P09 – Scripting for Data Science

1. Introduction

On April 15, 1912, the RMS Titanic hit an iceberg and sank. More than 67 percent of the people on board died. (Read the full story on Kaggle: https://www.kaggle.com/c/titanic).

Your task in this exercise is to write a Python script that analyses Titanic passenger data in order to answer the following question:

“What kind of individuals where likely to survive the disaster?”

This lab description contains plenty of tutorial material, based on the introductory competition called *“Machine Learning from Disaster”* on Kaggle (https://www.kaggle.com/c/titanic), to solve this question. You should start by following these notes. Your superior task is, however, to

- somewhere deviate from the pattern outlined in these notes and
- explore some *programming* ideas of your own

Remember: This course is about scripting, not data analysis per se. So you should focus on scripting techniques, not fancy analysis algorithms.

Examples for ideas to explore are:

- Things you want to engross (e.g., functions, loops, certain data structures)
- Things you want to newly explore (e.g., list comprehension, lambda expressions)
- Use different libraries (e.g., Pandas)

2. Data

The Titanic passenger data is freely available on the internet in various versions; we use a pre-processed version with all 1309 passenger entries in order to make it easier to work with using Python and Excel.

Basically, after a header row containing the column names, each row contains data about one passenger. The training set **`titanic3_train.csv`** (80% of records) includes the information if an individual has survived or not; in the test data set (20% of records) **`titanic3_test.csv`**, this column has been omitted.

If you are curious (i.e., this is not necessary), see section 1.3 “Original data description” in the file **`P09_DataScience_TitanicDataDescription.pdf`** for more background information and an explanation of the columns.

3. Task

3.1 Task description

Write a Python script that predicts from the data in the training set **`titanic3_train.csv`** for any individual if he or she has survived or not (i.e., deduce the value in the column **`survived`** from the values in the other columns of the same dataset). As the column is included, you can easily compare and assess the quality of your predictor!

As a result, your script should output a CSV file of exactly the following format: Each line, separated by a semicolon (not surrounded by whitespaces!), has to contain the id of a record (between 1 and 1309, but only for training dataset records) as well as the predicted value for the **`survived`** column (0 or 1). A header row containing the verbatim phrase "**`key;value`**" on top has also to be present.

```
key;value
5;0
10;1
15;0
...
```

3.2 Submission of results

Experimenting with the training set is necessary and somewhat interesting, but it is really challenging to apply your predictions to data where you don't know the correct solution a priori – and compare the performance (i.e., prediction accuracy) with others!

In order to do so, apply your script to the test set in **`titanic3_test.csv`** (Attention: remember that this data set has no column called **`survived`**, so you probably must adapt column numbers etc.) and produce the respective output file.

Then, submit your solution to our own lab web service:

- https://openwhisk.eu-de.bluemix.net/api/v1/web/SPLab_Scripting/default/titanic.html

The service will compute a score for your result and display it on our public leader board (but only after submission).

You can (and should) submit several versions as your script evolves; please remember to use different submission names (e.g., your nickname or team name, followed by a submission number, e.g. hour_minute) in order to get your progress reflected in the list instead of it getting overwritten each time.

Keep track of your scores with/without post-accident knowledge...

3.3 Further resources

There are numerous tutorials online for approaching the data set of Titanic passengers using Python. Among the most interesting ones for further inspiration are:

- https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-python
  (Covered here in sections 4.1–4.5)
- https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-python-ii
- https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-random-forests
  (Covered here in the remainder of section 4.6–4.8)
- http://nbviewer.ipython.org/github/agconti/kaggle-titanic/blob/master/Titanic.ipynb
- http://triangleinequality.wordpress.com/2013/09/05/a-complete-guide-to-getting-0-79903-in-kaggles-titanic-competition-with-python/

These Tutorials can give you some additional ideas on what can be done to the data to extract it’s treasures, and show different ways how this can be written in Python.

