2.a)

$$1 = \sum_{i=1}^{n} w_i^{(T+1)} = \sum_{i=1}^{n} \frac{w_i^{(T)} \exp(-\beta_T y_i G_T(X_i))}{Z_T}$$

$$Z_T = \sum_{i=1}^{n} w_i^{(T)} \exp(-\beta_T y_i G_T(X_i))$$

$$= \sum_{y_i \neq G_T(X_i)} w_i^{(T)} \exp\left(\frac{1}{2} \ln\left(\frac{1-err_T}{err_T}\right)\right) + \sum_{y_i = G_T(X_i)} w_i^{(T)} \exp\left(-\frac{1}{2} \ln\left(\frac{1-err_T}{err_T}\right)\right)$$

$$= \sum_{y_i \neq G_T(X_i)} w_i^{(T)} \left(\frac{1-err_T}{err_T}\right)^{\frac{1}{2}} + \sum_{y_i = G_T(X_i)} w_i^{(T)} \left(\frac{1-err_T}{err_T}\right)^{-\frac{1}{2}}$$

$$= err_T \cdot \left(\frac{1-err_T}{err_T}\right)^{\frac{1}{2}} + (1-err_T)\left(\frac{1-err_T}{err_T}\right)^{-\frac{1}{2}}$$

$$= (1-err_T)^{\frac{1}{2}} \cdot (err_T)^{\frac{1}{2}} + (1-err_T)^{\frac{1}{2}} (err_T)^{\frac{1}{2}}$$

$$= 2\sqrt{(1-err_T)\, err_T}$$

2.b.

Base case:

$T = 1$

$$w_i^{(T+1)} = \frac{1}{n} \frac{\exp(-\beta_T y_i G_T(X_i))}{Z_T}$$

$$= \frac{1}{n Z_T} e^{-y_i \beta_T G_T(X_i)}$$

$$= \frac{1}{n \prod_{t=1}^{T} Z_T} e^{-y_i M(X_i)}$$

since $M(X_i) = \sum_{t=1}^{T} \beta_T G_T(X_i) = \beta_T G_T(X_i)$ when $T = 1$

Inductive Hypothesis:

for all $t \leq T$

$$w_i^{(t+1)} = \frac{1}{n \prod_{i=1}^{t} z_i} e^{-y_i M(x_i)}$$

Inductive step:

Let $t = T+1$

$$w_i^{(T+2)} = \frac{w_i^{(T+1)} \exp(-y_i \beta_{T+1} G_{T+1}(x_i))}{z_{T+1}}$$

By inductive hypothesis

$$= \left(\frac{1}{n \prod_{i=1}^{T} z_i} e^{-y_i \sum_{j=1}^{T} \beta_j G_j(x_i)} \right) \cdot \left(\frac{\exp(-y_i \beta_{T+1} G_{T+1}(x_i))}{z_{T+1}} \right)$$

$$= \frac{1}{n \prod_{i=1}^{T+1} z_i} \left(e^{-y_i \left(\sum_{j=1}^{T} \beta_j G_j(x_i) + \beta_{T+1} G_{T+1}(x_i) \right)} \right.$$

$$= \frac{1}{n \prod_{i=1}^{T+1} z_i} e^{-g_i M(x_i)}$$

thus it also holds for $t = T+1$

By inductive proof it holds for all t.

2.c.

$$\sum_{i=1}^{n} e^{-y_i M(x_i)} = \sum_{y_i \neq \operatorname{sgn}(M(x_i))} e^{|M(x_i)|} + \sum_{y_i = \operatorname{sgn}(M(x_i))} e^{-|M(x_i)|}$$

$$\geq \sum_{y_i \neq \operatorname{sgn}(M(x_i))} 1 + C \qquad C = \sum_{y_i = \operatorname{sgn}(M(x_i))} e^{-|M(x_i)|}$$

where $C > 0$

$$\geq B + C$$

$$\geq B$$

$e^{|M(x_i)|} \geq 1$

$e^{-|M(x_i)|} > 0$

2.d.

$$err_t \leq 0.49 \Rightarrow Z_t \leq 2\sqrt{0.49 \cdot 0.51}$$

$$= 0.9997$$

$$< 0.9998$$

$$B \leq \sum_{i=1}^{n} e^{-y_i M(X_i)} = \sum_{i=1}^{n} w_i^{(T+1)} \cdot n \prod_{t=1}^{T} Z_t$$

$$= n \prod_{t=1}^{T} Z_t$$

$$< n \cdot (0.9998)^T \xrightarrow[T \to \infty]{} 0$$

Since $B < n \cdot (0.9998)^T$ it follows $B \to 0$ as $T \to \infty$ since B can't be negative.

