Crunching Numbers with NumPy and Cython

Brian Lester

Interactions

January 2, 2019

Me

- Work on NLP at Interactions specializing in Deep Learning
- Published/Maintain several python packages
  - string-distance
  - text-rank
  - mead-baseline

- We use chat bots to "facilitate customer care interaction" aka to build better chat bots
- I mostly do DL and am a big Nerd so if you every want to talk about ML for NLP I'm game
- Python is my daily driver
- Have several packages on PyPI including ones written in cython (string-distance), the subject of today's talk

The Problem

Suppose I have a numpy array of N points in 2-space; what's the super-fast, pure-numpy way to get a symmetric, NxN matrix of pairwise manhattan distances? Here's a simple example and a slow, naive solution:

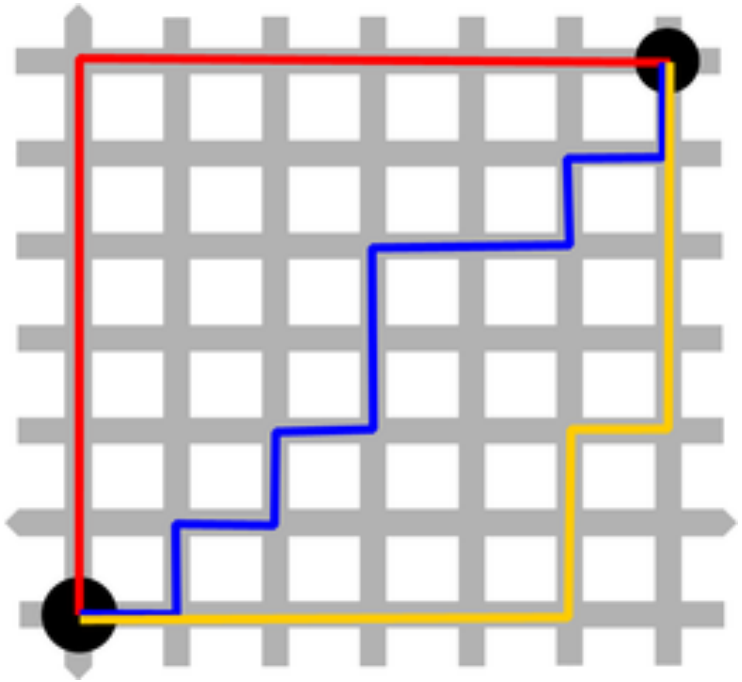
Figure: Ann Arbor Machine Learning and Data Science Slack

- a2mads is the Ann Arbor Machine Learning and Data Science Slack
- There aren't a lot of messages flying about but when someone asks a question there is normally a lot if discussion
- This user was asking that given a list of points what is the distance between a point and every other point in the list for each item in the list
- There was a long thread about how to optimize his solution, we will walk through some of them.

Manhattan Distance

- The distance between points measured along axes at right angles

- Distance if you can only move along the axes
- Contrast with euclidean distance where we can move diagonally and go straight there.
- Like you have directions to somewhere in manhattan, you can't smash through a building, you need to drive along the streets



- Here we can see example of moving between points as the Manhattan distance does

Manhattan Distance

- The distance between points measured along axes at right angles
- Also called the L1, cityblock, or taxi distance

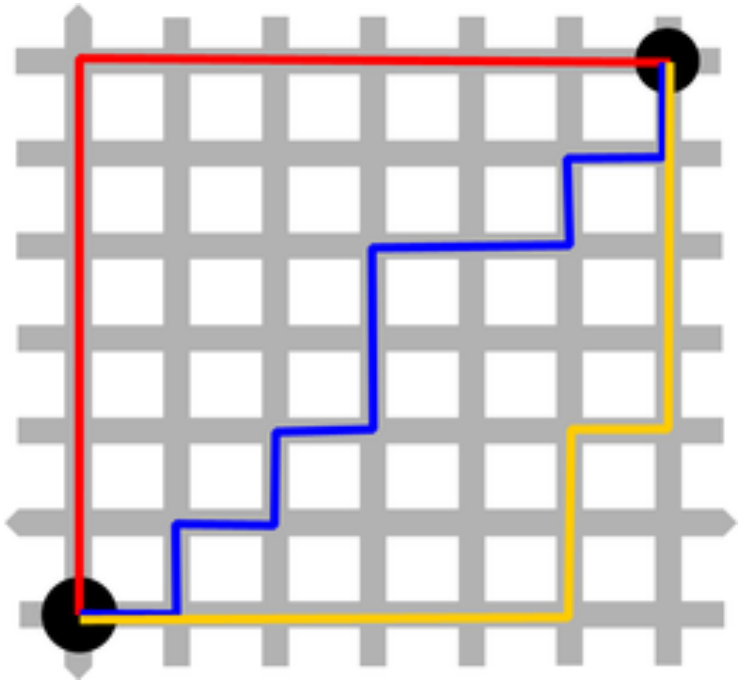
- Distance if you can only move along the axes
- Contrast with euclidean distance where we can move diagonally and go straight there.
- Like you have directions to somewhere in manhattan, you can't smash through a building, you need to drive along the streets

Manhattan Distance

A point with M dimensions is a vector of M numbers where the values at index i is the distance along the i^{th} axis.

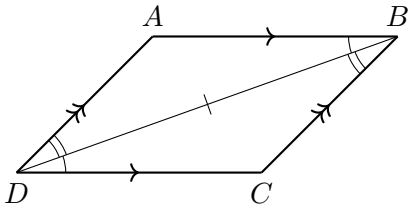
$$X \in \mathbb{R}^M$$
$$Y \in \mathbb{R}^M$$
$$\text{ManhattanDistance}(X, Y) = \sum_i^M |X_i - Y_i|$$

- While the Manhattan distance works for real numbers we will only be looking at Ints in code



- As suggested by the calculation for Manhattan Distance all of these paths have the same length, the next few slides sketch out why these are all the same but that isn't very important to us
- Point out the absolute distance between point dims on the red line, 6 up 6 over for a distance of 12
- All paths are length 12

Proof of Equal Lengths



Proof.
let $\overline{AB} \parallel \overline{DC}$ and $\overline{AD} \parallel \overline{BC}$ by the definition of a parallelogram.

$$\angle ABD \cong \angle CDB$$
$$\angle ADB \cong \angle CBD$$
$$\overline{DB} = \overline{DB}$$
$$\triangle ADB \cong \triangle CBD$$
$$\overline{AB} = \overline{DC}$$
$$\overline{AD} = \overline{BC}$$

□

- DB is a transversal of both sets of parallel lines
- ABD and CDB are congruent because they are alternate interior angles
- ADB and DBC are congruent because they are alternate interior angles
- It is the same length as itself
- Triangles are congruent by angle side angle
- AB and CD are the same because they are the same sides of congruent triangles
- AD and CB are the same because they are the same sides of congruent triangles

All Paths Equal

- Let R be a move 1 step *right* and U be a move 1 step *up*.
- Let the source point be $(0, 0)$ in our new coordinate system and the target be (p, q).
- Any possible path from source to target will have p R moves and q U moves.
- There are $p + q$ moves in total.
- There are $\binom{p+q}{p}$ possible paths all of the same length
- If you make an extra R or U move you will not be at your target and will need to make extra steps so it will not be the shortest path.

- This is a quick overview on why all paths between two points have the same length
- You can scramble the order of moves and the path still has the same number of Rs and Us but ends at the same spot
- It also sketches why the path length is always the length + height of the rectangle we make with the points as boxes
- This matches the formula where we take the absolute difference between points and sum across dimensions
- you can have a more rigorous proof via induction / contradiction that is based on the fact that if there was any shorter way to get to the next point then we could have gotten to this point through that but I didn't write it out here.
- The basic point is that because all paths are equal we can just look at the differences between the start and end, we don't need to care about the path take there.

NumPy

- NumPy: A collection of linear algebra functions that operate on a `ndarray`

- `ndarray` has a data type
  - all the items in it need to be the same and they can be stored contiguously in memory.
  - So instead of following pointers for to the object for each element in a python list we can access the element directly
  - This makes access fast and because they are together in memory we get better cache utilization
- `ndarray` is a multi dimensional array
  - a thing is a scalar
  - a list of things is an array (or vector)
  - a list of arrays is a matrix
  - a list of matrices is a `ndarray`
  - You can index like `x[i, j, k]`
  - can have an arbitrary dimensions
- A C extension is code written in C that can be called from python. You can use the speed of C and let go of the GIL. Loses ease of Python

NumPy

- NumPy: A collection of linear algebra functions that operate on a `ndarray`
- `ndarray` is a typed container that can represent multi-dimensional arrays

- `ndarray` has a data type
  - all the items in it need to be the same and they can be stored contiguously in memory.
  - So instead of following pointers for to the object for each element in a python list we can access the element directly
  - This makes access fast and because they are together in memory we get better cache utilization
- `ndarray` is a multi dimensional array
  - a thing is a scalar
  - a list of things is an array (or vector)
  - a list of arrays is a matrix
  - a list of matrices is a `ndarray`
  - You can index like `x[i, j, k]`
  - can have an arbitrary dimensions
- A C extension is code written in C that can be called from python. You can use the speed of C and let go of the GIL. Loses ease of Python

NumPy

- NumPy: A collection of linear algebra functions that operate on a `ndarray`
- `ndarray` is a typed container that can represent multi-dimensional arrays
- The majority of functionality is implemented in C extensions for speed

- `ndarray` has a data type
  - all the items in it need to be the same and they can be stored contiguously in memory.
  - So instead of following pointers for to the object for each element in a python list we can access the element directly
  - This makes access fast and because they are together in memory we get better cache utilization
- `ndarray` is a multi dimensional array
  - a thing is a scalar
  - a list of things is an array (or vector)
  - a list of arrays is a matrix
  - a list of matrices is a `ndarray`
  - You can index like `x[i, j, k]`
  - can have an arbitrary dimensions
- A C extension is code written in C that can be called from python. You can use the speed of C and let go of the GIL. Loses ease of Python

Cython

- Cython is a compiler for Python and Cython

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- It creates C extensions

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- It creates C extensions
- It can compile pure python code

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- It creates C extensions
- It can compile pure python code
- You can call python code from inside the cython code (and therefore from inside the C extension)

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- It creates C extensions
- It can compile pure python code
- You can call python code from inside the cython code (and therefore from inside the C extension)
- Makes it easy to wrap C libraries

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Cython

- Cython is a compiler for Python and Cython
- Cython is also a language that looks more like python than C, but adds types
- It creates C extensions
- It can compile pure python code
- You can call python code from inside the cython code (and therefore from inside the C extension)
- Makes it easy to wrap C libraries
- Some large libraries like SpaCy are all cython

- Cython is a superset of python, adds things like static types
- Compiling just python gives a slight speed boost
- You can also write code that releases the GIL so multithreading is possible
- We will see an example of why calling python code is nice

Let get to the code

Python Brute

```
def pairwise_manhattan_python_v1(
    points: List[List[int]]
) -> List[List[int]]:
    results = []
    # For each pair of points
    for x in points:
        dists = []
        for y in points:
            # Manhattan distance
            dist.append(
                sum(abs(x_i - y_i) for x_i, y_i in zip(x, y))
            )
        results.append(dist)
    return results
```

- It is 2020, python2 is dead, lets all start using type hints
- Input is a list of N points where each point is a List of M integers
- This is the most brute force approach, for all combinations of points calculate the Manhattan distance

Python Manhattan Distance

```
sum(abs(x_i - y_i) for x_i, y_i in zip(x, y))
```

- This is the calculation of manhattan distance directly translated from Math

Python Brute

```python
def pairwise_manhattan_python_v1(
    points: List[List[int]]
) -> List[List[int]]:
    results = []
    # For each pair of points
    for x in points:
        dists = []
        for y in points:
            # Manhattan distance
            dist.append(
                sum(abs(x_i - y_i) for x_i, y_i in zip(x, y))
            )
        results.append(dist)
    return results
```

- $\mathrm{O}(N^2M)$ in runtime
- $\mathrm{O}(N^2)$ in memory

- Big-O notation
- for x in points $\rightarrow$ a loop over N points $\mathrm{O}(N)$
- for y in points $\rightarrow$ a loop over N points for each point in the N points, $\mathrm{O}(N^2)$
- for x_i, y_i in zip(x, y) $\rightarrow$ a loop over M dimentions, $\mathrm{O}(N^2M)$
- Memory used is $\mathrm{O}(N^2)$ for the result matrix, external memory datastructures are not used

Python Brute

```
def pairwise_manhattan_python_v1(
    points: List[List[int]]
) -> List[List[int]]:
    results = []
    # For each pair of points
    for x in points:
        dists = []
        for y in points:
            # Manhattan distance
            dist.append(
                sum(abs(x_i - y_i) for x_i, y_i in zip(x, y))
            )
        results.append(dist)
    return results
```

- $\mathrm{O}(N^2M)$ in runtime
- $\mathrm{O}(N^2)$ in memory

- When optimizing you want to save all the time you can in the inner most loop
- every second wasted in the setup is only wasted once
- every second wasted in the inner loop is wasted over and over
- Python function calls have overhead so by manually inlining the Manhattan Distance calculation we get a speed up
- Without in-lining we pay for the call overhead $\mathrm{O}(N^2)$ times

Python Cached

```python
def pairwise_manhattan_python_v2(
    points: List[List[int]]
) -> List[List[int]]:
    # Preallocate the results
    results = [[None] * len(points) for _ in range(len(points))]
    # Look at every pair of you and the ones after you, distances
    # with points before you were calculated when they looked at you
    for i in range(len(points)):
        for j in range(i, len(points)):
            # Manhattan distance calculation
            dist = sum(
                abs(p1 - p2) for p1, p2 in zip(points[i], points[j])
            )
            # Manhattan distance is symmetric
            results[i][j] = dist
            results[j][i] = dist
    return results
```

- This is the only algorithmic improvemnt (not really, Big O is still the same but we are cutting out work) to our code
- The rest of our improvements are all computational should all show similar speed curves
- The Manhattan distance is symetric $(\text{ManhattanDistance}(X, Y) = \text{ManhattanDistance}(Y, X))$
- The distance from point i to point j is the same as the distance from point j to point i

Symmetric Manhattan Distance

$$\text{ManhattanDistance}(X, Y) = \sum_i^M |X_i - Y_i|$$

Proof.

$\text{ManhattanDistance}(X, Y) = \text{ManhattanDistance}(Y, X)$

Case (1)

let $x - y > 0$

$$\begin{aligned} y - x &< 0 \\ |x - y| &= x - y \\ |y - x| &= -(y - x) \\ -(y - x) &= x - y \end{aligned}$$

- The Manhattan distance is a distance metric and it be one mathematically it needs to be symmetrical but we can see why here
- The absoulte difference is the same when reversed so the sum is of the same values to the total sum is the same

- in the case where the difference is greater than one
- The switched must be negative
- Because we are positive value is just the difference
- The switched is the difference multiplied -1 because it negative and need to be positive
- distribute the multiplication and rearrange

Symetric Manhattan Distance

$$\text{ManhattanDistance}(X, Y) = \sum_i^M |X_i - Y_i|$$

Proof.

$\text{ManhattanDistance}(X, Y) = \text{ManhattanDistance}(Y, X)$

Case (2)

let $x - y < 0$

$$\begin{aligned} y - x &> 0 \\ |y - x| &= y - x \\ |x - y| &= -(x - y) \\ -(x - y) &= y - x \end{aligned}$$

- in the case where the difference is less than one
- The switched must be positive
- Our result if the difference multiplied by -1 because it negative and need to be positive
- Because the switched is positive its value is just the difference
- distribute the multiplication and rearrange

Symmetric Manhattan Distance

$$\text{ManhattanDistance}(X, Y) = \sum_i^M |X_i - Y_i|$$

Proof.
$\text{ManhattanDistance}(X, Y) = \text{ManhattanDistance}(Y, X)$

Case (3)

let $x - y = 0$

$$\begin{aligned} y - x &= 0 \\ |x - y| &= 0 \\ |y - x| &= 0 \end{aligned}$$

□

- In the case we are zero the switch is zero so we have the same value.

Python Cached

```python
def pairwise_manhattan_python_v2(
    points: List[List[int]]
) -> List[List[int]]:
    # Preallocate the results
    results = [[None] * len(points) for _ in range(len(points))]
    # Look at every pair of you and the ones after you, distances
    # with points before you were calculated when they looked at you
    for i in range(len(points)):
        for j in range(i, len(points)):
            # Manhattan distance calculation
            dist = sum(
                abs(p1 - p2) for p1, p2 in zip(points[i], points[j])
            )
            # Manhattan distance is symmetric
            results[i][j] = dist
            results[j][i] = dist
    return results
```

- We can reuse the result from one direction and skip computing it for the other
- Big O throws away constants so the is still $\mathrm{O}(N^2M)$
- We could use a similar trick to reduce memory usage by just having user of the results always index with some rule where the tuple of points need to be sorted or something similar, not worth it so still $\mathrm{O}(N^2)$ in memory.
- You could optimize a little bit here by initalizing the results matrix with zeros and skipping computing your distance to your self.

Numpy Brute

```python
def pairwise_manhattan_numpy_v1(
    points: List[np.ndarray]
) -> np.ndarray:
    results = np.zeros((len(points), len(points)), dtype=np.int32)
    for i in range(len(points)):
        for j in range(len(points)):
            results[i, j] = np.sum(np.abs(points[i] - points[j]))
    return results
```

- This is a pure port of original python version including the deplicated computation
- This version introduces a lot of numpy concepts but we might as well look at the next version to start

Numpy Cached

```python
def pairwise_manhattan_numpy_v2(
    points: List[np.ndarray]
) -> np.ndarray:
    results = np.zeros((len(points), len(points)), dtype=np.int32)
    for i in range(len(points)):
        for j in range(i, len(points)):
            dist = np.sum(np.abs(points[i] - points[j]))
            results[i, j] = dist
            results[j, i] = dist
    return results
```

- Preallocating the results as a large matrix
- Loop over all unique pairs of points
- Calculate the manhattan distance with numpy and save results into the matrix
- This was basically the first solution in the a2mads thread

Numpy Manhattan Distance

```
np.sum(np.abs(points[i] - points[j]))
```

- There are no loops visible in this code!
- This introduces us to vectorization!

Vectorization

- Code that can transparently operate on lists of numbers

```
def add(x, y):
    if isinstance(x, list):
        return [x_i + y_i for x, y in zip(x, y)]
    return x + y
```

- Moves the loops out of slow python into fast C

- A large part of performant python is trying to remove loops by vectorizing code
- The name implies it might be parallel processing but that depends on the back end choices
- Vectorization generally does these element wise applications but there are also reductions

Reductions

- Like vectorization operates on a list
- Reduces a list into a scalar by performing some operation

```
def mean(x):
    return sum(a for a in x) / len(x)
```

- This reduction is from a list to a scalar but as we get to more dims we can decide how far we want to reduce things

Numpy Manhattan Distance

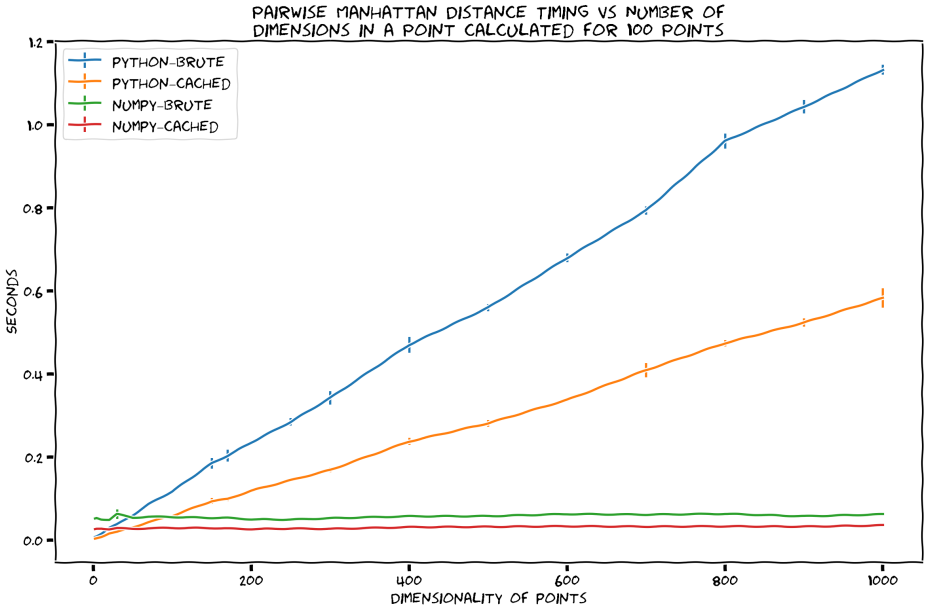
```
np.sum(np.abs(points[i] - points[j]))
```

- points i and j are M dimensional points
- The subtraction and the `np.abs` is a vectorized function
- The `np.sum` is a reduction

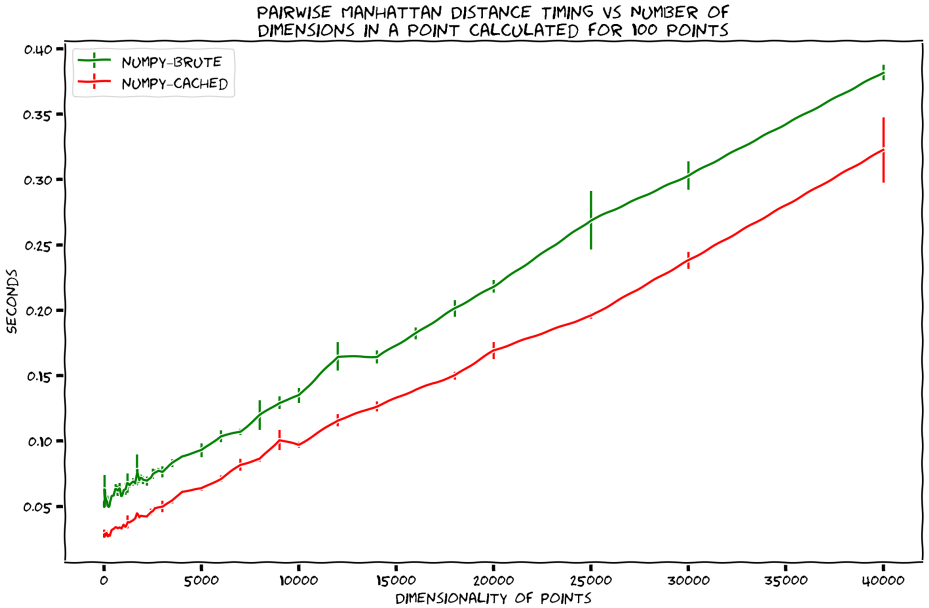
Numpy Cached

```python
def pairwise_manhattan_numpy_v2(
    points: List[np.ndarray]
) -> np.ndarray:
    results = np.zeros((len(points), len(points)), dtype=np.int32)
    for i in range(len(points)):
        for j in range(i, len(points)):
            dist = np.sum(np.abs(points[i] - points[j]))
            results[i, j] = dist
            results[j, i] = dist
    return results
```

- Then only thing we are vectorizing here is the manhattan distance calculation
- We expect this to be about the same speed as the python v2 as we grow the number of points
- We expect this to show speed gains as we increase the number of dimensions



- This is a plot of the timing vs the number of dimentions in a point.
- Graphs created by running each implementation on randomly generated data (same data across implemetations) 5 times. The point it the mean of the times and the error bar is the standard dev of the times.
- For the `python_brute` and `python_cached` implementations we can see the predicted $\mathrm{O}(M)$ growth.
- At small input sizes we can see there is some overhead for converting lists into numpy that counteracts the advangate of vectorization



- Here we can see the same $\mathrm{O}(M)$ growth for numpy implemenetations where we have vectorized the manhattan distance
- This highlights how we are making computational improvments, not algorithmic ones
- I had to scale out to a lot more points to see this trend because when python code is in the graph it looks flat
- Note the very different scale on the y-axis

Numpy Broadcast

```python
def pairwise_manhattan_numpy_v3(points: np.ndarray) -> np.ndarray:
    results = []
    for i in range(len(points)):
        results.append(np.sum(np.abs(points[i] - points), axis=1))
    return np.stack(results)
```

- We can see that one of the loops is gone!
- If you guessed the answer is vectorization you are right
- This is specifically the idea of broadcasting which help you vectorize code

Broadcasting

- Vectorization can speed up code

Broadcasting

- Vectorization can speed up code
- But what do I do when the shapes don't match?

Broadcasting

- Vectorization can speed up code
- But what do I do when the shapes don't match?
- If i want to add a scalar to a vector do I have to copy the scalar into a vector to vectorize it?

- Copying would be a waste of memory (need to have the scalar multiple times) and computation (time taken to make the copies of the scalar)

Broadcasting

- Vectorization can speed up code
- But what do I do when the shapes don't match?
- If i want to add a scalar to a vector do I have to copy the scalar into a vector to vectorize it?
- Broadcasting to the rescue!

- Copying would be a waste of memory (need to have the scalar multiple times) and computation (time taken to make the copies of the scalar)

Roll Your Own Broadcasting

```
def add_scalar_to_vector(scalar, vector):
    return [v + scalar for v in vector]
```

- The scalar is added to each element in the vector as if the scalar was expanded to a vector and a normal vectorized add was done.

Broadcasting Rules

1. If there is a mismatch in the number of dimensions the shape with fewer is padded with 1s on its leading side.
   - $2 \times 3 + 3 \rightarrow 2 \times 3 + 1 \times 3$
2. If the size of some dimension doesn't match and one of the sizes is 1 it is repeated to match the other size.
   - $5 \times 4 + 5 \times 1 \rightarrow 5 \times 4 + 5 \times 4$
3. If there is a mismatch and neither size is 1 an exception is thrown.
   - $3 \times 4 + 4 \times$ $\rightarrow$ ERROR

Broadcasting Examples

$$X \in \mathbb{R}^{2\times 3} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad Y \in \mathbb{R}^{1\times 3} \quad \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \rightarrow \qquad Y' \in \mathbb{R}^{2\times 3} \quad \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix} \qquad X + Y \in R^{2\times 3} \quad \begin{bmatrix} 2 & 4 & 4 \\ 5 & 7 & 7 \end{bmatrix}$$

$$X \in \mathbb{R}^{2\times 3} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad Z \in \mathbb{R}^{2\times 1} \quad \begin{bmatrix} 2 \\ 3 \end{bmatrix} \rightarrow \qquad Z' \in \mathbb{R}^{2\times 3} \quad \begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} \qquad X + Z \in R^{2\times 3} \quad \begin{bmatrix} 3 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}$$

- Here we see examples of broadcasting
- The first one is broadcasting over the rows
- The second one is broadcasting over the columns

Numpy Broadcast

```python
def pairwise_manhattan_numpy_v3(points: np.ndarray) -> np.ndarray:
    results = []
    for i in range(len(points)):
        results.append(np.sum(np.abs(points[i] - points), axis=1))
    return np.stack(results)
```

- points i is $\in \mathbb{R}^M$
- points is $\in \mathbb{R}^{N \times M}$
- The result of broadcasting vectorization for points is `[N, M]`
- There is this new kwarg for `np.sum`, `axis`
- This is a way to apply reductions to multi dimensional inputs

Multidimensional Reduction

```
y = np.sum(x)
```

$x \in \mathbb{R}^{4 \times 3}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$y \in \mathbb{R}$

$$78$$

- We said before that a reduction turned a vector into a scalar, but what about a high dim?
- Without an axis to reduces the whole things, results in 78

Reduction Along an Axis

```
y = np.sum(x, axis=1)
```

$x \in \mathbb{R}^{4 \times 3}$

$$\xrightarrow{\qquad\qquad}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$y \in \mathbb{R}^4$

$$\begin{bmatrix} 6 \\ 15 \\ 24 \\ 33 \end{bmatrix}$$

- The axis argument allows for a reduction that only reduces the number of dims by one
- Reduction over the axis 1 is applied across the columns for matrices resulting in a vector

Reduction Along an Axis

```
y = np.sum(x, axis=0)
```

$x \in \mathbb{R}^{4 \times 3}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$y \in \mathbb{R}^3$

$$\begin{bmatrix} 22 & 26 & 30 \end{bmatrix}$$

- Reduction over axis 0 is applied across the rows for matrices
- There is also a `keepdims` argument to keep the sinlge dimension of size 1 so instead of getting a vector of size 3 you get one of size `[1, 3]`

Numpy Broadcast

```python
def pairwise_manhattan_numpy_v3(points: np.ndarray) -> np.ndarray:
    results = []
    for i in range(len(points)):
        results.append(np.sum(np.abs(points[i] - points), axis=1))
    return np.stack(results)
```

- Reducing `[N, M]` over axis 1 gives us `N`
- Results is a list of vectors of size N
- `np.stack` puts them on top of each other to form a single matrix
- Side Note: Type hints for array shapes has no support which sucks. To make broadcasting work you often need specific shapes but you can't communicate these constraints via type hints

Numpy Broadcast

```python
def pairwise_manhattan_numpy_v3(points: np.ndarray) -> np.ndarray:
    results = []
    for i in range(len(points)):
        results.append(np.sum(np.abs(points[i] - points), axis=1))
    return np.stack(results)
```

- One thing to notice here is that

Numpy Double Broadcast

```python
def pairwise_manhattan_numpy_v4(points: np.ndarray) -> np.ndarray:
    exp_points = np.expand_dims(points, 1)  # [N, 1, M]
    return np.sum(np.abs(exp_points - points), axis=-1)
```

- All the loops are gone!
- We add a dimension of size 1 into the points
- When we do the subtraction the points `[N, M]` is expanded to `[1, N, M]`
- `[N, 1, M]` + `[1, N, M]` $\rightarrow$ `[N, N, M]`
- Reduction over axis -1 (same as negative indexing in python, sum over the last dim)
- Reduction gives us `[N, N]` aka our answer

Numpy Double Broadcast

```python
def pairwise_manhattan_numpy_v4(points: np.ndarray) -> np.ndarray:
    exp_points = np.expand_dims(points, 1)  # [N, 1, M]
    return np.sum(np.abs(exp_points - points), axis=-1)
```

- Now that we are double broadcasting there is no way to express the fact that we can skip the repeated work of calculating the distance between flipped points.

Numpy Double Broadcast

```python
def pairwise_manhattan_numpy_v4(points: np.ndarray) -> np.ndarray:
    exp_points = np.expand_dims(points, 1)  # [N, 1, M]
    return np.sum(np.abs(exp_points - points), axis=-1)
```

- Still $\mathrm{O}(N^2M)$ in runtime
- Now $\mathrm{O}(N^2M)$ in memory

- We are going to revisit Big-O for this implementation
- The runtime is still the same, remember the loops are still there, they are just in C not python
- We have this itermediate result of size `[N, N, M]` so our memory usage is higher
- We'll see that this extra memory usage can cause problems later.

Cython Implementation

```
@cython.wraparound(False)
@cython.boundscheck(False)
cpdef int[:, :] pairwise_manhattan(int[:, :] points):
    cdef int n = points.shape[0]
    cdef int m = points.shape[1]
    results = np.zeros((n, n), dtype=np.int32)
    cdef int[:, :] results_view = results
    cdef int i, j, k, dist
    for i in range(n):
        for j in range(i, n):
            dist = 0
            for k in range(m):
                dist += abs(points[i, k] - points[j, k])
            results_view[i, j] = dist
            results_view[j, i] = dist
    return results
```

- The cython code looks a lot like our original python code
- we can just write loops, we don't have to think about vectorization
- There are a lot of annotations that help speed things up.

Cython Decorators

```
@cython.wraparound(False)
@cython.boundscheck(False)
```

- These are directives that tell cython how to compile code
- `wrapsround` turns off the ability to do negative indexing
- `boundscheck` turns off rasing an error when you access past the end of an array
- Bpth of these ease of use thing require the length of the array. C arrays don't carry their lengths so we need to ask the python object for the length. Accessing python objects are slow and require the GIL

Cython Function Definition

```
cpdef int[:, :] pairwise_manhattan(int[:, :] points):
```

- `cpdef` is a utility to make this function callable by python code and other cython code. C function calls are faster and there is slight overhead from the `cpdef`. Cython in a tight inner loop (like a function of manhattan distance) should use `cdef`
- `int[:, :]` before the function name is the return type
- It is also a type of the input (points)
- We will get to what the types are in a second

Cython Variable Declaration

```
cdef int n = points.shape[0]
cdef int m = points.shape[1]
results = np.zeros((n, n), dtype=np.int32)
cdef int[:, :] results_view = results
cdef int i, j, k, dist
```

- `cdef` is used to statically declare variables. This tells the compiler what the type is so it can compile it to C.
- `int[:, :]` is a typed memoryview
- a type memoryview is a look into the memory of an object that supports the buffer interface
- buffer interface is way that objects will organize their memory layout so other things can access the content. NumPy arrays support this interface
- results is actually a dynamically created numpy array, it it known to the python interpreter, the gil is needed to make it, the ref counter is incremented by one when we make it, and result will be safe to use in python functions after we erturn it. Making it so easy to create python objects is a strength of cython
- results view is a typed memory view of the underlying memory of results, this lets us access from C, without gil

Cython Manhattan Distance

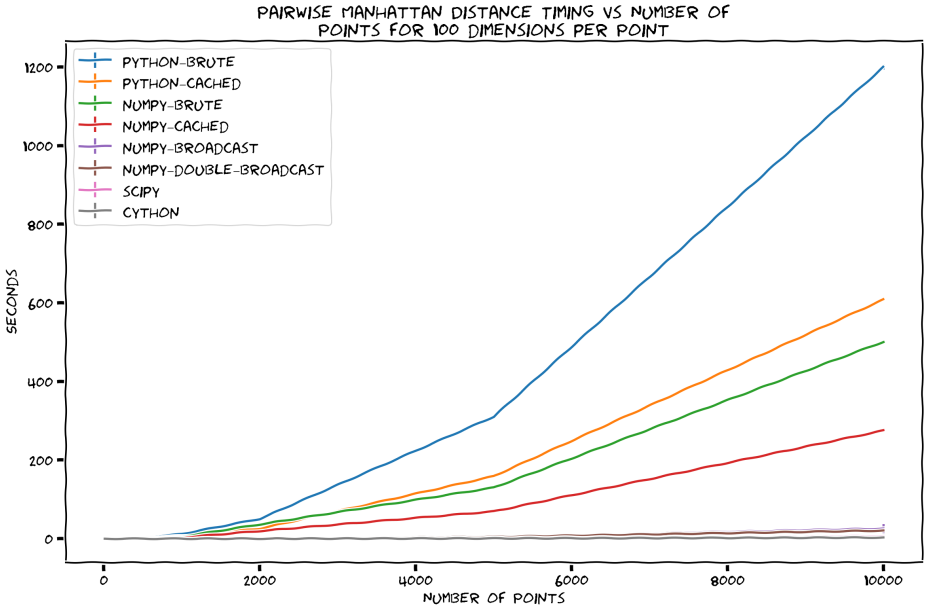
```
from libc.stdlib cimport abs
cimport cython

@cython.wraparound(False)
@cython.boundscheck(False)
cdef inline int manhattan_distance(int[:] x, int[:] y):
    cdef int i
    cdef int m = x.shape[0]
    cdef int dist = 0
    for i in range(m):
        dist += abs(x[i] - y[i])
    return dist
```

- I didn't do this in the code so it looks like the others but I would if this was real production code.
- cdef means this is only callable from cython but it has fast C function calls instead of slow python calls
- the inline keyword means when this is complied this code is directly insearted, not called as a function
- This gives us the speed of having code inline but the reusablilty/nice SWE practice of having a seperate function for it.
- We import abs from the C standard library so we always use the C version, if not cython will swap the version (C vs python) based on the datatypes and we wouldn't be able to release the GIL

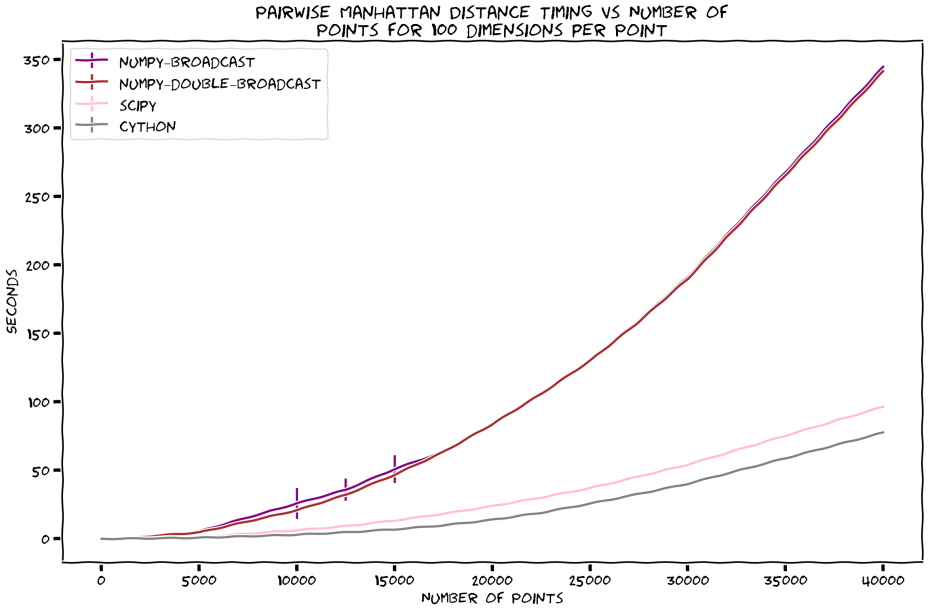
Cython Implementation

```
@cython.wraparound(False)
@cython.boundscheck(False)
cpdef int[:, :] pairwise_manhattan(int[:, :] points):
    cdef int n = points.shape[0]
    cdef int m = points.shape[1]
    results = np.zeros((n, n), dtype=np.int32)
    cdef int[:, :] results_view = results
    cdef int i, j, k, dist
    for i in range(n):
        for j in range(i, n):
            dist = 0
            for k in range(m):
                dist += abs(points[i, k] - points[j, k])
            results_view[i, j] = dist
            results_view[j, i] = dist
    return results
```

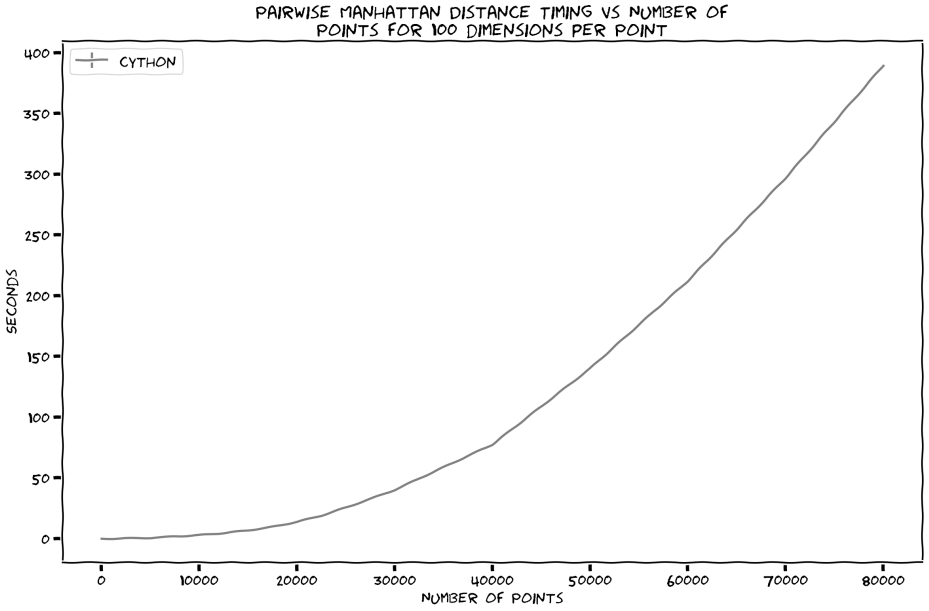
- There are a lot of optimizations left of the table, for example operating on batch points at a time in the inner loops for better cache coherence
- we could skip computing the distance to ourselves because we know it is zero
- These kind of optimizations are very hard (impossible) to express with NumPy
- We didn't use it but Cython also supports defining objects and structs which makes it amenable to speeding up code that isn't all numeric arrays
- NumPy is basically array only if you want speed.



- Here we can see the timing curves of various implemetations as we scale up the number of points.
- We can see the $\mathrm{O}(N^2M)$ growth for our algos
- The numpy broadcasting, scipy, and cython solutions are hard to see, we'll zoom in
- SciPy is a set of more specific/advanced scientific functions that includes a manhattan distance function



- Here we are looking at only some implementations
  - `numpy_broadcast`
  - `numpy_double_broadcast`
  - `scipy`
  - `cython`
- We can start to see the $\mathrm{O}(N^2M)$ growth



- Finally we are scaling out to a lot of points.
- `cython` only
- the $\mathrm{O}(N^2M)$ growth is obvious
- We are at 8 times as many points but only a thrid of the runtime as the python impl

Interactive Cython

- Cython is a compilied Language
- This makes rapid iteration harder than in python
- Jupyter notebooks have nice cython features to speed up exploratory work
- Notebook: Jupyter Notebook

- Cython C extensions are normally built via the `setup.py` file
- A change in cython code isn't reflected until you recompile it
- I am not a huge fan of jupyter notebooks but they can compile cython code for to allow for quick iteration
- Show the notebook

Conclusion

- Vectorization and Broadcasting in NumPy lets us push loops from python into C
- Cython lets us write fast loops and is useful for speeding up difficult to vectorize code
- If you have a custom use case cython can often beat the more general NumPy solution

Links

- Notebook: https://nbviewer.jupyter.org/github/blester125/MIPy-Talk-Jan-2-2020/blob/master/scripts/cython-jupyter-example.ipynb
- Slides: https://github.com/blester125/MIPy-Talk-Jan-2-2020/blob/master/slides/slides.pdf
- Twitter: https://twitter.com/BrianLester125
- GitHub: https://github.com/blester125

- The link to the slides is in a GitHub repo that has all these implementations, code to reproduce the graphs, and a docker container that has optimized versions of NumPy installed.