CS 529 Assignment 2

Mohit Mayank – mayank@purdue.edu

October 13, 2022

PUID: 033744160

1

1.
   - assumption: features of the class are independent of each other
   - Naive-Bayes is useful when this assumption of independence and equality holds true where this model could outperform other models.

2. KNN could be better than Logistic Regression when we sufficient domain knowledge about the problem at hand since KNN depends on distance measure, as this would support derivation of an appropriate measure. Also, KNN would perform better for models with less number of parameters and dataset size.

3.

$$Entropy = -\frac{150}{200} \cdot \log \frac{150}{200} - \frac{50}{200} \cdot \log \frac{50}{200}$$

4.

Mean of A	3.4
Mean of B	23.4
Std. Dev. of A	1.067
Std. Dev. of B	0.8
Prior of A	$20/30 = 0.67$
Prior of B	$10/30 = 0.33$

5. Given:

$$P(y = 0) = \frac{1}{2}$$
$$P(y = 1) = \frac{1}{2}$$

From Gaussian distribution:

$$P(x^{(t)}|y = y_i) = \frac{1}{\sigma_t\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-\mu_t)^2}{\sigma_t^2}}$$

For $x^{(1)}$:

$$\mu(x^{(1)}|y = 1) = 6$$
$$\sigma(x^{(1)}|y = 1) = 5.65$$
$$\mu(x^{(1)}|y = 0) = 0$$
$$\sigma(x^{(1)}|y = 0) = 5.65$$

$$P(x^{(1)}|y = 1) = \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{5.65^2}}$$
$$P(x^{(1)}|y = 0) = \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x)^2}{5.65^2}}$$

For $x^{(1)}$:

$$\mu(x^2|y = 1) = 7$$
$$\sigma(x^2|y = 1) = 4.24$$
$$\mu(x^2|y = 0) = 6$$
$$\sigma(x^2|y = 0) = 1.41$$

$$P(x^{(2)}|y = 1) = \frac{1}{4.24\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-7)^2}{4.24^2}}$$
$$P(x^{(2)}|y = 0) = \frac{1}{1.41\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{1.41^2}}$$

Using Naive Bayes for $y = 0$:

$$P(y = 0|x) = P(x^{(1)}|y = 0) * P(x^{(2)}|y = 0)$$
$$P(y = 0|x) = \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x)^2}{5.65^2}} * \frac{1}{1.41\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{1.41^2}}$$

Similarly for $y = 1$:

$$P(y=1|x) = P(x^{(1)}|y=1) * P(x^{(2)}|y=1)$$
$$P(y=1|x) = \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{5.65^2}} * \frac{1}{4.24\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-7)^2}{4.24^2}}$$

From above:

$$\begin{aligned} P(y=0|x).P(y=1|x) = {} & \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x)^2}{5.65^2}} * \frac{1}{1.41\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{1.41^2}} \\ & * \frac{1}{5.65\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-6)^2}{5.65^2}} * \frac{1}{4.24\sqrt{2\pi}}.\exp^{-\frac{1}{2}.\frac{(x-7)^2}{4.24^2}} \end{aligned}$$

2

1. Information gain calculation:

$$
\begin{aligned}
\text{InfoGain}(S, A) &= H(S) - \sum_{v \in V} \left(\frac{|S_v|}{|S|} \right) H(S_v) \\
H(S) &= \frac{-9}{16} \ln\left(\frac{9}{16}\right) - \frac{7}{16} \ln\left(\frac{7}{16}\right) \\
&= 0.685314 \\
\text{InfoGain(S,"Color")} &= H(S) - \left(\frac{13}{16} \left(\frac{-8}{13} \ln\left(\frac{8}{13}\right) - \frac{5}{13} \ln\left(\frac{5}{13}\right) \right) + \frac{3}{16} \left(\frac{-1}{3} \ln\left(\frac{1}{3}\right) - \frac{2}{3} \ln\left(\frac{2}{3}\right) \right) \right) \\
&= 0.02461 \\
\text{InfoGain(S,"Size")} &= H(S) - \left(\frac{8}{16} \left(\frac{-6}{8} \ln\left(\frac{6}{8}\right) - \frac{2}{8} \ln\left(\frac{2}{8}\right) \right) + \frac{8}{16} \left(\frac{-3}{8} \ln\left(\frac{3}{8}\right) - \frac{5}{8} \ln\left(\frac{5}{8}\right) \right) \right) \\
&= 0.07336 \\
\text{InfoGain(S,"Shape")} &= H(S) - \left(\frac{12}{16} \left(\frac{-6}{12} \ln\left(\frac{6}{12}\right) - \frac{6}{12} \ln\left(\frac{6}{12}\right) \right) + \frac{4}{16} \left(\frac{-3}{4} \ln\left(\frac{3}{4}\right) - \frac{1}{4} \ln\left(\frac{1}{4}\right) \right) \right) \\
&= 0.02461
\end{aligned}
$$

Since **size** has the highest information gain, we choose it as the root of the tree.

2. Below is the Decision Tree diagram:

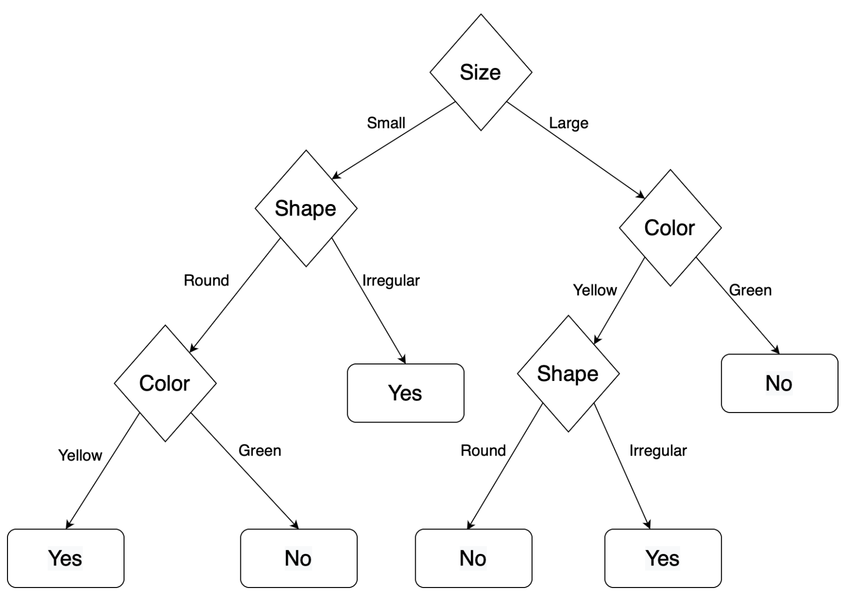


Figure 1: Decision Tree

3. If we use numerical features as separate categories, we would be creating too many categories (this would be even more huge in case of real valued features). With such features, it is highly likely that test data would contain unique values (eg: train data might contain values: 3.1, 3.112, 3.3, 4.55 and test data might contain some value like 3.35), in such cases the decision tree would fail to predict.

   We need to create bins (or ranges) to solve this problem if we really need to use Decision Trees for such a problem.

In [76]:
```
import pandas as pd

df = pd.read_csv('Pima.csv', names=['Pregnancies', 'Glucose', 'BloodPressure
assert df.shape[0] == 768
assert len(df.columns) == 9
print('Successfully verified 768 rows and 9 columns!')
df
```

```
Successfully verified 768 rows and 9 columns!
```

Out[76]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	
...	...	...	...	...	...	...	
763	10	101	76	48	180	32.9	
764	2	122	70	27	0	36.8	
765	5	121	72	23	112	26.2	
766	1	126	60	0	0	30.1	
767	1	93	70	31	0	30.4	

768 rows × 9 columns

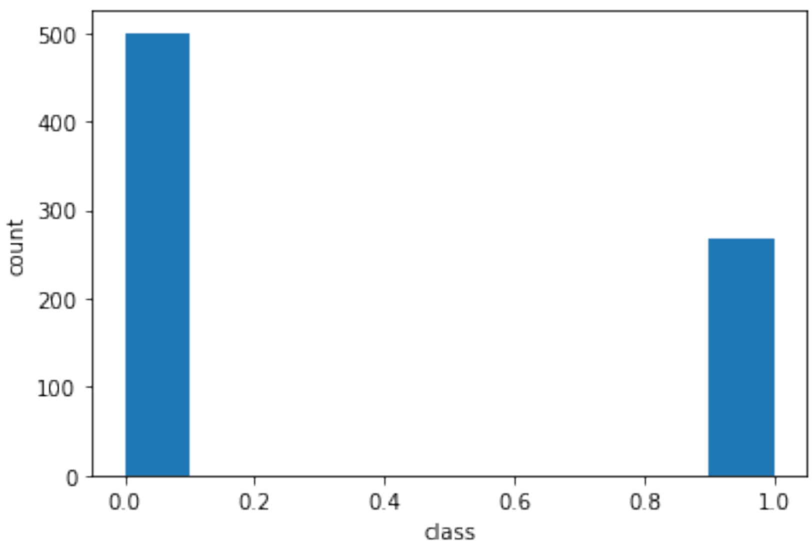
In [77]:
```
df.describe()
```

Out[77]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Di
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

In [78]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

plt.xlabel('class')
plt.ylabel('count')
plt.hist(df['Label'])
# val_counts = np.unique(df['Label'], return_counts=True)
# x = val_counts[0]
# y = val_counts[1]

