Aprendizagem Automática (Machine Learning)

Exercises

Software configuration

This tutorial uses Python 3.6 with the Spyder IDE and libraries such as Matplotlib, Numpy and Scikit-Learn. The easiest way to set up the software for this tutorial is to use the free Anaconda package distributed by Continuum Analytics™ which you can download from:

https://www.anaconda.com/download

To open the Spyder IDE[1] in Windows click the link in the Anaconda folder on your Start menu (probably in All Programs). If you are running Spyder for the first time, it may ask you to select a folder (the Spyder workspace) for your Python projects. Choose a convenient folder.

To test your instalation, copy the following code and paste it on the iPython console, on the bottom-right of the Spyder interface:

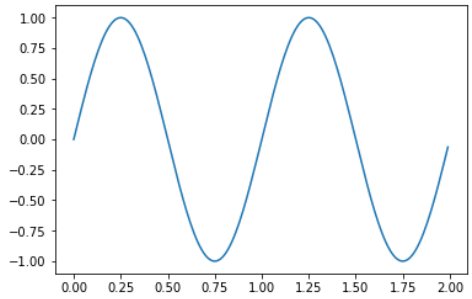
```
%matplotlib inline
from pylab import *
t = arange(0.0, 2.0, 0.01)
s = sin(2*pi*t)
plot(t, s)
show()
```

Press enter to execute the code. This should be the result:

```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 29 2018, 16:14:56)
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: %matplotlib inline
   ...: from pylab import *
   ...: t = arange(0.0, 2.0, 0.01)
   ...: s = sin(2*pi*t)
   ...: plot(t, s)
   ...: show()
   ...:
   ...:
```



[1] Integrated Development Environment, consisting of a source-code editor, console and other tools to make development easier.

Chapter 1

Python basics

Variables: data types and scope. Flow control and functions. Importing modules. Files and plotting.

1.1 Using the console; basic data types

Note: when copying text from the pdf file, some characters may not be copied correctly. Check the code carefully after pasting or type the code yourself.

Open the Spyder IDE. On the lower right corner of the Spyder interface you can find the iPython console. This is an interactive Python console that you can use to immediately execute commands. We will start with some simple commands just to illustrate some aspects of Python. We will start by creating two string variables. Strings in Python can be delimited with `"` or `'`, but must begin and end with the same delimiter. You can also use three delimiters to create multi-line strings. Type this in the console

```
first_string = "single line string"
second_string = '''this string
has two lines'''
```

This created two string variables. Note that the first string is delimited with a single double-quote character (`"`) while the second string is delimited by a sequence of three single-quote characters (`'`, repeated three times) in order to allow line breaks within the string. Also note that in Python the most common convention for compound names is to use all lower-case and separate the words with the underscore character.

Now type the following directly in the console:

```
first_string
second_string
print(second_string)
```

You should see this output as you type and enter the commands:

```
first_string
Out [3] : 'single line string'
```

```
second_string
Out [4] : 'this string\nhas two lines'

print(second_string)
this string
has two lines
```

Console commands that return values echo these values if they are not attributed to any variable. Note, however, that the echo may not display exactly as the `print` command does. In this case, the line break is represented with the escape sequence `\n`. You can use these escape sequences when creating strings too.

Strings, like everything else in Python, are objects. You can use the `dir` command to list the methods belonging to an object:

```
dir(first_string)
```

Typing this, you will see a long list of methods available in the string object. By convention, methods whose name begins with an underscore are private and not meant to be called from outside the object[1].

Also note that even constants are objects in Python. Try these examples:

```
first_string.upper()
Out[12]: 'SINGLE LINE STRING'

first_string.startswith('single')
Out[13]: True

first_string.startswith('abc')
Out[14]: False

'abc'.startswith('a')
Out[15]: True

'ABC'.lower()
Out[16]: 'abc'

'ABC'.lower
Out[17]: <function lower>
```

Note that the fourth and fifth examples use methods from constant objects. Also note that, when invoking a function (or method) it is always necessary to write the parenthesis even when there are no arguments. Otherwise the command is interpreted as merely referencing the function object (functions are also objects in Python) instead of executing it.

To index elements or slices of ordered arrays, like a string, we use the square brackets followed by an index value (zero-based) or a slice with the syntax `[beginning:end:step]`, where the step is optional. Here are some examples. Try them out (one at a time) to understand how slicing works.

```
first_string[0]
first_string[1]
first_string[2:4]
```

[1] There is a difference between a single and a double underscore, but we will not worry about that in this course.

```
first_string[-4:]
first_string[4:]
first_string[:4]
```

Note that indexes are zero-based and negative indexes count from the end, as illustrated here with the string `'Ludwig'`:

```
+---+---+---+---+---+---+
| L | u | d | w | i | g |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

Slices are set at the intervals between elements in the array:

```
+---+---+---+---+---+---+
| L | u | d | w | i | g |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Thus, a slice `[1:3]` will span the second and third elements.

Apart from strings, other common data types are integers and floating point numbers, lists, tuples and dictionaries. Here are some examples:

```
nothing = None                          # no value
minimum_wage = 505                      # integer
pi_squared = 9.8696                     # float
first_name = 'Ludwig'                   # string
years = [1999,2000,2001]                # list
coordinates = (23.4, 12.6, 13.5)        # tuple
animal_classes = {'fox':'mammal',       # dictionary
                  'snake':'reptile',
                  'fly':'insect'}
```

Dictionaries are defined by a comma-separated list of `key:value` pairs. The values can be any objects but the keys must be immutable objects. These are objects whose data cannot be changed, and include numbers, strings and tuples. Lists cannot be used as keys in dictionaries because list contents can be changed. To illustrate this, try these commands:

```
a_list = [1,2,3]
a_list[1] = 4
a_string = 'abc'
a_string[1] = 'x'
a_tuple = (1,2,3)
a_tuple[1] = 4
```

You will notice that trying to change the contents of a string or tuple will raise an exception

1.2 The first script

Instead of using the console directly, it is often better to write the code to a source file. Create a new file in Spyder and save it with a `.py` extension in the same folder to where you extracted this session's material. Let us start by reading the file `planets.csv`, containing the name, mean distance from the Sun (in Astronomical Units) and orbital period (in Earth years) for each planet in our solar system. To read a text file, we simply create a file object using the `open` function and then read all lines to a list of strings. Finally, we close the file to free it.

```
1 fil = open('planets.csv')
2 lines = fil.readlines()
3 fil.close()
```

If you do not care when the file is freed, you can read a file in a single line of code:

```
1 lines = open('planets.csv').readlines()
```

Python's garbage collection will eventually dispose of the file object and free up the file. This is often adequate for most purposes.

You can run your script by pressing `F5`. Since the `lines` variable has a global scope, you can access its contents in the console:

```
lines
Out[36]:
['Name,mean distance from Sun (AU),orbital period (Earth years)\n',
 'Mercury,0.39,0.24\n',
 'Venus,0.72,0.62\n',
 'Earth,1,1\n',
 'Mars,1.52,1.88\n',
 'Jupiter,5.2,11.86\n',
 'Saturn,9.54,29.46\n',
 'Uranus,19.18,84.01\n',
 'Neptune,30.06,164.8\n']
```

This list of strings contains all lines in the text file. Also note that the newline characters are included in the strings. String objects have a `strip()` method that returns the string without any whitespace in the beginning and end of the string. This is useful if you need to clean up the strings after reading.

As an exercise, let's write a function that receives a planet's name and returns its orbital period reading this file. This is the function, type it in the beginning of your script:

```
1 def planet_period(planet):
2     """Return orbital period
3     Uses planets.csv
4     """
5     lines = open('planets.csv').readlines()
6     for line in lines:
7         parts = line.split(',')
8         if parts[0] == planet:
9             return float(parts[2])
```

A function definition begins with the keyword `def` followed by the name of the function, the parameters in parentheses and then a colon (`:`). When you press enter after a colon the Spyder editor

should automatically indent your code. This indentation is part of Python's syntax and indicates a code block, so be careful of the proper alignment of your indented blocks.

Lines 2 and 3 are the docstring for the function. This is optional but may be useful if you use many functions because it is displayed by the `help` function using the command `help(function_name)`. You can also use the `help` function on any other function or method (in which case you must give the object too, for example, `help(line.split)`).

Then we read all the file contents and loop through the lines. The `for variable in iterable:` construct loops through all elements of an iterable object. In this case, the list with all strings. The block inside the loop is indented, and includes splitting the line on the commas to separate the fields and then checking if the first field is equal to the planet name. This if block is also indented, and the `return` command immediately exits the function returning the values indicated after the command. Note the `float` call to typecast the string into a floating number.

Now run your script (`F5`) to declare this function. Then you should get this result in the console if everything is working:

```
planet_period('Mars')
Out[38]: 1.88
```

Since the function declaration has a global scope in the script, you can access it in the console after running the script. However, you cannot access variables with a local scope inside the function.

1.3 Using libraries: Numpy

Now we will use the Numpy library to store the planet data in a two-dimensional matrix. To use a module or a library in Python we must import it into our namespace. Here is one way to do this:

```
1 import numpy
2
3 def load_planet_data(file_name):
4     """Return matrix with orbital radius and period"""
5     rows = []
6     lines = open(file_name).readlines()
7     for line in lines[1:]:
8         parts = line.split(',')
9         rows.append( (float(parts[1]),float(parts[2])) )
10    return numpy.array(rows)
11
12 data = load_planet_data('planets.csv')
```

In this example, first we import the Numpy library. Note that the convention in Python is to group all `import` statements at the beginning of the module (the `.py` file). Then we read the file contents as a list of strings and, starting from the second line to skip the column headers, we split the fields on the commas and append tuples to the `rows` list. Note that the tuple is formed using parentheses, and contains the distance from the Sun and the orbital period.

After the loop ends (note the indentation) we return an array object from the Numpy library created using the list of tuples. An array of arrays (lists or tuples), in this case, results in a two-dimensional matrix, where each row corresponds to each of the inner elements of the outer array.

Finally, in this script we call the function and store the resulting matrix in the `data` variable. Note the indentation, indicating that this call is outside the function definition. Run the script and type the following commands in the console, one at a time:

```
data
data[:,0]
data[:,[0]]
data[:,0]>1
data[data[:,0]>1,:]
```

The first one outputs the matrix object, which is a two-dimensional array. You can check the values to see if you have all data correctly loaded. The second returns a one-dimensional array with the values of the first column, since we specify that we want all elements with all indexes in the first dimension (the rows) and index 0 in the second dimension (columns).

The third outputs all values in the first column but keeps the two dimensions of the array, returning a column. Note the difference between the two, as this is sometimes an important detail when manipulating data.

The fourth command returns a boolean array with the result of comparing all values in the first column with the value of 1. Finally, the fifth shows how you can use these boolean masks to pick out specific values from the array. Indexing with a boolean mask will return those indexes with values of `True`.

Exercises

- Select the rows corresponding to planets with orbital periods greater than 10 years.
- How many planets have orbital periods greater than 10 years? Hint: you can use the `len` function to find the length of an array or the `numpy.sum` function to find the total sum of array elements, which considers `True` to be 1 and `False` to be 0.
- Select the orbital periods of the planets whose orbital periods in years are greater than twice the orbital radius in AU. Note that you can use algebraical operators, such as sum or multiplication, with array objects.

1.4 Plotting

We'll start by plotting our data, using the Matplotlib library. Here is an example of the basics:

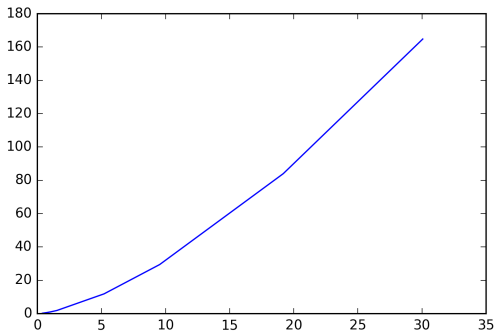
```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(data[:,0],data[:,1])
plt.show()
plt.close()
```

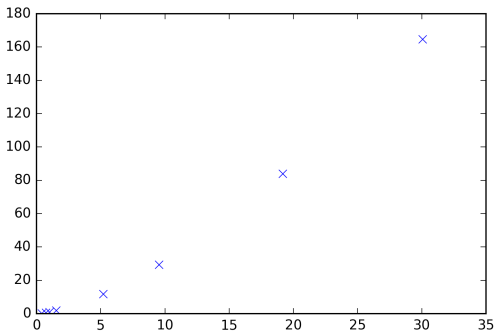
First we import the pyplot module in the matplotlib library but, to avoid having to write `matplotlib.pyplot` every time we use one of its functions, we use the `as` keyword to give it an alias, so now we can refer to it as `plt`.

Then we call the `figure` function to create a new figure for plotting, `plot` to plot the data, giving two arrays for the X and Y values, and then `show()` for displaying the graph. Finally, `close()` frees all the figure objects.

By default, this will display a line joining all the points. But we can specify the format of the plot using a format string as the third argument. Simply replace the `plot` command above with `plt.plot(data[:,0],data[:,1],'x')` [2]

You can also save the figures with the `savefig` function. For example, using `plt.savefig('planets.png',dpi=300)` after plotting and before closing the figure will save the image as a png file with a resolution of 300 dots per inch. This should be the result of your plots:





Using the iPython console

When running your scripts without the iPython console's inline plotting feature, you need to call the (show) method for each plot you wish to display. This will create a window showing the plot and suspend execution until the window is closed. You can use the `close` method to dispose of the plot object.

The iPython console has an inline plotting feature that not only shows the plots in the console when you use the (show) command but also displays any plots active in memory when the script ends if they are not visible. Thus, when using the iPython console, such as when running your scripts in Spyder, you do not need the (show) if you do not close your plots. But note that this is a feature of the iPython console and, without calling the (show) method, your plots will not be displayed if your script is run without the iPython console.

You can disable the iPython console's inline plotting feature with the magic command

```
%matplotlib
```

or enable it again with

```
%matplotlib inline
```

Note that these commands starting with `%` are not Python instructions but special commands for the iPython console and only work with this console.

[2] You can find more information on plotting here: http://matplotlib.org/users/pyplot_tutorial.html

1.5 Exercise 1

The mass of the Sun can be computed from the orbital distance and velocity of any planet from the following equation:

$$M_{Sun} = \frac{v^2 r}{G}$$

Where v is the orbital velocity in m/s, r is the orbital radius in m and G is the gravitational constant $6.67 \times 10^{-11} Nm^2 kg^{-2}$.

The file *planets.csv* has data on the orbital radii and periods of the planets in our solar system. The periods are in Earth years (one Earth year is 3.16×10^7 seconds long) and the radii are in astronomical units (one AU is 1.496×10^{11} meters). To obtain the orbital velocity you need to divide the perimeter of the orbit by the period, in the appropriate units. The perimeter of a circle is $2\pi r$, and you can obtain the value of π from the `numpi.pi` constant.

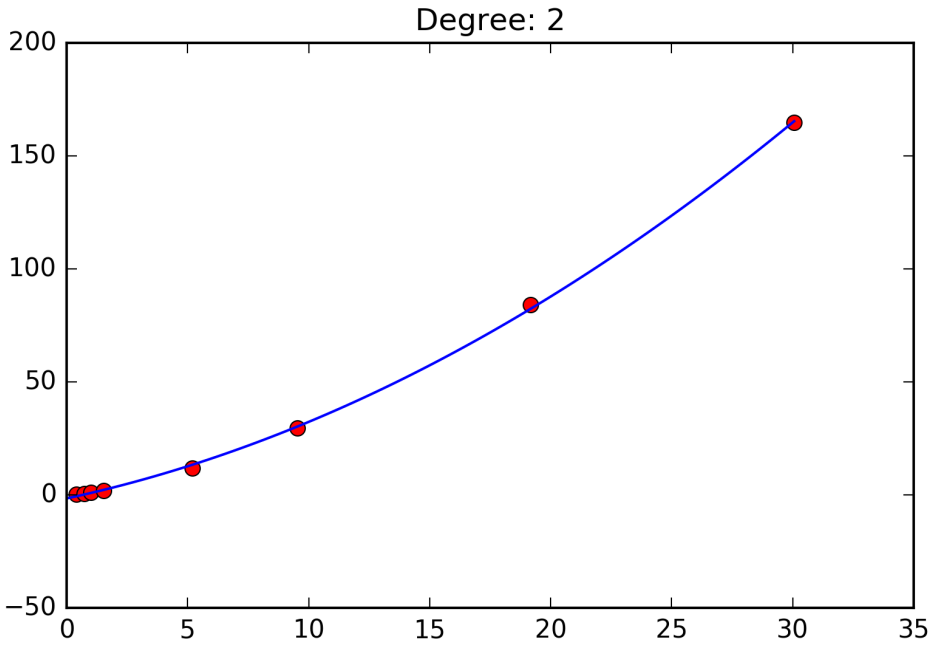
$$v = \frac{2\pi r}{\tau}$$

Compute the mass of the sun using the data for each planet and then compute the average and the standard deviation of those estimates. You can use the `numpy.mean()` and `numpy.std()` functions for these statistics. Note that this is not the most accurate way of computing the mass of the Sun, but it provides an approximate estimate and the purpose of this exercise is to practice some simple computations with arrays.

Note that you can distribute algebraic operations over all elements of a Numpy array easily. Any operation involving two arrays with the same dimensions will be applied over all corresponding elements [3]. Also, Python accepts numbers written in scientific notation. For example, 6.67e-11 for the gravitational constant. The average result should be a mass of approximately 1.98×10^{30} grams for the Sun, with a standard deviation of 2.95×10^{28}

Finally, using Lecture 2 as a reference, fit a second degree polynomial to the original data, with the radius in AU and the period in years, and plot the data superimposed with the polynomial curve. The result should look like this:

[3] This is also possible in some cases where the arrays do not have the same dimensions. See more on broadcasting algebraic operations here, if necessary: http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html



1.6 Exercise 2

The file *polydata.csv* has the data for the polynomial fitting example of lecture 2. Following the lecture notes for lecture 2, fit polynomial curves of degree 3 and 15 and plot each curve superimposed with the data. Answer the following questions:

- What is the *hypothesis class* you are using in each case (degree 3 and degree 15)?
- What is the corresponding *model* for each *hypothesis class*?
- What is the *hypothesis* in each case?
- How was each *hypothesis* chosen?

Chapter 2

Linear Regression

Linear regression using polynomials. Training, validation and test. Selecting the best model by validation on the best hypothesis.

An important goal in this tutorial is to show how sampling and noise in data can have a large effect on the hypotheses we find to model the data. This is especially important when dealing with small data sets.

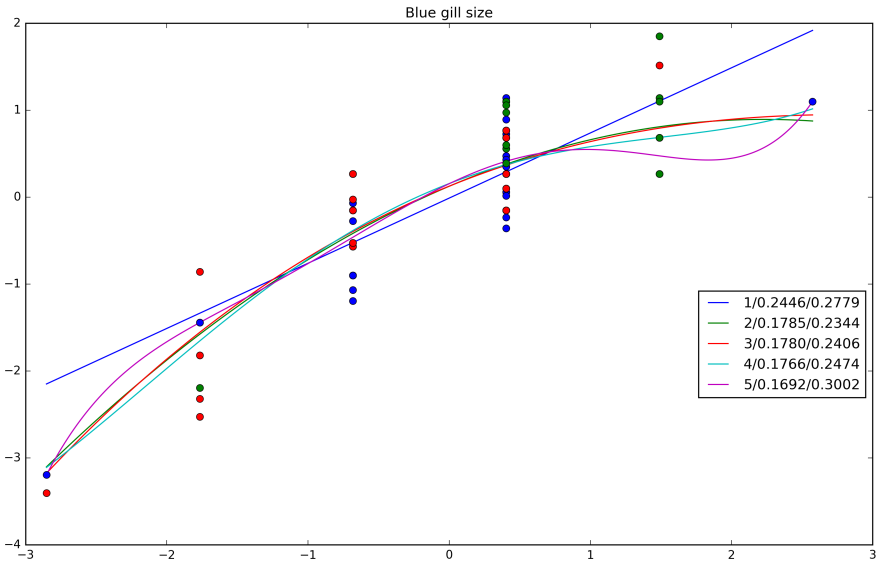
2.1 Exercise 1: regression

Download the `bluegills.txt` file to your working folder. This file contains data on the size of bluegill fish (in millimetres) of different ages (in years) [1]. You can easily load this data using the `loadtxt()` function from the Numpy library. Remember also to standardize the data. See lecture 4 for examples on how to do this. Also, check the online documentation for the Numpy functions you will use (`loadtxt()`, `std()`, `mean()`).

After standardizing the data, split it into three sets: half the points for *training*, one quarter for *validation* and one quarter for *test*. The training set will be used to fit each polynomial curve. The validation set will be used to estimate the true error of each polynomial curve and thus select the best one. The test set will be used to obtain an unbiased estimate of the selected curve since selecting curves based on the error measure will make that measure biased. Remember to shuffle the data randomly before splitting it because the order of the data in the file may not be random.

Now select the best polynomial curve from polynomials of degree 1 through 6. See lecture 3 for examples. Your script should report the best degree for the polynomial curve and also the test error of the best hypothesis. Also, do the plot of the different polynomials. It should look something like this, with different colors for the training, validation and test sets. The labels show the degree and the training and validation errors.

[1] Data from [1] and https://onlinecourses.science.psu.edu/stat501/node/325



To plot several lines in the same chart, you can create the figure, plot the data, and then plot each line before calling the `plt.show()` function. To add a legend, see the tutorial on http://matplotlib.org/users/legend_guide.html. For formatting a string, you can use the string `format()` method. There are many examples online. See, for example, http://www.python-course.eu/python3_formatted_output.php.

2.2 Exercise 2: smaller data set

Repeat the previous exercise with the `yield.txt` data set. This is a hypothetical result from an experiment giving different yields as a function of temperature. Note that this data set is much smaller (only 15 points). Run your script several times and note how the results vary significantly as a function of the random attribution of data points to training, validation or test sets. This is an important issue to bear in mind when dealing with real data.

2.3 Questions

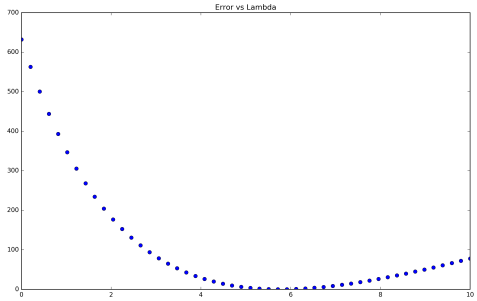
After these exercises, you should be able to answer the following questions:

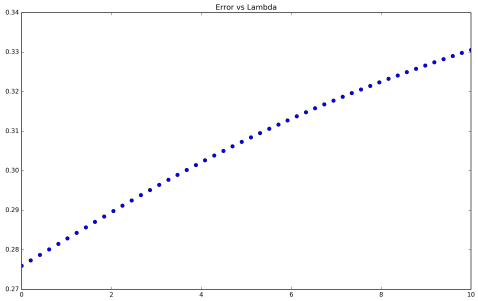
- What is the difference between the hypothesis class, the model and the hypothesis?
- What is the difference between the training error, the test error and the true error of a hypothesis?
- Why can we not use the training error to estimate the true error?
- Why do we need the test error? Why not use the validation error to estimate the true error?

2.4 Optional exercise: Ridge regression

Based on the example in Lecture 3, repeat the bluegill regression but using Ridge regression expanding the original data to degree 5. Explore λ values from 0.001 through 10, on a linear scale. Note that

the validation error will depend very much on the attribution of data points to training or validation sets. Run your code several times to see how validation error varies with λ. Here are two examples of possible results:





These examples illustrate how sampling and selection of data for training and validation can have a large effect on the results.

Chapter 3

Cross-validation and Logistic Regression

Classification with logistic regression. Model selection with cross-validation.

3.1 Exercise 2: Model selection with cross-validation

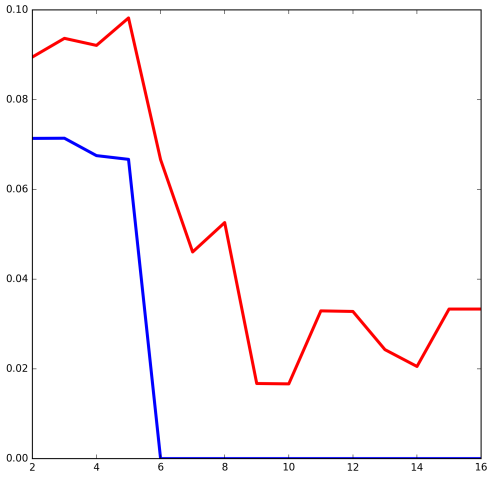
Download the `data.txt` file to your working folder. This file contains the synthetic data set shown on lecture 5. Use lecture 5 as a guide for this exercise. Optionally, download the `t3_aux.py` file with auxiliary functions for this exercise, which you can import into your own module (`poly_16features(X)` and `create_plot`).

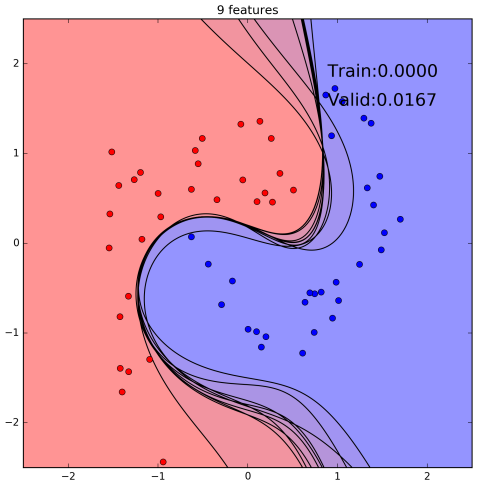
Load the data, standardize it and split it into the training and test sets. Note that, in regression, as we did in the previous tutorial, we also standardize the values to predict so that the results are more reproducible and to avoid numerical instability. In classification, however, the values to predict are class labels and it makes no sense to standardize those. So apply standardization only to the feature vectors. Then expand the data to 16 features:

1. x_1
2. x_2
3. $x_1 \times x_2$
4. x_1^2
5. x_2^2
6. x_1^3
7. x_2^3
8. $x_1^2 \times x_2$
9. $x_1 \times x_2^2$
10. x_1^4
11. x_2^4
12. $x_1^3 \times x_2$
13. $x_1 \times x_2^3$
14. $x_1^2 \times x_2^2$
15. x_1^5
16. x_2^5

Use one third of the data for the test set. The training set will be used to select the number of features to use by cross-validation. Remember to shuffle the data randomly before splitting it because the order of the data in the file may not be random. Use stratified sampling for splitting the data and for the cross-validation folds. See lecture 5 for more details.

