Modeling through Statsmodels/Sklearn

Customization vs Rapid Development

Building our own models is great*!

- Understand the assumptions
- Get EXACTLY what you need

Unfortunately, it takes a LOT of time!

*HAHAHAHAHAHAHAHAHAHA.... 😭

UPDATED TITLE

Modeling through Statsmodels/Sklearn: You're going to hate me

Statsmodels

Let's make statistics in Python easy!

Importing Statsmodels

We can import `statsmodels` in one of two ways:

1. With support for R-style formulas:

```
import statsmodels.formula.api as sm
```

2. Using pre-built numpy arrays as inputs:

```
import statsmodels.api as sm
```

Let's start with option 1...

Preparing a Dataset

When using formulas, we prepare our dataset by importing the data into a Pandas `DataFrame`. We should take care that each of our variables has a name with

1. **No spaces**
2. No symbols
3. Made up of letters and numbers (also can't have a number as the first character)

Preparing a Dataset

Our code so far might look something like:

```
import statsmodels.formula.api as smf
import pandas as pd, numpy as np

data = pd.read_csv("https://github.com/dustywhite7/Econ8320/blob/"
                + "master/AssignmentData/assignment8Data.csv?raw=true")
```

Assuming that our data set has already been cleaned

Regression Equations

`statsmodels` incorporates `R`-style regression equations by using the `patsy` library behind the scenes. The pattern is as follows:

```
"dependent variable ~ independent variable + another independent variable + any other independent variables"
```

The regression equation will be held in a string (unlike in `R`)

Implementing a Model

The first thing we might try is a simple linear regression:

```
reg = smf.ols("hhincome ~ educ", data=data).fit()
print(reg.summary())
```

Or, I might want to try regressing education on **logged** household incomes:

```
reg = smf.ols("np.log(hhincome) ~ educ", data=data).fit()
print(reg.summary())
```

Advancing our Model

It might be useful to create effects by including dummy variables for the values in our `race` column.

```
reg = smf.ols("np.log(hhincome) ~ educ + C(race)",
        data=data).fit()
print(reg.summary())
```

The `C()` command indicates that we would like to consider the `race` variable as a **C**ategorical variable, not a numeric variable.

Additional Transformations

Sometimes we want to include transformed variables in our model:

```
# Square a variable using the I() function for
#   mathematical transformations
reg = smf.ols("np.log(hhincome) ~ age + I(age**2)",
        data=data).fit()
```

```
# Combine variables using the I() function for
#   mathematical transformations
reg = smf.ols("np.log(hhincome) ~ I(age-educ-5)",
        data=data).fit()
```

Robust Modeling

If we want to utilize robust standard errors, we can update our regression results:

```
reg = smf.ols("np.log(hhincome) ~ educ + C(race)",
        data=data).fit()
# Use White's (1980) Standard Error
reg.get_robustcov_results(cov_type='HC0')
print(reg.summary())
```

More robust modeling

```
reg = smf.ols("np.log(hhincome) ~ educ + C(race)",
        data=data).fit()
# Use Cluster-robust Standard Errors
reg.get_robustcov_results(cov_type='cluster',
        groups=data['race']) # Need to specify groups
print(reg.summary())
```

Robust Modeling

Below are some of the covariance options that we have:

1. `HC0`: White's (1980) Heteroskedasticity robust standard errors
2. `HC1`, `HC2`, `HC3`: MacKinnon and White's (1985) alternative robust standard errors, with `HC3` being designed for improved performance in small samples
3. `cluster`: Cluster robust standard errors
4. `hac-panel`: Panel robust standard errors

Time Series Models

We have multiple time series options available.

- ARIMA models
- VAR models
- Exponential Smoothing models

Modeling Discrete Outcomes

If we have a binary dependent variable, we are able to use either Logit or Probit models to estimate the effect of exogenous variables on our outcome of interest. To fit a Logit model:

```
import statsmodels.api as sm

myformula = "married ~ hhincome + C(race) + educ"
model = sm.Logit.from_formula(myformula, data=data).fit()
```

Modeling Count Data

When modeling count data, we have options such as Poisson and Negative Binomial models.

```
myformula = "nchild ~ hhincome + C(race) + educ + married"

model = sm.Poisson.from_formula(myformula, data=data).fit()
```

`patsy`: Using Regression Equations

Breaking out our regression equations!

