Gliwice, 24.06.2020

Semester: 2

Group: 2

Section: FB_Monday_10_15

COMPUTER PROGRAMMING LABORATORY

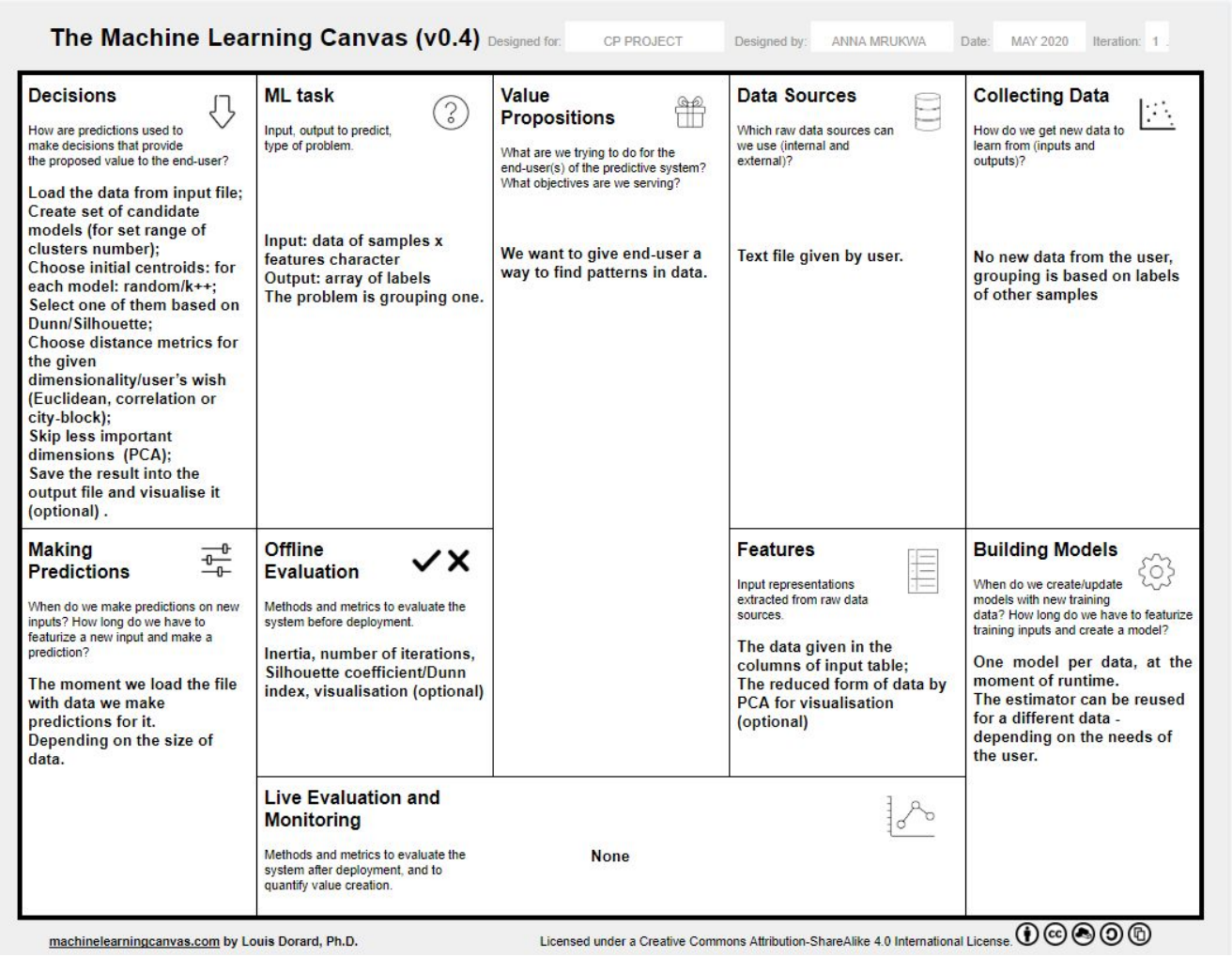
Author: Anna Mrukwa

E-mail: annamru993@student.polsl.pl

Tutor: Franciszek Bińczyk

1. Task topic

During the process of preparing my project I created ML Canvas dedicated to this programme. It helped me to get a clearer look at the architecture and applications of the whole composition. Because of that I place it below as an important part of the whole procedure.



