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]
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'data.csv'
```

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

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

Preparing The Data

It may help to understand the dimensions of the data.

```
auction_data.shape
```

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
```

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)
```

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)
```

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_
```

```
lm1.coef_
```

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)
```

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

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]
```

```
lm1_summary_tables[1]
```

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]
)
```

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)
```

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

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]
)
```

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)
```

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

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

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.