INF582: Data Science – Learning from Data

École Polytechnique

Lab 3: Supervised Learning I

k-Nearest Neighbors Classification and Perceptron Algorithm

Jean-Baptiste Bordes, Fragkiskos Malliaros, Nikolaos Tziortziotis and Michalis Vazirgiannis

December 18, 2015

1 Description

The goal of this lab is to study the following linear classification algorithms: (i) *k-Nearest Neighbors algorithm* and (ii) *Perceptron algorithm*. Initially, we discuss the basic characteristics of each algorithm and then we examine how the algorithms can be applied on the handwritten digit classification problem.

2 k-Nearest Neighbors Classification Algorithm

The *k-Nearest Neighbors algorithm* (k-NN) is a simple, very intuitive and commonly used method in the classification task. Recall that, in classification, the goal is to identify the class of a new instance (observation) based on a training dataset in which the class label of each instance is known.

In the k-NN algorithm, the predicted class label of a new instance is based on the already known classes of the k most similar neighbors. That way, the class of a new instance is specified by the majority rule between the classes of the k nearest neighbors. Variable k is the parameter of the algorithm and it can take positive integer values. In the extreme case where $k = 1$, the object is simply assigned to the class of that single nearest neighbor. In the case where $k > 1$ is an even number and the majority rule does not hold (i.e., equal number of neighbors from each class), the label is selected randomly.

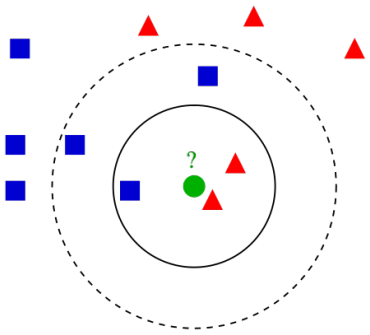

Figure 1: Example of k-NN classification. The test sample (green circle) can be classified either to the first class of blue squares or to the second class of red triangles. If $k = 3$ (solid line circle) it is assigned to the second class because there are 2 triangles and only 1 square inside the inner circle. If $k = 5$ (dashed line circle) it is assigned to the first class. (Source: Wikipedia).

Note that, neighbors-based classification is a type of instance-based learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

The training examples used be the k-NN algorithm are vectors in a multidimensional feature space, each one associated with a class label. Let $\mathcal{X} = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_m}, y_m)\}$, where $\mathbf{x_i} = (x_1, x_2, \ldots, x_n)$,

$i = 1, \ldots, m$ and y_i the class label, be the $m \times n$ training dataset. The *training phase* of the algorithm consists only of storing the feature vectors and class labels of the training samples.

In the classification phase, k is a user-defined constant, and an unlabeled instance $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is classified by assigning the label which is most frequent among the k training samples nearest to that query point. In order to find the nearest neighbors of the new instance, a *similarity* (or distance) measure between the instance and the training examples should be defined. Typically, the choice of a similarity measure depends on the type of the features in the data. In the case of real-valued features (i.e., $x_i \in \mathbb{R}, i = 1, \ldots, n$), the *Euclidean distance* is the most commonly used measure:

$$d(\mathbf{x_i}, \mathbf{x_j}) = \sqrt{\sum_{f=1}^{m} (x_{im} - x_{jm})^2}.$$

In the case of discrete variables, such as for text classification, the *Hamming distance*[1] can be used. Another measures for the similarity between instances include the correlation coefficient (e.g., *Pearson* correlation coefficient).

Algorithm 1 provides the pseudocode of the k-NN classifier.

Algorithm 1 k-Nearest Neighbors Classification

Input: Training data $\mathcal{X} = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_m}, y_m)\}$, where $\mathbf{x_i} = (x_1, x_2, \ldots, x_n)$, $i = 1, \ldots, m$
New unlabeled instance $\mathbf{x}$
Parameter k
Output: Class label y of $\mathbf{x}$

1: Compute the distance of the test instance x from each training instance
2: Sort the distances in ascending (or descending) order
3: Use the sorted distances to select the k nearest neighbors of $\mathbf{x}$
4: Assign $\mathbf{x}$ to a class based on the majority rule of the k nearest neighbors

Note that, while computing the Euclidean distance between instance vectors, the features should be on the same scale. Although this is part of the preprocessing task, we stress out that if the data is not normalized, the performance of the k-NN classifier can be heavily affected. One way to normalize the values of the features is by the applying the *min-max* normalization, where the value v of a numeric attribute x is transformed to v' in the range $[0, 1]$ by computing the $v' = (v - \min(x))/(\max(x) - \min(x))$, where $\min(x)$ and $\max(x)$ the minimum and maximum values of attribute x. Another way to normalize the data is by computing the *z-score* $z_v = (v - \mu_x)/\sigma_x$, where μ_x is the mean value of attribute x and σ_x the standard deviation.

Additional properties of k-NN

Although k-NN algorithm is very simple, it typically performs well in practice and is easily implementable. However, it has been observed that when the class distribution is skewed, the majority voting rule does not perform well. That is, instances of a more frequent class tend to dominate the prediction of the new instance, because they tend to be common among the k nearest neighbors due to their large number. One way to overcome this problem is to weight the classification, taking into account the distance from the test instance to each of its k nearest neighbors. The class of each of the k nearest neighbors is multiplied by a weight proportional to the inverse of the distance from that instance to the test instance. The algorithm is also sensitive to noisy features and may perform badly

[1] Wikipedia's lemma for *Hamming distance*: `http://en.wikipedia.org/wiki/Hamming_distance`.

in high dimensions (curse of dimensionality). In these cases, the performance of the algorithm can be improved applying feature selection or dimensionality reduction techniques. Additionally, the running time of the k-NN algorithm is high; for each test instance, we have to search through all training data to find the nearest neighbors. This point can be improved using appropriate data structures that support fast nearest neighbor search and make k-NN computationally tractable even for large data sets (these generally seek to reduce the number of distance evaluations actually performed).

Choise of parameter k

The value of parameter k often depends on the properties of the dataset. Generally, larger values of k reduce the effect of noise on the classification, but make boundaries between classes less distinct. On the other hand, small values of k create many small regions for each class and may lead to overfit. In practice, we can apply cross-validation in order to choose an appropriate value of k[2]. A rule of thumb in machine learning is to pick k near the square root of the size of the training set.

2.1 Handwritten Digit Recognition with k-NN

In this lab, we will implement and apply the k-NN classifier to recognize handwritten digits from the MNIST database[3].

2.1.1 Description of the Dataset

The MNIST dataset consists of handwritten digit images ($0-9$) and it is divided in $60,000$ examples for the training set and $10,000$ examples for testing. Figure 2 depicts the first 10 images of the dataset.

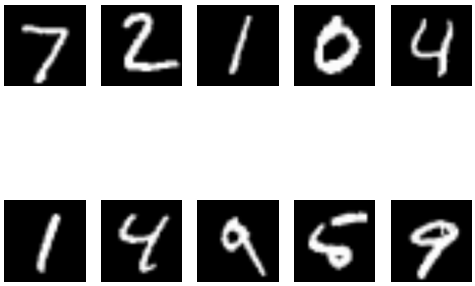

Figure 2: Example of 10 digits of the training set.

All digit images have been size-normalized and centered in a fixed size image of 28×28 pixels. Each pixel of the image is represented by a value in the range of $[0, 255]$, where 0 corresponds to black, 255 to white and anything in between is a different shade of grey. In our case, the pixels are the features of our dataset; therefore, each image (instance) has 784 features. That way, the training set has dimensions $60,000 \times 784$ and the test set $10,000 \times 784$. Regarding the class labels, each figure (digit) belongs to the category that this digit represents (e.g., digit 2 belongs to category 2). Due to time constraints, in the experiments that will be performed in the lab, we will use subsets of the above training and test sets. The code that imports the MNIST dataset has been implemented in the `loadMnist.py` Python script.

[2]A technique based on cross-validation for the selection of k is described here: `http://goo.gl/u3FXsg`.
[3]The MNIST database: `http://yann.lecun.com/exdb/mnist/`.

2.1.2 Pipeline of the Task

Here we describe the basic steps of the pipeline for the classification task, as given in the `kNN/main.py` Python script.

Initially, the data is loaded; variables `trainingImage` and `trainingLabels` contain the training instances and their class labels respectively. In a similar way, the test data and their class are loaded. Note that the `loadMnist()` function has already been implemented in the `loadMnist.py` script. The data is stored in the `Data` directory. Recall that each instance is a digit with $28 \times 28 = 784$ features (pixels).

```
# Load training and test data
trainingImages, trainingLabels = loadMnist('training')
testImages, testLabels = loadMnist('testing')
```

Since the dataset is relatively large, we keep a subset of the training and test data. This is happening due to time constraints of the lab and the fact that the k-NN algorithm is computationally expensive.

```
# Keep a subset of the training and test data
trainingImages = trainingImages[:2000,:]
trainingLabels = trainingLabels[:2000]

testImages = testImages[:50,:]
testLabels = testLabels[:50]
```

The next commands are for illustration purposes; they depict the first ten digits (images) of the test data.

```
# Show the first ten digits
fig = plt.figure('First 10 Digits')
for i in range(10):
a = fig.add_subplot(2,5,i+1)
plt.imshow(testImages[i,:].reshape(28,28), cmap=cm.gray)
plt.axis('off')

plt.show()
```

The next part of the code performs the classification of the test dataset using the k-NN algorithm. The `kNN()` function implements the k-Nearest Neighbors algorithm and the body of the function should be filled in the lab. It takes as input the parameter k (i.e., number of naighbors), the training data and their class labels, as well all the test data. In this case, we use the $k = 5$ nearest neighbors. As we have already discussed, the k-NN classifier is not based on a model that has been built upon the training data. The prediction of the class labels of new instances occurs during the classification phase based on the training set.

```
# Run kNN algorithm
k = 5
predictedDigits = zeros(testImages.shape[0])

for i in range(testImages.shape[0]):
print "Current Test Instance: " + str(i+1)
predictedDigits[i] = kNN(5, trainingImages, trainingLabels, testImages[i,:])
```

Finally, we compute the accuracy of the k-NN classifier. In particular, we compute the predicted labels of the test data with the true class labels contained in the `testLabels` variable.

```
# Calculate accuracy
successes = 0
```

```
for i in range(testImages.shape[0]):
if predictedDigits[i] == testLabels[i]:
successes += 1

accuracy = successes/float(testImages.shape[0])
print
print "Accuracy: " + str(accuracy)
```

2.1.3 Tasks to be Performed

- Fill the file `kNN/KNN.py` that implements the k-NN algorithm. As distance function, you can use the Euclidean distance.

```
def kNN(k, X, labels, y):
# k: number of nearest neighbors
# X: training data
# labels: class labels of training data
# y: predicted labels of test data

# Add your code here

return label
```

- Change the variable k and compute the accuracy of the results. What do you observe?
- Consider the size of the training set (recall that we have $60,000$ training instances) and examine the performance of the classifier for different cases. What do you observe? Is there any trade-off between the accuracy and the running time?

3 Classification Algorithm using Adaline

Adaline (Adaptive Linear Element) is a supervised classification algorithm designed to discriminate a feature vector in two classes. It was designed in 1960 by Bernard Widrow and his graduate student Ted Hoff at Stanford University [3]. It is linear classifier; this means that it is defined using a linear combination of the feature vector components with a set of weights. The geometrical interpretation of this process is an hyperplane separating the feature space in two parts.

3.1 Classification using hyperplanes

Let us denote the training dataset $\mathcal{X}$ as in previous section. Formally, an hyperplane defined by vector $w \in \mathbb{R}^n$ can be used to classify any vector $x \in \mathcal{X}$ as class 1 if $w^T x \geq 0$ and class 0 if $w^T x < 0$. If C stands for the class provided by this classifier and H the Heaviside function[4], then:

$$C(x) = H\Big(\sum_{i=1}^{n} w_i x_i\Big).$$

If there are p classes (with $p > 2$), the classification can be processed in *one against all*: for every class $j \in \{1, \ldots, p\}$, the training dataset is split in a subset of positive examples $\{(\mathbf{x_k}, y_k = j)\}$ and negative

[4]Wikipedias lemma for *Heaviside step function* `http://en.wikipedia.org/wiki/Heaviside_step_function`.

examples $\{(\mathbf{x_k}, y_k \neq j)\}$ and a weight vector w_j is trained. Then, the output of the system is defined by:

$$C(x) = \underset{j \in \{1,\ldots,p\}}{\arg\max} (w_j^T x).$$

3.2 Training of the Classifier using Widrow-Hoff Algorithm

We reason here in a two classes situation. The Widrow-Hoff algorithm is based on stochastic gradient descent to minimise the least square error $E(w) = \sum_{i=1}^{m} (C(x_i) - y_i)^2$. At every step, a single element $(\mathbf{x}, y)$ is chosen in $\mathbf{X}$ and the optimisation is performed on an approximated least square error: $e(w) = (w^T \mathbf{x} - y)^2$. The gradient $- \bigtriangledown e$ shows the steepest descent to minimize $e(w)$ and the corresponding translation is applied on w. Here is the pseudo-code:

- As long as the number of iterations is less than a fixed amount $\max_{iteration}$
  - Choose an element $(\mathbf{x}, y) \in \mathbf{X}$
  - update the weight vector: $w \leftarrow w - \epsilon \, (C(x) - y) \, x$

ϵ is a parameter of the algorithm.

3.3 Pipeline and Tasks to be Performed

The pipeline of this task is similar to the one of the k-NN classifier. The goal is to implement the training and classification parts of the Adaline algorithm described above. Then, the classifier will be applied on the MNIST dataset for the handwritten digit recognition problem. The pipeline is described in the `adaline/main2.py` script

After loading the data (similar to the k-NN task), we initialize the parameters of Adaline algorithm and we perform the training and classification tasks. Note that, we train the classifier for each class separately (ten classes in total):

```
w = numpy.random.rand(10,784)
eta = 0.0025
n = 5000
iteration_number = 40
results = zeros((iteration_number))

for k in range(iteration_number):
  for j in range(10):
        w[j,:] = training_adaline(w[j,:],n,trainingImages,trainingLabels,j,eta)

  for i in range(testImages.shape[0]):
    predictedDigits[i] = classify_adaline(testImages[i,:],w)
```

Then, we compute the accuracy of the classification at each iteration and we plot the results.

- In `adaline/adaline.py`, fill the code of the function `training-adaline` updating a weight vector using Widrow-Hoff algorithm.
- In `adaline/adaline.py`, fill the code of the function `classify-adaline` performing multi-class classification using a weight vector.
- In `main2.py`, plot the good classification rate versus the number of iterations of the training step using the MNIST dataset from 1 to 200 000. Comment your results.

References

[1] Jiawei Han, Micheline Kamber, Jian Pei. ”Data Mining: Concepts and Techniques”. The Morgan Kaufmann Series in Data Management Systems, 2006.

[2] Tom M. Mitchell. ”Machine learning”. Burr Ridge, IL: McGraw Hill 45, 1997.

[3] B. Widrow and M. E. Hoff. Adaptive switching circuits. 1960 IRE WESCON, pages 96-104, 1960