Lab 6 (Part 2): k-Nearest Neighbors

You are expected, but not required, to complete this portion of the lab by **10:00pm Y**. The official, final deadline for both Part 1 and Part 2 is **11:59pm Thursday, October 29**.

k-Nearest Neighbors is a simple yet popular method for classifying data points. Per its name, it works by looking for training points near a given test point, where "near" is defined by some sort of metric. In particular, the classifier finds the *k* nearest training points to the test point, and counts up how many of each neighbor corresponds to a certain classification: the classification with the most nearby points is the one applied to the test point.

Contents

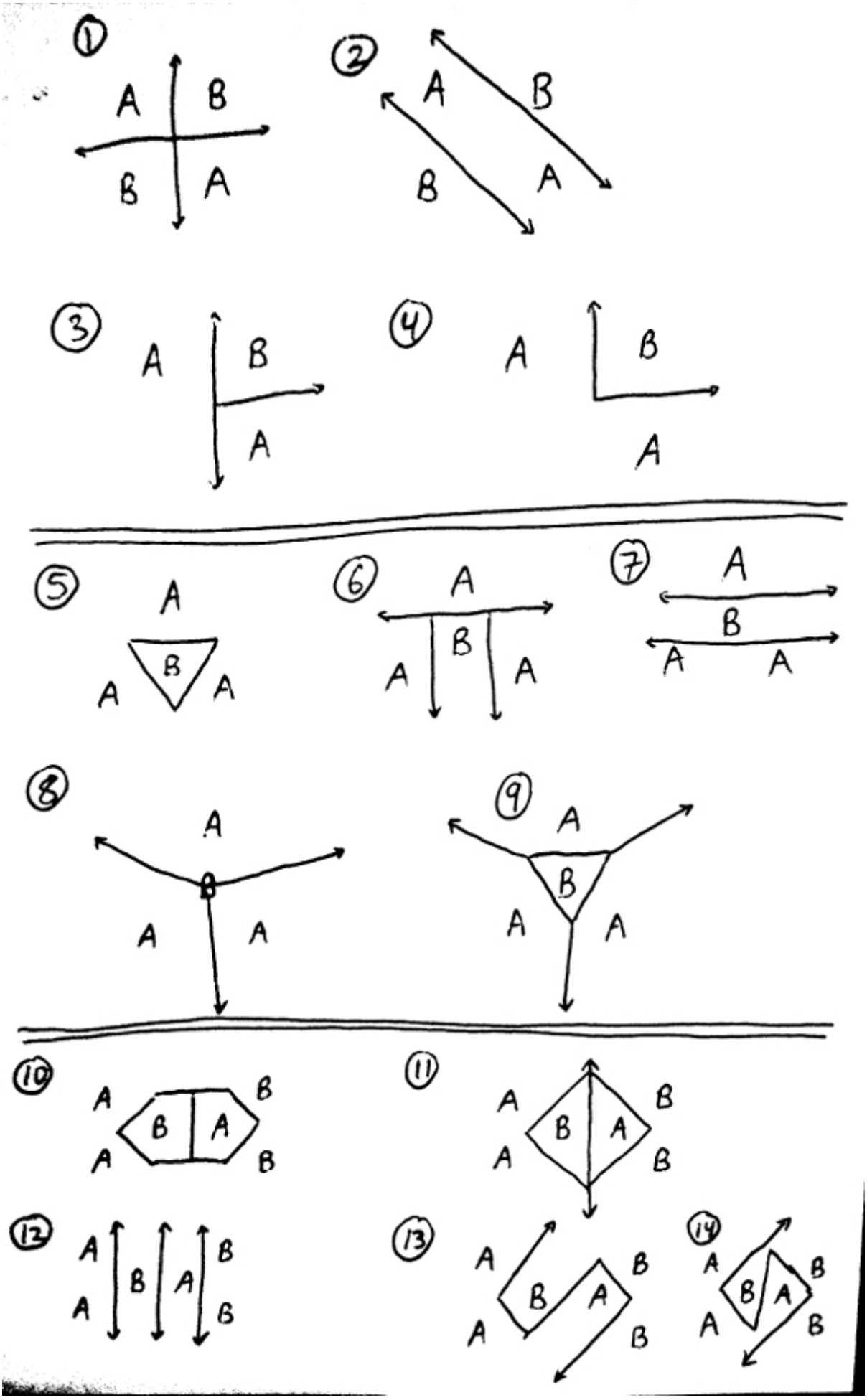
Part 2A: Drawing boundaries

Before we start coding, let's warm up by reviewing decision boundaries. We will be considering decision boundaries for ID trees as well as for kNN (with *k* = 1).

Below are sketches of 4 different data sets in Cartesian space, with numerous ways of drawing decision boundaries for each. For each sketch, answer the following question by filling in BOUNDARY_ANS_n (for n from 1 to 14) with an int 1, 2, 3, or 4:

Which method could create the decision boundaries drawn in the sketch?

1. **Greedy, disorder-minimizing ID tree, using only tests of the form "X > threshold" or "Y > threshold"**
2. **1-nearest neighbors using Euclidean distance**
3. **Both**
4. **Neither**


1
A
B
B
A
2
A
B
B
A
3
A
B
A
4
A
B
A
5
A
B
A
A
6
A
B
A
A
7
A
B
A
A
8
A
A
A
9
A
B
A
A
10
A
B
A
B
A
B
A
B
11
A
B
A
B
A
B
A
B
12
A
B
A
B
A
B
A
B
13
A
B
A
B
A
B
14
A
B
A
B
A
B

For explanations of the answers to some of these questions, search for "#BOUNDARY_ANS_n" in tests.py, for the appropriate value of n.

Part 2B: Distance metrics

When doing k-nearest neighbors, we aren't just restricted to straight-line (Euclidean) distance to compute how far away two points are from each other. We can use many different distance metrics. In 6.034, we cover four such metrics:

- **Euclidean distance**: the straight-line distance between two points.

$$\text{EUCLIDEAN}(\vec{\mathbf{u}}, \vec{\mathbf{v}}) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 +$$

- **Manhattan distance**: the distance between two points assuming you can only move parallel to axes. In two dimensions, you can conceptualize this as only being able to move along the "grid lines."

$$\text{MANHATTAN}(\vec{\mathbf{u}}, \vec{\mathbf{v}}) = \sum_{i=1}^{n} |u_i - v_i| = |u_1 - v_1| + |u_2 - v_2| + \cdot$$

- **Hamming distance**: the number of differing corresponding components between two vectors.

$$\text{HAMMING}(\vec{\mathbf{u}}, \vec{\mathbf{v}}) = \sum_{\substack{1 \le i \le n \\ u_i \neq v_i}} 1 = \delta_{u_1 v_1} + \delta_{u_2 v_2} + \cdots + \delta_{u_n v_n}$$

- **Cosine distance**: compares the angle between two vectors, where each point is represented as a vector from the origin to the point's coordinates. Note that cosine is a similarity measure, not a difference measure, because cos(0)=1 means two vectors are "perfectly similar," cos(90 deg)=0 represents two "perfectly orthogonal" vectors, and cos(180 deg)=-1 corresponds to two "maximally dissimilar" (or "perfectly opposite") vectors. For our code, however, we need this metric to be a difference measure so that smaller distances means two vectors are "more similar," to match all of the other distance metrics. Accordingly, we will define the cosine distance to be 1 minus the cosine of the angle between two vectors, so that our cosine distance varies between 0 (perfectly similar) and 2 (maximally dissimilar). Thus, the formula for cosine distance is:

$$\text{COSINE}(\vec{\mathbf{u}}, \vec{\mathbf{v}}) = 1 - \frac{\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}}{\|\vec{\mathbf{u}}\| \|\vec{\mathbf{v}}\|}$$

It's also possible to transform data instead of using a different metric -- for instance, transforming to polar coordinates -- but in this lab we will only work with distance metrics in Cartesian coordinates.

We'll start with some basic functions for manipulating vectors, which may be useful for `cosine_distance`. For these, we will represent an *n*-dimensional vector as a list or tuple of *n* coordinates. `dot_product` should compute the dot product of two vectors, while `norm` computes the length of a vector. (There is a simple implementation of `norm` that uses `dot_product`.) Implement both functions:

```
def dot_product(u, v):
```

```
def norm(v):
```

Next, you'll implement each of the four distance metrics. For the rest of the k-nearest neighbors portion of this lab, we will represent data points as **Point** objects (described below in the API). Because Point objects are iterable (e.g. you can do `for coord in point:`), you should be able to call your vector methods on them directly. Each distance function should take in two Points and return the distance between them as a float or int. *Python hint*: The built-in Python keyword `zip` may be useful. To learn more, type `help(zip)` in a Python command line, or read about it online.

Implement each distance function:

```
def euclidean_distance(point1, point2):
```

```
def manhattan_distance(point1, point2):
```

```
def hamming_distance(point1, point2):
```

```
def cosine_distance(point1, point2):
```

Part 2C: Classifying Points

Now that we have defined four distance metrics, we will use them to classify points using k-nearest neighbors with various values of k.

First, write a function `get_k_closest_points` that takes in a point (a Point object), some data (a list of Point objects), a number k, and a distance metric (a function, such as euclidean_distance), and returns the k points in the data that are nearest to the input point, according to the distance metric. In case of a tie, sort by `point.coords` (i.e. break ties lexicographically by coordinates). This function can be written cleanly in about five lines.

```
def get_k_closest_points(point, data, k, distance_metric):
```

A note about ties: There are two ways to get ties when classifying points using k-nearest neighbors.

1. If two or more training points are equidistant to the test point, the "k nearest" points may be undefined. If the equidistant points have the same classification, it doesn't make a difference, but in some cases they could cause the classification to be ambiguous. In this lab, these cases are handled in `get_k_closest_points` by breaking ties lexicographically.
2. If the two most likely classifications are equally represented among the k nearest training points, the classification is ambiguous. In other words, the k-nearest neighbors are voting on a classification, and if the vote is tied, there needs to be some mechanism for breaking the tie. In this lab, we will assume that there are no voting ties, so you don't need to worry about tie-breaking. (All of the test cases in this section are designed to ensure that there will not be any voting ties.)

Your task is to use that to write a function that classifies a point, given a set of data (as a list of Points), a value of k, and a distance metric (a function):

```
def knn_classify_point(point, data, k, distance_metric):
```

Python hint: To get the mode of a list, there's a neat one-line trick using `max` with the `key` argument, the `list.count` function, and converting a list to a set to get the unique elements. `.count` is a built-in list method that counts how many times an item occurs in a list. For example, `[10, 20, 20, 30].count(20)` -> `2`.

Part 2D: Choosing the best k and distance metric via cross-validation

There are many possible implementations of cross validation, but the general idea is to separate your data into a training set and a test set, then use the test set to determine how well the training parameters worked. For k-nearest neighbors, this means choosing a value of k and a distance metric, then testing each point in the test set to see whether they are classified correctly. The "cross" part of cross-validation comes from the idea that you can re-separate your data multiple times, so that different subsets of the data take turns being in the training and test sets, respectively.

For the next function, you'll implement a specific type of cross-validation known as leave-one-out cross-validation. If there are n points in the data set, you'll separate the data n times into a training set of size n-1 (all the points except one) and a test set of size one (the one point being left out). Implement the function `cross_validate`, which takes in a set of data, a value of k, and a distance metric, and returns the fraction of points that were classified correctly using leave-one-out cross-validation.

```
def cross_validate(data, k, distance_metric):
```

Use your cross-validation method to write a function that takes in some data (a list of Points) and tries every combination of reasonable values of k and the four distance metrics we defined above, then returns a tuple `(k, distance_metric)` representing the value of k and the distance metric that produce the best results with cross validation:

```
def find_best_k_and_metric(data):
```

Part 2E: More Multiple Choice

Questions 1-3: More tree classification

These questions refer to the k-nearest neighbors data on classifying trees, from 2014 Quiz 2, Part B.

Recall the following terms:

- Overfitting: Occurs when small amounts of noise in the training data are treated as meaningful trends in the population. (i.e. when relying too heavily on the data, or overutilizing the data, causes a poor classification rate)
- Underfitting: Occurs when the algorithm blurs out differences in the training data that reflect genuine trends in the population. (i.e. when ignoring information in the data, or underutilizing the data, causes a poor classification rate)

kNN Question 1: Consider the results of cross-validation using k=1 and the Euclidean distance metric. Why is 1 not a good value for k? Answer "Overfitting" or "Underfitting" in kNN_ANSWER_1.

kNN Question 2: Consider the results of cross-validation using k=7 and the Euclidean distance metric. Why is 7 not a good value for k? Answer "Overfitting" or "Underfitting" in kNN_ANSWER_2.

kNN Question 3: Euclidean distance doesn't seem to do very well with any value of k (for this data). Which distance metric yields the highest classification rate for this dataset? Answer 1, 2, 3, or 4 in kNN_ANSWER_3:

1. Euclidean distance
2. Manhattan distance
3. Hamming distance
4. Cosine distance

Questions 4-7: Choosing metrics

kNN Question 4: Suppose you wanted to classify a mysterious drink as coffee, energy drink, or soda based on the amount of caffeine and amount of sugar per 1-cup serving. Coffee typically contains a large amount of caffeine, lemonade typically contains a large amount of sugar, and energy drinks

typically contain a large amount of both. Which distance metric would work best, and why? Answer 1, 2, 3, or 4 in kNN_ANSWER_4:

1. Euclidean distance, because you're comparing standard numeric quantities
2. Manhattan distance, because it makes sense to array the training data points in a grid
3. Hamming distance, because the features "contains caffeine" and "contains sugar" are boolean values
4. Cosine distance, because you care more about the ratio of caffeine to sugar than the actual amount

kNN Question 5: Suppose you add a fourth possible classification, water, which contains no caffeine or sugar. Now which distance metric would work best, and why? Answer 1, 2, 3, or 4 in kNN_ANSWER_5:

1. Euclidean distance, because you're still comparing standard numeric quantities, and it's now more difficult to compare ratios
2. Manhattan distance, because there are now four points, so it makes even more sense to array them in a grid
3. Hamming distance, because the features "contains caffeine" and "contains sugar" are still boolean values
4. Cosine distance, because you still care more about the ratio of caffeine to sugar than the actual amount

kNN Question 6: While visiting Manhattan, you discover that there are 3 different types of taxis, each of which has a distinctive logo. Each type of taxi comes in many makes and models of cars. Which distance metric would work best for classifying additional taxis based on their logos, makes, and models? Answer 1, 2, 3, or 4 in kNN_ANSWER_6:

1. Euclidean distance, because there's a lot of variability in make and model
2. Manhattan distance, because the streets meet at right angles (and because you're in Manhattan and classifying taxicabs)
3. Hamming distance, because the features are non-numeric
4. Cosine distance, because it will separate taxis by how different their makes and models are

kNN Question 7: Why might an ID tree work better for classifying the taxis from Question 6? Answer 1, 2, 3, or 4 in kNN_ANSWER_7:

1. k-nearest neighbors requires data to be graphed on a coordinate system, but the training data isn't quantitative
2. Some taxis may have the same make and model, and k-nearest neighbors can't handle identical training points
3. An ID tree can ignore irrelevant features, such as make and model
4. Taxis aren't guaranteed to stay within their neighborhood in Manhattan

For explanations of the answers to some of these questions, search for "#kNN_ANSWER_n" in tests.py, for the appropriate value of n.

Point API

The `Point` class supports iteration (to iterate through the coordinates).

A `Point` has the following attributes:

`name`
The name of the point (a string), if defined.

`coords`
The coordinates of the point, represented as a vector (a tuple or list of numbers).

`classification`
The classification of the point, if known.