Getting Started With Python II

Getting Started *with Pandas:* Kaggle's Titanic Competition

To recap the last tutorial: we got comfortable with Python for re-implementing the models we originally imagined in Excel. By using a programming language, we were able to (1) use more powerful constructs and methods, like arrays to store and retrieve variables, and (2) to write scripted steps that can be repeated in the future without us performing the work by hand.

However, you may be thinking that you found it easier to understand what's in the data back when you were using Excel. (When you wanted to see a column, you just scrolled over to look at it, rather than counting through the indices 0 to 8.) On the other hand, you might have statistics friends who tell you that life is better with the software R, which has the concept of a "data.frame". Well, in this third tutorial we will take a slight detour from our modeling work in order to bridge that gap.

Python has another great package called Pandas, which makes data exploration and data cleaning much easier to do than manipulating arrays. It also lets you write code that's easier to read. Pandas has the concept of a DataFrame, too, which is like a spreadsheet with more programmatic power. Finally, if you go searching for additional tutorials in the forums someday, you'll often find that the author uses Pandas.

This tutorial is a little different than the first two: this is not a cohesive script to be run, nor part of a sample .py found on the Data Page. Instead, this tutorial is meant to entered line by line on your python command line, so that

you can learn some of the methods at your disposal and see what occurs. You might even deviate from this tutorial with other variables that interest you. Finally, at times the output from your command will be very long-winded, so not everything is printed in its entirety here.

Ready? If you have it installed, this would be a great time to utilize `ipython` or `ipython notebook`. Otherwise, run `python`.

Numpy Arrays

Let's review what our *train.csv* data looked like in python up to this point. Run the following to load the data again:

```
import csv as csv
import numpy as np

csv_file_object = csv.reader(open('../csv/train.csv', 'rb'))
header = csv_file_object.next()
data=[]

for row in csv_file_object:
    data.append(row)
data = np.array(data)
```

Now type `print data`

```
[['1' '0' '3' ..., '7.25' '' 'S']
 ['2' '1' '1' ..., '71.2833' 'C85' 'C']
 ['3' '1' '3' ..., '7.925' '' 'S']
 ...,
 ['889' '0' '3' ..., '23.45' '' 'S']
 ['890' '1' '1' ..., '30' 'C148' 'C']
 ['891' '0' '3' ..., '7.75' '' 'Q']]
```

This is familiar... an array of strings that the csv package was able to read.

Look at the first 15 rows of the Age column: `data[0:15,5]`

```
array(['22', '38', '26', '35', '35', '', '54', '2', '27',
'14', '4', '58', '20', '39', '14'],
 dtype='|S82')
```

Great, that command gives just the ages, and they are still stored as strings. What type of object is this whole column, though?

```
type(data[0::,5])
```

```
numpy.ndarray
```

So, any slice we take from the data is still a Numpy array. Now let's see if we can take the mean of the passenger ages. They will need to be floats instead of strings, so set this up as:

```
ages_onboard = data[0::,5].astype(np.float)
```

```
ValueError: could not convert string to float:
```

Hmm. This seemed to be working for the first few rows, but then produced an error when numpy got to the missing value ' ' in the 6th row. There is surely a way to use Python to filter out the missing values, then convert to float, then take the mean -- but this isn't sounding easy anymore. So let's try again with Pandas.

Pandas DataFrame

The first thing we have to do is import the Pandas package. It turns out that Pandas has its own functions to read or write a .csv file, so we are no longer actually using the **csv package** in the commands below. Let's create a new object called 'df' for storing the pandas version of *train.csv*. (This means you can still refer to the original 'data' numpy array for the rest of this tutorial anytime you want to compare and contrast.)

```
import pandas as pd
import numpy as np

# For .read_csv, always use header=0 when you know row 0 is the
header row
df = pd.read_csv('train.csv', header=0)
```

Now look at what's there by typing:

```
df
```

```
(...long list of stuff...!)
...
...
...
891 rows × 12 columns
```

That wasn't so helpful. So let's look at just the first few rows:

```
df.head(3)
```

```
(a short list of stuff!)
...
3 rows × 12 columns
```

You notice it has column names, and it has the index of rows labelled down the side. (Note: you can also try `df.tail(3)` and you can feed it any number of rows.) Now, compared to the original data array, what kind of object is this?

```
type(df)
```

```
pandas.core.frame.DataFrame
```

Recall that using the **csv package** before, every value was interpreted as a string. But how does Pandas interpret them using its own csv reader?

```
df.dtypes
```

```
PassengerId  int64
Survived     int64
Pclass       int64
Name        object
Sex         object
Age        float64
SibSp        int64
Parch        int64
Ticket      object
Fare       float64
Cabin       object
Embarked    object
```

```
dtype: object
```

Pandas is able to infer numerical types whenever it can detect them. So we have values already stored as integers. When it detected the existing decimal points somewhere in Age and Fare, it converted those columns to float. There are two more very valuable commands to learn on a dataframe:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
```

There's a lot of useful info there! You can see immediately we have 891 entries (rows), and for most of the variables we have complete values (891 are non-null). But not for Age, or Cabin, or Embarked -- those have nulls somewhere. Now try:

```
df.describe()
```

```
       PassengerId    Survived      Pclass         Age  ...
count   891.000000  891.000000  891.000000  714.000000  ...
mean    446.000000    0.383838    2.308642   29.699118  ...
std     257.353842    0.486592    0.836071   14.526497  ...
min       1.000000    0.000000    1.000000    0.420000  ...
25%     223.500000    0.000000    2.000000   20.125000  ...
50%     446.000000    0.000000    3.000000   28.000000  ...
75%     668.500000    1.000000    3.000000   38.000000  ...
max     891.000000    1.000000    3.000000   80.000000  ...
```

This is also very useful: pandas has taken all of the

numerical columns and quickly calculated the mean, std, minimum and maximum value. Convenient! But also a word of caution: we know there are a lot of missing values in Age, for example. How did pandas deal with that? It must have left out any nulls from the calculation. So if we start quoting the "average age on the Titanic" we need to caveat how we derived that number.

Data Munging

One step in any data analysis is the data cleaning. Thankfully pandas makes things easier to filter, manipulate, drop out, fill in, transform and replace values inside the dataframe. Below we also learn the syntax that pandas allows for referring to specific columns.

Referencing and filtering

Let's acquire the first 10 rows of the Age column. In pandas this is

```
df['Age'][0:10]
```

```
0 22
1 38
2 26
3 35
4 35
5 NaN
6 54
7 2
8 27
9 14
Name: Age, dtype: float64
```

--> And try this alternative syntax: `df.Age[0:10]`

--> Without counting indices, can you show the Cabin column now?

Similar to before, let's understand what kind of object is this. Type:

```
type(df['Age'])
```

```
pandas.core.series.Series
```

A single column is neither an numpy array, nor a pandas dataframe -- but rather a pandas-specific object called a data Series.

At this point, we'd really like to get than mean value:

```
df['Age'].mean()
```

```
29.69911764705882
```

That matches what was reported in df.describe().

--> See if you can obtain the .`median` of Age as well.

The next thing we'd like to do is look at more specific subsets of the dataframe. Again pandas makes this very convenient to write. Pass it a [list] of the columns desired:

```
df[ ['Sex', 'Pclass', 'Age'] ]
```

```
   Sex    Pclass Age
0  male   3      22.0
1  female 1      38.0
2  female 3      26.0
3  female 1      35.0
4  male   3      35.0
5  male   3       NaN
   ...    ...    ...
   ...    ...    ...

[891 rows x 3 columns]
```

Filtering data is another important tool if we are investigating the data by hand. The .describe() command had indicated that the maximum age was 80. What do the older passengers look like in this data set? This is written by passing the criteria of df as a *where* clause into df:

```
df[df['Age'] > 60]
```

```
(a medium list of stuff!)
...
...
22 rows × 12 columns
```

If you were most interested in the mix of the gender and Passenger class of these older people, you would want to combine the two skills you just learned and get only a few columns for the same *where* filter:

```
df[df['Age'] > 60][['Sex', 'Pclass', 'Age', 'Survived']]
```

```
     Sex    Pclass Age   Survived
33   male   2      66.0  0
54   male   1      65.0  0
96   male   1      71.0  0
116  male   3      70.5  0
170  male   1      61.0  0
252  male   1      62.0  0
275  female 1      63.0  1
280  male   3      65.0  0
     ...    ...    ...   ...
     ...    ...    ...   ...
[22 rows x 4 columns]
```

From visual examination of all 22 cases, it seems they were mostly men, mostly(?) 1st class, and mostly perished.

Now it's time to investigate all of those missing Age values, because we will need to address them in our model if we hope to use all the data for more advanced algorithms. To filter for missing values, use

```
df[df['Age'].isnull()][['Sex', 'Pclass', 'Age']]
```

```
(a long list of stuff!)
...
...
...
177 rows × 3 columns
```

Here the only thing we did was print all 177 cases, but the same syntax can be used later if we take action on them.

It will also be useful to combine multiple criteria (joined by the syntax &). To practice even more functionality in the same line of code, let's take a count of the males in each class.

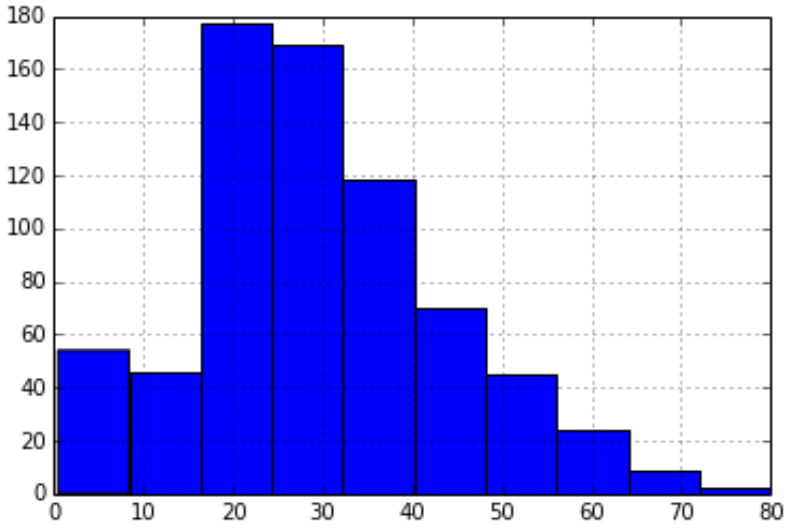
```
for i in range(1,4):
```

```
    print i,  len(df[ (df['Sex'] == 'male') & (df['Pclass']
== i) ])
```

```
1 122
2 108
3 347
```

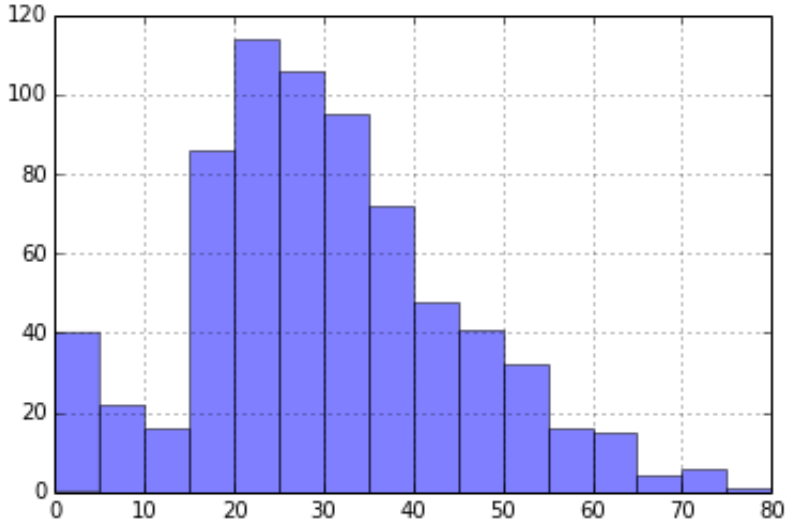
Before we finish the initial investigation by hand, let's use one other convenience function of pandas to derive a histogram of any numerical column. The histogram function is really a shortcut to the more powerful features of the matplotlib/pylab packages, so let's be sure that's imported. Type the following:

```
import pylab as P
df['Age'].hist()
P.show()
```



Inside the parentheses of .hist(), you can also be more explicit about options of this function. Before you invoke it, you can also be explicit that you are dropping the missing values of Age:

```
df['Age'].dropna().hist(bins=16, range=(0,80), alpha = .5)
P.show()
```



Cleaning the data

Ok now that we are comfortable with the syntax, we are ready to begin transforming the values in the dataframe into the shape we need for machine learning. First of all, it's hard to run analysis on the string values of "male" and "female". Let's practice transforming it in three ways -- twice for fun and once to make it useful. We'll store our transformation in a new column, so the original Sex isn't changed.

In Pandas, adding a column is as easy as naming it and passing it new values.

```
df['Gender'] = 4
```

Show some .head() rows of the dataframe to see what we just accomplished. Well, now let's make it mean something that's actually derived from the Sex column.

