In [79]:

```
import numpy as np

######## DO NOT MODIFY THIS FUNCTION ########
def draw_rand_label(x, label_list):
    seed = abs(np.sum(x))
    while seed < 1:
        seed = 10 * seed
    seed = int(1000000 * seed)
    np.random.seed(seed)
    return np.random.choice(label_list)
#############################################
```

In [80]:

```
banknote = np.genfromtxt("data_banknote_authentication.txt", delimiter=",")
```

Submission

In [130]:

```
import numpy as np

######## DO NOT MODIFY THIS FUNCTION ########
def draw_rand_label(x, label_list):
    seed = abs(np.sum(x))
    while seed < 1:
        seed = 10 * seed
    seed = int(1000000 * seed)
    np.random.seed(seed)
    return np.random.choice(label_list)
#############################################


class Q1:
    def feature_means(self, banknote):
        avgs = []
        for i in range(banknote.shape[-1] - 1):
            avgs.append(np.average(banknote[:, i]))
        return avgs

    def covariance_matrix(self, banknote):
        return np.cov(banknote[:, :4], rowvar=False)

    def feature_means_class_1(self, banknote):
        return self.feature_means(banknote[banknote[:, 4] == 1])

    def covariance_matrix_class_1(self, banknote):
        return self.covariance_matrix(banknote[banknote[:, 4] == 1])


class HardParzen:
    def __init__(self, h):
        self.h = h

    def train(self, train_inputs, train_labels):
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.label_list = np.unique(train_labels)

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        counts = np.ones((num_test, len(self.label_list)))
        classes_pred = np.zeros(num_test)

        for obs_index, observation in enumerate(test_data):
            # Get distance from point ex and every training example.
            euclidean_distances = np.sqrt(
                np.sum((self.train_inputs - observation) ** 2, axis=1)
            )

            # Get an array of all the points that are within the parzen window of t
he example point.
            neighbour_indices = euclidean_distances <= self.h

            for category_index, category in enumerate(self.label_list):
                counts[obs_index, category_index] = sum(
                    self.train_labels[neighbour_indices] == category
                )
```

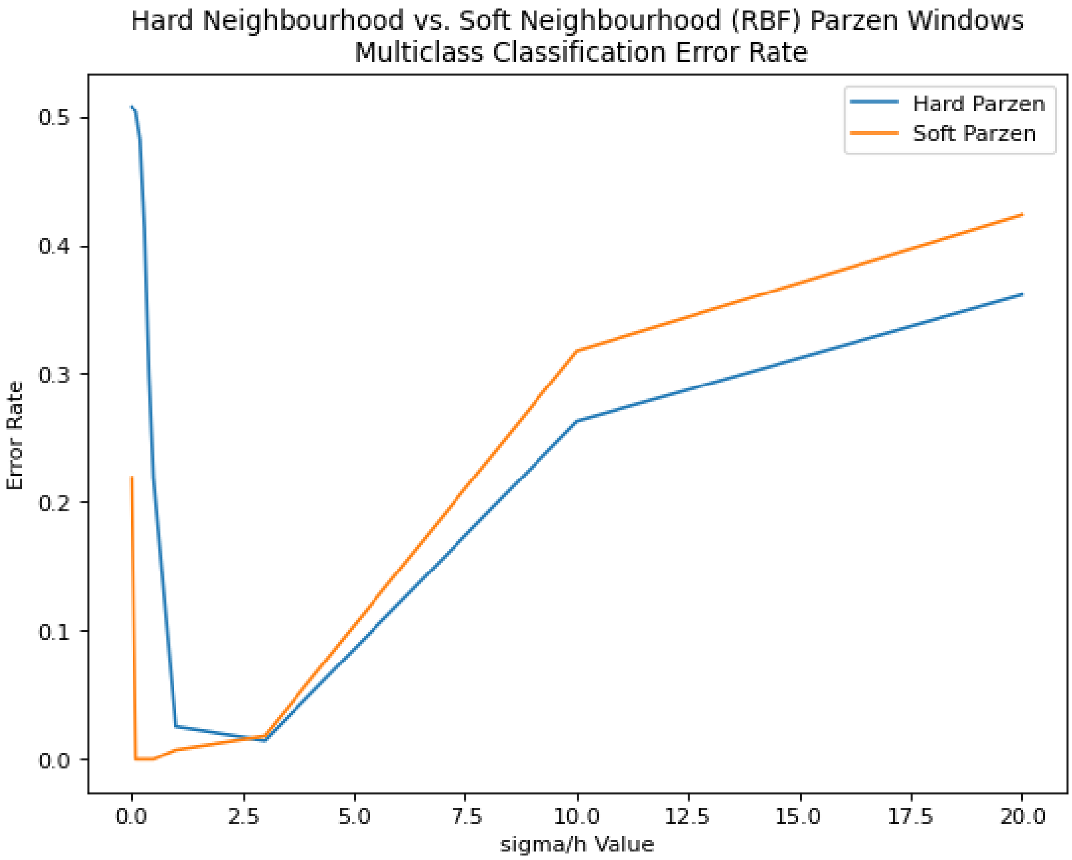
```
            # If there are no points in the window, draw randomly, otherwise perfor
m a vote.
            if np.sum(counts[obs_index, :]) == 0:
                classes_pred[obs_index] = int(
                    draw_rand_label(observation, self.label_list)
                )
            else:
                classes_pred[obs_index] = int(
                    self.label_list[np.argmax(counts[obs_index, :])]
                )

        return classes_pred


class SoftRBFParzen:
    def __init__(self, sigma):
        self.sigma = sigma

    def train(self, train_inputs, train_labels):
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.label_list = np.unique(train_labels)

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        counts = np.zeros((num_test, len(self.label_list)))
        classes_pred = np.zeros(num_test)

        for obs_index, observation in enumerate(test_data):
            # Get distance from point observation and every training example.
            euclidean_distances = np.sqrt(
                np.sum((self.train_inputs - observation) ** 2, axis=1)
            )
            sumk = (
                1
                / (
                    ((2 * np.pi) ** (self.train_inputs.shape[1] / 2))
                    * (self.sigma ** ((self.train_inputs.shape[1])))
                )
            ) * np.exp(-0.5 * ((euclidean_distances ** 2) / (self.sigma ** 2)))

            for category_index, category in enumerate(self.label_list):
                counts[obs_index, category_index] = sum(
                    sumk[self.train_labels == category]
                )

            classes_pred[obs_index] = self.label_list[np.argmax(counts[obs_index,
:])]

        return classes_pred


def split_dataset(banknote):
    train = banknote[[i for i in range(banknote.shape[0]) if i % 5 <= 2]]
    validation = banknote[[i for i in range(banknote.shape[0]) if i % 5 == 3]]
    test = banknote[[i for i in range(banknote.shape[0]) if i % 5 == 4]]
    return (train, validation, test)


class ErrorRate:
    def __init__(self, x_train, y_train, x_val, y_val):
```

```python
        self.x_train = x_train
        self.y_train = y_train
        self.x_val = x_val
        self.y_val = y_val

    def hard_parzen(self, h):
        HP = HardParzen(h)
        HP.train(train_inputs=self.x_train, train_labels=self.y_train)
        preds = HP.compute_predictions(test_data=self.x_val)
        return preds[preds != self.y_val].shape[0] / preds.shape[0]

    def soft_parzen(self, sigma):
        SP = SoftRBFParzen(sigma)
        SP.train(train_inputs=self.x_train, train_labels=self.y_train)
        preds = SP.compute_predictions(test_data=self.x_val)
        return preds[preds != self.y_val].shape[0] / preds.shape[0]


def get_test_errors(banknote):
    train, val, test = split_dataset(banknote)
    ER = ErrorRate(
        x_train=train[:, :4], x_val=val[:, :4], y_train=train[:, 4], y_val=val[:, 4
]
    )

    # Get Errors
    params = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
    hard_errors = []
    soft_errors = []
    for p in params:
        hard_errors.append(ER.hard_parzen(h=p))
        soft_errors.append(ER.soft_parzen(sigma=p))

    # Get h* and sigma*
    hstar = params[np.argmin(hard_errors)]
    sigmastar = params[np.argmin(soft_errors)]

    # Get and return test error
    ERnew = ErrorRate(
        x_train=train[:, :4], x_val=test[:, :4], y_train=train[:, 4], y_val=test[:,
4]
    )
    v1 = ERnew.hard_parzen(h=hstar)
    v2 = ERnew.soft_parzen(sigma=sigmastar)

    return np.array([v1, v2])
```

