HW5_TiagoGoncalves

December 31, 2018

```
In [5]: #Exercise 1
        #a)
        import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.cluster import KMeans

        #Load Data
        data = np.genfromtxt('heightWeightData.txt', delimiter=",")

        #Split Data into Labels and Features
        #Convert labels into 0-1
        labels = data[:,0] - 1
        #Features
        features = data[:, 1:3]

        #Create Classifier K-Means, with K=2
        kmeans = KMeans(n_clusters=2, random_state=0)
        #Obtain clusters with k-means classifier
        print("Fitting data....")
        kmeans.fit(features)
        #Make predictions on our data
        #Predictions
        y_hat = np.array(kmeans.predict(features))
        #Compare predictions with ground truth
        #Count elements of class 0 in labels
        class_0 = 0
        for label in labels:
            if label==0:
                class_0+=1

        #Count elements of class 0 in labels
        class_1 = 0
        for label in labels:
            if label==1:
                class_1+=1

        #Count elements of class 0 in labels
        pred_0 = 0
        for label in y_hat:
            if label==0:
                pred_0+=1

        #Count elements of class 0 in labels
        pred_1 = 0
        for label in y_hat:
            if label==1:
                pred_1+=1

        #Number of elements per class
        print("Elements per class in original data: ", "\nClass 0: ", class_0, "\nClass 1: ", c
        print("Elements per cluster in predictions: ", "\nCluster 0: ", pred_0, "\nCluster 1: "
        print("According to the results, one can say that K-Means algorithm, with K=2 performs
```

```
Fitting data...
Elements per class in original data:
Class 0:  73
Class 1:  137
Elements per cluster in predictions:
Cluster 0:  138
Cluster 1:  72
According to the results, one can say that K-Means algorithm, with K=2 performs well, when comp
```

```
In [2]: #b)
        #New approach from: https://github.com/alexkimxyz/XMeans/blob/master/xmeans.py

        """
        Implementation of XMeans algorithm based on
        Pelleg, Dan, and Andrew W. Moore. "X-means: Extending K-means with Efficient Estimatio
        ICML. Vol. 1. 2000.
        https://www.cs.cmu.edu/~dpelleg/download/xmeans.pdf
        """

        import numpy as np
        from sklearn.cluster import KMeans

        EPS = np.finfo(float).eps


        def loglikelihood(R, R_n, variance, M, K):
            """
            See Pelleg's and Moore's for more details.
            :param R: (int) size of cluster
            :param R_n: (int) size of cluster/subcluster
            :param variance: (float) maximum likelihood estimate of variance under spherical G
```

```
    :param M: (float) number of features (dimensionality of the data)
    :param K: (float) number of clusters for which loglikelihood is calculated
    :return: (float) loglikelihood value
    """
    if 0 <= variance <= EPS:
        res = 0
    else:
        res = R_n * (np.log(R_n) - np.log(R) - 0.5 * (np.log(2 * np.pi) + M * np.log(va
        if res == np.inf:
            res = 0
    return res


def get_additonal_k_split(K, X, clst_labels, clst_centers, n_features, K_sub, k_means_a
    bic_before_split = np.zeros(K)
    bic_after_split = np.zeros(K)
    clst_n_params = n_features + 1
    add_k = 0
    for clst_index in range(K):
        clst_points = X[clst_labels == clst_index]
        clst_size = clst_points.shape[0]
        if clst_size <= K_sub:
            # skip this cluster if it is too small
            # i.e. cannot be split into more clusters
            continue
        clst_variance = np.sum((clst_points - clst_centers[clst_index]) ** 2) / float(c
        bic_before_split[clst_index] = loglikelihood(clst_size, clst_size, clst_varianc
                                                      1) - clst_n_params / 2.0 * np.log
        kmeans_subclst = KMeans(n_clusters=K_sub, **k_means_args).fit(clst_points)
        subclst_labels = kmeans_subclst.labels_
        subclst_centers = kmeans_subclst.cluster_centers_
        log_likelihood = 0
        for subclst_index in range(K_sub):
            subclst_points = clst_points[subclst_labels == subclst_index]
            subclst_size = subclst_points.shape[0]
            if subclst_size <= K_sub:
                # skip this subclst_size if it is too small
                # i.e. won't be splittable into more clusters on the next iteration
                continue
            subclst_variance = np.sum((subclst_points - subclst_centers[subclst_index])
                subclst_size - K_sub)
            log_likelihood = log_likelihood + loglikelihood(clst_size, subclst_size, su
                                                             K_sub)
        subclst_n_params = K_sub * clst_n_params
        bic_after_split[clst_index] = log_likelihood - subclst_n_params / 2.0 * np.log
        # Count number of additional clusters that need to be created based on BIC com
        if bic_before_split[clst_index] < bic_after_split[clst_index]:
            add_k += 1
        return add_k


class XMeans(KMeans):
    def __init__(self, kmax=50, max_iter=1000, **k_means_args):
        """
        :param kmax: maximum number of clusters that XMeans can divide the data in
        :param max_iter: maximum number of iterations for the `while` loop (hard limit
        :param k_means_args: all other parameters supported by sklearn's KMeans algo (
        """
        if 'n_clusters' in k_means_args:
            raise Exception("`n_clusters` is not an accepted parameter for XMeans algo
        if kmax < 1:
            raise Exception("`kmax` cannot be less than 1")
        self.KMax = kmax
        self.max_iter = max_iter
        self.k_means_args = k_means_args

    def fit(self, X, y=None):
        K = 1
        K_sub = 2
        K_old = K
        n_features = np.size(X, axis=1)
        stop_splitting = False
        iter_num = 0
        while not stop_splitting and iter_num < self.max_iter:
            K_old = K
            kmeans = KMeans(n_clusters=K, **self.k_means_args).fit(X)
            clst_labels = kmeans.labels_
            clst_centers = kmeans.cluster_centers_
            # Iterate through all clusters and determine if further split is necessary
            add_k = get_additonal_k_split(K, X, clst_labels, clst_centers, n_features,
            K += add_k
            # stop splitting clusters when BIC stopped increasing or if max number of
            stop_splitting = K_old == K or K >= self.KMax
            iter_num = iter_num + 1
        # Run vanilla KMeans with the number of clusters determined above
        kmeans = KMeans(n_clusters=K_old, **self.k_means_args).fit(X)
        self.labels_ = kmeans.labels_
        self.cluster_centers_ = kmeans.cluster_centers_
        self.inertia_ = kmeans.inertia_
        self.n_clusters = K_old
```

In [3]:
```
#Create X-Means Instance
x_means = XMeans(kmax=50, max_iter=1000)
print("Fitting data...")
x_means.fit(features)
print("Predicted labels: \n", x_means.labels_)
```

```
        print("Predicted cluster centers: \n", x_means.cluster_centers_)
        print("Predicted number of clusters: ", x_means.n_clusters)
```

```
Fitting data...
Predicted labels:
 [0 0 0 0 0 0 2 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0 2 0 0
 0 2 0 0 1 0 0 1 1 0 1 0 0 2 0 0 2 1 2 1 0 0 0 1 0 2 0 0 0 0 1 1 0 0 2 1 1
 0 0 1 0 1 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 2 2 1 1 0 1 1 1 0 0 0 0
 1 0 0 0 0 1 2 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0
 0 0 1 1 1 0 0 2 0 0 1 1 1 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 0 2
 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 1 1 1 1 0 1]
Predicted cluster centers:
 [[163.83        57.36255469]
 [181.37072464  74.63976812]
 [188.15538462 106.31607692]]
Predicted number of clusters:  3
```

What's the criterion to choose the number of clusters?

```
In [4]: print("To choose the best number of clusters, i.e. the best model, the authors proposed
        print("Assuming that we have the data, D, and a a family of alternative models, Mj (whe
        print("In this case, since all models are of the K-Means assumed type (spherical Gaussi
        print("This is also known as the Schwarz criterion.")
```

```
To choose the best number of clusters, i.e. the best model, the authors proposed the BIC scorin
Assuming that we have the data, D, and a a family of alternative models, Mj (where each solutio
In this case, since all models are of the K-Means assumed type (spherical Gaussians), to approx
This is also known as the Schwarz criterion.
```

```
In [ ]: #b) Another approach that uses pyclustering library, that is based on another article.
        #In case of usage of pyclustering
        #!pip install pyclustering

        from pyclustering.cluster import cluster_visualizer
        from pyclustering.cluster.xmeans import xmeans
        from pyclustering.cluster.center_initializer import kmeans_plusplus_initializer
        from pyclustering.utils import read_sample
        from pyclustering.samples.definitions import SIMPLE_SAMPLES
        # Read sample 'simple3' from file.
        sample = features
        # Prepare initial centers - amount of initial centers defines amount of clusters from w
        # start analysis.
        amount_initial_centers = 2
        initial_centers = kmeans_plusplus_initializer(sample, amount_initial_centers).initializ
        # Create instance of X-Means algorithm. The algorithm will start analysis from 2 cluste
        # number of clusters that can be allocated is 20.
        xmeans_instance = xmeans(sample, initial_centers, 20)
```

```
xmeans_instance.process()
# Extract clustering results: clusters and their centers
clusters = xmeans_instance.get_clusters()
centers = xmeans_instance.get_centers()
# Visualize clustering results
visualizer = cluster_visualizer()
visualizer.append_clusters(clusters, sample)
visualizer.append_cluster(centers, None, marker='*')
visualizer.show()
```