Getting Started With Excel

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

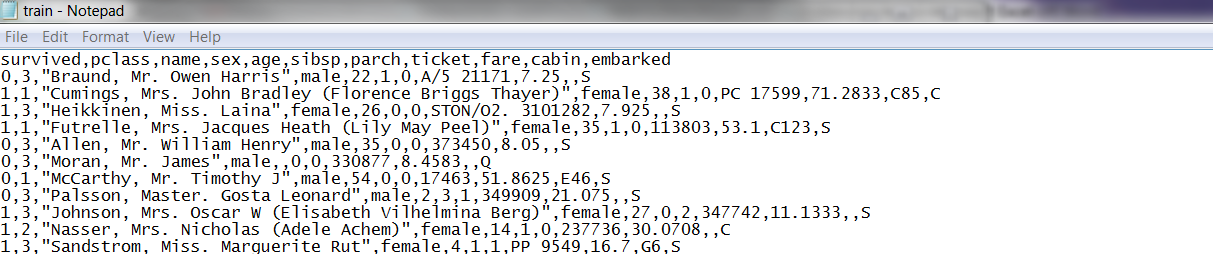
For those who are not experienced with handling large data sets, logging into the Kaggle website for the first time may be slightly daunting. Many of these competitions have a six figure prize and data which can, at times, be extremely involved. Here at Kaggle, we understand that this may seem like an insurmountable barrier to entry, so we have created a "getting started" competition to guide you through the initial steps required to get your first decent submission on the board.

The challenge

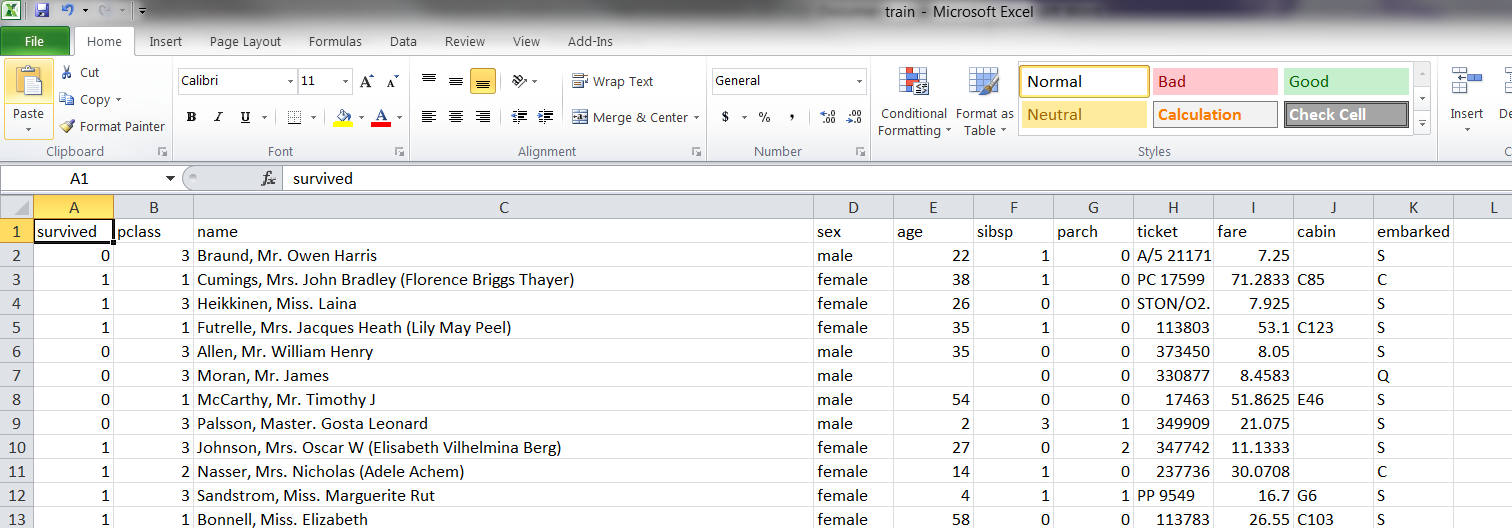
The competition is simple: we want you to use the Titanic passenger data (name, age, gender, socio-economic class, etc) to try to predict who will survive and who will die.

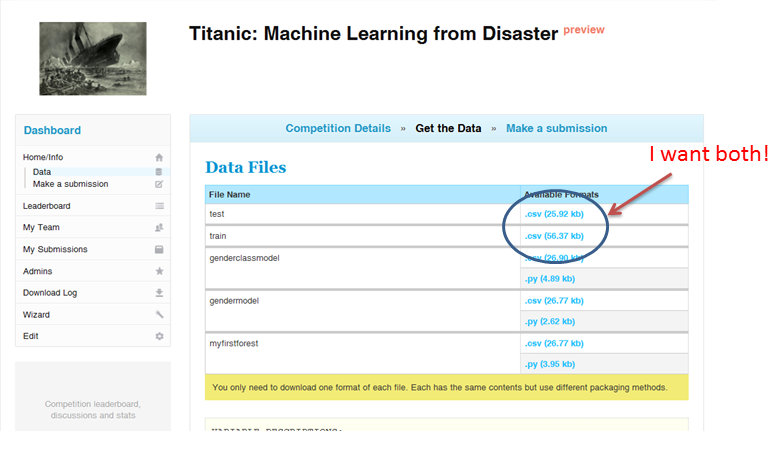
I want to compete! What do I do next?

The first thing to do is get the data from the Kaggle website. You will need two files: *train.csv* and *test.csv*. The .csv filename stands for comma separated values, where every value (name, age, gender, etc) in a row is separated by a comma. This allows Excel to interpet the data as columns.  If you load the file up in notepad--or your text editor of choice--it would look like this:



 However, in Excel it looks like this:

To get the data…Click on [Get The Data](https://www.kaggle.com/c/titanic/data), and this will take you to the Data page:



In order to download, click on the blue file extensions (.csv). The first time you do this you will be taken to the rules acceptance page. You must accept the competition rules in order to download the data. These rules govern how many submissions you can make per day, the maximum team size, and other competition specific details. Click "I Understand and Accept" and then re-click the filenames.  You will need both files.

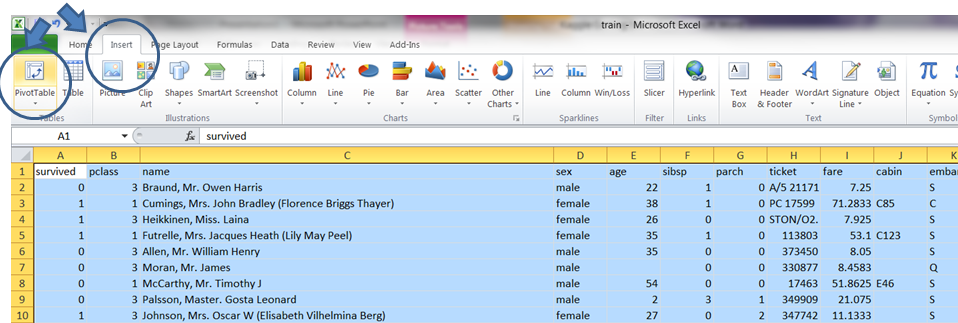
I have the data what do I have to do?!

You have two files, *train.csv* and *test.csv*. *Train.csv* will contain the details of a subset of the passengers on board (891 to be exact) and will tell you their details and whether they survived or not. Using the patterns you find in the *train.csv* data, you will have to predict whether the other 418 passengers on board (found in *test.csv*) survived.

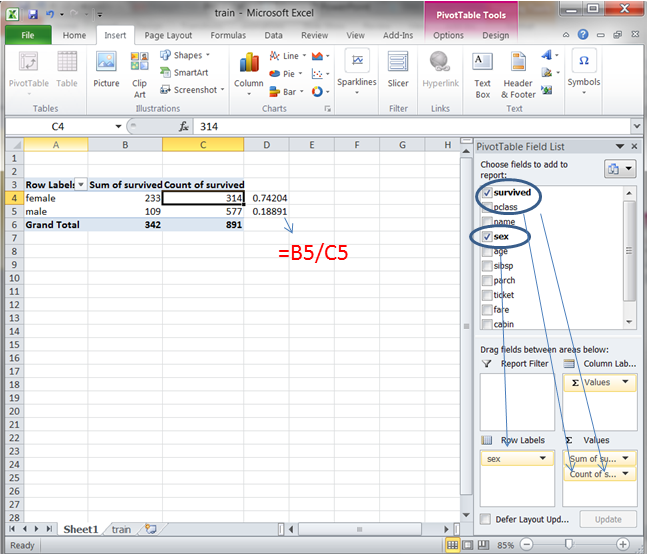
So I need to find patterns in the data… Where do I start?!

Open up *train.csv* in Excel. As you can see,we have given you the details of a number of passengers on board the Titanic. We have given you the Pclass, the class of their ticket (1st, 2nd or 3rd), their name, their sex, age, Subsp (which is the number of siblings / spouses they had on board with them), Parch (which is the number of parents / children they had on board with them), their ticket type, the fare they paid for their ticket, their cabin number, and Embarcation point (where they got on: Queenstown, Cherbourg or Southampton). We also give you whether they survived or not (1 = yes, 0 = sadly, not). This is the information you are going to use to make your predictions. We want to find if there is a relationship between one of the variables and ultimate survival.

In data science, your intuition is often a great place to start. If you have ever seen the film *Titanic*, they try to save the women and children first.  This would be a good guess to start! Excel has a helpful tool for this kind of exploration called a pivot table. Highlight the entire set of data and go to 'Insert'--> 'Pivot Tables'.



This should create a new spreadsheet in your document. On the right should be all the variables that you selected and four boxes at the bottom. We are interested in who survived, so you want to see how this value varies as you select other variables. On the right hand side, drag the word 'survived', which has a check box next to it, down to the bottom right hand box. This will show the sum of the survived box. Now, to see how many women and men survived, drag the 'Sex' variable to the 'Row labels'. Since Survived is a 1 or 0, the sum of this is the total number who survived. If you want to find the proportion, drag the Survived variable again into the values box (so now there are two in there) click on the down arrow, select value field settings, and change 'Sum' to 'count'. This will tell you the total number of rows (or passengers in this case). Now, in the cell next to the table you can just type = B4 / C4, which would show the proportion on females that survived.

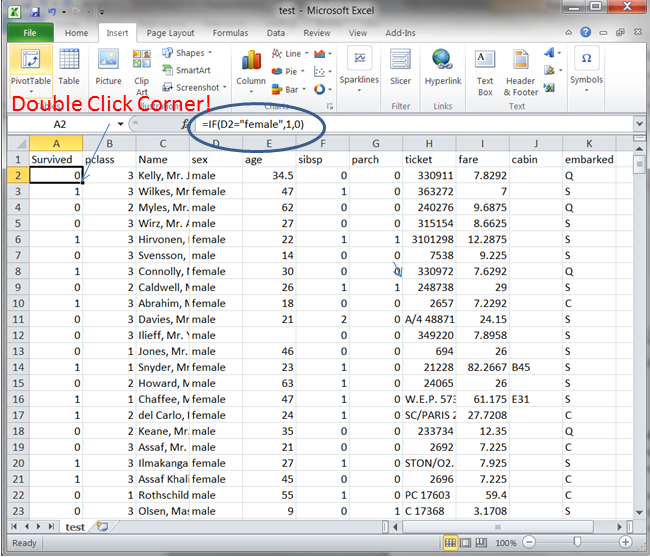


From this you can see that almost 75% of the females survived! However, only 19% of the males lived to tell about it. This is quite a promising first guess!