# counts, bins = np.histogram(df['Label'])
# print(counts, bins)
#
# # plt.hist(x=counts, bins=bins)
# plt.stairs(counts, bins)
```

Out[78]:
```
(array([500.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 268.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <BarContainer object of 10 artists>)
```



Question 3

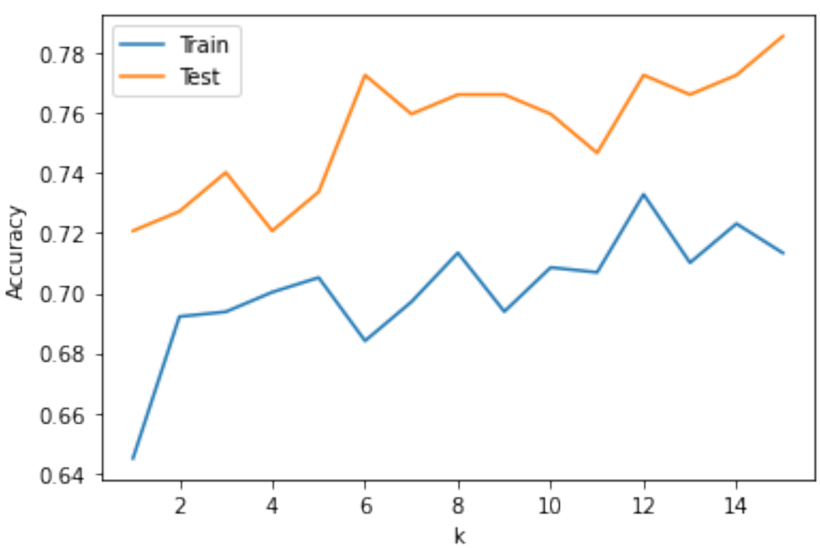
In [79]:
```
from sklearn.model_selection import train_test_split

y = df['Label']
X = df.drop(columns=['Label'])
np.random.seed(1)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8)
```

In [80]:
```
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

avg_scores = []
test_scores = []
for k in range(1, 16):
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(estimator=knn, X=X_train, y=y_train)
    avg_score = np.mean(scores)
    avg_scores.append(avg_score)
    print(f'k={k} score={avg_score}')

    # check
    knn.fit(X_train, y_train)
    score = knn.score(X_test, y_test)
    test_scores.append(score)
    # print(f'=====> k={k} test_score={score}')
    # end check

print(avg_scores)
best_k = np.argmax(avg_scores) + 1
print(f'best k={best_k}')

best_test_k = np.argmax(test_scores) + 1
print(f'best test k={best_test_k}')

x = range(1, 16)
y = avg_scores
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.plot(x, y)

y = test_scores
plt.plot(x, y)
plt.legend(['Train', 'Test'])
```

```
k=1 score=0.6449020391843263
k=2 score=0.6921764627482341
k=3 score=0.6937758230041318
k=4 score=0.7003198720511795
k=5 score=0.7051979208316673
k=6 score=0.6840863654538184
k=7 score=0.6971078235372518
k=8 score=0.7134346261495401
k=9 score=0.6938557910169265
k=10 score=0.7085699053711847
k=11 score=0.7069305611088896
k=12 score=0.7329734772757563
k=13 score=0.710169265627082 5
k=14 score=0.7231640677062509
k=15 score=0.7133946421431427
[0.6449020391843263, 0.6921764627482341, 0.6937758230041318, 0.7003198720511
795, 0.7051979208316673, 0.6840863654538184, 0.6971078235372518, 0.713434626
1495401, 0.6938557910169265, 0.7085699053711847, 0.7069305611088896, 0.73297
34772757563, 0.7101692656270825, 0.7231640677062509, 0.7133946421431427]
best k=12
best test k=15
```

Out[80]:

```
<matplotlib.legend.Legend at 0x1536cbca0>
```



In [81]:

```python
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(X_train, y_train)
score = knn.score(X_test, y_test)
print(f'k={best_k} test_error={1-score}')
```

```
k=12 test_error=0.2272727272727273
```

In [82]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
print(X_train.shape, X_test.shape)
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

knn.fit(X_train_norm, y_train)
score = knn.score(X_test_norm, y_test)
print(f'k={best_k} standardized test_error={1-score}')
```

```
(614, 8) (154, 8)
k=12 standardized test_error=0.20779220779220775
```

Q4

- Yes, centralization and standardization affects the data.
- This is because KNN uses raw feature values, hence if some value is much larger than the others, it would dominate the outcome. We can clearly see values of the feature "Insulin" is much higher than "DPF", thus without normalization "Insulin" would have higher weightage. If such scale difference is not desirable, normalization would give better results.

Problem 4

Q1

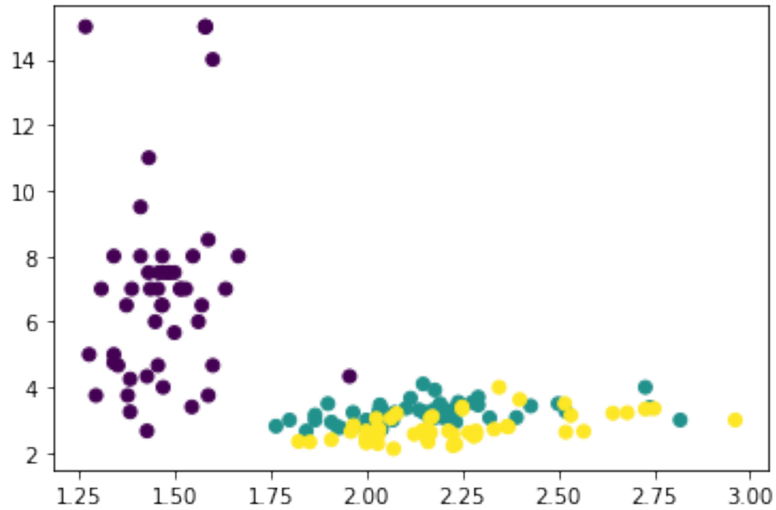
Note: I am not standardizing the input.

In [126…

```
import math

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

%matplotlib inline

filename = 'iris-2.data'
data = pd.read_csv(filename, names=['sepal_length', 'sepal_width', 'petal_le
data['sepal_ratio'] = data['sepal_length'] / data['sepal_width']
data['petal_ratio'] = data['petal_length'] / data['petal_width']
df = data.drop(['sepal_length', 'sepal_width', 'petal_length', 'petal_width'
df['class'], _ = pd.factorize(df['class'])
# df.head()

x = df['sepal_ratio']
y = df['petal_ratio']
df['color'] = df['class'].replace({
    'Iris-setosa': 'red',
    'Iris-versicolor': 'blue',
    'Iris-virginica': 'green'
})
plt.scatter(x, y, c=df['color'])
```

Out[126]:

```
<matplotlib.collections.PathCollection at 0x12c8821f0>
```



Q2

In [127…

```
from math import dist

from pandas import Series
np.random.seed(18)
df['coords'] = list(zip(x, y))

def dx2(coord1, centroid):
    # return np.sqrt((coord1[0] - centroid[0]) ** 2 + (coord1[1] - centroid[
    return (coord1[0] - centroid[0]) ** 2 + (coord1[1] - centroid[1]) ** 2

new_df = df[["sepal_ratio", "petal_ratio"]].to_numpy(dtype='float32')[:,:2]

#kmeans++ algorithm
def kpp_init(k):
    # Take one center
    _centroids = new_df[np.random.choice(df.shape[0], 1), :]

    for _ in range(k-1):
        dists = cdist(new_df, _centroids, 'euclidean')
        # for each point, get the closest centroid
        closest_cluster = np.min(dists, axis=1)
        # keep track of the furthest point from their closest centroid
        farthest_point = new_df[[np.argmax(closest_cluster)], :]
        # update _centroids
        _centroids = np.append(_centroids, farthest_point, axis=0)
    return _centroids

# noinspection PyPep8Naming
def kmeanspp_centroid(coords: Series, k):
    # Take one center
    # centroid = coords.sample().iloc[0]
    # _centroids = [centroid]
    _centroids = coords.sample()
```

```
    chosen = [_centroids.index[0]]

    # Take k-1 new centers with prob D(x)^2 / sum of D(x)^2
    # weights signify the prob distribution
    # while len(_centroids) < k:
    #     prob_dist = df['coords'].apply(dx2, args=(centroid,))
    #     centroid = df.sample(weights=prob_dist)['coords'].iloc[0]
    #     _centroids.append(centroid)

    while len(_centroids) < k:
        max_dist = -1
        farthest_point = None
        for index, coord in coords.items():
            # for each point, get the closest centroid
            closest_centroid = np.argmin([(coord[0] * centroid[0]) ** 2 + (c
                                            for centroid in _centroids])
            # print(f'{k=} index of closest cent={closest_centroid}')
            d = dist(coord, _centroids.iloc[closest_centroid])
            # keep track of the furthest point from their closest centroid
            if d > max_dist and index not in chosen:
                max_dist = d
                farthest_point = coord
                chosen.append(index)
        # update _centroids
        _centroids = pd.concat([_centroids, pd.Series([farthest_point], dtyp

    return _centroids

# centroids = {}
# for _k in range(1, 6):
#     centroids[_k] = kmeanspp_centroid(df['coords'], _k)
# pprint(centroids)
```

Q3

Note: DBI score is used as the clustering objective.

In [128…

```
from scipy.spatial.distance import cdist
from sklearn.metrics import davies_bouldin_score
%matplotlib inline

feature_df = pd.concat([x, y], axis=1)


# def update_centroids(_df, centers, _centroids):
#     sums = {}
#     n = len(centers)
#     for i in range(n):
#         coords = _df.iloc[i]['coords']
#         center = centers[i]
#         if center not in sums:
```

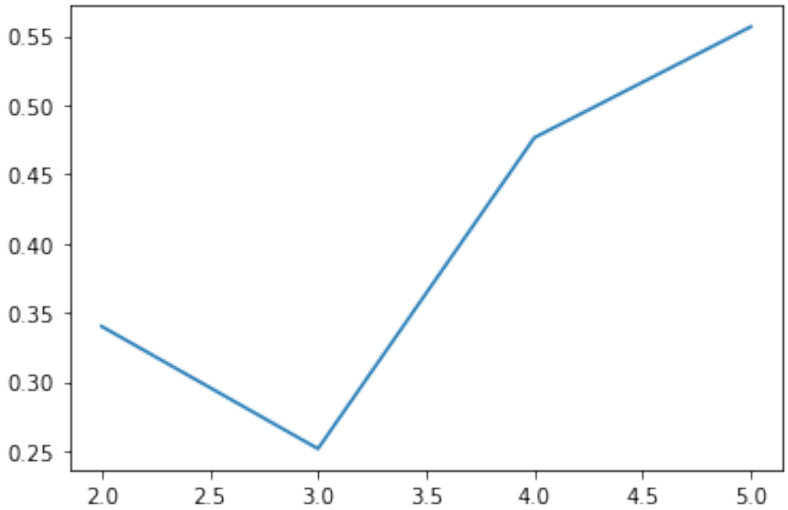
```
#             sums[center] = (0, 0)
#         sums[center] = (coords[0] + sums[center][0], coords[1] + sums[cent
#     for i, _sum in sums.items():
#         _centroids.iloc[i] = (_sum[0] / n, _sum[1] / n)
#     return _centroids


def dist2(coords, _centroids):
    dists = [math.dist(coords, centroid) for centroid in _centroids]
    return dists


def kmeans(k, break_early=True):
    scores = []
    # centroids = kmeanspp_centroid(df['coords'], k)
    centroids = kpp_init(k)

    distances = df['coords'].apply(dist2, args=(centroids,))
    points = np.array([np.argmin(d) for d in distances])
    df['centers'] = points

    for _ in range(50):
        prev_points = points
        centroids = []
        for idx in range(k):
            filtered_df = df[df['centers'] == idx]
            if len(filtered_df) == 0:
                print('dead')
            cent_x = filtered_df['sepal_ratio'].mean(axis=0)
            cent_y = filtered_df['petal_ratio'].mean(axis=0)
            centroids.append((cent_x, cent_y))
        centroids = np.vstack(centroids)
        distances = df['coords'].apply(dist2, args=(centroids,))
        points = np.array([np.argmin(d) for d in distances])

        if not break_early:
            # For plotting, we need scores for all iterations
            score = davies_bouldin_score(feature_df, points)
            scores.append(score)
        elif np.array_equal(points, prev_points):
            # Otherwise, we're only interested in the final score
            # print(f'Finished after {i} iterations')
            break

    # print(f'{k=} centers={pd.unique(points)}')
    dbi = davies_bouldin_score(feature_df, points)
    print(f'{k=} {dbi=}')

    return dbi, scores, points, centroids

    # prev = pd.Series(dtype=int)
    # centers = pd.Series(dtype=int)
    # for i in range(50):
```

```
    #     # print(centroids[k])
    #     centers = df['coords'].apply(dist2, args=(centroids,))
    #     print(f'{k=} centers={pd.unique(centers)}')
    #     centroids = update_centroids(df, centers, centroids)
    #     if prev.equals(centers):
    #         # print(f'{i=} {prev=} {centers=}')
    #         print(f'Finished in {i} iterations')
    #         break
    #     prev = centers
    #
    # print(f'{k=} centers={pd.unique(centers)}')
    # scores[k] = davies_bouldin_score(feature_df, centers)
    # print(f'{k=} accuracy={scores[k]}')
#
#     # cmap = {
#     #     0: 'purple',
#     #     1: 'red',
#     #     2: 'green',
#     #     3: 'blue',
#     #     4: 'orange',
#     # }
#     # df['color'] = centers.replace(cmap)
#     # # df[df['coords'] in centroids[k]]['color'] = 'black'
#     # plt.scatter(x, y, c=df['color'])
#     # plt.show()
#
#     # accuracy = accuracy_score(df['class'], centers)
#     # print(f'{k=} {accuracy=}')
#

_x = []
_y = []
for i in range(2, 6):
    acc, _, _, _ = kmeans(k=i)
    _x.append(i)
    _y.append(acc)

plt.plot(_x, _y)
```

```
k=2 dbi=0.34024794906992745
k=3 dbi=0.25177597158316023
k=4 dbi=0.47653943938097953
k=5 dbi=0.5566814573344967
```

Out[128]:

```
[<matplotlib.lines.Line2D at 0x12c732700>]
```



Q4

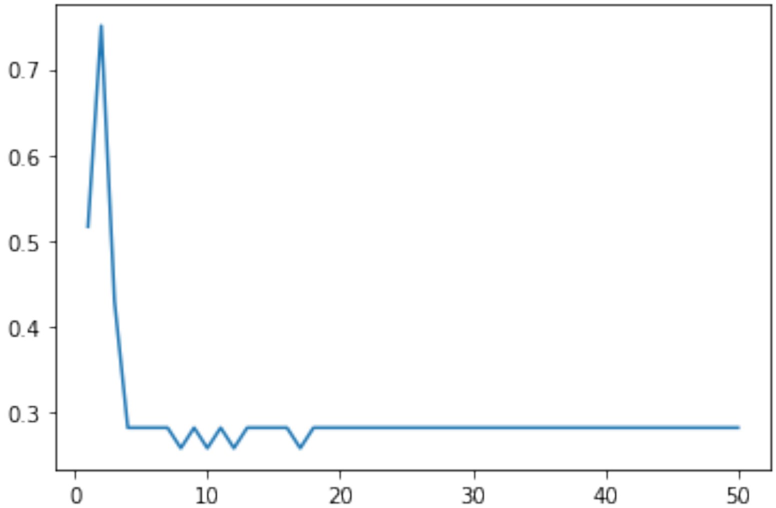
- Cluster size of 3 was chosen as the DBI score for $k = 3$ was the lowest. This is expected since the original dataset also has 3 clusters, thus our algorithm is correctly guessing the optimum number of clusters to some extent.

In [129…

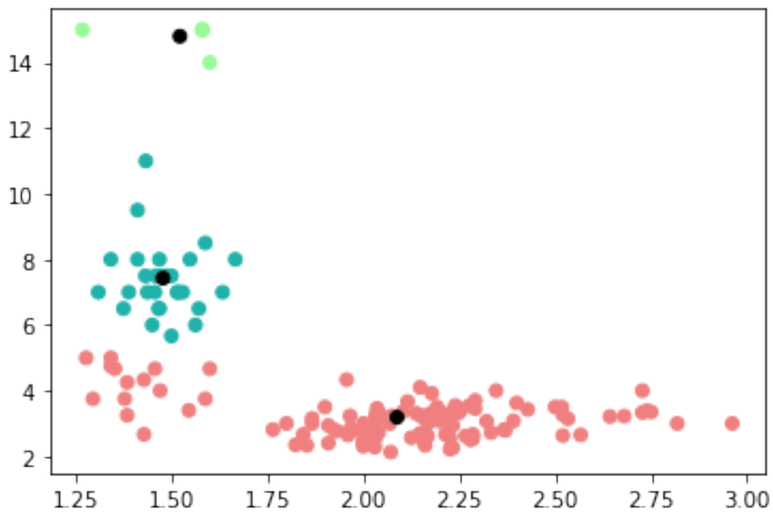
In [130…

```
clusters = 3
_, scores, centers, coords = kmeans(clusters, break_early=False)
scores = dbis
_x = range(1, 51)
_y = scores

colors = []
all_colors = ['lightcoral', 'palegreen', 'lightseagreen', 'hotpink', 'orange
for i, center in enumerate(centers):
    colors.append(all_colors[center])

plt.plot(_x, _y); plt.show()

_x = np.array(x)
_y = np.array(y)
for coord in coords:
    _x = np.append(_x, coord[0])
    _y = np.append(_y, coord[1])
    colors.append('black')
# plt.xlim(0, 14)
plt.scatter(_x, _y, c=colors)
```

```
k=3 dbi=0.2569306928383754
```



Out[130]: <matplotlib.collections.PathCollection at 0x12c384ee0>



In [130…

In [72]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
```

In [73]:
```
_train_data = pd.read_csv("optdigits.tra", header=None)
_test_data = pd.read_csv("optdigits.tes", header=None)

train_data = _train_data.drop(columns=_train_data.columns[-1], axis=1)
train_labels = _train_data[_train_data.columns[-1:]]

test_data = _test_data.drop(columns=_test_data.columns[-1], axis=1)
test_labels = _test_data[_test_data.columns[-1:]]

train_features = train_data.to_numpy()
train_labels = train_labels.to_numpy()
train_features = np.concatenate((np.ones((train_data.shape[0], 1)), train_fe
labels = np.unique(train_labels)

test_features = test_data.to_numpy()
test_labels = test_labels.to_numpy()
test_features = np.concatenate((np.ones((test_data.shape[0], 1)), test_featu
```

In [74]:
```
def one_vs_all(decision, labels):
    decisions = []
    for label in labels:
        label_class = np.where(decision == label, 1, 0)
        decisions.append(label_class)
    return decisions


_train = one_vs_all(train_labels, labels)
_test = one_vs_all(test_labels, labels)
```

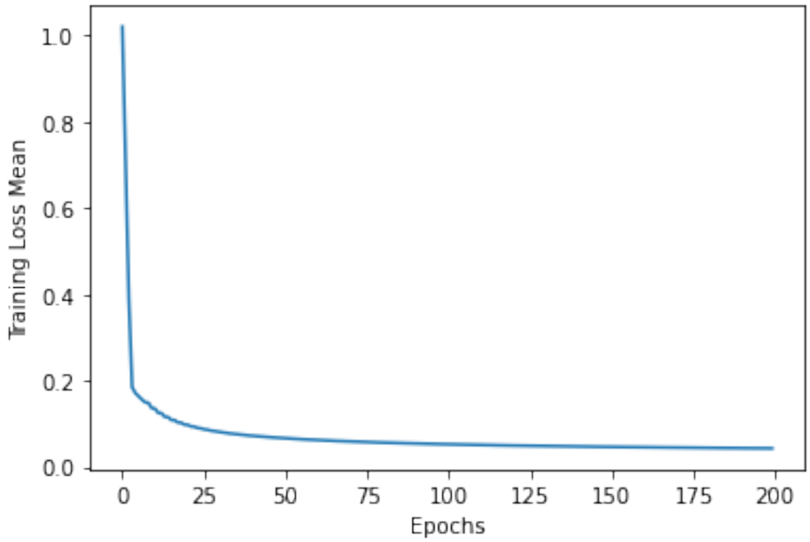
In [75]:
```
weights = []

def sigmoid(z):
        return 1 / (1 + np.exp(-z))


def loss(y, y_pred):
    losses = (y * np.log(y_pred)) + ((1 - y) * np.log(1 - y_pred))
    return -np.mean(losses)


def train(train_features, train_labels, test_features, test_labels, lam, alp
    weights = np.zeros((1, train_features.shape[1]))
    train_errors = []
    test_errors = []
    for epoch in range(max_epochs):
```

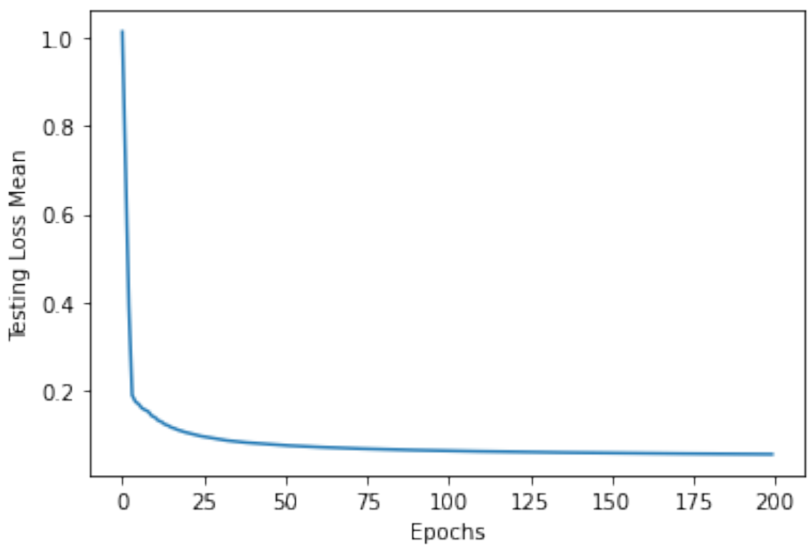
```
        wx = np.dot(train_features, weights.T)
        y_pred_train = sigmoid(wx)

        wx = np.dot(test_features, weights.T)
        y_pred_test = sigmoid(wx)

        dw = np.dot((y_pred_train - train_labels).T, train_features) * (1 /

        delta = dw + (lam * weights)

        weights = weights - (alpha * delta)

        wx = np.dot(train_features, weights.T)
        y_pred_train = sigmoid(wx)

        wx = np.dot(test_features, weights.T)
        y_pred_test = sigmoid(wx)

        train_loss = loss(train_labels, y_pred_train)
        test_loss = loss(test_labels, y_pred_test)
        train_errors.append(train_loss)
        test_errors.append(test_loss)
    return weights, train_errors, test_errors


epochs = 200
_training_losses = np.empty((10, epochs))
_test_losses = np.empty((10, epochs))
for i in labels:
    weight, train_loss, test_loss = train(train_features, _train[i],
                                          test_features, _test[i], 0, 0.01,
                                          epochs)
    _training_losses[i, :] = train_loss
    _test_losses[i, :] = test_loss
    weights.append(weight)

training_losses = np.mean(_training_losses, axis=0)
testing_losses = np.mean(_test_losses, axis=0)
```

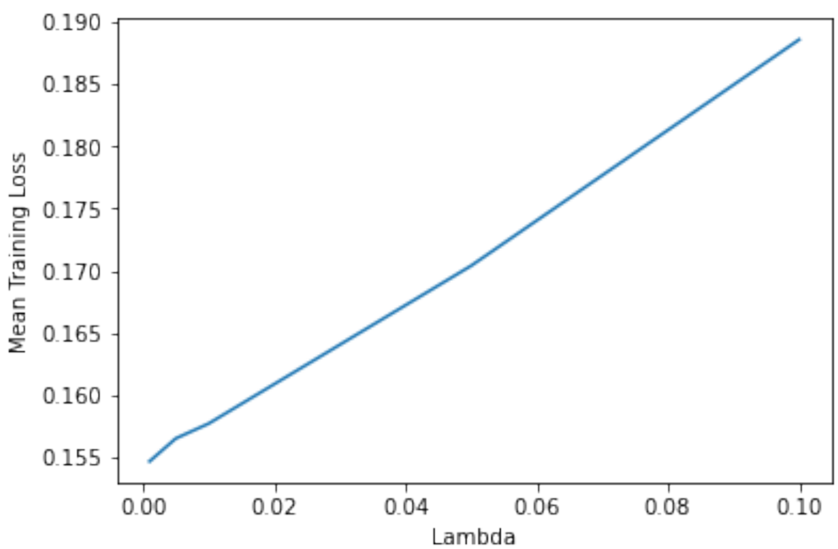
In [76]:

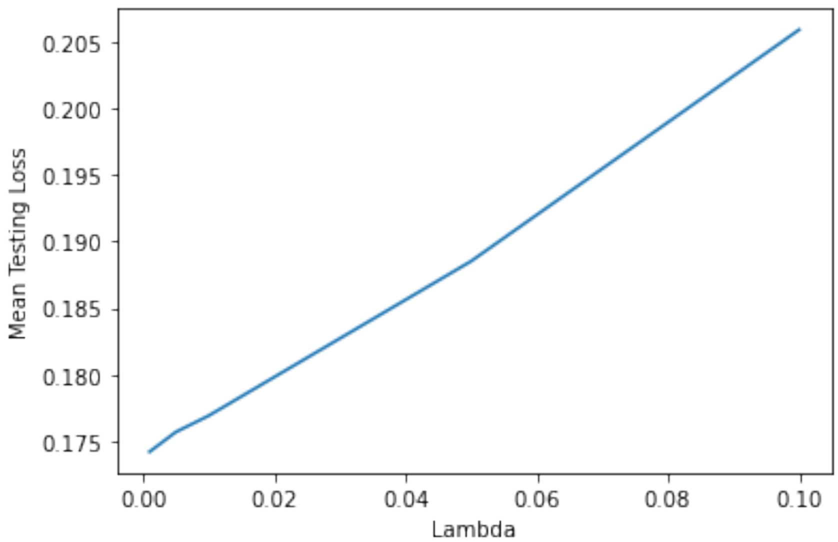
```
def predict(features_array_train, features_array_test, decision_train, decis
            one_vs_all_class_weights):

    def _predict(features, weight):
        wx = np.dot(features, weight.T)
        y_pred = sigmoid(wx)
        return y_pred

    train_prediction_list = []
    for i in labels:
        prediction_i = _predict(features_array_train, one_vs_all_class_weigh
        train_prediction_list.append(prediction_i)

    train_prediction_array = np.asarray(train_prediction_list)
```

```
    train_preds = np.argmax(train_prediction_array, axis=0)
    # training_accuracy = accuracy_score(decision_train, train_preds)

    test_prediction_list = []
    for i in labels:
        prediction_i = _predict(features_array_test, one_vs_all_class_weight
        test_prediction_list.append(prediction_i)

    test_prediction_array = np.asarray(test_prediction_list)
    test_preds = np.argmax(test_prediction_array, axis=0)
    # test_accuracy = accuracy_score(decision_test, test_preds)


plt.plot(range(epochs), training_losses)
plt.xlabel('Epochs')
plt.ylabel('Training Loss Mean')
plt.show()

plt.plot(range(epochs), testing_losses)
plt.xlabel('Epochs')
plt.ylabel('Testing Loss Mean')
plt.show()

predict(train_features, test_features, train_labels, test_labels,
        weights)

lambdas = [0.001, 0.005, 0.01, 0.05, 0.1]

training_loss = []
test_loss = []

for lam in lambdas:
    _training_loss = []
    _testing_loss = []
    for i in labels:
        weight, train_loss, _test_loss = train(train_features, _train[i],
                                               test_features, _test[i], lam,
                                               epochs)
        _training_loss.append(train_loss)
        _testing_loss.append(_test_loss)
    training_loss.append(np.mean(_training_loss))
    test_loss.append(np.mean(_testing_loss))

plt.plot(lambdas, training_loss)
plt.xlabel('Lambda')
plt.ylabel('Mean Training Loss')
plt.show()

plt.plot(lambdas, test_loss)
plt.xlabel('Lambda')
plt.ylabel('Mean Testing Loss')
plt.show()
```


Training Loss Mean
Epochs
0.0
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200


Testing Loss Mean
Epochs
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200


Mean Training Loss
Lambda
0.155
0.160
0.165
0.170
0.175
0.180
0.185
0.190
0.00
0.02
0.04
0.06
0.08
0.10



Mean Testing Loss
0.175
0.180
0.185
0.190
0.195
0.200
0.205
0.00
0.02
0.04
0.06
0.08
0.10
Lambda

In [76]: