LAB 3: BAYESIAN LEARNING AND BOOSTING

NILS BORE & MARTIN HJELM

October 13, 2015

1. Introduction

In this lab you will implement a Bayes Classifier and the Adaboost algorithm that improves the performance of a weak classifier by aggregating multiple hypotheses generated across different distributions of the training data. Some predefined functions for visualization and basic operations are provided, but you will have to program the key algorithms yourself.

In this exercise we will work with four well known machine learning datasets for classification and see how different modeling assumptions affect the classification. Each dataset is provided as a csv file `datasetnameX.txt` for the feature vectors and the labeling in a file `datasetnameY.txt`. You will be provided with functions that will import the datasets into Python. The dataset we will work with are the following:

1.1. **Wine.** The wine dataset contains the results of chemical analysis of wines grown in the same region in Italy but from three different cultivators. The dataset contains 178 instances, where the feature vector consists of 13 different attributes derived from the analysis. Our task is to classify instances as to belonging to one of the cultivators.

1.2. **Iris.** This dataset contains 150 instances of 3 different types of iris plants. The feature consists of 4 attributes describing the characteristics of the Iris. Our task is to classify instances as belonging to one of the types of Irises.

1.3. **Olivetti Faces.** This dataset contains 400 different images of the faces of 40 different persons, each person is represented by 10 images. The images were taken at different times with varying lightning and facial expressions against a dark homogeneous background. They are all black and white, and 64x64. Our task is to classify instances as belonging to one of the people.

1.4. **Vowel.** This dataset contains 528 instances of utterances of 11 different vowels. Our task is to classify instances as belonging to one of the types of the vowels.

More information on the different datasets can be found at https://archive.ics.uci.edu/ml/.

2. Code skeleton & Python packages

You will use Python for this lab assignment. You have the option to use either pure Python or a Jupyter notebook. There are code skeletons available for both in `lab3.py` and `lab3.ipynb` respectively. At the beginning of both is a short description of Jupyter and how to install it.

You will need the following Python packages for this exercise: `numpy`, `scipy`, `matplotlib` and `sklearn`.

3. Bayesian Learning

3.1. **Bayesian model fitting.** In Bayesian model fitting we wish to infer the model parameters, α, given the data, D. Using Bayes' theorem we can write the posterior for α as,

$$P(\alpha|D) = \frac{P(D|\alpha)P(\alpha)}{P(D)}. \tag{1}$$

Here $P(D|\alpha)$ is the likelihood of the data under our model with the given parameters. $P(\alpha)$ is the prior probability of the parameters and $P(D)$ is a normalizing constant, the model evidence. In words this becomes,

$$Posterior = \frac{Likelihood \times Prior}{Evidence} \tag{2}$$

3.2. **Bayesian Classification.** In a classification scenario we want to classify a set of points as belonging to one of a given set of classes, C. To classify a point $\mathbf{x}^*$ as belonging to the class k we want to compute the class posterior for each of the classes. A straightforward application of Bayes' theorem gives,

$$p(k|\mathbf{x}^*) = \frac{p_k(\mathbf{x}^*|k)\, p(k)}{\sum_{l \in C} p_l(\mathbf{x}^*|l)}, \tag{3}$$

where $p_k(\mathbf{x}^*|k)$ is the class conditional density, $p(k)$ is the prior probability of a point belonging to class k and the sum in the denominator is a normalizing constant. To classify a point we pick the class that has max posterior probability.

3.3. **Modeling.** In this lab we will assume that the density that best models the data for each of the classes, indexed by k, is multivariate Gaussian,

$$p_k(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}_k|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)\boldsymbol{\Sigma}_k{}^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)^{\mathrm{T}}\right) \tag{4}$$

where $\mathbf{x}$ is a d-dimensional row vector, $\boldsymbol{\mu}_k$ is the mean vector, $\boldsymbol{\Sigma}_k$ is the covariance matrix and $|\boldsymbol{\Sigma}_k|$ the determinant.