Making my predictions

Gender seems to be a strong indicator of survival, with women having a much better chance! To make your first predictions based on this, now open *test.csv* in Excel and insert a new column in the first column, and give it the header Survived. *Your submission to Kaggle MUST have your predictions in a column named 'Survived'* [*(more info here)*](https://www.kaggle.com/c/titanic/details/submission-instructions)*!*

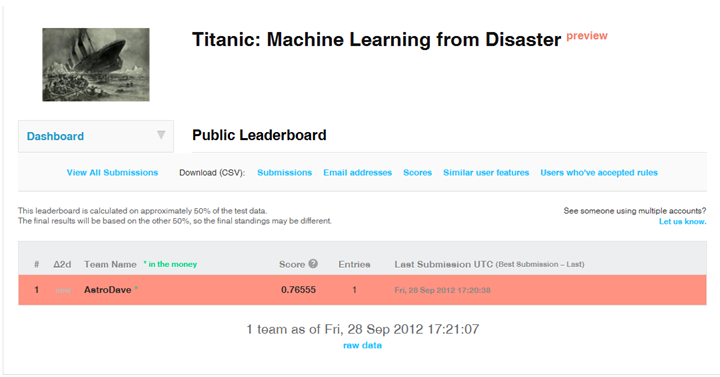
To make a model which states that if the passenger is female then she survives in the first cell write a logical 'if' statement. Type =if(E2="female",1,0) which means if(sex=female, then make this cell = 1, if not then make this cell = 0). Then double click on the bottom right hand corner of this box so it drags down through all rows. You've now created a calculated formula for each row in Excel, but for Kaggle you want a definitive value, 0 or 1. So re-paste the same column onto itself with Paste As > Values.  Save the file as something memorable. I'll use 'genderbasedmodel.csv'.



I’ve made my predictions, how well have I done?

One last step:  Kaggle can only accept 2 columns in your submitted answer, PassengerId and Survived. Delete any extra columns, and save (or Save As) this new version of your csv which only has those 2 columns. Now go back to your internet browser and the competition page and click 'Make a submission'.

* This should bring up your team page. This is where you can set up your team. (On Kaggle you are always a member of a team, whether it is one person or 20 people.) Each team has a team leader. Teams can be added to at a later date, however members *cannot be removed,* so choose carefully! You can compete anonymously as well, if don’t want people to see your name.
* Hit 'Continue' and the first thing you may notice is 'You have 2 (of 2 entries) left today'. Kaggle limits the number of submissions you make so you can't use the leaderboard score to gain and unfair advantage with your submissions. Click on the button 'Click or Drop Submission Here' and select the file you would like to submit (here, 'genderbasedmodel.csv') and click 'Submit'.



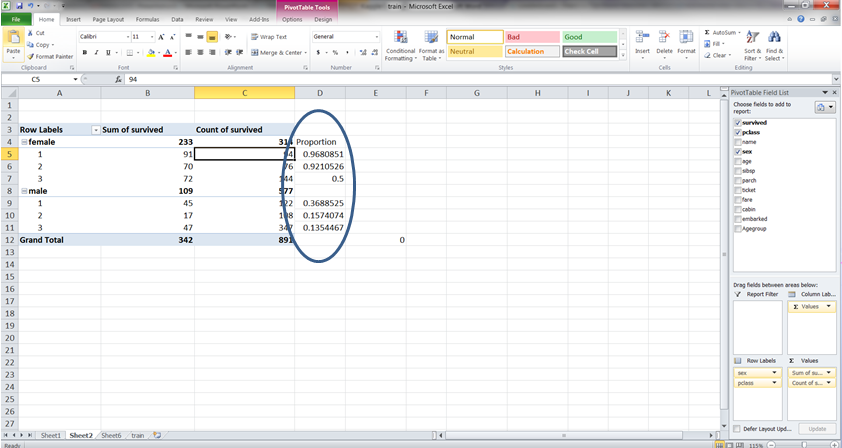
* Once you have submitted, you will be taken to the leaderboard and shown your score and ranking on the leaderboard. Each competition is scored based on different evaluation metrics, whose details are described on the [Evaluation page](https://www.kaggle.com/c/titanic/details/evaluation).  In this competition the metric is simply the fraction of passengers you got correct.

My second submission: I want to do better!

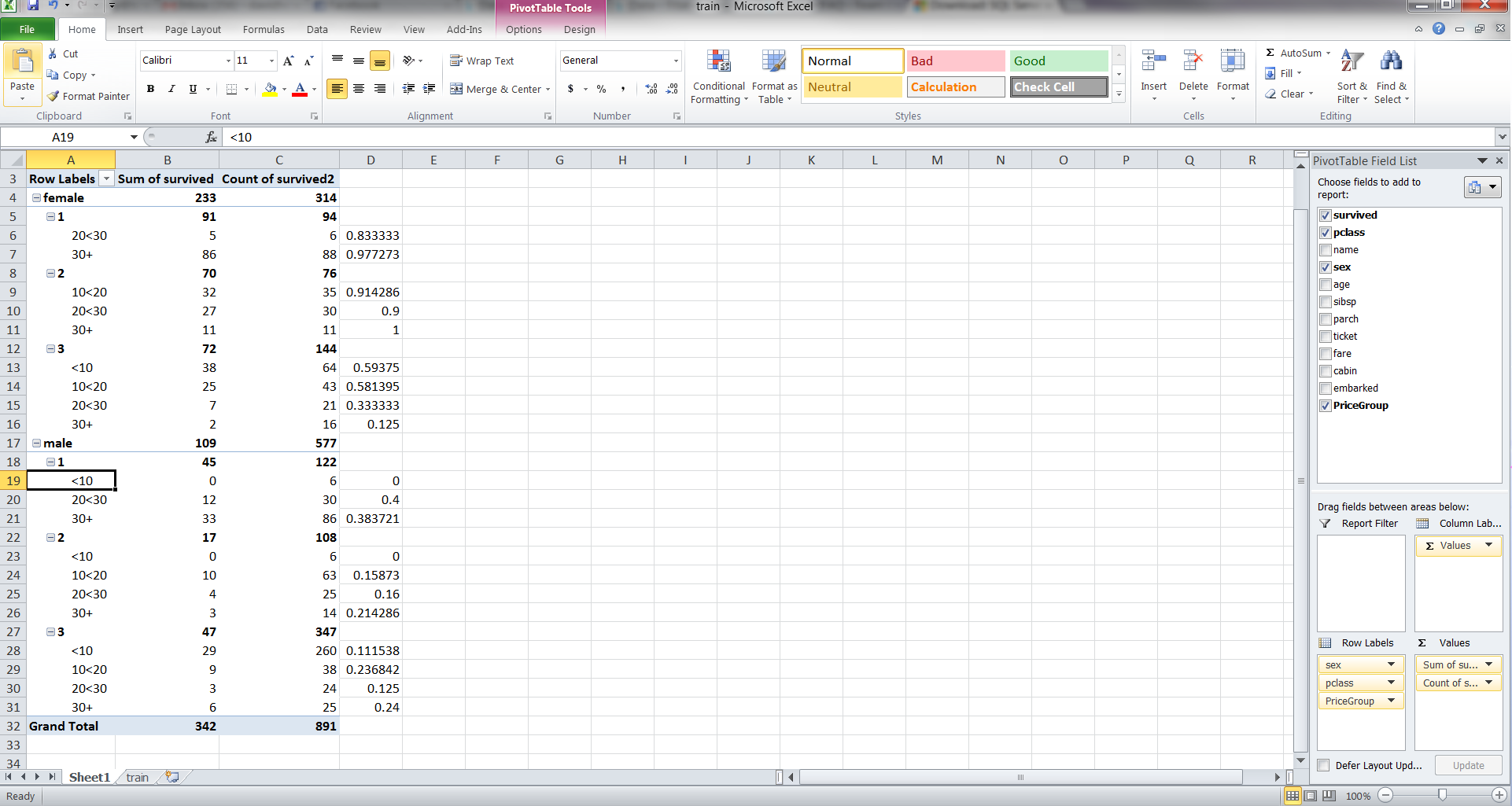
Your first submission will likely not be your best at the end of the competition. As the competition grows and evolves, people learn about the data, posts appear on the forum discussing techniques which give insight, and you will think of new ways to improve your model. Let's try to improve ours.

Going back to the *train.csv*, you want to improve what you currently have. Extending the earlier hypothesis that maybe age may also be predictive, you can add age to the pivot table. Drag age to the Row labels (where sex was) and it will show the number of passengers who were match on both gender and an age. However, as you may notice, the age is not binned up in any way. You may want to group these up, perhaps starting with just adults and children. Just like before, make an extra column with a 'if' statement. In this case, it will read =IF(F2>18,"adult","child"), where F2 was the age column. Now recreate the pivot table (note you have to re-make it) and add the same variables before, you can see if there are more patterns!

According to the table, adult women (over 18) had a 78% chance of survival and male adults only had an 18% chance of survival. You can see that this isn’t much changed from the original proportions. This tells us that there is not much additional information in the age variable. Let's look at one more variable: the class of passenger. Adding this to the pivot table we can see the results:



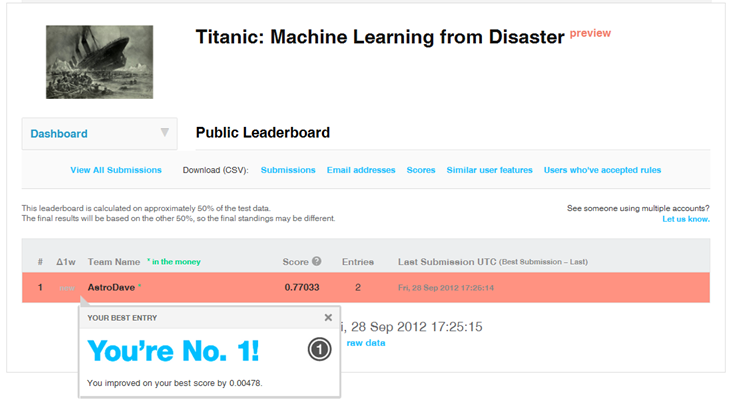
As you can see, the proportions have now changed dramatically, meaning that there is some predictive merit to this variable. However, this still doesn't surpass the male/female divide. Now let's bin up what people paid for their ticket, so that each class is split into payments of: (i) less than $10, (ii) between $10 and $20, (iii) between $20 and $30, and (iv) greater than $30:



Now we have the proportions for many different variables. I decide to just assume that any group with more than half survivors I will model to always survive, and all those with less than half will model to never survive. So as you can see, all the males will still not survive; however now women in third class who paid more than $20 will also not survive. This small improvement should make a difference on the leaderboard! So now going to *test.csv*  I do a nested IF statement in each cell of a new first column I create called Survived:

=IF(E2="male",0,IF(C2=3,IF(J2>20,0,1),1))

and then re-paste my formula column with Paste As Values, reduce my file to only 2 columns, Survived & PassengerId, then save to a csv file, then submit! As you see, we now differentiated between those who paid a lot for their third class ticket, and those who did not, and look at the effect on my submitted score!



Conclusion

Here Excel was used to get the basic understanding of the data. From just this basic bit of work, we have found that people did not necessarily pay the same amount of money for a ticket; some people paid more for a third class than a first class!

This concludes our first tutorial. We have been able to download the data and make a simple model based on our initial understanding of the problem. We made a submission to Kaggle, and then built on it and made an improvement.   As we incorporate more models and more complicated ideas, we may want to move to something slightly more sophisticated than Excel. [The next tutorial](https://www.kaggle.com/c/titanic/details/getting-started-with-python) will look at using Python, a simple scripting language which can help automate our analysis.

Getting Started With Python

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

Recapping our work with Excel: we have been able to successfully download the competition data and submit 2 models: one based on just the gender and another on the gender, class, and price of ticket. This is good for an initial submission, however problems arise when we move to slightly more complicated models and the time taken to formulate approaches in Excel takes longer. What do you do if you want to make a more complicated model but don’t have the time to manually find the proportion of survivors for a given variable? We should make the computer do the hard work for us!

I want to add more variables but it takes so much time!

Programming scripts are a great way to speed up your calculations and avoid the arduous task of manually calculating the pivot table ratios. There are many languages out there, each with its own advantages and disadvantages. Here we are going to use Python version 2.7, which is an easy to use scripting language. If you do not have this installed, please visit the [Python web site](https://www.python.org/downloads/) and folllow the instructions there -- or, you could install one specific distribution of Python called [Anaconda](http://continuum.io/downloads) that already bundles the most useful libraries for data science. (Another advantage of Anaconda is that it includes iPython (Interactive Python) which makes the interface easier for stepping through lines of programming one by one.)

NOTE in either case: if you use Python version 3.x, you may discover some Python syntax has changed in that version, which can cause errors on this tutorial [as people point out in the forum](https://www.kaggle.com/c/titanic-gettingStarted/forums/t/4937/titanic-problem-with-getting-started-with-python).

When you have things installed, to begin just type  python , or  ipython , or ipython notebook.

One of the great advantages of Python is its packages. Of these packages, the most useful (for Kaggle competitions) are the Numpy, Scipy, Pandas, matplotlib and csv package. In order to check whether you have these, just go to your python command line and type  import numpy  (and so on). If you don’t you will need them! This tutorial is going to guide you through making the same submissions as before, only this time using Python.

Python: Reading in your *train.csv*

Python has a nice csv reader, which reads each line of a file into memory. You can read in each row and just append a list. From there, you can quickly turn it into an array.

# The first thing to do is to import the relevant packages

# that I will need for my script,

# these include the Numpy (for maths and arrays)

# and csv for reading and writing csv files

# If i want to use something from this I need to call

# csv.[function] or np.[function] first

import csv as csv

import numpy as np

# Open up the csv file in to a Python object

csv\_file\_object = csv.reader(open('../csv/train.csv', 'rb'))

header = csv\_file\_object.next() # The next() command just skips the

# first line which is a header

data=[] # Create a variable called 'data'.

for row in csv\_file\_object: # Run through each row in the csv file,

data.append(row) # adding each row to the data variable

data = np.array(data) # Then convert from a list to an array

# Be aware that each item is currently

# a string in this format

Although you've seen this data before in Excel, just to be sure let's look at how it is stored now in Python.  Type  print data  and the output should be something like

[['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']]

You can see this is an array with just values (no descriptive header). And you can see that each value is being shown in quotes, which means it is stored as a string. Unfortunately in the output above, the full set of columns is being obscured with "...," so let's print the first row to see it clearly.  Type  print data[0]

['1' '0' '3' 'Braund, Mr. Owen Harris' 'male' '22' '1' '0' 'A/5 21171' '7.25' '' 'S']

and to see the last row, type  print data[-1]

['891' '0' '3' 'Dooley, Mr. Patrick' 'male' '32' '0' '0' '370376' '7.75' '' 'Q']

and to see the 1st row, 4th column, type  print data[0,3]

Braund, Mr. Owen Harris

I have my data now I want to play with it

Now if you want to call a specific column of data, say, the gender column, I can just type data[0::,4], remembering that "0::" means all (from start to end), and Python starts indices from 0 (not 1). You should be aware that the csv reader works by default with strings, so you will need to convert to floats in order to do numerical calculations. For example, you can turn the Pclass variable into floats by using data[0::,2].astype(np.float). Using this, we can calculate the proportion of survivors on the Titanic:

# The size() function counts how many elements are in

# in the array and sum() (as you would expects) sums up

# the elements in the array.

number\_passengers = np.size(data[0::,1].astype(np.float))

number\_survived = np.sum(data[0::,1].astype(np.float))

proportion\_survivors = number\_survived / number\_passengers

Numpy has some lovely functions. For example, we can search the gender column, find where any elements equal female (and for males, 'do not equal female'), and then use this to determine the number of females and males that survived:

women\_only\_stats = data[0::,4] == "female" # This finds where all

# the elements in the gender

# column that equals “female”

men\_only\_stats = data[0::,4] != "female" # This finds where all the

# elements do not equal

# female (i.e. male)

We use these two new variables as a "mask" on our original train data, so we can select only those women, and only those men on board, then calculate the proportion of those who survived:

# Using the index from above we select the females and males separately

women\_onboard = data[women\_only\_stats,1].astype(np.float)

men\_onboard = data[men\_only\_stats,1].astype(np.float)

# Then we finds the proportions of them that survived

proportion\_women\_survived = \

np.sum(women\_onboard) / np.size(women\_onboard)

proportion\_men\_survived = \

np.sum(men\_onboard) / np.size(men\_onboard)

# and then print it out

print 'Proportion of women who survived is %s' % proportion\_women\_survived

print 'Proportion of men who survived is %s' % proportion\_men\_survived

Now that I have my indication that women were much more likely to survive, I am done with the training set.

Reading the test data and writing the gender model as a csv

As before, we need to read in the test file by opening a python object to read and another to write. First, we read in the *test.csv* file and skip the header line:

test\_file = open('../csv/test.csv', 'rb')

test\_file\_object = csv.reader(test\_file)

header = test\_file\_object.next()

Now, let's open a pointer to a new file so we can write to it (this file does not exist yet). Call it something descriptive so that it is recognizable when we upload it:

prediction\_file = open("genderbasedmodel.csv", "wb")

prediction\_file\_object = csv.writer(prediction\_file)

We now want to read in the test file row by row, see if it is female or male, and write our survival prediction to a new file.

prediction\_file\_object.writerow(["PassengerId", "Survived"])

for row in test\_file\_object: # For each row in test.csv

if row[3] == 'female': # is it a female, if yes then

prediction\_file\_object.writerow([row[0],'1']) # predict 1

else: # or else if male,

prediction\_file\_object.writerow([row[0],'0']) # predict 0

test\_file.close()

prediction\_file.close()

Now you have a file called 'genderbasedmodel.csv', which you can submit!

On the Data page you will find all of the steps above in a single python script named 'gendermodel.py'. One advantage of python is that you can quickly run all of the steps you did again in the future -- if you receive a new training file, for example.

Pythonising the second submission

By now you have created your first python submission. Let's complicate things and try and submit the same submission as before, binning up the ticket price into the four bins and modeling the outcome on class, gender, and ticket price. This part assumes that you have completed the section 'Reading in your train.csv' and you have a data array as before.  On the Data page you will find a python script named 'genderclassmodel.py' to follow along... but be sure to type (or paste) each line of code yourself, to help you learn what is happening.

The idea is to create an table which contains just 1's and 0's. The array will be a survival reference table, whereby you read in the test data, find out passenger attributes, look them up in the survival table, and determine if they should be predicted to survive or not. In the case of a model that uses gender, class, and ticket price, you will need an array of 2x3x4 ( [female/male] , [1st / 2nd / 3rd class], [4 bins of prices] ).

The script will systematically will loop through each combination and use the 'where' function in python to search the passengers that fit that combination of variables. Just like before, you can ask what indices in your data equals female, 1st class, and paid more than $30. The problem is that looping through requires bins of equal sizes, i.e. $0-9,  $10-19,  $20-29,  $30-39.  For the sake of binning let's say everything equal to and above 40 "equals" 39 so it falls in this bin. So then you can set the bins:

# So we add a ceiling

fare\_ceiling = 40

# then modify the data in the Fare column to = 39, if it is greater or equal to the ceiling

data[ data[0::,9].astype(np.float) >= fare\_ceiling, 9 ] = fare\_ceiling - 1.0

fare\_bracket\_size = 10

number\_of\_price\_brackets = fare\_ceiling / fare\_bracket\_size

# I know there were 1st, 2nd and 3rd classes on board

number\_of\_classes = 3

# But it's better practice to calculate this from the data directly

# Take the length of an array of unique values in column index 2

number\_of\_classes = len(np.unique(data[0::,2]))

# Initialize the survival table with all zeros

survival\_table = np.zeros((2, number\_of\_classes, number\_of\_price\_brackets))

Now that these are set up, you can loop through each variable and find all those passengers that agree with the statements:

for i in xrange(number\_of\_classes): #loop through each class

for j in xrange(number\_of\_price\_brackets): #loop through each price bin

women\_only\_stats = data[ \#Which element

(data[0::,4] == "female") \#is a female

&(data[0::,2].astype(np.float) \#and was ith class

== i+1) \

&(data[0:,9].astype(np.float) \#was greater

>= j\*fare\_bracket\_size) \#than this bin

&(data[0:,9].astype(np.float) \#and less than

< (j+1)\*fare\_bracket\_size)\#the next bin

, 1] #in the 2nd col

men\_only\_stats = data[ \#Which element

(data[0::,4] != "female") \#is a male

&(data[0::,2].astype(np.float) \#and was ith class

== i+1) \

&(data[0:,9].astype(np.float) \#was greater

>= j\*fare\_bracket\_size) \#than this bin

&(data[0:,9].astype(np.float) \#and less than

< (j+1)\*fare\_bracket\_size)\#the next bin

, 1]

Notice that  data[ *where function*, 1]  means it is finding the Survived column for the conditional criteria which is being called. As the loop starts with i=0 and j=0, the first loop will return the Survived values for all the 1st-class females *(i + 1)* who paid less than 10 *((j+1)\*fare\_bracket\_size)* and similarly all the 1st-class males who paid less than 10.  Before resetting to the top of the loop, we can calculate the proportion of survivors for this particular combination of criteria and record it to our survival table:

  survival\_table[0,i,j] = np.mean(women\_only\_stats.astype(np.float))

survival\_table[1,i,j] = np.mean(men\_only\_stats.astype(np.float))

At the end we will get a matrix which will be shaped as a 2x3x4 array-- or think of this as two 3x4 arrays: The first corresponding to females, with the rows giving the class and columns giving the fare bracket, and the second corresponding similarly to the males.

Note! A Runtime warning will show when the loop is run, but it won't affect the output. This approach created a problem if there are no passengers in a given category. For example, in reality no females paid less than $10 for a first class ticket, so Python will return a nan for the mean, since it is dividing by zero. To deal with these, we could set them to 0 using a simple statement:

survival\_table[ survival\_table != survival\_table ] = 0.

What does our survival table look like?  Type print survival\_table

[[[ 0. 0. 0.83333333 0.97727273]

[ 0. 0.91428571 0.9 1. ]

[ 0.59375 0.58139535 0.33333333 0.125 ]]

[[ 0. 0. 0.4 0.38372093]

[ 0. 0.15873016 0.16 0.21428571]

[ 0.11153846 0.23684211 0.125 0.24 ]]]

Each of these numbers is the proportion of survivors for that criteria of passengers. For example, 0.91428571 signifies 91.4% of female, Pclass = 2, in the Fare bin of 10-19. The numbers should look familiar to you from the Pivot table in the previous Excel tutorial.  For our second model, let's again assume any probability greater than or equal to 0.5 should result in our predicting survival -- and less than 0.5 should not. We can update our survival table with:

survival\_table[ survival\_table < 0.5 ] = 0

survival\_table[ survival\_table >= 0.5 ] = 1

Now we have a survival table. Type  print survival\_table  again if you like.

When we go through each row of the test file we can find what criteria fit each new passenger and assign them a 1 or 0 according to our survival table.  As previously, let's open up the test file to read (and skip the header row), and also a new file to write to, called 'genderclassmodel.csv':

test\_file = open('../csv/test.csv', 'rb')

test\_file\_object = csv.reader(test\_file)

header = test\_file\_object.next()

predictions\_file = open("../csv/genderclassmodel.csv", "wb")

p = csv.writer(predictions\_file)

p.writerow(["PassengerId", "Survived"])

As with the previous model, we can take the first passenger, look at his/her gender, class, and price of ticket, and assign a Survived label. The problem is that each passenger in the test.csv file is not binned. We should loop through each bin and see if the price of their ticket falls in that bin. If so, we can break the loop (so we don’t go through all the bins) and assign that bin:

for row in test\_file\_object: # We are going to loop

# through each passenger

# in the test set

for j in xrange(number\_of\_price\_brackets): # For each passenger we

# loop thro each price bin

try: # Some passengers have no

# Fare data so try to make

row[8] = float(row[8]) # a float

except: # If fails: no data, so

bin\_fare = 3 - float(row[1]) # bin the fare according Pclass

break # Break from the loop

if row[8] > fare\_ceiling: # If there is data see if

# it is greater than fare

# ceiling we set earlier

bin\_fare = number\_of\_price\_brackets-1 # If so set to highest bin

break # And then break loop

if row[8] >= j \* fare\_bracket\_size\

and row[8] < \

(j+1) \* fare\_bracket\_size: # If passed these tests

# then loop through each bin

bin\_fare = j # then assign index

break

...

There are a couple of things to notice here. We *try* to make the relevant Fare variable (row[8]) into a float, since, in the case of empty data, the script cannot make it a float. If there is no fare entry we'll assume a fare bin simply correlated to the Passenger class. For example, if the passenger is third class they are put in the first bin ($0-9), second class into the second bin ($10-19), etc. The other thing to notice is that we assign the bin\_fare to equal j ... So although there are four bins, they must go from 0 to 3 because we will be using these as indices of our survival table. This little loop determines the index of the bin to look up in the survival table.

Now that we have determined the binned ticket price (bin\_fare), we can see if the passenger is female (row[3]), find their Pclass (row[1]), and then grab the relevant element in survival\_table. We need to convert this from the float in the survival\_table into an integer (int) that we write in our prediction file for Kaggle:

...

if row[3] == 'female': #If the passenger is female

p.writerow([row[0], "%d" % \

int(survival\_table[0, float(row[1])-1, bin\_fare])])

else: #else if male

p.writerow([row[0], "%d" % \

int(survival\_table[1, float(row[1])-1, bin\_fare])])

# Close out the files.

test\_file.close()

predictions\_file.close()

We have now inserted a 1 or 0 prediction, according to gender, class, and how much she/he paid in fare. We can now submit the file genderclassmodel.csv.

Conclusion

Just like in Excel, here we built predictions that take into account several features. But type  print survival\_table  again: what do you notice about the predictions for men? Surely some of the men survived, but our model can only predict 0. This suggests one source of error that's reflected in our leaderboard score, and it may already be prompting new ideas for improving your next model.

Yet in contrast to Excel, we have created a script now that can easily be altered to add more variables. For example, we could include Age, where they Embarked, or even their Name. All these variables may themselves have complications, so you will need to think of ways to make them useful. In this tutorial, in order to fill in any missing values of the fare, we assumed the Passenger Class can correlate simply to which fare bin to use. Using python we developed an extensible model without too much effort.

We are almost ready to apply Machine Learning on this data using python. However before we jump in, it would be advantageous to take a brief detour to learn tools that makes some of the work here easier.

[In the next tutorial](https://www.kaggle.com/c/titanic/details/getting-started-with-python-ii) we will explore python's Pandas package.

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

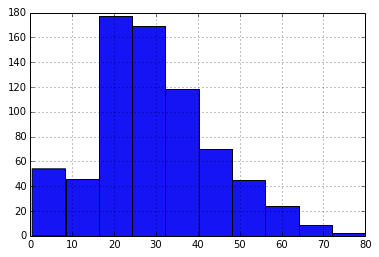
3 347

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()

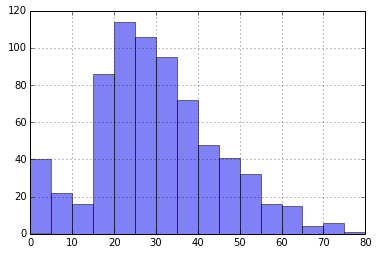
P.show()



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() )

Iambda 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](https://www.kaggle.com/c/titanic-gettingStarted/forums/t/6130/genderclassmodel-pandas-py). At the very least, you should be able to read and understand his script.)

[In the next tutorial](https://www.kaggle.com/c/titanic/details/getting-started-with-random-forests) we will take advantage of your new skills with python, in order to apply Machine Learning on this data for the first time.

Getting Started With Random Forests

Getting Started *with Random Forests:* Titanic Competition

The last step in this series of tutorials will be a quick guide to multivariate models that truly learn from your data. As far as competing and doing well in Kaggle competitions, this is the most important page, however there is much more to learn beyond this. We hope this gives you a taste of how simple it can be to apply a sophisticated algorithm.

Quick Recap:

The previous tutorials allowed us to open the data in both Excel and in the scripting language Python. We created a simple model in both and wrote a csv to use as our submission. We are going to build on these skills to create what is known as a random forest™ (or, an ensemble of decision trees).

The beauty of Python

This is where the effort you put into getting Python running pays off! As was mentioned in the previous tutorial, Python has handy, built-in packages to help you manipulate arrays, compute complex math functions, read in complex file formats, and in this case, create random forests. You will need to install the scikit-learn package, which has the handy RandomForestClassifier class. (If you originally installed the Anaconda distribution of python, then scikit-learn is already installed too.) For more on this package, visit the [scikit random forest page.](http://scikit-learn.org/dev/modules/generated/sklearn.ensemble.RandomForestClassifier.html)

What is a Random Forest?

As with all the important questions in life, this is best deferred to the [Wikipedia page](http://en.wikipedia.org/wiki/Random_forest). A random forest is an ensemble of [decision trees](http://en.wikipedia.org/wiki/Decision_tree) which will output a prediction value, in this case survival. Each decision tree is constructed by using a random subset of the training data. After you have trained your forest, you can then pass each test row through it, in order to output a prediction. Simple! Well not quite! This particular python function requires floats for the input variables, so all strings need to be converted, and any missing data needs to be filled.

