CS 189 Introduction to Machine Learning
Spring 2021 Jonathan Shewchuk HW7

Due: Thursday, May 6 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 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, $R_{ij} =$ `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. $R_{ij} =$ `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: $R = U\Sigma V^T$. Write an expression for R_{ij}, user i's rating for movie j, in terms of only the singular values and the components of U and V.

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

(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 $(n \times m)$, Σ (as a vector of m singular values in descending order, **not** a diagonal matrix), and V^T $(m \times m)$ in that order.

(d) To measure the training performance of the model, we can use the 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`.

(e) Our model as constructed may achieve 100% training accuracy, but is prone to overfitting. Instead, we would like to use lower-dimensional representations for x_i and y_j in order 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. Make sure to label your plots, series, and axes! 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%.

(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 2 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).

2 Regularized and Kernel k-Means

Recall that in k-means clustering we attempt to minimize the objective

$$\min_{C_1,C_2,\ldots,C_k} L = \sum_{i=1}^{k} \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2, \text{ where}$$

$$\mu_i = \text{argmin}_{\mu_i \in \mathbb{R}^d} \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2 = \frac{1}{|C_i|} \sum_{X_j \in C_i} X_j, \quad i = 1, 2, \ldots, k.$$

The sample points are $\{X_1, \ldots, X_n\}$, where $X_j \in \mathbb{R}^d$. C_i is the set of sample points assigned to cluster i and $|C_i|$ is the cluster's cardinality. Each sample point is assigned to exactly one cluster.

(a) What is the minimum value of the objective when $k = n$ (the number of clusters equals the number of sample points)?

(b) (Regularized k-means) Suppose we add a regularization term to the above objective. The regularized k-means objective is now

$$\min_{\mu_i \in \mathbb{R}^d} \sum_{i=1}^{k} \left(\lambda \|\mu_i\|_2^2 + \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2 \right).$$

Show that the optimum of

$$\min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2$$

is obtained at

$$\mu_i = \frac{1}{|C_i| + \lambda} \sum_{X_j \in C_i} X_j.$$

(c) (Kernel k-means) Suppose we have a dataset $\{X_i\}_{i=1}^n, X_i \in \mathbb{R}^\ell$ that we want to split into k clusters, i.e., finding the best k-means clustering (without regularization). Furthermore, suppose we know *a priori* that this data is best clustered in an impractically high-dimensional feature space $\mathbb{R}^m$ with an appropriate metric. Fortunately, instead of having to deal with the (implicit) feature map $\Phi : \mathbb{R}^\ell \to \mathbb{R}^m$ and (implicit) distance metric[1], we have a kernel function $\kappa(X_1, X_2) = \Phi(X_1) \cdot \Phi(X_2)$ that we can compute quickly on the raw samples. How should we perform the kernelized counterpart of k-means clustering?

Derive the missing portion of this algorithm, and show your work in deriving it. The primary issue to consider is that although we define the means μ_i in the usual way, we can't ever compute Φ explicitly because it's way too big. Therefore, in the step where we determine which cluster each sample point is assigned to, we must use the kernel function κ to obtain the right result. Review the lecture on kernels

[1] Just as how the interpretation of kernels in kernelized ridge regression involves an implicit prior/regularizer as well as an implicit feature space, we can think of kernels as generally inducing an implicit distance metric as well. Think of how you would represent the squared distance between two points in terms of pairwise inner products and operations on them.

if you don't remember how that's done.

Algorithm 1: Kernel k-means

Require: Data matrix $X \in \mathbb{R}^{n \times d}$; number of clusters K; kernel function $\kappa(X_1, X_2)$
Ensure: Cluster class(j) assigned for each sample point X_j.
function KERNEL-K-MEANS(X, K)
  Randomly initialize class(j) to be an integer in $1, 2, \ldots, K$ for each X_j.
  while *not converged* **do**
    for $i \leftarrow 1$ ***to*** K **do**
      Set $S_i \leftarrow \{j \in \{1, 2, \ldots, n\} : \text{class}(j) = i\}$.
    for $j \leftarrow 1$ ***to*** n **do**
      Set class(j) $\leftarrow \arg\min_k$(Action item: derive the missing portion here)
  Return S_i for $i \leftarrow 1, 2, \ldots, K$.
end function

(d) The expression you derived may have unnecessary terms or redundant kernel computations, especially when you compute class(j) for every $j \in [1, n]$. Explain how to eliminate them; that is, how to perform the computation quickly without doing irrelevant computations or redoing computations already done.

3 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 meta-learner 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.

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

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

(c) Let B (for "bad") be the number of sample points out of n that the meta-learner 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.

(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)?

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