Modeling Performance

In addition to solving the model quickly, getting the answer also requires building the model quickly

- In this section we present some methods used to build models using docplex in python.
- The constraint we are building is for a classroom scheduling model. In this model, we cannot have any potential sections overlap at any time for a specific room.
- $\sum_{s \in o(t) \backslash \mathrm{t}} C_s \leq \|s \in o(t) \backslash \mathrm{t}\| - \|s \in o(t) \backslash \mathrm{t}\| \sum_{u \in \partial(t)} C_u \; \forall \; t \in time\ periods$

We apply three approaches

- We want to use as much out-of-the box functionality as possible
  - Looping: In looping we iterate through lists and add variables to create the constraints
  - Pandas: We leverage the pandas package to do the looping with single commands and group the data together
  - Multiprocessing: We use the python multiprocessing package to send multiple jobs to run in parallel.
- We randomly generate three potential sections (start times) for 10,000 different classes and then for 1,000 different classes. We build the constraint to ensure there are no overlaps.
- The timing does not include the time required to create the data structures.

Looping

```
for t in potentialTimes.timeID.unique():

            #find the times that overlap with the given time

            lhsVars = []

            for idx, row in overlappingTimes[(overlappingTimes.timeID_x == t) &
                        (overlappingTimes.timeID_y != t)].iterrows():

                for idx2, row2 in potentialTimes[potentialTimes.timeID ==
                            row.timeID_y].iterrows():

                    lhsVars.append(row2.scheduleSection)

            rhsVars = []

            for idx3, row3 in potentialTimes[potentialTimes.timeID ==
                        t].iterrows():

                rhsVars.append(row3.scheduleSection)

            m.add_constraint(m.sum(lhsVars) <= len(lhsVars) - len(lhsVars) *
                        m.sum(rhsVars), 'overlap_%s'%t)
```

Looping results

- 10,000 classes
- $\mu = 17.40$
- $\sigma = 0.16$
- $n = 5$

- 1,000 classes
- $\mu = 2.03$
- $\sigma = 0.09$
- $n = 5$

Pandas

```
gb = potentialTimes.groupby(by='timeID').scheduleSection.sum()

overlaps = overlappingTimes[overlappingTimes.timeID_x !=
overlappingTimes.timeID_y].merge(potentialTimes,how='inner',left_on =
      'timeID_y',right_on='timeID')

numOverlaps = overlaps.groupby(by='timeID_x').timeID_y.count()

overlappingSections = overlaps.groupby(by='timeID_x').scheduleSection.sum()

m.add_constraints([(overlappingSections[idx] <= numOverlaps[idx] - numOverlaps[idx]
            * sections,'overlap_%s'%idx) for idx, sections in gb.iteritems()])
```

Pandas results

- $10{,}000$ classes
- $\mu = 15.02$
- $\sigma = 0.38$
- $n = 5$

- $1{,}000$ classes
- $\mu = 0.27$
- $\sigma = 0.02$
- $n = 5$

- The pandas coding is more compact, but requires getting used to creating merges and groupbys.
- Pandas is slowed by the merge, which in this case is close to a Cartesian multiplier.

Multiprocessing

```
# Create queues
task_queue = Queue()
done_queue = Queue()

# Start worker processes
for i in range(NUMBER_OF_PROCESSES):
      Process(target=worker, args=(task_queue, done_queue)).start()
NUM_TASKS = 0

task_queue.put((aggregate, potentialTimes[['timeID','varNames']], 'timeID',
             'varNames', 'gb')))
gb = 0
NUM_TASKS += 1
      task_queue.put((merge, (overlappingTimes[overlappingTimes.timeID_x !=
overlappingTimes.timeID_y],potentialTimes[['timeID','varNames']],'inner','timeID_y'
```

Multiprocessing continued

```
task_queue.put((merge, (overlappingTimes[overlappingTimes.timeID_x !=
            overlappingTimes.timeID_y], potentialTimes[['timeID','varNames']],
            'inner', 'timeID_y', 'timeID', 'overlaps')))

overlaps = 0

NUM_TASKS += 1

overlappingSections = 0

NUM_TASKS += 1

numOverlaps = 0

NUM_TASKS += 1


      for idx in range(NUM_TASKS):
            result = done_queue.get()
            if result[0] == 'gb':
                  gb = result[1].map(lambda x: m.sum(getVarList(m,x,'!')))
            if result[0] == 'overlaps':
```

Multiprocessing continued

```
for idx in range(NUM_TASKS):
        result = done_queue.get()
        if result[0] == 'gb':
                gb = result[1].map(lambda x: m.sum(getVarList(m,x,'!')))
        if result[0] == 'overlaps':
                overlaps = result[1]
                task_queue.put((aggregate, (result[1], 'timeID_x', 'varNames',
                                'overlappingSections')))
                task_queue.put((getCount, (result[1], 'timeID_x', 'varNames',
                                'numOverlaps')))
                if result[0] == 'overlappingSections':
                        overlappingSections = result[1].map(lambda x:
                                m.sum(getVarList(m,x,'!')))
                if result[0] == 'numOverlaps':
                        numOverlaps = result[1]
```

Multiprocessing continued

```
m.add_constraints([(overlappingSections[idx] <= numOverlaps[idx] - numOverlaps[idx]
      * sections,'overlap_%s'%idx) for idx, sections in gb.iteritems()])
```

Multiprocessing results

- $10{,}000$ classes
- $\mu = 2.90$
- $\sigma = 0.23$
- $n = 5$

- $1{,}000$ classes
- $\mu = 1.70$
- $\sigma = 0.03$
- $n = 5$

- Multiprocessing requires far more coding, but can leverage more of the processors on your system.
- Docplex variable objects cannot be pickled (used to store until the multiprocessing can pick up the data to process), so the variable names must be passed back and forth. As a result, packages such as dask will not work to aggregate docplex variables in a DataFrame.
- There is some overhead in creating the queues and starting the processes, so it is not always the fastest.

Conclusion

- Docplex supports many ways to build model constraints.
- It may be easiest to start with an intuitive method and migrate to a fast method
- Leverage python packages such as pandas