CS189 HW7

David Chen

April 2022

Contents

1 Write-up and Honor Code

Collaborated with: N/A.

I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

Signed: David Chen

2 Movie Recommender System

2.1 Part (a)

Recall that the matrix R can be expressed as a sum of outer products

$$\begin{aligned} R &= UDV^T \\ &= \sum_{i=1}^{d} \delta_i u_i v_i^T. \end{aligned}$$

To get entry $R_{i,j} = UDV^T$, we want to find the dot product of the ith row of UD with the jth column of V^T, which is equivalent to the jth row of V. The ith row of UD is simply the ith row of U, but each entry $u_{i,k}$ is scaled by δ_i. Therefore, the dot product between these two vectors is given by

$$\begin{aligned} R_{i,j} &= (UD_i \cdot V_j)_{i,j} \\ &= \begin{bmatrix} \delta_1 U_{i,1} \\ \delta_2 U_{i,2} \\ \vdots \\ \delta_d U_{i,d} \\ \vdots \\ 0 \end{bmatrix} \cdot \begin{bmatrix} V_{j,1} \\ V_{j,2} \\ \vdots \\ V_{j,n} \end{bmatrix} \\ &= \sum_{k=1}^{d} \delta_k U_{i,k} V_{j,k}. \end{aligned}$$

2.2 Part (b)

From the previous section, we have that entry $R_{i,j}$ can be expressed as $\sum_{k=1}^{d} \delta_k U_{i,k} V_{j,k}$. We can transform this to create a direct inner product by using $U_i^{'}$, which is the length d vector where the first d terms of U_i are scaled by the singular values δ_i. In expression form,

$$U_i^{'} = \begin{bmatrix} \delta_1 U_{i,1} \\ \delta_2 U_{i,2} \\ \vdots \\ \delta_d U_{i,d} \end{bmatrix}.$$

Using this vector as x_i and using row j of matrix V, truncated to the first d entries, as y_j gives

$$\begin{aligned} x_i \cdot y_j &= \sum_{k=1}^{d} \delta_k U_{i,k} V_{j,k} \\ &= R_{i,j}. \end{aligned}$$

2.3 Part (c)

Code included in the attached Python Notebook.

2.4 Part (d)

Code included in the attached Python Notebook.

2.5 Part (e)

Code included in the attached Python Notebook.

2.6 Part (f)

We start by deriving the closed-form solution of x_i that minimizes the loss function when we treat y_j as a constant. This gives our objective of

$$\begin{aligned}\min_{x_i} L\left(\{x_i\},\{y_j\}\right) &= \min_{x_i} \sum_{(i,j)\in S} \left(x_i \cdot y_j - R_{i,j}\right)^2 + \sum_{i=1}^{n} \|x_i\|_2^2 + \sum j = 1^m \|y_j\|_2^2 \\ &= \min_{x_i} \sum_{(i,j)\in S} \left(x_i \cdot y_j - R_{i,j}\right)^2 + \sum_{i=1}^{n} \|x_i\|_2^2.\end{aligned}$$

Taking the derivative of this objective function with respect to a single point x_k and setting it to 0 gives

$$\begin{aligned}\frac{\partial}{\partial x_k} L\left(\{x_i\},\{y_j\}\right) &= \frac{\partial}{\partial x_k}\left(\sum_{(i,j)\in S} \left(x_i \cdot y_j - R_{i,j}\right)^2 + \sum_{i=1}^{n} \|x_i\|_2^2\right) \\ &= \sum_{(k,j)\in S} 2y_j\left(x_k \cdot y_j - R_{k,j}\right) + 2x_k \\ &= \sum_{(k,j)\in S} 2y_j y_j^T x_k - \sum_{(k,j)\in S} 2y_j R_{k,j} + 2x_k \\ &= 0 \\ \therefore \sum_{(k,j)\in S} y_j R_{k,j} &= x_k^* + \sum_{(k,j)\in S} y_j y_j^T x_k^* \\ &= I x_k^* + \sum_{(k,j)\in S} y_j y_j^T x_k^* \\ &= \left(I + \sum_{(k,j)\in S} y_j y_j^T\right) x_k^* \\ \therefore x_k^* &= \left(I + \sum_{(k,j)\in S} y_j y_j^T\right)^{-1} \left(\sum_{(k,j)\in S} y_j R_{k,j}\right).\end{aligned}$$

Note that a similar argument is used for the closed-form solution of y_j holding the x_is constant, giving

$$y_k^* = \left(I + \sum_{(i,k)\in S} x_i x_i^T\right)^{-1} \left(\sum_{(i,k)\in S} x_i R_{i,k}\right).$$

Now, we can use this closed-form solution in our algorithms.
Code included in attached Python Notebook.

3 Regularized and Kernel k-Means

3.1 Part (a)

The minimum value of the objective function when $k = n$ is 0, since each cluster will simply contain one point, and the mean of that cluster will be the value of that point, giving a total objective value of 0.

3.2 Part (b)

Taking the gradient with respect to μ_i of our objective function for a single cluster and setting it equal to 0 gives

$$
\begin{aligned}
\frac{\partial}{\partial \mu_i}\left(\lambda\|\mu_i\|_2^2 + \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2\right) &= 2\lambda\mu_i - \sum_{X_j \in C_i} 2\left(X_j - \mu_i\right) \\
&= 2\lambda\mu_i + 2\sum_{X_j \in C_i} \mu_i - 2\sum_{X_j \in C_i} X_j \\
&= 2\lambda\mu_i + 2|C_i|\mu_i - 2\sum_{X_j \in C_i} X_j \\
&= 2\mu_i\left(|C_i| + \lambda\right) - 2\sum_{X_j \in C_i} X_j \\
&= 0 \\
\therefore 2\mu_i\left(|C_i| + \lambda\right) &= 2\sum_{X_j \in C_i} X_j \\
\therefore \mu_i &= \frac{1}{|C_i| + \lambda}\sum_{X_j \in C_i} X_j.
\end{aligned}
$$

3.3 Part (c)

Recall that our goal is to minimize the sum of the squared distances from points to their cluster means. Therefore, for each individual point, we want to choose the class that minimizes the aforementioned sum. First, we define the mean of cluster k as

$$\mu_k = \frac{1}{|S_k|} \sum_{x_i \in S_k} \Phi\left(x_i\right),$$

since we are applying kernel Φ to each point. Now, we can write the squared distance of point x_j to each class k, which is given by

$$\begin{aligned}
\left\|\Phi\left(x_j\right) - \mu_k\right\|^2 &= \Phi\left(x_j\right)^2 - 2\Phi\left(x_j\right)\mu_k + {\mu_k}^2 \\
&= \Phi\left(x_j\right) \cdot \Phi\left(x_j\right) - 2\Phi\left(x_j\right)\frac{1}{|S_k|}\sum_{x_i \in S_k}\Phi\left(x_i\right) + \frac{1}{|S_k|^2}\sum_{x_i \in S_k}\sum_{x_a \in S_k}\Phi\left(x_i\right) \cdot \Phi\left(x_a\right) \\
&= \kappa\left(x_j, x_j\right) - \frac{2}{|S_k|}\sum_{x_i \in S_k}\Phi\left(x_j\right) \cdot \Phi\left(x_i\right) + \frac{1}{|S_k|^2}\sum_{x_i \in S_k}\sum_{x_a \in S_k}\kappa\left(x_i, x_a\right) \\
&= \kappa\left(x_j, x_j\right) - \frac{2}{|S_k|}\sum_{x_i \in S_k}\kappa\left(x_j, x_i\right) + \frac{1}{|S_k|^2}\sum_{x_i \in S_k}\sum_{x_a \in S_k}\kappa\left(x_i, x_a\right).
\end{aligned}$$

Note that, for all classes k, the first term $\kappa\left(x_j, x_j\right)$ stays constant. Therefore, we can remove it from our argmin, which gives objective function

$$argmin_k\left(-\frac{2}{|S_k|}\sum_{x_i \in S_k}\kappa\left(x_j, x_i\right) + \frac{1}{|S_k|^2}\sum_{x_i \in S_k}\sum_{x_a \in S_k}\kappa\left(x_i, x_a\right)\right).$$

3.4 Part (d)

First, as was shown in the previous part, we removed the constant $\kappa(x_j, x_j)$ term from our argmin, since this term does not vary across classes. Next, notice that $\frac{1}{|S_k|^2} \sum_{x_i \in S_k} \sum_{x_a \in S_k} \kappa(x_i, x_a)$ does not depend on the current point x_j. Therefore, we only need to compute this once for each cluster, then we can store the value and access it rather than computing it again for the same cluster. Finally, recall that $\kappa(x_i, x_j) = \kappa(x_j, x_i)$. Therefore, while calculating term $-\frac{2}{|S_k|} \sum_{x_i \in S_k} \kappa(x_j, x_i)$, we can keep track of all the $\kappa(x_j, x_i)$ and reuse them for future calculations with either $\kappa(x_j, x_i)$ or $\kappa(x_i, x_j)$ rather than having to recalculate the kernel function.

4 The Training Error of AdaBoost

4.1 Part (a)

First, note that because $\sum_{i=1}^{n} w_i^T = 1$, $err_T = \sum_{y_i \neq G_T(X_i)} w_i^T$. Therefore, taking the summation of both sides gives

$$
\begin{aligned}
\sum_{i=1}^{n} w_i^{T+1} &= \sum_{i=1}^{n} \frac{1}{Z_T} w_i^T e^{(-\beta_T y_i G_T(X_i))} \\
\therefore Z_T &= \sum_{i=1}^{n} w_i^T e^{(-\beta_T y_i G_T(x_i))} \\
&= \sum_{y_i = G_T(X_i)} w_i^T e^{(-\beta_T y_i G_T(X_i))} + \sum_{y_i \neq G_T(X_i)} w_i^T e^{(-\beta_T y_i G_T(X_i))} \\
&= \sum_{y_i = G_T(X_i)} w_i^T e^{(-\beta_T)} + \sum_{y_i \neq G_T(X_i)} w_i^T e^{(\beta_T)} \\
&= e^{(-\beta_T)} \left(\sum_{y_i = G_T(X_i)} w_i^T + e^{(2\beta_T)} \sum_{y_i \neq G_T(X_i)} w_i^T \right) \\
&= e^{(-\beta_T)} \left(1 - err_T + e^{(2\beta_T)} err_T \right) \\
&= e^{(-\beta_T)} \left(1 - err_T + \frac{1 - err_T}{err_T} err_T \right) \\
&= 2 \left(1 - err_T\right) \left(\frac{1 - err_T}{err_T} \right)^{-\frac{1}{2}} \\
&= 2 \sqrt{err_T \left(1 - err_T\right)}.
\end{aligned}
$$

4.2 Part (b)

Note that the expression for any weight at any time is given by the product of all the previous updates multiplied by the initial weight. This gives

$$\begin{aligned}
w_i^{T+1} &= w_i^0 * \prod_{t=1}^{T} Update_i \\
&= \frac{1}{n} \prod_{t=1}^{T} \frac{1}{Z_t} w_i^t e^{-(\beta_t y_i G_t(X_i))} \\
&= \frac{1}{n \prod_{t=1}^{T} Z_t} e^{-\sum_{t=1}^{T} \beta_t y_i G_t(X_i)} \\
&= \frac{1}{n \prod_{t=1}^{T} Z_t} e^{-y_i \sum_{t=1}^{T} \beta_t G_t(X_i)} \\
&= \frac{1}{n \prod_{t=1}^{T} Z_t} e^{-y_i M(X_i)}.
\end{aligned}$$

4.3 Part (c)

$$\begin{aligned}
\sum_{i=1}^{n} &= \sum_{correct} e^{-|M(X_i)|} + \sum_{incorrect} e^{|M(X_i)|} \\
&\geq \sum_{incorrect} e^{|M(X_i)|} \\
&\geq \sum_{incorrect} 1 \\
&\geq B.
\end{aligned}$$

4.4 Part (d)

Since all $err_t \leq 0.49$, we have that all $Z_t < 0.9998$. Therefore,

$$\begin{aligned}
w_i^{T+1} &= \frac{1}{n\prod_{t=1}^{T} Z_t} e^{-y_i M(X_i)} \\
&\geq \frac{1}{n(0.9998)^T} e^{-y_i M(X_i)} \\
\therefore \lim_{t\to\infty} w_i^{T+1} &\geq 0 \\
&\geq B.
\end{aligned}$$

Since B is lower-bounded by 0 and we showed that B is also upper bounded by 0 as $T \to \infty$, we have that $B \to 0$.

4.5 Part (e)

Since AdaBoost uses short decision trees, each decision tree will only have a few features that it operates on. Furthermore, AdaBoost gives higher voting power to the stronger tress and lower voting power to the weaker trees. Therefore, the features of the stronger trees will have great impact on the prediction, whereas the features of the weak trees will have little impact on the prediction. This shows how AdaBoost selects a few strong features to base its prediction off of.

CS189 Homework 7

Imports

In [1]:
```
import os
import scipy
from scipy import io
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
```

Question 2: Movie Recommender System

Setup

In [2]:
```
# Recall we're doing unsupervised learning, so no labels in training data
R = io.loadmat("movie_data/movie_train.mat")["train"]
val_data = np.loadtxt('movie_data/movie_validate.txt', dtype=int, delimiter=','

# Helper method to get training accuracy
def get_train_acc(R, user_vecs, movie_vecs):
    num_correct, total = 0, 0
    for i in range(R.shape[0]):
        for j in range(R.shape[1]):
            if not np.isnan(R[i, j]):
                total += 1
                if np.dot(user_vecs[i], movie_vecs[j])*R[i, j] > 0:
                    num_correct += 1
    return num_correct/total

# Helper method to get validation accuracy
def get_val_acc(val_data, user_vecs, movie_vecs):
    num_correct = 0
    for val_pt in val_data:
        user_vec = user_vecs[val_pt[0]-1]
        movie_vec = movie_vecs[val_pt[1]-1]
        est_rating = np.dot(user_vec, movie_vec)
        if est_rating*val_pt[2] > 0:
            num_correct += 1
    return num_correct/val_data.shape[0]

# Helper method to get indices of all rated movies for each user,
# and indices of all users who have rated that title for each movie
def get_rated_idxs(R):
    user_rated_idxs, movie_rated_idxs = [], []
    for i in range(R.shape[0]):
        user_rated_idxs.append(np.argwhere(~np.isnan(R[i, :])).reshape(-1))
    for j in range(R.shape[1]):
        movie_rated_idxs.append(np.argwhere(~np.isnan(R[:, j])).reshape(-1))
    return np.array(user_rated_idxs), np.array(movie_rated_idxs)
```

Part (c)

In [21]:
```
# Part (c): SVD to learn low-dimensional vector representations
def svd_lfm(R):

    # Fill in the missing values in R
    R = np.nan_to_num(R)

    # Compute the SVD of R
    U, s, Vh = linalg.svd(R, full_matrices=False)

    # Construct user and movie representations
    user_vecs = np.multiply(U, s)
    movie_vecs = Vh.T

    return user_vecs, movie_vecs
```

Part (d)

In [18]:
```
# Part (d): Compute the training MSE loss of a given vectorization
def get_train_mse(R, user_vecs, movie_vecs):

    # Compute the training MSE loss
    # too lazy to try to batch it
    mse_loss = 0
    for i in range(R.shape[0]):
        for j in range(R.shape[1]):
            entry = R[i][j]
            if np.isnan(entry):
                continue
            pred = user_vecs[i] @ movie_vecs[j]
            mse_loss += (pred - entry) ** 2

    return mse_loss
```

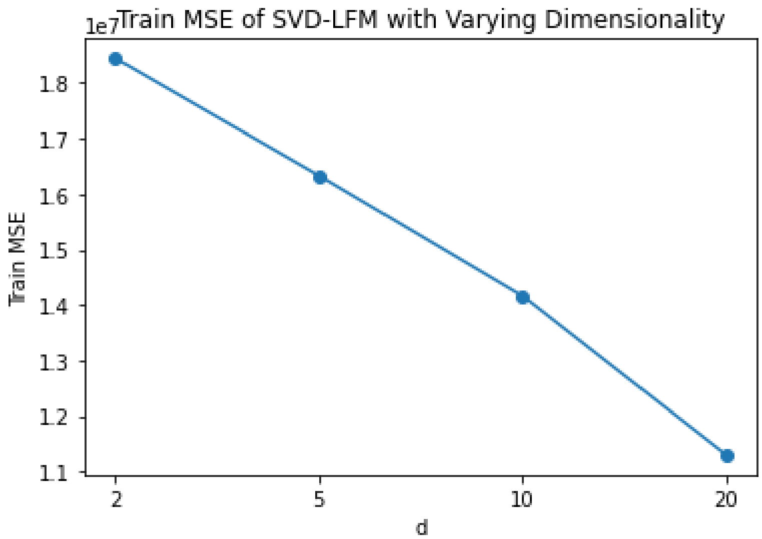
Part (e)

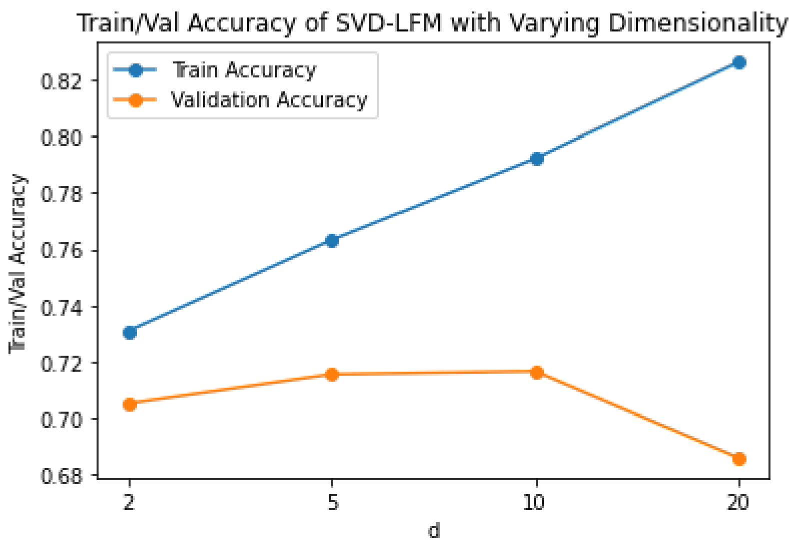
In [26]:
```
d_values = [2, 5, 10, 20]
train_mses, train_accs, val_accs = [], [], []
user_vecs, movie_vecs = svd_lfm(np.copy(R))
for d in d_values:
    train_mses.append(get_train_mse(np.copy(R), user_vecs[:, :d], movie_vecs[:,
    train_accs.append(get_train_acc(np.copy(R), user_vecs[:, :d], movie_vecs[:,
    val_accs.append(get_val_acc(val_data, user_vecs[:, :d], movie_vecs[:, :d]))
plt.clf()
plt.plot([str(d) for d in d_values], train_mses, 'o-')
plt.title('Train MSE of SVD-LFM with Varying Dimensionality')
plt.xlabel('d')
plt.ylabel('Train MSE')
plt.savefig(fname='train_mses.png', dpi=600, bbox_inches='tight')
plt.show()

plt.clf()
plt.plot([str(d) for d in d_values], train_accs, 'o-')
plt.plot([str(d) for d in d_values], val_accs, 'o-')
plt.title('Train/Val Accuracy of SVD-LFM with Varying Dimensionality')
```

```python
plt.xlabel('d')
plt.ylabel('Train/Val Accuracy')
plt.legend(['Train Accuracy', 'Validation Accuracy'])
plt.savefig(fname='trval_accs.png', dpi=600, bbox_inches='tight')
plt.show()

print(f"Validation Accuracies in order: {val_accs}.")
```






```
Validation Accuracies in order: [0.7051490514905149, 0.7154471544715447, 0.716
5311653116531, 0.6859078590785908].
```

d=10 gives the best performance. Even though we are still overfitting, having too little dimensions led to high bias and having too many dimensions led to high variance, whereas d=10 is the optimal choice for the bias-variance tradeoff.

Part (f)

In [24]:
```python
# Part (f): Learn better user/movie vector representations by minimizing loss
best_d = 10 #TODO(f): Use best from part (e)
np.random.seed(20)
user_vecs = np.random.random((R.shape[0], best_d))
movie_vecs = np.random.random((R.shape[1], best_d))
user_rated_idxs, movie_rated_idxs = get_rated_idxs(np.copy(R))
```

```
# Part (f): Function to update user vectors
def update_user_vecs(user_vecs, movie_vecs, R, user_rated_idxs):

    # Update user_vecs to the loss-minimizing value
    ##### TODO(f): Your Code Here #####
    for userIdx in range(len(user_rated_idxs)):
        userList = user_rated_idxs[userIdx]
        firstTerm = np.identity(best_d)
        secondTerm = np.zeros((best_d, ))
        for movieRated in userList:
            movieVec = movie_vecs[movieRated]
            firstTerm += np.outer(movieVec, movieVec)
            secondTerm += R[userIdx][movieRated] * movieVec
        user_vecs[userIdx] = np.linalg.inv(firstTerm) @ secondTerm

    return user_vecs

# Part (f): Function to update user vectors
def update_movie_vecs(user_vecs, movie_vecs, R, movie_rated_idxs):

    # Update movie_vecs to the loss-minimizing value
    ##### TODO(f): Your Code Here #####
    for movieIdx in range(len(movie_rated_idxs)):
        movieList = movie_rated_idxs[movieIdx]
        firstTerm = np.identity(best_d)
        secondTerm = np.zeros((best_d, ))
        for userRated in movieList:
            userVec = user_vecs[userRated]
            firstTerm += np.outer(userVec, userVec)
            secondTerm += R[userRated][movieIdx] * userVec
        movie_vecs[movieIdx] = np.linalg.inv(firstTerm) @ secondTerm

    return movie_vecs

# Part (f): Perform loss optimization using alternating updates
train_mse = get_train_mse(np.copy(R), user_vecs, movie_vecs)
train_acc = get_train_acc(np.copy(R), user_vecs, movie_vecs)
val_acc = get_val_acc(val_data, user_vecs, movie_vecs)
print(f'Start optim, train MSE: {train_mse:.2f}, train accuracy: {train_acc:.4f
for opt_iter in range(20):
    user_vecs = update_user_vecs(user_vecs, movie_vecs, np.copy(R), user_rated_
    movie_vecs = update_movie_vecs(user_vecs, movie_vecs, np.copy(R), movie_rat
    train_mse = get_train_mse(np.copy(R), user_vecs, movie_vecs)
    train_acc = get_train_acc(np.copy(R), user_vecs, movie_vecs)
    val_acc = get_val_acc(val_data, user_vecs, movie_vecs)
    print(f'Iteration {opt_iter+1}, train MSE: {train_mse:.2f}, train accuracy:
```

```
/var/folders/2b/pv38k5d552q5yst0qsc9wpzr0000gn/T/ipykernel_20307/42569185.py:3
5: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different length
s or shapes) is deprecated. If you meant to do this, you must specify 'dtype=o
bject' when creating the ndarray.
  return np.array(user_rated_idxs), np.array(movie_rated_idxs)
```

```
Start optim, train MSE: 27574866.30, train accuracy: 0.5950, val accuracy: 0.5
799
Iteration 1, train MSE: 13421216.24, train accuracy: 0.7611, val accuracy: 0.6
431
Iteration 2, train MSE: 11474959.41, train accuracy: 0.7876, val accuracy: 0.6
789
Iteration 3, train MSE: 10493324.86, train accuracy: 0.8007, val accuracy: 0.6
989
Iteration 4, train MSE: 10040997.98, train accuracy: 0.8069, val accuracy: 0.7
084
Iteration 5, train MSE: 9792296.83, train accuracy: 0.8098, val accuracy: 0.71
00
Iteration 6, train MSE: 9649312.88, train accuracy: 0.8117, val accuracy: 0.71
00
Iteration 7, train MSE: 9561491.69, train accuracy: 0.8130, val accuracy: 0.70
60
Iteration 8, train MSE: 9503837.41, train accuracy: 0.8138, val accuracy: 0.71
17
Iteration 9, train MSE: 9463660.97, train accuracy: 0.8144, val accuracy: 0.71
11
Iteration 10, train MSE: 9434168.95, train accuracy: 0.8147, val accuracy: 0.7
087
Iteration 11, train MSE: 9411512.64, train accuracy: 0.8150, val accuracy: 0.7
119
Iteration 12, train MSE: 9393397.49, train accuracy: 0.8152, val accuracy: 0.7
103
Iteration 13, train MSE: 9378404.19, train accuracy: 0.8155, val accuracy: 0.7
125
Iteration 14, train MSE: 9365635.88, train accuracy: 0.8156, val accuracy: 0.7
122
Iteration 15, train MSE: 9354518.75, train accuracy: 0.8157, val accuracy: 0.7
125
Iteration 16, train MSE: 9344681.51, train accuracy: 0.8158, val accuracy: 0.7
136
Iteration 17, train MSE: 9335879.18, train accuracy: 0.8159, val accuracy: 0.7
144
Iteration 18, train MSE: 9327944.20, train accuracy: 0.8160, val accuracy: 0.7
146
Iteration 19, train MSE: 9320755.69, train accuracy: 0.8161, val accuracy: 0.7
149
Iteration 20, train MSE: 9314221.76, train accuracy: 0.8163, val accuracy: 0.7
160
```

This method ended up getting a similar result to part (e). I believe that this is because our dataset is large, so the difference between excluding NaN data points and setting the NaN data points to the optimal value is slight, due to the abundance of other data to draw from.

In [ ]: