# PCA & SVD

## Q1:

Assume we have $\boldsymbol{u}_M$ vectors define the sub-space, thus we get orthonormality properties of them.

$$\boldsymbol{u}_i^T \boldsymbol{u}_j = 0 \text{ for } i \neq j$$

and

$$\boldsymbol{u}_i^T \boldsymbol{u}_i = 1$$

The variance of the projected data is $\boldsymbol{u}_{M+1}^T \boldsymbol{S} \boldsymbol{u}_{M+1}$ where $\boldsymbol{S}$ is the data covariance matrix, and we need to maximize the projected variance with the constraints of the orthonormality properties. Applying the Lagrange multiplier, we get

$$\boldsymbol{u}_{M+1}^T \boldsymbol{S} \boldsymbol{u}_{M+1} + \lambda_1(1 - \boldsymbol{u}_1^T \boldsymbol{u}_1) + ... + \lambda_{M+1}(1 - \boldsymbol{u}_{M+1}^T \boldsymbol{u}_{M+1}) + \lambda_{M+2} \boldsymbol{u}_1^T \boldsymbol{u}_{M+1} + ... + \lambda_{2M+2} \boldsymbol{u}_M^T \boldsymbol{u}_{M+1}$$

We denote the previous Lagrangian expression as $A$, and then take the derivative respect to $\boldsymbol{u}_{M+1}$,

$$\frac{\partial A}{\partial \boldsymbol{u}_{M+1}} = 2\boldsymbol{S}\boldsymbol{u}_{M+1} - 2\lambda_{M+1}\boldsymbol{u}_{M+1} + \lambda_{M+2}\boldsymbol{u}_1^T + ... + \lambda_{2M+2}\boldsymbol{u}_M^T \overset{!}{=} 0 \tag{1}$$

$$2\boldsymbol{S}\boldsymbol{u}_{M+1} + \lambda_{M+2}\boldsymbol{u}_1^T + ... + \lambda_{2M+2}\boldsymbol{u}_M^T = 2\lambda_{M+1}\boldsymbol{u}_{M+1}$$

We multiply $\boldsymbol{u}_M$ at both sides of the equation, since $\boldsymbol{u}_{1:M}$ has orthonormality, then we get:

$$\boldsymbol{S}\boldsymbol{u}_{M+1} = \lambda_{M+1}\boldsymbol{u}_{M+1}$$

this indicates that the new vector $\boldsymbol{u}_{M+1}$ is an eigenvector of $\boldsymbol{S}$, and the variance is given by

$$\boldsymbol{u}_{M+1}^T \boldsymbol{S} \boldsymbol{u}_{M+1} = \lambda_{M+1}$$

and so the variance will be maximum when we set $\boldsymbol{u}_{M+1}$ to be the first principal component which has the largest eigenvalue.

## Q2:

Let the new rating vector be:

$$\boldsymbol{m} = [0, 3, 0, 0, 4]$$

the prediction from the recommend system gives by

$$\boldsymbol{p} = \boldsymbol{m}\boldsymbol{V}^T = [1.74, 2.84]$$

which means Leslie may be more interested in the class (whatever the name is) of movies which "Casablanca" and "Titanic" belong to than the other.

## Q3:

# homework_10_dim_reduction_notebook

January 20, 2019

## 1 Programming assignment 10: Dimensionality Reduction

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline
```

### 1.1 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.

### 1.2 PCA Task

Given the data in the matrix X your tasks is to: * Calculate the covariance matrix $\Sigma$. * Calculate eigenvalues and eigenvectors of $\Sigma$. * 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.

#### 1.2.1 The given data X

```
In [9]: 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)])

In [12]: X.shape

Out[12]: (17, 2)
```

### 1.2.2 Task 1: Calculate the covariance matrix Σ

In [13]:
```
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

    """
    # TODO
    Sigma = np.cov(X.T)
    return Sigma
```

### 1.2.3 Task 2: Calculate eigenvalues and eigenvectors of Σ.

In [15]:
```
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

    """
    # TODO
    L, U = np.linalg.eig(S)
    return L, U
```

In [17]:
```
get_eigen(get_covariance(X))
```

Out[17]:
```
(array([5.625, 0.375]), array([[ 0.70710678, -0.70710678],
        [ 0.70710678,  0.70710678]]))
```

### 1.2.4 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 $\lambda_i$ then $\alpha \cdot u_i$ is also an eigenvector of $M$ with the same eigenvalue $\lambda_i$, where $\alpha$ is an arbitrary scalar (including $\alpha = -1$).

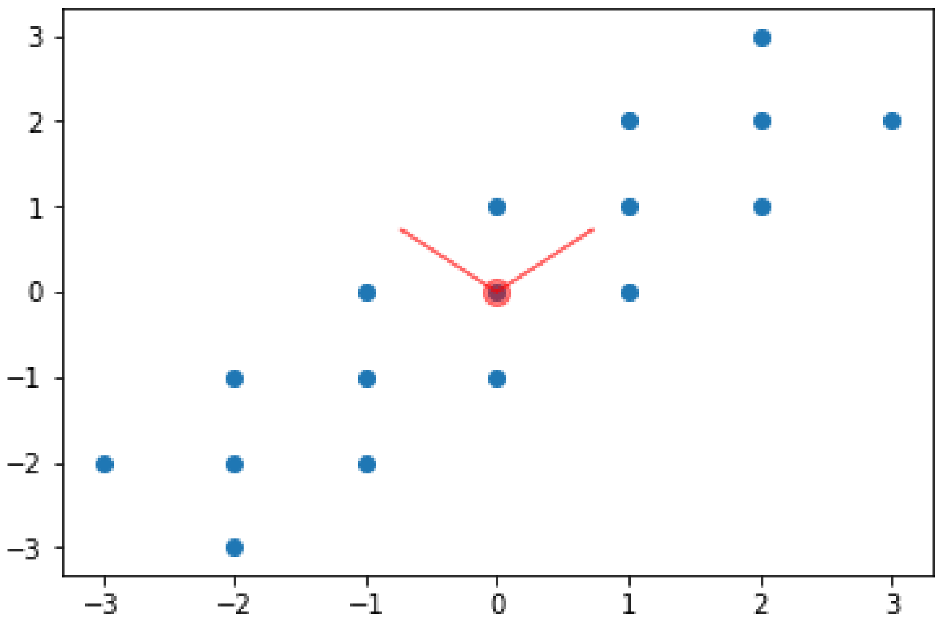
Thus, the signs of the eigenvectors are arbitrary, and you can flip them without changing the meaning of the result. Only their direction matters. The particular result depends on the algorithm used to find them.

In [19]:
```
# 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='red', alpha=0.5);
```




What do you observe in the above plot? Which eigenvector corresponds to the smallest eigenvalue?

Write your answer here:

[YOUR ANSWER] There are two vectors point at two directions. The smallest eigenvalue point at the upper left, which means the in this direction, the data are closest to each other than any other direction in the domain.

#### 1.2.5 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 correspondi

             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

             """
             # TODO
             idx = L.argsort()[::-1]
             L_sorted = L[idx]
             U_sorted = U[:,idx]
             X_t = X @ U_sorted[-1]

             return X_t
```

```
In [26]: X_t = transform(X, U, L)
```

```
In [27]: print(X_t)
```

```
[-3.53553391 -2.12132034 -0.70710678  0.70710678  2.12132034  3.53553391
 -2.82842712 -1.41421356  0.          1.41421356  2.82842712 -3.53553391
 -2.12132034 -0.70710678  0.70710678  2.12132034  3.53553391]
```

### 1.3 Task SVD

#### 1.3.1 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 [28]: M = np.array([[1, 2], [6, 3],[0, 2]])
```

In [29]:
```
U, S, V = np.linalg.svd(M, 0)
S_diag = np.diag(S)
print(S_diag)
```

```
[[7.02571561 0.        ]
 [0.         2.15390813]]
```

In [42]:
```
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.

    """
    # TODO
    U, S, V = np.linalg.svd(M, 0)
    V_trun = V[:, 0]
    M_t = M @ V_trun
    return M_t
```

In [43]:
```
M_t = reduce_to_one_dimension(M)
```

In [44]:
```
print(M_t)
```

```
[-1.90211303 -6.68109819 -1.05146222]
```

# Matrix Factorization

## Q4:

# Programming assignment 10: Matrix Factorization

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
```

## 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 (https://www.sejda.com/html-to-pdf) or `wkhtmltopdf` for Linux (tutorial (https://www.cyberciti.biz/open-source/html-to-pdf-freeware-linux-osx-windows-software/))
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.

## 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 $\lambda$ 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.

## 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_restaurants)).tocs
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 (https://en.wikipedia.org/wiki/Cold_start_(computing))](https://en.wikipedia.org/wiki/Cold_start_(computing)), 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_entries nonz

    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
```

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

In [6]:

```python
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

    """

    # YOUR CODE HERE
    user_means = np.mean(matrix, axis=1)
    user_means = user_means.reshape(-1, 1) # reshape to [N, 1]

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

## 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 have been

    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
```

In [19]:

```
print(M_train.shape)
M_shifted.shape
```

```
(3529, 2072)
```

Out[19]:

```
(3529, 2072)
```

## 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 the unshif
    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.nor

    return mean_sse_loss + regularization_loss
```

# 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.

### 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 Q, 'randon
             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)
    # YOUR CODE HERE

    N = matrix.shape[0]
    D = matrix.shape[1]
    if init == 'random':
        Q = np.random.rand(N, k)/k
        P = np.random.rand(k, D)/k
    elif init == 'svd':
        U, S, V = np.linalg.svd(matrix, full_matrices=False)
        Q = U*S
        P = V

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

## Task 3: Implement the alternating optimization approach

In [15]:

```
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 factorize
                        nnz refers to the number of non-zero entries. Note that this may be differen
                        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 det

    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 e
                        iterations (see eval_every for details). After we stop training, we restore
                        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 im
                        of the validation error, we decrease our patience by 1, else we reset it to

    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
                        loss over time.

    train_losses      : list of floats
                        Training loss for every evaluation iteration, can be used for plotting the t
                        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.

    """

    # YOUR CODE HERE
    # initialize the params
    Q, P = initialize_Q_P(M, k, init)
    best_Q = Q
    best_P = P
    lost_patience = 0
    it = 0
    validation_losses=[]
    train_losses=[]
    clf_p = Ridge(reg_lambda, fit_intercept=False, solver='auto')
    clf_q = Ridge(reg_lambda, fit_intercept=False, solver='auto')

    idx_x = non_zero_idx[:,0]
    idx_y = non_zero_idx[:,1]
    value = M[idx_x, idx_y]
    value = np.asarray(value).reshape(-1)

    idx_tuple = non_zero_idx.transpose()
    idx_tuple = tuple(map(tuple, idx_tuple))
    # print(value.shape)

    # start trainning
    while lost_patience<patience:

        it += 1
        current_train_loss = loss(value, idx_tuple, Q, P, reg_lambda)
        train_losses.append(current_train_loss)

        if it%eval_every==0:
            current_val_loss = loss(val_values, val_idx, Q, P, reg_lambda)
            validation_losses.append(current_val_loss)

        if len(train_losses)>=2 and train_losses[-1]>=train_losses[-2]:
            lost_patience += 1
        else:
            lost_patience = 0

        if len(train_losses)>=2 and train_losses[-1]<train_losses[-2]:
            best_Q = Q
            best_P = P

        # update Q
        clf_q.fit(P.transpose(), M.transpose())
        Q = clf_q.coef_

        # update P
```

```
        clf_p.fit(Q, M)
        P = clf_p.coef_.transpose()

        converged_after = it - patience*eval_every
        print("Iteration %s,    trainning loss %.2f,    validation loss %.2f" % (it,
                                                                                  train_losses[-1],
                                                                                  validation_losses[-1]))

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

### Train the latent factor (nothing to do here)

In [16]:

```
Q, P, val_loss, train_loss, converged = latent_factor_alternating_optimization(M_shifted, nonzero_ir
                                                                             k=100, val_idx=val_ic
                                                                             val_values=val_values
                                                                             reg_lambda=1e-4, init
                                                                             max_steps=100, patier
```

```
Iteration 1,    trainning loss 2218582.81,    validation loss 3019.77
Iteration 2,    trainning loss 1577870.02,    validation loss 3532.65
Iteration 3,    trainning loss 1306264.02,    validation loss 3733.20
Iteration 4,    trainning loss 1266097.77,    validation loss 3839.26
Iteration 5,    trainning loss 1255187.13,    validation loss 3880.03
Iteration 6,    trainning loss 1251088.68,    validation loss 3893.93
Iteration 7,    trainning loss 1249164.49,    validation loss 3897.90
Iteration 8,    trainning loss 1248111.33,    validation loss 3898.21
Iteration 9,    trainning loss 1247467.32,    validation loss 3897.06
Iteration 10,    trainning loss 1247035.46,    validation loss 3895.21
Iteration 11,    trainning loss 1246722.09,    validation loss 3893.01
Iteration 12,    trainning loss 1246480.16,    validation loss 3890.59
Iteration 13,    trainning loss 1246284.90,    validation loss 3888.04
Iteration 14,    trainning loss 1246122.53,    validation loss 3885.41
Iteration 15,    trainning loss 1245984.76,    validation loss 3882.73
Iteration 16,    trainning loss 1245866.22,    validation loss 3880.01
Iteration 17,    trainning loss 1245763.15,    validation loss 3877.28
Iteration 18,    trainning loss 1245672.77,    validation loss 3874.53
Iteration 19,    trainning loss 1245592.94,    validation loss 3871.77
Iteration 20,    trainning loss 1245521.97,    validation loss 3869.01
Iteration 21,    trainning loss 1245458.48,    validation loss 3866.24
Iteration 22,    trainning loss 1245401.41,    validation loss 3863.48
Iteration 23,    trainning loss 1245349.86,    validation loss 3860.71
Iteration 24,    trainning loss 1245303.14,    validation loss 3857.94
Iteration 25,    trainning loss 1245260.66,    validation loss 3855.16
Iteration 26,    trainning loss 1245221.96,    validation loss 3852.38
Iteration 27,    trainning loss 1245186.64,    validation loss 3849.60
Iteration 28,    trainning loss 1245154.39,    validation loss 3846.81
Iteration 29,    trainning loss 1245124.92,    validation loss 3844.01
Iteration 30,    trainning loss 1245098.00,    validation loss 3841.21
Iteration 31,    trainning loss 1245073.40,    validation loss 3838.40
Iteration 32,    trainning loss 1245050.95,    validation loss 3835.59
Iteration 33,    trainning loss 1245030.47,    validation loss 3832.78
Iteration 34,    trainning loss 1245011.80,    validation loss 3829.97
Iteration 35,    trainning loss 1244994.81,    validation loss 3827.16
Iteration 36,    trainning loss 1244979.36,    validation loss 3824.34
Iteration 37,    trainning loss 1244965.33,    validation loss 3821.53
Iteration 38,    trainning loss 1244952.61,    validation loss 3818.73
Iteration 39,    trainning loss 1244941.10,    validation loss 3815.93
Iteration 40,    trainning loss 1244930.70,    validation loss 3813.13
Iteration 41,    trainning loss 1244921.33,    validation loss 3810.34
Iteration 42,    trainning loss 1244912.90,    validation loss 3807.56
Iteration 43,    trainning loss 1244905.34,    validation loss 3804.79
Iteration 44,    trainning loss 1244898.57,    validation loss 3802.02
Iteration 45,    trainning loss 1244892.55,    validation loss 3799.27
Iteration 46,    trainning loss 1244887.20,    validation loss 3796.52
Iteration 47,    trainning loss 1244882.47,    validation loss 3793.79
Iteration 48,    trainning loss 1244878.31,    validation loss 3791.06
Iteration 49,    trainning loss 1244874.68,    validation loss 3788.35
Iteration 50,    trainning loss 1244871.54,    validation loss 3785.65
Iteration 51,    trainning loss 1244868.83,    validation loss 3782.96
```

```
Iteration 52,   trainning loss 1244866.54,   validation loss 3780.28
Iteration 53,   trainning loss 1244864.61,   validation loss 3777.62
Iteration 54,   trainning loss 1244863.03,   validation loss 3774.97
Iteration 55,   trainning loss 1244861.77,   validation loss 3772.33
Iteration 56,   trainning loss 1244860.79,   validation loss 3769.70
Iteration 57,   trainning loss 1244860.08,   validation loss 3767.09
Iteration 58,   trainning loss 1244859.60,   validation loss 3764.49
Iteration 59,   trainning loss 1244859.35,   validation loss 3761.91
Iteration 60,   trainning loss 1244859.30,   validation loss 3759.33
Iteration 61,   trainning loss 1244859.43,   validation loss 3756.78
Iteration 62,   trainning loss 1244859.73,   validation loss 3754.23
Iteration 63,   trainning loss 1244860.18,   validation loss 3751.70
Iteration 64,   trainning loss 1244860.76,   validation loss 3749.19
Iteration 65,   trainning loss 1244861.48,   validation loss 3746.68
Iteration 66,   trainning loss 1244862.30,   validation loss 3744.20
Iteration 67,   trainning loss 1244863.23,   validation loss 3741.72
Iteration 68,   trainning loss 1244864.25,   validation loss 3739.26
Iteration 69,   trainning loss 1244865.35,   validation loss 3736.82
Iteration 70,   trainning loss 1244866.53,   validation loss 3734.38
```

In [1]:

```
# I am sure there is sth wrong in the model, but I can't find it.
```

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

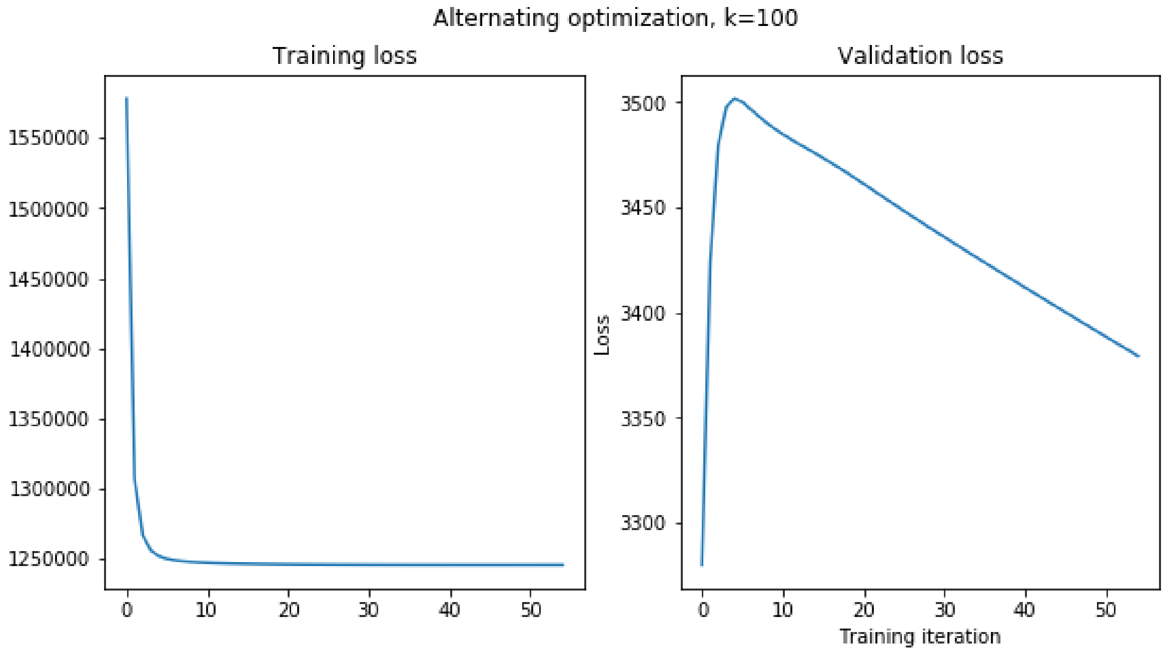
In [199]:

```
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()
```




In [ ]:

```

```

# Autoencoders

## Q5:

1. Since the activation functions are identity, and the dimensions of NN is smaller than the data dimensions, this simple means the NN try to copy the data but without enough memory to hold all the information, thus, it will lost some part of information. Thus, the reconstruction can only rely on this “incomplete” copy and the error won’t be zero.

2. The original data can be fully and densely represented by a data set with lower dimensions (e.g less or equal than K), then it is possible.