Regression in Python

Kaleb Cervantes

Intro

Last semester, we went over how to do regression in R. This is fairly straightfoward as we just use the functions `lm` and `glm` to make the models. In R, we simply needed the data, and the formula. However this is more complicated in Python.

First I will want to import the following libraries:

- `sklearn` for the linear and logistic regression functions
- `numpy` for matrix stuff
- `pandas` for viewing dataframes.

```
import sklearn.linear_model as skl
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
```

I will also be using the auction verification dataset from UCI repository. This is because it has both a numeric and binary response for linear and logistic regression respectively. The head of the data is given in the following three tables. The first two are predictors, and the third are responses.

```
auction_data = pd.read_csv("data.csv")

auction_data.iloc[0:4, 0:4]
```

	process.b1.capacity	process.b2.capacity	process.b3.capacity	process.b4.capacity
0	0	0	2	1
1	0	0	2	1

	process.b1.capacity	process.b2.capacity	process.b3.capacity	process.b4.capacity
2	0	0	2	1
3	0	0	2	1

```
auction_data.iloc[0:4, 4:7]
```

	property.price	property.product	property.winner
0	59	1	0
1	59	2	0
2	59	4	0
3	59	6	0

```
auction_data.iloc[0:4, 7:9]
```

	verification.result	verification.time
0	False	163.316667
1	False	200.860000
2	False	154.888889
3	False	108.640000

Preparing The Data

It may help to understand the dimensions of the data.

```
auction_data.shape
```

```
(2043, 9)
```

Since this data has a fairly large amount of observations, it may help to split the data into training and test sets. For this, it may help to identify the following responses:

- `verification.result` for logistic regression
- `verification.time` for linear regression

Handling Predictors

It may help to do some exploration with the predictors first. Now it may be important to note that — although all of the predictors are stored as integers — there may be some categorical data. From reading the documentation, this seems to be the case for the following:

- `property.product` — product code for product currently being verified.
- `property.winner` — `0` if price was verified, otherwise bidder code for bidder currently being verified

The above columns will be transformed using dummy variables, with their default value `0` being ignored. Conveniently, the responses are also the last two columns in the dataframe. This means that our predictors are the first seven columns. Since indeces in Python begin at 0, the following code chunk will extract the predictors and add dummy variables.

```
X = pd.get_dummies(
    auction_data.iloc[:, 0:7],
    columns= ["property.product", "property.winner"],
    drop_first=True
)
```

It may also help to note that the dimensions have changed.

```
X.shape
```

```
(2043, 14)
```

Even though most of these are dummy variables, they will add more coefficients to the regression model.

Splitting the Data

Now that the predictors have been handled, we can split the data. This process may need to be repeated later on depending on the circumstances. This is actually one of the areas where Python is more straightfoward than — base — R.

the function `sklearn.model_selection.train_test_split` splits the into training and test data. The first inputs for this function are the predictor matrix and response vector — or vectors if splitting for multiple responses. By default this function does a 75-25 split for training and testing. We can specify the split by putting the desired ratio for either group in the parameters `test_size` or `train_size`. There is also the function `random_state` which

can be used to specify the seed set for random sample used. Dr. Kerr used the R equivelent for reproducabiliity so I intend on doing the same.

```
y_lin = auction_data["verification.time"]
y_log = auction_data["verification.result"]

(
    X_train, X_test,
    y_lin_train, y_lin_test,
    y_log_train, y_log_test
) = train_test_split(
    X, y_lin, y_log,
    test_size = 0.4,
    random_state = 2022
)
```

Now that the data has been split, we can finally fit our model.

Linear Models

Similar to `lm` in R, `sklearn.linear_model.LinearRegression` is an object for our linear model. The function `fit` will be used to actually fit the model.

In order to see the R^2 coefficient, we use the function `score`. We will first check this for the training data.

```
lm1 = (
    skl
    .LinearRegression()
    .fit(X_train, y_lin_train)
)

lm1.score(X_train, y_lin_train)
```

```
0.6605544724598729
```

From this it appears that `lm1` only accounts for about 66% of the variability in the model. Now we try to see what this value would be for the test data.

```
lm1.score(X_test, y_lin_test)
```

```
0.6182826928671714
```

From this it appears that only about 62% of the variability in the test data is accounted for by the model. This is not perfect, but given the R^2 for the training data this seems ok.

If we want to see the coefficients of the model, we have to access the attributes `coef_` for the predictors and `intercept_` for the intercept.

```
lm1.intercept_
```

```
8190.3895882599245
```

```
lm1.coef_
```

```
array([  5850.77182834,     84.72484738,  -1143.55239713,   2362.55568998,
          -48.5921103 ,   9920.36748465, -10062.44991919,  -7612.87468259,
        -5228.64958719,  -9800.90224685,  -5832.05652052,   1195.38994045,
         1986.24482746,    691.72620791])
```

In R, the functions `lm`, `summary.lm`, and `plot.lm` do most of the above and much more. Unfortunately, Python doesn't allow for diagnostics to be done as easily. There are other packages beside sklearn that do them, but they are not as efficient as doing them in R. This is why — although I prefer Python as a language over R — R is much better for regression and diagnostics.

Logistic Regression

When we split the data, we included both responses. Luckily this means that the splitting portion has been done for the logistic regression. Doing this is very similar to linear regression. It is important to note that by default, logistic regression in python applies an l2 penalty. This is similar to ridge regression. To remove the penalty, I set the parameter `penalty` to `"none"`.

```
glm1 = (
    skl
    .LogisticRegression("none")
    .fit(X_train, y_log_train)
)
```

Now that the logistical model is fit, I will use `score` to check accuracy. Here `score` returns the number of correct predictions divided by the total number of predictions.

```
glm1.score(X_train, y_log_train)
```

```
0.9069387755102041
```

```
glm1.score(X_test, y_log_test)
```

```
0.902200488997555
```

The logistic model predicted with an accuracy of about 90% for both the test and training sets.

Unfortunately, a lot of the model diagnostics still have to be manually done as there are not simple `plot.glm` or `summary.glm` in `scikitlearn`.

statsmodels

`statsmodels` is a python library that allows users to use R-style formulas in Python. This would be useful for mixed model stuff, but in this document I will mostly use it to read model summaries.

Since we are using this library, we will use the function `add_constant` to add the constant term to the training data matrix.

```
import statsmodels.api as sm

from statsmodels.tools.tools import add_constant
```

Linear Regression Tables

Now we will view the model summary tables. There is a third table, but we did not go over what it shows in STAT 632. As such I will only show the first two.

```
X_train_sm = add_constant(X_train)

lm1_summary_tables = (
```

```
    sm
    .OLS(y_lin_train, X_train_sm)
    .fit()
    .summary()
    .tables
)

lm1_summary_tables[0]
```

Table 4: OLS Regression Results

Dep. Variable:	verification.time	R-squared:	0.661
Model:	OLS	Adj. R-squared:	0.657
Method:	Least Squares	F-statistic:	168.2
Date:	Sat, 14 May 2022	Prob (F-statistic):	7.69e-272
Time:	16:40:08	Log-Likelihood:	-12425.
No. Observations:	1225	AIC:	2.488e+04
Df Residuals:	1210	BIC:	2.496e+04
Df Model:	14		
Covariance Type:	nonrobust		

```
lm1_summary_tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	8190.3896	2227.571	3.677	0.000	3820.059	1.26e+04
process.b1.capacity	5850.7718	261.338	22.388	0.000	5338.046	6363.498
process.b2.capacity	84.7248	226.182	0.375	0.708	-359.028	528.478
process.b3.capacity	-1143.5524	643.268	-1.778	0.076	-2405.598	118.493
process.b4.capacity	2362.5557	385.525	6.128	0.000	1606.185	3118.926
property.price	-48.5921	27.998	-1.736	0.083	-103.523	6.339
property.product_2	9920.3675	547.889	18.107	0.000	8845.449	1.1e+04
property.product_3	-1.006e+04	636.074	-15.820	0.000	-1.13e+04	-8814.520
property.product_4	-7612.8747	655.433	-11.615	0.000	-8898.785	-6326.964
property.product_5	-5228.6496	687.055	-7.610	0.000	-6576.601	-3880.698
property.product_6	-9800.9022	598.886	-16.365	0.000	-1.1e+04	-8625.932
property.winner_1	-5832.0565	1137.179	-5.129	0.000	-8063.118	-3600.995
property.winner_2	1195.3899	780.814	1.531	0.126	-336.509	2727.289
property.winner_3	1986.2448	765.638	2.594	0.010	484.120	3488.370
property.winner_4	691.7262	966.348	0.716	0.474	-1204.177	2587.629

From this we can see that both of the categorical variables have significant and insignificant levels. I will check Dr. Kerr's notes on how to handle these.

The variable `process.b2.capacity` is not significant at any resonable level of α.

We also notice that `property.price` and `process.b3.capacity` are not significant at the $\alpha = 0.05$ level, but would be significant at the $\alpha = 0.1$ level.

Logistic Regression Table

```
(
    sm
    .Logit(y_log_train, X_train_sm)
    .fit()
    .summary()
    .tables[1]
)
```

```
Optimization terminated successfully.
         Current function value: 0.249993
         Iterations 8
```

	coef	std err	z	P>|z|	[0.025	0.975]
const	-11.7327	1.591	-7.375	0.000	-14.851	-8.615
process.b1.capacity	-1.3342	0.208	-6.405	0.000	-1.742	-0.926
process.b2.capacity	-0.3079	0.135	-2.287	0.022	-0.572	-0.044
process.b3.capacity	-0.0702	0.328	-0.214	0.831	-0.713	0.573
process.b4.capacity	-0.0463	0.238	-0.195	0.846	-0.512	0.419
property.price	0.1464	0.021	7.026	0.000	0.106	0.187
property.product_2	-0.2652	0.396	-0.669	0.503	-1.042	0.511
property.product_3	1.1049	0.373	2.960	0.003	0.373	1.837
property.product_4	1.3371	0.410	3.257	0.001	0.533	2.142
property.product_5	-0.8076	0.434	-1.863	0.063	-1.658	0.042
property.product_6	1.3737	0.394	3.488	0.000	0.602	2.146
property.winner_1	5.1913	0.794	6.542	0.000	3.636	6.747
property.winner_2	2.1573	0.294	7.329	0.000	1.580	2.734
property.winner_3	-0.9584	0.498	-1.926	0.054	-1.934	0.017
property.winner_4	0.1983	0.430	0.461	0.644	-0.644	1.041

Reduced Models

This is a bit more complicated in Python than it is in R. In R, we were able to remove predictors in the formula. In Python, we have to remove the corresponding columns in the dataframe or matrix. The following will remove the columns and refit the model.

Reduced Linear Model

```
X_train_lin_reduced = X_train.drop(
    ["process.b2.capacity", "process.b3.capacity", "property.price"],
    1
)
X_test_lin_reduced = X_test.drop(
    ["process.b2.capacity", "process.b3.capacity", "property.price"],
    1
)

lm2 = (
    skl
    .LinearRegression()
    .fit(X_train_lin_reduced, y_lin_train)
)

lm2.score(X_train_lin_reduced, y_lin_train)
```

```
0.6585338975456794
```

```
lm2.score(X_test_lin_reduced, y_lin_test)
```

```
0.6124603279912988
```

From the above, we can see that there does not seem to be a significant difference in the R^2 from reducing the models. We can now look at the new table for the coefficients.

```
(
    sm
    .OLS(y_lin_train, add_constant(X_train_lin_reduced))
    .fit()
```

```
    .summary()
    .tables[1]
)
```

	coef	std err	t	P>|t|	[0.025	0.975]
const	3083.6728	491.000	6.280	0.000	2120.370	4046.976
process.b1.capacity	5532.9222	228.843	24.178	0.000	5083.949	5981.895
process.b4.capacity	2385.9731	385.147	6.195	0.000	1630.345	3141.601
property.product_2	9808.2027	532.179	18.430	0.000	8764.109	1.09e+04
property.product_3	-1.013e+04	629.158	-16.101	0.000	-1.14e+04	-8895.575
property.product_4	-7245.2568	630.906	-11.484	0.000	-8483.045	-6007.469
property.product_5	-5565.8221	655.438	-8.492	0.000	-6851.741	-4279.904
property.product_6	-9645.2083	578.390	-16.676	0.000	-1.08e+04	-8510.452
property.winner_1	-6320.4783	1104.305	-5.723	0.000	-8487.038	-4153.918
property.winner_2	1010.4613	761.877	1.326	0.185	-484.281	2505.204
property.winner_3	1719.5593	747.140	2.302	0.022	253.729	3185.390
property.winner_4	487.4516	955.656	0.510	0.610	-1387.470	2362.373

Reduced Logistic Model

```
X_train_log_reduced = X_train.drop(
    ["process.b3.capacity", "process.b4.capacity"],
    1
)
X_test_log_reduced = X_test.drop(
    ["process.b3.capacity", "process.b4.capacity"],
    1
)

glm2 = (
    skl
    .LogisticRegression("none")
    .fit(X_train_log_reduced, y_log_train)
)

glm2.score(X_train_log_reduced, y_log_train)
```

```
0.9044897959183673
```

```
glm2.score(X_test_log_reduced, y_log_test)
```

```
0.9009779951100244
```

```
(
    sm
    .Logit(y_log_train, add_constant(X_train_log_reduced))
    .fit()
    .summary()
    .tables[1]
)
```

```
Optimization terminated successfully.
         Current function value: 0.250025
         Iterations 8
```

	coef	std err	z	P>|z|	[0.025	0.975]
const	-11.8459	1.535	-7.718	0.000	-14.854	-8.838
process.b1.capacity	-1.3453	0.201	-6.709	0.000	-1.738	-0.952
process.b2.capacity	-0.3021	0.133	-2.273	0.023	-0.562	-0.042
property.price	0.1458	0.021	7.075	0.000	0.105	0.186
property.product_2	-0.2775	0.389	-0.713	0.476	-1.040	0.485
property.product_3	1.1044	0.373	2.960	0.003	0.373	1.836
property.product_4	1.3415	0.410	3.269	0.001	0.537	2.146
property.product_5	-0.8201	0.422	-1.942	0.052	-1.648	0.008
property.product_6	1.3697	0.393	3.487	0.000	0.600	2.140
property.winner_1	5.1986	0.792	6.563	0.000	3.646	6.751
property.winner_2	2.1678	0.292	7.426	0.000	1.596	2.740
property.winner_3	-0.9585	0.497	-1.929	0.054	-1.933	0.016
property.winner_4	0.1852	0.419	0.442	0.658	-0.635	1.006

Conclusion

Regression can be done in Python and with the use of libraries like `statsmodels`, may offer the same tools that can be used in R. I think that with how common Python is, it is worth learning these methods. However they are not as simple as they are in R.