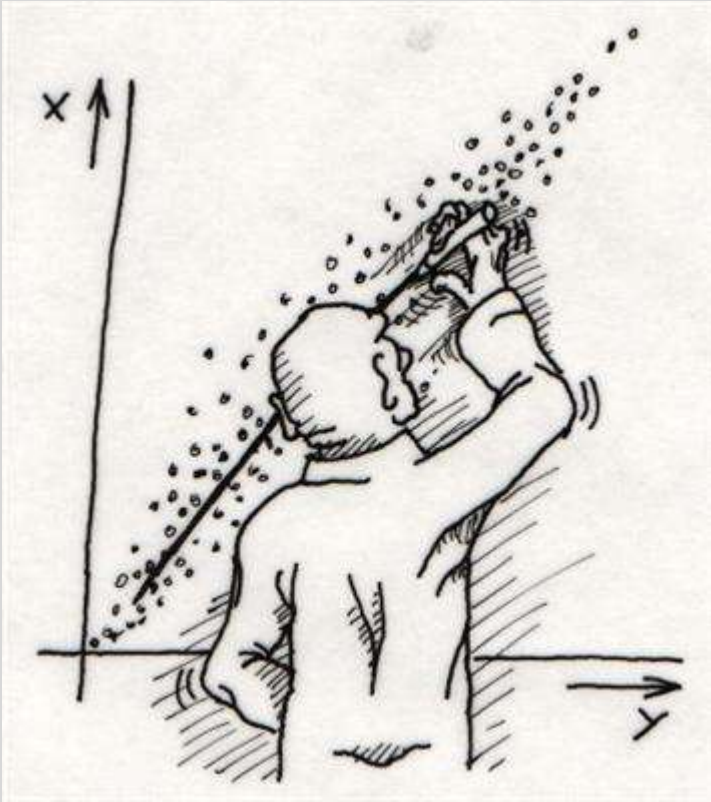


DigitalHouse > Coding School

DATA SCIENCE

MÓDULO 3

Feature Engineering

Agosto de 2017

Pensando en Model Validation

En las presentaciones anteriores estudiamos las ideas fundamentales de machine learning, pero todos los ejemplos asumieron que teníamos datos numéricos en un formato [n_samples, m_features].

En aplicaciones reales, los datos rara vez vienen ordenados de esa manera. Con esto en mente, uno de los pasos más importantes para usar machine learning en la práctica es la ingeniería de features o ***feature engineering:***

> **Esto es, tomar cualquier información que tengas sobre el problema a resolver y convertirlo en números que puedas usar para construir tu matriz de features.**

Normalmente este proceso es conocido como *vectorización*, ya que involucra la idea de convertir datos arbitrarios en **vectores bien formados.**

En esta sección vamos a estudiar algunas tareas comunes de feature engineering:

features para representar datos categóricos,

features para representar texto

features para representar imágenes

Adicionalmente vamos a discutir estos temas:

features derivadas para incrementar la complejidad del modelo

imputación de datos faltantes (missing data).

Recuerdan la definición de datos categóricos?

Imaginemos que estamos explorando ciertos datos de precios de propiedades y, junto con las features numéricas como “precio” y “habitaciones”, también tenés información sobre el “barrio”.

Por ejemplo, tus datos podrían verse así:

In [1]:

```
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'}
 ]
```

Features Categóricas

Usemos la técnica *one-hot encoding*, que crea columnas extra indicando la presencia o ausencia de una categoría con un valor de 1 o 0, respectivamente.

Cuando tus datos vienen como una lista de diccionarios, la clase **DictVectorizer** puede hacer esto automáticamente:

In [3]:

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

Out[3]:

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

Features Categóricas

Para ver el significado de cada columna, podemos inspeccionar los nombres de las features:

In [4]:

```
vec.get_feature_names()
```

Out[4]:

```
['neighborhood=Fremont',
  'neighborhood=Queen Anne',
  'neighborhood=Wallingford',
  'price',
  'rooms']
```

Con estas features categóricas codificadas de esta manera, podemos proceder como de costumbre para ajustar un modelo con Scikit-Learn

Hay una clara desventaja en esta aproximación: si tu categoría tiene muchos valores posibles, esto puede incrementar en gran medida el tamaño de tu dataset. Sin embargo, como los datos codificados contienen principalmente valores ceros, una **representación dispersa** podría ser una solución eficiente:

In [5]:

```
vec = DictVectorizer(sparse=True, dtype=int)
 vec.fit_transform(data)
```

Out[5]:

```
<4x5 sparse matrix of type '<class 'numpy.int64'>'
 with 12 stored elements in Compressed Sparse Row format>
```

Muchos (aunque no todos) los estimadores de Scikit-Learn aceptan representaciones dispersas cuando se ajustan y evalúan modelos.

sklearn.preprocessing.OneHotEncoder y **sklearn.feature_extraction.FeatureHasher** son dos funcionalidades adicionales que Scikit-Learn incluye para soportar este tipo de codificación.

Otra necesidad común en feature engineering es convertir texto a un conjunto de valores numéricos representativos.

Por ejemplo, la mayoría del mining automático de datos de redes sociales se basa en alguna forma de codificación del texto como números. Uno de los métodos más simples es codificar los datos por medio del conteo de palabras (*word counts):* tomás cada fragmento de texto, contás las ocurrencias de cada palabra en él y ponés los resultados en una tabla.

Por ejemplo, consideremos el siguiente dataset de tres frases:

In [6]:

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

Para vectorizar este dataset basado en el conteo de palabras, podríamos construir una columna representado cada palabra: “problem”, “evil”, “horizon“,... etc.

Para esto usamos CountVectorizer:

In [7]:

```
from sklearn.feature_extraction.text import CountVectorizer

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

Out[7]:

```
<3x5 sparse matrix of type '<class 'numpy.int64'>'
with 7 stored elements in Compressed Sparse Row format>
```

NOTA: Term frequency-inverse document frequency (TF–IDF) es una **técnica alternativa** que funciona mejor con ciertos algoritmos de clasificación.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

El resultado es una matriz dispersa que almacena la frecuencia de aparición de cada palabra; es fácil inspeccionar esta matriz si la convertimos a un DataFrame con columnas etiquetadas:

In [8]:

```
import pandas as pd
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

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

Otra necesidad común es codificar imágenes apropiadamente para su análisis usando machine learning.

La aproximación más simple es la que mostramos en la práctica guiada de procesamiento de dígitos, en la Introducción a Machine Learning: simplemente usamos los valores de los píxeles. Pero hay aproximaciones que pueden resultar más efectivas en otros contextos.

Un sumario exhaustivo de técnicas de extracción de features para imágenes está fuera del alcance de esta clase, pero hay excelente implementaciones de los métodos estándares en el proyecto Scikit-Image.

Veremos un pipeline completo en el último módulo.

Otro tipo útil de feature es aquel que es matemáticamente derivado de otros features de entrada.

Veremos más adelante este tema en detalle, pero estudiemos ahora cómo construir features polinómicos de nuestros datos de entrada.

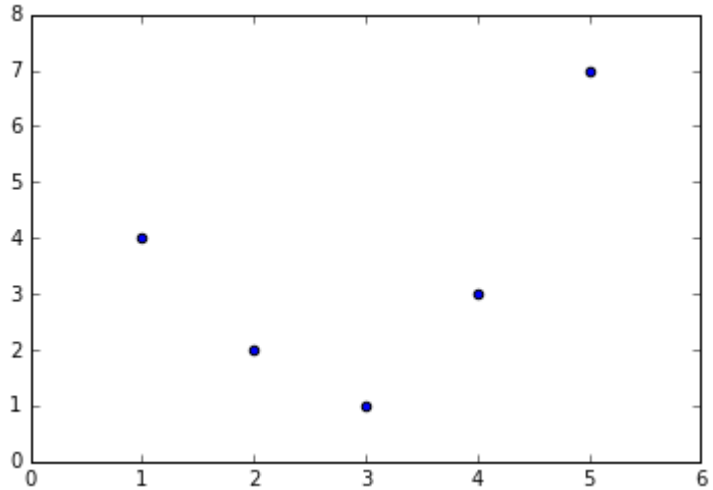
Podemos convertir una **regresión lineal** en una **regresión polinómica**, no cambiando el modelo pero tranformando la entrada!

Estos es conocido a veces como *basis function regression*,

Por ejemplo, este dataset claramente no puede ser descrito correctamente por una línea recta:

In [10]:

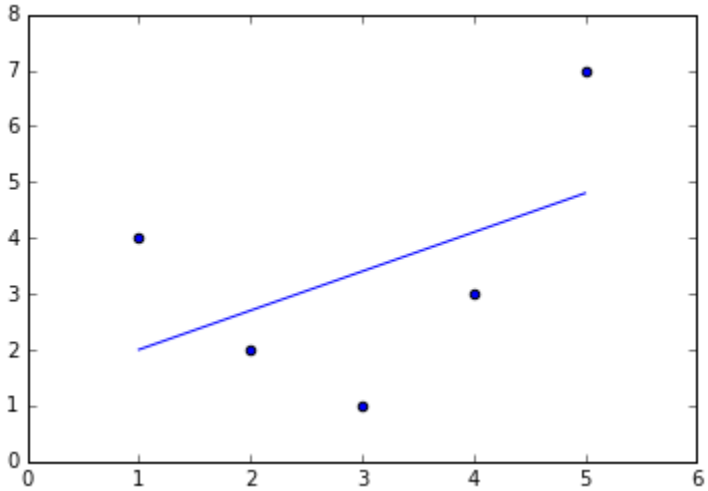
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

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



Sin embargo, podemos ajustar una línea a los datos usando LinearRegression y obtener un resultado óptimo:

In [11]:

```
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);
```



Claramente, necesitamos “un modelo más sofisticado” para describir la relación entre x e y.

Una aproximación a esto es transformar los datos, agregando columnas extra de features para agregar más flexibilidad al modelo. Por ejemplo, podemos agregar features polinómicas a los datos de esta forma:

In [12]:

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

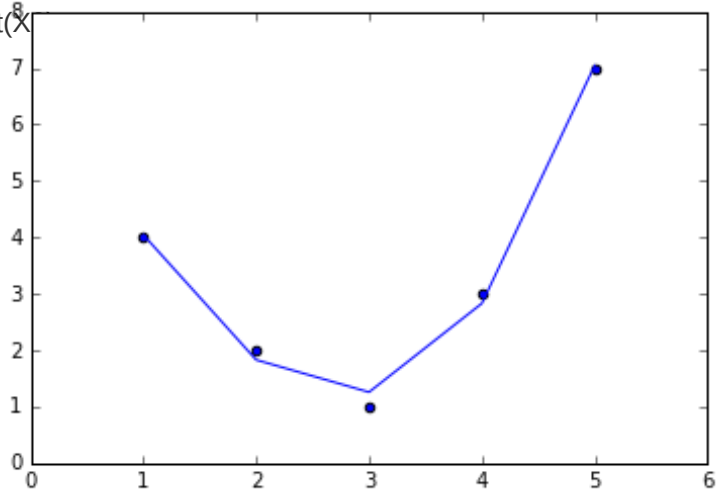
```
[[  1.   1.   1.]
 [  2.   4.   8.]
 [  3.   9.  27.]
 [  4.  16.  64.]
 [  5.  25. 125.]]
```

Features Derivadas

La matriz de features derivada tiene una columna representando x, una segunda columna representando x^2, y una tercer columna representando x^3.
Computar una regresión lineal en esta entrada expandida nos da un ajuste mucho más cercano a nuestros datos:

In [13]:

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



La idea de mejorar una solución, no cambiando el modelo, pero tranformando la entrada, es fundamental para muchos de los métodos más poderosos de machine learning.

Más generalmente, este es un tema que motiva la creación de las técnicas conocidas como *kernel methods*.

Imputación de datos faltantes

Otra necesidad común en feature engineering es el manejo de datos faltantes.

Por ejemplo, podríamos tener un dataset como este:

In [14]:

```
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])
```

Para aplicar un modelo de machine learning a estos datos, primero debemos reemplazar los datos perdidos con algún valor apropiado. Esto se conoce como imputación de datos faltantes y las estrategias van desde las simples (reemplazar valores faltantes con la media de la columna) hasta las más sofisticadas (como usar modelos robustos para imputación)

Imputación de datos faltantes

Como un ejemplo simple de imputación, usaremos la media.

Scikit-Learn provee la clase **Imputer**

In [15]:

```
from sklearn.preprocessing import Imputer
imp = Imputer(strategy='mean')
X2 = imp.fit_transform(X)
X2
```

Out[15]:

```
array([[ 4.5,  0. ,  3. ],
       [ 3. ,  7. ,  9. ],
       [ 3. ,  5. ,  2. ],
       [ 4. ,  5. ,  6. ],
       [ 8. ,  8. ,  1. ]])
```

Si queremos armar una secuencia de transformaciones, puede ser tedioso hacerlo a mano. Por ejemplo, podríamos querer hacer algo como esto:

1. Imputar valores perdidos usando la media
2. Transformar features a cuadráticas
3. Ajustar una regresión lineal

Pipelines

Para organizar este tipo de pipeline de procesamiento, Scikit-Learn provee una clase **Pipeline**

In [17]:

```
from sklearn.pipeline import make_pipeline

model = make_pipeline(Imputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
```

Este **Pipeline** se comporta como un objeto estándar de Scikit-Learn, y **aplicará todos los pasos especificados a cualquier dato de entrada**.

In [18]:

```
model.fit(X, y)  # X with missing values, from above
 print(y)
 print(model.predict(X))
```

```
[14 16 -1  8 -5]
 [ 14.  16.  -1.   8.  -5.]
```

Todos los pasos del modelo son aplicados automáticamente.

Notar que por cuestiones de simplicidad de la demostración, hemos aplicado el modelo a los datos de entrenamiento; por eso fue capaz de predecir perfectamente el resultado.