Middle East Technical University | Department of Computer Engineering

CENG 499

Introduction to Machine Learning

Fall '2022-2023

Homework 2

Due date: December 1, 2022, 23:55

Objectives

This assignment aims to fulfill the following objectives:

- To familiarize you with the cross-validation technique which is crucial for model tuning and model evaluation, and enable you to gain hands-on experience with it.
- To provide hands-on experience with the K Nearest Neighbor method and how to tune its hyper-parameters with cross-validation.
- To familiarize you with Kmeans and Kmeans++ algorithms at the implementation level for clustering problems and provide hands-on experience in how to pinpoint the number of clusters with the elbow method.
- To familiarize you with Hierarchical Agglomerative Clustering and Silhouette method for clustering and evaluating clustering results, respectively.

The assignment involves both coding and writing a report. You can prepare your report in Latex or Word (the format of the report is up to you). There should be a single report submitted for all parts.

Part 1

When a problem to be solved has abundant data so as to form training, validation, and test datasets, a machine model could be trained on the training dataset, its hyperparameters (if there are any) could be determined with the validation dataset and its expected generalization performance could be measured on the test dataset. Separation of a dataset into three reduces the data amount for training especially if the dataset contains a relatively small number of instances (the fewer instances there are in a training dataset, the worse generalization performance we expect since the training dataset fails to represent the actual data generation process's data distribution). One way to alleviate this problem is to omit the validation dataset and perform model training and testing on a training dataset and a test dataset (instead of dividing a whole dataset into three, we divide it into 2 parts namely, the training dataset and testing dataset). The test dataset is used to assess the trained model's expected generalization performance. One crucial point regarding the test dataset is that it should contain enough instances to yield reliable results. With the training dataset, we need to carry out both the parameter search and hyperparameter tuning procedures. We could extract a validation dataset but it will be arbitrary and reduce the training

sample amount. We could pick instances randomly to form a validation dataset, but by chance, we may have picked some outlier or noisy instances so the results attained on this single validation dataset may not be reliable for parameter tuning. Instead, we should form multiple validation datasets and average results attained on them for parameter tuning. This is the very basic approach considered by the cross-validation technique. The training dataset is divided into K parts. For each hyperparameter configuration, K-1 parts are used for training the model and the remaining partition is used to assess the performance of the model with that hyperparameter configuration in an iterative manner (first 1^{st} partition is set as the test dataset, $2^{nd}, 3^{rd}, 4^{th}...K^{th}$ partitions are used for training. Second, for the same hyperparameter value, 2^{nd} partition is considered to be the test dataset, and the rest ($1^{st}, 3^{rd}, 4^{th}...K^{th}$) is regarded as the training, this procedure is repeated for each partition and for each hyperparameter configuration). In total, K many test datasets and training datasets are used for each hyperparameter configuration and the average performance score over these test datasets provides an idea about the generalization capability of that hyperparameter configuration. This average score can be used to pick the best hyperparameter configuration. After determining the best-performing hyperparameter values, the model can be trained with all K partitions and the trained model's generalization performance could be measured on the original test dataset (the test dataset that we set aside just before the beginning of cross-validation). The cross-validation procedure is summarized in the following figure.

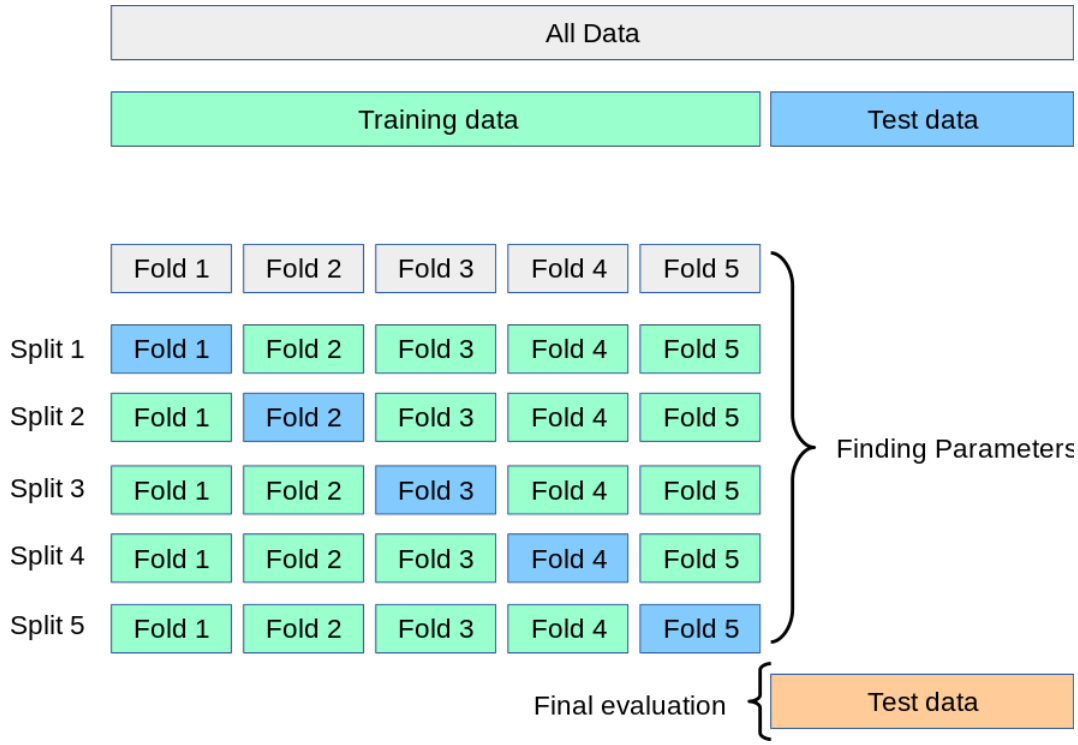

Figure 1: Cross-validation technique applied on a dataset for hyperparameter tuning. A dataset is divided into training (referred to as TRAIN1) and test (referred to as TEST1) parts (the training part is used for both training the model and tuning hyperparameters whereas the test part is solely used to assess the generalization performance of the trained model. The training part is further divided into K parts and for each hyperparameter configuration, one part is designated as a test dataset and the remaining ones are regarded as a training dataset in an iterative manner. Depending on the average score attained on these partitions, the best hyperparameter configuration could be determined. After identification of the best-performing hyperparameter values, the model is trained on the training part (TRAIN1) and its final generalization score is calculated on the test part (TEST1). Generally K values of 5, 10 or 30 are used. The image has been taken from [1].

Since during training K-1 parts are used, the reduction in the number of training samples is smaller compared to designating a separate validation dataset. In addition, because many validation datasets are used to assess the performance of a hyperparameter configuration, a more reliable (unbiased) result is obtained. Of course, all these come with an extra computation cost (for each hyperparameter configuration, the model needs to be trained K times). If the model being trained involves some randomness, the cross-validation procedure should be repeated many times to eliminate random results (even if the model is deterministic, it still is better to repeat cross-validation by shuffling the original dataset to further obtain more reliable results). In other words, the results attained during cross-validation should be statistically significant. Typically, K values of 5, 10, or 30 are considered.

In spite of all its merits, there are mainly two pitfalls that should be considered with cross-validation. The first one is that while partitioning the training part into K parts, there is a risk of introducing a class imbalance problem, especially if we pick partition instances randomly (for instance for a classification task, for one class an instance may not be present in a partition which is considered to be a test dataset). To eliminate this problem, the partition should be formed depending on the original class distribution of the training part. Such a strategy is coined as stratification [2]. The second pitfall emerges when the cross-validation technique is used on a whole dataset for both model hyperparameter tuning and model performance assessment (without sparing a test dataset out of the whole dataset). Especially if the problem to be solved features a small dataset and sparing a portion of it as a test dataset may not be possible (Sample size may be too small), direct application of the cross-validation technique could be considered on the whole dataset. With such an approach, the same dataset is used for both hyperparameter tuning and model evaluation, which leads to optimistic evaluation results [3,4,5]. Since the hyperparameters are tuned to perform well on the dataset, the model can be considered to overfit the dataset. Since the evaluation is done on the same dataset, the results may be misleading [4] (It may not provide reliable results about the model's generalization capability, it is likely to be optimistic). This serious problem can be alleviated via the nested cross-validation technique [4, 5].

The K-nearest neighbor (KNN) method is a lazy method (also it is a non-parametric density estimation method [6]) which can be used for both classification and regression problems, it is "lazy" in the sense that it does not require training (it does not have parameters to tune, non-parametric). For a given instance whose label is not known, it finds the K nearest neighbors of that instance from a training data set depending on a distance/similarity metric and labels the given data sample. Labeling can be done via majority voting or weighted majority voting (some other strategies could also be utilized, for instance, when labels of the neighbors are considered to be an input, a neural network could be trained for the final decision). In the first strategy, every nearest neighbor has equal importance (their decisions count equally) on the final label decision and the data sample is labeled with the label that has been voted the most by the neighbors. In the second strategy (a weighted score is calculated for each K nearest neighbor depending on their distance/similarity to the given data sample), closer neighbors have more contribution (since they are closer to the given data sample or they are more similar) to the final decision. For classification problems, with majority voting, the class that attains the most votes is considered as the predicted label of a given point. With weighted majority voting, the class label that attains the most cumulative weight is returned as the resulting label (different weight combination strategies also could be considered). For a regression task, with majority voting, values of the K nearest neighbors are averaged. With weighted voting, a weighted average of the values can be calculated as the label of a given instance (other combination schemes could be considered too).

KNN requires two main hyperparameters to be tuned: K and distance/similarity metric. Depending on data distribution and the number of features of data samples, the optimal hyperparameter configuration varies. For K=1, the method is called nearest neighbor (NN), which is a commonly used form of KNN for classification and regression tasks. A given data sample is labeled with the label of the closest data point. As the distance measure, several metrics (and more) could be considered: Minkowski, Euclidian (where p=2 for Minkowski), cosine, and Mahalanobis. The cosine similarity is based on the angle between two vectors. The smaller the angle between two vectors, the more similar these vectors are to each other. Mahalanobis distance [7] takes into account the possible feature correlations (which affect the data

distribution) of data samples by incorporating the covariance matrix in calculating the distance between two data points (one feature may lead data to spread out/elongate in its direction, the covariance matrix captures such distributions) (If the covariance matrix is the identity matrix, Mahalanobis distance becomes equivalent to the Euclidian distance). Here are the formulas for these distance/similarity measures:

$$\text{Minkowski}(\vec{x}, \vec{y}, p) = (\sum_{i=1}^{d} |x_i - y_i|^p)^{1/p}, \text{where } d \text{ is the dimension of x and y vectors.}$$

$$\text{Cosine}(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| * \|\vec{y}\|}, \text{where } \cdot \text{ denotes the dot product operation.}$$

$$\text{Mahalanobis}(\vec{x}, \vec{y}, S^{-1}) = \sqrt{(\vec{x} - \vec{y})^T S^{-1} (\vec{x} - \vec{y})}, \text{where } S^{-1} \text{ is the reverse of the covariance matrix of a dataset.}$$

Part 1 Specifications

- You are expected to implement the following class (whose file name should be **Distance.py**) for the distance/similarity metrics:

```
class Distance:
    @staticmethod
    def calculateCosineDistance(x, y):
        pass
    @staticmethod
    def calculateMinkowskiDistance(x, y, p=2):
        pass
    @staticmethod
    def calculateMahalanobisDistance(x,y, S_minus_1):
        pass
```

  `S_minus_1` parameter of `calculateMahalanobisDistance` specifies the inverse of the covariance matrix of a dataset. This matrix can be calculated with **numpy.cov** [8] and **numpy.linalg.inv** [9] for a given dataset. All the datasets in this assignment are 2D numpy arrays. Each data sample (instance) is represented with a row vector.

- For the KNN method, you are expected to implement the following class (whose file name should be **Knn.py**):

```
class KNN:
    def __init__(self, dataset, data_label, similarity_function,
    similarity_function_parameters=None, K=1):
        """
        :param dataset: dataset on which KNN is executed, 2D numpy array
        :param data_label: class labels for each data sample, 1D numpy
    array
        :param similarity_function: similarity/distance function, Python
    function
        :param similarity_function_parameters: auxiliary parameter or
    parameter array for distance metrics
        :param K: how many neighbors to consider, integer
        """
        self.K = K
        self.dataset = dataset
        self.dataset_label = data_label
        self.similarity_function = similarity_function
        self.similarity_function_parameters =
    similarity_function_parameters
```

```
15
16      def predict(self, instance):
17          pass
```

The `similarity_function` is assigned to one of the `Distance` class functions, `similarity_function_parameters` variable holds additional parameters for the similarity/distance function (i.e it holds p for the Minkowski distance or S^{-1} for the Mahalanobis distance). The `predict` function takes a data instance (as numpy array) and returns the predicted label for that instance via **majority voting** among the K nearest neighbors.

- You are provided with a classification problem dataset and an parital implementation that loads the dataset (**Knnexperiment.py**). On this dataset, you are expected to perform 10-fold cross-validation (with stratification) for hyperparameter tuning (via grid search). You are free to determine which hyperparameter configurations to test (there should be at fewest 5 configurations). Please repeat this cross-validation procedure 5 times (by shuffling the original dataset) and compute confidence intervals of **accuracy** performance scores for each hyperparameter configuration. In your report, please specify these hyperparameter values and attained confidence intervals for each hyperparameter configuration. In addition, please add some comments on how you have picked the best-performing hyperparameter values. For stratified cross-validation, you may refer to the following Scikitlearn documentation [10]. For data shuffling with numpy, you may refer to [11].
- You may prefer to consider other performance metrics as well (it is up to you) (i.e. precision, recall, .etc).
- Your whole experimentation code for this part should be implemented in **Knnexperiment.py**.
- For this part, you are expected to make use of your own implementation for KNN. Library implementations are not allowed.

Note: Although the KNN method does not feature any randomness, in this part, by repeating the cross-validation procedure multiple times, we aim to reduce the dependency on a single fixed partitioning, whose results may be biased and misleading. By shuffling the dataset, we introduce different partitionings to the method and by averaging the attained scores we obtain more statistically reliable results. In other words, we want our results not to depend on a single partitioning of the dataset.

Part 2

The unsupervised learning paradigm aims to extract knowledge with data samples that lack label information. One important problem in this paradigm is to cluster data samples and extract insightful knowledge from these groupings. To this end, this part and the next part are devoted to the clustering problem.

The Kmeans algorithm is a special case of the expectation-maximization procedure where the likelihood of data samples is maximized with respect to a model parameter by considering hidden/latent variables (the factors that we don't observe directly) [12, 13]. It consists of two main steps: the expectation (E) step and the maximization (M) step. The procedure starts with an initial guess of the parameters. Later by iterating through the E and M steps, the procedure maximizes the likelihood of data samples. In the E step, hidden variables are estimated with the current parameter values and these estimated hidden variable values are used to improve the current parameter values. For Kmeans, the information which specifies a data point belongs to which cluster (cluster label) is a hidden variable and the mean vectors of clusters are parameters. Kmeans begins its operations by initializing the cluster means randomly. Then it estimates the cluster labels of data samples by finding the closest cluster mean among the current cluster mean vectors and assigning them to the closest cluster (it labels the data samples with the label of the

closest cluster mean) (this constitutes the E-step for KMeans, we estimate labels of data samples by using the current cluster means). Next, the current cluster mean vectors are updated with the current cluster labels (simply, an average (mean) vector of data points in a cluster is calculated. This constitutes the M step, the current cluster means are updated with the current cluster labels). In other words, in the E step, the current mean values are fixed and the cluster labels are updated. In the M step, the cluster labels are fixed and the cluster means are updated.

For Kmeans, these operations can be formulated as the following loss function [12]:

$$L = \sum_{s}^{n} \sum_{k}^{K} l_k^s \|x_s - \mu_k\|^2 \tag{1}$$

where n is the number of data samples (x_s is the s^{th} data sample), K is the number of clusters (μ_k is the mean vector of the k^{th} cluster) , l_k^s specifies cluster label for a data sample. If an instance s belongs to a cluster k, l_k^s is 1 for that cluster (for the other clusters it is 0). Minimization of this loss function corresponds to finding compact clusters whose data points are close to their cluster centers and not scattered around. Furthermore, we can interpret the loss function as the sum of inner cluster variances [14] ($Variance = \mathbb{E}[(x - \mu)^2]$). Hence Kmeans aims to minimize variance within each cluster (it aims to form clusters whose inner variances are minimum).

As the main distance/similarity measure Euclidean distance is considered for KMeans (the method requires a centroid notation to work (which is available in Euclidean space), datasets may contain data features for which averaging/calculation mean is meaningless, i.e. marital status: single, married (mean/average cannot be defined))). This introduces bias to KMeans and leads KMeans to perform well on the spherical-shaped datasets (if clusters are of arbitrary shape, it is highly likely to perform poorly). Such a restriction leads to the Kmedoid method [15], which considers a data sample as the cluster representative rather than the mean of cluster data points (for Kmeans, clusters are summarized/identified as a mean vector of data points, for Kmedoids a single data point is considered). In the M step, instead of calculating a centroid (mean of cluster data samples), the data point that is the closest to all cluster data points is picked to be the new cluster center. Similarly in the E step, data points are labeled depending on their closeness to these cluster representatives. With this method, other distance/similarity metrics could be considered for forming clusters. The loss function for the method is a generalization of that of Kmeans:

$$L = \sum_{s}^{n} \sum_{k}^{K} l_k^s d(x_s, c_k)$$

where $d(x, y)$ is a distance/similarity metric (function), c_k is the representative data point for the cluster k. Minimization of this loss function corresponds to forming clusters whose data points are close to the cluster representative data point (More similar data points are grouped together).

The expectation maximization procedure may get stuck in local optima while maximizing the likelihood of data samples. The similar handicap is valid for Kmeans (since it is a specific instance of the EM procedure). So Kmeans is not guaranteed to yield an optimal clustering result for a given dataset (neither is Kmedoid).

Kmeans (also Kmedoid) strongly depends on the initially picked cluster centers while generating a final clustering result. Several strategies could be considered to this end. We could run Kmeans (by selecting the initial cluster centers randomly) many times (e.g 50 times) and pick the one that attains the smallest loss error (Equation 1) after running. Or we can pick instances that are distant from each other depending on a fixed distance threshold value. A better approach is from the Kmeans++ algorithm [16]: picking initial clusters with a probability score depending on the distance between data samples. Kmeans++ differs from Kmeans solely in picking initial cluster representatives. As the first step, Kmeans++ picks an instance randomly as the first cluster center. The rest of the cluster centers (the remaining K-1 centers) are sampled with a probability score depending on the distances between data samples and already picked cluster centers (D^2 weighting). After picking all cluster centers, standard E and M steps of Kmeans are

applied. With this strategy, Kmeans++ has been shown to complete faster and yield better clustering results compared to Kmeans [16].

Along with how to pick initial cluster centers, the most important hyperparameter of Kmeans is the number of clusters to form. If we have knowledge about the data generation process, we might know the exact number of clusters beforehand (i.e. handwritten digit dataset, 10 different clusters are expected). If we do not have access to such information, we may try to project the dataset into a lower dimension (e.g 3D, 2D) and visually determine the number of clusters from the projected data. This may not be possible for every dataset (especially for real datasets, dimensionality reduction introduces data loss, which may culminate in uninterpretable visual results). Another possibility is to apply information-theoretic measures [17]. As the last resort (but not the least), we can apply the elbow method, which requires plotting the clustering loss (Equation 1) for possible several K values and suggests picking the K value where an elbow shape occurs (after this point we do not further see a sharp decrease in the loss score).

In this part, you are expected to implement both Kmeans (by picking initial cluster centers randomly) and Kmeans++ and apply both of these methods on two datasets by utilizing the elbow method. For both methods, the Euclidean distance metric should be considered as the distance/similarity measure.

Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to k random $\boldsymbol{x}^t$

Repeat

  For all $\boldsymbol{x}^t \in \mathcal{X}$

$$b_i^t \leftarrow \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\| \\ 0 & \text{otherwise} \end{cases}$$

  For all $\boldsymbol{m}_i, i = 1, \ldots, k$

$$\boldsymbol{m}_i \leftarrow \sum_t b_i^t \boldsymbol{x}^t / \sum_t b_i^t$$

Until $\boldsymbol{m}_i$ converge

Figure 2: Pseudo code for Kmeans [12]. Equivalently b_i^ts could be checked for convergence. When none of b_i^ts change, the algorithm terminates.

1a. Take one center c_1, chosen uniformly at random from $\mathcal{X}$.
1b. Take a new center c_i, choosing $x \in \mathcal{X}$ with probability $\frac{D(x)^2}{\sum_{x \in \mathcal{X}} D(x)^2}$.
1c. Repeat Step 1b. until we have taken k centers altogether.
2-4. Proceed as with the standard `k-means` algorithm.

Figure 3: Pseudo code for Kmeans++ [16]. $D(x)$ denotes the shortest distance from data point x to the closest cluster center which has already been chosen.

Part 2 Specifications

- For Kmeans, you are expected to implement the following class (whose file name should be **Kmeans.py**):

```
class KMeans:
    def __init__(self, dataset, K=2):
        """
        :param dataset: 2D numpy array, the whole dataset to be clustered
        :param K: integer, the number of clusters to form
        """
        self.K = K
        self.dataset = dataset
        # each cluster is represented with an integer index
```

```
        # self.clusters stores the data points of each cluster in a dictionary
        self.clusters = {i: [] for i in range(K)}
        # self.cluster_centers stores the cluster mean vectors for each cluster in a dictionary
        self.cluster_centers = {i: None for i in range(K)}
        # you are free to add further variables and functions to the class

    def calculateLoss(self):
        """Loss function implementation of Equation 1"""

    def run(self):
        """Kmeans algorithm implementation"""
        return self.cluster_centers, self.clusters, self.calculateLoss()
```

The constructor takes two parameters: `dataset` and `K`. The dataset to be fed is a 2D numpy array where each row vector represents a single data point (the total number of instances is equal to the number of rows of this array). In the constructor, two variables (of type Python dictionary) are defined to store cluster points and centers: `self.clusters` and `self.cluster_centers`. Each cluster is represented with an integer index (the whole index range is 0 to K-1). Mainly, you are expected to complete the functions: `calculateLoss` and `run`. The `calculateLoss` function simply calculates the loss function of Kmeans (Equation 1). The `run` function implements Kmeans and after running, it returns the cluster centers, cluster data points, and the loss value for the final clustering. If you need to add other variables and auxiliary functions you can add them to the class implementation.

- For Kmeans++, you are expected to implement the following class (whose file name should be **Kmeansplusplus.py**):

```
class KMeansPlusPlus:
    def __init__(self, dataset, K=2):
        """
        :param dataset: 2D numpy array, the whole dataset to be clustered
        :param K: integer, the number of clusters to form
        """
        self.K = K
        self.dataset = dataset
        # each cluster is represented with an integer index
        # self.clusters stores the data points of each cluster in a dictionary
        self.clusters = {i: [] for i in range(K)}
        # self.cluster_centers stores the cluster mean vectors for each cluster in a dictionary
        self.cluster_centers = {i: None for i in range(K)}
        # you are free to add further variables and functions to the class

    def calculateLoss(self):
        """Loss function implementation of Equation 1"""

    def run(self):
        """Kmeans++ algorithm implementation"""
        return self.cluster_centers, self.clusters, self.calculateLoss()
```

It is identical to that of Kmeans but the `run` function should implement the Kmeans++ algorithm.

- You are given two datasets and two Python files, namely: **Kmeansexperiment.py** and **Kmeansplusplusexperiment.py** (which already loads the datasets). In these files, you are expected to

implement the elbow method for the methods separately by employing the two datasets. To get a loss value for a particular K value (for both methods) you can run the algorithms 10 times (with the same K values) and pick the lowest loss value as the result. In order to further alleviate randomness, this procedure can be repeated 10 times from scratch to obtain confidence intervals on loss values for each K value. Depending on these confidence intervals, the most suitable K values could be picked for both datasets with both methods. For instance, let's say we have two datasets A and B, first, we would like to consider Kmeans and we would like to determine a suitable K value for A. By starting from 2 to (say) 10 we run Kmeans on A. First, we run for K=2 10 times on A. Since each time different initial cluster centers are picked, it is highly like to get different loss values for each run. After completing 10 runs, we pick the lowest loss score for K=2, say α_1. We repeat the same procedure for the second time (K=2, 10 runs, picking the smallest loss value) to get α_2. We repeat this until we get α_{10}. The confidence interval calculated from $\alpha_1, \alpha_2, \alpha_3 ... \alpha_{10}$ is for the loss value of K=2 on the dataset A. The whole steps need to be repeated for the rest of the K values (K=3, K=4, ...). With these calculated confidence intervals (on dataset A), a K versus Loss graph could be drawn (it provides insight into the performance of Kmeans on A) and a suitable K value for A can be determined. Similarly, the whole process can be applied to dataset B and Kmeans++ could be employed in the same manner for clustering.

- In your report, please provide the K versus Loss graphs and comment on the best number of clusters for each method and dataset (In total 4 graphs).
- For this part, you are expected to make use of your own implementation for Kmeans and Kmeans++. Library implementations are not allowed.
- For this part, you can consider 10 as the highest K value to test during experiments.
- In the report, please provide all plots of the elbow method (K versus Loss) with confidence intervals and comment about the results attained for both methods and datasets, separately (i.e. what is the most suitable cluster number for each dataset (for each method separately)?).
- In your report, please provide a worst-case running time analysis for Kmeans with respect to the number of data points (N), data sample vector dimension (d), cluster number (K), and the number of iterations (I).

Note: In order to better grasp the relation between EM and Kmeans, the reader is encouraged to refer to Gaussian mixture models (GMM), which can be thought be the generalization of KMeans where clusters are represented with Gaussian distributions (with separate mean vectors and covariance matrices) and the labeling of data points to clusters are performed probabilistically [12, 13, 18]. Kmeans is a specific instance of GMM where the covariance matrix is the identity matrix (this is the reason why KMeans is biased towards spherical clusters) and labeling is done deterministically (by measuring distance to the cluster centers) (for the closest cluster the probability score is 1 and for the other clusters, it is 0).

Part 3

Kmeans and Kmeans++ methods introduced in the previous part are intrinsically random and tend to form spherical clusters. Even if Kmedoids can be utilized to alleviate this tendency, it still strongly depends on the initial configuration of clusters (it still features randomness), which requires multiple runs to obtain satisfactory results. Instead of determining initial cluster centers (in other words, enforcing initial cluster centers for a dataset), we may look forward to capturing natural groupings (due to data distribution) within a dataset itself without assuming anything about where clusters may be formed (in a sense, by introducing initial cluster centers, we dictate how clusters should be formed). This is the basic approach that has been considered by the hierarchical clustering approach. Hierarchical clustering (HC)

aims to form clusters in a bottom-up or top-down approach without a notion of a cluster center. In this part, we consider the bottom-up approach (Hierarchical Agglomerative Clustering (HAC)). With HAC, initially, every data instance is considered to be a cluster. By starting from these clusters iteratively two closest/the most similar clusters are merged until there is a whole single cluster formed. How to merge to cluster (linkage criterion) becomes the main problem since a similarity/distance measure between clusters (we consider every single instance as a cluster) needs to be defined, which is a harder problem compared to instance-instance distance/similarity measurement. To this end in this part, three linkage criteria are considered: single, complete, and average. Let $C_i = \{x_m^i | m = 1...M\}$ and $C_j = \{x_l^j | l = 1...L\}$ denote two clusters with M and L elements (instances), respectively. x_m^i denotes the m^{th} data instance of cluster i, similarly, x_l^j is l^{th} instance of cluster j. With these cluster definitions, the linkage criteria are defined as follows:

$$\text{Single Linkage}(C_i, C_j) = \min_{m,l} D(x_m^i, x_l^j)$$

$$\text{Complete Linkage}(C_i, C_j) = \max_{m,l} D(x_m^i, x_l^j)$$

$$\text{Average Linkage}(C_i, C_j) = \frac{1}{N * M} \sum_{m}^{M} \sum_{l}^{L} D(x_m^i, x_l^j)$$

where D is instance-instance similarity/distance function/metric.

By keeping the track of distance values at which two clusters are merged, a tree-like diagram which is called a dendrogram could be constructed. On this structure by picking a suitable distance value, any demanded number of clusters could be obtained (it could be likened to a cut operation on the structure, each branch on the cut represents a cluster).

In order to assess the quality of the clustering results of HAC, we can utilize the silhouette method [19, 20] (the silhouette method could also be utilized for other clustering methods). A silhouette score provides an idea of how close a data point is to its cluster instances and how well it is separated from other clusters. The silhouette score for m^{th} data instance of cluster i (x_m^i) is calculated as follows:

$$\alpha(x_m^i) = \frac{1}{|C_i| - 1} \sum_{n, n \neq i} D(x_m^i, x_n^i)$$

$$\beta(x_m^i) = \min_{j \neq i} \frac{1}{|C_j|} \sum_{l} D(x_m^i, x_l^j)$$

$$s(x_m^i) = \begin{cases} \frac{\beta(x_m^i) - \alpha(x_m^i)}{\max(\alpha(x_m^i), \beta(x_m^i))} & |C_i| > 0 \\ 0 & |C_i| = 1 \end{cases}$$

where $|C_i|$ denotes the number of elements of cluster i. $\alpha()$ finds the average distance within a cluster with respect to the given data point. $\beta()$ gets the average distance of the given point to the closest cluster instances. The value range of silhouette is [-1, 1]. A value of 1 indicates that the sample point is close to its cluster and far away from other clusters. On the other hand, -1 indicates that the data point may be in the wrong cluster. The average silhouette value of all data points gives an idea about the compactness of overall clustering.

$$\text{Average silhouette value} = \frac{\sum_c \sum_l s(x_l^c)}{\sum_c |C_c|}$$

Furthermore, when silhouette values of individual data instances of each cluster are sorted and displayed on a single plot [19, 20], we could elicit a very insightful idea about how well the clustering results are and we could determine the best number of clusters. When silhouette values of data points of all clusters are above the average silhouette value or close to it and clusters are of similar size (they contain a similar number of data points), such clustering can be considered desirable [20].

Part 3 Specifications

- For this part, you are given a dataset (as a 2D numpy array) for HAC clustering and a partial implementation that loads it, namely **part3.py**.
- You are expected to perform clustering experiments via scikitlearn's `AgglomerativeClustering` [21] on the given dataset and report the results.
- You are expected to try different HAC hyperparameter values (namely for linkage criterion and instance-instance distance/similarity measure). You should consider the following hyperparameter values: ['single', 'complete'] and ['euclidean', 'cosine'], respectively (in total 4 hyperparameter configurations).
- For each hyperparameter configuration, please add the resulting dendrogram plots in your report.
- For each hyperparameter configuration, please perform silhouette analysis for the K values of 2, 3, 4, and 5 to get the best K value. Among these 4 best configurations (('single', 'euclidean', K_1), ('single, 'cosine', K_2), ('complete', 'euclidean', K_3), ('complete', 'cosine', K_4)), report the one that attains the highest average silhouette score.
- In your report, please add all silhouette value plots and comment on them.
- You can make use of code segments provided in the scikitlearn documentation.
- The whole experimentation code of this part should be in the `part3.py` file and this file should be submitted.
- In your report, please provide a worst-case run time analysis for HAC with respect to the number of data points (N) and dimension (D) of data instances and comment on which clustering method (Kmeans or HAC) you would prefer to use with a dataset consisting of 1 million data points each of which has a dimension of 120000 (i.e 200x200 RGB image).

Regulations

1. You are expected to write your code in Python by using scikitlearn, Numpy, copy, and Matplotlib libraries.
2. Falsifying results or changing the composition of training, validation, and test data is strictly forbidden, and you will receive 0 if this is the case. Your programs will be examined to see if you have actually reached the results and if it is working correctly.
3. **Commenting:** Since all implementations are going to be inspected manually, comments are of great importance for the evaluation procedure. Please add extensive explanatory comments in all of your implementations.
4. **Late Submission:** You have a total of 5 late days for all homework without receiving a penalty. As soon as you have depleted your quota, penalization will be in effect. The late submission penalty will be calculated using $5d^2$, that is, 1 day late submission will cost you 5 points, 2 days will cost you 20 points, and 3 days will cost you 45 points. No late submission is accepted after reaching a total of 3 late days (No matter whether you have still a remaining late-day quota or not).
5. **Cheating:** Using any piece of code that is not your own is strictly forbidden and constitutes cheating. This includes friends, previous homework, or the internet. However, example code snippets shared on Scikitlearn's website can be used. **We have a zero-tolerance policy for cheating**. People involved in cheating will be punished according to university regulations.

6. **Discussion:** You must follow ODTUClass for discussions and possible updates/corrections/clarifications on a daily basis. **For the previous assignment, we received many questions that could have been asked on the discussion forum. Please ask your questions on ODTUClass unless you really think that your question is private and does not concern anyone**.

7. **Evaluation:** Your assignment is going to be graded manually.

Submission

Submission will be done via the ODTUClass system. For all parts, you are expected to upload a single pdf file named **report.pdf**. For Part 1, you are expected to upload **Distance.py**, **Knn.py**, and **Knnexperiment.py** files. For Part 2 you are expected to upload **KMeans.py** and **Kmeansplusplus.py**, **Kmeansexperiment.py** and **Kmeansplusplusexperiment.py** files. For Part 3, you are expected to upload **part3.py**.

References

1. https://scikit-learn.org/stable/modules/cross_validation.html
2. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 19.
3. Cawley, G.C.; Talbot, N.L.C. On over-fitting in model selection and subsequent selection bias in performance evaluation. J. Mach. Learn. Res 2010,11, 2079-2107.
4. https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html
5. https://machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/
6. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 8.
7. https://en.wikipedia.org/wiki/Mahalanobis_distance
8. https://numpy.org/doc/stable/reference/generated/numpy.cov.html
9. https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
11. https://numpy.org/doc/stable/reference/random/generated/numpy.random.shuffle.html
12. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 7.
13. Pattern Recognition and Machine Learning, Christopher M. Bishop, Section 9.
14. Neural Networks and Learning Machines, 3rd edition, Simon Haykin, Chapter 5.
15. Kaufman, Leonard; Rousseeuw, Peter J. (1990-03-08), "Partitioning Around Medoids (Program PAM)", Wiley Series in Probability and Statistics, Hoboken, NJ, USA: John Wiley and Sons, Inc., pp. 68–125.
16. Arthur, D.; Vassilvitskii, S. (2007). "k-means++: the advantages of careful seeding" (PDF). Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 1027–1035.
17. https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set

18. https://scikit-learn.org/stable/modules/mixture.html
19. https://en.wikipedia.org/wiki/Silhouette_(clustering)
20. https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html
21. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
22. Lecture notes
23. Announcements Page
24. Discussion Forum