Lecture 6 - Recommender Systems

Review
(1) Why do we use ensembles?
(2) How do you build a decision tree classifier? How are the rules selected

Looking forward
Yesterday we learned how to predict overall whether a movie is good or bad. But how can we predict whether you will like the movie? In other words, how can we recommend content for specific users?

Today
- Recommender problems
- Content-based recommendation
  - Linear regression
  - Regression for content-based recommendation
- Collaborative filtering
  - Matrix factorization
  - Low-rank matrix factorization
  - Learning low-rank factorizations

Recommender problems
A common problem in machine learning is that of content recommendation.
Ex. Netflix, Amazon, etc.

For today we'll use movie recommendations as a running example.

Problem: How can we recommend movies for a specific user based on features of the movie and based on the ratings of other users?

Naive method: Predict movies with the highest average user rating (which you haven't seen)
Problem: Not specific to your taste in movies (maybe you hate popular movies).

Content-based Recommendation

Idea: Use the features of the movie and your movie ratings to build a movie recommender model for you.

How? Regression

After linear regression

Notation

a = user a (i.e. you)

D_a = indexes of movies you have seen (and rated)

$x^{(i)}$ = feature vector for i^{th} movie

$y^{(i)}$ = your rating for the i^{th} movie (1-5 scale)

$S_a = \{(x^{(i)}, y^{(i)}), i \in D_a\}$ = set of all movie features and your ratings for all the movies you have seen

Why regression?

We can formulate the content-based recommendation task as a regression problem.

So far, we have only been doing classification where we are trying to predict a discrete label (e.g. +1/-1).

We could solve this problem as a classification problem with 5 classes (1,2,3,4,5), but it's easier to work in the regression setting where we can predict any real # and we try to get as close to the true rating as possible.

Linear regression

So far, we've only been doing classification

With a linear classifier our prediction has been
$h(x; \theta, \theta_0) = \text{sign}(\theta \cdot x + \theta_0)$

? How?

But sometimes we may want to predict a real # rather than just +1 or -1.

We can do this by simply predicting the real # $\theta \cdot x + \theta_0$

$\boxed{\text{Linear regressor: } f(x; \theta, \theta_0) = \theta \cdot x + \theta_0}$

? How do we learn θ and θ_0?

Perceptron won't work because how do we define a mistake?
If the target is 2, is predicting 3 a mistake? what about 2.1?
2.00000...1?

Clearly the process for learning a linear regressor model is going to be different from the process of learning a linear classifier model.

Idea: Use loss functions

For now ignore θ_0

Remember: A loss function is a function $J(\theta)$ which gives a quantitative measure of how far away we are from a perfect model. Our goal is to minimize the loss function

Linear regression objective: minimize the difference between the actual and the predicted value according to a loss function

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}(y^{(i)} - \theta \cdot x^{(i)})$$

$J(\theta)$ is the average loss across all training points

In linear regression, we typically use squared loss:
$\text{Loss}_s(z) = \frac{z^2}{2}$

$$\boxed{J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{(y^{(i)} - \theta \cdot x^{(i)})^2}{2}}$$

Now we can learn θ simply by choosing the θ which minimizes $J(\theta)$. This can be done with gradient descent.

Problem: If the size of the training set is small, it may be difficult to learn all the coordinates of θ.

Idea: If our model struggles to learn coordinates of θ, we should give it something to default to.

Specifically, we want the model to default to 0 when it's not sure.

How? Regularization.

This is similar to controlling the margins in SVM.

$$\boxed{J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{(y^{(i)} - \theta \cdot x^{(i)})^2}{2} + \frac{\lambda}{2} \|\theta\|^2}$$

Since we want to minimize $J(\theta)$ coordinates of theta will tend to 0 if not otherwise informed by the training data.

This builds a regressor which generalizes better.

<u>Content based recommendations using linear regression</u>
Notation (from page 6-2)

Directly apply regression to movies we have seen:

$$\boxed{J(\theta) = \sum_{i \in D_a} \frac{(y^{(i)} - \theta \cdot x^{(i)})^2}{2} + \frac{\lambda}{2} \|\theta\|^2}$$

Optimize with gradient descent

Essentially the regressor builds a θ which learns how to use the features of movies you have seen to predict movies with similar features.

Ex. If you like lots of big budget action movies, the model will learn to predict high ratings for other big budget action movies.

Collaborative filtering
This is great, but how can we do better?
We have a huge number of other users who have rated movies, so how can we make use of this information in addition to the information from movie features.

Collaborative filtering makes use of both movie features and other users' ratings.

There are multiple methods for performing collaborative filtering. The one we will study is called low-rank matrix factorization.

Matrix factorization
Represent movie ratings as a matrix.
- rows are users
- columns are movies

Example

$$\begin{array}{cccc} \text{movie } 1 & \text{movie } 2 & \text{movie } 3 & \\ \end{array}$$

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix} \begin{array}{l} \text{user } 1 \\ \text{user } 2 \end{array}$$

In reality, the matrix will be massive and will mostly be ?'s because most users have not seen or rated most movies.

Goal: Find a matrix X with predictions for every user's rating of every movie.

Naive Solution

What if we do something similar to linear regression?

We want to ensure that for ratings we do know, our predictions are similar, and we want some sort of regularization to enforce generalizability.

Let D be the set of all ratings in Y that we know.
Then the known ratings are $\{Y_{ai}, (a,i) \in D\}$
a = user, i = movie

Based on the intuition above, our objective function would be:

$$J(\theta) = \sum_{ai \in D} \frac{(Y_{ai} - X_{ai})^2}{2} + \frac{\lambda}{2} \sum_{ai} X_{ai}^2$$

(?) Does this work? If not, why not? What happens?

→ For ratings Y_{ai} that we know, we set $X_{ai} = Y_{ai}$.
For ratings we don't know, the regularization forces $X_{ai} = 0$

So we predict a rating of 0 for all movies each user hasn't seen.

not exactly b/c of λ, but close enough

$$Y = \begin{bmatrix} 5 & ? & ? \\ 1 & 2 & ? \end{bmatrix} \rightarrow X \begin{bmatrix} 5 & 0 & ? \\ 1 & 2 & 0 \end{bmatrix}$$

This is terrible!

actually $\frac{1}{1+\lambda} Y_{ai}$

(?) How can we fix this?

Low-rank factorization

Problem: X has too much freedom to take on bad solutions.
Idea: Constrain the possible matrices that X could be.

Matrix factorization: Require that $X = UV^T$ for matrices U, V

Example

$$\begin{bmatrix} 5 & 7 \\ 10 & 14 \\ 30 & 42 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 6 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \end{bmatrix}$$

Problem: If U and V can be any size, then X is unconstrained because it can still take on any values.

Idea: Require that U and V have low rank (are small).
This constrains the possible products $X = UV^T$.

Rank 1 Case - most constrained case
U and V have just one column and are vectors rather than matrices.

$$X = UV^T = \begin{bmatrix} u^{(1)} \\ u^{(2)} \\ \vdots \\ u^{(n)} \end{bmatrix} \times \begin{bmatrix} v^{(1)} & v^{(2)} & \dots & v^{(m)} \end{bmatrix}$$

? What is the prediction for the rating that user a gives to movie i?
i.e. What is X_{ai}?

$X_{ai} = u^{(a)} \cdot v^{(i)}$

So we can see that $u^{(a)}$ is associated with the a^{th} user and $v^{(i)}$ is associated with the i^{th} movie.

$v^{(i)}$ will represent the overall rating of the movie
$u^{(a)}$ will represent how much user a agrees with the overall ratings of movies

Rank k case
In general we can let U and V have k columns rather than just one column.
But we keep k small to maintain the constraint.

6-8

$$X = n \begin{array}{c} \quad k \\ \begin{bmatrix} -u^{(1)}- \\ -u^{(2)}- \\ \vdots \\ -u^{(n)}- \end{bmatrix} \end{array} \times k \begin{array}{c} \quad m \\ \begin{bmatrix} | & | & & | \\ v^{(1)} & v^{(2)} & \cdots & v^{(m)} \\ | & | & & | \end{bmatrix} \end{array} \qquad k << n, m$$

The rows of U and V are now vectors rather than scalars.
$u^{(a)}$ is a length k vector representing preferences for user a
$v^{(i)}$ is a length k vector representing features of movie i
$X_{ai} = u^{(a)} \cdot v^{(i)}$ = predicted rating = how well preferences of user align with features of movie

now this is a dot product of vectors

Learning low-rank factorizations
Goal is to learn X, which we will do by learning U and V.

Now instead of writing the loss function in terms of X, we write it in terms of U and V.

$$\boxed{J(U,V) = \underbrace{\sum_{(a,i)\in D} \frac{(Y_{ai} - \overbrace{[UV^T]_{ai}}^{X_{ai}})^2}{2}}_{\text{enforces accurate predictions for known ratings}} + \underbrace{\frac{\lambda}{2}\sum_{a=1}^{n}\sum_{j=1}^{k} U_{aj}^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\sum_{j=1}^{k} V_{ij}^2}_{\text{regularization to help generalize}}}$$

Looks similar to naive solution, but low-rank requirement constrains $X = UV^T$ and prevents bad solutions.

Goal: Find U, V which minimizes $J(U,V)$.

7 How?

Problem: It's hard to optimize U and V simultaneously
Idea: It's easy to optimize one at a time.
Solution: Alternate between optimizing U and V.

If we assume that V is fixed, then we can easily optimize for U and vice versa. In fact, if you examine $J(U,V)$ closely, you'll see that each vector $u^{(a)}$ and $v^{(i)}$ are independent and can be optimized separately.

Procedure

1) Initialize the movie feature vectors $v^{(1)}, v^{(2)}, \ldots, v^{(m)}$ randomly.
2) Fix $v^{(1)}, v^{(2)}, \ldots, v^{(m)}$ and separately optimize $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$ by minimizing

$$\sum_{i:(a,i)\in D} \frac{(Y_{ai} - u^{(a)} \cdot v^{(i)})^2}{2} + \frac{\lambda}{2} \|u^{(a)}\|^2$$

The minimization can be done by computing the derivative with respect to $u^{(a)}$, setting it equal to 0, and solving for $u^{(a)}$.

3) Fix $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$ and separately optimize $v^{(1)}, v^{(2)}, \ldots, v^{(m)}$ by minimizing

$$\sum_{a:(a,i)\in D} \frac{(Y_{ai} - u^{(a)} \cdot v^{(i)})^2}{2} + \frac{\lambda}{2} \|v^{(i)}\|^2$$

4) Repeat steps 2 and 3 several times.