Python Data Wrangling and Classification

K Arnold

Example Dataset: Titanic Passengers

- https://www.openml.org/d/40945
- http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.html
- http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3info.txt
- https://www.encyclopedia-titanica.org/

Download the data, if we don't have it alreday:

```r
data_filename <- "data/titanic.csv"
if (!file.exists(data_filename)) {
  dir.create("data")
  download.file("https://www.openml.org/data/get_csv/16826755/phpMYEkMl", data_filename)
}
```

Python Setup

```r
library(reticulate)                                                    (r)
py_config()
```

```
## python:         /Users/ka37/Library/r-miniconda/envs/r-reticulate/bin/python
## libpython:      /Users/ka37/Library/r-miniconda/envs/r-reticulate/lib/libpython3.6m.dylib
## pythonhome:     /Users/ka37/Library/r-miniconda/envs/r-reticulate:/Users/ka37/Library/r-miniconda/envs/
## version:        3.6.11 | packaged by conda-forge | (default, Aug  5 2020, 20:19:23)  [GCC Clang 10.0.1
## numpy:          /Users/ka37/Library/r-miniconda/envs/r-reticulate/lib/python3.6/site-packages/numpy
## numpy_version:  1.19.1
```

The Python Data Science Toolbox

- **Pandas** (`pd`): the main library for wrangling tabular data in Python. (analogous to *tidyverse*)
- **NumPy** (`np`): the underlying math library. Gives us `array`s of numbers. Conventionally imported as `np`.
- **scikit-learn**: the main library for machine learning in Python.

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

Pandas

Loading data

Data frames in R are automatically converted into Pandas `DataFrame`s:

```
titanic <- read_csv("data/titanic.csv", na = "?")                    (r)
```

```
r.titanic.__class__                                                  (py)
```

```
## <class 'pandas.core.frame.DataFrame'>
```

Pandas can read CSV files itself. (CSV is such a quirky data format, so read the docs for all the parameters you can set.)

```
titanic = pd.read_csv("data/titanic.csv", na_values="?")             (py)
```

Exploring data structure

```python
titanic.shape
```

```
## (1309, 14)
```

```python
num_people, num_variables = titanic.shape
print(f"{num_people} people, {num_variables} variables about each")
```

```
## 1309 people, 14 variables about each
```

```python
titanic.head()
```

```
##    pclass  survived  ...   body                        home.dest
## 0       1         1  ...    NaN                     St Louis, MO
## 1       1         1  ...    NaN  Montreal, PQ / Chesterville, ON
## 2       1         0  ...    NaN  Montreal, PQ / Chesterville, ON
## 3       1         0  ...  135.0  Montreal, PQ / Chesterville, ON
## 4       1         0  ...    NaN  Montreal, PQ / Chesterville, ON
##
## [5 rows x 14 columns]
```

```python
titanic.info()
```

(py)

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 1309 entries, 0 to 1308
## Data columns (total 14 columns):
##  #   Column     Non-Null Count  Dtype
## ---  ------     --------------  -----
##  0   pclass     1309 non-null   int64
##  1   survived   1309 non-null   int64
##  2   name       1309 non-null   object
##  3   sex        1309 non-null   object
##  4   age        1046 non-null   float64
##  5   sibsp      1309 non-null   int64
##  6   parch      1309 non-null   int64
##  7   ticket     1309 non-null   object
##  8   fare       1308 non-null   float64
##  9   cabin      295 non-null    object
##  10  embarked   1307 non-null   object
##  11  boat       486 non-null    object
##  12  body       121 non-null    float64
##  13  home.dest  745 non-null    object
## dtypes: float64(3), int64(4), object(7)
## memory usage: 143.3+ KB
```

```
titanic.describe()
```
(py)

```
##              pclass     survived  ...         fare        body
## count  1309.000000  1309.000000  ...  1308.000000  121.000000
## mean      2.294882     0.381971  ...    33.295479  160.809917
## std       0.837836     0.486055  ...    51.758668   97.696922
## min       1.000000     0.000000  ...     0.000000    1.000000
## 25%       2.000000     0.000000  ...     7.895800   72.000000
## 50%       3.000000     0.000000  ...    14.454200  155.000000
## 75%       3.000000     1.000000  ...    31.275000  256.000000
## max       3.000000     1.000000  ...   512.329200  328.000000
##
## [8 rows x 7 columns]
```

Tidying data

Drop unneeded columns

```py
titanic2 = titanic.drop(['ticket', 'body'], axis = 1)
```

Rename columns

```py
titanic3 = titanic2.rename(columns={
  "pclass": "passenger_class",
  "survival": "survived",
  "sibsp": "num_siblings_or_spouses_aboard",
  "parch": "num_parents_or_children_aboard",
  "ticket": "ticket_num",
  "embarked": "embarked_from_port",
  "boat": "lifeboat",
})
```

Note that most (but not all!) Pandas methods make a *new* `DataFrame` (they don't modify the existing one).

Dropping missing data

This dataset has a lot of missing data in some columns. For demonstration purposes, we'll drop people where this data is missing, without investigating why. But in general:

Be careful about dropping missing data if you don't know why it's missing!

```py
titanic4 = titanic3.dropna(subset = ['age', 'fare', 'embarked_from_port'])
```

Querying data

Each column of a `pd.DataFrame` is a `pd.Series`, which is a NumPy `array` with (optional) labels.

```
titanic4['passenger_class']                    (py)
```

```
## 0       1
## 1       1
## 2       1
## 3       1
## 4       1
##        ..
## 1301    3
## 1304    3
## 1306    3
## 1307    3
## 1308    3
## Name: passenger_class, Length: 1043, dtype: int64
```

You can use Boolean operations on a `Series` to get another `Series`:

```
is_first_class = titanic4['passenger_class'] =(py)
is_first_class
```

```
## 0        True
## 1        True
## 2        True
## 3        True
## 4        True
##         ...
## 1301    False
## 1304    False
## 1306    False
## 1307    False
## 1308    False
## Name: passenger_class, Length: 1043, dtype: bool
```

How many rows does this Series have? How many columns?

Filtering data

You can use a Boolean series to query data. This syntax means: filter the data frame to include only the rows that correspond to a `True`:

```python
titanic4[is_first_class]
```

```
##      passenger_class  survived  ... lifeboat
## 0                  1         1  ...        2                    St Louis, MO
## 1                  1         1  ...       11  Montreal, PQ / Chesterville, ON
## 2                  1         0  ...      NaN  Montreal, PQ / Chesterville, ON
## 3                  1         0  ...      NaN  Montreal, PQ / Chesterville, ON
## 4                  1         0  ...      NaN  Montreal, PQ / Chesterville, ON
## ..               ...       ...  ...      ...                              ...
## 316                1         0  ...      NaN Geneva, Switzerland / Radnor, PA
## 317                1         1  ...        A Geneva, Switzerland / Radnor, PA
## 319                1         1  ...        3                              NaN
## 321                1         0  ...      NaN                      Halifax, NS
## 322                1         1  ...        8   New York, NY / Washington, DC
##
## [282 rows x 12 columns]
```

You can combine queries using Boolean operations (but they need to be the element-wise versions: `&`, `|`, and `~` instead of `and`, `or`, and `not`).

```python
had_companions = titanic4['num_siblings_or_spou
titanic4[is_first_class & had_companions]
```

```
##      passenger_class  survived  ... lifeboat
## 1                  1         1  ...       11  Mont
## 2                  1         0  ...      NaN  Mont
## 3                  1         0  ...      NaN  Mont
## 4                  1         0  ...      NaN  Mont
## 6                  1         1  ...       10
## ..               ...       ...  ...      ...
## 304                1         1  ...        5
## 310                1         0  ...      NaN
## 311                1         1  ...        8
## 312                1         0  ...      NaN
## 314                1         1  ...        4
##
```

Counting

You can get the counts of how many times each item occurs in a `Series`:

```
titanic4['passenger_class'].value_counts()                                    (py)
```

```
## 3    500
## 1    282
## 2    261
## Name: passenger_class, dtype: int64
```

Separating data into features and outcomes

sklearn needs the features to be in a separate data frame from the outcomes, so we need to split them apart ourselves. If we want to predict survival, we can create `y` as:

```python
y = titanic4['survived'] == 1
```
(py)

To create `X`, we can either drop the columns we don't want (see "tidying data" above) or directly ask for a list of columns we do want:

```python
#titanic4.columns # this can help you look at the column names
numeric_features = [
  'age', 'num_siblings_or_spouses_aboard', 'num_parents_or_children_aboard']
# We'll use these later
categorical_features = ['passenger_class', 'sex', 'embarked_from_port']

X = titanic4[numeric_features]
```
(py)

```
X.info()
```
(py)

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 1043 entries, 0 to 1308
## Data columns (total 3 columns):
##  #   Column                          Non-Null Count  Dtype
## ---  ------                          --------------  -----
##  0   age                             1043 non-null   float64
##  1   num_siblings_or_spouses_aboard  1043 non-null   int64
##  2   num_parents_or_children_aboard  1043 non-null   int64
## dtypes: float64(1), int64(2)
## memory usage: 32.6 KB
```

Scikit-Learn (`sklearn`)

Documentation and imports

The documentation is very well structured:

- the User Guide gives narrative documentation with background and examples (e.g., logistic regression)
- the API Reference gives the nitty-gritty details about individual classes and functions
- the Examples show worked examples of using most components.

