Machine learning lab assignment 3
k-nearest neighbors classifier

Kevin Serrano, *EMARO+, UNIGE*

***Abstract*—This report introduces the k-NN algorithm and its performance is tested using the semeion-digits dataset. It is implemented using MATLAB and some of its hyper-parameters such as distance and weighting functions are discussed. Additionally, confusion matrix terminology is introduced to better asses the performance of classifiers taking into account multiple rates rather than just accuracy. Finally, grid search is used to distinguish differences on performance when training k-NN with several hyper-parameter combinations.**

***Index Terms*—k-nearest neighbors, classifier, machine learning, confusion matrix, grid search.**

I. Introduction

A. k-NN algorithm

The **k-nearest neighbors algorithm** (k-NN) is a non-parametric method used for regression and classification.

Throughout this report we'll be using the k-NN algorithm to solve a classification problem. In k-NN classification, the output is a class membership. An observation will be assigned to the class most common among its k nearest neighbors, with k being an integer, in other words, an observation is classified by a majority vote of its neighbors.

As most classification algorithms, it has two main steps (training and prediction). The training step only consists in locally storing all the available observations, leaving all the computation to be carried out in the prediction step. This is often referred to as instance-based learning or lazy learning. The k-NN algorithm is among the simplest algorithms of machine learning.

A useful technique to improve performance can be to assign weight to each neighbor's contribution, so that nearer neighbors contribute more than distant ones.

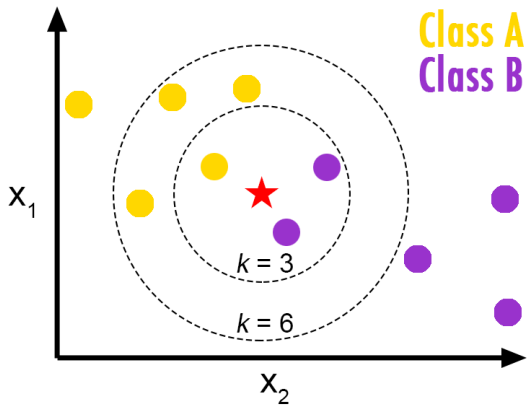

Fig. 1: 2D algorithm visualization example

	predicted YES	predicted NO
actual YES	TP	FN
actual NO	FP	TF

Fig. 2: Binary confusion matrix. TP = true positive, TN = true negative, FN = false negative, FP = false positive.

B. Confusion matrix terminology

A confusion matrix table is a table often used to describe the performance of a classification model on a set of data for which true values are known. The confusion matrix itself is relatively simple to understand, but the related terminology might be confusing [1].

The basic terms are:

- True positives (TP): actual and predicted outcome is positive.
- True negatives (TN): actual and predicted outcome is negative.
- False positives (FP): prediction is positive, but actual value is negative. Also known as *type I error*
- False negatives (FN): prediction is negative, but actual value is positive. Also known as *type II error*

Using the 4 previous terms, we can compute several rates that helps us evaluate the performance of a classifier.

- Accuracy: overall, how often is the classifier correct.

$$\frac{TP+TN}{\text{total}}$$

- Misclassification rate: overall, how often is the classifier wrong.

$$\frac{FP+FN}{\text{total}} = 1 - \text{accuracy}$$

- **Sensitivity** or **recall**: Also known as true positive rate. When it is actually YES, how often does it predict YES.

$$\frac{TP}{TP+FN}$$

- **Specificity**: Also known as true negative rate. When it is actually NO, how often does it predict NO.

$$\frac{TN}{TN+FP}$$

- **Precision**: When it predicts YES, how often it is correct.

$$\frac{TP}{TP+FP}$$

- **F1 score**: Harmonic (weighted) average of the precision and recall where F1 reaches its best value at 1 and worst at 0.

$$2\,\frac{\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}$$

Although a binary confusion matrix was used to introduce the previous concepts, it can be generalized to multi-class scenarios since every confusion matrix can be decomposed into multiple binary matrices using a *one-vs-all* approach. Thus, for multi-class problems we'll compute rates per class.

II. Approach

A. Training step (fitting model)

As previously mentioned, k-NN is a lazy learning algorithm and there is no computation in this step. Nevertheless, we can detail here the different distance functions which will help us determine the k closest neighbors.

- Euclidean distance: *ordinary* straight-line distance between two points in Euclidean space.

$$d(\boldsymbol{x},\boldsymbol{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{1}$$

- Manhattan (city block) distance: absolute difference of two points' Cartesian coordinates. Multiple shortest paths can be obtained with this metric, whereas the euclidean one has a *unique* shortest path. It can be interpreted as the shortest path(s) a car would take between two intersections.

$$d(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{n}|x_i - y_i| \tag{2}$$

- Minkowksi distance: can be considered a generalization of both, Euclidean and Manhattan distance.

$$d(\boldsymbol{x},\boldsymbol{y}) = \left(\sum_{i=1}^{n}|x_i - y_i|^q\right)^{1/q} \tag{3}$$

Additionally, we introduce three different distance weighting functions.

- `equal`: All neighbors get equal weight of 1.
- `invdist`: Weight is 1/distance of each corresponding neighbor.
- `rank`: Weight is 1/rank where rank corresponds to the ordinal value of each neighbor.

TABLE I: Example list of nearest neighbors.

rank	distance	**class**
1	0.3	blue
2	0.4	red
3	0.5	green
4	0.6	green
5	1.5	red
6	2.9	red

We'll provide the possible different outcomes when using each of the previously presented weighting functions by taking as reference the premeditated example in table I. The weighted sums, using each of the different weighting functions, are shown in table II and it is easy to see how each function affects the predicted class $\hat{y}$.

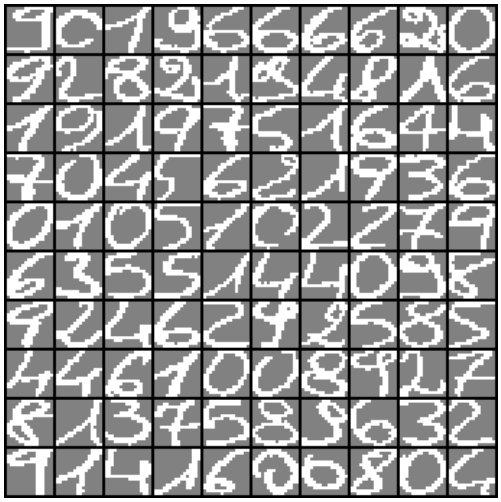

Fig. 3: Randomly picked samples of semeion-digits dataset.

TABLE II: Prediction ($\hat{y}$) using different weighting functions.

	blue	red	green	$\hat{y}$
`equal`	1	**3**	2	red
`invdist`	3.33	3.51	**3.66**	green
`rank`	**1**	0.86	0.58	blue

B. Testing step (prediction)

Prediction step can be summarized as:

1) Computing distance between an *unseen* pattern/observation and all the other elements in the training set.
2) Extract the k-nearest neighbors. Choose the k shortest distances along with the respective class which they belong.
3) Classify observation according to the class most common among the (weighted) neighbors.

We can already appreciate the simplicity of this algorithm since it only consists of mainly three steps. However, this simplicity comes with a high computational cost. Taking into account an $m \times n$ training set, the algorithm complexity (per *unseen* observation) is approximately $O(mnk)$ depending on the method used to extract the k nearest neighbors.

III. Experimental results & analysis

The semeion-digits dataset will be used throughout this report. It consists of 1593 scanned handwritten digits from around 80 persons, stretched in a rectangular 16×16 box in a binary scale. These binary images are then *flattened* into 256 dimensional vectors, yielding a dataset with 1593 observations and 256 features.

The k-NN algorithm was implemented in MATLAB as a class. The following snippet exemplifies its use.

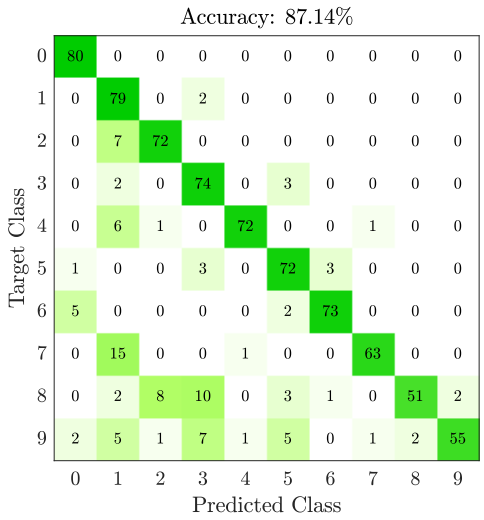

Fig. 4: k-NN confusion matrix (k = 10).

```
% 50% for training - 50% for testing
[train, test] = stratified_split(X, t, 0.50);

% create classifier object
k = 10;
kNN = kNNKlassifier(k);

% set hyper-parameters
kNN.distfcn = 'Euclidean';
% (by default) also possible: 'Manhattan'
kNN.weightfcn = 'equal';
% (by default) also possible: 'invdist' or 'rank'

% train classifier
kNN.learn(train.X, train.t);

% get predicted output and compute its accuracy
y_pred = kNN.predict(test.X);
acc = mean( test.t == y_pred ); % accuracy
```

The results obtained after executing the previous script can be better visualized with the confusion matrix in figure 4. Since there are 10 classes (one per digit) the dimension of the matrix will be 10×10 whose rows indicate the true label/class, and columns the prediction.

The percentage of samples per class is roughly equal throughout all the semeion dataset, that is, each class takes up around 10% of the whole dataset. Since we're using stratified split, these percentages are preserved. Sorting out all the evident computations, the test set contains around 80 samples per class. Knowing this can help us get a better grasp of the numbers contained in the confusion matrix in figure 4. Moreover, we can see that the most *confused* classes are $1 \leftarrow 7$ and $3 \leftarrow 8$.

Going back to the terminology presented in the introduction, we can say that the diagonal elements are the equivalent of true positives/negatives depending on the class to be analyzed. Additionally, rows represent false negatives and columns represent false positives (omitting diagonal values). Accuracy and misclassification rates are class-independent since they only rely on right or wrong predictions; the remaining class-dependent rates are displayed in table III.

TABLE III: k-NN class-dependent rates

k-nn performance

class (digit)	specificty	sensitivity	precision	F1
0	98.708	100.000	90.909	95.238
1	94.299	97.531	68.103	80.203
2	98.410	91.139	87.805	89.441
3	96.557	93.671	77.083	84.571
4	99.678	90.000	97.297	93.506
5	97.943	91.139	84.706	87.805
6	99.357	91.250	94.805	92.994
7	99.683	79.747	96.923	87.500
8	99.688	66.234	96.226	78.462
9	99.687	69.620	96.491	80.882
avg.	98.401	87.033	89.035	87.060

metric

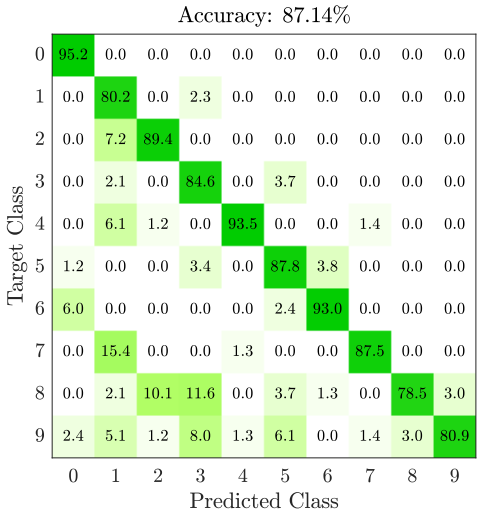

Fig. 5: k-NN confusion matrix with F1 scores (k = 10).

To get a better understanding on the relation between performance rates and the confusion matrix we'll take digit 8 as an example. From table III we notice it has low sensitivity (large number of false negatives, dense row-vector) and high precision (small number of false positives, sparse column-vector). In less technical terms, around 30% of actual 8 samples get misclassified, but when we predict 8 we can be pretty confident the sample is an actual 8. The F1 score conveys a balance between these two concepts and we will therefore mainly use this score to asses the classifier's performance.

Now, both the confusion matrix and the class-dependent rates provide useful complementary information. The confusion matrix is visually attractive, but numbers in it are hard to understand when not given any context on how it was computed. On the other hand, rates are numerically easy to interpret since they range from 1 to 0 (or 100 to 0) where 1 is the best performance and 0 is worst. Combining them while still retaining the best of both worlds is possible; the *F1-normalized* confusion matrix in figure 5 is both visually and numerically intuitive.

A. How to choose k?

This is one of the main questions to answer when using k-NN. There are multiple approaches to answer this question which, in machine learning, are often referred to as hyper-parameter optimization [2]. Along with k, we have other factors in play such as the distance and weighting functions. Therefore, we'll perform several a *grid search* or *parameter sweep* through a manually specified subset of hyper-parameters.

$$K \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20\}$$
$$\texttt{distfcn} \in \{\texttt{Euclidean}, \texttt{Manhattan}\}$$
$$\texttt{weightfcn} \in \{\texttt{equal}, \texttt{invdist}, \texttt{rank}\}$$

Regarding the `distfcn` hyper-parameter, the distance measured by either of the functions is exactly the same since all our features are binary, thus we can discard it from the grid search.

Algorithm 1 General structure for grid search using a k-NN classifier.

Ensure: hyper-parameter subsets initialized
  procedure GRIDSEARCH(`weightfcn`, K, `iters`)
    for all $w \in$ `weightfcn` **do**
      for all $k \in K$ **do**
        for $i = 1$; $i \leq$ `iters`; i++ **do**
          Random train-test split
          Train kNNClassifier(k, w)
          Test kNNClassifier(k, w)
          Get and store performance metrics
        end for
      end for
    end for
  end procedure

Binarizing the confusion matrix induces class imbalance, leadint to a positive-negative ratio of 1:9 approximately. Hence, we can avoid displaying true negative rates since they won't greatly contribute to the performance analysis. Figure 6 shows the performance average rates obtained from grid search. It is interesting to note that the shape is mostly preserved among all different metrics due to the stratified splitting. Also, even though the F1 score is considered a balance between precision and recall, this property doesn't hold when averaging though all classes (can also be noted from table III).

Given that shape is preserved, we could use any of the presented metrics to choose an appropriate value for k. On that account, we'll use accuracy as performance metric. Figure 7 shows the averaged results obtained from the grid search. When using `equal` weighting we notice a drop when $k = 2$ because of *tie handling*. As a tie-breaker, we choose our prediction based on the ascending order in which tied classes appear. The other two methods are more immune to ties since distances and ranks among neighbors are unlikely to be the same. Another interesting behavior to be noticed is that rank-weighting has a relatively constant performance as the number

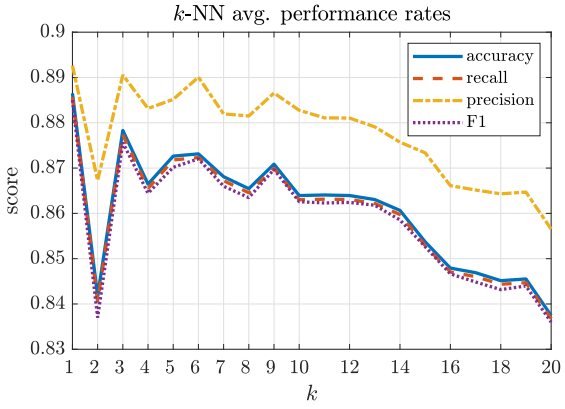

Fig. 6: Grid search performance using `weightfcn = 'equal'`. Overall, predictions contain slightly less false positives than false negatives.

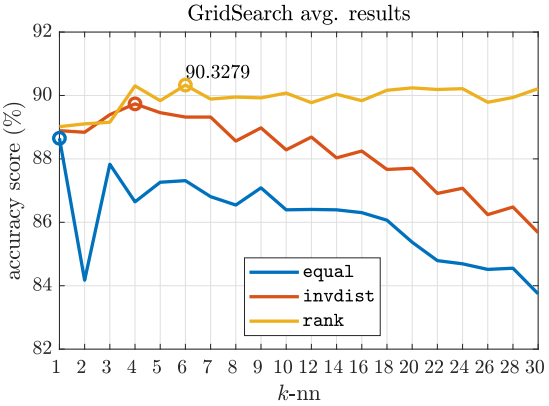

Fig. 7: Grid search averaged results. Circles indicate the highest score achieved by the corresponding `weighfcn`.

of neighbors increases, whereas with tho other two methods performance seems to decrease as k increases.

The [6, `rank`] hyper-parameter pair yielded the highest average accuracy. Seeing that accuracy performance remains high and constant when using rank-weighting we can say it is a *safe* choice because it downplays the role of k when $k \geq 5$.

Refer to the appendix for a more graphical intuition on how the three different weighting functions work. In the appendix we made use of the t-SNE algorithm to represent the semeion-dataset in a 2-dimensional space and ran our k-nn implementation over this reduced dataset.

IV. CONCLUSIONS

Despite the simplicity of the k-nn classification algorithm, results can give useful insight on the dataset itself. It classifies unseen data based on a similarity measure (e.g. distance functions) which gives us a couple of benefits:

- Predictions are easy to interpret
- Distribution-free method. Meaning that it doesn't assume that data follows a specific distribution.
- Few hyper-parameters to tweak (e.g. k, weighting functions and distance functions)
- No training, but computationally expensive at prediction when having large datasets.

To get the most out of this method it is helpful to perform a grid search to discover which combination of hyper-parameters give the *best* results. Although there are many other methods that nowadays outperform k-NN classifiers, they have been used since the 70s and are still integrated in ensemble methods to further boost classification performance.

APPENDIX

Before going further in this section, we must clarify that the information and figures presented here are merely auxiliary. The dimensionality reduction t-SNE technique was used since it is particularly well suited for the visualization of high-dimensional datasets.

Dimensionality reduction techniques are out of the scope of this report but we'll provide a brief intuition on how t-SNE works. t-SNE attempts to solve the crowding problem. It models each high-dimensional object by a two or three-dimensional point such that similar objects are modeled by nearby points and dissimilar ones by distant points.

Results shown in further figures were using the actual embedded data and in no way intend to replicate or accurately explain the behavior obtained when using the actual high-dimensional dataset but rather provide a good approximation.

All 1593 observations in the semeion-dataset were embedded using t-SNE. Figures 8 and 9 are rather self-explanatory and interesting differences can be appreciated depending on the weighting function used, particularly in the northwest region of the plots.

REFERENCES

[1] K. Markham, "Simple guide to confusion matrix terminology," http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/, 2014, accessed 01/11/17.

[2] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning," *CoRR*, vol. abs/1502.02127, 2015. [Online]. Available: http://arxiv.org/abs/1502.02127

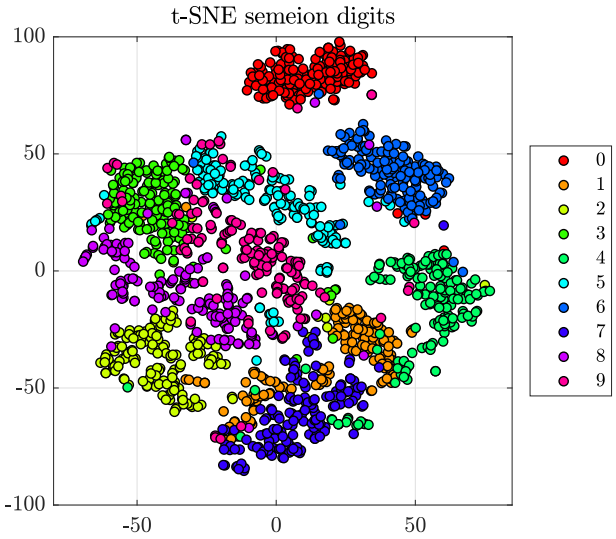

Fig. 8: Raw t-SNE embedding of semeion-data using euclidean distance

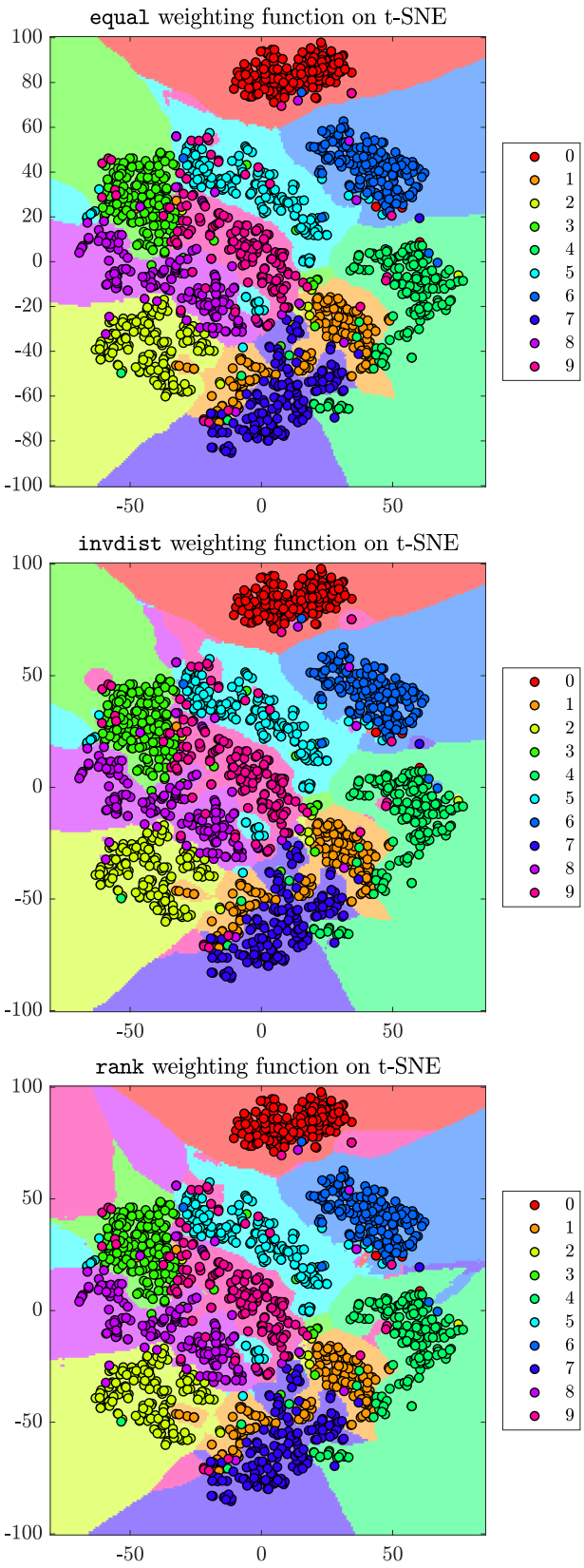

Fig. 9: Approximate Voronoi tesselations using 10-NN classification. Each new point will be classified according to the color of the region it lies in.