Homework 1 - Practical Report

October 7, 2021

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

[2]:
```
######## 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)
##############################################
```

[3]:
```
iris = np.loadtxt('../data/iris.txt')
```

1 Q1: Implementation - no need for a report

[4]:
```
class Q1:

    def feature_means(self, iris):
        features = iris[:, :-1]
        means = np.mean(features, axis=0)
        return means

    def covariance_matrix(self, iris):
        features = iris[:, :-1]
        features = [features[:, 0], features[:, 1], features[:, 2], features[:,␣
↪3]]
        return np.cov(features)

    def feature_means_class_1(self, iris):
        iris_class_1 = np.array([x for x in iris if x[4] == 1])
        features_class_1 = iris_class_1[:, :-1]
        mean_class_1 = np.mean(features_class_1, axis=0)
        return mean_class_1
```

```
    def covariance_matrix_class_1(self, iris):
        iris_class_1 = np.array([x for x in iris if x[4] == 1])
        features_class_1 = iris_class_1[:, :-1]
        iris_class_1 = [features_class_1[:, 0], features_class_1[:, 1],
                        features_class_1[:, 2], features_class_1[:, 3]]
        return np.cov(iris_class_1)
```

2 Q2: Implementation - no need for a report

[5]:
```
class HardParzen:
    def __init__(self, h):
        self.h = h

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

    def compute_distances(self, x, train_inputs):
        distances = []
        for (i, x_train) in enumerate(train_inputs):
            distances.append(np.linalg.norm(x - x_train))

        return np.array(distances)

    def one_hot(self, y):
        one_hot = np.zeros(self.n_classes)
        one_hot[int(y) - 1] = 1
        return one_hot

    def get_neighbors_idx(self, x):
        distances = self.compute_distances(x, self.train_inputs)
        neighbors_idx = np.array([i for i in range(len(distances)) if␣
↪distances[i] < self.h])
        return neighbors_idx

    def compute_predictions(self, test_data):
        num_test = test_data.shape[0]
        classes_pred = np.zeros(num_test)
        for (i, x_test) in enumerate(test_data):

            neighbors_idx = self.get_neighbors_idx(x_test)
            if len(neighbors_idx) == 0:
                classes_pred[i] = draw_rand_label(x_test, self.label_list)
```

```
            else:
                one_hot_scores = []
                for neighbor_idx in neighbors_idx:
                    one_hot_yi = self.one_hot(self.train_labels[neighbor_idx])
                    one_hot_scores.append(one_hot_yi)
                one_hot_scores = np.array(one_hot_scores)
                classes_pred[i] = np.argmax(np.sum(one_hot_scores, axis=0)) + 1
        return classes_pred
```

3 Q3: Implementation - no need for a report

[6]:
```
class SoftRBFParzen:
    def __init__(self, sigma):
        self.sigma = sigma

    def train(self, train_inputs, train_labels):
        self.label_list = np.unique(train_labels)
        self.train_inputs = train_inputs
        self.train_labels = train_labels
        self.num_features = train_inputs.shape[1] - 1
        self.n_classes = len(np.unique(train_labels))

    def RBF_kernel(self, xi, x):
        fraction = 1 / (((2. * math.pi) ** (self.num_features / 2.)) * (self.
↪sigma ** self.num_features))
        euclidean_distance = np.linalg.norm(xi - x) ** 2.
        exp = math.exp((-1 / 2) * (euclidean_distance / (self.sigma ** 2)))
        score = fraction * exp
        return score

    def get_scores(self, x, X):
        scores = []
        for (i, xi) in enumerate(X):
            scores.append(self.RBF_kernel(xi, x))
        return np.array(scores)

    def one_hot(self, y):
        one_hot = np.zeros(self.n_classes)
        one_hot[int(y) - 1] = 1
        return one_hot

    def compute_predictions(self, test_data):
        # For each test datapoint
        num_test = test_data.shape[0]
        classes_pred = np.zeros(num_test)
        for (i, x_test) in enumerate(test_data):
```

```
            train_scores = self.get_scores(x_test, self.train_inputs)
            one_hot_scores = []
            for (j, y_train) in enumerate(self.train_labels):
                one_hot_yi = self.one_hot(y_train)
                one_hot_scores.append(train_scores[j] * one_hot_yi)
            one_hot_scores = np.array(one_hot_scores)
            classes_pred[i] = np.argmax(np.sum(one_hot_scores, axis=0)) + 1
        return classes_pred
```

4 Q4: Implementation - no need for a report

[7]:
```
def split_dataset(iris):
    train = []
    val = []
    test = []
    for (i, ex) in enumerate(iris):
        # Training dataset
        if i % 5 == 0 or i % 5 == 1 or i % 5 == 2:
            train.append(ex)

        # Validation dataset
        if i % 5 == 3:
            val.append(ex)

        # Test dataset
        if i % 5 == 4:
            test.append(ex)

    train = np.array(train)
    val = np.array(val)
    test = np.array(test)

    return train, val, test
```

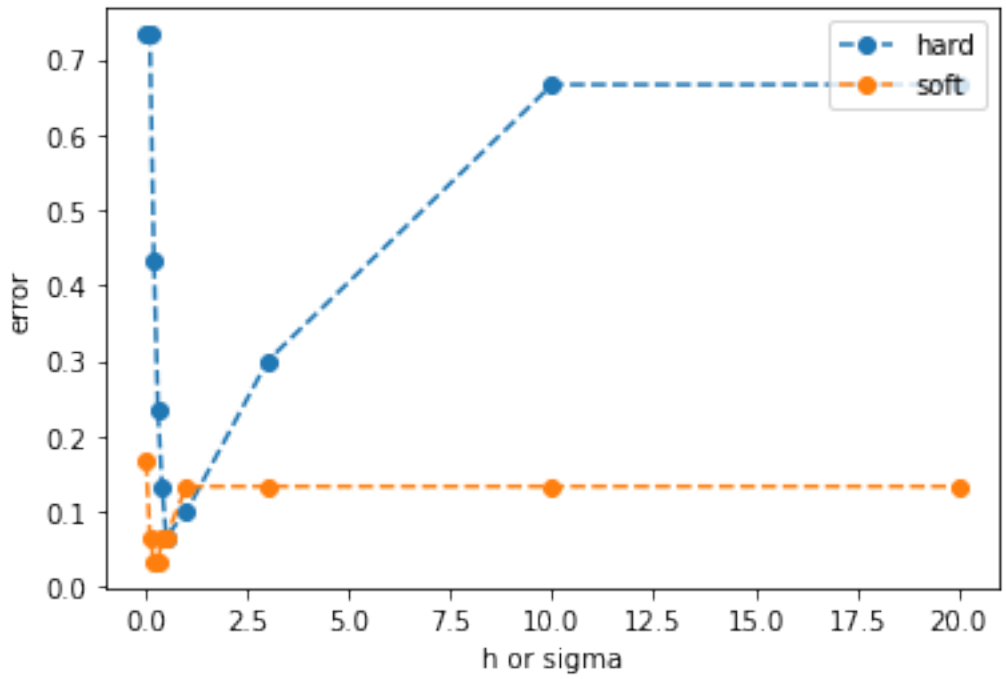
5 Q5: Implementation and Report

[8]:
```
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 calc_error(self, model_type, param):
        if model_type == 'soft':
            model = SoftRBFParzen(sigma=param)
        elif model_type == 'hard':
            model = HardParzen(h=param)
        else:
            raise Exception('Model type is undefined!')

        model.train(train_inputs=self.x_train, train_labels=self.y_train)
        y_pred = model.compute_predictions(self.x_val)

        num_misclassified = 0
        for i in range(len(self.y_val)):
            if self.y_val[i] != y_pred[i]:
                num_misclassified += 1
        return num_misclassified / len(self.y_val)

    def hard_parzen(self, h):
        return self.calc_error(model_type='hard', param=h)

    def soft_parzen(self, sigma):
        return self.calc_error(model_type='soft', param=sigma)
```

[9]:
```
train, val, test, = split_dataset(iris)

x_train = train[:, :-1]
y_train = train[:, -1]

x_val = val[:, :-1]
y_val = val[:, -1]

x_test = test[:, :-1]
y_test = test[:, -1]

er = ErrorRate(x_train=x_train, y_train=y_train, x_val=x_val, y_val=y_val)

hard_parzen_errors = []
hs = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
for h in hs:
    hard_parzen_errors.append(er.hard_parzen(h=h))

soft_parzen_errors = []
sigmas = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
for sigma in sigmas:
    soft_parzen_errors.append(er.soft_parzen(sigma=sigma))
```

[10]:
```
plt.plot(hs, hard_parzen_errors, linestyle='--', marker='o', label='hard')
plt.plot(sigmas, soft_parzen_errors, linestyle='--', marker='o', label='soft')
plt.legend(loc='upper right')

plt.ylabel('error')
plt.xlabel('h or sigma')

plt.savefig("../reports/error.pdf", dpi=1200)
plt.show()
```



[11]:
```
min_hard_parzen_error = np.min(hard_parzen_errors)
corresponding_h = hs[np.argmin(hard_parzen_errors)]
print('Minimum error of HardParzen is {} by {} h.'.
 ↪format(min_hard_parzen_error, corresponding_h))


min_soft_parzen_error = np.min(soft_parzen_errors)
corresponding_sigma = hs[np.argmin(soft_parzen_errors)]
print('Minimum error of SogtParzen is {} by {} sigma.'.
 ↪format(min_soft_parzen_error, corresponding_sigma))
```

```
Minimum error of HardParzen is 0.06666666666666667 by 0.5 h.
Minimum error of SogtParzen is 0.03333333333333333 by 0.2 sigma.
```

[12]:
```
print('HardParzen Errors:', hard_parzen_errors)
print('SoftParzen Errors:', soft_parzen_errors)
```

```
HardParzen Errors: [0.7333333333333333, 0.7333333333333333, 0.43333333333333335,
0.23333333333333334, 0.13333333333333333, 0.06666666666666667, 0.1, 0.3,
0.6666666666666666, 0.6666666666666666]
SoftParzen Errors: [0.16666666666666666, 0.06666666666666667,
0.03333333333333333, 0.03333333333333333, 0.06666666666666667,
0.06666666666666667, 0.13333333333333333, 0.13333333333333333,
0.13333333333333333, 0.13333333333333333]
```

1. For HardParzen, from h=0.01 to h=0.5, error decreases; similarly, for SoftParzen, from h=0.01 to h=0.2.
2. The minimum value of the hyper-parameter for errors on the validation set for the HardParzen model is 0.5 and for the SoftParzen model is 0.2.
3. For HardParzen, for hs greater than 0.5, error increases, and at some point, it won't change anymore; similarly, for SoftParzen, for sigmas greater than 0.2.
4. Overall observation, the SoftParzen has a smaller error than HardParzen.
5. For larger values of hyper-parameters, larger than 0.5 for Harparzen and larger than 0.2 for SoftParzen, models will become under-fitted (based on lectures slide + train error).
6. For smaller values of hyper-parameters, smaller than 0.5 for Harparzen and 0.2 for SoftParzen, models will become over-fitted (based on lectures slide + train error).
7. After 10.0 value of "h" for the HardParzen, the error doesn't change, which might mean that the maximum distance of every point in the validation set to training set is less than 10.0, and every training points fall into every validation point's neighbors. We also can say a similar explanation for SoftParzen, for sigma values larger than 1.0.

6 Q6: Implementation - no need for a report

[13]:
```
def get_test_errors(iris):
    train, val, test, = split_dataset(iris)

    x_train = train[:, :-1]
    y_train = train[:, -1]

    x_val = val[:, :-1]
    y_val = val[:, -1]

    x_test = test[:, :-1]
    y_test = test[:, -1]

    er = ErrorRate(x_train=x_train, y_train=y_train, x_val=x_val, y_val=y_val)

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

```
    smallest_hard_parzen_error = float("inf")
    optimal_hard_parzen_h = -1
    for h in Hs:
        current_hard_parzen_error = er.hard_parzen(h=h)
        if current_hard_parzen_error < smallest_hard_parzen_error:
            smallest_hard_parzen_error = current_hard_parzen_error
            optimal_hard_parzen_h = h

    sigmas = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
    smallest_soft_parzen_error = float("inf")
    optimal_soft_parzen_sigma = -1
    for sigma in sigmas:
        current_soft_parzen_error = er.soft_parzen(sigma=sigma)
        if current_soft_parzen_error < smallest_soft_parzen_error:
            smallest_soft_parzen_error = current_soft_parzen_error
            optimal_soft_parzen_sigma = sigma

    test_er = ErrorRate(x_train=x_train, y_train=y_train, x_val=x_test,␣
↪y_val=y_test)
    hard_parzen_error = test_er.hard_parzen(h=optimal_hard_parzen_h)
    soft_parzen_error = test_er.soft_parzen(sigma=optimal_soft_parzen_sigma)
    return np.array([hard_parzen_error, soft_parzen_error])
```

7 Q7: Report

Discussion on the running time complexity of the HardParzen and SoftParzen methods:

n = number of training examples
d = dimention of each example
p = number of validation examples

For HardParzen:
Training phase technically does not exist, since all computation is done during prediction, so we have O(1) for time complexity.
Training time complexity: O(1)

For the prediction time complexity for a single point, we do the following steps: 1. Computing distances of a test point to all training points: O(nd) 2. Finding neighbors which fall into radius h: it depends on h. O(number of points in the neighborhood) 3. Counting neighbors' votes: O(number of points in the neighborhood)

The maximum value of points which fall into the neighborhood is n; so:
Prediction time complexity: O(nd + n + n) -> O(nd)

For SoftParzen:
Similarly:
Training time complexity: O(1)

For the prediction time complexity for a single point, we do the following steps: 1. Computing

scores of each training points: O(nk) 2. Counting neighbors' votes: O(n)

Prediction time complexity: O(nd + n) -> O(nd)

The hyperparameter h or changes don't affect the time complexity of models because there is a step for "computing distances" or "computing neighbors' scores" which has much more effect on time complexity.

8 Q8: Implementation - no need for a report

[14]:
```
def random_projections(X, A):
    return (1/math.sqrt(2)) * np.matmul(X, A)
```

9 Q9: Implementation and Report

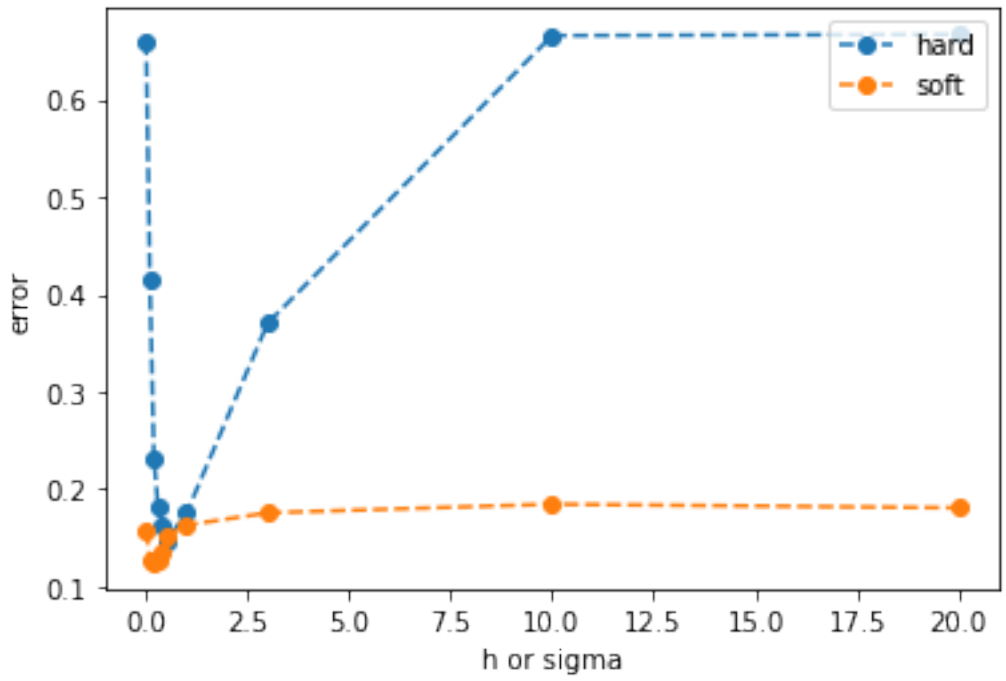
[15]:
```
class Q9:

    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 calc_error(self, model_type, params):

        val_errors = -1 * np.ones((500, 10))
        for i in range(500):
            for (j, param) in enumerate(params):
                if model_type == 'soft':
                    model = SoftRBFParzen(sigma=param)
                elif model_type == 'hard':
                    model = HardParzen(h=param)
                else:
                    raise Exception('Model type is undefined!')

                A = np.random.normal(0, 1, (4, 2))
                X = random_projections(self.x_train, A)
                projected_x_val = random_projections(self.x_val, A)
                model.train(train_inputs=X, train_labels=self.y_train)
                y_pred = model.compute_predictions(projected_x_val)

                num_misclassified = 0
                for k in range(len(self.y_val)):
                    if self.y_val[k] != y_pred[k]:
                        num_misclassified += 1
                val_errors[i][j] = num_misclassified / len(self.y_val)
```

```
        return val_errors

    def hard_parzen(self, hs):
        return self.calc_error(model_type='hard', params=hs)

    def soft_parzen(self, sigmas):
        return self.calc_error(model_type='soft', params=sigmas)
```

[16]:
```
train, val, test, = split_dataset(iris)
x_train = train[:, :-1]
y_train = train[:, -1]

x_val = val[:, :-1]
y_val = val[:, -1]

x_test = test[:, :-1]
y_test = test[:, -1]

q9 = Q9(x_train=x_train, y_train=y_train, x_val=x_val, y_val=y_val)

hs = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
hard_parzen_errors = q9.calc_error(model_type='hard', params=hs)

sigmas = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0, 3.0, 10.0, 20.0]
soft_parzen_errors = q9.calc_error(model_type='soft', params=sigmas)
```

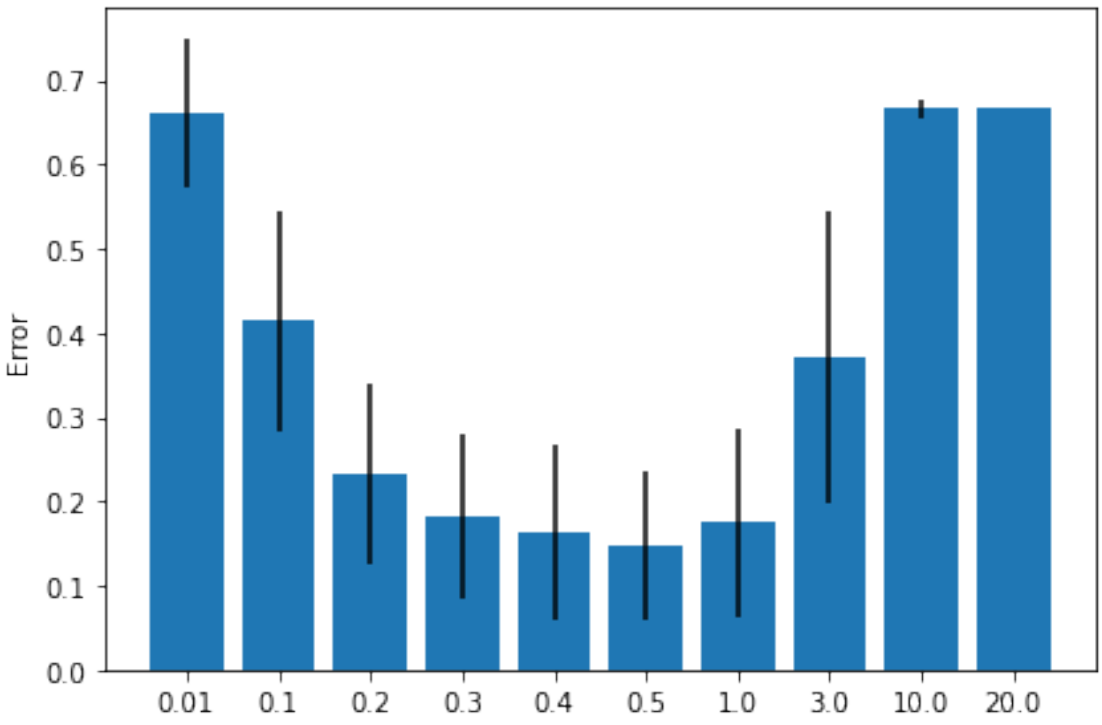
[17]:
```
hard_parzen_means_errors_of_hs = np.mean(hard_parzen_errors, axis=0)
soft_parzen_means_errros_of_sigmas = np.mean(soft_parzen_errors, axis=0)
```

[18]:
```
plt.plot(hs, hard_parzen_means_errors_of_hs, linestyle='--', marker='o',␣
 ↪label='hard')
plt.plot(sigmas, soft_parzen_means_errros_of_sigmas, linestyle='--',␣
 ↪marker='o', label='soft')
plt.legend(loc='upper right')

plt.ylabel('error')
plt.xlabel('h or sigma')

plt.savefig("../reports/projected_errors.pdf", dpi=1200)

plt.show()
```

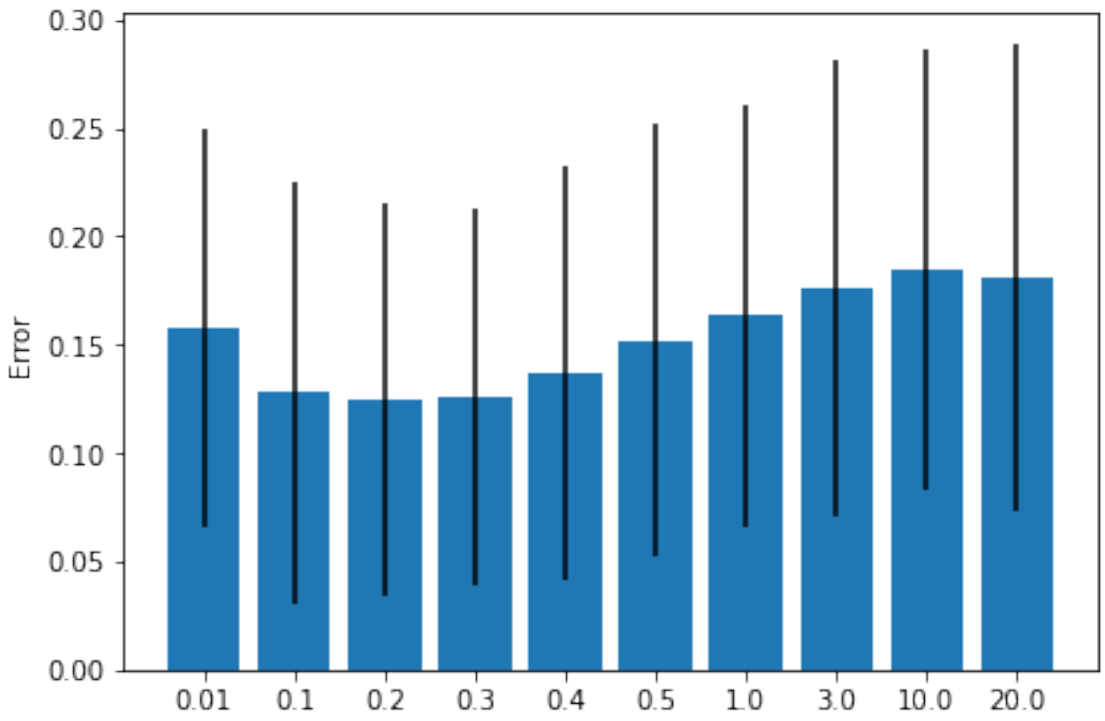


[19]:
```
hard_means = []
hard_stds = []

soft_means = []
soft_stds = []
for i in range(len(hs)):
    hard_means.append(np.mean(hard_parzen_errors[:, i]))
    hard_stds.append(np.std(hard_parzen_errors[:, i]))

    soft_means.append(np.mean(soft_parzen_errors[:, i]))
    soft_stds.append(np.std(soft_parzen_errors[:, i]))
```

[20]:
```
fig, ax = plt.subplots()
x_pos = [i for i in range(len(hs))]
ax.set_ylabel('HardParzen Error')
ax.bar(x_pos, hard_means, yerr=hard_stds)
ax.set_ylabel('Error')
ax.set_xticks(x_pos)
ax.set_xticklabels(['{}'.format(h) for h in hs])

plt.savefig("../reports/hard_parzen_bar_error.pdf", dpi=1200)

plt.tight_layout()
plt.show()
```



[21]:
```
fig, ax = plt.subplots()
x_pos = [i for i in range(len(sigmas))]
ax.set_ylabel('SoftParzen Error')
ax.bar(x_pos, soft_means, yerr=soft_stds)
ax.set_ylabel('Error')
ax.set_xticks(x_pos)
ax.set_xticklabels(['{}'.format(sigma) for sigma in sigmas])

plt.savefig("../reports/soft_parzen_bar_error.pdf", dpi=1200)
plt.tight_layout()
plt.show()
```



[22]:
```
min_hard_parzen_error = np.min(hard_parzen_means_errors_of_hs)
corresponding_h = hs[np.argmin(hard_parzen_means_errors_of_hs)]
print('Minimum error of HardParzen is {} by {} h.'
 ↪format(min_hard_parzen_error, corresponding_h))


min_soft_parzen_error = np.min(soft_parzen_means_errros_of_sigmas)
corresponding_sigma = hs[np.argmin(soft_parzen_means_errros_of_sigmas)]
print('Minimum error of SogtParzen is {} by {} sigma.'
 ↪format(min_soft_parzen_error, corresponding_sigma))
```

```
Minimum error of HardParzen is 0.14713333333333364 by 0.5 h.
Minimum error of SogtParzen is 0.12480000000000036 by 0.2 sigma.
```

[23]:
```
print('hard_parzen_means_errors_of_hs:', hard_parzen_means_errors_of_hs)
print()
print('soft_parzen_means_errros_of_sigmas:', soft_parzen_means_errros_of_sigmas)
```

```
hard_parzen_means_errors_of_hs: [0.6604     0.41453333 0.23273333 0.182
0.16326667 0.14713333
 0.17533333 0.3704     0.66546667 0.66666667]

soft_parzen_means_errros_of_sigmas: [0.15786667 0.12773333 0.1248     0.12626667
0.13646667 0.15186667
```

```
0.1632      0.1758      0.18466667 0.18093333]
```

Note: I got the same value for h and sigma as before, so it means that our projection preserves the distance of points, and by using random projection, we could reduce the dimension of the problem.