R to Python for Data Analysis

Leonardo Uchoa

3/31/2020

Contents

1 The tables

That's my R to python port. It's intended to make my approach to learning python faster and its mostly composed of data wrangling routine tools. Many of those are already listed in other sources.

Table 1: Data Wrangling

R	Python
dim	df.shape (pd)
str	df.dtypes / df.info (pd)
unique	np.unique (np)
sort	np.sort (np)
rbind	np.hstack (np)
summary	df.describe (pd)
`group_by`	df.groupby (pd)
count	`df.value_count` (pd)
—	np.bincount (np)
apply	df.apply (pd)
if.else	df.where[case,true,false] (pd)
table	pd.crosstab
mutate(df, c=a-b)	df.assign(c=df['a']-df['b']) (pd)
colSums(is.na())	df.isnull().sum() (pd)
na.omit	df.dropna(axis=X) (pd)
imputation	df.fillna(df.mean()) (pd)
colnames() <-	df.colnames (pd)

Table 2: Linear Algebra

R	Python
eigen	np.linalg.eig (np)
%*%	np.dot
—	np.matmul

Table 3: Abreviations

Module Abreviation	Module
pd	Pandas
np	Numpy

Usefull examples

These are convertions of the commands I use the most and some other (because they're different from what I'm used to do in R) when analysing data in R.

I wrote this document using Rstudio and Rmarkdown. So in order to load python within R the thing to do was to use `reticulate`. But I've also installed all my python packages using anaconda because it handles compatability between libraries more efficiently. In that case you can load `reticulate` set it's path with `use_python` for it to load packages installed via anaconda (they're **not** the same).

```
library(reticulate)
use_python("/home/leonardo/anaconda3/bin/python")
```

```
import numpy as np
import pandas as pd
```

2 Counting per column

Source: StackOverflow.

```
df = pd.DataFrame(np.random.randint(0, 2, (10, 4)), columns=list('abcd'))
df.apply(pd.Series.value_counts)
```

```
##    a  b  c  d
## 0  6  3  2  4
## 1  4  7  8  6
```

3 Multi argument iteration with zip

Zip allows us to construct tuples for iterating over multiple arguments.

```
players = [ "Sachin", "Sehwag", "Gambhir", "Dravid", "Raina" ]
scores = [100, 15, 17, 28, 43 ]

# Lets see how it constructs the tuples

print(tuple(zip(players, scores)))

# Now we just need to iterate over them
```

```
## (('Sachin', 100), ('Sehwag', 15), ('Gambhir', 17), ('Dravid', 28), ('Raina', 43))
```

```
for pl, sc in zip(players, scores):
    print ("Player :  %s     Score : %d" %(pl, sc))
```

```
## Player :  Sachin     Score : 100
## Player :  Sehwag     Score : 15
## Player :  Gambhir     Score : 17
## Player :  Dravid     Score : 28
## Player :  Raina     Score : 43
```

4 Categorical data encoding

Source: Chapter 4 of Python Machine Learning [2].

For this section we're working the toy data bellow

```
df = pd.DataFrame([
['green', 'M', 10.1, 'class2'],
['red', 'L', 13.5, 'class1'],
['blue', 'XL', 15.3, 'class2']])
df.columns = ['color', 'size', 'price', 'classlabel']

df
```

```
##    color size  price classlabel
## 0  green    M   10.1     class2
## 1    red    L   13.5     class1
## 2   blue   XL   15.3     class2
```

In both approaches bellow we use a dictionary the create the mapping identifier for the `map` method. Remember that according to w3schools a dictionary is

> *A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.*

4.1 Encoding ordinals - create labels manually

```
#create the dict mapping from ordinal to integer
size_mapping = {'XL': 3,'L': 2,'M': 1}

#use map to in the desired column get the mapped values
df['size'] = df['size'].map(size_mapping)
```

4.2 Encoding nominals - creating labels automatically

```
class_mapping = {label: idx for idx, label in enumerate(np.unique(df['classlabel']))}
```

Now what that command is doing is looping through the iterators `idx` and `label` (created by the `enumerate` function) in the unique values of the `classlabel` column and assigning both to `label` and `idx`. Let's see

```
print(list(
    enumerate(np.unique(df['classlabel']))
  ))
```

```
## [(0, 'class1'), (1, 'class2')]
```

So iterating through the list we get to assign "class1"/"class2" to `label` and 0/1 to `idx`[1]. Finally the last step to map

```
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

```
##    color  size  price  classlabel
## 0  green     1   10.1           1
## 1    red     2   13.5           0
## 2   blue     3   15.3           1
```

Want to get the mapping backwards? Access the `items` method in the `class_mapping` object and loop again

[1] Note the inversion in 'label: idx for idx, label'

```
inv_class_mapping = {a:b for b,a in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

```
##    color  size  price classlabel
## 0  green     1   10.1     class2
## 1    red     2   13.5     class1
## 2   blue     3   15.3     class2
```

Ps.: There's also an object in `skitlearn` module `preprocessing` that does this: `LabelEncoder`

4.3 Creating Dummies in Design Matrix

First way: Pandas

Now to create dummy variables for the design matrix, there's a simple way using pandas

```
df_dm = pd.get_dummies(df[['price', 'color', 'size']], drop_first = True)
df_dm
```

```
##    price  size  color_green  color_red
## 0   10.1     1            1          0
## 1   13.5     2            0          1
## 2   15.3     3            0          0
```

The `drop_first = True` is important. Otherwise we woud get another column name "color_blue" with an $(0, 0, 1)^T$ entry which we do not need because when the other 2 columns are 0 we already encode the blue color.

Second way: sklearns' OneHotEncoder

The OneHotEncoder function arguments are (name, transformer, columns) tuples

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

#creat the object
color_ohe = OneHotEncoder(categories='auto', drop='first')
c_transf = ColumnTransformer([
('onehot', color_ohe, [0]), # respectively 'transformer_name',transformer_object,column
('nothing', 'passthrough', [1, 2]) # respectively 'action_1','action_2', columns for action_1/2
])

#note that the function input is an np array
c_transf.fit_transform(df[['color', 'size', 'price']].values).astype(float)
```

```
## array([[ 1. ,  0. ,  1. , 10.1],
##        [ 0. ,  1. ,  2. , 13.5],
##        [ 0. ,  0. ,  3. , 15.3]])
```

5 Train-test Split and basic pre-process

Source: Chapter 4 of Python Machine Learning [2].

5.1 Train-test Split

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

wine = datasets.load_wine()
wine.data.shape
```

```
## (178, 13)
```

```
wine.target.shape
```

```
## (178,)
```

```
X = wine.data
y = wine.target

X_train, X_test, y_train, y_test = train_test_split(X,y,
test_size=0.3, # split
random_state=0, #  set.seed
stratify=y) #stratification of data based on target frequencies
```

5.2 Basic Continuous pre-process

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)

mms = MinMaxScaler()
X_train_mms = mms.fit_transform(X_train)
X_test_mms = mms.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)

lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
```

6 Basic Pipeline

6.1 Cross Validation

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

pipe_lr = make_pipeline(StandardScaler(),
  PCA(n_components=2),
  LogisticRegression(random_state=1,
  solver='lbfgs'))

# First way

from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
scores = []

for k, (train, test) in enumerate(kfold):
  pipe_lr.fit(X_train[train], y_train[train])
  score = pipe_lr.score(X_train[test], y_train[test])
  scores.append(score)
  print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
    np.bincount(y_train[train]), score))

# Second and Better way

from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
  X=X_train,y=y_train,cv=10,n_jobs=1)

print('CV accuracy scores: %s' % scores)
```

6.2 Sample Size Learning Curves

One thing to note here is, according to [2]

> Note that we passed max_iter=10000 as an additional argument when instantiating the LogisticRegression object (which uses 1,000 iterations as a default) to avoid convergence issues for the smaller dataset sizes or extreme regularization parameter values.

Which can be valuable for other algorithms too. Note also that stratified k-fold CV is the default routine.

```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve


pipe_lr = make_pipeline(StandardScaler(),
LogisticRegression(penalty='l2',
random_state=1,
solver='lbfgs',
```

```
max_iter=10000))

train_sizes, train_scores, test_scores =\ learning_curve(estimator=pipe_lr,
X=X_train,
y=y_train,
train_sizes=np.linspace(
0.1, 1.0, 10),
cv=10,
n_jobs=1)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)


plt.plot(train_sizes, train_mean, color='blue', marker='o',
markersize=5, label='Training accuracy')

plt.fill_between(train_sizes,train_mean + train_std, train_mean - train_std,alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean,color='green', linestyle='--', marker='s', markersize=5,label='Validati

plt.fill_between(train_sizes,
test_mean + test_std,
test_mean - test_std,
alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training examples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.show()
```

6.3 Validation Curves

Here stratified k-fold CV is the default routine.

```
from sklearn.model_selection import validation_curve

param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

train_scores, test_scores = validation_curve(
estimator=pipe_lr,
X=X_train,
y=y_train,
param_name='logisticregression__C',
param_range=param_range,
cv=10)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
```

```
test_std = np.std(test_scores, axis=1)

plt.plot(param_range, train_mean,
color='blue', marker='o',
markersize=5, label='Training accuracy')
plt.fill_between(param_range, train_mean + train_std,
train_mean - train_std, alpha=0.15,
color='blue')
plt.plot(param_range, test_mean,
color='green', linestyle='--',
marker='s', markersize=5,
label='Validation accuracy')

plt.fill_between(param_range,
test_mean + test_std,
test_mean - test_std,
alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.show()
```

6.4 Tuning hyperparameters via grid search

Here we're grid searching hyperparameters for an SVM with two different kernel functions and their respective parameters[2]. Thats is: for the linear basis function we search for the range and for the rbf we search the range and gamma.

Here we have two dictionaries:

- `{'svc__C': param_range,'svc__kernel': ['linear']}` and
- `{'svc__C': param_range,'svc__gamma': param_range,'svc__kernel': ['rbf']}`

who compose our grids.

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),SVC(random_state=1))

param_range = [0.0001, 0.001, 0.01, 0.1,1.0, 10.0, 100.0, 1000.0]

param_grid = [{'svc__C': param_range,'svc__kernel': ['linear']},
{'svc__C': param_range,'svc__gamma': param_range,
'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,param_grid=param_grid
scoring='accuracy',cv=10,refit=True,n_jobs=-1)
```

[2]Using RandomizedSearchCV in scikit-learn, we can perform Randomized Grid Search. See [3]

```
gs = gs.fit(X_train, y_train)

clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

As note in [2] a great thing to keep in mind is

> Please note that fitting a model with the best settings (gs.best_estimator_) on the training set manually via clf.fit(X_train, y_train) after completing the grid search is not necessary. The GridSearchCV class has a refit parameter, which will refit the gs.best_estimator_ to the whole training set automatically if we set refit=True (default).

6.5 Nested Cross Validation

See [4]. Following [2], chapter 6, let's compare a decision tree and an svc.

```
from sklearn.tree import DecisionTreeClassifier

gs_svc = GridSearchCV(estimator=pipe_svc,param_grid=param_grid,scoring='accuracy',cv=2)
scores_svc = cross_val_score(gs_svc, X_train, y_train,scoring='accuracy', cv=5)

gs_tree = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
param_grid=[{'max_depth': [1, 2, 3,4, 5, 6,7, None]}],scoring='accuracy',cv=2)

scores_tree = cross_val_score(gs_tree, X_train, y_train,scoring='accuracy',cv=5)
```

7 References

[1]. Pandas: https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_r.html#quick-reference

[2]. Raschka, S. and Mirjalili, V., 2019. Python Machine Learning. Birmingham: Packt Publishing, Limited.

[3]. Random search for hyper-parameter optimization. Bergstra J, Bengio Y. Journal of Machine Learning Research. pp. 281-305, 2012

[4]. Bias in Error Estimation When Using Cross-Validation for Model Selection, BMC Bioinformatics, S. Varma and R. Simon, 7(1): 91, 2006