Intermediate Python

Programming for Professionals

Threading

And Synchronization

POP QUIZ: COMMON TERM

What is a thread?

5 MINUTES

I AM A THREAD. WHAT AM I?

Asynchronous function call

- I have parameters
- I have a return value
- When the function exits, I am done. I own a stack frame:
  - Locals
  - Parameters
  - Return address
  - so on

I own thread specific storage: Thread Local Storage (TLS)

If I own a window, I also have a GUI queue

THREAD CONCEPTS

- Multi-Threading
- Preemptive Multi-Threading
- Parallelism
- Impact on performance
- Scheduling
- Process / Thread Priority
- Synchronization
- Maintainability
- Compute thread
- GUI Thread

MULTI-THREADING VERSUS PARALLELISM / CONCURRENCY

- Responsiveness
- UI Thread
- Task specific allocation Parallelism
- Multi-threading
- Performance
- Maximize multiple cores
- Thread Pools

MULTI-THREADING PERFORMANCE

Multiple threads may improve performance but that is not guaranteed. It depends on the application, hardware, and other factors.

Oversubscription and undersubscription can adversely affect performance. Oversubscription is when there are more threads than processor cores. Threads will compete for processor execution and ready threads can become suspended. Undersubscription is when there are much fewer threads than processor cores. This results in minimal parallelism and underutilized processor cores.

Constant starting and stopping of short running threads can also affect performance.

THREAD SCHEDULING

Thread scheduling is a combination of process priority, thread priority, and round-robin preemptive scheduling.

Preemptive scheduling means each thread receives one or more quantums of execution; but can be preempted if a high priority thread starts.

A thread is preempted after completing the quantum(s). The schedule then looks for the next thread to schedule – either a higher priority thread or round-robin fashion.

Threads running over their base priority are eroded 1 priority each time slice. However, thread priority will not erode beneath the thread's base priority. Threads may also receive a priority boost for a variety of reasons.

MAINTAINABILITY

- Creating a thread is easy
- Writing a responsible thread is much harder
  - Normal priority threads
  - No global variables
  - Limited if any synchronization
  - Using thread pool when possible
  - No dependences, such as file access.

THREADING MODULE

The Thread class is found in the threading module. There are other thread constructs in this module:

- Configuration
- Get the current thread
- Enumerate threads
- Synchronization objects
- And more

CREATING A THREAD

Creating a thread is simple:

1. Import threading module
2. Define a method to be used as a thread
3. Initialize a Thread object with the thread method
4. Execute the thread with the start method
5. The thread will execute until the thread method exits

```
import threading
import time
def thethread():
    print('Worker\n')
    return

for i in range(5):
    t = threading.Thread( \
    target=thethread)
    t.start()

time.sleep(10)
```

LOGGING WITHIN A THREAD

Log instead of print to trace or audit a thread. Logging inserts the thread identity into the message.

For logging, import the logging module. Use the debug method to log:

```
import threading
import time
import logging


logging.basicConfig(level=logging.DEBUG, \
    format="[%(levelname)s] (%(threadName)-10s) %(message)s")


def thethread(arg):
    print('Worker %d\n'%arg)
    logging.debug('Worker %d\n'%arg)
    return


threads = []
for i in range(5):
    t = threading.Thread( \
        target=thethread,args=(i,))
    t.start()


time.sleep(5)
```

PLAN B: CREATE THREAD

You can also create a thread through inheritance.

- Create a class that inherits from threading.Thread
- Override the *run* method
- Implement the thread as the run method
- Create an instance of the class
- Call *thread*.start to execute the thread

This approach is particularly useful for maintaining state information for the thread as a member of the class.

```
import threading
import datetime
import time


class ThreadClass(threading.Thread):
        def run(self):
                date=datetime.datetime.now()
                print("%s, Hello World! at: %s\n" \
                        %(self.getName(), date))


for i in range(2):
        t=ThreadClass()
        t.daemon=True
        t.start()
time.sleep(5)
print("")
```

WAIT FOR A THREAD

The threading.join() method allows one thread to block until another thread completes. This avoids other less efficient methods for synchronization especially polling.

```
import threading

import datetime


class ThreadClass(threading.Thread):

  def run(self):

    date=datetime.datetime.now()

    print("%s, Hello World! at: %s\n" \

        %(self.getName(), date))


t=ThreadClass()

t.daemon=True

t.start()


t.join()
```

DAEMON THREADS

You can have one or more threads in an application. The threads are either daemon or non-daemon. Daemon threads are background threads and not sufficient to keep the application alive. Daemon threads are also called worker threads.

Threads are non-daemon by default. All non-daemon threads must finish before the application will exit naturally.

The attribute threadObject.daemon = True sets a thread status to daemon thread.

SYNCHRONIZATIONS

LOCK

Locks are the simplest of the synchronization objects. Protect resources that come and quantity of 1 or blocks of code that are not thread safe. You can place gates around code or resources. Only one thread is allowed between the gates at a time. All other threads will block until the current thread exits the gate.

Here are the common methods of the lock object:

- acquire. enter gate. If the lock is owned, the thread blocks.
- release. leave gate. If threads waiting, one thread enters. If multiple threads are waiting / blocked, the order of release may be undetermined.

LOCK – EXAMPLE CODE

```
import threading

import time



thelock=threading.Lock()

def thethread():

        name=threading.current_thread().name

        while(True):

                #protect resource

                thelock.acquire()

                print("%s enters gate"%name)

                time.sleep(5)

                print("%s leaves gates"%name)

                thelock.release()

                time.sleep(5)

        count=5

        while(count):

                t = threading.Thread(target=thethread)

                t.setDaemon(True)

                t.start()

                count-=1
```

SEMAPHORE

Semaphores synchronize access to limited resources. Imagine threads as taxi drivers competing for available cabs. There are four drivers and two cabs. There are not enough cars for every driver; there will always be a driver waiting. When a driver acquires a cab, they must keep the car for an entire 8 hour shift. They then release the cab for any available driver.

Here are the common semaphore methods:

- Set the resource count in the semaphore constructor. The default is 1.
- acquire. acquires any number of resources. If exceeds availability, the thread blocks.
- release. relinquish a number of resources. If presently over the threshold, may release blocked threads.

SEMAPHORE – EXAMPLE CODE

```
import threading

import time

sema=threading.Semaphore(3)

def thethread():

    name=threading.current_thread().name

    while(True):

        print("%s Requesting car\n"%name)

        sema.acquire(True)

        print("%s Shift started\n"%name)

        # 8 hour shift

        time.sleep(8)

        sema.release()

        print("%s Off work\n"%name)

        time.sleep(16)

count=5

while(count):

    t = threading.Thread(target=thethread)

    t.setDaemon(True)

    t.start()

    count-=1
```

EVENT

Events are custom synchronization objects. Developers decide when an event is signaled and non-signaled. There is no predefined behavior for events. It is truly at the discretion of the developer.

Here are the common event methods:

- is_set. returns true if the event is signaled
- set. sets the event to signaled
- clear. sets an event to non-signaled
- wait. Block if event is non-signaled

EVENT – EXAMPLE CODE

```
import threading

import time

theevent=threading.Event()

def thethread():

    while(True):

        print("Thread running\n")

        if(not theevent.isSet()):

            print("Thread about to block\n")

        theevent.wait()

        time.sleep(2)

t = threading.Thread(target=thethread)

t.start()

time.sleep(5)

print("Set Event")

theevent.set()

time.sleep(10)

print("Reset Event")

theevent.clear()

time.sleep(5)
```

PYTHON GLOBAL INTERPRETER LOCK (GIL)

THE BAD NEWS

Hard to describe the GIL as anything but the *bad news*. Simply stated, within a process execution is limited to one thread at a time – even in a multi-threading and multiple core architecture. When running a thread, there is a mutex / lock for the Python interpreter. The lock serializes access to the Python interpreter.

GIL has been long debated and caused considerable controversy. *However, it made sense at the time.*

PROBLEM SOLVED

GIL solved several problems.

- Reference counting in a multi-threaded environment
  - Deadlocks
  - Memory leaks
- Enforced a thread safe environment – especially important for C libraries.
- Single threaded programs run more efficiently

SINGLE-THREADED COUNTDOWN

```
3.7095768451690674
```


```
import time

from threading import Thread

COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1

start = time.time()

countdown(COUNT)

end = time.time()

print('Time taken in seconds -', end - start)
```

Example code from: https://realpython.com/python-gil/

MULTI-THREADED COUNTDOWN


```
import time

from threading import Thread


COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))

t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print('Time taken in seconds -', end - start)
```

```
3.710641860961914
```

GIL – SOLUTION 1

Multi-processing versus multi-thread is one solution for the GIL problem. However, it does make your code *more complex* than otherwise necessary.

Note: the reference is to multi-processing; not multiple processors.

Each Python process receives a separate Python Interpreter. You still have a lock but a different lock from everyone else.

The tools for multiprocessing are found in the Multiprocessing module using pools.

MULTI-PROCESSING COUNTDOWN


```
from multiprocessing import Pool

import time

COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1



if __name__ == '__main__':

    pool = Pool(processes=2)

    start = time.time()

    r1 = pool.apply_async(countdown, [COUNT//2])

    r2 = pool.apply_async(countdown, [COUNT//2])

    pool.close()

    pool.join()

    end = time.time()

    print('Time taken in seconds -', end - start)
```

```
3.4133529663085938
```

GIL – SOLUTION 2

The most popular implementation of python is CPython. Solution 2 is stop using CPython. Other Python implementations may not implement the GIL lock:

- IronPython
- Jython
- PyPy
- And so on

PLATFORM

```
>>>
>>> import platform
>>> platform.python_implementation()
'CPython'
>>>
>>>
```

How do you know the flavor of the current implementation of Python?

Threading

Lab 10

LAB: RESTAURANT

This lab:

Create a restaurant simulation with customers and cashiers.

RESTAURANT

1. Import threading
2. Import time
3. There are five cashiers waiting to provide service. Create a semaphore with a usage count of 5 – one for each cashier
4. You have ten customers each a thread. Define a common thread, called customer, for each person. In the thread:
    1. Print “waiting for cashier”. Include the thread name.
    2. Acquire the semaphore
    3. Sleep for 5 seconds
    4. Print “leaving restaurant”. Include the thread name.
    5. Release semaphore
5. This program should have a properly constructed main (entry point method)
6. Start the 10 threads (customers)

MACHINE LEARNING

FOUNDATIONS

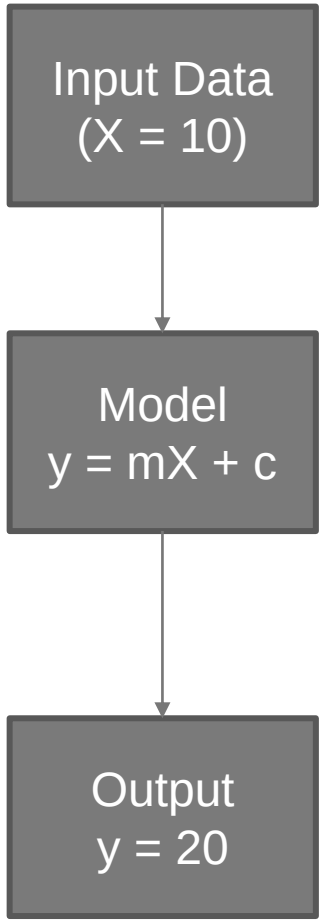
POP QUIZ: COMMON TERM

What is a machine learning?

5 MINUTES

WHAT IS A MACHINE LEARNING MODEL

Computer algorithms that use data to learn and make estimations.



CHOOSING A MODEL

The code for a model is often very simple.

In fact most efficient models are simple. The complexity of a model will depend on the goal you are trying to achieve.

For example, a linear model to estimate today's temperature based on historical temperature values.

The model's efficacy relies on having the correct values for its internal parameters.

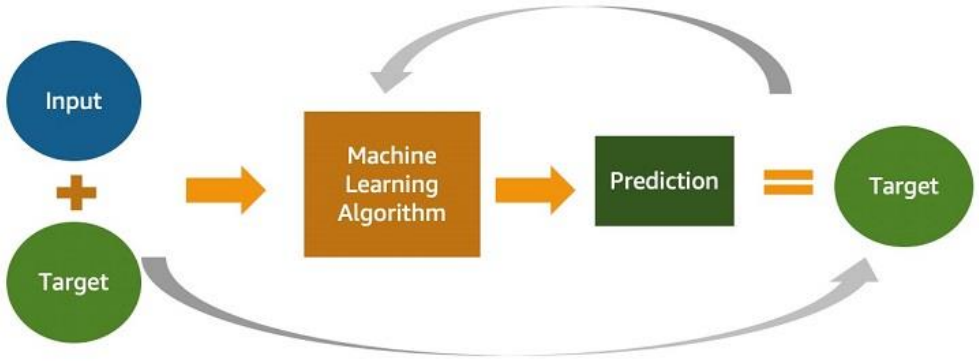
```
def estimate_temperature(today_date):
        estimate = m*today_date + c
        return estimate
```

MODEL TRAINING

The model's efficacy relies on having the correct values for its internal parameters.

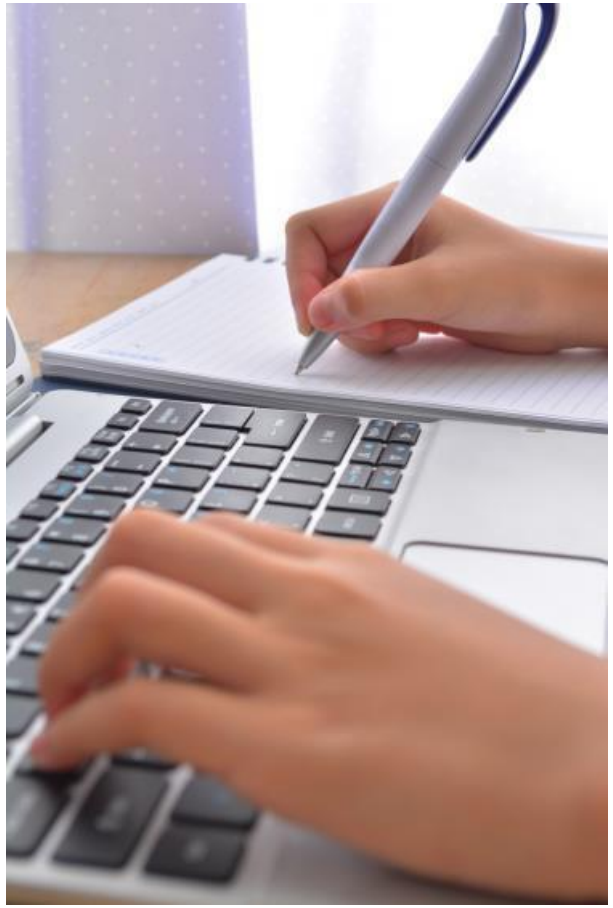
Discovering the right parameters might requires a training process.

During the training process, the model's estimate is compared with known values and its internal parameters are adjusted until there is no improvement in the model prediction error.



