```
import numpy as np
# import random
import matplotlib.pyplot as plt
######## 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):
        return [np.sum(banknote[:,i])/len(banknote[:,i]) for i in range(banknote.shape[1]-1)]

    def covariance_matrix(self, banknote):
        return np.cov(banknote[:, :-1].T)

    def feature_means_class_1(self, banknote):
        class1=np.array([banknote[i] for i in range(banknote.shape[0]) if banknote[i][-1]==1.
        return [np.sum(class1[:,i])/len(class1[:,i]) for i in range(class1.shape[1]-1)]

    def covariance_matrix_class_1(self, banknote):
        class1=np.array([banknote[i] for i in range(banknote.shape[0]) if banknote[i][-1]==1.
        return np.cov(class1[:, :-1].T)

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=list(np.unique(train_labels))
        self.num_class = len(np.unique(train_labels))

    def euclidean_distances(self,x, Y):
      return (np.sum((np.abs(x - Y)) ** 2, axis=1)) ** (1.0 / 2)

    def compute_predictions(self, test_data):
        test_size = test_data.shape[0]
        counts = np.ones((test_size, self.num_class))
        classes_pred = np.zeros(test_size)
        for (i, ex) in enumerate(test_data):
            distances = self.euclidean_distances(ex, self.train_inputs)
            neighbour_idx = []
```

```
            neighbour_idx = np.array([j for j in range(len(distances)) if distances[j] < self
            if len(neighbour_idx)==0:
                output=draw_rand_label(ex,self.label_list)
                classes_pred[i]=output

            else:
              for k in neighbour_idx:
                counts[i, int(self.train_labels[k])] += 1
              classes_pred[i] = np.argmax(counts[i, :])

        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.num_class = len(np.unique(train_labels))

    def euclidean_distances(self,x, Y):
      return (np.sum((np.abs(x - Y)) ** 2)) ** (1.0 / 2)

    def RBF(self,Xi, X, sigma):
        coef=((2*np.pi)**(len(Xi)/2))*(sigma**(len(Xi)))
        dist = self.euclidean_distances(X,Xi)**2
        return (1/coef)*np.exp((-1*dist)/(2*(sigma**2)))

    def compute_predictions(self, test_data):
        test_size = test_data.shape[0]
        counts = np.ones((test_size,self.num_class))
        classes_pred = np.zeros(test_size)

        for (i, ex) in enumerate(test_data):
          weights=[]
          summ=0
          for j in range(len(self.train_inputs)):
            wi=self.RBF(self.train_inputs[j],ex, self.sigma)
            counts[i, int(self.train_labels[j])]+=wi

          classes_pred[i] = np.argmax(counts[i, :])

        return classes_pred


def split_dataset(banknote):
        train = [0,1,2,5]
        validation=[3]
        test=[4]
```

```
        train_cols = list(range(0,banknote.shape[1]-1))
        target_ind = [banknote.shape[1] - 1]

        inds = list(range(banknote.shape[0]))
        train_inds = [inds[i] for i in range(len(inds)) if inds[i]%5 in train]
        validation_inds = [inds[i] for i in range(len(inds)) if inds[i]%5 in validation]
        test_inds = [inds[i] for i in range(len(inds)) if inds[i]%5 in test]

        train_set = banknote[train_inds, :]
        train_set = train_set[:, train_cols + target_ind]
        test_set = banknote[test_inds, :]
        test_set = test_set[:, train_cols + target_ind]
        validation_set = banknote[validation_inds, :]
        validation_set = validation_set[:, train_cols + target_ind]

        return train_set, validation_set, test_set


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):
        h_parzen = HardParzen(h)
        h_parzen.train(self.x_train, self.y_train)
        classes_pred_knn = h_parzen.compute_predictions(self.x_val)
        conf_mat = confusion_matrix(self.y_val, classes_pred_knn)
        total_num = np.sum(conf_mat)
        num_correct = np.sum(np.diag(conf_mat))
        return 1.0 - num_correct / total_num

    def soft_parzen(self, sigma):
        s_parzen = SoftRBFParzen(sigma)
        s_parzen.train(self.x_train, self.y_train.astype('int32') )
        classes_pred_knn = s_parzen.compute_predictions(self.x_val)
        conf_mat = confusion_matrix(self.y_val.astype('int32') , classes_pred_knn.astype('int
        total_num = np.sum(conf_mat)
        num_correct = np.sum(np.diag(conf_mat))
        return 1.0 - num_correct / total_num


def confusion_matrix(true_labels, pred_labels):
    n_classes=2
    matrix = np.zeros((n_classes, n_classes))
    for (true, pred) in zip(true_labels, pred_labels):
        matrix[int(true), int(pred)] += 1
    return matrix
```

```
def get_test_errors(banknote):
    train_set, validation_set, test_set= split_dataset(banknote)
    ee=ErrorRate(train_set[:, :-1],train_set[:, -1].astype('int32'), test_set[:, :-1], test_s
    errror_hard_parzen=ee.hard_parzen(3)
    errror_soft_parzen=ee.soft_parzen(0.5)
    return [errror_hard_parzen,errror_soft_parzen]


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

Question 5

```
banknote = np.genfromtxt("data_banknote_authentication.txt", delimiter=",")
train_set, validation_set, test_set= split_dataset(banknote)
ee=ErrorRate(train_set[:, :-1],train_set[:, -1].astype('int32'), validation_set[:, :-1], vali
hyperparameter=[0.01,0.1,0.2,0.3,0.4,0.5,1,3,10,20]
errror_hard_parzen=[ee.hard_parzen(k) for k in [0.01,0.1,0.2,0.3,0.4,0.5,1,3,10,20]]
errror_soft_parzen=[ee.soft_parzen(k) for k in [0.01,0.1,0.2,0.3,0.4,0.5,1,3,10,20]]


h_star=hyperparameter[np.argmin(errror_hard_parzen)]
sigma_star=hyperparameter[np.argmin(errror_soft_parzen)]
```

```
errror_hard_parzen
```

```
[0.5072992700729927,
 0.5036496350364963,
 0.4817518248175182,
 0.4124087591240876,
 0.2992700729927007,
 0.21897810218978098,
 0.025547445255474477,
 0.014598540145985384,
 0.26277372262773724,
 0.36131386861313863]
```

```
errror_soft_parzen
```

```
[0.44160583941605835,
 0.007299270072992692,
 0.0,
 0.0,
 0.0,
 0.0,
 0.007299270072992692,
 0.018248175182481785,
```

```
0.3175182481751825,
0.4233576642335767]
```

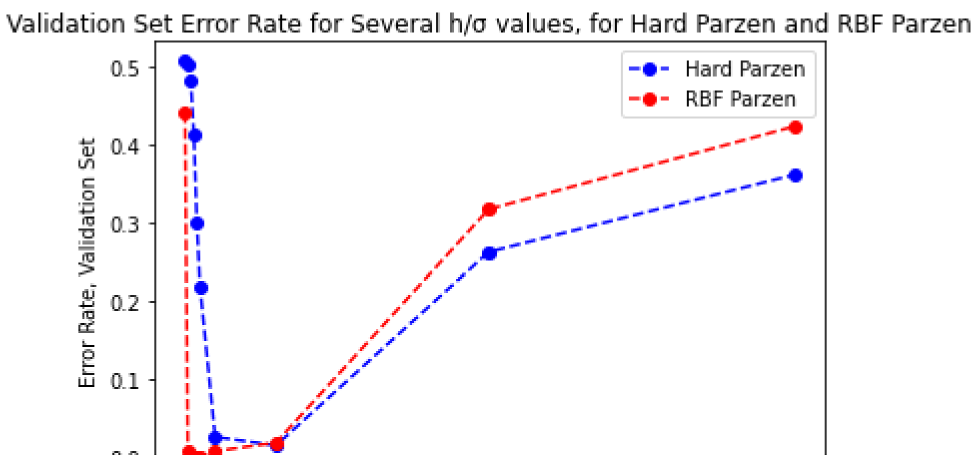
```
h_star, sigma_star
```

```
(3, 0.2)
```

```
print(np.array(errror_soft_parzen).mean())
print(np.array(errror_hard_parzen).mean())
```

```
0.12153284671532846
0.30875912408759126
```

```
plt.plot(hyperparameter, errror_hard_parzen, '--bo')
plt.plot(hyperparameter, errror_soft_parzen, '--ro')
plt.xticks([0, 1, 3, 10, 15, 20])
# plt.xscale('log')
plt.xlabel('Value of h/σ')
plt.ylabel('Error Rate, Validation Set')
plt.title('Validation Set Error Rate for Several h/σ values, for Hard Parzen and RBF Parzen')
plt.legend(('Hard Parzen','RBF Parzen'))
plt.show()
```



Observations

The Hard Parzen algorithm labels unseen points by serving a majority class vote on the set of neighbors within a distance of h of the unseen point, and exclude all other.

as Expected and shown from the plot above: When the value of h is small, the model picks only the neighbors that are closest to the data sample, thus forming a very complex decision boundary. Such a model fails to generalize well on the test data set, thereby showing poor results. we see validation-set error rates close to 50% for h values such as 0.01. This might be because of the sparsity in the data. if a data point within the validation set is too far (distance > 0.01) from any point in the training set then a label is selected at random and Since the current task is binary classification, a validation accuracy of 50% is not unexpected in this scenario.

As we increase the h, the model starts to generalize well, and we observe a meaningful reduction in the error rate (The lowest error rate of ≈ 0.014598540145985384% is observed for h=3.0) but increasing the value too much would again drop the performance showing that augmenting the range of the neighborhood of points we consider in order to compute the majority vote that will determine the predicted label, is not a perfect solution. For high values of h, like h=10, h=20, where points that are very far from our target point, start having an equal vote in determining the label of the test point, we observe a substantial decrease in accuracy. This is because points that are too far away from the test point are not relevant.

Therefore we conclude that the optimal value h$*$ chosen from the list of values that we have evaluated is h=3.0, where the validation set error rate is close to 0.014598540145985384%.

We see that the value of h governs the degree of smoothing there is an optimum choice for h that is neither too large nor too small

The RBF Parzen algorithm labels unseen points by executing a weighted majority class vote on the whole of the training set. The weight of each point in the training set is determined by computing the RBF kernel function of this point and the unseen point that must be labeled.

as Expected and shown from the plot above: When the value of sigma is small we have very high validation set error rates. This is due to the fact that many of the validation points will not have training set neighbors that are close enough to them to influence a meaningful vote towards the correct vote. Instead because of the very small sigma and the exponentially decaying nature of the weight, with regards to the distance, the vote of training points that are close enough to contribute towards the correct label will receive an equal amount of weight as the points that are very far away. when the value of σ increases, we start observing good accuracy and reach an optimum σ^* for $\sigma = 0.2, 0.3, 0.4 and 0.5$and the validation error rate equals 0.0%. Similar to Hard Parzen when we increase the value of σ, we start observing high validation set error rates again.

Question 7 - Complexity

h=the size of the neighborhood in Hard Parzen sigma=the value of sigma for the Soft Parzen with RBF kernel n=the number of points in the training set d=the dimentions of each point l=the number of classes for training the Hard Parzen algorithm we just need to load the training set from the disk so the runtime is O(n)

for testing the Hard Parzen algorithm we need to compute the distance between the unseen data point and each data point in the training set o(nd) and perform a majority vote o(n) which results in a total runtime of O(nd + n) simplifies to O(nd).So for a test set of size m, the total run time would be O(mnd).

The hyperparameter h does not change the runtime of the hard parzen algorithm, since we can consider at most n points as neighbors of our unseen point for any value of h.

for training the Soft Parzen algorithm we just need to load the training set from the disk so the runtime is O(n)

For testing the Soft Parzen algorithm with RBF kernel, we need to compute the distance between the unseen data point and each training data point in the training set. So for a test set of size m the run time would be O(mnd). (Computing the `RBF kernel runs in constant time.)

sigma does not change the runtime on Soft Parzen since we consider the vote of all points in the training set this has nothing to do with the runtime complexity. so as the the number of possible labels they can't change the runtime complexity.

Question 9

```
projections = np.random.normal(0, 1, (500,4,2))
```

```
projected_train_sets = np.array([random_projections(train_set[:, :-1], projection) for projec
projected_train_sets.shape
```

```
(500, 824, 2)
```

```
projected_validation_sets = np.array([random_projections(validation_set[:,0:4], projection) f
projected_validation_sets.shape
```

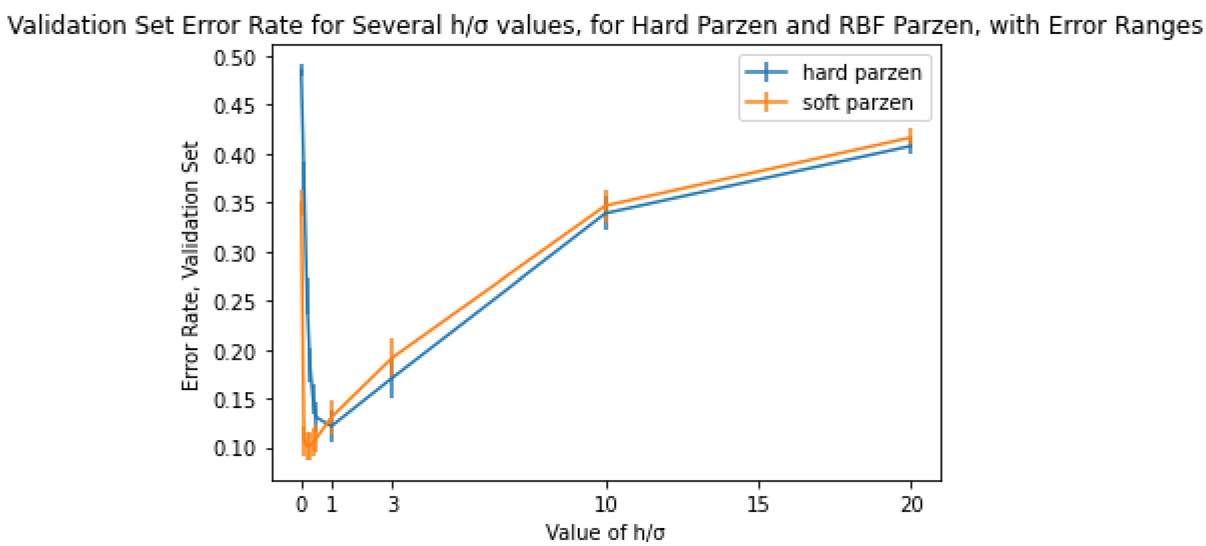
```
(500, 274, 2)
```

```
error_rate_objects = [ErrorRate(projected_train_set, train_set[:, -1], projected_validation_s
```

```
h_err=np.zeros((500,10))
s_err=np.zeros((500,10))
hyperparameter=[0.01,0.1,0.2,0.3,0.4,0.5,1,3,10,20]
for ee_idx in range(len(error_rate_objects)):
```

```
    # if (ee_idx %10==0):
          # print("ee isdx is :", ee_idx)
          # print()
          # print()
          # print()
    for p_idx in range(len(hyperparameter)):
        ee=error_rate_objects[ee_idx]
        errror_hard_parzen=ee.hard_parzen(hyperparameter[p_idx])
        errror_soft_parzen=ee.soft_parzen(hyperparameter[p_idx])
        h_err[ee_idx][p_idx]=errror_hard_parzen
        s_err[ee_idx][p_idx]=errror_soft_parzen


plt.errorbar(x=hyperparameter, y=h_err.mean(axis=0), yerr=0.2 * h_err.std(axis=0), label='har
plt.errorbar(x=hyperparameter, y=s_err.mean(axis=0), yerr=0.2 * s_err.std(axis=0), label='sof
plt.xticks([0, 1, 3, 10, 15, 20])
plt.xlabel('Value of h/σ')
plt.ylabel('Error Rate, Validation Set')
plt.title('Validation Set Error Rate for Several h/σ values, for Hard Parzen and RBF Parzen,
plt.legend()
```



The plot above shows the results after running 500 simulations with projections of the data on a matrix $A_{(2,4)}$ containing 8 independent variables drawn uniformly from a Gaussian distribution of mean 0 and variance 1. the plot demonstrates a sudden drop in Error Rate initially, until reaches a minimum error rate somewhere close to 0, and ended increasing in Error Rate as h/σ increases, which is somehow similar to the plot we had in the plot in Q5 without any projection.

One thing worth mentioning is that because of reducing the dimension of our inputs form 4D to 2D which kind of hide some information available previously the Error Rate values after this projection on data which reduce the dimension of our input features has a higher average than when we perform the model on the original input without any projection, and also the optimal value of our hyperparamter, where the error is minimum also decreases since we kind of remove some dimensions.