BIS634 Yuechen Liu HW5

Exercise 1 (25 points):

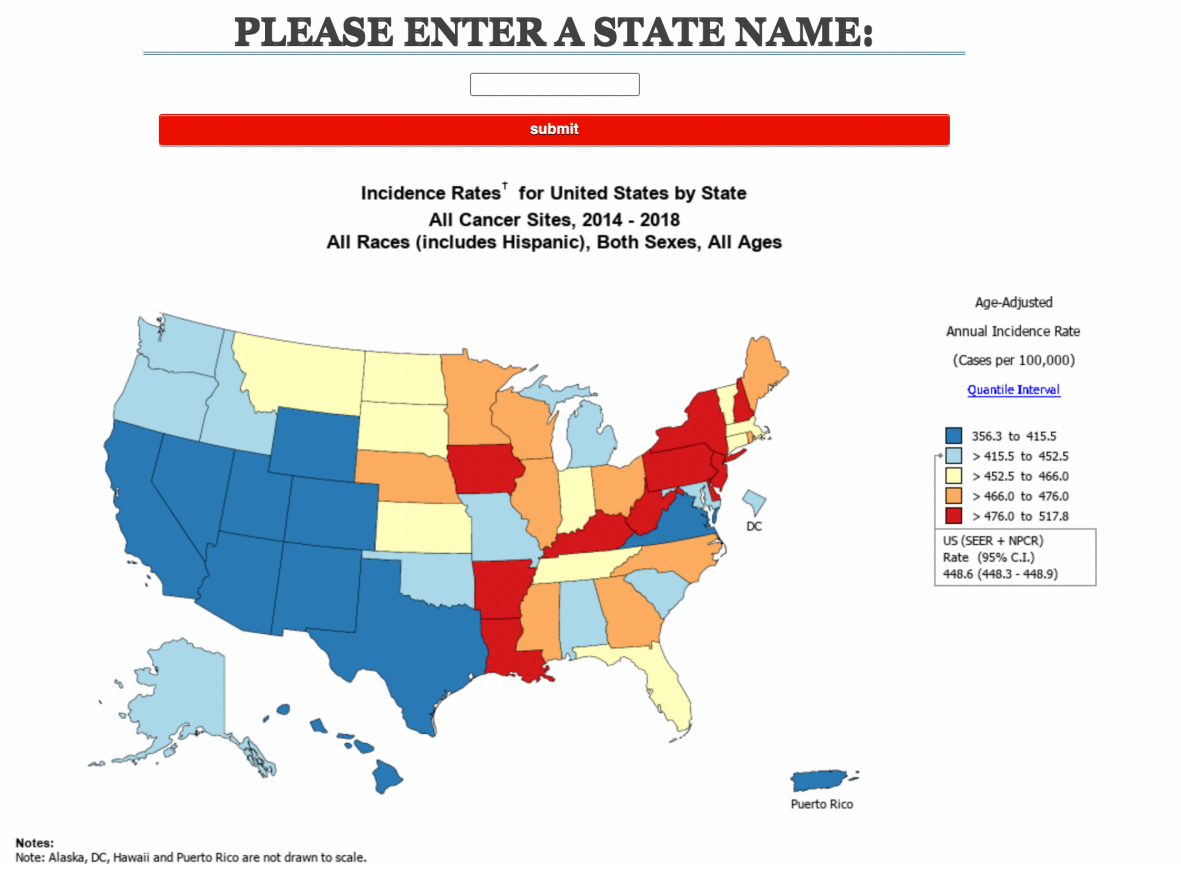
Perform any necessary data cleaning (e.g. you'll probably want to get rid of the numbers in e.g. "Connecticut(7)" which refer to data source information as well as remove lines that aren't part of the table). Include the cleaned CSV file in your homework submission, and make sure your readme includes a citation of where the original data came from and how you changed the csv file. (5 points)

Answer: I select the rows with over 10 elements using csv lib. Citation: The original data came from "https://statecancerprofiles.cancer.gov/incidencerates/index.php?stateFIPS=00&areatype=state&cancer=001&race=00&sex=0&age=001&stage=999&year=0&type=incd&sortVariableName=name&sortOrder=asc#results" 1)download the data from above address 2)create a new .csv file to save the cleaned data: "new_incd.csv" file 3)set a number 10 to select the rows, because the useful data has over 10 dims 4)get rid of the numbers in e.g."Connecticut(7)": manually delete numbers, '(', and ')'. 5)manually organize the names of the columns.

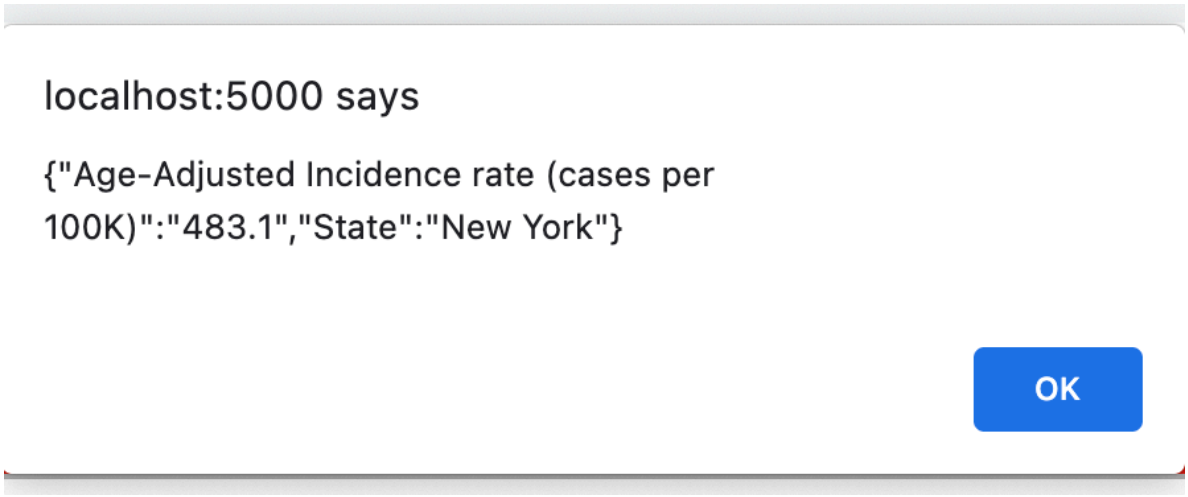
Using Flask, implement a server that provides three routes (5 points each)You've now completed most of this course, so you're now qualified to choose the next step. Take this exercise one step beyond that which is described above in a way that you think is appropriate, and discuss your extension in your readme. (e.g. you might show maps, or provide more data, or use CSS/JS to make the page prettier or more interactive, or use a database, or...)

Answer: 1)include two .html files (homepage.html and errorpage.html) 2)Searching for state names is not case-sensitive and space-insensitive 3)test: http://localhost:5000/ return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"} http://localhost:5000/state/Kentucky return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"} http://localhost:5000/info?state_name=Kentucky return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"} 4)use css to decorate, saving to style.css file.

The homepage:



The info page:



The error page:

Exercise 2 (25 points):

Extend the basic binary tree example from slides2 into a binary search tree that is initially empty (5 points). Provide an add method that inserts a single numeric value at a time according to the rules for a binary search tree (5 points).

Answer:

```
class Tree:
    def __init__(self, value=None):
        self.value = value
        self.left = None
        self.right = None

    def add(self, item):
        if self.value is None:
            self.value = item
        elif item < self.value:
            if self.left:
                self.left.add(item)
            else:
                self.left = Tree(item)
        elif item > self.value:
            if self.right:
                self.right.add(item)
            else:
                self.right = Tree(item)

    def __contains__(self, item):
        if self.value == item:
            return True
        elif self.left and item < self.value:
            return item in self.left
        elif self.right and item > self.value:
            return item in self.right
        else:
            return False
```

Replace the contains method of the tree with the following contains method. The change in name will allow you to use the in operator; e.g. after this change, 55 in my_tree should be True in the above example, whereas 42 in my_tree would be False. Test this.

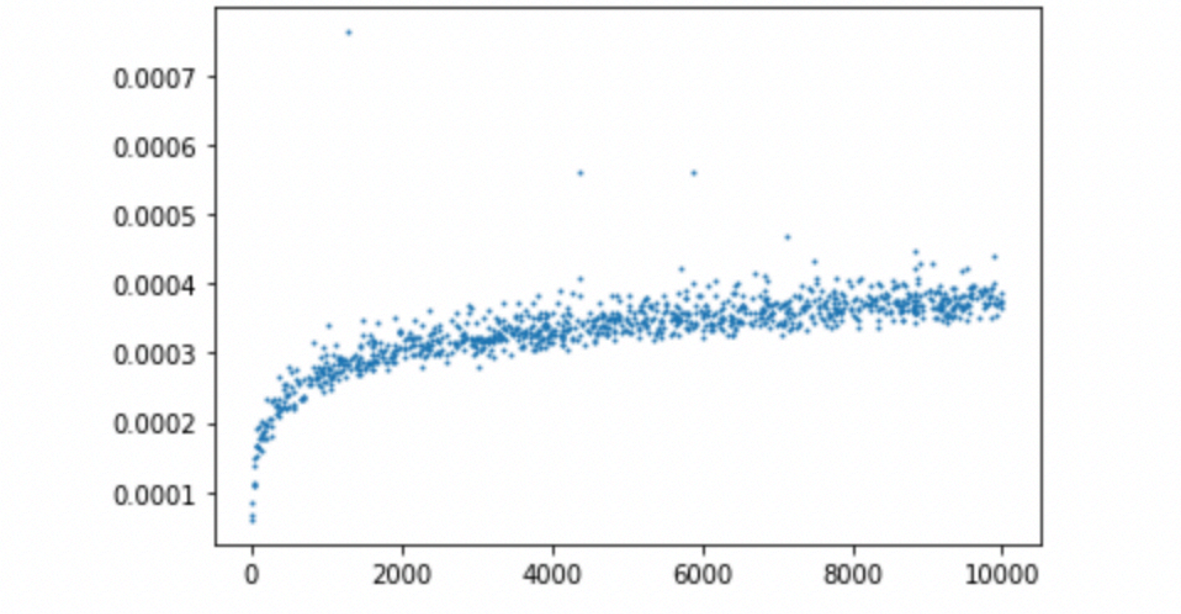
Answer: The test tree is [55, 62, 37, 49, 71, 14, 17]. I test if 55, 42,71, and 4 are in the tree, and the answer is true, false,true, false.

```
import random
my_tree = Tree()
test_tree = [55, 62, 37, 49, 71, 14, 17]
for item in test_tree:
    my_tree.add(item)
print(55 in my_tree)
print(42 in my_tree)
print(71 in my_tree)
print(4 in my_tree)
```

```
True
False
True
False
```

Using various sizes n of trees (populated with random data) and sufficiently many calls to in (each individual call should be very fast, so you may have to run many repeated tests), demonstrate that in is executing in O(log n) times; on a log-log plot, for sufficiently large n, the graph of time required for checking if a number is in the tree as a function of n should be almost horizontal. (5 points).

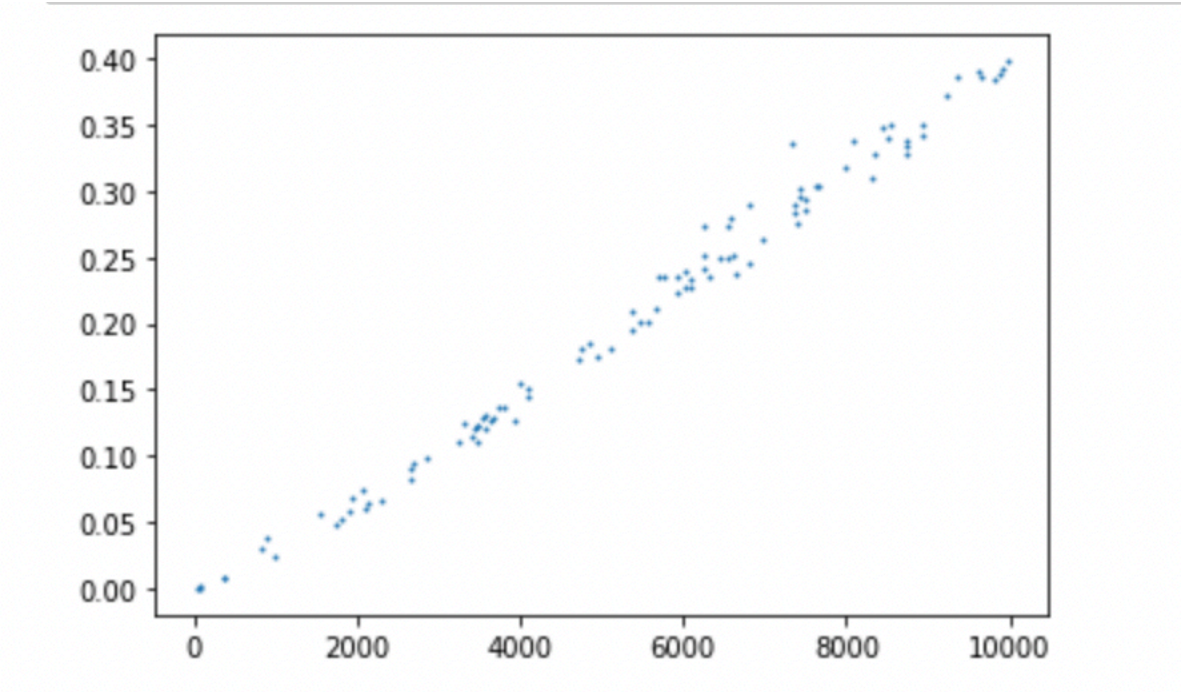
Answer: As shown in the below plot, the in is executing in O(log n) times.



This speed is not free. Provide supporting evidence that the time to setup the tree is O(n log n) by timing it for various sized ns and showing that the runtime lies between a curve that is O(n) and one that is O(n**2). (5 points)

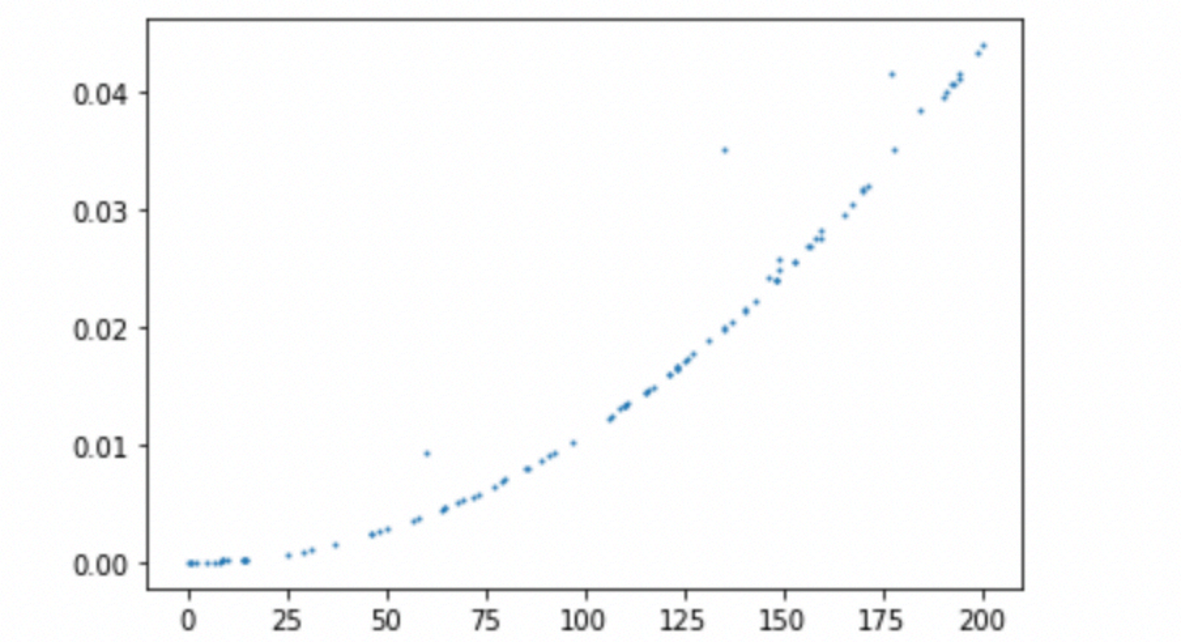
Answer: This way is to draw two graphs of extream situation,O(n) and O(n**2), which means that the runtime must lie between those two curves. If the input secquence is random enough, the tree is almost balanced and the setup time is O(n), as shown below:

```
time_per_test = []
n_samples = 100
repeat_times = 10
for _ in range(n_samples):
    n = random.randint(0, 10000)
    secquence = [random.random() for _ in range(n)]
    start = time.time()
    for _ in range(repeat_times):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    time_per_test.append((end - start, n))

plt.scatter([x[1] for x in time_per_test], [x[0] for x in time_per_test], s=1)
plt.show()
```



However, if the input is inversely sorted, than the setup time is O(n**2):**

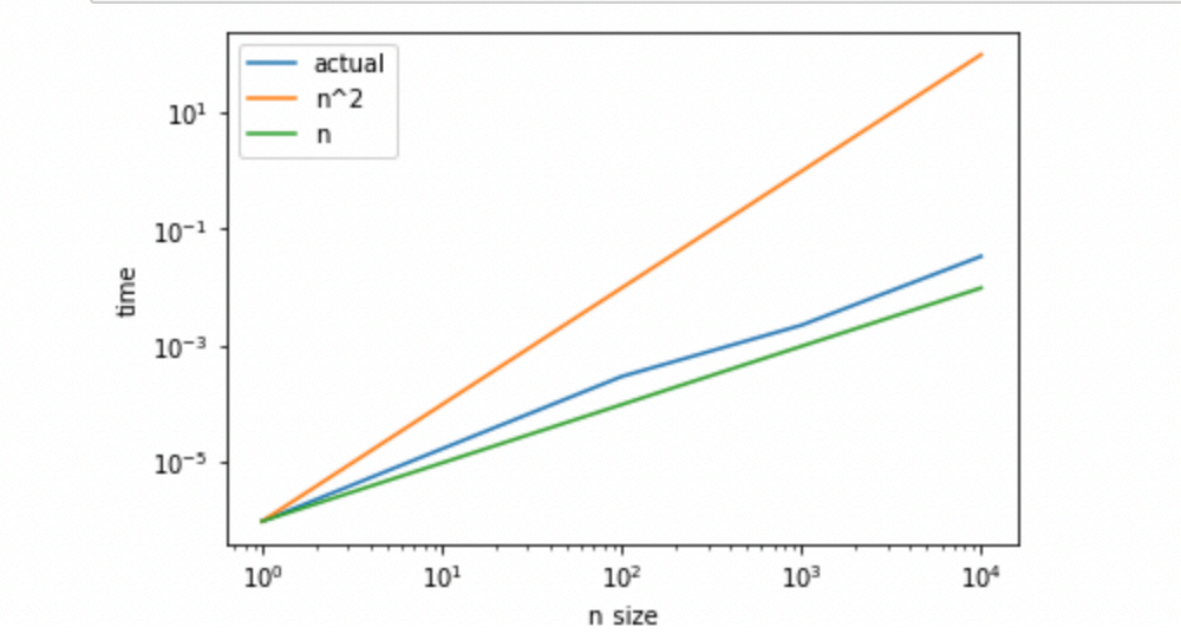
```
time_per_test = []
n_samples = 100
repeat_times = 10
for _ in range(n_samples):
    n = random.randint(0, 200)
    secquence = [random.random() for _ in range(n)]
    secquence = sorted(secquence)
    start = time.time()
    for _ in range(repeat_times):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    time_per_test.append((end - start, n))

plt.scatter([x[1] for x in time_per_test], [x[0] for x in time_per_test], s=1)
plt.show()
```



If I create a function and put actual time spend, n2, and n in one graph:

```
def timefunc(n):
    times = []
    n = [random.randint(1,n) for _ in range(n)]
    for attempt in [i for i in range(100)]:
        start = time.time
        my_tree = Tree()
        for item in n:
            my_tree.add(item)
        end = time.time()
        times.append(end - start)
    return min(times)
```

```
ns = [1,100,1000,10000] #Select different n size
actual = [timefunc(n) for n in ns]
s = actual [0]
quadratic = [s*n**2 for n in ns]
linear = [s*n for n in ns]
plt.plot(ns,actual)
plt.plot(ns,quadratic)
plt.plot(ns,linear)
plt.xscale('log')
plt.yscale('log')
plt.ylabel('time')
plt.xlabel('n_size')
plt.legend(['actual','n^2','n'],loc = 'upper left')
plt.show()
```



As we can see from the above graph, the actual running time lies between O (n**2) and O (n).

Exercise 3 (50 points):

Citation:
https://github.com/diana12333/QuadtreeNN/blob/anewbranch/QuadTree_Report.pdf https://en.wikipedia.org/wiki/Quadtree
https://cloud.tencent.com/developer/article/1487842
https://scipython.com/blog/quadtrees-2-implementation-in-python/
https://jrtechs.net/data-science/implementing-a-quadtree-in-python

Normalize the seven quantitative columns to a mean of 0 and standard deviation 1. (3 points)

Answer:

Normalize the data

```
for c in data.columns[:-1]:
    data[c] = (data[c] - np.mean(data[c])) / np.std(data[c])
```

```
data.head()
```

	AREA	PERIMETER	MAJORAXIS	MINORAXIS	ECCENTRICITY	CONVEX_AREA	EXTENT	CLASS
0	1.479830	2.004354	2.348547	-0.212943	2.018337	1.499659	-1.152921	Cammeo
1	1.147870	1.125853	0.988390	0.945568	0.410018	1.192918	-0.602079	Cammeo
2	1.135169	1.317214	1.451908	0.253887	1.212956	1.126504	0.405611	Cammeo
3	0.293436	0.115300	0.261439	0.198051	0.239751	0.233857	-0.275351	Cammeo
4	1.166345	1.487053	1.316442	0.523419	0.952221	1.299855	-0.206013	Cammeo

```
data.value_counts('CLASS')
```

```
CLASS
Osmancik    2180
Cammeo      1630
dtype: int64
```

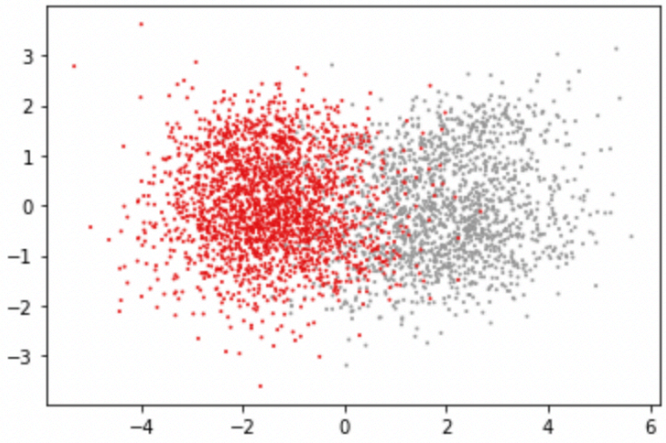
Plot this on a scatterplot, color-coding by type of rice.

Answer:

PCA

```
pca = decomposition.PCA(n_components=2)
data_reduced = pca.fit_transform(data[[
    'AREA',
    'PERIMETER',
    'MAJORAXIS',
    'ECCENTRICITY',
    'CONVEX_AREA',
    'EXTENT'
    ]])
pc0 = data_reduced[:, 0]
pc1 = data_reduced[:, 1]
```

```
plt.scatter(pc0, pc1,
    c=data.CLASS.map({'Osmancik': 0, 'Cammeo': 1}),
    s=1, cmap='Set1')
plt.show()
```



Comment on what the graph suggests about the effeciveness of using k-nearest neighbors on this 2-dimensional reduction of the data to predict the type of rice. (4 points)

Answer: The PCA model is able to reduce the dimensionality of the data to 2. The gaussion distance of two samples seems to be small if they belong to the same class.

Implement a two-dimensional k-nearest neighbors classifier (in particular, do not use sklearn for k-nearest neighbors here): given a list of (x, y; class) data, store this data in a quad-tree (14 points). Given a new (x, y) point and a value of k (the number of nearest neighbors to examine), it should be

able to identify the most common class within those k nearest neighbors (14 points).

Answer:

```
class Point: #the points in the training set
    def __init__(self, x, y, payload=None):
        self.x, self.y = x, y
        self.payload = payload

    def distance_to(self, other):
        try:
            other_x, other_y = other.x, other.y
        except AttributeError:
            other_x, other_y = other
        return np.hypot(self.x - other_x, self.y - other_y)

class Rect: #Rectangular store the shape and other attribute of the quadtree
    #cx: center point x-axis
    #cy: center point y-axis
    #w: the width of the rectangular
    #h: the height of the rectangular
    def __init__(self, cx, cy, w, h):
        self.cx, self.cy = cx, cy
        self.w, self.h = w, h
        self.west_edge, self.east_edge = cx - w/2, cx + w/2
        self.north_edge, self.south_edge = cy - h/2, cy + h/2

    def contains(self, point): #validate if a point is inside of the rectangular
        try:
            point_x, point_y = point.x, point.y
        except AttributeError:
            point_x, point_y = point

        return (point_x >= self.west_edge and
                point_x <  self.east_edge and
                point_y >= self.north_edge and
                point_y < self.south_edge)

    def intersects(self, other):#validate if the rectangular is in the center and the radius
        return not (other.west_edge > self.east_edge or
                    other.east_edge < self.west_edge or
                    other.north_edge > self.south_edge or
                    other.south_edge < self.north_edge)
```

```python
class QuadTree:
    #A class implementing a quadtree.
    #boundary: the rectangular that store the shape and cneter
    #depth: the max number of points can be stored in this quad tree

    def __init__(self, boundary, max_points=4, depth=0):
        self.boundary = boundary
        self.max_points = max_points
        self.points = []
        self.depth = depth
        self.divided = False

    def divide(self): #divide the quad tree into 4 sub quadtree
        cx, cy = self.boundary.cx, self.boundary.cy
        w, h = self.boundary.w / 2, self.boundary.h / 2
        self.nw = QuadTree(Rect(cx - w/2, cy - h/2, w, h),
                           self.max_points, self.depth + 1)
        self.ne = QuadTree(Rect(cx + w/2, cy - h/2, w, h),
                           self.max_points, self.depth + 1)
        self.se = QuadTree(Rect(cx + w/2, cy + h/2, w, h),
                           self.max_points, self.depth + 1)
        self.sw = QuadTree(Rect(cx - w/2, cy + h/2, w, h),
                           self.max_points, self.depth + 1)
        self.divided = True

    def insert(self, point): #insert the points into the sub quadtree.
        if not self.boundary.contains(point):
            # The point does not lie inside boundary: bail.
            return False
        if len(self.points) < self.max_points:
            # There's room for our point without dividing the QuadTree.
            self.points.append(point)
            return True

        # No room: divide if necessary, then try the sub-quads.
        if not self.divided:
            self.divide()

        return (self.ne.insert(point) or
                self.nw.insert(point) or
                self.se.insert(point) or
                self.sw.insert(point))
```

```
    def query_circle(self, boundary, centre, radius, found_points):
        """Find the points in the quadtree that lie within radius of centre.

        boundary is a Rect object (a square) that bounds the search circle.
        There is no need to call this method directly: use query_radius.

        """

        if not self.boundary.intersects(boundary):
            # If the domain of this node does not intersect the search
            # region, we don't need to look in it for points.
            return False

        # Search this node's points to see if they lie within boundary
        # and also lie within a circle of given radius around the centre point.
        for point in self.points:
            if (boundary.contains(point) and
                    point.distance_to(centre) <= radius):
                found_points.append(point)

        # Recurse the search into this node's children.
        if self.divided:
            self.nw.query_circle(boundary, centre, radius, found_points)
            self.ne.query_circle(boundary, centre, radius, found_points)
            self.se.query_circle(boundary, centre, radius, found_points)
            self.sw.query_circle(boundary, centre, radius, found_points)
        return found_points

    def query_radius(self, centre, radius, found_points):
        """Find the points in the quadtree that lie within radius of centre."""

        boundary = Rect(*centre, 2*radius, 2*radius)
        return self.query_circle(boundary, centre, radius, found_points)


    def __len__(self):
        npoints = len(self.points)
        if self.divided:
            npoints += len(self.nw)+len(self.ne)+len(self.se)+len(self.sw)
        return npoints
```

```python
class KNN(): #K Neaarest Neighbor algorithm
    def __init__(self, k = 1) -> None:
        self.qtree = QuadTree(Rect(0, 0, 25, 25))
        self.k = k
        self.pca = decomposition.PCA(n_components=2)

    def fit(self, X, y):
        X = self.pca.fit_transform(X)
        for _x, _y in zip(X, y):
            self.qtree.insert(Point(_x[0], _x[1], _y))

    def predict(self, X):
        k = self.k
        pred = []
        # binary search for radius
        r_max = 5
        r_min = 0
        while True:
            found_points = []
            r = (r_max + r_min) / 2
            self.qtree.query_radius((X[0], X[1]), r, found_points)
            if len(found_points) > k:
                r_max = r
            elif len(found_points) < k:
                r_min = r
            else:
                break
        for p in found_points:
            pred.append(p.payload)
        return np.bincount(pred).argmax()

    def score(self, X, y):
        X = self.pca.transform(X)
        pred = []
        for x in X:
            pred.append(self.predict(x))
        pred = np.array(pred)
        y = np.array(y)
        return confusion_matrix(y, pred)

    def get_params(self, *args, **kwargs):
        return dict()
```

```python
_data = data.sample(frac=1)
X = _data[[
    'AREA',
    'PERIMETER',
    'MAJORAXIS',
    'ECCENTRICITY',
    'CONVEX_AREA',
    'EXTENT'
    ]]
Y = _data.CLASS.map({'Osmancik': 0, 'Cammeo': 1})
```

Using 5-fold cross-validation with your k-nearest neighbors implementation, give the confusion matrix for predicting the type of rice with k=1. (4 points) Repeat for k=5. (4 points)

Answer: When k = 1:

When k = 1:

```
for i in range(0, len(X), int(len(X)/5)):
    i = int(i)
    x_test = X.iloc[i: i+int(len(X)/5)]
    x_train = X.drop(x_test.index)
    y_test = Y[x_test.index]
    y_train = Y.drop(x_test.index)
    knn = KNN(k=1)
    knn.fit(x_train, y_train)
    print(knn.score(x_test, y_test))
```

```
[[386  51]
 [ 53 272]]
[[381  45]
 [ 47 289]]
[[417  34]
 [ 35 276]]
[[377  56]
 [ 46 283]]
[[394  39]
 [ 49 280]]
```

When k = 5:

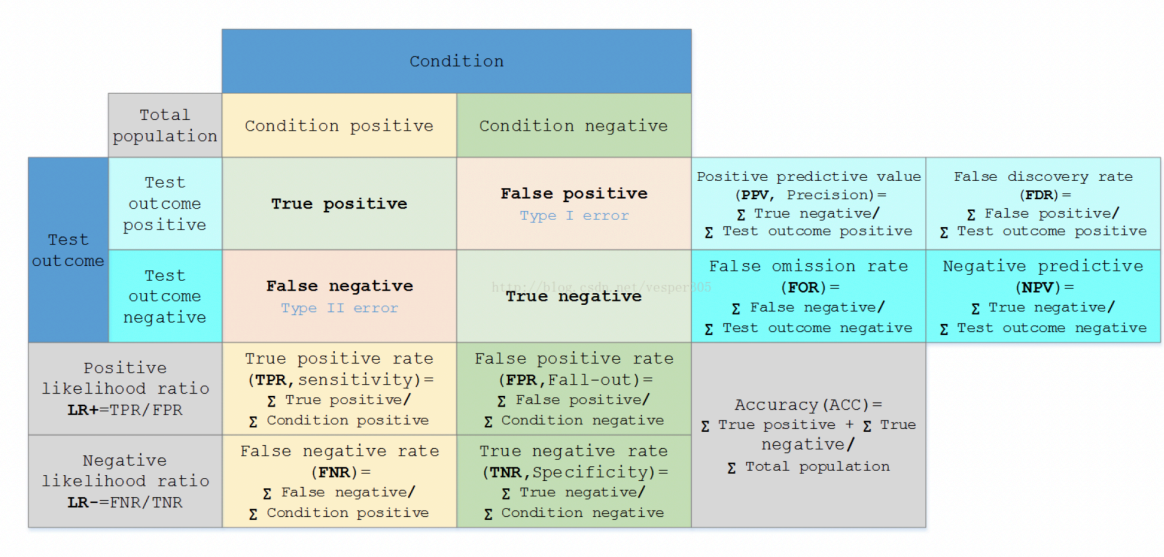
When k = 5:

```
: for i in range(0, len(X), int(len(X)/5)):
    i = int(i)
    x_test = X.iloc[i: i+int(len(X)/5)]
    x_train = X.drop(x_test.index)
    y_test = Y[x_test.index]
    y_train = Y.drop(x_test.index)
    knn = KNN(k=5)
    knn.fit(x_train, y_train)
    print(knn.score(x_test, y_test))
```

```
[[399  38]
 [ 31 294]]
[[396  30]
 [ 28 308]]
[[427  24]
 [ 28 283]]
[[402  31]
 [ 37 292]]
[[406  27]
 [ 47 282]]
```

Provide a brief interpretation of what the confusion matrix results mean. (4 points)

Answer:



		Condition			
	Total population	Condition positive	Condition negative		
Test outcome	Test outcome positive	**True positive**	**False positive** Type I error	Positive predictive value (**PPV**, Precision) = Σ True negative/ Σ Test outcome positive	False discovery rate (**FDR**) = Σ False positive/ Σ Test outcome positive
	Test outcome negative	**False negative** Type II error	**True negative**	False omission rate (**FOR**) = Σ False negative/ Σ Test outcome negative	Negative predictive (**NPV**) = Σ True negative/ Σ Test outcome negative
Positive likelihood ratio **LR+**=TPR/FPR		True positive rate (**TPR**, sensitivity) = Σ True positive/ Σ Condition positive	False positive rate (**FPR**, Fall-out) = Σ False positive/ Σ Condition negative	Accuracy (ACC) = Σ True positive + Σ True negative/ Σ Total population	
Negative likelihood ratio **LR−**=FNR/TNR		False negative rate (**FNR**) = Σ False negative/ Σ Condition positive	True negative rate (**TNR**, Specificity) = Σ True negative/ Σ Condition negative		

According to confusion matrix definition, we can see the difference of results between when k = 1 and when k = 5. Citation: https://blog.csdn.net/vesper305/article/details/44927047

Obviously, the overall true positive (Osmancik is correctly classified as Osmancik) and true negative (Cammeo is correctly classified as Cammeo) is higher in the results when k = 5 than in the results when k = 1; meanwhile, the overall false positive(Osmancik is wrongly classified as Cammeo) and false negative(Cammeo is wrongly classified as Osmancik) is lower in the results when k = 5 than in the results when k = 1. As a result, we can see that the consitency of knn when k=5 is better.

Perform any necessary data cleaning (e.g. you'll probably want to get rid of the numbers in e.g. "Connecticut(7)" which refer to data source information as well as remove lines that aren't part of the table). Include the cleaned CSV file in your homework submission, and make sure your readme includes a citation of where the original data came from and how you changed the csv file. (5 points)

In [1]:
```
import pandas as pd
import numpy as np
import csv
```

In [1]:
```
#select the rows with over 10 elements using csv lib.
with open('./new_incd.csv','w',encoding='utf8',newline='') as writer:
```

```
    csv_writer = csv.writer(writer,)
    with open('./incd.csv',newline='') as f:
        data = csv.reader(f)
    # data = pd.read_csv('./incd.csv')
        for i in data:
            if len(i) > 10: #The useful data begins at row 10.
                if i[0][-3]=='(' and i[0][-1] == ')':
                    i[0] = i[0][:-3]
                csv_writer.writerow(i)
#column names need to be cleaned by hands
```

Citation: The original data came from "https://statecancerprofiles.cancer.gov/incidencerates/index.php?stateFIPS=00&areatype=state&cancer=001&race=00&sex=0&age=001&stage=999&year=0&type=incd&sortVariableName=nan 1)download the data from above address 2)create a new .csv file to save the cleaned data: "new_incd.csv" file 3)set a number 10 to select the rows, because the useful data has over 10 dims 4)get rid of the numbers in e.g."Connecticut(7)": manually delete numbers, '(', and ')'. 5)manually organize the names of the columns.

In [1]:

```
from flask import Flask,redirect,url_for,request,send_file,render_template
import pandas as pd
data = pd.read_csv('./new_incd.csv')
temp_states= list(data['State'].str.upper())
for i in range(len(temp_states)):
    temp_states[i] = temp_states[i].replace(" ","")

app = Flask(__name__, template_folder='./templates')

@app.route('/')
def index():
    return render_template('homepage.html')

@app.route('/state/<string:name>')
def search_state(name):
    state_name = name.replace(" ","").upper()
    if state_name.upper() in temp_states:
        index = temp_states.index(state_name.upper())
```

```
            aair = data.iloc[index,:]["Age-Adjusted Incidence rate (cases per 100K)"]
            return {"State":str(data.loc[index,'State']),"Age-Adjusted Incidence rate (cases per 100K)":
                    str(aair)}
        else:
            return render_template('errorpage.html'),404

@app.route('/info',methods=["GET"])
def info():
    if request.method == 'GET':
        if request.args.get('state_name'):
            ori_state_name = request.args.get('state_name')
            state_name = ori_state_name.replace(" ","").upper()
            if state_name.upper() in temp_states:
                index = temp_states.index(state_name.upper())
                aair = data.iloc[index,:]["Age-Adjusted Incidence rate (cases per 100K)"]
                return {"State":str(data.loc[index,'State']),"Age-Adjusted Incidence rate (cases per 10(
                        str(aair)}
            else:
                return render_template('errorpage.html'),404

        else:
            return render_template('errorpage.html'),404
    else:
        return render_template('errorpage.html'),404
    return render_template('errorpage.html'),404
if __name__=='__main__':
    app.run(host="localhost")

#Readme
# 1)include two .html files (homepage.html and errorpage.html)
# 2)Searching for state names is not case-sensitive and space-insensitive
# 3)test:
# http://localhost:5000/
# return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"}

# http://localhost:5000/state/Kentucky
# return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"}

# http://localhost:5000/info?state_name=Kentucky
```

```
# http://localhost:5000/info?state_name=Kentucky
# return {"Age-Adjusted Incidence rate (cases per 100K)":"517.8","State":"Kentucky"}
```

```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://localhost:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [01/Dec/2021 15:13:31] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [01/Dec/2021 15:13:54] "GET /info?state_name=New%20York HTTP/1.1" 200 -
127.0.0.1 - - [01/Dec/2021 15:13:54] "GET /?state_name=New+York HTTP/1.1" 200 -
127.0.0.1 - - [01/Dec/2021 15:14:56] "GET /?state_name=kk HTTP/1.1" 200 -
127.0.0.1 - - [01/Dec/2021 15:14:56] "GET /info?state_name=kk HTTP/1.1" 404 -
127.0.0.1 - - [01/Dec/2021 15:15:00] "GET /info?state_name=nl HTTP/1.1" 404 -
127.0.0.1 - - [01/Dec/2021 15:15:00] "GET /?state_name=nl HTTP/1.1" 200 -
```

You've now completed most of this course, so you're now qualified to choose the next step. Take this exercise one step beyond that which is described above in a way that you think is appropriate, and discuss your extension in your readme. (e.g.

	State	FIPS	Met Healthy People Objective of ***?	Age-Adjusted Incidence rate (
2	US (SEER+NPCR)	0	***	448.6
3	Kentucky	21000	***	517.8
4	New Jersey	34000	***	486.7
5	Iowa	19000	***	484.1
6	West Virginia	54000	***	483.5
7	New York	36000	***	483.1
8	Louisiana	22000	***	482.4
9	Pennsylvania	42000	***	480
10	Delaware	10000	***	479.6
11	New Hampshire	33000	***	479.3
12	Arkansas	5000	***	479
13	Maine	23000	***	476
14	Mississippi	28000	***	474.4
15	Rhode Island	44000	***	472.8
16	Minnesota	27000	***	469.5

17	North Carolina	37000	***	468.9
18	Georgia	13000	***	468.5
19	Wisconsin	55000	***	468.5
20	Nebraska	31000	***	467.7
21	Ohio	39000	***	467.5
22	Illinois	17000	***	466.8
23	Tennessee	47000	***	466
24	Connecticut	9000	***	465.1
25	Montana	30000	***	462.9
26	Florida	12000	***	460.2
27	South Dakota	46000	***	459.1
28	Indiana	18000	***	457.9
29	Kansas	20000	***	457.8
30	Massachusetts	25000	***	457.8
31	Vermont	50000	***	457.4
32	North Dakota	38000	***	453.2
33	Maryland	24000	***	452.5
34	Missouri	29000	***	452.3
35	Alabama	1000	***	450.8
36	Oklahoma	40000	***	450.2
37	South Carolina	45000	***	449.6
38	Michigan	26000	***	448.8
39	Idaho	16000	***	445.1
40	Washington	53000	***	442.4
41	Oregon	41000	***	430.5
42	District of Columbia	11001	***	426
43	Alaska	2900	***	418.6
44	Hawaii	15000	***	415.5
45	Texas	48000	***	411.2
46	Virginia	51000	***	411

47	Utah	49000	***	405.4
48	California	6000	***	402.4
49	Wyoming	56000	***	402.1
50	Nevada	32000	***	398.8
51	Colorado	8000	***	396.6
52	Arizona	4000	***	385.7
53	New Mexico	35000	***	373.2
54	Puerto Rico	72001	***	356.3

YCKellyLiu / **BIS634** Private

Code Issues Pull requests Actions Projects Security Insights Settings

main **BIS634** / HW5_YuechenLiu / **HW5Q2.ipynb** Go to file ···

YCKellyLiu Add files via upload Latest commit 338d537 21 hours ago **History**

1 contributor

355 lines (355 sloc) | 66.5 KB Raw Blame

In [7]:

```
class Tree:
    def __init__(self, value=None):
        self.value = value
        self.left = None
        self.right = None

    def add(self, item):
        if self.value is None:
            self.value = item
        elif item < self.value:
            if self.left:
                self.left.add(item)
            else:
```

```
                self.left = Tree(item)
        elif item > self.value:
            if self.right:
                self.right.add(item)
            else:
                self.right = Tree(item)

    def __contains__(self, item):
        if self.value == item:
            return True
        elif self.left and item < self.value:
            return item in self.left
        elif self.right and item > self.value:
            return item in self.right
        else:
            return False
```

Replace the contains method of the tree with the following **contains** method. The change in name will allow you to use the in operator; e.g. after this change, 55 in my_tree should be True in the above example, whereas 42 in my_tree would be False. Test this.

In [8]:

```
import random
my_tree = Tree()
test_tree = [55, 62, 37, 49, 71, 14, 17]
for item in test_tree:
    my_tree.add(item)
print(55 in my_tree)
print(42 in my_tree)
print(71 in my_tree)
print(4 in my_tree)
```

```
True
False
True
False
```

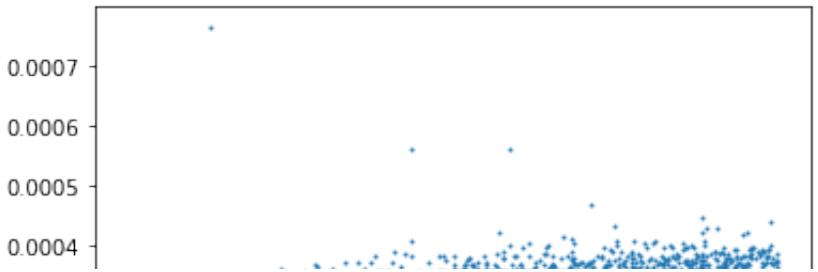
Using various sizes n of trees (populated with random data) and sufficiently many calls to in (each individual call should be

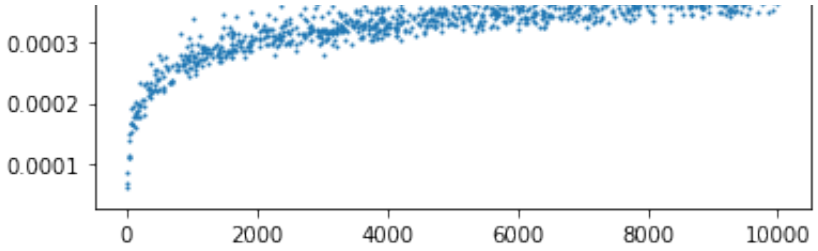
very fast, so you may have to run many repeated tests), demonstrate that in is executing in O(log n) times; on a log-log plot, for sufficiently large n, the graph of time required for checking if a number is in the tree as a function of n should be almost horizontal. (5 points).

In [4]:
```
import random
import time
import matplotlib.pyplot as plt
```

In [9]:
```
time_per_test = []
n_samples = 1000
repeat_times = 100
for _ in range(n_samples):
    n = random.randint(0, 10000)
    my_tree = Tree()
    for _ in range(n):
        my_tree.add(random.random())
    start = time.time()
    for _ in range(repeat_times):
        my_tree.__contains__(random.random())
    end = time.time()
    time_per_test.append((end - start, n))
```

In [11]:
```
plt.scatter([x[1] for x in time_per_test], [x[0] for x in time_per_test], s=1)
plt.show()
```





As shown in the above plot, the in is executing in O(log n) times.

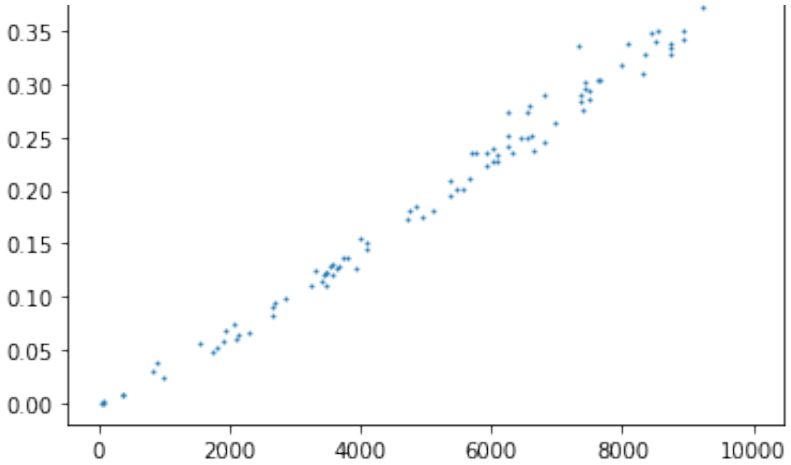
This speed is not free. Provide supporting evidence that the time to setup the tree is O(n log n) by timing it for various sized ns and showing that the runtime lies between a curve that is O(n) and one that is O(n**2). (5 points)

This way is to draw two graphs of extream situation,O(n) and O(n**2), which means that the runtime must lie between those two curves.

If the input secquence is random enough, the tree is almost balanced and the setup time is O(n), as shown below

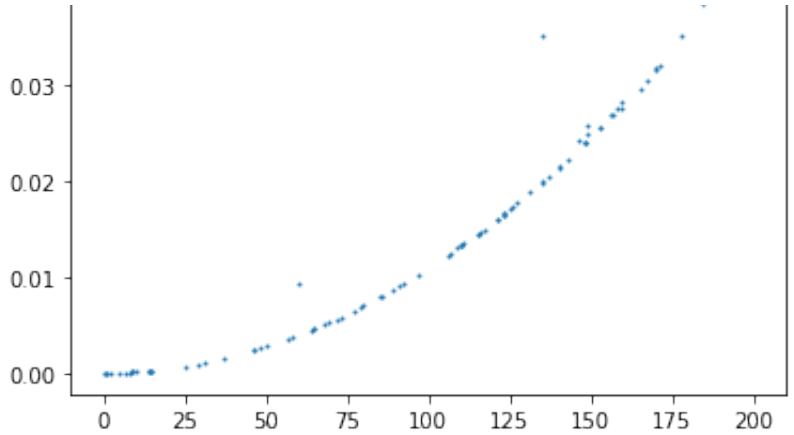
In [12]:

```
time_per_test = []
n_samples = 100
repeat_times = 10
for _ in range(n_samples):
    n = random.randint(0, 10000)
    secquence = [random.random() for _ in range(n)]
    start = time.time()
    for _ in range(repeat_times):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    time_per_test.append((end - start, n))

plt.scatter([x[1] for x in time_per_test], [x[0] for x in time_per_test], s=1)
plt.show()
```



However, if the input is inversely sorted, than the setup time is O(n**2)

In [13]:
```
time_per_test = []
n_samples = 100
repeat_times = 10
for _ in range(n_samples):
    n = random.randint(0, 200)
    secquence = [random.random() for _ in range(n)]
    secquence = sorted(secquence)
    start = time.time()
    for _ in range(repeat_times):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    time_per_test.append((end - start, n))

plt.scatter([x[1] for x in time_per_test], [x[0] for x in time_per_test], s=1)
plt.show()
```

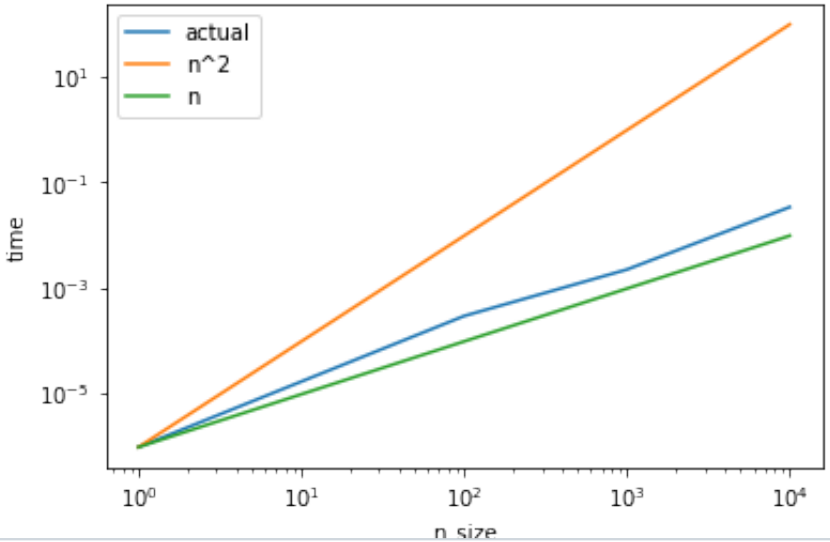


If I create a function and put actual time spend, n**2, and n in one graph.

In [13]:
```
def timefunc(n):
    times = []
    n = [random.randint(1,n) for _ in range(n)]
    for attempt in [i for i in range(100)]:
        start = time.time
        my_tree = Tree()
        for item in n:
            my_tree.add(item)
        end = time.time()
        times.append(end - start)
    return min(times)
```

In [14]:
```
ns = [1,100,1000,10000] #Select different n size
actual = [timefunc(n) for n in ns]
s = actual [0]
quadratic = [s*n**2 for n in ns]
linear = [s*n for n in ns]
plt.plot(ns,actual)
plt.plot(ns,quadratic)
plt.plot(ns,linear)
```

```
plt.xscale('log')
plt.yscale('log')
plt.ylabel('time')
plt.xlabel('n_size')
plt.legend(['actual','n^2','n'],loc = 'upper left')
plt.show()
```



In [1]:
```
import pandas as pd
from sklearn import decomposition
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_score, cross_val_predict, cross_validate
```

In [2]:
```
data = pd.read_excel('/Users/ycliu/Desktop/Rice_Osmancik_Cammeo_Dataset.xlsx', engine='openpyxl')
```

Normalize the data

In [3]:
```
for c in data.columns[:-1]:
    data[c] = (data[c] - np.mean(data[c])) / np.std(data[c])
```

In [4]:
```
data.head()
```

Out[4]:

	AREA	PERIMETER	MAJORAXIS	MINORAXIS	ECCENTRICITY	CONVEX_AREA	EXTENT	CLASS
0	1.479830	2.004354	2.348547	-0.212943	2.018337	1.499659	-1.152921	Cammeo
1	1.147870	1.125853	0.988390	0.945568	0.410018	1.192918	-0.602079	Cammeo
2	1.135169	1.317214	1.451908	0.253887	1.212956	1.126504	0.405611	Cammeo
3	0.293436	0.115300	0.261439	0.198051	0.239751	0.233857	-0.275351	Cammeo
4	1.166345	1.487053	1.316442	0.523419	0.952221	1.299855	-0.206013	Cammeo

In [5]:
```
data.value_counts('CLASS')
```

Out[5]:
```
CLASS
Osmancik    2180
Cammeo      1630
dtype: int64
```

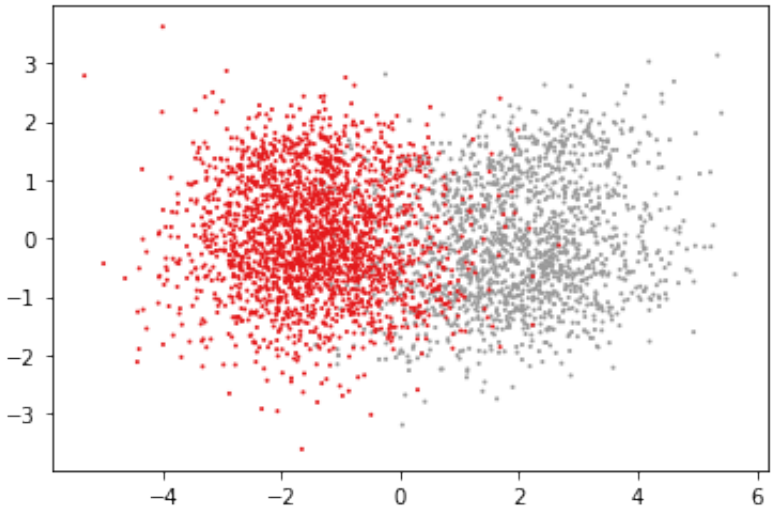
PCA

In [6]:
```
pca = decomposition.PCA(n_components=2)
data_reduced = pca.fit_transform(data[[
    'AREA',
    'PERIMETER',
    'MAJORAXIS',
    'ECCENTRICITY',
```

```
    'CONVEX_AREA',
    'EXTENT'
    ]])
pc0 = data_reduced[:, 0]
pc1 = data_reduced[:, 1]
```

In [7]:

```
plt.scatter(pc0, pc1,
    c=data.CLASS.map({'Osmancik': 0, 'Cammeo': 1}),
    s=1, cmap='Set1')
plt.show()
```



The PCA model is able to reduce the dimensionality of the data to 2. The gaussion distance of two samples seems to be small if they belong to the same class.

Implementation of the KNN algorithm and Quad Tree

Citation: https://cloud.tencent.com/developer/article/1487842 https://scipython.com/blog/quadtrees-2-implementation-in-python/ https://jrtechs.net/data-science/implementing-a-quadtree-in-python

In [8]:

```
class Point: #the points in the training set
    def __init__(self, x, y, payload=None):
        self.x, self.y = x, y
        self.payload = payload

    def distance_to(self, other):
        try:
            other_x, other_y = other.x, other.y
        except AttributeError:
            other_x, other_y = other
        return np.hypot(self.x - other_x, self.y - other_y)

class Rect: #Rectangular store the shape and other attribute of the quadtree
    #cx: center point x-axis
    #cy: center point y-axis
    #w: the width of the rectangular
    #h: the height of the rectangular
    def __init__(self, cx, cy, w, h):
        self.cx, self.cy = cx, cy
        self.w, self.h = w, h
        self.west_edge, self.east_edge = cx - w/2, cx + w/2
        self.north_edge, self.south_edge = cy - h/2, cy + h/2

    def contains(self, point): #validate if a point is inside of the rectangular
        try:
            point_x, point_y = point.x, point.y
        except AttributeError:
            point_x, point_y = point

        return (point_x >= self.west_edge and
                point_x <  self.east_edge and
                point_y >= self.north_edge and
                point_y < self.south_edge)

    def intersects(self, other):#validate if the rectangular is in the center and the radius
        return not (other.west_edge > self.east_edge or
```

```
                    other.east_edge < self.west_edge or
                    other.north_edge > self.south_edge or
                    other.south_edge < self.north_edge)

class QuadTree:
    #A class implementing a quadtree.
    #boundary: the rectangular that store the shape and cneter
    #depth: the max number of points can be stored in this quad tree

    def __init__(self, boundary, max_points=4, depth=0):
        self.boundary = boundary
        self.max_points = max_points
        self.points = []
        self.depth = depth
        self.divided = False

    def divide(self): #divide the quad tree into 4 sub quadtree
        cx, cy = self.boundary.cx, self.boundary.cy
        w, h = self.boundary.w / 2, self.boundary.h / 2
        self.nw = QuadTree(Rect(cx - w/2, cy - h/2, w, h),
                                    self.max_points, self.depth + 1)
        self.ne = QuadTree(Rect(cx + w/2, cy - h/2, w, h),
                                    self.max_points, self.depth + 1)
        self.se = QuadTree(Rect(cx + w/2, cy + h/2, w, h),
                                    self.max_points, self.depth + 1)
        self.sw = QuadTree(Rect(cx - w/2, cy + h/2, w, h),
                                    self.max_points, self.depth + 1)
        self.divided = True

    def insert(self, point): #insert the points into the sub quadtree.
        if not self.boundary.contains(point):
            # The point does not lie inside boundary: bail.
            return False
        if len(self.points) < self.max_points:
            # There's room for our point without dividing the QuadTree.
            self.points.append(point)
            return True
```

```
        # No room: divide if necessary, then try the sub-quads.
        if not self.divided:
            self.divide()

        return (self.ne.insert(point) or
                self.nw.insert(point) or
                self.se.insert(point) or
                self.sw.insert(point))

    def query_circle(self, boundary, centre, radius, found_points):
        """Find the points in the quadtree that lie within radius of centre.

        boundary is a Rect object (a square) that bounds the search circle.
        There is no need to call this method directly: use query_radius.

        """

        if not self.boundary.intersects(boundary):
            # If the domain of this node does not intersect the search
            # region, we don't need to look in it for points.
            return False

        # Search this node's points to see if they lie within boundary
        # and also lie within a circle of given radius around the centre point.
        for point in self.points:
            if (boundary.contains(point) and
                    point.distance_to(centre) <= radius):
                found_points.append(point)

        # Recurse the search into this node's children.
        if self.divided:
            self.nw.query_circle(boundary, centre, radius, found_points)
            self.ne.query_circle(boundary, centre, radius, found_points)
            self.se.query_circle(boundary, centre, radius, found_points)
            self.sw.query_circle(boundary, centre, radius, found_points)
        return found_points

    def query_radius(self, centre, radius, found_points):
        """Find the points in the quadtree that lie within radius of centre."""
```

```
            Find the points in the quadtree that lie within radius of centre.

        boundary = Rect(*centre, 2*radius, 2*radius)
        return self.query_circle(boundary, centre, radius, found_points)


    def __len__(self):
        npoints = len(self.points)
        if self.divided:
            npoints += len(self.nw)+len(self.ne)+len(self.se)+len(self.sw)
        return npoints
```

In [9]:

```
class KNN(): #K Neaarest Neighbor algorithm
    def __init__(self, k = 1) -> None:
        self.qtree = QuadTree(Rect(0, 0, 25, 25))
        self.k = k
        self.pca = decomposition.PCA(n_components=2)

    def fit(self, X, y):
        X = self.pca.fit_transform(X)
        for _x, _y in zip(X, y):
            self.qtree.insert(Point(_x[0], _x[1], _y))

    def predict(self, X):
        k = self.k
        pred = []
        # binary search for radius
        r_max = 5
        r_min = 0
        while True:
            found_points = []
            r = (r_max + r_min) / 2
            self.qtree.query_radius((X[0], X[1]), r, found_points)
            if len(found_points) > k:
                r_max = r
            elif len(found_points) < k:
                r_min = r
```

```
            else:
                break
        for p in found_points:
            pred.append(p.payload)
        return np.bincount(pred).argmax()

    def score(self, X, y):
        X = self.pca.transform(X)
        pred = []
        for x in X:
            pred.append(self.predict(x))
        pred = np.array(pred)
        y = np.array(y)
        return confusion_matrix(y, pred)

    def get_params(self, *args, **kwargs):
        return dict()
```

In [10]:

```
_data = data.sample(frac=1)
X = _data[[
    'AREA',
    'PERIMETER',
    'MAJORAXIS',
    'ECCENTRICITY',
    'CONVEX_AREA',
    'EXTENT'
    ]]
Y = _data.CLASS.map({'Osmancik': 0, 'Cammeo': 1})
```

When k = 1:

In [11]:

```
for i in range(0, len(X), int(len(X)/5)):
    i = int(i)
    x_test = X.iloc[i: i+int(len(X)/5)]
    x_train = X.drop(x_test.index)
    y_test = Y[x_test.index]
```

```
    y_train = Y.drop(x_test.index)
    knn = KNN(k=1)
    knn.fit(x_train, y_train)
    print(knn.score(x_test, y_test))
```

```
[[386  51]
 [ 53 272]]
[[381  45]
 [ 47 289]]
[[417  34]
 [ 35 276]]
[[377  56]
 [ 46 283]]
[[394  39]
 [ 49 280]]
```

When k = 5:

In [12]:

```
for i in range(0, len(X), int(len(X)/5)):
    i = int(i)
    x_test = X.iloc[i: i+int(len(X)/5)]
    x_train = X.drop(x_test.index)
    y_test = Y[x_test.index]
    y_train = Y.drop(x_test.index)
    knn = KNN(k=5)
    knn.fit(x_train, y_train)
    print(knn.score(x_test, y_test))
```

```
[[399  38]
 [ 31 294]]
[[396  30]
 [ 28 308]]
[[427  24]
 [ 28 283]]
[[402  31]
 [ 37 292]]
[[406  27]
```

```
 [ 47 282]]
```

According to confusion matrix definition, we can see the difference of results between when k = 1 and when k = 5. Citation: https://blog.csdn.net/vesper305/article/details/44927047

Obviously, the overall true positive (Osmancik is correctly classified as Osmancik) and true negative (Cammeo is correctly classified as Cammeo) is higher in the results when k = 5 than in the results when k = 1; meanwhile, the overall false positive(Osmancik is wrongly classified as Cammeo) and false negative(Cammeo is wrongly classified as Osmancik) is lower in the results when k = 5 than in the results when k = 1.

From the result we can see that the consitency of knn when k=5 is better.