CSE 446 HW4

Zachary McNulty

May 2019

Collaborators: Pei Lee Yap, Conrad Spiekerman, Valerie Liao

Expectation Maximization

Problem 1

(a) We can start with the standard trick of considering the log-likehood as both have the same optimal $\hat{w}$:

$$\max_w \prod_{i \in S} p(y_i \mid x_i, w)$$

$$\equiv \max_w log\left(\prod_{i \in S} p(y_i \mid x_i, w)\right)$$

$$= \max_w \sum_{i \in S} log(p(y_i \mid x_i, w))$$

$$= \max_w \sum_{i \in S} log\left(\frac{\lambda_i^{y_i}}{y_i!}e^{-\lambda_i}\right)$$

$$= \max_w \sum_{i \in S} \left(y_i log(\lambda_i) + log(e^{-\lambda_i}) - log(y_i!)\right)$$

$$= \max_w \sum_{i \in S} (y_i log(\lambda_i) - \lambda_i - log(y_i!))$$

$$= \max_w \sum_{i \in S} \left(y_i w^T x_i - e^{w^T x_i} - log(y_i!)\right)$$

Firstly, note that the $log(y_i!)$ term does not depend on w and is thus constant. Therefore we can ignore it in this maximization problem. Furthermore, as always we can rewrite this as an equivalent minimization problem by multiplying through by negative one.

$$= \min_w \sum_{i \in S} \left(e^{w^T x_i} - y_i w^T x_i\right)$$

Note that the right term is simply a linear transformation with respect to w and is thus convex (the minus sign does not effect its convexity as $-a^T x$ and $a^T x$ are both convex). The first term is a strictly nondecreasing univariate convex function, $f(z) = e^z$, composed with a linear transformation $w^T x$ which is also convex. For any univariate nondecreasing, convex $g(x)$ and convex $f(x)$ it holds that $g(f(x))$ is convex and thus $e^{w^T x}$ is convex. Furthermore, the sum of convex functions is convex and thus the entire summand is convex. The function is clearly differentiable and thus some kind of gradient descent can be used to solve for $\hat{w}$. Since n, the number of songs in the database, is likely quite large and you probably have lots of users, stochastic gradient descent should be used. As the problem is convex, all local minimizers are global minimizers so we do not have to worry about getting stuck in a local minimum.

(b) Assuming the user has k different moods, our goal will be to find a set of weights $\{w_j\}_{j=1}^k$ that best model her music preferences for each of the specific moods. Furthermore, we would like to know which of the $i \in S$ songs correspond to each mood. Define an additional variable z_{ij} for each song/mood and let $z_{ij} = 1$ if song i is suitable for mood j and zero otherwise. Assuming that for each song x_i suitable for mood j it holds that $Y_i \sim Poisson(e^{w_j^T x_i})$ our goal is to maximize:

$$\max \prod_{i \in S} \prod_{j=1}^{k} P(Y_i \mid x_i, w_j)^{I\{z_{ij}=1\}} \equiv \max \sum_{i \in S} \sum_{j=1}^{k} I\{z_{ij} = 1\} log\left(P(Y_i \mid x_i, w_j)\right) \tag{1}$$

where π_j represents the probability a point is sampled from distribution $j \in \{1, 2, ..., k\}$ when drawn randomly (i.e. the true proportion of songs that would be suitable for mood j).

Initialization

Run K-means on the set of x_i for $i \in S$. For each partition j of the songs in S initialize w_j to be the solution to the maximization problem defined in $1a$) across only this partition. Initialize $\pi_j = \frac{1}{k}$ for all j.

E step

In this step, the proportion of points drawn from each distribution π_j and the parameters of the k poisson distributions, $\lambda_j = e^{w_j^T x_i}$, are fixed for all $\jmath \in \{1, 2, ..., k\}$ (calculated from initialization or previous M step). For each song $i \in S$ and mood j in the range $\{1, 2, ..., k\}$, we simply need to compute r_{ij} using equation 2 below.

$$\boxed{r_{ij} = \frac{\pi_j P(Y_i \mid x_i, Y_i \sim Poisson(e^{w_j^T x_i}))}{\sum_{\ell=1}^{k} \pi_\ell P(Y_i \mid x_i, Y_i \sim Poisson(e^{w_\ell^T x_i}))}} \tag{2}$$

Note that:

$$P(Y_i \mid x_i, Y_i \sim Poisson(e^{w_j^T x_i})) = \frac{e^{Y_i w_j^T x_i}}{Y_i!} e^{-e^{w_j^T x_i}}$$

M Step

In this step, the relative probabilities r_{ij} are all fixed, calculated from the last round of the E step. Here, we need to calculate the new parameters w_j that help define our Poisson distributions. Furthermore, we need to estimate the overall probability a sampled point comes from each distribution, π_j (i.e. the true proportion of songs that would be suitable for mood j). Let $|S|$ be the number of songs in S.

$$\boxed{N_j = \sum_{i \in S} r_{ij}}$$

$$\boxed{\pi_j = \frac{N_j}{|S|}}$$

$$\begin{aligned} w_j &= \arg\max_w \sum_{i \in S} r_{ij} log\left(P(Y_i \mid x_i, Y_i \sim Poisson(e^{w_j^T x_i}))\right) \\ &\equiv \arg\max_w \sum_{i \in S} r_{ij}\left(y_i w^T x_i - e^{w^T x_i} - log(y_i!)\right) \\ &\rightarrow \boxed{w_j = \arg\min_w \sum_{i \in S} r_{ij}\left(e^{w^T x_i} - y_i w^T x_i\right)} \end{aligned}$$

where N_j is just an intermediate term used in calculation. For w_j, we showed in part a) above that $e^{w^T x_i} - y_i w^T x_i$ is convex and thus naturally its weighted sum with nonnegative coefficients (as $r_{ij} \geq 0$ by definition) is naturally also convex. Thus we can solve for w_j using traditional convex optimization techniques like SGD.

Convergence Criteria

Define a hyperparameter $\delta > 0$ that will be tuned according to the data. Stop the EM algorithm when the log-likelihood (seen in the equation below) changes by less than δ in a given iteration.

$$\sum_{i \in S} \sum_{j=1}^{k} r_{ij} log \left(P(Y_i \mid x_i, Y_i \sim Poisson(e^{w_j^T x_i})) \right)$$

Recommendation System

Problem 2

(a) Since this predictor outputs the same prediction, the mean rating of that movie among users who rated that movie, for all users, it is clear that all columns of $\hat{R}$ will be identical. Thus we can write $\hat{R}$ as a rank one matrix in the following way (Let $\mathbf{1} \in \mathbb{R}^n$ be the vector of all ones and $\mu \in \mathbb{R}^m$ be defined such that μ_i is the average rating for the i^{th} movie across all users who rated that movie). In the case where the movie has not been rated by any users, we set $\mu_i = 0$:

$$\hat{R} = \mu \mathbf{1}^T$$

Thus, we can write our predictor for R_{ij} as the dot product of the two 1D vectors:

$$\hat{R}_{ij} = \langle \mu_i, \mathbf{1} \rangle$$

This yields an error $\boxed{\epsilon_{test}(\hat{R}) = 1.06}$

Problem 2a Code

```
# LOAD DATA ===========================================================================
data = []
with open('./data/ml-100k/u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])

data = np.array(data)

n = len(data) # n= 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*n)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train], :]
test = data[perm[num_train::], :]

# COMMON FUNCTIONS ====================================================================

def find_error(X, R_hat):
    '''

    :param X: data matrix in sparse representation: rows are in form (user_index,
                                                      movie_index, rating)
    :param R_hat: predicted movie preferences; this is a function that takes in (j, i),
                                                      the user
                  index and the movie index, and outputs a number in [1, 5]
    :return: MSE on the given data set
    '''

    n = X.shape[0]

    # rows of X are (user_index, movie_index, rating)
    # R_hat is a function that takes in (user_index, movie_index) and outputs predicted
                                                      rating
    return 1/n * sum([(X[k, 2] - R_hat(X[k, 0], X[k, 1])) ** 2 for k in range(n)])

# =====================================================


mu = np.zeros(shape=(num_items, ))

# counts the number of ratings for each movie
counts = np.zeros(shape=(num_items, ), dtype=int)
for i in range(train.shape[0]):
    next_rating = train[i, 2]
    next_movie = train[i, 1]
    mu[next_movie] += next_rating
    counts[next_movie] += 1
```

```
# this will ensure that mu is set equal to zero when no ratings are available
# for a specific movie
counts = np.where(counts == 0, 1, counts)
mu = mu / counts

R_hat = lambda user_index, movie_index: mu[movie_index]
print('Test Error Part a: ', find_error(test, R_hat))
```

(b) Note that once we generate $\tilde{R}$ and take its SVD we have the decomposition:

$$\tilde{R} = U\Sigma V^*$$

The corresponding rank d approximation of $\tilde{R}$ would use only the first d singular values and their associated left/right singular vectors in U, V^*. Thus, we can see the rank d approximation must be (using python indexing):

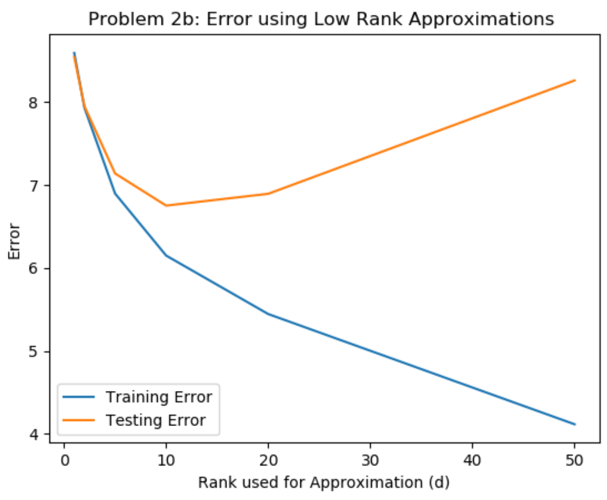
$$\tilde{R}_d = U[:, 0:d]\Sigma[0:d, 0:d]V^*[0:d, :] = U_d\Sigma_d V_d^*$$

If we were interested in determining the (i, j) entry of $\tilde{R}_d$, we could consider the element by element method to matrix multiplication: If $C = AB$ then $C_{ij} = A_{row_i}B_{col_j}$. In this case, this implies:

$$\tilde{R}_d[i, j] = (US)_{row_i}V^*_{col_j} = U[i, 0:d]\Sigma[0:d, 0:d]V^*[0:d, j]$$

Thus, we can see the desired $\{u_i\} \in \mathbb{R}^d$ are just the rows of our matrix U_dS_d and $\{v_j\}$ are just the columns of V^* (rows of V, up to complex conjugation). Thus for a given movie, user pair (i, j) we would predict the rating R_{ij} to be:

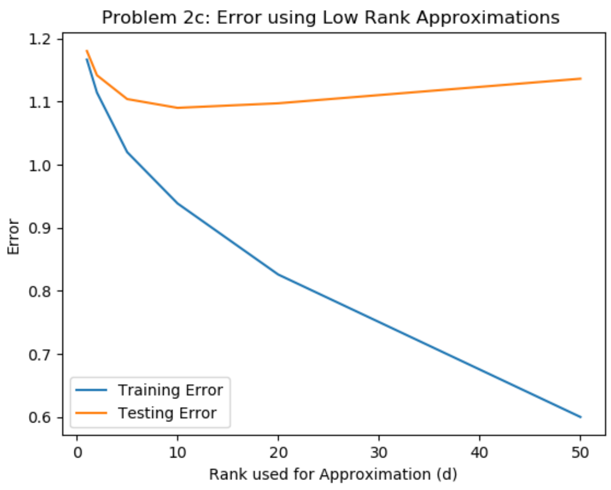
$$\hat{R}_{ij} = \langle U_d, row_i S_d, V^*_{col_j}\rangle$$



Problem 2b Code

```
# LOAD DATA ==========================================================================
data = []
with open('./data/ml-100k/u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])

data = np.array(data)

n = len(data) # n= 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*n)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train], :]
test = data[perm[num_train::], :]

# COMMON FUNCTIONS ===================================================================

def find_error(X, R_hat):
    '''

    :param X: data matrix in sparse representation: rows are in form (user_index,
                                                   movie_index, rating)
    :param R_hat: predicted movie preferences; this is a function that takes in (j, i),
                                                   the user
                  index and the movie index, and outputs a number in [1, 5]
    :return: MSE on the given data set
    '''

    n = X.shape[0]

    # rows of X are (user_index, movie_index, rating)
    # R_hat is a function that takes in (user_index, movie_index) and outputs predicted
                                                   rating
    return 1/n * sum([(X[k, 2] - R_hat(X[k, 0], X[k, 1])) ** 2 for k in range(n)])

# ======================================================
matrix = np.zeros(shape=(num_items, num_users))
for entry in range(train.shape[0]):
    next_user = train[entry, 0]
    next_movie = train[entry, 1]
    next_rating = train[entry, 2]
    matrix[next_movie, next_user] = next_rating

[u, s, vh] = np.linalg.svd(matrix, full_matrices=False)

d_vals = [1, 2, 5, 10, 20, 50]
all_train_errors = []
all_test_errors = []

for d in d_vals:

    # First, note that the lowrank (rank d) approximation can be acheived through the
                                                   matrix
    # multiplication U[:, :d] * S[:d, :d] * Vh[:d, :] = U_d S_d Vh_d.
    # To find the (i,j) = (movie_index, user_index) entry in this low rank approximation
                                                   of our matrix, we can multiply the
    # ith row of U_dS_d by the jth column of Vh_d
    Rd_hat = lambda user_index, movie_index: np.inner(u[movie_index, :d]*s[:d], vh[:d,
                                                   user_index])

    train_error = find_error(train, Rd_hat)
    test_error = find_error(test, Rd_hat)
    all_test_errors.append(test_error)
    all_train_errors.append(train_error)

    #print('Test Error Part b (d = ', d, '): ', test_error)

plt.figure(1)
plt.plot(d_vals, all_train_errors)
plt.plot(d_vals, all_test_errors)
plt.title('Problem 2b: Error using Low Rank Approximations ')
plt.xlabel('Rank used for Approximation (d)')
plt.ylabel('Error')
plt.legend(['Training Error', 'Testing Error'])
plt.show()
```

(c) As we can see in Figure **??**, the test error is roughly around 1.1 for the best value of d ($d \approx 10$). Surprisingly, this is slightly worse than the naive predictor we generated in part a) that completely ignored variation between users and just always predicted the average. It could be that by setting all the unknown entries to a single entry, we are inadvertently adding highly correlated structure to the system. It is this artificial structure that gets picked up by the SVD rather than the true structure in the data, making our predictors $\langle u_i, v_j \rangle$ perform poorly.



Problem 2c Code

```python
# LOAD DATA ===========================================================================
data = []
with open('./data/ml-100k/u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])

data = np.array(data)

n = len(data) # n= 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*n)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train], :]
test = data[perm[num_train::], :]

# COMMON FUNCTIONS ====================================================================

def find_error(X, R_hat):
    '''

    :param X: data matrix in sparse representation: rows are in form (user_index,
                                                    movie_index, rating)
    :param R_hat: predicted movie preferences; this is a function that takes in (j, i),
                                                    the user
                  index and the movie index, and outputs a number in [1, 5]
    :return: MSE on the given data set
    '''

    n = X.shape[0]

    # rows of X are (user_index, movie_index, rating)
    # R_hat is a function that takes in (user_index, movie_index) and outputs predicted
                                                    rating
    return 1/n * sum([(X[k, 2] - R_hat(X[k, 0], X[k, 1])) ** 2 for k in range(n)])

# =====================================================


matrix = np.zeros(shape=(num_items, num_users)) - 1 # negative entries will be our sign
                                                that no ratings are present
for entry in range(train.shape[0]):
    next_user = train[entry, 0]
    next_movie = train[entry, 1]
    next_rating = train[entry, 2]
    matrix[next_movie, next_user] = next_rating


# Replace unknown entries with the mean rating
mean_rating = np.mean(train[:, 2])
matrix = np.where(matrix == -1, mean_rating, matrix)

[u, s, vh] = np.linalg.svd(matrix, full_matrices=False)

d_vals = [1,2,5,10,20,50]
all_train_errors = []
all_test_errors = []

for d in d_vals:

    # First, note that the lowrank (rank d) approximation can be acheived through the
                                                    matrix
    # multiplication U[:, :d] * S[:d, :d] * Vh[:d, :] = U_d S_d Vh_d.
    # To find the (i,j) = (movie_index, user_index) entry in this low rank approximation
                                                    of our matrix, we can multiply the
    # ith row of U_dS_d by the jth column of Vh_d
    Rd_hat = lambda user_index, movie_index: np.inner(u[movie_index, :d]*s[:d], vh[:d,
                                                    user_index])

    train_error = find_error(train, Rd_hat)
    test_error = find_error(test, Rd_hat)
    all_test_errors.append(test_error)
    all_train_errors.append(train_error)

    #print('Test Error Part b (d = ', d, '): ', test_error)

plt.figure(2)
plt.plot(d_vals, all_train_errors)
plt.plot(d_vals, all_test_errors)
plt.title('Problem 2c: Error using Low Rank Approximations ')
plt.xlabel('Rank used for Approximation (d)')
plt.ylabel('Error')
plt.legend(['Training Error', 'Testing Error'])
plt.show()
```

(d) Suppose we fix $\{v_j\}_{j=1}^n$. The goal is the to determine the optimal $\{u_i\}_{i=1}^m$. Consider the partial derivative of this loss function with respect to u_i:

$$\nabla_{u_i} L = 2 \sum_{(j,R_{ij}) \in train} v_j(\langle u_i, v_j\rangle - R_{ij}) + 2\lambda u_i = 0$$

$$\lambda u_i = \sum_{(j,R_{ij}) \in train} v_j(R_{ij} - \langle u_i, v_j\rangle)$$

$$\lambda u_i + \sum_{(j,R_{ij}) \in train} v_j \langle u_i, v_j\rangle = \sum_{(j,R_{ij}) \in train)} v_j R_{ij}$$

$$\lambda u_i + \sum_{(j,R_{ij}) \in train} v_j v_j^T u_i = \sum_{(j,R_{ij}) \in train} v_j R_{ij}$$

$$\left(\lambda I + \sum_{(j,R_{ij}) \in train)} v_j v_j^T\right) u_i = \sum_{(j,R_{ij}) \in train} v_j R_{ij}$$

Thus, for any fixed $i \in \{1, 2, ..., m\}$ it holds:

$$\boxed{u_i = \left(\lambda I + \sum_{(i,j,R_{ij}) \in train} v_j v_j^T\right)^{-1} \sum_{(i,j,R_{ij}) \in train} v_j R_{ij}}$$

Similarly, when we fix $\{u_i\}$ then we can determine the optimal $\{v_j\}$ by taking the partial derivative of this loss function with respect to v_j. Thus, for any fixed $j \in \{1, 2, ..., n\}$ it holds:

$$\boxed{v_j = \left(\lambda I + \sum_{(i,R_{ij}) \in train} u_i u_i^T\right)^{-1} \sum_{(i,R_{ij}) \in train} u_i R_{ij}}$$

In both cases rather than actually calculating the inverse, we will just solve the linear system. Since $u_i, v_j \in \mathbb{R}^d$ and $d << n, d << m$ this system is much smaller than the entire $m \times n$ matrix R we aim to approximate (this system is size $\mathbb{R}^{d\times d}$ due to the outer products).

Note that we can quickly calculate the sum: $\sum u_i u_i^T$ by writing it as a matrix multiply. Let U, V be defined as below and it holds:

$$U = \begin{bmatrix} -u_1- \\ -u_2- \\ ... \\ -u_m- \end{bmatrix} \rightarrow U_k^T U_k = \sum_{i,R_{ij},j=k} u_i u_i^T$$

where U_k are only the rows i of U where $(k, i, R_{ij}) \in train$. We can write as a matrix multiply $\sum v_j v_j^T$ in a similar way.

The initial vectors $\{u_i\}_{i=1}^m$ and $\{v_j\}_{j=1}^n$ were initialized by drawing randomly from the multivariate gaussian $\mathcal{N}(0, \sigma^2 I)$

Coarsely tuning the parameters λ and σ we chose $\lambda = 10$ and $\sigma = 5$ to be suitable choices. This procedure will generate a sets $\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n$ and we will just use:

$$\hat{R}_{ij} = \langle u_i, v_j\rangle$$

as our predictor.

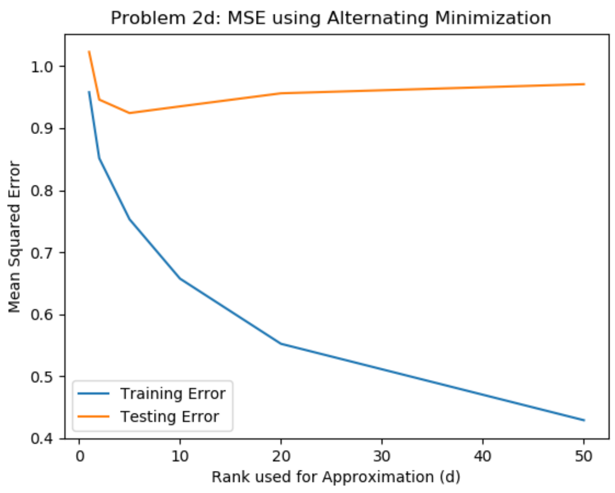


Figure 1: MSE on the training and testing datasets running alternating maximization with $\lambda = 10$ and $\sigma = 5$

Problem 2d Code

```
# LOAD DATA ============================================================================
data = []
with open('./data/ml-100k/u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])

data = np.array(data)

n = len(data) # n= 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*n)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train], :]
test = data[perm[num_train::], :]

# COMMON FUNCTIONS =====================================================================

def find_error(X, R_hat):
    '''

    :param X: data matrix in sparse representation: rows are in form (user_index,
                                                   movie_index, rating)
    :param R_hat: predicted movie preferences; this is a function that takes in (j, i),
                                                   the user
                  index and the movie index, and outputs a number in [1, 5]
    :return: MSE on the given data set
    '''

    n = X.shape[0]

    # rows of X are (user_index, movie_index, rating)
    # R_hat is a function that takes in (user_index, movie_index) and outputs predicted
                                                   rating
    return 1/n * sum([(X[k, 2] - R_hat(X[k, 0], X[k, 1])) ** 2 for k in range(n)])

# =====================================================



# choose hyperparameters
lam = 10 # lambda; used for regularization
sigma = 5  # standard deviation for normal distributions used for initializing {u_i}, {
                                               v_j}
delta = 0.1  # convergence condition
```

```
d_vals = [1, 2, 5, 10, 20, 50]
all_train_errors = []
all_test_errors = []

# Rather than searching through all of train to find appropriate indices every time, we
                                                  will just do
# it initially. Row indices in train that correspond to each respective i, j
index_map_i = {i : np.where(train[:, 1] == i)[0] for i in range(num_items)}
index_map_j = {j : np.where(train[:, 0] == j)[0] for j in range(num_users)}

for d in d_vals:

    # Initialize {u_i}, {v_i}. Let U, V be matrices whose rows are u_i, v_i respectively
    U = sigma * np.random.randn(num_items, d)
    V = sigma * np.random.randn(num_users, d)

    prev_U = np.copy(U)
    prev_V = np.copy(V)

    not_converged = True
    itr = 0

    print('Running d value : ', d)
    while not_converged:
        itr += 1
        print('iteration: ', itr)

        # Fix {u_i} and solve for {v_i}
        for j in range(num_users):
            indices = index_map_j[j]
            U_j = U[train[indices, 1], :]  # only take u_i who have a corresponding (j, i
                                              , R_ij) datapoint with current
                                               j
            A = lam * np.eye(d) + np.dot(U_j.T, U_j)
            b = np.dot(U_j.T, train[indices, 2])
            V[j, :] = np.linalg.solve(A, b)

        # Fix {v_j} and solve for {u_i}
        for i in range(num_items):
            indices = index_map_i[i] #  j values in (j, i, R_ij) with current i
            V_i = V[train[indices, 0], :]  # only take the v_j who have a corresponding (
                                              j, i, R_ij) datapoint with
                                              current i
            A = lam * np.eye(d) + np.dot(V_i.T, V_i)
            b = np.dot(V_i.T, train[indices, 2])
            U[i, :] = np.linalg.solve(A, b)

        # check convergence
        #max_diff = max(np.max(np.abs(U - prev_U)),  np.max(np.abs(V - prev_V)))
        #print(max_diff, np.max(V), np.max(U))
        if np.max(np.abs(U - prev_U)) < delta and np.max(np.abs(V - prev_V)) < delta:
            not_converged = False
        else:
            prev_U = np.copy(U)
            prev_V = np.copy(V)

    R_hat = lambda user_index, movie_index: np.inner(U[movie_index, :], V[user_index, :])

    all_train_errors.append(find_error(train, R_hat))
    all_test_errors.append(find_error(test, R_hat))





plt.figure(2)
plt.plot(d_vals, all_train_errors)
plt.plot(d_vals, all_test_errors)
plt.title('Problem 2d: MSE using Alternating Minimization  ')
plt.xlabel('Rank used for Approximation (d)')
plt.ylabel('Mean Squared Error')
plt.legend(['Training Error', 'Testing Error'])

plt.show()
```

(e) Here we choose parameter $\lambda = 0.001$ and $\sigma = 1$ with a batch size of 500 and a step size of $\eta = 2 * 10^{-2}$. For convergence, we stopped iterating SGD once both $\nabla_U L$ and $\nabla_V L$ had no entry above $\delta = \frac{0.05}{\eta} = 25$. Again, this will generate a sets $\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n$ and we will just use:

$$\hat{R}_{ij} = \langle u_i, v_j \rangle$$

as our predictor.

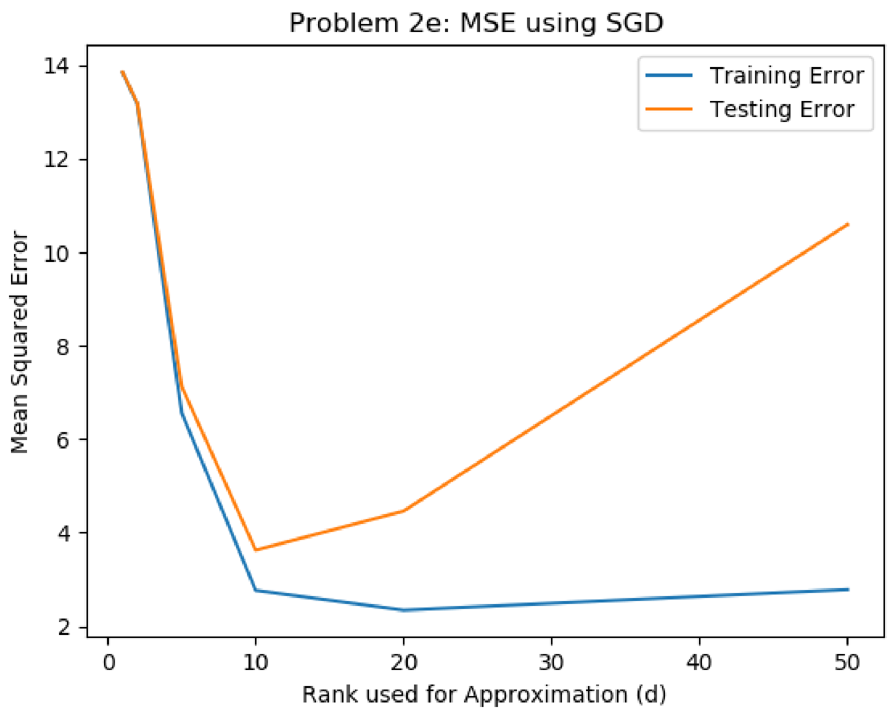


Figure 2

Problem 2e Code

```
# LOAD DATA ==========================================================================
data = []
with open('./data/ml-100k/u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])

data = np.array(data)

n = len(data) # n= 100,000
num_users = max(data[:,0])+1 # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1 # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*n)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train], :]
test = data[perm[num_train::], :]

# COMMON FUNCTIONS ====================================================================

def find_error(X, R_hat):
    '''

    :param X: data matrix in sparse representation: rows are in form (user_index,
                                              movie_index, rating)
    :param R_hat: predicted movie preferences; this is a function that takes in (j, i),
                                              the user
                  index and the movie index, and outputs a number in [1, 5]
    :return: MSE on the given data set
    '''

    n = X.shape[0]

    # rows of X are (user_index, movie_index, rating)
    # R_hat is a function that takes in (user_index, movie_index) and outputs predicted
                                                   rating
    return 1/n * sum([(X[k, 2] - R_hat(X[k, 0], X[k, 1])) ** 2 for k in range(n)])


# Problem 2e WITH PYTORCH =========================================================

import torch as torch
import torch.optim as optim

# choose hyperparameters
lam = 0.001  # lambda; used for regularization
sigma = 1 # standard deviation for normal distributions used for initializing {u_i}, {v_j
                                                }
delta = 0.05  # convergence condition
batch_size = 500  # minibatch size to use in stochastic gradient descent
eta = 2e-3 # learning rate for stochastic gradient descent

device = torch.device('cpu')

d_vals = [1,2, 5, 10, 20, 50]
#d_vals = [1,2,5,10]
N = train.shape[0]
all_train_errors = []
all_test_errors = []

torch.random.manual_seed(14564678) # 1546345

def loss_function(U,V, batch, lam):
    loss = torch.tensor([[0]], dtype=torch.float)
    for row_num in range(batch.shape[0]):
        (j, i, R_ij) = batch[row_num, :]
        u_i = U[i, :]
        v_j = V[j, :]
        temp = torch.mm(u_i.view(1, -1), v_j.view(-1, 1))
        loss += (temp - R_ij).pow(2)

    loss += lam * torch.norm(U, p='fro')
    loss += lam * torch.norm(V, p='fro')
    return loss

for d in d_vals:

    # Initialize {u_i}, {v_i}. Let U, V be matrices whose rows are u_i, v_i respectively
    U = sigma * torch.randn(num_items, d, device=device, dtype=torch.float)
    U.requires_grad = True
    V = sigma * torch.randn(num_users, d, device=device, dtype=torch.float)
    V.requires_grad = True

    not_converged = True
    itr = 0

    print('Running d value : ', d)
    while not_converged:
        itr += 1
        print('iteration: ', itr)

        shuffled_indices = list(range(train.shape[0]))
        np.random.shuffle(shuffled_indices)

        # LOOP OVER ALL MINIBATCHES
        for batch_num in range(train.shape[0] // batch_size):

            data_indices = shuffled_indices[batch_num * batch_size : (batch_num + 1) *
                                                    batch_size]
            batch = train[data_indices, :]

            loss = loss_function(U=U, V=V, batch=batch, lam=lam)
            #print(loss.item())
            loss.backward()
            # check convergence
            with torch.no_grad():
                U -= eta * U.grad
                V -= eta * V.grad
                if eta * torch.max(torch.abs(U.grad)).item() < delta and eta * torch.max(
                                                                         torch.abs(V.grad)) < delta
                                                                         :
                    not_converged = False
                    break

                #p(eta * torch.max(torch.abs(U.grad)).item())
                #p(eta * torch.max(torch.abs(V.grad)).item())
                #p(loss_function(U,V, train, lam))
                #print('hi')

                # zero out the gradients so they dont accummulate for next iteration
                U.grad.zero_()
                V.grad.zero_()




    R_hat = lambda user_index, movie_index: torch.mm(U[movie_index, :].view(1, -1), V[
                                                      user_index, :].view(-1, 1)).item()

    all_train_errors.append(find_error(train, R_hat))
    all_test_errors.append(find_error(test, R_hat))





plt.figure(2)
plt.plot(d_vals, all_train_errors)
plt.plot(d_vals, all_test_errors)
plt.title('Problem 2e: MSE using SGD  ')
plt.xlabel('Rank used for Approximation (d)')
plt.ylabel('Mean Squared Error')
plt.legend(['Training Error', 'Testing Error'])

plt.show()
```

(f) Here we can see that in part d, the solution obtained by alternating minimization seems to generalize to the test set much worse than the solution obtained via stochastic gradient descent in part e. While the training error in 2d continuously decreases, the test error flat-lines almost immediately. On the other hand, in 2e the test error initially decreases with the training error before eventually curving back upward.

Deep Learning Architectures

Problem 3

(a) Here we chose the learning rate $\eta = 0.0001$ and the momentum $m = 0.6$

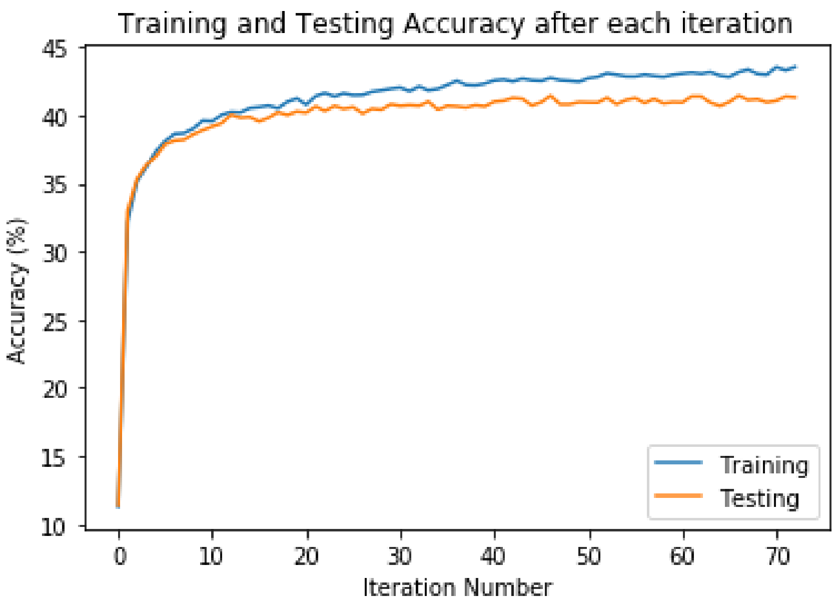


Figure 3: Accuracy plotted per iteration on both the training and test set. One iteration is defined to be 2000 updates during SGD (2000 minibatches of size 4). In this case, the neural network is just a linear transformation (no hidden layers)

(b) Here we chose the learning rate $\eta = 0.001$ and the momentum $m = 0.7$. We chose the hidden layer to have $M = 200$ nodes.

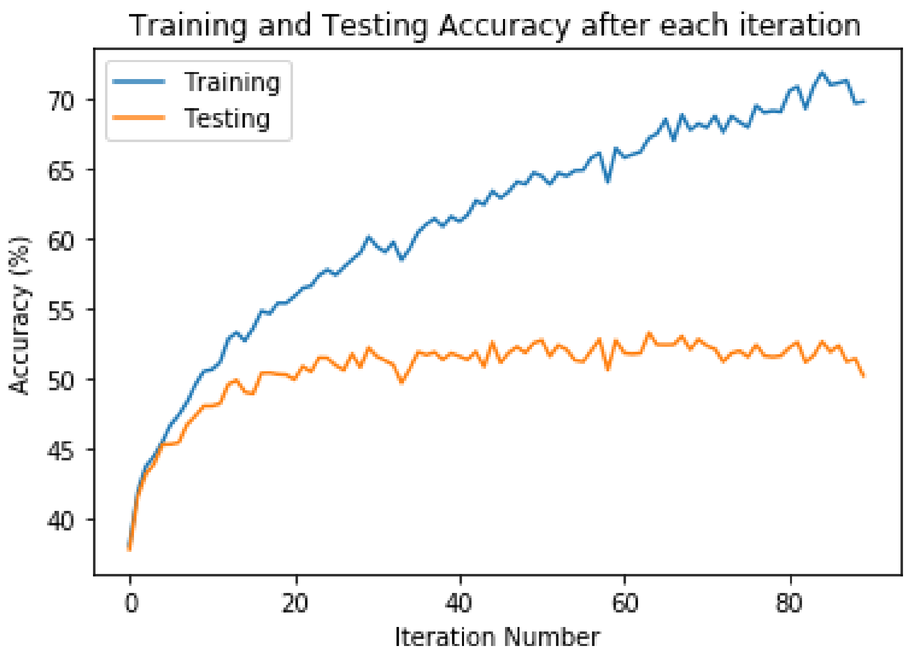


Figure 4: Accuracy plotted per iteration on both the training and test set. One iteration is defined to be 2000 updates during SGD (2000 minibatches of size 4). In this case, the neural network is just a linear transformation (no hidden layers)

(c) Here we chose the learning rate $\eta = 0.001$ and the momentum $m = 0.9$. We chose the convolutional layer to have $M = 100$ filters and to have a filter size of $(4x4)$. We chose the Max Pooling layer to have a kernel size of $(8x8)$

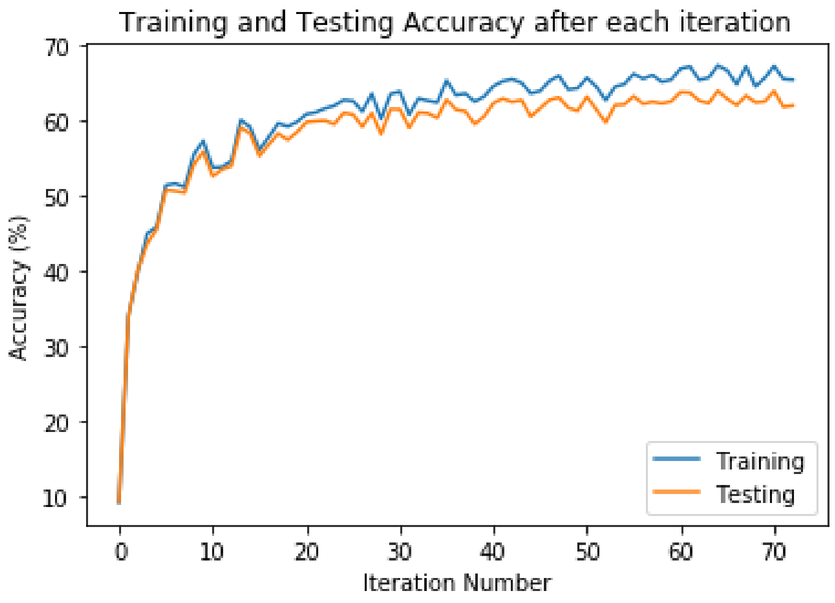


Figure 5: Accuracy plotted per iteration on both the training and test set. One iteration is defined to be 2000 updates during SGD (2000 minibatches of size 4). In this case, the neural network is just a linear transformation (no hidden layers)