Scales

May 23, 2022

1 Scales

[1]:
```
# Let's bring in pandas as normal
import pandas as pd

# Heres an example. Lets create a dataframe of letter grades in descending␣
↪order. We can also set an index
# value and here we'll just make it some human judgement of how good a student␣
↪was, like "excellent" or "good"

df=pd.DataFrame(['A+', 'A', 'A-', 'B+', 'B', 'B-', 'C+', 'C', 'C-', 'D+', 'D'],
                index=['excellent', 'excellent', 'excellent', 'good', 'good',␣
↪'good',
                       'ok', 'ok', 'ok', 'poor', 'poor'],
               columns=["Grades"])
df
```

[1]:
```
          Grades
excellent     A+
excellent      A
excellent     A-
good          B+
good           B
good          B-
ok            C+
ok             C
ok            C-
poor          D+
poor           D
```

[2]:
```
# Now, if we check the datatype of this column, we see that it's just an␣
↪object, since we set string values
df.dtypes
```

[2]:
```
Grades    object
dtype: object
```

[3]:
```python
# We can, however, tell pandas that we want to change the type to category,␣
↪using the astype() function
df["Grades"].astype("category").head()
```

[3]:
```
excellent    A+
excellent     A
excellent    A-
good         B+
good          B
Name: Grades, dtype: category
Categories (11, object): [A, A+, A-, B, ..., C+, C-, D, D+]
```

[9]:
```python
# We see now that there are eleven categories, and pandas is aware of what␣
↪those categories are. More
# interesting though is that our data isn't just categorical, but that it's␣
↪ordered. That is, an A- comes
# after a B+, and B comes before a B+. We can tell pandas that the data is␣
↪ordered by first creating a new
# categorical data type with the list of the categories (in order) and the␣
↪ordered=True flag
my_categories=pd.CategoricalDtype(categories=['D', 'D+', 'C-', 'C', 'C+', 'B-',␣
↪'B', 'B+', 'A-', 'A', 'A+'],
                           ordered=True)
# then we can just pass this to the astype() function

# Acá creamos una serie llamada 'grades' con los datos de la columna "Grades"␣
↪convertidos a categóricos y con orden
grades=df["Grades"].astype(my_categories)
grades.head()
```

[9]:
```
excellent    A+
excellent     A
excellent    A-
good         B+
good          B
Name: Grades, dtype: category
Categories (11, object): [D < D+ < C- < C ... B+ < A- < A < A+]
```

[11]:
```python
# Now we see that pandas is not only aware that there are 11 categories, but it␣
↪is also aware of the order of
# those categoreies. So, what can you do with this? Well because there is an␣
↪ordering this can help with
# comparisons and boolean masking. For instance, if we have a list of our␣
↪grades and we compare them to a C
# we see that the lexicographical comparison returns results we were not␣
↪intending.
```

```
# Hacemos un query sobre la df donde las categorias y su orden no fueron
↪asignadas
df[df["Grades"]>"C"]
```

[11]:
```
      Grades
ok        C+
ok        C-
poor      D+
poor       D
```

[13]:
```
# Vemos que grades sigue siendo un dtype:object, por eso el query anterior dió
↪algo no esperado
df.dtypes
```

[13]:
```
Grades    object
dtype: object
```

[14]:
```
# So a C+ is great than a C, but a C- and D certainly are not. However, if we
↪broadcast over the dataframe
# which has the type set to an ordered categorical


# OJO QUE grades es una SERIE! Por ende no estamos haciendo query sobre un df,
↪sino sobre una serie
# Acá el resultado es el esperado porque usamos las categorías ordenadas
grades[grades>"C"]
```

[14]:
```
excellent    A+
excellent     A
excellent    A-
good         B+
good          B
good         B-
ok           C+
Name: Grades, dtype: category
Categories (11, object): [D < D+ < C- < C ... B+ < A- < A < A+]
```

[7]:
```
# We see that the operator works as we would expect. We can then use a certain
↪set of mathematical operators,
# like minimum, maximum, etc., on the ordinal data.
```

[8]:
```
# Sometimes it is useful to represent categorical values as each being a column
↪with a true or a false as to
# whether the category applies. This is especially common in feature
↪extraction, which is a topic in the data
# mining course. Variables with a boolean value are typically called dummy
↪variables, and pandas has a built
# in function called get_dummies which will convert the values of a single
↪column into multiple columns of
```

```
# zeros and ones indicating the presence of the dummy variable. I rarely use␣
↪it, but when I do it's very
# handy.
```

[40]:
```
# Theres one more common scale-based operation Id like to talk about, and thats␣
↪on converting a scale from
# something that is on the interval or ratio scale, like a numeric grade, into␣
↪one which is categorical. Now,
# this might seem a bit counter intuitive to you, since you are losing␣
↪information about the value. But its
# commonly done in a couple of places. For instance, if you are visualizing the␣
↪frequencies of categories,
# this can be an extremely useful approach, and histograms are regularly used␣
↪with converted interval or ratio
# data. In addition, if youre using a machine learning classification approach␣
↪on data, you need to be using
# categorical data, so reducing dimensionality may be useful just to apply a␣
↪given technique. Pandas has a
# function called cut which takes as an argument some array-like structure like␣
↪a column of a dataframe or a
# series. It also takes a number of bins to be used, and all bins are kept at␣
↪equal spacing.

# Lets go back to our census data for an example. We saw that we could group by␣
↪state, then aggregate to get a
# list of the average county size by state. If we further apply cut to this␣
↪with, say, ten bins, we can see
# the states listed as categoricals using the average county size.

# let's bring in numpy
import numpy as np

# Now we read in our dataset
df=pd.read_csv("datasets/census.csv")

# And we reduce this to country data
df=df[df['SUMLEV']==50]

# And for a few groups
# Creamos una serie que agrupó todas las ciudades de cada estado y les calculó␣
↪el promedio de habitantes a cada estado.
df=df.set_index('STNAME').groupby(level=0)['CENSUS2010POP'].agg(np.average)

df.head()
```

[40]:
```
STNAME
Alabama         71339.343284
```

```
Alaska         24490.724138
Arizona       426134.466667
Arkansas       38878.906667
California    642309.586207
Name: CENSUS2010POP, dtype: float64
```

[36]:
```
# Now if we just want to make "bins" of each of these, we can use cut()
# Creamos una serie nueva con bins

# Los bins con grupos con rango de valores, es una variable categórica y␣
↪agrupan a los estados que tienen
# número de habitantes dentro de ese rango.

# a la serie que creamos antes df le aplicamos pd.cut() para generar 10 bins
df2= pd.cut(df,10)
```

[38]:
```
print(type(df2))
print(df2.head(20))
```

```
<class 'pandas.core.series.Series'>
STNAME
Alabama                  (11706.087, 75333.413]
Alaska                   (11706.087, 75333.413]
Arizona                (390320.176, 453317.529]
Arkansas                 (11706.087, 75333.413]
California             (579312.234, 642309.586]
Colorado                (75333.413, 138330.766]
Connecticut            (390320.176, 453317.529]
Delaware               (264325.471, 327322.823]
District of Columbia   (579312.234, 642309.586]
Florida                (264325.471, 327322.823]
Georgia                  (11706.087, 75333.413]
Hawaii                 (264325.471, 327322.823]
Idaho                    (11706.087, 75333.413]
Illinois                (75333.413, 138330.766]
Indiana                  (11706.087, 75333.413]
Iowa                     (11706.087, 75333.413]
Kansas                   (11706.087, 75333.413]
Kentucky                 (11706.087, 75333.413]
Louisiana                (11706.087, 75333.413]
Maine                   (75333.413, 138330.766]
Name: CENSUS2010POP, dtype: category
Categories (10, interval[float64]): [(11706.087, 75333.413] < (75333.413,
138330.766] < (138330.766, 201328.118] < (201328.118, 264325.471] ...
(390320.176, 453317.529] < (453317.529, 516314.881] < (516314.881, 579312.234] <
(579312.234, 642309.586]]
```

[35]:
```
# Podemos ver que a partir del número de habitantes, se crearon 6 rangos de␣
↪habitantes y
# a cada estado se le asingno una de ellos
set(df2)
```

[35]:
```
{Interval(11706.087, 91082.751, closed='right'),
 Interval(91082.751, 169829.442, closed='right'),
 Interval(169829.442, 248576.133, closed='right'),
 Interval(248576.133, 327322.823, closed='right'),
 Interval(406069.514, 484816.205, closed='right'),
 Interval(563562.896, 642309.586, closed='right')}
```

[11]:
```
# Here we see that states like alabama and alaska fall into the same category,␣
↪while california and the
# disctrict of columbia fall in a very different category.

# Now, cutting is just one way to build categories from your data, and there␣
↪are many other methods. For
# instance, cut gives you interval data, where the spacing between each␣
↪category is equal sized. But sometimes
# you want to form categories based on frequency  you want the number of items␣
↪in each bin to the be the
# same, instead of the spacing between bins. It really depends on what the␣
↪shape of your data is, and what
# youre planning to do with it.
```