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:
    from google.colab import drive
    %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 2.1: Introduction to Pandas

Pandas is an open-source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It is based on the dataframe concept found in the R programming language. For this class, Pandas will be the primary means by which we manipulate data to be processed by neural networks.

The data frame is a crucial component of Pandas. We will use it to access the auto-mpg dataset. You can find this dataset on the UCI machine learning repository. For this class, we will use a version of the Auto MPG dataset, where I added column headers. You can find my version at https://data.heatonresearch.com/.

UCI took this dataset from the StatLib library, which Carnegie Mellon University maintains. The dataset was used in the 1983 American Statistical Association Exposition. It contains data for 398 cars, including mpg, cylinders, displacement, horsepower , weight, acceleration, model year, origin and the car's name.

The following code loads the MPG dataset into a data frame:

In [2]:
```
# Simple dataframe
import os
import pandas as pd

pd.set_option('display.max_columns', 7)
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv")
display(df[0:5])
```

	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
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

5 rows × 9 columns

The **display** function provides a cleaner display than merely printing the data frame. Specifying the maximum rows and columns allows you to achieve greater control over the display.

In [3]:
```
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

It is possible to generate a second data frame to display statistical information about the first data frame.

In [4]:
```
# Strip non-numerics
df = df.select_dtypes(include=['int', 'float'])

headers = list(df.columns.values)
fields = []

for field in headers:
    fields.append({
        'name' : field,
        'mean': df[field].mean(),
        'var': df[field].var(),
        'sdev': df[field].std()
    })

for field in fields:
    print(field)
```

```
{'name': 'mpg', 'mean': 23.514572864321615, 'var': 61.089610774274405, 'sde
v': 7.815984312565782}
{'name': 'cylinders', 'mean': 5.454773869346734, 'var': 2.8934154399199943,
'sdev': 1.7010042445332094}
{'name': 'displacement', 'mean': 193.42587939698493, 'var': 10872.1991522473
64, 'sdev': 104.26983817119581}
{'name': 'weight', 'mean': 2970.424623115578, 'var': 717140.9905256768, 'sde
v': 846.8417741973271}
{'name': 'acceleration', 'mean': 15.568090452261291, 'var': 7.60484823361138
1, 'sdev': 2.7576889298126757}
{'name': 'year', 'mean': 76.01005025125629, 'var': 13.672442818627143, 'sde
v': 3.697626646732623}
{'name': 'origin', 'mean': 1.5728643216080402, 'var': 0.6432920268850575, 's
dev': 0.8020548777266163}
```

This code outputs a list of dictionaries that hold this statistical information. This information looks similar to the JSON code seen in Module 1. If proper JSON is needed, the program should add these records to a list and call the Python JSON library's **dumps** command.

The Python program can convert this JSON-like information to a data frame for better display.

In [5]:
```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
df2 = pd.DataFrame(fields)
display(df2)
```

	name	mean	var	sdev
0	mpg	23.514573	61.089611	7.815984
1	cylinders	5.454774	2.893415	1.701004
2	displacement	193.425879	10872.199152	104.269838
3	weight	2970.424623	717140.990526	846.841774
4	acceleration	15.568090	7.604848	2.757689
5	year	76.010050	13.672443	3.697627
6	origin	1.572864	0.643292	0.802055

Missing Values

Missing values are a reality of machine learning. Ideally, every row of data will have values for all columns. However, this is rarely the case. Most of the values are present in the MPG database. However, there are missing values in the horsepower column. A common practice is to replace missing values with the median value for that column. The program calculates the median. The following code replaces any NA values in horsepower with the median:

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

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])
print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")

print("Filling missing values...")
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)
# df = df.dropna() # you can also simply drop NA values

print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")
```

```
horsepower has na? True
Filling missing values...
horsepower has na? False
```

Dealing with Outliers

Outliers are values that are unusually high or low. We typically consider outliers to be a value that is several standard deviations from the mean. Sometimes outliers are simply errors; this is a result of observation error. Outliers can also be truly large or small values that may be difficult to address. The following function can remove such values.

In [7]:
```
# Remove all rows where the specified column is +/- sd standard deviations
def remove_outliers(df, name, sd):
    drop_rows = df.index[(np.abs(df[name] - df[name].mean())
                          >= (sd * df[name].std()))]
    df.drop(drop_rows, axis=0, inplace=True)
```

The code below will drop every row from the Auto MPG dataset where the horsepower is two standard deviations or more above or below the mean.

In [8]:
```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore

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

# create feature vector
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)

# Drop the name column
df.drop('name',1,inplace=True)

# Drop outliers in horsepower
print("Length before MPG outliers dropped: {}".format(len(df)))
remove_outliers(df,'mpg',2)
print("Length after MPG outliers dropped: {}".format(len(df)))

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(df)
```

```
Length before MPG outliers dropped: 398
Length after MPG outliers dropped: 388
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	or
0	18.0	8	307.0	130.0	3504	12.0	70	
1	15.0	8	350.0	165.0	3693	11.5	70	
...	...	...	...	...	...	...	...	
396	28.0	4	120.0	79.0	2625	18.6	82	
397	31.0	4	119.0	82.0	2720	19.4	82	

388 rows × 8 columns

Dropping Fields

You must drop fields that are of no value to the neural network. The following code removes the name column from the MPG dataset.

In [9]:
```
import os
import pandas as pd

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

print(f"Before drop: {list(df.columns)}")
df.drop('name', 1, inplace=True)
print(f"After drop: {list(df.columns)}")
```

```
Before drop: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'a
cceleration', 'year', 'origin', 'name']
After drop: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'ac
celeration', 'year', 'origin']
```

Concatenating Rows and Columns

Python can concatenate rows and columns together to form new data frames. The code below creates a new data frame from the **name** and **horsepower** columns from the Auto MPG dataset. The program does this by concatenating two columns together.

In [10]:
```
# Create a new dataframe from name and horsepower

import os
import pandas as pd

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

col_horsepower = df['horsepower']
```

```
col_name = df['name']
result = pd.concat([col_name, col_horsepower], axis=1)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(result)
```

	name	horsepower
0	chevrolet chevelle malibu	130.0
1	buick skylark 320	165.0
...	...	...
396	ford ranger	79.0
397	chevy s-10	82.0

398 rows × 2 columns

The **concat** function can also concatenate rows together. This code concatenates the first two rows and the last two rows of the Auto MPG dataset.

In [11]:
```
# Create a new dataframe from first 2 rows and last 2 rows

import os
import pandas as pd

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

result = pd.concat([df[0:2],df[-2:]], axis=0)

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 0)
display(result)
```

	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

4 rows × 9 columns

Training and Validation

We must evaluate a machine learning model based on its ability to predict values that it has never seen before. Because of this, we often divide the training data into a validation and training set. The machine learning model will learn from the training data but ultimately be evaluated based on the validation data.

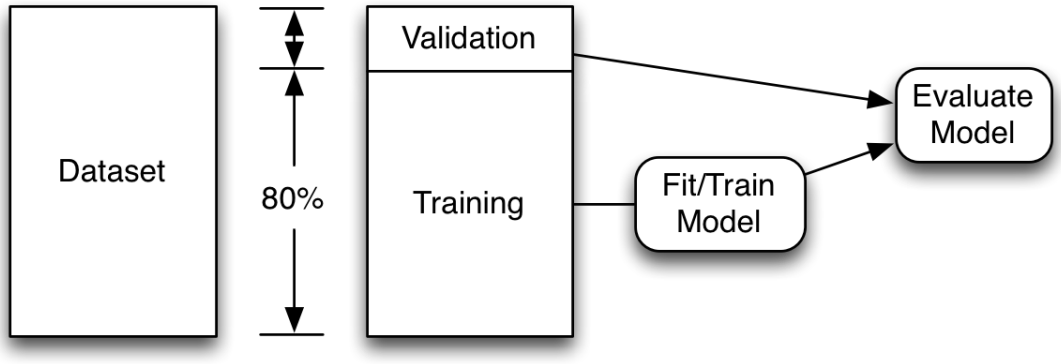
- **Training Data** - **In Sample Data** - The data that the neural network used to train.
- **Validation Data** - **Out of Sample Data** - The data that the machine learning model is evaluated upon after it is fit to the training data.

There are two effective means of dealing with training and validation data:

- **Training/Validation Split** - The program splits the data according to some ratio between a training and validation (hold-out) set. Typical rates are 80% training and 20% validation.
- **K-Fold Cross Validation** - The program splits the data into several folds and models. Because the program creates the same number of models as folds, the program can generate out-of-sample predictions for the entire dataset.

The code below splits the MPG data into a training and validation set. The training set uses 80% of the data, and the validation set uses 20%. Figure 2.TRN-VAL shows how we train a model on 80% of the data and then validated against the remaining 20%.

Figure 2.TRN-VAL: Training and Validation



In [12]:
```
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','?'])

# Usually a good idea to shuffle
df = df.reindex(np.random.permutation(df.index))
```

```
mask = np.random.rand(len(df)) < 0.8
trainDF = pd.DataFrame(df[mask])
validationDF = pd.DataFrame(df[~mask])

print(f"Training DF: {len(trainDF)}")
print(f"Validation DF: {len(validationDF)}")
```

```
Training DF: 333
Validation DF: 65
```

Converting a Dataframe to a Matrix

Neural networks do not directly operate on Python data frames. A neural network requires a numeric matrix. The program uses a data frame's **values** property to convert the data to a matrix.

In [13]:
```
df.values
```

Out[13]:
```
array([[20.2, 6, 232.0, ..., 79, 1, 'amc concord dl 6'],
       [14.0, 8, 304.0, ..., 74, 1, 'amc matador (sw)'],
       [14.0, 8, 351.0, ..., 71, 1, 'ford galaxie 500'],
       ...,
       [20.2, 6, 200.0, ..., 78, 1, 'ford fairmont (auto)'],
       [26.0, 4, 97.0, ..., 70, 2, 'volkswagen 1131 deluxe sedan'],
       [19.4, 6, 232.0, ..., 78, 1, 'amc concord']], dtype=object)
```

You might wish only to convert some of the columns, to leave out the name column, use the following code.

In [14]:
```
df[['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
```

Out[14]:
```
array([[ 20.2,   6. , 232. , ...,  18.2,  79. ,   1. ],
       [ 14. ,   8. , 304. , ...,  15.5,  74. ,   1. ],
       [ 14. ,   8. , 351. , ...,  13.5,  71. ,   1. ],
       ...,
       [ 20.2,   6. , 200. , ...,  15.8,  78. ,   1. ],
       [ 26. ,   4. ,  97. , ...,  20.5,  70. ,   2. ],
       [ 19.4,   6. , 232. , ...,  17.2,  78. ,   1. ]])
```

Saving a Dataframe to CSV

Many of the assignments in this course will require that you save a data frame to submit to the instructor. The following code performs a shuffle and then saves a new copy.

In [15]:
```
import os
import pandas as pd
import numpy as np
```

```
path = "."

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

filename_write = os.path.join(path, "auto-mpg-shuffle.csv")
df = df.reindex(np.random.permutation(df.index))
# Specify index = false to not write row numbers
df.to_csv(filename_write, index=False)
```

```
Done
```

Saving a Dataframe to Pickle

A variety of software programs can use text files stored as CSV. However, they take longer to generate and can sometimes lose small amounts of precision in the conversion. Generally, you will output to CSV because it is very compatible, even outside of Python. Another format is Pickle. The code below stores the Dataframe to Pickle. Pickle stores data in the exact binary representation used by Python. The benefit is that there is no loss of data going to CSV format. The disadvantage is that generally, only Python programs can read Pickle files.

In [16]:
```
import os
import pandas as pd
import numpy as np
import pickle

path = "."

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

filename_write = os.path.join(path, "auto-mpg-shuffle.pkl")
df = df.reindex(np.random.permutation(df.index))

with open(filename_write,"wb") as fp:
    pickle.dump(df, fp)
```

Loading the pickle file back into memory is accomplished by the following lines of code. Notice that the index numbers are still jumbled from the previous shuffle? Loading the CSV rebuilt (in the last step) did not preserve these values.

In [17]:
```
import os
import pandas as pd
import numpy as np
import pickle

path = "."

df = pd.read_csv(
```

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

filename_read = os.path.join(path, "auto-mpg-shuffle.pkl")

with open(filename_write,"rb") as fp:
    df = pickle.load(fp)

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

	mpg	cylinders	displacement	...	year	origin	name
387	38.0	6	262.0	...	82	1	oldsmobile cutlass ciera (diesel)
361	25.4	6	168.0	...	81	3	toyota cressida
...	...	...	...	...	...	...	...
358	31.6	4	120.0	...	81	3	mazda 626
237	30.5	4	98.0	...	77	1	chevrolet chevette

398 rows × 9 columns

Module 2 Assignment

You can find the first assignment here: assignment 2

In [ ]:

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 2.2: Categorical and Continuous Values

Neural networks require their input to be a fixed number of columns. This input format is very similar to spreadsheet data; it must be entirely numeric. It is essential to represent the data so that the neural network can train from it. Before we look at specific ways to preprocess data, it is important to consider four basic types of data, as defined by [Cite:stevens1946theory]. Statisticians commonly refer to as the levels of measure:

- Character Data (strings)
  - **Nominal** - Individual discrete items, no order. For example, color, zip code, and shape.
  - **Ordinal** - Individual distinct items have an implied order. For example, grade level, job title, Starbucks(tm) coffee size (tall, vente, grande)
- Numeric Data
  - **Interval** - Numeric values, no defined start. For example, temperature. You would never say, "yesterday was twice as hot as today."
  - **Ratio** - Numeric values, clearly defined start. For example, speed. You could say, "The first car is going twice as fast as the second."

Encoding Continuous Values

One common transformation is to normalize the inputs. It is sometimes valuable to normalize numeric inputs in a standard form so that the program can easily compare these two values. Consider if a friend told you that he received a 10-dollar discount. Is this a good deal? Maybe. But the cost is not normalized. If your friend purchased a car, the discount is not that good. If your friend bought lunch, this is an excellent discount!

Percentages are a prevalent form of normalization. If your friend tells you they got 10% off, we know that this is a better discount than 5%. It does not matter how much the purchase price was. One widespread machine learning normalization is the Z-Score:

$$z = \frac{x - \mu}{\sigma}$$

To calculate the Z-Score, you also need to calculate the mean(μ or $\bar{x}$) and the standard deviation (σ). You can calculate the mean with this equation:

$$\mu = \bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

The standard deviation is calculated as follows:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

The following Python code replaces the mpg with a z-score. Cars with average MPG will be near zero, above zero is above average, and below zero is below average. Z-Scores more that 3 above or below are very rare; these are outliers.

In [2]:
```
import os
import pandas as pd
from scipy.stats import zscore

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)

df['mpg'] = zscore(df['mpg'])
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	-0.706439	8	307.0	...	70	1	chevrolet chevelle malibu
1	-1.090751	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	0.574601	4	120.0	...	82	1	ford ranger
397	0.958913	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

Encoding Categorical Values as Dummies

The traditional means of encoding categorical values is to make them dummy variables. This technique is also called one-hot-encoding. Consider the following data set.

In [3]:
```
import pandas as pd

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

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

```
display(df)
```

	id	job	area	...	retail_dense	crime	product
0	1	vv	c	...	0.492126	0.071100	b
1	2	kd	c	...	0.342520	0.400809	c
...	...	...	...	...	...	...	...
1998	1999	qp	c	...	0.598425	0.117803	c
1999	2000	pe	c	...	0.539370	0.451973	c

2000 rows × 14 columns

The *area* column is not numeric, so you must encode it with one-hot encoding. We display the number of areas and individual values. There are just four values in the *area* categorical variable in this case.

In [4]:
```
areas = list(df['area'].unique())
print(f'Number of areas: {len(areas)}')
print(f'Areas: {areas}')
```

```
Number of areas: 4
Areas: ['c', 'd', 'a', 'b']
```

There are four unique values in the *area* column. To encode these dummy variables, we would use four columns, each representing one of the areas. For each row, one column would have a value of one, the rest zeros. For this reason, this type of encoding is sometimes called one-hot encoding. The following code shows how you might encode the values "a" through "d." The value A becomes [1,0,0,0] and the value B becomes [0,1,0,0].

In [5]:
```
dummies = pd.get_dummies(['a','b','c','d'],prefix='area')
print(dummies)
```

```
   area_a  area_b  area_c  area_d
0       1       0       0       0
1       0       1       0       0
2       0       0       1       0
3       0       0       0       1
```

We can now encode the actual column.

In [6]:
```
dummies = pd.get_dummies(df['area'],prefix='area')
print(dummies[0:10]) # Just show the first 10
```

```
   area_a  area_b  area_c  area_d
0       0       0       1       0
1       0       0       1       0
..    ...     ...     ...     ...
8       0       0       1       0
9       1       0       0       0

[10 rows x 4 columns]
```

For the new dummy/one hot encoded values to be of any use, they must be merged back into the data set.

In [7]:
```
df = pd.concat([df,dummies],axis=1)
```

To encode the *area* column, we use the following code. Note that it is necessary to merge these dummies back into the data frame.

In [8]:
```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','area','income','area_a',
                  'area_b','area_c','area_d']])
```

	id	job	area	income	area_a	area_b	area_c	area_d
0	1	vv	c	50876.0	0	0	1	0
1	2	kd	c	60369.0	0	0	1	0
2	3	pe	c	55126.0	0	0	1	0
3	4	11	c	51690.0	0	0	1	0
4	5	kl	d	28347.0	0	0	0	1
...	...	...	...	...	...	...	...	...
1995	1996	vv	c	51017.0	0	0	1	0
1996	1997	kl	d	26576.0	0	0	0	1
1997	1998	kl	d	28595.0	0	0	0	1
1998	1999	qp	c	67949.0	0	0	1	0
1999	2000	pe	c	61467.0	0	0	1	0

2000 rows × 8 columns

Usually, you will remove the original column *area* because the goal is to get the data frame to be entirely numeric for the neural network.

In [9]:
```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)

df.drop('area', axis=1, inplace=True)
```

```
display(df[['id','job','income','area_a',
                'area_b','area_c','area_d']])
```

	id	job	income	area_a	area_b	area_c	area_d
0	1	vv	50876.0	0	0	1	0
1	2	kd	60369.0	0	0	1	0
...	...	...	...	...	...	...	...
1998	1999	qp	67949.0	0	0	1	0
1999	2000	pe	61467.0	0	0	1	0

2000 rows × 7 columns

Removing the First Level

The **pd.concat** function also includes a parameter named *drop_first*, which specifies whether to get k-1 dummies out of k categorical levels by removing the first level. Why would you want to remove the first level, in this case, *area_a*? This technique provides a more efficient encoding by using the ordinarily unused encoding of [0,0,0]. We encode the *area* to just three columns and map the categorical value of *a* to [0,0,0]. The following code demonstrates this technique.

In [10]:
```
import pandas as pd

dummies = pd.get_dummies(['a','b','c','d'],prefix='area', drop_first=True)
print(dummies)
```

```
   area_b  area_c  area_d
0       0       0       0
1       1       0       0
2       0       1       0
3       0       0       1
```

As you can see from the above data, the *area_a* column is missing, as it **get_dummies** replaced it by the encoding of [0,0,0]. The following code shows how to apply this technique to a dataframe.

In [11]:
```
import pandas as pd

# Read the dataset
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# encode the area column as dummy variables
dummies = pd.get_dummies(df['area'], drop_first=True, prefix='area')
df = pd.concat([df,dummies],axis=1)
df.drop('area', axis=1, inplace=True)
```

```
# display the encoded dataframe
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','income',
                  'area_b','area_c','area_d']])
```

	id	job	income	area_b	area_c	area_d
0	1	vv	50876.0	0	1	0
1	2	kd	60369.0	0	1	0
2	3	pe	55126.0	0	1	0
3	4	11	51690.0	0	1	0
4	5	kl	28347.0	0	0	1
...	...	...	...	...	...	...
1995	1996	vv	51017.0	0	1	0
1996	1997	kl	26576.0	0	0	1
1997	1998	kl	28595.0	0	0	1
1998	1999	qp	67949.0	0	1	0
1999	2000	pe	61467.0	0	1	0

2000 rows × 6 columns

Target Encoding for Categoricals

Target encoding is a popular technique for Kaggle competitions. Target encoding can sometimes increase the predictive power of a machine learning model. However, it also dramatically increases the risk of overfitting. Because of this risk, you must take care of using this method.

Generally, target encoding can only be used on a categorical feature when the output of the machine learning model is numeric (regression).

The concept of target encoding is straightforward. For each category, we calculate the average target value for that category. Then to encode, we substitute the percent corresponding to the category that the categorical value has. Unlike dummy variables, where you have a column for each category with target encoding, the program only needs a single column. In this way, target coding is more efficient than dummy variables.

In [13]:
```
# Create a small sample dataset
import pandas as pd
import numpy as np
```

```
np.random.seed(43)
df = pd.DataFrame({
    'cont_9': np.random.rand(10)*100,
    'cat_0': ['dog'] * 5 + ['cat'] * 5,
    'cat_1': ['wolf'] * 9 + ['tiger'] * 1,
    'y': [1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
})

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

	cont_9	cat_0	cat_1	y
0	11.505457	dog	wolf	1
1	60.906654	dog	wolf	0
2	13.339096	dog	wolf	1
3	24.058962	dog	wolf	1
4	32.713906	dog	wolf	1
5	85.913749	cat	wolf	1
6	66.609021	cat	wolf	0
7	54.116221	cat	wolf	0
8	2.901382	cat	wolf	0
9	73.374830	cat	tiger	0

We want to change them to a number rather than creating dummy variables for "dog" and "cat," we would like to change them to a number. We could use 0 for a cat and 1 for a dog. However, we can encode more information than just that. The simple 0 or 1 would also only work for one animal. Consider what the mean target value is for cat and dog.

In [14]:
```
means0 = df.groupby('cat_0')['y'].mean().to_dict()
means0
```

Out[14]:
```
{'cat': 0.2, 'dog': 0.8}
```

The danger is that we are now using the target value (y) for training. This technique will potentially lead to overfitting. The possibility of overfitting is even greater if a small number of a particular category. To prevent this from happening, we use a weighting factor. The stronger the weight, the more categories with fewer values will tend towards the overall average of y. You can perform this calculation as follows.

In [15]:
```
df['y'].mean()
```

Out[15]: 0.5

You can implement target encoding as follows. For more information on Target Encoding, refer to the article "Target Encoding Done the Right Way", that I based this code upon.

In [16]:
```
def calc_smooth_mean(df1, df2, cat_name, target, weight):
    # Compute the global mean
    mean = df[target].mean()

    # Compute the number of values and the mean of each group
    agg = df.groupby(cat_name)[target].agg(['count', 'mean'])
    counts = agg['count']
    means = agg['mean']

    # Compute the "smoothed" means
    smooth = (counts * means + weight * mean) / (counts + weight)

    # Replace each value by the according smoothed mean
    if df2 is None:
        return df1[cat_name].map(smooth)
    else:
        return df1[cat_name].map(smooth),df2[cat_name].map(smooth.to_dict())
```

The following code encodes these two categories.

In [17]:
```
WEIGHT = 5
df['cat_0_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_0', target='y', weight=WEIGHT)
df['cat_1_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_1', target='y', weight=WEIGHT)

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

display(df)
```

	cont_9	cat_0	cat_1	y	cat_0_enc	cat_1_enc
0	11.505457	dog	wolf	1	0.65	0.535714
1	60.906654	dog	wolf	0	0.65	0.535714
2	13.339096	dog	wolf	1	0.65	0.535714
3	24.058962	dog	wolf	1	0.65	0.535714
4	32.713906	dog	wolf	1	0.65	0.535714
5	85.913749	cat	wolf	1	0.35	0.535714
6	66.609021	cat	wolf	0	0.35	0.535714
7	54.116221	cat	wolf	0	0.35	0.535714
8	2.901382	cat	wolf	0	0.35	0.535714
9	73.374830	cat	tiger	0	0.35	0.416667

Encoding Categorical Values as Ordinal

Typically categoricals will be encoded as dummy variables. However, there might be other techniques to convert categoricals to numeric. Any time there is an order to the categoricals, a number should be used. Consider if you had a categorical that described the current education level of an individual.

- Kindergarten (0)
- First Grade (1)
- Second Grade (2)
- Third Grade (3)
- Fourth Grade (4)
- Fifth Grade (5)
- Sixth Grade (6)
- Seventh Grade (7)
- Eighth Grade (8)
- High School Freshman (9)
- High School Sophomore (10)
- High School Junior (11)
- High School Senior (12)
- College Freshman (13)
- College Sophomore (14)
- College Junior (15)
- College Senior (16)
- Graduate Student (17)
- PhD Candidate (18)
- Doctorate (19)

- Post Doctorate (20)

The above list has 21 levels and would take 21 dummy variables to encode. However, simply encoding this to dummies would lose the order information. Perhaps the most straightforward approach would be to simply number them and assign the category a single number equal to the value in the parenthesis above. However, we might be able to do even better. A graduate student is likely more than a year so you might increase one value.

High Cardinality Categorical

If there were many, perhaps thousands or tens of thousands, then one-hot encoding is no longer a good choice. We call these cases high cardinality categorical. We generally encode such values with an embedding layer, which we will discuss later when introducing natural language processing (NLP).

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 2.3: Grouping, Sorting, and Shuffling

We will take a look at a few ways to affect an entire Pandas data frame. These techniques will allow us to group, sort, and shuffle data sets. These are all essential operations for both data preprocessing and evaluation.

Shuffling a Dataset

There may be information lurking in the order of the rows of your dataset. Unless you are dealing with time-series data, the order of the rows should not be significant. Consider if your training set included employees in a company. Perhaps this dataset is ordered by the number of years the employees were with the company. It is okay to have an individual column that specifies years of service. However, having the data in this order might be problematic.

Consider if you were to split the data into training and validation. You could end up with your validation set having only the newer employees and the training set longer-term employees. Separating the data into a k-fold cross validation could have similar problems. Because of these issues, it is important to shuffle the data set.

Often shuffling and reindexing are both performed together. Shuffling randomizes the order of the data set. However, it does not change the Pandas row numbers. The following code demonstrates a reshuffle. Notice that the program has not reset the row indexes' first column. Generally, this will not cause any issues and allows tracing back to the original order of the data. However, I usually prefer to reset this index. I reason that I typically do not care about the initial position, and there are a few instances where this unordered index can cause issues.

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', '?'])

#np.random.seed(42) # Uncomment this line to get the same shuffle each time
df = df.reindex(np.random.permutation(df.index))

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

	mpg	cylinders	displacement	...	year	origin	name
117	29.0	4	68.0	...	73	2	fiat 128
245	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
88	14.0	8	302.0	...	73	1	ford gran torino
26	10.0	8	307.0	...	70	1	chevy c20

398 rows × 9 columns

The following code demonstrates a reindex. Notice how the reindex orders the row indexes.

In [3]:
```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df.reset_index(inplace=True, drop=True)
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	29.0	4	68.0	...	73	2	fiat 128
1	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
396	14.0	8	302.0	...	73	1	ford gran torino
397	10.0	8	307.0	...	70	1	chevy c20

398 rows × 9 columns

Sorting a Data Set

While it is always good to shuffle a data set before training, during training and preprocessing, you may also wish to sort the data set. Sorting the data set allows you to order the rows in either ascending or descending order for one or more columns. The following code sorts the MPG dataset by name and displays the first car.

In [4]:
```
import os
import pandas as pd

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

df = df.sort_values(by='name', ascending=True)
print(f"The first car is: {df['name'].iloc[0]}")
```

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

```
The first car is: amc ambassador brougham
```

	mpg	cylinders	displacement	...	year	origin	name
96	13.0	8	360.0	...	73	1	amc ambassador brougham
9	15.0	8	390.0	...	70	1	amc ambassador dpl
...	...	...	...	...	...	...	...
325	44.3	4	90.0	...	80	2	vw rabbit c (diesel)
293	31.9	4	89.0	...	79	2	vw rabbit custom

398 rows × 9 columns

Grouping a Data Set

Grouping is a typical operation on data sets. Structured Query Language (SQL) calls this operation a "GROUP BY." Programmers use grouping to summarize data. Because of this, the summarization row count will usually shrink, and you cannot undo the grouping. Because of this loss of information, it is essential to keep your original data before the grouping.

We use the Auto MPG dataset to demonstrate grouping.

In [5]:
```
import os
import pandas as pd

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

You can use the above data set with the group to perform summaries. For example, the following code will group cylinders by the average (mean). This code will provide the grouping. In addition to **mean**, you can use other aggregating functions, such as **sum** or **count**.

In [6]:
```
g = df.groupby('cylinders')['mpg'].mean()
g
```

Out[6]:
```
cylinders
3    20.550000
4    29.286765
5    27.366667
6    19.985714
8    14.963107
Name: mpg, dtype: float64
```

It might be useful to have these **mean** values as a dictionary.

In [7]:
```
d = g.to_dict()
d
```

Out[7]:
```
{3: 20.55,
 4: 29.28676470588236,
 5: 27.366666666666664,
 6: 19.985714285714284,
 8: 14.963106796116508}
```

A dictionary allows you to access an individual element quickly. For example, you could quickly look up the mean for six-cylinder cars. You will see that target encoding, introduced later in this module, uses this technique.

In [8]:
```
d[6]
```

Out[8]:
```
19.985714285714284
```

The code below shows how to count the number of rows that match each cylinder count.

In [9]:
```
df.groupby('cylinders')['mpg'].count().to_dict()
```

Out[9]:
```
{3: 4, 4: 204, 5: 3, 6: 84, 8: 103}
```

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

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 2.5: Feature Engineering

Feature engineering is an essential part of machine learning. For now, we will manually engineer features. However, later in this course, we will see some techniques for automatic feature engineering.

Calculated Fields

It is possible to add new fields to the data frame that your program calculates from the other fields. We can create a new column that gives the weight in kilograms. The equation to calculate a metric weight, given weight in pounds, is:

$$m_{(kg)} = m_{(lb)} \times 0.45359237$$

The following Python code performs this transformation:

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

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

df.insert(1, 'weight_kg', (df['weight'] * 0.45359237).astype(int))
pd.set_option('display.max_columns', 6)
pd.set_option('display.max_rows', 5)
df
```

Out[2]:

	mpg	weight_kg	cylinders	...	year	origin	name
0	18.0	1589	8	...	70	1	chevrolet chevelle malibu
1	15.0	1675	8	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	1190	4	...	82	1	ford ranger
397	31.0	1233	4	...	82	1	chevy s-10

398 rows × 10 columns

Google API Keys

Sometimes you will use external APIs to obtain data. The following examples show how to use the Google API keys to encode addresses for use with neural networks. To use these, you will need your own Google API key. The key I have below is not a real key; you need to put your own there. Google will ask for a credit card, but there will be no actual cost unless you use a massive number of lookups. YOU ARE NOT required to get a Google API key for this class; this only

shows you how. If you want to get a Google API key, visit this site and obtain one for **geocode**.

You can obtain your key from this link: Google API Keys.

In [3]:
```
if 'GOOGLE_API_KEY' in os.environ:
    # If the API key is defined in an environmental variable,
    # the use the env variable.
    GOOGLE_KEY = os.environ['GOOGLE_API_KEY']
else:
    # If you have a Google API key of your own, you can also just
    # put it here:
    GOOGLE_KEY = 'REPLACE WITH YOUR GOOGLE API KEY'
```

Other Examples: Dealing with Addresses

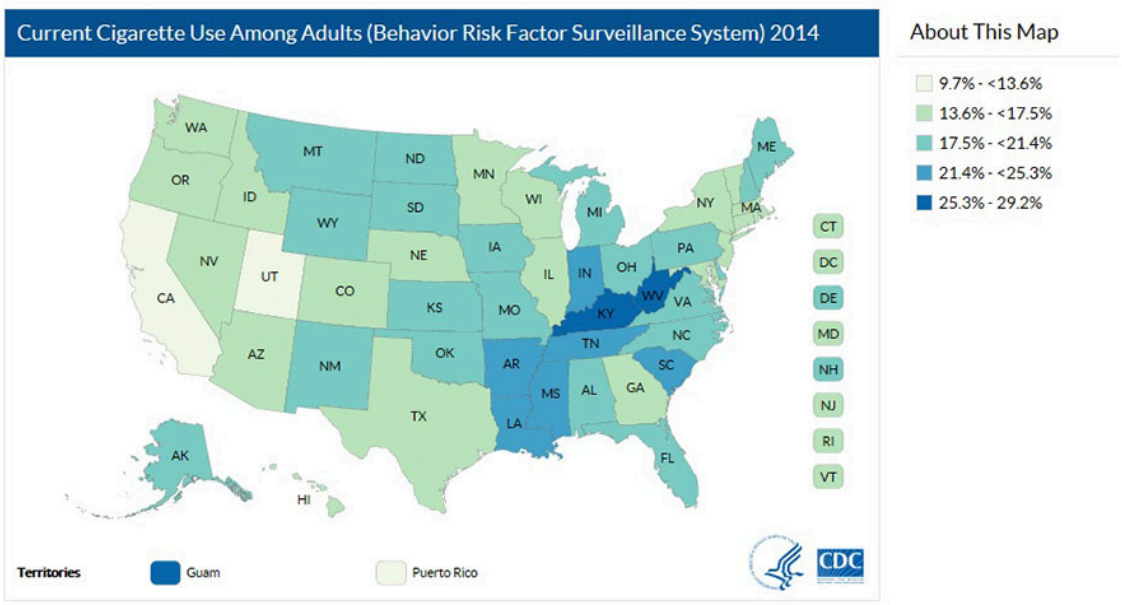
Addresses can be difficult to encode into a neural network. There are many different approaches, and you must consider how you can transform the address into something more meaningful. Map coordinates can be a good approach. latitude and longitude can be a useful encoding. Thanks to the power of the Internet, it is relatively easy to transform an address into its latitude and longitude values. The following code determines the coordinates of Washington University:

In [4]:
```
import requests

address = "1 Brookings Dr, St. Louis, MO 63130"

response = requests.get(
    'https://maps.googleapis.com/maps/api/geocode/json?key={}&address={}' \
    .format(GOOGLE_KEY,address))

resp_json_payload = response.json()

if 'error_message' in resp_json_payload:
    print(resp_json_payload['error_message'])
else:
    print(resp_json_payload['results'][0]['geometry']['location'])
```

```
{'lat': 38.6481653, 'lng': -90.3049506}
```

They might not be overly helpful if you feed latitude and longitude into the neural network as two features. These two values would allow your neural network to cluster locations on a map. Sometimes cluster locations on a map can be useful. Figure 2.SMK shows the percentage of the population that smokes in the USA by state.

Figure 2.SMK: Smokers by State



The above map shows that certain behaviors, like smoking, can be clustered by the global region.

However, often you will want to transform the coordinates into distances. It is reasonably easy to estimate the distance between any two points on Earth by using the great circle distance between any two points on a sphere:

The following code implements this formula:

$$\Delta\sigma = \arccos\left(\sin\phi_1 \cdot \sin\phi_2 + \cos\phi_1 \cdot \cos\phi_2 \cdot \cos(\Delta\lambda)\right)$$

$$d = r\,\Delta\sigma$$

In [5]:
```
from math import sin, cos, sqrt, atan2, radians

URL='https://maps.googleapis.com' + \
    '/maps/api/geocode/json?key={}&address={}'

# Distance function
def distance_lat_lng(lat1,lng1,lat2,lng2):
    # approximate radius of earth in km
    R = 6373.0

    # degrees to radians (lat/lon are in degrees)
    lat1 = radians(lat1)
    lng1 = radians(lng1)
    lat2 = radians(lat2)
    lng2 = radians(lng2)

    dlng = lng2 - lng1
    dlat = lat2 - lat1

    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlng / 2)**2
```

```
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    return R * c

# Find lat lon for address
def lookup_lat_lng(address):
    response = requests.get( \
        URL.format(GOOGLE_KEY,address))
    json = response.json()
    if len(json['results']) == 0:
        raise ValueError("Google API error on: {}".format(address))
    map = json['results'][0]['geometry']['location']
    return map['lat'],map['lng']


# Distance between two locations

import requests

address1 = "1 Brookings Dr, St. Louis, MO 63130"
address2 = "3301 College Ave, Fort Lauderdale, FL 33314"

lat1, lng1 = lookup_lat_lng(address1)
lat2, lng2 = lookup_lat_lng(address2)

print("Distance, St. Louis, MO to Ft. Lauderdale, FL: {} km".format(
        distance_lat_lng(lat1,lng1,lat2,lng2)))
```

```
Distance, St. Louis, MO to Ft. Lauderdale, FL: 1685.3019808607426 km
```

Distances can be a useful means to encode addresses. It would help if you considered what distance might be helpful for your dataset. Consider:

- Distance to a major metropolitan area
- Distance to a competitor
- Distance to a distribution center
- Distance to a retail outlet

The following code calculates the distance between 10 universities and Washington University in St. Louis:

In [6]:
```
# Encoding other universities by their distance to Washington University

schools = [
    ["Princeton University, Princeton, NJ 08544", 'Princeton'],
    ["Massachusetts Hall, Cambridge, MA 02138", 'Harvard'],
    ["5801 S Ellis Ave, Chicago, IL 60637", 'University of Chicago'],
    ["Yale, New Haven, CT 06520", 'Yale'],
    ["116th St & Broadway, New York, NY 10027", 'Columbia University'],
    ["450 Serra Mall, Stanford, CA 94305", 'Stanford'],
    ["77 Massachusetts Ave, Cambridge, MA 02139", 'MIT'],
    ["Duke University, Durham, NC 27708", 'Duke University'],
    ["University of Pennsylvania, Philadelphia, PA 19104",
         'University of Pennsylvania'],
```

```
    ["Johns Hopkins University, Baltimore, MD 21218", 'Johns Hopkins']
]

lat1, lng1 = lookup_lat_lng("1 Brookings Dr, St. Louis, MO 63130")

for address, name in schools:
    lat2,lng2 = lookup_lat_lng(address)
    dist = distance_lat_lng(lat1,lng1,lat2,lng2)
    print("School '{}', distance to wustl is: {}".format(name,dist))
```

```
School 'Princeton', distance to wustl is: 1354.4830895052746
School 'Harvard', distance to wustl is: 1670.6297027161022
School 'University of Chicago', distance to wustl is: 418.0815972177934
School 'Yale', distance to wustl is: 1508.217831712127
School 'Columbia University', distance to wustl is: 1418.2264083295695
School 'Stanford', distance to wustl is: 2780.6829398114114
School 'MIT', distance to wustl is: 1672.4444489665696
School 'Duke University', distance to wustl is: 1046.7970984423719
School 'University of Pennsylvania', distance to wustl is: 1307.19541200423
School 'Johns Hopkins', distance to wustl is: 1184.3831076555425
```

In [ ]: