Matrix Programming

Recommendations

NumPy is a numerical library in Python

- Provides matrices and tools to manipulate them
- Plus a large library of linear algebra operations

Create a complete program

What papers should scientists read?

One good answer is "what their colleagues read"

- Input: scientists' ratings of papers
- Output: what they should read

Based on example from Segaran's

Programming Collective Intelligence

What criteria can we use to recommend papers?

1. The way the paper was rated by other people.

2. The similarity between those raters and the previous ratings for an individual.

Plan:

1. Process people's ratings of various papers and store in NumPy array
2. Introduce two similarity measures
3. Generate recommendations

Input is triples: person, paper, score

This is (very) *sparse* data

So store it in a dictionary

```
raw_scores = {
  'Bhargan Basepair' : {
    'Jackson 1999' : 2.5,
    'Chen 2002' : 3.5,
  },
  'Fan Fullerene' : {
    'Jackson 1999' : 3.0,
    ...
```

Should use DOI or some other unique ID

Turn this dictionary into a dense array (SciPy contains sparse arrays for large data sets)

```
raw_scores = {
  'Bhargan Basepair' : {
    'Jackson 1999' : 2.5,
    'Chen 2002' : 3.5,
  },
  'Fan Fullerene' : {
    'Jackson 1999' : 3.0,
    ...
```

	Jackson 1999	Chen 2002	...
Bhargan Basepair	2.5	3.5	
Fan Fullerene	3.0		
...			


```
def prep_data(all_scores):
  # Names of all people in alphabetical order.
  people = all_scores.keys()
  people.sort()

  # Names of all papers in alphabetical order.
  papers = set()
  for person in people:
    for title in all_scores[person].keys():
      papers.add(title)
  papers = list(papers)
  papers.sort()
```

```python
def prep_data(scores):
  ...
  # Create and fill array.
ratings = numpy.zeros((len(people), len(papers)))
  for (person_id, person) in enumerate(people):
    for (title_id, title) in enumerate(papers):
      r = scores[person].get(title, 0)
      ratings[person_id, title_id] = float(r)

  return people, papers, ratings
```


Next step is to compare sets of ratings

Many ways to do this

We will consider:

– Inverse sums of squares

– Pearson correlation coefficient

Remember: 0 in matrix means "no rating"

Doesn't make sense to compare ratings unless both people have read the paper

Limit our metrics by *masking* the array

Inverse sum of squares uses the distance between N-dimensional vectors

In 2D, this is:

$$\text{distance}^2 = (x_A-x_B)^2 + (y_A-y_B)^2$$

Define similarity as:

$$\text{sim}(A, B) = 1 / (1 + \text{norm}(D))$$

where D is the vector of differences between the papers that both have rated

No diffs => sim(A, B) = 1

Many/large diffs => sim(A, B) is small


```
def sim_distance(prefs, left_index, right_index):

  # Where do both people have preferences?
  left_has_prefs  = prefs[left_index,  :] > 0
  right_has_prefs = prefs[right_index, :] > 0
  mask = numpy.logical_and(left_has_prefs,
                           right_has_prefs)

 # Not enough signal.
 if numpy.sum(mask) < EPS:
    return 0
```

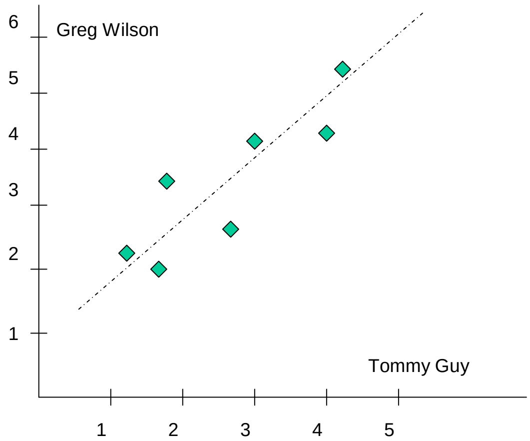
```
def sim_distance(prefs, left_index, right_index):
  ...

  # Return sum-of-squares distance.
  diff = prefs[left_index, mask] –
         prefs[right_index, mask]
  sum_of_squares = numpy.linalg.norm(diff) ** 2
  return 1/(1 + sum_of_squares)
```


What if two people rate many of the same papers but one always rates them lower than the other?

If they *rank* papers the same, but use a different scale, we want to report that they rate papers the same way

Pearson's Correlation reports the correlation between two individuals rather than the absolute difference.

Pearson's Correlation Score measures the error of a best fit line between two individuals.



Each dot is a paper, and its X and Y values correspond to the ratings of each individual.

In this case, the score is high, because the line fits quite well.

To calculate Pearson's Correlation, we need to introduce two quantities:

The *standard deviation* is the divergence from the mean:

$$\text{StDev}(X) = E(X^2) - E(X)^2$$

The *covariance* measures how two variables change together:

$$\text{Cov}(X,Y) = E(XY) - E(X)E(Y)$$

Pearson's Correlation is:

r = Cov(X,Y) / (StdDev(X) * StdDev(Y))

Use NumPy to calculate both terms

If a and b are Nx1 arrays, then `numpy.cov(a,b)` returns an array of results

Variance (a)	Covariance (a,b)
Covariance (a,b)	Variance (b)

Use it to calculate numerator and denominator

```
def sim_pearson(prefs, left_index, right_index):
  # Where do both have ratings?
  rating_left  = prefs[left_index,  :]
  rating_right = prefs[right_index, :]
  mask = numpy.logical_and(rating_left > 0,
                           rating_right > 0)

  # Summing over Booleans gives number of Trues
  num_common = sum(mask)

  # Return zero if there are no common ratings.
  if num_common == 0:
    return 0
```



```
def sim_pearson(prefs, left_index, right_index):
  ...
  # Calculate Pearson score "r"
  varcovar = numpy.cov(rating_left[mask],
                       rating_right[mask])
  numerator = varcovar[0, 1]
  denominator = sqrt(varcovar[0, 0]) *
                sqrt(varcovar[1,1])
  if denominator < EPS:
    return 0
  r = numerator / denominator
  return r
```

Now that we have the scores we can:

1. Find people who rate papers most similarly

2. Find papers that are rated most similarly

3. Recommend papers for individuals based on the rankings of other people and their similarity with this person's previous rankings

To find individuals with the most similar ratings, apply a similarity algorithm to compare each person to every other person

Sort the results to list most similar people first

```python
def top_matches(ratings, person, num,
  similarity):

  scores = []
  for other in range(ratings.shape[0]):
    if other != person:
      s = similarity(ratings, person, other)
      scores.append((s, other))

  scores.sort()
  scores.reverse()
```


Use the same idea to compute papers that are most similar

Since both similarity functions compare rows of the data matrix, we must transpose it

And change names to refer to papers, not people

```python
def similar_items(paper_ids, ratings, num=10):
  result = {}
  ratings_by_paper = ratings.T
  for item in range(ratings_by_paper.shape[0]):
    temp = top_matches(ratings_by_paper,
                       item, num, sim_distance)
    scores = []
    for (score, name) in temp:
      scores.append((score, paper_ids[name]))
    result[paper_ids[item]] = scores
  return result
```


Finally suggest papers based on their rating by people who rated other papers similarly

Recommendation score is the weighted average of paper scores, with weights assigned based on the similarity between individuals

Only recommend papers that have not been rated yet

```
def recommendations(prefs, person_id, similarity):

  totals, sim_sums = {}, {}
  num_people, num_papers = prefs.shape

  for other_id in range(num_people):
    # Don't compare people to themselves.
    if other_id == person_id:
      continue
    sim = similarity(prefs, person_id, other_id)
    if sim < EPS:
      continue
```



```
def recommendations(prefs, person_id, similarity):
  ...
  for other_id in range(num_people):
    ...
    for title in range(num_papers):
      # Only score papers person hasn't seen yet.
      if prefs[person_id, title] < EPS and \
         prefs[other_id, title] > 0:
        if title in totals:
          totals[title] += sim * \
                           prefs[other_id, title]
        else:
          totals[title] = 0
```

```
def recommendations(prefs, person_id, similarity):
  ...
  for other_id in range(num_people):
    ...
    for title in range(num_papers):
      ...
      # Sum of similarities
      if title in sim_sums:
        sim_sums[title] += sim
      else:
        sim_sums[title] = 0
```



```
def recommendations(prefs, person_id, similarity):
  ...
  # Create the normalized list
  rankings = []
  for title, total in totals.items():
    rankings.append((total/sim_sums[title], title))

  # Return the sorted list
  rankings.sort()
  rankings.reverse()
  return rankings
```

Major points:

1. Mathematical operations on matrix were all handled by NumPy

2. We still had to take care of data (re)formatting

created by

Richard T. Guy

November 2010