4. Tutorial[1]

4.1 Reading in the training data

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

```
import csv as csv
import numpy as np

# Open up the CSV file into a Python object
with open('titanic3_train.csv', 'r') as f:
    csv_file_object = csv.reader(f, delimiter=';')
    header = csv_file_object.__next__() #next() skips the first line holding the column headers
    data=[]
    for row in csv_file_object:  #Run through each row in the CSV, add it to the data variable
        data.append(row)

# Then convert from a list to an array
# (Be aware that each item is currently a string in this format)
data = np.array(data)
print(data)
```

Compare how the data looks with what you see with e.g. Excel or a text editor. 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])`

1The following notes are based on Kaggle’s notes (see links above). You should start to follow these notes to get an idea. Then, deviate from them at any time to try individual ideas (compare section 1).

To see the 1st row, 4th column, type `print(data[0,3])`

4.2 Play with the training data

Now if you want to call a specific column of data, say, the gender column, you can just type `data[0::,5]`, 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::,1].astype(np.float)`. Using this, you can calculate the proportion of survivors on the Titanic:

```
# The size() function counts how many elements are in
# the array and sum() (as you would expect) sums up
# the elements in the array.
number_passengers = np.size(data[0::,2].astype(np.float))
number_survived = np.sum(data[0::,2].astype(np.float))
proportion_survivors = number_survived / number_passengers
print(proportion_survivors)
```

NumPy has some lovely functions. For example, you 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:

```
# This finds where all the elements in the gender column equal “female”
women_only_stats = data[0::,5] == "female"
# This finds where all the elements do not equal female (i.e. male)
men_only_stats = data[0::,5] != "female"
```

You can use these two new variables as a "mask" on the original train data, so you 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,2].astype(np.float)
men_onboard = data[men_only_stats,2].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 you have your indication that women were much more likely to survive, you are done for the moment with the training set.

4.3 The first submission using the test data

According to your experience with the training data, your proposal is to calculate the survival probability purely based on a person's gender. Naïve, without machine 'learning' (no training set involved), but simple! First, you read in the **`titanic3_test.csv`** file and skip the header line:

```
# Open up the CSV file in to a Python object
test_file = open('titanic3_test.csv', 'r')
test_file_object = csv.reader(test_file, delimiter=';')
header = test_file_object.next()
```

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

```
prediction_file = open("submission1_genderbased.csv", "w")
prediction_file_object = csv.writer(prediction_file, delimiter=';')
```

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

```
prediction_file_object.writerow(["key", "value"])
for row in test_file_object:
    if row[4] == 'female':
        prediction_file_object.writerow([row[0],'1']) # predict 1
    else: #it is male
        prediction_file_object.writerow([row[0],'0']) # predict 0
test_file.close()
prediction_file.close()
```

Now you have a file called **`submission1_genderbased.csv`**, which you should submit (see section 3.2 again how to do this)!

4.4 Submission two (optional)

Let's complicate things and try and submit a new solution, binning up the ticket price into the four bins and modelling the outcome on **`class`**, **`gender`**, and ticket **`price`**. This part assumes that you have completed section 4.1 and you have a data array as before. Now, training data will be involved to actually 'learn' from existing data.

First, you have to care for missing data: Some of the **`fare`** values have no value at all, which will later give an error if you try to convert them to **`float`**. Add the following lines to assign them the mean value of all the fares in this **`pclass`**:

```
#fill up empty fare values with the mean of the corresponding pclass
```

```
#1. calculate mean fare of each pclass
fare_mean_per_pclass = []
for i in range(len(np.unique(data[0::,1]))): #loop over all possible pclass's (i == pclass-1)
    fare_mean_per_pclass.append(0) #initialize ith pclass fare mean
    cnt = 0 #number of tickets with a value in pclass i
    for j in range(len(data[0::,10])):
        if data[j, 1].astype(np.int) == i+1:
            try: #try to cast the jth fare value as float
                fare_mean_per_pclass[i] += data[j, 10].astype(np.float)
                cnt += 1
            except: #if it can't be casted to float: ignore it
                fare_mean_per_pclass[i] += 0 #just ignore empty values
    if cnt > 0: fare_mean_per_pclass[i] /= float(cnt) #calculate mean
print("Mean fare per pclass: ", fare_mean_per_pclass)

for i in range(len(data[0::,10])): #2. replace the empty values with the pclass mean
    try:
        test = float(data[i, 10]) #if this works, all is well with row i
    except ValueError:
        data[i, 10] = fare_mean_per_pclass[data[i, 1].astype(np.int)-1] #index: cur. pclass-1
```

The idea is now to create a table which contains just **`1`**'s and **`0`**'s. This array will be a survival reference table created from the training data. Then, 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 (based on the similar passengers in table). In the case of a model that uses **`gender`**, **`class`**, and ticket **`price`**, you will need an array of size **`2x3x4`** ([**`female`**/**`male`**] , [1st / 2nd / 3rd **`class`**], [4 bins of **`prices`**]).

The script will systematically loop through each combination and 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::,10].astype(np.float) >= fare_ceiling, 10 ] = 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 1
number_of_classes = len(np.unique(data[0::,1]))
```

```
# 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
        #Which element is a female, and was ith class, was greater than this bin,
        #and less than the next bin -> give the the 3rd col (survived)
        women_only_stats = data[ \
                            (data[0::,5] == "female") \
                            &(data[0::,1].astype(np.float) == i+1) \
                            &(data[0:,10].astype(np.float) >= j*fare_bracket_size) \
                            &(data[0:,10].astype(np.float) < (j+1)*fare_bracket_size) \
                            , 2]

        #Which element is a male, and was ith class, was greater than this bin,
        #and less than the next bin
        men_only_stats = data[ \
                          (data[0::,5] != "female") \
                          &(data[0::,1].astype(np.float) == i+1) \
                          &(data[0:,10].astype(np.float) >= j*fare_bracket_size) \
                          &(data[0:,10].astype(np.float) < (j+1)*fare_bracket_size) \
                          , 2]
```

Notice that **`data[ where_function, 2]`** 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`** list-- or think of this as two **`3x4`** lists: 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: Two Numpy runtime warnings will shown when the loop is run, but they won't affect the output.

This approach creates 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** (“not a number”) 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 the survival table look like? Type **print survival_table**

```
[[[ 0.          0.          1.          0.97222222]
  [ 0.          0.88571429  0.82352941  1.        ]
  [ 0.52873563  0.58        0.4         0.11764706]]

 [[ 0.14285714  0.          0.38235294  0.31      ]
  [ 0.          0.14634146  0.13333333  0.17647059]
  [ 0.14814815  0.19512195  0.18518519  0.16666667]]]
```

Each of these numbers is the proportion of survivors for that criteria of passengers. For example, **0.88571429** signifies 91.4% of **female**, **pclass==2**, in the **fare** bin of 10-19. Let's assume any probability greater than or equal to **0.5** should result in our predicting **survival==1**, and less than **0.5** should not. You can update the 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 you go through each row of the test file you 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 **submission2_genderclassbased.csv**:

```
#open the test set and a new submission file
test_file = open('titanic3_test.csv', 'rb')
test_file_object = csv.reader(test_file , delimiter=';')
header = test_file_object.next()
predictions_file = open("submission2_genderclassbased.csv", "w")
p = csv.writer(predictions_file, delimiter=';')
p.writerow(["key", "value"])
```

As with the previous model, you 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 **`titanic3_test.csv`** file is not binned. You should loop through each bin and see if the price of their ticket falls in that bin. If so, you can break the loop (so you don't go through all the bins) and assign that bin:

```
# We are going to loop through each passenger in the test set
for row in test_file_object:
    # For each passenger we loop throu each price bin
    for j in xrange(number_of_price_brackets):
        try: # Some passengers have no fare data so try to make…
            row[9] = float(row[9]) # a float
        except: # If fails: no data, so…
            bin_fare = 3 - float(row[1]) # bin the fare according to pclass
            break
        if row[9] > fare_ceiling: # If there is data see if it is greater than fare ceiling
            bin_fare = number_of_price_brackets-1 # If so set to highest bin
            break
        # If passed these tests then loop through each bin
        if row[9] >= j * fare_bracket_size \
           and row[9] < (j+1) * fare_bracket_size:
              bin_fare = j # If passed these tests then assign index
              break
        #…
```

There are a couple of things to notice here. You *try* to make the relevant **`fare`** variable (**`row[9]`**) into a **`float`**, since, in the case of empty data, the script cannot make it a **`float`**. If there is no **`fare`** entry you'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 you assign the **`bin_fare`** to equal **`j`** ... so although there are four bins, they must go from 0 to 3 because you 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 you have determined the binned ticket price (**`bin_fare`**), you can see if the passenger is **`female`** (**`row[5]`**), find their **`pclass`** (`row[1]`), and then grab the relevant element in **`survival_table`**. You need to convert this from the **`float`** in the **`survival_table`** into an integer (**`int`**) that you write in our prediction file for the submission:

```
        #…
        if row[4] == 'female': #If the passenger is female
            p.writerow([row[0], "%d" % int(survival_table[0, int(row[1])-1, bin_fare])])
        else: #passenger is male
            p.writerow([row[0], "%d" % int(survival_table[1, int(row[1])-1, bin_fare])])

# Close out the files.
```

```
test_file.close()
predictions_file.close()
```

You have now inserted a **`1`** or **`0`** prediction, according to gender, class, and how much she/he paid in **`fare`**. You can now submit the file **`submission2_genderclassbased.csv`**.

4.5 Conclusion and further explorations
You have 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 your model can only predict **`0`**. This suggests one source of error that's reflected in your leaderboard score, and it may already be prompting new ideas for improving your next model.

You have created a script now that can easily be altered to add more variables. For example, you 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, you assumed the passenger class can correlate simply to which fare bin to use. Using Python you developed an extensible model without too much effort.

You are almost ready to apply Machine Learning on this data using Python. However before you jump in, it would be advantageous to take a brief detour to learn tools that makes some of the work here easier. See Kaggle’s next tutorial at https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-python-ii to e.g. explore Python's Pandas package. But beware: The data Kaggle uses is different!

- You possibly have different columns names and certainly have different columns indexes (e.g., we added a new 0th column called **`id`**)
- You may have different values due to a different division into training- and test data

Revise the given scripts in order to reflect your data, but adopt any ideas you find interesting.

4.6 Getting Started With Random Forests
This last step will be a quick guide to multivariate models that truly learn from your data. We hope this gives you a taste of how simple it can be to apply a sophisticated algorithm (here, a so-called random forest™ or, an ensemble of decision trees).

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 the **`scikit-learn`** package, which has the handy **`RandomForestClassifier`** class. (**`scikit-learn`** already ships with Anaconda). For more on this package, visit the scikit random forest page.

What is a random forest? As with all the important questions in life, this is best deferred to the Wikipedia page. A random forest is an ensemble of decision trees 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 again to be converted, and any missing data needs again to be filled.

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 (see section 4.4). 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.

4.7 Submission three

In order to use random forests we need to import the following.

```
import csv as csv
import numpy as np
from sklearn.ensemble import RandomForestClassifier
```

As mentioned before, we need to convert all **`strings`** to **`floats`** and any missing data needs to be filled. As long as we use arrays to store our data, it's important to know exactly the content of each array column of our data[2].

2Alternatively the **`pandas`** package defines another data structure which allows which allows to access the data by name. The **`pandas`** package is introduced in a separate tutorial (https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-python-ii) and will not be used in the following.

We start with the preparation of the training data used to create the prediction model. First, we read the training file the same way we did in the previous tutorials and turn the data into an array.

```
# Open up the CSV file into a Python object
with open('../data/titanic3_train.csv', 'r') as f:     # Load the training file
    csv_file_object = csv.reader(f, delimiter=';')
    csv_file_object.next()      # next() skips the first line holding the column headers
    orig_train_data = []
    for row in csv_file_object: # Run through each row in the csv, add it to the data variable
        orig_train_data.append(row)

# Then convert from a list to an array
# (Be aware that each item is currently a string in this format)
orig_train_data = np.array(orig_train_data)
```

The columns and their corresponding indices contained in the original training data are given by following table:

```
# COLUMN CONTENT     ORIG.INDEX
# id                0
# pclass             1
# survived           2
# name               3
# surname            4
# sex               5
# age               6
# sibsp             7
# parch              8
# ticket            9
# fare             10
# cabin             11
# embarked          12
# boat             13
# body              14
# home.dest         15
```

To create the prediction model only the data is used that can easily be converted into **`floats`**. So the training data finally contains the following data at the index position TRAIN.INDEX:

```
# Prepare the data structure used for training
# DATA USED FOR TRAINING    ORIG.INDEX      TRAIN.INDEX
# survived                   2             0
# pclass                    1             1
# sibsp                     7             2
# parch                    8            3
# sex                      5            4
# age                      6            5
# fare                    10            6
# embarked                    12            7
```

This leads to a matrix with a number of rows equal to the number of rows in the original training data and 8 columns.

```
rows = len(orig_train_data[0::, 0])     # Number of rows in the training data
cols = 8                                # Number of columns in the training data
train_data = np.zeros((rows, cols))  # Array to store the data used for training
```

The first column shall contain the information if the passenger survived and will be used as label for the data similar as we did in the previous tutorials. This column contains only 0’s or 1’s and can simply be converted to **`float`**.

```
# SURVIVED: Store data 'survived' in train_data
train_data[0::, 0] = orig_train_data[0::, 2].astype(np.float)
```

The next three columns can also simply be converted since they contain only numbers and no empty entries.

```
# PCLASS: Store data 'pclass' in train_data
train_data[0::, 1] = orig_train_data[0::, 1].astype(np.float)

# SIBSP: Store data 'sibsp' in train_data
train_data[0::, 2] = orig_train_data[0::, 7].astype(np.float)

# PARCH: Store data 'parch' in train_data
train_data[0::, 3] = orig_train_data[0::, 8].astype(np.float)
```

The information about gender is originally stored as “female” and “male” and must be converted to 0 for “female” and 1 for “male”. If an entry is empty, it will be filled with the most frequent gender. Therefor, we first have to get the most frequent gender and afterwards store the converted values in the training data.

```
# SEX: Prepare data 'sex' and store it in train_data
# First: get the most frequent gender
gender_data = orig_train_data[0::, 5]
num_female = sum(gender_data == 'female')
num_male = sum(gender_data == 'male')
# Set the most frequent gender (female = 0, male = 1)
most_freq_gender = 0 if num_female >= num_male else 1
# Second: store gender data in train_data
for i, sex in enumerate(gender_data):
    if sex == '':
        train_data[i, 4] = most_freq_gender  # Most freq. gender is used if 'sex' is undefined
    if sex == 'female':
        train_data[i, 4] = 0
    if sex == 'male':
        train_data[i, 4] = 1
```

The column containing the age of the passengers is incomplete. Therefore, we have to calculate the median of the existing entries and replace empty ones with the median.

```
# AGE: Prepare data 'age' and store it in train_data
# First: get the median age
# Convert 'age' to float, empty values to 0
age_data = [0 if age == '' else float(age) for age in orig_train_data[0::, 6]]
median_age = np.nanmedian(age_data)
# Second: store age data in train_data
# Alternative: train_data[0::, 5] = [median_age if age == 0 else age for age in age_data]
for i, age in enumerate(age_data):
    if age == 0:
        train_data[i, 5] = median_age    # Most freq. age is used if 'age' is undefined
    else:
        train_data[i, 5] = age
```

Similar to the information about age, we have to replace missing fares with median values. Since the fares depend on their respective “pclass”, three different median fares have to be calculated. Afterwards, the missing entries can be replaced by the corresponding median.

```
# FARE: Prepare data 'fare' and store it in train_data
# First: get the 'fare' and 'pclass' data
# Convert 'fare' to float, empty values to 0
fare_data = [0 if fare == '' else float(fare) for fare in orig_train_data[0::, 10]]
fare_data = np.array(fare_data)     # Convert from a list to an array
pclass_data = train_data[0::, 1]    # Get the 'pclass' values from train_data
pclass_data_unique = list(enumerate(np.unique(pclass_data)))  # Get the unique 'pclass' values
# Second: replace fares with value 0 with the median fare of the corresponding 'pclass'
for i, unique_pclass in pclass_data_unique:
    # Get array of fares corresponding to the current pclass
    pclass_fare = fare_data[pclass_data == unique_pclass]
    # Calculate the median of the previously received fares
    median_fare = np.nanmedian(pclass_fare)
    pclass_fare[pclass_fare == 0] = median_fare  # Replace fares with value 0 with median fare
    fare_data[pclass_data == unique_pclass] = pclass_fare
# Third: store fare data in train_data
train_data[0::, 6] = fare_data[0::]
```

The last column used for training contains the information about the port where the passengers embarked. Three different ports are possible and have to be converted to 0, 1 or 2. But first, we again have to replace missing values with the most common port.

Therefor the most common port has to be determined. Once the missing values are replaced, the data can be converted and stored in the training data.

