Activity 1: All Possible Regressions

This is a quick assignment. I would expect it to only take a couple of hours to complete. The start of this notebook develops the example. You will then parallelize the exammple using `joblib`.

Run this on a machine that has at least 4 cores. Typically this is your laptop. The ugrad machines are also OK. Google Colab is not adequate. Fill out the requested cells toward the bottom of the notebook.

Due date: Tuesday September 12, 2022, 9:00 pm EDT.

Instructions for Submission: Submit via Gradescope.

Preparing the Environment

You will need a couple of packages installed to make this work. Stop your instance of jupyter lab and then run

```
conda install pandas pandoc
jupyter lab
```

At this point, you should be ready to go.

Example code

This is a Python reimplementation of the Section 3.4. in *Matloff*, Parallel Computing for Data Science. It is based on data from https://www.kaggle.com/divan0/multiple-linear-regression. The notebook asks the question what combination of variables best predict the price of a house.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

%matplotlib inline

#importing dataset using panda
dataset = pd.read_csv('../data/kc_house_data.csv')
#to see what my dataset is comprised of
dataset.head()
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0

5 rows × 21 columns

In [3]:
```
#dropping the id and date column
dataset = dataset.drop(['id','date'], axis = 1)

# clean out NaN and inf values
dataset = dataset[~dataset.isin([np.nan, np.inf, -np.inf]).any(1)]
```

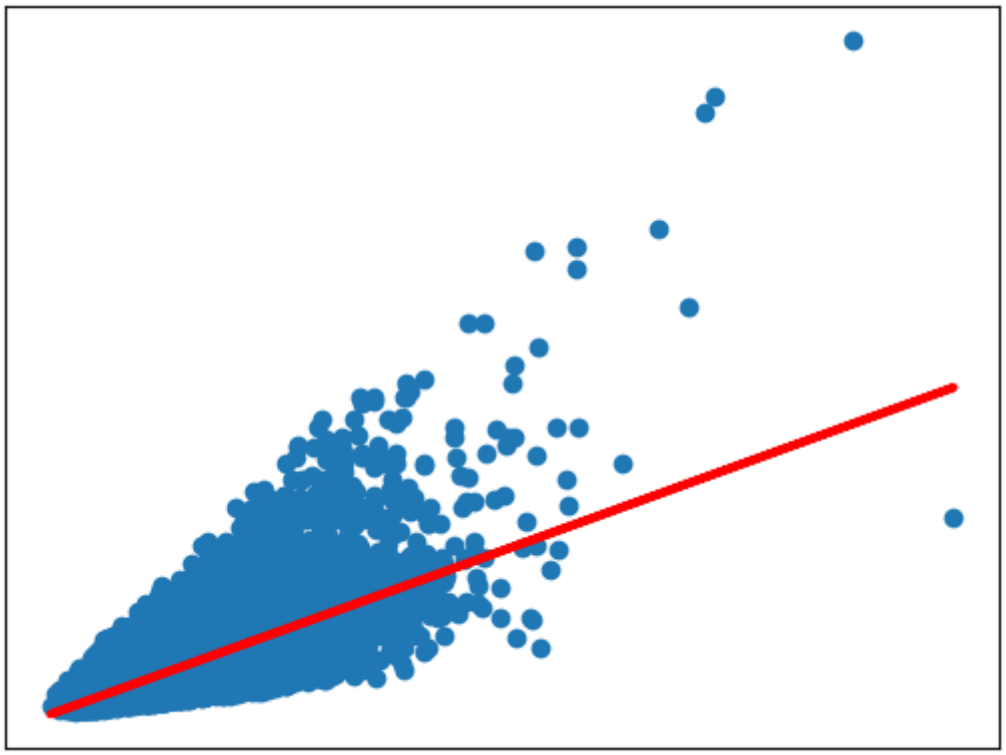
```
/tmp/ipykernel_27/3445377822.py:5: FutureWarning: In a future version of panda
s all arguments of DataFrame.any and Series.any will be keyword-only.
  dataset = dataset[~dataset.isin([np.nan, np.inf, -np.inf]).any(1)]
```

Let's first do a simple regression. How does square footage predict price?

In [4]:
```
from sklearn.linear_model import LinearRegression

X = np.array(dataset.sqft_living)
Y = np.array(dataset.price)

# shape X into matrix of a single column
X = X.reshape((X.shape[0],1))

model = LinearRegression()
model.fit(X,Y)
r_sq = model.score(X,Y)
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.4928817196006995
```

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt

# scatter data
plt.plot(X,Y,'o')
# best fit line
y_pred = model.intercept_ + model.coef_ * X
plt.plot(X, y_pred, color='red', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```



We see that there is a strong correlation between square footage and house price. The coefficient of determination measures the strength of the correlation and varies between 0 (no correlation) and 1.0 (perfectly correlated).

Multiple Linear Regression

Adding more variables often improves the score.

In [6]:
```
X = np.array(dataset[['sqft_living','condition','yr_built']])
Y = np.array(dataset.price)

# shape X into matrix of a single column
X = X.reshape((X.shape[0],3))

model = LinearRegression()
model.fit(X,Y)
r_sq = model.score(X,Y)
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.5249639858277426
```

but some variables are confounding

In [7]:
```
X = np.array(dataset[['zipcode','floors','waterfront']])
Y = np.array(dataset.price)

# shape X into matrix of a single column
X = X.reshape((X.shape[0],3))

model = LinearRegression()
model.fit(X,Y)
```

```
r_sq = model.score(X,Y)
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.1358874997770878
```

This leads to a first parallel program. What are the right set of variables? A brute force approach called *All Possible Regressions* examines all combinations. So, let's build a big matrix that and we will regress on subsets. We will look at all combinations of 1, 2, or 3 variables.

In [8]:
```
from itertools import combinations, chain

Y = np.array(dataset.price)
X = np.array(dataset.drop(['price'], axis=1))

## Let's choose all combinations of 1, 2, and 3 columns.
col_idxs = np.array(range(X.shape[1]))
combos = list(chain(combinations(col_idxs, 1), combinations(col_idxs, 2), combi
```

In [9]:
```
model = LinearRegression()

# do in a for loop (dumbest way)
r_sq_best = 0.0
for combo in combos:
    Xp = X[:,combo]
    model = model.fit(Xp,Y)
    r_sq = model.score(Xp,Y)
    if r_sq > r_sq_best:
        r_sq_best = r_sq
        combo_best = combo

print(r_sq_best, combo_best)
```

```
0.6095149101819037 (2, 6, 14)
```

The outcome is kind of crazy. The fields are `sqft_living`, `view`, and `latitude`. Latitude is probably a somewhat accurate proxy for wealth in this area, e.g. N of town richer than south of town. But, this is the kind of outcome that would not translate to other regions, i.e. is likely specific to this data. *Neat*.

Back to performance.

In [10]:
```
%%timeit
model = LinearRegression()

r_sq_best = 0.0
for combo in combos:
    Xp = X[:,combo]
    model = model.fit(Xp,Y)
    r_sq = model.score(Xp,Y)
    if r_sq > r_sq_best:
        r_sq_best = r_sq
        combo_best = combo
```

```
1.01 s ± 27.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Let's see if flattening the loop matters. Replace for loop with a list comprehension of all combination of variables.

In [11]:
```
def r_sq_regression (combo):
    Xp = X[:,combo]
    model = LinearRegression()
    model = model.fit(Xp,Y)
    return model.score(Xp,Y)
```

In [12]:
```
r_sq_list = [ (combo, r_sq_regression(combo)) for combo in combos ]
r_sq_arr = np.array(r_sq_list, dtype=object)
r_sq_idx = np.argmax(r_sq_arr[:,1])
print(r_sq_arr[r_sq_idx])
```

```
[(2, 6, 14) 0.6095149101819037]
```

In [13]:
```
%%timeit
r_sq_list = [ (combo, r_sq_regression(combo)) for combo in combos ]
r_sq_arr = np.array(r_sq_list, dtype=object)
r_sq_idx = np.argmax(r_sq_arr[:,1])
```

```
1.02 s ± 21.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It didn't seem to help. But, this is a step toward parallelization.

Exercise (This is the assignment)

1. Use `joblib.Parallel` and `joblib.delayed` to parallelize the computation of the calls to `r_sq_regression`.
   A. In one cell, print the answer to verify that your parallel program is correct.
   B. In another cell time the computation. What is the speedup at `n_jobs=4`?
   C. Estimate the optimized fraction of the code ($p) for this computation. Show your work.

In [14]:
```
# TODO code for 1A
def r_sq_regression (combo):
    Xp = X[:,combo]
    model = LinearRegression()
    model = model.fit(Xp,Y)
    return model.score(Xp,Y)

from joblib import Parallel, delayed
partials = Parallel(n_jobs=4)(delayed(r_sq_regression)(i) for i in combos)

print(max(partials))
print(combos[partials.index(max(partials))])
```

```
0.6095149101819037
(2, 6, 14)
```

In [15]:
```
%%timeit
# TODO code for 1B
partials = Parallel(n_jobs=4)(delayed(r_sq_regression)(i) for i in combos)
```

```
576 ms ± 92.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

TODO Answer for 1B The speedup at n_jobs=4 is 542 ms ± 11.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

PART 1C: Formula for Amdahl's Law is S = 1 / (1 - p + p/s) where S is the average time it takes (speedup), lowercase s is for how many cores there are. Solving for the optimized fraction of the code p for this computation would be:

(reference) example in class using a speedup of 80 with 100 processors: 80 = 1 / ((1-p)+p/100)

IN THIS CASE, using my sample of a speedup of 548 ms with 4 processors 542 = 1 / ((1 - p) + p/4)

542 = 1 / (1 - 3p/4)

1/542 = 1 - 3p/4

3p/4 = 1 - 1/542

p = 4/3 * (1 - 1/542)

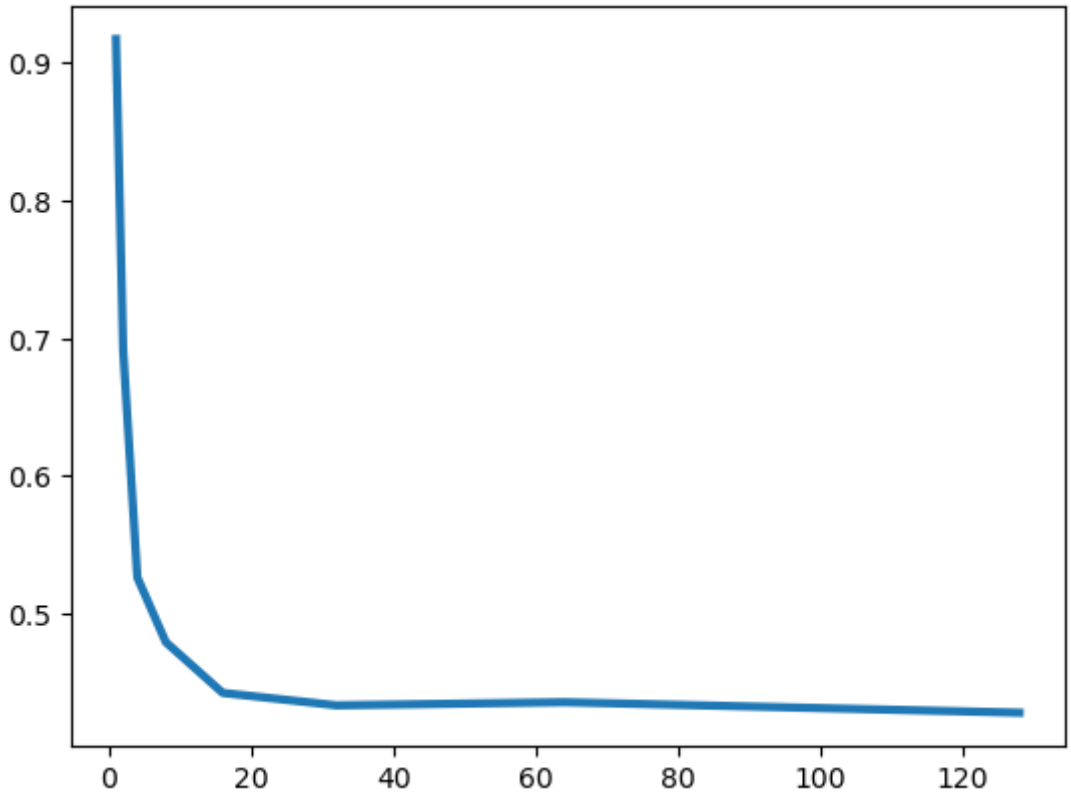
p = 1082/813

1. Use the batch size parameter to vary the number of tasks in each batch from 1,2,...128 @ n_jobs=4. You will need to look at the joblib documentation to read up about batch size.
   A. Plot your results (use %timeit -o to capture output)
   B. Model the problem as having two performance components: a fixed startup cost per batch (C_B) and perfect parallelism. Estimate the batch startup cost.

In [16]:
```
# code for 2A
from joblib import Parallel, delayed

powers = [1, 2, 4, 8, 16, 32, 64, 128]
time_results = []

for i in powers:
    time = %timeit -o Parallel(n_jobs=4, batch_size=i)(delayed(r_sq_regression)
    time_results.append(time)

#batch_size="auto" #auto keeps track of the time for a batch to complete
```

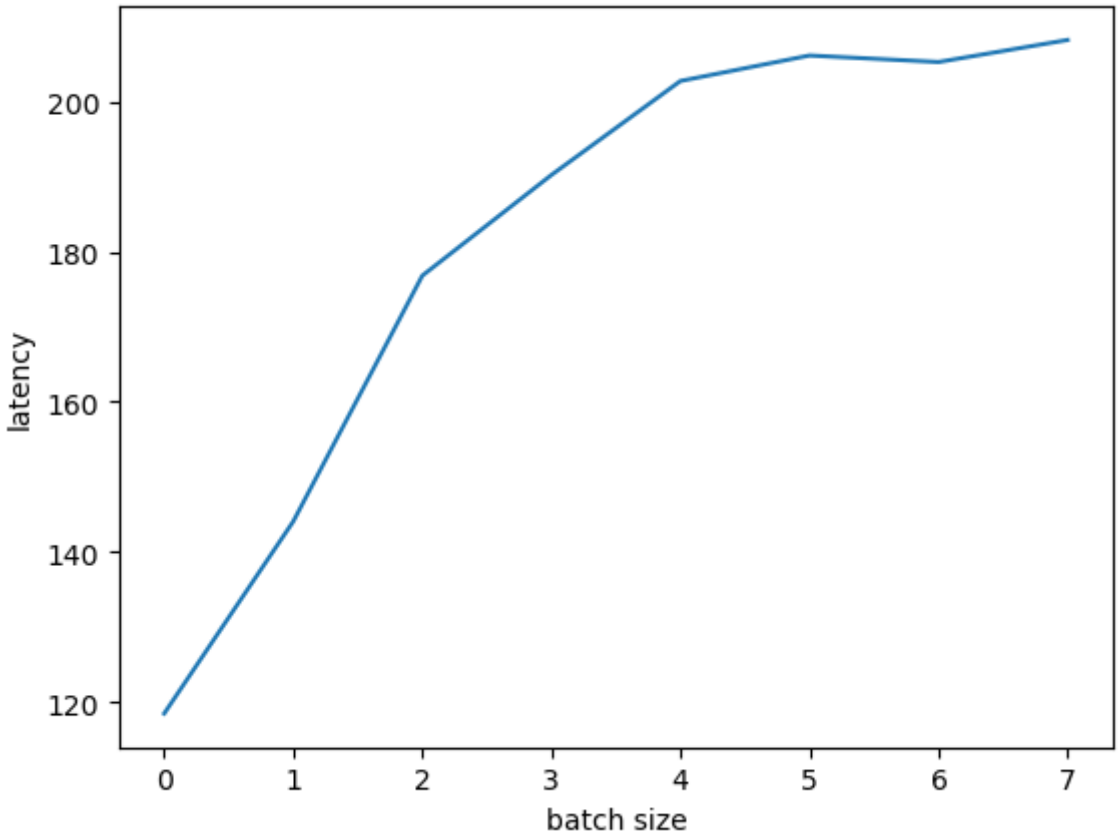
```
917 ms ± 63.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
692 ms ± 36.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
526 ms ± 9.15 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
479 ms ± 4.44 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
442 ms ± 4.72 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
433 ms ± 3.93 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
436 ms ± 9.51 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
428 ms ± 3.45 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [17]:
```
%matplotlib inline

# TODO plot for 2A
import matplotlib.pyplot as plt

results_avgs = [time.average for time in time_results] # get averages of speedu
plt.plot(powers, results_avgs, linewidth=3, label="speedup")

plt.show()
```



In [22]:
```
# code for 2B
# a fixed startup cost per batch ( CB )is the initialization time required for
# perfect parallelism is after start cost,the remainder computation runs linear
import matplotlib.pyplot as plt
# Model the problem as having two performance components: a fixed startup cost
def two_performance_components(batch_size, cb, t):
    l = len(results_avgs)
    n = batch_size
    batch_time = cb * (l/n)
    perfect_parallel = l * (t/4)
    return batch_time + perfect_parallel
plt.xlabel('batch size')
plt.ylabel('latency')
plt.plot(two_performance_components(np.array(results_avgs), 9, 20))
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0xffff52c1b7c0>]
```



TODO Answer for 2B Formula to represent execution time E(n) of this problem, given perfect paralellism:

(Equation based in class): E(b) = Cb *(l/b) + (l* t)/s where Cb is the batch start up cost per batch, l is the # of items our function is running, b is number of batches, t tracks the amount of time it takes per item, s is # of cores (and with perfect paralellism, its speedup is 4). Thus the equation becomes:

E(n) = Cb *(l/b) + (l* t)/4

Using this equation, a good estimate is around 8.5ms for Cb, the batch start up cost per batch.

1. Run the job with `prefer='threads'` and `prefer='processes'`. You do not need to vary batch size for this part. Which is more efficient? Why? Consider our discussion of parallel threads in python.

In [ ]:
```
# Code for 3
%timeit -o Parallel(n_jobs=4, prefer='threads')(delayed(r_sq_regression)(combo)
%timeit -o Parallel(n_jobs=4, prefer='processes')(delayed(r_sq_regression)(comb
```

```
1.16 s ± 41.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

TODO Answer to 3

threads results: 1.17 s ± 88.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
processes results: 527 ms ± 4.44 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Processes results is more efficient. In class, we discussed that parallel threads in python are ineffective in which we would need 'optimized packages that release the GIL' (Global Interpreter Lock). Releasing the lock during execution means other threads could access the Python interpreter while a function is already running.