The Machine Learning Canvas (v0.4) Designed for: CP PROJECT Designed by: ANNA MRUKWA Date: MAY 2020 Iteration: 1

Decisions How are predictions used to make decisions that provide the proposed value to the end-user? **Load the data from input file; Create set of candidate models (for set range of clusters number); Choose initial centroids: for each model: random/k++; Select one of them based on Dunn/Silhouette; Choose distance metrics for the given dimensionality/user's wish (Euclidean, correlation or city-block); Skip less important dimensions (PCA); Save the result into the output file and visualise it (optional) .**	**ML task** Input, output to predict, type of problem. **Input: data of samples x features character Output: array of labels The problem is grouping one.**	**Value Propositions** What are we trying to do for the end-user(s) of the predictive system? What objectives are we serving? **We want to give end-user a way to find patterns in data.**	**Data Sources** Which raw data sources can we use (internal and external)? **Text file given by user.**	**Collecting Data** How do we get new data to learn from (inputs and outputs)? **No new data from the user, grouping is based on labels of other samples**
Making Predictions When do we make predictions on new inputs? How long do we have to featurize a new input and make a prediction? **The moment we load the file with data we make predictions for it. Depending on the size of data.**	**Offline Evaluation** Methods and metrics to evaluate the system before deployment. **Inertia, number of iterations, Silhouette coefficient/Dunn index, visualisation (optional)**		**Features** Input representations extracted from raw data sources. **The data given in the columns of input table; The reduced form of data by PCA for visualisation (optional)**	**Building Models** When do we create/update models with new training data? How long do we have to featurize training inputs and create a model? **One model per data, at the moment of runtime. The estimator can be reused for a different data - depending on the needs of the user.**
	Live Evaluation and Monitoring Methods and metrics to evaluate the system after deployment, and to quantify value creation.	**None**		

machinelearningcanvas.com by Louis Dorard, Ph.D. Licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

2. Project analysis

As mentioned above, the problem to solve here is finding patterns in the data and grouping them accordingly into clusters, whose number does not have to be specified by the user. To do so, I am using the k-means algorithm. It is simple grouping algorithm that is based on the following steps:

- Initialize algorithm by choosing initial centroids - I decided to implement both Mac Queen's random initialization, consisting of randomly chosen points from a given dataset and k++ initialization, explained below.
- To complete initialization, we label all the points belonging to the given set - we assign them to the clusters based on their distance (measured in the

Euclidean, correlation or cityblock metrics) to the closest one. Each such group consisting of data points is a cluster with a common label.

- After completing initialization recalculating the centroids begins. As the name of the method suggests, we choose new centroid coordinates based on the mean of coordinates in each dimension of all the points belonging to the cluster.
- By doing it, we possibly moved the centroids and previously assigned labels are no longer corresponding to the actual smallest distance between a point and a member of the centroid group. To fix this, we label all the points again.

The two last steps are repeated for the determined number of iterations. This improves the result of clustering. We provide the labels of each point to the user.

Most of the basic problems are already solved in my implementation of this algorithm in C and their solutions can be applied similarly. To describe new issues more precisely I skip discussing these problems and carry on with the rest of this implementation.

The part that proved to be more challenging was creating adequate assignment operators and appropriate constructors. As I created my own structure containing the dynamically allocated array and two parameters describing its dimensions, I could not simply copy a pointer from one object to another.

I also added another initialization method to k-means - it speeds up the algorithm, choosing not random, but more different initial centroids. K++ iniltialization chooses the first centroid at random. For the next ones, however, we compute the distance from each point to the closest centroid. Then we choose the next centroid using weighted probability proportional to this distance squared.

Moreover, to enhance the whole process and enable future visualisation, I implemented one more unsupervised learning algorithm - this one for feature extraction. Principal Component Analysis lets us rotate the coordinate system in which we position the data in such a way that each dimension is independent; more intuitional explanation would be a comparison to finding the best way to take a photo of a group of people: we want to see them as clearly as possible without unnecessary obscuring of the photo. To do so, we rotate the camera until we get the best version of the image. The line we project the people onto and the line orthogonal to it are what we are looking for. The implementation consists of calculating the covariance matrix for the input data and then calculating eigenvectors of this matrix. As we want to reduce the number of dimensions of our input and preserve as much of the initial information as possible, we sort the eigenvectors by their eigenvalues in the descending order. We preserve only the first N eigenvectors, where N is the number of dimensions we want the data to be reduced to. Then we multiply the initial data by the remaining eigenvectors.

I did not want to use any external libraries computing the eigenvectors and eigenvalues of the matrix, so I wrote QR decomposition combined with the power method. To ensure the orthonormalization every iteration, I also applied modified Gram-Schmidt algorithm to prevent numerical errors.

As I wanted the program to choose the best number of clusters by itself, not by the specification of the user, I also implemented Dunn Index and Silhouette score. Dunn Index calculates the smallest distance between two clusters in the current grouping and then divides it by the greatest defined distance within the cluster. The linkages can be chosen by the user. This shows us how much the points with the same labels are different and how much different clusters are mixed together. The bigger the index, the better the clustering. Silhouette score

informs us about the same. I calculate it for the whole set as the average score for every point belonging to this set. Silhouette score for one point is defined as: $s = \frac{b-a}{max(a,b)}$, where a - mean intra cluster distance (from this point to all the other in the same cluster), b - mean distance to the closest cluster the sample does not belong to. Scores close to 1 mean great assignment, 0 - overlapping clusters, negative values - samples are in the wrong clusters.

3. External specification

The easiest way to download the program is by clicking here. I strongly recommend reading README and taking a look at the specially created exemplary file.

The input file is not checked in this implementation - the program will not inform you about the typos and will not work correctly. This is because I wanted to speed up the whole process, as loading the data takes up a lot of time.

Besides that, most of the basic functions (such as vector division) check for the incorrect behaviour, terminating with a message. This should help you diagnose the problem in case of possible different usage of the header parts.

K-means and PCA implementations imitate the parts of scikit-learn pipelines: you can use fit, predict for k-means, as well as fit_predict, grouping your data and returning the labels, whereas for PCA you get fit, transform and fit_transform, performing the decomposition and transforming your data.

While using GapSearch or SilhouetteSearch, remember that after fitting the estimator you used remained unchanged - the desired one with the best score is still inside the searching object and you have to extract the labels from it.

4. Internal specification

Here I present classes, methods and functions used in the project:

1. In the file Vectors.h:

```
enum class dist_ { Euclidean, correlation, cityblock };
```

this enum class holds all the possible distance metrics the user can choose from; it prevents unnecessary typos that could occur with the string names.

`class vectors` - class for storing matrices and possible operations on them.

public members:

`int n_samples = 0;` number of rows of the matrix, automatically set to 0

`int n_features = 0;` number of columns of the matrix, automatically set to 0

`double* coords;` pointer to array of doubles, dynamically allocated later on

`vectors() = default;` default constructor, creates NULL pointer, usage unadvised

`vectors(int samples, int features, double* data)` - constructor creating the matrix of given dimensions: samples - number of rows, features - number of columns, data - a deep copy of a given array is created. The memory used by this parameter should be freed later on, this constructor doesn't free it.

`vectors(int samples, int features)` - constructor allocating the memory of the given size (samples*features) and setting the dimensions.

`vectors(int square_dim)` - convenience constructor for square matrices

`vectors(const vectors& v1)` - copy constructor. Creates a deep copy of a given object - doesn't only copy pointer coords but creates a new one and copies the values in the array.

`vectors(std::ifstream& datafile)` - constructor loading the matrix from the stream datafile. Number of rows is the same as the number of lines in the file, the number of columns is the same as the number of entrances in each line. It does not check if the file is correct - errors may occur.

`~vectors()` - object destructor, frees the memory.

`vectors& operator=(const vectors& v)` - assigns the number of rows and columns to these numbers in v. Reallocates the coords, creates a deep copy of v.coords.

`void shape_show()` - prints the matrix dimensions to the console: (n_samples, n_features).

`double sum()` - returns the sum of all elements of the array.

`double sum_of_column(int column)` - returns the sum of elements in the column of the given index (int column).

`double sum_of_row(int row) const` - returns the sum of elements in the row of the given index (int row)

`double mean_of_row(int row) const` - returns the mean of elements in the row of the given index (int row)

`double mean_of_column(int column)` - returns the mean of elements in the column of the given index (int column)

`void change_size(int samples, int features)` - changes n_samples to samples, n_features to features and reallocates memory.

`void substract(double value)` – substracts value from all the elements of the array.

`void substract(double value, int column)` – substracts value from all the elements of the column given by the index (int column).

`void substract(double* v, int col)` – substracts given vector from the column specified by given index (int col).

`double* substract(double value, int row) const` - returns vector specified by the row index decreased by the given value.

`void divide(double value, int column)` – divides all the elements of the column specified by index (int column) by value.

`void divide(double value)`– divides all the elements of the matrix by value.

`vectors transpose()` – returns the transpose of the matrix.

`void leave_n_cols(int col)` - method preserving col columns out of matrix.

private:

`void get_dimensions(std::ifstream& datafile)` – counts number of lines and columns in a given file and sets the dimensions of the object accordingly.

`void load_data(std::ifstream& datafile)` – loads the data from the given file into the object. The dimensions of the vectors should be specified before the usage.

this header contains also functions operating on the vectors class:

`vectors indices(int l)` - creates the matrix of `l` n_features and and 1 n_samples, returning it filled with the appropriate indices.

`double length_of_column(const vectors& v1, int column)` – returns the length of column under the index (int column) of object v1.

`double length_of_row(const vectors& v1, int row)` – returns the length of row under the index (int row) of object v1.

`double length_of_row(double* v, int dimension)` – this function calculates length of given array on specified number of dimensions. Dimension shouldn't be bigger than the number of allocated doubles.

`void normalise(vectors* v, int col)` - normalises the column under given index (col) to the unit vector.

`void normalise(vectors* v)` - normalises the whole matrix.

`vectors center(const vectors& v1)` - returns centered matrix.

`vectors standardise(const vectors& v1)` - returns standardised matrix.

`vectors std_base(int dimension)` - returns matrix being standard basis for dimension-space.

`bool operator==(const vectors& v1, const vectors& v2)` - compares n_dimensions, n_features and variables in v1 and v2.

`void operator<<(std::ostream& out, const vectors& some_vector)` - writes into out all the elements in the array, preserving its specified dimensions.

`double* operator*(const vectors& A, double* x)` - returns the result of multiplication A*x. x should not be shorter than A.n_features. If longer, additional dimensions are skipped.

`vectors operator*(const vectors& A, const vectors& B)` - returns result of matrix multiplication A*B. A.n_features should be equal to B.n_samples.

`double row_product(double* d1, double* d2, int dimension)` - returns result of dot product of d1 and d2. Their lengths should be equal to dimension.

`double col_product(vectors x, int col1, int col2)` - returns result of dot product of columns under indexes col1 and col2.

`double correlation_distance(const vectors& v1, const vectors& v2, int row1, int row2)` - returns correlation distance between rows of two matrices: row1 in v1 and row2 in v2.

`double Euclidean_distance(const vectors& v1, const vectors& v2, int row1, int row2)` - returns Euclidean distance between rows of two matrices: row1 in v1 and row2 in v2.

`double cityblock_distance(const vectors& v1, const vectors& v2, int row1, int row2)` - returns cityblock distance between rows of two matrices: row1 in v1 and row2 in v2.

`double distance(const vectors& v1, const vectors& v2, int row1, int row2, dist_metric = dist_::Euclidean)` - returns distance in dist_ metric between rows of two matrices: row1 in v1 and row2 in v2. The default is Euclidean distance.

`double min_distance(const vectors& v1, const vectors& v2, int row1, dist_ metric = dist_::Euclidean)` - returns distance in dist_ metric between row of two matrices: row1 in v1 and closest row to row1 of v2.

`double min_distance(const vectors& v1, const vectors& v2, int row1, int index, dist_ metric = dist_::Euclidean)` - returns distance in dist_ metric between row of two matrices: row1 in v1 and closest row to v1 from rows in v2 up to index, exclusively.

`int argmin_distance(const vectors& v1, const vectors& v2, int row1, dist_ metric = dist_::Euclidean)` - returns index of row in v2 closest to row1 in v1 in dist_ metric.

`double abs(double a, double b)`- returns absolute difference between a and b.

`bool are_same(const vectors& v1, const vectors& v2, double tol = 1e-4)` - checks if the matrices are the same with some tolerance tol.

`void orthogonalise(vectors* x, int col1, int col2)` - orthogonalises column col1 of x with respect to column col2 of x.

`vectors Gram_Schmidt(vectors x)` - returns orthomalized x, assuming consecutive vectors are in columns, not rows.

2. In the file initialization.h:

`enum class init_ {random, kpp};` - this enum class holds all the possible initialization methods the user can choose from; it prevents unnecessary typos that could occur with the string names.

`void swap(double* a, double* b)` - swaps a and b.

`void fisher_yates(int desired_dimensions, vectors* ind)` - randomly swaps values in ind.coords up to desired_dimensions index (exclusively) with all others values in ind.coords.

`void random_init(vectors* centres, vectors x)` - chooses centres->n_samples samples from x at random and fills centres->coords with them.

`void first_centroid(vectors* centres, vectors x)` - chooses 1 sample at random from x as the first centroid for k++ initialization and fills the first row of centres with the values.

`int weighted_random(vectors weights)` - returns index of one of the points by probability based on the given weights.coords.

`void next_centroid(vectors* centres, vectors x, vectors* weights, int c_index, dist_ metric)` - chooses next centroid for k++ initialization from x based on metric and appends it in c_index row of centres.

`void kpp_init(vectors* centres, vectors x, dist_ metric)` - chooses centres->n_samples samples from x by k++ method and fills centres->coords with them.

`void initialize(vectors* centres, vectors x, init_ init, dist_ metric)` - chooses centres->n_samples samples from x by init and fills centres->coords with them.

3. In the file kmeans.h:

`void label_points(vectors* labels, vectors x, vectors centroids, dist_ metric)` - fills labels->coords with indices of rows of centroids being closest to the consecutive samples in x measured with metric.

`void calculate_centroids(vectors labels, vectors x, vectors* centroids)` - calculates centroids->coords as average on each axis of samples in x with the same labels.coords under the appropriate indices.

`double calculate_inertia(vectors labels, vectors x, vectors centroids, dist_ metric)` - returns inertia for given parameters.

`void kmeans_algorithm(vectors* centroids, vectors* labels, dist_ metric, int n_clusters, init_ initialization, double* inertia, int max_iter, int* n_iter, vectors x)` - performs k-means clustering on x for n_clusters. Fills centroids and labels matrices and counts n_iter for this run. Calculates inertia of the model.

`class kmeans` - k - means estimator.

public members:

`vectors centroids` - Coordinates of cluster centers and their dimensions.

`dist_ metric` - one of dist_: Euclidean, correlation or cityblock.

`int n_clusters` - the number of clusters to form as well as the number of centroids to generate, should be int greater than 1.

`init_ initialization` - one of init_: random or kpp.

`double inertia` - sum of squared distances of samples to their closest cluster center.

`int max_iter` - maximum number of iterations of the k-means algorithm for a single run.

`int n_iter` - number of iterations run.

`vectors labels` - labels of each point.

`int n_init` - Number of times the k-means algorithm will be run. The final results will be the best output of n_init consecutive runs in terms of inertia.

`kmeans(int clusters_n, dist_ metrics = dist_::correlation, init_ init = init_::kpp, int iter = 1000, int init_n = 10)` - constructor setting all the parameters for the future clustering.

`kmeans() = default` - default constructor, standalone usage not recommended.

`~kmeans()` - destructor.

`kmeans(const kmeans& estim)` - copy constructor; creates deep copy of all parameters, labels and centroids too.

`kmeans(const kmeans& estim, int clusters_n)` - copy constructor; creates copy of all parameters except estim.labels, estim.centroids and estim.n_clusters. n_clusters is set to clusters_n. New estimator should be fitted independently later on.

`kmeans& operator=(const kmeans& estim)` - deep assignment operator.

`void fit(vectors data)` - compute k-means clustering for data.

`vectors predict(vectors data)` - predict the closest cluster each sample in data belongs to.

`vectors fit_predict(vectors data)` - convenience method equivalent to calling fit and then predict on data.

4. In the file sorting.h:

`bool descending_sort(int i, int j)` - checks if i is bigger than j.

`class sort_indices` - convenience class for sorting functions.

public members:

`vectors b` - vectors which coords is sorted

`sort_indices(vectors to_sort)` - constructor

`bool operator()(int i, int j) const` - checks if b.coords[i] is bigger than b.coords[j].

`void sort_by_idx(vectors* x, vectors idx)` - sorts *x by the order specified in idx.

`void sort_two(vectors* basic, vectors* dependent)` - sorts *basic in descending order and the columns in *dependent accordingly.

5. In the file pca.h:

`vectors covariance_matrix(vectors data)` - returns approximate covariance matrix of data.

`void QR_algorithm(vectors x, vectors* eigenvec, vectors* eigenvals, double tol, int iter)` - finds eigenvectors and eigenvalues of x approximating to tol, iterating at most iter times. Uses QR algorithm combined with orthogonal power method.

`class PCA` - performs Principal Component Analysis. Using standardised vectors is recommended for performance reasons.

public members:

`int reduced_dims` - number of features data should have left after the transformation

`std::string reduce` - if not "YES", keeps all features

`vectors eigenvectors` - reduced eigenvectors of the covariance matrix of data used to fit

`vectors eigenvalues` - reduced eigenvalues of the covariance matrix of data used to fit

`double tol` - tolerance for singular values of coords of eigenvectors

`int n_iter` - number of iterations for the QR method

`PCA(std::string red = "YES", int dims = 2, double tolerance = 1e-6)` - constructor

`~PCA()` - destructor

`PCA(const PCA& t)` - deep copy constructor

`void fit(vectors data)` - fit model with data

`vectors transform(vectors data)` - apply dimensionality reduction to data

`vectors fit_transform(vectors data)` - convenience method, equivalent to fitting model with data and applying dimensionality reduction on it.

6. In the file dunn.h:

`enum class inter_ {centroid, closest, furthest, avg}` - this enum class holds all the possible distances between clusters the user can choose from; it prevents unnecessary typos that could occur with the string names.

`inter_::centroid` - distance between centroids of the clusters

`inter_::closest` - distance between closest samples of two distinct clusters

`inter_::furthest` - distance between furthest samples of two distinct clusters

`inter_::avg` - average distance between all of the samples of two distinct clusters

`enum class inter_ {centroid, closest, furthest, avg}` - this enum class holds all the possible distances within the cluster the user can choose from; it prevents unnecessary typos that could occur with the string names.

`intra_::centroid` - average distance from all the samples in the cluster to its centroid

`intra_::furthest` - distance between furthest samples of the cluster

`intra_::avg` - average distance between all of the samples within cluster

`double single_linkage(vectors labels, vectors data, int c1, int c2, dist_ metric)` - returns closest distance in metric distance between two samples of data belonging to clusters c1 and c2 respectively.

`double complete_linkage(vectors labels, vectors data, int c1, int c2, dist_ metric)` - returns the distance in metric distance between the most remote samples of data belonging to clusters c1 and c2 respectively.

`double avg_linkage(vectors labels, vectors data, int c1, int c2, dist_ metric)` - returns the average distance in metric distance between all of the samples of data belonging to clusters c1 and c2 respectively.

`double inter_linkage(vectors labels, vectors data, int c1, int c2, dist_ metric, inter_ link)` - returns the distance between clusters c1 and c2 in metric, defined as link.

`double inter_dist(kmeans* est, vectors data, inter_ metric)` - returns smallest distance between two clusters of *est, defined as one of: inter_::closest, inter::_furthest, inter_::avg.

`double inter_centroid(vectors centroids, dist_ metric)` - returns smallest distance between centroids.

`double inter_distance(kmeans *estim, vectors data, inter_ metric = inter_::centroid)` - returns the smallest distance between two clusters of all clusters, defined as metric.

`double intra_centroid(kmeans* est, vectors data, int c)` - returns the average distance of samples in data belonging to cluster c from centroid c.

`double intra_linkage(kmeans* estim, vectors data, int c, intra_ metric)` - returns distance within cluster c defined as metric.

`double intra_distance(kmeans* est, vectors data, intra_ metric= intra_::avg)` - returns the biggest distance within a cluster of all clusters, defined as metric.

`double dunn_index(kmeans *estim, vectors data, inter_ metric1 = inter_::centroid, intra_ metric2 = intra_::avg)` - returns Dunn Index for estim and the data in specified linkage metrics.

`class DunnSearch` - chooses the best estimator and its number of clusters by computing Dunn Index for specified range.

public members:

`kmeans estimator` - after fitting, estimator of the greatest Dunn Index from min_clusters to max_clusters interval, before: template estimator given by user.

`intra_ intra` - measurement method of distance within clusters.

`inter_ inter` - measurement method of distance between clusters.

`int max_clusters` - biggest number of clusters for estimator.

`int min_clusters` - smallest number of clusters for estimator.

`double index` - Dunn Index value for estimator after fitting.

`DunnSearch(kmeans est, inter_ inter_d = inter_::centroid, intra_ intra_d = intra_::avg, int min = 2, int max = 20)` - constructor; creates a deep copy of est.

`~DunnSearch()` - destructor.

`DunnSearch(const DunnSearch& estim)` - deep copy constructor.

`double single_idx(kmeans* est, vectors data, int clusters_n)` - returns a single Dunn Index for *est, automatically fits *est and changes its number of clusters.

`void fit(vectors data)` - performs k-means clustering for kmeans estimators with n_cluster between max_clusters and min_clusters, chooses the estimator with the highest Dunn Index.

7. In the file silhouette.h:

`double avg_to_cluster(vectors labels, vectors data, int sample, int c, dist_metric)` - calculates the average distance of the sample in data to samples in cluster c in metric.

`double min_avg(vectors labels, vectors data, int sample, int n_clusters, dist_metric)` - calculates the distance of the sample in data to the closest cluster it doesn't belong to, in metric distance.

`double _for_sample(kmeans* est, vectors data, int sample)` - returns Silhouette Score for sample in data, *est should be fitted.

`double silhouette(kmeans* est, vectors data)` - returns average Silhouette Score for data after clustering.

`class SilhouetteSearch` - chooses the best estimator and its number of clusters by computing Silhouette Score for specified range.

public members:

`kmeans estimator` - after fitting, estimator of the greatest Silhouette Score from min_clusters. to max_clusters interval, before: template estimator given by user.

`int max_clusters` - biggest number of clusters for estimator.

`int min_clusters` - smallest number of clusters for estimator.

`double coefficient` - Silhouette Score value for estimator

`SilhouetteSearch(kmeans est, int min = 2, int max = 20)` - constructor; creates a deep copy of est.

`~SilhouetteSearch()` - destructor.

`SilhouetteSearch(const SilhouetteSearch& estim)` - deep copy constructor.

`double single_coefficient(kmeans* est, vectors data, int clusters_n)` - calculates a single Silhouette Score for *est, automatically fits *est and changes its number of clusters.

`void fit(vectors data)` - performs k-means clustering for kmeans estimators with n_cluster between max_clusters and min_clusters, chooses the estimator with the highest Silhouette Score.

5. Source code

I decided against putting the source code in this file for it would definitely lower its quality. All of the files are available in the already linked repository. It is organized in 8 source files, excluding tests:

- Vectors.h - provides class and functions for operations on matrices.
- initialization.h - entails functions choosing some part of given vectors (random or k++ initialization).
- kmeans.h - runs k-means algorithm.
- dunn.h - runs k-means algorithm with sweep by Dunn index
- silhouette.h - runs k-means algorithm with sweep by Silhouette Score.
- sorting.h - sorts two vectors in descending order of one of them.
- pca.h - performs dimension reduction by Principal Component Analysis.
- Example.cpp - contains main() function and shows exemplary flow of the program.

6. Testing

I decided to implement unit tests for this project so that debugging and manual tests will not be necessary so often. I used Native Unit Tests provided by Visual Studio. All of them can be seen here. I will not put them here as it would obscur the whole report. I also performed a separate test on PCA, as I wanted to check its correctness on more complex data than small matrices. I checked whether the transformation of the iris dataset in this implementation and solution provided in the scikit-learn package gives the same result. My program proved to be successful. The same applies to DunnSearch (I checked this one with my own implementation in Python as well as some verified codes available on GitHub, as there is none in scikit-learn), and SilhouetteSearch - the values of the calculated numbers of clusters were the same.

7. Additional remarks

To improve the stability and performance of the k-means algorithm, each training of the model is repeated 10 times - after that, I choose the version with the smallest inertia. This also made the choice of number of clusters in Search classes more stable - this value stopped jumping through the whole desired interval, settling for one characteristic value for the dataset.