Why use `patsy`?

- We could just select our variables manually, and creating a column of ones is trivial (remember??)
- Patsy allows us to separate our endogenous and exogenous variables AND to
    - "Dummy out" categorical variables
    - Easily transform variables (square, or log transforms, etc.)
    - Use identical transformations on future data

Why use `patsy` when `statsmodels` handles it for us?

By breaking out our regression equations, we can use the same data splits and processing steps for both `statsmodels` and for `sklearn` (which does not use `patsy`)!

Getting Started

```
import patsy as pt
import pandas as pd
import numpy as np

data = pd.read_csv("https://github.com/dustywhite7/Econ8320/blob/"
                + "master/AssignmentData/assignment8Data.csv?raw=true")

# To create y AND x matrices
y, x = pt.dmatrices("hhincome ~ year + educ + married + age",
                data = data)

# To create ONLY an x matrix
x = pt.dmatrix("~ year + educ + married + age",
                data = data)
```

These regression equations automatically include an intercept term.

Categorical Variables

```
# To create y AND x matrices
eqn = "hhincome ~ C(year) + educ + married + age"
y, x = pt.dmatrices(eqn, data = data)
```

Categorical variables can be broken out into binary variables using the `C()` syntax inside of the regression equation.

In this case, there would be binary variables for each unique value of `year`.

Transforming Variables

```
# To create y AND x matrices
eqn = "I(np.log(hhincome)) ~ C(year) + educ + married + age + I(age**2)"
y, x = pt.dmatrices(eqn, data = data)
```

We can transform variables using the `I()` syntax inside of the regression equation. We then use any numeric transformation that we choose to impose on our data.

In this case, we logged our dependent variable, `hhincome`, and added the square of our `age` term.

SUPER IMPORTANT $\rightarrow$ Same Transformation on New Data!

```
# To create a new x matrix based on our previous version

xNew = pt.build_design_matrices([x.design_info], dataNew)
```

We can create a new matrix in the SAME SHAPE as our original `x` matrix by using the `build_design_matrices()` function in `patsy`.

We pass a list containing the old design matrix information, as well as the new data from which to construct our new matrix.

Why Does Recreating our `X` array Matter?

- Ensures that we always have the same number of categories
- Maintains consistency in our model
- Makes our work replicable
- AGAIN - ❤️ streamlines the use of `statsmodels` and `sklearn` in the same workflow ❤️

scikit-learn

see 🤖 learn

learn, 🤖, learn!

Predictive Modeling

What `statsmodels` does for regression analysis, `sklearn` does for predictive analytics and machine learning.

- Likely the most popular machine learning library today
- Has a standard API to make using the library VERY simple.

```
# Import some data...
data = pd.read_csv("https://github.com/dustywhite7/Econ8310/
raw/master/DataSets/occupancyTrain.csv")

# Build x, y matrices
y, x = pt.dmatrices("Occupancy ~ -1 + Light + CO2", data=data)
```

Decision Tree Classification (and Regression)

Classification and Regression Trees (CARTs) are the standard jumping-off point for exploring machine learning. They are very easy to implement in `sklearn`:

```
from sklearn import tree
from sklearn.metrics import accuracy_score

clf = tree.DecisionTreeClassifier()
clf = clf.fit(x, y)

pred = clf.predict(new_xs)

print(accuracy_score(new_ys, pred))
```

Support Vector Machines

We also implement Support Vector Machines for both classification and regression:

```
from sklearn import svm
from sklearn.metrics import accuracy_score

clf = svm.SVC()
clf = clf.fit(x, y)

pred = clf.predict(new_xs)

print(accuracy_score(new_ys, pred))
```

Can you see the API pattern yet?

Random Forest Models

Again, available in both classification and regression flavors, these models are aggregations of many randomized Decision Trees.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

clf = RandomForestClassifier(n_estimators=50)
clf = clf.fit(x, y)

pred = clf.predict(new_xs)

print(accuracy_score(new_ys, pred))
```

There MUST be a pattern here...

More from `sklearn`

Many other tools are also available to aid in the data cleaning process through `sklearn`. Some of these are:

- Principal Component Analysis (PCA)
- Factor Analysis
- Many Cross-Validation Algorithms
- Hyperparameter Tuning
    - Finding the correct parameters for a decision tree or random forest, for example
- Model Evaluation Tools
- Plotting decision trees

Lab time!