In a fully Bayesian treatment we would place a prior over the parameters and marginalize them out, but to simplify things we instead choose to find the parameters by the maximum likelihood (ML) estimate. We also assume that all of the data points are independent and identically distributed (i.i.d). We write the likelihood for the ML-estimate as,

$$\arg\max_{\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k} \mathcal{L}(\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k|D_k) = p_k(D_k|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) \propto \prod_{\{i|c_i=k\}}^{N} p_k(\mathbf{x}_i|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k), \tag{5}$$

where D_k are the points in D belonging to class k.

A less complicated form of the Gaussian to work with is the log transform, the log-likelihood,

$$\ln(p_k(\mathbf{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_\mathbf{k})) = -\frac{1}{2}\ln(|\boldsymbol{\Sigma}_k|) - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathrm{T}} - \frac{d}{2}\ln(2\pi) \tag{6}$$

To maximize log likelihood we take the derivate with respect to both $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_\mathbf{k}$ and equate to zero,

$$\frac{d\ln(\mathcal{L})}{d\boldsymbol{\mu}_k} = 0, \quad \frac{d\ln(\mathcal{L})}{d\boldsymbol{\Sigma}_k} = 0. \tag{7}$$

After some manipulation we arrive at the following expressions for the ML-estimate of $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$. c_i denotes the class of the i:th training instance:

$$\boldsymbol{\mu}_k = \frac{\sum_{\{i|c_i=k\}} \mathbf{x}_i}{N_k} \tag{8}$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k}\sum_{\{i|c_i=k\}} (\mathbf{x}_i-\boldsymbol{\mu}_k)^{\mathrm{T}}(\mathbf{x}_i-\boldsymbol{\mu}_k) \tag{9}$$

3.4. **Assignment 1.** Write a function, `mlParams(X,labels)`, that computes the ML-estimates of $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ for the different classes in the dataset. `X` here is a set of row vectors, and `labels` are the class labels for each of the data points. The function should return a $C \times d$-array `mu` that contains the class means, a $d \times d \times C$-array `sigma` that contains the class covariances. The covariance should be computed using matrix multiplication and not by applying a library function.

Use the provided function, `genBlobs()`, that returns Gaussian distributed data points together with class labels, to generate some test data. Compute the ML-estimates for the data and plot the 95%-confidence interval using the function `plotGaussians`.

3.5. **Classification.** The next step is to program the discriminant function based on the log posterior, $\delta_k(\mathbf{x}) = \ln(p(k|\mathbf{x}))$ for predicting the class of an unseen instance, $\mathbf{x}^*$. Using Eq. 3 and the log transform we can write the discriminant function as,

$$\begin{aligned}\delta_k(\mathbf{x}^*) &= \ln(p(k|\mathbf{x}^*)) = \ln(p_k(\mathbf{x}^*|k)) + \ln(p(k)) - \ln\sum_{l\in C} p_l(\mathbf{x}^*|l) \\ &= -\frac{1}{2}\ln(|\mathbf{\Sigma}_k|\,) - \frac{1}{2}(\mathbf{x}^* - \boldsymbol{\mu}_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} - \frac{d}{2}\ln(2\pi) + \ln(p(k)) - \ln\sum_{l\in C} p_l(\mathbf{x}^*|l) \\ &= -\frac{1}{2}\ln(|\mathbf{\Sigma}_k|\,) - \frac{1}{2}(\mathbf{x}^* - \boldsymbol{\mu}_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} + \ln(p(k)) + \mathcal{C}.\end{aligned} \tag{10}$$

When classifying new data points, we can ignore $\mathcal{C}$ when comparing the values given by Eq. 10 as it will not vary with our test data or class assignments.

We compute the class prior, $p(k)$, as the frequency of the occurrences of the different classes,

$$p(k) = \frac{N_k}{N}. \tag{11}$$

Observe that $\mathbf{\Sigma}_k^{-1}$ should never be computed explicitly as finding the inverse is time consuming and inexact, instead solve the equation system, $\mathbf{\Sigma}_k\mathbf{y} = (\mathbf{x}^* - \boldsymbol{\mu}_k)^T$.

