Open in Colab

T81-558: Applications of Deep Neural Networks

Module 2: Python for Machine Learning

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [Video] [Notebook]
- Part 2.2: Categorical Values [Video] [Notebook]
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [Video] [Notebook]
- **Part 2.4: Using Apply and Map in Pandas for Keras** [Video] [Notebook]
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 2.4: Apply and Map

If you've ever worked with Big Data or functional programming languages before, you've likely heard of map/reduce. Map and reduce are two functions that apply a task you create to a data frame. Pandas supports functional programming techniques that allow you to use functions across en entire data frame. In addition to functions that you write, Pandas also provides several standard functions for use with data frames.

Using Map with Dataframes

The map function allows you to transform a column by mapping certain values in that column to other values. Consider the Auto MPG data set that contains a field **origin_name** that holds a value between one and three that indicates the geographic origin of each car. We can see how to use the map function to transform this numeric origin into the textual name of each origin.

We will begin by loading the Auto MPG data set.

In [2]:
```
import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

The **map** method in Pandas operates on a single column. You provide **map** with a dictionary of values to transform the target column. The map keys specify what values in the target column should be turned into values specified by those keys. The following code shows how the map function can transform the numeric values of 1, 2, and 3 into the string values of North America, Europe, and Asia.

In [3]:
```
# Apply the map
df['origin_name'] = df['origin'].map(
    {1: 'North America', 2: 'Europe', 3: 'Asia'})

# Shuffle the data, so that we hopefully see
# more regions.
df = df.reindex(np.random.permutation(df.index))

# Display
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
display(df)
```

	mpg	cylinders	displacement	...	origin	name	origin_name
45	18.0	6	258.0	...	1	amc hornet sportabout (sw)	North America
290	15.5	8	351.0	...	1	ford country squire (sw)	North America
313	28.0	4	151.0	...	1	chevrolet citation	North America
82	23.0	4	120.0	...	3	toyouta corona mark ii (sw)	Asia
33	19.0	6	232.0	...	1	amc gremlin	North America
...	...	...	...	...	...	...	...
329	44.6	4	91.0	...	3	honda civic 1500 gl	Asia
326	43.4	4	90.0	...	2	vw dasher (diesel)	Europe
34	16.0	6	225.0	...	1	plymouth satellite custom	North America
118	24.0	4	116.0	...	2	opel manta	Europe
15	22.0	6	198.0	...	1	plymouth duster	North America

398 rows × 10 columns

Using Apply with Dataframes

The **apply** function of the data frame can run a function over the entire data frame. You can use either a traditional named function or a lambda function. Python will execute the provided function against each of the rows or columns in the data frame. The **axis** parameter specifies that the function is run across rows or columns. For axis = 1, rows are used. The following code calculates a series called **efficiency** that is the **displacement** divided by **horsepower**.

In [4]:
```
efficiency = df.apply(lambda x: x['displacement']/x['horsepower'], axis=1)
display(efficiency[0:10])
```

```
45     2.345455
290    2.471831
313    1.677778
82     1.237113
33     2.320000
249    2.363636
27     1.514286
7      2.046512
302    1.500000
179    1.234694
dtype: float64
```

You can now insert this series into the data frame, either as a new column or to replace an existing column. The following code inserts this new series into the data frame.

In [5]:
```
df['efficiency'] = efficiency
```

Feature Engineering with Apply and Map

In this section, we will see how to calculate a complex feature using map, apply, and grouping. The data set is the following CSV:

- https://www.irs.gov/pub/irs-soi/16zpallagi.csv

This URL contains US Government public data for "SOI Tax Stats - Individual Income Tax Statistics." The entry point to the website is here:

- https://www.irs.gov/statistics/soi-tax-stats-individual-income-tax-statistics-2016-zip-code-data-soi

Documentation describing this data is at the above link.

For this feature, we will attempt to estimate the adjusted gross income (AGI) for each of the zip codes. The data file contains many columns; however, you will only use the following:

- **STATE** - The state (e.g., MO)
- **zipcode** - The zipcode (e.g. 63017)
- **agi_stub** - Six different brackets of annual income (1 through 6)
- **N1** - The number of tax returns for each of the agi_stubs

Note, that the file will have six rows for each zip code for each of the agi_stub brackets. You can skip zip codes with 0 or 99999.

We will create an output CSV with these columns; however, only one row per zip code. Calculate a weighted average of the income brackets. For example, the following six rows are present for 63017:

zipcode	agi_stub	N1
63017	1	4710
63017	2	2780
63017	3	2130
63017	4	2010
63017	5	5240
63017	6	3510

We must combine these six rows into one. For privacy reasons, AGI's are broken out into 6 buckets. We need to combine the buckets and estimate the actual AGI of a zipcode. To do this, consider the values for N1:

- 1 = 1 to 25,000
- 2 = 25,000 to 50,000
- 3 = 50,000 to 75,000
- 4 = 75,000 to 100,000
- 5 = 100,000 to 200,000
- 6 = 200,000 or more

The median of each of these ranges is approximately:

- 1 = 12,500
- 2 = 37,500
- 3 = 62,500
- 4 = 87,500
- 5 = 112,500
- 6 = 212,500

Using this, you can estimate 63017's average AGI as:

```
>>> totalCount = 4710 + 2780 + 2130 + 2010 + 5240 + 3510
>>> totalAGI = 4710 * 12500 + 2780 * 37500 + 2130 * 62500
    + 2010 * 87500 + 5240 * 112500 + 3510 * 212500
>>> print(totalAGI / totalCount)

88689.89205103042
```

We begin by reading the government data.

In [6]:
```
import pandas as pd
```

```
df=pd.read_csv('https://www.irs.gov/pub/irs-soi/16zpallagi.csv')
```

First, we trim all zip codes that are either 0 or 99999. We also select the three fields that we need.

In [7]:
```
df=df.loc[(df['zipcode']!=0) & (df['zipcode']!=99999),
          ['STATE','zipcode','agi_stub','N1']]

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

	STATE	zipcode	agi_stub	N1
6	AL	35004	1	1510
7	AL	35004	2	1410
8	AL	35004	3	950
9	AL	35004	4	650
10	AL	35004	5	630
...	...	...	...	...
179785	WY	83414	2	40
179786	WY	83414	3	40
179787	WY	83414	4	0
179788	WY	83414	5	40
179789	WY	83414	6	30

179184 rows × 4 columns

We replace all of the **agi_stub** values with the correct median values with the **map** function.

In [8]:
```
medians = {1:12500,2:37500,3:62500,4:87500,5:112500,6:212500}
df['agi_stub']=df.agi_stub.map(medians)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)
display(df)
```

	STATE	zipcode	agi_stub	N1
6	AL	35004	12500	1510
7	AL	35004	37500	1410
8	AL	35004	62500	950
9	AL	35004	87500	650
10	AL	35004	112500	630
...	...	...	...	...
179785	WY	83414	37500	40
179786	WY	83414	62500	40
179787	WY	83414	87500	0
179788	WY	83414	112500	40
179789	WY	83414	212500	30

179184 rows × 4 columns

Next, we group the data frame by zip code.

In [9]:
```
groups = df.groupby(by='zipcode')
```

The program applies a lambda across the groups and calculates the AGI estimate.

In [11]:
```
df = pd.DataFrame(groups.apply(
    lambda x:sum(x['N1']*x['agi_stub'])/sum(x['N1']))) \
    .reset_index()

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

	zipcode	0
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	99929	45911.214953

29872 rows × 2 columns

We can now rename the new **agi_estimate** column.

In [13]:
```
df.columns = ['zipcode','agi_estimate']

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

	zipcode	agi_estimate
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	99929	45911.214953

29872 rows × 2 columns

Finally, we check to see that our zip code of 63017 got the correct value.

In [14]:
```
df[ df['zipcode']==63017 ]
```

Out[14]:

	zipcode	agi_estimate
19909	63017	88689.892051