k-Nearest Neighbors

Centre for Data Science, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- Consider a person who is going to vote in the next election.
- There is no information about the voter other than his name.
- A smart way will be to know how the neighbors are voting and based on that it can be predicted.
- This is the basic logic behind the nearest neighbors classification.

The Model

- Nearest neighbors is one of the simplest predictive models.
- No mathematical assumptions, and it doesn't require any sort of heavy machinery.
- It requires some notion of distance.
- Assumption that points that are close to one another are similar.
- It neglects a lot of information as it is only based on the neighbors.

k-NN classification

- A simple code to predict the voting of a person with neighbors details is given below.

```
from typing import List
from collections import Counter
def raw_majority_vote(labels: List[str]) -> str:
  votes = Counter(labels)
  winner, _ = votes.most_common(1)[0]
  return winner
assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

k nearest neighbors

- In a situation when suppose there are equal no. of neighbors in two different categories then, we have several options:

1. Pick one of the winners at random.
2. Weight the votes by distance and pick the weighted winner.
3. Reduce k until we find a unique winner.

- We will be implementing the third way.

k-NN classification

- One thing is sure, that eventually it will work, even if it has to go till the last neighbor(i.e. one neighbor).

```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to
        farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count for count in
        vote_counts.values() if count==winner_count])
    if num_winners == 1:
        return winner # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again
            without the farthest
# Tie, so look at first 4, then 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

k-NN classification

- Below it the function given to create a classifier, which can be used as k-nn classifier.

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance
class LabeledPoint(NamedTuple):
  point: Vector
  label: str
def knn_classify(k: int,labeled_points:
    List[LabeledPoint],new_point:  Vector) -> str:
  # Order the labeled points from nearest to farthest.
  by_distance = sorted(labeled_points,
  key=lambda lp: distance(lp.point, new_point))
  # Find the labels for the k closest
  k_nearest_labels = [lp.label for lp in
      by_distance[:k]]
  # and let them vote.
  return majority_vote(k_nearest_labels)
```

Iris Dataset

- Iris dataset is a staple of machine learning.
- It contains a bunch of measurements for 150 flowers representing three species of iris.
- For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species.
- It can be downloaded from UCI repository.

Iris dataset

- The data is comma-separated.

```
import requests
data=requests.get("https://archive.ics.uci.edu/ml/
            machine-learning-databases/iris/iris.data")
with open('iris.dat', 'w') as f:
  f.write(data.text)
from typing import Dict
import csv
from collections import defaultdict
def parse_iris_row(row: List[str]) -> LabeledPoint:
  """
  sepal_length, sepal_width, petal_length, petal_width,
      class
  """
  measurements = [float(value) for value in row[:-1]]
  # class is e.g. "Iris-virginica"; we just want
      "virginica"
  label = row[-1].split("-")[-1]
  return LabeledPoint(measurements, label)
```

Iris dataset

```
with open('iris.data') as f:
  reader = csv.reader(f)
  iris_data = [parse_iris_row(row) for row in reader]
# We'll also group just the points by species/label
#so we can plot them
points_by_species: Dict[str, List[Vector]] =
    defaultdict(list)
for iris in iris_data:
  points_by_species[iris.label].append(iris.point)
```

Iris dataset

- Graphical representation of data with all possible pair details.

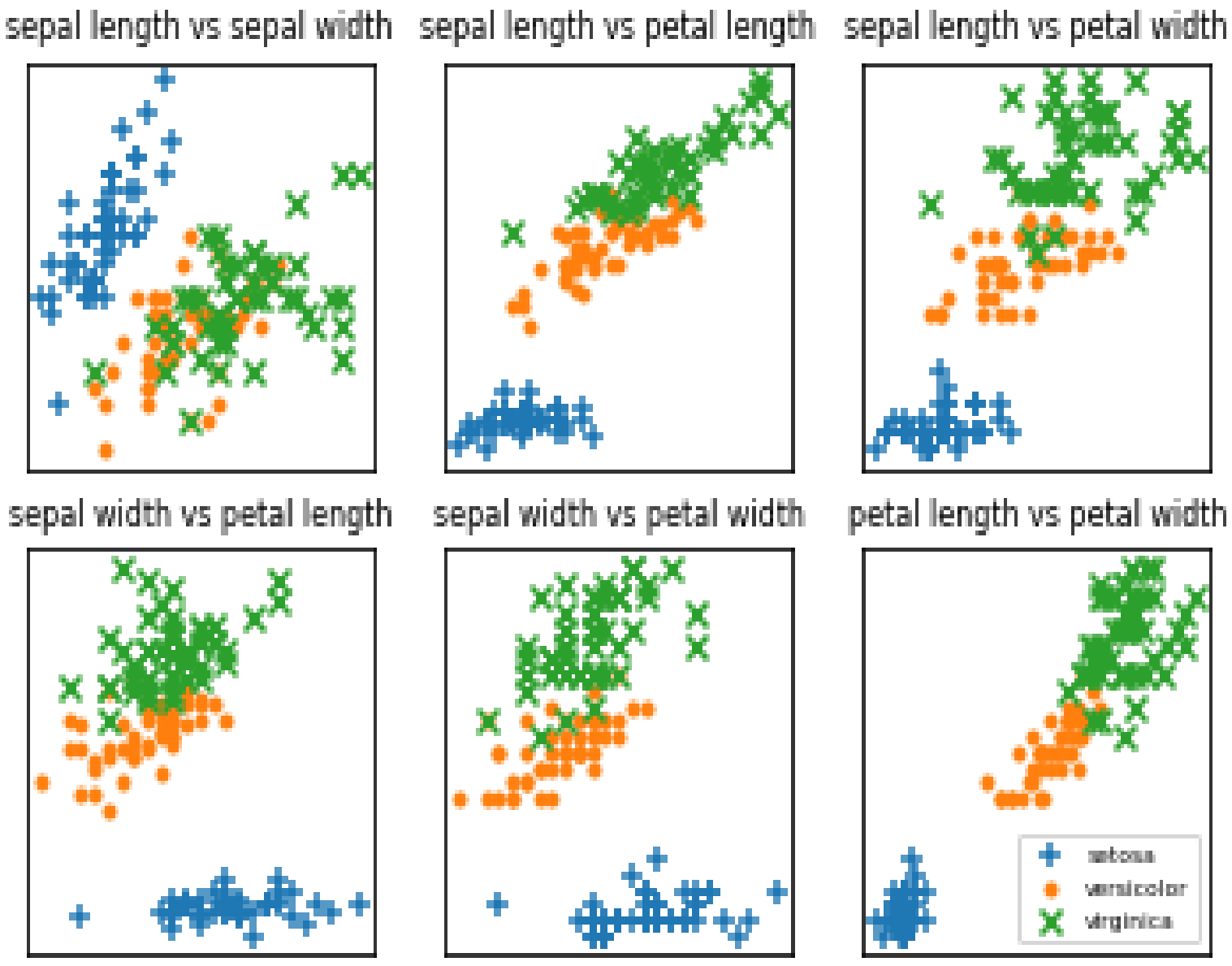
```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal
    length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4)
    if i < j]
marks = ['+', '.', 'x'] # we have 3 classes, so 3
    markers
fig, ax = plt.subplots(2, 3)
for row in range(2):
  for col in range(3):
    i, j = pairs[3 * row + col]
    ax[row][col].set_title(f"{metrics[i]} vs
        {metrics[j]}", fontsize=8)
    ax[row][col].set_xticks([])
    ax[row][col].set_yticks([])
    for mark, (species, points) in
        zip(marks,points_by_species.items()):
      xs = [point[i] for point in points]
      ys = [point[j] for point in points]
      ax[row][col].scatter(xs,ys,marker=mark,label=spec
```

Iris dataset

- Graphical representation of data with all possible pair details.

```
    ax[-1][-1].legend(loc='lower right', prop={'size': 6})
    plt.show()
```

Iris plots


sepal length vs sepal width
sepal length vs petal length
sepal length vs petal width
sepal width vs petal length
sepal width vs petal width
petal length vs petal width
setosa
versicolor
virginica

Iris dataset

- Let's perform the k-NN classification on this datset.

```
import random
from scratch.machine_learning import split_data
random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
from typing import Tuple
# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] =
    defaultdict(int)
num_correct = 0
for iris in iris_test:
  predicted = knn_classify(5, iris_train, iris.point)
  actual = iris.label
  if predicted == actual:
    num_correct += 1
  confusion_matrix[(predicted, actual)] += 1
pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

Iris dataset

- On this simple dataset, the predictions are almost perfect.
- Correct prediction ratio=0.977777777777777
- $defaultdict(< class'int' >, \{('setosa','setosa') : 13\}$
- $('versicolor','versicolor') : 15$
- $('virginica','virginica') : 16$
- $\{('virginica','versicolor') : 1\}$

Curse of dimensionality

- k-nearest neighbors algorithm runs into trouble in higher dimensions.
- Points in high-dimensional spaces tend not to be close to one another at all.
- It can be seen by randomly generating pairs of points in the d-dimensional “unit cube” in a variety of dimensions, and calculating the distances between them.
- For every dimension from 1 to 100, we’ll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension.

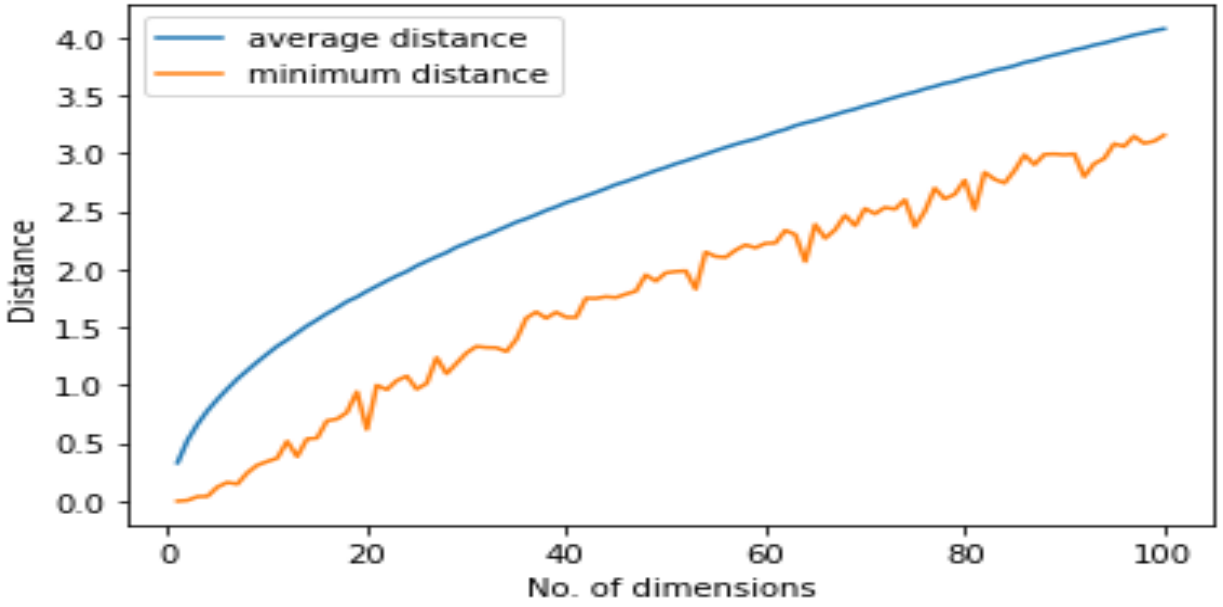
Curse of dimensionality

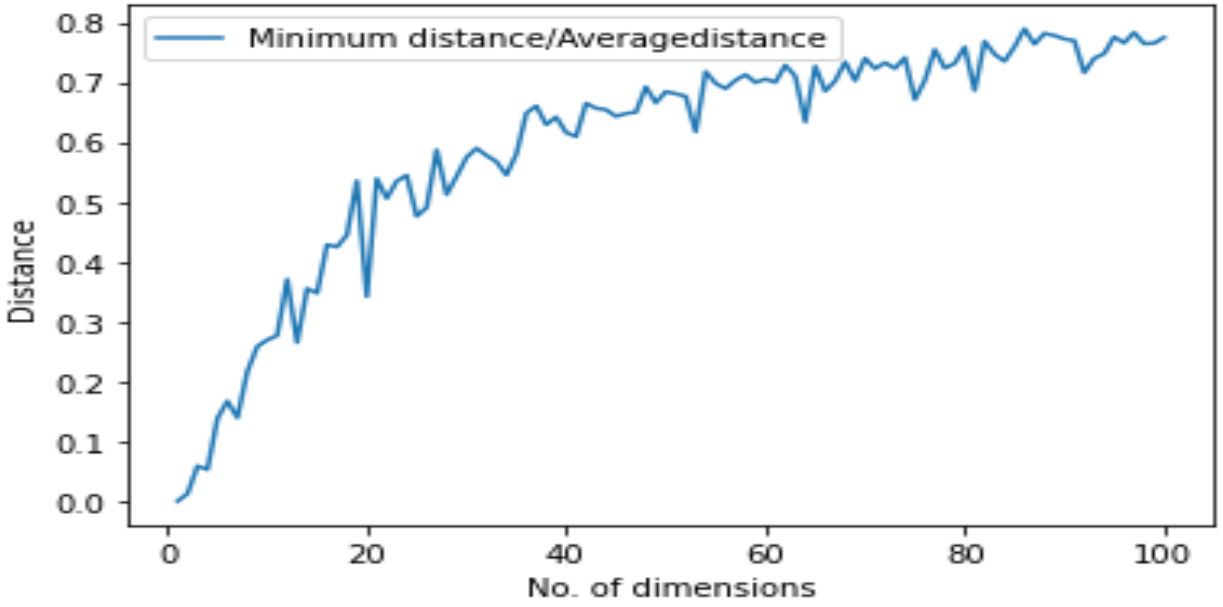
```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
def random_distances(dim: int, num_pairs: int) ->
    List[float]:
    return [distance(random_point(dim), random_point(dim))
        for _ in
         range(num_pairs)]
import tqdm
dimensions = range(1, 101)
avg_distances = []
min_distances = []
random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of
    Dimensionality"):
    distances = random_distances(dim, 10000) # 10,000
        random pairs
    avg_distances.append(sum(distances) / 10000) # track
        the average
    min_distances.append(min(distances)) # track the minimum
```

Curse of dimensionality

```
min_avg_ratio = [min_dist / avg_dist for min_dist, avg_dist
    in
          zip(min_distances, avg_distances)]
plt.xlabel('No. of dimensions')
plt.ylabel('Distance')
plt.plot(dimensions,avg_distances,label='average distance')
plt.plot(dimensions,min_distances,label='minimum distance')
plt.legend()
plt.show()
min_avg_ratio = [min_dist / avg_dist for min_dist, avg_dist
    in zip(min_distances, avg_distances)]
plt.xlabel('No. of dimensions')
plt.ylabel('Distance')
plt.plot(dimensions,min_avg_ratio,label='Minimum
    distance/Averagedistance')
plt.legend()
plt.show()
```

Plots


average distance
minimum distance
Distance
No. of dimensions
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20
40
60
80
100


Minimum distance/Averagedistance
Distance
No. of dimensions
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100

Curse of dimensionality

- In above 1st graph, observe that as no. of dimensions increases, the average distance between point increases and also minimum distance between points in each dimension increases.
- Ratio of closest distance and average distance also goes on increasing, depicting with increase in closest distance is increasing more than increase in average distance.
- In low dimensional datasets, the closest points tend to be much closer than average.
- But two points are close if they are close in every dimension, and with increase in dimension, it gives opportunity for point to be farther away from every other point.

Curse of dimensionality

- Suppose that there are data points belonging to very high dimensions.
- In this case, even if we determine distance between two points and it is found to be very low, it is still a possibility that the points are no-where close to each other.
- In this case it is important to explore the data and find the necessary and important dimensions.
- Thus, reduce the dimensions and then find the distance.

References

[1] Data Science from Scratch_First Principles with Python by Joel Grus, *O'Reilly*.

Thank You
Any Questions?