Note that $\mathbf{\Sigma}_k$ most of the time is symmetric positive definite (if not you can add a diagonal matrix $1E-6I$ to it). This means that $\mathbf{\Sigma}_k$ can be factorized into $\mathbf{\Sigma}_k = \mathbf{L}\mathbf{L}^T$ by Cholesky factorization, where $\mathbf{L}$ is an upper triangular matrix. The factorization can be used in computing the product containing the inverse covariance, $\mathbf{\Sigma}_k^{-1}$, in the following manner,

$$\begin{aligned}&\text{Solve :}\mathbf{\Sigma}_k\mathbf{y} = (\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} \\ &\text{Decompose : } \mathbf{\Sigma}_k = \mathbf{L}\mathbf{L}^T \\ &\text{Solve : } \mathbf{L}\mathbf{v} = (\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} \\ &\text{Solve : } \mathbf{L}^T\mathbf{y} = \mathbf{v}\end{aligned} \tag{12}$$

Solving for an upper triangular matrix should roughly half the solution time. In Python the code for the generic problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ is:

```
L = np.linalg.cholesky(A)
y = np.linalg.solve(L,b)
x = np.linalg.solve(L.H,y)
```

Using the factorization we can also write the determinant as:

$$\ln(|\mathbf{\Sigma}_k|) = \ln(|\mathbf{L}\mathbf{L}^T|) = \ln(|\mathbf{L}||\mathbf{L}|) = \ln(\prod_{i=1}^{d} l_{i,i}^2) = 2\sum_{i=1}^{d} \ln l_{i,i} \tag{13}$$

3.6. **Assignment 2.**

(1) Write a function `computePrior(labels)` that estimates and returns the class prior in `X`.

(2) Write a function `classify(X,prior,mu,sigma,covdiag)` that computes the discriminant function values for all classes and data points, and classifies each point to belong to the max discriminant value. The function should return an $N \times 1$ matrix containing the predicted class value for each point.

(3) In this lab we are interested in the modeling assumption's influence on the classification results. We will study two different cases, one where we assume all features are uncorrelated (naive Bayes) and one where they are correlated. This means that we will have two different covariances matrices when classifying, **one diagonal and one non-diagonal**. You should therefore add a boolean parameter `covdiag` to the `classify` function that makes it possible to choose classification based on the full or the diagonal covariance matrix. When the covariance matrix is diagonal we can skip the Cholesky factorization and compute the inverse and the log determinant sum directly.

3.7. **Assignment 3.** We now have all functions we need for doing the training and classification. Use the provided function `testClassifier` to test the accuracy for each of the datasets. `testClassifier` runs a loop that does the following things:

0. Uses the provided random partitioning function to split the dataset into a training and test dataset.
1. Trains your classifier on the training partition.
2. Evaluates the performance of the classifier on the test partition.

Run `testClassifier` for each of the datasets for the two assumptions of feature dependence by setting the function parameter `covdiag` to `True` or `False`.

Answer the following questions:

(1) Does the feature independence assumption have any effect on the classification accuracy for the different datasets?
(2) If so why does some of the datasets have more difference than others?
(3) When can an independence assumption be reasonable and when not?
(4) How does the standard deviation differ for the two assumptions and what does that imply?

4. Boosting

4.1. **Boosting the Bayes Classifier.** Boosting aggregates multiple hypotheses generated by the same learning algorithm invoked over different distributions of training data into a single composite classifier. Boosting generates a classifier with a smaller error on the training data as it combines multiple hypotheses which individually have a larger error (but lower than 50%). Boosting requires unstable classifiers whose learning algorithms are sensitive to changes in the training examples.

The idea of boosting is to repeatedly apply a weak learning algorithm on various distributions of the training data and to aggregate the individual classifiers into a single overall classifier. After each iteration the distribution of training instances is changed based on the error the current classifier exhibits on the training set. The weight ω_i of an instance $(\mathbf{x}_i, c_i)$ specifies its relative importance, which can be interpreted as if the training set would contain ω_i identical copies of the training example $(\mathbf{x}_i, c_i)$. The weights ω_i of correctly classified instances $(\mathbf{x}_i, c_i)$ are reduced, whereas those of incorrectly classified instances are increased. Thereby the next invocation of the learning algorithm will focus on the incorrect examples.

In order to be able to boost the Bayes classifier, the algorithm for computing the MAP parameters and the discriminant function has to be modified such that it can deal with fractional (weighted) instances. Assume, that ω_i is the weight assigned to the i:th training instance. Without going into a straightforward detailed derivation the Equations 8-9 for the MAP parameter with weighted instances become:

$$\boldsymbol{\mu}_k = \frac{\sum_{\{i|c_i=k\}} \omega_i \mathbf{x}_i}{\sum_{\{i|c_i=k\}} \omega_i} \tag{14}$$

$$\boldsymbol{\Sigma}_k^2 = \frac{\sum_{\{i|c_i=k\}} \omega_i \left(\mathbf{x}_i - \boldsymbol{\mu}_k\right)^T \left(\mathbf{x}_i - \boldsymbol{\mu}_k\right)}{\sum_{\{i|c_i=k\}} \omega_i} \tag{15}$$

4.2. **Assignment 4:** Extend the old `mlParams` function to `mlParams(X, labels, W)` that handles weighted instances. Again `X` is $N \times d$ matrix of feature vectors, `labels` a $N \times 1$-vector containing the corresponding labels and `W` is a $N \times 1$ vector of weights. The signature should look like

```
def mlParams(X, labels, W)
    ...
    return (mu, sigma)
```

Here, the return parameters `mu` and `sigma` are identical to the old `mlParams`. The function computes the maximum posterior parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ for a dataset D according to Equations 14-15. Assume the usual data format for the first two parameters. Test your function `mlParams(X, labels, W)`, for a uniform weight vector with $\boldsymbol{\omega} = 1/N$. The MAP parameters should be identical to those obtained with the previous version of `mlParams`.

4.3. **The Adaboost algorithm.** The Adaboost algorithm repeatedly invokes a weak learning algorithm, in our case `mlParams` and after each round updates the weights of training instances $(\mathbf{x}_i, c_i)$. In the following, t will denote the iteration round of the algorithm. Adaboost proceeds in the following manner (repeating steps 1-4 several times).

0. Initialize all weights uniformly $\omega_i^1 = 1/N$.
1. Train weak learner using distribution $\boldsymbol{\omega}^t$.
2. Get weak hypothesis h^t and compute its error ϵ^t with respect to the weighted distribution $\boldsymbol{\omega}^t$. In case of the Bayes classifier a single hypothesis h^t is represented by $(\boldsymbol{\mu}_k^t, \boldsymbol{\Sigma}_k^t)$.
$$\epsilon^t = \sum_{i=1}^{N} \omega_i^t \left(1 - \delta(h^t(\mathbf{x}_i), c_i)\right)$$
where $h^t(\mathbf{x}_i)$ is the classification of instance $\mathbf{x}_i$ made by the hypothesis h^t. The function $\delta(h^t(\mathbf{x}_i), c_i)$ is 1 if $h^t(\mathbf{x}_i) = c_i$ and 0 otherwise.
3. Choose $\alpha^t = \frac{1}{2}\left(\ln(1-\epsilon^t) - \ln(\epsilon^t)\right)$.
4. Update weights according to
$$\omega_i^{t+1} = \frac{\omega_i^t}{Z^t} \times \begin{cases} e^{-\alpha^t} & \text{if } h^t(\mathbf{x}_i) = c_i \\ e^{\alpha^t} & \text{if } h^t(\mathbf{x}_i) \neq c_i \end{cases}$$
where Z^t is a normalization factor ensuring that $\sum_i \omega_i^{t+1} = 1$.

The overall classification of the boosted classifier of an unseen instance $\mathbf{x}$ is obtained by aggregating the votes casted by each individual Bayes classifier $h^t = (\boldsymbol{\mu}_k^t, \boldsymbol{\Sigma}_k^t)$. As we have higher confidence in classifiers that have a low error (large α^t), their votes count relatively more. The final classification $H(\mathbf{x})$ is the class c_{max} that receives the majority of votes

$$H(\mathbf{x}) = c_{max} = \arg\max_{c_i} \sum_{t=1}^{T} \alpha^t \delta(h^t(\mathbf{x}), c_i) \tag{16}$$

4.4. **Assignment 5:**

(1) Modify `computePrior` to have the signature `computePrior(labels, W)`, taking the boosting weights $\boldsymbol{\omega}$ into account. We can look at the weights as taking a particular training point $\mathbf{x}_i$ into account ω_i times. So if previously there was N_k points in a particular class, we should now think about "how many times we count" each point. Note that the prior probabilities should still sum to one.

(2) Implement the Adaboost algorithm and apply it to the Bayes classifier. Design a function `trainBoost(X, labels, T)` that generates a set of boosted hypothesis, where the parameter `T` determines the number of hypotheses. Use the modified `computePrior(labels, W)`. The signature in Python should look like

```
def trainBoost(X, labels, T):
    ...
    return (priors, mus, sigmas, alphas)
```

(3) Design a function

```
def classifyBoost(X, priors, mus, sigmas, alphas):
    ...
    return c
```

that classifies the instances in data by means of the aggregated boosted classifier according to Equation 16. The resulting classifications are returned in the vector `c`.

Observe: The return parameter `alphas` ($T \times 1$) holds the classifier vote weights α^t. `mus`, `sigmas` and `priors` contain lists of length T with MAP parameters and priors for every classifier. `c` will be a $N \times 1$ vector. Note that you have to compute and store all the hypothesis generated with `mlParams(data, labels, W)` for each of the different distributions $\boldsymbol{\omega}^{t+1}$ and later aggregate their classifications to obtain the overall classification.

Compute the classification accuracy of the boosted classifier on some data sets using `testClassifier` and compare it with those of the basic classifier (see Assignment 3):

(1) Is there any improvement in classification accuracy? Why/why not?
(2) Plot the decision boundary of the boosted classifier and compare it with that of the basic. What differences do you notice? Is the boundary of the boosted version more complex?
(3) Can we make up for not using a more advanced model in the basic classifier (e.g. independent features) by using boosting?

You may use the function `plotBoundary` provided in the code skeleton to plot the decision boundary for different datasets and parameters.

5. The End

If you have followed the instructions you should be done now. Please report your findings carefully by saving your plots and classification results. Use these to reason about the questions in the lab description.

6. Appendix: Numpy

6.1. **Numpy Creation of Matrices.** During the creation of the `sigma` matrix, we have to create a three dimensional matrix in `numpy`. Below is an example of how we might achieve this.

```
import numpy

# example values
```

```
d = 7
C = 4
# a matrix that we want to fill with values
As = numpy.zeros((d, d, C))
for i in range(C):
    A = numpy.zeros((d, d))
    # ...
    # somehow fill A with values
    # A could also be created directly, e.g. through some product
    # ...
    As[:, :, i] = A
```

6.2. **Numpy Logical Indexing.** Given a data vector `X` of row vectors and a label vector y we want to extract the data points for one of the labeled classes. If we assume that `y` consists of four different classes and we want to extract all vectors from `X` for a corresponding label we can do the following:

```
import numpy
X,y = getData()
Xcl1 = X[y==0,:]
Xcl2 = X[y==1,:]
Xcl3 = X[y==2,:]
Xcl4 = X[y==3,:]
```

6.3. **Broadcasting.** Sometimes we want to subtract or add a row vector `u` to a data vector `X` consisting of row vectors. Instead of iterating over all rows in `X` we can make use of something in numpy called broadcasting. Broadcasting vectorizes arithmetic array operations such that looping occurs in C instead of Python. It can be used in the following way:

```
import numpy

# Subtract the vector using for loop
X,y = getData()
u = getSubtractionVec()
for i_row in range(0,X.shape[0]):
    X[i_row,:] = X[i_row,:] - u

# Subtract using broadcasting
X,y = getData()
u = getSubtractionVec()
X  = X - u
```