worksheet_12

March 18, 2024

1 Worksheet 12

Name: Daniyal Ahmed UID: U11469883

1.0.1 Topics

- Introduction to Classification
- K Nearest Neighbors

1.0.2 Introduction to Classification

a) For the following examples, say whether they are or aren't an example of classification.

1. Predicting whether a student will be offered a job after graduating given their GPA.
2. Predicting how long it will take (in number of months) for a student to be offered a job after graduating, given their GPA.
3. Predicting the number of stars (1-5) a person will assign in their yelp review given the description they wrote in the review.
4. Predicting the number of births occuring in a specified minute.

1 and 3 are classifcations since in number 3 the label can be the number of stars while in number 1 the labels can be whether they got the job or not.

b) Given a dataset, how would you set things up such that you can both learn a model and get an idea of how this model might perform on data it has never seen?

You would use Instance-Based Classifiers on this model, since you can used stored data to see how the model reacts on unseen data

c) In your own words, briefly explain:

- underfitting
- overfitting

and what signs to look out for for each.

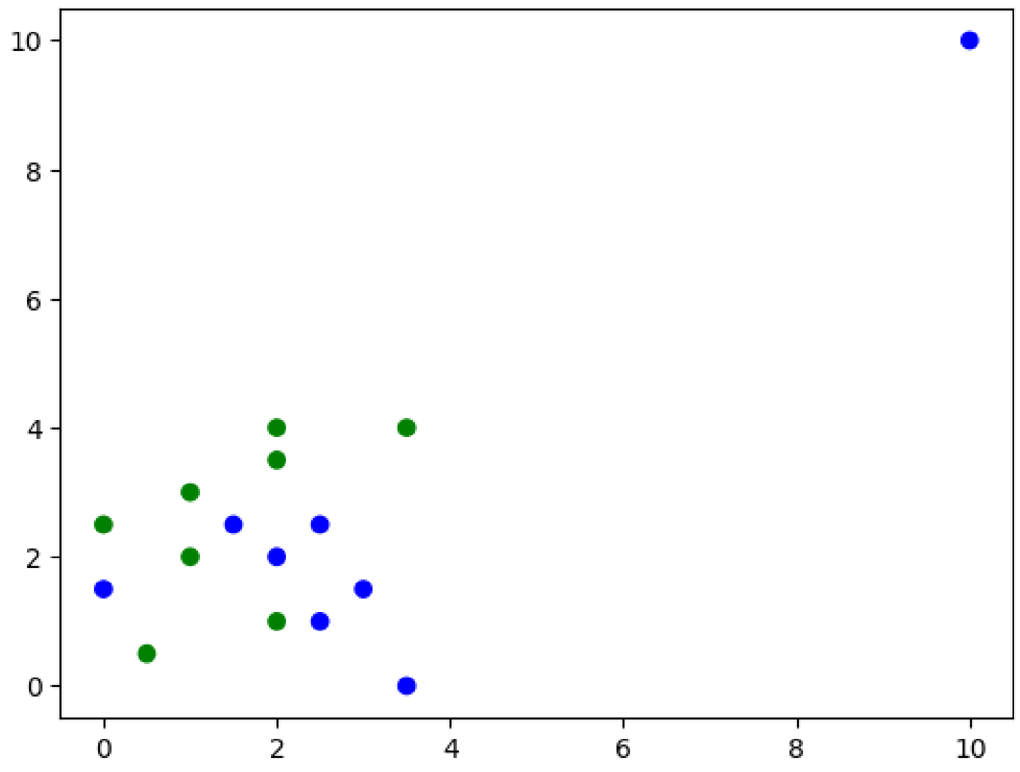
Underfitting the Dataset usually happens whens model has not analyzed the data enough or learned enough from a data. Basically when a model is underfitted it is not following the data set closely enough. While when a model is overfitted, it is following the data set too closely and not predicting what some labels can be rather just looking at the data set for any thing that looks similar enouugh. In conclusion Overfitting is when the model references the dataset too much but Underfitting is when it doesn't reference it enough

1.0.3 K Nearest Neighbors

```
[1]: import numpy as np
     import matplotlib.pyplot as plt

     data = {
         "Attribute A" : [3.5, 0, 1, 2.5, 2, 1.5, 2, 3.5, 1, 3, 2, 2, 2.5, 0.5, 0.,␣
      ↪10],
         "Attribute B" : [4, 1.5, 2, 1, 3.5, 2.5, 1, 0, 3, 1.5, 4, 2, 2.5, 0.5, 2.5,␣
      ↪10],
         "Class" : [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0],
     }
```

a) Plot the data in a 2D plot coloring each scatter point one of two colors depending on its corresponding class.

```
[2]: colors = np.array([x for x in 'bgrcmyk'])
     plt.scatter(data["Attribute A"], data["Attribute B"],␣
      ↪color=colors[data["Class"]].tolist())
     plt.show()
```



Outliers are points that lie far from the rest of the data. They are not necessarily invalid points however. Imagine sampling from a Normal Distribution with mean 10 and variance 1. You would expect most points you sample to be in the range [7, 13] but it's entirely possible to see 20 which, on average, should be very far from the rest of the points in the sample (unless we're VERY (un)lucky). These outliers can inhibit our ability to learn general patterns in the data since they are not representative of likely outcomes. They can still be useful in of themselves and can be analyzed in great depth depending on the problem at hand.

b) Are there any points in the dataset that could be outliers? If so, please remove them from the dataset.

Yes there is one outlier in the data set is at point 10 10

[3]:
```
data["Attribute A"].remove(10)
data["Attribute B"].remove(10)
data["Class"].remove(0)
print(data)
```

```
{'Attribute A': [3.5, 0, 1, 2.5, 2, 1.5, 2, 3.5, 1, 3, 2, 2, 2.5, 0.5, 0.0],
'Attribute B': [4, 1.5, 2, 1, 3.5, 2.5, 1, 0, 3, 1.5, 4, 2, 2.5, 0.5, 2.5],
'Class': [1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}
```

Noise points are points that could be considered invalid under the general trend in the data. These could be the result of actual errors in the data or randomness that we could attribute to oversimplification (for example if missing some information / feature about each point). Considering noise points in our model can often lead to overfitting.

c) Are there any points in the dataset that could be noise points?

Yes, the Point at (3.5,4) and the point at (3.5,0)

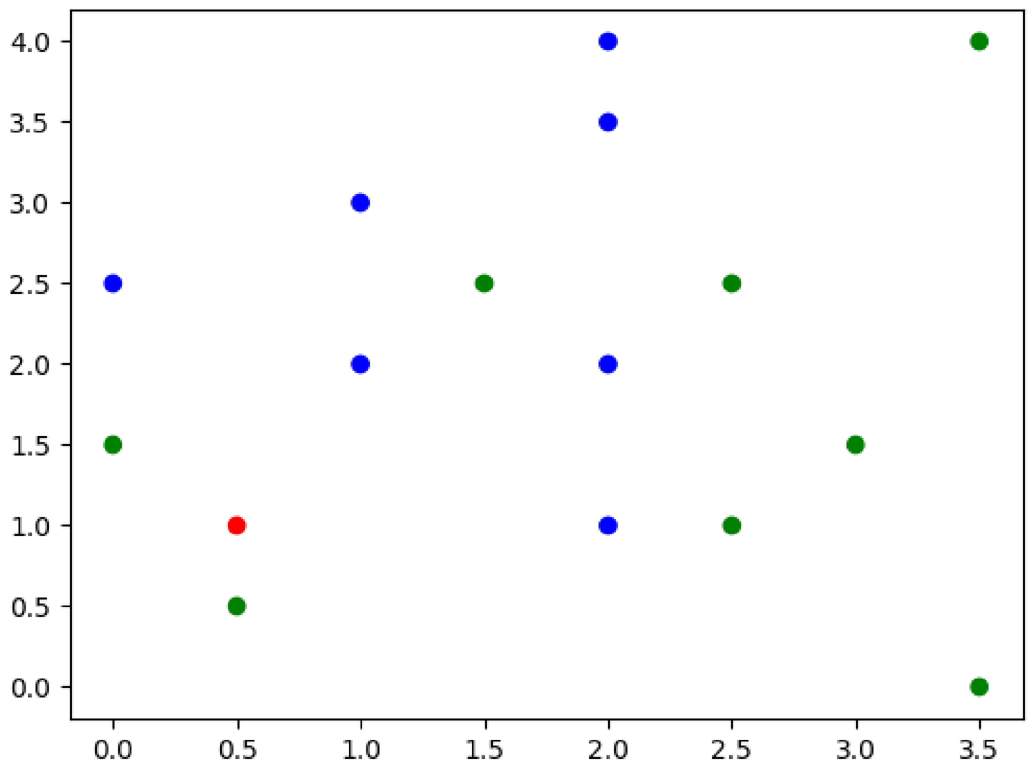
For the following point

A	B
0.5	1

d) Plot it in a different color along with the rest of the points in the dataset.

[4]:
```
data["Attribute A"].append(.5)
data["Attribute B"].append(1.0)
data["Class"].append(2)
print(len(data["Attribute A"]))
print(len(data["Attribute B"]))
print(len(data["Class"]))
colors2 = np.array([x for x in 'bgrcmyk'])

plt.scatter(data["Attribute A"], data["Attribute B"],␣
 ↪color=colors2[data["Class"]].tolist())
plt.show()
```

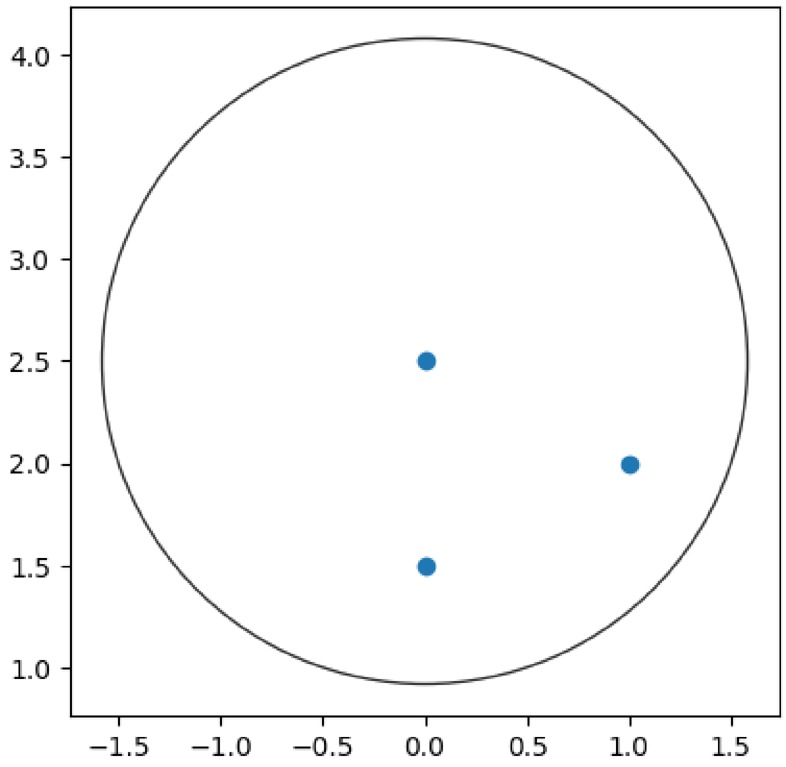
```
16
16
16
```



e) Write a function to compute the Euclidean distance from it to all points in the dataset and pick the 3 closest points to it. In a scatter plot, draw a circle centered around the point with radius the distance of the farthest of the three points.

[5]:
```python
def n_closest_to(example, n):


    distances = []
    #Assuming n is our dataset
    x=n["Attribute A"]
    y=n["Attribute B"]
    for i in range(len(x) ):
         if(x[i] != example[0] and y[i] != example[1]):
            distances.append([np.sqrt((example[0]-x[i])**2 +␣
 ↪(example[1]-y[i])**2),(x[i],y[i])])

    minimum = []


    #Find the three closest points by the minimum distance
    minimum.append(min(distances , key = lambda x: x[0]))
    distances.remove(minimum[0])
    minimum.append(min(distances , key = lambda x: x[0]))
    distances.remove(minimum[1])
    minimum.append(min(distances , key = lambda x: x[0]))
    distances.remove(minimum[2])

    print(minimum)
    return minimum



minimum = n_closest_to((.5,1), (data) )
max_point = max(minimum, key = lambda x: x[0])
print(max_point)
location = max_point[1]

radius = max_point[0]
_, axes = plt.subplots()
x, y = zip(minimum[0][1], minimum[1][1], minimum[2][1])

axes.scatter(x,y)
cir = plt.Circle(location, radius, fill = False, alpha=0.8)
axes.add_patch(cir)
axes.set_aspect('equal') # necessary so that the circle is not oval
plt.show()
```

```
[[0.7071067811865476, (0, 1.5)], [1.118033988749895, (1, 2)],
[1.5811388300841898, (0.0, 2.5)]]
[1.5811388300841898, (0.0, 2.5)]
```



f) Write a function that takes the three points returned by your function in e) and returns the class that the majority of points have (break ties with a deterministic default class of your choosing). Print the class assigned to this new point by your function.

[41]:
```python
def majority(points):
    count = 0
    count2 = 0
    for i in range(0,len(points)):
        if(data["Class"][findIndex(data,points[i][1])]==1):
            count+=1
        if(data["Class"][findIndex(data,points[i][1])]==0):
            count2+=1

    if(count>count2):
        #data["Class"][-1]=1
        print("Assigning 1")
        return 1
    if(count2>count):
        #data["Class"][-1]=0
        print("Assinging 0")
        return 0
    if(count == count2):
        #data["Class"][-1]=1
        print("Assigning 1")
        return 1

#Small function I wrote to find the index of the point in the data set
def findIndex(data,points):
    for i in range(len(data["Attribute A"])):
        if(data["Attribute A"][i] == points[0] and data["Attribute B"][i] ==␣
 ↪points[1]):
            return i


data["Class"][-1] = majority(minimum)
```

```
Assinging 0
```

g) Re-using the functions from e) and f), you should be able to assign a class to any new point. In this exercise we will implement Leave-one-out cross validiation in order to evaluate the performance of our model.

For each point in the dataset:

- consider that point as your test set and the rest of the data as your training set
- classify that point using the training set
- keep track of whether you were correct with the use of a counter

Once you've iterated through the entire dataset, divide the counter by the number of points in the dataset to report an overall testing accuracy.

[42]:
```
import copy

count = 0
for i in range(len(data["Attribute A"])):

    training_set ={}
    actual_class = data["Class"][i]


    #Making deep copy of the original data and removing the i-th element so we␣
 ↪can use it as a training set hence we are generally just removing the the␣
 ↪test set
    training_set = {
        "Attribute A" : copy.deepcopy(data["Attribute A"]) ,
        "Attribute B" : copy.deepcopy(data["Attribute B"]) ,
        "Class" : copy.deepcopy(data["Class"]),
    }

    #removing that value from the training set
    training_set["Attribute A"].pop(i)
    training_set["Attribute B"].pop(i)
    training_set["Class"].pop(i)

    point = (data["Attribute A"][i], data["Attribute B"][i])

    #Running it through the two functions we prevuiously created
    prediction = majority(n_closest_to(point, training_set))
    if prediction == actual_class:
        count += 1

print("overall accuracy = ", count/len(data["Attribute A"]))
```

```
[[1.5811388300841898, (2, 3.5)], [1.8027756377319946, (2.5, 2.5)], [2.5, (1.5,
2.5)]]
Assigning 1
[[0.7071067811865476, (0.5, 1.0)], [1.118033988749895, (1, 2)],
[1.118033988749895, (0.5, 0.5)]]
Assinging 0
[[0.7071067811865476, (1.5, 2.5)], [1.118033988749895, (0, 1.5)],
[1.118033988749895, (0.0, 2.5)]]
Assigning 1
[[0.7071067811865476, (3, 1.5)], [1.118033988749895, (2, 2)],
[1.4142135623730951, (3.5, 0)]]
Assigning 1
[[1.118033988749895, (1.5, 2.5)], [1.118033988749895, (1, 3)],
[1.118033988749895, (2.5, 2.5)]]
Assigning 1
[[0.7071067811865476, (1, 2)], [0.7071067811865476, (1, 3)],
[0.7071067811865476, (2, 2)]]
Assinging 0
[[1.118033988749895, (3, 1.5)], [1.4142135623730951, (1, 2)],
[1.5811388300841898, (1.5, 2.5)]]
Assigning 1
[[1.4142135623730951, (2.5, 1)], [1.5811388300841898, (3, 1.5)],
[1.8027756377319946, (2, 1)]]
Assigning 1
[[0.7071067811865476, (1.5, 2.5)], [1.118033988749895, (2, 3.5)],
[1.118033988749895, (0.0, 2.5)]]
Assinging 0
[[0.7071067811865476, (2.5, 1)], [1.118033988749895, (2, 1)],
[1.118033988749895, (2, 2)]]
Assinging 0
[[1.4142135623730951, (1, 3)], [1.5811388300841898, (1.5, 2.5)],
[1.5811388300841898, (2.5, 2.5)]]
Assigning 1
[[0.7071067811865476, (1.5, 2.5)], [0.7071067811865476, (2.5, 2.5)],
[1.118033988749895, (2.5, 1)]]
Assigning 1
[[0.7071067811865476, (2, 2)], [1.118033988749895, (2, 3.5)],
[1.118033988749895, (3, 1.5)]]
Assinging 0
[[1.118033988749895, (0, 1.5)], [1.5811388300841898, (1, 2)],
[1.5811388300841898, (2, 1)]]
Assinging 0
[[1.118033988749895, (1, 2)], [1.118033988749895, (1, 3)], [1.5811388300841898,
(0.5, 1.0)]]
Assinging 0
[[0.7071067811865476, (0, 1.5)], [1.118033988749895, (1, 2)],
[1.5811388300841898, (0.0, 2.5)]]
Assinging 0
overall accuracy =  0.375
```

1.1 Challenge Problem

For this question we will re-use the "mnist_784" dataset.

a) Begin by creating a training and testing datasest from our dataset, with a 80-20 ratio, and random_state=1. You can use the `train_test_split` function from sklearn. By holding out a portion of the dataset we can evaluate how our model generalizes to unseen data (i.e. data it did not learn from).

[8]:
```python
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml


#Fetching the dataset
X, y = fetch_openml(name="mnist_784", version=1, return_X_y=True,
as_frame=False)


#Splitting the dataset on Test and train data
X_train, X_test, Y_train, Y_test = train_test_split(X, y, train_size=.8,␣
↪test_size=0.2, random_state=1)

print(X_train)
```

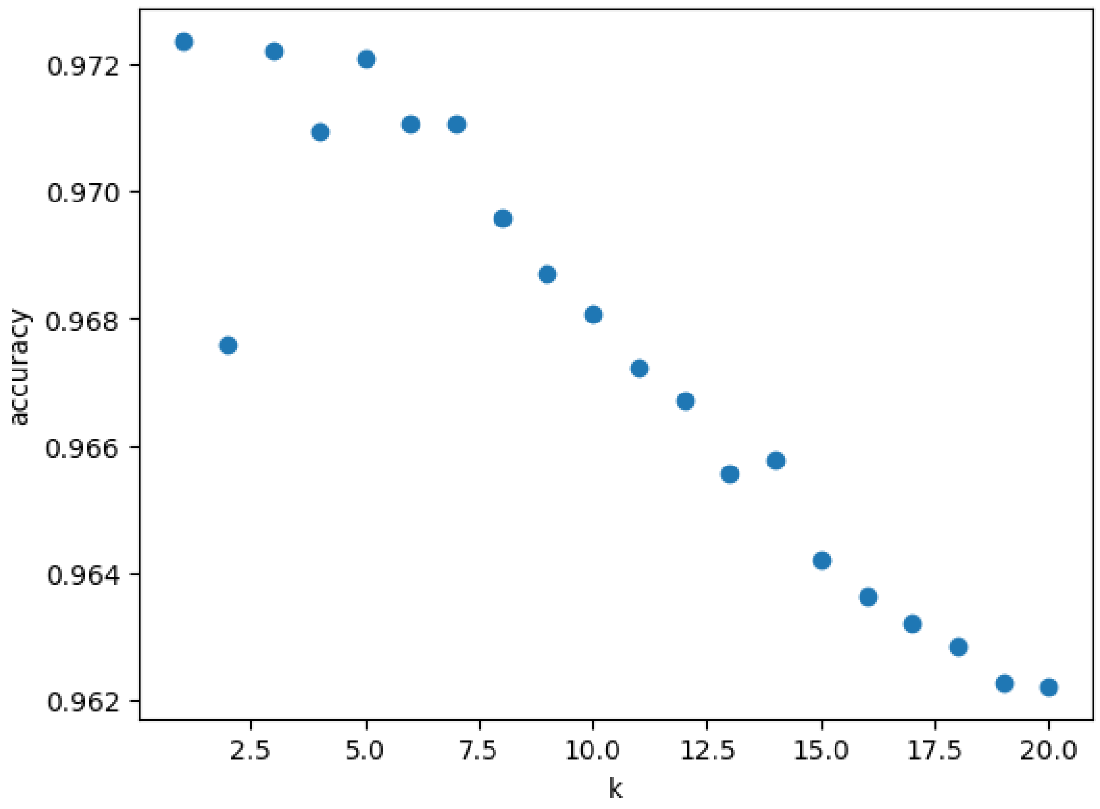
```
C:\Users\eggsc\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n
2kfra8p0\LocalCache\local-packages\Python310\site-
packages\sklearn\datasets\_openml.py:1002: FutureWarning: The default value of
`parser` will change from `'liac-arff'` to `'auto'` in 1.4. You can set
`parser='auto'` to silence this warning. Therefore, an `ImportError` will be
raised from 1.4 if the dataset is dense and pandas is not installed. Note that
the pandas parser may return different data types. See the Notes Section in
fetch_openml's API doc for details.
  warn(
[[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]
```

b) For K ranging from 1 to 20:

1. train a KNN on the training data
2. record the training and testing accuracy

Plot a graph of the training and testing set accuracy as a function of the number of neighbors K (on the same plot). Which value of K is optimal? Briefly explain.

[9]:
```python
from sklearn.neighbors import KNeighborsClassifier

score = []



#Finding the best k value from 1 to 20
for i in range(1, 21):

    knn = KNeighborsClassifier(n_neighbors=i)

    knn.fit(X_train, Y_train)

    score.append(knn.score(X_test, Y_test))


plt.scatter(range(1, 21), score)
plt.xlabel('k')
plt.ylabel('accuracy')
plt.show()
```

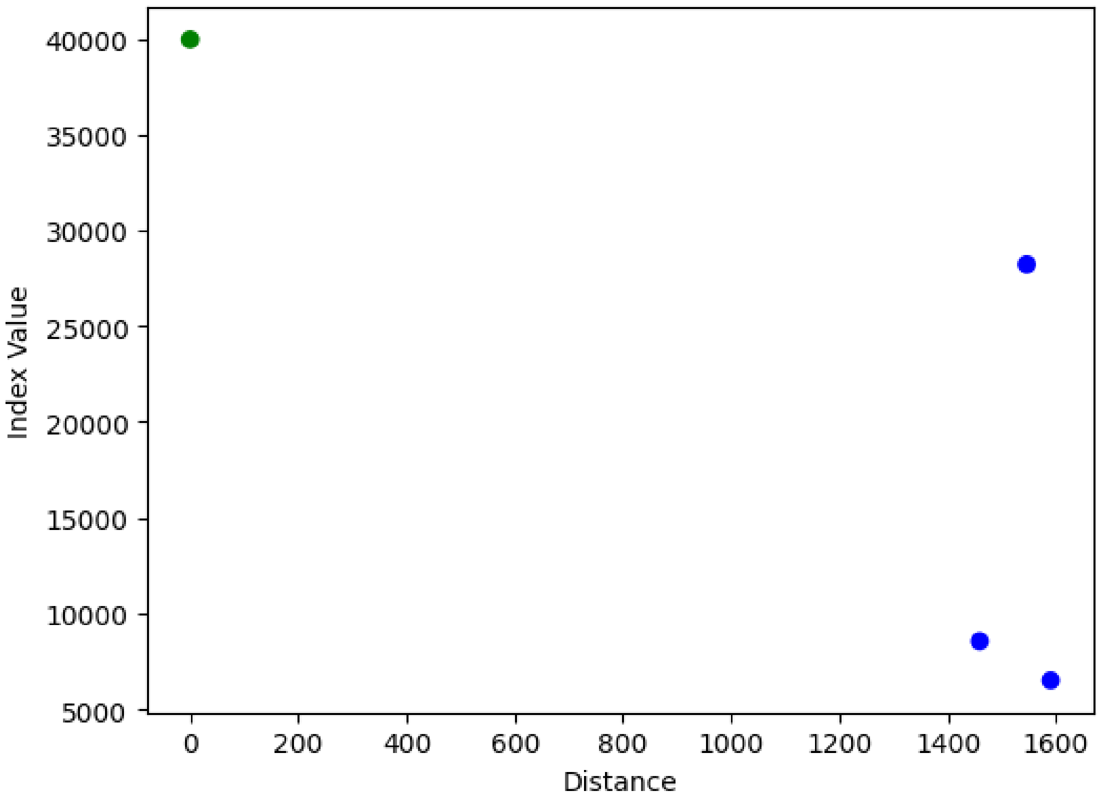


c) Using the best model from b), pick an image at random and plot it next to its K nearest neighbors

[10]:
```
import random


#The best model in y opinion is the one with k = 3
knn = KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, Y_train)


#Randomly selecting a point from the dataset
random_point = random.randint(0, len(X_train))



#Finding the 3 nieghbors of the random point One of the closest points is the␣
↪point itself so thats why we do 4 instead of 3
dis , points = knn.kneighbors(X_train[random_point].reshape(1, -1), 4,␣
↪return_distance=True)
```

```python
print(dis)
print(points)

col=[1,0,0,0]
#GREEN ONE IS THE ORIGINAL POINT
plt.scatter(dis, points, color = colors[col].tolist())
plt.xlabel("Distance")
plt.ylabel("Index Value")
plt.show()
```

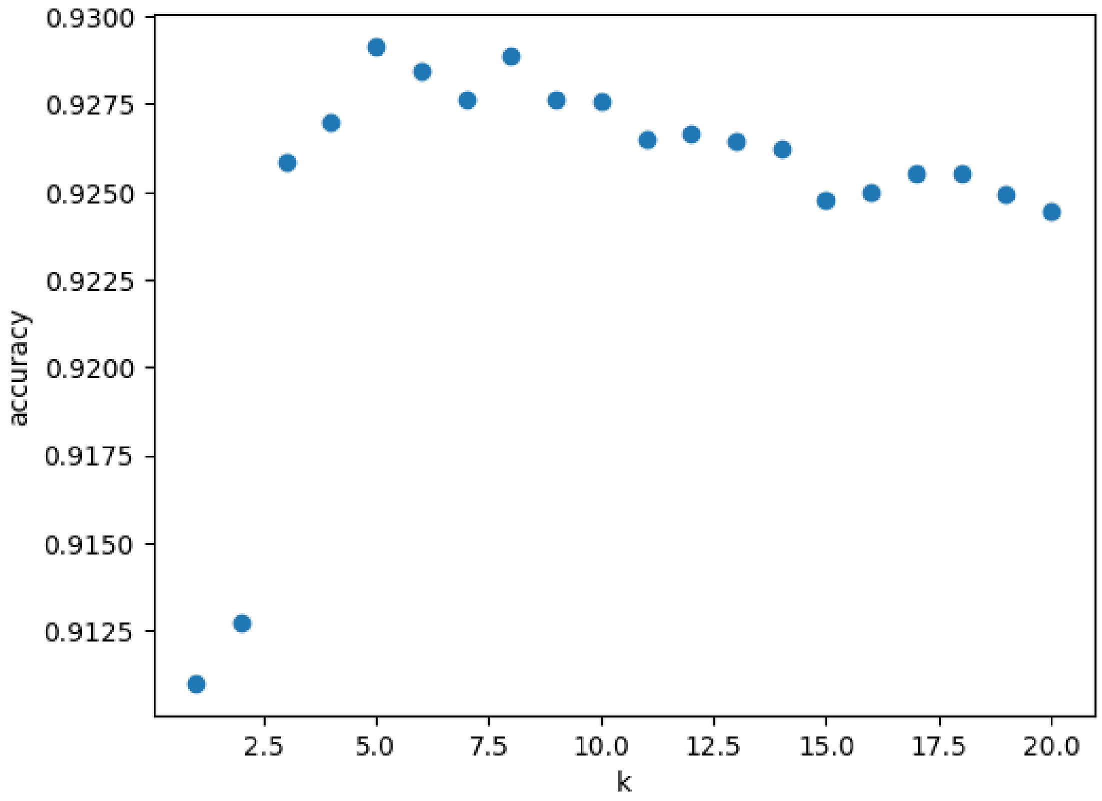
```
[[   0.         1459.40981222 1546.29913018 1590.59202815]]
[[39968  8557 28228  6515]]
```



d) Using a dimensionality reduction technique discussed in class, reduce the dimensionality of the dataset before applying a KNN model. Repeat b) and discuss similarities and differences to the previous model. Briefly discuss your choice of dimension and why you think the performance / accuracy of the model has changed.

[39]:
```python
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier



score = []
#Finding the best k value from 1 to 20
for i in range(1, 21):

    #Uses the first 10 rows of the singular value matrix that is basically what␣
 ↪the TruncatedSVD does
    model = make_pipeline(TruncatedSVD(n_components=10),␣
 ↪KNeighborsClassifier(n_neighbors=i))
    model.fit(X_train, Y_train)

    score.append(model.score(X_test, Y_test))


print(score)
plt.scatter(range(1, 21), score)
plt.xlabel('k')
plt.ylabel('accuracy')
plt.show()
```

```
[0.911, 0.9127142857142857, 0.9258571428571428, 0.927, 0.9291428571428572,
0.9284285714285714, 0.9276428571428571, 0.9288571428571428, 0.9276428571428571,
0.9275714285714286, 0.9265, 0.9266428571428571, 0.9264285714285714,
0.9262142857142858, 0.9247857142857143, 0.925, 0.9255, 0.9255,
0.9249285714285714, 0.9244285714285714]
```



I noticed that the accuaracy was very Similar and I also noticed that the more dimensions I use the better that the accuracy gets. The pattern of the data also generally is very similar to how it was beforehand. My choice of dimeninsion relied on keeping as much accuracy as possible while reducing as my dimensions as possible. I tried trial and error over and over again increasing the dimensions by a few to see what is best. I landed at 10 dimensions since I get above 90 percent accuracy on all iterations of KNN

1.2 Midterm Prep (Part 1)

Compete in the Titanic Data Science Competition on Kaggle: https://www.kaggle.com/c/titanic

Requirements:

1. Add at least 2 new features to the dataset (explain your reasoning below)
2. Use KNN (and only KNN) to predict survival
3. Explain your process below and choice of K
4. Make a submission to the competition and provide a link to your submission below.
5. Show your code below

[37]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
```

```
"""LINK:https://www.kaggle.com/competitions/titanic/submissions#14611"""

file=pd.read_csv("train.csv")

LE = LabelEncoder()



#My Two Features
features = ["Sex", "Age"]



#file["Age"] = file["Age"].fillna(file["Age"].)

file = file.dropna(subset=features)


file["Sex"] = LE.fit_transform(file['Sex'])

X = file[features]
Y = file["Survived"]

test = pd.read_csv("test.csv")

#test = test.dropna(subset=features)


#Filling in the missing values with the mean of the column This is fill␣
 ↪Kaggle's requirement of thier being 418 rows
test["Age"] = test["Age"].fillna(test["Age"].mean())

test['Sex']= LE.fit_transform(test['Sex'])



"""After trying multiple Nieghbors, 5 was the best, I determined this by simply␣
 ↪trial and error. My methodology was nearly doubling the nieghbors each times
Until my score on kaggle got lower, when it did, I went ahead and tried to find␣
 ↪a 'sweet spot' by decreasing the number of neighbors one by one and ending␣
 ↪up on five"""

knn= KNeighborsClassifier(n_neighbors=5)


knn.fit(X,Y)
```

```python
pre =knn.predict(test[features])

#Saving the CSV file
csv = pd.DataFrame({
    'PassengerId': test['PassengerId'],
    'Survived': pre
})

csv.to_csv('sub.csv', index=False)
```

My Methodology:

The features I chose to add were Age and Sex since I thought these were very related on prioritizing lifeboats. Furthermore, I chose my K as a sort of trial and error methodology. Secondly in the training data I removed any rows that did not contain a number in either the Age or Sex rows given that these were going to be useless and make it harder for KNN to process as well. Finally I made the Nan values for Age in the test data set the mean as a sort of last resort since kaggle still relied on this data and I needed a number there if was going to use the model to predict it.