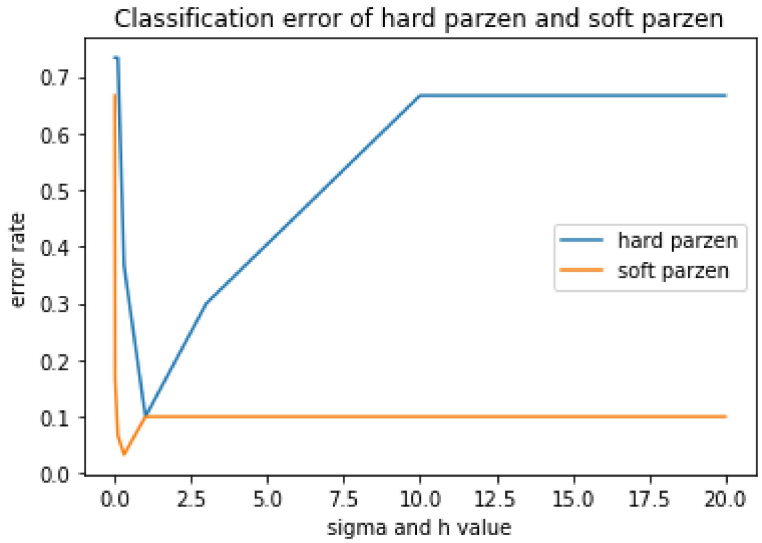
IFT 6390, Homework 1

Marc-Antoine Provost

Question 5

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
iris = np.loadtxt('iris.txt')

radius_hard = [0.001, 0.01, 0.1, 0.3, 1.0, 3.0, 10.0, 15.0, 20.0]
validation_error_hard = [0.7333333333333334, 0.7333333333333334, 0.7333333333333334, 0.3666666666666667, 0.09999999999999998, 0.30000000000000004, 0.6666666666666667, 0.6666666666666667, 0.6666666666666667]
validation_error_soft = [1.3333333333333335, 0.3333333333333326, 0.13333333333333333, 0.06666666666666665, 0.19999999999999996, 0.19999999999999996, 0.19999999999999996, 0.19999999999999996, 0.19999999999999996]

plt.plot(radius_hard, validation_error_hard, label = "hard parzen")
plt.plot(radius_hard, np.divide(validation_error_soft, 2), label = "soft parzen")
plt.xlabel('sigma and h value')
plt.ylabel('error rate')
plt.legend()
plt.title("Classification error of hard parzen and soft parzen")
plt.show()
```



As we increase the lenght in the k-NN algorithm with hard neighbourhood, we consider a larger portion of neighbors, thus the increase in the error rate. To illustrate this point, let us consider an example with a small lenght. In this case, our model will have very low bias, because it only considers a small window of points to predict the class of our new test point (but at the same time has high variance because we are sensitive to outliers). When selecting a wider lenght, all test points will belong to the same class; the majority class. Thus, increasing our hyperparameter h will result in higher bias, but smaller variance. I.e. we will have a larger classification error rate as we increase our hypeparameter.

The same can be said for the hyperparameter sigma in the k-NN algorithm with kernel density estimation. In this case, we are now taking a weighted vote instead of computing the vote of all neighbors in a specified lenght. With a small sigma, we give more weight to the points that are close to our test point, resulting in a smaller error rate. As sigma increase, the weight is more equally distributed, thus increasing the error rate.

Question 7

As the hyperparameter h increases, so does the running time for the method hard_parzen. By increasing our hyperparameter h, we include a larger number of points to compute the number of predictions and the error rate on.

When increasing the hyperparameter sigma, the running time seems to decrease a bit. This can be due to the fact that with a small sigma, more computation is needed to calculate different weights, while with a bigger sigma the weights are more or less similar.

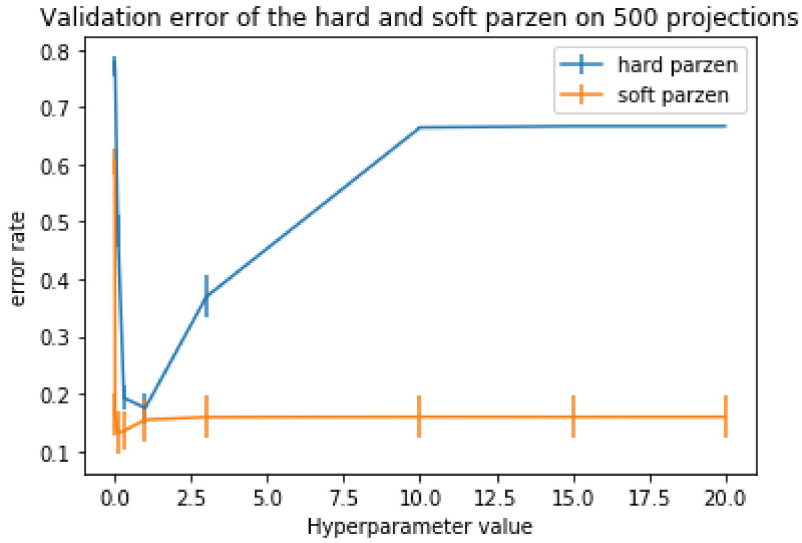
Question 9

In [ ]:

```
N = 500
splitted_data = split_dataset(iris)
training = splitted_data[0]
validation = splitted_data[1]
test = splitted_data[2]
x_train = splitted_data[0][:, 0:4]
y_train = splitted_data[0][:, 4]
x_val = splitted_data[1][:, 0:4]
y_val = splitted_data[1][:, 4]
validation_error_hard = np.zeros((500, 9))
validation_error_soft = np.zeros((500, 9))
hyperparameter = [0.001, 0.01, 0.1, 0.3, 1.0, 3.0, 10.0, 15.0, 20.0]

for i in range(N):
    random_matrix = np.random.normal(0, 1, (4, 2))
    modified_training = random_projections(x_train, random_matrix)
    modified_validation = random_projections(x_val, random_matrix)
    a = ErrorRate(modified_training, y_train, modified_validation, y_val)
    for (index, value) in enumerate(hyperparameter):
        validation_error_hard[i, index] = a.hard_parzen(value)

for i in range(N):
    random_matrix = np.random.normal(0, 1, (4, 2))
    modified_training = random_projections(x_train, random_matrix)
    modified_validation = random_projections(x_val, random_matrix)
    a = ErrorRate(modified_training, y_train, modified_validation, y_val)
    for (index, value) in enumerate(hyperparameter):
        validation_error_soft[i, index] = a.soft_parzen(value)

avg_val_hard = np.mean(validation_error_hard, axis=0)
avg_val_soft = np.mean(validation_error_soft, axis=0)
np.divide(avg_val_soft, 2)
np.std(validation_error_hard, axis = 0)
np.std(validation_error_soft, axis = 0)
```

In [32]:
```
avg_val_soft = [0.60446667, 0.163, 0.13146667, 0.1354, 0.1546, 0.15906667, 0.1
5966667, 0.1596, 0.15966667]
avg_val_hard = [0.77413333, 0.76646667, 0.48446667, 0.19266667, 0.1752,0.36926
667, 0.66426667, 0.66666667, 0.66666667]
hyperparameter = [0.001, 0.01, 0.1, 0.3, 1.0, 3.0, 10.0, 15.0, 20.0]
hard_std = [6.83747842e-02, 7.07786534e-02, 1.42870633e-01, 1.02724032e-01, 1.
31823518e-01, 1.87153781e-01, 1.03732562e-02, 1.33226763e-15, 1.33226763e-15]
error_hard = np.dot(hard_std, 0.2)
soft_std = [0.11348813, 0.18681213, 0.17870666, 0.17186967, 0.18206597, 0.1861
4838, 0.18669233, 0.18669233, 0.18669233]
error_soft = np.dot(soft_std, 0.2)


plt.errorbar(hyperparameter, avg_val_hard, yerr = error_hard, label= "hard par
zen")
plt.errorbar(hyperparameter, avg_val_soft, yerr = error_soft, label= "soft par
zen")
plt.xlabel("Hyperparameter value")
plt.ylabel("error rate")
plt.title("Validation error of the hard and soft parzen on 500 projections")
plt.legend()
```

Out[32]: <matplotlib.legend.Legend at 0xcb49c10>



The results are quite similar as the ones from question 5. This can be explained by our dimensionality reduction technique (random projection). Indeed, we reduced our number of features and it still captured the essence of our initial four features. Dimensionality reduction helps avoid the curse of dimensionality and reduce time and storage space.