QueryingDataFrame_ed

July 14, 2023

1 Querying DataFrame

In this lecture we're going to talk about querying DataFrames. The first step in the process is to understand Boolean masking. Boolean masking is the heart of fast and efficient querying in numpy and pandas, and it's analogous to bit masking used in other areas of computational science. By the end of this lecture you'll understand how Boolean masking works, and how to apply this to a DataFrame to get out data you're interested in.

A Boolean mask is an array which can be of one dimension like a series, or two dimensions like a data frame, where each of the values in the array are either true or false. This array is essentially overlaid on top of the data structure that we're querying. And any cell aligned with the true value will be admitted into our final result, and any cell aligned with a false value will not.

[14]:
```
# Let's start with an example and import our graduate admission dataset. First␣
↪we'll bring in pandas
import pandas as pd
# Then we'll load in our CSV file
df = pd.read_csv('datasets/Admission_Predict.csv', index_col=0)
# And we'll clean up a couple of poorly named columns like we did in a previous␣
↪lecture
df.columns = [x.lower().strip() for x in df.columns]
# And we'll take a look at the results
df.head()
```

[14]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                 337          118                  4  4.5  4.5  9.65
2                 324          107                  4  4.0  4.5  8.87
3                 316          104                  3  3.0  3.5  8.00
4                 322          110                  3  3.5  2.5  8.67
5                 314          103                  2  2.0  3.0  8.21

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
```

```
5                  0             0.65
```

[15]:
```
# Boolean masks are created by applying operators directly to the pandas Series␣
↪or DataFrame objects.
# For instance, in our graduate admission dataset, we might be interested in␣
↪seeing only those students
# that have a chance higher than 0.7

# To build a Boolean mask for this query, we want to project the chance of␣
↪admit column using the
# indexing operator and apply the greater than operator with a comparison value␣
↪of 0.7. This is
# essentially broadcasting a comparison operator, greater than, with the␣
↪results being returned as
# a Boolean Series. The resultant Series is indexed where the value of each␣
↪cell is either True or False
# depending on whether a student has a chance of admit higher than 0.7
admit_mask=df['chance of admit'] > 0.7
admit_mask
```

[15]:
```
Serial No.
1        True
2        True
3        True
4        True
5       False
        …
396      True
397      True
398      True
399     False
400      True
Name: chance of admit, Length: 400, dtype: bool
```

[16]:
```
# This is pretty fundamental, so take a moment to look at this. The result of␣
↪broadcasting a comparison
# operator is a Boolean mask - true or false values depending upon the results␣
↪of the comparison. Underneath,
# pandas is applying the comparison operator you specified through␣
↪vectorization (so efficiently and in
# parallel) to all of the values in the array you specified which, in this␣
↪case, is the chance of admit
# column of the dataframe. The result is a series, since only one column is␣
↪being operator on, filled with
# either True or False values, which is what the comparison operator returns.
```

```
[17]: # So, what do you do with the boolean mask once you have formed it? Well, you
       ↪can just lay it on top of the
      # data to "hide" the data you don't want, which is represented by all of the
       ↪False values. We do this by using
      # the .where() function on the original DataFrame.
      df.where(admit_mask).head()
```

```
[17]:            gre score  toefl score  university rating  sop  lor  cgpa  \
      Serial No.
      1              337.0        118.0                4.0  4.5  4.5  9.65
      2              324.0        107.0                4.0  4.0  4.5  8.87
      3              316.0        104.0                3.0  3.0  3.5  8.00
      4              322.0        110.0                3.0  3.5  2.5  8.67
      5                NaN          NaN                NaN  NaN  NaN   NaN

                 research  chance of admit
      Serial No.
      1               1.0             0.92
      2               1.0             0.76
      3               1.0             0.72
      4               1.0             0.80
      5               NaN              NaN
```

```
[18]: # We see that the resulting data frame keeps the original indexed values, and
       ↪only data which met
      # the condition was retained. All of the rows which did not meet the condition
       ↪have NaN data instead,
      # but these rows were not dropped from our dataset.
      #
      # The next step is, if we don't want the NaN data, we use the dropna() function
      df.where(admit_mask).dropna().head()
```

```
[18]:            gre score  toefl score  university rating  sop  lor  cgpa  \
      Serial No.
      1              337.0        118.0                4.0  4.5  4.5  9.65
      2              324.0        107.0                4.0  4.0  4.5  8.87
      3              316.0        104.0                3.0  3.0  3.5  8.00
      4              322.0        110.0                3.0  3.5  2.5  8.67
      6              330.0        115.0                5.0  4.5  3.0  9.34

                 research  chance of admit
      Serial No.
      1               1.0             0.92
      2               1.0             0.76
      3               1.0             0.72
      4               1.0             0.80
      6               1.0             0.90
```

[19]:
```
# The returned DataFrame now has all of the NaN rows dropped. Notice the index␣
↪now includes
# one through four and six, but not five.
#
# Despite being really handy, where() isn't actually used that often. Instead,␣
↪the pandas devs
# created a shorthand syntax which combines where() and dropna(), doing both at␣
↪once. And, in
# typical fashion, the just overloaded the indexing operator to do this!

df[df['chance of admit'] > 0.7].head()
```

[19]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                 337          118                  4  4.5  4.5  9.65
2                 324          107                  4  4.0  4.5  8.87
3                 316          104                  3  3.0  3.5  8.00
4                 322          110                  3  3.5  2.5  8.67
6                 330          115                  5  4.5  3.0  9.34

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
6                  1             0.90
```

[20]:
```
# I personally find this much harder to read, but it's also very more common␣
↪when you're reading other
# people's code, so it's important to be able to understand it. Just reviewing␣
↪this indexing operator on
# DataFrame, it now does two things:

# It can be called with a string parameter to project a single column
df["gre score"].head()
```

[20]:
```
Serial No.
1    337
2    324
3    316
4    322
5    314
Name: gre score, dtype: int64
```

[21]:
```
# Or you can send it a list of columns as strings
df[["gre score","toefl score"]].head()
```

[21]:
```
             gre score  toefl score
Serial No.
1                  337          118
2                  324          107
3                  316          104
4                  322          110
5                  314          103
```

[22]:
```
# Or you can send it a boolean mask
df[df["gre score"]>320].head()
```

[22]:
```
             gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                  337          118                  4  4.5  4.5  9.65
2                  324          107                  4  4.0  4.5  8.87
4                  322          110                  3  3.5  2.5  8.67
6                  330          115                  5  4.5  3.0  9.34
7                  321          109                  3  3.0  4.0  8.20

             research  chance of admit
Serial No.
1                   1             0.92
2                   1             0.76
4                   1             0.80
6                   1             0.90
7                   1             0.75
```

[23]:
```
# And each of these is mimicing functionality from either .loc() or .where().
↪dropna().
```

[24]:
```
# Before we leave this, lets talk about combining multiple boolean masks, such␣
↪as multiple criteria for
# including. In bitmasking in other places in computer science this is done␣
↪with "and", if both masks must be
# True for a True value to be in the final mask), or "or" if only one needs to␣
↪be True.

# Unfortunatly, it doesn't feel quite as natural in pandas. For instance, if␣
↪you want to take two boolean
# series and and them together
(df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [24], line 7
      1 # Before we leave this, lets talk about combining multiple boolean␣
↪masks, such as multiple criteria for
```

```
      2 # including. In bitmasking in other places in computer science this is␣
↪done with "and", if both masks must be
      3 # True for a True value to be in the final mask), or "or" if only one␣
↪needs to be True.
      4
      5 # Unfortunatly, it doesn't feel quite as natural in pandas. For␣
↪instance, if you want to take two boolean
      6 # series and and them together
----> 7 (df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)

File /opt/conda/lib/python3.9/site-packages/pandas/core/generic.py:1527, in␣
↪NDFrame.__nonzero__(self)
   1525 @final
   1526 def __nonzero__(self) -> NoReturn:
-> 1527     raise ValueError(
   1528         f"The truth value of a {type(self).__name__} is ambiguous. "
   1529         "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
   1530     )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.
↪item(), a.any() or a.all().
```

[ ]:
```
# This doesn't work. And despite using pandas for awhile, I still find I␣
↪regularly try and do this. The
# problem is that you have series objects, and python underneath doesn't know␣
↪how to compare two series using
# and or or. Instead, the pandas authors have overwritten the pipe | and␣
↪ampersand & operators to handle this
# for us
(df['chance of admit'] > 0.7) & (df['chance of admit'] < 0.9)
```

