EECS-553 Homework 1

Vipransh Sinha

February 5, 2025

1. Curse of Dimensionality

Consider a synthetic dataset where training and test data are generated with n samples in d dimension according to X_{train} = np.random.randn(d)/np.sqrt(d). Here $1/np.sqrt(d)$ normalization ensures that the points approximately lie on the unit Euclidean ball. Set $k = 1$ i.e. we are investigating vanilla nearest neighbor classifier. Let x be a test example similarly drawn from np.random.randn(d). Let $d(x)$ be the distance to the nearest neighbor i.e.

$$d(x) = \min_{x' \in X_{train}} \|x - x'\|_2.$$

We will plot the expected distance $\mathbb{E}[d(x)]$ as a function of d and n. Concretely, we vary $d \in \{2, 4, 6, 8, 10\}$ and $n = \{100, 200, 500, 1000, 2000, 5000\}$. In order to estimate the expectation, average over sufficiently many test points x (e.g 100 realizations).

- Plot $\mathbb{E}[d(x)]$ as a function of n and create a separate curve for each choice of d. Create the plot in semilogx for better visualization
- Comment on the influence of d on the distance to the nearest neighbor
- Suppose we wish to ensure $\mathbb{E}[d(x)] \leq \epsilon$ for some $\epsilon > 0$. Based on these experiments, how do you expect n should grow as a function of d and ϵ (your best guess e.g exponential vs polynomial dependence)?

Solution
Plotting $\mathbb{E}[d(x)]$ as a function of n, we get the following graph below (code found in index). As dimension (d) increases, the expected distance ($\mathbb{E}[d(x)]$) to the closest nearest neighbor increases. Since we are on a semilog graph, the rate at which $\mathbb{E}[d(x)]$ changes as n increases is distorted. To ensure that $\mathbb{E}[d(x)] \leq \epsilon$, n should grow at: $n \propto e^{c\epsilon d}$, where c is arbitrary.

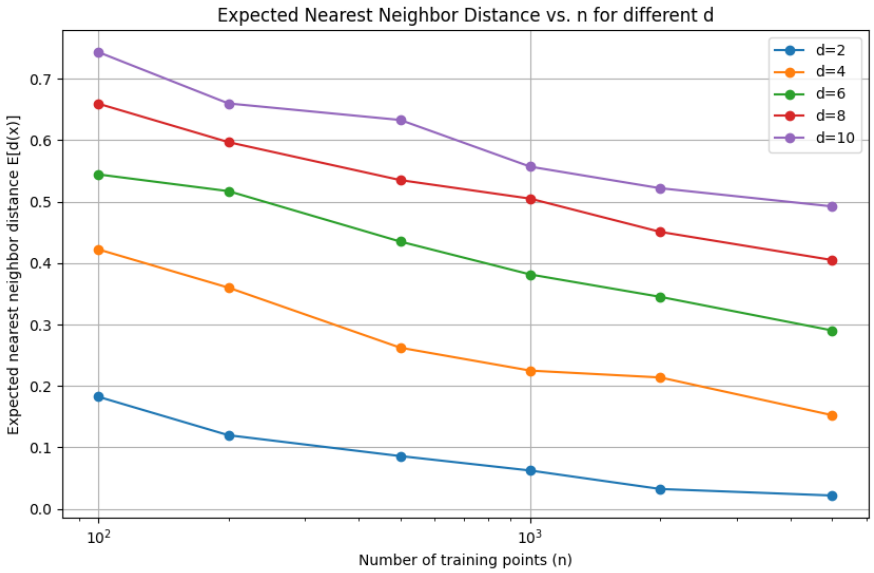

Figure 1: Exercise 1 Graph

2. Fast kNN implementations

Consider a synthetic dataset where training and test data are generated with n samples in $d = 2$ dimension according to Xtrain = np.random.randn(n,2) , ytrain = sign(np.random.rand(n)) (same for test). We will evaluate the computational performance of different kNN implementations on this dataset. In sklearn.KNeighborsClassifier, you can set algorithm to one of algo = ('balltree', 'kdtree', 'brute'). Set $k = 5$ neighbors i.e set clf = neighbors.KNeighborsClassifier(5, algorithm=algo). Use $n_{test} = 5000$ test examples for your evaluations.