```
df['Gender'] = df['Sex'].map( lambda x: x[0].upper() )
```

lambda x is an built-in function of python for generating an anonymous function in the moment, at runtime. Remember that x[0] of any string returns its first character.

What do the .head() rows of the dataframe look like now?

But of course what we really need is a binary integer for

female and male, similar to the way Survived is stored. As a matter of consistency, let's also make Gender into values of 0 and 1's. We have a precedent of analyzing the women first in all of our previous arrays, so let's decide female = 0 and male = 1. So, for real this time:

```
df['Gender'] = df['Sex'].map( {'female': 0, 'male': 1}
).astype(int)
```

See what the .head() rows of the dataframe look like now.

--> Can you make a new column to do something similar for the Embarked values?

Now it's time to deal with the missing values of Age, because most machine learning will need a complete set of values in that column to use it. By filling it in with guesses, we'll be introducing some noise into a model, but if we can keep our guesses reasonable, some of them should be close to the historical truth (whatever it was...), and the overall predictive power of Age might still make a better model than before. We know the average [known] age of all passengers is 29.6991176 -- we could fill in the null values with that. But maybe the median would be better? (to reduce the influence of a few rare 70- and 80-year olds?) The Age histogram did seem positively skewed. These are the kind of decisions you make as you create your models in a Kaggle competition.

For now let's decide to be more sophisticated, that we want to use the age that was typical in each passenger class. And decide that the median might be better. Let's build another reference table to calculate what each of these medians are:

```
median_ages = np.zeros((2,3))
median_ages
```

yields:

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

And then populating the array,

```
for i in range(0, 2):
    for j in range(0, 3):
        median_ages[i,j] = df[(df['Gender'] == i) & \
                              (df['Pclass'] == j+1)]
['Age'].dropna().median()

median_ages
```

yields:

```
array([[ 35. , 28. , 21.5],
       [ 40. , 30. , 25. ]])
```

We could fill in the missing ages directly into the Age column. But to be extra cautious and not lose the state of the original data, a more formal way would be to create a new column, AgeFill, and even record which ones were originally null (and thus artificially guessed).

Make a copy of Age:

```
df['AgeFill'] = df['Age']

df.head()
```

Take a look at just the rows with missing values, and limit it to the columns important to us right now:

```
df[ df['Age'].isnull() ]
[['Gender','Pclass','Age','AgeFill']].head(10)
```

```
    Gender Pclass Age AgeFill
5  1      3      NaN     NaN
17 1      2      NaN     NaN
19 0      3      NaN     NaN
26 1      3      NaN     NaN
28 0      3      NaN     NaN
29 1      3      NaN     NaN
31 0      1      NaN     NaN
32 0      3      NaN     NaN
36 1      3      NaN     NaN
42 1      3      NaN     NaN
```

Use some code to fill in AgeFill based on our median_ages table. Here we happen to use the alternate syntax for referring to an existing column, like df.Age rather than df['Age']. There's a *where* clause on df and referencing its column AgeFill, then assigning it an appropriate value out of median_ages.

```
for i in range(0, 2):
    for j in range(0, 3):
        df.loc[ (df.Age.isnull()) & (df.Gender == i) &
(df.Pclass == j+1),\
                'AgeFill'] = median_ages[i,j]
```

View the exact same 10 rows we just looked at:

```
df[ df['Age'].isnull() ]
[['Gender','Pclass','Age','AgeFill']].head(10)
```

```
   Gender Pclass Age AgeFill
5  1      3      NaN    25.0
17 1      2      NaN    30.0
19 0      3      NaN    21.5
26 1      3      NaN    25.0
28 0      3      NaN    21.5
29 1      3      NaN    25.0
31 0      1      NaN    35.0
32 0      3      NaN    21.5
36 1      3      NaN    25.0
42 1      3      NaN    25.0
```

This confirms we accomplished exactly what we wanted.

Let's also create a feature that records whether the Age was originally missing. This is relatively simple by allowing pandas to use the integer conversion of the True/False evaluation of its built-in function, pandas.isnull()

```
df['AgeIsNull'] = pd.isnull(df.Age).astype(int)
```

Now that we have 3 new numerical columns, Gender, AgeFill, AgeIsNull... perhaps you want to run `df.describe()` to see the summary statistics of the whole dataframe again.

Feature Engineering

Let's create a couple of other features, this time using simple math on existing columns. Since we know that Parch is the number of parents or children onboard, and SibSp is the number of siblings or spouses, we could collect those together as a FamilySize:

```
df['FamilySize'] = df['SibSp'] + df['Parch']
```

We can also create artificial features if we think it may be advantageous to a machine learning algorithm -- of course, it might not. For example, we know Pclass had a large effect on survival, and it's possible Age will too.
One artificial feature could incorporate whatever predictive power might be available from both Age and Pclass by multiplying them. This amplifies 3rd class (3 is a higher multiplier) at the same time it amplifies older ages. Both of these were less likely to survive, so in theory this could be useful.

```
df['Age*Class'] = df.AgeFill * df.Pclass
```

We could make some histograms of our new columns to understand them better. Go back and find the .hist() commands above.

We know we'd like to have better predictive power for the men, so you might be wishing you could pull out more information from the Name column -- for example the honorary or pedestrian title of the men? We won't accomplish that in this tutorial, but you may find ideas in the Kaggle forums.

Final preparation

We have our data almost ready for machine learning. But most basic ML techniques will not work on strings, and in python they almost always require the data to be an array-- the implementations we will see in the **sklearn package** are not written to use a pandas dataframe. So the last two things we need to do are (1) determine what columns we have left which are not numeric, and (2) send our pandas.DataFrame back to a numpy.array.

In pandas you could always see the column types from the `.info()` method. You can also see them directly:

```
df.dtypes
```

```
PassengerId   int64
Survived      int64
Pclass        int64
Name         object
Sex          object
Age         float64
SibSp         int64
Parch         int64
Ticket       object
Fare        float64
Cabin        object
Embarked     object
Gender        int64
AgeFill     float64
AgeIsNull     int64
FamilySize    int64
Age*Class   float64
dtype: object
```

With a little manipulation, we can require .dtypes to show only the columns which are 'object', which for pandas means it has strings:

```
df.dtypes[df.dtypes.map(lambda x: x=='object')]
```

```
Name        object
Sex         object
Ticket      object
Cabin       object
Embarked    object
dtype: object
```

(You may already have already transformed 'Embarked' in your own work above.)

The next step is to drop the columns which we will not use:

```
df = df.drop(['Name', 'Sex', 'Ticket', 'Cabin', 'Embarked'],
axis=1)
```

We can also drop 'Age' even though it is numeric, since we

copied and filled that to a better column AgeFill. The original 'Age' still has the missing values which won't work well in our future model.

```
df = df.drop(['Age'], axis=1)
```

An alternate command is to drop any rows which still have missing values:

```
df = df.dropna()
```

But remember that . `dropna()` removes an observation from df even if it only has 1 NaN, anywhere, in any of its columns. It could delete most of your dataset if you aren't careful with the state of missing values in other columns!

Now we have a clean and tidy dataset that is ready for analysis.

The final step is to convert it into a Numpy array. Pandas can always send back an array using the . `values` method. Assign to a new variable, train_data:

```
train_data =  df.values
train_data
```

```
array([[ 1. , 0. , 3. , ..., 0. , 1. , 66. ],
       [ 2. , 1. , 1. , ..., 0. , 1. , 38. ],
       [ 3. , 1. , 3. , ..., 0. , 0. , 78. ],
       ...,
       [ 889. , 0. , 3. , ..., 1. , 3. , 64.5],
       [ 890. , 1. , 1. , ..., 0. , 0. , 26. ],
       [ 891. , 0. , 3. , ..., 0. , 0. , 96. ]])
```

Compare that to the original array we used in the last tutorial! Type:

```
data
```

```
 array([['1' '0' '3' ..., '7.25' '' 'S']
```

```
       ['2' '1' '1' ..., '71.2833' 'C85' 'C']
       ['3' '1' '3' ..., '7.925' '' 'S']
       ...,
       ['889' '0' '3' ..., '23.45' '' 'S']
       ['890' '1' '1' ..., '30' 'C148' 'C']
       ['891' '0' '3' ..., '7.75' '' 'Q']],
      dtype='|S82')
```

Conclusion

Here we used Pandas as a much easier tool to access the data in a csv file and manipulate it.

If you found this tutorial helpful but would like more practice, you could re-do the previous Python tutorial, but this time write your script using pandas. (In fact, one Kaggler did just that in the forums. At the very least, you should be able to read and understand his script.)

In the next tutorial we will take advantage of your new skills with python, in order to apply Machine Learning on this data for the first time.