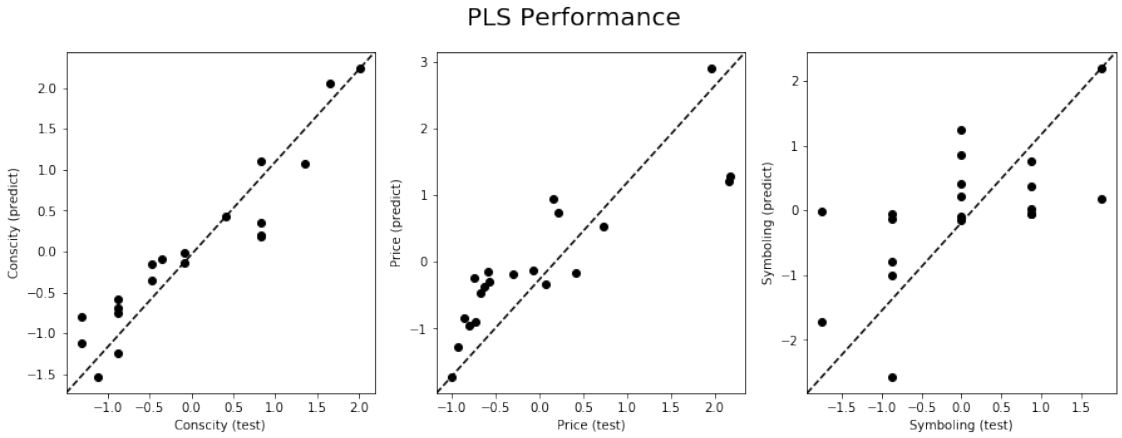
We also present the result all four of above algorithms by graphs, and we can closely see how our algorithm's performance in three dependent variables (price, conscity, and symboling).

In the following graph, X axis represents true values of dependent variable, and Y axis represents predicted values by algorithms. The more the shape of scatters is close to a 45 degree line, the more accurate the algorithm is.









From graphs above, Ridge and PLS both perform good on the first dependent variable:

Conscity. The distribution of scatters is very tight and close to the line. OLS's scatters are also close to a line-shape, but its distribution is comparatively looser, which shows that the variance of estimators by OLS is overall larger than Ridge and PLS.
On the other hand, OLS actually performs well in estimating the second dependent variable: price. This may because all features are highly correlated to pirce, and dropping "less important" features in other algorithms actually lose some important information and thus weakening the advantages of other algorithms.
All algorithms we test perform not well on the third dependent variable: Symboling. This dependent variable may not have linear relation with our features. To improve our model in the future, we can try nonlinear models to find Symboling's estimators.

8 Conclusion

Partial Least Square (PLS) is a powerful algorithm that can solve the problem of insufficient data and the problem of high-correlation between independent variables. Both problems are common but very difficult to solve in real life analysis. By using PLS, we can solve this difficulties by finding latent variables which explain the most covariance between $\mathbf{X}$ and $\mathbf{Y}$ as well as variance of $\mathbf{X}/\mathbf{Y}$. By discarding latent variables with minor importance, PLS can also lower the noise in data. When all latent components are used, PLS behaves the same as ordinary least square (OLS). However, PLS can also have a solution when OLS does not work. Thus, PLS is an important and useful algorithm. We implemented PLS in python and optimized it (`https://github.com/Juncheng-Dong/Partial-Least-Square`). To summarize, PLS can work normally as OLS when OLS has a solution while it can also work when OLS cannot work because of data insufficiency and multi-collinearity. PLS finds the latent variables that best explains variance and covariance in the data. PLS also gives the user some freedom to choose how many latent variables to use when doing prediction so that noise in the data can be eliminated.

References

[1] PAUL GELADI, and BRUCE R KOWALSKI.

PARTIAL LEAST-SQUARES REGRESSION: A TUTORIAL . 1985.

[2] PAUL GELADI, and BRUCE R KOWALSKI.

AN EXAMPLE OF 2-BLOCK PREDICTIVE PARTIAL LEAST-SQUARES REGRESSION WITH SIMULATED DATA. 1985.

[3] Kee Siong Ng. *A Simple Explanation of Partial Least Squares*.
`http://users.cecs.anu.edu.au/ kee/pls.pdf`. 2013.

[4] Herv´e Abdi *Partial Least Squares (PLS) Regression*
`https://personal.utdallas.edu/ herve/Abdi-PLS-pretty.pdf`

[5] Data Source:
http://archive.ics.uci.edu/ml/datasets/Automobile