(a) Verify that 'brute' is the fastest algorithm for training i.e for running cld.fit(Xtrain, ytrain). Try $n = 1000, n = 10000, n = 100000$ and report the time using time.time().

(b) Verify that 'brute' becomes slower at inference for sufficiently large n. By inference, we mean the time it takes to run clf.predict(Xtest). Specifically, evaluate on the grid $n \in [1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000]$ and plot the inference time of the three kNN methods as a function of n. For which n choice, 'brute' becomes the slowest option for the first time? The precise n might be hardware dependent and it does not have to be exact as long as you observe a trasition in your plot. You can again use time.time() for time measurements. You might have to try larger n choice (like $n = 10^6$ or more) if needed.

Solution

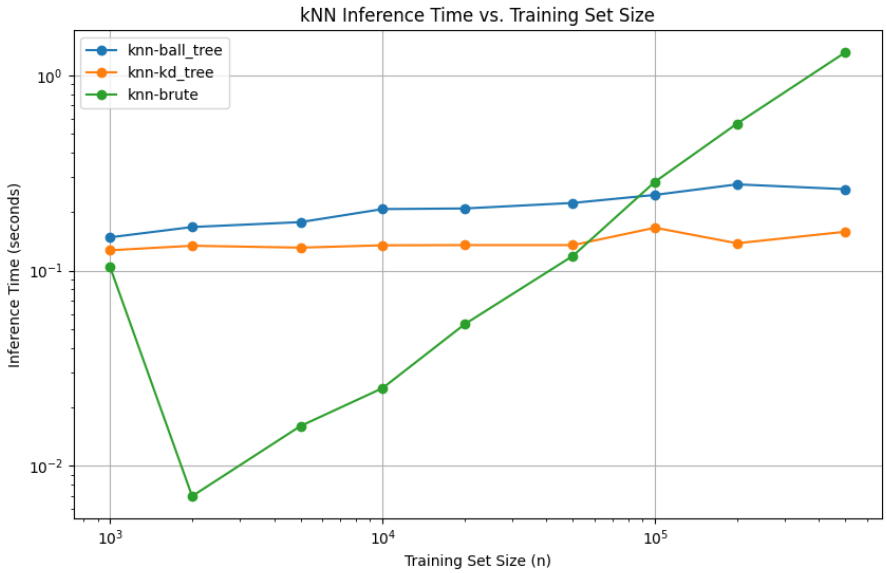

Figure 2: Graph for Exercise 2b

a) After running each of the kNN implementations at different dataset sizes, I am finding that brute is by far the fastest for dataset sizes of [1000, 5000, 10000, 100000]. Rounded to 10 decimal places, at dataset sizes of 1,000 and 5,000 we get a runtime of 0 seconds. At a dataset size of 10,000 we get a runtime of 0.001 seconds (which is 4 times faster than kdtree and 5 times faster than balltree. At a dataset size of 100,000 we get a runtime of 0.009 seconds (which is $\sim$ 7.5 times faster than kdtree, and $\sim$ 6 times faster than balltree). The code and detailed runtimes found in Index.

b) The above graph shows that at around $n \approx 10^5$ training set size, brute starts to become the slowest kNN algorithm.

3. PSD matrices

(a) Prove that if all eigenvalues of a smmetric matrix A are positive, then $x^T Ax > 0$ for all $x \neq 0$ (and hence A is positive definite).

(b) A Gram matrix is any $d \times d$ matrix whose $(i, j)^{th}$ entry is $\langle x_i, x_i \rangle$ for some vectors $x_1, \ldots, x_d$. Show that Gram matrices are positive semi-definite.

