CS434H Machine Learning and Data Mining – Homework 1

k-Nearest Neighbor Classification

Overview and Objectives. In this homework, we are going to implement a kNN classifier. The assignment will help you understand how to apply the concepts from lecture to applications.

How to Do This Assignment.

- Each question that you need to respond to is clearly marked in a blue "Task Box" with its corresponding point-value listed. Most questions involve programming. There is a written component to Q3 and Q6 which you need to include in a report. Don't forget to submit this report in addition to the code.
- We prefer a typeset report (LaTeX / Word) but will accept scanned written work if it is legible. If we can't read your work, we can't give you credit. Submit your solutions to Canvas as a zip including your report and any code the assignment asks you to develop. You do not need to include the data files.
- Programming should be done in Python and numpy. If you don't have Python installed in your machine, you can install it from https://www.python.org/downloads/. This is also the link showing how to install numpy: https://numpy.org/install/. You can also search through the internet for numpy tutorials if you haven't used it before. Google is your friend!

You are NOT allowed to...

- Use machine learning package such as `sklearn`.
- Use data analysis package such as `panda` or `seaborn`.
- Discuss low-level details or share code / solutions with other students.

Advice. Start early. Start early. Start early. This is a programming assignment. Give yourself time to debug!

How to submit. Submit a zip file to Canvas. Inside, you will need to have all your working code and `hw1-report.pdf`.

1 The Code Structure

This assignment consists of four Python files:

- hw1.py: This file contains the main function and the code that drives the algorithm. You will see that I have provided some functions for you such as some testing functions and a function for computing accuracy.
- nn.py: This file contains the NearestNeighbor class. You will need to fill out some functions in this class.
- rplsh.py: This file contains the RPLSH (Random Projection Locality Sensitive Hashing) class. You will need to implement this entire class.
- rplsh_nn.py: This file contains the RPLSHNearestNeighbor class, which is a subclass of the NearestNeighbor class. If you set things up correctly, you will end up writing a small amount of code in this class.

Please follow the structure of this code base. For the functions I specifically ask you to complete, please do not change the arguments that they take. I will be relying on them being the same so that I can run scripts to help with grading. You may add new instance functions and instance variables as needed.

To run the code, the command line arguments are:

hw1 <training set file name> <test set file name> <mode>

where mode = 0 means "normal" kNN and mode = 1 means the LSH version of kNN. **Please do not change this.**

2 k-Nearest Neighbor (kNN) [50pts]

In this section, we'll implement our first machine learning algorithm of the course – k Nearest Neighbors. We are considering a binary classification problem where the goal is to classify whether a person has an annual income more or less than $50,000 given census information. As no validation split is provided, you'll need to perform cross-validation to fit good hyperparameters. I will be evaluating your code on a private test set that I have not released to the class.

Data Analysis. We've done the data processing for you for this assignment; however, getting familiar with the data before applying any algorithm is a *very* important part of applying machine learning in practice. Quirks of the dataset could significantly impact your model's performance or how you interpret the outcome of your experiments. For instance, if I have an algorithm that achieved 70% accuracy on this task – how good or bad is that? We'll see!

We've split the data into two subsets – a **training** set with 8,000 labelled rows, and a **test** set with 2,000 unlabelled rows. These can be downloaded from Canvas as "train.csv" and "test_pub.csv" respectively. Both files will come with a header row, so you can see which column belongs to which feature.

Below you will find a table listing the attributes available in the dataset. We note that categorizing some of these attributes into two or a few categories is reductive (e.g. only 14 occupations) or might reinforce a particular set of social norms (e.g. categorizing sex or race in particular ways). For this homework, we reproduced this dataset from its source without modifying these attributes; however, it is useful to consider these issues as machine learning practitioners.

attribute name: type. *list of values*
id: numerical. Unique for each point. Don't use this as a feature (it will hurt, badly).
age: numerical. ***workclass***: categorical. *Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay* ***education-num***: ordinal. *1:Preschool, 2:1st-4th, 3:5th-6th, 4:7th-8th, 5:9th, 6:10th, 7:11th, 8:12th, 9:HS-grad, 10:Some-college, 11:Assoc-voc, 12:Assoc-acdm, 13:Bachelors, 14:Masters, 15:Prof-school, 16:Doctorate* ***marital-status***: categorical. *Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse* ***occupation***: categorical. *Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces* ***relationship***: categorical. *Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried* ***race***: categorical. *White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black* ***sex***: categorical. *0:Male, 1:Female* ***capital-gain***: numerical. ***capital-loss***: numerical. ***hours-per-week***: numerical. ***native-country***: categorical. *United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinada&Tobago, Peru, Hong, Holand-Netherlands*
income: ordinal. 0: <=50K, 1: >50K This is the class label. Don't use this as a feature.

Our dataset has three types of attributes – numerical, ordinal, and nominal. Numerical attributes represent continuous numbers (e.g. hours-per-week worked). Ordinal attributes are a discrete set *with a natural ordering*, for instance different levels of education. Nominal attributes are also discrete sets of possible values; however, there is no clear ordering between them (e.g. native-country). These different attribute types require different preprocessing. As discussed in class, numerical fields have been normalized.

For nominal variables like workclass, marital-status, occupation, relationship, race, and native-country, we've transformed these into one column for each possible value with either a 0 or a 1. For example, the first instance in the training set reads: `[0, 0.136, 0.533, 0.0, 0.659, 0.397, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]` where all the zeros and ones correspond to these binarized variables.

2.1 k-Nearest Neighbors

In this part, you will need to implement the k-NN classifier algorithm with the Euclidean distance. The kNN algorithm is simple as you already have the preprocessed data. To make a prediction for a new data point, there are three main steps:

1. Calculate the distance of that data point to every training example.
2. Get the k nearest points (i.e. the k with the lowest distance).
3. Return the majority class of these neighbors as the prediction

Implementing this using native Python operations will be quite slow, but lucky for us there is the `numpy` package. `numpy` makes matrix operations quite efficient and easy to code. How will matrix operations help us? The next question walk you through figuring out the answer. Some useful things to check out in `numpy`: broadcasting, `np.linalg.norm` `np.argsort()` or `np.argpartition()`, and array slicing.

Distances and vector norms are closely related concepts. For instance, an L_2 norm of a vector $\mathbf{x}$ (defined below) can be interpreted as the Euclidean distance between $\mathbf{x}$ and the zero vector:

$$||x||_2 = \sqrt{\sum_{i=1}^{d} x_i^2} \tag{1}$$

The Euclidean distance between $\mathbf{x}$ and $\mathbf{z}$ can be written as an L_2 norm (convince yourself of this). You can compute norms efficiently with `numpy` using `np.linalg.norm`

In kNN, we need to compute distance between every training example $\mathbf{x}_i$ and a new point $\mathbf{z}$ in order to make a prediction. Computing this for one $\mathbf{x}_i$ can be done by applying an arithmetic operation between $\mathbf{x}_i$ and $\mathbf{z}$, then taking a norm. In `numpy`, arithmetic operations between matrices and vectors are sometimes defined by "broadcasting", even if standard linear algebra doesn't allow for them. For example, given a matrix X of size $n \times d$ and a vector $\mathbf{z}$ of size d, `numpy` will happily compute $Y = X - z$ such that Y is the result of the vector z being subtracted from each row of X. **Combining this with your answer from the previous question can make computing distances to every training point quite efficient and easy to code.**

▶ Q1 Implement kNN Classifier [10pts] Implement k-Nearest Neighbors using Euclidean distance by completing the skeleton code provided in `nn.py`. Specifically, you'll need to finish:

- `def __init__(self, train_X, train_Y)`
  This is the constructor for the NearestNeighbor class. Note that you have to pass in the training data here so you will need to store it in an instance variable. Here train_X is an n-by-d 2D numpy array, corresponding to the training data features. Each row contains the n features of a single d-dimensional data point. The argument train_Y is a n-by-1 numpy array of the class labels for each data instance in train_X.
- `def get_nearest_neighbors(self,query,k)`
  Using the training dataset you stored in the constructor, return the indices of the k nearest examples to the query point. This is where the bulk of the computation will happen in your algorithm so you are strongly encouraged to review the paragraph above this task box to get hints for how to do it efficiently in numpy.
- `def classify(self, query, k)`
  Using the training dataset you stored in the constructor, return the prediction of a kNN classifier for the query point. Should use the previous function.
- `def classify_dataset(self, queries_X, k)`
  Same as the function above but now applied to an m-by-d numpy array of queries

The code to load the data is in the sanity_check() and run_cross_validation_test() functions. I've also included code to compute accuracy in the nn.py file. You'll want to read over these carefully.

2.2 K-Fold Cross Validation

We do not provide class labels for the test set or a validation set, so you'll need to implement K-fold Cross Validation[1] to check if your implementation is correct and to select hyperpameters. As discussed in class, K-fold Cross Validation divides the training set into K segments and then trains K models – leaving out one of the segments each time and training on the others. Then each trained model is evaluated on the left-out fold to estimate performance. Overall performance is estimated as the mean and variance of these K fold performances.

▶ Q2 Implement k-fold Cross Validation [8pts] Next we'll be implementing k-fold cross validation by finishing the following function in `nn.py`:

- `cross_validation(train_X, train_y, num_folds, k)`
  Given a $n \times d$ matrix of training examples and a $n \times 1$ column-vector of their corresponding labels, perform K-fold cross validation with `num_folds` folds for a k-NN classifier. To do so, split the data in `num_folds` equally sized chunks then evaluate performance for each chunk while considering the other chunks as the training set. Return the average and variance of the accuracies you observe.

For simplicity, you can assume `num_folds` evenly divides the number of training examples. You may find the numpy split and vstack functions useful.

▶ Q3 Hyperparameter Search [8pts] To search for the best hyperpameters, run 4-fold cross-validation to estimate our accuracy. For each k in 1,3,5,7,9,99,999,and 8000, report:

- accuracy on the training set when using the entire training set for kNN (call this **training accuracy**),
- the mean and variance of the 4-fold cross validation accuracies (call this **validation accuracy**).

Skeleton code for this is present in `hw1.py` and labeled as Q3 Hyperparmeter Search. Finish this code to generate these values – should likely make use of `accuracy`, and `cross_validation`.

Questions: What is the best number of neighbors (k) you observe? When k = 1, is training error 0%? Why or why not? What trends (train and cross-valdiation accuracy rate) do you observe with increasing k? How do they relate to underfitting and overfitting?

2.3 Random Projection Locality Sensitive Hashing

The next step is to implement Random Projection Locality Sensitive Hashing (or RPLSH for short), just like we described in class. Remember the intuition behind the hashing – data points that are nearby will have the same hash code and thus map to the same hash bucket.

▶ Q4 Random Projection Locality Sensitive Hashing [8pts] Implement a RPLSH by completing the skeleton code provided in `rplsh.py`. Specifically, you'll need to finish:

- `def get_hash_code(self,x)`
  Creates the hash code for the data instance x, which is a 1-by-d numpy array. Here, d is the number of features. To generate random hyperplanes, sample each hyperplane coefficient from a Gaussian with mean 0 and variance 1. You will need to store these hyperplane coefficients in an instance variable of some sort.
- `def hash_dataset(self,dataset)`
  Takes a dataset (a n-by-d numpy array) and inserts all the data instances into its internal hash table. The hash code is computed using the random hyperplanes.
- `def get_hash_entries(self,x)`
  Returns a numpy array of indices corresponding to data points in the training data that are approximately close to x. More precisely, these indices are the data points that map to the same hash bucket as x does.

2.4 k-Nearest Neighbor with RPLSH

[1] Note that K here has nothing to do with k in kNN – just overloaded notation.

Once you have finished your RPLSH class, the next task is to implement a RPLSH-based k-Nearest Neighbor classifier.

▶ Q5 RPLSH-based k-Nearest Neighbor Classifier [10pts] Implement a RPLSH-based k-Nearest Neighbor classifier by completing the skeleton code provided in `rplsh_knn.py`. Note that the RPLSHNearestNeighbor class is a subclass of the NearestNeighbor classifier (take advantage of inheritance!). You will need to implement the following functions:

- `def __init__(self, train_X, train_Y, num_projections, num_hash_tables)`
  This is the constructor for the RPLSHNearestNeighbor class. Note that in addition to the parameters that the NearestNeighbor constructor takes, this constructor takes the number of projections (i.e. the number of random hyperplanes) and the number of hash tables (which we refer to as L in our notes). Don't forget to call super() to call the base class constructor.
- `def get_nearest_neighbors(self,query,k)`
  This function gets the k nearest neighbors as in the base class but it should use the RPLSH to help find nearest neighbors.

What should you do if your hash bucket has 0 items? In this case, have your classifier return a randomly chosen class label or if you want to be smarter, have it output the most common class label.

▶ Q6 Cross validation to select M (the number of hyperplanes) and L (the number of hash tables [6 pts]) You now need to implement a second hyperparameter search (like in Q3) for M and L. To make things easier, select the best k you determined from Q3 and hold it fixed. You will try out a small range of values for$M$=2,4,6,8 and L=1,2,3.
Questions: What are the best configurations of M, L and k that you found? What are the training and test accuracies with this best configuration? Are there any trends in accuracy that you noticed for M and L?

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?