Mining of Massive Datasets: k-Nearest Neighbor, Decision Trees

Mauro Sozio
Marie Al Ghossein (TA), Maroua Bahri (TA)
Arnaud Guerquin (TA), Pierre-Alexandre Murena (TA)
firstname.lastname@telecom-paristech.fr

1 k-Nearest Neighbor

In this lab session, we are going to train and test a k-nearest neighbor classifier on a collection of documents. In particular, we are going to train a binary classifier which determines whether a given text document deals either with "apple" the fruit or "apple" the software company.

The lab consists of the following three steps:

1. Collecting a set of documents (in English), each of them dealing with either "apple" the fruit or "apple" the company. They can be downloaded from the Web, we recommend at least 10 text documents.

2. Splitting such a dataset in *training set* and *test set*. To this purpose, we use a hold-out evaluation method where $\frac{2}{3}$ of the documents are used as training set and the remaining $\frac{1}{3}$ as test set. We recommend to do the splitting uniformly at random.

3. Learning the k-nearest neighbor classifier on the training set and evaluation of the classifier on the test set. You should report the *accuracy* of the classifier which is defined as the number of documents in the test set that are classified correctly divided by the number of documents in the test set. Study the accuracy for at least 3 different values of k.

4. Evaluate your classifier on the set of documents provided ("text.txt").

1.1 k-nearest neighbor classifier in Python

We are going to use the sci-kit learn library (`http://scikit-learn.org/`) in Python, which is a project sponsored by INRIA, Telecom ParisTech (where the main contributor graduated in 2004) and Google. The first step is to turn a collection of documents into vectors in the Euclidean space. To this end, we are going to use *CountVectorizer* (see documentation at `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html` and the tutorial at `http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html`). The first thing to do is to import the relevant libraries.

```
import sklearn as sk
from sklearn.feature_extraction.text import CountVectorizer
```

Next we use CountVectorizer to turn our training set into a set of points in the euclidean space. In particular the method

`|count_vect.fit_transform()|`

receives in argument a list of strings, each of them representing the text of the corresponding document. It builds a so called term-document matrix, where each row represents a document while each column represents a term (or word in the document). Each entry of such a matrix represents the number of occurrences of the corresponding term in the corresponding document. This is done as follows.

```
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(trainingFiles)
```

One could specify to ignore stopwords (such as articles and common adjectives) which are not informative and might lead to bad results. This is done by specifying

`|stop_words='english'|`

as an argument of

`CountVectorizer()`

Next, we need to convert the results as an array, which is done as follows:

```
training=X_train_counts.toarray()
```

Next, we are going to train our classifier on our training set. We use KNeighborsClassifier (see documentation at `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`) as follows:

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=k)
neigh.fit(training,trainingCl)
```

where k specifies the value of the parameter k in the k-nearest neighbor classifier, while trainingCl is a list specifying for each of the document its training class (i.e. in our case whether the document is about the fruit or the computer).

For evaluating the classifier on the test set we can use the method

`transform(testFiles)`

of $CountVectorizer()$ to turn the documents into vector as well as the method $score(test, testCl)$ of KNeighborsClassifier, which computes the accuracy of the classifier on $test$ (an array of text documents) whose class is specified by a list $testCl$.

1.2 Possible Questions at the Exam

Argue why the choice for k in the k-nearest neighbor classifier is important.

2 Decision Tree Classifier

To solve this exercise, you are provided with a skeleton in Python as well as a Jupyter notebook. We are going to focus on the Iris flower dataset, which was introduced by Ronald Fisher in a 1936 research paper about classification. It is a well-known dataset in data mining and machine learning which is relatively easy to study. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). For each sample, four features are considered: the length and the width of the sepals and petals, in centimetres. We are going to train a decision tree classifier which is able to predict the species (class) of a given flower given the values of those four features. Informations about the Iris dataset can be found here: `http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html`. In our case, we have split the dataset into training and test sets in a "bad way" leading to overfitting. After building the decision tree, you are asked to overcome the overfitting problem.

Overfitting. In classification tasks, the main goal is to be able to build a model (e.g. a decision tree) which determines the (unknown) class of a given instance. The effectiveness of a given model is measured by how well the model is able to determine the class in *unseen* data. To be able to determine the effectiveness of a model, the dataset is split into training set and test set. The model is then built starting from the training set and evaluated on the test set, which has not been seen by the classifier. We have overfitting when the model classifies very well the instances in the training set but performs poorly on the test set (and likely on other unseen data). This happens when the model "memorizes" the training data rather than "learning" from the data and generalizing to unseen data. Therefore, "complicated" models are often the results of overfitting, with one extreme case being when the model is as large as the training data itself. In our class, we have seen a few strategies to overcome overfitting.

The exercise consists of the following steps:

1. Building the decision tree and print the resulting decision tree on a pdf file.
2. Computing the accuracy of the decision tree.
3. Overcoming the overfitting issues.

2.1 Building the Decision tree

We start by loading the data by means of *numpy.load*() as follows:

