LISTADO DE TRANSFORMACIÓN DE VARIABLES

# INTRODUCCIÓN

A continuación, se enumera un listado suficientemente amplio de las transformaciones de variables que más frecuentemente se suelen utilizar, dándose una descripción funcional y algún ejemplo en Python para cada una de éstas.

Recuérdese que los modelos basados en árboles suelen ser robustos y aceptan “casi todo”, sin embargo, otros como las regresiones o los svm, necesitan y se recomienda, la aplicación de ciertas transformaciones en los datos.

<https://towardsdatascience.com/catalog-of-variable-transformations-to-make-your-model-works-better-7b506bf80b97>

# RECOMENDACIONES PREVIAS

Hay muchas técnicas de transformación de variables para el uso de modelización. Muchas de ellas, como algunas que ya se han visto están implementadas en librerías de Python como *scikit-learn* y *categorical encoders*.

En dichas transformaciones hay muchos parámetros que van desde por ejemplo una media y una desviación típica, hasta una tabla de conversión. Así pues un error común, es la aplicación de una transformación de variables en el dataset training y después aplicarla, pero con otros parámetros en el test:

Regla: Transformación del training y del test deben ser aplicadas usando los mismos parámetros y estos tienen que ser obtenido sólo y exclusivamente del dataset training

De no aplicar la anterior regla, no se podrán comparar los resultados. Para ello, como se verá los “*Transformadores*” de *scikit learn* resolverán esta cuestión con el uso de funciones como *fit()* y *transform()*

Diagrama, Tabla

Descripción generada automáticamente

En los casos en los que se aplica *cross-validation*, porque por ejemplo puedan existir outiliers y los parámetros puedan ser sensibles al muestreo aplicado, dichos parámetros de la transformación deberían ser derivados sólo a partir de los training y por tanto deben matenerse estos en los datos de validación y no transformar, antes de comenzar el proceso de *cross-validation*, es decir, antes de aplicar dicho proceso, se toma el dataset sin transformar. No obstante, cuando hay muchos datos, hay quién justifica la transformación previa del training dataset por la simplicidad que puede suponer ésta en el momento de crear el pipeline de modelización posterior, además dado que el modelo final va a estar basado en todo el training dataset, la transformación o no previa al *cross-validation* resulta aún discutible frente a la regla anterior que no lo es.

# LISTADO DE TRANSFORMACIONES

En función del tipo de variable, se aplican unas u otras transformaciones. A continuación se hace una enumeración de todas ellas junto con los principales parámetros y algunos de sus usos a tener en cuenta cuando se aplican:

## Transformaciones para variables numéricas

Las principales resultan ser las que se enumeran a continuación:

### Estandarización

Bastante conocida y habitual.

Expresión técnica:

Texto

Descripción generada automáticamente con confianza baja

Aplicaciones y/o usos: en muchas herramientas analíticas, esta transformación viene ya incluida, no obstante, donde suele predominar su uso es en los modelos de regresión lineal, svm, redes neuronales fundamentalmente.

Implementación técnica (ejemplo sencillo):

import pandas as pd

from sklearn.preprocessing import StandardScaler

train\_data = pd.DataFrame({'var1':[0, 0, 1, 1]})

test\_data = pd.DataFrame({'var1':[1.5]})

scaler = StandardScaler()

#Se calculan los parámetros en base al training

scaler.fit(train\_data)

print(scaler.transform(train\_data))

#¿Cuáles son los parámetros de la transformación? ¿Cuál sería el resultado de esta #función?

#Los parámetros se pueden ver si se desea con las siguientes líneas

print(scaler.mean\_)

print(scaler.var\_)

#Se aplican los parámetros al dataset test (que tiene una única observación, no se realiza #una transformación de éste

print(scaler.transform(test\_data))

# [[2.]] Compruébese “a mano” que efectivamente se obtiene este número

### Min – max scaling

Bastante conocida y habitual, aunque es bastante sensible a los outliers, cosa que no ocurre tanto con la anterior transformación

Expresión técnica:

En este sentido hay algunas variantes en la transformación, pero la que se usa en Python es la siguiente:

Otra variante sería la siguiente:

Aplicaciones y/o usos: los modelos donde más predomina su uso es en aquellos donde exista tratamiento de imágenes. Al ser una serie de valores cerrados (por ejemplo entre 0 y 255), su uso se muestra necesario en este tipo de aplicaciones

Implementación técnica (ejemplo sencillo):

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

train\_data = pd.DataFrame({'var1':[-1, 0, 1, 2]})

test\_data = pd.DataFrame({'var1':[2.5]})

scaler = MinMaxScaler()

#Se calculan los parámetros en base al training

scaler.fit(train\_data)

print(scaler.transform(train\_data))

#¿Cuáles son los parámetros de la transformación? ¿Cuál sería el resultado de esta #función?

#Los parámetros se pueden ver si se desea con las siguientes líneas

print(scaler.transform(test\_data))

# [[1.16666667]] Compruébese “a mano” que efectivamente se obtiene este número

### Logarithmic Tranformation

También bastante conocida y habitual, como es conocido, permite en ocasiones corregir la asimietría para conseguir una forma más “normal”.

Una distribución muy conocida y asociada a esta transformación, es la *distribución log-normal*. Su construcción viene como sigue:

Se observa en este caso:

Texto

Descripción generada automáticamente con confianza media

Expresión técnica:

Sobre esta transformación hay varias variantes, ya que hay que tener en cuenta, que al aplicar un logaritmo directamente, se podría tener problema ante valores nulos y sobre todo con los negativos, por tanto, cabe considerar las siguientes:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Aplicaciones y/o usos: algunos de los usos de esta transformación serían cuando se ajustan distribuciones, como la del precio de la vivienda por metro cuadrado o por ejemplo los retornos diarios de un determinado activo financiero

Implementación técnica (ejemplo sencillo):

En este caso, la implementación es directa y no requiere “memorizar parámetros” que se ajusten con los datos

### Box – Cox Tranformation

Esta transformación se aplicará y se recordará de nuevo en la sesión de series temporales que es donde se aplica habitualmente, pero también puede aplicarse a cualquier variable para tratar de “normalizarla”.

Expresión técnica:

Aunque su expresión es sencilla, no resulta trivial la estimación del parámetro *lambda*, que se estima a partir de los datos de la serie temporal

Texto

Descripción generada automáticamente con confianza media

Aplicaciones y/o usos: permite en ocasiones corregir la tendencia en varianza en las series temporales, algo que es clave para la aplicación de modelos econométricos que se analizan más adelante

Implementación técnica (ejemplo sencillo):

import pandas as pd

import numpy as np

from sklearn.preprocessing import PowerTransformer

train\_data = pd.DataFrame({'var1':[1, 0.1, 1.5, 2]})

pt = PowerTransformer(method='box-cox')

#Se calculan los parámetros en base al training

pt.fit(train\_data)

train\_data\_trans = pt.transform(train\_data)

print(train\_data\_trans)

#Sabiendo que lambda vale: 0.79614218¿Cuál sería el resultado de esta función?

#Los parámetros se pueden ver si se desea con las siguientes líneas

print(pt.lambdas\_)

### Yeo – Johnson Tranformation

Esta transformación tiene la misma idea que el caso de Box – Cox, pero se admite la existencia de valores negativos, lo que la anterior no soportaba.

Expresión técnica:

Aunque su expresión es sencilla, no resulta trivial la estimación del parámetro *lambda*, que se estima a partir de los datos de la serie temporal

Texto, Carta

Descripción generada automáticamente

Aplicaciones y/o usos: permite en ocasiones corregir la tendencia en varianza en las series temporales, algo que es clave para la aplicación de modelos econométricos que se analizan más adelante

Implementación técnica (ejemplo sencillo):

import pandas as pd

import numpy as np

from sklearn.preprocessing import PowerTransformer

train\_data = pd.DataFrame({'var1':[1, 0.1, 1.5, 2]})

pt = PowerTransformer(method= yeo-johnson)

#Se calculan los parámetros en base al training

pt.fit(train\_data)

train\_data\_trans = pt.transform(train\_data)

print(train\_data\_trans)

#Sabiendo que lambda vale: 1.34950574¿Cuál sería el resultado de esta función?

#Los parámetros se pueden ver si se desea con las siguientes líneas

print(pt.lambdas\_)

### Clipping

Esta transformación consiste en acotar y censurar bien por arriba o por abajo, los valores de una distribución continua.

Expresión técnica:

Su expresión básicamente es la de establecer un percentil, superior e inferior, de modo que cualquier valor que los supere en los extremos, tomará a dichos extremos

Aplicaciones y/o usos: la transformación en sí soluciona el problema de los outiliers, aunque puede crear situaciones engañosas, debe tenerse cuidado al emplearla

Implementación técnica (ejemplo sencillo): Aquí se hace uso de la función *quantile()* de Python, junto con la función *clip()* que aplica directamente las acotaciones

import pandas as pd

# Multiple columns can be clipped at once.

p01 = df[['Value','Age']].quantile(0.01)

p99 = df[['Value','Age']].quantile(0.99)

df[['Value','Age']] = df[['Value','Age']].clip(p01,p99,axis=1)

### Binning

Esta transformación consiste en agrupar la variable continua en distintos intervalos, acorde a un criterio, posteriormente cabe usar el índice del grupo como variable categórica o como un rango numérico

Expresión técnica:

En este caso la variable pasa de un continuo a unos pocos valores, en base a un criterio, es lo que se conoce como segmentaciones

Aplicaciones y/o usos: en credit-scoring, el binning resulta esencial para la construcción de un modelo de tarjeta de puntuación

Implementación técnica (ejemplo sencillo): Tras el criterio a aplicar, la función que rige la transformación es *cut()* que aplica directamente los extremos de los intervalos creados

import pandas as pd

#when bins is integer, it will be the number of bins and bin widths are to be equal.

df['Value\_binned'] = pd.cut(df['Value'], bins=10, labels=False)

df[['Value','Value\_binned']].head(5)

# Value Value\_binned

# 0 110500000.0 9

# 1 77000000.0 6

# 2 118500000.0 9

# 3 72000000.0 6

# 4 102000000.0 8

#when bins is list of values, they are to be bin intervals.

df['Value\_binned'] = pd.cut(df['Value'], bins=[-float('inf'),5\*10\*\*7, 10\*10\*\*7, float('inf')], labels=False)

df[['Value','Value\_binned']].head(5)

### Ranking

Consiste en convertir los valores continuos en rangos (comenzando por ejemplo por 1)

Expresión técnica:

Se entiende mejor aplicándola y viendo cómo se re-ordenan los datos

Aplicaciones y/o usos: homogeneiza el efecto de una variable continua, en vez de dejar que los valores fluctúen a lo largo de todo su espectro, permite que estos estén controlados entre 1 y el máximo número de observaciones

Implementación técnica (ejemplo sencillo):

import pandas as pd

x = pd.Series([5, 4, 0, -1, 3, 10])

y = pd.Series([5, 2.5, 0, -1, 2.5, 10])

rank = x.rank()

print(rank.values)

# [5. 4. 2. 1. 3. 6.]: the rank of values starting from 1

rank = y.rank()

print(rank.values)

# [5. 3.5 2. 1. 3.5 6. ] : if there are ties, the aveage rank is assigned.

### RankGauss

Consiste en convertir los valores continuos en rangos y después convertir estos valores en otros acordes a una distribución *normal*.

Expresión técnica:

Se entiende mejor aplicándola y viendo cómo se re-ordenan los datos

Aplicaciones y/o usos: se aplicó en kaggle en el problema Porto Segro’s Safe Driver Prediction

Implementación técnica (ejemplo sencillo):

import matplotlib.pyplot as plt

%matplotlib inline

from sklearn.preprocessing import QuantileTransformer

transformer = QuantileTransformer(n\_quantiles=100, random\_state=1111, output\_distribution='normal')

transformer.fit(df[['Age']])

df['Age\_RankGauss'] = transformer.transform(df[['Age']])

fig, (ax1, ax2) = plt.subplots(1,2, figsize=[12,5])

ax1.hist(df['Age'],bins=100)

ax1.title.set\_text('Original Data')

ax2.hist(df['Age\_RankGauss'],bins=100)

ax2.title.set\_text('RankGauss-ed Data')

plt.show()

## Transformaciones para variables categóricas

Las principales resultan ser las que se enumeran a continuación:

### One – hot encoding

Con esta transformación se convierte una columna categórica a múltiples binarias 0 – 1, tantas como el número de niveles distintos existan en la columna original. Si hay 4 niveles en la variable categórica, *one – hot encoding* creará 4 columnas por tanto

Tabla

Descripción generada automáticamente

Aplicaciones y/o usos: es una transformación muy usual y en algunos contexto como R, existe el tipo *factor* que “por debajo” permite la aplicación de esta transformación que tiene 2 debilidades. Por un lado, si hay muchas categorías, se puede “inflar” el dataset enormemente, mientras que por otro lado, en modelos con variable explicada continua, puede crear problemas de consistencia. No obstante, ambos problemas podrían solucionarse mediante agrupación de niveles mediante grupos tipo “otros”

Implementación técnica (ejemplo sencillo):

La función que permite esta transformación es *get\_dummies()* del módulo *OneHotEncoder* de *scikit-learn preporcessing* que permite la generación de matrices dispersas para ahorrar recursos

import pandas as pd

df\_x = pd.DataFrame({'v1':['A','B','D','C','B','C'], 'v2':[10,11,2,0,30,50]})

df\_x

# v1 v2

# 0 A 10

# 1 B 11

pd.get\_dummies(df\_x,columns=['v1'])

# get\_dummies function will remove original column and generate one-hot encoding # columns

# v2 v1\_A v1\_B v1\_C v1\_D

# 0 10 1 0 0 0

# 1 11 0 1 0 0

# 2 2 0 0 0 1

### Label encoding

Esta transformación hay que usarla con cuidado. Básicamente es transformar las categorías a números, pero el peligro claro, es que dichos números no tienen por qué guardar relación de orden y magnitud con la categoría en sí que representan

Diagrama, Tabla

Descripción generada automáticamente

Aplicaciones y/o usos: no es muy recomendable su aplicación en contextos de modelización, salvo que se tenga justificaciones claras. En modelos tipo árbol, quizás su aplicación sería más intuitiva que frente a una regresión lineal. En variables ordinales, es quizás donde más sentido puede tener su aplicación

Implementación técnica (ejemplo sencillo):

La función que permite esta transformación es *LabelEncoder()* del módulo *OneHotEncoder* de *scikit-learn preporcessing* que permite la generación de matrices dispersas para ahorrar recursos

from sklearn.preprocessing import LabelEncoder

import pandas as pd

df\_x = pd.DataFrame({'v1':['A','B','D','C','B','C'], 'v2':[10,11,2,0,30,50]})

le = LabelEncoder()

le.fit(df\_x['v1'])

df\_x['v1\_le'] = le.transform(df\_x['v1'])

df\_x

# # v1 v2 v1\_le

# 0 A 10 0

# 1 B 11 1

# 2 D 2 3

# 3 C 0 2

# 4 B 30 1

# 5 C 50 2

### Feature hashing

Similar al *one – hot – encoding* pero los valores a utilizar van a ser más de 2 y no tiene por qué generarse tantas variables como categorías

Tabla

Descripción generada automáticamente

Aplicaciones y/o usos: no es muy recomendable su aplicación en contextos de modelización, salvo que se tenga justificaciones claras.

Implementación técnica (ejemplo sencillo):

La función que permite esta transformación es *FeatureHasher()* del módulo *feature\_extraction* de *scikit-learn* o bien con *HashingEncoder()* de *category\_encoders.hashing*

from sklearn.feature\_extraction import FeatureHasher

import pandas as pd

df\_x = pd.DataFrame({'v1':['A','B','D','C','B','C'], 'v2':[10,11,2,0,30,50]})

fh = FeatureHasher(n\_features=3, input\_type='string')

hashed = fh.transform(df\_x[['v1']].astype(str).values)

hashed = pd.DataFrame(hashed.todense())

hashed.columns = ['v1\_fh'+str(i) for i in hashed.columns]

pd.concat([df\_x,hashed],axis=1)

# v1 v2 v1\_fh0 v1\_fh1 v1\_fh2

# 0 A 10 0.0 1.0 0.0

# 1 B 11 0.0 0.0 -1.0

# 2 D 2 0.0 0.0 1.0

# 3 C 0 -1.0 0.0 0.0

# 4 B 30 0.0 0.0 -1.0

# 5 C 50 -1.0 0.0 0.0

### Binary & Base – N Encoding

Es una transformación que permite pasar de una columna a otras de tipo binario (como en el caso del *one – hot – encoding*, pero minimizando el número de columnas a crear según se muestra en el siguiente esquema

Diagrama

Descripción generada automáticamente

La anterior transformación se generaliza a una base logarítmica distinta de 2 con *Base – N Encoding*:

Gráfico, Diagrama, Gráfico de embudo

Descripción generada automáticamente

Aplicaciones y/o usos: su aplicación puede ofrecer resultados interesantes, al ahorrar almacenamiento y conservar gran parte de la información original, pero debe tenerse cuidado con la codificación a realizar.

Implementación técnica (ejemplo sencillo):

La función que permite esta transformación es *category\_encoder()* del módulo *BinaryEncoder*

from category\_encoders import BinaryEncoder

import pandas as pd

df\_x = pd.DataFrame({'v1':['A','B','D','C','B','C'], 'v2':[10,11,2,0,30,50]})

be = BinaryEncoder()

df\_be = be.fit\_transform(df\_x['v1'])

pd.concat([df\_x,df\_be],axis=1)

# v1 v2 v1\_0 v1\_1 v1\_2

# 0 A 10 0 0 1

# 1 B 11 0 1 0

# 2 D 2 0 1 1

# 3 C 0 1 0 0

# 4 B 30 0 1 0

# 5 C 50 1 0 0

from category\_encoders import BaseNEncoder

import pandas as pd

df\_x = pd.DataFrame({'v1':['A','B','D','C','B','C'], 'v2':[10,11,2,0,30,50]})

bne = BaseNEncoder(base=3)

df\_bne = bne.fit\_transform(df\_x['v1'])

pd.concat([df\_x,df\_bne],axis=1)

# v1 v2 v1\_0 v1\_1 v1\_2

# 0 A 10 0 0 1

# 1 B 11 0 0 2

# 2 D 2 0 1 0

### Frecuency Encoding

En esta transformación de una única columna de categorías, se pasa también a una única columna numérica, cuyos valores van a ser las frecuencias de la columna original

Diagrama

Descripción generada automáticamente

Aplicaciones y/o usos: en ocasiones la frecuencia de los niveles, tienen capacidad de influencia en la *variable explicada* y por tanto es una transformación muy a tener en cuenta, aunque la variable transformada, está descontextualizada con respecto a la original. También cabe encadenar transformaciones y convertir esas frecuencias en rangos mediante una *rank transformation* anteriormente explicada

Implementación técnica (ejemplo sencillo):

Actualmente no parece haber librería que soporte esta transformación directamente en Python, por tanto habría que programarla y tener en cuenta que hay que hacer los cálculos con respecto a un *training* para después aplicar las frecuencias obtenidas por categorías al *test*, a diferencia de las anteriores transformaciones categóricas, se necesita “memorizar los parámetros”.

import pandas as pd

df\_xtrain = pd.DataFrame({'v1':['B','B','A','C','B','C'], 'v2':[10,11,2,0,30,50]})

df\_xtest = pd.DataFrame({'v1':['C','A','B'], 'v2':[5,1,20]})

freq = df\_xtrain['v1'].value\_counts()

df\_xtrain['v1\_fe'] = df\_xtrain['v1'].map(freq)

# test data should be applied the same encoding rule as training data.

df\_xtest['v1\_fe'] = df\_xtest['v1'].map(freq)

# ¿Cuáles son los valores del training y del test tras esta transformación?

### Target Encoding

Esta transformación es similar a la anterior, en el sentido de que trata de convertir una variable categórica en una única variable continua, con la diferencia de que los valores numéricos creados van a estar basados en los valores de la *variable explicada*.

Diagrama

Descripción generada automáticamente

Aplicaciones y/o usos: transformación que puede crear predictores potentes, pero puede hacer que nuestros análisis caígan en el *data leaking* si no se toma el cuidado necesario. Por ello se recomienda realizar un *k – folding* de al menos orden 4, que evite por ejemplo situaciones en los que en el dataset pueda haber una ordenación por la *target* o alguna otra circunstancia sistemática más sutil y que no se tenga en cuenta a priori

Diagrama

Descripción generada automáticamente

Implementación técnica (ejemplo sencillo):

En este caso si existe función *TargetEncoder()* del módulo *category\_encoder* que permite hacer la transformación, que como se ve a continuación, se complica si se introduce un *k-folding*

from category\_encoders import target\_encoder

import pandas as pd

from sklearn.model\_selection import KFold

df\_xtrain= pd.DataFrame({'v1':['A','B','D','C','B','C','A','B','D','A','B','C','A','B','D','C','B','C','B','A']})

df\_ytrain = pd.DataFrame({'y':[1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0]})

# Last record has new level 'E', not included in training set.

df\_xtest = pd.DataFrame({'v1':['C','A','B','E']})

# Target encoding for training set using Kfolding.

kf = KFold(n\_splits=4, shuffle=True, random\_state=1111)

train\_te = pd.DataFrame()

for tra\_idx, val\_idx in kf.split(df\_xtrain):

#if level is unknown from input data, return nan.

te=target\_encoder.TargetEncoder(handle\_missing='return\_nan',handle\_unknown='return\_nan')

te.fit(df\_xtrain.iloc[tra\_idx],df\_ytrain.iloc[tra\_idx])

temp = te.transform(df\_xtrain.iloc[val\_idx])

train\_te = pd.concat([train\_te,temp],axis=0)

train\_te.sort\_index(inplace=True)

train\_te.columns = ['v1\_te']

pd.concat([df\_xtrain,train\_te,df\_ytrain],axis=1)

# v1 v1\_te y

# 0 A 0.349227 1

# 1 B 0.498419 0

# 2 D 0.071718 0

# 3 C 0.733401 1

# Target encoding for test set.

te= target\_encoder.TargetEncoder(handle\_missing='return\_nan',handle\_unknown='return\_nan')

te.fit(df\_xtrain,df\_ytrain)

test\_te = te.transform(df\_xtest)

test\_te.columns = ['v1\_te']

pd.concat([df\_xtest,test\_te],axis=1)

# v1 v1\_te

# 0 C 0.792806

# 1 A 0.400000

### Otras posibles transformaciones

Algunas transformaciones de variables categóricas menos frecuentes y que se comentan rápidamente serían las siguientes:

* *Helmert Encoding*: en este encoding, la media de la *variable dependiente* para un nivel es comparada con la media de dicha variable en todos los niveles previos

Tabla

Descripción generada automáticamente

* *Mean Encoding*: similar al *target encoding* pero aplicable en variables continuas donde se calcula la media de la *dependiente* por categoría
* *Leave* – one – out – Encoding: similar al *target encoding* solo que en el momento de promediar, no se tiene en cuenta la primera fila de la categoría a considerar, trata por tanto de corregir en alguna manera ese posible *data leaking* que podría presentarse en los datasets
* *James-Stein Encoding*: devuelve una nueva variable formada por una media ponderada de la *variable dependiente* de la característica observada y el valor de medio de dicha *variable dependiente*

Diagrama

Descripción generada automáticamente