DATA 601: HW2

Fall 2019

Due: Wed. Sep. 25, 2019 (by 23:55)

Learning Objectives

- Work with realworld datasets that can be represented using linear data structures.
- Apply vectorization concepts to an iterative problem.
- Explore different programming paradigms to solve problems.

This is an individual homework assignment.

Please complete this homework assignment within the Jupypter notebook environment, making use of Markdown and Code cells to properly format your answers.

Your completed Jupyter notebook is to be submitted via the HW2 dropbox on D2L.

Question 1 (8 points):

Visualizing trends in an index

This question asks you to plot the Bitcoin price index (BPI) along with 5-day and 10-day averages. Please execute the code cell below; it will read in a csv file containing the daily closing price from Sep. 1, 2018 to Aug. 31, 2019 (data obtained from [https://www.coindesk.com/price/ (https://www.coindesk.com/price/)](https://www.coindesk.com/price/)). Perform the following tasks. You may use any built-in Python functions as well as data strucutres and functions provided by the `numpy` library.

- Observe that the closing prices are at a daily interval. We therefore do not need the date information. Clean up the data and only retain the price information. Store the result in a list or a numpy array in floating point format.
- Recall that a [simple moving average (https://en.wikipedia.org/wiki/Moving_average#Simple_moving_average)](https://en.wikipedia.org/wiki/Moving_average#Simple_moving_average) is defined as the (unweighted) mean over the previous N days.
  Perform a simple moving average of the price index. The number of days N to average over should be adjustable. If you are using `numpy`, you may find the function [`np.convolve` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html) helpful.
- Plot the raw price index data along with 5-day and 10-day simple moving average. Plot on the same figure in order to help you visually ascertain the effect of the filter.
- What is the effect of the moving average filter? In what circumstances would you *not* want to use a moving average?

In [2]:

```python
import re

def fileToList( fname, regexp=r'\W+' ):
    file = open(fname, 'rt')
    text = file.read()
    file.close()
    # split based on provided regular expression and remove empty string
s
    # By default, matches words.
    return [x for x in re.split(regexp, text) if x]

bfile = "coindesk-bpi-close-data.csv"
bpi = fileToList( bfile, regexp=r'[,\r\n]+' )
# Print the head and tail.
print(bpi[:10:1])
print(bpi[-10::1])
```

```
['2018-09-01T22:59:59.404Z', '7198.0584362259', '2018-09-02T22:59:59.96
5Z', '7282.8585138311', '2018-09-03T22:59:59.783Z', '7260.2515925593',
'2018-09-04T22:59:59.331Z', '7361.2718642376', '2018-09-05T22:59:59.788
Z', '6913.1150129363']
['2019-08-27T23:00:00.000Z', '10122.1965329581', '2019-08-28T22:59:59.0
00Z', '9743.1620510451', '2019-08-29T22:59:59.000Z', '9487.9764335192',
'2019-08-30T22:59:59.000Z', '9590.7363369227', '2019-08-31T22:59:59.000
Z', '9624.9839105321']
```

Time series data, must check for monotone times.

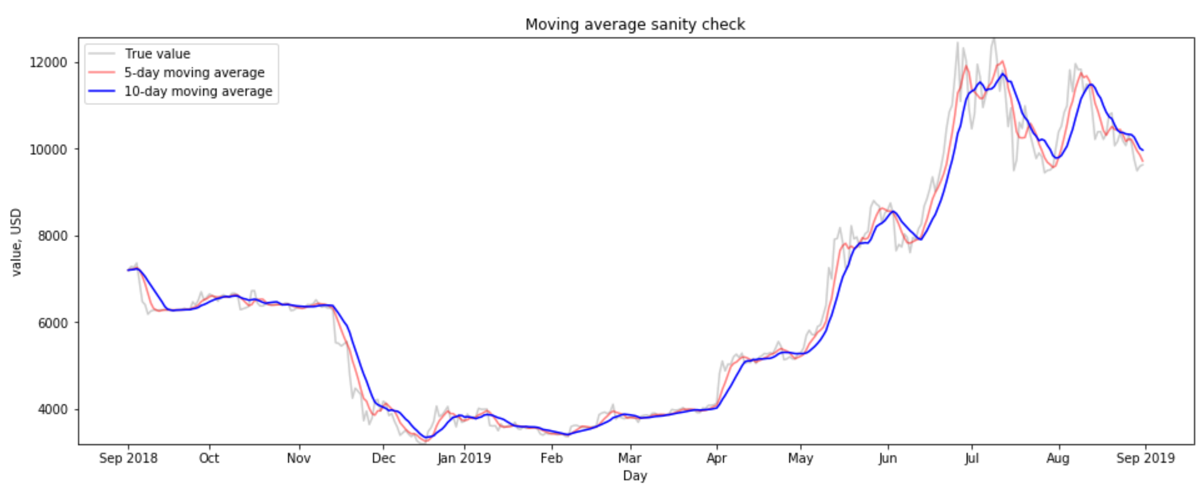
In [2]:

```
# make my life easier by invoking pandas
import pandas as pd
bitcoin_data = pd.read_csv('coindesk-bpi-close-data.csv', names=["timest
amp","value"])
bitcoin_data['timestamp'] = pd.to_datetime(bitcoin_data['timestamp'], in
fer_datetime_format=True)

%matplotlib inline
import matplotlib.pyplot as plt

plt.figure( figsize=(16,6) )
plt.plot( bitcoin_data['timestamp'], '-r' )

plt.title("Checking for monotonicity in the dataset")
plt.yticks([])
plt.ylabel("time")
plt.xlabel("entry number")

plt.show()
```

Close enough. Let's take the simple approach!

In [7]:

```
import numpy as np

coin_value = np.array(bitcoin_data['value'])
coin_value_count = len(coin_value)

# this is clever, but I'd dock myself points as I make it tough to adjus
t the window size.
coin_value_temp = np.sum( [np.pad( coin_value[0:coin_value_count-i], (i,
0), 'edge' ) for i in range(5)], axis=0 )
coin_value_5 = 0.2 * coin_value_temp

coin_value_temp += np.sum( [np.pad( coin_value[0:coin_value_count-i], (i
,0), 'edge' ) for i in range(5,10)], axis=0 )
coin_value_10 = 0.1 * coin_value_temp

# a better approach: https://stackoverflow.com/questions/14313510/how-to
-calculate-moving-average-using-numpy/14314054#14314054

plt.figure( figsize=(16,6) )
plt.plot( coin_value, '-k', alpha=0.2, label='True value' )
plt.plot( coin_value_5, '-r', alpha=0.5, label="5-day moving average" )
plt.plot( coin_value_10, '-b', alpha=1.0, label="10-day moving average"
)

plt.title("Moving average sanity check")
plt.ylabel("value, USD")
plt.ylim( [np.min(coin_value), np.max(coin_value)] )
plt.xlabel("Day")
plt.xticks( [0,          29,    61,    91,    120,        151,   179,
210,   240,   271,   301,   332,   363], \
            ['Sep 2018', 'Oct', 'Nov', 'Dec', 'Jan 2019', 'Feb', 'Mar',
'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep 2019'] )

plt.legend()

plt.show()
```



Moving average seems to smooth data. Handy at removing short-term trends, bad when short-term trends what you want.

In [5]:
```
# verify I have the labels in the right place
for i,v in enumerate([0, 29,    61,    91,    120,        151,   179,
210,   240,   271,   301,   332,   362]):
    print( "{}  ".format(bitcoin_data['timestamp'].loc[v]), end='' )
    if (i&0x3 == 3):
        print()
print()
```

```
2018-09-01T22:59:59.404Z  2018-10-01T03:59:59.835Z  2018-11-01T22:59:5
8.000Z  2018-12-01T22:59:59.000Z
2019-01-01T22:59:58.000Z  2019-02-01T23:00:00.000Z  2019-03-01T22:59:5
9.000Z  2019-04-01T22:59:58.000Z
2019-05-01T22:59:58.000Z  2019-06-01T22:59:59.000Z  2019-07-01T23:00:0
0.000Z  2019-08-01T22:59:59.000Z
2019-08-31T22:59:59.000Z
```

Question 2 (12 points):

Vectorized Race Simulation

1. Eight athletes are competing in a 1500 m race. Using `numpy` , write a vectorized race simulation according to the following criteria:

   - The granularity of the simulation is 1 s, i.e. each iteration in your simulation represents 1 second.
   - During each iteration, each athelete can randomly take 1, 2, 3, or 4 steps. Each step is 1 m long.
   - When the race is complete, return the winner and the winning time. There should not be any ties. If there is a tie, select a winner at random.

   Please pay attention to the following:

   - There should only be one loop in your simulation: the loop that advances the simulation by a second.
   - All other operations should be done using vectorized array operations and boolean indexing.
   - The following numpy functions will be helpful:
     - `numpy.random.randint` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randint.html)
     - `numpy.sum` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html)
     - `numpy.random.choice` (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.choice.html)

1. Run your simulation 10,000 times. For each run, record the winner and the winning time. Produce a bar chart showing the number of times each athelete won. Also display a bar chart showing the average winning time for each athlete.

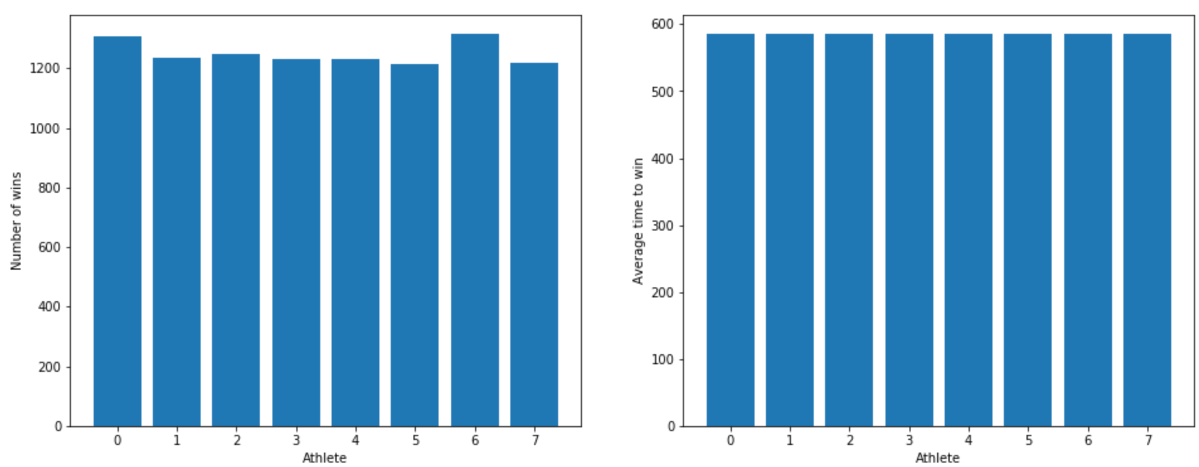
Part 1

In [19]:
```
import numpy as np  # this is redundant, but useful in case someone does
n't want to run prior cells

# Simple version first, where I use loops
def sim( distance=1500, athletes=8 ):
    """Simulate one race among a group of athletes."""

    time = 0
    dist = np.zeros(athletes, dtype=int)  # all these values are ints, s
o why not enforce that in the data?
    longest = np.max(dist)

    while longest < distance:

        # another bit of cleverness: Zeno's footrace!
        steps = max( int(distance - longest) >> 2, 1 )

        dist += np.sum( np.random.randint(low=1, high=5, size=(athletes,
steps)), axis=1)
        time += steps
        longest = np.max(dist)

    mask = (dist >= distance)
    index = np.random.choice( np.linspace(0,athletes-1,athletes, dtype=i
nt)[mask] )

    return (index, time)
```

Part 2

In [42]:

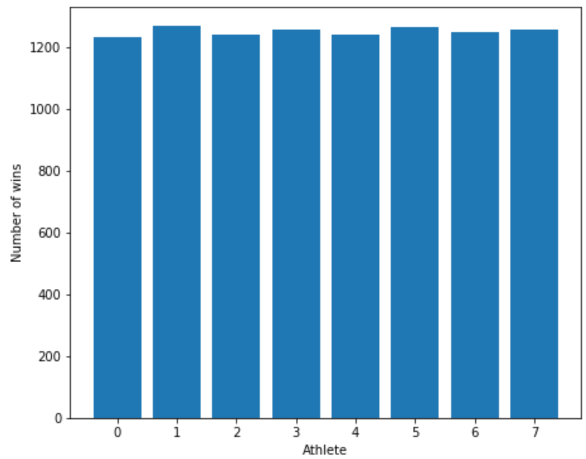
```
finish = 1500
ath = 8
sims = 10000

wins = np.zeros(ath, dtype=int)
times = np.zeros(ath, dtype=int)
for _ in range(sims):

    winner, time = sim(finish, ath)
    wins[winner] += 1
    times[winner] += time

plt.figure( figsize=(16,6) )
plt.subplot(1,2, 1)
plt.bar( np.linspace(0,ath-1,ath, dtype=int), wins, label='Total number of wins' )

plt.xlabel("Athlete")
plt.ylabel("Number of wins")


plt.subplot(1,2, 2)
plt.bar( np.linspace(0,ath-1,ath, dtype=int), times / wins, label='Average time to win' ) # note the potential division by zero!

plt.xlabel("Athlete")
plt.ylabel("Average time to win")


plt.show()
```



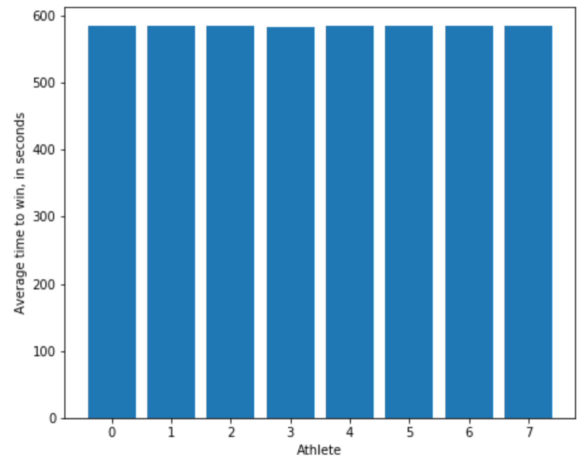
I should stress that the above code had multiple bugs on submission, and wouldn't have earned 100%. For instance, I was dividing by the wrong amount to normalize times, and I forgot about in-memory variables that I subsequently deleted from the code. I also never put units in, when I should have.

In [10]:

```
# ZERO LOOP SOLUTION!

# this one line generates distances for "ath" athletes over "sims" simul
ations
results = np.sum( np.cumsum( np.random.randint(low=1, high=5, size=(sims
,ath,finish)), axis=2 ) < finish, axis=2 )

wins = np.zeros(ath, dtype=int)
times = np.zeros(ath, dtype=int)

# more cleverness: use functions to hide loops
def accum(x, distance=1500):
    """Given one race outcome, update the global vars."""
    global wins, times

    athletes = len(x)
    least = x.min()
    mask = (x == least)
    index = np.random.choice( np.linspace(0,athletes-1,athletes, dtype=i
nt)[mask] )

    wins[index] += 1
    times[index] += x[index]

list( map( lambda x: accum(x, finish), results ) )  # technically, this
 isn't a loop!

# a student improved on this by adding a fractional jitter to every dist
ance,
#  allowing them to use `np.amin()` directly as opposed to my much-longe
r workaround.

plt.figure( figsize=(16,6) )
plt.subplot(1,2, 1)
plt.bar( np.linspace(0,ath-1,ath, dtype=int), wins, label='Total number
 of wins' )

plt.xlabel("Athlete")
plt.ylabel("Number of wins")


plt.subplot(1,2, 2)
plt.bar( np.linspace(0,ath-1,ath, dtype=int), times / wins, label='Avera
ge time to win' )

plt.xlabel("Athlete")
plt.ylabel("Average time to win, in seconds")  # see? It's a small thin
g, but greatly improves the user experience


plt.show()
```



Number of wins
0
200
400
600
800
1000
1200
Athlete
0 1 2 3 4 5 6 7



Average time to win, in seconds
0
100
200
300
400
500
600
Athlete
0 1 2 3 4 5 6 7