```
# EMBARKED: Prepare data 'embarked' and store it in train_data
# First: get the most common 'embarked' value
embarked_data = list(orig_train_data[0::, 12])
mc_embarked = max(set(embarked_data), key=embarked_data.count)
# Second: replace empty entries with the most common 'embarked' value
embarked_data = [mc_embarked if embarked == '' else embarked for embarked in embarked_data]
embarked_data = np.array(embarked_data)
# Third: convert all 'embarked' values to int
# Get the unique 'embarked' values
embarked_data_unique = list(enumerate(np.unique(embarked_data)))
for i, unique_embarked in embarked_data_unique:
    embarked_data[embarked_data == unique_embarked] = i
# Fourth: store embarked data in train_data
train_data[0::, 7] = embarked_data[0::].astype(np.float)
```

Now we have created our training data to train a random forest. Similar to the steps above, we now need to convert and fill the test data.

The Code below does the trick and will not be explained in depth because it’s almost similar to the one above.

```
# Open up the CSV file into a Python object
with open('../data/titanic3_test.csv', 'r') as f:     # Load the test file
    csv_file_object = csv.reader(f, delimiter=';')
    csv_file_object.next()   # next() skips the first line holding the column headers
    orig_test_data = []
    for row in csv_file_object: # Run through each row in the csv, add it to the data variable
        orig_test_data.append(row)

# Then convert from a list to an array
# (Be aware that each item is currently a string in this format)
orig_test_data = np.array(orig_test_data)
```

Since we want to predict if a passenger has survived or not, the column with the information about survival is missing in the test data. This differs from the training data and leads to slightly different indices.

```
# Overview of the test data
# DATA      ORIG.INDEX
# id        0
# pclass    1
# name      2
# surname   3
# sex       4
# age       5
# sibsp     6
# parch     7
# ticket    8
# fare      9
# cabin     10
# embarked  11
# boat      12
# body      13
# home.dest 14
```

```
# Prepare the data structure used for testing
# DATA USED FOR TESTING    ORIG.INDEX      TEST.INDEX
# pclass                    1              0
# sibsp                     6              1
# parch                     7              2
# sex                       4              3
# age                       5              4
# fare                      9              5
# embarked                  11             6
```

Since there is one column less in comparison with the training data, the size of the test data needs to be adjusted.

```
rows = len(orig_test_data[0::, 0])      # Number of rows in the test data
cols = 7                                # Number of columns in the test data
test_data = np.zeros((rows, cols))  # Array to store the data used for testing
```

The steps containing converting and filling missing data are equal to the ones concerning the training data and are not explained any further. Keep in mind that the indices differ!

```
# PCLASS: Store data 'pclass' in test_data
test_data[0::, 0] = orig_test_data[0::, 1].astype(np.float)
```

```
# SIBSP: Store data 'sibsp' in test_data
test_data[0::, 1] = orig_test_data[0::, 6].astype(np.float)
```

```
# PARCH: Store data 'parch' in test_data
test_data[0::, 2] = orig_test_data[0::, 7].astype(np.float)
```

```
# SEX: Prepare data 'sex' and store it in test_data
# First: get the most frequent gender
gender_data = orig_test_data[0::, 4]
num_female = sum(gender_data == 'female')
num_male = sum(gender_data == 'male')
```

```
# Set the most frequent gender (female = 0, male = 1)
most_freq_gender = 0 if num_female >= num_male else 1
# Second: store gender data in test_data
for i, sex in enumerate(gender_data):
    if sex == '':
        test_data[i, 3] = most_freq_gender  # Most freq. gender is used if 'sex' is undefined
    if sex == 'female':
        test_data[i, 3] = 0
    if sex == 'male':
        test_data[i, 3] = 1
```

```
# AGE: Prepare data 'age' and store it in test_data
# First: get the median age
# Convert 'age' to float, empty values to 0
age_data = [0 if age == '' else float(age) for age in orig_test_data[0::, 5]]
median_age = np.nanmedian(age_data)
# Second: store age data in test_data
# Alternative: test_data[0::, 5] = [median_age if age == 0 else age for age in age_data]
for i, age in enumerate(age_data):
    if age == 0:
        test_data[i, 4] = median_age    # Most freq. age is used if 'age' is undefined
    else:
        test_data[i, 4] = age
```

```
# FARE: Prepare data 'fare' and store it in test_data
# First: get the 'fare' and 'pclass' data
# Convert 'fare' to float, empty values to 0
fare_data = [0 if fare == '' else float(fare) for fare in orig_test_data[0::, 9]]
fare_data = np.array(fare_data)    # Convert from a list to an array
pclass_data = test_data[0::, 0]    # Get the 'pclass' values from test_data
pclass_data_unique = list(enumerate(np.unique(pclass_data)))  # Get the unique 'pclass' values
# Second: replace fares with value 0 with the median fare of the corresponding 'pclass'
for i, unique_pclass in pclass_data_unique:
    # Get array of fares corresponding to the current pclass
    pclass_fare = fare_data[pclass_data == unique_pclass]
    median_fare = np.nanmedian(pclass_fare)
    pclass_fare[pclass_fare == 0] = median_fare  # Replace fares with value 0 with median fare
    fare_data[pclass_data == unique_pclass] = pclass_fare
# Third: store fare data in test_data
test_data[0::, 5] = fare_data[0::]
```

```
# EMBARKED: Prepare data 'embarked' and store it in test_data
# First: get the most common 'embarked' value
embarked_data = list(orig_test_data[0::, 11])
mc_embarked = max(set(embarked_data), key=embarked_data.count)
# Second: replace empty entries with the most common 'embarked' value
embarked_data = [mc_embarked if embarked == '' else embarked for embarked in embarked_data]
embarked_data = np.array(embarked_data)
# Third: convert all 'embarked' values to int
# Get the unique 'embarked' values
embarked_data_unique = list(enumerate(np.unique(embarked_data)))
for i, unique_embarked in embarked_data_unique:
    embarked_data[embarked_data == unique_embarked] = i
# Fourth: store embarked data in test_data
test_data[0::, 6] = embarked_data[0::].astype(np.float)
```

At this point, the training and test data is ready. But before we can start with training and predicting, we need the ids of the test data to be stored together with the predictions. These ids can simply be received from the original test data.

```
test_ids = orig_test_data[0::, 0]
```

Now the data is complete and we want to predict. Using the predictive capabilities of the **`scikit-learn`** package is very simple. It can be carried out in three simple steps:

- initializing the model
- fitting it to the training data
- 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(..)`

To initialize the model, we use the following line of code:

```
forest = RandomForestClassifier(n_estimators=100)
```

Now we want to fit our forest to the training data. Remember that the first column of our training data is the “survived” column that is used to label the remaining training data. The **`.fit(..)`**-function takes two arguments. The first is the training data used to predict and the second are the labels.

```
forest = forest.fit(train_data[0::, 1::], train_data[0::, 0])
```

Once the forest fits to the data, we can finally do our predictions for the test data. 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.

```
output = forest.predict(test_data).astype(np.float)
```

The last thing to do is to write the predictions together with the passenger ids into a file to be submitted.

```
# Write the data into a file
predictions_file = open('submission3_randomforest.csv', 'w')
open_file_object = csv.writer(predictions_file, delimiter=';')
open_file_object.writerow(['key', 'value'])
open_file_object.writerows(zip(test_ids, output))
predictions_file.close()
```

4.8 Conclusions

It can happen that the score will not improve depending on the columns used to train and predict. This might seem strange, as your initial thoughts may have been *"This is more complicated, therefore should be better".* This gives us two lessons to bear in mind:

1. A simple model is not always a bad model. Sometimes, concise, simple views of data reveal their true patterns and nature.
2. 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 the Titanic tutorial is 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.

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

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

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 leaderboards show accuracy of .90 up to even 100% for this data set – but that's *probably not* from statistical modeling, but from looking up the answers somewhere else on the Internet, by combining the altsample dataset, or by doing web automation. No need to fool yourself - the focus is on scripting.