Fundamentals in Statistical Pattern Recognition

Lab 1: Linear Regression with Python

Dr André Anjos
Dr Sébastien Marcel

www.idiap.ch/~marcel

Idiap Research Institute
Martigny, Switzerland

www.idiap.ch

May 22, 2013
Outline

Python: Basics

Lab 1.1: Univariate Linear Regression with Gradient Descent

Lab 1.2: Multivariate Linear Regression with Gradient Descent

Lab 1.3: Multivariate Linear Regression with closed-form solution

Lab 1.4: Polynomial Regression (Challenge!)
**Python: Basics**

**Why Python?**

- full programming language (classes, exceptions, lambda functions, complex types);
- extensible via C, C++;
- modular;
- free;
- cross-platform;
- widely used;
- well documented;
- well supported.
Start the interpreter

On a terminal (double-click on the icon on the left of the desktop), type the following to start the Python interpreter:

```
$ python
Python 2.7.3 (default, Sep 26 2012, 21:51:14)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more ...
>>> 3 + 4
7
>>> print("Hello, World!")
Hello, World!
```

Try it

You can try all examples in yellow boxes.

Help is built-in:

```
>>> help(print)
```
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)
```
Exiting the interpreter

To leave the interpreter, use the `exit()` function:

```python
>>> exit()
```

Tip

You can also use the keyboard sequence CTRL-D, to exit.
Objects in Python are *dynamically* typed:

```python
>>> a = 7
>>> type(a)
<type 'int'>
>>> a/2
3
>>> b = 7.
>>> type(b)
<type 'float'>
>>> b/2
3.5
>>> c = 7+5j
>>> type(c)
<type 'complex'>
>>> c/2
(3.5+2.5j)
>>> d = '7'
>>> type(d)
<type 'str'>
```
Everything is an object:

```python
>>> import os
>>> type(os)
<type 'module'>
>>> andre = os
>>> help(andre)
>>> type(os.listdir)
<type 'builtin_function_or_method'>
>>> foo = os.listdir
>>> type(foo)
```
Sequences

Two main types of sequences: lists and tuples (read-only lists). We cover lists, but you can read more online about tuples.

```python
>>> lst = [7, 3., 'string']
>>> lst[0]  # N.B.: indexing is 0-based
7
>>> lst[1:3]  # or lst[1:]
[3.0, 'string']
>>> lst[-1]  # or lst[2]
'string'
>>> lst.remove(3.)  # powerful
>>> print lst
[7, 'string']
>>> lst.append('x')
>>> print lst
[7, 'string', 'x']
>>> len(lst)
3
>>> 3. in lst
False
>>> lst.index('x')
2
```
Sequences (part 2)

Explore: Use TAB completion

```python
>>> lst.<TAB><TAB>
```

Remember

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

More information

http://docs.python.org/2/library/stdtypes.html
# Number Operations

<table>
<thead>
<tr>
<th>Operators</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, /, *</td>
<td>standard (note: integer division)</td>
<td>4 / 3 (=1)</td>
</tr>
<tr>
<td>+=, -=, /=, *=</td>
<td>standard</td>
<td>x = 3; x += 1</td>
</tr>
<tr>
<td>%, %=</td>
<td>reminder of division</td>
<td>7 % 2</td>
</tr>
<tr>
<td>**, **=</td>
<td>exponent</td>
<td>4**2</td>
</tr>
<tr>
<td>//, //=</td>
<td>floor division</td>
<td>9.0//2.0</td>
</tr>
</tbody>
</table>

Use () to mark priority

More information

http://www.tutorialspoint.com/python/python_basic_operators.htm
Scope

Scope in Python is determined by **indentation**:

```python
>>> <control-flow statement> :
... <statement>
... <statement>
... [RETURN] #marks end of block
```

About indentation

- Good practice in **any** programming language
- **Enforced** in Python (default: 4 spaces)
- Never mix spaces and tabs ⇒ editor setup

More information

http://docs.python.org/2.7/reference/lexical_analysis.html#indentation
Functions

To create a function use the `def` keyword. Let’s try it:

```python
>>> def f(x, y=2, z=3):
    ... return (x * y) + z
    ...
>>> <RETURN>
```

Call `f`, what happens? → Quiz!

- `f(1.1)`
- `f(1, 5)`
- `f(0, z=4)`
To create a function use the `def` keyword. Let's try it:

```python
>>> def f(x, y=2, z=3):
...     return (x * y) + z
... <RETURN>
```

Call `f`, what happens? → Quiz!

- `f(1.1)` → `x=1.1; y=2; z=3 → f=5.2`, output is `float`
- `f(1, 5)` → `x=1; y=5; z=3 → f=8`, output is `int`
- `f(0, z=4)` → `x=0; y=2; z=4 → f=4`, output is `int`
Your Modules

You can create your own modules. Try this:

```python
$ gedit first.py

def f(x, y=2, z=3):
    return (x * y) + z

Save, the file, open the Python prompt and type:

>>> import first
>>> first.f(1.1)
5.2
```

More information

http://docs.python.org/2/tutorial/modules.html
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
Outline

Python: Basics

Lab 1.1: Univariate Linear Regression with Gradient Descent

Lab 1.2: Multivariate Linear Regression with Gradient Descent

Lab 1.3: Multivariate Linear Regression with closed-form solution

Lab 1.4: Polynomial Regression (Challenge!)
Univariate Linear Regression with Gradient Descent

Hypothesis

\[ h_\theta(x) = \theta_0 + \theta_1 x \]

Parameters

\[ \theta^T = [\theta_0 \quad \theta_1] \]

Cost function \( J(\theta) \)

\[
J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2
\]
Univariate Linear Regression with Gradient Descent

Gradient Descent: update iteratively $\theta_0$ and $\theta_1$

$$\theta_0^{\text{new}} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_0} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)$$

$$\theta_1^{\text{new}} = \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_1} = \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_i$$
Univariate Linear Regression with Gradient Descent

Your Tasks

Implement:
- the hypothesis: $H(X, \theta)$
- the cost function: $J(X, y, \theta)$
- the partial derivatives: $\partial J(\theta)/\partial \theta$
Univariate Linear Regression with Gradient Descent

Instructions:

```python
$ cd ex1  # change into directory for the exercise
$ ./ex1.py  # execute the main program
```

Predicting ‘salary’ from ‘age’ using Univariate Linear ...

Loading data...

Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features : 14
Initial parameters (theta): [ 0.  0.]
Max iterations : 15
Learning rate : 0.000200

Your h() function should return 1.0 => 0.0
Your h() function should return 1.5 => 0.0
Your h() function should return 2.0 => 0.0

000/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
001/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
...

$ gedit ex1_lab.py  # open this module
Univariate Linear Regression with Gradient Descent

Task 1: Implement the hypothesis $H$

$h_\theta(x) = \theta_0 + \theta_1 x$

Let's implement this on `ex1.lab.py`:

```python
def h(x, theta):
    ""
    ...
    ""

    # replace the next line with your solution
    return 0.
```

Tips

- Use `theta[0]` to access $\theta_0$;
- If unsure about operator precedence, use parenthesis `()`.
Univariate Linear Regression with Gradient Descent

Check:

$ ./ex1.py # execute the main program

Predicting ‘salary’ from ‘age’ using Univariate Linear ...

Loading data...

Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features : 14
Initial parameters (theta): [ 0.  0.]
Max iterations : 15
Learning rate : 0.000200

Your h() function should return 1.0 => 1.0 # CORRECT
Your h() function should return 1.5 => 1.5 # CORRECT
Your h() function should return 2.0 => 2.0 # CORRECT

000/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
001/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
...

...
Univariate Linear Regression with Gradient Descent

Task 1: Implement the hypothesis $H$

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Answer:

```python
def h(x, theta):
    
    # replace the next line with your solution
    return theta[0] + x*theta[1]
```

Univariate Linear Regression with Gradient Descent

Task 2: Implement the cost function $J(\theta)$

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2$$

Let's implement this on `ex1_lab.py`:

```python
def J(X, y, theta):
    """
    ...  
    """
    J_ = 0.
    n = len(X)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = 0.
        J_ += e_
    J_ /= (2*n)
    return J_
```
Let's implement this on `ex1_lab.py`:

```python
def J(X, y, theta):
    
    J_ = 0.
    n = len(X)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = 0.
        J_ += e_
    J_ /= (2*n)
    return J_
```

**Tips**

- Use the previously defined $h()$
- Check `help(range)` to understand what that is
Univariate Linear Regression with Gradient Descent

Check:

```
$ ./ex1.py # execute the main program
Predicting 'salary' from 'age' using Univariate Linear ...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features : 14
Initial parameters (theta): [ 0.  0.]
Max iterations : 15
Learning rate : 0.000200
Your h() function should return 1.0 => 1.0
Your h() function should return 1.5 => 1.5
Your h() function should return 2.0 => 2.0
000/015: J_train = 4.97830020e+05 | J_devel = 5.452251e+05 # Ok
001/015: J_train = 4.97830020e+05 | J_devel = 5.452251e+05 # Ok
...```
Univariate Linear Regression with Gradient Descent

Task 2: Implement the cost function $J(\theta)$

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2$$

Answer:

```python
def J(X, y, theta):
    """
    ...
    """
    J_ = 0.
    n = len(X)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = h(X[i], theta) - y[i]
        J_ += e_**2
    J_ /= (2*n)
    return J_
```
Univariate Linear Regression with Gradient Descent

Task 3: Implement the derivatives of the cost $\frac{\partial J(\theta)}{\partial \theta}$

$$
\theta_0^{\text{new}} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_0} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n}(h_\theta(x_i) - y_i)
$$

$$
\theta_1^{\text{new}} = \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_1} = \theta_1 - \alpha \frac{1}{n} \sum_{i=1}^{n}(h_\theta(x_i) - y_i)x_i
$$

Let’s implement this on `ex1_lab.py`

- Open the file and fill in the gaps in functions $dJ0$ and $dJ1$;
- Run `ex1.py` to make sure the gradient is going down
Check:

$ ./ex1.py  # execute the main program
Predicting ‘salary’ from ‘age’ using Univariate Linear ...
Loading data...
Number of training examples loaded  : 50
Number of development examples loaded : 50
Number of features                   : 14
Initial parameters (theta): [ 0.  0.]
Max iterations                      : 15
Learning rate                       : 0.000200
...
015/020: J_train = 1.06189053e+05  | J_devel = 1.617002e+05
Convergence detected (delta J = 5.71890e-09). Stopping at ...
Result (after 15 iterations):
Final parameters (theta): [ 0.25798845 15.18662164]
Univariate Linear Regression with Gradient Descent

Answers:

```python
def dj0(X, y, theta):
    dJ0_ = 0
    n = len(X)
    for i in range(n):
        dJ0_ += h(X[i], theta) - y[i]
    dJ0_ /= n
    return dJ0_

def dj1(X, y, theta):
    dJ1_ = 0
    n = len(X)
    for i in range(n):
        dJ1_ += (h(X[i], theta) - y[i]) * X[i]
    dJ1_ /= n
    return dJ1_
```
You can explore more options with this program. For details:

$ ./ex1.py --help

How it is done

When you become proficient in Python, you can look inside exercise files and find out details about reading text files and plotting.
Python: NumPy and ndarrays

- Fundamental package for scientific computing in Python
- Provides a multi-dimensional (or N-dimensional) array type: ndarray
- Linear algebra support for these types;
- Vectorized (pre-compiled) routines for most operations.
Python: NumPy and ndarrays

Creating arrays:

```python
>>> import numpy
>>> help(numpy.ndarray)
```

```python
# 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.  0.]
 [ 0.  0.]]
```

```python
>>> # other interesting initializers
>>> help(numpy.ones) #equivalent to Matlab’s
>>> help(numpy.array) #initializes from almost any object!
```
Python: NumPy and ndarrays

Interesting methods and properties:

```python
>>> X = numpy.array([[1,2,3],[4,5,6]])
>>> print X
[[1 2 3]
 [4 5 6]]
>>> X.reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> X.sum()
21
>>> X.mean()
3.5
>>> X.std()
1.707825127659933
>>> X.shape
(2, 3)
>>> X.dtype
dtype('int64')
>>> len(X)
2 #notice: number of rows == dimension 0 extent
>>> X.<TAB> #for much more
```
Indexing and slicing (1D):

```python
>>> X = numpy.array(numpy.arange(5.))
>>> print X
array([[ 0.,  1.,  2.,  3.,  4.]]
>>> X[1] #second element
1.0 #note: indexing starts at 0!
>>> X[-1] #last element
4.0
>>> X[1:3] #slice: second and third elements
array([ 1.,  2.])
>>> X[2:] #slice: from the third element
array([ 2.,  3.,  4.])
>>> X[-4:-1] # what do you think happens?
array([ 0.,  1.,  2.])
```
Indexing and slicing (1D):

```python
>>> X = numpy.array(numpy.arange(5.))
>>> print X
array([[ 0.,  1.,  2.,  3.,  4.]]
>>> X[1]  #second element
1.0  #note: indexing starts at 0!
>>> X[-1] #last element
4.0
>>> X[1:3] #slice: second and third elements
array([ 1.,  2.])
>>> X[2:] #slice: from the third element
array([ 2.,  3.,  4.])
>>> X[-4:-1] # what do you think happens?
array([ 1.,  2.,  3.])
```
Python: NumPy and ndarrays

Indexing and slicing (2D, continued):

```
>>> X = numpy.array(numpy.arange(25.)).reshape(5,5)
>>> print X
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.]]

>>> X[1]  # second row
array([ 5.,  6.,  7.,  8.,  9.])

>>> X[-1]
array([20., 21., 22., 23., 24.])

>>> X[:,0]  # first column
array([ 0.,  5., 10., 15., 20.])  # note: row vector

>>> X[:,1:3]  # second and third columns
array([[1., 2.],
       [6., 7.],
       [11., 12.],
       [16., 17.],
       [21., 22.]])

>>> Z = X[:,1:3]
>>> Z[0,0] = 3.1416

>>> print X  # what do you think happens?
```
Matlab Gotchas

- Array operations are applied element-wise. Example: $a \times b$ 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

Python: Basics

Lab 1.1: Univariate Linear Regression with Gradient Descent

Lab 1.2: Multivariate Linear Regression with Gradient Descent

Lab 1.3: Multivariate Linear Regression with closed-form solution

Lab 1.4: Polynomial Regression (Challenge!)
Multivariate Linear Regression with Gradient Descent

Hypothesis

\[ h_\theta(x) = \theta^T x \]

Parameters

\[ \theta^T = [\theta_0 \ \theta_1 \ \theta_2 \ \cdots \ \theta_d] \]

Cost function \( J(\theta) \)

\[ J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 \]
Multivariate Linear Regression with Gradient Descent

Gradient Descent: update $\theta$, iteratively

$$
\theta_0^{\text{new}} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_0} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)
$$

$$
\theta_d^{\text{new}} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_d} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i) x_{id}
$$

$d = 1, 2, \cdots$
Your Tasks

Implement:

• the hypothesis: \( H(X, \theta) \)
• the cost function: \( J(X, y, \theta) \)
• the partial derivatives: \( \frac{\partial J(\theta)}{\partial \theta} \)
Multivariate Linear Regression with Gradient Descent

Instructions:

$ cd ex2  # change into directory for the exercise
$ ./ex2.py --normalize sales age  # execute the main program

Predicting ‘salary’ from ‘[‘sales’, ‘age’]’ using Multivariate...

Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
Normalizing input data X := (X-mean(X))/std(X)

Initial parameters (theta): [ 0.  0.  0.]
Max iterations : 20
Learning rate : 0.000200

Your h() function should return 9.424800 => 0.0
Your h() function should return 15.708000 => 0.0
Your h() function should return 40.840800 => 0.0

000/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
...

$ gedit ex2_lab.py  # open this module
Multivariate Linear Regression with Gradient Descent

Task 1: Implement the hypothesis $H$

$h_\theta(x) = \theta^T x$

Let's implement this on ex2_lab.py:

```python
def h(x, theta):
    h_ = 0.
    for i in range(len(x)):
        # replace the next line with your solution
        h_ += 0.
    return h_
```

Tips

- The input $x$ does **NOT** contain the bias term
Multivariate Linear Regression with Gradient Descent

Check:

$ ./ex2.py --normalize age sales
Predicting 'salary' from ['age', 'sales'] using Multivariate...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
Normalizing input data X := (X-mean(X))/std(X)
Initial parameters (theta): [ 0.  0.  0.]
Max iterations : 20
Learning rate : 0.000200
Your h() function should return 9.424800 => 9.4248
Your h() function should return 15.708000 => 15.708
Your h() function should return 40.840800 => 40.8408
000/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
001/020: J_train = 0.00000000e+00 | J_devel = 0.000000e+00
...


Multivariate Linear Regression with Gradient Descent

Task 1: Implement the hypothesis $H$

$$h_\theta(x) = \theta^T x$$

Answer:

```python
def h(x, theta):
    h_ = theta[0]
    for i in range(len(x)):
        # replace the next line with your solution
        h_ += theta[i+1] * x[i]
    return h_
```
Multivariate Linear Regression with Gradient Descent

Task 2: Implement the cost function $J(\theta)$

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2$$

Let’s implement this on `ex2_lab.py`:

```python
def J(X, y, theta):
    ...
    J_ = 0.
    n = len(X)  # number of examples in X (== the number of rows)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = 0.
        J_ += e_
    J_ /= (2*n)
    return J_
```
Let's implement this on `ex2_lab.py`:

```python
def J(X, y, theta):
    ...
    J_ = 0.
    n = len(X)  # number of examples in X (== the number of rows)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = 0.
        J_ += e_
    J_ /= (2*n)
    return J_
```

**Tips**

- Remember `X[i]` returns a row from the matrix
- `X` does not contain a column for the bias parameter
Multivariate Linear Regression with Gradient Descent

Check:

```
$ ./ex2.py --normalize age sales
Predicting ‘salary’ from ‘[‘age’, ’sales’]’ using Multivariate...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
Normalizing input data X := (X-mean(X))/std(X)
Initial parameters (theta): [ 0.  0.  0.]
Max iterations : 20
Learning rate : 0.000200
Your h() function should return 9.424800 => 9.4248
Your h() function should return 15.708000 => 15.708
Your h() function should return 40.840800 => 40.8408
000/020: J_train = 4.9783020e+05 | J_devel = 5.452251e+05
001/020: J_train = 4.9783020e+05 | J_devel = 5.452251e+05
...```
Multivariate Linear Regression with Gradient Descent

Answer:

```python
def J(X, y, theta):
    J_ = 0.
    n = len(X)
    for i in range(n):
        # replace the next two lines with your solution
        e_ = h(X[i], theta) - y[i]
        J_ += e_**2
    J_ /= (2*n)
    return J_
```
Multivariate Linear Regression with Gradient Descent

Task 3: Implement the derivatives of the cost $\frac{\partial J(\theta)}{\partial \theta}$

$$
\theta_0^{\text{new}} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_0} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)
$$

$$
\theta_d^{\text{new}} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_d} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_{id}
$$

$d = 1, 2, \ldots$

Let's implement this on ex2_lab.py

- Open the file and fill in the gaps in function $dJ$
- Run ex2.py to make sure the gradient is going down
Multivariate Linear Regression with Gradient Descent

Check:

```
$ ./ex2.py --normalize -a 0.3 -t 50 age sales
Predicting 'salary' from ['age', 'sales'] using ...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
Normalizing input data X := (X-mean(X))/std(X)
Initial parameters (theta): [ 0. 0. 0.]
...
000/020: J_train = 4.97830020e+05 | J_devel = 5.452251e+05
...
045/050: J_train = 6.63860645e+04 | J_devel = 1.377290e+05
Convergence detected (delta J = 6.65023e-09). Stopping at ...
Final parameters (theta): [ -3.49306172e+01 1.28572200e+01 ...]```
Task 3: Implement the derivatives of the cost $\frac{\partial J(\theta)}{\partial \theta}$

\[
\theta_0^{new} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_0} = \theta_0 - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)
\]

\[
\theta_d^{new} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} \frac{\partial J(\theta)}{\partial \theta_d} = \theta_d - \alpha \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)x_i^d
\]

Answer:

```python
def dJ(X, y, theta):
    dJ_ = numpy.zeros(theta.shape)
    n = len(X)
    for i in range(n):
        tmp = h(X[i], theta) - y[i]
        dJ_[0] += tmp
        for j in range(1, len(theta)):
            dJ_[j] += tmp * X[i,j-1]
    dJ_ /= n
    return dJ_
```

```python
54/83
```
Explore now

You can explore more options with this program. For details:

$ ./ex2.py --help

What to try

• Turn-off normalization and find which $\alpha$ works better (converges faster w/o overflowing), how many iterations
• Turn back on normalization and enable iterative plotting using --plot
• Try mixing other variables (any number will work with your program). Which ones work best for you? Why?
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”.
NumPy: Vectorization

We will only cover the dot-product

- NumPy: http://docs.scipy.org/doc/numpy/reference/
- Scipy: http://docs.scipy.org/doc/scipy/reference/
NumPy: Matrix-Vector operations

Matrix (m \times n) Matrix (n \times o) multiplication

\[
A \times B = AB = \begin{bmatrix}
a_{1,1} & \cdots & a_{1,n} \\
a_{2,1} & \cdots & a_{2,n} \\
\vdots & \ddots & \vdots \\
a_{m,1} & \cdots & a_{m,n}
\end{bmatrix}
\begin{bmatrix}
b_{1,1} & \cdots & b_{1,o} \\
b_{2,1} & \cdots & b_{2,o} \\
\vdots & \ddots & \vdots \\
b_{n,1} & \cdots & b_{n,o}
\end{bmatrix}
\]

\[
= \begin{bmatrix}
a_{1,1}b_{1,1} + \cdots + a_{1,n}b_{n,1} & \cdots & a_{1,1}b_{1,o} + \cdots + a_{1,n}b_{n,o} \\
\vdots & \ddots & \vdots \\
a_{m,1}b_{1,1} + \cdots + a_{m,n}b_{n,1} & \cdots & a_{m,1}b_{1,o} + \cdots + a_{m,n}b_{n,o}
\end{bmatrix}
\]

In Python

\texttt{numpy.dot(A, B)}
NumPy: Matrix-Vector operations

Matrix \((m \times n)\) Vector \((n)\) multiplication

\[
A \times \mathbf{b} = Ab = \begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{bmatrix}
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_n
\end{bmatrix}
= \begin{bmatrix}
b_1 a_{1,1} + b_2 a_{1,2} + \cdots + b_n a_{1,n} \\
b_1 a_{2,1} + b_2 a_{2,2} + \cdots + b_n a_{2,n} \\
\vdots \\
b_1 a_{m,1} + b_2 a_{m,2} + \cdots + b_n a_{m,n}
\end{bmatrix}
\]

In Python

```
numpy.dot(A, b)
```
NumPy: Matrix-Vector operations

Transpose $A^T$ ($n \times m$) of $A$ ($m \times n$)

\[ A = \begin{bmatrix}
49 & 9 & 2 & 6200 \\
43 & 10 & 10 & 283 \\
51 & 9 & 3 & 169 \\
\end{bmatrix} \quad A^T = \begin{bmatrix}
49 & 43 & 51 \\
9 & 10 & 9 \\
2 & 10 & 3 \\
6200 & 283 & 169 \\
\end{bmatrix} \]

In Python

A.transpose(), or numpy.transpose(A)
In Python

```
numpy.linalg.inv(A), or
numpy.linalg.pinv(A) (for the -Moore-Penrose- pseudo-inverse)
```
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 functions 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)
>>> numpy.dot(t, col) # ok
array([14])
>>> numpy.dot(col, t) # not ok
```

If one of the is a column vector, it may go wrong

```python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
```
NumPy: Vectorization

Can use it with 2D arrays (matrices):

```python
>>> m = numpy.array(numpy.arange(4)).reshape(2,2)
>>> m
array([[0, 1],
       [2, 3]])
>>> numpy.dot(m, m)
array([[ 2,  3],
       [ 6, 11]])
>>> # matrix versus vector
>>> v = numpy.array([2.0, 3.0])
>>> numpy.dot(v, m) # means v * m
array([ 6., 11.]) # notice automatic type casting
>>> numpy.dot(m, v) # means m * v', using v's shape
array([ 3., 13.])
>>> numpy.dot(m, v.reshape(2,1)) # means m * v' literally
array([[ 3.],
       [13.]])
>>> numpy.dot(v.reshape(2,1), m) # not Ok => error
...  
ValueError: matrices 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, 0.0]]) #notice automatic type casting
```

More information

- `help(numpy.linalg)`
Outline

Python: Basics

Lab 1.1: Univariante Linear Regression with Gradient Descent

Lab 1.2: Multivariate Linear Regression with Gradient Descent

Lab 1.3: Multivariate Linear Regression with closed-form solution

Lab 1.4: Polynomial Regression (Challenge!)
Multivariate Linear Regression with Closed-form

Hypothesis

\[ h_\theta(x) = \theta^T x \]

Parameters

\[ \theta^T = [\theta_0 \ \theta_1 \ \theta_2 \ \cdots \ \theta_d] \]

Cost function \( J(\theta) \)

\[ J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 \]
Multivariate Linear Regression with Closed-form

Design matrix $X$ of size $n \times (d + 1)$

$$X = \begin{bmatrix}
1 & 49 & 9 & 2 & 6200 & 966 & 15.58065 \\
1 & 43 & 10 & 10 & 283 & 48 & 16.96113 \\
1 & 51 & 9 & 3 & 169 & 40 & 23.66864 \\
1 & 55 & 22 & 22 & 1100 & -54 & -4.90909 \\
1 & 44 & 8 & 6 & 351 & 28 & 7.977208 \\
1 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{bmatrix}$$

$$x_2^T = \begin{bmatrix}
1 & 43 & 10 & 10 & 283 & 48 & 16.96113 \\
\end{bmatrix}$$

$$y^T = \begin{bmatrix}
1161 & 600 & 379 & 651 & 497 & \ldots \\
\end{bmatrix}$$

Closed-form solution

$$\theta = (X^TX)^{-1}X^Ty$$
Multivariate Linear Regression with Closed-form

Your Tasks

Implement:

• the hypothesis in vectorized form: $\theta = (X^T X)^{-1} X^T y$
• the cost function in vectorized form: $J(X, y, \theta)$
Multivariate Linear Regression with Closed-form

Instructions:

```
$ cd ex3  # change into directory for the exercise
$ ./ex3.py sales age  # execute the main program
Predicting ‘salary’ from ‘[‘sales’, ‘age’]’ using Multivariate...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
J_train = 0.00000000e+00 | J_devel = 0.000000e+00
Final parameters (theta): [ 0.  0.  0.]
...$ gedit ex3_lab.py  # open this module
```
Multivariate Linear Regression with Closed-form

Task 1 and 2: Implement the missing bits

\[ \theta = (X^T X)^{-1} X^T y \]
\[ J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 \]

Let's implement this on ex3_lab.py:

```python
def theta(X, y):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    # replace the next line with your answer
    return numpy.zeros((Xp.shape[1],), dtype='float64')

def J(X, y, theta):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    # replace the next line with your answer
    return 0.
```
Let’s implement this on `ex3_lab.py`:

```python
def theta(X, y):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    # replace the next line with your answer
    return numpy.zeros((Xp.shape[1],), dtype='float64')

def J(X, y, theta):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    # replace the next line with your answer
    return 0.
```

**Tips**

- For $\theta$, first transpose $X_p$ to simplify the calculations;
- For $J(\theta)$, first calculate $h = X_p \cdot \theta$;
Check:

$ ./ex3.py sales age # execute the main program
Predicting ‘salary’ from ‘[’sales’, ’age’]’ using Multivariate...
Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 2
J_train = 6.63860645e+04 | J_devel = 1.377289e+05
Final parameters (theta): [ -3.49302762e+01 3.06297602e-02 ...]
Multivariate Linear Regression with Closed-form

Task 1 and 2: Implement the missing bits

\[ \theta = (X^T X)^{-1} X^T y \]

\[ J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 \]

Answer:

```python
def theta(X, y):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    Xp_t = Xp.transpose()
    return numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(Xp_t, Xp)), Xp_t), y)

def J(X, y, theta):
    Xp = numpy.hstack([numpy.ones((len(X),1)), X])
    h = numpy.dot(Xp, theta)
    return ((h - y)**2).mean() / 2
```
You can explore more options with this program. For details:

$ ./ex3.py --help

What to try

- Turn-on normalization, see what happens. Do you understand it?
Outline

Python: Basics

Lab 1.1: Univariante Linear Regression with Gradient Descent

Lab 1.2: Multivariate Linear Regression with Gradient Descent

Lab 1.3: Multivariate Linear Regression with closed-form solution

Lab 1.4: Polynomial Regression (Challenge!)
Polynomial Regression

**Trick**

- set new features:

  \[ x'_1 = x_1 \quad x'_2 = x_1^2 \quad x'_3 = x_1^3 \quad x'_4 = x_1^3 \ldots \]

  \[ x'_1 = x_1 \quad x'_2 = \sqrt{x_1} \quad x'_3 = x_2 \quad x'_4 = \sqrt{x_2} \ldots \]

- use the new features in the revised hypothesis:

  \[ h_\theta(x') = \theta_0 x_0 + \theta_1 x'_1 + \theta_2 x'_2 + \theta_3 x'_3 + \cdots = \theta^T x' \]

- use gradient descent or closed-form to find \( \theta \)
Polynomial Regression

Instructions:

$ cd ex4  # change into directory for the exercise
$ ./ex4.py --plot sales  # execute the main program

Predicting 'salary' from ['sales'] using Multivariate ...

Loading data...
Number of training examples loaded : 50
Number of development examples loaded : 50
Number of features available : 14
Number of user selected features : 1
Number of polynomial terms : 3
First training example : [ 6200.]
First training example transformed : [ 6.20000000e+03 ... 

J_train = 6.21409735e+04 | J_devel = 1.126583e+05

Final parameters (theta): [ 5.53308156e+02 1.15311024e-01 ... 

$ gedit ex4_lab.py  # open this module
Polynomial Regression

Input format

Example: You select sales, age, profits

\[
X = \begin{bmatrix}
\text{sales}_1 & \text{age}_1 & \text{profits}_1 \\
\text{sales}_2 & \text{age}_2 & \text{profits}_2 \\
\text{sales}_3 & \text{age}_3 & \text{profits}_3 \\
\vdots & \vdots & \vdots \\
\text{sales}_n & \text{age}_n & \text{profits}_n \\
\end{bmatrix}
\]

You can modify the variable transformation:

```python
def make_polynom(X):
    return numpy.hstack([
        X[:,(0,)], # returns first variable untouched
        X[:,(0,)]**2, # the square of the first variable
        X[:,(0,)]**3, # the cube of the first variable
    ])
```
Polynomial Regression

The Challenge!

1. Train the best model for predicting the salary:
   - Using any number variables;
   - Using polynomial expansion.

2. Select the best model that minimizes the cost on the development set
Polynomial Regression

The Challenge!

1. Train the best model for predicting the salary:
   - Using any number variables;
   - Using polynomial expansion.

2. Select the best model that minimizes the cost on the development set

A possible way through

- Using the plot option (``--plot``), start by looking at single variables and polynomial expansions of those;
- Choose the variable transformations that best allow you to map to the salary (linearly). Annotate, go to the next variable;
- By the end, try to combine the best variables you found.
Polynomial Regression

The Winner

./ex4.py --test --plot <the variables of your choice>
Next Steps

Explore more

- http://www.diveintopython.net/toc/index.html
- Free Python Books: http://pythonbooks.revolunet.com/
Sébastien Marcel and André Anjos

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