Machine Learning Assignment2 (group24)

source: https://github.com/WarClans612/machine_learning/tree/master/hw2

- Working environment

張翔中	劉昱劭	彭敬樺	周才錢
macOS	macOS/Ubuntu 16.04	Ubuntu 16.04	Windows

IDE	Jupyter Notebook

- Our files (click to see readme.md)

ML Assignment2

K-means

- `kmeans.py` is pure kmeans code.
- `kmeans_xy.ipynb` is kmeans by `x` & `y`, with k<=6
- `kmeans_xy_3.ipynb` is kmeans by `x` & `y`, with k=3. we calculate **accuracy** here.
- `kmeans_ss.ipynb` is kmeans by `speed` & `spin`, with k=3.

KD-Tree

- `kdtree/kd.py` is the source code of our kdtree.
- `kdtree/kd.ipynb` shows the visualized results.

I. K-means (click to see .ipynb file)

- Code (split in few partitions)
  - Plot the result of k means clustering

```python
def draw_plot():
    plt.figure(figsize=(5, 5))
    plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
    for i in centroids.keys():
        plt.scatter(*centroids[i], color=colmap[i])
    plt.xlim(plotx_min, plotx_max)
    plt.ylim(ploty_min, ploty_max)
    plt.show()
```

  - Assignment each data point to a cluster

```python
## Assignment Stage
def assignment(df, centroids):
    for i in centroids.keys():
        # sqrt((x1 - x2)^2 - (y1 - y2)^2)
        df['distance_from_{}'.format(i)] = (
            np.sqrt(
                (df['x'] - centroids[i][0]) ** 2
              + (df['y'] - centroids[i][1]) ** 2
            )
        )
    centroid_distance_cols = ['distance_from_{}'.format(i) for i in centroids.keys()]
    df['closest'] = df.loc[:, centroid_distance_cols].idxmin(axis=1)
    df['closest'] = df['closest'].map(lambda x: int(x.lstrip('distance_from_')))
    df['color'] = df['closest'].map(lambda x: colmap[x])
    return df
```

  - Update the center of each cluster

```python
## Update Stage
def update(k):
    for i in centroids.keys():
        centroids[i][0] = np.mean(df[df['closest'] == i]['x'])
        centroids[i][1] = np.mean(df[df['closest'] == i]['y'])
    return k
```

  - Main loop (randomly pick centroids at first)

```python
sumerr = []
K = 5 # K<=7
for k in range(1, K):
    centroids = { i+1: [df['x'][entry], df['y'][entry]] for i, entry in enumerate(random.sample(range(len(df)), k)) }
#    old_centroids = copy.copy(centroids)
    df = assignment(df, centroids)

    while True:
        closest_centroids = df['closest'].copy(deep=True)
        centroids = update(centroids)
        df = assignment(df, centroids)
        if closest_centroids.equals(df['closest']): break

    draw_plot()
    sumerr.append(cost_func())

```

  - Cost function (sum of error)

We use $J = \sum_{k=1}^{K} \sum_{i \in C_k} ||x_i - \mu_k||$ instead of $J = \sum_{k=1}^{K} \sum_{i \in C_k} ||x_i - \mu_k||^2$

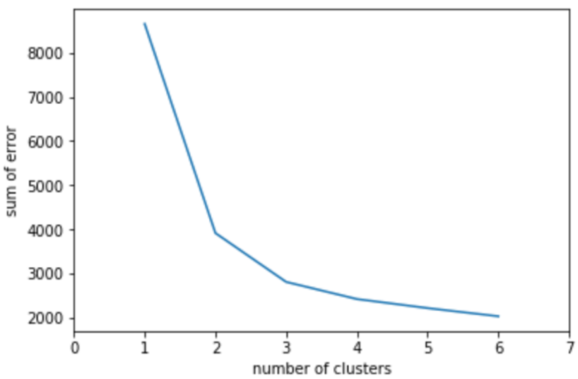
```python
# cost function
def cost_func():
    err = 0;
    for i in range(len(df)):
        j = df['closest'][i]
        err = err + df['distance_from_{}'.format(j)][i]
    return err
```

■ Confusion matrix (accuracy)

```python
from sklearn.metrics import confusion_matrix

pitch = pd.read_csv('data_noah.csv', usecols=['pitch_type'])
x = len(np.unique(pitch))
label = np.append(np.unique(pitch), np.unique(df['color']))
cfmx = pd.DataFrame(confusion_matrix(pitch, df['color']), index=label, columns= label)
cfmx = cfmx.iloc[:x, x:]
print(np.unique(pitch), np.unique(df['color']))

col = {'r': 'red', 'g': 'green', 'b': 'blue', 'c': 'cyan', 'm': 'magenta', 'y': 'yellow'}
def func(s):
    return ['background-color: {}; opacity: 0.6'.format(col[s.name])]*len(s)

cfmx.style.apply(func, axis=0)
```

- Cost function & accuracy

■ Sum of error among different numbers of K means clustering



■ Confusion matrix (k = 3)

	b	g	r
CH	1	161	0
CU	0	0	301
FF	595	263	0

■ accuracy

source: click here

we count $\boxed{\text{accuracy} = \frac{\text{accurate\_points}}{\text{total\_points}}}$

accuracy of k=3 is about 0.8.

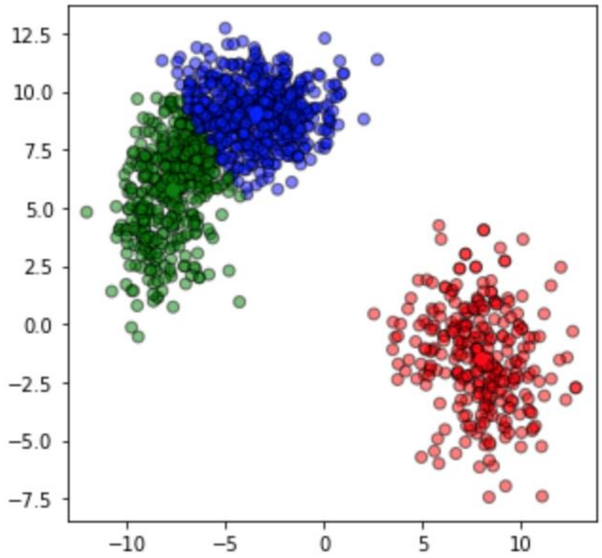
In [38]:

```python
accurate_points = 0

for row in cfmx.index:
    color = np.where(cfmx.loc[row] == cfmx.loc[row].max())[0][0]
    accurate_points = accurate_points + cfmx.loc[row][color]

#print(accurate_points)

total_points = cfmx.sum().sum()
#print(total_points)

accuracy = accurate_points/total_points
#print(accuracy)

row_name = ['accurate_points', 'total_points', 'accuracy']

pd.DataFrame([accurate_points,total_points,accuracy], columns=['value'], index=row_name)
```

Out[38]:

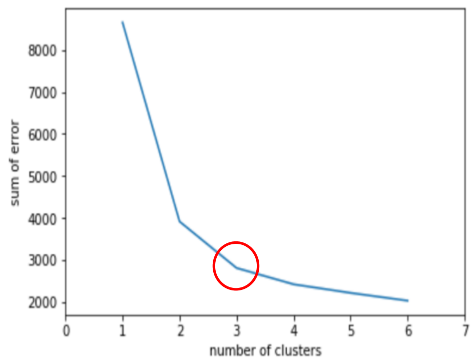
	value
accurate_points	1057.000000
total_points	1321.000000
accuracy	0.800151

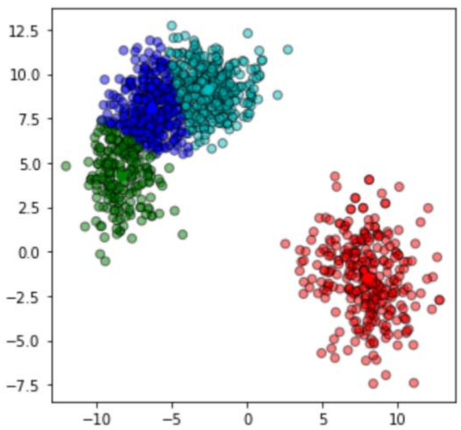
- The result of K-Means clustering



- Is k=3 the best clustering?

If we use the elbow method, we can say that k=3 is the best k, since adding more clusters didn't get significant decrease of sum of error. However, there could be room for discussion that k=3 may not be the best k. As we can see in the confusion matrix at k=3, nearly one third of four-seam fastballs are mixed with changeups. From the picture and confusion matrix of k=4 below, it is quite obvious that the four-seam fastballs are well split from the changeups, we can therefore combine blue and cyan clusters to make it k=3. It is k=3 at final but through an indirect way, which is not the case.

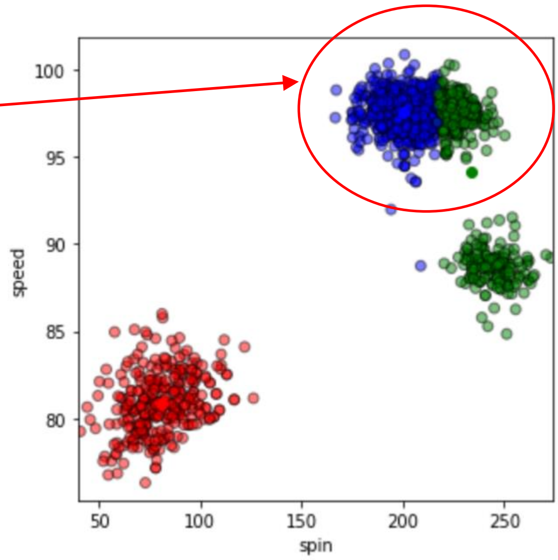




	b	c	g	r
CH	4	0	158	0
CU	0	0	0	301
FF	360	436	62	0

- Use another two or more attributes to partition
  - source: [click here]
  - Ex. Use speed and spin attributes
  - This result looks not good enough.

	b	g	r
CH	1	161	0
CU	0	0	301
FF	599	259	0



II. KD-Tree (click to see .ipynb file)

- ## KD-Tree code

We modifed the sample code from `astroML`, but we didn't use `astroML`'s library
source: http://www.astroml.org/book_figures/chapter2/fig_kdtree_example.html

- Build a KDTree class

```
class KDTree:
```

- init function & variable initialization

```
# class initialization function
def __init__(self, data, mins, maxs, prev):
    self.data = np.asarray(data)

    # data should be two-dimensional
    #assert self.data.shape[1] == 2

    if mins is None:
        mins = data.min(0)
    if maxs is None:
        maxs = data.max(0)

    self.mins = np.asarray(mins)
    self.maxs = np.asarray(maxs)
    self.sizes = self.maxs - self.mins

    self.prev = prev
    self.leaf = False

    self.child1 = None
    self.child2 = None
```

- Find the more spread dimension

```
if len(data) > 0:
    # sort on the dimension with the largest spread
    largest_dim = self.prev
    if self.prev == -1:
        largest_dim = np.argmax(self.sizes)
    else:
        largest_dim = (largest_dim+1)%2
    i_sort = np.argsort(self.data[:, largest_dim])
    self.data[:] = self.data[i_sort, :]
    #print("data: \n", self.data)

    # find split point
    N = self.data.shape[0]
```

- For leaf node

```
if N == 1:
    split_point = self.data[:, largest_dim]
    mins1 = self.mins.copy()
    mins1[largest_dim] = split_point
    maxs2 = self.maxs.copy()
    maxs2[largest_dim] = split_point
    self.leaf = True
    self.child1 = KDTree([], mins1, self.maxs, largest_dim)
    self.child2 = KDTree([], self.mins, maxs2, largest_dim)
```

- For non-leaf node, recursively create sub-trees.

```
else:
    split_point = np.median(self.data[:, largest_dim])
    split_point = find_nearest(self.data[:, largest_dim], split_point+0.1)
    #print("split_point: ", split_point)
    idx = np.where(self.data[:, largest_dim] == split_point)[0][0]
    #print('idx= ', idx)

    # create subnodes
    mins1 = self.mins.copy()
    mins1[largest_dim] = split_point
    maxs2 = self.maxs.copy()
    maxs2[largest_dim] = split_point
    #print("mins1, self.maxs: ", mins1, self.maxs)
    #print("self.mins, maxs2: ", self.mins, maxs2)
    #print("-------")
    # Recursively build a KD-tree on each sub-node
    self.child1 = KDTree(self.data[idx+1:], mins1, self.maxs, largest_dim)
    self.child2 = KDTree(self.data[:idx], self.mins, maxs2, largest_dim)
```

- draw_rectangle is used to plot the divided region of KD-Tree.

```
def draw_rectangle(self, ax, depth=None):
    """Recursively plot a visualization of the KD tree region"""
    if depth <= 1:
        #print('self.mins, *size: ', self.mins,  *self.sizes)
        #print()
        rect = plt.Rectangle(self.mins, *self.sizes, ec='r', fc='none')
        ax.add_patch(rect)
        pass

    if self.child2 is not None:
        if depth is None:
            self.child1.draw_rectangle(ax)
            self.child2.draw_rectangle(ax)
        elif depth > 0:
            self.child1.draw_rectangle(ax, depth - 1)
            self.child2.draw_rectangle(ax, depth - 1)
```

- depth is used to get the depth of KD-Tree.

```
def depth(self):
    current_depth = 0

    if self.child2:
        current_depth = max(current_depth, self.child2.depth())

    if self.child1:
        current_depth = max(current_depth, self.child1.depth())

    return current_depth + 1
```

● Result of KD-Tree

■ Visualize the KD-Tree

Use `kdtree` package to visualize the order of KD-Tree

KD-Tree Visualization

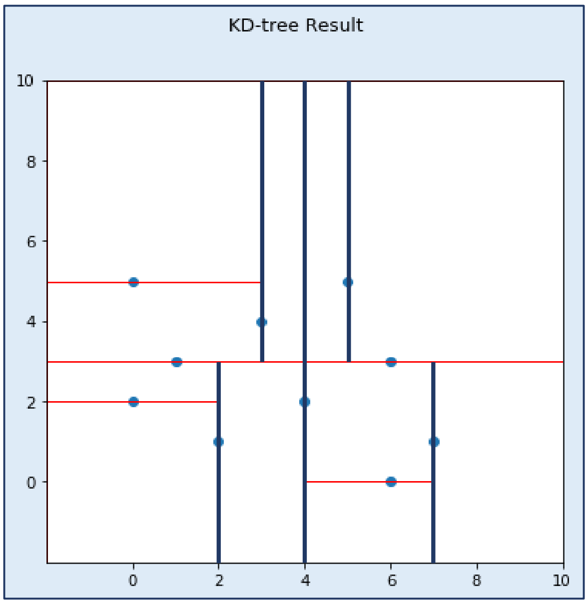
- Use another kdtree package
- source: https://github.com/stefankoegl/kdtree

In [6]:
```
import kdtree
#import pandas as pd
#points = pd.read_csv('points', sep=' ', header=None, names=head).values
tree = kdtree.create(points.tolist())
kdtree.visualize(tree)
```

```
level 1                      [4, 2]
level 2          [1, 3]                      [6, 3]
level 3    [2, 1]       [3, 4]       [7, 1]       [5, 5]
level 4  [0, 2]      [0, 5]      [6, 0]
```

■ Draw the 2-dim divided square of kd-tree

Use our `draw_rectangle` function to draw the divided rectangle.



- Draw the 2-dim divided square for each level of kd-tree

  We can see the each step of `draw_rectangle`.