[25]:
```
# One thing to watch out for is order of operations! A common error for new␣
↪pandas users is
# to try and do boolean comparisons using the & operator but not putting␣
↪parentheses around
# the individual terms you are interested in
df['chance of admit'] > 0.7 & df['chance of admit'] < 0.9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File /opt/conda/lib/python3.9/site-packages/pandas/core/ops/array_ops.py:305, in␣
↪na_logical_op(x, y, op)
    296 try:
    297     # For exposition, write:
    298     #  yarr = isinstance(y, np.ndarray)
   (…)
```

```
    303     # Then Cases where this goes through without raising include:
    304     #  (xint or xbool) and (yint or bool)
--> 305     result = op(x, y)
    306 except TypeError:

File /opt/conda/lib/python3.9/site-packages/pandas/core/roperator.py:54, in␣
 ↪rand_(left, right)
     53 def rand_(left, right):
---> 54     return operator.and_(right, left)

TypeError: ufunc 'bitwise_and' not supported for the input types, and the input␣
 ↪could not be safely coerced to any supported types according to the casting␣
 ↪rule ''safe''

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
File /opt/conda/lib/python3.9/site-packages/pandas/core/ops/array_ops.py:319, i␣
 ↪na_logical_op(x, y, op)
    318 try:
--> 319     result = libops.scalar_binop(x, y, op)
    320 except (
    321     TypeError,
    322     ValueError,
   (…)
    325     NotImplementedError,
    326 ) as err:

File /opt/conda/lib/python3.9/site-packages/pandas/_libs/ops.pyx:180, in pandas
 ↪_libs.ops.scalar_binop()

ValueError: Buffer dtype mismatch, expected 'Python object' but got 'double'

The above exception was the direct cause of the following exception:

TypeError                                 Traceback (most recent call last)
Cell In [25], line 4
      1 # One thing to watch out for is order of operations! A common error for
 ↪new pandas users is
      2 # to try and do boolean comparisons using the & operator but not puttin␣
 ↪parentheses around
      3 # the individual terms you are interested in
----> 4 df['chance of admit'] > 0.7 & df['chance of admit'] < 0.9

File /opt/conda/lib/python3.9/site-packages/pandas/core/ops/common.py:72, in␣
 ↪_unpack_zerodim_and_defer.<locals>.new_method(self, other)
     68             return NotImplemented
     70 other = item_from_zerodim(other)
```

```
---> 72 return method(self, other)

File /opt/conda/lib/python3.9/site-packages/pandas/core/arraylike.py:76, in␣
↪OpsMixin.__rand__(self, other)
     74 @unpack_zerodim_and_defer("__rand__")
     75 def __rand__(self, other):
---> 76     return self._logical_method(other, roperator.rand_)

File /opt/conda/lib/python3.9/site-packages/pandas/core/series.py:6254, in␣
↪Series._logical_method(self, other, op)
   6251 lvalues = self._values
   6252 rvalues = extract_array(other, extract_numpy=True, extract_range=True)
-> 6254 res_values = ops.logical_op(lvalues, rvalues, op)
   6255 return self._construct_result(res_values, name=res_name)

File /opt/conda/lib/python3.9/site-packages/pandas/core/ops/array_ops.py:395, i␣
↪logical_op(left, right, op)
    391 # For int vs int `^`, `|`, `&` are bitwise operators and return
    392 #   integer dtypes.  Otherwise these are boolean ops
    393 filler = fill_int if is_self_int_dtype and is_other_int_dtype else␣
↪fill_bool
--> 395 res_values = na_logical_op(lvalues, rvalues, op)
    396 # error: Cannot call function of unknown type
    397 res_values = filler(res_values)  # type: ignore[operator]

File /opt/conda/lib/python3.9/site-packages/pandas/core/ops/array_ops.py:328, i␣
↪na_logical_op(x, y, op)
    320         except (
    321             TypeError,
    322             ValueError,
   (…)
    325             NotImplementedError,
    326         ) as err:
    327             typ = type(y).__name__
--> 328             raise TypeError(
    329                 f"Cannot perform '{op.__name__}' with a dtyped [{x.
↪dtype}] array "
    330                 f"and scalar of type [{typ}]"
    331             ) from err
    333 return result.reshape(x.shape)

TypeError: Cannot perform 'rand_' with a dtyped [float64] array and scalar of␣
↪type [bool]
```

[28]:

```python
# The problem is that Python is trying to bitwise and a 0.7 and a pandas␣
↪dataframe, when you really want
# to bitwise and the broadcasted dataframes together
```

[29]:
```python
# Another way to do this is to just get rid of the comparison operator␣
↪completely, and instead
# use the built in functions which mimic this approach
df['chance of admit'].gt(0.7) & df['chance of admit'].lt(0.9)
```

[29]:
```
Serial No.
1      False
2       True
3       True
4       True
5      False
       …
396     True
397     True
398    False
399    False
400    False
Name: chance of admit, Length: 400, dtype: bool
```

[30]:
```python
# These functions are build right into the Series and DataFrame objects, so you␣
↪can chain them
# too, which results in the same answer and the use of no visual operators. You␣
↪can decide what
# looks best for you
df['chance of admit'].gt(0.7).lt(0.9)
```

[30]:
```
Serial No.
1      False
2      False
3      False
4      False
5       True
       …
396    False
397    False
398    False
399     True
400    False
Name: chance of admit, Length: 400, dtype: bool
```

[31]:
```python
# This only works if you operator, such as less than or greater than, is built␣
↪into the DataFrame, but I
# certainly find that last code example much more readable than one with␣
↪ampersands and parenthesis.
```

[32]:
```
# You need to be able to read and write all of these, and understand the implications of the route you are
# choosing. It's worth really going back and rewatching this lecture to make sure you have it. I would say
# 50% or more of the work you'll be doing in data cleaning involves querying DataFrames.
```

In this lecture, we have learned to query dataframe using boolean masking, which is extremely important and often used in the world of data science. With boolean masking, we can select data based on the criteria we desire and, frankly, you'll use it everywhere. We've also seen how there are many different ways to query the DataFrame, and the interesting side implications that come up when doing so.