threeCluster

May 16, 2023

[ ]:
```
import numpy as np
from numpy.random import multivariate_normal as mvn

import matplotlib.pyplot as plt

from scipy.stats import norm
from scipy.optimize import minimize

from math import log
from math import dist
from math import floor

from random import shuffle

from tqdm import tqdm

from weights import KNN
from weights import proximity

from accuracy import KNN_acc
from accuracy import Prox_acc
```

1 Complete Three Cluster Example

1.1 Generating the Points

We first write a function to generate three well-separated clusters, depending on the number of points desired and the centers and covariance matrix of the distributions

[ ]:
```
def clusters(m3, centers, covar):
        # m3 is the number of points around each distribution

        knownvals = [int(j*m3) for j in range(3)] # Points for which we know
 ↪the value of the label

        X = mvn(centers[0],covar, m3)
        X = np.append(X,mvn(centers[1],covar, m3), axis=0)
```

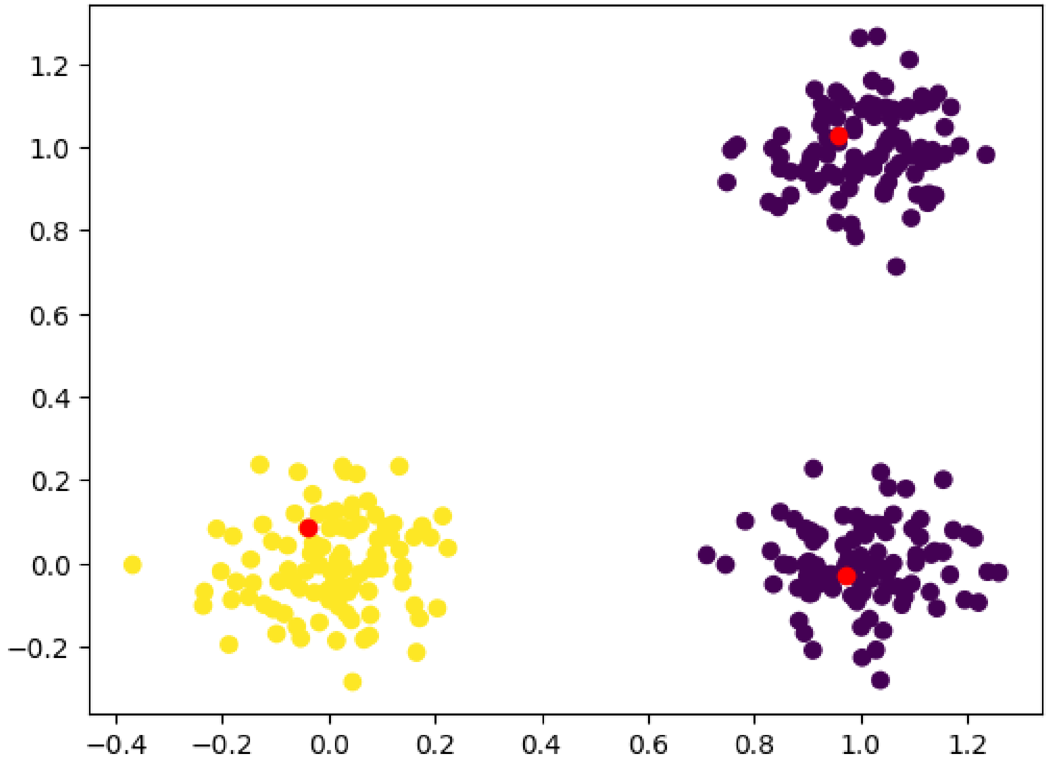
```
        X = np.append(X,mvn(centers[2],covar, m3), axis=0)

        y = [1 for i in range(m3)]
        y = y + [-1 for i in range(2*m3)]

        return X, y, knownvals
```

We use this to generate the points

[ ]:
```
M = 300 # Multiple of 3

centers = [[0,0],[1,0],[1,1]]
covar = 0.01*np.identity(2)

X, y, knownvals = clusters(int(M/3), centers, covar)
```

[ ]:
```
xs = X[:,0]
ys = X[:,1]

xs_k = [xs[j] for j in knownvals]
ys_k = [ys[j] for j in knownvals]

plt.scatter(xs, ys, c=y, cmap = "viridis")
plt.scatter(xs_k, ys_k, color="red")
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x1731de4f4d0>
```



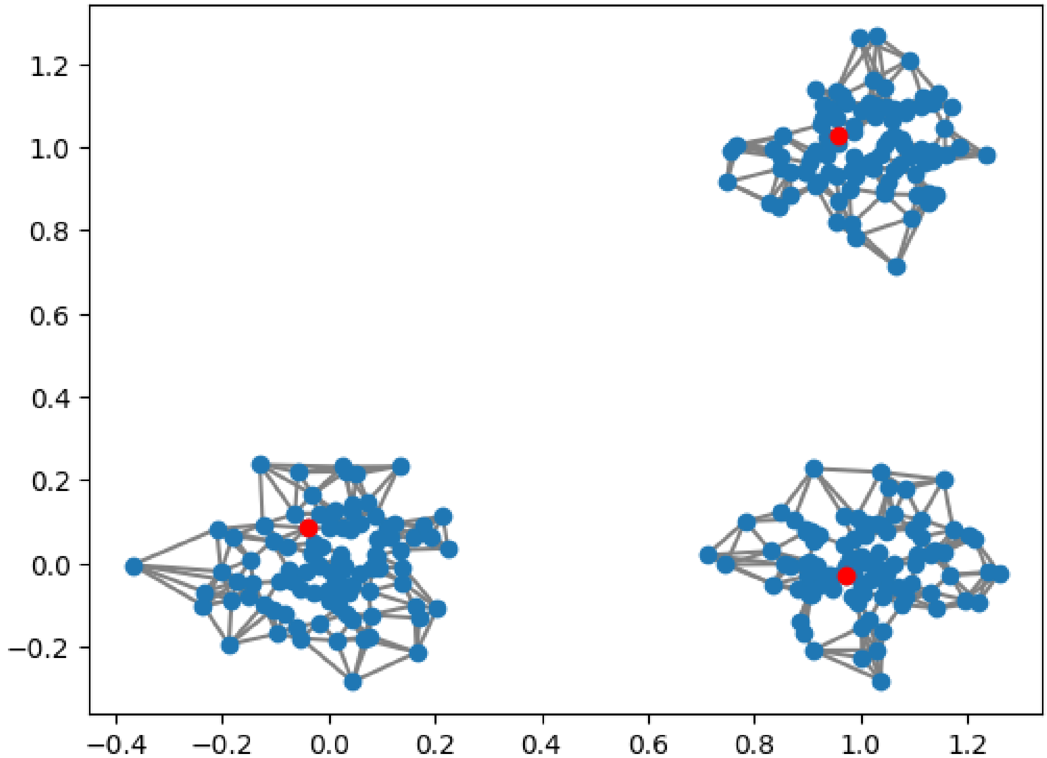
1.2 Build a Graph on the Points and Run Regression

To build a graph on the points, we have some choices to make: the choice of weight function and its parameters and whether we use a KNN approach or a full proximity graph approach

Here is an example of some of these graphs

1.2.1 KNN with uniform kernel

[ ]:
```
unif = lambda x1, x2: 1
k = 5

L, W = KNN(X, M, k, unif)

plt.scatter(xs, ys)
plt.scatter(xs_k, ys_k, color="red")

for i in range(M):
        for j in range(i,M):
                if W[i,j] > 0:
                        plt.plot([X[i,0],X[j,0]],[X[i,1],X[j,1]], color =
 ↪"gray", zorder = 0)
```



Performing the classification, we see that the model performs well with both probit and regression loss. Note that it was not necessary to tune parameters to get good results.

[ ]:
```
def probit(kvals, y, f): # Probits Loss
        return -sum([log(norm.cdf(y[j]*f[j])) for j in kvals])

def regression(kvals, y, f): # Regression Loss
        return sum([(y[j]-f[j])**2 for j in kvals])

def regular(lamb,C_inv,f): # Regularization
        f_T = np.array(f).T
        return lamb*f_T.dot(C_inv).dot(f)

def to_minimize(f,kvals,y,lamb,C_inv, loss): # Funtional to minimize
    return loss(kvals,y,f) + regular(lamb,C_inv,f)
```

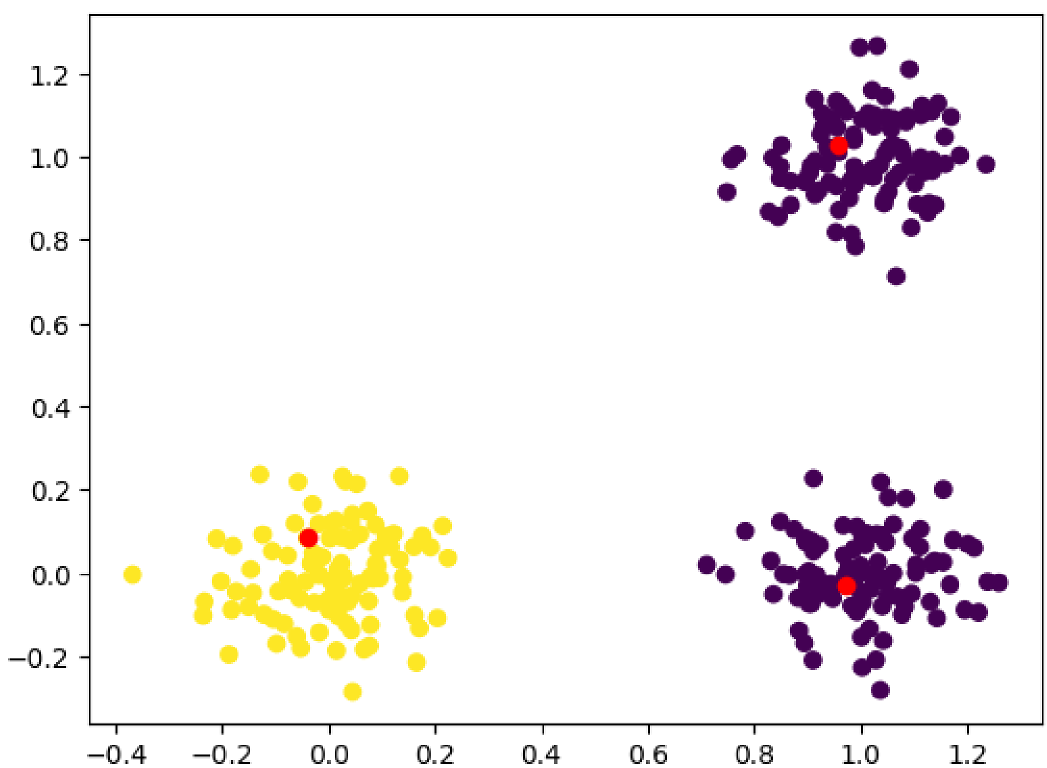
[ ]:
```
tau = 1
alpha = 2
lamb = (tau**(2*alpha))/2
C = np.linalg.matrix_power(((L + (tau**2)*np.eye(M))),-alpha)
C_inv = np.linalg.inv(C)
```

Probit Loss

[ ]:
```
loss = probit

f0 = np.zeros(M)
result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss),␣
 ↪method='BFGS') # Perform minimization

f_star = result.x
y_pred = np.sign(f_star) # Predicted labels
```

[ ]:
```
xs = X[:,0]
ys = X[:,1]

xs_k = [xs[j] for j in knownvals]
ys_k = [ys[j] for j in knownvals]

plt.scatter(xs, ys, c=y_pred, cmap = "viridis")
plt.scatter(xs_k, ys_k, color="red")
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x1732287d490>
```



```
[ ]: accuracy = sum([x[0] == x[1] for x in zip(y_pred,y)])/M
     print(accuracy)
```

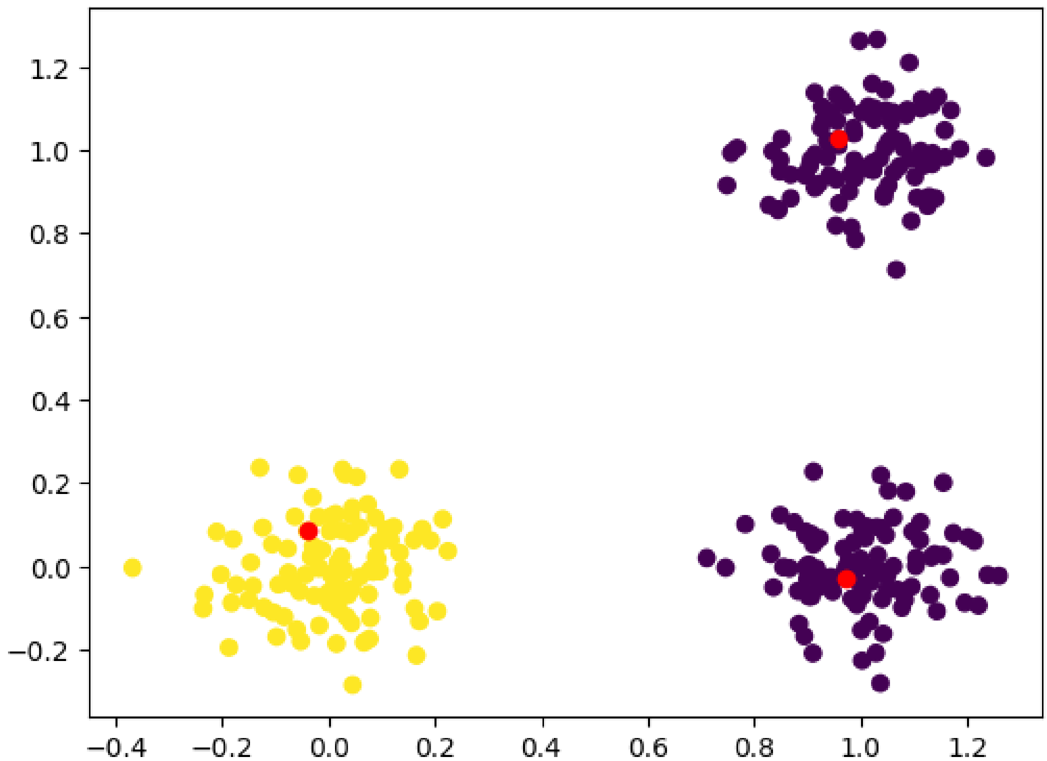
```
1.0
```

Regression Loss

```
[ ]: loss = regression

     f0 = np.zeros(M)
     result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss),␣
      ↪method='BFGS') # Perform minimization

     f_star = result.x
     y_pred = np.sign(f_star) # Predicted labels
```

```
[ ]: xs = X[:,0]
     ys = X[:,1]

     xs_k = [xs[j] for j in knownvals]
     ys_k = [ys[j] for j in knownvals]

     plt.scatter(xs, ys, c=y_pred, cmap = "viridis")
     plt.scatter(xs_k, ys_k, color="red")
```

```
[ ]: <matplotlib.collections.PathCollection at 0x1732287c690>
```




```
[ ]: accuracy = sum([x[0] == x[1] for x in zip(y_pred,y)])/M
     print(accuracy)
```

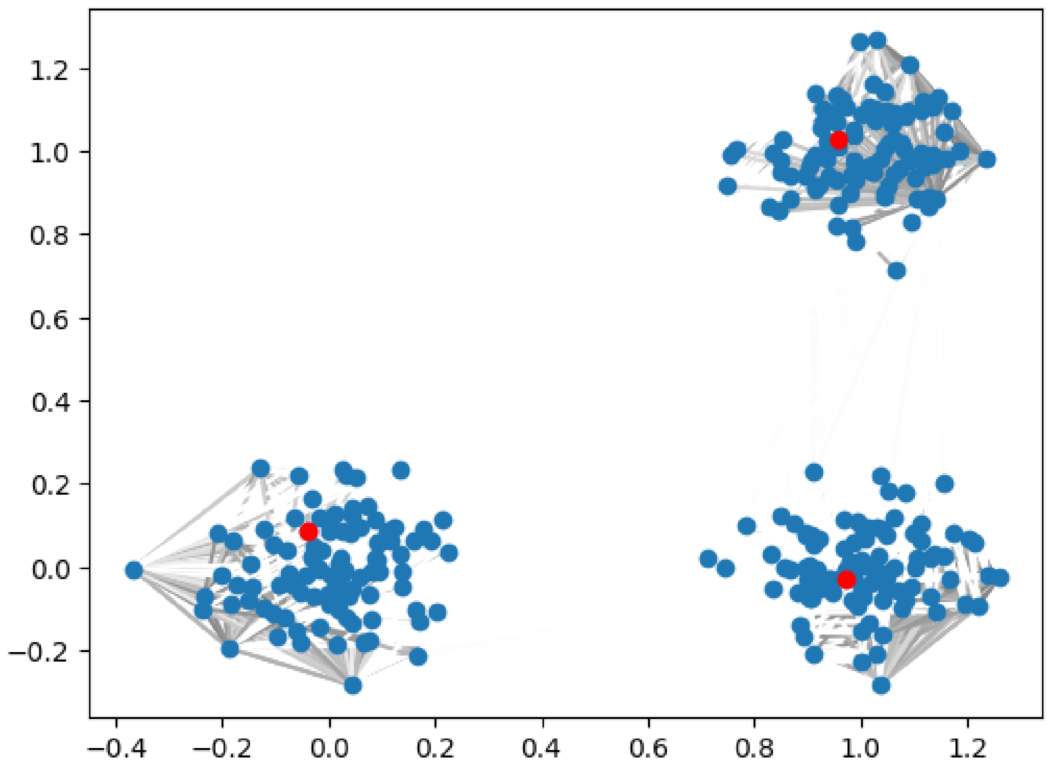
1.0

1.2.2 Proximity with RBF Kernel

We first need to make a guess at the gamma parameter in the RBF kernel. We use the first quartile of the distances between vertices. Then we adjust using our multplier

```
[ ]: dists = []
     for x1 in X:
             for x2 in X:
                     dists += [dist(x1,x2)]
     dists.sort()

     mult = 16
     gamma = dists[floor(len(dists)/4)]*mult

     rbf = lambda x1, x2: np.exp(gamma**2*-0.5*dist(x1,x2)**2)
```

[ ]:
```
L, W = proximity(X, M, 2, rbf)

plt.scatter(xs, ys)
plt.scatter(xs_k, ys_k, color="red")

l1 = [*range(M)] # Shuffle vertices to make plot more even
shuffle(l1)
l2 = [*range(M)]
shuffle(l2)

for i in l1:
        for j in l2:
                if W[i,j] > 0:
                        plt.plot([X[i,0],X[j,0]],[X[i,1],X[j,1]], color = str(1␣
 ↪- 0.5*W[i,j]), zorder = 0)
```



Now we do the classification

We need to tune the tau parameter here, we want tau^2 to be on the order of epsilon

[ ]:
```
W2 = np.copy(W)
W2 = W2.flatten()
W2.sort()
```

```
tau = W2[floor(len(W2)/2)]**(1/2)
```

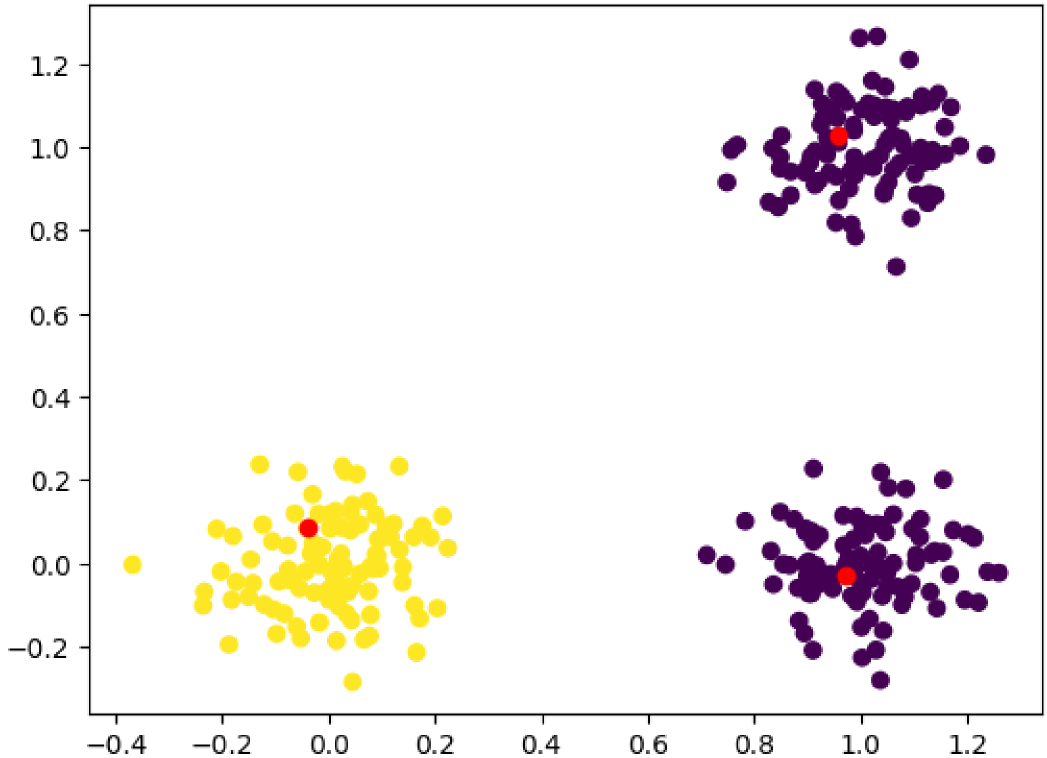
[ ]:
```
alpha = 2
lamb = (tau**(2*alpha))/2
C = np.linalg.matrix_power(((L + (tau**2)*np.eye(M))),-alpha)
C_inv = np.linalg.inv(C)
```

Probit Loss

[ ]:
```
loss = probit

f0 = np.zeros(M)
result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss),
    method='BFGS') # Perform minimization

f_star = result.x
y_pred = np.sign(f_star) # Predicted labels
```

[ ]:
```
xs = X[:,0]
ys = X[:,1]

xs_k = [xs[j] for j in knownvals]
ys_k = [ys[j] for j in knownvals]

plt.scatter(xs, ys, c=y_pred, cmap = "viridis")
plt.scatter(xs_k, ys_k, color="red")
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x17369dadad0>
```



[ ]:
```
accuracy = sum([x[0] == x[1] for x in zip(y_pred,y)])/M
print(accuracy)
```

```
1.0
```

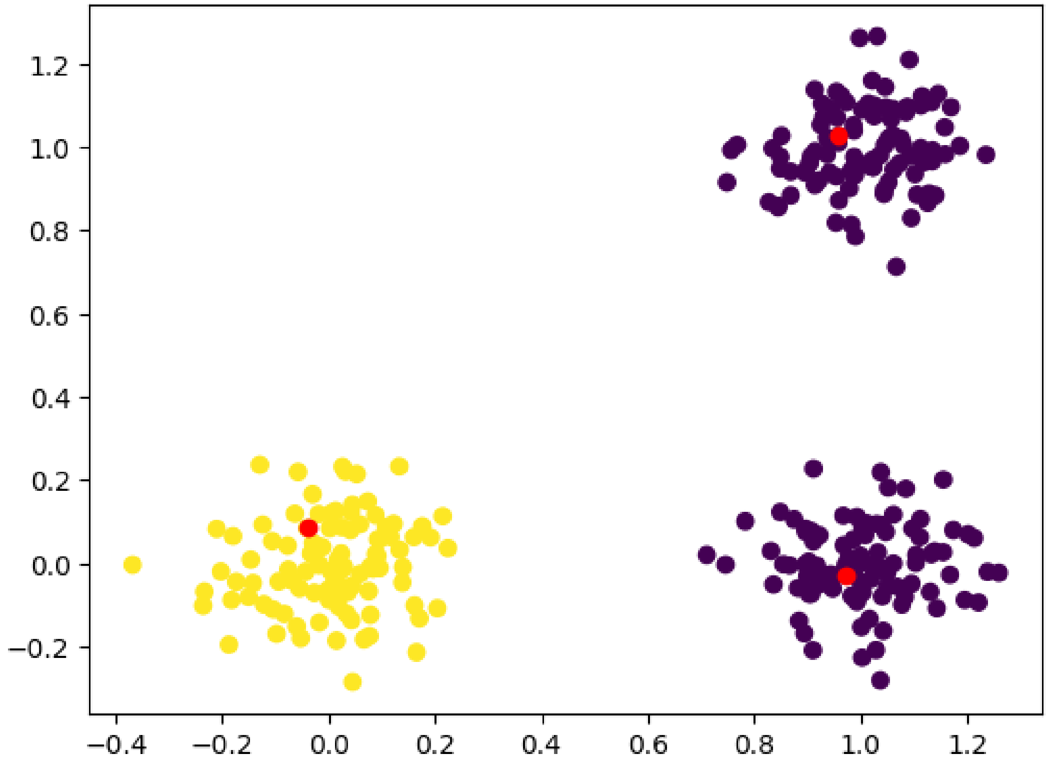
1.2.3 Regression Loss

[ ]:
```
loss = regression

f0 = np.zeros(M)
result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss),␣
↪method='BFGS') # Perform minimization

f_star = result.x
y_pred = np.sign(f_star) # Predicted labels
```

[ ]:
```
xs = X[:,0]
ys = X[:,1]

xs_k = [xs[j] for j in knownvals]
ys_k = [ys[j] for j in knownvals]
```

```
plt.scatter(xs, ys, c=y_pred, cmap = "viridis")
plt.scatter(xs_k, ys_k, color="red")
```

[ ]: <matplotlib.collections.PathCollection at 0x17362252350>



[ ]:
```
accuracy = sum([x[0] == x[1] for x in zip(y_pred,y)])/M
print(accuracy)
```

```
1.0
```

1.3 Validation of accuracy with multiple trials

We run this process 50 times for each case to confirm this result

1.3.1 KNN Graph

[ ]:
```
M = 300 # Multiple of 3

centers = [[0,0],[1,0],[1,1]]
covar = 0.01*np.identity(2)

k = 5
tau = 1
```

```
alpha = 2
lamb = (tau**(2*alpha))/2

sum_acc = 0
for j in tqdm(range(50)):
        X, y, knownvals = clusters(int(M/3), centers, covar)

        sum_acc += KNN_acc(X, y, knownvals, alpha = alpha, tau = tau, lossf =␣
 ↪"probit", k = k, kernel = unif)

probit_accuracy = sum_acc/50

sum_acc = 0
for j in tqdm(range(50)):
        X, y, knownvals = clusters(int(M/3), centers, covar)

        sum_acc += KNN_acc(X, y, knownvals, alpha = alpha, tau = tau, lossf =␣
 ↪"regression", k = k, kernel = unif)

regression_accuracy = sum_acc/50
```

```
100%|      | 50/50 [09:13<00:00, 11.07s/it]
100%|      | 50/50 [04:40<00:00,  5.62s/it]
```

```
1.0
1.0
```

[ ]:
```
print(probit_accuracy)
print(regression_accuracy)
```

```
1.0
1.0
```

1.3.2 Proximity Graph

[ ]:
```
M = 300 # Multiple of 3

centers = [[0,0],[1,0],[1,1]]
covar = 0.01*np.identity(2)

tau = 1
alpha = 2
lamb = (tau**(2*alpha))/2

sum_acc = 0
for j in tqdm(range(50)):
```

```
        X, y, knownvals = clusters(int(M/3), centers, covar)

        dists = []
        for x1 in X:
                for x2 in X:
                        dists += [dist(x1,x2)]
        dists.sort()

        mult = 16
        gamma = dists[floor(len(dists)/4)]*mult

        rbf = lambda x1, x2: np.exp(gamma**2*-0.5*dist(x1,x2)**2)

        L, W = proximity(X, M, 2, rbf)

        W2 = np.copy(W)
        W2 = W2.flatten()
        W2.sort()
        tau = W2[floor(len(W2)/2)]**(1/2)

        alpha = 2
        lamb = (tau**(2*alpha))/2
        C = np.linalg.matrix_power(((L + (tau**2)*np.eye(M))),-alpha)
        C_inv = np.linalg.inv(C)

        loss = probit

        f0 = np.zeros(M)
        result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss), method='BFGS') # Perform minimization

        f_star = result.x
        y_pred = np.sign(f_star) # Predicted labels

        sum_acc += sum([x[0] == x[1] for x in zip(y_pred,y)])/M

probit_accuracy = sum_acc/50

sum_acc = 0
for j in tqdm(range(50)):
        X, y, knownvals = clusters(int(M/3), centers, covar)

        dists = []
        for x1 in X:
                for x2 in X:
                        dists += [dist(x1,x2)]
        dists.sort()
```

```
        mult = 16
        gamma = dists[floor(len(dists)/4)]*mult

        rbf = lambda x1, x2: np.exp(gamma**2*-0.5*dist(x1,x2)**2)

        L, W = proximity(X, M, 2, rbf)

        W2 = np.copy(W)
        W2 = W2.flatten()
        W2.sort()
        tau = W2[floor(len(W2)/2)]**(1/2)

        alpha = 2
        lamb = (tau**(2*alpha))/2
        C = np.linalg.matrix_power(((L + (tau**2)*np.eye(M))),-alpha)
        C_inv = np.linalg.inv(C)

        loss = regression

        f0 = np.zeros(M)
        result = minimize(to_minimize, f0, args=(knownvals,y,lamb,C_inv,loss),
 ↪method='BFGS') # Perform minimization

        f_star = result.x
        y_pred = np.sign(f_star) # Predicted labels

        sum_acc += sum([x[0] == x[1] for x in zip(y_pred,y)])/M

regression_accuracy = sum_acc/50
```

```
100%|      | 50/50 [14:44<00:00, 17.70s/it]
100%|      | 50/50 [04:18<00:00,  5.17s/it]
```

[ ]:
```
print(probit_accuracy)
print(regression_accuracy)
```

```
1.0
1.0
```

We can see that with the choices made above, both the disconnected and O(Eps) graphs classify the data with 100% accuracy