03-04: Scales ¶

Now that we've covered the mechanics of pandas, we can afford to sidetrack a bit and talk for a moment about data types and scales. We've already seen that pandas supports a number of different computational data types such as strings, integers, floating point numbers. What this doesn't capture is what we call the **scale of the data**.

As a data scientist, there's at least 4 different scales worth knowing about.

Types of Scales

Types of Scales: Ratio Scale

- units are equally spaced.
- Mathematical operations are all valid.
- For instance, height and weight.

Types of Scales: Interval Scale

- Units are equally spaced
- There is no true zero.
- Multiplication and division are usually not defined.
- ie. Temperature: there is never an absence a temperature, 0 degree celsius is still a temperature. or the Compass, where a location that has a bearing of 0deg from North is still a direction heading toward north.

Types of Scales: Ordinal Scale:

- Order of the units are important, but they are **not** evenly spaced.
- Letter grades such as A+, A, B are a good example.
- Most common in machine learning but relatively challenging dataset to work with.

Types of Scales: Nominal Scale

- Categories of data, but the categories have no order with respect to one another.
- ie. Names of the players in a team.
- Changing orders of the items or applying mathematical functions to these data are meaningless.
- Categorial data are very common, and those that only have 2 possible values we label them as **binary categories**.

Using Pandas

In [1]:

```
# Let's bring in pandas as normal
import pandas as pd

# Here's an example. Lets create a dataframe of letter grades in descending orde
r. We can also set an index
# value and here we'll just make it some human judgement of how good a student w
as, like "excellent" or "good"

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

Out[1]:

	Grades
excellent	A+
excellent	A
excellent	A-
good	B+
good	B
good	B-
ok	C+
ok	C
ok	C-
poor	D+
poor	D

In [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
```

Out[2]:

```
Grades    object
dtype: object
```

Using Pandas: Converting from ordinal data to categorical data

In [3]:

```
# We can, however, tell pandas that we want to change the type to category, usin
g the astype() function
df["Grades"].astype("category").head()
```

Out[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+]
```

Sidetrack: Grade bandings are actually 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

In [4]:

```
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
grades=df["Grades"].astype(my_categories)
grades.head()
```

Out[4]:

```
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+]
```

Using Boolean Masks on Ordered Categorical Data

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.

In [5]:

```
# 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 ord
ering this can help with
# comparisons and boolean masking.

df[df["Grades"]>"C"]
```

Out[5]:

	Grades
ok	C+
ok	C-
poor	D+
poor	D

In [ ]:

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

grades[grades>"C"]
```

In [ ]:

```
# We see that the operator works as we would expect. We can then use a certain s
et of mathematical operators,
# like minimum, maximum, etc., on the ordinal data.
```

Feature Extraction

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.

Converting Interval/Ratio scale data into Categorical Data

There's one more common scale-based operation I'd like to talk about, and that's 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 it's 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 you're 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 eries. It also takes a number of bins to be used, and all bins are kept at equal spacing.

In [7]:

```
# 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 wit
h, 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
df=df.set_index('STNAME').groupby(level=0)['CENSUS2010POP'].agg(np.average)

df.head()
```

Out[7]:

```
STNAME
Alabama         71339.343284
Alaska          24490.724138
Arizona        426134.466667
Arkansas        38878.906667
California     642309.586207
Name: CENSUS2010POP, dtype: float64
```

In [8]:

```
# Now if we just want to make "bins" of each of these, we can use cut()
pd.cut(df,10)
```

Out[8]:

```
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]
Maryland                (201328.118, 264325.471]
Massachusetts           (453317.529, 516314.881]
Michigan                 (75333.413, 138330.766]
Minnesota                 (11706.087, 75333.413]
Mississippi               (11706.087, 75333.413]
Missouri                  (11706.087, 75333.413]
Montana                   (11706.087, 75333.413]
Nebraska                  (11706.087, 75333.413]
Nevada                  (138330.766, 201328.118]
New Hampshire            (75333.413, 138330.766]
New Jersey              (390320.176, 453317.529]
New Mexico                (11706.087, 75333.413]
New York                (264325.471, 327322.823]
North Carolina           (75333.413, 138330.766]
North Dakota              (11706.087, 75333.413]
Ohio                     (75333.413, 138330.766]
Oklahoma                  (11706.087, 75333.413]
Oregon                   (75333.413, 138330.766]
Pennsylvania            (138330.766, 201328.118]
Rhode Island            (201328.118, 264325.471]
South Carolina           (75333.413, 138330.766]
South Dakota              (11706.087, 75333.413]
Tennessee                 (11706.087, 75333.413]
Texas                    (75333.413, 138330.766]
Utah                     (75333.413, 138330.766]
Vermont                   (11706.087, 75333.413]
Virginia                  (11706.087, 75333.413]
Washington              (138330.766, 201328.118]
West Virginia             (11706.087, 75333.413]
Wisconsin                (75333.413, 138330.766]
Wyoming                   (11706.087, 75333.413]
Name: CENSUS2010POP, dtype: category
Categories (10, interval[float64]): [(11706.087, 75333.413] < (7533
3.413, 138330.766] < (138330.766, 201328.118] < (201328.118, 264325.
471] ... (390320.176, 453317.529] < (453317.529, 516314.881] < (5163
14.881, 579312.234] < (579312.234, 642309.586]]
```

Summary

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 you’re planning to do with it.

In [ ]:

```

```