LAB 3: BAYESIAN LEARNING AND BOOSTING

NILS BORE & MARTIN HJELM

September 26, 2016

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 three well known machine learning datasets for classification and see how different modeling assumptions affect the classification. Each dataset is provided in the form of two csv files: `datasetnameX.txt` for the feature vectors and `datasetnameY.txt` for the corresponding labels. You will be provided with functions that will import the datasets into Python. The datasets we will work with are the following:

Figure 1. Example of Iris flower, by David Iliff. License: CC-BY-SA 3.0.

1.1. **Iris.** This dataset contains 150 instances of 3 different types of iris plants. The feature consists of 2 attributes describing the characteristics of the Iris. Our task is to classify instances as belonging to one of the types of Irises.

1.2. **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.

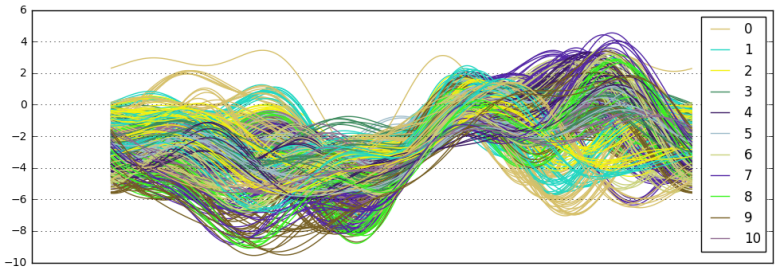

FIGURE 2. Visualization of the different classes of vowels.

FIGURE 3. A few example images from the Olivetti faces dataset.

1.3. **(Voluntary) 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 lighting and facial expressions against a dark homogeneous background. They are all grayscale, and 64x64 pixels. Our task is to classify instances as belonging to one of the people.

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`. For nicer plots, you may also use the `seaborn` library. To enable this, uncomment lines 12-13 in `labfuns.py`.

3. REALLY Useful Tips For Python and Numpy

3.1. First of all.

- Use logical indexing and broadcasting.
- Iterate over classes not data points.

- Try to avoid for loops when possible. If you cannot think in matrix multiplications, do the for loop first then simplify.
- Remember that 2/3 might not be the same as 2./3., that is, integer division vs. float division.
- Comment your code.

3.2. **Numpy Creation of Matrices.** Here are some simple instructions for matrices. It is good to preallocate the matrix before usage to avoid resizing, etc.

```
import numpy as np

# Dimensions of Matrices
rows = 7
cols = 4
depth = 5

# Creating matrices
A = np.zeros((rows,cols)) # 2D Matrix of zeros
print(A.shape)
>> (7,4)
A = np.zeros((depth,rows,cols))  # 3D Matrix of zeros
A = np.ones((rows,cols)) # 2D Matrix of ones
A = np.array([(1,2,3),(4,5,6),(7,8,9)]) # 2D 3x3 matrix with values
# Turn it into a square diagonal matrix with zeros of-diagonal
d = np.diag(A) # Get diagonal as a row vector
d = np.diag(d) # Turn a row vector into a diagonal matrix
print(d)
>> array([[1, 0, 0],
       [0, 5, 0],
       [0, 0, 9]])
# This works
print(1.0/A)
>> array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667],
       [ 0.14285714,  0.125     ,  0.11111111]])
```

3.3. **Numpy reshaping.** Sometimes you will need to convert a vector of size (N,) to a matrix (N,1) or a matrix to vector

```
a = np.array((1,2,3))
print(a.shape)
>>(3,)
print(a.reshape(-1,1).shape)
>>(3,1)
print(a.reshape(1,-1).shape)
>>(1,3)
```

```
print(a.reshape(1,-1).reshape(-1).shape)
>>(3,)
```

3.4. **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 the specific classes for the corresponding label we can do the following:

```
X,y = getData()
classes = np.unique(y) # Get the unique examples
# Iterate over both index and value
for jdx,class in enumerate(classes):
    idx = y==class # Returns a true or false with the length of y
    # Or more compactly extract the indices for which y==class is true,
    # analogous to MATLAB's find
    idx = np.where(y==class)[0]
    xlc = X[idx,:] # Get the x for the class labels. Vectors are rows.
```

3.5. **Broadcasting.** Sometimes we want to subtract or add a row vector `u` to a matrix `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:

```
# Subtract the vector using for loop
X = np.array([(1,2,3),(4,5,6),(7,8,9)])
u = np.array((1,2,3))
for i_row in range(0,X.shape[0]):
    X[i_row,:] = X[i_row,:] - u
print(X)
>> array([[0, 0, 0],
       [3, 3, 3],
       [6, 6, 6]])

# Subtract using broadcasting
X = np.array([(1,2,3),(4,5,6),(7,8,9)])
u = np.array((1,2,3))
X = X - u
print(X)
>> array([[0, 0, 0],
       [3, 3, 3],
       [6, 6, 6]])
```

4. Bayesian Learning

4.1. **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,

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

4.2. **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,

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

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.

4.3. **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,

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

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,

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

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

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

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

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

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

For simplicity, we will assume that all of the feature dimensions are uncorrelated, with no off diagonal covariance elements. This allows us to further simplify the expression for the covariance matrix at diagonal indices (m, m) and $(m, n), m \neq n$ respectively to

$$\mathbf{\Sigma}_k(m,m) = \frac{1}{N_k}\sum_{\{i|c_i=k\}} (\mathbf{x}_i(m) - \boldsymbol{\mu}_k(m))^2, \text{ and } \mathbf{\Sigma}_k(m,n) = 0 \text{ for } m \neq n. \tag{10}$$

This is knowm as the *naive Bayes* assumption, making it a Naive Bayes Classifier.

4.4. **Assignment 1.** Write a function, `mlParams(X,labels)`, that computes the ML-estimates of $\boldsymbol{\mu}_k$ and $\mathbf{\Sigma}_k$ 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 (again, ignore the `W` argument for now). The function should return a $C \times d$-array `mu` that contains the class means, a $C \times d \times d$-array `sigma` that contains the class covariances. The covariance should be implemented using your own code 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`.

4.5. **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,

$$\begin{aligned}\delta_k(\mathbf{x}^*) &= \ln(p(k|\mathbf{x}^*)) = \ln(p_k(\mathbf{x}^*|k)) + \ln(p(k)) - \ln \sum_{l \in C} p_l(\mathbf{x}^*|l) \\ &= -\frac{1}{2}\ln(|\mathbf{\Sigma}_k|) - \frac{1}{2}(\mathbf{x}^* - \boldsymbol{\mu}_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} - \frac{d}{2}\ln(2\pi) + \ln(p(k)) - \ln \sum_{l \in C} p_l(\mathbf{x}^*|l) \\ &= -\frac{1}{2}\ln(|\mathbf{\Sigma}_k|) - \frac{1}{2}(\mathbf{x}^* - \boldsymbol{\mu}_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x}^* - \boldsymbol{\mu}_k)^{\mathrm{T}} + \ln(p(k)) + \mathcal{C}.\end{aligned} \tag{11}$$

Since we have diagonal covariance matrices $\mathbf{\Sigma}_k$, there is no need to compute the inverse of the full matrices. Rather, these products can be computed for each dimension separately.

When classifying new data points, we can ignore $\mathcal{C}$ when comparing the values given by Eq. 11 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,

$$p(k) = \frac{N_k}{N}. \tag{12}$$

4.6. **Assignment 2.**

(1) Write a function `computePrior(labels)` that estimates and returns the class prior in `X` (ignore the `W` argument).
(2) Write a function `classifyBayes(X,prior,mu,sigma)` 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 a length N vector containing the predicted class value for each point.

4.7. **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 the `vowels` and `iris` 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. Evaluate the performance of the classifier on the test partition.

Run `testClassifier` for the datasets and take note of the accuracies. Use `plotBoundary` to plot the decision boundary of the 2D `iris` dataset.

Answer the following questions:

(1) When can a feature independence assumption be reasonable and when not?

(2) How does the decision boundary look for the Iris dataset? How could one improve the classification results for this scenario by changing classifier or, alternatively, manipulating the data?

5. Boosting

5.1. **Boosting a Weak 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 Equations 8, 10 for the MAP parameter with weighted instances become:

$$\boldsymbol{\mu}_k = \frac{\sum_{\{i|c_i=k\}} \omega_i \mathbf{x}_i}{\sum_{\{i|c_i=k\}} \omega_i} \tag{13}$$

$$\boldsymbol{\Sigma}_k(m,m) = \frac{1}{\sum_{\{i|c_i=k\}} \omega_i} \sum_{\{i|c_i=k\}} \omega_i (\mathbf{x}_i(m) - \boldsymbol{\mu}_k(m))^2, \text{ and } \boldsymbol{\Sigma}_k(m,n) = 0 \text{ for } m \neq n. \tag{14}$$

5.2. **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 length N vector containing the corresponding labels and `W` is a $N \times 1$ matrix of weights. The signature should look like

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

Here, the types of 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 13-14. 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`.

5.3. **The Adaboost algorithm.** The Adaboost algorithm repeatedly invokes a weak learning algorithm, for example the `BayesClassifier` that we implemented, 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. Further, we assume that $\boldsymbol{\omega}$ is always normalized, $\sum_i \omega_i = 1$. Adaboost proceeds as follows (repeating steps 1-4 for every t).

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 the the learned parameters $(\boldsymbol{\mu}_k^t, \boldsymbol{\Sigma}_k^t)$. The weighted error is given by

$$\epsilon^t = \sum_{i=1}^{N} \omega_i^t \left(1 - \delta(h^t(\mathbf{x}_i), c_i)\right),$$

   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 classifier. 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

$$H(\mathbf{x}) = c_{max} = \arg\max_{c_i} \sum_{t=1}^{T} \alpha^t \delta(h^t(\mathbf{x}), c_i) \tag{15}$$

5.4. **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(base_classifier, X, labels, T)` that generates a set of boosted hypotheses, where the parameter `T` determines the number of hypotheses. Use the modified `computePrior(labels, W)`. The signature in Python should look like

```
def trainBoost(base_classifier, X, labels, T):
    ...
    return classifiers, alphas
```

Note that a new classifier of type `base_classifier` can be trained by calling `new_classifier = base_classifier.trainClassifier(X, y, W)`, which can then be used for classification by calling `new_classifier.classify(X)`.

(3) Design a function

```
def classifyBoost(X, classifiers, alphas, Nclasses):
    ...
    return yPred
```

that classifies the instances in data by means of the aggregated boosted classifier according to Equation 15. The resulting classifications are returned in the vector `yPred`.

Observe: The return parameter `alphas`, a length T list, holds the classifier vote weights α^t. `classifiers` should be a length T list of trained classifiers. `yPred` will be a length N vector of predicted class labels. Note that you have to compute and store all the hypotheses generated with `classifiers[t].classify(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` from `labfuns.py` and compare it with those of the basic classifier on the `vowels` and `iris` data sets (see Assignment 3):

(1) Is there any improvement in classification accuracy? Why/why not?
(2) Plot the decision boundary of the boosted classifier on `iris` 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 `labfuns.py` to plot the decision boundary for different datasets and parameters.

5.5. **Assignment 6.** We have implemented a class `DecisionTreeClassifier` based upon skLearns decision tree classifier. The skLearn implementation is similar to the one

used in the first lab, however, here the values are continuous and we use the default Gini index to compute the split.

Test the decision tree classifier on the `vowels` and `iris` data sets. Repeat but now by passing it as an argument to the `BoostClassifier` object. Answer questions 1-3 in assignment 5 for the decision tree.

5.6. **Assignment 7.** If you had to pick a classifier, naive Bayes or a decision tree or the boosted versions of these, which one would you pick? Motivate from the following criteria:

- Outliers
- Irrelevant inputs: part of the feature space is irrelevant
- Predictive power
- Mixed types of data: binary, categorical or continuous features, etc.
- Scalability: the dimension of the data, D, is large or the number of instances, N, is large, or both.

6. 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.

For the interested, there is also the voluntary assignment at the end. This showcases how we can use the already implemented code to classify and visualize real world data.

7. Voluntary Assignment

Note that this part is completely voluntary!

Figure 4. Example of `visualizeOlivettiVectors` output.

7.1. **Classifying Faces.** Now we can use the implemented classifiers to achieve something a bit more tangible. For this section, we will refer mostly to the code. In `labfuns.py`, there is a function called `visualizeOlivettiVectors`, which takes in columns of training vectors from the Olivetti data set together with a single test point. It then visualizes these images together. The idea is that we will take a test point at random, classify it, and then visualize it together with the training vectors belonging to that class to see how well the classification worked.

To begin with, we should try some different combinations of classifiers and boosted classifiers to see how good results we can expect. While a boosted `BayesClassifier` might be too computationally demanding to try out, you can try the basic bayes classifier as well as `DecisionTreeClassifier` with and without boosting. In some combinations, you should be able to approach accuracies of 90%.

When you have identified a good combination, simply use it with the supplied code to visualize the classifications. An example output is shown in Figure 4.