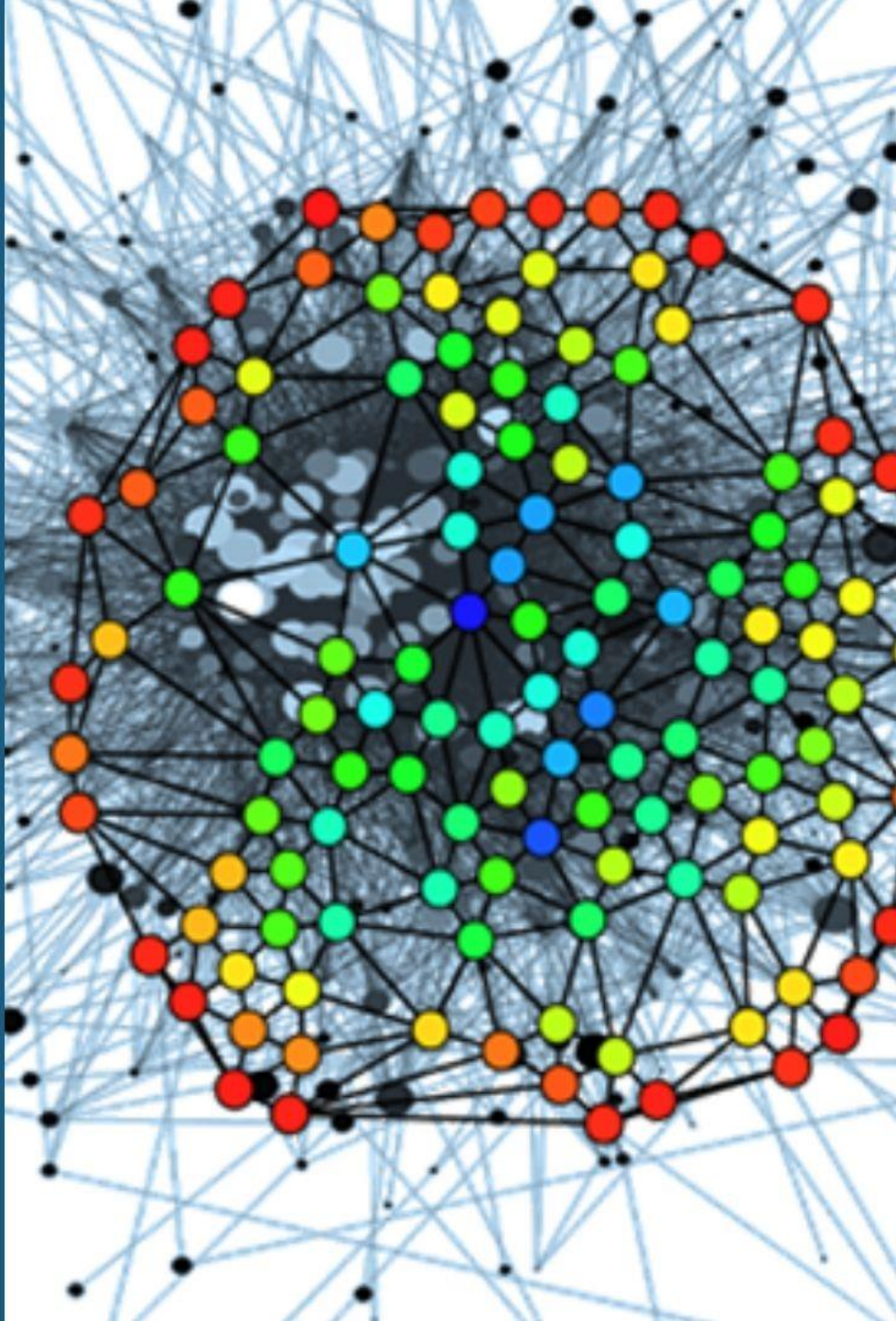
MACHINE LEARNING & IMAGE PROCESSING

WEEK 06 - PART I

FEATURE ENGINEERING

Thakshila Dasun
BSc. Hons in Mechanical Engineering
(Mechatronics Specialization)
CIMA, UK
Academy of Innovative Education

AIE
ACADEMY OF INNOVATIVE EDUCATION



Topics Covered

- Features for representing *categorical data*
- Features for representing *text*
- Features for representing *images*.
- *Derived features* for increasing model complexity
- *Imputation* of missing data (*vectorization*)

Categorical Features (1)

- One common type of non-numerical data is *categorical* data.

```
data = [
    {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
    {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
    {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
    {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
]
```

- You might be tempted to encode this data with a straightforward numerical mapping:

```
{'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};
```

Categorical Features (2)

- It turns out that this is not generally a useful approach in Scikit-Learn: the package's models make the fundamental assumption that numerical features reflect algebraic quantities.
- Thus such a mapping would imply, for example, that *Queen Anne* < *Fremont* < *Wallingford*, or even that *Wallingford - Queen Anne = Fremont*, which does not make much sense.
- In this case, one proven technique is to use one-hot encoding, which effectively creates extra columns indicating the presence or absence of a category with a value of 1 or 0, respectively.
- When your data comes as a list of dictionaries, Scikit-Learn's DictVectorizer will do this for you:

Categorical Features (3)

- Notice that the 'neighborhood' column has been expanded into three separate columns.
- Representing the three neighborhood labels, and that each row has a 1 in the column associated with its neighborhood

```
from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer(sparse=False, dtype=int)
vec.fit_transform(data)
```

```
array([[      0,       1,       0, 850000,       4],
       [      1,       0,       0, 700000,       3],
       [      0,       0,       1, 650000,       3],
       [      1,       0,       0, 600000,       2]], dtype=int64)
```

Categorical Features (3)

- There is one clear disadvantage of this approach: if your category has many possible values, this can *greatly* increase the size of your dataset.
- However, because the encoded data contains mostly zeros, a SPARSE output can be a very efficient solution
- Many (though not yet all) of the Scikit-Learn estimators accept such sparse inputs when fitting and evaluating models.

Text Features(1)

- Another common need in feature engineering is to convert text to a set of representative numerical values.
- For example, most automatic mining of social media data relies on some form of encoding the text as numbers.
- One of the simplest methods of encoding data is by *word counts*: you take each snippet of text, count the occurrences of each word within it, and put the results in a table.

```
sample = ['problem of evil',
          'evil queen',
          'horizon problem']
```

Text Features(2)

- For a vectorization of this data based on word count, we could construct a column representing the word "problem," the word "evil," the word "horizon," and so on.
- While doing this by hand would be possible, the tedium can be avoided by using Scikit-Learn's CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(sample)
X
```

Text Features(3)

- The result is a sparse matrix recording the number of times each word appears

	evil	horizon	of	problem	queen
0	1	0	1	1	0
1	1	0	0	0	1
2	0	1	0	1	0

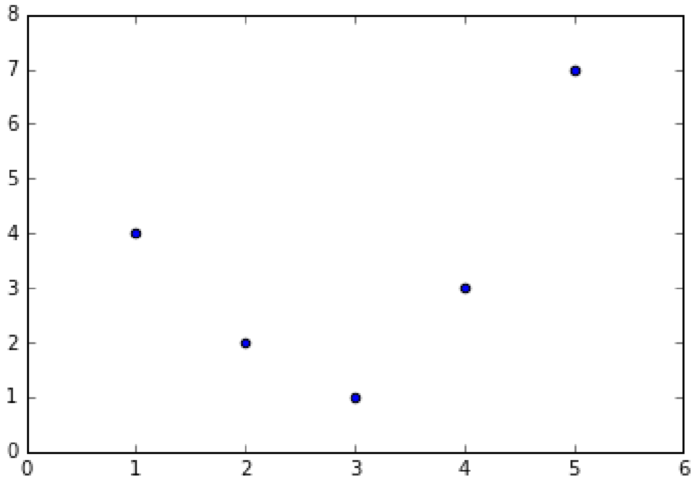
Derived Features (1)

- Another useful type of feature is one that is mathematically derived from some input features.
- when we constructed POLYNOMIAL FEATURES from our input data, We saw that we could convert a linear regression into a POLYNOMIAL REGRESSION not by changing the model, but by transforming the input!
- This is sometimes known as *basis function regression*

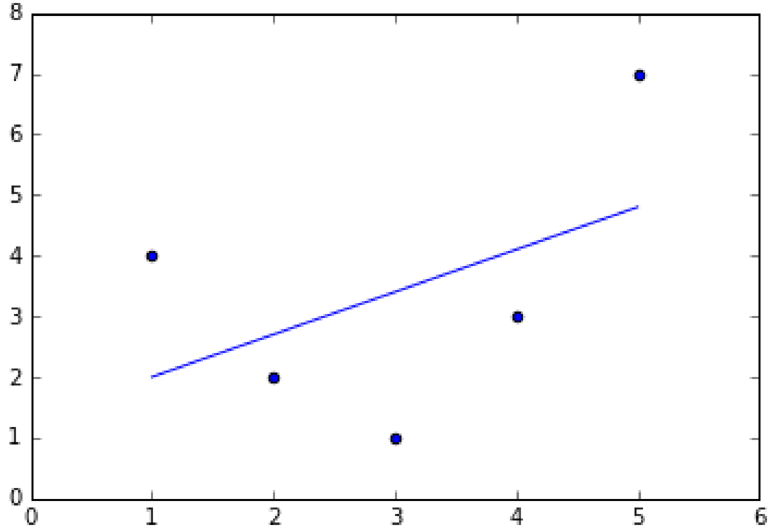
Derived Features (2)

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
```




```
from sklearn.linear_model import LinearRegression
X = x[:, np.newaxis]
model = LinearRegression().fit(X, y)
yfit = model.predict(X)
plt.scatter(x, y)
plt.plot(x, yfit);
```



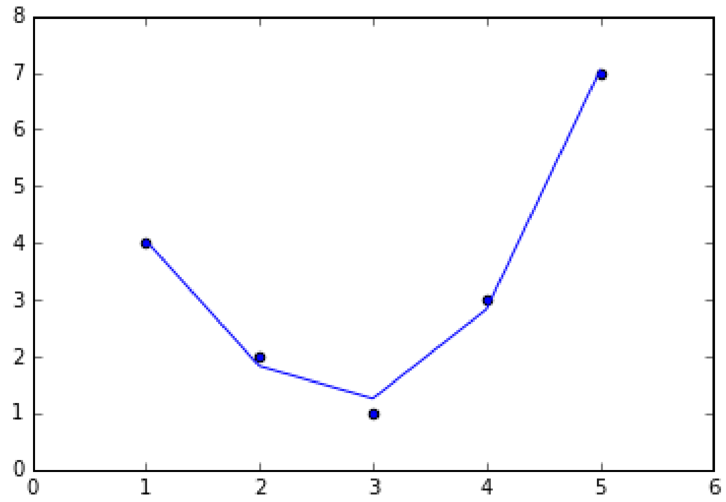
Derived Features (3)

- It's clear that we need a more sophisticated model to describe the relationship between x and y.
- One approach to this is to transform the data, adding extra columns of features to drive more flexibility in the model.
- For example, we can add polynomial features to the data this way:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
X2 = poly.fit_transform(X)
print(X2)
```

```
[[   1.    1.    1.]
 [   2.    4.    8.]
 [   3.    9.   27.]
 [   4.   16.   64.]
 [   5.   25.  125.]]
```

```
model = LinearRegression().fit(X2, y)
yfit = model.predict(X2)
plt.scatter(x, y)
plt.plot(x, yfit);
```



Derived Features (4)

- The derived feature matrix has one column representing x, and a second column representing x2, and a third column representing x3. Computing a linear regression on this expanded input gives a much closer fit to our data:
- This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods

Imputation of Missing Data (1)

- Another common need in feature engineering is handling of missing data.
- in Handling Missing Data, and saw that often the NaN value is used to mark missing values.
- For example, we might have a dataset that looks like this:

```
from numpy import nan
X = np.array([[ nan, 0,   3  ],
              [ 3,   7,   9  ],
              [ 3,   5,   2  ],
              [ 4,   nan, 6  ],
              [ 8,   8,   1  ]])
y = np.array([14, 16, -1,  8, -5])
```

Imputation of Missing Data (2)

- When applying a typical machine learning model to such data, we will need to first replace such missing data with some appropriate fill value.
- This is known as *imputation* of missing values, and strategies range from simple (e.g., replacing missing values with the mean of the column) to sophisticated (e.g., using matrix completion or a robust model to handle such data).
- The sophisticated approaches tend to be very application-specific, and we won't dive into them here. For a baseline imputation approach, using the mean, median, or most frequent value