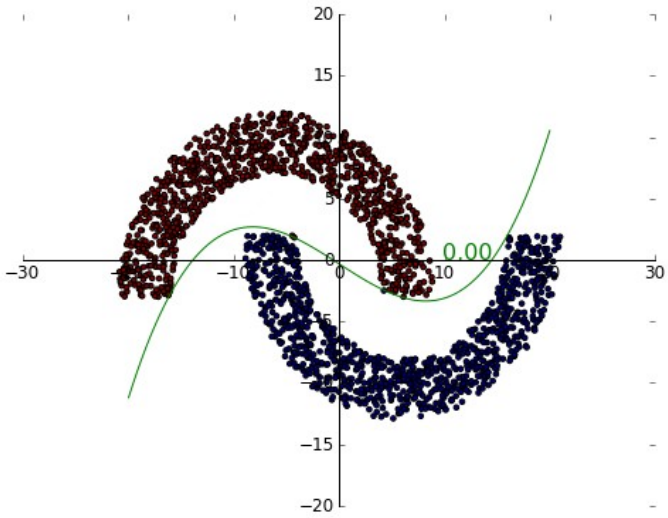
Stochastic Gradient Descent Experiments



Abstract

This project is about various approaches to machine learning with a goal of implementing a linear classification scheme for the given data. Different specialized algorithms adapted from the more general gradient descent are experimented with. Programming language of choice is Python, using the matplotlib and numpy libraries.

Markus Viljanen

6.5.2013

majuvi@utu.fi

http://users.utu.fi/majuvi

A simple perceptron

Algorithm for a perceptron

Perceptron learning algorithm is quite simple.Assume we have a set of training examples $Y=\{(x_1,y_1),\ldots,(x_n,y_n)\}$ where $x\in X=\{1\}\times\mathbb{R}^d$ and $y\in\{-1,1\}$, with the hypothesis set $h\in H$ specified by the functional form $h(\vec{x})=sign(\vec{w}^T\vec{x})$. Perceptron is an iterative method for minimizing the in sample error $E_{\text{in}}(h)=\frac{1}{N}\sum_{n=1}^{N}[\![h(\vec{x}_n)\neq f(\vec{y}_n)]\!]$. It works like this: In each iteration, pick a random (x_n,y_n) and compute $\rho(t)=\vec{w}^T(t)\vec{x}(t)$. If $y(t)p(t)\leq 0$ update $\vec{w}$ by $\vec{w}(t+1)=\vec{w}(t)+y(t)\vec{x}(t)$. Iteration steps are repeated until a threshold such as a number of iteration steps is reached. It can be proven that for linearly seperable data the perceptron algorithm does in fact converge to the right solution (**Problem 1.3**). I would write the proof here if the OpenOffice formula editor wasn't such a pain to work with.

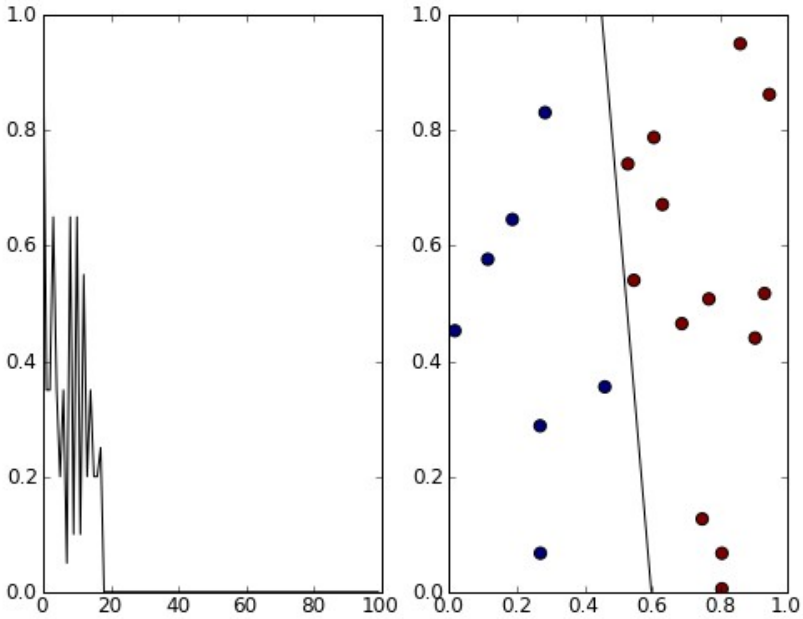
A simplified sample source code would be:

```
X = vstack( (ones(n), s*random.random((2,n))) )
F = array([-0.8,1,0.5])
Y = sign(dot(F,X))
W = zeros(3)

for i in range(iterations):

        r = random.randint(len(X))
        x = X[:,r]
        y = Y[r]

        if (sign(dot(W,x)) != y):
                W = W+Y[r]*X[:,r]
```

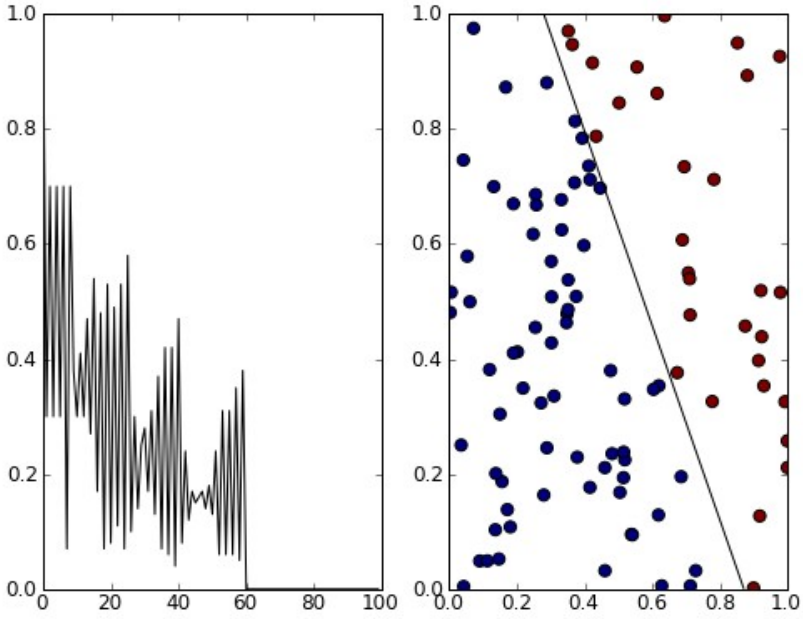
In generating our graphics we have an adapted version where in each iteration an $\vec{x}_n$ is picked such that $sign(\vec{w}^T \vec{x}_n) \neq \vec{y}_n$. This results in graphs that appear to converge much faster with no extended flat surfaces. With regards to performace computationally it makes no difference, in fact we could just update the graph when $\vec{w}(t)$ is modified.

