Recommender Systems

ref. from book "**Data Science from Scratch**", Chap 23

```
begin
    using Test
    using Random
    using PlutoUI
    using LinearAlgebra
    using CSV
    using StringEncodings
    using Printf
    using Plots

    push!(LOAD_PATH, "./src")
    using YaWorkingData: DT, pca, transform
```

TOC

- Recommending What is Popular
- User-Based Collaborative Filtering
- Item-Based Collaborative Filtering
- Matrix Factorization

back to TOC

Recommending What is Popular

```
Vector{String}[String["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "(
```

```
const Users_Interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
```

```
    ["libsvm", "regression", "support vector machines"]
```

15

AbstractArray

```
begin
    const VVS = Vector{Vector{S}} where S <: AbstractString
    const VVF = Vector{Vector{Float64}}
    const VS = Vector{String}
    const VP = Vector{Pair{String, Integer}}
    const VT_IF = Vector{Tuple{Integer, Float64}}
    const DSI = Dict{String, Integer}
    const AVT = AbstractArray{T, N} where {T <: Any, N}
```

gen_popular_interests (generic function with 1 method)

```
function gen_popular_interests(ds::VVS)::DSI
  count_word = Dict{String, Integer}()
  for ui_s ∈ ds, w ∈ ui_s
    count_word[w] = get(count_word, w, 0) + 1
  end
  count_word
```

most_popular_new_interests (generic function with 1 method)

```
function most_popular_new_interests(ui::VS, pi::DSI; max_res=5)::VP
    suggestions = filter(((i, _f)=t) -> i ∉ ui,
        sort(collect(pi), by=t -> t[2], rev=true)
    )[1:max_res]
```

pop_interests =

```
Dict("R" ⇒ 4, "MySQL" ⇒ 1, "Hadoop" ⇒ 2, "statsmodels" ⇒ 2, "artificial intell
```

Vector{Vector{String}} (alias for Array{Array{String, 1}, 1})

If you are user 2 (`Users_Interests[2]`) we recommend: `[("R" => 4), ("Python" => 4), ("Java" => 3), ("Big Data" => 3), ("statistics" => 3)]`:

Test Passed

```
begin
    @test most_popular_new_interests(Users_Interests[2], pop_interests) == [
        ("R" => 4), ("Python" => 4), ("Java" => 3), ("Big Data" => 3),
        ("statistics" => 3)
    ]
```

If you are user 4, who is already interested in many of those things, you would get instead `[("HBase" => 3), ("Java" => 3), ("Big Data" => 3), ("Hadoop" => 2), ("statsmodels" => 2)]`:

Test Passed

```julia
@test most_popular_new_interests(Users_Interests[4], pop_interests) == [
    ("HBase" => 3), ("Java" => 3), ("Big Data" => 3), ("Hadoop" => 2),
    ("statsmodels" => 2)
```

back to TOC

User-Based Collaborative Filtering

One way of taking a user's interests into account is to look for users who are somehow similar to him/her, and then suggest the things that those users are interested in.

In order to do that we need to measure how similar two users are and for this we need a "distance". A typical one for this case is the cosine distance. To apply it, we will binary encode the user preference's vector. With this encoding, similar user (which means users with vectors pointing nearly in the same direction of the feature space) will have a similarity close to 1. Conversely, very dissimilar users will a similarity of 0. Otherwise the similarity will fall between 0 and 1.

To achieve this, let us start collecting the known (unique) interests.

```julia
const Uniq_Interests = Iterators.flatten(Users_Interests) |>
    collect |>
    sort |>
```

```
Test Passed
```

```julia
@test Uniq_Interests[1:6] == String["Big Data", "C++", "Cassandra", "HBase",
```

```
(36, 15)
```

```
gen_binary_vect (generic function with 2 methods)
```

```julia
begin

function gen_binary_vect(user_int::VS, uniq_int::VS)::BitArray
    v = BitArray(undef, length(uniq_int))
    for (ix, ui) ∈ enumerate(uniq_int)
        v[ix] = ui ∈ user_int ? 1 : 0
    end
    v
end

gen_binary_vect(user_int::VVS, uniq_int::VS) = gen_binary_vect.(user_int, uniq_int)

```

```
BitVector[BitVector: [true, false, true, true, true, false, true, false, false,
```

```julia
## NOTE: we need to prevent broadcasting on the second array => Ref()
```

```
36
```

Now `user_interest_vect[i][j]` is 1 if user i has interest in j and 0 otherwise.

Let us define the cosine similarity:

```
cosine_similarity (generic function with 1 method)
```

```
function cosine_similarity(v₁::AVT, v₂::AVT)::Float64 # where {T <: Any}
    dot(v₁, v₂) / (norm(v₁) * norm(v₂))
```

```
Test Passed
```

```
begin
    @test cosine_similarity([1., 1, 1], [2., 2, 2]) ≈ 1. ## "same direction"
    @test cosine_similarity([-1., -1], [2., 2]) ≈ -1.    ## "opposite direction"
    @test cosine_similarity([1., 0], [0., 1]) ≈ 0.       ## "orthogonal"
```

Because we have a small dataset, it is not a problem to compute the pairwise similarities between all of our users (*it is symetric*) :

```
User_Similarities =
 Vector{Float64}[Float64[1.0,  0.338062,  0.0,  0.0,  0.0,  0.154303,  0.0,  0.0,  0.188
```

```
# user_similarities = [
#   [cosine_similarity(u_i, u_j) for (i, u_i) ∈ enumerate(user_interest_vect)
#           for (j, u_j) ∈ enumerate(user_interest_vect) if j > i]
# ]

User_Similarities = [
    [cosine_similarity(u_i, u_j) for u_i ∈ User_Interest_Vect]
            for u_j ∈ User_Interest_Vect
```

`user_similarities[i][j]` gives us the similarity between users i and j (i < j)

```
Vector{Vector{Float64}} (alias for Array{Array{Float64, 1}, 1})
```

```
Test Passed
```

```
## Users 1 and 10 share interests in Hadoop, Java, and Big Data
@test 0.56 ≤ User_Similarities[1][10] ≤ 0.58  ## several shared interests
```

```
Test Passed
```

```
## Users 1 and 9 share only one interest: Big Data
@test 0.18 ≤ User_Similarities[1][9] ≤ 0.20  ## "only one shared interest"
```

In particular, `user_similarities[i]` is the vector of user i's similarities to every other user. We can use this to write a function that finds the most similar users to a given user (avoiding including the user himself/herself, nor any users with zero similarity). Then we will sort the results from most similar to least similar:

```
most_similar_users (generic function with 1 method)
```

```julia
function most_similar_users(user_simil::VVF, user_id::Integer)::VT_IF
    type_ = eltype(user_simil[user_id])
    filter(
        ((uid, sim)=t) -> uid ≠ user_id && sim > zero(type_),
        collect(enumerate(user_simil[user_id]))
    ) |> a -> sort(a, by=t -> t[end], rev=true)
```

```
Tuple{Integer, Float64}[(10, 0.566947), (2, 0.338062), (9, 0.188982), (14, 0.169
```

```
Vector{Tuple{Integer, Float64}} (alias for Array{Tuple{Integer, Float64}, 1})
```

How do we use this to suggest new interests to a user?

For each interest we can add up the user similarities of the other iuser interested in it:

```
user_based_suggestions (generic function with 1 method)
```

```julia
function user_based_suggestions(user_id::Integer, user_simil, user_inter::VVS;
        incl_curr_interests=false)
    #
    suggs = Dict{String, Float64}()
    for (o_uid, simil) ∈ most_similar_users(user_simil, user_id),
        inter ∈ user_inter[o_uid]
        suggs[inter] = get(suggs, inter, 0.) + simil
    end

    suggs = sort(collect(suggs), by=t -> t[end], rev=true)  ## Vector of pairs

    incl_curr_interests && (return suggs)

    [(sugg, w) for (sugg, w) ∈ suggs if sugg ∉ user_inter[user_id]]
```

```
Tuple{String, Float64}[("MapReduce", 0.566947), ("Postgres", 0.507093), ("MongoDB"
```

This seems to make sense for someone whose stated interests are "Big Data" and database related.

Note however that this approach does not scale well and in large dimensional vector spaces, vectors are far apart.

back to TOC

Item-Based Collaborative Filtering

An alternative approach is to compute similarities between interests directly. We can then generate suggestions for each user by aggregating interests that are similar to his/her current interests.

First let us transpose our user*interest*vect (a list of column vector) into a matrix of row vector.

```
((15), Vector{BitVector} (alias for Array{BitArray{1}, 1}))
```

```
Matrix{Bool}[1×15 Matrix{Bool}:                        ,  1×15 Matrix{Bool}:
               1  0  0  0  0  0  0  0  1  1  0  0  0  0  0    0  0  0  0  0  1  0  0
```

```
(36)
```

```
Test Passed
```

```
@test Interest_User_Matrix[1] == [1 0 0 0 0 0 0 0 1 1 0 0 0 0 0]
```

With this we can now use the cosine similarity again. If same users are interested in two topics, their similarity will be 1 and conversely when two users are interested in different topices tehir similarity will be 0.

```
Vector{Float64}[Float64[1.0,  0.0,  0.408248,  0.333333,  0.816497,  0.0,  0.666667,  0
```

```
const Interest_Similarities = [
    [cosine_similarity(uvectᵢ, uvectⱼ) for uvectⱼ ∈ Interest_User_Matrix]
    for uvectᵢ ∈ Interest_User_Matrix
```

```
most_similar_interests (generic function with 1 method)
```

```
function most_similar_interests(inter_id::Integer, inter_similarities::VVF,
        uniq_inters)
    simils = inter_similarities[inter_id]

    [(uniq_inters[o_inter_id], sim) for (o_inter_id, sim) ∈ enumerate(simils)
            if inter_id ≠ o_inter_id && sim > 0.] |>
    pairs -> sort(pairs, by=pair -> pair[end], rev=true)
```

We can, for example, find the interests most similar to "big Data" (interest 1) using the following, which suggest these similar interests:

```
msit1 =
  Tuple{String, Float64}[("Hadoop",  0.816497),  ("Java",  0.666667),  ("MapReduce",  0.
```

```
Test Passed
```

```
begin
    @test msit1[1][1] == "Hadoop"
    @test 0.815 < msit1[1][2] < 0.817
    @test msit1[2][1] == "Java"
    @test 0.666 < msit1[2][2] < 0.667
```

Thus now we can create recommendations for a user by summing up the similarities of the interests similar to his/her.

```
item_based_suggestions (generic function with 1 method)
```

```
function item_based_suggestions(user_id::Integer, user_inter_vect, users_inter,
        inter_similarities::VVF, uniq_inters;
        incl_curr_interests=false)
    #
    suggs = Dict{String, Float64}()
```

```julia
    user_inter_v = user_inter_vect[user_id]
    for (inter_id, is_inter) ∈ enumerate(user_inter_v)
        if is_inter == 1
            similar_inters = most_similar_interests(inter_id, inter_similarities,
                uniq_inters)
            for (inter, sim) ∈ similar_inters
                suggs[inter] = get(suggs, inter, 0.) + sim
            end
        end
    end
    # sort...
    suggs = collect(suggs) |>
        p -> sort(p, by=pair -> pair[end], rev=true)
    #
    incl_curr_interests && (return suggs)
    [(sugg, w) for (sugg, w) ∈ suggs if sugg ∉ users_inter[user_id]]
```

```
ibs1 =
 Tuple{String, Float64}[("MapReduce", 1.86181), ("Postgres", 1.3165), ("MongoDB",
```

```julia
ibs1 = item_based_suggestions(1, User_Interest_Vect, Users_Interests,
```

```
Test Passed
```

```julia
begin
    @test ibs1[1][1] == "MapReduce"
    @test 1.86 < ibs1[1][2] < 1.87

    @test ibs1[2][1] ∈ ("Postgres", "MongoDB")  # A tie
    @test 1.31 < ibs1[2][2] < 1.32
```

back to TOC

Matrix Factorization

```
"./recommender_data/ml-100k/u.data"
```

```julia
begin
    const MOVIES = "./recommender_data/ml-100k/u.item"
    const RATINGS = "./recommender_data/ml-100k/u.data"
```

```julia
struct Rating
    user_id:: String
    movie_id::String
    rating::Float32
```

```julia
begin
    movies = Dict{String, String}()

    for row ∈ CSV.File(open(read, MOVIES, enc"ISO-8859-1"); delim="|")
        movies[string(row[1])] = row[2]
    end
```

```
Test Passed
```

```
String["519", "788", "1164", "774", "599", "491", "1195", "1470", "1377", "228
```

```
"Treasure of the Sierra Madre, The (1948)"
```

```
begin
    ratings= Rating[]

    for row ∈ CSV.File(open(read, RATINGS, enc"ISO-8859-1"); delim="\t")
        push!(ratings, Rating(string(row[1]), string(row[2]), Float32(row[3])))
    end
```

```
Test Passed
```

```
starwars_ratings =
 Dict("172" ⇒ Float32[], "50" ⇒ Float32[], "181" ⇒ Float32[])
```

```
##
## Create a dictionary for ratings by movie ids
##
# starwars_ratings = Dict{String, Vector{Float32}}(
#   movie_id => Float32[] for (movie_id, title) ∈ movies if occursin(r"Star Wars|Empire Strikes|Jedi", title)
# )
starwars_ratings = reduce(
    (hsh, (movie_id, title)=r) -> (
        occursin(r"Star Wars|Empire Strikes|Jedi", title) &&
            (hsh[movie_id] = Float32[]); hsh
        ),
    movies;
    init=Dict{String, Vector{Float32}}()
```

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

```
begin
    ## Iterate over ratings, accumulating the Star Wars ones
    for rating ∈ ratings
        rating.movie_id ∈ keys(starwars_ratings) &&
            (push!(starwars_ratings[rating.movie_id], rating.rating))
    end

    ## Compute avg rating for each movie
    avg_ratings = [
        (sum(title_ratings) / length(title_ratings), movie_id)
            for (movie_id, title_ratings) ∈ starwars_ratings
    ]

    ## then print them in order
    with_terminal() do
        for (avg_rating, movie_id) ∈ sort(avg_ratings, by=t -> t[1], rev=true)
            @printf("%.2f %5s\n", avg_rating, movies[movie_id])
        end
    end
```

Ok, let us try to come up with a model to predict these ratings. Our first step is to split the ratings data into 3 subsets: train, validation and test

```
Vector{Rating} (alias for Array{Rating, 1})
```

```
(69999, 5001, 24999)
```

```
begin
    Random.seed!(42)
    Random.shuffle!(ratings)
    s₁, s₂ = round(Int, length(ratings) * .7), round(Int, length(ratings) * .05)

    train, valid = ratings[1:s₁], ratings[s₁+1:s₁+1+s₂]
    test = ratings[s₁+2+s₂:end]

    length(train), length(valid), length(test)
```

```
Test Passed
```

```
(Rating("807", "1066", 5.0), Vector{Rating} (alias for Array{Rating, 1}))
```

Let us define a baseline model which will predict the average rating. We will use MSE (Mean Squared Error) as our metric and check how the baseline does on our test set.

```
1.2706535f0
```

```
begin
    bl_avg_ratings = map(row -> row.rating, train) |>
        sum |> s -> s/length(train)

    bl_error = map(row -> (row.rating - bl_avg_ratings) ^ 2, test) |>
        sum |> s -> s/length(test);
```

```
Test Passed
```

Given our embeddings, the predicted ratings are given by the matrix product of the user embeddings and the movie embeddings. For a given user and movie, that value is just the dot product of the corresponding embeddings.

Let us start by creating the embeddings. We will represent them as dictionaries where the keys are IDs and the values are vectors, which will allow us to easily retrieve the embedding for a given ID:

```
Dict("1" ⇒ Float32[0.773162, 0.912781], "519" ⇒ Float32[0.512218, 0.231151], "78
```

```
begin
    const EMB_DIM = 2

    user_ids = map(r -> r.user_id, ratings) |> unique
    movie_ids = map(r -> r.movie_id, ratings) |> unique

    user_vects = Dict{String, Vector{Float32}}(
        user_id => rand(Float32, EMB_DIM) for user_id ∈ user_ids
    )
    movie_vects = Dict{String, Vector{Float32}}(
        movie_id => rand(Float32, EMB_DIM) for movie_id ∈ movie_ids
    )
```

Now, it is time to write our taining loop.

```
train_loop (generic function with 1 method)
```

```
function train_loop(ds::Vector{Rating}, ds_name::Symbol;
        η::Union{Float32, Nothing}=nothing)
    loss = 0.
    for (ix, rating) ∈ enumerate(ds)
        movie_v = movie_vects[rating.movie_id]
        user_v = user_vects[rating.user_id]
        ŷ = dot(user_v, movie_v)
        err = ŷ - rating.rating
        loss += err ^ 2
        if !isnothing(η)
            ## we have ŷ ≡ m₀ × u₀ + m₁ × u₁ + ... +mₖ × uₖ
            ## thus each uⱼ contributes to output with coefficient mⱼ
            ## and conv. each mⱼ contributes to output with coefficient uⱼ
            ∇user = movie_v * err # [err * mⱼ for mⱼ ∈ movie_v]
            ∇movie = user_v * err
            #
            user_v .-= η * ∇user
            movie_v .-= η * ∇movie
        end
        ix % 10_000 == 0 && @printf("\t%5d: avg_loss: %3.6f\n", ix, loss / ix)
    end
    avg_loss = loss / length(ds)
    @printf("\tFinal avg_loss(%12s): %3.6f\n", ds_name, avg_loss)
    avg_loss
```

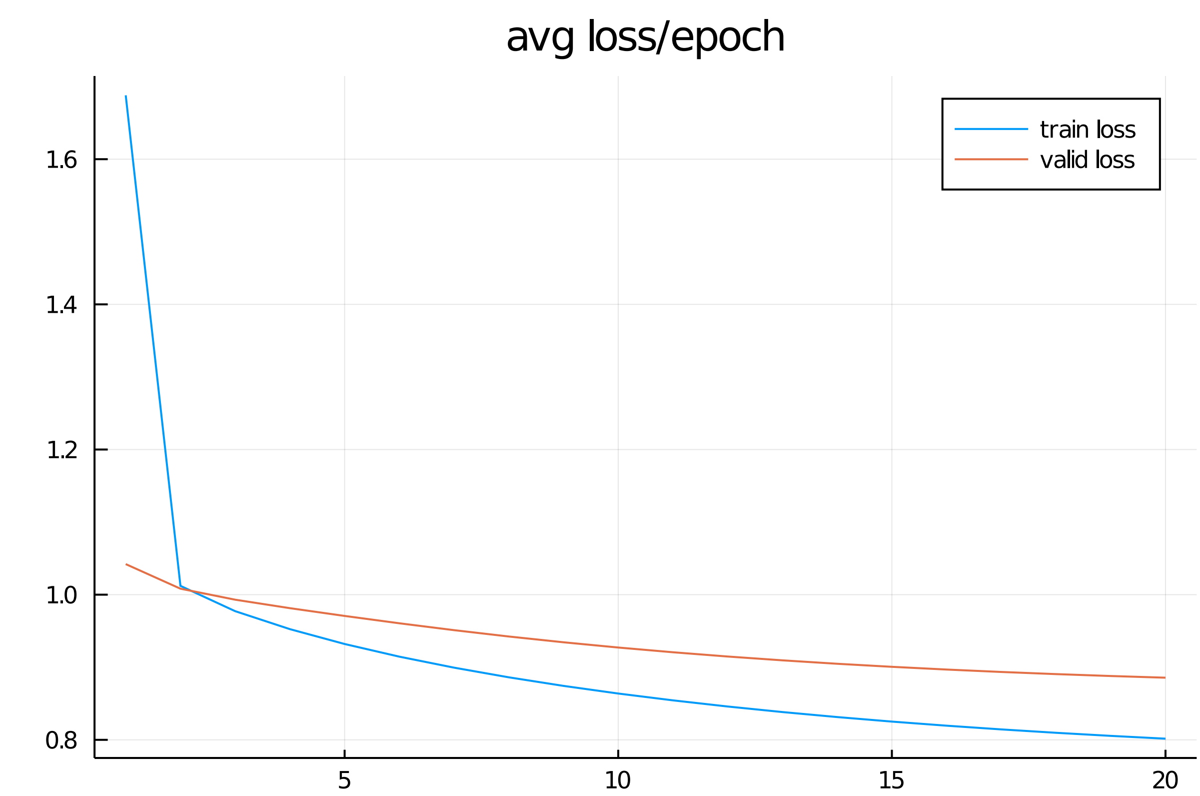
```
1 => η: 0.04500    10000: avg_loss: 4.526434
    20000: avg_loss: 3.063639
    30000: avg_loss: 2.461263
    40000: avg_loss: 2.142522
    50000: avg_loss: 1.937317
    60000: avg_loss: 1.792576
    Final avg_loss(    Training): 1.688028
    Final avg_loss(  Validation): 1.042134
2 => η: 0.04050    10000: avg_loss: 1.029605
    20000: avg_loss: 1.023878
    30000: avg_loss: 1.016865
    40000: avg_loss: 1.019998
    50000: avg_loss: 1.017876
    60000: avg_loss: 1.013756
    Final avg_loss(    Training): 1.012141
    Final avg_loss(  Validation): 1.008258
3 => η: 0.03645    10000: avg_loss: 0.983533
    20000: avg_loss: 0.983199
    30000: avg_loss: 0.977911
    40000: avg_loss: 0.982449
```

```
begin
    η = Float32(0.05)
    avg_train_losses, avg_valid_losses = [], []
```

```julia
    avg_test_loss = nothing

    with_terminal() do
        for epoch ∈ 1:20
            global η *= Float32(0.9)
            @printf("%2d => η: %2.5f", epoch, η)
            #
            avg_train_loss = train_loop(train, :Training; η)
            avg_valid_loss = train_loop(valid, :Validation)
            push!(avg_train_losses, avg_train_loss)
            push!(avg_valid_losses, avg_valid_loss)
        end
        #
        global avg_test_loss = train_loop(test, :Test)
    end
```

```
(avg_test_loss = "92.96%")
```




```julia
begin
    plot(avg_train_losses, title="avg loss/epoch", label="train loss")
    plot!(avg_valid_losses, label="valid loss")
```

Now, we will inspect the learned vectors. Because there is no reason to expect the two components to be particularly meaningful, we are going to use PCA.

```
Vector{Float32}[Float32[0.712455, 0.701718], Float32[-0.701556, 0.712614]]
```

```julia
begin
    orig_vects = values(movie_vects) |> collect
    comps = pca(orig_vects, 2)
```

Let us transform our vectors to represent the principal components and join in the movie IDs and average ratings:

```
Tuple{String, Float32, String, Vector{Float32}}[("1", 3.87832, "_", Float32[1.9991
```

```
begin
    ratings_by_movie = Dict{String, DT}()
    for rating in ratings
        id = rating.movie_id
        ary = get(ratings_by_movie, id, DT{Float32}())
        push!(ary, rating.rating)
        ratings_by_movie[id] = ary
    end
    vs = [
        (
            movie_id,
            sum(ratings_by_movie[movie_id]) / length(ratings_by_movie[movie_id]),
            get(movies, movie_id, "_"),
            vect
        )
        for (movie_id, vect) ∈ zip(keys(movie_vects), transform(orig_vects, comps))
    ]
end
```

```
Tuple{String, Float32, String, Vector{Float32}}[("1449", 4.625, "Pather Panchali (1
```

```
# top 25 by first principal component
```

```
Tuple{String, Float32, String, Vector{Float32}}[("1567", 1.0, "Careful (1992)", Fl
```

```
# bottom 25 by first principal component
```

The top 25 are all highly rated, while the bottom 25 are *mostly* low-rated (or unrated in the training data), which suggests that the first principal component is mostly capturing "how good is this movie?"

It is hard to make much sense of the second component; and, indeed the 2-dimensional embeddings performed only slightly better than the one-dimensional embeddings, suggesting that whatever the second component captured is possibly very subtle...