Mahdokht Afravi
Sept 6, 2019
CS5461 ML

k-nearest-neighbors

Problem

The goal of this program is to build a model that implements the k-nearest-neighbors algorithm. This model will either be a classifier or regression model, depending on user specification, that will receive training data on which to train the model, and testing data which will predict the class or likely class for each test sample provided to the predictor.

Development Environment

This program was developed on two platforms with the Python 3.7 library, though the program was mostly tested using the CuPy instead of NumPy package. This development environment consisted of the PyCharm Ultimate IDE on a CUDA-enabled GPU (GeForce GTX 1050). The other environment did not have a dedicated graphics card, which seriously hindered the testing time and allowed only a limited number of test cases, which are not presented in this paper.

Algorithms implemented

The algorithm used for the model's predictor function is the k-nearest-neighbors. The implementation was designed for the following 4 user choices:

1. Weighted classifier
2. Un-weighted classifier
3. Weighted regression
4. Un-weighted regression

Therefore, the predictor first checks whether a regression or classification model will be used, and then checks if the selected model will calculate results using weights or not. The implemented algorithm for the KNN classifier model is as follows:

$\text{classifier(test\_data): guess}$

1. Create a list for the guesses for each of the test samples
2. Create a list for the labels for each of the possible classifications
3. For every test sample x
    a. Create a list for the votes of the label for each neighbor
    b. For every possible classification l
        i. Find the delta sum of the neighbors whose whose labels are equal to l
        ii. If we are using weights, multiple this value by the sum of the weights of the neighbors
    c. Find the largest of these votes to be the guess for this test sample

The implemented algorithm for the KNN regression model is as follows:

regression(test_data): guess

1. Create a list of the means of neighbors for each test sample
2. For every test sample x
   a. Save the sum of the weights multiplied by labels of the k nearest neighbors
   b. If we are using weights, divide this sum by the sum of the k nearest neighbors' weights
   c. If we are not using weights, divide this sum by k

Experimental results

As expected, the running times for regression . Two datasets were used to train and test the model: the MNIST dataset and the Solar Particle dataset. Training results' times are not reported as the fit() function only stores the training data for the predict() function to later use. The results for MNIST dataset are presented in Figures 1 and 2. The tests presented from these results use the classification and regression algorithms, with both weighted and unweighted models. All results presented included 5000 training samples and up to 500 test cases.

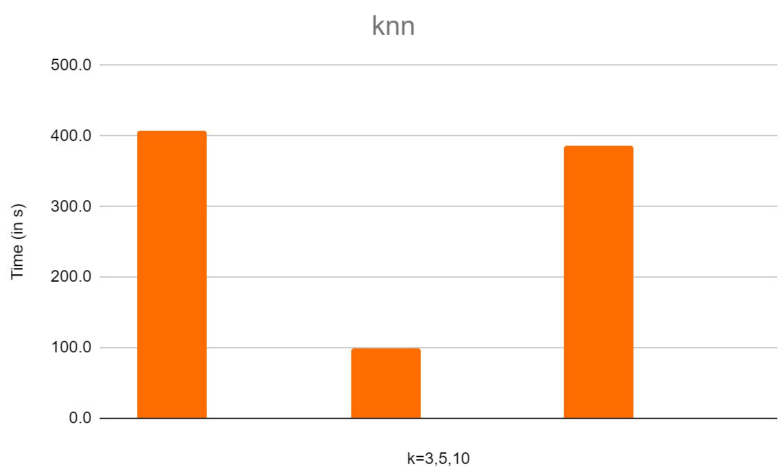

Fig 1: Average times for MNIST tests when k = 3, 5, 10

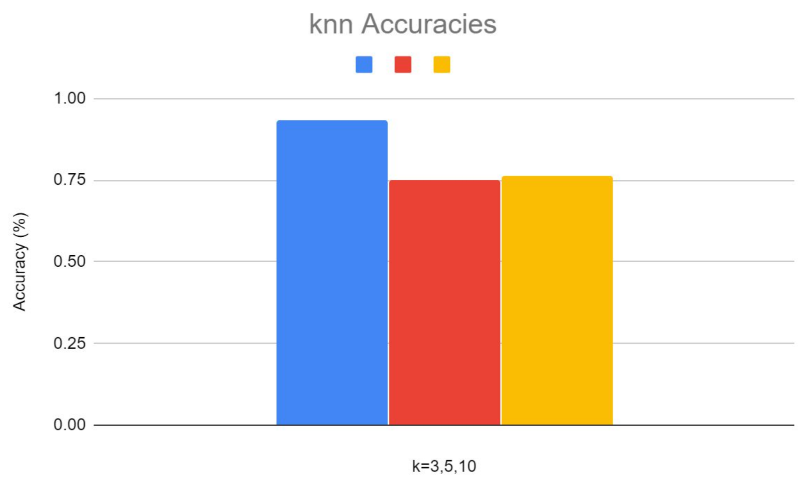

Fig 2: Average accuracies for MNIST tests when k = 3, 5, 10

Below, in Figures 3 and 4, I present the results for the Solar Particle dataset using k values of 3 and 5. Again, the tests presented from these results use the classification and regression algorithms, with both weighted and unweighted models. All results presented included 5000 training samples and up to 500 test cases.

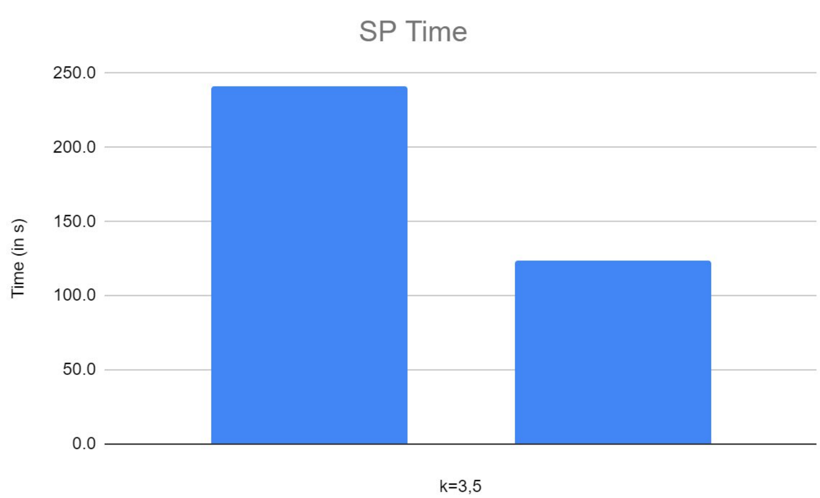

Fig 3: Average times for Solar Particle tests when k = 3, 5

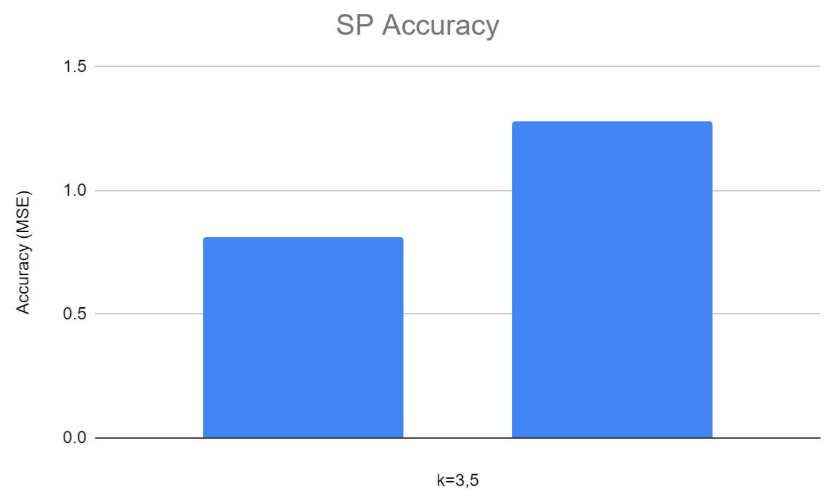

Fig 4: Average accuracies for Solar Particle tests when k = 3, 5

Optimization

In order to improve the efficiency of the program, I implemented an algorithm for attribute selection. The algorithm is as follows:

attr_sel(train): void

1. Find the means of each of the features across all training samples
2. Remove those features whose means are less than 0.25 of the average of features