Experiments in $\mathbb{R}^2$ with various sample sizes

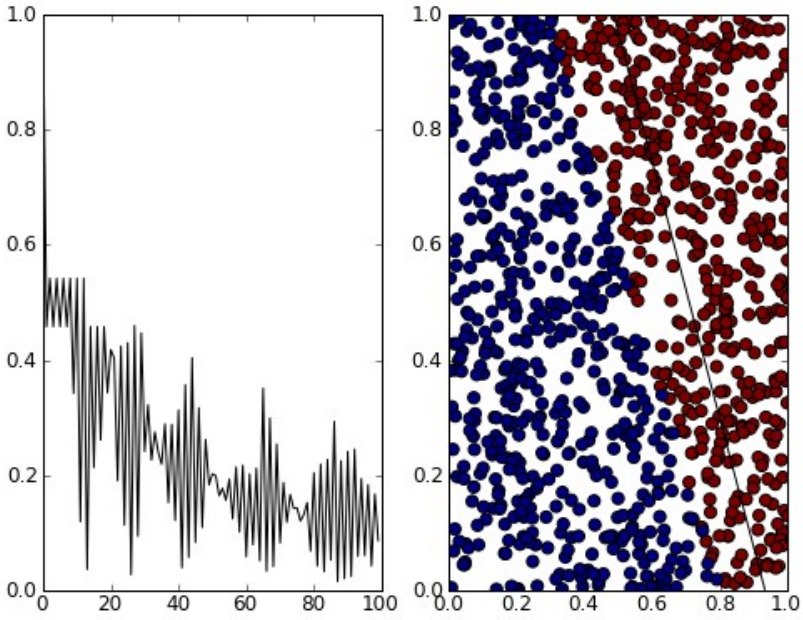
The following graphs are my solutions to **Problem 1.4** in the book. The file *perceptron.py* contains necessary tools to create the graphs by adjusting sample size in the source code. In the following graphs perceptron algorithm is ran with random samples which are linearly seperatable by an unknown line. The image on the left presents the evolution of the in sample error as a function of iteration step:



Running a perceptron on 20 random samples.



Running a perceptron on 100 random samples.

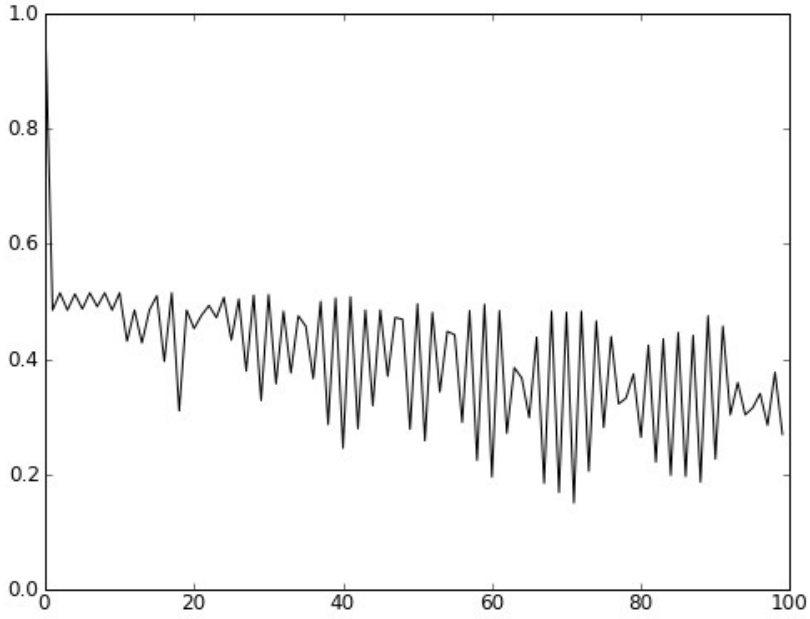


Running a perceptron on 1000 random samples.

It is clear that with more samples the perceptron takes longer to converge.

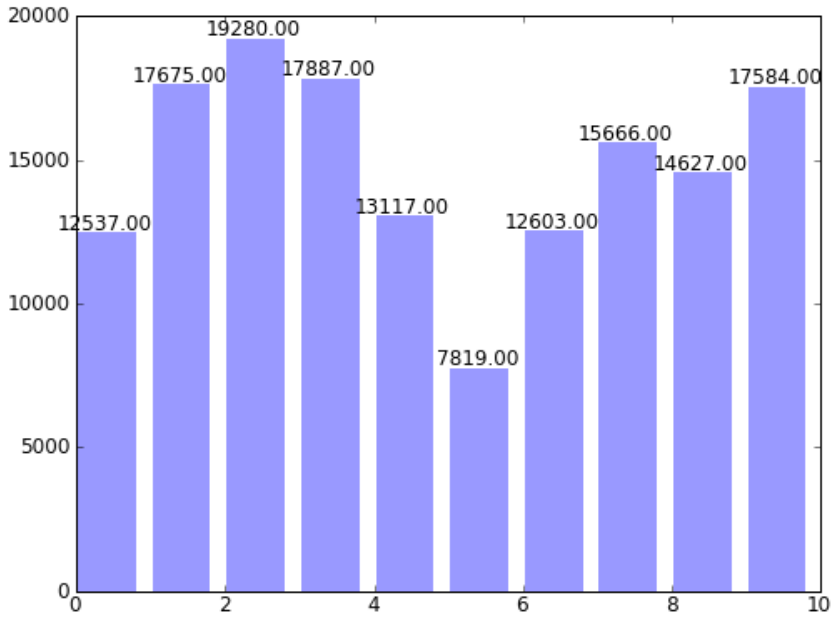
Behavior in higher dimensions

Next we explore the relationship with more dimensions by applying the perceptron to $\mathbb{R}^{10}$ using 1000 samples:



Evolution of a perceptron with 10 dimensions and 1000 samples.

This appears to converge so slowly that it is a reasonable question to ask how many iteration it will take to converge. Running it so that every single point $\vec{x}_n$ is correctly classified takes on average 10000 iteration steps:



Iteration steps taken to achieve perfect classification for 10 seperate runs.

We shall not explore this relationship further, being content for now knowing that it will converge.

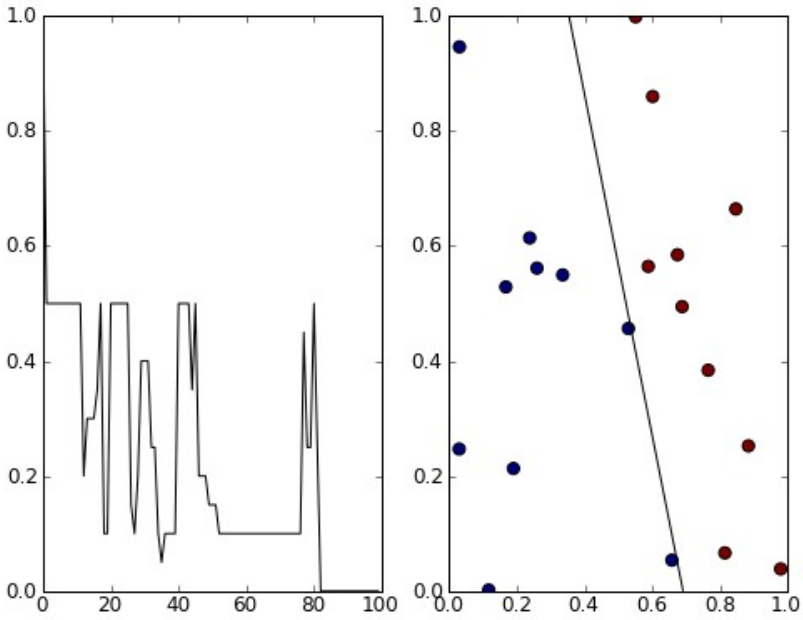
Adaptive Linear Neuron (Adaline)

Algorithm for the modified perceptron

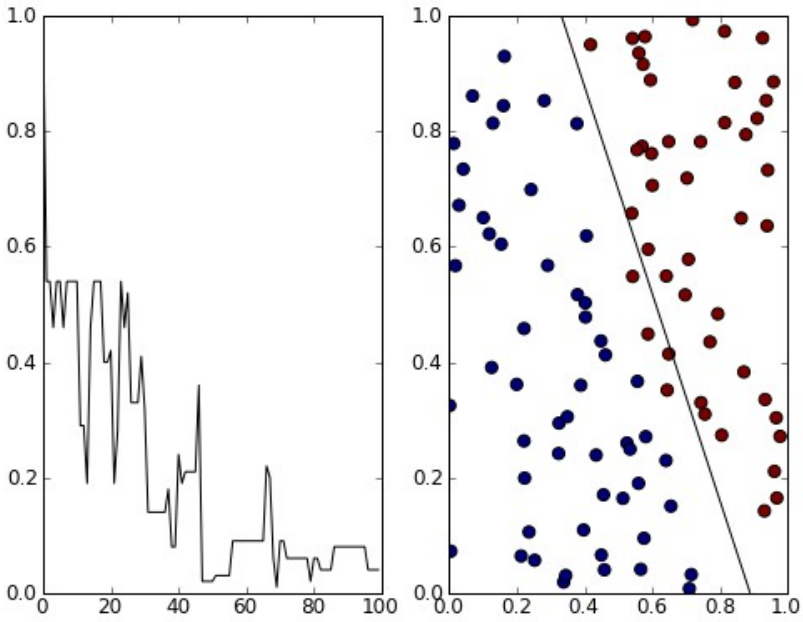
The perceptron algorithm does not take into account how close the misclassified point $\vec{x}_n$ is to the line $\vec{w}(t)$, moving it always the same amount. Adapted algorithm calculates the distance and uses it to have different step sizes. Assuming the perceptron framework, we make the following modification: In each iteration, pick a random (x_n, y_n) and compute $\rho(t)=\vec{w}^T(t)\vec{x}(t)$. If $y(t)p(t)\leq 1$ update $\vec{w}$ by $\vec{w}(t+1)=\vec{w}(t)+\eta*(y(t)-\rho(t))\vec{x}(t)$ where η is some constant.

Improvements with the same samples

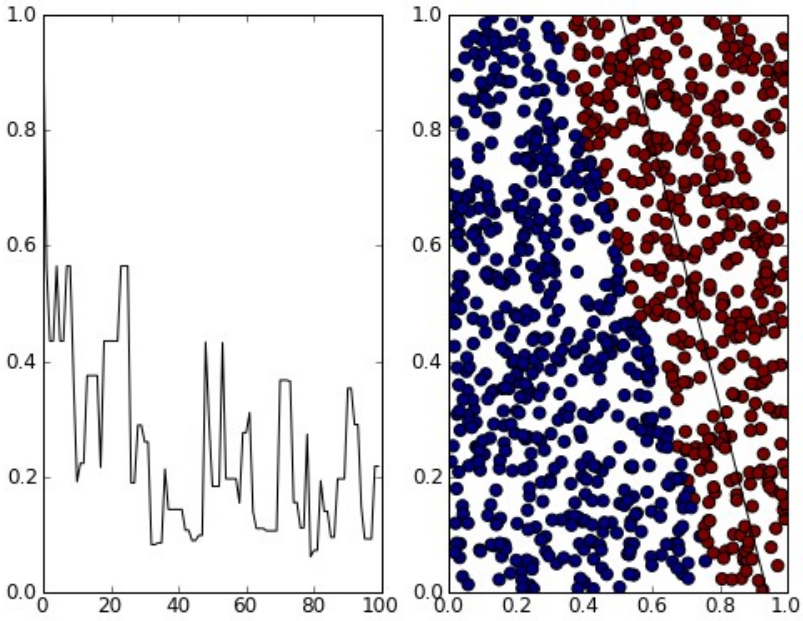
The following samples are solutions to **Problem 1.5**. The graphing algorithm can be found in *adaline.py.* Because Adaline has a somewhat smarter convergence, the flatlines where we pick a correctly classified point and do nothing are included. Thus the graphs here are not directly comparable to the perceptron graphs.



Adaline with 10 samples.

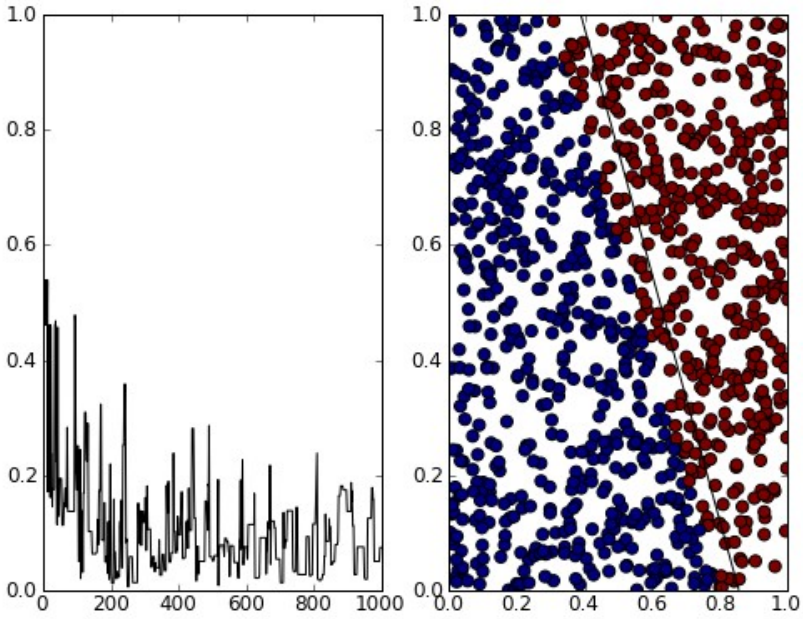


Adaline with 100 samples.



Adaline with 1000 samples.

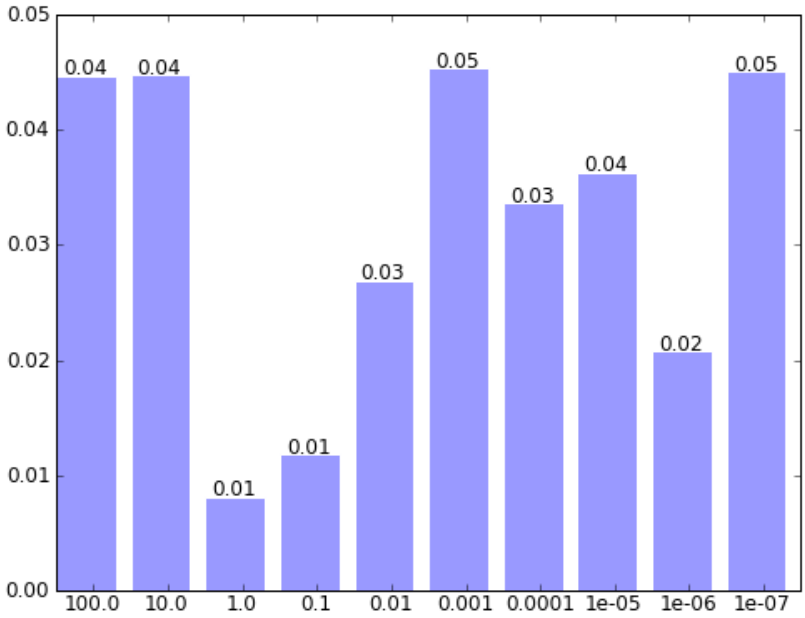
And finally, to allow direct comparison to the perceptron algorithm, we have 1000 samples with the same graphing algorithm:



Adaline with 1000 samples, using the same graphing algorithm.

Optimal values for the adjustment constant

No comment was made for the value of η . A simple experiment can be performed to experimentally determine the optimal value. Let us generate 10000 samples for plenty of work. Iterate only for 100 and then average the final in sample error over 10 runs like this:



Average final error for different values of η .

The experimental result, between 1 and 0.1 agree with the book's Authors' optimal value.

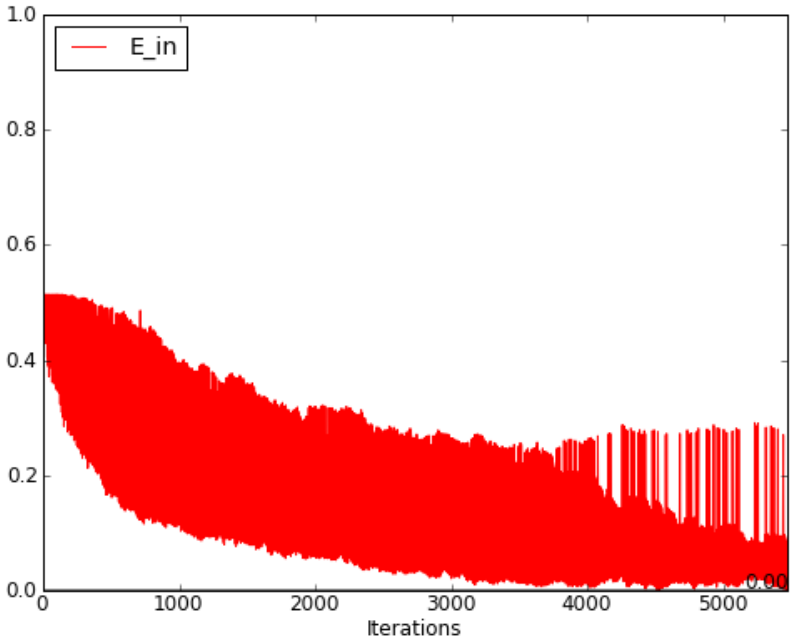
First test on real data

Definition of the task

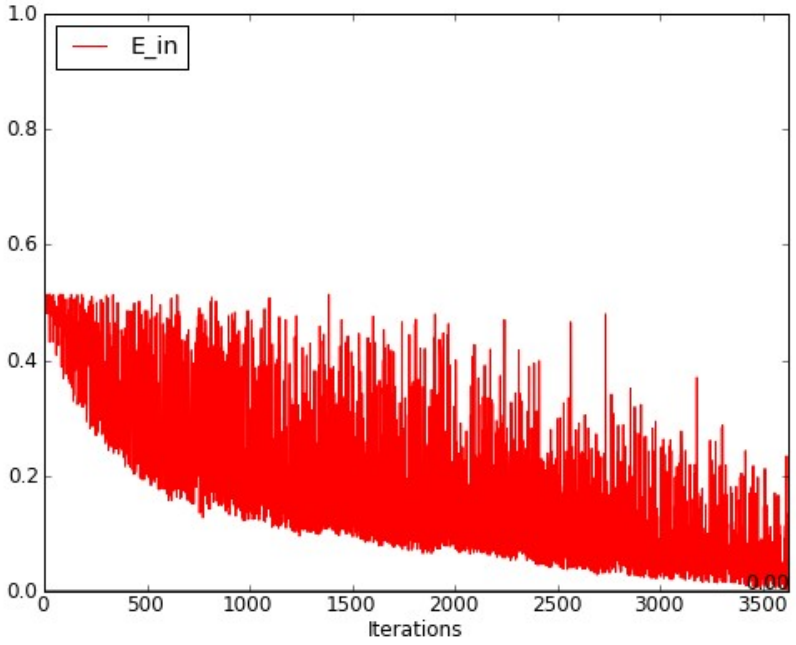
The course project has provided three files for learning. *Traininput.txt* contains $X=\mathbb{R}^d$ with $d=1050$ and *trainlabels.txt* $y\in\{-1,1\}$. The file *Testinput.txt* has a corresponding data set, but no labels. It is supposed to act as the final test of our algorithm, with the algorithm generating the output file *Testoutput.txt*.

Running the perceptron and adaline on real data

First we shall use a naïve approach in purely iterating to zero in sample error to see if the data is linearly seperable. This is just to test our algorithms work. Everything is as before, with the in sample error defined by $E_{\text{in}}(h)=\frac{1}{N}\sum_{n=1}^{N}[\![h(\vec{x}_n)\neq f(\vec{y}_n)]\!]$. There is no iteration limit, and to produce these nice graphs the whole error is very inefficently computed at every iteration.



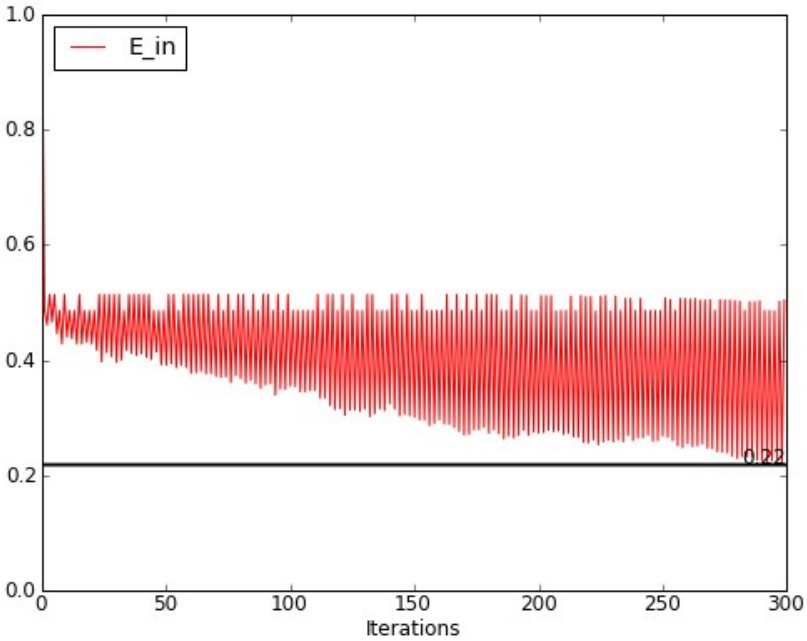
Perceptron on the real data achieves zero error at 5000+ iterations.



Adaline does better but not by a lot, getting to zero error at 3250+.

Adding a pocket to the perceptron

Since the error seems to very reliably oscillate within 0.2 points of the previous error, sometimes being 100% more than the previous error itself, it seems wise to have a good result stored in case shit hits the fan and we run out of computing time. A pocket perceptron checks to see if the error achieved with a new $\vec{w}(t)$ is better than any previously encountered error, and if so, it saves the $\vec{w}(t)$ that gave us this result. This is a very simple addition, and we can visualize it by adding a horizontal line to highlight the best known minimized error. With the same data:



Pocket perceptron set to stop at 300 iterations.

Going general with gradient descent methods

Principles of generalization

Fixed learning rate gradient descent is defined by the following:

1. Initialize the weights at time $t=0$ to $\vec{w}(0)$
2. **for** t = 0,1,2,... **do**
3. Compute the gradient $g_t=\nabla E_{\text{in}}(\vec{w}(t))$
4. Set the direction to move, $\vec{v}_t=-\vec{g}_t$
5. Update the weights: $\vec{w}(t+1)=\vec{w}(t)+\eta\vec{v}_t$
6. Iterate to the next step until it is time to stop
7. **endfor**
8. Return the final weights

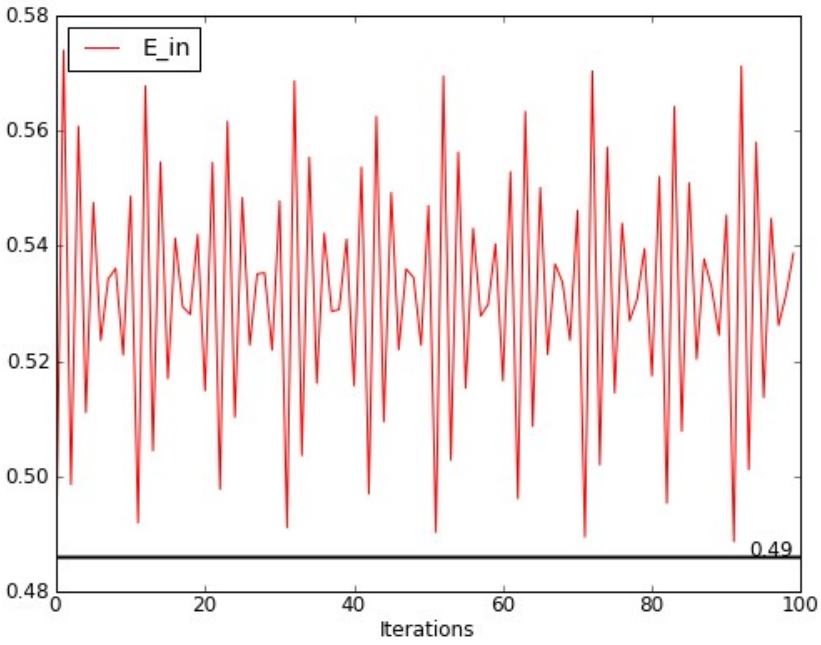
By defining $E_n(\vec{w})=[\![sign(\vec{w}^{\mathrm{T}}\vec{x}_n)\neq y_n]\!]$ (**Problem 3.5**) one can quickly show derive

$$\nabla E_{\text{in}}=\begin{cases}-\vec{x}_n & y-\vec{w}^{\mathrm{T}}\vec{x}>0\wedge sign(\vec{w}^{\mathrm{T}}\vec{x})\neq y_n\\ \vec{x}_n & y-\vec{w}^{\mathrm{T}}\vec{x}<0\wedge sign(\vec{w}^{\mathrm{T}}\vec{x})\neq y_n\\ 0 & sign(\vec{w}^{\mathrm{T}}\vec{x})=y_n\end{cases} \text{ when } .$$

Also it is easily verified that $E_n(\vec{w})\geq[\![sign(\vec{w}^{\mathrm{T}}\vec{x}_n)\neq y_n]\!]\forall n$ and therefore $\frac{1}{N}\sum_{n=1}^{N}E_n(\vec{w})$ is an upper boung for the in sample classification error $E_{\text{in}}(\vec{w}(t))$. Applying the above algorithm to $\frac{1}{N}\sum_{n=1}^{N}E_n(\vec{w})$ gives us a gradient descent algorithm that suspiciously looks like a version of a perceptron.

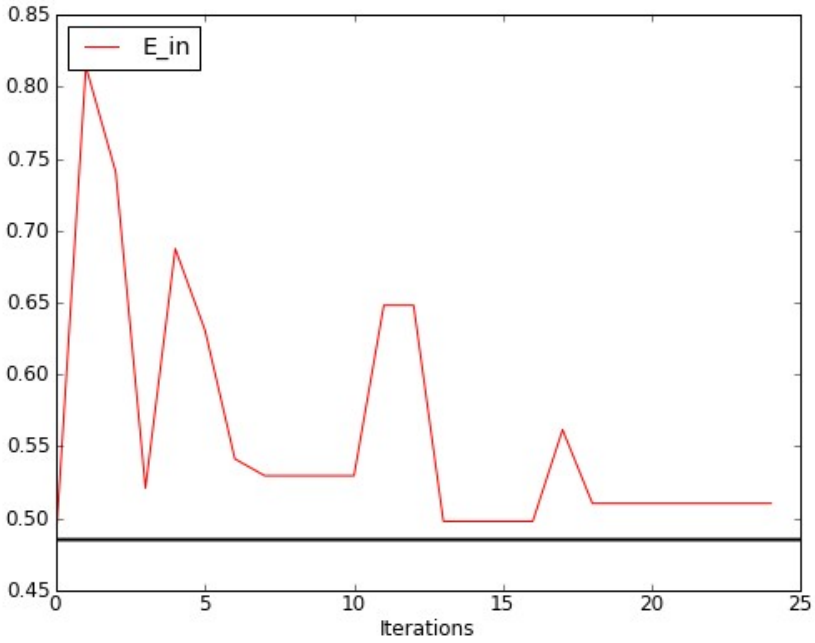
Batch gradient descent

Batch gradient descent is based on the idea of simultaneously taking into account every point in X . Algorithm that implemenets the above reasoning with runs on the data produce graphs that look like the hearth rate of an arrhythmic reptile. By tinkering with the constant in the fixed learning rate algorithm, we can get this oscillation to be large or small. Of course the problem lies in getting stuck in a local minimum. Relying on the slope of greatest descent promised benefits of a very quick convergence, but the setbacks of having many local minima for a non-convex function seem to render it unusable.



Batch gradient descent stuck very quickly into a local minima.

To fix this, one could try introducing some randomness in where to go, allowing bad directions as well. When we reduce this randomness and the probability of bad directions over time, we get a widely used method known as simulated annealing in the field of artificial intelligence. However with this many dimensions picking a random direction still results very quickly in local minima, as can be seen from the following graph:

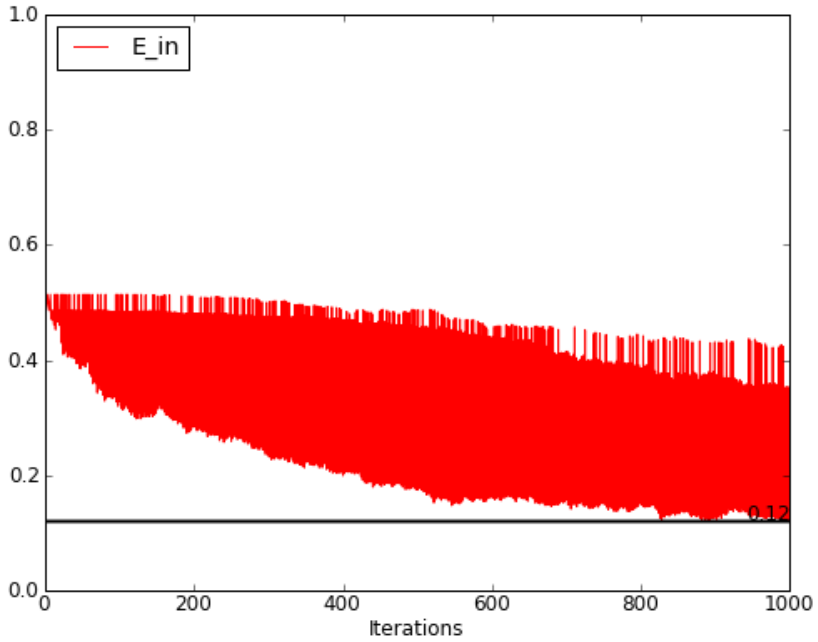


Simulated annealing gets stuck quickly again.

While it was a nice idea, at least for this data set I couldn't get it to work.

Stochastic gradient descent

To combat the problems arrising with many features, we change the gradient descent algorithm into stochastic gradient descent by considering one random point $\vec{x}_n$ at a time instead of the whole matrix X of all $\vec{x}_i$ as rows.



Looking familiar? Perceptron makes the reappearance.

In fact, this is just the perceptron algorithm re-expressed with fancier looking error functions.

Regression and validation on the real data

Motivation

While the perceptron achieving an in-sample error of 0.0 seems comforting, if we set aside a validation set of 200 samples and run the percepton learning algorithm on the 800 samples left, we get a stable validation error of 0.25. That is, 25% of the samples are misclassified. This isn't very suprising as looking at the perceptron convergence graph while lower end of the error oscillations is at zero, the higher is at 0.25. This means a correction of classification at one point, or equivalently changing that point, can result in such a jump.

To fix this problem, we define two sets: Y_{in} and Y_{val} such that $Y_{\text{in}} \cup Y_{\text{val}} = Y$. In the algorithm we choose 20% of the samples to be in Y_{val} as per the book recommendation. To proceed, we shall construct a complete function $\vec{w}^*$ like before that achieves $E_{\text{in}} = 0$ in Y_{in} . From this function we generate n functions $\{\vec{w}_1, \ldots, \vec{w}_n\}$ we are define different *hard order constraints* on the function. For example, one option is $\vec{w}_i = \{w | w \in \mathbb{R}^i\}$. In our sample it is is simpler to set the weights to zero.

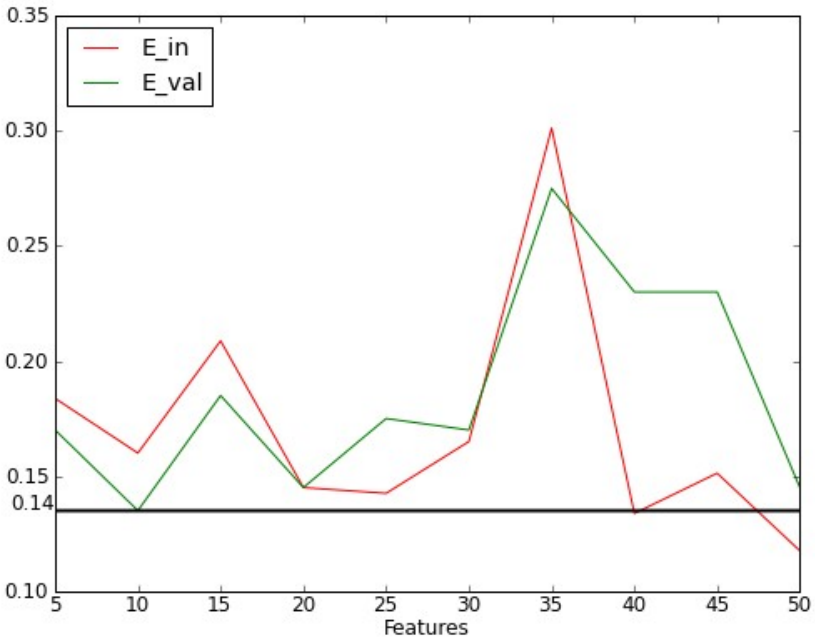
Because going trough the $\binom{|d|}{i}$ where $d = len(x)$ different non-zero weights for each function

$\vec{w}_i$ in Y_{in} to get the best one to evaluate in Y_{val} is inpractical, I decided to evaluate with just the one function $\vec{w}^*$, and then select the particular i non-zero weights $\psi=\{w_1,\dots,w_i\}, w_i\in\mathbb{N}, w_i\leq d$ of this function for each $\vec{w}_i$ for which $|\vec{w}^*[j]|\leq|\vec{w}^*[i]|\forall\, j\notin\psi, i\in\psi$. In other words, my hypothesis is that the largest absolute weights will have more influece to E_{in} than the small weights. We will take i weights as non-zero beginning from the greatest absolute weight to i 'th greatest and then re-evaluate in Y_{in} to get the actual $H=\{\vec{w}_1,\dots,\vec{w}_n\}$ for calculating E_{val}. Alternative scheme would be to classify a distance measure that measures the distance of each weight from other weights that does not rely on the real number system. Word of which below.

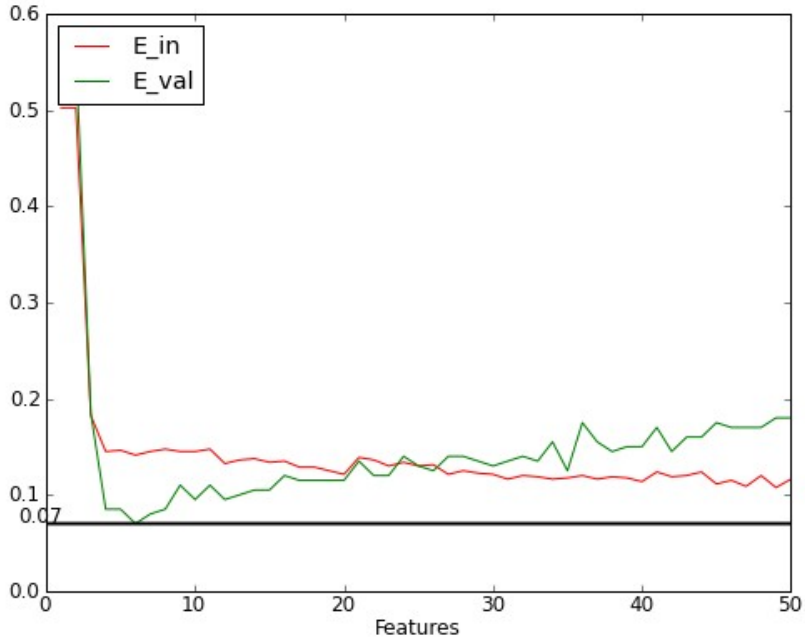
Soft order constraints are useless for linear classification. To see why consider $\lambda\vec{w}^{\mathrm{T}}\vec{w}\leq C$ now obviously $\vec{w}^{\mathrm{T}}\vec{x}=0$ is the same line as $\alpha\vec{w}^{\mathrm{T}}\vec{x}=0$. So taking $\vec{w}_2\leftarrow\alpha\vec{w}$ we have $\lambda\alpha^2\vec{w}^{\mathrm{T}}\vec{w}\leq C$ and the constraint is satisfied by scaling down α with the same function.

Picking the final hypothesis for the real data

Now that we have explained the idea, we shall pick the $\vec{w}_i$ by iterating over i features as described. Experiments indicate that over 50 features ($i\geq 50$) bring no actual benefit to the validation set, making the value of E_{val} quickly converge to 0.5 for up to $i\geq 1050$. Because generating the actual $\vec{w}_i$ for graphing purposes takes a long time, below I have plotted functions for $i\in\{5,10,\dots 50\}$. From the below graph algorithm chooses 10 features as optimal (though maybe 20-30 is better). This gives us a final $\vec{w}^*=(-28.98, 29.1, 29.15, \dots, 164.6, -172.45)$ for example. With models that are more sophisticated and less intuition relying with regards to the significance of the absolute weights it might just be that another -0.1 in the estimated E_{out} is possible. Here of course one should remember that the validation set is slightly biased.



Picking the correct hypothesis using the validation set with steps of 5



Here is a nice illustration of the relationship between E_{in} and E_{val} as features increase.

The final algorithm

Last but not least here is the full source code for the final learning algorithm. It automatically reads the appropriate files and outputs the presumably missing *Testoutput.txt file*.

```
'''
       Stochastic Gradient Descent Experiments
                     Machine learning 2013
                            Markus Viljanen
'''

from pylab import *
from numpy  import *


#==========    READ X ==========

filename = 'traininput.txt'
print "Reading X from", filename, "..."
f = open(filename, 'r')

X = None
first = True

for line in f:
```

```
        x = array(line.split(), dtype='float64')

        if first:
                print "x.dtype: ", x.dtype
                features = len(line.split())
                X = array([zeros(features)])
                X[0]=x
                first = False
        else:
                X = append(X,[x], axis = 0)

print "x.features: ", features
print "X.shape: ", X.shape

f.close()


#==========    READ Y ==========

filename = 'trainlabels.txt'
print "Reading Y from", filename, "..."
f = open(filename, 'r')

Y = array([])
first = True

for line in f:

        y = array(line, dtype='int32')
        Y = append(Y,y)

print "Y.shape: ", Y.shape

f.close()


#==========    Linear regression ==========
#print "=========START=========="

#print "Evaluating linear regression..."

#Xc = dot( inv( dot(transpose(X), X) ), transpose(X))
#Wlin = dot( Xc, Y )
```

```
#print X.shape
#print Wlin.shape
#diff = (sign(dot(X,Wlin)) != Y)
#error = E(diff)
#print "E_in:", error

#==========    Perceptron ==========

def E(diff,N):
        return float(sum( diff ))/N

def E2(W,X,Y,N):
        return dot((sign(dot(X,W)) != Y),ones(N))/N

def DE2(W,x,y,N):
        if (sign(dot(x,W)) != y):
                if ((dot(x,W)-y) >= 0):
                        return -x
                else:
                        return x
        else:
                return zeros(len(x))

print "========= FIND w- ========="

Wd=None

for f in [len(X[0])]:
        iterations = 100000
        W = zeros(len(X[0]))
        TW = W[:f]
        TX = X[:800,:f]
        TY = Y[:800]

        print "Using", f,"features ..."
        data = len(TX)
        features = len(TX[0])

        for i in range(iterations):

                r = random.randint(data)
                TW = TW+DE2(TW,TX[r,:],TY[r],data)

        print "Error:", E2(TW,TX,TY,data)
        Wd = TW
```

```
print "========= Model selection (w-,H) ========="

start = 50
best_f = start
best_w = Wd
best_validation = 1.0
best_take = 0

Fs = []
Errors = []
Validations = []

for f in [x for x in range(start,0,-1)]:
        iterations = 1000
        take = argsort(abs(Wd))[-f:]
        #print take
        TW = Wd[take]
        TX = X[:800,take]
        TY = Y[:800]

        ZX = X[800:,take]
        ZY = Y[800:]

        print "Using", f,"features ..."
        data = len(TX)
        features = len(TX[0])

        for i in range(iterations):

                r = random.randint(data)
                TW = TW+DE2(TW,TX[r,:],TY[r],data)

        Error = E2(TW,TX,TY,data)
        Validation = E2(TW,ZX,ZY,len(ZX))
        print "Error:", Error
        print "Validation:", Validation
        Fs.append(f)
        Errors.append(Error)
        Validations.append(Validation)

        if Validation < best_validation:
                best_f = f
                best_take = take
                best_validation = Validation
```

```
print "========= Final results ========="
print "f:", best_f
print "W*:", Wd[best_take]
print "Validation:", best_validation



#==========    Read Z ==========

filename = 'testinput.txt'
print "Reading Z from", filename, "..."
f = open(filename, 'r')

Z = None
first = True
features = 0

for line in f:

        z = array(line.split(), dtype='float64')

        if first:
                print "z.dtype: ", z.dtype
                features = len(line.split())
                Z = array([zeros(features)])
                Z[0]=z
                first = False
        else:
                Z = append(Z,[z], axis = 0)

print "z.features: ", features
print "Z.shape: ", Z.shape

f.close()

#==========    Calculate output ==========

W = zeros(features)
W[best_take] = Wd[best_take]
Y =  sign(dot(Z, W))
print "Calculated output:", Y

#==========    Read Z ==========
```

```
filename = 'testoutput.txt'
print "Writing Y = Zw* to", filename, "..."
f = open(filename, 'w')

for y in Y:
        f.write(str(int(y))+"\n")
f.close()
print len(Y), "entries written."
print "All done!"

fig = figure(1)

ax1 = fig.add_subplot(111)

ax1.plot(Fs, Errors, color='red', label="E_in")
ax1.plot(Fs, Validations, color='green', label="E_val")
ax1.legend(loc='upper left')
ax1.axhline(best_validation, color='black', linewidth=2.0)
ax1.text(Fs[-1], best_validation, '%.2f' % best_validation, ha='right', va= 'bottom')

ax1.set_xlabel('Features')

savefig("features.png",dpi=72)
```