homework_12_matrix_factorization_Yiman Li

February 2, 2020

0.1 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook. 2. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)). 3. Concatenate your solutions for other tasks with the output of Step 2. On linux, you can use `pdfunite`, there are similar tools for other platforms, too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` version 5.5 or later by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

1 Matrix Factorization

[20]:
```
import time
import scipy.sparse as sp
import numpy as np
from scipy.sparse.linalg import svds
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
%matplotlib inline
```

1.1 Restaurant recommendation

The goal of this task is to recommend restaurants to users based on the rating data in the Yelp dataset. For this, we try to predict the rating a user will give to a restaurant they have not yet rated based on a latent factor model.

Specifically, the objective function (loss) we wanted to optimize is:

$$\mathcal{L} = \min_{P,Q} \sum_{(i,x)\in W} (M_{ix} - \mathbf{q}_i^T \mathbf{p}_x)^2 + \lambda \sum_x \|\mathbf{p}_x\|^2 + \lambda \sum_i \|\mathbf{q}_i\|^2$$

where W is the set of (i, x) pairs for which the rating M_{ix} given by user i to restaurant x is known. Here we have also introduced two regularization terms to help us with overfitting where λ is hyper-parameter that control the strength of the regularization.

Hint 1: Using the closed form solution for regression might lead to singular values. To avoid this issue perform the regression step with an existing package such as scikit-learn. It is advisable to use ridge regression to account for regularization.

Hint 2: If you are using the scikit-learn package remember to set `fit_intercept = False` to only learn the coefficients of the linear regression.

1.1.1 Load and Preprocess the Data (nothing to do here)

[21]:
```
ratings = np.load("exercise_12_matrix_factorization_ratings.npy")
```

[22]:
```
# We have triplets of (user, restaurant, rating).
ratings
```

[22]:
```
array([[101968,   1880,       1],
       [101968,    284,       5],
       [101968,   1378,       2],
       ...,
       [ 72452,   2100,       4],
       [ 72452,   2050,       5],
       [ 74861,   3979,       5]], dtype=int64)
```

Now we transform the data into a matrix of dimension [N, D], where N is the number of users and D is the number of restaurants in the dataset. We store the data as a sparse matrix to avoid out-of-memory issues.

[23]:
```
n_users = np.max(ratings[:,0] + 1)
n_restaurants = np.max(ratings[:,1] + 1)
M = sp.coo_matrix((ratings[:,2], (ratings[:,0], ratings[:,1])), shape=(n_users,␣
 ↪n_restaurants)).tocsr()
M
```

[23]:
```
<337867x5899 sparse matrix of type '<class 'numpy.int64'>'
        with 929606 stored elements in Compressed Sparse Row format>
```

To avoid the cold start problem, in the preprocessing step, we recursively remove all users and restaurants with 10 or less ratings.

Then, we randomly select 200 data points for the validation and test sets, respectively.

After this, we subtract the mean rating for each users to account for this global effect.

Note: Some entries might become zero in this process – but these entries are different than the 'unknown' zeros in the matrix. We store the indices for which we the rating data available in a separate variable.

[24]:
```
def cold_start_preprocessing(matrix, min_entries):
    """
    Recursively removes rows and columns from the input matrix which have less␣
 ↪than min_entries nonzero entries.

    Parameters
    ----------
    matrix      : sp.spmatrix, shape [N, D]
                  The input matrix to be preprocessed.
    min_entries : int
                  Minimum number of nonzero elements per row and column.

```

```
    Returns
    -------
    matrix      : sp.spmatrix, shape [N', D']
                   The pre-processed matrix, where N' <= N and D' <= D

    """
    print("Shape before: {}".format(matrix.shape))

    shape = (-1, -1)
    while matrix.shape != shape:
        shape = matrix.shape
        nnz = matrix>0
        row_ixs = nnz.sum(1).A1 > min_entries
        matrix = matrix[row_ixs]
        nnz = matrix>0
        col_ixs = nnz.sum(0).A1 > min_entries
        matrix = matrix[:,col_ixs]
    print("Shape after: {}".format(matrix.shape))
    nnz = matrix>0
    assert (nnz.sum(0).A1 > min_entries).all()
    assert (nnz.sum(1).A1 > min_entries).all()
    return matrix
```

1.1.2 Task 1: Implement a function that subtracts the mean user rating from the sparse rating matrix

[25]:
```
def shift_user_mean(matrix):
    """
    Subtract the mean rating per user from the non-zero elements in the input␣
↪matrix.

    Parameters
    ----------
    matrix : sp.spmatrix, shape [N, D]
             Input sparse matrix.
    Returns
    -------
    matrix : sp.spmatrix, shape [N, D]
             The modified input matrix.

    user_means : np.array, shape [N, 1]
                 The mean rating per user that can be used to recover the␣
↪absolute ratings from the mean-shifted ones.

    """
```

```
    # TODO: Compute the modified matrix and user_means
    nnz_mask = (matrix>0)
    user_means = matrix.sum(1) / nnz_mask.sum(1)
    subtract_mask = sp.csr_matrix(user_means).multiply(nnz_mask)
    matrix = matrix-subtract_mask

    assert np.all(np.isclose(matrix.mean(1), 0))
    return matrix, user_means
```

1.1.3 Split the data into a train, validation and test set (nothing to do here)

[26]:
```
def split_data(matrix, n_validation, n_test):
    """
    Extract validation and test entries from the input matrix.

    Parameters
    ----------
    matrix          : sp.spmatrix, shape [N, D]
                      The input data matrix.
    n_validation    : int
                      The number of validation entries to extract.
    n_test          : int
                      The number of test entries to extract.

    Returns
    -------
    matrix_split    : sp.spmatrix, shape [N, D]
                      A copy of the input matrix in which the validation and␣
 ↪test entries have been set to zero.

    val_idx         : tuple, shape [2, n_validation]
                      The indices of the validation entries.

    test_idx        : tuple, shape [2, n_test]
                      The indices of the test entries.

    val_values      : np.array, shape [n_validation, ]
                      The values of the input matrix at the validation indices.

    test_values     : np.array, shape [n_test, ]
                      The values of the input matrix at the test indices.

    """

    matrix_cp = matrix.copy()
    non_zero_idx = np.argwhere(matrix_cp)
```

```
    ixs = np.random.permutation(non_zero_idx)
    val_idx = tuple(ixs[:n_validation].T)
    test_idx = tuple(ixs[n_validation:n_validation + n_test].T)

    val_values = matrix_cp[val_idx].A1
    test_values = matrix_cp[test_idx].A1

    matrix_cp[val_idx] = matrix_cp[test_idx] = 0
    matrix_cp.eliminate_zeros()

    return matrix_cp, val_idx, test_idx, val_values, test_values
```

[27]:
```
M = cold_start_preprocessing(M, 20)
```

```
Shape before: (337867, 5899)
Shape after: (3529, 2072)
```

[28]:
```
n_validation = 200
n_test = 200
# Split data
M_train, val_idx, test_idx, val_values, test_values = split_data(M,␣
 ↪n_validation, n_test)
```

[29]:
```
# Remove user means.
nonzero_indices = np.argwhere(M_train)
M_shifted, user_means = shift_user_mean(M_train)
# Apply the same shift to the validation and test data.
val_values_shifted = val_values - user_means[np.array(val_idx).T[:,0]].A1
test_values_shifted = test_values - user_means[np.array(test_idx).T[:,0]].A1
```

1.1.4 Compute the loss function (nothing to do here)

[30]:
```
def loss(values, ixs, Q, P, reg_lambda):
    """
    Compute the loss of the latent factor model (at indices ixs).
    Parameters
    ----------
    values : np.array, shape [n_ixs,]
        The array with the ground-truth values.
    ixs : tuple, shape [2, n_ixs]
        The indices at which we want to evaluate the loss (usually the nonzero␣
 ↪indices of the unshifted data matrix).
    Q : np.array, shape [N, k]
        The matrix Q of a latent factor model.
    P : np.array, shape [k, D]
        The matrix P of a latent factor model.
    reg_lambda : float
        The regularization strength
```

```
    Returns
    -------
    loss : float
           The loss of the latent factor model.

    """
    mean_sse_loss = np.sum((values - Q.dot(P)[ixs])**2)
    regularization_loss =  reg_lambda * (np.sum(np.linalg.norm(P, axis=0)**2) +␣
↪np.sum(np.linalg.norm(Q, axis=1) ** 2))

    return mean_sse_loss + regularization_loss
```

1.2 Alternating optimization

In the first step, we will approach the problem via alternating optimization, as learned in the lecture. That is, during each iteration you first update Q while having P fixed and then vice versa.

1.2.1 Task 2: Implement a function that initializes the latent factors Q and P

[31]:
```
def initialize_Q_P(matrix, k, init='random'):
    """
    Initialize the matrices Q and P for a latent factor model.

    Parameters
    ----------
    matrix : sp.spmatrix, shape [N, D]
             The matrix to be factorized.
    k      : int
             The number of latent dimensions.
    init   : str in ['svd', 'random'], default: 'random'
             The initialization strategy. 'svd' means that we use SVD to␣
↪initialize P and Q, 'random' means we initialize
             the entries in P and Q randomly in the interval [0, 1).

    Returns
    -------
    Q : np.array, shape [N, k]
        The initialized matrix Q of a latent factor model.

    P : np.array, shape [k, D]
        The initialized matrix P of a latent factor model.
    """
    np.random.seed(0)
    N, D = np.shape(matrix)
    # TODO: Compute Q and P
    if init == 'svd':
```

```
        U, S, V = svds(matrix, k=k)
        S_x = np.diag(S)
        Q = U.dot(S_x)
        P = V
    elif init == 'random':
        Q = np.random.randn(N, k)
        P = np.random.randn(k, D)
    else:
        raise ValueError

    assert Q.shape == (matrix.shape[0], k)
    assert P.shape == (k, matrix.shape[1])
    return Q, P
```

1.2.2 Task 3: Implement the alternating optimization approach

[32]:
```
def latent_factor_alternating_optimization(M, non_zero_idx, k, val_idx,␣
↪val_values,
                                           reg_lambda, max_steps=100,␣
↪init='random',
                                           log_every=1, patience=5,␣
↪eval_every=1):
    """
    Perform matrix factorization using alternating optimization. Training is␣
↪done via patience,
    i.e. we stop training after we observe no improvement on the validation loss␣
↪for a certain
    amount of training steps. We then return the best values for Q and P oberved␣
↪during training.

    Parameters
    ----------
    M                 : sp.spmatrix, shape [N, D]
                        The input matrix to be factorized.

    non_zero_idx      : np.array, shape [nnz, 2]
                        The indices of the non-zero entries of the un-shifted␣
↪matrix to be factorized.
                        nnz refers to the number of non-zero entries. Note that␣
↪this may be different
                        from the number of non-zero entries in the input matrix␣
↪M, e.g. in the case
                        that all ratings by a user have the same value.

    k                 : int
                        The latent factor dimension.
```

```
    val_idx             : tuple, shape [2, n_validation]
                          Tuple of the validation set indices.
                          n_validation refers to the size of the validation set.

    val_values          : np.array, shape [n_validation, ]
                          The values in the validation set.

    reg_lambda          : float
                          The regularization strength.

    max_steps           : int, optional, default: 100
                          Maximum number of training steps. Note that we will stop␣
↪early if we observe
                          no improvement on the validation error for a specified␣
↪number of steps
                          (see "patience" for details).

    init                : str in ['random', 'svd'], default 'random'
                          The initialization strategy for P and Q. See function␣
↪initialize_Q_P for details.

    log_every           : int, optional, default: 1
                          Log the training status every X iterations.

    patience            : int, optional, default: 5
                          Stop training after we observe no improvement of the␣
↪validation loss for X evaluation
                          iterations (see eval_every for details). After we stop␣
↪training, we restore the best
                          observed values for Q and P (based on the validation␣
↪loss) and return them.

    eval_every          : int, optional, default: 1
                          Evaluate the training and validation loss every X steps.␣
↪If we observe no improvement
                          of the validation error, we decrease our patience by 1,␣
↪else we reset it to *patience*.

    Returns
    -------
    best_Q              : np.array, shape [N, k]
                          Best value for Q (based on validation loss) observed␣
↪during training

    best_P              : np.array, shape [k, D]
```

```
                        Best value for P (based on validation loss) observed␣
↪during training

    validation_losses : list of floats
                        Validation loss for every evaluation iteration, can be␣
↪used for plotting the validation
                        loss over time.

    train_losses      : list of floats
                        Training loss for every evaluation iteration, can be␣
↪used for plotting the training
                        loss over time.

    converged_after   : int
                        it - patience*eval_every, where it is the iteration in␣
↪which patience hits 0,
                        or -1 if we hit max_steps before converging.

    """

    # TODO: Compute best_Q, best_P, validation_losses, train_losses and␣
↪converged_after
    nnz_mask = sp.coo_matrix((np.ones(len(non_zero_idx)), (non_zero_idx[:
↪,0],non_zero_idx[:,1])), shape=M.shape, dtype="uint8").tocsr()
    nnz_mask_col = nnz_mask.tocsc()

    cols = nnz_mask.T.tolil().rows
    rows = nnz_mask.tolil().rows

    reg = Ridge(alpha=reg_lambda, fit_intercept=False)

    Q,P = initialize_Q_P(M, k, init)
    train_losses = []
    validation_losses = []
    best_val_loss = best_Q = best_P = converged_after = -1
    train_idx = tuple(non_zero_idx.T)

    bef = -1
    times = []
    for it in range(max_steps):
        if bef != -1:
            times.append(time.time()-bef)
        bef = time.time()

        if it % eval_every == 0:
            train_loss = loss(M[train_idx].A1, train_idx, Q, P, reg_lambda)
```

```
            train_losses.append(train_loss)

            val_loss = loss(val_values, val_idx, Q, P, reg_lambda)
            validation_losses.append(val_loss)

            if best_val_loss < 0 or val_loss < best_val_loss:
                best_val_loss = val_loss
                best_Q = Q
                best_P = P
                current_patience = patience
            else:
                current_patience -= 1

            if current_patience == 0:
                converged_after = it - patience*eval_every
                break

        print("Iteration {}, training loss: {:.3f}, validation loss: {:.3f}".
↪format(it, train_loss, val_loss))

        # fix Q and update P
        # fix Q and update P
        for rating_idx in range(M.shape[1]):
            nnz_idx = cols[rating_idx]
            res = reg.fit(Q[nnz_idx], np.squeeze(M[nnz_idx, rating_idx].
↪toarray()))
            P[:, rating_idx] = res.coef_

        for user_idx in range(M.shape[0]):
            nnz_idx = rows[user_idx]
            res = reg.fit(P[:, nnz_idx].T, np.squeeze(M[user_idx, nnz_idx].
↪toarray()))
            Q[user_idx, :] = res.coef_

    print("Converged after {} iterations, on average {:.3f}s per iteration".
↪format(converged_after, np.mean(times)))

    return best_Q, best_P, validation_losses, train_losses, converged_after
```

1.2.3 Train the latent factor (nothing to do here)

[33]:
```
Q, P, val_loss, train_loss, converged =␣
↪latent_factor_alternating_optimization(M_shifted, nonzero_indices,
                                                                                ␣
↪k=100, val_idx=val_idx,
```

```
↪val_values=val_values_shifted,
↪reg_lambda=1e-4, init='random',
↪max_steps=100, patience=10)
```

```
Iteration 0, training loss: 15601179.586, validation loss: 21937.249
Iteration 1, training loss: 2516.317, validation loss: 753.995
Iteration 2, training loss: 540.570, validation loss: 523.066
Iteration 3, training loss: 194.908, validation loss: 513.264
Iteration 4, training loss: 95.040, validation loss: 532.319
Iteration 5, training loss: 57.519, validation loss: 548.417
Iteration 6, training loss: 40.974, validation loss: 535.613
Iteration 7, training loss: 32.765, validation loss: 534.745
Iteration 8, training loss: 28.356, validation loss: 530.761
Iteration 9, training loss: 25.839, validation loss: 527.236
Iteration 10, training loss: 24.336, validation loss: 524.362
Iteration 11, training loss: 23.395, validation loss: 521.984
Iteration 12, training loss: 22.792, validation loss: 517.433
Iteration 13, training loss: 22.386, validation loss: 511.743
Iteration 14, training loss: 22.102, validation loss: 507.319
Iteration 15, training loss: 21.895, validation loss: 504.711
Iteration 16, training loss: 21.737, validation loss: 502.432
Iteration 17, training loss: 21.613, validation loss: 500.809
Iteration 18, training loss: 21.511, validation loss: 499.392
Iteration 19, training loss: 21.425, validation loss: 498.078
Iteration 20, training loss: 21.350, validation loss: 497.056
Iteration 21, training loss: 21.283, validation loss: 495.862
Iteration 22, training loss: 21.223, validation loss: 494.797
Iteration 23, training loss: 21.167, validation loss: 493.988
Iteration 24, training loss: 21.114, validation loss: 493.125
Iteration 25, training loss: 21.065, validation loss: 492.341
Iteration 26, training loss: 21.017, validation loss: 491.748
Iteration 27, training loss: 20.972, validation loss: 491.345
Iteration 28, training loss: 20.929, validation loss: 490.784
Iteration 29, training loss: 20.887, validation loss: 490.268
Iteration 30, training loss: 20.847, validation loss: 489.816
Iteration 31, training loss: 20.807, validation loss: 489.395
Iteration 32, training loss: 20.769, validation loss: 488.989
Iteration 33, training loss: 20.732, validation loss: 488.596
Iteration 34, training loss: 20.695, validation loss: 488.198
Iteration 35, training loss: 20.659, validation loss: 487.797
Iteration 36, training loss: 20.624, validation loss: 487.379
Iteration 37, training loss: 20.590, validation loss: 486.958
Iteration 38, training loss: 20.556, validation loss: 486.527
Iteration 39, training loss: 20.523, validation loss: 486.110
```

```
Iteration 40, training loss: 20.490, validation loss: 485.691
Iteration 41, training loss: 20.458, validation loss: 485.294
Iteration 42, training loss: 20.427, validation loss: 484.889
Iteration 43, training loss: 20.396, validation loss: 484.509
Iteration 44, training loss: 20.366, validation loss: 484.115
Iteration 45, training loss: 20.336, validation loss: 483.745
Iteration 46, training loss: 20.306, validation loss: 483.358
Iteration 47, training loss: 20.277, validation loss: 482.976
Iteration 48, training loss: 20.248, validation loss: 482.730
Iteration 49, training loss: 20.220, validation loss: 482.474
Iteration 50, training loss: 20.192, validation loss: 482.170
Iteration 51, training loss: 20.164, validation loss: 481.861
Iteration 52, training loss: 20.137, validation loss: 481.542
Iteration 53, training loss: 20.110, validation loss: 481.214
Iteration 54, training loss: 20.084, validation loss: 480.872
Iteration 55, training loss: 20.058, validation loss: 480.517
Iteration 56, training loss: 20.032, validation loss: 480.160
Iteration 57, training loss: 20.006, validation loss: 479.798
Iteration 58, training loss: 19.981, validation loss: 479.445
Iteration 59, training loss: 19.956, validation loss: 479.091
Iteration 60, training loss: 19.931, validation loss: 478.747
Iteration 61, training loss: 19.907, validation loss: 478.401
Iteration 62, training loss: 19.883, validation loss: 478.065
Iteration 63, training loss: 19.859, validation loss: 477.726
Iteration 64, training loss: 19.835, validation loss: 477.395
Iteration 65, training loss: 19.811, validation loss: 477.063
Iteration 66, training loss: 19.788, validation loss: 476.738
Iteration 67, training loss: 19.765, validation loss: 476.413
Iteration 68, training loss: 19.742, validation loss: 476.094
Iteration 69, training loss: 19.720, validation loss: 475.775
Iteration 70, training loss: 19.697, validation loss: 475.463
Iteration 71, training loss: 19.675, validation loss: 475.150
Iteration 72, training loss: 19.653, validation loss: 474.844
Iteration 73, training loss: 19.631, validation loss: 474.538
Iteration 74, training loss: 19.609, validation loss: 474.238
Iteration 75, training loss: 19.588, validation loss: 473.939
Iteration 76, training loss: 19.567, validation loss: 473.645
Iteration 77, training loss: 19.546, validation loss: 473.352
Iteration 78, training loss: 19.525, validation loss: 473.063
Iteration 79, training loss: 19.504, validation loss: 472.776
Iteration 80, training loss: 19.483, validation loss: 472.493
Iteration 81, training loss: 19.463, validation loss: 472.212
Iteration 82, training loss: 19.442, validation loss: 471.934
Iteration 83, training loss: 19.422, validation loss: 471.658
Iteration 84, training loss: 19.402, validation loss: 471.386
Iteration 85, training loss: 19.382, validation loss: 471.115
Iteration 86, training loss: 19.363, validation loss: 470.848
Iteration 87, training loss: 19.343, validation loss: 470.583
```

```
Iteration 88, training loss: 19.323, validation loss: 470.321
Iteration 89, training loss: 19.304, validation loss: 470.061
Iteration 90, training loss: 19.285, validation loss: 469.804
Iteration 91, training loss: 19.266, validation loss: 469.549
Iteration 92, training loss: 19.247, validation loss: 469.297
Iteration 93, training loss: 19.228, validation loss: 469.047
Iteration 94, training loss: 19.209, validation loss: 468.801
Iteration 95, training loss: 19.191, validation loss: 468.556
Iteration 96, training loss: 19.172, validation loss: 468.314
Iteration 97, training loss: 19.154, validation loss: 468.075
Iteration 98, training loss: 19.136, validation loss: 467.838
Iteration 99, training loss: 19.118, validation loss: 467.603
Converged after -1 iterations, on average 4.454s per iteration
```

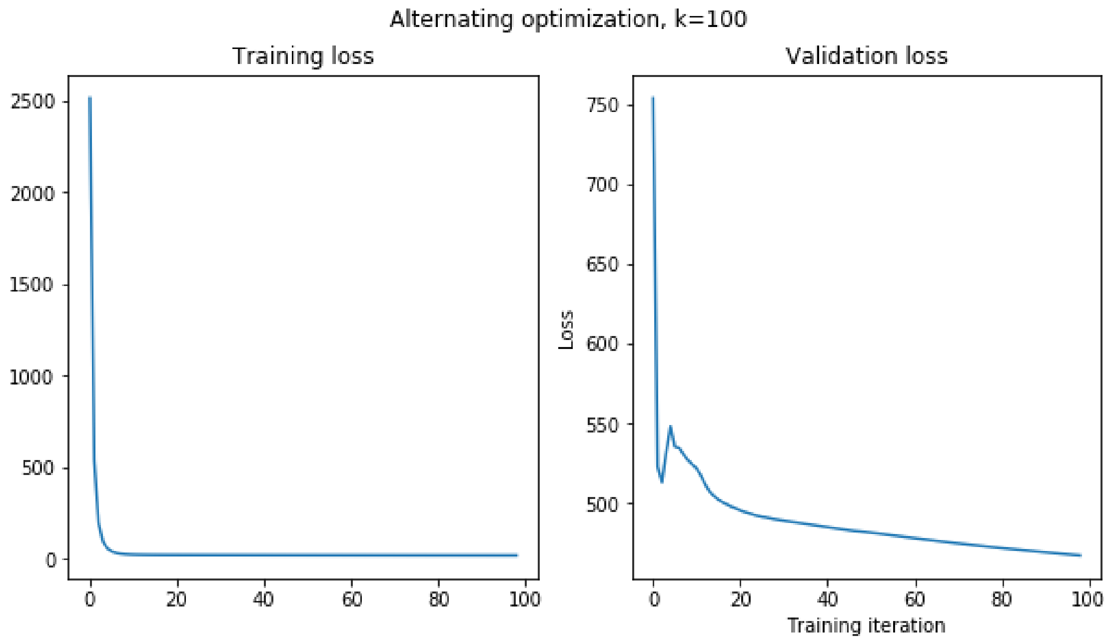
1.2.4 Plot the validation and training losses over for each iteration (nothing to do here)

[34]:
```
fig, ax = plt.subplots(1, 2, figsize=[10, 5])
fig.suptitle("Alternating optimization, k=100")

ax[0].plot(train_loss[1::])
ax[0].set_title('Training loss')
plt.xlabel("Training iteration")
plt.ylabel("Loss")


ax[1].plot(val_loss[1::])
ax[1].set_title('Validation loss')
plt.xlabel("Training iteration")
plt.ylabel("Loss")

plt.show()
```


Alternating optimization, k=100
Training loss
Validation loss
2500
2000
1500
1000
500
0
750
700
650
600
550
500
Loss
0
20
40
60
80
100
Training iteration