```
import numpy as np
training_data = np.load('training_data.npy')
```

You are provided with the files

`training_data, training_class, test_data, test_class`

containing the training data, the class for each instance in the training data, the test data, and the class for each instance in the test data, respectively. The type of all those files is *numpy.ndarray* (see `https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html` for the documentation), therefore the type of *training_data* and the other variables is *numpy.ndarray*.

We will build the decision tree by means of $tree.DecisionTreeClassifier$ (`http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`), which is done as follows:

```
clf = tree.DecisionTreeClassifier(criterion='gini', random_state=RandomState(130))
clf = clf.fit(data,class)
```

where 'gini' specifies the criterion according to which the decision tree is built (as we have seen during our class), while $random\_state = RandomState(130)$ makes the algorithm to build the tree deterministic. Do not change those parameters in this part of the exercise. Prediction can be done by $clf.predict()$.

In order to visualize the decision tree and print it on a pdf file, you can use *graphviz* `http://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html`. This yields a '.dot' file which should be converted into a pdf file by means of the *dot* command in Linux or by using the Python modules *pydotplus* or *pydot* (see for example `http://scikit-learn.org/stable/modules/tree.html#tree` and `http://pydotplus.readthedocs.io/reference.html`).

2.2 Accuracy

You should compute the accuracy of the classifier, which is the fraction of instances in the test set that are classified correctly. This should be done for both decision trees, with and without overfitting.

2.3 Overcoming Overfitting

The decision tree built on *training_data* is very good at classifying the instances in the training data but performs poorly on the test set. This is a classical example of overfitting. In order to overcome this problem several strategies are possible, such as *early stopping rule*, *postprocessing*, etc that we mentioned during our class (see the slides on the decision trees). You are free to tune the parameters of the $tree.DecisionTreeClassifier$ so as to overcome the problem of overfitting. You should try to do that by looking only at the structure of the tree, in particular without taking into account the accuracy of the decision tree on the test set.

2.4 Possible Questions at the Exam

At the exam we might ask you one or more of the following questions: 1) what makes you think that you had an overfitting problem? How did you overcome such a problem? Explain why the new decision tree you obtained might perform better at predicting previously unseen instances.

3 Naive Bayes Classifier (Optional)

In this lab session we focus on Naive-Bayes classifiers. We are also going to evaluate the models we build with a k-fold cross evaluation. We are going to train a Naive Bayes classifier for texts that deal with (i) apple the IT company, (ii) apple the fruit, (iii) microsoft the IT company and (iv) banana the fruit. Use the collection of texts in the folder "TechFruits": one text on each line.To this end, we recommend you to use the Naive Bayes classifier available at the scikit-learn library `http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html`.

3.1 k-fold cross validation

In this part of the question, we are going to implement a k-fold cross validation. Even if there are modules taking care of this, it is helpful to implement it for yourself. In order to evaluate a classifier with such an evaluation method, we first split the instances into k subsets with the same size (you can assume that the number of instances is a multiple of k). Each subset is used as test set exactly once while the rest of the subsets are used as training set. Therefore, this requires to run a given classifier k times. Each time, the accuracy is computed as the fraction of instances (documents) in the test set that have been classified correctly. The overall accuracy of the classifier is then measured as the average accuracy of the classifier over the k evaluations. In order to implement the k-fold cross validation the following could be helpful:

```
import numpy as np
a=[1,2,3,4,5,6,7,8,9,10]
a=np.array(a)
b=a[0:4]
c=a[6:]
np.append(b,c,axis=0)
#prints array([ 1,  2,  3,  4,  7,  8,  9, 10])
```