Welcome to Jupyter!

- colab.research.google.com --> New Notebook

```
print("hello world")

    hello world
```

- Click shift+return (mac) to run the cell
- Once the cell has successfully run, a green checkmark will appear
- Or you can run all cells at once

NumPy Basics

- NumPy is a library of one-line vectorized math functions that have been optimized

```
# task: take the dot product of these two vectors

v_1 = [1, 2, 3, 4, 5]
v_2 = [6, 7, 8, 9, 10]
```

Non-vectorized code:

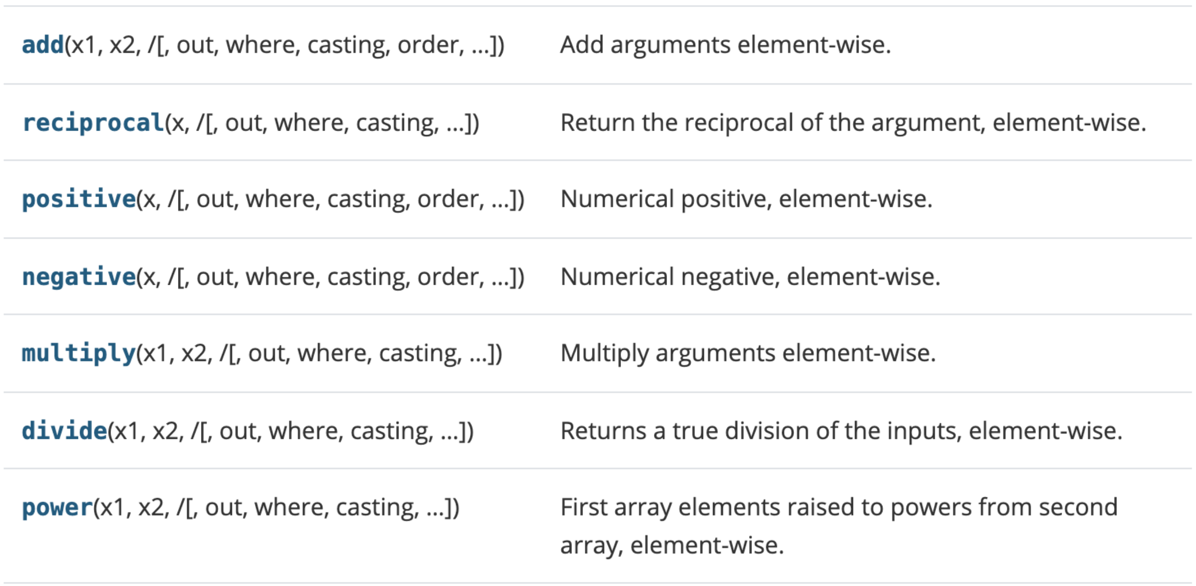
```
agg = 0
assert len(v_1) == len(v_2)
for i in range(len(v_1)):
  agg += v_1[i]*v_2[i]
print(agg)

    130
```

Vectorized code:

```
import numpy as np
```

```
print(np.dot(v_1, v_2))

    130
```

- Fewer lines of code
- Intuitive linear algebra notation (important for later!)
- There exists a NumPy function for any math operation you will need



add(x1, x2, /[, out, where, casting, order, ...])	Add arguments element-wise.
reciprocal(x, /[, out, where, casting, ...])	Return the reciprocal of the argument, element-wise.
positive(x, /[, out, where, casting, order, ...])	Numerical positive, element-wise.
negative(x, /[, out, where, casting, order, ...])	Numerical negative, element-wise.
multiply(x1, x2, /[, out, where, casting, ...])	Multiply arguments element-wise.
divide(x1, x2, /[, out, where, casting, ...])	Returns a true division of the inputs, element-wise.
power(x1, x2, /[, out, where, casting, ...])	First array elements raised to powers from second array, element-wise.

Importing TensorFlow

```
# magic command to convert to v1
%tensorflow_version 1.x

import tensorflow as tf
```

```
    TensorFlow 1.x selected.
```

- We use *tf* as convention

Tensors

- A tensor is a generalization of a matrix that allows for an arbitrary number of dimensions
- Scalar - rank 0, vector - rank 1, matrix - rank 2, Tensor - rank 3+
- For example, colored images are rank 3 tensors *(x, y, RGB channels)*

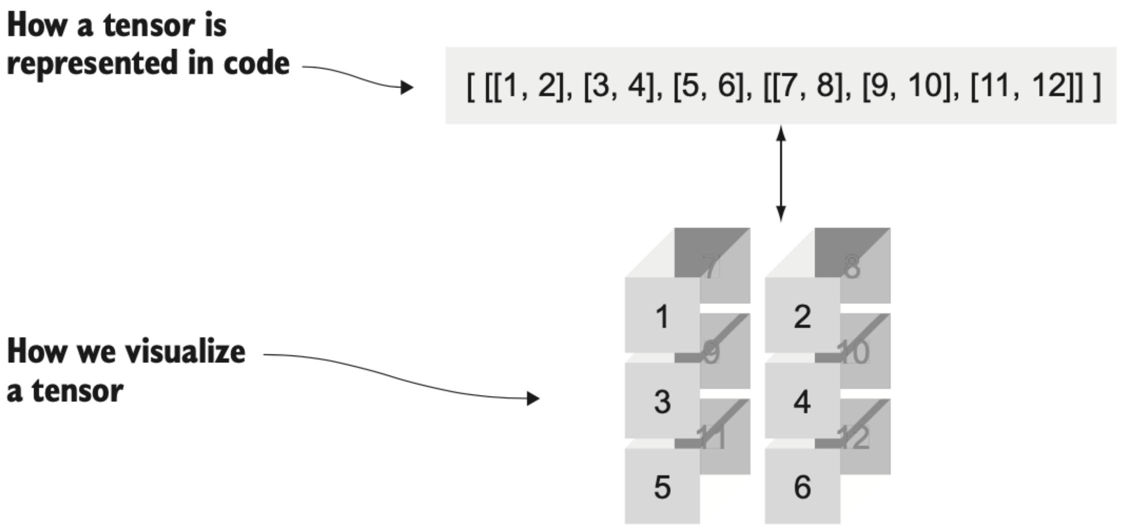


Figure 2.2 You can think of this tensor as being multiple matrices stacked on top of one another. To specify an element, you must indicate the row and column, as well as which matrix is being accessed. Therefore, the rank of this tensor is 3.

```
A = [[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10], [11, 12]]]
print(A[0][1][0])  # accessing a value from a 3-d tensor requires 3 indices
```

```
    3
```

```
# same matrix in 3 types

m1 = [[1.0, 2.0], [3.0, 4.0]]

m2 = np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32)

m3 = tf.constant([[1.0, 2.0], [3.0, 4.0]])

print(type(m1))
print(type(m2))
print(type(m3))
```

```
    <class 'list'>
    <class 'numpy.ndarray'>
    <class 'tensorflow.python.framework.ops.Tensor'>
```

```
# all tensorflow operations must be applied to tensor types (like m3)

t1 = tf.convert_to_tensor(m1, dtype=tf.float32)
```

```
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)

print(type(t1))
print(type(t2))
print(type(t3))
```

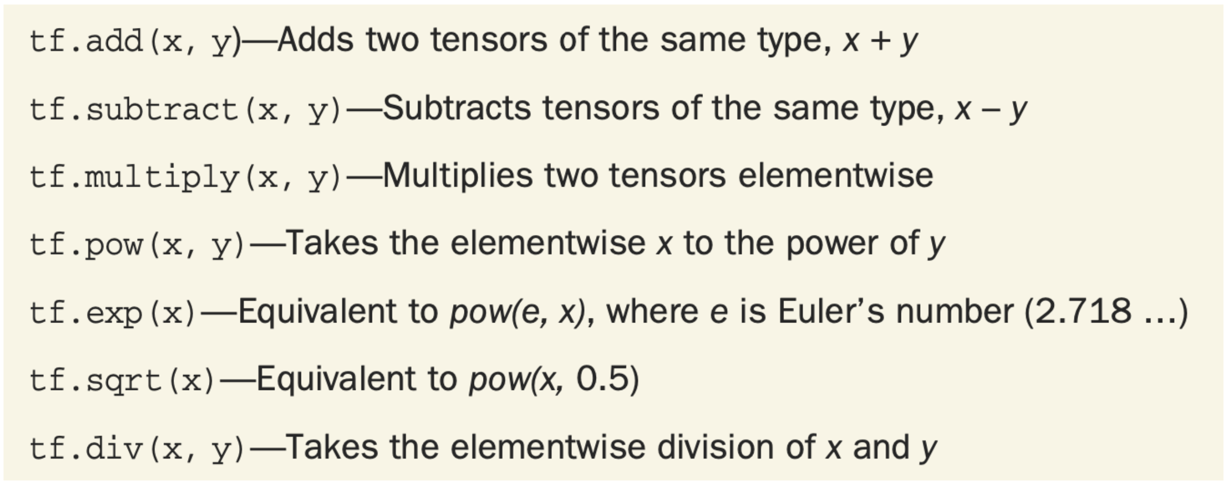
```
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

```
# printing the object itself gives us more information

print(t1)
```

```
Tensor("Const_1:0", shape=(2, 2), dtype=float32)
```

Executing Operations

- Similar to NumPy functions on vectors, Tensorflow has a set of functions to operate on tensors



`tf.add(x, y)`—Adds two tensors of the same type, *x* + *y*

`tf.subtract(x, y)`—Subtracts tensors of the same type, *x* – *y*

`tf.multiply(x, y)`—Multiplies two tensors elementwise

`tf.pow(x, y)`—Takes the elementwise *x* to the power of *y*

`tf.exp(x)`—Equivalent to *pow(e, x)*, where *e* is Euler's number (2.718 …)

`tf.sqrt(x)`—Equivalent to *pow(x,* 0.5)

`tf.div(x, y)`—Takes the elementwise division of *x* and *y*

Sessions

- TensorFlow defines computation in the form of a Graph
- Sessions allows TF to execute graphs and allocate/free resources for efficiency
- We must be within a session to perform any operations

```
# use an operation to negate an arbitrary vector

x = tf.constant([[1., 2.]])
neg_op = tf.negative(x)  # define the operation

with tf.Session() as sess:  # start a session
  result = sess.run(neg_op)  # execute the operation

print(result)
```

```
[[-1. -2.]]
```

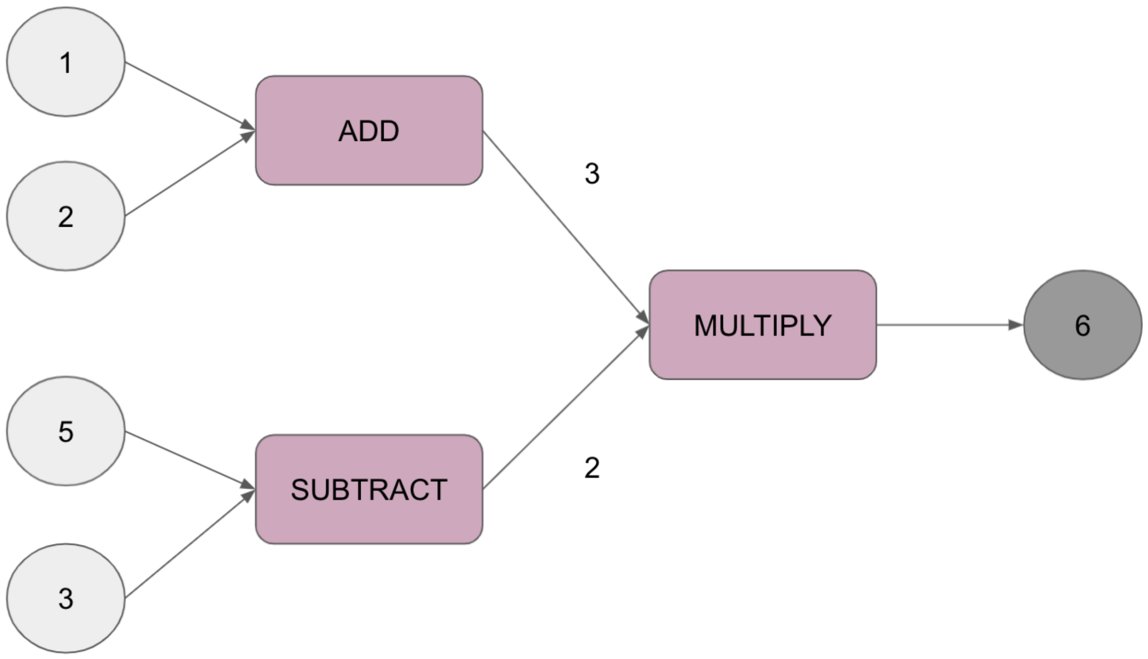
- Can use interactive sessions so that we don't have to re-open the session every time we want to execute an operation

```
# use an operation to negate an arbitrary vector

sess = tf.InteractiveSession()

x = tf.constant([[1., 2.]])
neg_op = tf.negative(x)  # define the operation
result = neg_op.eval()  # execute the operation using eval()

print(result)

sess.close()  # be sure to free computation and memory resources
```

```
[[-1. -2.]]
```

Graph Computing

- The essence of TensorFlow is in its representation of computation as a graph
- Allows for intuitive decoupling of parallel tasks



- Nodes indicate states
- Easy to see where parallelization is possible

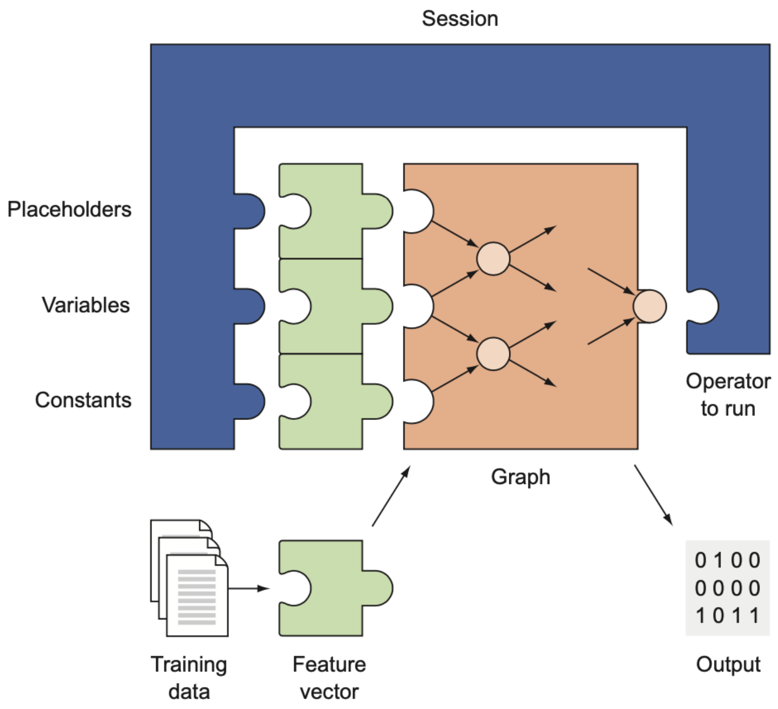


Figure 2.4 The session dictates how the hardware will be used to process the graph most efficiently. When the session starts, it assigns the CPU and GPU devices to each of the nodes. After processing, the session outputs data in a usable format, such as a NumPy array. A session optionally may be fed placeholders, variables, and constants.

- In essence the session is an abstraction for the computation that goes on under the hood

Variables

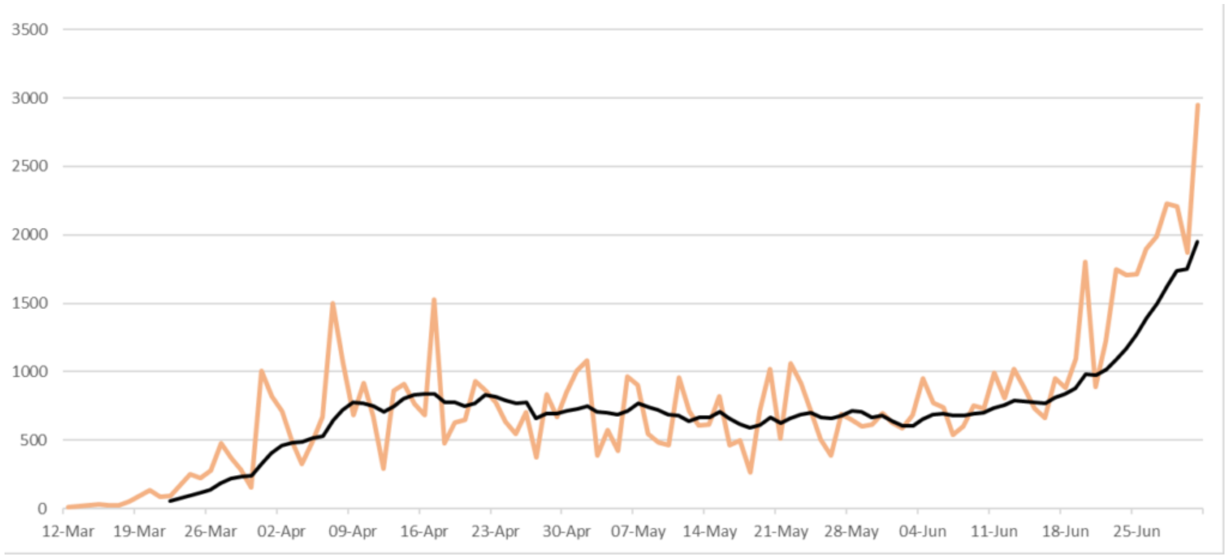
- So far we've only been working with TensorFlow constants
- Any interesting applications will involve values that change over time

```
# parse through a vector of real numbers, detect spikes of magnitude >5

sess = tf.InteractiveSession()

raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]

spike = tf.Variable(False)  # initialize spike variable to have value False
spike.initializer.run()  # initialize the variable

for i in range(1, len(raw_data)):
  if raw_data[i] - raw_data[i-1] > 5:  # if spike detected
    tf.assign(spike, True).eval()  # change variable to True
  else:
    tf.assign(spike, False).eval()  # change variable to False
  print("Spike", spike.eval())

sess.close()
```

```
    Spike False
    Spike True
    Spike False
    Spike False
    Spike True
    Spike False
    Spike True
```

Solving a mathematical problem in TensorFlow

- Suppose we have a stream of data from an unknown distribution
- We want to predict the average at the current time-step
- How would we do this?



Exponential Moving Average

$$s_0 = x_0$$
$$s_t = \alpha x_t + (1 - \alpha) s_{t-1}, \quad t > 0$$

where α is the *smoothing factor*, and $0 < \alpha < 1$.

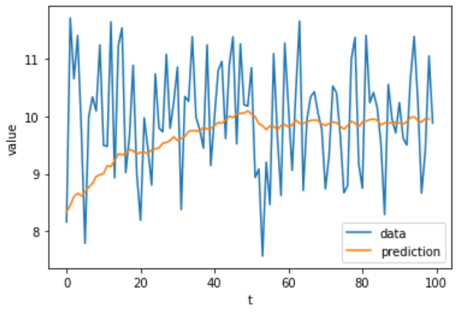
```
# run an exponential averaging algorithm on a randomly generated dataset

raw_data = np.random.normal(10, 1, 100)  # generate 100 data points following N(10, 1)

alpha = tf.constant(0.05)  # user-defined hyperparameter
curr_value = tf.placeholder(tf.float32)  # unassigned but will be initialized later in the session
prev_avg = tf.Variable(raw_data[0], dtype='float32')  # variable to hold average of previous values
update_avg = alpha * curr_value + (1 - alpha) * prev_avg  # operation to update predicted average

init = tf.global_variables_initializer()  # initializes all variables at once

averages = []

with tf.Session() as sess:
  sess.run(init)
  for t in range(1, len(raw_data)):
    curr_avg = sess.run(update_avg, feed_dict={curr_value:raw_data[t]})  # compute update_avg operation by feeding in i'th datapoint
    averages.append(curr_avg)
    sess.run(tf.assign(prev_avg, curr_avg))  # update prev_avg
    print(raw_data[t], curr_avg)
```

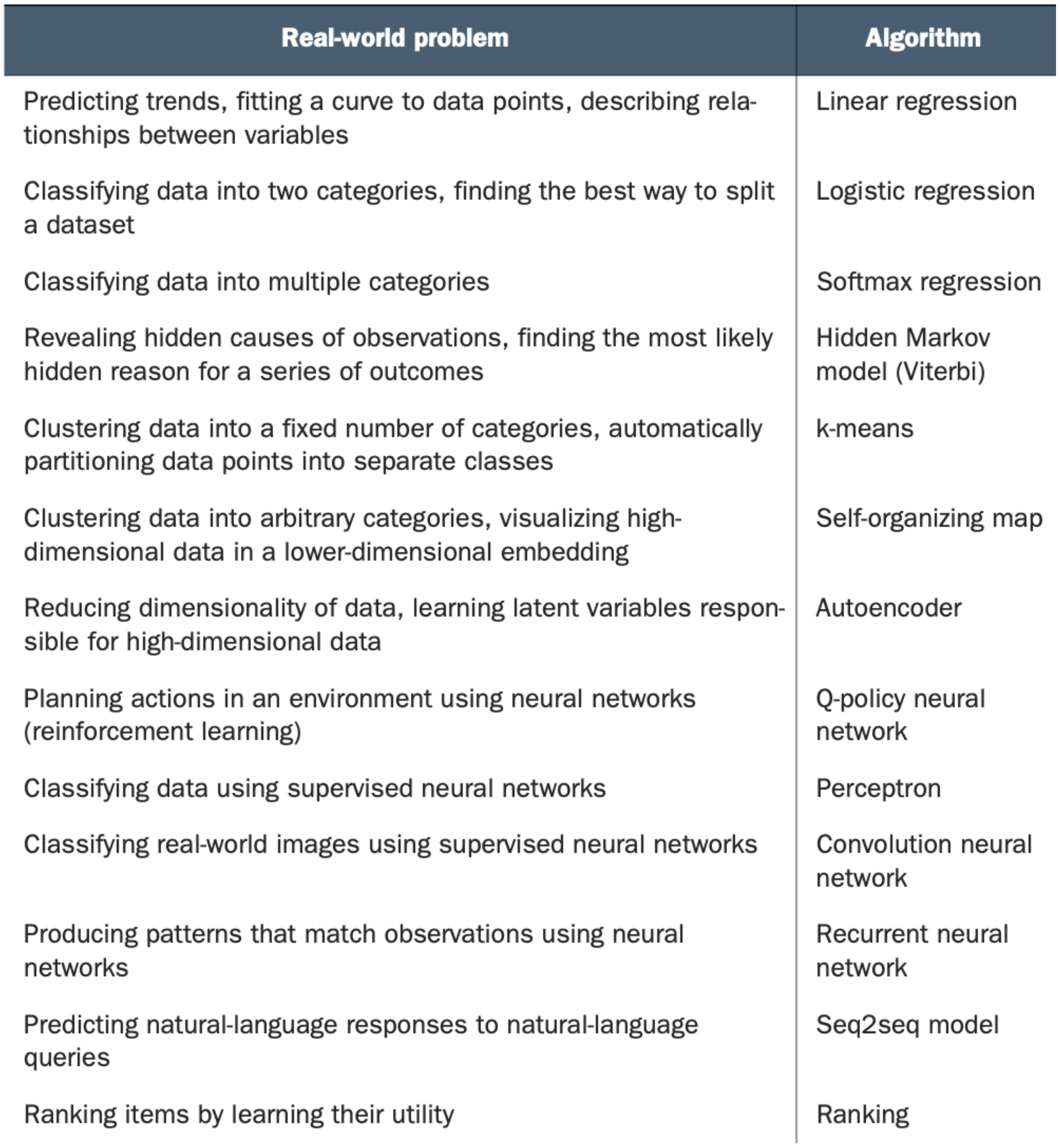
```
10.799188572824251 9.8405075
10.95713126177494 9.896338
9.611098385039757 9.882076
10.89089252995961 9.932516
11.384740540263975 10.005127
9.522279613071419 9.980985
11.263585385315624 10.0451145
10.202728154917397 10.052996
10.17939837977944 10.059316
10.847786334507617 10.098739
8.935183776106042 10.040561
9.08521119029834 9.992793
7.568485118627644 9.871578
9.20234447996171 9.838117
8.46584890663585 9.769503
11.093730304342344 9.835714
9.657225868361497 9.82679
8.62255553194045 9.766578
11.277254414014628 9.842112
10.16380633608974 9.858196
9.061338840299769 9.818353
10.4278445328769 9.848826
11.66039629420202 9.939405
8.71081924595316 9.877976
9.941043799288883 9.881129
10.340673944242766 9.904106
10.430169033949177 9.9304085
10.05256927803258 9.936516
9.78294560729089 9.928837
8.73916874063854 9.869353
9.328492027025112 9.842311
10.52855779348223 9.876623
10.407660680602339 9.903174
9.688486790258839 9.89244
8.670866935386973 9.831361
8.792996847894853 9.779442
10.99253337447294 9.8400955
11.37488826670432 9.916835
9.178347839688449 9.87991
8.749350649333874 9.823382
11.410597048828478 9.902743
10.239983575273948 9.919605
10.416824486118742 9.944467
10.154429318812653 9.954965
9.544164084176748 9.934424
8.290674164584845 9.852237
10.558824878976006 9.887566
9.956955053047286 9.891035
9.712246826895417 9.882095
10.240980036843343 9.900039
9.622533896564306 9.886164
9.501091820253535 9.86691
10.66429044735684 9.906779
11.391029256376886 9.980991
10.261762180389145 9.995029
```

```
8.663229111643963 9.928439
9.395859394624035 9.901811
11.054250150557314 9.959433
9.885606971096191 9.955742
```

```
import matplotlib.pyplot as plt

plt.plot(raw_data, label="data")  # plot data observations
plt.plot(averages, label="prediction")  # plot EMA

plt.legend()  # show legend

plt.xlabel("t")  # x-axis label
plt.ylabel("value")  # y-axis label
```

```
Text(0, 0.5, 'value')
```





Real-world problem	Algorithm
Predicting trends, fitting a curve to data points, describing relationships between variables	Linear regression
Classifying data into two categories, finding the best way to split a dataset	Logistic regression
Classifying data into multiple categories	Softmax regression
Revealing hidden causes of observations, finding the most likely hidden reason for a series of outcomes	Hidden Markov model (Viterbi)
Clustering data into a fixed number of categories, automatically partitioning data points into separate classes	k-means
Clustering data into arbitrary categories, visualizing high-dimensional data in a lower-dimensional embedding	Self-organizing map
Reducing dimensionality of data, learning latent variables responsible for high-dimensional data	Autoencoder
Planning actions in an environment using neural networks (reinforcement learning)	Q-policy neural network
Classifying data using supervised neural networks	Perceptron
Classifying real-world images using supervised neural networks	Convolution neural network
Producing patterns that match observations using neural networks	Recurrent neural network
Predicting natural-language responses to natural-language queries	Seq2seq model
Ranking items by learning their utility	Ranking

✓ 0s completed at 1:14 PM