CHAPTER 8

Recommendation Engines: Building a User-Facing Data Product at Scale

Recommendation engines, also called recommendation systems, are the quintessential data product and are a good starting point when you're explaining to non–data scientists what you do or what data science really is. This is because many people have interacted with recommendation systems when they've been suggested books on Amazon.com or gotten recommended movies on Netflix. Beyond that, however, they likely have not thought much about the engineering and algorithms underlying those recommendations, nor the fact that their behavior when they buy a book or rate a movie is generating data that then feeds back into the recommendation engine and leads to (hopefully) improved recommendations for themselves and other people.

Aside from being a clear example of a product that literally uses data as its fuel, another reason we call recommendation systems "quintessential" is that building a solid recommendation system end-to-end requires an understanding of linear algebra *and* an ability to code; it also illustrates the challenges that Big Data poses when dealing with a problem that makes intuitive sense, but that can get complicated when implementing its solution at scale.

In this chapter, Matt Gattis walks us through what it took for him to build a recommendation system for Hunch.com—including why he made certain decisions, and how he thought about trade-offs between various algorithms when building a large-scale engineering system and infrastructure that powers a user-facing product.

Matt graduated from MIT in CS, worked at SiteAdvisor, and co-founded Hunch as its CTO. Hunch is a website that gives you recommendations of any kind. When they started out, it worked like this: they'd ask people a bunch of questions (people seem to love answering questions), and then someone could ask the engine questions like, "What cell phone should I buy?" or, "Where should I go on a trip?" and it would give them advice. They use machine learning to give better and better advice. Matt's role there was doing the R&D for the underlying recommendation engine.

At first, they focused on trying to make the questions as fun as possible. Then, of course, they saw things needing to be asked that would be extremely informative as well, so they added those. Then they found that they could ask merely 20 questions and then predict the rest of them with 80% accuracy. They were questions that you might imagine and some that are surprising, like whether people were competitive versus uncompetitive, introverted versus extroverted, thinking versus perceiving, etc.—not unlike MBTI.

Eventually Hunch expanded into more of an API model where they crawl the Web for data rather than asking people direct questions. The service can also be used by third parties to personalize content for a given site—a nice business proposition that led to eBay acquiring Hunch.com.

Matt has been building code since he was a kid, so he considers software engineering to be his strong suit. Hunch requires cross-domain experience so he doesn't consider himself a domain expert in any focused way, except for recommendation systems themselves.

The best quote Matt gave us was this: "Forming a data team is kind of like planning a heist." He means that you need people with all sorts of skills, and that one person probably can't do everything by herself. (Think Ocean's Eleven, but sexier.)

A Real-World Recommendation Engine

Recommendation engines are used all the time—what movie would you like, knowing other movies you liked? What book would you like, keeping in mind past purchases? What kind of vacation are you likely to embark on, considering past trips?

There are plenty of different ways to go about building such a model, but they have very similar feels if not implementation. We're going to show you how to do one relatively simple but complete version in this chapter.

To set up a recommendation engine, suppose you have *users*, which form a set U; and you have *items* to recommend, which form a set V. As Kyle Teague told us in Chapter 6, you can denote this as a bipartite graph (shown again in Figure 8-1) if each user and each item has a node to represent it—there are lines from a user to an item if that user has expressed an opinion about that item. Note they might not always *love* that item, so the edges could have weights: they could be positive, negative, or on a continuous scale (or discontinuous, but many-valued like a star system). The implications of this choice can be heavy but we won't delve too deep here—for us they are numeric ratings.

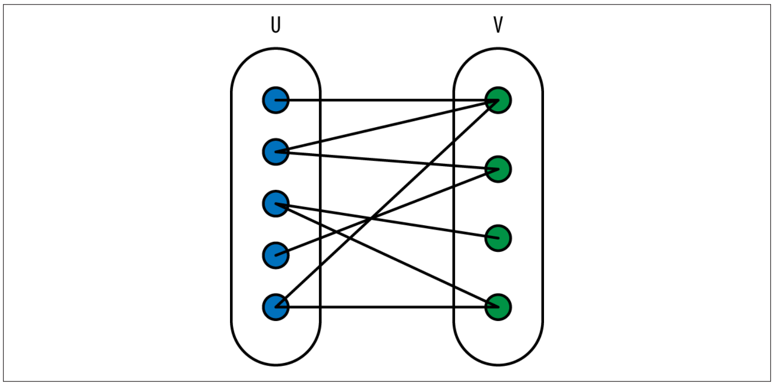

Figure 8-1. Bipartite graph with users and items (television shows) as nodes

Next up, you have training data in the form of some preferences—you know some of the opinions of some of the users on some of the items. From those training data, you want to predict other preferences for your users. That's essentially the output for a recommendation engine.

You may also have metadata on users (i.e., they are male or female, etc.) or on items (the color of the product). For example, users come to your website and set up accounts, so you may know each user's gender, age, and preferences for up to three items.

You represent a given user as a vector of features, sometimes including only metadata—sometimes including only preferences (which would lead to a sparse vector because you don't know all the user's opinions) —and sometimes including both, depending on what you're doing with the vector. Also, you can sometimes bundle all the user vectors together to get a big user matrix, which we call U, through abuse of notation.

Nearest Neighbor Algorithm Review

Let's review the nearest neighbor algorithm (discussed in Chapter 3): if you want to predict whether user A likes something, you look at a user B *closest* to user A who has an opinion, then you assume A's opinion is the same as B's. In other words, once you've identified a similar user, you'd then find something that user A hadn't rated (which you'd assume meant he hadn't ever seen that movie or bought that item), but that user B *had* rated and liked and use that as your recommendation for user A.

As discussed in Chapter 3, to implement this you need a metric so you can measure distance. One example when the opinions are binary: Jaccard distance, i.e., 1–(the number of things they both like divided by the number of things either of them likes). Other examples include cosine similarity or Euclidean distance.

Which Metric Is Best?

You might get a different answer depending on which metric you choose. But that's a good thing. Try out lots of different distance functions and see how your results change and think about why.

Some Problems with Nearest Neighbors

So you *could* use nearest neighbors; it makes some intuitive sense that you'd want to recommend items to people by finding similar people and using those people's opinions to generate ideas and recommendations. But there are a number of problems nearest neighbors poses. Let's go through them:

Curse of dimensionality

There are too many dimensions, so the closest neighbors are too far away from each other to realistically be considered "close."

Overfitting
Overfitting is also a problem. So one guy is closest, but that could be pure noise. How do you adjust for that? One idea is to use k-NN, with, say, k=5 rather than k=1, which increases the noise.

Correlated features
There are tons of features, moreover, that are highly correlated with each other. For example, you might imagine that as you get older you become more conservative. But then counting both age and politics would mean you're double counting a single feature in some sense. This would lead to bad performance, because you're using redundant information and essentially placing double the weight on some variables. It's preferable to build in an understanding of the correlation and project onto smaller dimensional space.

Relative importance of features
Some features are more informative than others. Weighting features may therefore be helpful: maybe your age has nothing to do with your preference for item 1. You'd probably use something like covariances to choose your weights.

Sparseness
If your vector (or matrix, if you put together the vectors) is too sparse, or you have lots of missing data, then most things are unknown, and the Jaccard distance means nothing because there's no overlap.

Measurement errors
There's measurement error (also called reporting error): people may lie.

Computational complexity
There's a calculation cost—computational complexity.

Sensitivity of distance metrics
Euclidean distance also has a scaling problem: distances in age outweigh distances for other features if they're reported as 0 (for don't like) or 1 (for like). Essentially this means that raw euclidean distance doesn't make much sense. Also, old and young people might think one thing but middle-aged people something else. We seem to be assuming a linear relationship, but it may not exist. Should you be binning by age group instead, for example?

Preferences change over time
: User preferences may also change over time, which falls outside the model. For example, at eBay, they might be buying a printer, which makes them only want ink for a short time.

Cost to update
: It's also expensive to update the model as you add more data.

The biggest issues are the first two on the list, namely overfitting and the curse of dimensionality problem. How should you deal with them? Let's think back to a method you're already familiar with—linear regression—and build up from there.

Beyond Nearest Neighbor: Machine Learning Classification

We'll first walk through a simplification of the actual machine learning algorithm for this—namely we'll build a separate linear regression model for each item. With each model, we could then predict for a given user, knowing their attributes, whether they would like the item corresponding to that model. So one model might be for predicting whether you like *Mad Men* and another model might be for predicting whether you would like Bob Dylan.

Denote by $f_{i,j}$ user i's stated preference for item j if you have it (or user i's attribute, if item j is a metadata item like age or is_logged_in). This is a subtle point but can get a bit confusing if you don't internalize this: you are treating metadata here *also* as if it's an "item." We mentioned this before, but it's OK if you didn't get it—hopefully it will click more now. When we said we could predict what you might like, we're also saying we could use this to predict your *attribute*; i.e., if we didn't *know* if you were a male/female because that was missing data or we had never asked you, we might be able to predict that.

To let this idea settle even more, assume we have three numeric attributes for each user, so we have $f_{i,1}$, $f_{i,2}$, and $f_{i,3}$. Then to guess user i's preference on a new item (we temporarily denote this estimate by p_i) we can look for the best choice of β_k so that:

$$p_i = \beta_1 f_{1,i} + \beta_2 f_{2,i} + \beta_3 f_{3,i} + \epsilon$$

The good news: You know how to estimate the coefficients by linear algebra, optimization, and statistical inference: specifically, linear regression.

The bad news: This model only works for one item, and to be complete, you'd need to build as many models as you have items. Moreover, you're not using other items' information at all to create the model for a given item, so you're not leveraging other pieces of information.

But wait, there's more good news: This solves the "weighting of the features" problem we discussed earlier, because linear regression coefficients *are* weights.

Crap, more bad news: overfitting is *still* a problem, and it comes in the form of having huge coefficients when you don't have enough data (i.e., not enough opinions on given items).

> Let's make more rigorous the preceding argument that huge coefficients imply overfitting, or maybe even just a bad model. For example, if two of your variables are exactly the same, or are nearly the same, then the coefficients on one can be 100,000 and the other can be −100,000 and really they add nothing to the model. In general you should always have some *prior* on what a reasonable size would be for your coefficients, which you do by normalizing all of your variables and imagining what an "important" effect would translate to in terms of size of coefficients—anything much larger than that (in an absolute value) would be suspect.

To solve the overfitting problem, you impose a Bayesian prior that these weights shouldn't be too far out of whack—this is done by adding a penalty term for large coefficients. In fact, this ends up being equivalent to adding a prior matrix to the covariance matrix. That solution depends on a single parameter, which is traditionally called λ.

But that begs the question: how do you choose λ? You could do it experimentally: use some data as your training set, evaluate how well you did using particular values of λ, and adjust. That's kind of what happens in real life, although note that it's not exactly consistent with the idea of estimating what a reasonable size would be for your coefficient.

You can't use this penalty term for large coefficients and assume the "weighting of the features" problem is still solved, because in fact you'd be penalizing some coefficients way more than others if they start out on different scales. The easiest way to get around this is to normalize your variables before entering them into the model, similar to how we did it in Chapter 6. If you have some reason to think certain variables should have larger coefficients, then you can normalize different variables with different means and variances. At the end of the day, the way you normalize is again equivalent to imposing a prior.

A final problem with this prior stuff: although the problem will have a unique solution (as in the penalty will have a unique minimum) if you make λ large enough, by that time you may not be solving the problem you care about. Think about it: if you make λ absolutely huge, then the coefficients will all go to zero and you'll have no model at all.

The Dimensionality Problem

OK, so we've tackled the overfitting problem, so now let's think about overdimensionality—i.e., the idea that you might have tens of thousands of items. We typically use both Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) to tackle this, and we'll show you how shortly.

To understand how this works before we dive into the math, let's think about how we reduce dimensions and create "latent features" internally every day. For example, people invent concepts like "coolness," but we can't *directly* measure how cool someone is. Other people exhibit different patterns of behavior, which we internally map or reduce to our one dimension of "coolness." So coolness is an example of a latent feature in that it's unobserved and not measurable directly, and we could think of it as reducing dimensions because perhaps it's a combination of many "features" we've observed about the person and implictly weighted in our mind.

Two things are happening here: the dimensionality is reduced into a single feature and the latent aspect of that feature.

But in this algorithm, we don't decide which latent factors to care about. Instead we let the machines do the work of figuring out what the important latent features are. "Important" in this context means they explain the variance in the answers to the various questions—in other words, they model the answers efficiently.

Our goal is to build a model that has a representation in a low dimensional subspace that gathers "taste information" to generate recommendations. So we're saying here that taste is *latent* but can be approximated by putting together all the observed information we *do* have about the user.

Also consider that most of the time, the rating questions are binary (yes/no). To deal with this, Hunch created a separate variable for every question. They also found that comparison questions may be better at revealing preferences.

Time to Brush Up on Your Linear Algebra if You Haven't Already

A lot of the rest of this chapter likely won't make sense (and we want it to make sense to you!) if you don't know linear algebra and understand the terminology and geometric interpretation of words like *rank* (hint: the linear algebra definition of that word has nothing to do with ranking algorithms), *orthogonal*, *transpose*, *base*, *span*, and *matrix decomposition*. Thinking about data in matrices as points in space, and what it would mean to transform that space or take subspaces can give you insights into your models, why they're breaking, or how to make your code more efficient. This isn't just a mathematical exercise for the sake of it—although there is elegance and beauty in it—it can be the difference between a start-up that fails and a start-up that gets acquired by eBay. We recommend Khan Academy's excellent free online introduction to linear algebra if you need to brush up your linear algebra skills.

Singular Value Decomposition (SVD)

Hopefully we've given you some intuition about what we're going to do. So let's get into the math now starting with singular value decomposition. Given an $m \times n$ matrix X of rank k, it is a theorem from linear algebra that we can always compose it into the product of three matrices as follows:

$$X = USV^{\tau}$$

where U is $m \times k$, S is $k \times k$, and V is $k \times n$, the columns of U and V are pairwise orthogonal, and S is diagonal. Note the standard statement of SVD is slightly more involved and has U and V both square unitary matrices, and has the middle "diagonal" matrix a rectangular. We'll be using this form, because we're going to be taking approximations to X of increasingly smaller rank. You can find the proof of the existence of this form as a step in the proof of existence of the general form here.

Let's apply the preceding matrix decomposition to our situation. X is our original dataset, which has users' ratings of items. We have m users, n items, and k would be the rank of X, and consequently would also be an upper bound on the number d of latent variables we decide to care about—note we choose d whereas m, n, and k are defined through our training dataset. So just like in k-NN, where k is a tuning parameter (different k entirely—not trying to confuse you!), in this case, d is the tuning parameter.

Each row of U corresponds to a *user*, whereas V has a row for each *item*. The values along the diagonal of the square matrix S are called the "singular values." They measure the importance of each latent variable—the most important latent variable has the biggest singular value.

Important Properties of SVD

Because the columns of U and V are orthogonal to each other, you can order the columns by singular values via a base change operation. That way, if you put the columns in decreasing order of their corresponding singular values (which you do), then the dimensions are ordered by importance from highest to lowest. You can take lower rank approximation of X by throwing away part of S. In other words, replace S by a submatrix taken from the upper-left corner of S.

Of course, if you cut off part of S you'd have to simultaneously cut off part of U and part of V, but this is OK because you're cutting off the *least important* vectors. This is essentially how you choose the number of latent variables d—you no longer have the original matrix X anymore, only an approximation of it, because d is typically much smaller than k, but it's still pretty close to X. This is what people mean when they talk about "compression," if you've ever heard that term thrown around. There is often an important interpretation to the values in the matrices U and V. For example, you can see, by using SVD, that the "most important" latent feature is often something like whether someone is a man or a woman.

How would you actually use this for recommendation? You'd take X, fill in all of the empty cells with the average rating for that item (you don't want to fill it in with 0 because that might mean something in the rating system, and SVD can't handle missing values), and then compute the SVD. Now that you've decomposed it this way, it means that you've captured latent features that you can use to compare users if you want to. But that's not what you want—you want a prediction. If you multiply out the U, S, and V^{τ} together, you get an approximation to X—or a prediction, $\hat{X}$—so you can predict a rating by simply looking up the entry for the appropriate user/item pair in the matrix $\hat{X}$.

Going back to our original list of issues with nearest neighbors in "Some Problems with Nearest Neighbors" on page 202, we want to avoid the problem of missing data, but that is not fixed by the preceding SVD approach, nor is the computational complexity problem. In fact, SVD is extremely computationally expensive. So let's see how we can improve on that.

Principal Component Analysis (PCA)

Let's look at another approach for predicting preferences. With this approach, you're still looking for U and V as before, but you don't need S anymore, so you're just searching for U and V such that:

$$X \equiv U \cdot V^{\tau}$$

Your optimization problem is that you want to minimize the discrepency between the actual X and your approximation to X via U and V measured via the squared error:

$$argmin \Sigma_{i,j} \left(x_{i,j} - u_i \cdot v_j\right)^2$$

Here you denote by u_i the row of U corresponding to user i, and similarly you denote by v_j the row of V corresponding to item j. As usual, items can include metadata information (so the age vectors of all the users will be a row in V).

Then the dot product $u_i \cdot v_j$ is the *predicted preference* of user *i* for item *j*, and you want that to be as close as possible to the *actual preference* $x_{i,j}$.

So, you want to find the best choices of U and V that minimize the squared differences between prediction and observation on everything you actually know, and the idea is that if it's really good on stuff you know, it will also be good on stuff you're guessing. This should sound familiar to you—it's mean squared error, like we used for linear regression.

Now you get to choose a parameter, namely the number *d* defined as *how may latent features you want to use*. The matrix U will have a row for each user and a column for each latent feature, and the matrix V will have a row for each item and a column for each latent feature.

How do you choose *d*? It's typically about 100, because it's more than 20 (as we told you, through the course of developing the product, we found that we had a pretty good grasp on someone if we ask them 20 questions) and it's as much as you care to add before it's computationally too much work.

The resulting latent features are the basis of a well-defined subspace of the total *n*-dimensional space of potential latent variables. There's no reason to think this solution is unique if there are a bunch of missing values in your "answer" matrix. But that doesn't necessarily matter, because you're just looking for *a solution*.

Theorem: The resulting latent features will be uncorrelated

We already discussed that correlation was an issue with k-NN, and who wants to have redundant information going into their model? So a nice aspect of these latent features is that they're uncorrelated. Here's a sketch of the proof:

Say we've found matrices U and V with a fixed product $U \cdot V = X$ such that the squared error term is minimized. The next step is to find the best U and V such that their entries are small—actually we're minimizing the sum of the squares of the entries of U and V. But we can modify U with any invertible $d \times d$ matrix G as long as we modify V by its inverse: $U \cdot V = (U \cdot G) \cdot \left(G^{-1} \cdot V\right) = X$.

Assume for now we only modify with determinant 1 matrices G; i.e., we restrict ourselves to volume-preserving transformations. If we ignore for now the size of the entries of V and concentrate only on the size of the entries of U, we are minimizing the surface area of a d-dimensional parallelepiped in n space (specifically, the one generated by the columns of U) where the volume is fixed. This is achieved by making the sides of the parallelepiped mutually orthogonal, which is the same as saying the latent features will be uncorrelated.

But don't forget, we've ignored V! However, it turns out that V's rows will also be mutually orthogonal when we force U's columns to be. This is not hard to see if you keep in mind X has its SVD as discussed previously. In fact, the SVD and this form $U \cdot V$ have a lot in common, and some people just call this an SVD algorithm, even though it's not quite.

Now we allow modifications with nontrivial determinant—so, for example, let G be some scaled version of the identity matrix. Then if we do a bit of calculus, it turns out that the best choice of scalar (i.e., to minimize the sum of the squares of the entries of U and of V) is in fact the *geometric mean* of those two quantities, which is cool. In other words, we're minimizing the arithmetic mean of them with a single parameter (the scalar) and the answer is the geometric mean.

So that's the proof. Believe us?

Alternating Least Squares

But how do you do this? How do you actually find U and V? In reality, as you will see next, you're not first minimizing the squared error and then minimizing the size of the entries of the matrices U and V. You're actually doing both at the same time.

So your goal is to find U and V by solving the optimization problem described earlier. This optimization doesn't have a nice closed formula like ordinary least squares with one set of coefficients. Instead, you need an iterative algorithm like gradient descent. As long as your problem is convex you'll converge OK (i.e., you won't find yourself at a local, but not global, maximum), and you can force your problem to be convex using regularization.

Here's the algorithm:

- Pick a random V.

- Optimize U while V is fixed.
- Optimize V while U is fixed.
- Keep doing the preceding two steps until you're not changing very much at all. To be precise, you choose an ϵ and if your coefficients are each changing by less than ϵ, then you declare your algorithm "converged."

Theorem with no proof: The preceding algorithm will converge if your prior is large enough

If you enlarge your prior, you make the optimization easier because you're artificially creating a more convex function—on the other hand, if your prior is huge, then all your coefficients will be zero anyway, so that doesn't really get you anywhere. So actually you might not want to enlarge your prior. Optimizing your prior is philosophically screwed because how is it a *prior* if you're back-fitting it to do what you want it to do? Plus you're mixing metaphors here to some extent by searching for a close approximation of X at the same time you are minimizing coefficients. The more you care about coefficients, the less you care about X. But in actuality, you only want to care about X.

Fix V and Update U

The way you do this optimization is user by user. So for user i, you want to find:

$$argmin_{u_i} \Sigma_{j \in P_i} \left(p_{i,j} - u_i{}^{*} v_j \right)^2$$

where v_j is fixed. In other words, you just care about *this user* for now.

But wait a minute, this is the same as linear least squares, and has a closed form solution! In other words, set:

$$u_i = \left(V_{*,i}^{\tau} V_{*,i} \right)^{-1} V_{*,i}^{\tau} P_{*i},$$

where $V_{*,i}$ is the subset of V for which you have preferences coming from user i. Taking the inverse is easy because it's $d \times d$, which is small. And there aren't that many preferences per user, so solving this many times is really not that hard. Overall you've got a doable update for U.

When you fix U and optimize V, it's analogous—you only ever have to consider the users that rated that movie, which may be pretty large for popular movies but on average isn't; but even so, you're only ever inverting a $d \times d$ matrix.

Another cool thing: because each user is only dependent on their item's preferences, you can parallelize this update of U or V. You can run it on as many different machines as you want to make it fast.

Last Thoughts on These Algorithms

There are lots of different versions of this approach, but we hope we gave you a flavor for the trade-offs between these methods and how they can be used to make predictions. Sometimes you need to extend a method to make it work in your particular case.

For example, you can add new users, or new data, and keep optimizing U and V. You can choose which users you think need more updating to save computing time. Or if they have enough ratings, you can decide not to update the rest of them.

As with any machine learning model, you should perform cross-validation for this model—leave out a bit and see how you did, which we've discussed throughout the book. This is a way of testing overfitting problems.

Thought Experiment: Filter Bubbles

What are the implications of using error minimization to predict preferences? How does presentation of recommendations affect the feedback collected?

For example, can you end up in local maxima with rich-get-richer effects? In other words, does showing certain items at the beginning give them an unfair advantage over other things? And so do certain things just get popular or not based on luck?

How do you correct for this?

Exercise: Build Your Own Recommendation System

In Chapter 6, we did some exploratory data analysis on the GetGlue dataset. Now's your opportunity to build a recommendation system with that dataset. The following code isn't for GetGlue, but it is Matt's code to illustrate implementing a recommendation system on a relatively small dataset. Your challenge is to adjust it to work with the GetGlue data.

Sample Code in Python

```
import math,numpy

pu    =    [[(0,0,1),(0,1,22),(0,2,1),(0,3,1),(0,5,0)],[(1,0,1),
(1,1,32),(1,2,0),(1,3,0),(1,4,1),(1,5,0)],[(2,0,0),(2,1,18),
(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),(3,1,40),(3,2,1),
(3,3,0),(3,4,0),(3,5,1)],[(4,0,0),(4,1,40),(4,2,0),(4,4,1),
(4,5,0)],[(5,0,0),(5,1,25),(5,2,1),(5,3,1),(5,4,1)]]

pv     =     [[(0,0,1),(0,1,1),(0,2,0),(0,3,1),(0,4,0),(0,5,0)],
[(1,0,22),(1,1,32),(1,2,18),(1,3,40),(1,4,40),(1,5,25)],
[(2,0,1),(2,1,0),(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),
(3,1,0),(3,2,1),(3,3,0),(3,5,1)],[(4,1,1),(4,2,0),(4,3,0),
(4,4,1),(4,5,1)],[(5,0,0),(5,1,0),(5,2,1),(5,3,1),(5,4,0)]]

V = numpy.mat([[ 0.15968384,  0.9441198 ,  0.83651085],
               [ 0.73573009,  0.24906915,  0.85338239],
               [ 0.25605814,  0.6990532 ,  0.50900407],
               [ 0.2405843 ,  0.31848888,  0.60233653],
               [ 0.24237479,  0.15293281,  0.22240255],
               [ 0.03943766,  0.19287528,  0.95094265]])

print V

U = numpy.mat(numpy.zeros([6,3]))
L = 0.03

for iter in xrange(5):

    print "\n----- ITER %s -----"%(iter+1)

    print "U"
    urs = []
    for uset in pu:
        vo = []
        pvo = []
        for i,j,p in uset:
            vor = []
            for k in xrange(3):
                vor.append(V[j,k])
            vo.append(vor)
            pvo.append(p)
        vo = numpy.mat(vo)
        ur = numpy.linalg.inv(vo.T*vo +
            L*numpy.mat(numpy.eye(3))) *
            vo.T * numpy.mat(pvo).T
        urs.append(ur.T)
    U = numpy.vstack(urs)
    print U

    print "V"
    vrs = []
    for vset in pv:
        uo = []
        puo = []
        for j,i,p in vset:
            uor = []
            for k in xrange(3):
                uor.append(U[i,k])
            uo.append(uor)
            puo.append(p)
        uo = numpy.mat(uo)
             vr  =  numpy.linalg.inv(uo.T*uo  +  L*numpy.mat(num
py.eye(3))) * uo.T * numpy.mat(puo).T
        vrs.append(vr.T)
    V = numpy.vstack(vrs)
    print V

    err = 0.
    n = 0.
    for uset in pu:
        for i,j,p in uset:
            err += (p - (U[i]*V[j].T)[0,0])**2
            n += 1
    print math.sqrt(err/n)

print
print U*V.T
```