Middle East Technical University | Department of Computer Engineering

CENG 499

Introduction to Machine Learning

Fall '2024-2025

Homework 2

Due date: December 26, 2024, 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 evaluation, and to 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 hyperparameters with cross-validation.
- To familiarize you with Kmeans and Kmedoid algorithms for clustering problems and provide hands-on experience in pinpointing the number of clusters with the elbow method.
- To familiarize you with the Hierarchical Agglomerative Clustering and DBSCAN algorithms for clustering and evaluating clustering results with the silhouette method.
- To familiarize you with the t-SNE, PCA, UMAP, and autoencoder dimensionality reduction methods and provide hands-on experience (at the implementation level with PCA and autoencoder) to visualize data instances in 1-D, 2-D, and 3-D spaces.

The assignment involves both coding and writing a report. You are expected to prepare your report in Latex. There should be a single report submitted for all parts.

Part 0

The first assignment involved two of the main learning problems (classification and regression), and this assignment's focus is on the clustering problem which is strongly related to the density estimation problem. With the density estimation problem, we aim to approximate the probability distribution $q(\vec{x})$ of an unknown data generation process with some model $p(\vec{x})$. In the machine learning literature, $p(\vec{x})$ can be a parametric or non-parametric model ([2] Chapters 4 and 8). In a parametric model, the density function modeled is determined via its parameters. For instance, we may assume that $p(\vec{x})$ is a Gaussian model whose parameters are the mean and covariance or a multilayer perceptron whose parameters are the trainable weights and biases in layers. Parametric models make assumptions about the distribution of data instances, which may not hold for given data instances (we may assume the data instances are distributed via a Gaussian distribution and try to model the density via a Gaussian distribution model.

However, the actual data distribution may have been sampled via some other distribution function). On the other hand, non-parametric approaches do not impose a parameter and assume anything about the distribution of data instances. As examples of non-parametric density estimation models, the k-nearest neighbor method, and Parzen window method can be given ([13] Chapter 2). They originate from the following common formalism ([13] Chapter 2):

$$p(\vec{x}) \approx \frac{k/n}{V}$$

where k is the number of data instances that fall into the space region (e.g., hypersphere, hypercube) that is centered at $\vec{x}$, V is the volume of the region and n is the total number of data instances. Parzen window fixes the space volume V and determines the k value from datasets. On the other hand, the k-nearest neighbor fixes k and determines V from datasets. Both approaches converge to true probability density values provided that some criteria are met (e.g., infinitely many data instances) ([13] Chapter 2).

In clustering, we aim to group data instances depending on their distance/similarity. We expect similar instances to be close to each other in the original data instance space as they exhibit similar behavior/effect/outcome. As a result, if we model the data distribution of each cluster/grouping separately via parametric or non-parametric models, we can combine them to obtain a density function of the following form:

$$p(\vec{x}) = \sum_{i=1}^{K} p(\vec{x}, cluster = i) = \sum_{i=1}^{K} p(\vec{x}|cluster = i)p(cluster = i)$$

where K is the total number of clusters, $p(\vec{x}|cluster = i)$ is the probability function for the likelihood of a data instance for cluster i (the existence likelihood of a data instance in cluster i) and $p(cluster = i)$ is the likelihood of cluster i (the likelihood of observing/existence of cluster i). So here $p(\vec{x})$ calculates a weighted average from cluster density models ($p(\vec{x}|cluster = i)$). Furthermore, by checking the value of $p(\vec{x} = \vec{x_j}|cluster = i)$ for a given data instance $\vec{x_j}$ we can find out which cluster is more likely to generate the given data instance. In other words, it gives an idea about how likely the data instance belongs to a particular cluster.

If we utilize Gaussian models for clusters, we obtain the Gaussian mixture model ([3] Chapter 10) and it is capable of modeling any data distribution with the desired accuracy given that enough number of cluster models are employed ([33] Chapter 3). For learning parameter values of parametric models, as discussed previously, the maximum likelihood approach could be utilized. For instance, for the Gaussian model (if we assume a given dataset is distributed via an unknown Gaussian distribution), the MLE of the parameters are ([2] Chapter 4):

$$\vec{\mu}_{MLE} = \frac{\sum_{i=1}^{N} \vec{x_i}}{N}$$

$$\Sigma = \frac{\sum_{i}^{N} (\vec{x} - \vec{\mu}_{MLE})^T (\vec{x} - \vec{\mu}_{MLE})}{N - 1}$$

When we try to apply MLE to learn the parameters (mean and covariances of all clusters and the likelihood of clusters) of a Gaussian mixture model, we cannot obtain a closed-form solution similar to the Gaussian case above ([3] Chapter 10). In this case, the expectation-maximization method could be employed ([13] Chapter 9). The expectation maximization procedure aims to find parameter values (θ) that maximize the likelihood function $p(D|\theta)$ by introducing hidden/latent variables which are denoted Z and working with $P(D, Z|\theta)$ ($p(D|\theta) = \sum_{z_i} p(D, Z = z_i|\theta)$), where D is the set of data instances. It consists of two main iterative steps: expectation and maximization. In the E step, the likelihood of the hidden variables ($P(Z|D, \theta_m)$) are calculated depending on the dataset D and current parameter values θ_m (θ values at the m^{th} iteration). In the M step, these values are used to find new parameter values (θ_{m+1}). At the very beginning of the procedure, all parameter values are initialized and these two steps (E and M) are repeated till convergence occurs. The EM method is guaranteed to converge however it may converge to a local optimal solution ([13] Chapter 9) (global optimality is not guaranteed).

Both approaches (parametric, nonparametric) can severely suffer from the curse of dimensionality problem since the number of data instances required in order to model data instance distribution as correctly as possible increases exponentially as the dimensionality of data instances increases ([13] Chapter 1).

Part 1

The maximum likelihood estimation approach results in a point estimation for parameter and hyperparameter values that solve a given problem ($\theta_{MLE} = \arg\max_\theta p(D|\theta)$, both model parameters and hyperparameters are contained in θ). So when the dataset for a given problem changes, the values obtained for model parameters and hyperparameters can change. In order to obtain a good generalization capability with methods, datasets should be rich enough to encompass possible future data instance characteristics and models should be able to learn as much information as possible from these datasets. When data is not abundant, when MLE is utilized, models can encounter underfitting or overfitting problems ([13] Chapter 1). To alleviate the underfitting problem, one can pick a complex model or increase the complexity of a method if possible (e.g., adding more layers to an MLP). To alleviate the overfitting problem with respect to model parameters, regularization techniques could be employed ([5] Chapter 4). These techniques restrain model parameters so that they do not increase the complexity of models. To alleviate the overfitting problem with respect to model hyperparameters, one can utilize the cross-validation technique ([13] Chapter 1). With this technique, hyperparameter values are subject to different validation datasets, and the hyperparameter values that attain the best average performance on these validation datasets are determined. With the regularization techniques, methods' parameter values, with the cross-validation method models' parameter values can be kept under control so that models can generalize in the future.

When a problem to be solved has abundant data 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 two 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, which 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 selected some outlier or noisy data instances, so the results attained on this single validation dataset may not be reliable for parameter tuning. Instead, we can form multiple validation datasets and average results achieved on them for parameter tuning. This is the fundamental 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 (1^{st} partition is set as the test dataset, $2^{\text{nd}}, 3^{\text{rd}}, 4^{\text{th}}...\text{K}^{\text{th}}$ partitions are used for training. Second, for the same hyperparameter value, 2^{nd} partition is considered to be the test dataset, and the rest ($1^{\text{st}}, 3^{\text{rd}}, 4^{\text{th}}...\text{K}^{\text{th}}$) is regarded as the training dataset, this procedure is repeated for each partition and 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 we set aside just before the beginning of cross-validation). The cross-validation procedure is summarized in the following figure.

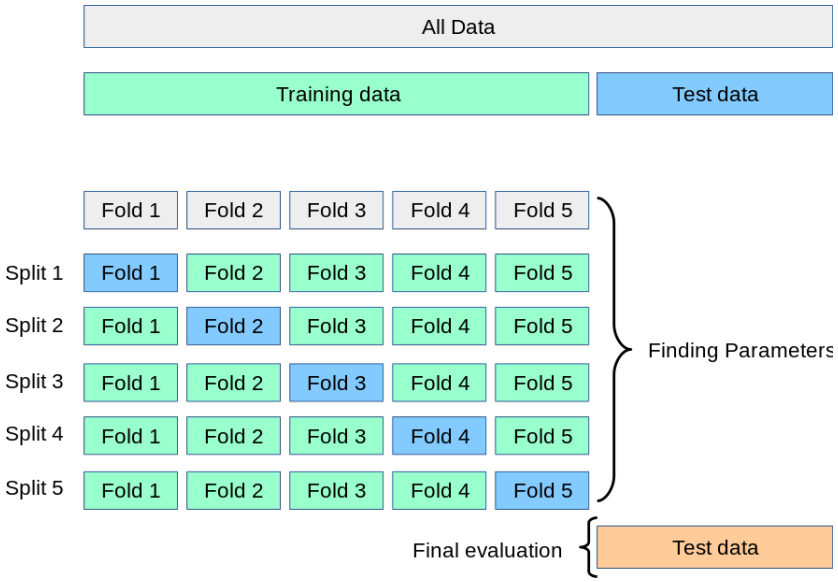

Figure 1: Cross-validation technique applied on a dataset for hyperparameter tuning. A dataset is divided into training (referred to as TRAIN1) and test (referred to as TEST1) parts (the training part is used for both training the model and tuning hyperparameters, whereas the test part is solely used 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 performance 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 performance (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 K-1 parts are used during training, 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 must 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 is still better to repeat cross-validation by shuffling the original dataset to obtain more reliable results further). In other words, the results attained during cross-validation should be statistically significant. Typically, K values of 5, 10, or 30 are considered.

Despite 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 example, for a classification task, for one class, no data instance may be present in a partition which is considered to be a test dataset). The partitions should be formed depending on the original class distribution of the training part to eliminate this problem. Such a strategy is coined as stratification ([2] Chapter 20). 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 entire 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 [4, 6, 7]. 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 [6] (It may not provide reliable results about the model's generalization capability, and it is likely to be optimistic). This serious problem can be alleviated via the nested cross-validation technique [6, 7]. The K-nearest neighbor (KNN) method is a lazy method (also, it is a non-parametric density estimation method ([2] Chapter 8)) 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 like MLPs, 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/measure 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 it) to the final decision. For classification problems, with majority voting, the class that attains the most votes is considered the predicted label of a given point. With weighted majority voting, the class label with the highest cumulative weight is returned as the resulting label (different weight combination strategies could also 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).

Since originally KNN is a non-parametric density estimation method, we can apply it to model cluster densities and utilize it for solving a classification problem. Suppose that data instances that have the same class label form a cluster, clusters are identified and KNN has been utilized to model the probability distribution of these clusters ([13] Chapter 2). So for each cluster i:

$$p(\vec{x}|cluster = i) = \frac{k_i/n_i}{V}$$

where k_i is the number of data instances belonging to class/cluster i in a space region whose volume is V and n_i is the total number of data instances in class/cluster i. Furthermore, we can estimate the cluster/class likelihood probabilities depending on how many data instances they contain (if a cluster has few data instances, it is less likely to observe/encounter that cluster, otherwise we are highly likely to observe/encoder it):

$$p(cluster = i) = \frac{n_i}{N}, \sum_{c=1}^{K} n_c = N$$

For a given new data instance $\vec{x}_{new}$, its cluster/class can be determined by checking the posterior probability scores for each cluster/class:

$$p(cluster = i|\vec{x} = \vec{x}_{new}) = \frac{p(\vec{x} = \vec{x}_{new}|cluster = i)p(cluster = i)}{p(\vec{x} = \vec{x}_{new})} \tag{1}$$

$$= \frac{\frac{k_i/n_i}{V} * \frac{n_i}{N}}{\frac{k/N}{V}} = \frac{k_i}{k} \tag{2}$$

Where k is the number of data instances in the space region and k_i is the number of data instances belonging to cluster/class i. In order to minimize the error probability of misclustering/mislabeling, the

new data instance should be assigned/labeled with the label of the cluster/class that attains the highest posterior probability ($p(cluster = i|\vec{x} = \vec{x}_{new})$). As the posterior probability calculation above indicates, this strategy corresponds to labeling with majority voting. Since posterior probabilities are calculated, the minimum risk minimization strategy could be utilized while labeling data instances provided that a risk matrix is given for a problem.

KNN requires tuning two main hyperparameters: K and distance/similarity metric. The optimal hyperparameter configuration varies depending on data distribution and the characteristics of the features of data samples (e.g., number of features, discrete/continuous-valued, etc.). 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, Minkowski (p=2 leads to Euclidian distance), 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 [8] 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:

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

$$\text{Cosine}(\vec{x}, \vec{y}) = 1 - \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| * \|\vec{y}\|}, \text{where } \cdot \text{ denotes the dot product operation.}$$

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

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** [9] and **numpy.linalg.inv** [10] 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

    def predict(self, instance):
        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 (e.g., 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 a partial 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 least five configurations). Please repeat this cross-validation procedure five 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 [11]. For data shuffling with numpy, you may refer to [12].
- You may prefer to consider other performance metrics (it is up to you) (e.g., precision, recall, f1 score, 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.

for data shuffling can we use scikit and numpy implenetations ??

Note: Although the KNN method does not feature any randomness (multiple runs of a particular hyperparameter setting will yield the same/identical results), 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 performance 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 from data samples lacking label information. One crucial problem in this paradigm is to cluster data samples depending on their similarities/dissimilarities/closeness and extract insightful knowledge from these groupings. To this end, this and the following parts are devoted to the clustering problem and clustering analysis. The K-means 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) ([2] Chapter 7), ([13] Chapter 9). 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, the procedure maximizes the likelihood of data samples by iterating through the E and M steps. 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 K-means, the information that specifies a data point belongs to which cluster (cluster label) is a hidden variable, and the mean vectors of clusters are parameters. K-means 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 nearest cluster (it labels the data samples with the label of the closest cluster mean) (this constitutes the E-step for K-means, 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 (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 K-means, these operations can be formulated as the following loss function ([2] Chapter 7):

$$L = \sum_{s}^{n} \sum_{k}^{K} l_k^s \|x_s - \mu_k\|^2 \tag{3}$$

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). Minimizing 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] Chapter 5) ($Variance = \mathbb{E}[(x - \mu)^2]$). Hence, K-means aims to minimize variance within each cluster (it aims to form clusters whose inner variances are minimal).

As the main distance/similarity measure Euclidean distance is considered for K-means (the method requires a centroid notation to work (which is available in Euclidean space), datasets may contain data features for which averaging/calculating mean is meaningless, e.g., marital status: single, married (mean/average cannot be defined))). This introduces bias to K-means and leads K-means 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 K-medoids method [15] which considers a data sample as the cluster representative rather than the mean of cluster data points (for K-means, clusters are summarized/identified as a mean vector of data points, for K-medoids a single data point is considered). In the M step, instead of calculating a centroid (mean of cluster data samples), the data point closest to all cluster data points is picked to be the new cluster center. Similarly, data points are labeled in the E step 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 K-means:

$$L = \sum_{s}^{n} \sum_{k}^{K} l_k^s d(x_s, c_k) \tag{4}$$

where $d(x, y)$ is an arbitrary distance/similarity metric (function, possibly not Euclidean distance), c_k is the representative data point for the cluster k. Minimizing this loss function corresponds to forming clusters whose data points are close to the cluster representative data point (more similar data points are grouped).

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

K-means (also K-medoids) strongly depend on the initially picked cluster centers while generating a final clustering result. Several strategies could be considered to this end. We could run K-means (by selecting the initial cluster centers randomly) many times (e.g., 50 times) and pick the one that attains the smallest loss error (Equation 3) after running. Or we can choose instances that are distant from each other depending on a fixed distance threshold value. A better approach is from the K-means++ algorithm [16]: picking initial clusters with a probability score depending on the distance between data samples. K-means++ differs from K-means solely in determining initial cluster representatives. As the first step, K-means++ 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 selecting all cluster centers, standard E and M steps of K-means are applied. With this strategy, K-means++ has been shown to complete faster and yield better clustering results compared to K-means [16].

Along with how to pick initial cluster centers, the most critical hyperparameter of K-means 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 (e.g., in a handwritten digit dataset, ten different clusters are expected). If we cannot access such information, we can use the elbow method, which requires plotting the clustering loss (Equation 3) 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). As a direct alternative to the loss value in the elbow method, the average silhouette score can be considered (please refer to Part 3 for further information about it) could be considered. Similar to K versus Loss, a K versus Average Silhouette Score graph can be drawn and a suitable "elbow point" point could be identified. Furthermore, silhouette scores of data instances in clusters can be plotted (silhouette graphs, please refer to Part 3 for further information), and depending on the distribution of values, a suitable K value could be identified. Another possibility is to apply information-theoretic measures [17]. As a last resort (but not the least), we can project the dataset into a lower dimension (e.g., 2D, 3D) and visually determine the number of clusters from the projected data.

Both K-means and K-means++ algorithms utilize the Euclidean distance to measure similarity/dissimilarity between data instances, which restricts their applicability to data instances with many features (high dimensional) and mixed feature values (discrete and continuous). From the Euclidean distance perspective, data instances look-alike in a high dimensional space, which is a severe problem likely to lead to inaccurate clustering results. For example, let's consider the following 4-D data instances: [3, 3, 3, 0], [0, 1, 0, $\sqrt{6}$], [0, 0, 0, $\sqrt{5}$]. These instances are equally distant to [1, 1, 1, 0]. As can be seen, this data instance is a scaled form of the first data instance. We expect it to be more similar to the first data instance (they share the same direction if we regard them as vectors), but due to the Euclidean distance measure, the other data instances are not different from the first one. Another important consideration is that when some features of a data instance have discrete values, we cannot utilize the Euclidean distance to quantify similarity/dissimilarity between data instances. For example, one feature may represent occupation or marital status, and we cannot calculate a squared distance between different occupations or marital statuses. On the other hand, since K-medoids is capable of working with an arbitrary similarity/distance measure, it can easily be employed with these types of data instances. Furthermore, since other distance metrics could be utilized, clustering results may be better than those obtained via Euclidean distance. However, such an appealing merit comes with a computation overhead. K-means calculates the cluster centers by averaging data instances in clusters. In contrast, K-medoids has to measure the similarity

between each possible data instance pair in clusters to find cluster centers.

As humans, we are very accustomed to 1-D, 2-D, and 3-D spaces and understand relationships between objects and their behavior in these spaces. When data instances have a dimensionality higher than 3, we cannot fully observe them, and our common sense knowledge about 1-D, 2-D, and 3-D spaces may no longer apply to those data instances. For instance, most of the volume of a sphere with radius 1 in high dimensions lies in a thin shell near the surface of the sphere ([13] Chapter 1), which is against our common sense with spheres in 3-D. Data instances for different problem types can have different dimensionality/number of features, and every feature of data instances may not help solve a problem. The number of features plays an essential role in the computation and storage demands of a problem. It directly or indirectly determines the complexity of the problem, thus the number of parameters in an ML algorithm. For instance, if data instances of the problem/task lie in a high dimensional space, say 500, the function to be learned by an MLP to solve the problem should be more complex compared to the function when they had lower dimensionality, say 3, (it is more complex in the sense that there are more dimensions thus the search space for solution functions is larger. In this regard, more weights and hidden layers may be needed to learn the complex function). A similar argument could be made for other methods, such as decision trees, SVMs, .etc as well. The more features, the more branching there are in a decision tree (the bigger the tree is).

To reduce the complexity and storage demands of a problem, we can utilize dimensionality reduction methods ([2] Chapter 6). After reducing the dimensionality of data instances, we can use less complex ML algorithms to solve the problem. However, dimensionality reduction could be considered a data compression operation since it tries to represent a higher dimensional data point with a lower dimensional data point, and information loss could occur. Thus, helpful information for a problem solution could be lost, which can render the problem unsolvable (algorithms can perform very poorly due to the fact that data instances do not contain enough information after the reduction operation). In general, the success of applying dimensionality reduction operation is task/problem, data (dataset), and algorithm dependent.

We have previously mentioned that one way to determine the number of clusters in a clustering problem is to utilize dimensionality reduction methods. Simply, we can map/transform original data instances as 1-D, 2-D, or 3-D data instances and visualize them. We can pick a K value depending on the number of clusters formed in the visualization. The visualization quality depends on data instances (their distribution) and the dimensionality reduction method utilized. Various dimensionality reduction algorithms handle the dimensionality reduction problem by approaching it with different assumptions such as Principal Component Analysis (PCA), Multidimensional Scaling, Linear Discriminant Analysis (LDA), Isomap, Locally Linear Embedding (LLE) ([2] Chapter 6), Self-Organizing Maps (SOM) ([14] Chapter 9), T-distributed Stochastic Neighbor Embedding (t-SNE) [18, 19], Uniform Manifold Approximation and Projection (UMAP) [20, 21], Autoencoder [22, 23]. Since there are many within the scope of this assignment, we focus on the four widely used methods: PCA, t-SNE, UMAP, and autoencoder. PCA is the simplest among them and aims to linearly project data instances onto particular vectors such that the projected data instances have the highest variance (scatter) ([2] Chapter 6), which turns out to find eigenvectors of the covariance matrix of data instances. T-SNE, on the other end, aims to model the distribution of data instances in a high dimension and maps these data instances into a low dimension (1-D, 2-D, or 3-D) by preserving the modeled data distribution [18]. Like t-SNE, UMAP aims to form a topological representation of data instances in a high dimension and map these data instances to a low dimensional space by preserving the modeled representation [20].

In this part, you are expected to employ K-means and K-medoids algorithms on two datasets by utilizing the elbow technique (with K-means loss and silhouette score), and dimensionality reduction methods (namely, PCA, t-SNE, UMAP, and autoencoder).

Part 2 Specifications

- You are given two datasets and two Python files, namely: **K-meansexperiment.py** and **K-medoidssexperiment.py** (which already load the datasets). In these files, you are expected to

implement the elbow method with both K-means loss and silhouette score for the methods separately by employing the two datasets. To get a loss value/an average silhouette score for a particular K value (for both methods), you can run the algorithms ten times (with the same K values) and pick the lowest loss value/average silhouette value as the result. To further alleviate randomness, this procedure can be repeated ten times from scratch to obtain an average loss value/silhouette score for each K value (similarly, a confidence interval can be calculated with these scores if need be). Depending on these average loss values/silhouette scores, 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 K-means, and we would like to determine a suitable K value for A. By starting from 2 to (say) 10 we run K-means on A. First, we run for K=2 10 times on A. Since each time different initial cluster centers are picked, it is highly likely to get different loss values/average silhouette values for each run. After completing ten runs, we choose the lowest loss score/average silhouette score for K=2, say α_1. We repeat the same procedure for the second time (K=2, ten runs, picking the smallest loss value/average silhouette score) to get α_2. We repeat this until we get α_{10}. The average loss/average silhouette value calculated from $\alpha_1, \alpha_2, \alpha_3 ... \alpha_{10}$ is for the loss value/average silhouette value of K=2 on dataset A. These steps need to be repeated for the rest of the K values (K=3, K=4, ...). With these average loss values/silhouette values (on dataset A), a K versus loss/average silhouette graph could be drawn (it provides insight into the performance of K-means on A), and a suitable K value for A can be determined. Similarly, the whole process can be applied to dataset B, and K-medoids could be employed similarly for clustering. Confidence intervals calculated via $\alpha_i s$ could also be shown in K versus loss/silhouette graphs to represent experiment results better and provide more detailed information.

- In your report, please provide the K versus Loss and K versus Silhouette graphs separately and comment on the best number of clusters for each method and dataset (total of 8 graphs, two datasets, and two algorithms).

- For this part, you are expected to make use of the scikitlearn implementation for the K-means method and silhouette score calculations. The library scikit-learn-extra [32] should be considered for the implementation of the K-medoids algorithm. You can consider ten as the highest K value to test during experiments.

- In the report, please provide all plots of the elbow method (K versus Loss, K versus Average Silhouette Score) with confidence intervals (you can provide confidence intervals as a table and depict them on K versus loss and K versus silhouette graphs) and comment about the results attained for both methods and datasets separately (e.g., what is the most suitable cluster number for each dataset (for each method and evaluation criteria (loss/silhouette) individually)?, Do elbow locations identified via K-mean losses and average silhouette scores match for each method and dataset?).

- For this part, you are expected to implement the PCA and autoencoder methods (for t-SNE and UMAP you can utilize libraries).

- You are provided a Python file named **part2_dimensionalityreduction.py**, which already loads the datasets of this part. In this file, you are expected to employ the four dimensionality reduction methods to visualize the datasets in 2-D. You are expected to utilize your own implementations for the PCA and autoencoder methods. You can try different hyperparameter configurations for t-SNE and UMAP to enhance visualization results (please ensure that the hyperparameter configurations you have tested should be explicit in the file). In the report, please provide the "best" dimensionality reduction results (as matplotlib scatter plots) of each method for both datasets (the definition of "best" is up to you) (if you would like, you can add 3-D visualizations too). Also, please briefly discuss whether the number of clusters you have identified via the elbow method matches the number of clusters in visualizations for both datasets separately. For t-SNE and UMAP implementation

hyperparameters, you can refer to [19] and [21], respectively. For the autoencoder architecture, in your implementation, you can consider a single fixed set of hyperparameters (learning rate, iteration count, number of hidden layers, activation functions utilized, optimizer utilized) for projecting datasets (due to the time constraints of the assignment). A better approach is to try testing different hyperparameters with the model in order to obtain the best results. The PCA method does not possess a hyperparameter to tune.

- You are provided as file name **pca.py** where you are expected to implement the PCA method using solely the numpy library. In **part2_dimensionalityreduction.py**, you are expected to import this file for performing dimensionality reduction via the PCA method. In this file a PCA class definition template is given:

```python
import numpy as np

class PCA:
    def __init__(self, projection_dim: int):
        """
        Initializes the PCA method
        :param projection_dim: the projection space dimensionality
        """
        self.projection_dim = projection_dim
        # keeps the projection matrix information
        self.projection_matrix = None

    def fit(self, x: np.ndarray) -> None:
        pass
    def transform(self, x: np.ndarray) -> np.ndarray:
        pass
```

With this class, you are expected to implement two functions: fit and transform. The fit function should extract the projection matrix of PCA from a given data matrix. Here a data matrix is a Nxd matrix where each data instance is represented as a row vector of size d and there are N data instances. The transform function should return the projected version of a given data matrix utilizing the extracted projection matrix in the fit function. Further implementation details are provided in **pca.py**.

- Similarly, you are provided a file named **autoencoder.py** where you are expected to implement the autoencoder method using solely numpy and torch libraries. In **part2_dimensionalityreduction.py**, you are expected to import this file for performing data projection via the autoencoder method. In the file, two class definition templates are provided:

```python
import numpy as np
import torch.nn as nn
import torch
class AutoEncoderNetwork(nn.Module):
    def __init__(self, input_dim: int, output_dim: int):
        super().__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        """
        Your autoencoder model definition should go here
        """
    def project(self, x: torch.Tensor) -> torch.Tensor:
        pass
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        pass
```

The AutoencoderNetwork class should implement a regression MLP architecture where the number of inputs is equal to the number of outputs and one of the hidden layers (bottleneck layer) should have two nodes in order to perform a projection from the original input data space to 2-D. The neural network should approximate the identity function ($f(x) = x$). In other words, the network should learn to generate the input data instances at its output layer as closely as possible. To this end, it can be trained with the reconstruction loss:

$$Reconstruction\ Loss = \frac{1}{N}\sum_{i=1}^{N} ||\vec{x_i} - \vec{y_i}||^2$$

where $\vec{x_i}$ is an original data instance and $\vec{y_i}$ is the output of the autoencoder network when $\vec{x_i}$ is fed to it. Since the bottleneck layer has a dimensionality of 2, the network part from the input layer to it (the encoder part) has to perform an encoding operation because typically the input dimensionality is higher than 2, thus high dimensional data instances have to be encoded into 2-D. The layer between the bottleneck layer and the output layer (the decoder part) has to perform a decoding operation since from 2-D data it has to learn to recover original data instances. Since the architecture throttles forward pass within its bottleneck layer, the architecture has to learn the most useful way of encoding and decoding data instances. The output of the bottleneck layer can be used to visualize data instances in 2-D (if there were 3 nodes in this layer, then we would be able to project data instances into 3-D, but for this assignment, 2 nodes should be employed in this layer.). The encoder part and decoder parts can have an arbitrary number of other hidden layers whose node count is higher than 2 in order to simplify the data encoding and decoding operations (with further hidden layers, the encoder and decoder parts can implement more complex encoding and decoding operations). For the AutoencoderNetwork class, you are expected to first define your architecture in the constructor, the structure of the neural network is up to you (it should contain at least one hidden layer, which is the bottleneck layer). The forward function should implement the ordinary regression network forward pass operations. The project function should return the output of the bottleneck layer of the network. This class is mainly utilized by the AutoEncoder class whose definition and details are given below. Further implementation details are provided in **autoencoder.py**.

```
1  import numpy as np
2  import torch.nn as nn
3  import torch
4
5  class AutoEncoder:
6      def __init__(self, input_dim: int, projection_dim: int, learning_rate:
       float, iteration_count: int):
7          """
8          Initializes the Auto Encoder method
9          :param input_dim: the input data space dimensionality
10         :param projection_dim: the projection space dimensionality
11         :param learning_rate: the learning rate for the auto encoder
       neural network training
12         :param iteration_count: the number epoch for the neural network
       training
13         """
14         self.input_dim = input_dim
15         self.projection_matrix = projection_dim
16         self.iteration_count = iteration_count
17         self.autoencoder_model = AutoEncoder(input_dim, projection_dim)
18         """
19             Your optimizer and loss definitions should go here
20         """
21
```

```
    def fit(self, x: torch.Tensor) -> None:
        pass

    def transform(self, x: torch.Tensor) -> torch.Tensor:
        pass
```

The class implementation first requires some MLP training-related hyperparameter values to be set (the MLP architecture and the hyperparameter values are up to you) and creates the autoencoder neural network using the AutoEncoder class. In the fit function, you are expected to train the network created in the constructor by employing the reconstruction loss. Similarly, the transform function should return the output of the bottleneck layer of the network trained via the fit function for a given data matrix. In other words, this function should return the projected version of a given data matrix. Further implementation details are provided in **autoencoder.py**.

- In your report, please provide a worst-case running time analysis for both K-means and K-medoids with respect to the number of data points (N), data sample vector dimension (d), cluster number (K), and the number of iterations (I).

Note: To better grasp the relation between EM and K-means, the reader is encouraged to refer to Gaussian mixture models (GMM), which can be thought be the generalization of K-means 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 ([2] Chapter 7), ([13] Chapter 9), [24]. K-means is a specific instance of GMM where the covariance matrix is the identity matrix (this is the reason why K-means is biased towards spherical clusters) and labeling is done deterministically (by measuring the distance to the cluster centers) (for the closest cluster the probability score is 1 and for the other clusters, it is 0).

Note: K-medoids algorithm has an extra hyperparameter compared to K-means: distance metric. Within the scope of this assignment, we have considered it as fixed (as cosine). But in general, while applying it to other works/projects, we need to consider other possible metrics and test them during experiments. To this end, we can repeat the experiment procedure described above for each possible distance measure from scratch and report the results of the distance measure that attains the smallest average loss/silhouette value at the elbow location.

Part 3

K-means and K-means++ methods introduced in the previous part are intrinsically random and tend to form spherical clusters. Even if K-medoids can be utilized to alleviate this tendency, it still strongly depends on the initial configuration of clusters, 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 the density-based spatial clustering of applications with noise (DBSCAN) [29, 30, 31] method considers. The method labels data instances as cluster members or noise (data points outside any cluster) depending on how data instances are connected (e.g., density reachable [29]) to each other. It can form arbitrarily shaped clusters, unlike K-means, and features no parameters (e.g., cluster mean like K-means) and three main hyperparameters: the distance metric, the minimum number of neighbor data instances for a data point to be regarded as a core point, and the maximum instance (ϵ) with which two points are determined as neighbors or not. Once core data points (in densely populated data instance space regions) are identified, clusters are expanded with other data instances depending on how they are related to these core points. Unlike K-means, K-medoids, and DBSCAN, hierarchical clustering (HC) aims to form clusters in a bottom-up or top-down approach without the notion of cluster center/core data point. In this part, we

consider the bottom-up approach (Hierarchical Agglomerative Clustering (HAC)). With HAC, initially, every data instance is considered a cluster. By starting from these clusters iteratively, the two closest/the most similar clusters are merged until a single cluster is formed. How to merge two clusters (linkage criterion) becomes the main problem since a similarity/distance measure between clusters needs to be defined, which is a harder problem than instance-instance distance/similarity measurement. To this end, in this part, three linkage criteria are considered ([2] Chapter 7), [34]: 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 ([2] Chapter 7), [34]:

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

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

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

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

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

To assess the quality of the clustering results of HAC and DBSCAN, we can utilize the silhouette method [25, 26], ([5] Chapter 21) (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:

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

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

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

where $|C_i|$ denotes the number of elements of cluster i. $\alpha()$ finds the average distance within a cluster with respect to a given data point. $\beta()$ gets the average distance of the given point to the closest cluster instances. The value range of the silhouette score 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 [25, 26], we could elicit a very insightful idea about how well the clustering results are, and we could determine the optimal 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 [26]. Readers are encouraged to refer to ([28] Chapter 2) for more detail regarding the silhouette method application.

Part 3 Specifications

- For this part, you are given a dataset (as a 2D numpy array) for clustering with HAC and DBSCAN methods, and a partial implementation that loads it, namely **part3.py**.
- You are expected to perform clustering experiments via scikitlearn's `AgglomerativeClustering` [27] and `DBSCAN` [31] 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, four hyperparameter configurations).
- For the DBSCAN algorithm, you are expected to try different hyperparameter values (namely distance/similarity metric, the maximum neighborhood distance (ϵ), and the minimum number of neighbor data instances). You should consider a range of hyperparameter values which are up to you (these hyperparameter values should be visible in your implementation) and report the results whose average silhouette score is among the four highest average silhouette scores attained by all DBSCAN configurations.
- For the best four DBSCAN hyperparameter configurations, perform silhouette analysis to get the best K value. Please add all silhouette value plots for these four best hyperparameter configurations and comment on them. Among these, report the one that attains the highest average silhouette score.
- For each HAC hyperparameter configuration, please add the resulting dendrogram plots in your report.
- For each HAC hyperparameter configuration, please perform silhouette analysis for the K values of 2, 3, 4, and 5 to get the best K value. Among these four 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 for HAC and comment on them.
- You can make use of code segments provided in the scikitlearn documentation.
- The whole experimentation code (with HAC and DBSCAN) of this part should be in the **part3.py** file, and this file should be submitted.
- Similar to Part 2, you are given a Python file named **part3_dimensionalityreduction.py**, which already loads the dataset of this part. In this file, you are expected to employ the three dimensionality reduction methods (PCA, t-SNE, and UMAP) for visualization of the dataset of this part in 2-D. You can try different hyperparameter configurations for t-SNE and UMAP to enhance visualization results (please ensure that the hyperparameter configurations you have tested are explicit in the file). In the report, please provide the "best" dimensionality reduction results (as matplotlib scatter plots) of each method (the definition of "best" is up to you) (if you would like, you can add 3-D visualizations too). Also, please briefly discuss whether the number of clusters you have identified via the silhouette method for HAC and DBSCAN matches the number of clusters in visualizations for the dataset. For t-SNE and UMAP implementation hyperparameters, you can refer to [19] and [21], respectively. **For PCA, you can utilize your implementation (of Part 2) or the scikitlearn implementation.**
- 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 (K-means or HAC) you would prefer to use with a dataset consisting of 1 million data points each of which has a dimension of 120000 (e.g., 200x200 RGB image).

Note: The silhouette graphs introduced in this section could be utilized for assessing clustering results of K-means and K-medoids algorithms as well however these graphs are likely to change from run to run due to the randomness in these methods, which renders inspecting results difficult. Still, as an alternative to the elbow method for determining the number of clusters, these graphs can easily be employed with these methods.

Regulations

1. You are expected to write your code in Python using scikitlearn, scikit-learn-extra, numpy, torch, copy, pickle, umap-learn, 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 the reported results are obtained from them and if it is working correctly.

3. **Late Submission:** You have a total of 72 late hours 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 (0-24 hours) late submission will cost you 5 points, 2 days (24-48 hours) will cost you 20 points, and 3 days (48-72 hours) will cost you 45 points. No late submission is accepted after 3 days from the deadline of the assignment.

4. **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.

5. **Discussion:** You must follow ODTUClass for discussions and possible updates/corrections/clarifications on a daily basis.

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

Submission

Submissions will be done via the ODTUClass system. You are expected to upload a single pdf file named **report.pdf** for all parts. For Part 1, you are expected to upload **Distance.py**, **Knn.py**, and **Knnexperiment.py** files. For Part 2 you are expected to upload **pca.py**, **autoencoder.py**, **Kmeansexperiment.py**, **Kmedoidsexperiment.py** and **part2_dimensionalityreduction.py** files. For Part 3, you are expected to upload **part3.py** and **part3_dimensionalityreduction.py**.

References

[1] https://scikit-learn.org/stable/modules/cross_validation.html

[2] Introduction to Machine Learning, 4th edition, Ethem Alpaydın

[3] Pattern Classification, 2nd Edition by Richard O. Duda, Peter E. Hart, David G. Stork

[4] 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

[5] Murphy, K.P., 2022. Probabilistic machine learning: an introduction. MIT Press.

[6] https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html

[7] https://machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/

[8] https://en.wikipedia.org/wiki/Mahalanobis_distance

[9] https://numpy.org/doc/stable/reference/generated/numpy.cov.html

[10] https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html

[11] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

[12] https://numpy.org/doc/stable/reference/random/generated/numpy.random.shuffle.html

[13] Pattern Recognition and Machine Learning, Christopher M. Bishop.

[14] Neural Networks and Learning Machines, 3rd edition, Simon Haykin.

[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] Van der Maaten, L. and Hinton, G., 2008. Visualizing data using t-SNE. Journal of machine learning research, 9(11).

[19] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

[20] McInnes, L., Healy, J. and Melville, J., 2018. Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426.

[21] https://umap-learn.readthedocs.io/en/latest/

[22] Kramer, M.A., 1991. Nonlinear principal component analysis using auto-associative neural networks. AIChE journal, 37(2), pp.233-243.

[23] https://en.wikipedia.org/wiki/Autoencoder

[24] https://scikit-learn.org/stable/modules/mixture.html

[25] https://en.wikipedia.org/wiki/Silhouette_(clustering)

[26] https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

[27] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

[28] Kaufman, Leonard and Rousseeuw, Peter. (1990). Finding Groups in Data: An Introduction To Cluster Analysis. 10.2307/2532178.

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96). AAAI Press, 226–231

[30] https://scikit-learn.org/stable/modules/clustering.html#dbscan

[31] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN

[32] https://scikit-learn-extra.readthedocs.io/en/stable/

[33] Deep Learning, Goodfellow, I. and Bengio, Y. and Courville, A., 2016

[34] Lecture notes

[35] Announcements Page

[36] Discussion Forum