
Recommendation System: Factorization Machine

Jiaying Wang, Sailun Xu, Joyce Xu

Overview

Idea:

FMs model all interactions between variables using factorized parameters.

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j$$

Overview

Factorization Machines uses collaborative filtering to predict with real value feature vector

Advantage

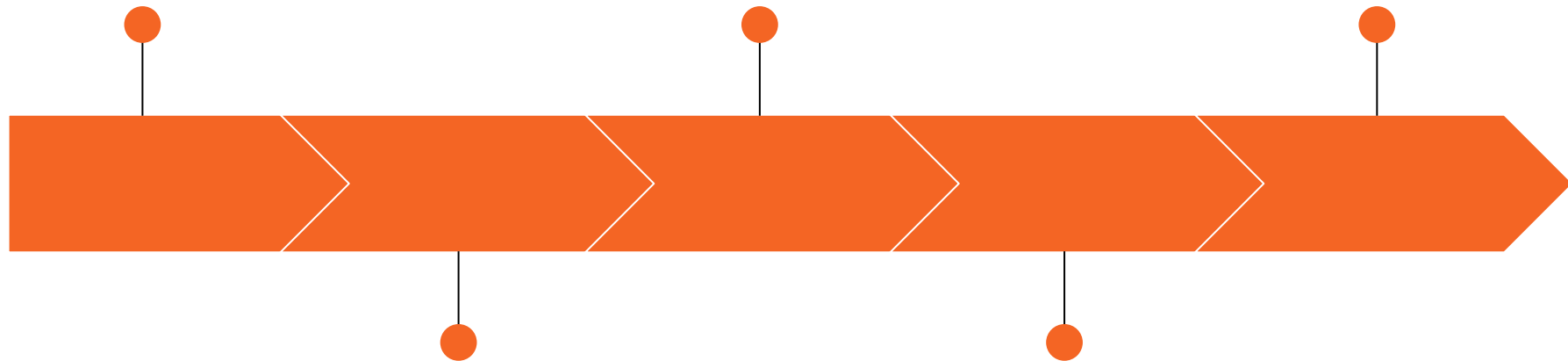
Compare to SVM, works well with sparse matrix

Project Timeline

Prediction & gradient
& vanilla gradient
descent

Profiling

Optimization: In place,
fully vectorize



Using Minimize.py to
do gradient descent

Optimization: Cython

Prediction & Gradient & Vanilla Gradient Descent

Prediction & gradient

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j$$

$$\frac{\partial}{\partial \theta} \hat{y}(\mathbf{x}) = \begin{cases} 1, & \text{if } \theta \text{ is } w_0 \\ x_i, & \text{if } \theta \text{ is } w_i \\ x_i \sum_{j=1}^n v_{j,f} x_j - v_{i,f} x_i^2, & \text{if } \theta \text{ is } v_{i,f} \end{cases}$$

We implement prediction and gradient, and do gradient check to ensure they are correct

Prediction & gradient

Problem: gradient w/r to matrix **V** super slow with double loop

```
dLdV = np.zeros((m, k))
XV = X.dot(V)
for i in range(m):
    temp = np.array(X[:,i]).reshape(-1)
    temp2 = temp ** 2
    for f in range(k):
        v_ifX = V[i][f] * temp2
        Xv_ifX = temp * (XV[:,f].reshape(-1))
        dLdV[i][f] = diff.dot(v_ifX - Xv_ifX)
dLdV = dLdV * 2.0 / n
return compress(dLdw0, dLdwi, dLdV, m, k)
```

Prediction & gradient

Result:

By now, after 300 epochs(go through all data), the RMSE has reached **1.13**, close to the baseline **1.12** set by predicting the mean for all features

Minimize.py

Minimize.py

Interface: optimized gradient descent based on given function and its gradient

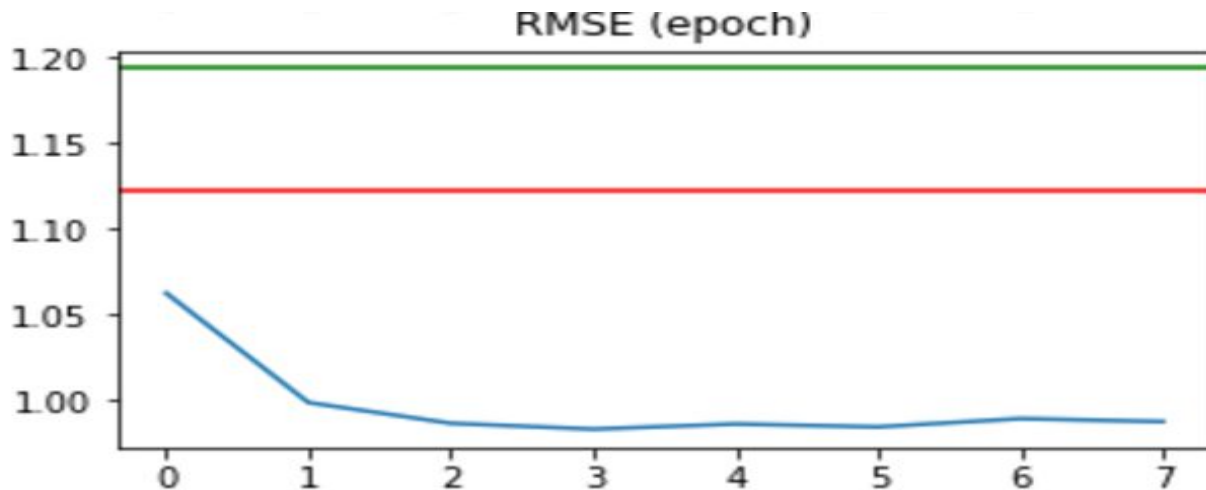
Each iteration would be significantly slower, yet improvement through each iteration would be magnificent.

Before: 300 epochs, **now:** within 10 epochs to get similar RMSE.

Minimize.py

Result:

RMSE go down to 0.98, close to the result given by the original paper(0.91), much lower than the baseline (red -> mean, green -> mode) now.



Profiling

Profiling

Problem:


Result has been good, yet we still need to tune hyper parameter: k , which determine the size of \mathbf{V} (according to the paper, actually higher k is, better the result will be)

So far, it is extremely slow! (each epoch for hours) We need to find which parts are most costly, and optimize them!

Profiling

Not surprisingly, gradient w/r to the matrix is the slowest part, as we have nested loop! (partial results below:)

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	28.776	28.776	<string>:1(<module>)
80000	0.509	0.000	5.177	0.000	BatchGD.py:11(predictionForOne)
21	0.000	0.000	0.004	0.000	BatchGD.py:117(compress)
41	0.000	0.000	0.000	0.000	BatchGD.py:125(extract)
20	0.109	0.005	3.064	0.153	BatchGD.py:135(lossH)
20	15.120	0.756	25.590	1.280	BatchGD.py:171(gradHVec)
1	0.001	0.001	28.776	28.776	BatchGD.py:222(runMinVec)
40	0.454	0.011	5.772	0.144	BatchGD.py:23(prediction)
40	0.141	0.004	5.318	0.133	BatchGD.py:28(<listcomp>)
160040	0.060	0.000	0.581	0.000	_methods.py:31(_sum)
13	0.000	0.000	0.000	0.000	_methods.py:37(_any)
42	0.000	0.000	0.000	0.000	fromnumeric.py:1364(ravel)
160040	0.331	0.000	0.965	0.000	fromnumeric.py:1710(sum)
13	0.000	0.000	0.000	0.000	fromnumeric.py:1866(any)
42	0.000	0.000	0.002	0.000	function_base.py:4951(append)



Optimization: Cython

Optimization: cython

We vectorize the gradient to make nested loop into single loop: we see that many things could be pre-computed: (element-wise square, transpose, XV)

```
dLdV = np.zeros((m, k))
XV = X.dot(V)
xT = X.T
xT2 = xT ** 2
diffT = diff.reshape(n,1)
for i in range(m):
    v_ifX = (V[i, np.newaxis].T).dot(xT2[i, np.newaxis])
    Xv_ifX = (xT[i][:, np.newaxis] * XV).T
    res = (v_ifX - Xv_ifX).dot(diffT)
    dLdV[i] = res.ravel()
dLdV *= 2/n
```

Optimization: cython

We also use cython to compile c code from python:

Performance improvement:

```
%timeit -n2 -r3 gradHVec(H, X_train,y_train,m, k, 0.05)
```

179 ms \pm 7.89 ms per loop (mean \pm std. dev. of 3 runs, 2 loops each)

```
%timeit -n2 -r3 gradHVecc(H, X_train,y_train,m, k, 0.05)  
# %lprun -f gradHVecc gradHVecc(H, X_train,y_train,m, k, 0.05)
```

184 ms \pm 9.83 ms per loop (mean \pm std. dev. of 3 runs, 2 loops each)

**Optimization: In place, fully
vectorize**

Optimization: In place, fully vectorize:

We still want to improve as it is still slow:

In place:

We allocate result(Xv_ifX) first, and make:

```
np.multiply(xT[i], XVT, Xv_ifX)
```

instead of:

```
Xv_ifX = xT[i] * XVT
```

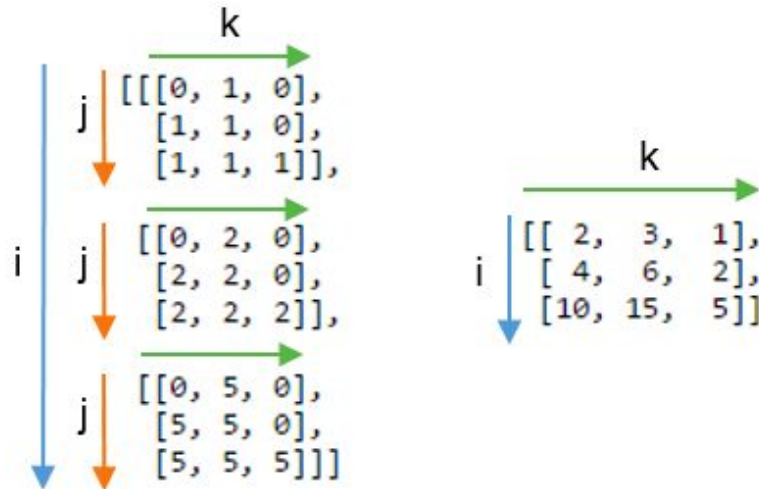
By specifying the output place

Optimization: In place, fully vectorize:

Result: quite confusing, that there is no significant improvement.

Optimization: In place, fully vectorize:

Fully vectorize: `np.einsum`



Optimization: In place, fully vectorize:

Fully vectorize: `np.einsum`

This is the best vectorization we could make: no for loop at all!

```
v_ifX = np.einsum('ij, jk -> jki', x2, V)
Xv_ifX = np.einsum('ij, kj -> ikj', xT, XVT)
dLdV = ((v_ifX - Xv_ifX).dot(diffT)).reshape(m, k)
dLdV *= 2/n
```

Optimization: In place, fully vectorize:

Result: it is actually much lower, the problem is that the memory overhead is a big deal: if instead of loop, we store everything in a big matrix, then its dimension is about: $90000 \times 3000 \times 30$, so that there is lot of page swap, making it much slower.

Optimization: In place, fully vectorize:

Remedy: We thus partition the data, and do fully vectorized form within each partition:

Final Result:

Halves the time for taking gradient, each epoch goes down to several minutes!

Note:

As we don't have for loop anymore, cython is not particularly advantageous now

TODO:

TODO:

Tune the hyperparameter

More epochs

Larger k for better result
