Machine Learning homework 10 solution

Dimensionality Reduction & Matrix Factorization

Wiktor Jurasz - M.Nr. 03709419

January 20, 2019

1 Problem 1

For given model MLE estimates for μ_{ML}:

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N} x_i = \bar{x} \tag{1}$$

Thus

$$\bar{Ax} = \frac{1}{N}\sum_{i=1}^{N} Ax_i = A\frac{1}{N}\sum_{i=1}^{N} x_i = A\mu_{ML} \tag{2}$$

MLE estimate for W_{ML}

$$W_{ML} = U(\Lambda - \sigma^2 I)^{\frac{1}{2}}V \tag{3}$$

The Covariance of x

$$Cov(x) = S = U\Lambda U^T \tag{4}$$

The Covariance of Ax

$$Cov(x) = ASA^T = AU\Lambda U^T A^T = (AU)\Lambda(AU)^T \tag{5}$$

We can see that in transformed space eigenvectors are also transformed by A, so from equation 3 we have:

$$Wtrans_{ML} = AU(\Lambda - \sigma^2 I)^{\frac{1}{2}}V = AW_{ML} \tag{6}$$

Model is given by:

$$x_i = z_i + \mu + \epsilon_i \tag{7}$$

By applying transformation A we can see that error ϵ_i is also transformed by A

$$Ax_i = Az_i + A\mu + A\epsilon_i \tag{8}$$

As Φ is a covariance of the error we know (from lecture) that

$$\Phi trans = A\Phi A^T \Rightarrow \Phi trans_{ML} = A\Phi_{ML}A^T \tag{9}$$

Finally by assuming orthogonality of A and $\Phi = \sigma^2 I$ we can show that if noise ϵ_i of original space has distribution $N(0, \sigma^2 I)$ then this property is preserved in transformed space.

$$A\Phi A^T = A\sigma^2 I A^T = \sigma^2 AA^T = \sigma^2 I \tag{10}$$

2 Problem 2

Projected data can be obtained by computing $M * V$. However we are only interested in prediction for Leslie in concept space, so we can multiply just the new row by V. The result of this computation is $1x2$ vector $[1.74, 2.84]$ which gives us information how much Leslie would like movies from both of concepts.

3 Problem 3

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline
```

3.1 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. Download the notebook in HTML (click File > Download as > .html) 3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf or `wkhtmltopdf` for Linux (tutorial) 4. Concatenate your solutions for other tasks with the output of Step 3. On a Linux machine you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

3.2 PCA Task

Given the data in the matrix X your tasks is to: * Calculate the covariance matrix Σ. * Calculate eigenvalues and eigenvectors of Σ. * Plot the original data X and the eigenvectors to a single diagram. What do you observe? Which eigenvector corresponds to the smallest eigenvalue? * Determine the smallest eigenvalue and remove its corresponding eigenvector. The remaining eigenvector is the basis of a new subspace. * Transform all vectors in X in this new subspace by expressing all vectors in X in this new basis.

3.2.1 The given data X

```
In [5]: X = np.array([(-3,-2),(-2,-1),(-1,0),(0,1),
                      (1,2),(2,3),(-2,-2),(-1,-1),
                      (0,0),(1,1),(2,2), (-2,-3),
                      (-1,-2),(0,-1),(1,0), (2,1),(3,2)])
```

3.2.2 Task 1: Calculate the covariance matrix Σ

```
In [20]: def get_covariance(X):
             """Calculates the covariance matrix of the input data.

             Parameters
             ----------
             X : array, shape [N, D]
                 Data matrix.

             Returns
             -------
             Sigma : array, shape [D, D]
                 Covariance matrix
```

```
    """
    return np.cov(X, rowvar=False)
```

3.2.3 Task 2: Calculate eigenvalues and eigenvectors of Σ.

```
In [3]: def get_eigen(S):
            """Calculates the eigenvalues and eigenvectors of the input matrix.

            Parameters
            ----------
            S : array, shape [D, D]
                Square symmetric positive definite matrix.

            Returns
            -------
            L : array, shape [D]
                Eigenvalues of S
            U : array, shape [D, D]
                Eigenvectors of S

            """
            return np.linalg.eig(S)
```

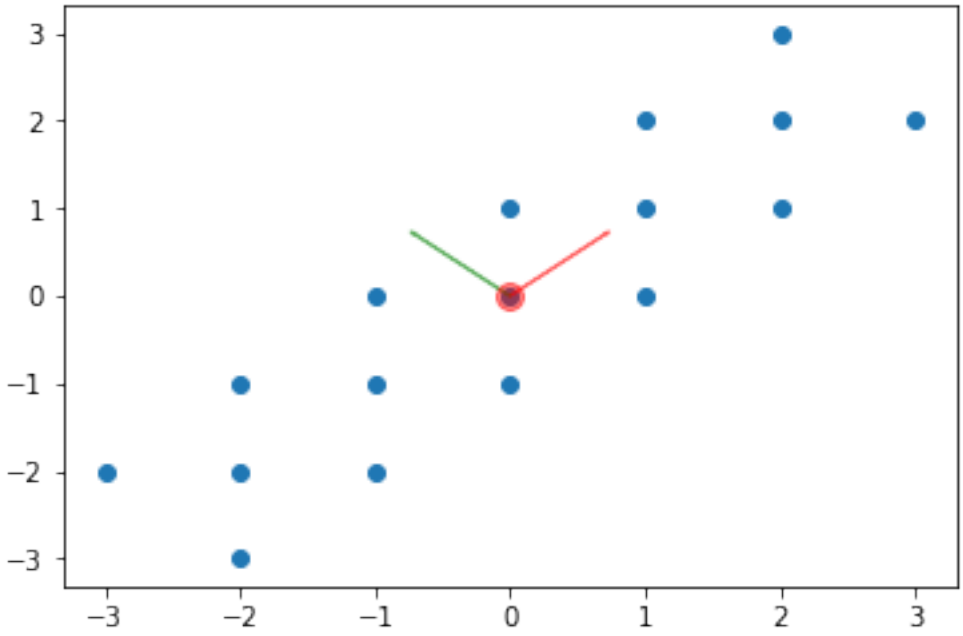
3.2.4 Task 3: Plot the original data X and the eigenvectors to a single diagram.

Note that, in general if u_i is an eigenvector of the matrix M with eigenvalue λ_i then $\alpha \cdot u_i$ is also an eigenvector of M with the same eigenvalue λ_i, where α is an arbitrary scalar (including $\alpha = -1$).

Thus, the signs of the eigenvectors are arbitrary, and you can flip them without changing the meaning of the result. Only their direction matters. The particular result depends on the algorithm used to find them.

```
In [24]: # plot the original data
         plt.scatter(X[:, 0], X[:, 1])

         # plot the mean of the data
         mean_d1, mean_d2 = X.mean(0)
         plt.plot(mean_d1, mean_d2, 'o', markersize=10, color='red', alpha=0.5)
         # calculate the covariance matrix
         Sigma = get_covariance(X)
         # calculate the eigenvector and eigenvalues of Sigma
         L, U = get_eigen(Sigma)
         plt.arrow(mean_d1, mean_d2, U[0, 0], U[1, 0], width=0.01, color='red', alpha=0.5)
         plt.arrow(mean_d1, mean_d2, U[0, 1], U[1, 1], width=0.01, color='green', alpha=0.5);
```



What do you observe in the above plot? Which eigenvector corresponds to the smallest eigenvalue?
Write your answer here:
Green vector corresponds to smaller eigenvalue as it points to the direction of lower variance.

3.2.5 Task 4: Transform the data

Determine the smallest eigenvalue and remove its corresponding eigenvector. The remaining eigenvector is the basis of a new subspace. Transform all vectors in X in this new subspace by expressing all vectors in X in this new basis.

```
In [25]: def transform(X, U, L):
             """Transforms the data in the new subspace spanned by the eigenvector corresponding to the

             Parameters
             ----------
             X : array, shape [N, D]
                 Data matrix.
             L : array, shape [D]
                 Eigenvalues of Sigma_X
             U : array, shape [D, D]
                 Eigenvectors of Sigma_X

             Returns
             -------
             X_t : array, shape [N, 1]
                 Transformed data

             """
             largest_index = np.argmax(L)
             largest_eigenvector = U[:, largest_index]
             return X@largest_eigenvector
```

```
In [37]: X_t = transform(X, U, L)
```

3.3 Task SVD

3.3.1 Task 5: Given the matrix M find its SVD decomposition $M = U \cdot \Sigma \cdot V$ and reduce it to one dimension using the approach described in the lecture.

```
In [36]: M = np.array([[1, 2], [6, 3],[0, 2]])
```

```
In [63]: def reduce_to_one_dimension(M):
             """Reduces the input matrix to one dimension using its SVD decomposition.

             Parameters
             ----------
             M : array, shape [N, D]
                 Input matrix.

             Returns
             -------
             M_t: array, shape [N, 1]
                 Reduce matrix.

             """
             U,S,V = np.linalg.svd(M,full_matrices=False)
             return M*V[0,:]
```

```
In [65]: M_t = reduce_to_one_dimension(M)
```

```
In [ ]:
```

4 Problem 4

```
In [1]: 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
```

4.1 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. Download the notebook in HTML (click File > Download as > .html) 3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf or `wkhtmltopdf` for Linux (tutorial) 4. Concatenate your solutions for other tasks with the output of Step 3. On a Linux machine you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

4.2 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 coeficients of the linear regression.

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

```
In [2]: ratings = np.load("ratings.npy")

In [3]: # We have triplets of (user, restaurant, rating).
        ratings

Out[3]: array([[101968,   1880,      1],
               [101968,    284,      5],
               [101968,   1378,      2],
               ...,
               [ 72452,   2100,      4],
               [ 72452,   2050,      5],
               [ 74861,   3979,      5]])
```

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.

```
In [4]: 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_restaurant
        M

Out[4]: <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.

```
In [5]: def cold_start_preprocessing(matrix, min_entries):
            """
            Recursively removes rows and columns from the input matrix which have less than min_ent
```

```
    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
```

4.2.2 Task 1: Implement a function that substracts the mean user rating from the sparse rating matrix

```
In [6]: 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

            """

            # YOUR CODE HERE
            user_means = matrix.mean(1)
            matrix = matrix - user_means
```

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

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

In [7]:
```
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 h

    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
```

In [8]:
```
M = cold_start_preprocessing(M, 20)
```

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

```
In [9]: 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)
```

```
In [10]: # 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
```

4.2.4 Compute the loss function (nothing to do here)

```
In [11]: 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
             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.
             return mean_sse_loss + regularization_loss
```

4.3 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.

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

```
In [12]: 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
             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)
    if init=='random':
        Q = np.random.random((matrix.shape[0], k))
        P = np.random.random((k, matrix.shape[1]))
    else:
        Q,_,P = svds(matrix, full_matrices=False)

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

4.3.2 Task 3: Implement the alternating optimization approach

```
In [38]: 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 pati
    i.e. we stop training after we observe no improvement on the validation loss for a cer
    amount of training steps. We then return the best values for Q and P oberved during tr

    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
                        nnz refers to the number of non-zero entries. Note that this may b
                        from the number of non-zero entries in the input matrix M, e.g. in
                        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
                    no improvement on the validation error for a specified number of s
                    (see "patience" for details).

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

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 lo
                    iterations (see eval_every for details). After we stop training, w
                    observed values for Q and P (based on the validation loss) and ret

eval_every        : int, optional, default: 1
                    Evaluate the training and validation loss every X steps. If we obs
                    of the validation error, we decrease our patience by 1, else we re

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

best_P            : np.array, shape [k, D]
                    Best value for P (based on validation loss) observed during traini

validation_losses : list of floats
                    Validation loss for every evaluation iteration, can be used for pl
                    loss over time.

train_losses      : list of floats
                    Training loss for every evaluation iteration, can be used for plot
                    loss over time.

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

"""
validation_losses = []
train_losses = []
reg = Ridge(alpha=reg_lambda, fit_intercept=False)
Q,P = initialize_Q_P(M, k, init)
t = 0
```

```
    current_p = 0
    converged_after = -1
    best_Q, best_P = Q, P
    best_valid_lost = float('inf')
    w_indexes = []
    h_indexes = []
    for el in nonzero_indices:
        w_indexes.append(el[0])
        h_indexes.append(el[1])
    m_idx = (np.array(w_indexes), np.array(h_indexes))
    for step in range(max_steps):
        if current_p == patience:
            converged_after = step - patience*eval_every
            break;
        P = reg.fit(X=Q, y=M).coef_
        Q = reg.fit(X=P, y=M.T).coef_
        if step % eval_every == 0:
            pred = Q@P.T
            train_losses.append(((M[m_idx].T - pred[m_idx])**2).mean())
            val_loss = ((val_values - pred[val_idx])**2).mean()
            validation_losses.append(val_loss)
            if val_loss < best_valid_lost:
                best_valid_lost = val_loss
                best_P = P
                best_Q = Q
            if validation_losses[len(validation_losses) - 1] <= val_loss:
                current_p = current_p + 1
            else:
                current_pent_p = 0
        if step % log_every == 0:
            print("Iteration {}, training loss: {}, validation loss: {}",
                  step, train_losses[len(train_losses) -1],
                  validation_losses[len(validation_losses) -1])
    return best_Q, best_P, validation_losses, train_losses, converged_after
```

4.3.3 Train the latent factor (nothing to do here)

```
In [ ]: 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)
```

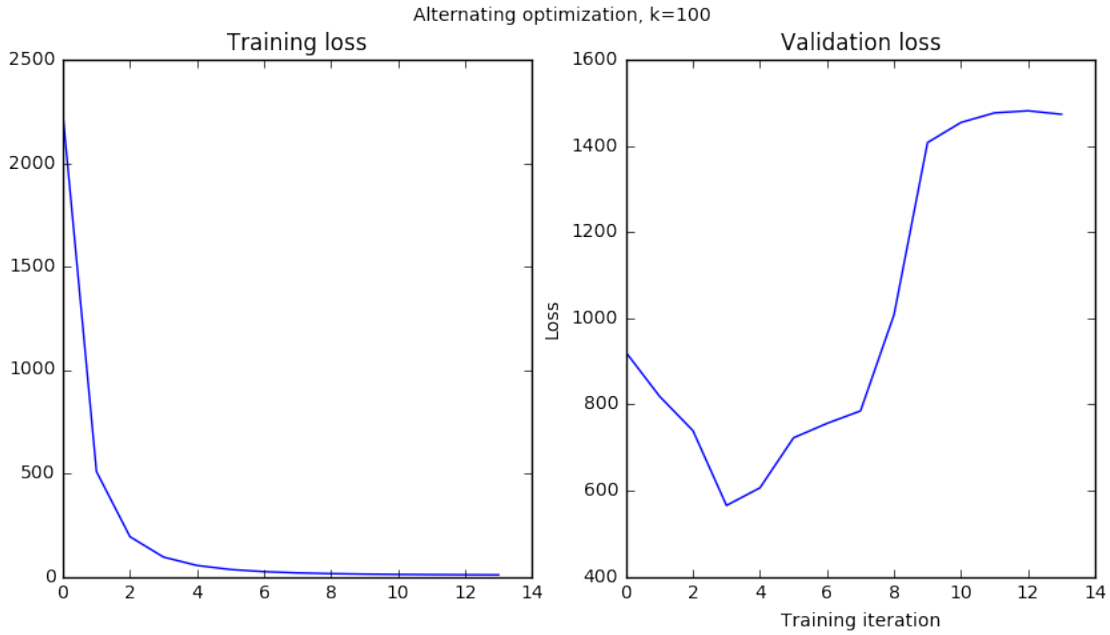
4.3.4 Plot the validation and training losses over for each iteration (nothing to do here)

```
In [15]: 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()
```



I didn't manage to finish *latent_factor_alternating_optimization* function implementation.

5 Problem 5

With identity activation function autoencoder is not able to approximate non-linear function. Thus in order for K dimensional reduction to give 0 reconstruction error all the data points in D would have to exist in lower dimension K perfectly, that is without any noise. Even if on point is slightly "off" the hyperplane this information is lost during encoding. Usually it is the case that collected data contain some noise.

Perfect reconstruction is only possible if $D - K$ dimensions are linear combinations of the other K dimensions.