How do I clean and fill?

In the previous pandas tutorial you gained the skills to clean and fill in data using the pandas package. Now you can put that into practice.

Not all types of data can be converted into floats. For example, Names would be very difficult. In these cases let's decide to neglect these columns. Although they are strings, the categorical variables like male and female can be converted to 1 and 0, and the port of embarkment, which has three categories, can be converted to a 0, 1 or 2 (Cherbourg, Southamption and Queenstown). This may seem like a non-sensical way of classifying, since Queenstown is not twice the value of Southampton-- but random forests are somewhat robust when the number of different attributes are not too numerous.

Converting from categorical strings to floats is intuitive. However, filling in data can be more tricky. Some data cannot be trivially filled (such as Cabin) without complete knowledge of every cabin and ticket price for the entire ship.  Nonetheless, Fare price can be estimated if you know the class, or the age of a passenger can be estimated using the median age of the people on board. Fortunately for us, the amount of missing data here is not too large, so the method for which you choose to fill the data shouldn’t have too much of an effect on your predictive result.

My data is complete and floating nicely, I want to predict

Using the predictive capabilities of the scikit-learn package is very simple. In fact, it can be carried out in three simple steps: initializing the model, fitting it to the training data, and predicting new values.

Note that almost all of the model techniques in scikit-learn share a few common named functions, once they are initialized. You can always find out more about them in the documentation for each model. These are

*some-model-name*.fit( )

*some-model-name*.predict( )

*some-model-name*.score( )

At this point, it is assumed that you have read in the training data into an array train\_data, where the first column [0] is the Survived column, that the Name, Cabin, and Ticket columns have been removed, and also that the Gender and Embarked columns have been converted to numbers. (If you just completed the previous pandas tutorial this means you also need to drop the PassengerId column.)

# Import the random forest package

from sklearn.ensemble import RandomForestClassifier

# Create the random forest object which will include all the parameters

# for the fit

forest = RandomForestClassifier(n\_estimators = 100)

# Fit the training data to the Survived labels and create the decision trees

forest = forest.fit(train\_data[0::,1::],train\_data[0::,0])

# Take the same decision trees and run it on the test data

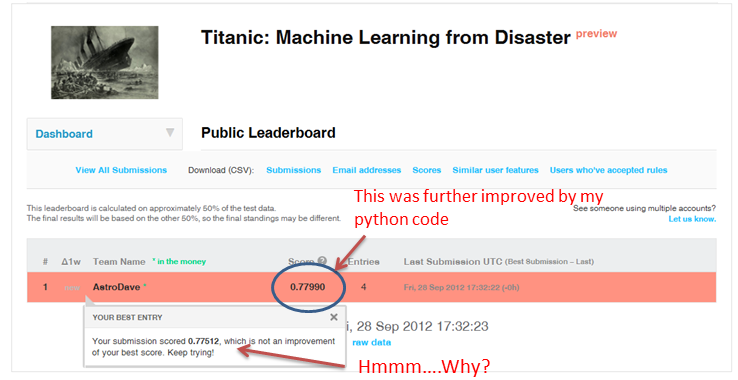
output = forest.predict(test\_data)

There is much more to this script, which you can find on the Data page.

The output will be an array with a length equal to the number of passengers in the test set and a prediction of whether they survived. You can change the parameters as you see fit, as described on the RandomForestClassifier [documentation page](http://scikit-learn.org/dev/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

Word of Warning!!!

Looking at the leaderboard, I now see:



It appears that I have not improved my score! This seems strange, as your initial thoughts are "This is more complicated, therefore should be better!" This gives us three lessons to bear in mind:

* A simple model is not always a bad model. Sometimes, concise, simple views of data reveal their true patterns and nature.
* This is not the final score for my new submission! I have not done as well on the public leaderboard, but who knows what the private score may hold? I made my previous model on the assumptions of the training data: we still don't know how these will hold up in the private leaderboard.
* Because the data set is very small, the differences in scores can be just one or two flips in decisions between survived or not survived. This means it will be very hard to determine the quality of the model from this data set. The aim of our Titanic Tutorial was to show you an easy way into more difficult problems, so don't be too disheartened if your super-complicated random forest doesn't beat the gender based model!

Conclusion

Now you have a budding set of skills to get you going in the larger Kaggle competitions. That, of course, is the primary prize of this Competition. There is still some improvement possible in your model, so please feel free to post comments on the forums, questions (and answers!). Chances are the experiences you have and questions you ask will be shared by others.

Know that a score of 0.79 - 0.81 is doing well on this challenge, and 0.81-0.82 is really going beyond the basic models! The dataset here is smaller than normal, so there is less signal to tap your models into. NOTE: You may see some people on the Leaderboard show accuracy of .90 up to even 100% -- but that's *probably not* from statistical modeling, just trying to look up the answers somewhere else on the Internet (which defeats the entire purpose.) So seriously, ignore those people! A discussion in the forum talks further about [What accuracy should I be aiming for?](https://www.kaggle.com/c/titanic-gettingStarted/forums/t/4894/what-accuracy-should-i-be-aiming-for)

In terms of improving your model from here, you could consider any of these paths to try on your own:

* 1. Revisit your assumptions about how you cleaned and filled the data.
  2. Be creative with additional feature engineering, so that your chosen model has more columns to train from.
  3. Use the sklearn documentation to experiment with different parameters for your random forest.
  4. Consider a different model approach. For example, a logistic regression model is often used to predict binary outcomes like 0/1.

*Added note:* If you would like to try out a different tool than Python for accomplishing the same kind of analysis, we now have links to outside tutorials for [Getting Started with R](https://www.kaggle.com/c/titanic/details/new-getting-started-with-r).

We also have [Further Reading](https://www.kaggle.com/c/titanic/details/further-reading-watching) with additional tutorials and suggestions.

Good luck with the competition!