On average, the number of features decreased by 390-400 for MNIST and decreased by 40 in the Solar Particles dataset. The time decreased but not significantly. I expected a decrease in time if the number of features decreased significantly, as reflected in the MNIST dataset, and thankfully the optimization does improve when preprocessing the data.

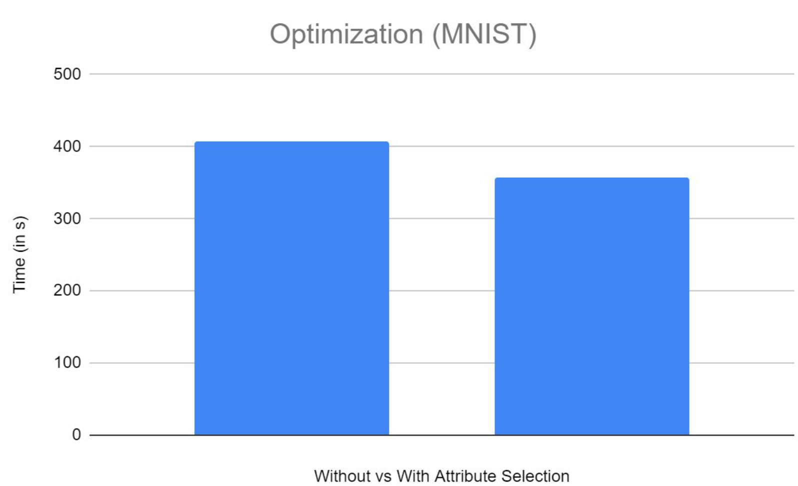

Figure 5: MNIST results with attribute selection implemented

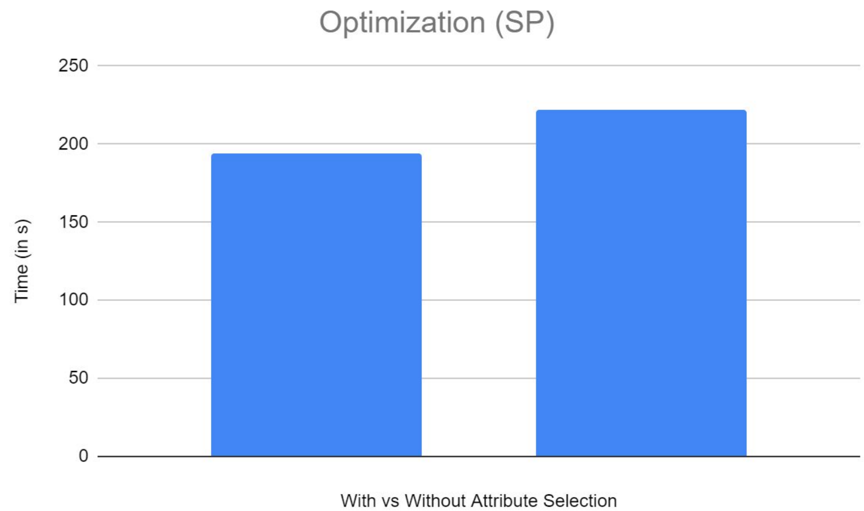

Figure 6: SP results with attribute selection implemented

For the MNIST dataset, on average, the number of features removed was 395 (out of 784). On average, the number of features removed from the Solar Particle dataset was 40 (out of 50). Since the use of attribute selection did not affect accuracy significantly, give or take 3%, it would be beneficial to implement a better algorithm as well as other optimization approaches in addition to the one used in this program.

Discussion of results

There are some limitations to my program. Firstly, my implementation may not be taking full advantage of the features of Python array slicing, and this may explain why my results do not reflect a significant decrease in time despite my implementation of optimization.

Another concern lies in my optimization algorithm, which used a fixed value for the threshold for feature variance. For the purposes of this assignment, I kept the features whose means were less than 0.25 of the average of all of the features. This, however, is dependent on the training data's features, meaning, which features are dropped will vary greatly. For smaller test cases, this did not affect accuracy but did improve efficiency. For larger test cases and varying values of k, accuracy did decrease and the time efficiency improved.

Appendix

The appendix includes the source code of the described program implemented in Python 3.7. Methods/Files authored by me are denoted with an author tag (@author mmafr). The code is also available at https://github.com/mahdafr/19w_cs5361-labs on GitHub.

```python
import mnist
import cupy as np
import time
import math


class knn(object):
    # Constructor
    def __init__(self, k=3, weighted=True, classify=True):
        self.k = k
        self.weighted = weighted
        self.classify = classify
        # @author mmafr
        self.n_matrix = None    # the list of the distances
        self.n_weights = None
        self.n_labels = None

    # train the model
    def fit(self, x, y):
        self.x_train = x.astype(np.float32)
        self.y_train = y

    # test the model
    # @author mmafr
    def predict(self, test):
        s = ' (weighted)' if self.weighted else ''
        print('Classification' + s if self.classify else 'Regression' + s)
        # test = self.attr_selection(test)  # for optimization
        self.buildNeighborList(test)
        if self.classify:
            return self.classifier(test)
        else:
            return self.regression(test)

    # KNN classifier: weighted or unweighted?
    # @author mmafr
    def classifier(self, test):
        guess = np.zeros(test.shape[0])
        label = np.unique(self.y_train)     # every possible class
        for x in range(test.shape[0]):
            votes = np.zeros(label.shape[0])  # sums up to k
            for l in range(label.shape[0]):     # for each label
                d = self.delta(l, self.n_labels[x])
                votes[l] = d*np.sum(self.n_weights[x])
                votes = np.around(votes, 3)
            guess[x] = np.argmax(votes)
            # print("Prediction for Test=" + str(x) + " is " + str(guess[x]))
        return guess

    # @author mmafr
    def delta(self, a, b):
        return (b==a).sum()

    # KNN regression: weighted or unweighted?
    # @author mmafr
    def regression(self, test):
        mean = np.zeros(test.shape[0])
        for x in range(test.shape[0]):  # for each test example
            mean[x] = np.sum(self.n_weights[x]*self.n_labels[x])
            if self.weighted:   # divide by sum of the weights
                mean[x] = mean[x]/np.sum(self.n_weights[x])
            else:   # divide by k
                mean[x] = mean[x]/self.k
            # print("Prediction for Test=" + str(x) + " is " + str(mean[x]))
        mean = np.around(mean, 0)   # round for picking a class
        return mean
```

```python

    # calculate the distances from each test point to other points
    # @author mmafr
    def buildNeighborList(self, test):
        n_tmp = np.zeros(shape=(test.shape[0], self.x_train.shape[0]))
        for x1 in range(test.shape[0]):     # foreach test sample
            for x2 in range(self.x_train.shape[0]):     # for each train point
                n_tmp[x1][x2] = (self.distance(test[x1], self.x_train[x2]))
        srtd = np.argsort(n_tmp, axis=1)    # sort list of neighbors
        # build the knn matrix and weights
        self.k_neighbors(n_tmp, srtd)
        self.weights()
        print("Built neighbor matrix.")

    # calculate the euclidean distance between point x1 and x2
    # @author mmafr
    def distance(self, x1, x2):
        d = 0
        for f in range(len(x1)):
            d += pow((x1[f] - x2[f]), 2)
        return math.sqrt(d)

    # determines the list of the k nearest-neighbors
    # @author mmafr
    def k_neighbors(self, neighbors, nearest):
        # each test sample has k nearest neighbors
        self.n_matrix = np.zeros(shape=(neighbors.shape[0], self.k))
        self.n_labels = np.zeros(self.n_matrix.shape)
        # self.attr_selection()   # preprocessing: optimize time
        for i in range(self.n_matrix.shape[0]):
            for j in range(self.k):
                self.n_matrix[i][j] = neighbors[i][nearest[i][j]]
                self.n_labels[i][j] = self.y_train[int(nearest[i][j])]

    # drop the features with the lowest variance
    # @author mmafr
    def attr_selection(self, test):
        start = self.x_train.shape[1]
        mean = np.mean(self.x_train, axis=0)   # means of features
        mask = (self.x_train > 0.25*np.mean(mean))
        self.x_train = np.asarray(self.x_train[:, mask.any(axis=0)])
        x_test = test[:, mask.any(axis=0)]
        print("Dropped " + str(start - self.x_train.shape[1]) + " features.")
        return x_test

    # get the weights, list of ones if not weighted
    #  @author mmafr
    def weights(self):
        if self.classify:
            self.n_weights = np.zeros(self.n_matrix.shape)
            self.n_weights[: , 0] = 1   # all others are 0
        else:
            self.n_weights = np.ones(self.n_matrix.shape)
        if self.weighted:  # if weighted, w=1/d(x,xi)
            self.n_weights = 1/self.n_matrix
        # mask = np.isin(self.sorted, self.neighbor)
        # weight = 1/weight[mask]


if __name__ == "__main__":
    TESTS = 10
    TRAIN = 1000
    THEK = 3
    print('MNIST dataset')
    x_train, y_train, x_test, y_test = mnist.load()
    x_train = x_train[:TRAIN:]
```

```
    y_train = y_train[:TRAIN:]
    x_test = x_test[:TESTS]
    y_test = y_test[:TESTS]
    print('Training size = ' + str(x_train.shape[0]))
    print('Testing size = ' + str(x_test.shape[0]))
    print("K = " + str(THEK))
    model = knn(classify=True, weighted=False)

    start = time.time()
    model.fit(x_train, y_train)
    elapsed_time = time.time()-start
    print('Elapsed_time training  {0:.6f} '.format(elapsed_time))

    start = time.time()
    pred = model.predict(x_test)
    elapsed_time = time.time()-start
    print('Elapsed_time testing  {0:.6f} '.format(elapsed_time))

    y_test = np.asarray(y_test)     # for CuPy
    print('Accuracy:', np.sum(pred == y_test)/len(y_test))

    # print('\nSolar particle dataset')
    # dir = 'D:\Google Drive\skool\CS 5361\datasets\lab1\\'
    # x_train = np.load(dir + 'x_ray_data_train.npy')
    # y_train = np.load(dir + 'x_ray_target_train.npy')
    # x_test = np.load(dir + 'x_ray_data_test.npy')
    # y_test = np.load(dir + 'x_ray_target_test.npy')
    # y_test = np.asarray(y_test)  # for CuPy
    # x_train = x_train[:TRAIN]
    # y_train = y_train[:TRAIN]
    # x_test = x_test[:TESTS:]
    # y_test = y_test[:TESTS:]
    # print('Training size = ' + str(x_train.shape[0]))
    # print('Testing size = ' + str(x_test.shape[0]))
    # print("K = " + str(THEK))
    # model = knn(k=THEK, classify=True, weighted=True)
    #
    # start = time.time()
    # model.fit(x_train, y_train)
    # elapsed_time = time.time()-start
    # print('Elapsed_time training  {0:.6f} '.format(elapsed_time))
    #
    # start = time.time()
    # pred = model.predict(x_test)
    # elapsed_time = time.time()-start
    # print('Elapsed_time testing  {0:.6f} '.format(elapsed_time))
    #
    # print('Mean square error:', np.mean(np.square(pred-y_test)))
```