Fundamentals in Statistical Pattern Recognition

Lab 2: Logistic Regression with Python and BOB

Dr André Anjos
Dr Sébastien Marcel

www.idiap.ch/~marcel

Idiap Research Institute
Martigny, Switzerland

May 22, 2013
Package for Lab 2

Please make sure you download and unzip:

http://www.idiap.ch/~marcel/lectures/lectures/epfl2013/FSPR_lab2.zip

Instructions:

$ mkdir ~/work
$ cd ~/work
$ wget <URL as above>
$ unzip <Package name as above>.zip
$ cd lab2
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
Most of Python magic lives in outside modules. You need to import modules to use their functionality:

```python
>>> import os
```

If you don’t know it, ask for help:

```python
>>> help(os)
```

To access an object inside another module (or other object), use the . (dot) operator:

```python
>>> help(os.listdir)
```
Explore: Use TAB completion

```python
>>> lst = [7, 3., 'string']
>>> lst.<TAB>
```

Remember

In doubt call help, e.g., help(lst.remove)

More information

http://docs.python.org/2/library/stdtypes.html
Always start an outside editor (e.g. gedit) and a prompt you can use for consultation. You can optionally have a second prompt you use.
Loops with for

To create a for loop, use the for keyword:

```python
>>> for <variable> in <iterable> :
...     <statement>
...     <statement>
...     [RETURN] #marks end of block
```

Try this:

```python
>>> for k in [1, 2, 3.14, 'bla']:
...     print k
...     [RETURN] #marks end of block
1
2
3.14
bla
```

Formal definition

http://docs.python.org/2/reference/compound_stmts.html#the-for-statement
Python: NumPy and ndarrays

Creating arrays:

```python
>>> import numpy
>>> help(numpy.ndarray)
>>> # basics: numpy.ndarray(<SHAPE>, <DATA-TYPE (dtype)>)
>>> X = numpy.ndarray((2,2), dtype=numpy.uint8)
>>> print X
[[168 247]
 [ 20 161]] #notice: uninitialized
>>> X = numpy.zeros((2,2), dtype=numpy.float64)
>>> print X
[[ 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00]]
>>> # other interesting initializers
>>> help(numpy.ones) #equivalent to Matlab’s
>>> help(numpy.array) #initializes from almost any object!
```

More information

NumPy: Vectorization

- Disclaimer: This is an abuse of the term in most cases
  - It means "parallelize" normally.
- What we mean is: *use pre-compiled routines to operate on matrices and vectors so as to speed-up computation*
- This is mostly deployed for linear algebra operations.
  - Examples:
    - Dot products between vectors, vectors and matrices or matrices;
    - Linear system solving;
    - Value Decomposition;
    - etc.
- Also possible to speed-up simple mathematical operations, such as "square all values of this array".
Recap LR Minimization LR+LBFGS Multi-class Regularization Challenge

NumPy: Vectorization

Remember

Array operations are applied element-wise.

That means:

```python
>>> t = numpy.array(numpy.arange(4))
>>> t
array([0, 1, 2, 3])
>>> t*t
array([0, 1, 4, 9])
```

# use the function numpy.dot() for dot-products
```python
>>> numpy.dot(t, t)
14
```

# if both objects are 1D, numpy will do what you expect
```python
>>> col = t.reshape(4,1)
```

# if one of the is a column vector, it may go wrong
```python
>>> numpy.dot(t, col)  # ok
array([[14]])
>>> numpy.dot(col, t)  # not ok
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
```
NumPy: Vectorization

Interesting operators and linear algebra:

```python
>>> m = numpy.array(numpy.arange(4)).reshape(2,2)
>>> m
array([[0, 1],
       [2, 3]])

>>> m.transpose()
array([[0, 2],
       [1, 3]])

>>> numpy.linalg.inv(m)
array([[-1.5,  0.5],
       [ 1. ,  0. ]])  #notice automatic type casting
```

More information

- `help(numpy.linalg)
Matlab Gotchas

- Array operations are applied element-wise. Example: `a * b` means multiply `a` with `b` elementwise. Use the `numpy.matrix` class if you want the default to mean *matrix multiplication* (not advised though);
  - Element Broadcasting
- 0-based indexing (instead of 1-based);
- slicing gives a reference to existing data, **not** a copy.

More information

http://www.scipy.org/NumPy_for_Matlab_Users
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
**Hypothesis**

\[ h_\theta(x) = g(\theta^T x), \text{ with} \]

\[ \theta^T = [\theta_0 \quad \theta_1 \quad \theta_2 \quad \cdots \quad \theta_d], \text{ and} \]

\[ g(x) = \frac{1}{1 + e^{-x}} \]

**(Convex) Cost function** \( J(\theta) \)

\[ J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] \]

**Cost Derivative**
The Iris Flower Database (1/3)

Task: Discriminate between 3 types of Iris Flowers

Examples (setosa, versicolor, virgnica)

Reference

The Iris Flower Database (2/3)

There are 50 examples of each class. 4 variables are described: Petal length and width, Sepal length and width (plen, pwid, slen, swid)
The “Setosa” variant is linearly separable from the other two.
Your Tasks

Implement (all in \textit{vectorized} form):

1. the LR hypothesis: $H(X, \theta)$
2. the cost function: $J(X, y, \theta)$
3. the cost derivative: $dJ(\theta)/d\theta$
4. a function to compute the overall classification error
Multivariate Logistic Regression with Gradient Descent

Instructions:

```bash
$ cd ex1  # change into directory for the exercise
$ ./ex1.py -v plen pwid -- setosa versicolor
$ gedit ex1_lab.py  # open this module
```

Let's spice it up

In case you need testing, you have to implement it yourself!

```bash
$ gedit mytest.py
```

```python
#!/usr/bin/env python
import numpy
import ex1_lab
X = numpy.array([[1.0, 1.0], [2.0, 2.0]])
theta = numpy.ones([0.0, 1.0, 1.0])
print ex1_lab.h(X, theta)
```

```bash
$ chmod 755 ./mytest.py
$ ./mytest.py
```

$ 19/70
Task 1: LR Hypothesis

\[ h_\theta(x) = g(\theta^T x), \text{ with} \]
\[ \theta^T = [\theta_0, \theta_1, \theta_2, \ldots, \theta_d], \text{ and} \]
\[ g(x) = \frac{1}{1 + e^{-x}} \]

Test cases

<table>
<thead>
<tr>
<th>( \theta^T )</th>
<th>( X )</th>
<th>( \theta^T X )</th>
<th>( g(\theta^T X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(1.0, 1.0)^T</td>
<td>2.0</td>
<td>( \sim 0.8808 )</td>
</tr>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(2.0, 2.0)^T</td>
<td>4.0</td>
<td>( \sim 0.9820 )</td>
</tr>
<tr>
<td>(0.0, 0.5, 0.5)</td>
<td>(3.0, 3.0)^T</td>
<td>3.0</td>
<td>( \sim 0.9526 )</td>
</tr>
</tbody>
</table>
Multivariate Logistic Regression with Gradient Descent

Task 1: LR Hypothesis

\[ h_\theta(x) = g(\theta^T x), \text{ with} \]
\[ \theta^T = [\theta_0 \ \theta_1 \ \theta_2 \ \cdots \ \theta_d], \text{ and} \]
\[ g(x) = \frac{1}{1 + e^{-x}} \]

solution:

```python
return 1. / (1 + numpy.exp(-numpy.dot(Xp, theta)))
```
Multivariate Logistic Regression with Gradient Descent

Task 2: LR Cost

\[ J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] \]

Test cases

<table>
<thead>
<tr>
<th>\theta^T</th>
<th>X</th>
<th>y</th>
<th>J(\theta, X, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(1.0, 1.0)^T</td>
<td>1.0</td>
<td>~0.1269</td>
</tr>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(2.0, 2.0)^T</td>
<td>1.0</td>
<td>~0.0181</td>
</tr>
<tr>
<td>(0.0, 0.5, 0.5)</td>
<td>(3.0, 3.0)^T</td>
<td>0.0</td>
<td>~3.0486</td>
</tr>
</tbody>
</table>

First line of cost on ex1.py:

\[ J_{\text{train}} = 6.93147181e-01 \mid J_{\text{devel}} = 6.931472e-01 \]
Multivariate Logistic Regression with Gradient Descent

Task 2: LR Cost

\[ J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] \]

Solution

\[
\begin{align*}
  h_\_ &= h(X, \theta) \\
  \text{logh} &= \text{numpy.nan_to_num}(\text{numpy.log}(h_\_)) \\
  \text{log1h} &= \text{numpy.nan_to_num}(\text{numpy.log}(1 - h_\_)) \\
  \text{return} &= -(y*\text{logh} + ((1-y)*\text{log1h})).\text{mean()}
\end{align*}
\]

Tip

To avoid NaN and ±Inf, use numpy.nan_to_num, to convert NaN to 0 and ±Inf into very large floating point numbers.
Multivariate Logistic Regression with Gradient Descent

Task 3: LR Cost Derivative

\[
\frac{dJ}{d\theta} = \frac{1}{n} \sum_{i=1}^{n} (h_{\theta}(x_i) - y_i)x_{i,j} \forall j = 0..d
\]

Test cases

<table>
<thead>
<tr>
<th>$\theta^T$</th>
<th>$X$</th>
<th>$y$</th>
<th>$dJ(\theta, X, y)/d\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(1.0, 1.0)$^T$</td>
<td>1.0</td>
<td>$[-0.1192, -0.1192, -0.1192]$</td>
</tr>
<tr>
<td>(0.0, 1.0, 1.0)</td>
<td>(2.0, 2.0)$^T$</td>
<td>1.0</td>
<td>$[-0.0180, -0.0360, -0.0360]$</td>
</tr>
<tr>
<td>(0.0, 0.5, 0.5)</td>
<td>(3.0, 3.0)$^T$</td>
<td>0.0</td>
<td>$[0.9526, 2.8577, 2.8577]$</td>
</tr>
</tbody>
</table>

Second line of cost on ex1.py:

```
J_train = 7.03311660e-01 | J_devel = 7.152138e-01
```
Task 3: LR Cost Derivative

\[
\frac{dJ}{d\theta} = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j} \quad \forall j = 0..d
\]

Solution

```python
return ((h(X, theta) - y) * Xp.T).T.mean(axis=0)
```

Tip

`ndarray.T` is the same as `ndarray.transpose()`
NumPy Arrays: Advanced Indexing

Advanced indexing in NumPy:

```python
>>> import numpy
>>> h = numpy.array([0.5, 0.1, 0.2, 0.9, 0.3, 0.7])
>>> h < 0.5
array([False,  True,  True, False,  True, False], dtype=bool)
>>> z = h[h < 0.5]
>>> z
array([ 0.1, 0.2, 0.3])
>>> z[0] = 0.8
>>> h
array([ 0.5, 0.1, 0.2, 0.9, 0.3, 0.7]) #N.B.: unchanged!
```
Multivariate Logistic Regression with Gradient Descent

Task 4: Classification Error Rate (CER)

Definitions:

False Negative (FN): \( h_\theta(x) < 0.5, \text{ but } y = 1 \)
False Positive (FP): \( h_\theta(x) \geq 0.5, \text{ but } y = 0 \)

Classification Error: \( CE = FP + FN \)
Classification Error Rate: \( CER = \frac{FP+FN}{N} \)

\( N \) is the total number of examples

Expected final output:

000/100: \( J_{\text{train}} = 6.93147181e-01 \) (50%)
001/100: \( J_{\text{train}} = 7.03311660e-01 \) (50%)
002/100: \( J_{\text{train}} = 5.21725342e-01 \) (45%)
003/100: \( J_{\text{train}} = 4.89047755e-01 \) (50%)
004/100: \( J_{\text{train}} = 4.52491010e-01 \) (0%)
Multivariate Logistic Regression with Gradient Descent

Task 4: Classification Error Rate (CER)

Definitions:

- False Negative (FN): \( h_\theta(x) < 0.5 \), but \( y = 1 \)
- False Positive (FP): \( h_\theta(x) \geq 0.5 \), but \( y = 0 \)

Classification Error: \( CE = FP + FN \)
Classification Error Rate: \( CER = \frac{FP + FN}{N} \)
\( N \) is the total number of examples

Solution (uses advanced indexing):

```python
h_ = h(X, theta)
h_[h_<0.5] = 0.0
h_[h_>=0.5] = 1.0
errors = (h_ != y).sum()
return float(errors)/len(X)
```
Do more!

1. Activate plotting using the `--plot` option;
2. Change the input variables and their number (find the best configuration);
3. Play with the number of iterations and the early stop criteria (how is it implemented?);
4. Activate normalization and checkout what happens with the learning rate. Does it oscillate more than before? Why?
5. Increase the polynomial degree using `--polynomial-degree`. What happened to the decision boundary when you set it to 2 or 3? Is that still true for other variable combinations?
6. Try to separate different classes. For example: virginica and versicolor. Is it easier or more difficult? Why?
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
The learning rate $\alpha$

A better $\alpha$

- What is the best value?
- If you had to guess, how would you set it?
The learning rate $\alpha$

A better $\alpha$

- What is the best value?
- If you had to guess, how would you set it?

As you can probably imagine...

That depends on the surface’s curvature change rate!
The learning rate $\alpha$

- **Objective:** find the best set of parameters $\theta$ for a given surface $J(\theta)$ in which $J$ is as small as possible;
- **Constraints:**
  - $J$ convexity: strict, strong
  - differentiability (once? twice?)
  - dependence on initial guess for $\theta$
Minimization

Depending on your constraints, you can resort to different methods.

1. Newton’s Method: use $d^2 J / d\theta^2$ (Hessian) to determine learning rate (Constraint: need double differentiability);

2. L-BFGS (Limited memory Broyden-Fletcher-Goldfarb-Shanno): approximates Hessian computation. (Constraint: need to be differentiable);

3. Others: Nelder-Mead, Conjugate Gradient, etc;

4. Brute force: Simulated Annealing;

References

http://en.wikipedia.org/wiki/Mathematical_optimization
http://en.wikipedia.org/wiki/Newton%27s_method_in_optimization
http://en.wikipedia.org/wiki/BFGS_method
Minimization using SciPy

Using L-BFGS in SciPy:

```python
import scipy.optimize
theta, cost, d = scipy.optimize.fmin_l_bfgs_b(J, theta0, dJ)
# theta: optimized theta
# cost : the cost at the optimal theta
# d    : a python dictionary with extra information
```

You must define J and dJ so they work like this:

```python
def J(theta, X, y):
    # return "old" J(X, y, theta)

def dJ(theta, X, y):
    # returns "old" dJ(X, y, theta)
```

References

http://docs.scipy.org/doc/scipy/reference/optimize.html
Minimization using SciPy

Clean solution: Python classes

class MyClass:

def __init__(self, par1, par2, par3):
    self.par1 = par1
    self.par2 = par2
    self.par3 = par3

def doSomethingWithoutParameters(self):
    print par1, par2, par3

def doAnotherThingWithParameters(self, par4):
    self.doSomethingWithoutParameters()
    print par4

References

http://docs.python.org/2/tutorial/classes.html
Let’s apply that to our problem, so it conforms to the optimization API:

```python
class Trainer:
    def __init__(self, normalize=True):
        self.normalize = normalize

    def J(self, theta, X, y):
        # returns J(X, y, theta)

    def dJ(self, theta, X, y):
        # return dJ(X, y, theta)

    def train(self, X, y):
        from scipy.optimize import fmin_l_bfgs_b
        theta, cost, d = fmin_l_bfgs_b(self.J, theta0, self.dJ, (X, y))
        return theta
```
This is **Machine** Learning right?

class Machine:

    def __init__(self, theta):
        self.theta = theta

    def __call__(self, X):
        # applies the hypothesis given the input

    def J(self, X, y):
        # calculates the cost

    def dJ(self, X, y):
        # calculates the cost derivative

    def CER(self, X, y):
        # calculates the classification error rate
class Trainer:

    def __init__(self, normalize=True):
        self.normalize = normalize

    def J(self, theta, machine, X, y):
        machine.theta = theta
        return machine.J(X, y)

    def dJ(self, theta, machine, X, y):
        machine.theta = theta
        return machine.dJ(X, y)

    def train(self, X, y):
        from scipy.optimize import fmin_l_bfgs_b
        machine = machine(theta0)
        theta, cost, d = fmin_l_bfgs_b(self.J, theta0, self.dJ, (machine, X, y))
        machine.theta = theta
        return machine
Loading and Saving Machine states

In your normal workflow, you normally don’t train and test in a single program. You train in one program and you test in another:

- Reproduce results;
- Avoid malfunctioning.

Saving the Machine State

Use *Hierarchical Data Format* (HDF, version 5 - or just HDF5).

- Supports different data types
- Supported by all major software frameworks: Python, R, S-plus, Octave, Matlab, etc.

Reference

http://www.hdfgroup.org/HDF5
Loading and Saving Machine states

Use Bob’s interface to HDF5 loading and saving:

```python
>>> import bob
>>> import numpy

# Open new file for writing, truncate it
>>> f = bob.io.HDF5File('test.hdf5', 'w')

# Open new file for writing, truncate it
>>> t = numpy.array(numpy.arange(25.)).reshape(5,5)

# Open new file for writing, truncate it
>>> f.set("myvar", t)

# Open new file for writing, truncate it
>>> del f # flush file

# Open new file for writing, truncate it
>>> f = bob.io.HDF5File('test.hdf5', 'r')

# Open new file for writing, truncate it
>>> f.paths() # List all variables and folders inside file
["/myvar"]

# Open new file for writing, truncate it
>>> f.read("myvar") # Loads it back
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24.]])
```
External Utilities for HDF5 files

List contents: h5ls

$ h5ls test.hdf5
myvar Dataset {5, 5}

Dump contents and explain: h5dump

$ h5dump test.hdf5
HDF5 "test.hdf5" {
GROUP "/" {
  DATASET "myvar" {
    DATATYPE H5T_IEEE_F64LE
    DATASPACE SIMPLE { ( 5, 5 ) / ( 5, 5 ) }
    DATA {
      ...
  }
}

Find differences in files and objects: h5diff

$ h5diff test.hdf5 test.hdf5
Other things you can do

HDF5 files in Bob are not only used for Machine loading and saving:

- Saving and loading temporary images, audio or features;
- Exchanging data with other software frameworks (e.g. Matlab);
- Saving multiple machines and settings so we represent a complex setup (see it in action at ex3).
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
Your Tasks

1. Fill-in the Machine class with the missing bits
Instructions:

$ cd ex2  # change into directory for the exercise
$ ./ex2.py -v plen pwid -- setosa versicolor
$ gedit ex2_lab.py  # open this module

Tips

- Inside the class scope, self == this in C++;
- Read the documentation of each function before filling it in;
- Use the methods set and read to read and load from the HDF5 file
Logistic Regression with LBFGS, Machines and Trainers

Expected output:

$ ./ex2.py -v plen pwid -- setosa versicolor
Predicting Iris Flower classes using Multivariate ...
Loading data...
Splitting data into training, development and test sets...
Number of training examples loaded : 40
Number of development examples loaded : 40
Number of test examples loaded : 20
Classes selected : setosa x versicolor
Variables selected : plen, pwid
Polynomial degree : 1
Settings:
  * initial guess = [0.0, 0.0, 0.0]
  * cost (J) = 0.693147
  * CER = 50%
Training using scipy.optimize.fmin_l_bfgs_b()...
** LBFGS converged successfully **
Final settings:
  * theta = [-29.381094091206364, 7.9883693629508308,...
  * cost (J) = 3.59013e-06
Solution (1/2):

```python
def __call__(self, X, pre_normed=False):
    Xnorm = X if pre_normed else self.normalize(X)
    return 1. / (1. + numpy.exp(-numpy.dot(Xnorm, self.theta)))

def J(self, X, y, pre_normed=False):
    Xnorm = X if pre_normed else self.normalize(X)
    h = self(Xnorm, pre_normed=True)
    logh = numpy.nan_to_num(numpy.log(h))
    log1h = numpy.nan_to_num(numpy.log(1-h))
    return -(y*logh + ((1-y)*log1h)).mean()

def dJ(self, X, y, pre_normed=False):
    Xnorm = X if pre_normed else self.normalize(X)
    return ((self(Xnorm, pre_normed=True) - y) * Xnorm.T).T.mean(axis=0)
```
Logistic Regression with LBFGS, Machines and Trainers

Solution (2/2):

def CER(self, X, y, pre_normed=False):

    Xnorm = X if pre_normed else self.normalize(X)
    h = self(Xnorm, pre_normed=True)
    h[h<0.5] = 0.0
    h[h>=0.5] = 1.0
    errors = (h != y).sum()
    return float(errors)/len(Xnorm)

def save(self, h5f):

    h5f.set('theta', self.theta)
    h5f.set('subtract', self.norm[0])
    h5f.set('divide', self.norm[1])

def load(self, h5f):

    self.theta = h5f.read('theta')
    self.norm = h5f.read('subtract'), h5f.read('divide')
The `train()` method:

```python
def train(self, X, y):
    theta0 = numpy.zeros(X.shape[1])
    machine = Machine(theta0)
    if self.normalize:
        machine.set_norm(X)

    print 'Settings:'
    print ' * initial guess = %s' % ([k for k in theta0],)
    print ' * cost (J) = %g' % (machine.J(X, y),)
    print ' * CER = %g%%' % (100*machine.CER(X, y),)
    print 'Training using scipy.optimize.fmin_l_bfgs_b()...

    theta, cost, d = scipy.optimize.fmin_l_bfgs_b(
        self.J,
        theta0,
        self.dJ,
        (machine, X, y),
    )

    ...
```
Do more!

1. Make sure you understand what is happening on the `train()` method of the Trainer class;

2. Activate plotting using the `--plot` option;

3. Change the input variables and their number (find the best configuration);

4. Increase the polynomial degree using `--polynomial-degree`. What happened to the decision boundary when you set it to 2 or 3? Is that still true for other variable combinations? In particular, try this: `./ex2.py -p -d 2 -v plen swid -- virginica versicolor`;

5. Try this: `./ex2.py -p -d 5 -v slen pwid -- virginica versicolor` Does it make sense? Explain. Activate normalization now. Does it work better? What do you think happened?
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
Multi-class classification

Multi-class problem as multiple binary classification problems

1 vs 2+3
\[ h_{\theta(1)} \]

2 vs 1+3
\[ h_{\theta(2)} \]

3 vs 1+2
\[ h_{\theta(3)} \]

One vs all

\[
\text{class} = \arg \max_k h_{\theta(k)} = \arg \max_k P(y = k | x; \theta(k))
\]

with \( k = 1 .. 3 \)
Multi-class classification

Your Tasks

1. Create a new dataset (only targets) such that you can try individual classifiers for all 3 Iris Flower types;
2. Implement a prediction function based on \( \text{argmax} \);
3. Implement classification error as a confusion matrix.
Multi-class classification

Confusion Matrix (For our 3-class problem):

\[
CM = \begin{pmatrix} 
\frac{TP_0}{N_0} & \frac{FP_{(1 \rightarrow 0)}}{N_1} & \frac{FP_{(2 \rightarrow 0)}}{N_2} \\
\frac{FP_{(0 \rightarrow 1)}}{N_0} & \frac{TP_1}{N_1} & \frac{FP_{(2 \rightarrow 1)}}{N_2} \\
\frac{FP_{(0 \rightarrow 2)}}{N_0} & \frac{FP_{(1 \rightarrow 2)}}{N_1} & \frac{TP_2}{N_2}
\end{pmatrix}
\]

Where:

- \(TP_i/N_i\) is the True Positive Rate for class \(i\). That is, the number of correctly classified examples of class \(i\) by the classifier pool;
- \(FP_{(i \rightarrow j)}/N_i\) is the False Positive Rate for class \(j\) by the classifier pool from examples of class \(i\). That is, the number of elements of class \(i\) that were incorrectly classified as \(j\) by our classification system.
Task 1:

$ cd ex3  # change into directory for the exercise
$ ./ex3_train.py -v plen swid -- setosa
$ gedit ex3_lab.py  # fix make_subset()

Expected output:

Final settings:
* $\theta = [1.4042630947339283, -9.4307795910292125, 7.99609943...$
* $\text{cost (J)} = 2.54096e-06$
* $\text{CER} = 0$

Development set statistics:
* $\text{cost (j)} = 2.11435e-06$
* $\text{CER} = 0$
Multi-class classification

Task 1:

```bash
$ cd ex3  # change into directory for the exercise
$ ./ex3_train.py -v plen swid -- setosa
$ gedit ex3_lab.py  # fix make_subset()
```

Solution:

```python
def make_subset(y, cls):
    retval = numpy.zeros(len(y))
    retval[y == cls] = 1.
    return retval
```
Multi-class classification

Tasks 2 and 3 (decision and confusion matrix):

$ ./ex3_train.py --variable plen swid -- setosa
$ ./ex3_train.py --variable plen swid -- versicolor
$ ./ex3_train.py --variable plen swid -- virginica
$ ./ex3_evaluate.py setosa.hdf5 versicolor.hdf5 virginica.hdf5
$ gedit ex3_lab.py # fix confusion_matrix() and predict_class()

Expected output:

Confusion matrix for train set:
[[ 1. 0. 0. ]
 [ 0. 0.95 0. ]
 [ 0. 0.05 1. ]]

Confusion matrix for devel set:
[[ 1. 0. 0. ]
 [ 0. 0.9 0.15]
 [ 0. 0.1 0.85]]
Solution

```python
def predict_class(scores):
    return scores.argmax(axis=1)

def confusion_matrix(scores):
    retval = numpy.ones((3,3), dtype='float64')
    for real in range(3):
        col_scores = numpy.vstack([k[real] for k in scores]).T
        predictions = predict_class(col_scores)
        for predicted_as in range(3):
            retval[predicted_as, real] = len(predictions[predictions == predicted_as]) / float(len(predictions))
    return retval
```
Do more!

1. Activate plotting using the `--plot` option to visualize the confusion matrix;

2. Can you get a perfect scoring by having 100% on all main diagonals?
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
Regularization

Objective

Penalize very large values for $\theta$.
Avoid overflow.
Regularization

Regularized Cost function $J(\theta)$

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] + \frac{\lambda}{2} \sum_{j=2}^{d} \theta_{2j}^2$$

Regularized Derivative of Cost function $dJ(\theta)/d\theta$

$$\frac{dJ}{d\theta_0} = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j}$$

$$\frac{dJ}{d\theta_t} = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j} + \frac{\lambda \theta_t}{n}$$
Regularization

Your Tasks

1. Re-implement the cost $J$ to account for regularization;
2. Re-implement the cost $dJ/d\theta$ to account for regularization.
Task 1 & 2:

```bash
$ # continue at ex3 directory
$ ./ex3_train.py -d 2 -l 0.0001 --variable plen swid -- setosa
$ gedit logreg.py # fix Machine’s J() and dJ()
```

Expected output:

Final settings:

* `theta` = [0.073741150081068199, -0.17861442649719333, 0.31124102303627926, -1.114442687295276, -0.29665490666188432, 1.1993517280124333]

* cost (J) = 0.000345887
* CER = 0%

Development set statistics:

* cost (J) = 4.26815e-05
* CER = 0%
Task 1: Regularized Cost function $J(\theta)$

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] + \frac{\lambda}{2} \sum_{j=2}^{d} \theta_j^2$$

Task 2: Regularized Derivative of Cost function $dJ(\theta)/d\theta$

$$\frac{dJ}{d\theta_0} = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j}$$

$$\frac{dJ}{d\theta_t} = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j} + \frac{\lambda \theta_t}{n}$$
Task 1: Regularized Cost function $J(\theta)$

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))] + \frac{\lambda}{2} \sum_{j=2}^{d} \theta_d^2$$

Solution:

...  

```python
regularization_term = regularizer*(self.theta[1:])**2).sum()
main_term = -(y*logh + ((1-y)*log1h)).mean()
return main_term + regularization_term
```
Task 2: Regularized Derivative of Cost function $dJ(\theta)/d\theta$

\[
\begin{align*}
\frac{dJ}{d\theta_0} &= \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j} \\
\frac{dJ}{d\theta_t} &= \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{i,j} + \frac{\lambda \theta_t}{n}
\end{align*}
\]

Solution:

```python
... 
retval = ((self(Xnorm, pre_normed=True) - y) * Xnorm.T).T.mean(axis=0) 
retval[1:] += (regularizer*self.theta[1:])/len(Xnorm) 
return retval
```
Outline

Recap

Lab 2.1: Logistic Regression using Gradient Descent (Warm-up)

Cost Minimization using SciPy

Lab 2.2: Logistic Regression using L-BFGS

Lab 2.3: Multi-class Logistic Regression

Lab 2.4: Logistic Regression with Regularization

Lab 2.5: Multi-class Logistic Regression with Regularization (Challenge!)
And the award goes to...

**Challenge**

Find the best multi-class classifier (preferably with 100% on all main diagonals of the confusion matrix! Try the following:

- Use the infrastructure you have pre-coded for ex3;
- Modify the number of variables you use for classification;
- Modify the polynomial expansion degree;
- Play with the regularization term (be careful not to set it too high);
- Make your choice based on the results of your development set;
- Use the program ./ex3_evaluate.py to evaluate your individual machines by the end.

**Tips**

- Start by verifying combinations of 2 variables, then 3 and 4
- If that does not work, try introducing polynomial terms. If the program does not converge, use normalization;
- Visualize the decision boundary. If you decide to do so, use regularization to try to smooth it.
Thank you for your attention

Sébastien Marcel and André Anjos

Idiap Research Institute
Martigny, Switzerland
marcel AT idiap DOT ch
http://www.idiap.ch/~marcel
andre DOT anjos AT idiap DOT ch
http://andreanjos.org