It's conventional to import only what you actually need from `sklearn`:

```py
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_validate
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import accuracy_score, precision_score, recall_score
```

Train-Test Split

First, we'll hold out a test set of 10% of the passengers. We'll set a random seed so that this process is reproducible:

```
np.random.seed(0)                                                                              (py)
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.1)
X_train.shape, y_train.shape
```

```
## ((938, 3), (938,))
```

```
X_test.shape, y_test.shape                                                                     (py)
```

```
## ((105, 3), (105,))
```

Classifier API

All classifiers have the same basic interface: construct, `fit`, and `predict`.

We'll create a `LogisticRegression` object called `clf`, with the regularization parameter `C` set to 0.1.

```py
clf = LogisticRegression(C = 0.1, solver = "lbfgs")
clf.fit(X, y);
y_pred = clf.predict(X)
```

Metrics

The `sklearn.metrics` module implements a variety of useful metrics.

```
accuracy_score(y_true = y, y_pred = y_pred)                                    (py)
```

```
## 0.6184084372003835
```

```
precision_score(y_true = y, y_pred = y_pred)                                   (py)
```

```
## 0.651685393258427
```

```
recall_score(y_true = y, y_pred = y_pred)                                      (py)
```

```
## 0.13647058823529412
```

Remember that "recall" = true positive rate = *sensitivity*. sklearn doesn't directly implement *specificity*, but it does give us "precision" = positive predictive value (see Wikipedia).

Cross Validation

```python
cv_results = cross_validate(clf, X_train, y_train, cv=5,
  scoring=['accuracy', 'precision', 'recall'])

# Wrap the results in a DataFrame:
cv_results = pd.DataFrame(cv_results).reset_index()
```
(py)

We can now access this data in R.

```r
py$cv_results
```
(r)

```
##   index   fit_time  score_time test_accuracy test_precision test_recall
## 1     0 0.03214216 0.005925179     0.5957447      0.5333333   0.1038961
## 2     1 0.03038025 0.005378008     0.6329787      0.8333333   0.1298701
## 3     2 0.03310490 0.011760235     0.6276596      0.7058824   0.1558442
## 4     3 0.03105903 0.008498907     0.5882353      0.4782609   0.1447368
## 5     4 0.02601981 0.004664183     0.6203209      0.6666667   0.1315789
```

Column Transformers

Column transformers let us apply preprocessing steps to subsets of columns. For example, we'll scale the numeric features:

```py
numeric_feature_proc = StandardScaler()
```

and one-hot-encode the categorical features:

```py
categorical_feature_proc = OneHotEncoder()
```

And we'll apply each pre-processor to its corresponding columns:

```py
preprocessor = make_column_transformer(
  (numeric_feature_proc,     numeric_features),
  (categorical_feature_proc, categorical_features),
  remainder = 'drop')
```

Pipelines

Pipelines put several steps in sequence. Like *workflows* in `tidymodels`, we can use pipelines to say that the data should be preprocessed before running the model:

```
clf = make_pipeline(preprocessor, LogisticRegression())
```
(py)

Now we can use all of our features!

```
X = titanic4.drop(["survived"], axis = 1)
```
(py)

Redo the train-test split:

```
np.random.seed(0)
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.1)
```
(py)

Pipelines have same API as models (`fit`, `predict`)

```py
clf.fit(X, y);
```

Just as a demo, let's predict and score on the full training set. Remember that this is an overestimate of the accuracy we'd get on truly unseen data.

```py
y_pred = clf.predict(X)
accuracy_score(y_true = y, y_pred = y_pred)
```

```
## 0.7909875359539789
```

CV with pipelines

A pipeline behaves exactly like a classifier (it has `fit` and `predict`), so we can use exactly the same code to validate it.

```
cv_results = cross_validate(clf, X_train, y_train, cv=5,                (py)
  scoring=['accuracy', 'precision', 'recall'])

# Wrap the results in a DataFrame:
cv_results = pd.DataFrame(cv_results).reset_index()
```

We can now access this data in R.

```
py$cv_results                                                            (r)
```

```
##   index   fit_time  score_time test_accuracy test_precision test_recall
## 1     0 0.06636715 0.019825697     0.8138298      0.8088235   0.7142857
## 2     1 0.03014016 0.009572983     0.7925532      0.7567568   0.7272727
## 3     2 0.03230286 0.007397175     0.7553191      0.6781609   0.7662338
## 4     3 0.01353335 0.005396843     0.7433155      0.7121212   0.6184211
## 5     4 0.03007913 0.010706902     0.8074866      0.8030303   0.6973684
```