Question 5

In [129]:

```
params = [0.01,0.1,0.2,0.3,0.4,0.5,1.0,3.0,10.0,20.0]
y1 = []
y2 = []
for p in params:
    y1.append(ER.hard_parzen(h=p))
    y2.append(ER.soft_parzen(sigma=p))

plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.plot(params, y1, label="Hard Parzen")
plt.plot(params, y2, label="Soft Parzen")
plt.plot()

plt.xlabel("sigma/h Value")
plt.ylabel("Error Rate")
plt.title("Hard Neighbourhood vs. Soft Neighbourhood (RBF) Parzen Windows\n Multicl
ass Classification Error Rate")
plt.legend()
plt.show()
plt.close()
```



At very small values of sigma and h, we have a large error rate and this is a clear demonstration of overfitting. This is also the case in large values of sigma and h, which is expected and is a display of underfitting.

Question 7

In both train() methods, since we do not perform any operations besides assigning variables, its time complexity is O(1).

In both compute_predictions() methods, our time complexity is O(N^2) where N is the number of observations in our test_data. We perform a nested for loop containing constant time operations.

Question 8

In [171]:

```
def random_projections(X, A):
    return (1 / np.sqrt(2)) * np.matmul(X, A)
```

Question 9

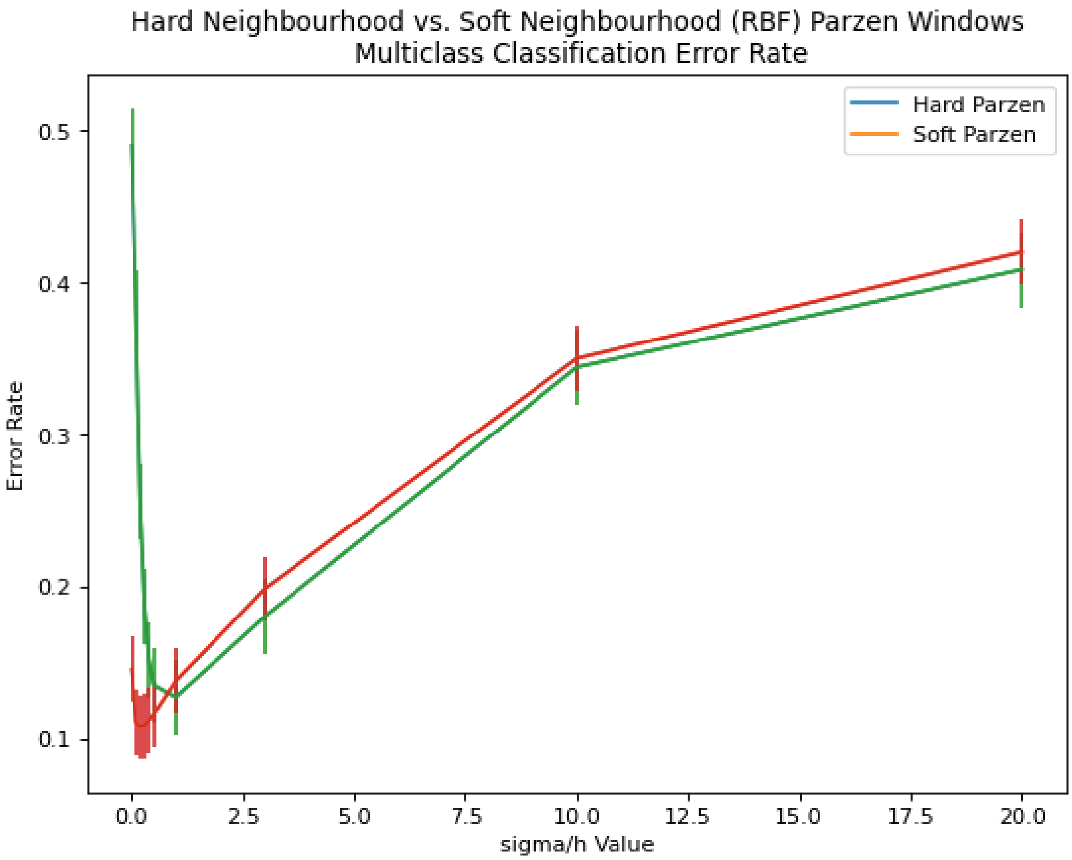
In [170]:

```
# Setup
num_samples = 500
params = [0.01,0.1,0.2,0.3,0.4,0.5,1.0,3.0,10.0,20.0]
hard_errors = np.zeros(shape=(num_samples, len(params)))
soft_errors = np.zeros(shape=(num_samples, len(params)))
projections = []
for i in range(num_samples):
    projections.append(np.random.normal(0, 1, (4, 2)))

train, valid, _ = split_dataset(banknote)
x_train = train[:, :4]
y_train = train[:, 4]
x_val = valid[:, :4]
y_val = valid[:, 4]

# Get Errors
for proj_index, proj in enumerate(projections):

    proj_train = random_projections(x_train, proj)
    proj_val = random_projections(x_val, proj)
    ER = ErrorRate(
        x_train=proj_train, y_train=y_train, x_val=proj_val, y_val=y_val
    )
    for p_index, p in enumerate(params):
        hard_errors[proj_index, p_index] = ER.hard_parzen(h=p)
        soft_errors[proj_index, p_index] = ER.soft_parzen(sigma=p)

hard_means = np.mean(hard_errors, axis=0)
soft_means = np.mean(soft_errors, axis=0)

plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.plot(params, hard_means, label="Hard Parzen")
plt.plot(params, soft_means, label="Soft Parzen")
plt.errorbar(params, hard_means, yerr=0.2*np.std(hard_means))
plt.errorbar(params, soft_means, yerr=0.2*np.std(soft_means))

plt.xlabel("sigma/h Value")
plt.ylabel("Error Rate")
plt.title("Hard Neighbourhood vs. Soft Neighbourhood (RBF) Parzen Windows\n Multicl
ass Classification Error Rate")
plt.legend()
```

```
CPU times: user 26min 41s, sys: 9.77 s, total: 26min 51s
Wall time: 28min 23s
```

Out[170]:

```
<matplotlib.legend.Legend at 0x7fd5da396d10>
```



Archive

In [81]:

```
class Q1:
    def feature_means(self, banknote):
        avgs = []
        for i in range(banknote.shape[-1]-1):
            avgs.append(np.average(banknote[:, i]))
        return avgs

    def covariance_matrix(self, banknote):
        return np.cov(banknote[:,:4], rowvar=False)

    def feature_means_class_1(self, banknote):
        return self.feature_means(banknote[banknote[:,4] == 1])

    def covariance_matrix_class_1(self, banknote):
        return self.covariance_matrix(banknote[banknote[:,4] == 1])
```

In [106]:

```
class HardParzen:
    def __init__(self, h):
        self.h = h

    def train(self, train_inputs, train_labels):
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.label_list = np.unique(train_labels)

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        counts = np.ones((num_test, len(self.label_list)))
        classes_pred = np.zeros(num_test)

        for i, observation in enumerate(test_data):
            # Get distance from point ex and every training example.
            distances = np.sqrt(np.sum((self.train_inputs - observation) ** 2, axis
=1))

            # Get an array of all the points that are within the parzen window of t
he example point.
            neighbour_indices = []
            parzen_radius = self.h
            neighbour_indices = np.array([j for j in range(len(distances)) if dista
nces[j] < parzen_radius])

            # If there are no points in the window, draw randomly, otherwise perfor
m a vote.
            if neighbour_indices.size == 0:
                classes_pred[i] = draw_rand_label(observation, self.train_labels)
            else:
                for k in neighbour_indices:
                    counts[i, int(self.train_labels[k])] += 1
                classes_pred[i] = np.argmax(counts[i, :])

        return classes_pred
```

In [127]:

```
class HardParzen:
    def __init__(self, h):
        self.h = h

    def train(self, train_inputs, train_labels):
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.label_list = np.unique(train_labels)

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        counts = np.ones((num_test, len(self.label_list)))
        classes_pred = np.zeros(num_test)

        for i, observation in enumerate(test_data):
            # Get distance from point ex and every training example.
            euclidean_distances = np.sqrt(np.sum((self.train_inputs - observation)
** 2, axis=1))

            # Get an array of all the points that are within the parzen window of t
he example point.
            neighbour_indices = euclidean_distances <= self.h

            for category_index, category in enumerate(self.label_list):
                counts[i, category_index] = sum(self.train_labels[neighbour_indices
] == category)

            # If there are no points in the window, draw randomly, otherwise perfor
m a vote.
            if neighbour_indices.size == 0:
                classes_pred[i] = draw_rand_label(observation, self.train_labels)
            else:
                classes_pred[i] = self.label_list[np.argmax(counts[i, :])]

        return classes_pred
```

In [125]:

```
train, val, test = split_dataset(banknote)

HP = HardParzen(h=3)
HP.train(train_inputs=train[:, :4], train_labels=train[:, 4])
preds = HP.compute_predictions(test_data=val[:, :4])
```

In [96]:

```
class SoftRBFParzen:
    def __init__(self, sigma):
        self.sigma = sigma

    def train(self, train_inputs, train_labels):
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.label_list = np.unique(train_labels)

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        classes_pred = np.zeros(num_test)

        for i, observation in enumerate(test_data):
            # Get distance from point ex and every training example.
            euclidean_distances = np.sqrt(np.sum((self.train_inputs - observation)
** 2, axis=1))

            # get sum of k
            ksum = 0
            kysum = np.ones(len(self.label_list))
            int_sum = np.zeros((test_data.shape[0], len(self.label_list)))

            for j, dist in enumerate(euclidean_distances):
                k = (1/(((2*np.pi)**len(self.label_list))*(self.sigma**(2*len(self.
label_list)))))*np.exp(-0.5*((dist**2)/(self.sigma**2)))
                ksum += k
                kysum[int(self.train_labels[j])] += k

            # get intermediate sum
            int_sum = (1/ksum)*(kysum)

            classes_pred[i] = np.argmax(int_sum)

        return classes_pred
```

In [97]:

```
def split_dataset(banknote):
    train = banknote[[i for i in range(banknote.shape[0]) if i % 5 <= 2]]
    validation = banknote[[i for i in range(banknote.shape[0]) if i % 5 == 3]]
    test = banknote[[i for i in range(banknote.shape[0]) if i % 5 == 4]]
    return (train, validation, test)
```

In [98]:

```
class ErrorRate:
    def __init__(self, x_train, y_train, x_val, y_val):
        self.x_train = x_train
        self.y_train = y_train
        self.x_val = x_val
        self.y_val = y_val

    def hard_parzen(self, h):
        HP = HardParzen(h)
        HP.train(train_inputs=self.x_train, train_labels=self.y_train)
        preds = HP.compute_predictions(test_data=self.x_val)
        return preds[preds != self.y_val].shape[0]/preds.shape[0]

    def soft_parzen(self, sigma):
        SP = SoftRBFParzen(sigma)
        SP.train(train_inputs=self.x_train, train_labels=self.y_train)
        preds = SP.compute_predictions(test_data=self.x_val)
        return preds[preds != self.y_val].shape[0]/preds.shape[0]
```

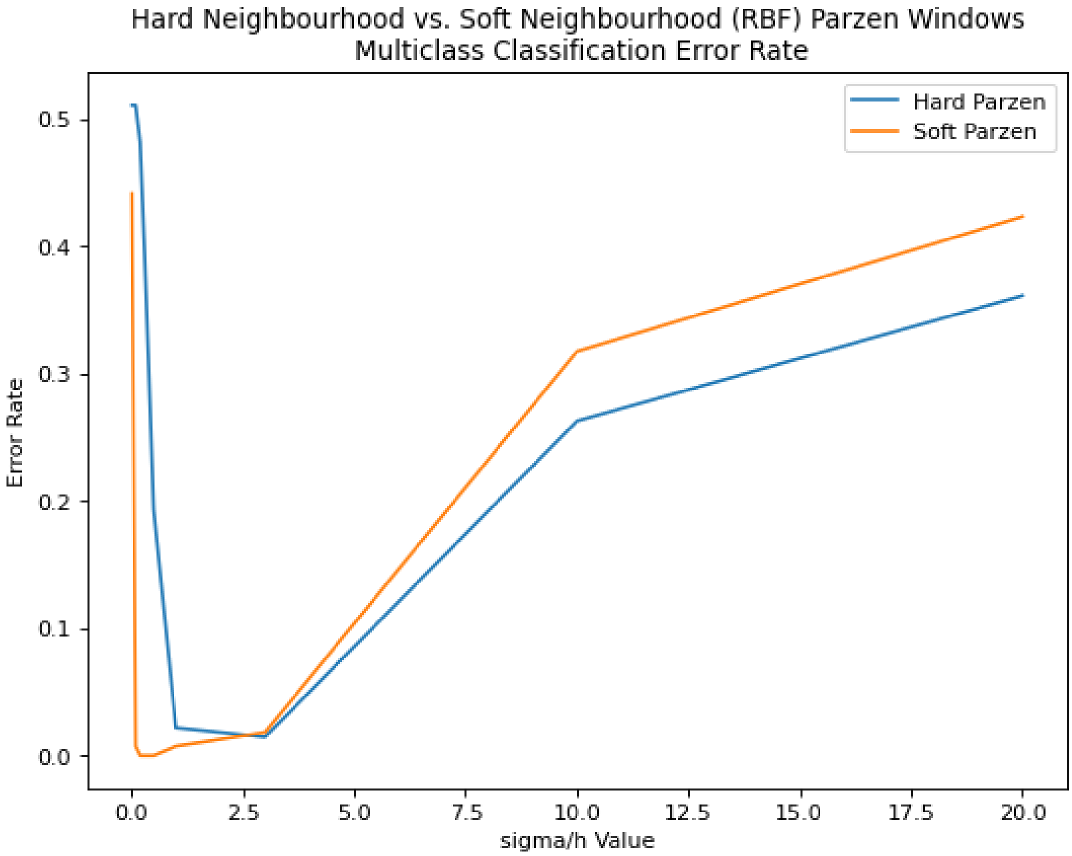
In [99]:

```
import matplotlib.pyplot as plt
```

In [107]:

```
params = [0.01,0.1,0.2,0.3,0.4,0.5,1.0,3.0,10.0,20.0]

hard_errors = []
soft_errors = []
for p in params:
    hard_errors.append(ER.hard_parzen(h=p))
    soft_errors.append(ER.soft_parzen(sigma=p))

plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.plot(params, hard_errors, label="Hard Parzen")
plt.plot(params, soft_errors, label="Soft Parzen")
plt.xlabel("sigma/h Value")
plt.ylabel("Error Rate")
plt.title("Hard Neighbourhood vs. Soft Neighbourhood (RBF) Parzen Windows\n Multiclass Classification Error Rate")
plt.legend()
plt.show()
plt.close()
```

```
/Users/Matteo/miniconda/envs/mila3.7/lib/python3.7/site-packages/ipykerr
/Users/Matteo/miniconda/envs/mila3.7/lib/python3.7/site-packages/ipykerr
```



In [114]:

```
train[:, :4].shape[1]
```

Out[114]:

4

In [108]:

```
def get_test_errors(banknote):
    train, val, test = split_dataset(banknote)
    ER = ErrorRate(x_train=train[:, :4], x_val=val[:, :4],
                  y_train=train[:, 4], y_val=val[:, 4])

    # Get Errors
    params = [0.01,0.1,0.2,0.3,0.4,0.5,1.0,3.0,10.0,20.0]
    hard_errors = []
    soft_errors = []
    for p in params:
        hard_errors.append(ER.hard_parzen(h=p))
        soft_errors.append(ER.soft_parzen(sigma=p))

    print(hard_errors)
    print(soft_errors)

    # Get h* and sigma*
    hstar = params[np.argmin(hard_errors)]
    sigmastar = params[np.argmin(soft_errors)]

    print(hstar)
    print(sigmastar)

    # Get and return test error
    ERnew = ErrorRate(x_train=train[:, :4], x_val=test[:, :4],
                  y_train=train[:, 4], y_val=test[:, 4])
    v1 = ERnew.hard_parzen(h=hstar)
    v2 = ERnew.soft_parzen(sigma=sigmastar)

    return np.array([v1,v2])
```

In [109]:

```
get_test_errors(banknote)
```

```
/Users/Matteo/miniconda/envs/mila3.7/lib/python3.7/site-packages/ipykerr
/Users/Matteo/miniconda/envs/mila3.7/lib/python3.7/site-packages/ipykerr
```

```
[0.5109489051094891, 0.5109489051094891, 0.48175182481751827, 0.39416058
[0.4416058394160584, 0.0072992700729927005, 0.0, 0.0, 0.0, 0.0, 0.007299
3.0
0.2
```

Out[109]:

```
array([0.01094891, 0.        ])
```

In [75]:

```
def random_projections(X, A):
    return (1/np.sqrt(2))*np.dot(X, A)
```

In [ ]: