KNN vs. KMeans

# Introduction

The dataset used for this study was the Iris dataset consisting of 150 samples with 4 flower features classifying 3 types of flowers: setosa, versicolor, virginica. **KMeans** was used as an unsupervised clustering algorithm to classify the data around 3 centroids, then **K nearest-neighbors (KNN)** was used on the Kmean clusters classification to predict the class of a test set. A KNN class and a Kmean class which implements the respective algorithms was developed in python with code in the Appendix and on GitHub at <https://github.com/tljstewart/machine-learning/blob/main/knn.ipynb>

A screenshot of a computer

Description automatically generated with medium confidence

## KNN

KNN is shorthand for K Nearest-Neighbor and is a form of supervised learning. KNN doesn’t train in the since that a neural network trains and adjusts weights, KNN just compares the unknown sample to the known samples you’ve provided.

One passes in a dataset with labels, then KNN is able to make a prediction when one passes subsequent unlabeled samples. To predict the class of the unknown data point one must define how many neighbors to be used. One should use an odd number to ensure a majority vote can be determined, in this case 3, 4 and 5 nearest neighbors were used and compared.

KNN algorithm genius comes from computing the class of the nearest neighbors based on distance, there are many ways to compute distance however, in this case we use the Euclidean distance, d, defined by the equation:

A picture containing text

Description automatically generated

In the KNN case we are working with feature vectors **X** the essence of this equation is the same, however, it may be represented as an **n** dimensional distance formula where **X** is

## KMeans

KMeans is an algorithm that uses centroids and the Euclidian distance, the centroids are updated each iteration until it converges on a solution, while the class of the data point is determined by its nearest centroid. Therefore, KMeans has two primary steps, cluster assignment step for each data point and a centroid update step. There are essentially three stopping criteria that can be adopted to stop the K-means algorithm:

1. Centroids of newly formed clusters do not change
2. Points remain in the same cluster
3. Maximum number of iterations are reached

# Results

**KMeans** was used as an unsupervised clustering algorithm to classify the data around 3 centroids, then **K nearest neighbors (KNN)** was used on this classification to predict the class of a test set, it was determined by comparing to the ground truth data that KMeans had a **81.2% classification accuracy**.

Chart, scatter chart

Description automatically generated

Figure 1. The Kmean classification of 3 flowers, with cluster centroids marked with X

**KNN** was also used on the ground truth data, where the data set was split into a training set and a test set, the k parameter that determines how many neighbors to use for the euclidean distance was tested at 3, 4 and 5. A k value of 3 and 4 had **100%** accuracy on the ground truth data, where k=5 showed a drop in accuracy to **96%.** Also, note that even values for k, such as 4, should be avoided as we rely on a majority vote for classification and even number do not exhibit this characteristic (i.e., need a tie-breaker)

Chart, scatter chart

Description automatically generated

Figure 2. Ground Truth dataset, blue is Setosa, red is versicolor and green is virginica

Chart, scatter chart

Description automatically generated

Figure 3. KNN only, Training set marked with ‘o’, test set marked with ‘\*’

In Figure 3. one can see the training set marked with ‘o’ while the test set is marked with ‘\*’ this yielded a 100% accuracy with k=3 but dropped to 96% with k=5. This could be to the fact that feature 1 and feature 2 are highly correlate for Setosa (blue) and virginica (green), this correlation does not maximize the information in the feature set, in fact it duplicates it, a different feature from the 4 would have been a better choice.

Chart, scatter chart

Description automatically generated

Figure 4. KMean and KNN, KMean marked with ‘o’, KNN marked with ‘\*’

In Figure 4. One can observe KMeans clustering marked with ‘o’ and KNN marked with ‘\*’. Furthermore, one may observer the misclassification by KMeans, where ‘\*’ marks the data point a different color than KMeans; notably the teal on blue and the blue on green. The correct data points show green on green, blue on blue and slight red on orange.

**IMPORTANT NOTE**: It’s important to realize here that KMeans assign numerical labels in a clustering mechanism, however, the number KMeans chooses to label a cluster may not be the same label in your ground truth data, which could lead to erroneous accuracy, in this specific study an erroneous 22% accuracy was observed, however, in this case label 0 by KMeans needs to be swapped with label 1, this resulted in a 82% accuracy comparing KMeans predictions to KNN and Ground Truth predictions, as KNN had 100%

# Conclusion

KNN is a supervised algorithm that compares classification based on Euclidean distance to known samples. KNN with k value of 3 produced **100%** accuracy on the ground truth data, where k=5 showed a drop in accuracy to **96%**. KMeans is an unsupervised clustering algorithm for classification that uses centroids and euclidean distance for the data points to the centroids to classify. KMeans clustering showed an **82%** accuracy compared to KNN on ground truth data.

# Appendix

**import** **numpy** **as** **np**

**from** **sklearn** **import** datasets

**from** **sklearn.model\_selection** **import** train\_test\_split

**import** **matplotlib.pyplot** **as** **plt**

**import** **matplotlib.colors** **as** **cm**

**from** **collections** **import** Counter

cmap = cm.ListedColormap(['#FF0000','#0000FF','#00FF00'])

iris\_dataset = datasets.load\_iris()

X, y = iris\_dataset.data, iris\_dataset.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1234)

*# print(iris\_dataset.target\_names) #['setosa' 'versicolor' 'virginica']*

print(iris\_dataset.feature\_names) *#['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']*

*# print(iris\_dataset.target) # 0 1 2 reference the labels ['setosa' 'versicolor' 'virginica']*

*# print(iris\_dataset.data) # this is the feature vector, it contains 4 feature columns, however, only 2 features are used so that it may be plotted.*

*# print(f"X\_train shape: {X\_train.shape}\n")*

*# print(f"X\_train[0]: {X\_train[0]}\n")*

*# print(f"y\_train shape: {y\_train.shape}\n")*

*# print(f"y\_train:\n {y\_train}")*

*# fig = plt.figure()*

*# ax = fig.gca()*

*# ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolor='k', s=50)*

*## KMEAN CLUSTERING (K here has nothing to do with K in KNN)*

np.random.seed(42)

**class** **KMEAN**:

*#where K is the max number of cluster centroids*

**def** \_\_init\_\_(self, K=5, max\_iters=100, plot\_steps=**False**):

self.K = K

self.max\_iters = max\_iters

self.plot\_steps = plot\_steps

*#list of sample indicies for each cluster*

self.clusters = [[] **for** \_ **in** range(self.K)] *#init an empty [] list*

*#mean feature vector for each cluster*

self.centroids = []

**def** predict(self, X):

self.X = X

self.n\_samples, self.n\_features = X.shape

*#init centroids*

random\_sample\_idxs = np.random.choice(self.n\_samples, self.K, replace=**False**)

self. centroids = [self.X[idx] **for** idx **in** random\_sample\_idxs]

*#optimization*

**for** \_ **in** range(self.max\_iters):

*#update clusters*

self.clusters = self.\_create\_clusters(self.centroids)

*#update ccentroids*

centroids\_old = self.centroids

self.centroids = self.\_get\_centroids(self.clusters)

*#check if converged*

**if** self.\_is\_converged(centroids\_old, self.centroids):

**break**

*#return cluster labels*

**return** self.\_get\_cluster\_labels(self.clusters)

**def** \_get\_cluster\_labels(self, clusters):

labels = np.empty(self.n\_samples)

**for** cluster\_idx, cluster **in** enumerate(clusters):

**for** sample\_idx **in** cluster:

labels[sample\_idx] = cluster\_idx

**return** labels

**def** \_create\_clusters(self, centroids):

clusters = [[] **for** \_ **in** range(self.K)]

**for** idx, sample **in** enumerate(self.X):

centorid\_idx = self.\_closest\_centroid(sample, centroids)

clusters[centorid\_idx].append(idx)

**return** clusters

**def** \_closest\_centroid(self, sample, centroids):

distances = [self.euclidean\_distance(sample, point) **for** point **in** centroids]

closest\_idx = np.argmin(distances)

**return** closest\_idx

**def** \_get\_centroids(self, clusters):

centroids = np.zeros((self.K, self.n\_features))

**for** cluster\_idx, cluster **in** enumerate(clusters):

cluster\_mean = np.mean(self.X[cluster], axis=0)

centroids[cluster\_idx] = cluster\_mean

**return** centroids

**def** \_is\_converged(self, centroids\_old, centroids):

distances = [self.euclidean\_distance(centroids\_old[i], centroids[i]) **for** i **in** range(self.K)]

**return** sum(distances) == 0

**def** euclidean\_distance(self, x1, x2):

*# print(f"x1: {x1}")*

*# print(f"x1: {x1.shape}")*

*# print(f"x2: {x2}")*

**return** np.sqrt(np.sum((x1-x2)\*\*2)) *#this actually compute on all features in the feature vector, not just the 2 or 3 we use to plot... (because plotting n dimesion is difficult)*

**def** plot(self, compare=**False**, feats=[], labels=[]):

fig, ax = plt.subplots(figsize=(12,8))

**for** i, index **in** enumerate(self.clusters):

point = self.X[index].T

ax.scatter(\*point, s=100)

**for** point **in** self.centroids:

ax.scatter(\*point, marker='x', s=300, linewidth=2) *#color="black", linewidth=2)*

**if** compare==**True**:

ax.scatter(feats[:, 0], feats[:, 1], c=labels, cmap=cmap, s=20, marker='\*')

ax.set\_xlabel("Feature 1")

ax.set\_ylabel("Feature 2")

ax.set\_title("Kmean Predidtion and Ground Truth, Flower Class")

ax.legend(iris\_dataset.target\_names, loc='best')

**else**:

ax.set\_xlabel("Feature 1")

ax.set\_ylabel("Feature 2")

ax.set\_title("Kmean Cluster Predidtions, Flower Class")

ax.legend(iris\_dataset.target\_names, loc='best')

plt.show()

**return** fig,ax

**import** **numpy** **as** **np**

**from** **sklearn.datasets** **import** make\_blobs

*#X, y = make\_blobs(centers=4, n\_samples=500, n\_features=2, shuffle=True, random\_state=42)*

*#X, y = make\_blobs(centers=3, n\_samples=500, n\_features=2, shuffle=True, random\_state=40)*

print(X.shape)

X\_kmeans = X[:, :2] *# need to force X to have 2 features, this only works for 2 feature... something about the plot function....*

print(X\_kmeans.shape)

clusters = len(np.unique(y)) *#get how many labels in y*

print(clusters)

kmean = KMEAN(K=clusters, max\_iters=150, plot\_steps=**True**)

y\_pred\_by\_kmean = kmean.predict(X\_kmeans)

print(y\_pred\_by\_kmean)

print(y)

*#its important to realize here that kmean assign numerical labels in a clustering mechanism, however, the number kmeans chooses to label a cluster may not be*

*#the same label in your ground truth data, which could lead to erroneous accurarcy.*

kmean.plot()

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

(150, 4)

(150, 2)

3

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.

1. 1. 2. 2. 2. 0. 2. 0. 2. 0. 2. 0. 0. 0. 0. 0. 0. 2. 0. 0. 0. 0. 0. 0.

0. 0. 2. 2. 2. 2. 0. 0. 0. 0. 0. 0. 0. 0. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 2. 0. 2. 0. 2. 2. 1. 2. 2. 2. 2. 2. 2. 0. 0. 2. 2. 2. 2. 0.

2. 0. 2. 0. 2. 2. 0. 0. 2. 2. 2. 2. 2. 0. 0. 2. 2. 2. 0. 2. 2. 2. 0. 2.

2. 2. 0. 2. 2. 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 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]

Chart, scatter chart

Description automatically generated

Out[187]:

(<Figure size 864x576 with 1 Axes>,

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fb5e0a0e690>)

## KNN

KNN is K Nearest Neighbor and is a form of supervised learning.

We pass in a dataset with labels, then to predict the class of the an unknown data point. First we define how many neighbors to be used. One should use an odd number to ensure a majority vote can be determined, in this case 3, 4 and 5 neartest neighbors were used and compared.

KNN needs to comupute the nearest neighbors based on distance, there are many different ways to compute distance however, in this case we use the euclidean distance  is defined by equation:

In the KNN case we are working with feature vectors , the essence of this equation is the same, however, it may be represented as an  dimensional distance forumla where  is $\mathbb{R^n}$

In [188]:

*# KNN (K here has nothing to do with K in KMEAN)*

**class** **KNN**:

**def** \_\_init\_\_(self, k=3):

self.k = k

*#there isn't training in knn, it's a compute prediction not a weights computation or a best fit line.*

**def** fit(self, X, y):

self.X\_train = X

self.y\_train = y

*#capital X to designate prediction for mulitple samples*

**def** predict(self, X):

predicted\_labels = [self.\_predict(x) **for** x **in** X]

*#A note on syntax here:*

*# the [] is called list comprehension syntax --> [self.\_predict(x) for x in X] says, return a prediction for a sinlge sample little x for ever sample in big X*

*# exmaple code: primes = [x for x in range(2, 50) if x not in noprimes]*

*# Translation: Primes = All "x for" which "x in range(2,50)" but only "if x not in noprimes"*

**return** np.array(predicted\_labels) *#conversion to numpy array*

*#lower case x to designate predict only 1 sample*

**def** \_predict(self, x):

*#compute the euclidean distance*

distances = [self.euclidean\_distance(x, x\_train) **for** x\_train **in** self.X\_train]

*#get k nearest samples, labels*

k\_indices = np.argsort(distances)[:self.k] *# this is an example of slicing, it slices out a section of data columns 2 to 5 like some\_data[2:5] #[:5] is equivalent to [0:5].*

k\_nearest\_labels = [self.y\_train[i] **for** i **in** k\_indices]

*#majority vote, most common class label*

most\_common = Counter(k\_nearest\_labels).most\_common(1)

**return** most\_common[0][0]

**def** euclidean\_distance(self, x1, x2):

*# print(f"x1: {x1}")*

*# print(f"x1: {x1.shape}")*

*# print(f"x2: {x2}")*

**return** np.sqrt(np.sum((x1-x2)\*\*2)) *#this actually compute on all features in the feature vector, not just the 2 or 3 we use to plot... (because plotting n dimesion is difficult)*

**def** accuracy(self, y\_ground\_truth, y\_pred):

correct\_preds = np.sum(y\_ground\_truth == y\_pred)

total\_samples = len(y\_ground\_truth)

print(f"total correct predictions: **{**correct\_preds**}**")

print(f"total samples: **{**total\_samples**}**")

accuracy = correct\_preds / total\_samples

**return** accuracy

In [213]:

k = 3

knn = KNN(k=k)

knn.fit(X\_train, y\_train)*#there isn't training in knn, it's a compute prediction not a weights computation or a best fit line.*

predictions = knn.predict(X\_test) *#here it will compare the samples in X\_test to X\_train, it will look at X\_test neartest neighbors in the X\_train with known lables.*

print("Custom KNN classification accuracy", knn.accuracy(y\_test, predictions))

plt.figure()

*#ax = fig.gca()*

*#ax = fig.gca(projection='3d')*

k = 4 *#you want to pick odd numbers to to ensure a majority vote*

knn = KNN(k=k)

knn.fit(X\_train, y\_train)*#there isn't training in knn, it's a compute prediction not a weights computation or a best fit line.*

predictions = knn.predict(X\_test) *#here it will compare the samples in X\_test to X\_train, it will look at X\_test neartest neighbors in the X\_train with known lables.*

print("Custom KNN classification accuracy", knn.accuracy(y\_test, predictions))

plt.figure()

k = 5

knn = KNN(k=k)

knn.fit(X\_train, y\_train)*#there isn't training in knn, it's a compute prediction not a weights computation or a best fit line.*

predictions = knn.predict(X\_test) *#here it will compare the samples in X\_test to X\_train, it will look at X\_test neartest neighbors in the X\_train with known lables.*

print("Custom KNN classification accuracy", knn.accuracy(y\_test, predictions))

plt.figure(figsize=(12,8))

*# ax.scatter3D(X\_test[:, 0], X\_test[:, 1], X\_test[:, 2], c=y\_test, cmap="gray", edgecolor='k', s=50)*

*# ax.scatter3D(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=cmap, edgecolor='k', s=50)*

*# plt.xlabel("Feature 1")*

*# plt.ylabel("Feature 2")*

*# plt.ylabel("Feature 3")*

*# plt.show()*

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=cmap, edgecolor='k', s=50)

plt.scatter(X\_test[:, 0], X\_test[:, 1], edgecolor='k', s=100, marker='\*')

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.title("Train Dataset, Test set \*")

*#plt.legend(iris\_dataset.target\_names, loc='best') #not working for some reason*

plt.show()

total correct predictions: 30

total samples: 30

Custom KNN classification accuracy 1.0

total correct predictions: 30

total samples: 30

Custom KNN classification accuracy 1.0

total correct predictions: 29

total samples: 30

Custom KNN classification accuracy 0.9666666666666667

<Figure size 432x288 with 0 Axes>

<Figure size 432x288 with 0 Axes>

Chart, scatter chart

Description automatically generated

In [218]:

plt.figure(figsize=(12,8))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolor='k', s=100)

plt.title("Ground Truth")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

*#plt.legend(iris\_dataset.target\_names, loc='best') #not working for some reason*

plt.show()

Chart, scatter chart

Description automatically generated

In [214]:

plt.figure(figsize=(12,8))

*# ax.scatter3D(X\_test[:, 0], X\_test[:, 1], X\_test[:, 2], c=y\_test, cmap="gray", edgecolor='k', s=50)*

*# plt.xlabel("Feature 1")*

*# plt.ylabel("Feature 2")*

*# plt.show()*

plt.scatter(X\_test[:, 0], X\_test[:, 1], edgecolor='k', s=100, marker='\*')

plt.title("Test set as \*")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Chart, scatter chart

Description automatically generated

In [199]:

*#there are issues adding to plots, espeically ones that return from fucntions... im chalking with one up as a juypter colab issue.*

[figg, axx] = kmean.plot(compare=**True**, feats=X, labels=y)

[figg, axx] = kmean.plot()

*#axx.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=cmap, edgecolor='k', s=50)*

*#axx.scatter(-1, -1, cmap=cmap, edgecolor='k', s=50)*

*#figg.canvas.draw()*

*#axx.scatter(X\_test[:, 0], X\_test[:, 1], edgecolor='k', s=100, marker='\*')*

*#axx.set\_xlabel("Feature 1")*

*#axx.set\_ylabel("Feature 2")*

*#axx.set\_title("Kmean Predidtion and Ground Truth")*

*#figg.legend(loc='best')*

*#figg.canvas.draw()*

*#axx.set(xlabel='Feature 1', ylabel='Feature 2', title='TITLE')*

k = 3

knn = KNN(k=k)

knn.fit(X, y)*#there isn't training in knn, it's a compute prediction not a weights computation or a best fit line.*

predictions = knn.predict(X\_test) *#here it will compare the samples in X\_test to X\_train, it will look at X\_test neartest neighbors in the X\_train with known lables.*

print("Using Kmean clustering labels to determine accuracy with Ground Truth Labels", knn.accuracy(y, y\_pred\_by\_kmean))

print(y\_pred\_by\_kmean)

print(y)

*#its important to realize here that kmean assign numerical labels in a*

*#clustering mechanism, however, the number kmeans chooses to label a cluster may not be*

*#the same label in your ground truth data, which could lead to erroneous accurarcy,*

*#this specific exmaple shows an erroneous 22% accuracy.*

*#in this case label 0 by kmean needs to be swapped with label 1*

y\_pred\_by\_kmean = y\_pred\_by\_kmean.astype(int)

**for** i, pred **in** enumerate(y\_pred\_by\_kmean):

**if** pred == 0:

y\_pred\_by\_kmean[i] = 1

**elif** pred == 1:

y\_pred\_by\_kmean[i] = 0

*# y\_pred\_by\_mean = np.where(y\_pred\_by\_kmean == 1, 5, y\_pred\_by\_kmean)*

*# # y\_pred\_by\_mean = np.where(y\_pred\_by\_kmean == 0, 1, y\_pred\_by\_kmean)*

*# y\_pred\_by\_mean = np.where(y\_pred\_by\_kmean == 1, 0, y\_pred\_by\_kmean)*

print("Using Kmean clustering labels to determine accuracy with Ground Truth Labels", knn.accuracy(y, y\_pred\_by\_kmean))

print(y\_pred\_by\_kmean)

print(y)

*# np.c\_ is the numpy concatenate function*

*# which is used to concat iris['data'] and iris['target'] arrays*

*# for pandas column argument: concat iris['feature\_names'] list*

*# and string list (in this case one string); you can make this anything you'd like..*

*# the original dataset would probably call this ['Species']*

df = pd.DataFrame(data= np.c\_[iris\_dataset['data'], iris\_dataset['target']],

columns= iris\_dataset['feature\_names'] + ['target'])

*# print(df)*

*# df.head()*

display(df)

print(iris\_dataset.target\_names)

Chart, scatter chart

Description automatically generated

Chart, scatter chart

Description automatically generated

total correct predictions: 34

total samples: 150

Using Kmean clustering labels to determine accuracy with Ground Truth Labels 0.22666666666666666

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 0 2 0 2 0 2 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0

2 2 2 2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 2 2 1 2 2 2 2

2 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2

2 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 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]

total correct predictions: 122

total samples: 150

Using Kmean clustering labels to determine accuracy with Ground Truth Labels 0.8133333333333334

[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 0

0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 2 1 2 1 2 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1

2 2 2 2 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1 2 2 0 2 2 2 2

2 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 1 2 2 2 1 2 2 2 1 2 2 2 1 2

2 1]

[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 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]