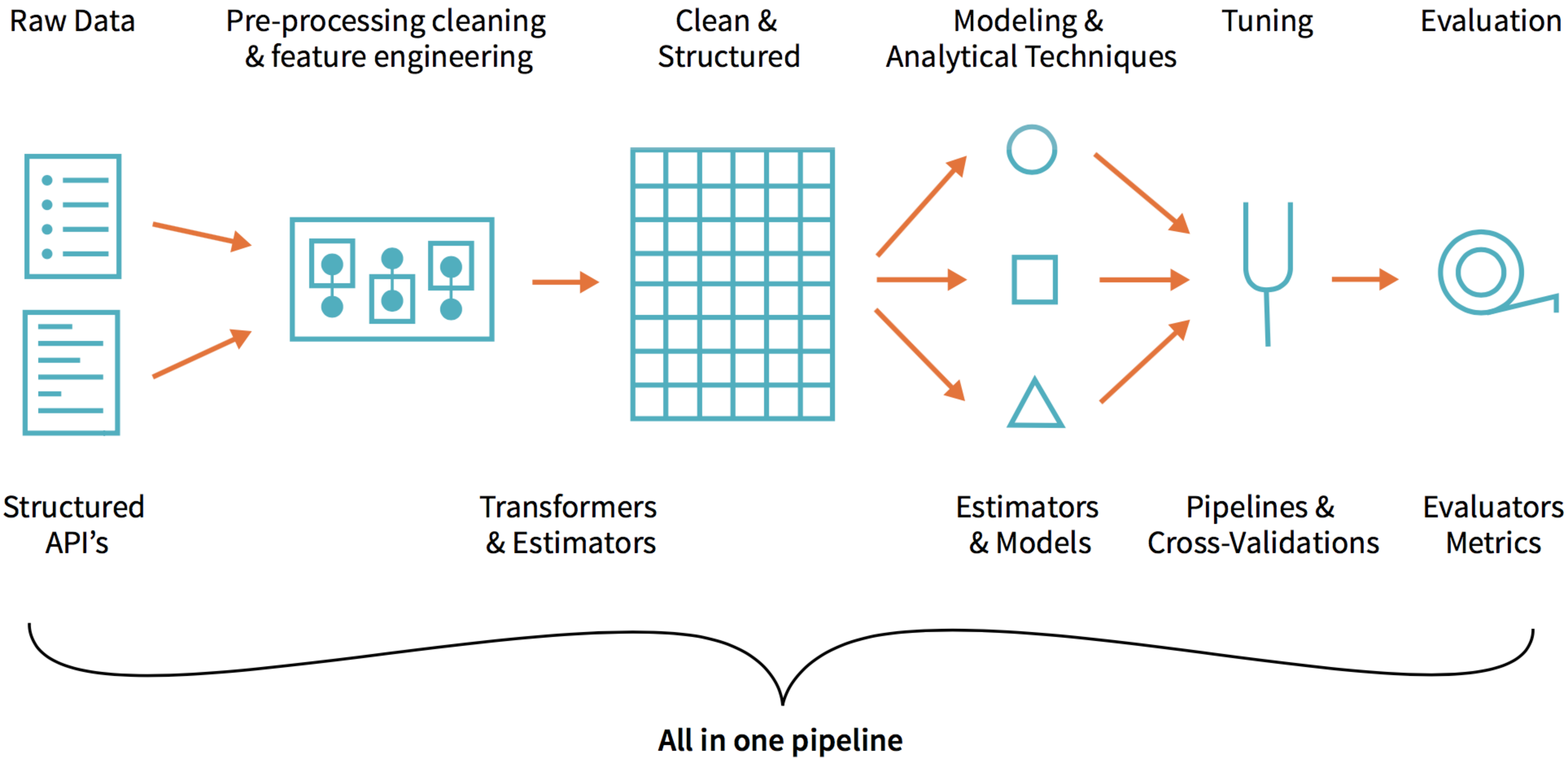
Lecture 6:
Modeling Libraries in Python

Section 6.1 Interfacing Between pandas and Model Code

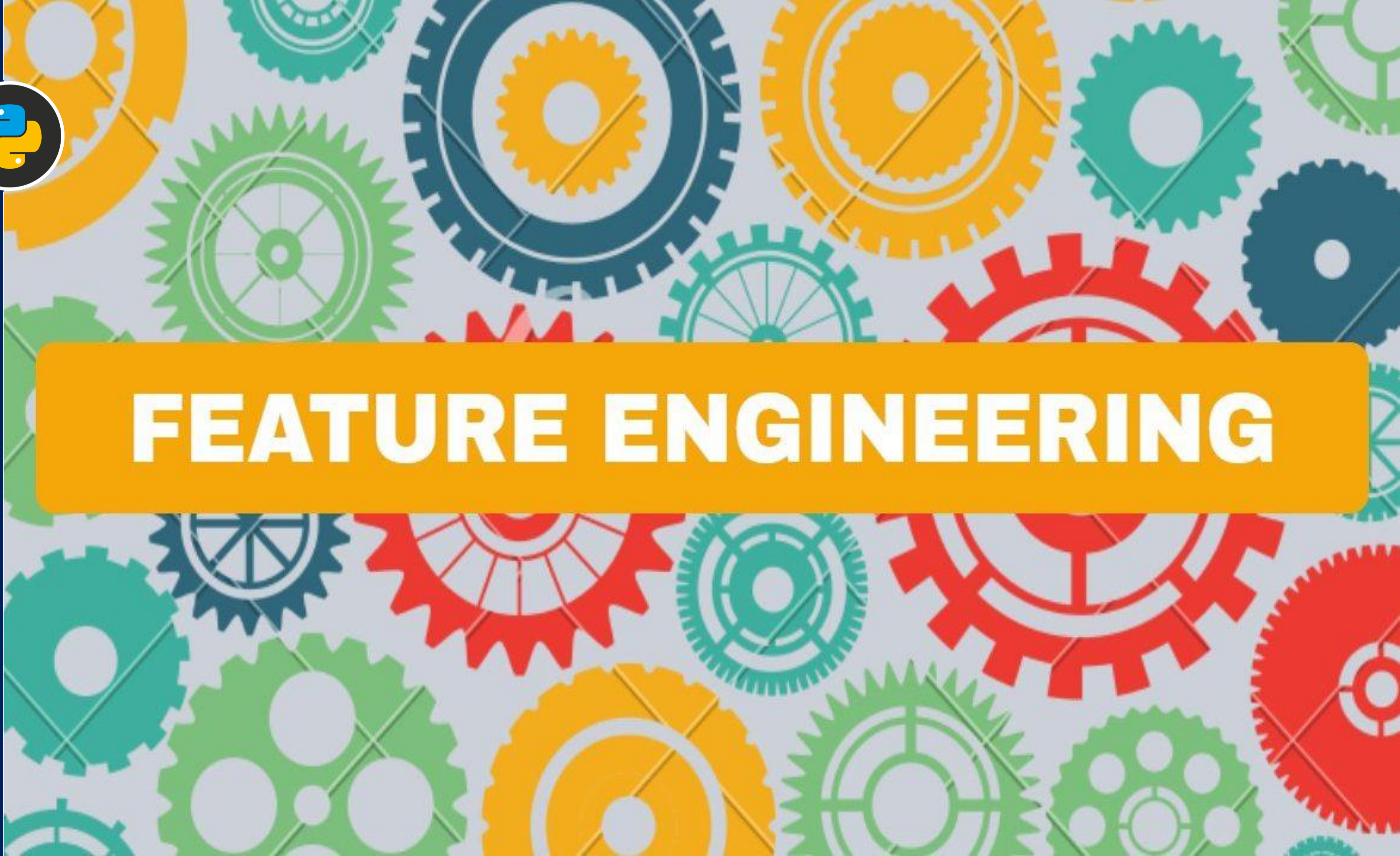
6.1 Interfacing Between pandas and Model Code

MAGIC CODE INSTITUTE
Leverage your tech skills

➢ Progress of a data science project:

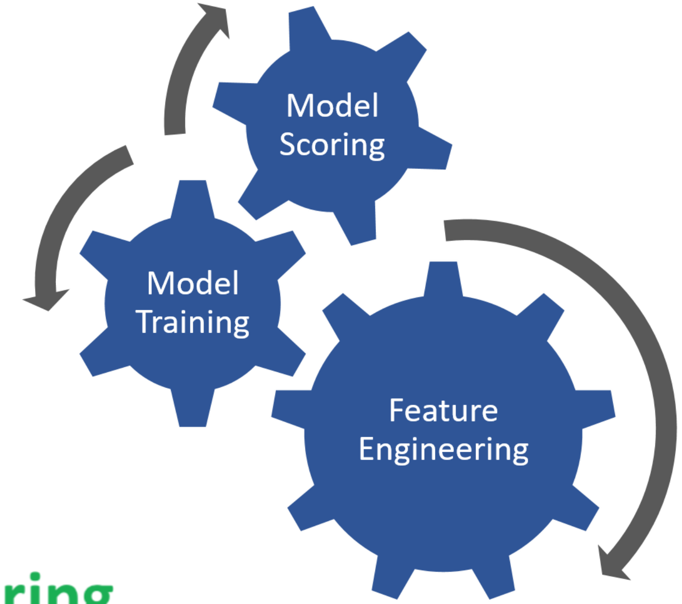


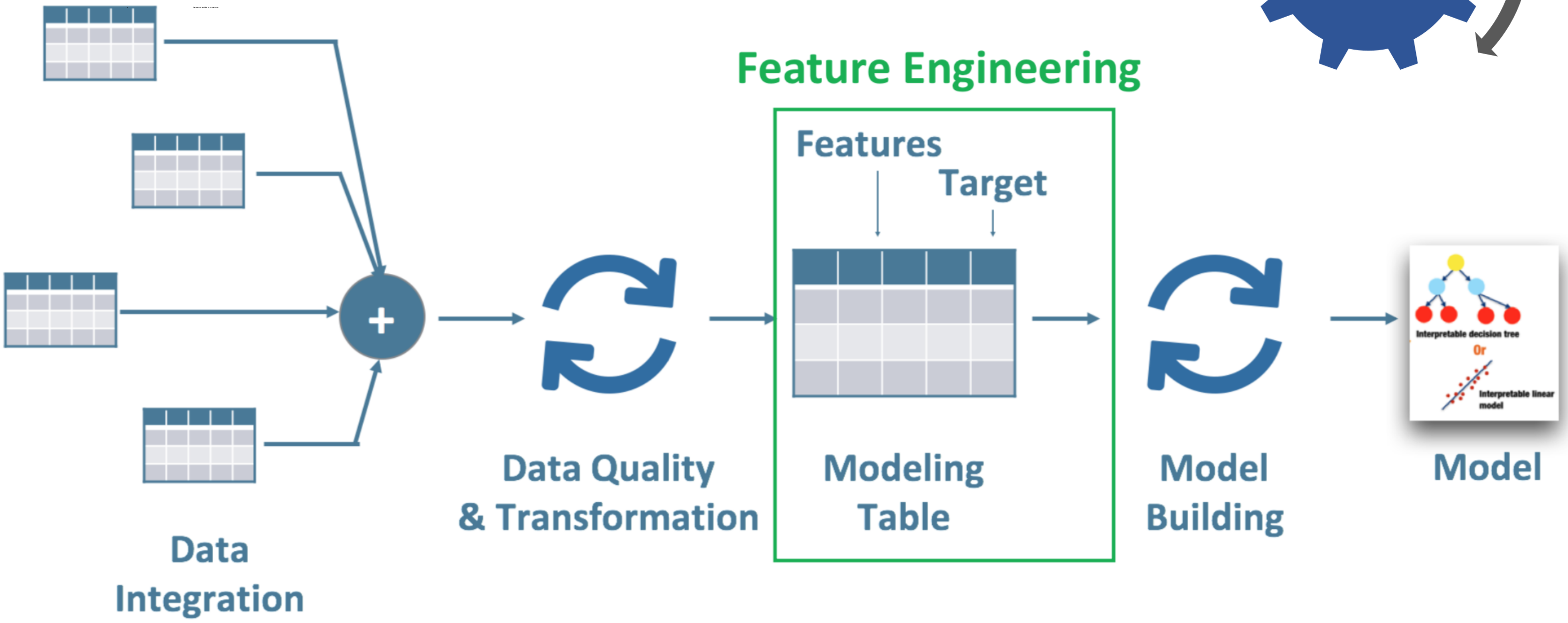
6.1 Interfacing Between pandas and Model Code



Why Feature Engineering is important:

- Data is initially in a raw form and not ready for analyzing / modeling purposes
- Feature engineering is the process of using domain knowledge of the data to create (hopefully) features
- Account for 80% - 90% of a data science project

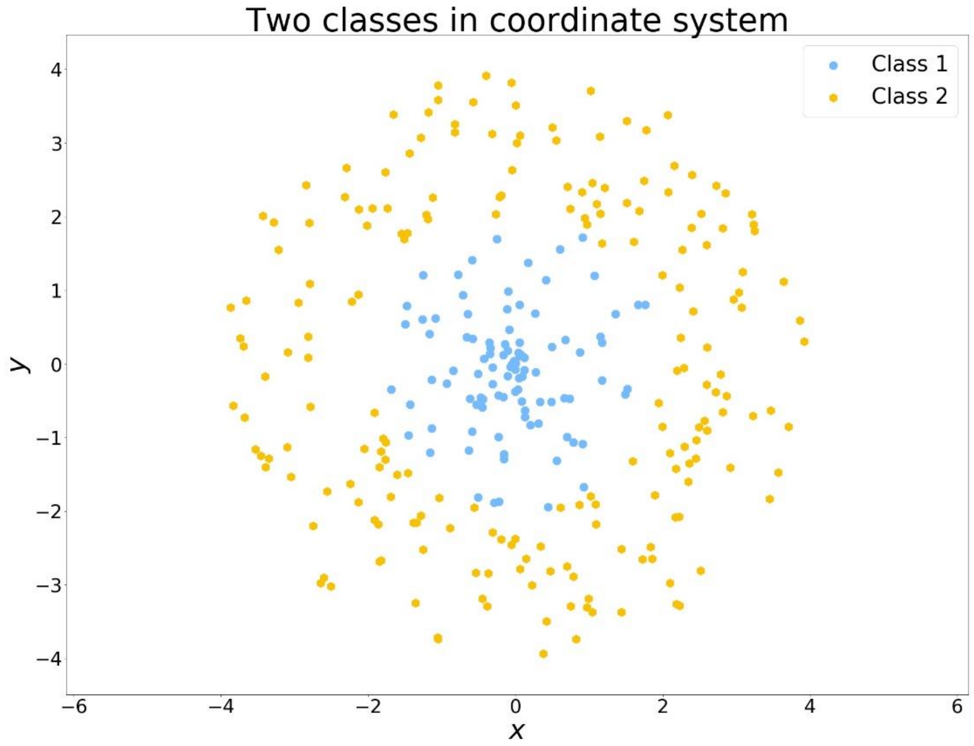


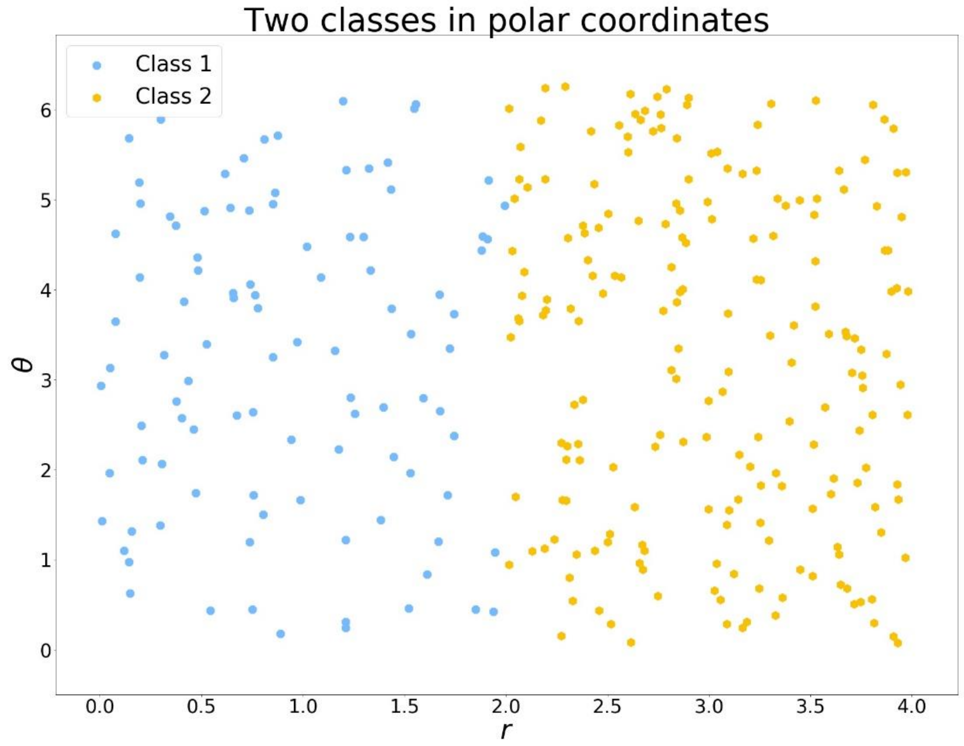


What is Feature Engineering?

- Even the raw dataset has features. Most of the time, the data will be in the form of a table.
- Each column is a feature. But these features may not produce the best results from the algorithm.
- Modifying, deleting and combining these features results in a new set that is more adept at training the algorithm.
- Feature engineering in machine learning is more than selecting the appropriate features and transforming them.
- Not only does feature engineering prepare the dataset to be compatible with the algorithm, but it also improves the performance of the machine learning models.

Example for the importance of Feature Engineering





- Original Data: non-linear boundary and not convenient for many machine learning algorithms
- Derived Data: linear boundary and will work with almost all machine learning algorithms

6.1 Interfacing Between pandas and Model Code

- A common workflow for model development:
  - use pandas for data loading and cleaning.
  - a modeling library to build the model.
- The point of contact between pandas & other analysis libraries:
  - NumPy arrays
- To turn a DataFrame into a NumPy array:
  - Use the .values property:
- To convert back to a DataFrame:
  - pass a two-dimensional ndarray with optional column names

```
df2 = pd.DataFrame(data.values,
                   columns=['one', 'two', 'three'])
df2
```

	one	two	three
0	1.0	0.01	-1.5
1	2.0	-0.01	0.0
2	3.0	0.25	3.6
3	4.0	-4.10	1.3
4	5.0	0.00	-2.0

```
data = pd.DataFrame(
        {'x0': [1, 2, 3, 4, 5],
         'x1': [0.01, -0.01, 0.25, -4.1, 0.],
         'y': [-1.5, 0., 3.6, 1.3, -2.]})
data
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

```
data.columns
```

```
Index(['x0', 'x1', 'y'], dtype='object')
```

```
data.values
```

```
array([[ 1.  ,  0.01, -1.5 ],
       [ 2.  , -0.01,  0.  ],
       [ 3.  ,  0.25,  3.6 ],
       [ 4.  , -4.1 ,  1.3 ],
       [ 5.  ,  0.  , -2.  ]])
```

MAGIC CODE INSTITUTE
Leverage your tech skills

Note:

- The `.values` attribute is intended to be used when your data is homogeneous - for example, all numeric types.
- If you have heterogeneous data, the result will be an `ndarray` of Python objects.

```
df3 = data.copy()
df3['strings'] = ['a', 'b', 'c', 'd', 'e']
df3
```

	x0	x1	y	strings
0	1	0.01	-1.5	a
1	2	-0.01	0.0	b
2	3	0.25	3.6	c
3	4	-4.10	1.3	d
4	5	0.00	-2.0	e

```
df3.values
```

```
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

- If you want to use a subset of the column

```
model_cols = ['x0', 'x1']
data.loc[:, model_cols].values
```

```
array([[ 1.  ,  0.01],
       [ 2.  , -0.01],
       [ 3.  ,  0.25],
       [ 4.  , -4.1 ],
       [ 5.  ,  0.  ]])
```

```
data[model_cols].values
```

```
array([[ 1.  ,  0.01],
       [ 2.  , -0.01],
       [ 3.  ,  0.25],
       [ 4.  , -4.1 ],
       [ 5.  ,  0.  ]])
```

```
data['category'] = pd.Categorical(
                    ['a', 'b', 'a', 'a', 'b'],
                    categories=['a', 'b'])
data
```

	x0	x1	y	category
0	1	0.01	-1.5	a
1	2	-0.01	0.0	b
2	3	0.25	3.6	a
3	4	-4.10	1.3	a
4	5	0.00	-2.0	b

Dealing with Categorical data

- If we wanted to replace the 'category' column with dummy variables:
  - create dummy variables
  - drop the 'category' column
  - join the result

```
dummies = pd.get_dummies(data.category,
                         prefix='category')
dummies
```

	category_a	category_b
0	1	0
1	0	1
2	1	0
3	1	0
4	0	1

```
data_with_dummies = data.drop('category',
                              axis=1).join(dummies)
data_with_dummies
```

	x0	x1	y	category_a	category_b
0	1	0.01	-1.5	1	0
1	2	-0.01	0.0	0	1
2	3	0.25	3.6	1	0
3	4	-4.10	1.3	1	0
4	5	0.00	-2.0	0	1

Section 6.2 Creating Model Descriptions with Patsy

Patsy:

- A Python library for describing statistical models (especially linear models) with a small string-based "formula syntax".
- well supported for specifying linear models in `statsmodels`
  - `"y ~ x0 + x1"`
  - `patsy.dmatrices` function takes a formula string along with a dataset (a `DataFrame` or a `dict` of arrays) and produces design matrices for a linear model

```
data = pd.DataFrame({
        'x0': [1, 2, 3, 4, 5],
        'x1': [0.01, -0.01, 0.25, -4.1, 0.],
        'y': [-1.5, 0., 3.6, 1.3, -2.]})
data
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

```
import patsy
y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

```
X
```

```
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

```
y
```

```
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)
```

MAGIC CODE INSTITUTE
Leverage your tech skills

- These Patsy `DesignMatrix` instances are `NumPy ndarrays` with additional metadata

```
np.asarray(X)
```

```
array([[ 1.  ,  1.  ,  0.01],
       [ 1.  ,  2.  , -0.01],
       [ 1.  ,  3.  ,  0.25],
       [ 1.  ,  4.  , -4.1 ],
       [ 1.  ,  5.  ,  0.  ]])
```

```
np.asarray(y)
```

```
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])
```

```
X.design_info.column_names
```

```
['Intercept', 'x0', 'x1']
```

```
y.design_info.column_names
```

```
['y']
```

- The `Intercept` term is a convention for linear models like ordinary least squares (OLS) regression.
- You can suppress the intercept by adding the term `"+ 0"` to the model

```
patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
# X here
```

```
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

```
np.linalg.lstsq(X, y)
```

```
(array([[ 0.31290976],
        [-0.07910564],
        [-0.26546384]]),
 array([19.63791494]),
 3,
 array([8.03737688, 3.38335321, 0.90895207]))
```

- The Patsy objects can be passed directly into algorithms

```
coef, resid, _, _ = np.linalg.lstsq(X, y)
```

```
coef
```

```
array([[ 0.31290976],
       [-0.07910564],
       [-0.26546384]])
```

- Get model coefficients

```
coef.squeeze()
```

```
array([ 0.31290976, -0.07910564, -0.26546384])
```

```
coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)
coef
```

```
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

- reattach the model column names to the fitted coefficients to obtain a Series

- Data Transformations in Patsy Formulas
  - Patsy will try to find the functions you use in the enclosing scope

```
y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)
X
```

```
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

- Some commonly used variable transformations include:
  - Standardizing (to mean 0 and variance 1):
  - Centering (subtracting the mean).

```
y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)
X
```

```
DesignMatrix with shape (5, 3)
  Intercept  standardize(x0)  center(x1)
          1         -1.41421        0.78
          1         -0.70711        0.76
          1          0.00000        1.02
          1          0.70711       -3.33
          1          1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

- Try some other numpy functions

FUNCTION	DESCRIPTION
expm1()	Calculate exp(x) – 1 for all elements in the array.
exp2()	Calculate 2**p for all p in the input array.
log10()	Return the base 10 logarithm of the input array, element-wise.
log2()	Base-2 logarithm of x.
log1p()	Return the natural logarithm of one plus the input array, element-wise.
logaddexp()	Logarithm of the sum of exponentiations of the inputs.
logad-dexp2()	Logarithm of the sum of exponentiations of the inputs in base-2.

FUNCTION	DESCRIPTION
tan()	Compute tangent element-wise.
arcsin()	Inverse sine, element-wise.
arccos()	Trigonometric inverse cosine, element-wise.
arctan()	Trigonometric inverse tangent, element-wise.
arctan2()	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
degrees()	Convert angles from radians to degrees.
rad2deg()	Convert angles from radians to degrees.
deg2rad	Convert angles from degrees to radians.
radians()	Convert angles from degrees to radians.
hypot()	Given the “legs” of a right triangle, return its hypotenuse.
unwrap()	Unwrap by changing deltas between values to 2*pi com-plement.

`patsy.center(x)`

A stateful transform that centers input data, i.e., subtracts the mean.

If input has multiple columns, centers each column separately.

Equivalent to `standardize(x, rescale=False)`

`patsy.standardize(x, center=True, rescale=True, ddof=0)`

A stateful transform that standardizes input data, i.e. it subtracts the mean and divides by the sample standard deviation.

Either centering or rescaling or both can be disabled by use of keyword arguments. The *ddof* argument controls the delta degrees of freedom when computing the standard deviation (cf. `numpy.std()`). The default of `ddof=0` produces the maximum likelihood estimate; use `ddof=1` if you prefer the square root of the unbiased estimate of the variance.

If input has multiple columns, standardizes each column separately.

6.3 Introduction to statsmodels

- Import modules
- dnorm: helper function to generate random numbers following normal distribution
- Y is constructed as:
  - Y = X * beta + eps (noise)

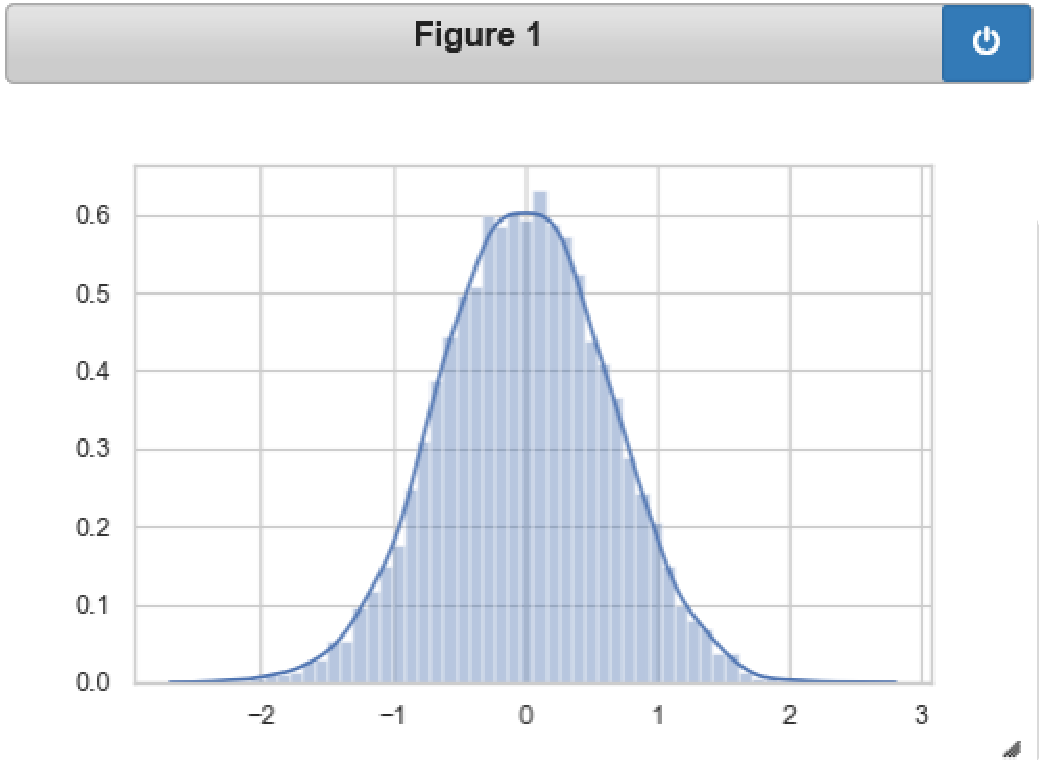
```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)

# For reproducibility
np.random.seed(12345)

N = 10000
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]
y = np.dot(X, beta) + eps
```

```
fig = plt.figure()
sns.distplot(X[:,0])
```

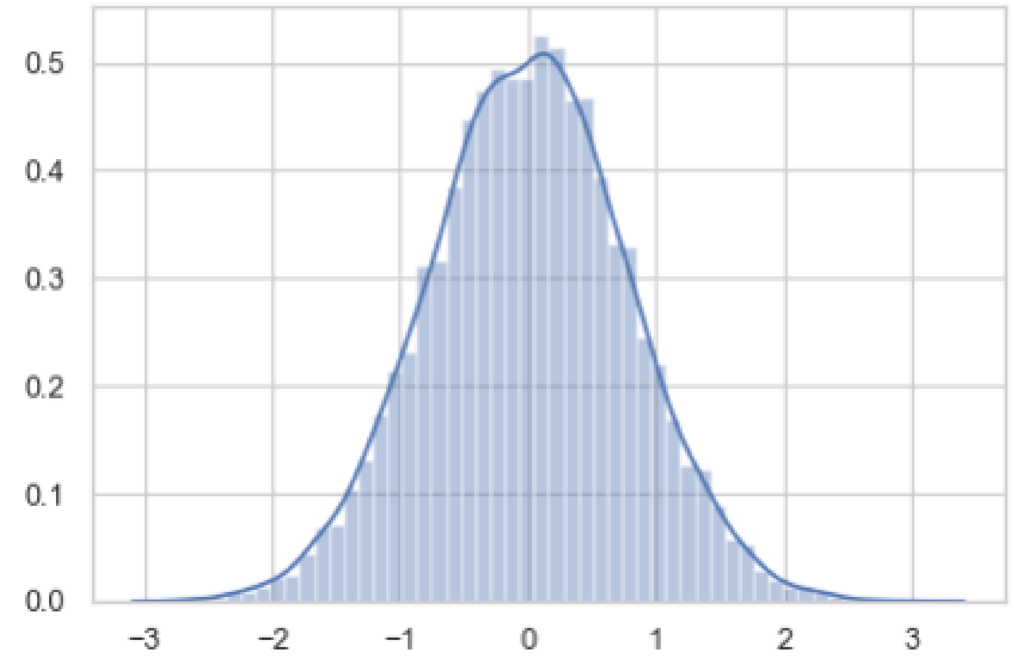


- Normally distributed with:
  - mean = 0
  - Variance = 0.4

- Normally distributed with:
  - mean = 0
  - Variance = 0.6 (0.2)

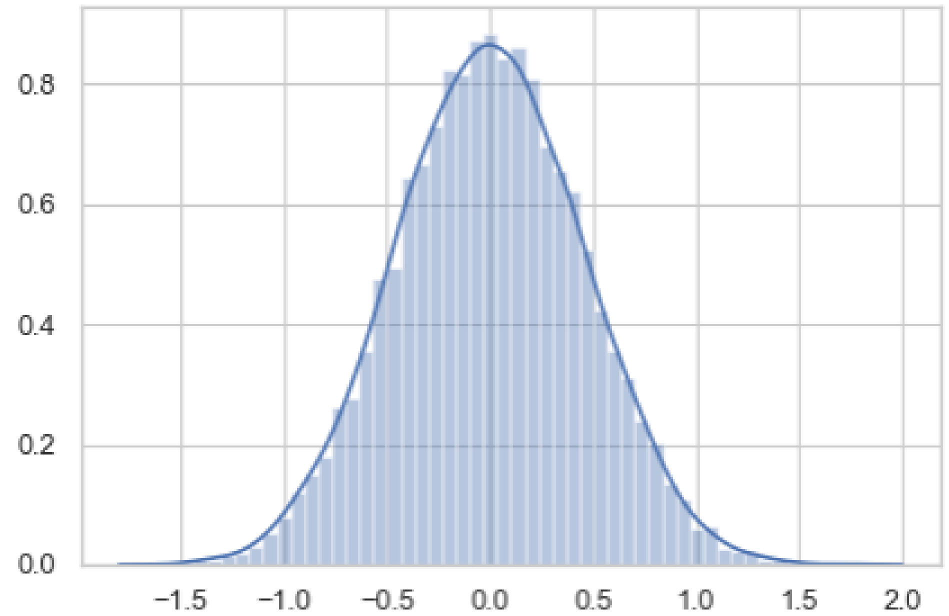
```
fig = plt.figure()
sns.distplot(X[:,1])
```

Figure 2




```
fig = plt.figure()
sns.distplot(X[:,2])
```

Figure 3




```
model = sm.OLS(y, X)
results = model.fit()
results.params
```

```
array([0.09166321, 0.30220827, 0.50473953])
```

```
print(results.summary())
```

```
                            OLS Regression Results
=======================================================================================
Dep. Variable:                      y   R-squared (uncentered):                   0.521
Model:                            OLS   Adj. R-squared (uncentered):              0.521
Method:                 Least Squares   F-statistic:                              3629.
Date:                Sat, 04 Apr 2020   Prob (F-statistic):                        0.00
Time:                        16:40:52   Log-Likelihood:                         -2708.3
No. Observations:               10000   AIC:                                      5423.
Df Residuals:                    9997   BIC:                                      5444.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.0917      0.005     18.242      0.000       0.082       0.102
x2             0.3022      0.004     73.569      0.000       0.294       0.310
x3             0.5047      0.007     71.712      0.000       0.491       0.519
==============================================================================
Omnibus:                        3.283   Durbin-Watson:                   1.979
Prob(Omnibus):                  0.194   Jarque-Bera (JB):                3.314
Skew:                          -0.041   Prob(JB):                        0.191
Kurtosis:                       2.966   Cond. No.                         1.71
==============================================================================
```

```
results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
results.params
```

```
Intercept    0.000858
col0         0.091678
col1         0.302207
col2         0.504745
dtype: float64
```

```
data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
data['y'] = y
data
```

	col0	col1	col2	y
0	-0.129468	1.493446	-0.258437	0.142901
1	0.302910	-0.895914	0.375710	-0.329110
2	-0.328522	-0.302596	0.313175	0.288018
3	-0.351475	0.310462	0.041303	-0.251139
4	1.243269	-0.677377	0.509666	0.127453
...	...	...	...	...
9995	-0.545716	0.280965	-0.707475	-0.295103
9996	1.361230	2.058392	-0.622691	0.461859
9997	-0.004241	1.140471	-0.331893	0.422961
9998	-0.768258	0.800278	-0.419750	-0.061137
9999	0.414251	2.285421	0.017529	0.491187

10000 rows × 4 columns

```python
data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
data['y'] = y
data
```

	col0	col1	col2	y
0	-0.129468	1.493446	-0.258437	0.142901
1	0.302910	-0.895914	0.375710	-0.329110
2	-0.328522	-0.302596	0.313175	0.288018
3	-0.351475	0.310462	0.041303	-0.251139
4	1.243269	-0.677377	0.509666	0.127453
...	...	...	...	...
9995	-0.545716	0.280965	-0.707475	-0.295103
9996	1.361230	2.058392	-0.622691	0.461859
9997	-0.004241	1.140471	-0.331893	0.422961
9998	-0.768258	0.800278	-0.419750	-0.061137
9999	0.414251	2.285421	0.017529	0.491187

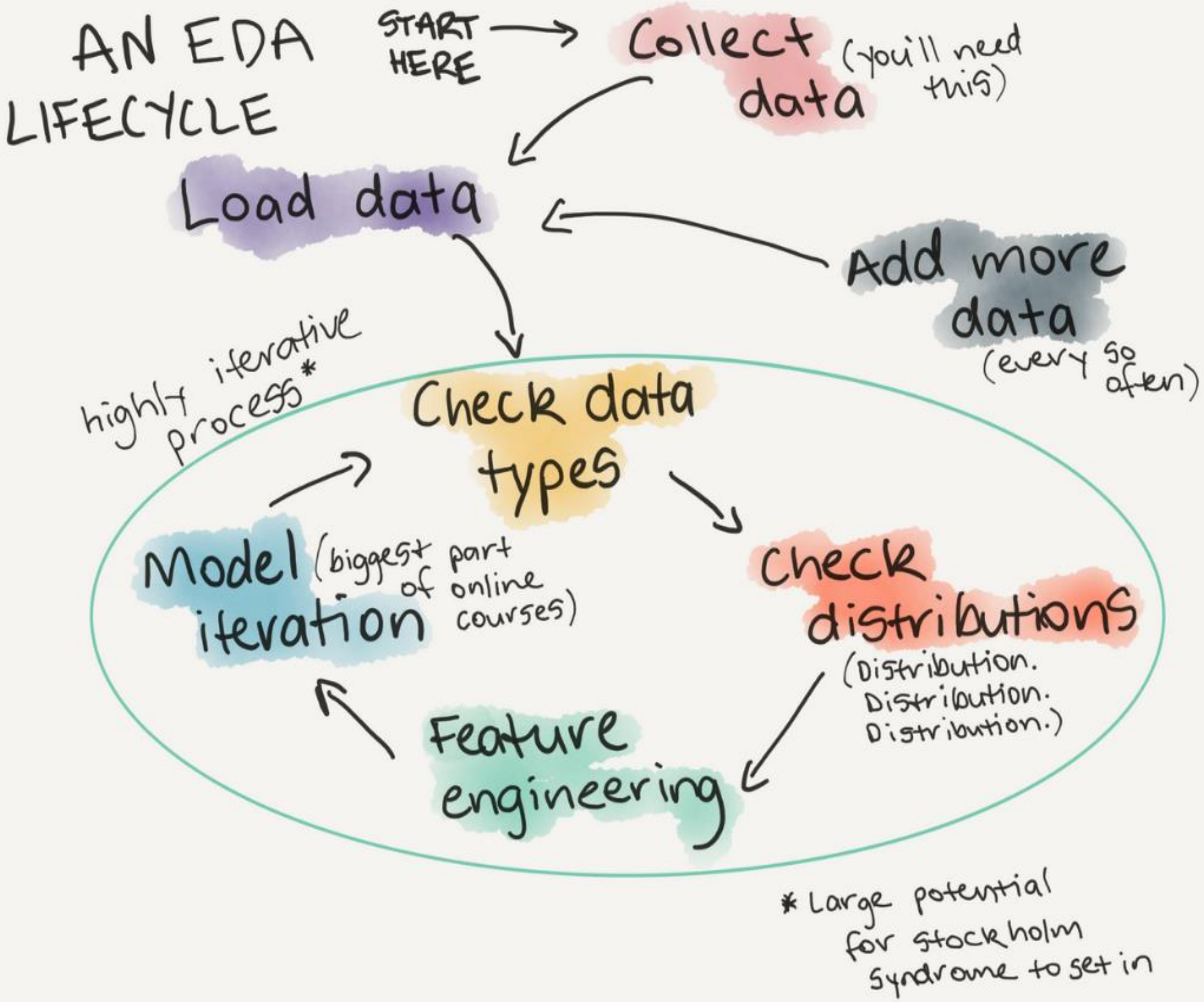
10000 rows × 4 columns

```python
results = smf.ols('y ~ col0 + col1 + col2',
                  data=data).fit()
results.params
```

```
Intercept    0.000858
col0         0.091678
col1         0.302207
col2         0.504745
dtype: float64
```

- Suppose instead that all of the model parameters are in a `DataFrame`
- use the `statsmodels` formula API and `Patsy` formula strings

THANKS FOR LISTENING!!!


AN EDA LIFECYCLE
START HERE
Collect data
(you'll need this)
Load data
Add more data
(every so often)
highly iterative process*
Check data types
Check distributions
(Distribution. Distribution. Distribution.)
Feature engineering
Model iteration
(biggest part of online courses)
* Large potential for Stockholm Syndrome to set in