# The KNN Algorithm

## 1. Load the data

```
def generate_data():
    np.random.seed(0)
    X_xor = np.random.randn(2000, 2)
    y_xor = np.logical_xor(X_xor[:, 0] > 0,
                           X_xor[:, 1] > 0)
    y_xor = np.where(y_xor, 1, -1)
    return X_xor, y_xor
Bring them in rbf space:
def get_rbf_kernel(X, gamma):
    print("gamma :", gamma)
    new_X = []

    for i, data in enumerate(X):
        new_data = []
        for j, center in enumerate(centers):
            d = np.exp(-sum(pow((np.subtract(data, list(center.values())[0])),
2)) / gamma)
            new_data.append(d)
        new_X.append(np.array(new_data))
    new_X = np.array(new_X)
    return new_X
```

## 2. Initialize K to your chosen number of neighbors

1. 31

## 3. For each example in the data (باkmeansداده را ابتدا کاهش می دهیم)

4.

```
def get_means(dict, part):
    kmeans = KMeans(n_clusters=19)
    kmeans.fit(dict)
    labels = kmeans.labels_
    new_dict = get_label_dict(dict, labels)
    means = []
    for key, value in new_dict.items():
        means.append(get_mean(value, part))
    return means
```

1.

## 3.1 Calculate the distance between the query example and the current example from the data.

```
def cluster_on_mahalanobis_distance(x, means: List[dict] = None, inv_covs=None):
    mahal = []
    for mean in means:
        dict1={"key":list(mean.keys())[0],"value":mahalanobis(x,
list(mean.values())[0], inv_covs[list(mean.keys())[0]])}
        mahal.append(dict1)
        # mahal.append(mahalanobis(x, list(mean.values())[0]))
    mahal= sorted(mahal, key = lambda i: i['value'])
    mahal_result=mahal[:21]
    a={1:0,-1:0}
    for item in mahal_result:
        a[item['key']]=a[item['key']]+1
    if a[1] > a[-1]:
        return 1
    return -1
```

## 3.2 Add the distance and the index of the example to an ordered collection

```
        mahal.append(dict1)
        # mahal.append(mahalanobis(x, list(mean.values())[0]))
    mahal= sorted(mahal, key = lambda i: i['value'])
    mahal_result=mahal[:21]
    a={1:0,-1:0}
    for item in mahal_result:
        a[item['key']]=a[item['key']]+1
    if a[1] > a[-1]:
        return 1
    return -1
```

## 4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances

## 5. Pick the first K entries from the sorted collection

```
mahal_result=mahal[:21]
```

6. Get the labels of the selected K entries

```
for item in mahal_result:
        a[item['key']]=a[item['key']]+1
```

7. If regression, return the mean of the K labels

no

8. If classification, return the mode of the K labels

```
if a[1] > a[-1]:
        return 1
    return -1
```

## Choosing the right value for K

To select the K that's right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

Here are some things to keep in mind:

1. As we decrease the value of K to 1, our predictions become less stable. Just think for a minute, imagine K=1 and we have a query point surrounded by several reds and one green (I'm thinking about the top left corner of the colored plot above),

but the green is the single nearest neighbor. Reasonably, we would think the query point is most likely red, but because K=1, KNN incorrectly predicts that the query point is green.

2. Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point). Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.

3. In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

```
Result :  gamma =0.5 , k=21:

gamma : 0.5

: Confusion Matrix

[2   199]]

[[198 1  ]

Accuracy Score : 0.9925
```