2.e

Since trees are short they will make decisions on a small number of features with the most info gain. And with each iteration of Adaboost a tree focuses on correcting errors of previous trees by reducing the weighted classification error thus selecting the features that are most relevant.

3.a)

$$R_{n \times m} = UDV^T$$

$$R_{ij} = (UDV^T)_{ij} = \sum_{k=1}^{Rank(R)} U_{ik} D_{kk} V_{kj}^T$$

$$= row_i(U) \cdot diag(D) \cdot col_j(V^T)$$

3.b $R = UDV^T$

Set $\sqrt{D} = \begin{bmatrix} \sqrt{\sigma_1} & & 0 \\ & \ddots & \\ 0 & & \sqrt{\sigma_k} \end{bmatrix}$

$\sqrt{D} \cdot \sqrt{D} = D$

So Set $x_i = \text{row}_i(U\sqrt{D})$

$y_i = \text{column}_j(\sqrt{D}V^T)$

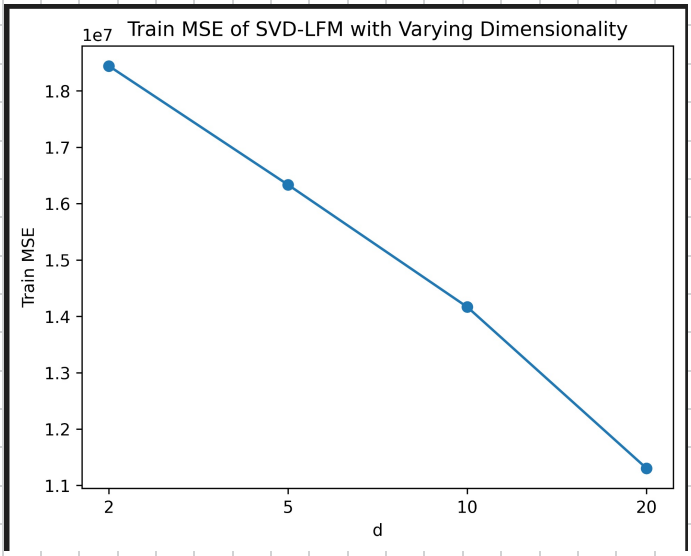
3.C.

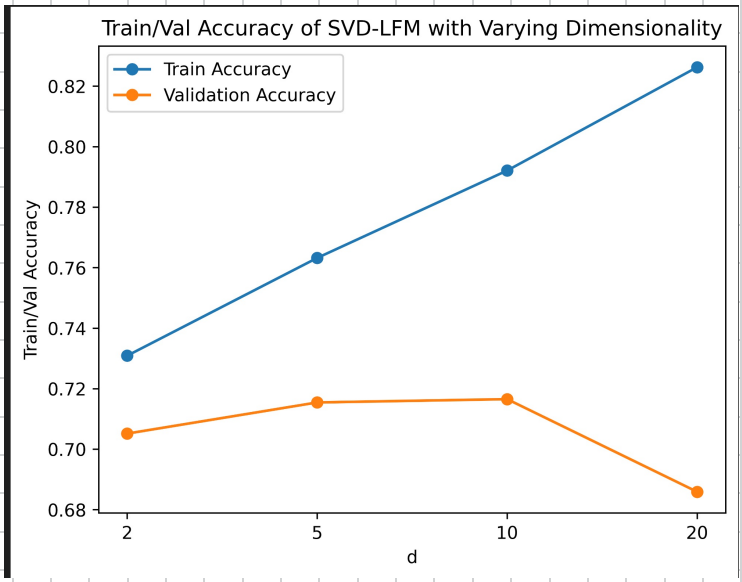
```python
# Part (c): SVD to learn low-dimensional vector representations
def svd_lfm(R):

    # Fill in the missing values in R
    ##### TODO(c): Your Code Here #####
    R = np.nan_to_num(R)
    # Compute the SVD of R
    ##### TODO(c): Your Code Here #####
    U, D, V_tr = scipy.linalg.svd(R, full_matrices=False)

    # Construct user and movie representations
    ##### TODO(c): Your Code Here #####
    D_sr = np.sqrt(D)
    D_diag = np.diag(D_sr)
    user_vecs = U @ D_diag
    movie_vecs = D_diag @ V_tr
    movie_vecs = movie_vecs.T
    return user_vecs, movie_vecs
```

3.D

```python
# 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
    ##### TODO(d): Your Code Here #####

    n = len(R)
    m = len(R[0])

    mse_loss = 0

    for i in range(n):
        for j in range(m):
            if not np.isnan(R[i][j]):
                mse_loss = mse_loss + (np.dot(user_vecs[i],movie_vecs[j]) - R[i][j]) ** 2


    return mse_loss
```

3.e.





seems like d = 10 leads to optimal performance.

3.5. for i,j s.t $R[i,j] \neq NaN$

$$L(\{x_i\}, \{y_i\}) = \sum_i \sum_j (x_i \cdot y_j - R_{ij})^2 + \sum_{i=1}^{n} \|x_i\|_2^2 + \sum_{j=1}^{m} \|y_j\|_2^2$$

$$\nabla_{x_i} L = \sum_j \nabla_{x_i} (x_i \cdot y_j - R_{ij})^2 + 2x_i$$

$$0 = \sum_j 2(x_i \cdot y_j - R_{ij}) y_j + 2x_i$$

$$0 = \sum_j (x_i \cdot y_j) y_j - \sum_j R_{ij} y_j + x_i$$

$$\sum_j R_{ij} y_j = \sum_j x_i^T y_j y_j + x_i$$

$$\sum_j R_{ij} y_j = \left(\sum_j y_j y_j^T + I\right) x_i$$

$$\left(\sum_j y_j y_j^T + I\right)^{-1} \sum_j R_{ij} y_j = x_i$$

$$\nabla_{y_j} L = \sum_i \nabla_{x_i} (x_i \cdot y_j - R_{ij})^2 + 2 y_j$$

$$0 = \sum_i 2 (x_i^T y_j - R_{ij}) x_i + 2 y_j$$

$$\sum_i R_{ij} x_i = \left(\sum_i x_i x_i^T + I \right) y_j$$

$$y_j = \left(\sum_i x_i x_i^T + I \right)^{-1} \sum_i R_{ij} x_i$$

```python
# Part (f): Learn better user/movie vector representations by minimizing loss
# begin solution
best_d = 10# TODO(f): Use best from part (e)
# end solution
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):
    n = len(user_vecs)
    m = len(movie_vecs[0])

    for i in range(n):
        y = np.zeros(m)
        YY = np.zeros((m,m))
        for j in user_rated_idxs[i]:
            y += R[i][j] * movie_vecs[j]
            YY += np.outer(movie_vecs[j],movie_vecs[j].T)
        user_vecs[i] = np.linalg.inv(YY + np.eye(m)) @ y
    return user_vecs
```

```python
# Part (f): Function to update user vectors
def update_movie_vecs(user_vecs, movie_vecs, R, movie_rated_idxs):
    n = len(movie_vecs)
    m = len(user_vecs[0])
    for j in range(n):
        x = np.zeros(m)
        XX = np.zeros((m,m))
        for i in movie_rated_idxs[j]:
            x += R[i][j] * user_vecs[i]
            XX += np.outer(user_vecs[i],user_vecs[i].T)
        movie_vecs[j] = np.linalg.inv(XX + np.eye(m)) @ x
    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}, val accuracy: {val_acc:.4f}')
for opt_iter in range(20):
    user_vecs = update_user_vecs(user_vecs, movie_vecs, np.copy(R), user_rated_idxs)
    movie_vecs = update_movie_vecs(user_vecs, movie_vecs, np.copy(R), movie_rated_idxs)
    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: {train_acc:.4f}, val accuracy: {val_acc:.4f}')
```

Performs:

```
Iteration 19, train MSE: 9320755.69, train accuracy: 0.8161, val accuracy: 0.7149
Iteration 20, train MSE: 9314221.76, train accuracy: 0.8163, val accuracy: 0.7160
```

In e for $d = 10$

train_acc ≈ 0.78

val_acc ≈ 0.71

MSE $\approx$ 14,000,000,

so MSE in part e) was higher while accuracies are roughly the same, but better in f)

4.a.

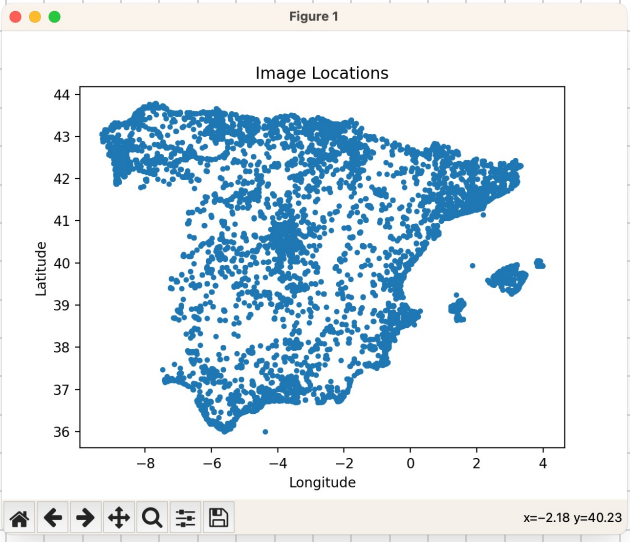


Figure 1
Image Locations
Latitude
Longitude
44
43
42
41
40
39
38
37
36
−8
−6
−4
−2
0
2
4
x=−2.18 y=40.23

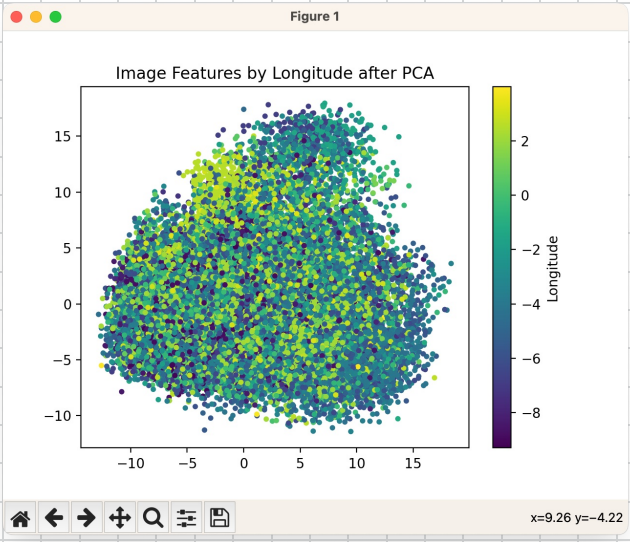


Figure 1
Image Features by Longitude after PCA
Longitude
15
10
5
0
−5
−10
−10
−5
0
5
10
15
2
0
−2
−4
−6
−8
x=9.26 y=−4.22

4.b

test image:

coordinates:

(37.38, -5.99)

1st nearest:

coordinates:

(37.38, -5.99)

2nd nearest:

coordinates:

(40.03, -3.609)

3rd nearest:

Coordinates:

(37.38, -5.99)

1st and 3rd seem to be correct.

```
# Part C: Find the 5 nearest neighbors of test image 53633239060.jpg
knn = NearestNeighbors(n_neighbors=3).fit(train_features)

# Use knn to get the k nearest neighbors of the features of image 53633239060.jpg
##### TODO(c): Your Code Here #####
index = np.where(test_files == "53633239060.jpg")[0]
distances, indices = knn.kneighbors([test_features[index[0]]])
print(distances)
print(train_files[indices])
print("test coordinate: ", test_labels[index[0]])
print("train coordinates: ", train_labels[indices])
```

```
[[0.28561533 0.5634429  0.56347881]]
[['31870484468.jpg' '4554482343.jpg' '53643099910.jpg']]
test coordinate:  [37.380455 -5.993931]
train coordinates:  [[[37.380424 -5.994328]
  [40.036533 -3.609201]
  [37.38637  -5.992387]]]
```

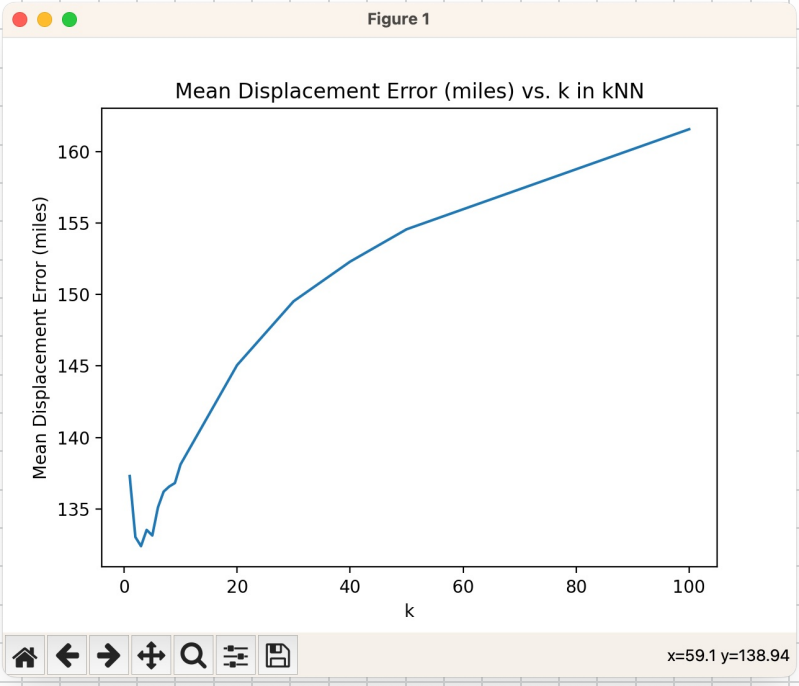
4.c

MDE: 209.86 miles

```
[37.38637  -5.992587]]]
Mean Displacement Error (Miles): 209.86265737332474
```

```
centroid_lat = np.mean(train_labels[:, 0])
centroid_lon = np.mean(train_labels[:, 1])
predictions = np.array([(centroid_lat, centroid_lon)] * len(test_features))
displacements = predictions - test_labels
displacements = np.array([((a[0] * 69)**2, (a[1] * 52)**2) for a in displacements ])
displacements = np.sqrt(np.sum(displacements, axis=1))
mde = np.mean(displacements)
print("Mean Displacement Error (Miles):", mde)
```

4.0



best K = 3

With lowest error = 132.4

```python
def grid_search(train_features, train_labels, test_features, test_labels, is_weighted=False, verbose=True):
    knn = NearestNeighbors(n_neighbors=100).fit(train_features)
    if verbose:
        print(f'Running grid search for k (is_weighted={is_weighted})')
    ks = list(range(1, 11)) + [20, 30, 40, 50, 100]
    mean_errors = []

    for k in ks:
        distances, indices = knn.kneighbors(test_features, n_neighbors=k)
        errors = []
        for i, nearest in enumerate(indices):
            # Evaluate mean displacement error in miles for each test image
            # Assume 1 degree latitude is 69 miles and 1 degree longitude is 52 miles
            y = test_labels[i]
            if (is_weighted):
                weights = [1 / (a + 1e-8) for a in distances[i]]
                mean_lat = np.average(train_labels[nearest][:, 0], weights=weights)
                mean_lon = np.average(train_labels[nearest][:, 1], weights=weights)
            else:
                mean_lat = np.mean(train_labels[nearest][:, 0])
                mean_lon = np.mean(train_labels[nearest][:, 1])
            e = np.sqrt(((y[0] - mean_lat) * 69)**2 + ((y[1] - mean_lon)*52)**2)
            errors.append(e)
        e = np.mean(np.array(errors))
        mean_errors.append(e)
        if verbose:
            print(f'{k}-NN mean displacement error (miles): {e:.1f}')

    # Plot error vs k for k Nearest Neighbors
    if verbose:
        plt.plot(ks, mean_errors)
        plt.xlabel('k')
        plt.ylabel('Mean Displacement Error (miles)')
        plt.title('Mean Displacement Error (miles) vs. k in kNN')
        plt.show()
    return min(mean_errors)
```

4.e

$K=1$ find the closest point so can capture complex patterns thus lower bias.

but very sensitive to the training data thus high variance.

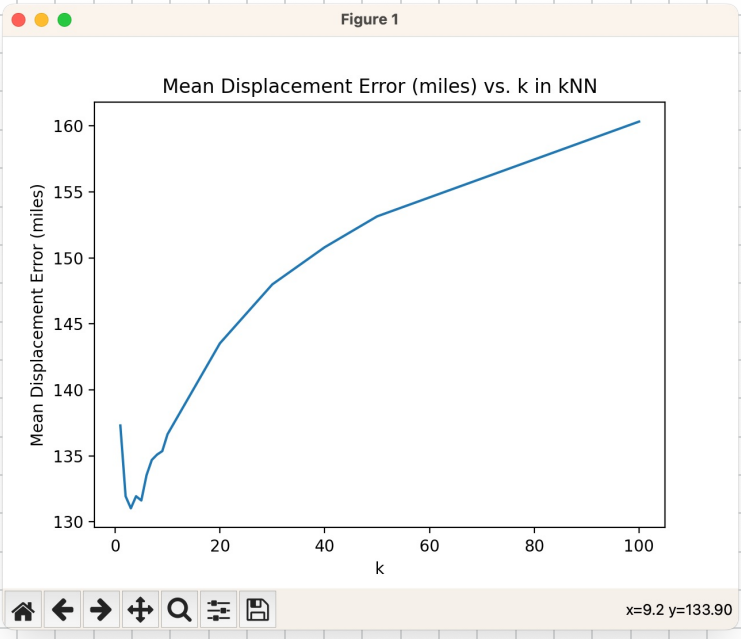
$K=n$ considers all training points, high bias because it doesn't capture underlying patterns, only overall average.

but variance is low since now model is less sensitive to small changes in training set.

for intermediate K there is a tradeoff.

We can see that the balance happens not at the extremes, although closer to $K=1$. Its probably because bias increases very fast with more K while variance decreases slower.

4.5

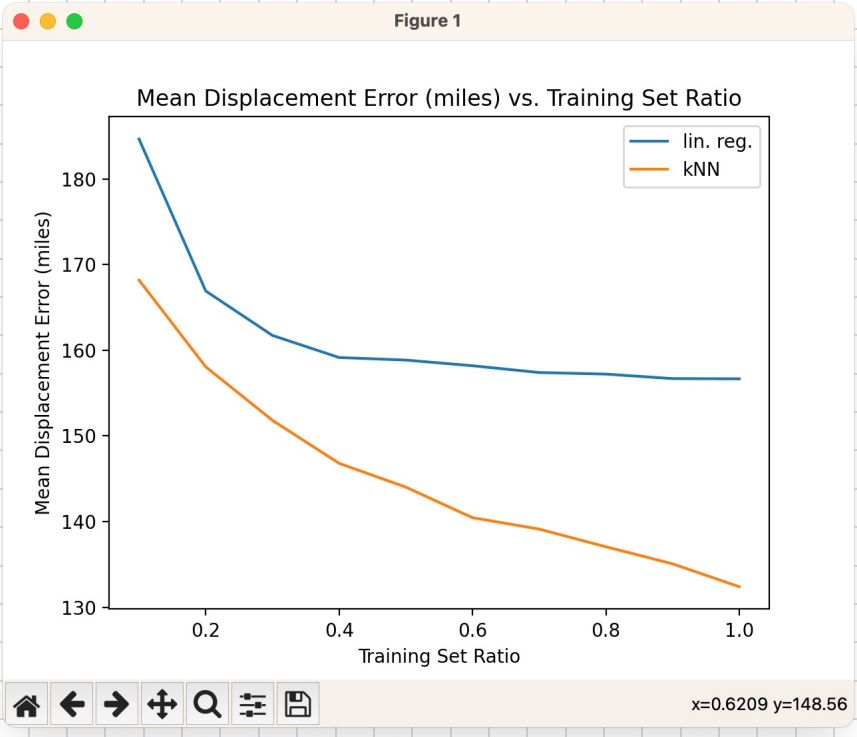


best value for k still 3 with error 131

the difference in error almost negligible usually about a mile closer than dj.

```
    y = test_labels[i]
    if (is_weighted):
        weights = [1 / (a + 1e-8) for a in distances[i]]
        mean_lat = np.average(train_labels[nearest][:, 0], weights=weights)
        mean_lon = np.average(train_labels[nearest][:, 1], weights=weights)
    else:
        mean_lat = np.mean(train_labels[nearest][:, 0])
        mean_lon = np.mean(train_labels[nearest][:, 1])
    e = np.sqrt(((y[0] - mean_lat) * 69)**2 + ((y[1] - mean_lon)*52)**2)
    errors.append(e)
```

4.6



kNN seems to be improving at a faster pace. while lin. reg seemed to plateau.

```python
# Part H: compare to linear regression for different # of training points
mean_errors_lin = []
mean_errors_nn = []
ratios = np.arange(0.1, 1.1, 0.1)
for r in ratios:
    num_samples = int(r * len(train_features))
    ##### TODO(h): Your Code Here #####
    indices = np.random.choice(num_samples, size=num_samples, replace=False)

    train_f_sample = train_features[indices]
    train_l_sample = train_labels[indices]

    model = LinearRegression()
    model.fit(train_f_sample, train_l_sample)

    y_pred = model.predict(test_features)
    e_lin = mean_dis_error(y_pred, test_labels)

    e_nn = grid_search(train_f_sample, train_l_sample, test_features, test_labels, verbose=False)

    mean_errors_lin.append(e_lin)
    mean_errors_nn.append(e_nn)

    print(f'\nTraining set ratio: {r} ({num_samples})')
    print(f'Linear Regression mean displacement error (miles): {e_lin:.1f}')
    print(f'kNN mean displacement error (miles): {e_nn:.1f}')
```

4. Appendix

```python
import matplotlib.pyplot as plt
import numpy as np

from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors


def plot_data(train_feats, train_labels):
    """
    Input:
        train_feats: Training set image features
        train_labels: Training set GPS (lat, lon)

    Output:
        Displays plot of image locations, and first two PCA dimensions vs longitude
    """
    # Plot image locations (use marker='.' for better visibility)
    plt.scatter(train_labels[:, 1], train_labels[:, 0], marker=".")
    plt.title('Image Locations')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.show()

    # Run PCA on training_feats
    ##### TODO(a): Your Code Here #####
    transformed_feats = StandardScaler().fit_transform(train_feats)
    transformed_feats = PCA(n_components=2).fit_transform(transformed_feats)

    # Plot images by first two PCA dimensions (use marker='.' for better visibility)
    plt.scatter(transformed_feats[:, 0],     # Select first column
                transformed_feats[:, 1],     # Select second column
                c=train_labels[:, 1],
                marker='.')
    plt.colorbar(label='Longitude')
    plt.title('Image Features by Longitude after PCA')
    plt.show()
```

```python
def grid_search(train_features, train_labels, test_features, test_labels, is_weighted=False, verbose=True):
    knn = NearestNeighbors(n_neighbors=100).fit(train_features)
    if verbose:
        print(f'Running grid search for k (is_weighted={is_weighted})')
    ks = list(range(1, 11)) + [20, 30, 40, 50, 100]
    mean_errors = []

    for k in ks:
        distances, indices = knn.kneighbors(test_features, n_neighbors=k)
        errors = []
        for i, nearest in enumerate(indices):
            # Evaluate mean displacement error in miles for each test image
            # Assume 1 degree latitude is 69 miles and 1 degree longitude is 52 miles
            y = test_labels[i]
            if (is_weighted):
                weights = [1 / (a + 1e-8) for a in distances[i]]
                mean_lat = np.average(train_labels[nearest][:, 0], weights=weights)
                mean_lon = np.average(train_labels[nearest][:, 1], weights=weights)
            else:
                mean_lat = np.mean(train_labels[nearest][:, 0])
                mean_lon = np.mean(train_labels[nearest][:, 1])
            e = np.sqrt(((y[0] - mean_lat) * 69)**2 + ((y[1] - mean_lon)*52)**2)
            errors.append(e)
        e = np.mean(np.array(errors))
        mean_errors.append(e)
        if verbose:
            print(f'{k}-NN mean displacement error (miles): {e:.1f}')

    # Plot error vs k for k Nearest Neighbors
    if verbose:
        plt.plot(ks, mean_errors)
        plt.xlabel('k')
        plt.ylabel('Mean Displacement Error (miles)')
        plt.title('Mean Displacement Error (miles) vs. k in kNN')
        plt.show()
    return min(mean_errors)
```

```python
def mean_dis_error(predictions, truth):
    displacements = predictions - truth
    displacements = np.array([((a[0] * 69)**2, (a[1] * 52)**2) for a in displacements ])
    displacements = np.sqrt(np.sum(displacements, axis=1))
    return np.mean(displacements)


def main():
    print("Predicting GPS from CLIP image features\n")

    # Import Data
    print("Loading Data")
    data = np.load('im2spain_data.npz')

    train_features = data['train_features']  # [N_train, dim] array
    test_features = data['test_features']    # [N_test, dim] array
    train_labels = data['train_labels']      # [N_train, 2] array of (lat, lon) coords
    test_labels = data['test_labels']        # [N_test, 2] array of (lat, lon) coords
    train_files = data['train_files']        # [N_train] array of strings
    test_files = data['test_files']          # [N_test] array of strings

    # Data Information
    print('Train Data Count:', train_features.shape[0])

    # Part A: Feature and label visualization (modify plot_data method)
    plot_data(train_features, train_labels)

    # Part C: Find the 5 nearest neighbors of test image 53633239060.jpg
    knn = NearestNeighbors(n_neighbors=3).fit(train_features)

    # Use knn to get the k nearest neighbors of the features of image 53633239060.jpg
    ##### TODO(c): Your Code Here #####
    index = np.where(test_files == "53633239060.jpg")[0]
    distances, indices = knn.kneighbors([test_features[index[0]]])
    print(distances)
    print(train_files[indices])
    print("test coordinate: ", test_labels[index[0]])
    print("train coordinates: ", train_labels[indices])
```

```python
    # Part D: establish a naive baseline of predicting the mean of the training set
    ##### TODO(d): Your Code Here #####

    centroid_lat = np.mean(train_labels[:, 0])
    centroid_lon = np.mean(train_labels[:, 1])
    predictions = np.array([(centroid_lat, centroid_lon)] * len(test_features))
    displacements = predictions - test_labels
    displacements = np.array([((a[0] * 69)**2, (a[1] * 52)**2) for a in displacements ])
    displacements = np.sqrt(np.sum(displacements, axis=1))
    mde = np.mean(displacements)
    print("Mean Displacement Error (Miles):", mde)


    # Part E: complete grid_search to find the best value of k
    grid_search(train_features, train_labels, test_features, test_labels)

    # Parts G: rerun grid search after modifications to find the best value of k
    grid_search(train_features, train_labels, test_features, test_labels, is_weighted=True)

    # Part H: compare to linear regression for different # of training points
    mean_errors_lin = []
    mean_errors_nn = []
    ratios = np.arange(0.1, 1.1, 0.1)
    for r in ratios:
        num_samples = int(r * len(train_features))
        ##### TODO(h): Your Code Here #####
        indices = np.random.choice(num_samples, size=num_samples, replace=False)

        train_f_sample = train_features[indices]
        train_l_sample = train_labels[indices]

        model = LinearRegression()
        model.fit(train_f_sample, train_l_sample)

        y_pred = model.predict(test_features)
        e_lin = mean_dis_error(y_pred, test_labels)

        e_nn = grid_search(train_f_sample, train_l_sample, test_features, test_labels, verbose=False)

        mean_errors_lin.append(e_lin)
        mean_errors_nn.append(e_nn)
```

```
        print(f'\nTraining set ratio: {r} ({num_samples})')
        print(f'Linear Regression mean displacement error (miles): {e_lin:.1f}')
        print(f'kNN mean displacement error (miles): {e_nn:.1f}')

    # Plot error vs training set size
    plt.plot(ratios, mean_errors_lin, label='lin. reg.')
    plt.plot(ratios, mean_errors_nn, label='kNN')
    plt.xlabel('Training Set Ratio')
    plt.ylabel('Mean Displacement Error (miles)')
    plt.title('Mean Displacement Error (miles) vs. Training Set Ratio')
    plt.legend()
    plt.show()


if __name__ == '__main__':
    main()
```

3. Appendix

```python
import os
import scipy.io
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

# Load training data from MAT file
R = scipy.io.loadmat('movie_data/movie_train.mat')['train']

# Load validation data from CSV
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]
```

```python
# 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, dtype=object), np.array(movie_rated_idxs, dtype=object)

# Part (c): SVD to learn low-dimensional vector representations
def svd_lfm(R):

    # Fill in the missing values in R
    ##### TODO(c): Your Code Here #####
    R = np.nan_to_num(R)
    # Compute the SVD of R
    ##### TODO(c): Your Code Here #####
    U, D, V_tr = scipy.linalg.svd(R, full_matrices=False)

    # Construct user and movie representations
    ##### TODO(c): Your Code Here #####
    D_sr = np.sqrt(D)
    D_diag = np.diag(D_sr)
    user_vecs = U @ D_diag
    movie_vecs = D_diag @ V_tr
    movie_vecs = movie_vecs.T
    return user_vecs, movie_vecs
```

```python
# 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
    ##### TODO(d): Your Code Here #####

    n = len(R)
    m = len(R[0])

    mse_loss = 0

    for i in range(n):
        for j in range(m):
            if not np.isnan(R[i][j]):
                mse_loss = mse_loss + (np.dot(user_vecs[i],movie_vecs[j]) - R[i][j]) ** 2


    return mse_loss
```

```python
# Part (e): Compute training MSE and val acc of SVD LFM for various d
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[:, :d]))
    train_accs.append(get_train_acc(np.copy(R), user_vecs[:, :d], movie_vecs[:, :d]))
    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.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')
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')


# Part (f): Learn better user/movie vector representations by minimizing loss
# begin solution
best_d = 10# TODO(f): Use best from part (e)
# end solution
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))
```

```python
# Part (f): Function to update user vectors
def update_user_vecs(user_vecs, movie_vecs, R, user_rated_idxs):
    n = len(user_vecs)
    m = len(movie_vecs[0])

    for i in range(n):
        y = np.zeros(m)
        YY = np.zeros((m,m))
        for j in user_rated_idxs[i]:
            y += R[i][j] * movie_vecs[j]
            YY += np.outer(movie_vecs[j],movie_vecs[j].T)
        user_vecs[i] = np.linalg.inv(YY + np.eye(m)) @ y
    return user_vecs

# Part (f): Function to update user vectors
def update_movie_vecs(user_vecs, movie_vecs, R, movie_rated_idxs):
    n = len(movie_vecs)
    m = len(user_vecs[0])
    for j in range(n):
        x = np.zeros(m)
        XX = np.zeros((m,m))
        for i in movie_rated_idxs[j]:
            x += R[i][j] * user_vecs[i]
            XX += np.outer(user_vecs[i],user_vecs[i].T)
        movie_vecs[j] = np.linalg.inv(XX + np.eye(m)) @ x
    return movie_vecs
```

```python
# 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}, val accuracy: {val_acc:.4f}')
for opt_iter in range(20):
    user_vecs = update_user_vecs(user_vecs, movie_vecs, np.copy(R), user_rated_idxs)
    movie_vecs = update_movie_vecs(user_vecs, movie_vecs, np.copy(R), movie_rated_idxs)
    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: {train_acc:.4f}, val accuracy: {val_acc:.4f}')
```