Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

July 7, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Introduction to `scikit-learn`
2. `statsmodels` + `patsy`
3. `pyMC3` + `arviz`
4. `pyArrow` - Apache Arrow Python bindings
5. More material

`scikit-learn`

`scikit-learn` is an open-source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities.

- Simple and efficient tools for predictive data analysis
- Accessible to everybody and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

This is one of several other "scikits" (e.g. scikit-image), which are scientific toolboxes built on top of `scipy`.

`scikit-learn`: what can we do with `scikit-learn`?

The `sklearn` package contains a large number of submodules which are specialized for different tasks/models,

- `sklearn.base`
- `sklearn.calibration`
- `sklearn.cluster`
- `sklearn.compose`
- `sklearn.covariance`
- `sklearn.datasets`
- `sklearn.decomposition`
- `sklearn.ensemble`
- sklearn.exceptions
- sklearn.experimental
- sklearn.feature_extraction
- sklearn.feature_selectio
- sklearn.gaussian_process
- sklearn.impute
- sklearn.inspection
- sklearn.isotonic
- sklearn.kernel_approximation
- sklearn.kernel_ridge
- sklearn.linear_model

scikit-learn: more submodules....

- sklearn.manifold
- sklearn.metrics
- sklearn.mixture
- sklearn.model_selection
- sklearn.multiclass
- sklearn.multioutput
- sklearn.naive_bayes
- sklearn.neighbors
- sklearn.neural_network
- sklearn.pipeline
- sklearn.preprocessing
- sklearn.random_projection
- sklearn.semi_supervised
- sklearn.svm
- sklearn.tree
- sklearn.utils

Statsmodels + patsy

statsmodels

> `statsmodels` is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration. An extensive list of result statistics is available for each estimator. The results are tested against existing statistical packages to ensure that they are correct.

```python
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.tsa.api as tsa
```

`statsmodels` uses slightly different terminology for refering to `y` / dependent / response and `x` / independent / explanatory variables. Specifically, it uses `endog` to refer to the `y` and `exog` to refer to the `x` variable(s).

This is particularly important when using the main API, less so when using the formula API.

OpenIntro loans data

> This data set represents thousands of loans made through the Lending Club platform, which is a platform that allows individuals to lend to other individuals. Of course, not all loans are created equal. Someone who is essentially a sure bet to pay back a loan will have an easier time getting a loan with a low-interest rate than someone who appears to be riskier. And for very risky people? They may not even get a loan offer or may have yet to accept the request due to a high-interest rate. It is important to remember that last part since this data set only represents loans actually made, i.e. do not mistake this data for loan applications!

For the full data dictionary, see here. We removed some columns to make the data set more reasonably sized and dropped any rows with missing values.

```
loans = pd.read_csv("data/openintro_loans.csv")
```

statsmodels: OLS use case

```
y = loans["loan_amount"]
X = loans[["homeownership", "annual_income", "debt_to_income", "
    interest_rate", "public_record_bankrupt"]]

model = sm.OLS(endog=y, exog=X)

# ValueError: Pandas data cast to numpy dtype of object. Check
    input data with np.asarray(data).
```

What do you think the issue is here?

The error occurs because X contains mixed types. Specifically, we have categorical data columns which cannot be directly converted to a numeric dtype. Hence, we need to take care of the dummy coding for statsmodels (with this interface).

```
X_dc = pd.get_dummies(X)
model = sm.OLS(endog=y, exog=X_dc)
```

statsmodels: fitting and summary

```
res = model.fit()
print(res.summary())
```

In contrast to pandas or scikit-learn, the summary is more detailed and provides you with more information and even references.

Please, run it and analyze the output information.

Most of the modelling interfaces are also provided by smf (statsmodels.formula.api) in which case patsy is used to construct the model matrices.

```
model = smf.ols(
"loan_amount ~ homeownership + annual_income + debt_to_income +
    interest_rate + public_record_bankrupt",
data = loans )

res = model.fit()
print(res.summary())
```

statsmodels: more features...

- Logistic regression models (GLM)

```
y = pd.get_dummies( possum["pop"] )
X = pd.get_dummies( possum.drop(["site","pop"], axis=1) )

model = sm.GLM(y, X, family = sm.families.Binomial())
res = model.fit()
print(res.summary())
```

- t-test and z-test for equality of means

```
cm = sm.stats.CompareMeans(
sm.stats.DescrStatsW( books.weight[books.cover == "hb"] ),
sm.stats.DescrStatsW( books.weight[books.cover == "pb"] ))

print(cm.summary())
```

- Contigency tables

```
gss = pd.DataFrame({"US": [454, 226], "Duke": [56,32]},
    index=["A great deal", "Not a great deal"])
tbl = sm.stats.Table2x2(gss.to_numpy())
print(tbl.summary())
```

patsy

> `patsy` is a Python package for describing statistical models (especially linear models, or models that have a linear component) and building design matrices. It is closely inspired by and compatible with the formula mini-language used in R and S.

> `patsy`'s goal is to become the standard high-level interface to describing statistical models in Python, regardless of what particular model or library is being used underneath.

```python
from patsy import ModelDesc
ModelDesc.from_formula("y ~ a + a:b + np.log(x)")

ModelDesc.from_formula("y ~ a*b + np.log(x) - 1")
```

Model matrix

```
from patsy import demo_data, dmatrix, dmatrices

data = demo_data("y", "a", "b", "x1", "x2")
pd.DataFrame(data)
```

Or you can simply create a dmatrix and return it as a DataFrame

```
dmatrix("a + a:b + np.exp(x1)", data, return_type='dataframe')
```

Design Info

One of the keep features of the design matrix object is that it retains all the necessary details (including stateful transforms) that are necessary to apply to new data inputs (e.g. for prediction).

```
d = dmatrix("a + a:b + np.exp(x1)",data, eturn_type='dataframe')
d.design_info
```

patsy: scikit-lego patsyTransformer

If you would like to use a `patsy` formula in a `scikitlearn` pipeline, it is possible via the `patsyTransformer` from the `scikit-lego` library.

```
from sklego.preprocessing import PatsyTransformer
df = pd.DataFrame({
"y": [2, 2, 4, 4, 6],
"x": [1, 2, 3, 4, 5],
"a": ["yes", "yes", "no", "no", "yes"]
})

X, y = df[["x", "a"]], df[["y"]].values
```

```
pt = PatsyTransformer("x*a + np.log(x)")
pt.fit_transform(X)

make_pipeline(pt, StandardScaler()).fit_transform(X)
```

pyMC3 + Arviz

> PyMC3 is a probabilistic programming package for Python that allows users to fit Bayesian models using a variety of numerical methods, most notably Markov chain Monte Carlo (MCMC) and variational inference (VI). Its flexibility and extensibility make it applicable to a large suite of problems. Along with core model specification and fitting functionality, PyMC3 includes functionality for summarizing output and for model diagnostics.

> ArviZ is a Python package for exploratory analysis of Bayesian models. Includes functions for posterior analysis, data storage, sample diagnostics, model checking, and comparison. The goal is to provide backend-agnostic tools for diagnostics and visualizations of Bayesian inference in Python, by first converting inference data into xarray objects.

```python
import pymc3  as pm
import arviz as az
```

pyMC3: Model basics

All models are derived from the `Model()` class, unlike what we have seen previously `pymc` makes heavy use of Python's context manager using the with statement to add model components to a model.

```
with pm.Model() as norm:
x = pm.Normal("x", mu=0, sigma=1)

x = pm.Normal("x", mu=0, sigma=1)
#### TypeError: No model on context stack....

with norm:
y = pm.Normal("y", mu=x, sigma=1, shape=3)
norm.vars
```

pyMC3: Beta-Binamial model

We will now build a basic model where we know what the solution should look like and compare the results.

```
with pm.Model() as beta_binom:
p = pm.Beta("p", alpha=10, beta=10)
x = pm.Binomial("x", n=20, p=p, observed=5)
```

In order to sample from the posterior we add a call to sample() within the model context.

```
with beta_binom:
trace = pm.sample(return_inferencedata=True, random_seed=1234)

##
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [p]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4
    _000 + 4_000 draws total) took 6 seconds.
```

pyMC3: inferenceData results

```
print(trace)
## Inference data with groups:
##      > posterior
##      > log_likelihood
##      > sample_stats
##      > observed_data

print(type(trace))
## <class 'arviz.data.inference_data.InferenceData'>
```

> **xarray: N-D labelled arrays and datasets in Python**
> xarray (formerly xray) is an open-source project and Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!
> ....

See here for more details on xarray + InferenceData

pyMC3: more...

- Posterio info

```
print(trace.posterior)## try and see the result
print(trace.posterior["p"].shape)
print(trace.sel(chain=0).posterior["p"].shape)
print(trace.sel(draw=slice(500, None, 10)).posterior["p"].
    shape)
```

- As DataFrame

  Posterior values, or subsets, can be converted to DataFrames via the `to_dataframe()` method

```
trace.posterior.to_dataframe()
trace.posterior["p"][0,:].to_dataframe()
```

- MultiTrace result

```
with beta_binom:
mt = pm.sample(random_seed=1234)
```

pyMC3: other useful diagnostics

Standard MCMC diagnostic statistics are available via summary() from ArviZ

```
az.summary(trace)
## mean sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk
    ess_tail  r_hat
## p  0.374  0.076   0.232    0.509      0.002    0.001
    1596.0    2654.0    1.0
```

individual methods are available for each statistic,

```
print(az.ess(trace, method="bulk"))
## <xarray.Dataset>
## Dimensi....

print(az.ess(trace, method="tail"))
## <xarray.Dataset>
## Dimensio....

print(az.rhat(trace))
print(az.mcse(trace))
```

Apach Arrow

> Apache Arrow is a software development platform for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.

> A critical component of Apache Arrow is its in-memory columnar format, a standardized, language-agnostic specification for representing structured, table-like datasets in-memory. This data format has a rich data type system (included nested and user-defined data types) designed to support the needs of analytic database systems, data frame libraries, and more.

```python
import pyarrow as pa
```

(Apache) Parquet

> … provides a standardized open-source columnar storage format for use in data analysis systems. It was created originally for use in Apache Hadoop with systems like Apache Drill, Apache Hive, Apache Impala, and Apache Spark, adopting it as a shared standard for high-performance data IO.

Core features:

> The values in each column are physically stored in contiguous memory locations, and this columnar storage provides the following benefits:
>
> - Column-wise compression is efficient and saves storage space
> - Compression techniques specific to a type can be applied as the column values tend to be of the same type
> - Queries that fetch specific column values need not read the entire row data thus, improving performance

(Apache) Parquet: feather...

> … is a portable file format for storing Arrow tables or data frames (from languages like Python or R) that utilizes the Arrow IPC format internally. Feather was created early in the Arrow project as a proof of concept for fast, language-agnostic data frame storage for Python (pandas) and R.

Core features:

- Different encoding techniques can be applied to different columns
- Direct columnar serialization of Arrow tables
- Supports all Arrow data types and compression
- Language agnostic
- Metadata makes it possible to read only the necessary columns for an operation

More materials and references

More materials and references

More materials

- Introduction to `pytorch`
- Fundamentals of AutoGrad
- Feedforward NN
- Convolutional NN
- pytorch and GPU
- SQL and Python

References

- Introduction to scikit-learn
- Statsmodels and patsy
- PyArrow - Apache Arrow Python bindings
- Introduction to Bayesian analysis in Python: PyMC and Arviz