CS 189 Introduction to Machine Learning
Spring 2023 Jonathan Shewchuk

HW7

Due: Friday, May 5 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 : _____

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: We need $\sum_{i=1}^n w_i^{(T+1)} = 1$, so

$$\begin{aligned} Z_T &= \sum_{i=1}^n w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i)) \\ &= \sum_{y_i = G_T(X_i)} w_i^{(T)} e^{-\beta_T} + \sum_{y_i \neq G_T(X_i)} w_i^{(T)} e^{\beta_T} \end{aligned}$$

$$
\begin{aligned}
&= e^{-\beta_T} \sum_{y_i = G_T(X_i)} w_i^{(T)} + e^{\beta_T} \sum_{y_i \neq G_T(X_i)} w_i^{(T)} \\
&= e^{-\beta_T}(1 - \mathrm{err}_T) + e^{\beta_T}\mathrm{err}_T \\
&= \sqrt{\frac{\mathrm{err}_T}{1-\mathrm{err}_T}}(1-\mathrm{err}_T) + \sqrt{\frac{1-\mathrm{err}_T}{\mathrm{err}_T}}\mathrm{err}_T \\
&= 2\sqrt{\mathrm{err}_T(1-\mathrm{err}_T)}.
\end{aligned}
$$

(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: By induction,

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

(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:

$$
\begin{aligned}
\sum_{i=1}^n e^{-y_i M(X_i)} &= \sum_{y_i M(X_i) \leq 0} e^{-y_i M(X_i)} + \sum_{y_i M(X_i) > 0} e^{-y_i M(X_i)} \\
&\geq \sum_{y_i M(X_i) \leq 0} 1 + \sum_{y_i M(X_i) > 0} 0 \\
&= B.
\end{aligned}
$$

(d) Use the formulas (2), (3), and (4) to show that if $\mathrm{err}_t \leq 0.49$ 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: As every $\mathrm{err}_t \leq 0.49$, every $Z_t \leq 2\sqrt{0.49 \cdot 0.51} < 0.9998$, which is a number strictly less than 1. Therefore, $\lim_{T\to\infty} \prod_{t=1}^T Z_t = 0$, and

$$
B \leq \sum_{i=1}^n e^{-y_i M(X_i)}
$$

$$= n\left(\prod_{t=1}^{T} Z_t\right)\sum_{i=1}^{n} w_i^{(T+1)}$$
$$= n\prod_{t=1}^{T} Z_t$$

also approaches zero in the limit as $T \to \infty$.

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

Solution: Short decision trees don't look at many features. If there are features that don't improve the metalearner's predictive power enough, they won't be used by any of the decision trees.

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:

$$R_{ij} = (x_i D) \cdot y_j$$

Where:
x_i is the ith row of U, as a row vector
D is the diagonal matrix of singular values
y_j is the jth row of V (jth col of $V^\top$), as a row vector.
Note that combining D with $V^\top$ or splitting D across U and $V^\top$ is also equally valid.

(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: From above, it follows that the user feature vectors should be the rows of UD, and the movie feature vectors should be the rows of V (columns of $V^\top$), such that on the training set $R_{ij} = (x_i D) \cdot y_j$.

(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.

Solution:

```
def svd_lfm(R):

    # Fill in the missing values in R
    R[np.isnan(R)] = 0

    # Compute the SVD of R
    U, D, Vt = scipy.linalg.svd(R, full_matrices=False)

    # Construct user and movie representations
    user_vecs = np.dot(U, np.diag(D))
    movie_vecs = Vt.T

    return user_vecs, movie_vecs
```

(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:

```
def get_train_mse(R, user_vecs, movie_vecs):

    # Compute the training MSE loss
    mse_loss = 0
    for i in range(R.shape[0]):
        for j in range(R.shape[1]):
            if not np.isnan(R[i, j]):
                mse_loss += (np.dot(user_vecs[i], movie_vecs[j]) - R[i, j])**2

    return mse_loss
```

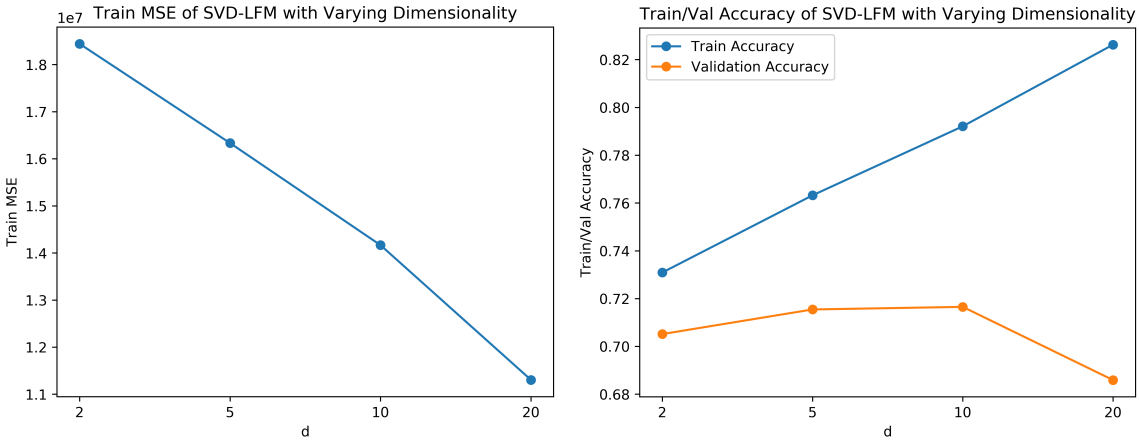
(e) Our model as constructed may achieve 100% training accuracy, but it is prone to overfitting. Instead, we would like to use lower-dimensional representations for x_i and y_j to approximate our known ratings closely while still generalizing well to unknown user/movie combinations. Specifically, we 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:



From the plots, $d = 10$ is optimal since it produces the highest validation accuracy of 71.65%.

(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 ($R_{ij} \neq$ `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: The general strategy for solving this problem.

- **Given:** dataset R with NaNs, randomly initialized vectors $x \in \mathbb{R}^{n\times d}$, $y \in \mathbb{R}^{m\times d}$.
- Consider y to be fixed, then we have for every x_i

$$\begin{aligned}\nabla_{x_i} L(\{x_i\}, \{y_j\}) &= \sum_{(i,j)\in S} 2(x_i \cdot y_j - R_{ij})y_j + 2x_i \\ &= 2\sum_{(i,j)\in S} (y_j y_j^\top x_i - R_{ij}y_j) + 2x_i.\end{aligned}$$

  This gradient is 0 at optimality (sufficient since the objective is convex when y_j is held constant).

$$\begin{aligned}\therefore \sum_{(i,j)\in S} (y_j y_j^\top x_i - R_{ij}y_j) + x_i &= 0. \\ \Big(\sum_{(i,j)\in S} y_j y_j^\top + I_d\Big)x_i &= \sum_{(i,j)\in S} R_{ij}y_j. \qquad \text{(for every } x_i\text{)} \\ \therefore x_i &= \Big(\sum_{(i,j)\in S} y_j y_j^\top + I_d\Big)^{-1} \sum_{(i,j)\in S} R_{ij}y_j.\end{aligned}$$

  Similarly, for each y_j if we set x to be fixed, we get

$$y_j = \Big(\sum_{(i,j)\in S} x_i x_i^\top + I_d\Big)^{-1} \sum_{(i,j)\in S} R_{ij}x_i. \qquad \text{(for every } y_j\text{)}$$

- For iteration $t \in [T]$ we alternate between updating x and y with these update rules.

We implement this algorithm in the code below.

```
best_d = 10
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
    for i in range(R.shape[0]):
        X = np.eye(user_vecs.shape[1])
        Y = np.zeros_like(movie_vecs[0])
        for j in user_rated_idxs[i]:
            X += np.outer(movie_vecs[j], movie_vecs[j])
            Y += R[i, j]*movie_vecs[j]
        user_vecs[i] = np.dot(np.linalg.inv(X), Y)

    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
    for j in range(R.shape[1]):
        X = np.eye(user_vecs.shape[1])
        Y = np.zeros_like(user_vecs[0])
        for i in movie_rated_idxs[j]:
            X += np.outer(user_vecs[i], user_vecs[i])
            Y += R[i, j]*user_vecs[i]
        movie_vecs[j] = np.dot(np.linalg.inv(X), Y)

    return movie_vecs
```

Our best run had a training MSE of 9.3×10^6, training accuracy of 81.63%, and validation accuracy of 72.44%, slightly higher than the best val results in part (e).

4 Nearest Neighbors for Regression, from A to Z

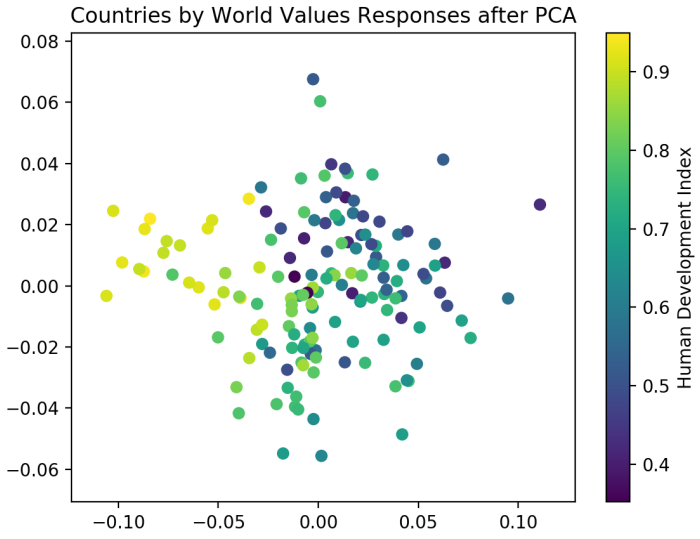
For this problem, we will use data from the UN to have some fun with the nearest neighbors classifier. You'll be modifying starter code in the provided `world_values` directory.

We are using the "World Values Survey" dataset, collected over several years from many countries. The survey asked, "Which of these are most important for you and your family?" There were 16 possible responses, including needs like "Freedom from Discrimination and Persecution" and "Better Transport and Roads." The data reported is the fraction of responses in each country that chose each option.

We would like to use these 16 features of each country (citizens' responses to the survey) to predict that country's HDI (Human Development Index), a value between 0 and 1. In reality, the HDI is a complex measure that accounts for factors like a country's life expectancy, education, and per capita income. Intuitively, though, you would expect citizens of countries with different HDI to have different priorities. For that reason, it may be reasonable to predict the HDI from survey data. (Note: throughout the problem we use RMSE, which stands for Root Mean Squared Error.)

(a) Let's visualize the data. Using sklearn, apply PCA to the data in the plot_pca method of world_values_utils.py. Plot the data in its first two PCA dimensions, colored by HDI.

Solution: Run the plotting code provided to see the graph below:



(b) In lecture, we covered k-nearest neighbors algorithms for classification problems. We decided that the class of a test point would be the plurality of the classes of the k nearest training points. That algorithm makes sense when the outputs are discrete, so we can vote. Here, the outputs are continuous. How would you adapt the k-nearest neighbors classifier for a regression problem? (This is an open-ended question with several possible answers.)

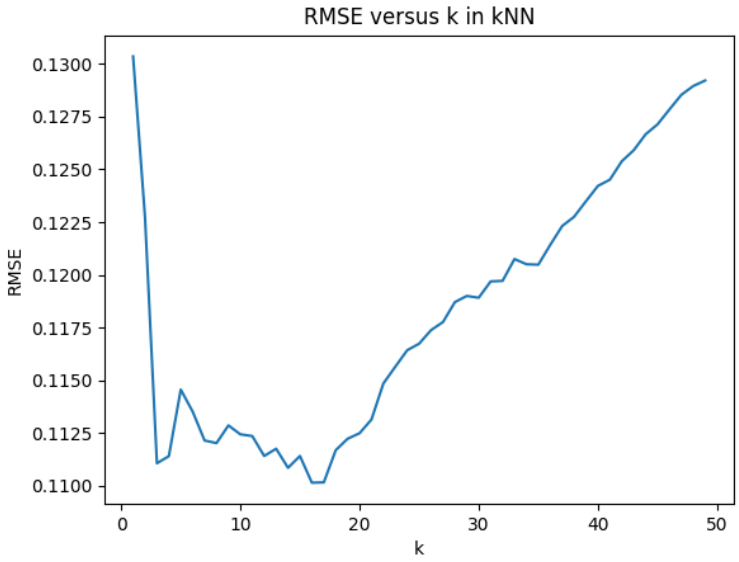
Solution: There are many possible answers. We could take the *average* of the labels of the neighbors or the *weighted average* (weighted by distance) or the *median*. Basically, any measure of central tendency.

(c) Modify the starter code in world_values_starter.py to find the 7 nearest neighbors of the USA. Which countries are the USA's 7 nearest neighbors (in order) from the data given?

Solution: In order, they are Ireland, the United Kingdom, Belgium, Finland, Malta, Austria, and France.

(d) The main hyperparameter of k-nearest neighbors is k itself. Use grid search in world_values_starter.py to create a plot of the RMSE 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 RMSE?

Solution: As k increases, it is initially helpful: averaging more neighbors gives better results. But as k gets too large, we are essentially averaging everything. There is a tradeoff between being too local and being too global. The best value of k is 16, which gives an RMSE of 0.110 (value may differ slightly due to randomness).



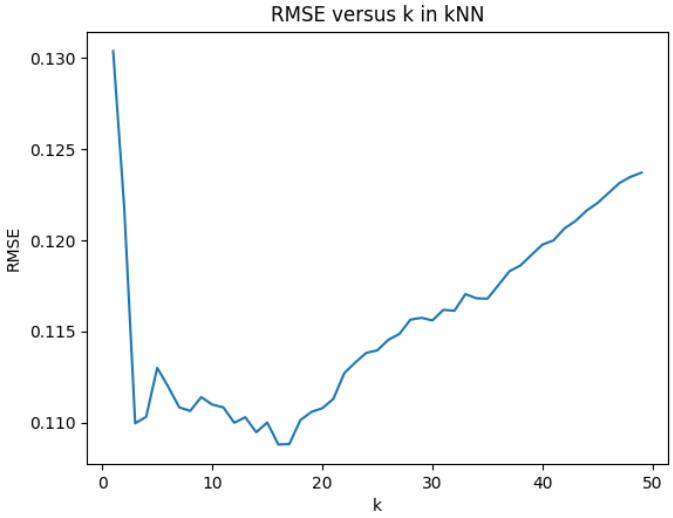
(e) Explain your plot in (d) in terms of bias and variance. Think about the spirit of bias and variance more than their precise definitions.

Solution: As k increases, bias *increases* and variance *decreases*. There is a still a trade-off, of course, but the model complexity shrinks as we add more k. Here are a few ways to understand it:

- When k equals 1, the training error is 0.
- When k equals n, we average all of them so we get a constant estimator, which has variance 0.
- Between 1 and n, k is averaging more points at test time, generating a smoother set of predictions.

(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 to the test point by modifying world_values_parameters.py. Plot the RMSE of k-NN regression with distance weighting vs. k, where k is the number of features. What is the best value of k? What is the RMSE? What happens as k gets very large, compared to part (d)?

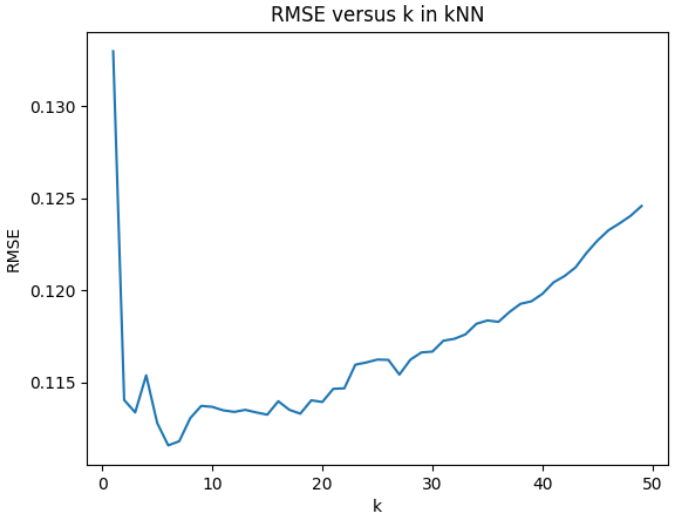
Solution: The graph is similar to the graph from earlier, though now the asymptotic behavior is not as bad. This is because as k gets large we do not return a global average but instead return an average that favors nearer neighbors. The best value of k is still 16, but it gives an RMSE of 0.109 (your value may differ slightly due to randomness).



(g) One of the drawbacks of the k-nearest neighbors classifier is that it is very sensitive to the scale of the features. For example, if one feature takes on values 0 or 0.1 and another takes on values 0 or 10, then neighbors will almost certainly agree in the second feature.

Add normalization to your k-nearest neighbors pipeline (continue to use distance weighting). Plot RMSE versus k. What is the best value of k? What is the RMSE?

Solution:



The new RMSE should be similar to before (ours was ~ 0.112). However, k gets a lot smaller. Now the best k is $k = 6$!