Solution

a) Since A is symmetric and has all positive eigenvalues, we can rewrite $A = PDP^T$, where

$D = \text{diag}(\lambda_1, \ldots, \lambda_n)$, and P is an orthogonal matrix. Thus we can rewrite

$$x^T Ax = x^T(PDP^T)x$$

Letting arbitrary $y = P^T x$, we can rewrite again as

$$\begin{aligned} &= y^T Dy \\ &= \lambda_1 y_1^2 + \ldots + \lambda_n y_n^2 \end{aligned}$$

Since all y terms are squared, and all λ values are positive by the problem statement, we have shown that $x^T Ax > 0$.

b) Let G be the Gram matrix, and $G_{i,j} = \langle x_i, x_i \rangle$. Let u be an arbitrary vector. Let's apply the condition of positive semi-definiteness and expand:

$$\begin{aligned} u^T Gu &= \sum_{i,j} u_i G_{i,j} u_j \\ &= \sum_{i,j} \langle x_i, x_j \rangle u_j \end{aligned}$$

using properties of the inner product, we get

$$\begin{aligned} &= \sum_j \langle \sum_i u_i x_i, x_j \rangle u_j \\ &= \langle \sum_i u_i x_i, \sum_j u_j x_j \rangle = \langle y, y \rangle \end{aligned}$$

for some arbitrary real y. $\langle y, y \rangle \geq 0$ always, thus showing Gram matrices to always be PSD.

4. Unconstrained Optimization

(a) Let A be an $m \times n$ matrix and $b \in \mathbb{R}^m$. Consider a convex function $f : \mathbb{R}^m \to \mathbb{R}$. Using the definition of convexity, prove that $g(x) = f(Ax + b)$ is convex.

(b) Prove that if f is strictly convex, f has at most one minimizer (recall that for convex functions, all local minima are also global minima).

(c) Consider the function $f(x) = \frac{1}{2} x^T Ax + b^T x + c$, where A is a symmetric $d \times d$ matrix. Derive the Hessian of f. Under what conditions on A, f is convex? Strictly convex?

Solution
a) Let $\lambda \in [0,1]$, and $y, z \in f$. We know f is convex, meaning that is must satisfy this condition:

$$f(\lambda y + (1-\lambda)z) \leq \lambda f(y) + (1-\lambda)f(z)$$

To prove that $g(x) = f(Ax+b)$, is convex, we must do similarly to above, and apply the condition of convexity:

$$\begin{aligned} g(\lambda x_1 + (1-\lambda)x_2) &= f(A(\lambda x_1 + (1-\lambda)x_2) + b) \\ &= f(\lambda A x_1 + (1-\lambda)Ax_2 + b) \end{aligned}$$

Since we know f is convex by the problem statement, we get

$$f(\lambda A x_1 + (1-\lambda)Ax_2 + b) \leq \lambda f(Ax_1 + b) + (1-\lambda)f(Ax_2 + b)$$

Applying the definition of g, we get

$$g(\lambda x_1 + (1-\lambda)x_2) \leq \lambda g(x_1) + (1-\lambda)g(x_2)$$

Thus, we have shown g to be convex.

b) Assume that 2 minimizers exist: x_1, x_2:

$$f(x_1) = f(x_2) = \min_x f(x)$$

The condition for strict convexity is:

$$f(\lambda x_1 + (1-\lambda)x_2) < \lambda f(x_1) + (1-\lambda)f(x_2)$$

Since $f(x_1) = f(x_2)$, we get

$$f(\lambda x_1 + (1-\lambda)x_2) < f(x_1)$$

This contradicts that x_1 and x_2 are both minimizers, thus showing that if f is strictly convex, then f has at most one minimizer.

c) $H = \nabla^2 f(x)$. $\nabla f(x) = Ax + b$. Taking the gradient once more, we find that $H = \nabla^2 f(x) = A$. By definition, f is convex if H is positive semi-definite. Thus to be convex, A must be symmetric, and all eigenvalues of A must be ≥ 0. To be strictly convex, A must be symmetric, and all eigenvalues must be > 0.

5. The Bayes Classifier and Excess Risk

Consider the 0-1 loss. In class, the Bayes classifier was defined and discussed for multiclass classification. The Bayes classifier h^* achieves the lowest risk $R(h^*) = R^*$, which is called the Bayes Risk. This means that quantity $R(h) - R^*$, which we call the excess risk, is non-negative for any classifier h.
In this problem, we consider binary classification with label $Y \in \{1, -1\}$, and define $\eta(x) := \Pr(Y = 1|X = x)$, the probability that $Y = 1$ given than X takes on the values x. For any classifier h, prove that the excess risk is given by

$$R(h) - R^* = \mathbb{E}_X[|2\eta(X) - 1| \mid 1_{\{h(X)\neq \text{sign}(2\eta(X)-1)\}}]$$

Here

$$\text{sign}(t) := \begin{cases} 1 & t \geq 0 \\ -1 & t < 0 \end{cases}$$

The convention $\text{sign}(0) = 1$ does not affect the problem. The results says that the excess risk depends on how much (on average) $\eta(X)$ deviates from $\frac{1}{2}$ at points where h disagrees with the Bayes classifier.
Hint: Refer to the proof for the Bayes classifier in the lecture notes, and for the binary case rewrite the proof in terms of η.

Solution
We can use the definition of risk of $R(h) = \mathbb{E}_X[P(h(X) \neq Y|X)] = \eta(X)1_{\{h(X)=0\}} + (1 - \eta(X))1_{\{h(X)=1\}}$ to help derive the formula for excess risk. Rewriting the problem we get:

$$\begin{aligned} R(h) - R^* &= R(h) - R(h^*) \\ &= \eta(X)1_{\{h(X)=0\}} + (1 - \eta(X))1_{\{h(X)=1\}} - (\eta(X)1_{\{h^*(X)=0\}} + (1 - \eta(X))1_{\{h^*(X)=1\}}) \end{aligned}$$

We can split this problem into the condition of $h(x) \neq h^*(x)$. In the case of $h(X) = 0, h^*(X) = 1$, we get

$$\begin{aligned} 1_{\{h(X)=0\}} = 1,\ 1_{\{h^*(X)=1\}} = 1 \\ \Rightarrow \eta(X) - (1 - \eta(X)) = 2\eta(X) - 1 \end{aligned}$$

In the case of $h(X) = 1, h^*(X) = 0$, we get

$$\begin{aligned} 1_{\{h(X)=1\}} = 1,\ 1_{\{h^*(X)=0\}} = 1 \\ \Rightarrow (1 - \eta(X)) - \eta(X) = 1 - 2\eta(X) \end{aligned}$$

We can take the absolute value, to get this:

$$\mathbb{E}_X[|2\eta(X) - 1 \mid 1_{\{h(x)\neq h^*(X)\}}]$$

We can use the definition in the problem to find the fact that $h(X) \neq h^*(X)$ is equivalent to $h(X) \neq \text{sign}(2\eta(X) - 1)$. Thus we have arrived the solution of:

$$R(h) - R^* = \mathbb{E}_X[|2\eta(X) - 1| \mid 1_{\{h(X) \neq \text{sign}(2\eta(X)-1)\}}]$$

Index

Code for Exercise 1:

```
import random as rand
import numpy as np
import matplotlib.pyplot as plt

def generate_data(n,d):
    X_train = np.random.randn(n, d)/np.sqrt(d)
    return X_train

def nearest_neigbor_distance(X_train, x):
    distances = np.linalg.norm(X_train - x, axis = 1)
    return np.min(distances)

def expected_distance(n, d):
    X_train = generate_data(n, d)
    distances = [nearest_neigbor_distance(X_train,
    np.random.randn(d)/np.sqrt(d)) for _ in range(100)]
    return np.mean(distances)

d_values = [2,4,6,8,10]
n_values = [100,200,500,1000,2000,5000]

expected_distances = np.zeros((len(d_values), len(n_values)))

for i, d in enumerate(d_values):
    for j, n in enumerate(n_values):
        expected_distances[i,j] = expected_distance(n, d)


plt.figure(figsize=(10, 6))
for i, d in enumerate(d_values):
    plt.plot(n_values, expected_distances[i], marker='o', label=f'd={d}')

plt.xscale("log")
plt.xlabel("Number of training points (n)")
plt.ylabel("Expected nearest neighbor distance E[d(x)]")
plt.title("Expected Nearest Neighbor Distance vs. n for different d")
plt.legend()
plt.grid(True)
plt.show()
```

Runtimes for Exercise 2:

```
Dataset size: n = 1000
Algorithm: ball_tree, Time taken: 0.0020022392 seconds
Algorithm: kd_tree, Time taken: 0.0009977818 seconds
Algorithm: brute, Time taken: 0.0000000000 seconds

Dataset size: n = 5000
Algorithm: ball_tree, Time taken: 0.0019919872 seconds
Algorithm: kd_tree, Time taken: 0.0030074120 seconds
Algorithm: brute, Time taken: 0.0000000000 seconds

Dataset size: n = 10000
Algorithm: ball_tree, Time taken: 0.0050034523 seconds
Algorithm: kd_tree, Time taken: 0.0049917698 seconds
Algorithm: brute, Time taken: 0.0010025501 seconds

Dataset size: n = 100000
Algorithm: ball_tree, Time taken: 0.0550014973 seconds
Algorithm: kd_tree, Time taken: 0.0669918060 seconds
Algorithm: brute, Time taken: 0.0090067387 seconds
```

Code for Exercise 2:

```
import random as rand
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

def generate_Xdata(n):
    X_train = np.random.randn(n, 2)
    return X_train

def generate_Ydata(n):
    Y_train = np.sign(np.random.rand(n))
    return Y_train

def evaluate_knn_times(n, algorithms):
    for i in n:
        print(f"\nDataset size: n = {i}")
        for algo in algorithms:
            clf = KNeighborsClassifier(n_neighbors=5, algorithm=algo)
            start = time.time()
            clf.fit(generate_Xdata(i), generate_Ydata(i))
```

```
            time_taken = time.time() - start
            print(f"Algorithm: {algo}, Time taken: {time_taken:.10f}
            seconds")


def inference_test(n_test, n_values_test, algorithms):
    X_test = np.random.randn(n_test, 2)
    inference_times = {algo: [] for algo in algorithms}
    for n in n_values_test:
        X_train = generate_Xdata(n)
        Y_train = generate_Ydata(n)
        for algo in algorithms:
            clf = KNeighborsClassifier(n_neighbors=5, algorithm=algo)
            clf.fit(X_train, Y_train)
            start = time.time()
            clf.predict(X_test)
            time_taken = time.time() - start
            inference_times[algo].append(time_taken)
    return inference_times

n_values = [1000, 5000, 10000, 100000]
n_values_test = [1000, 2000, 5000, 10000, 20000, 50000,
100000, 200000, 500000]
algorithms = ['ball_tree', 'kd_tree', 'brute']

#part a
evaluate_knn_times(n_values, algorithms)

#part b
test_times = inference_test(5000, n_values_test, algorithms)
plt.figure(figsize=(10, 6))
for algo in algorithms:
    plt.plot(n_values_test, test_times[algo],
    label=f'knn-{algo}', marker='o')

plt.xlabel("Training Set Size (n)")
plt.ylabel("Inference Time (seconds)")
plt.title("kNN Inference Time vs. Training Set Size")
plt.legend()
plt.xscale("log")
plt.yscale("log")
plt.grid()
plt.show()
```