CS 189/289A Introduction to Machine Learning
Spring 2023 Jonathan Shewchuk HW7

Due: Wednesday, May 1 at 11:59 pm

Deliverables:

1. Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment entitled "Homework 7 Write-Up". In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.

   - In your write-up, please state with whom you worked on the homework.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats.

     "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."

2. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 7 Code". Yes, you must submit your code twice: once in your PDF write-up following the directions as described above so the readers can easily read it, and once in compilable/interpretable form so the readers can easily run it. Do **NOT** include any data files we provided. Please include a short file named README listing your name, student ID, and instructions on how to reproduce your results. Please take care that your code doesn't take up inordinate amounts of time or memory to run. If your code cannot be executed, your solution cannot be verified.

1 Honor Code

Declare and sign the following statement:

"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature : Christian R. Aurora

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

2 The Training Error of AdaBoost

Recall that in AdaBoost, our input is an $n \times d$ design matrix X with n labels $y_i = \pm 1$, and at the end of iteration T the importance of each sample is reweighted as

$$w_i^{(T+1)} = w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i)), \quad \text{where} \quad \beta_T = \frac{1}{2} \ln\left(\frac{1 - \text{err}_T}{\text{err}_T}\right) \quad \text{and} \quad \text{err}_T = \frac{\sum_{y_i \neq G_T(X_i)} w_i^{(T)}}{\sum_{i=1}^n w_i^{(T)}}.$$

Note that err_T is the weighted error rate of the classifier G_T. Recall that $G_T(z)$ is ± 1 for all points z, but the metalearner has a non-binary decision function $M(z) = \sum_{t=1}^T \beta_t G_t(z)$. To classify a test point z, we calculate $M(z)$ and return its sign.

In this problem we will prove that if every learner G_t achieves 51% accuracy (that is, only slightly above random), AdaBoost will converge to zero training error. (If you get stuck on one part, move on; all five parts below can be done without solving the other parts, and parts (c) and (e) are the easiest.)

(a) We want to change the update rule to "normalize" the weights so that each iteration's weights sum to 1; that is, $\sum_{i=1}^n w_i^{(T+1)} = 1$. That way, we can treat the weights as a discrete probability distribution over the sample points. Hence we rewrite the update rule in the form

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

for some scalar Z_T. Show that if $\sum_{i=1}^n w_i^{(T)} = 1$ and $\sum_{i=1}^n w_i^{(T+1)} = 1$, then

$$Z_T = 2\sqrt{\text{err}_T(1 - \text{err}_T)}. \tag{2}$$

Hint: sum over both sides of (1), then split the right summation into misclassified points and correctly classified points.

Solution:

$Z^T = \frac{\sum_{i=1}^n [w_i^T e^{-\beta_T y_i G_T(x_i)}]}{\sum_{i=1}^n w_i^{T+1}}$

$Z^T = \frac{\sum_{i=1}^n [w_i^T e^{-(\frac{1}{2} ln(\frac{(1-err_T)}{(err_T)}) y_i G_T(x_i)}]}{\sum_{i=1}^n w_i^{T+1}}$

When $y_i == G_T(x_i)$ $y_i = 1$ and $G_T(x_i) = 1$:

$\sum_{i=1}^n w_i^T = 1$ and $\sum_{i=1}^n w_i^{T+1} = 1$

$\Rightarrow Z^T = (1) e^{-\frac{1}{2} ln(\frac{(1-err_T)}{err_T})^{-\frac{1}{2}}}$

$\Rightarrow Z^T = \sqrt{\frac{err_T}{1-err_T}}$

$\Rightarrow \sum_{i:y_i \neq G_T(x_i)}^n w_i^T = 1 - err_T$

When $y_i \neq G_T(x_i)$ $y_i = -1$ and $G_T(x_i) = -1$:

$\sum_{i=1}^n w_i^T = 1$ and $\sum_{i=1}^n w_i^{T+1} = 1$

$\Rightarrow Z^T = (1) e^{-\frac{1}{2} ln(\frac{(1-err_T)}{err_T})^{-\frac{1}{2}}} (-1)(-1)$

$\Rightarrow Z^T = \sqrt{\frac{1-err_T}{err_T}}$

$\Rightarrow \sum_{i:y_i \neq G_T(x_i)}^n w_i^T = err_T$

$Z_T = \sum_{i:y_i = G_T(x_i)}^n w_i^T \sqrt{\frac{err_T}{1-err_T}} + \sum_{i:y_i \neq G_T(x_i)}^n w_i^T \sqrt{\frac{1-err_T}{err_T}}$

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

$Z^T = \sqrt{(err_T)(1 - err_T)} + \sqrt{(err_T)(1 - err_T)}$

$Z^T = 2\sqrt{(err_T)(1 - err_T)}$

(b) The initial weights are $w_1^{(1)} = w_2^{(1)} = \cdots = w_n^{(1)} = \frac{1}{n}$. Show that

$$w_i^{(T+1)} = \frac{1}{n \prod_{t=1}^{T} Z_t} e^{-y_i M(X_i)}. \tag{3}$$

Solution:

$w^{(T+1)} = \frac{w_i^T e^{-\beta_T y_i G_T(x_i)}}{Z^T}$

$\beta_T = \frac{1}{2} ln(\frac{1-err_T}{err_T})$

$err_T = \frac{\sum y_i \neq G_T(x_i) w_i^T}{\sum_{i=1}^{n} w_i^T}$

When $y_i = G_T(x_i)$ $y_i = 1$ and $G_T(x_i) = 1$ and $\beta_T = \sqrt{\frac{err_T}{1-err_T}}$

When $y_i \neq G_T(x_i)$ $y_i = -1$ and $G_T(x_i) = -1$ and $\beta_T = \sqrt{\frac{1-err_T}{err_T}}$

$M(x_i) = \sum_{t=1}^{T} \beta_t G_t(x_i)$

Given that $w_n^t = \frac{1}{n}$

$w^{T+1} = \frac{1}{n} \frac{e^{-M(x_i)y_i}}{2\sqrt{err_T(1-err_T)}}$

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

(c) Let B (for "bad") be the number of sample points out of n that the metalearner classifies incorrectly. Show that

$$\sum_{i=1}^{n} e^{-y_i M(X_i)} \geq B. \tag{4}$$

Hint: split the summation into misclassified points and correctly classified points.

Solution:

$\sum_{i=1}^{n} e^{-y_i M(x_i)} = \sum_{i:y_i=G_T(x_i)} e^{-y_i M(x_i)} + \sum_{i:y\neq G_T(x_i)}^{n} e^{-y_i M(x_i)} \geq B$

$\sum_{i:y_i=G_T(x_i)} e^{-y_i M(x_i)} + \sum_{i:y\neq G_T(x_i)}^{n} e^{-y_i M(x_i)} \geq \sum_{i:y\neq G_T(x_i)}^{n} e^{-y_i M(x_i)}$

Subtract $\sum_{i:y\neq G_T(x_i)}^{n} e^{-y_i M(x_i)}$ from both sides:

$\sum_{i:y_i=G_T(x_i)} e^{-y_i M(x_i)} \geq 0$

$\therefore \sum_{i=1}^{n} e^{-y_i M(x_i)} \geq B$

(d) Use the formulas (2), (3), and (4) to show that if $\text{err}_t \leq 0.49$ for every learner G_t, then $B \to 0$ as $T \to \infty$. Hint: (2) implies that every $Z_t < 0.9998$. How can you combine this fact with (3) and (4)?

Solution:

$Z_T = 2\sqrt{(.49)(1-.49)} = 0.49989$

$w_i^{T+1} = \frac{1}{n} \cdot \frac{1}{(0.49)^t} e^{-y_i M(x_i)}$

$lim_{t\to\infty}(\frac{1}{n} \cdot \frac{1}{(0.49)^t} e^{-y_i M(x_i)} = 0$

Therefore as the number of classifiers increases the no. of sample

points out of n that the meta-learner classifies incorrectly approaches 0.

(e) Explain briefly why AdaBoost with short decision trees is a form of subset selection when the number of features is large.

Solution:

AdaBoost with short decision trees is a form of subset selection because short decision trees limit the no. of dichotomies, meaning less splits on less data. Therefore, by using short decision trees we can measure which features or subsets of data perform better. Similarly to finding the optimal split in decision trees by measuring entr

3 Movie Recommender System

In this problem, we will build a personalized movie recommender system! Suppose that there are $m = 100$ movies and $n = 24{,}983$ users in total, and each user has watched and rated a subset of the m movies. Our goal is to recommend more movies for each user given their preferences.

Our historical ratings dataset is given by a matrix $R \in \mathbb{R}^{n \times m}$, where R_{ij} represents the rating that user i gave movie j. The rating is a real number in the range $[-10, 10]$: a higher value indicates that the user was more satisfied with that movie. If user i did not rate movie j, $R_{ij} = \texttt{NaN}$.

The provided `movie_data/` directory contains the following files:

- `movie_train.mat` contains the training data, i.e. the matrix R of historical ratings specified above.
- `movie_validate.txt` contains user-movie pairs that don't appear in the training set (i.e. $R_{ij} = \texttt{NaN}$). Each line takes the form "`i,j,s`", where `i` is the user index, `j` is the movie index, and `s` indicates the user's rating of the movie. Contrary to the training set, the rating here is binary: if the user liked the movie (positive rating), `s = 1`, and if the user did not like the movie (negative rating), `s = -1`.

We also provide `movie_recommender.py`, containing starter code for building your recommender system.

The singular value decomposition (SVD) is a powerful tool to decompose and analyze matrices. In lecture, we saw that the SVD can be used to efficiently compute the principal coordinates of a data matrix for PCA. Here, we will see that SVD can also produce dense, compact featurizations of the variables in the input matrix (in our case, the m movies and n users). This application of SVD is known as Latent Semantic Analysis (Wikipedia), and we can use it to construct a Latent Factor Model (LFM) for personalized recommendation.

Specifically, we want to learn a feature vector $x_i \in \mathbb{R}^d$ for user i and a feature vector $y_j \in \mathbb{R}^d$ for movie j such that the inner product $x_i \cdot y_j$ approximates the rating R_{ij} that user i would give movie j.

(a) Recall the SVD definition for a matrix $R \in \mathbb{R}^{n \times m}$ from Lecture 21: $R = UDV^\top$. Write an expression for R_{ij}, user i's rating for movie j, in terms of only the contents of U, D, and V. **Solution:**

$R_{ij} = U_{ij} \cdot D_{ij} \cdot V_{ij}^T$

(b) Based on your answer above, what should we choose as our user and movie feature vector representations x_i and y_j to achieve 100% training accuracy (correctly predict all known ratings in R)?

Solution:

Based on my answer above we should chose U to represent x_i and V as y_i to achieve 100% training accuracy.

(c) In the provided `movie_recommender.py`, complete the code for part (c) by filling in the missing parts of the function `svd_lfm`. Start by replacing all missing (`NaN`) values in R with 0. Then, compute the SVD of the resulting matrix, and follow your above derivations to compute the feature vector representations for each user and movie. Note: do **not** center the data matrix; this is not PCA.

Once you are finished with the code, the **rows** of the `user_vecs` array should contain the feature vectors for users (so the ith row of `user_vecs` is x_i), and the **rows** of `movie_vecs` should contain the feature vectors for movies (so the jth row of `movie_vecs` is y_j).

Hint: we recommend using `scipy.linalg.svd` to compute the SVD, with `full_matrices = False`. This returns U ($n \times m$), D (as a vector of m singular values in descending order, **not** a diagonal matrix), and $V^\top$ ($m \times m$) in that order.

```
def svd_lfm(R):

    # Impute Nan values in R with 0
    idx = np.isnan(R[:,])
    R[idx] = 0

    # Compute eigenvalues and eigenvectors of R.T @ R
    eigenvalues, eigenvectors = np.linalg.eig(R.T @ R)

    # sort the eigenvalues indices in descending order
    eigenvaluesSortedIdx = np.argsort(eigenvalues)[::-1]

    # sort eigenvalues and eigenvectors in descending order
    eigenvalues = eigenvalues[eigenvaluesSortedIdx]

    # sort eigenvectors in descending order (right singular vectors: describes the columns of R)
    V= eigenvectors[:, eigenvaluesSortedIdx]

    # compute the SVD of R
    # construct Diagonal matrix D (singular values of R)
    # meaures the amount of variance captured by each corresponding vector in U, and V
    # i.e. the first/largest eigenvalue represents the importance/strength of the pattern associated with
     ↪ the
    # first columns of U and V
    D = np.diag(np.sqrt(eigenvalues))

    # compute U (left singular vectors of R: describe the rows of R)
    U = (R @ V @ np.linalg.pinv(D) / np.linalg.norm(R @ V @ np.linalg.pinv(D), axis=0))

    # construct the matrix R = UDV.T
    R_approx = U @ D @ V.T

    return U, V
```

(d) To measure the training performance of the model, we can use the mean squared error (MSE) loss,

$$\text{MSE} = \sum_{(i,j)\in S} (x_i \cdot y_j - R_{ij})^2 \quad \text{where } S := \{(i,j) : R_{ij} \neq \texttt{NaN}\}.$$

Complete the code to implement the training MSE computation within the function `get_train_mse`.

Solution:

```
# 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
    xy = user_vecs @ movie_vecs.T
    xyr = (xy - R) ** 2
    return np.nanmean(xyr)
```

(e) Our model as constructed may achieve 100% training accuracy, but it is prone to overfitting. Instead, we would like to use lower-dimensional representations for x_i and y_j to approximate our known ratings closely while still generalizing well to unknown user/movie combinations. Specifically, we want each x_i and y_j to be d-dimensional for some $d < m$, such that only the top d features are used to make predictions $x_i \cdot y_j$. The "top d features" are those corresponding to the d largest singular values: use this as a hint for how to prune your current user/movie vector representations to d dimensions.

In your code, compute pruned user/movie vector representations with $d = 2, 5, 10, 20$. Then, for each setting, compute the training MSE (using the function you implemented in part (d)), the training accuracy (using the provided `get_train_acc`), and the validation accuracy (using the provided `get_val_acc`). Plot the training MSE as a function of d on one plot, and the training and validation accuracies as a function of d together on a separate plot. The code for this part is already included in the starter code, so if your training MSE function from part (d) is implemented correctly, the required plots should be saved to your project directory.

Comment on which value of d leads to optimal performance.

Hint: as a sanity check, if implemented correctly, your best validation accuracy should be about 71%.

Solution:

```
# 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
best_d = 2
```

(f) For sparse data, replacing all missing values with zero, as we did in part (c), is not a very satisfying solution. A missing value in the training matrix R means that the user has not watched the movie; this does not imply that the rating should be zero. Instead, we can learn our user/movie vector representations by minimizing the MSE loss, which only incorporates the loss on rated movies ($R_{ij} \neq \texttt{NaN}$).

Let's define a loss function

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

where S has the same definition as in the MSE. This is similar to the original MSE loss, except with two additional regularization terms to prevent the norms of the user/movie vectors from getting too large.

Implement an algorithm to learn vector representations of dimension d, the optimal value you found in part (e), for users and movies by minimizing $L(\{x_i\}, \{y_j\})$.

We suggest employing an alternating minimization scheme. First, randomly initialize x_i and y_j for all i, j. Then, minimize the above loss function with respect to the x_i by treating the y_j as constant vectors, and subsequently minimize the loss with respect to the y_j by treating the x_i as constant vectors. Repeat these two steps for a number of iterations. Note that when one of the x_i or y_j are constant, minimizing the loss function with respect to the other component has a closed-form solution. **Derive this solution first in your report, showing all your work**.

The starter code provides a template for this algorithm. Start by inputting your best d value from part (e) to initialize the user and movie vectors, and then implement the functions to update the user and movie vectors (holding the other constant) to their loss-minimizing values.

- To improve efficiency, we recommend using the `user_rated_idxs` and `movie_rated_idxs` arrays provided, which contain the indices of movies that each user rated and the indices of users that rated each movie (respectively), to iterate through the non-`NaN` values of R in the update functions.
- Run these 2 update steps for 20 iterations. Include your final training MSE, training accuracy, and validation accuracy on your report, and compare these results with your best results from part (e).

Solution:

$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{m} 2(x_i y_j - R_{ij})(y_j) + 2x_i$

$\frac{\partial L}{\partial y_i} = \sum_{i=1}^{n} 2(x_i y_j - R_{ij})(x_i) + 2y_j$

```
    # Part (f): Learn better user/movie vector representations by minimizing loss
best_d = 2
np.random.seed(20)
randomized_user_vecs = np.random.random((R.shape[0], best_d))
randomized_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
    out = np.zeros(user_vecs.shape)
    learning_rate = .0001

    for i in range(R.shape[0]):
        temp = np.zeros(user_vecs[i].shape)
        for j in user_rated_idxs[i]:
```

```
            temp += (2 * ((user_vecs[i] @ movie_vecs[j] - R[i, j]) * movie_vecs[j]) + (2 * 1/5 * user_vecs[
    ↪ i]))
        out[i] = user_vecs[i] - learning_rate * temp

    return out

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

    learning_rate = .0001
    out = np.zeros(movie_vecs.shape)
    for j in range(R.shape[1]):
        temp = np.zeros(movie_vecs[j].shape)
        for i in movie_rated_idxs[j]:
            temp += 2 * (user_vecs[i] @ movie_vecs[j] - R[i, j]) * user_vecs[i] + (2 * 1/5 * movie_vecs[j])
        out[j] = movie_vecs[j] - learning_rate * temp

    return out
```

Iteration 20, train MSE: 22.53, train accuracy: 0.7027, val accuracy: 0.6762

4 IM2SPAIN: Nearest Neighbors for Geo-location

For this problem, we will use nearest neighbors (NN or k-NN) to predict latitude and longitude coordinates of images from their CLIP embeddings. You'll be modifying starter code in the provided `im2spain` directory.

We are using a dataset of images scraped from Flickr with geo-tagged locations within Spain. Each image has been processed with OpenAI's CLIP image model (`https://github.com/openai/CLIP`) to produce features that can be used with k-NN.

The CLIP model was not explicitly trained to predict coordinates from images, but from task-agnostic pre-training on a large web-crawl dataset of captioned images has learned a generally useful mapping from images to embedding vectors. These feature vectors turn out to encode various pieces of information about the image content such as object categories, textures, 3D shapes, etc. In fact, these very same features were used to filter out indoor images from outdoor images in the construction of this dataset.

Note: Throughout the problem we use MDE which stands for Mean Displacement Error (in miles). Displacement is the (technically spherical) distance between the predicted coordinates and ground truth coordinates. Since all our images are located within a relatively small region of the globe, we can approximate spherical distances with Euclidean distances by treating latitude/longitude as cartesian coordinates. Assume 1 degree longitude is equal to 69 miles and 1 degree latitude is 52 miles in this problem.

Deliverables: Include your modified `im2spain_starter.py` script in your submission. Your submitted file should include all modifications requested in this problem.

(a) Let's visualize the data. Using matplotlib and scikit-learn, plot the image locations and modify the code to apply PCA to the image features (remember to re-center the features first) in the `plot_data` method of `im2spain_starter.py`. Plot the data in its first two PCA dimensions, colored by longitude coordinate (east-west position).

Solution:

```
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
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()
```

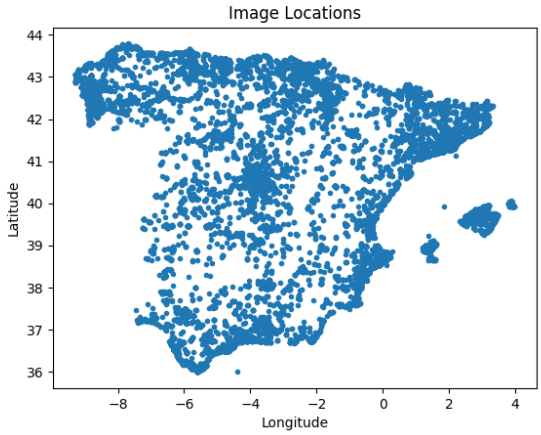


Image Locations
Latitude
Longitude
44
43
42
41
40
39
38
37
36
−8
−6
−4
−2
0
2
4

Figure 1: 4a

(b) Modify the starter code in `im2spain_starter.py` to find the three nearest neighbors in the training set of the test image file `53633239060.jpg`. Include those three image files (as images) in your submission. Now look at their coordinates. How many of the 3 nearest neighbors are "correct"?

Solution:

```
# Find the 3 nearest neighbors of test image 53633239060.jpg

# Fit the model with the training data
knn = NearestNeighbors(n_neighbors=3).fit(train_features)

# Find image data in test set
image = '53633239060.jpg'
image_data = test_features[np.where(test_files == image)]

# Find the 3 nearest neighbors of the `image`
nearestNeighbors = knn.kneighbors(image_data, 3, return_distance=False)[0]

# Extract the data of 3 nearest images to the test image
imageOneIdx = nearestNeighbors[0]
imageTwoIdx = nearestNeighbors[1]
imageThreeIdx = nearestNeighbors[2]

print(train_files[imageOneIdx])
print(train_files[imageTwoIdx])
print(train_files[imageThreeIdx])

Output:
31870484468.jpg
4554482343.jpg
53643099910.jpg
```

(c) Before we begin with our k-NN model, let's first establish a naive constant baseline of simply predicting the training set centroid (coordinate-wise average) location for every test image. Modify the code in `im2spain_starter.py` to implement the constant baseline. What is its MDE in miles?

Solution:

```
# (c): establish a naive baseline of predicting the mean of the training set
    meanLongitude = np.mean(train_labels[:, 0]) # mean longitude
    meanLatitude = np.mean(train_labels[:, 1]) # mean latitude

    baselinePredictions = np.full(test_labels.shape, (meanLongitude, meanLatitude))

    displacement = test_labels - baselinePredictions

    meanLongitudeDisplacement = (np.mean(displacement[:, 0])) * 52
    meanLatitudeDisplacement = (np.mean(displacement[:, 1])) * 69

    print(f"MDE Longitude in Miles: {meanLongitudeDisplacement}")
    print(f"MDE Latitude in Miles: {meanLatitudeDisplacement}")

    Output:
    MDE Longitude in Miles: -0.9376283288002014
    MDE Latitude in Miles: -1.9224216621369123
```

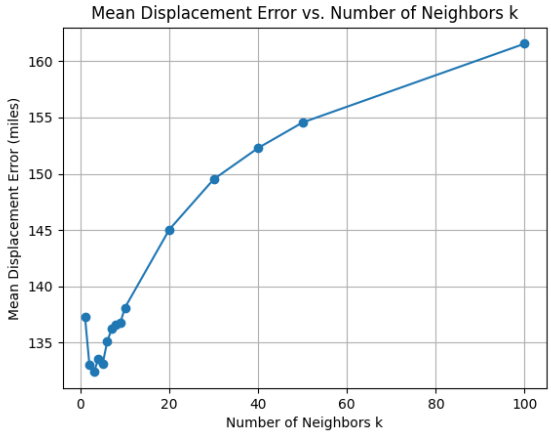
(d) The main hyperparameter of a k-nearest neighbor classifier is k itself. Use a 1-D grid search in `im2spain_starter.py` to create a plot of the MDE (in miles) of k-NN regression versus k, where k is the number of neighbors. Include your plot in your write-up. What is the best value of k? What is the lowest error?

Solution:

```
def grid_search(train_features, train_labels, test_features, test_labels, verbose=True):
    knn = NearestNeighbors().fit(train_features)
    ks = list(range(1, 11)) + [20, 30, 40, 50, 100]
    mean_errors = []

    for k in ks:
        knn.n_neighbors = k
        distances, indices = knn.kneighbors(test_features, return_distance=True)

        errors = []
        for i, nearest_indices in enumerate(indices):
            # Calculate average lat and lon of the k-nearest neighbors
            avg_lat = np.mean(train_labels[nearest_indices, 0])
            avg_lon = np.mean(train_labels[nearest_indices, 1])

            # Calculate displacement error in miles
            lat_error = abs(avg_lat - test_labels[i, 0]) * 69  # latitude conversion to miles
            lon_error = abs(avg_lon - test_labels[i, 1]) * 52  # longitude conversion to miles

            # Euclidean distance in miles as the mean error for this test point
            error = np.sqrt(lat_error**2 + lon_error**2)
            errors.append(error)

        mean_error = np.mean(errors)
        mean_errors.append(mean_error)
        if verbose:
            print(f'{k}-NN mean displacement error (miles): {mean_error:.2f}')

    # Plotting the results
    if verbose:
        plt.plot(ks, mean_errors, marker='o')
        plt.xlabel('Number of Neighbors k')
        plt.ylabel('Mean Displacement Error (miles)')
        plt.title('Mean Displacement Error vs. Number of Neighbors k')
        plt.grid(True)
        plt.show()

    return min(mean_errors)
```

Optimal K = 3

Lowest Error: 131.03


Mean Displacement Error vs. Number of Neighbors k
Mean Displacement Error (miles)
Number of Neighbors k
160
155
150
145
140
135
0
20
40
60
80
100

(e) Explain your plot in (d) in terms of bias and variance. (In the definitions of *bias* and *variance*, you should think of the ground truth function g and the predicted hypothesis h as both being in the form of a longitude and a latitude, and you should assume that we integrate the bias-squared and the variance over the probability distribution of Spain travel photos that people might take.) In particular, given n training points, how is the bias different for $k = 1$ versus $k = n$? How is the variance different for $k = 1$ versus $k = n$? What happens for intermediate values of k?

Solution:

As the number of neighbors k increases the amount of variance captured increases,

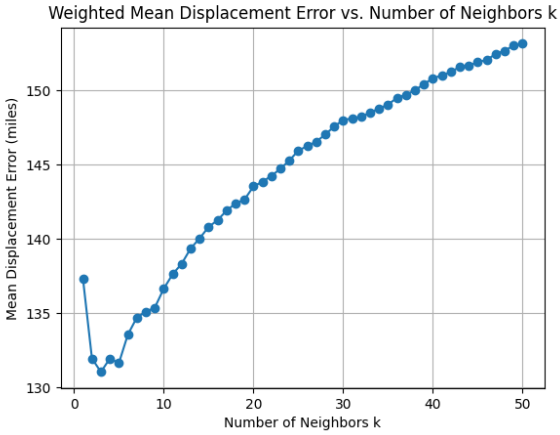
therefore the bias increases as well, consequently, the error increases also because the model is getting better and more prone to over fitting.(More biased towards training data).

(f) We do not need to weight every neighbor equally: closer neighbors may be more relevant. For this problem, weight each neighbor by the inverse of its distance (in feature space) to the test point by modifying `im2spain_starter.py`. Plot the error of k-NN regression with distance weighting vs. k, where k is the number of neighbors. What is the best value of k? What is the MDE in miles? How does performance compare to part (e)?

Note: When computing the inverse distance, add a small value (e.g. 10^{-8}) to the denominator to avoid division by zero.

Solution:

```
def weighted_grid_search(train_features, train_labels, test_features, test_labels, verbose=True):
knn = NearestNeighbors().fit(train_features)
ks = range(1, 51)  # Adjust range of k as needed
mean_errors = []

for k in ks:
    knn.n_neighbors = k
    distances, indices = knn.kneighbors(test_features, return_distance=True)

    errors = []
    for i, (dist, nearest_indices) in enumerate(zip(distances, indices)):
        weights = 1 / (dist + 1e-8)  # Avoid division by zero
        sum_weights = np.sum(weights)

        weighted_lat = np.sum(weights * train_labels[nearest_indices, 0]) / sum_weights
        weighted_lon = np.sum(weights * train_labels[nearest_indices, 1]) / sum_weights

        lat_error = abs(weighted_lat - test_labels[i, 0]) * 69
        lon_error = abs(weighted_lon - test_labels[i, 1]) * 52

        error = np.sqrt(lat_error**2 + lon_error**2)
        errors.append(error)

    mean_error = np.mean(errors)
    mean_errors.append(mean_error)
    if verbose:
        print(f'{k}-NN weighted mean displacement error (miles): {mean_error:.2f}')

# Plotting the results
if verbose:
    plt.plot(ks, mean_errors, marker='o')
    plt.xlabel('Number of Neighbors k')
    plt.ylabel('Mean Displacement Error (miles)')
    plt.title('Weighted Mean Displacement Error vs. Number of Neighbors k')
    plt.grid(True)
    plt.show()

return min(mean_errors)
```


Weighted Mean Displacement Error vs. Number of Neighbors k
Mean Displacement Error (miles)
Number of Neighbors k
130
135
140
145
150
0
10
20
30
40
50

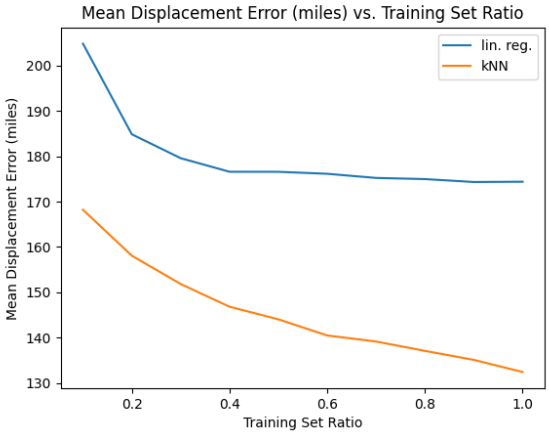
(g) k-NN yields a *non-parametric* model which means its complexity can grow without bound as we increase the amount of training data. This is in contrast to *parametric* models such as linear regression that assume a fixed number of parameters, so the complexity of the model is bounded even if trained with infinite data. (We typically think of modern deep neural nets as functionally non-parametric, though they technically have a finite parameter size, because when we have more data we usually add more parameters.)

Let's compare the performance of k-NN with linear regression at different sizes of training datasets to get a sense of their respective "scaling curves". Plot the test error of both k-NN and linear regression for various percentages of training data. Which method would you expect to continue improving with twice as much training data?

Note: use the optimal value of k at each training dataset size by running grid search.

Solution:

```
#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:
    # determine split size
    num_samples = int(r * len(train_features))
    print('samples:', num_samples)

    # Define regression model
    regression = LinearRegression()

    # split training data
    trainingData = train_features[:num_samples]
    trainingLabels = train_labels[:num_samples]

    regression.fit(trainingData, trainingLabels)
    linRegPred = regression.predict(test_features)

    # Calculate displacement error in miles
    lat_error = abs(linRegPred[:, 1] - test_labels[:, 1]) * 69
    lon_error = abs(linRegPred[:, 0] - test_labels[:, 0]) * 52

    # Euclidean distance in miles as the mean error for this test point
    e_lin = np.mean(np.sqrt(lat_error**2 + lon_error**2))
    mean_errors_lin.append(e_lin)

    e_nn = grid_search(trainingData, trainingLabels, test_features, test_labels, verbose=False)
    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}')


print(len(mean_errors_lin))
print(len(mean_errors_nn))

# 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()
```


Mean Displacement Error (miles) vs. Training Set Ratio
lin. reg.
kNN
Mean Displacement Error (miles)
200
190
180
170
160
150
140
130
0.2
0.4
0.6
0.8
1.0
Training Set Ratio