HANDS-ON EXERCISE

Create a machine learning model



PREPARING DATA – HANDS ON

The first step in creating a model, is to load and prepare data. There are various techniques to achieve this, but for this hands on we will keep things simple.

The model you are going to create predicts sales from a TV marketing budget dataset. The dataset consist only of two columns, TV and Sales.

You will use the *pandas* library to explore and generate some visualizations.

Open the Hands_On_Simple_Linear_Regression.ipynb notebook and follow the instructions.

EXPLORATORY DATA ANALYSIS

For a model to be accurate, the correct features must be selected to train the parameters used for label estimation.

Exploratory data analysis helps in understanding the training dataset and serve as a starting point for feature engineering.

There are many ways for EDA depending on the tools you are using.

In the hands-on exercise, you use pandas DataFrames.

```
import pandas as pd

# Reading csv file
file_path  = "tvmarketing.csv"
advertising = pd.read_csv(file_path)

# Display the first 5 rows
advertising.head()

# Display the last 5 rows
advertising.tail()

# Let's look at some statistical
information about the dataframe.
advertising.describe()
# Visualise the relationship between the
# features and the response using scatterplots
advertising.plot(x='TV',y='Sales',kind='scatter')
```

SELECTING A MODEL

The model you choose to train mainly depends on the problem you are trying to solve and the nature of your data.

In the hands-on lab, you perform a Simple Linear Regression to train a linear model which is represented mathematically by the equation below:

$$y = m.TV + c$$

Where m = slope, c = y-intercept, TV is a random variable or feature in our dataset, y is the label (Sales), the value we are predicting

Several python machine libraries exist that can be used to train and build models.

For this exercise, you use the *statsmodels* module.

```
# Import the statsmodels library
import statsmodels.formula.api as smf

# Define a formula - Sales is explained by TV
formula = 'Sales ~ TV'

# Create an untrained model using OLS Regression
model = smf.ols(formula,advertising)

# Check that the internal parameters have not yet
been trained
if not hasattr(model,'params'):
    print('Model selected but not yet trained')
```

FITTING A MODEL

Given a linear regression model:

$$y = m.TV + c$$

The training process involves selecting the right parameter values for m and c that can be used to estimate the label.

The statsmodels Ordinary Least Squares returns a new linear model without trained internal parameters.

Calling the model.fit() method, will train the parameters as shown in the example code.

```
# Train (fit) the sales model
fitted_model = model.fit()

# Print the internal parameters of the trained
model
print(f"Slope: {fitted_model.params[1]}")
print(f"Intercept: {fitted_model.params[0]}")
```

MAKING A PREDICTION

Once you have trained the model, you can use the model.predict() method to estimate labels given an input dataset consisting of the features used to train the model.

```
import random
# Making predictions on some sample values
sample =
advertising.iloc[random.randint(0,advertising.TV.
count() - 1)]
tv_value = {'TV': [sample.TV]}
approximate_sales =
fitted_model.predict(tv_value)[0]
```

SAVE MODEL TO A FILE

To reuse your trained model in a scoring/prediction script, you must save it as a file.

The joblib module provides a method to save the model as a file object.

```
# Save a model to a file
import joblib

model_filename = './sales_model.pkl'
joblib.dump(fitted_model,model_filename)
```

✓ 0.3s

```
['./sales_model.pkl']
```

LOADING A MODEL

Once you have a saved model, loading it from disk using *joblib* is easy.

```
# Load a model from a file
import joblib

model_filename = './sales_model.pkl'
loaded_model = joblib.load(model_filename)
print(loaded_model.params)
```

✓ 0.9s

```
Intercept    7.032594
TV           0.047537
dtype: float64
```

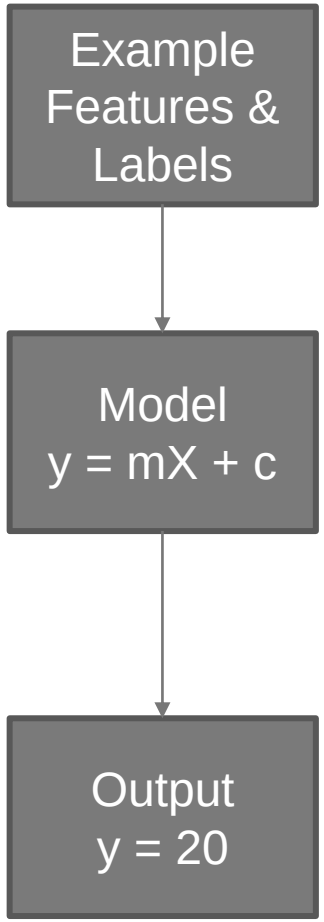
POP QUIZ:

What is the purpose of performing training?

5 MINUTES

SUPERVISED MACHINE LEARNING

Supervised learning is the process in which machine learning models learn from examples.



SUPERVISED VS UNSUPERVISED MACHINE LEARNING

Unsupervised learning

Train the model without prior-knowledge of the answer

Example: Model to draw realistic images such as DALL-E neural network.

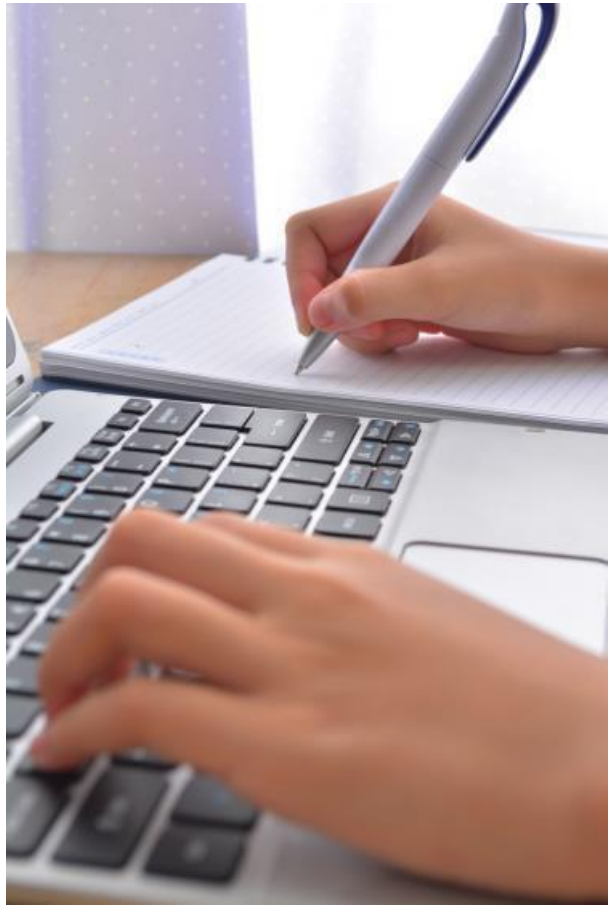
Supervised learning

Assess model performance by comparing estimates to known labels

Example: Model to predict daily temperatures based on historical weather data.

HANDS-ON EXERCISE

Supervised learning



SUPERVISED MACHINE LEARNING - SUMMARY

Supervised learning requires that you train a model to operate on a set of features and predict a label using a dataset with known label values.

In regression, the model is represented by a linear function y = f(x) where x is a vector that consists of multiple features and y is the label being predicted.

The goal of training is to have a function that predicts y accurately.

A machine learning model such as OLS is used to fit the x-values to f(x) that predicts y within reasonable accuracy.

General regression function

$$Y = f([x1,x2,x3,\ldots\ldots])$$

SUPERVISED MACHINE LEARNING ALGORITHMS

REGRESSION ALGORITHMS:

Predict a continuous variable Y, such as predicting the number of sales transactions for a TV as in the previous hands on.

Example is the Ordinary Least Squares algorithm.

We are not going to get into the details of how the algorithms work in this course.

CLASSIFICATION ALGORITHMS:

Predict to which category or class an observation belongs to.

The Y value is a probability between 0 and 1 for each of the individual classes indicating the probability that an observation belongs to that class.

Example is an Image Classification algorithm.

START WITH UNDERSTANDING THE DATASET

The first step is to explore the data used for training in order to understand the relationship between the features (X) and the label being predicted (Y).

During this stage you might carry out the following tasks:

- Data cleaning
- Handling missing values
- Feature engineering
- Normalization
- Data encoding

```
# Load the bike data into a pandas DataFrame
import pandas as pd
bike_share_data = pd.read_csv('daily-bike-share.csv')
bike_share_data.head()
```

✓ 0.6s

	instant	dteday	season	yr	mnth	holiday	weekday
0	1	1/1/2011	1	0	1	0	6
1	2	1/2/2011	1	0	1	0	0
2	3	1/3/2011	1	0	1	0	1
3	4	1/4/2011	1	0	1	0	2
4	5	1/5/2011	1	0	1	0	3

IDENTIFYING FEATURES AND LABELS

Using the pandas df.dtypes attribute, you can view all the columns in a dataset as well as their data types.

Identifying features and labels is not a straight forward task. This might involve several experiments and meetings with SMEs to select the right predictors to approximate the label.

In this example, we want to predict the number of bike rentals per day, so *rentals* is the label (Y), and all the other columns are candidate features (X)

```
bike_share_data.dtypes
✓ 0.4s
instant          int64
dteday          object
season           int64
yr               int64
mnth             int64
holiday          int64
weekday          int64
workingday       int64
weathersit       int64
temp           float64
atemp          float64
hum            float64
windspeed      float64
rentals          int64
dtype: object
```

FEATURE ENGINEERING

In simple terms, feature engineering is the process of deriving new features by transforming or combining existing features. Feature engineering involves a number of tasks which we cannot describe in a single slide.

In this example, we create a new ‘day’ column from the existing ‘dteday’ column. The new column represents the day of the month from 1 to 31.

```
bike_share_data['day'] = pd.DatetimeIndex(bike_share_data['dteday']).day
bike_share_data.head(3)
```

✓ 0.1s Python

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	rentals	day
0	1	1/1/2011	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331	1
1	2	1/2/2011	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131	2
2	3	1/3/2011	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120	3

DESCRIPTIVE STATISTICS

Descriptive statistics forms a basis for exploratory data analysis.

Using measures of central tendency and measures of dispersion, you can get insight into the distribution of numeric values in your dataset.

Common statistics to look for are:

- mean, standard deviation, min & max, quantiles.

Pandas DataFrame provide a method called *describe* which makes this easy.

```
features_label = ['temp','atemp','hum','windspeed','rentals']
bike_share_data[features_label].describe()
```

✓ 0.9s

	temp	atemp	hum	windspeed	rentals
count	731.000000	731.000000	731.000000	731.000000	731.000000
mean	0.495385	0.474354	0.627894	0.190486	848.176471
std	0.183051	0.162961	0.142429	0.077498	686.622488
min	0.059130	0.079070	0.000000	0.022392	2.000000
25%	0.337083	0.337842	0.520000	0.134950	315.500000
50%	0.498333	0.486733	0.626667	0.180975	713.000000
75%	0.655417	0.608602	0.730209	0.233214	1096.000000
max	0.861667	0.840896	0.972500	0.507463	3410.000000

POP QUIZ:

What insights can you draw from the descriptive statistics below:

```
features_label = ['temp','atemp','hum','windspeed','rentals']
bike_share_data[features_label].describe()
```

✓ 0.9s

	temp	atemp	hum	windspeed	rentals
count	731.000000	731.000000	731.000000	731.000000	731.000000
mean	0.495385	0.474354	0.627894	0.190486	848.176471
std	0.183051	0.162961	0.142429	0.077498	686.622488
min	0.059130	0.079070	0.000000	0.022392	2.000000
25%	0.337083	0.337842	0.520000	0.134950	315.500000
50%	0.498333	0.486733	0.626667	0.180975	713.000000
75%	0.655417	0.608602	0.730209	0.233214	1096.000000
max	0.861667	0.840896	0.972500	0.507463	3410.000000

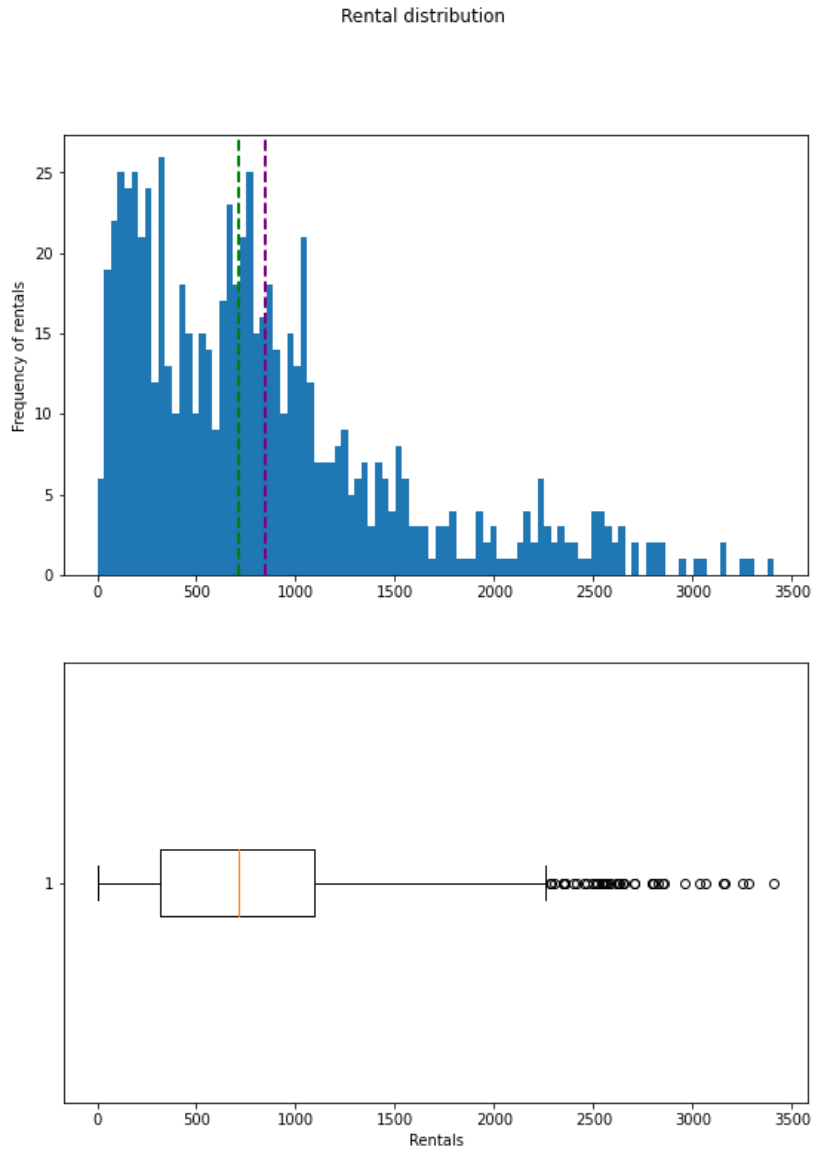
5 MINUTES

VISUALIZING WITH MATPLOTLIB

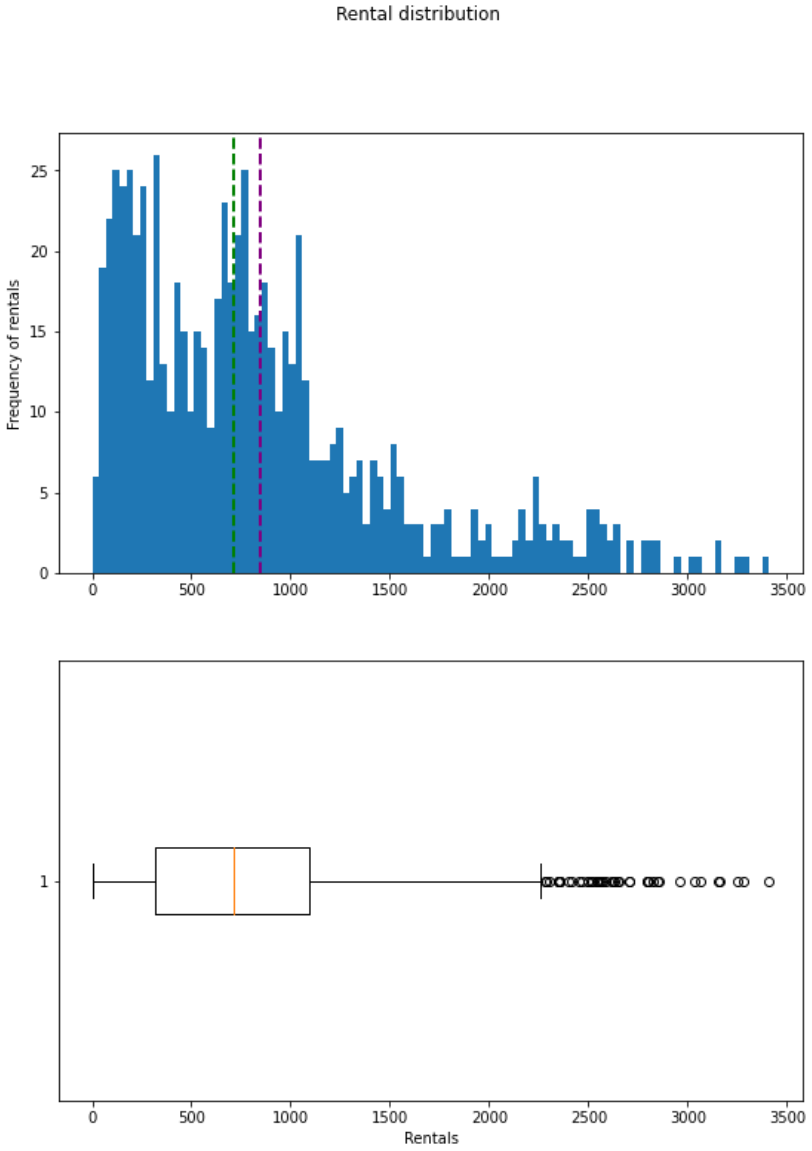
Visuals can provide more clarity into the nature of a dataset than a textual or tabular representation.

Matplotlib is a popular Python library that can be used to plot graphs.

In this example, we plot a line graph and histogram to visualize the distribution of bike rentals.



POP QUIZ:



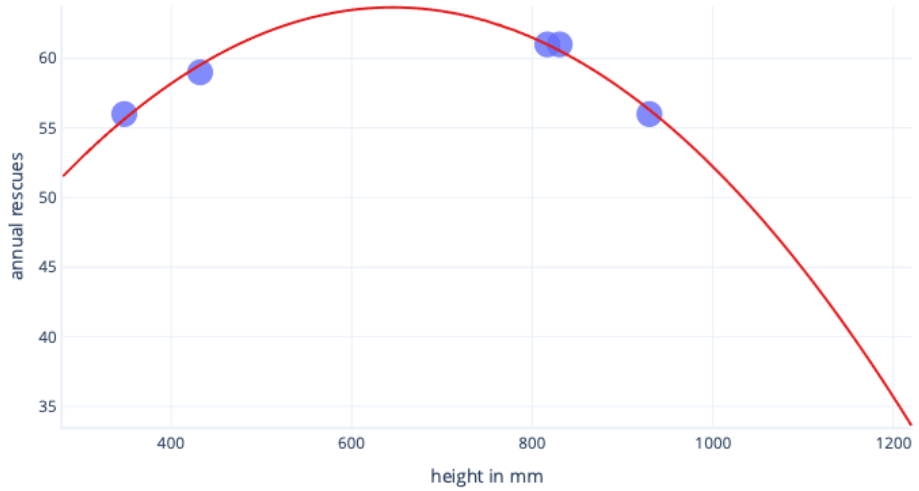
What can you say about the distribution of rentals represented by the histogram and boxplot?

5 MINUTES

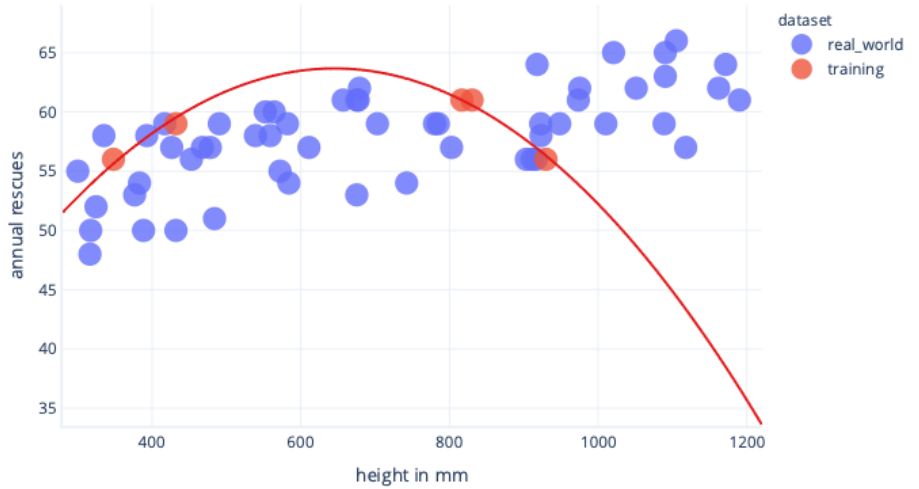
MODEL OVERFITTING

A model is said to be overfit if it performs well with the training data than it does with unknown data.

This shows that the model has memorized details of the training dataset instead of finding a set of parameters that generalize the model's internal function.



Model performance with training data



Model performance with real data

AVOIDING OVERFITTING – TRAIN/TEST SPLIT

The simplest approach to avoiding overfitting is using a simple dataset that is a better representation of the real-world observations.

Using a test dataset, you can stop training the model after it has learned the general parameter values and before it has been overfit.

A test dataset is created by splitting a large dataset into two portions. One for training and the other one for testing (validation)

```
from sklearn.model_selection import \
    train_test_split as tts

#Split the dataset into
# 80% for the training and 20% for validation
X_train, X_test, y_train, y_test = \
tts(X,y, test_size = 0.20, random_state = 0)
```

✓ 0.2s

The scikit-learn library provides a train_test_split function we can use to randomly split the data.

TRAIN THE MODEL

The next step is to run an experiment to train a model.

In this example we have a single run to train the model, but in real-world scenarios a model is trained using several iterations until a fit is found after model performance evaluation.

```
from sklearn.linear_model import LinearRegression

# Fit the linear regression model on the training dataset
model = LinearRegression().fit(X_train, y_train)
print(model)
```

✓ 0.3s

```
LinearRegression()
```

We use the Scikit-Learn LinearRegression

estimator to train the model.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

VALIDATE THE MODEL

The scikit-learn LinearRegression model exposes a predict() function which accepts a set of features as input.

Use the model to predict rentals for the X_test dataset and compare the results with the y_test values.

The comparison will help check if the model is performing well.

```
# Evaluate the model with test data
import numpy as np

predicted_rentals = model.predict(X_test)
print('Predicted rentals: ', np.round(predicted_rentals)[:5])
print('Actual rentals: ', np.round(y_test)[:5])
```

✓ 0.3s

```
Predicted rentals:  [1925. 1196. 1024.  -73.  300.]
Actual rentals:  [2418  754  222   47  244]
```

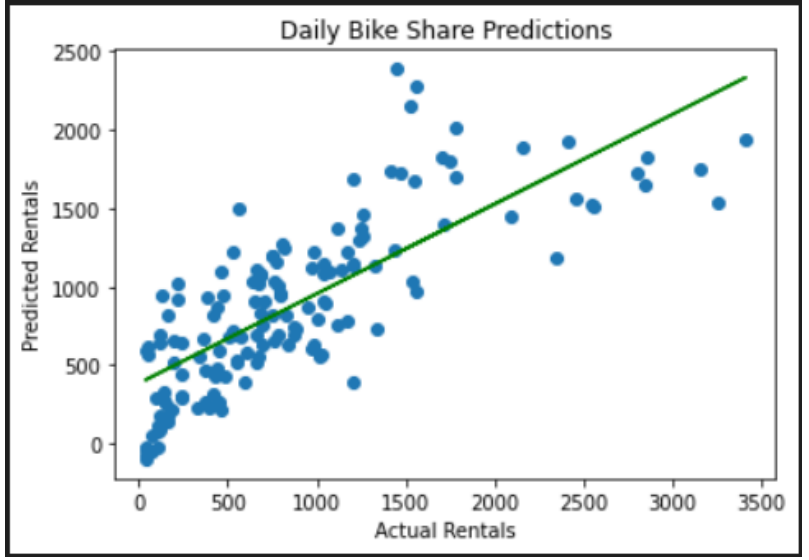
VALIDATE THE MODEL – USING A TRENDLINE

A line plot can provide a clear and visual comparison between the predicted labels and actual labels.

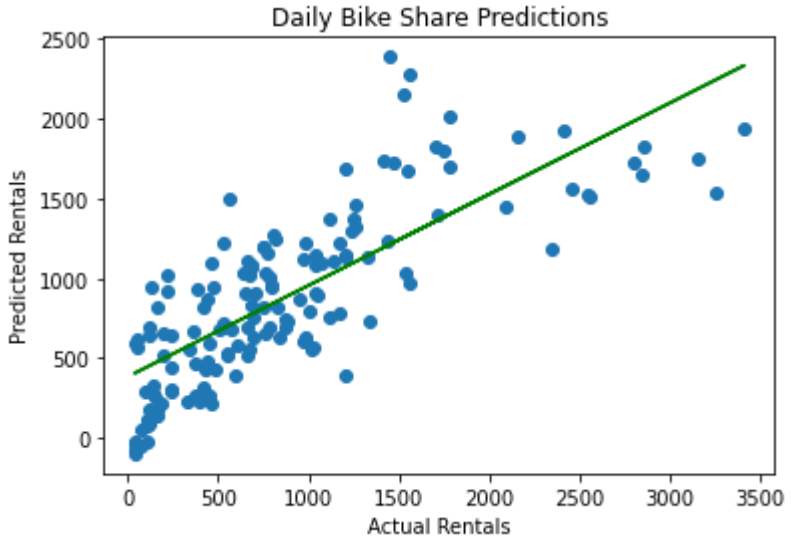
This can help to quickly determine if a model is performing well.

```
# Plot a trendline to compare the output
import matplotlib.pyplot as plt

%matplotlib inline

plt.scatter(y_test, predicted_rentals)
plt.xlabel('Actual Rentals')
plt.ylabel('Predicted Rentals')
plt.title('Daily Bike Share Predictions')

z = np.polyfit(y_test, predicted_rentals, 1)
p = np.poly1d(z)
plt.plot(y_test,p(y_test), color='green')
plt.show()
```

✓ 0.1s



POP QUIZ:



What can you say about the trendline above?

5 MINUTES

QUANTIFYING ERRORS – MODEL METRICS

You can notice that there is some difference between the line representing the ideal model function and the results.

The difference represents the variance between the predicted labels and actual values of the label in the validation dataset.

The residuals can be quantified using evaluation metrics:

- Mean Squared Error
- Root Mean Squared Error
- Coefficient of Determination (R-squared)

To learn more click the link below:

https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics

```
# Quantify errors using evaluation metrics
from sklearn.metrics import mean_squared_error, r2_score

mse = mean_squared_error(y_test, predicted_rentals)
print("MSE:", mse)

rmse = np.sqrt(mse)
print("RMSE:", rmse)

r2 = r2_score(y_test, predicted_rentals)
print("R2:", r2)
```

✓ 0.4s

```
MSE: 210673.0967793622
RMSE: 458.99139074645205
R2: 0.6013016737003889
```

MINIMIZE MODEL ERRORS – COST FUNCTIONS

Machine learning uses a cost function to determine how well a model is performing.

The number of mistakes that a model makes when predicting a label is known as an error, cost or loss.

The goal of a cost function is to minimize the cost, ideally have zero cost.

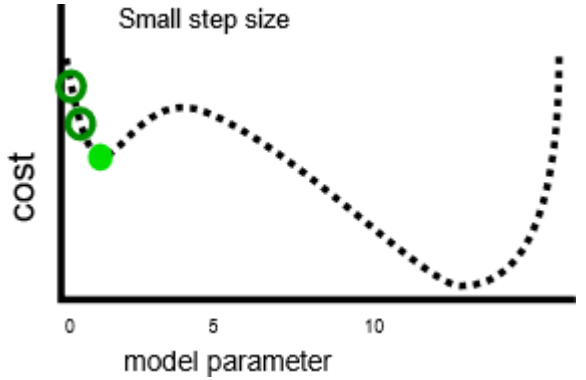
The cost function can change how long training takes and how well a model works

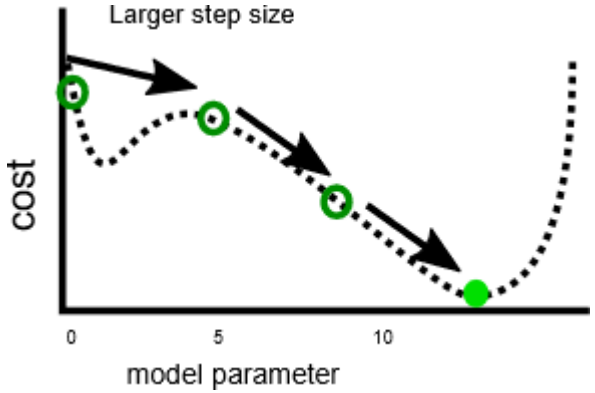
```python
import numpy

# SSD squares that difference and sums the result.
def sum_of_square_differences(estimate, actual):
    return numpy.sum((estimate - actual)**2)

# SAD converts differences into absolute
#  and then sums them.
def sum_of_absolute_differences(estimate, actual):
    return numpy.sum(numpy.abs(estimate - actual))
```

MINIMIZE MODEL ERRORS – GRADIENT DESCENT

Gradient descent is an optimization algorithm which uses calculus to estimate the rate of change of a model's internal parameters during training.

It calculates the slope of the relationship between each model parameter and the cost, then alters the parameters to move down the slope (gradient), hence the name gradient descent.





MODEL IMPROVEMENT - HYPERPARAMETERS

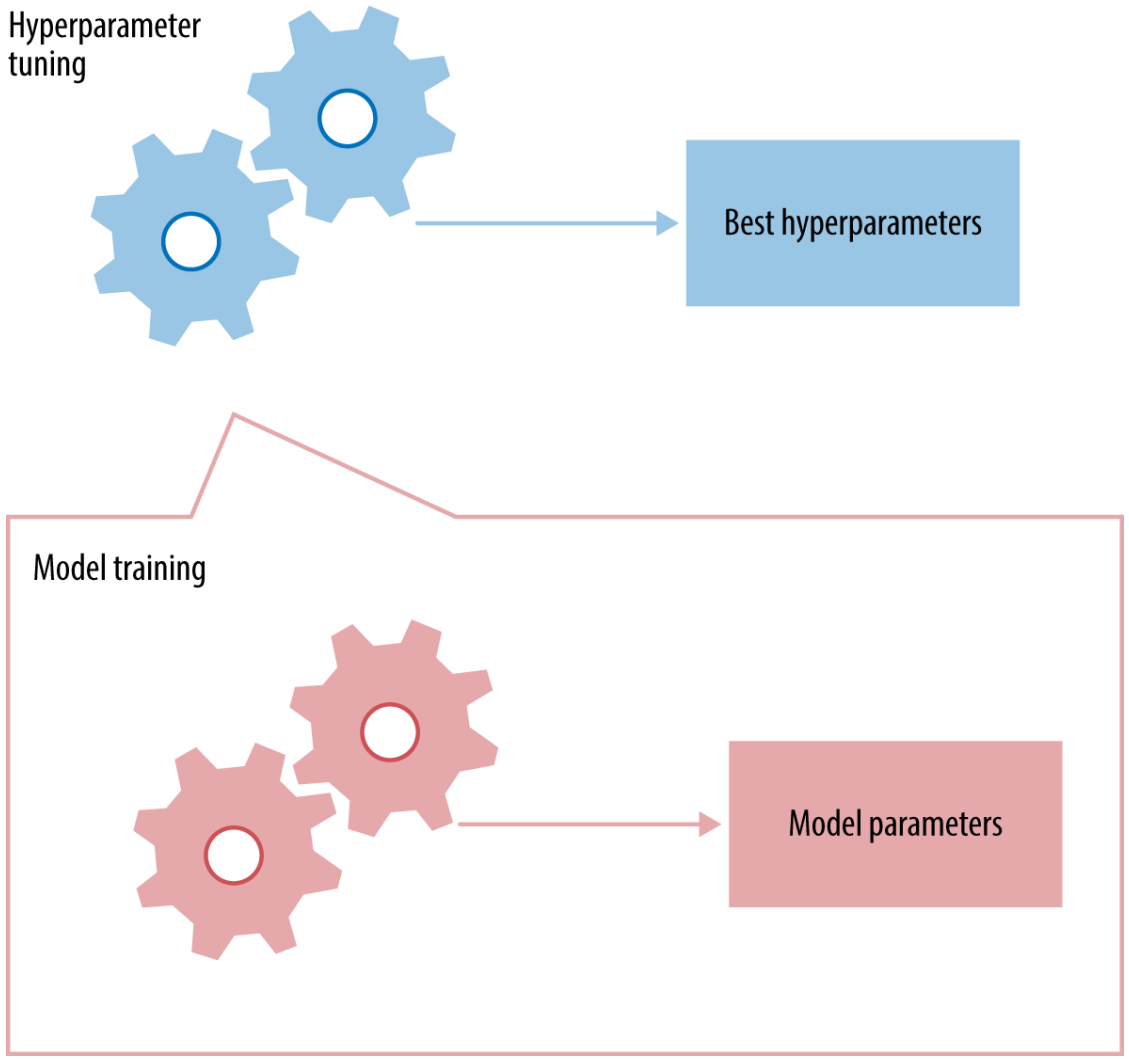
During each training iteration, the internal parameters of a model are adjusted towards a set of general values that represent an ideal prediction function.

The rate at which these parameters are modified is known as the learning rate.

The learning rate is also a hyperparameter because is can change the way the model is fit during each training iteration.

A high learning rate means the model can be trained faster.

The idea is to take an approach to find optimal parameter values during training.



Machine Learning

Lab 11

LAB: DIABETES PREDICTION

In this lab, you will train a Linear Regression model to predict diabetes given the sklearn diabetes dataset.

TRAINING NOTEBOOK

- Create a new training notebook named diabetes_prediction.ipynb
- Add the following import statements

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

These will allow you to use matplotlib, sklearn and numpy for exploratory data analysis, visualizations and model training.

GET THE DATA

Retrieve the dataset from sklearn.dataset using the code below:

```
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)
```

EXPLORATORY DATA ANALYSIS

- Use pandas to explore the dataset
- Using matplotlib, plot a histogram and boxplot for each feature and explain the distribution of features
- Plot a histogram to determine the distribution of the label
- What is the relationship between each feature and the label?

TRAIN TEST SPLIT

Split the dataset as follows:

- 80% for training
- 20% for testing

You can use numpy or sklearn train_test_split function to do this.

FIT THE MODEL

Using the diabetes_X_train and diabetes_y_train dataset, fit the model using a LinearRegression estimator.

MAKE PREDICTIONS

Use the diabetes_X_test to make predictions with the model you have just created.

VALIDATE MODEL METRICS

In a new notebook cell, view and evaluate model metrics as follows:

- Print model.coef_
- Call mean_squared_error(diabetes_y_test, diabetes_y_pred) and print the result
- Call r2_score(diabetes_y_test, diabetes_y_pred) and print the result

PLOT MODEL OUTPUT

In a new notebook cell, use matplotlib to plot two graphs:

- A scatter plot with diabetes_X_test and diabetes_y_test along the x and y axis respectively.
- A trendline with diabetes_X_test and diabetes_y_pred along the x and y axis respectively.
- What can you say about the trendline and test label dataset?

SAVE THE MODEL TO A FILE

In a new notebook cell, use joblib to save the model to disk

DATABASE

CRUD

RELATIONAL DATABASE

Python provides easy manipulation of relational databases and tables.

- SQL development
- Multi user access
- Schemas
- Constraints
- Object Relational Mapping

SQL LITE

- Open source database
- Lightweight database – not as functional as Oracle DB or MYSQL
- Niche. Lightweight ideal for mobile environments
- Portable
- Easy to use: local database – appears as a file in the file directory

CRUD

```
import sqlite3

connect=sqlite3.connect("enterprise1.db")
curs=connect.cursor()
curs.execute("""CREATE TABLE zoo (critter VARCHAR(20) PRIMARY KEY,
  count INT, damages FLOAT)""")
curs.close()

connect.close()
```

CRUD

```
import sqlite3

connect=sqlite3.connect("enterprise1.db")

curs=connect.cursor()

curs.execute('INSERT INTO zoo VALUES("Duck", 5, 0.0)')

curs.execute('INSERT INTO zoo VALUES("Bear", 15, 0.0)’)

con.commit()

curs.close()

connect.close()
```

CRUD

```
import sqlite3

connect=sqlite3.connect("enterprise1.db")

curs=connect.cursor()


for row in curs.execute("SELECT * FROM zoo"):

    print(row)


curs.close()

connect.close()
```

CRUD

```
conn = sqlite3.connect('databaza.db’)

c = conn.cursor()

data3 = str(input('Please enter name: '))

mydata = c.execute('DELETE FROM Zoo WHERE Name=?', (data3))

conn.commit()

c.close
```

OBJECT RELATIONAL MAPPING (ORM)

ORM abstracts the details of database manipulation, such as writing SQL commands. In addition, ORM is an object oriented approach to creating, adding records, querying, and updating a database. With ORM, you create a class that defines the database schema where fields are attributes of the class. Adding records, deleting records, or other operations are methods on database objects. Abstraction makes the database code more maintainable. However, you may lose access to some database specific features.

PYTHON ORM

The sqlalchemy module hosts the ORM capabilities. To install ORM:

```
pip install flask-sqlalchemy
```

Important methods:

- create_engine: creates the Sqlalchemy and initializes a connection string
- declarative_base: returns a class and the base class for any ORM mapped class
- sessionmaker: generates new session objects. Optionally, you can bind an engine or connection to the session object.

ORM – EXAMPLE - 1

Adjacent is typical code to create a database with ORM. This code creates a local database called movie.db. It will be created in the current directory.

The Movie class defines the database schema with three fields: role, name, and age.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

conn=sa.create_engine("sqlite:///movie.db")

theBase=declarative_base()

class Movie(theBase):
    __tablename__="Actors"
    role=sa.Column("role", sa.String, primary_key=True)
    name=sa.Column("name", sa.String)
    age=sa.Column("age", sa.Integer)

    def __init__(self, role, name, age):
        self.role=role
        self.name=name
        self.age=age

    def __repr__ (self):
        return "<Actors({}, {})>" \
               .format(self.name, self.age)
```

ORM – EXAMPLE - 2

Here is the remainder of the code to create a database.

The session constructor creates the tangible database.

The commit method persists any database changes.

```
theBase.metadata.create_all(conn)

first=Movie("Bob", "Robert Culp", 79)
second=Movie("Ted", "Elliot Gould", 79)
third=Movie("Carol", "Natalie Wood", 43)
fourth=Movie("Alice", "Dyan Cannon", 81)

print(first)

session=sessionmaker(bind=conn)
theSession=session()

theSession.add(first)
theSession.add_all([second, third, fourth])
theSession.commit()
```

ORM – EXAMPLE - 3

The database client code is straightforward. You must import the database code.

This code reads all of the records and displays the first.

```
import sqlalchemy as sa
from db4 import theBase, Movie
from sqlalchemy.orm import sessionmaker

conn=sa.create_engine("sqlite:///movie.db")
theBase.metadata.bind = conn

session=sessionmaker()
session.bind=conn
theBase=session()

theBase.query(Movie).all()


actor=theBase.query(Movie).first()
```

ORM – DATATYPES

SQLAlchemy supports the base data types Boolean, date, time, strings and numeric values as well as vendor-specific types like JSON and custom developer types.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import
declarative_base


conn = sa.create_engine("sqlite:///movie.db")


theBase = declarative_base()


def get_connection():
    return theBase,conn


class Product(theBase):
    __tablename__ = 'products'
    id= sa.Column(sa.Integer, primary_key=True)
    title= sa.Column('title', sa.String(32))
    in_stock= sa.Column('in_stock', sa.Boolean)
    quantity= sa.Column('quantity', sa.Integer)
    price= sa.Column('price', sa.Numeric)
```

ORM – RELATIONSHIPS

SQLAlchemy supports the following relationship types:

- One to Many
- Many to One
- One to One
- Many to Many

The sqlalchemly.relationship method and sqlalchemy.ForeignKey class are used to establish relationships between objects.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import
declarative_base


conn = sa.create_engine("sqlite:///movie.db")


theBase = declarative_base()


class Article(theBase):
    __tablename__ = 'articles'
    id = sa.Column(sa.Integer, primary_key=True)
    comments = sa.relationship("Comment")



class Comment(theBase):
    __tablename__ = 'comments'
    id = sa.Column(sa.Integer, primary_key=True)
    article_id = sa.Column(sa.Integer,
sa.ForeignKey('articles.id'))
```

MORE RESOURCES

- Object Relational Tutorial (1.x API) — SQLAlchemy 1.4 Documentation
- Relationships API — SQLAlchemy 1.4 Documentation

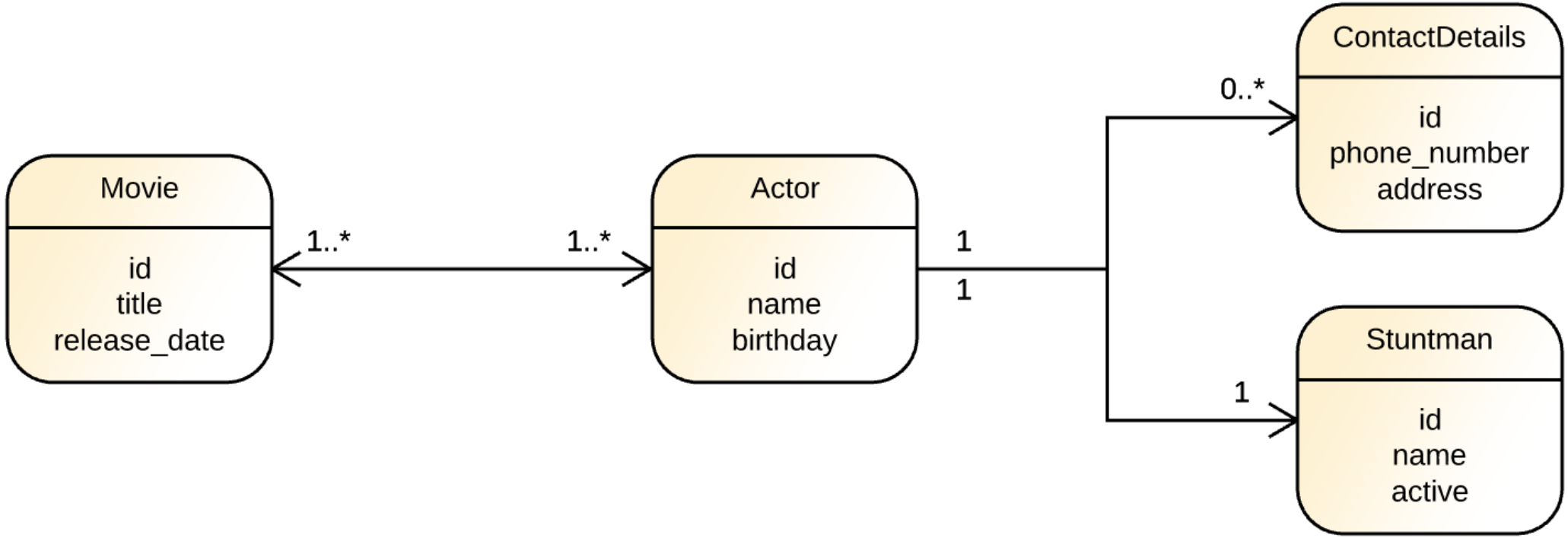
LAB: MOVIES

This lab:

In this lab, you will map four simple classes that represent movies, actors, stuntmen, and contact details.

LAB: MOVIES

Given the following entity relationship diagram, create a sqllite databases for the 4 tables and their relationships.



LAB: BASE CLASS

A base class with the following code has already been created in the labs folder.

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import
declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine("sqlite:///movie.db")
Session = sessionmaker(bind=engine)

Base = declarative_base()
```

Create and map the following child classes of the Base class:

- Movie
- Actor
- Stuntman
- ContactDetails

LAB: PERSIST DATA

1. Open PersistData.py and add the necessary import statements.

2. Write additional code to insert the records into the movies database.

3. Create a generic class to read data for any type of object. The class must have the following methods:

- read_all
- read_first
- get_one