MACHINE LEARNING HOMEWORK SHEET -10
DIMENSIONALITY REDUCTION & MATRIX FACTORIZATION

NAME: MAHALAKSHMI SABANAYAGAM
TUM ID: ge73yuw
DATE: 20.01.2019

1. PCA & SVD.

Problem 1: Latent space distribution $p(z) = \mathcal{N}(z|0,I)$

Conditional distribution for $x|z$, $x \in R^d$

$$p(x|z) = \mathcal{N}(x|Wz+\mu, \phi).$$

ϕ – symmetric, positive definite noise covariance matrix.

μ_{ML}, W_{ML}, ϕ_{ML} – Max. likelihood solution for x.

Transformation on x $(y) = Ax$ where A is non singular $d \times d$ matrix

To prove:

i.) $A\mu_{ML}$, AW_{ML}, $A\phi_{ML}A^T$ are coresponding max. likelihood solutions for y.

ii.) To preserve the original model A should be orthogonal and $\phi = \sigma^2 I$.

i.) Log likelihood for data x is

$$LL_x = -\frac{N}{2}\left[d\ln(2\pi) + \ln|WW^T+\phi|\right] - \frac{1}{2}\sum_{n=1}^{N}(x_n-\mu)^T(WW^T+\phi)^{-1}(x_n-\mu)$$

For transformed data $y = Ax$,

let μ be μ'

N be W'

ϕ be ϕ'

$$LL_y = -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|W'W'^T+\phi'| - \frac{1}{2}\sum_{n=1}^{N}(Ax_n-\mu')^T(W'W'^T+\phi')^{-1}(Ax_n-\mu')$$

$$\mu' = \frac{1}{N}\sum_{i=1}^{N}Ax_i = A\frac{1}{N}\sum_{i=1}^{N}x_i = A\mu_{ML}.$$

$$\therefore LL_y = -\frac{Nd}{2}\ln(2\pi) - \frac{N}{2}\ln|W'W'^T + \phi'| - \frac{1}{2}\sum_{n=1}^{N}(Ax_n - A\mu_{ML})^T (W'W'^T + \phi')^{-1}(Ax_n - A\mu_{ML})$$

$$= -\frac{Nd}{2}\ln(2\pi) - \frac{N}{2}\ln|W'W'^T + \phi'| - \frac{1}{2}\sum (x_n - \mu_{ML})^T A^T (W'W'^T + \phi')^{-1} A(x_n - \mu_{ML})$$

Now, substituting $W' = AW_{ML}$ & $\phi' = A\phi_{ML}A^T$ into LL_y,

$$= -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|AW_{ML}W_{ML}^T A^T + A\phi_{ML}A^T| -$$
$$\frac{1}{2}\sum_{n=1}^{N}\left[(x_n - \mu_{ML})^T A^T (AW_{ML}W_{ML}^T A^T + A\phi_{ML}A^T)^{-1} A(x_n - \mu_{ML})\right]$$

$$= -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|A(W_{ML}W_{ML}^T + \phi_{ML})A^T| -$$
$$\frac{1}{2}\sum_{n=1}^{N}\left[(x_n - \mu_{ML})^T A^T [A(W_{ML}W_{ML}^T + \phi_{ML})A^T]^{-1} A(x_n - \mu_{ML})\right]$$

$$= -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|A| - \frac{N}{2}\ln|W_{ML}W_{ML}^T + \phi_{ML}| - \frac{N}{2}\ln|A^T| -$$
$$\frac{1}{2}\sum_{n=1}^{N}\left[(x_n - \mu_{ML})^T A^T \underbrace{(A^T)^{-1}}_{I} (W_{ML}W_{ML}^T + \phi_{ML})^{-1} \underbrace{A^{-1}A}_{I} (x_n - \mu_{ML})\right]$$

$$= -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|A| - \frac{N}{2}\ln|A| - \frac{N}{2}\ln|W_{ML}W_{ML}^T + \phi_{ML}| -$$
$$\frac{1}{2}\sum_{n=1}^{N}(x_n - \mu_{ML})^T (W_{ML}W_{ML}^T + \phi_{ML})^{-1}(x_n - \mu_{ML})$$

$$= -\frac{N}{2}d\ln(2\pi) - \frac{N}{2}\ln|W_{ML}W_{ML}^T + \phi_{ML}| - \frac{1}{2}\sum_{n=1}^{N}\left[(x_n - \mu_{ML})^T (W_{ML}W_{ML}^T + \phi_{ML})^{-1}\right.$$
$$\left.(x_n - \mu_{ML})\right] - N\ln|A|.$$

Thus, LL_y is of the same form as LL_x. This shows the chosen W' & ϕ' are infact the max likelihood for y.

$$\Rightarrow \mu' = A\mu_{ML} \qquad \phi' = A\phi_{ML}A^T$$
$$W' = AW_{ML}$$

ii.) to preserve the original model,

(ie) $LL_x = LL_y$ we have $-N \ln|A|$ extra in LL_y. For it to be 0, $|A| = 1$.

If A is an orthogonal matrix, $A^T A = I$

$\therefore |A^T A| = 1$

$|A^T||A| = 1$

$|A|^2 = 1 \quad \because |A^T| = |A|$

$|A| = \pm 1$.

$\Rightarrow$ when A is orthogonal, $LL_y = LL_x$ $[\because -N \ln|A| = 0]$

Also, while considering log likelihood for x, we assumed the noise as $\sigma^2 I$. So Φ to preserve the original model, $\Phi = \sigma^2 I$.

Problem 2:

New I/P. [0, 3, 0, 0, 4]

Concept space is data. V

$$= [0 \ 3 \ 0 \ 0 \ 4] \begin{bmatrix} 0.58 & 0 \\ 0.58 & 0 \\ 0.58 & 0 \\ 0 & 0.71 \\ 0 & 0.71 \end{bmatrix} = [1.74 \ \ 2.84]$$

The values 1.74 and 2.84 are the projection of Leslie on to the concept space. The values show the strength of each concept (ie) how strongly Leslie prefers the movies in one particular concept.

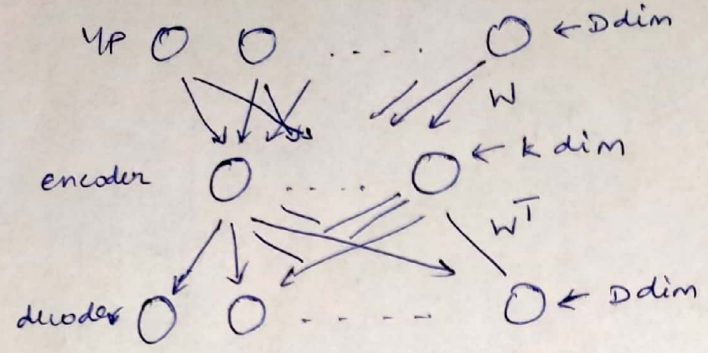
$\therefore$ From the concept space, we can say Leslie will like Casablanca more than Matrix and StarWars as we got 2.84 for second concept (> 1.74). Second concept includes Titanic & Casablanca.

5. Linear Autoencoder. (D dimension).

single k dimension hidden layer. No biases & activation function $\sigma(x) = x$.

i) Impossible to get 0 reconstruction error if $k < D$:

Considering one layer,



Let x be the input with D dimensions,
y be the latent code with k dimensions,
x' be the final reconstructed output with D dimensions.
W, be the weights.

$$\therefore \quad y = f(x, W)$$
$$= xW \quad \text{(linear autoencoder and identity activation)}$$
$$x' = f(y, W^T)$$
$$= yW^T = xWW^T$$

For reconstruction error to be 0, $x = x'$.

(ie) $x = xWW^T$

This can happen only if W is orthogonal matrix.
W could be orthogonal if the k largest eigen vectors of α is chosen. But, it will not reconstruct the input without any error as we are picking only k vectors out of D ($k < D$).

ii.) It will be possible to have an architecture which gives 0 reconstruction error when $k \geq D$.

a.) $k = D$:

In this case if W is eigen vectors then reconstruction error will be 0.

b.) $k > D$:

When $k > D$, we can have identity for first D nodes and 0 for $k-D$ nodes which will reconstruct the input x.

Programming assignment 10: Dimensionality Reduction

In [1]:

```
import numpy as np
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.

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.

The given data X

In [2]:

```
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)])
```

Task 1: Calculate the covariance matrix Σ

In [3]:

```
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
    return np.cov(X.T)
```

Task 2: Calculate eigenvalues and eigenvectors of Σ.

In [4]:

```
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.eigh(S)
    return L,U
```

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.

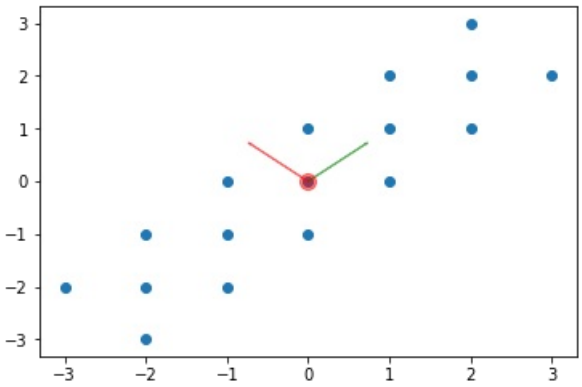
In [5]:

```
# 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)
print(L,U)
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);
```

```
[0.375 5.625] [[-0.70710678  0.70710678]
 [ 0.70710678  0.70710678]]
```



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

Write your answer here:

[From the plot, we observe that the given data is highly correlated. Eigen vectors show the direction along which the data is distributed. The vector drawn in red corresponds to the smallest eigen value as the variance along that direction is the lowest.]

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 [6]:

```
def transform(X, U, L):
    """Transforms the data in the new subspace spanned by the eigenvector corresponding to the largest eigen
value.

    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

    max_eig = np.argmax(L)
    y = np.reshape(U[max_eig], (-1,1))
    X_t = np.matmul(X,y)
    return X_t
```

In [7]:

```
X_t = transform(X, U, L)
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]]
```

Task SVD

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 [8]:

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

In [9]:

```
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, full_matrices=False)
    #print(u, u.shape)
    #print(s, s.shape)
    #print(v, v.shape)

    s_max = s[np.argmax(s)]
    s_max = np.reshape(s_max, (-1,1))
    u_modified = u[:,np.argmax(s)]
    u_modified = np.reshape(u_modified, (-1,1))
    M_t = np.matmul(u_modified,s_max)

    #Equivalent to M*v.T
    #exp = np.matmul(M,v.T)
    #print("exp ", exp)
    return M_t
```

In [10]:

```
M_t = reduce_to_one_dimension(M)
print(M_t)
```

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

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:

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

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.

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)).tocsr()
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)), 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 nonzero entr
ies.

    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]:

```
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-shi
fted ones.

    """

    # 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
```

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 set to z
ero.

    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
```

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 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.

    """
    print(values.shape)
    print(ixs.shape)
    print(Q.shape, P.shape)
    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, axi
s=1) ** 2))

    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, '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)

    # YOUR CODE HERE
    N,D = matrix.shape
    if init=='random':
        Q = np.random.random((N,k))
        P = np.random.random((k,D))

    elif init == 'svd':
        matrix = np.random.random((N,D))
        Q, D, P = svds(matrix,k)
        Q = Q*D

    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 [13]:

```
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
```

```
    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 evaluatio
n
                        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 improvemen
t
                        of the validation error, we decrease our patience by 1, else we reset it to *patienc
e*.

    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 validat
ion
                        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.

    """

    # YOUR CODE HERE
    #initialize Q,P
    Q, P = initialize_Q_P(M, k, init)

    train_losses = []
    validation_losses = []

    #find best Q, P
    M_nnz = M[non_zero_idx[:,0],non_zero_idx[:,1]]
    for i in range(max_steps):
        reg = Ridge(alpha=reg_lambda, solver='svd')
        converged_after = 0
        #while(converged_after < patience):
        reg.fit(X=Q,y=M)
        best_P = reg.coef_
        reg.fit(X=best_P, y=np.transpose(M))
        best_Q = reg.coef_

        P = np.transpose(best_P)
        Q = best_Q

        converged_after+=1
        if i % eval_every == 0:
            Mnew = np.matmul(Q,P)

            M_nnzNew = Mnew[non_zero_idx[:,0],non_zero_idx[:,1]]
            train_loss = np.sum(np.square(M_nnz-M_nnzNew))
            train_loss += reg_lambda * (np.sum(np.linalg.norm(P, axis=0)**2) + np.sum(np.linalg.norm(Q, axis
=1) ** 2))

            val_set = Mnew[val_idx[0],val_idx[1]]
            val_loss = np.sum(np.square(val_values - val_set))

            train_losses.append(train_loss)
```

```
            validation_losses.append(val_loss)


        if i % log_every == 0:
            print("Iteration {}".format(i) , "train_losses {}".format(train_loss), " validation_losses {}".f
ormat(val_loss))

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

Train the latent factor (nothing to do here)

In [14]:

```
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, l
og_every=10)
```

```
Iteration 0 train_losses 1594467.006164029  validation_losses 2810.797843438443
Iteration 10 train_losses 1247391.933411284  validation_losses 2773.9729138046264
Iteration 20 train_losses 1247103.5308121147  validation_losses 2763.329222959661
Iteration 30 train_losses 1246952.4986499017  validation_losses 2754.728381752725
Iteration 40 train_losses 1246857.4042473084  validation_losses 2754.3894316933065
Iteration 50 train_losses 1246786.3389146817  validation_losses 2756.5385137459334
Iteration 60 train_losses 1246730.0532530956  validation_losses 2759.0203560494083
Iteration 70 train_losses 1246686.89857531  validation_losses 2761.2214686431785
Iteration 80 train_losses 1246655.0136485763  validation_losses 2762.9882839729908
Iteration 90 train_losses 1246631.9296907696  validation_losses 2764.3198086159587
```

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()
```