Now select the best logistic regression model by 10-fold cross-validation. You can use the code samples on lecture 5 but make sure you understand the different steps. Plot the training and cross-validation error for the different models to select the best one. If several models have very similar cross-validation errors, prefer a model with fewer features. This is an example of what the results may be, but note that your actual result may be different due to the random assignment of points to training and test sets. The figures below show the error plot and the plot for the nine features model, which appears to be the best one in this case.



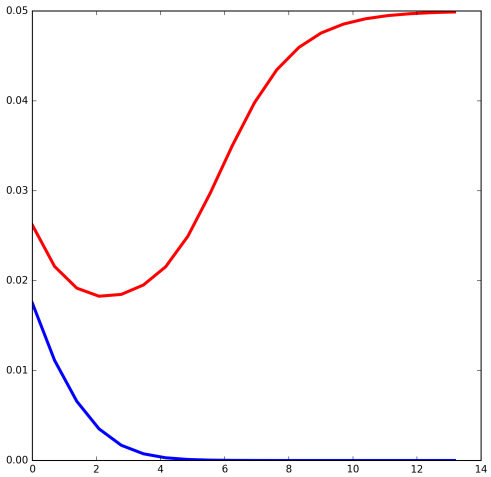


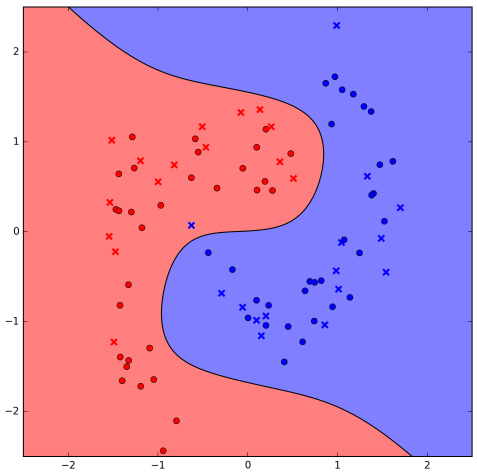
Try running your code a few times to get a feeling for how the results vary depending on the assignment of points to training and testing sets. Note that this is often the case with small data sets, although the effect is less pronounced the more data there is available.

Once you select the correct number of features, train a logistic regression with that number of features using the full training set and measure the test error.

3.2 Exercise 2: Regularization

Use cross validation to find the best C value for regularization of the 16-features model. Start with $C = 1.0$ and double the value of C at each iteration for 20 iterations, reporting the C value and the cross-validation error for each iteration. The figure below shows the plot of the training and cross-validation error as a function of the logarithm of C and the hypothesis trained with the best C value.





3.3 Questions

After these exercises, answer the following questions:

- Why does cross-validation give us a more reliable estimate of the true error of the model?
- What is the purpose of stratified sampling for splitting the data into the folds, training and test sets?
- How does regularization work on logistic regression?

3.4 Extra material: Plotting contours

This is not part of the course subject matter, but if you want to know how to plot the contours for the classifier, here is a code example. Note that this is only useful for two-dimensional data so, in practice, you will seldom have use for this sort of plots. The plot is done using the contour plotting functions from the `pyplot` module, computing the classification for all points in a two-dimensional grid. The `poly_16features` function is the function that expands the original data into the 16 features.

```
def poly_mat(reg,X_data,feats,ax_lims):
    """create score matrix for contour
    """
    Z = np.zeros((200,200))
    xs = np.linspace(ax_lims[0],ax_lims[1],200)
    ys = np.linspace(ax_lims[2],ax_lims[3],200)
    X,Y = np.meshgrid(xs,ys)
    points = np.zeros((200,2))
    points[:,0] = xs
    for ix in range(len(ys)):
        points[:,1] = ys[ix]
        x_points=poly_16features(points)[:,:feats]
        Z[ix,:] = reg.decision_function(x_points)
    return (X,Y,Z)

def create_plot(X_r, Y_r, X_t, Y_t, feats, best_c):
    """create imege with plot for best classifier"""
    ax_lims=(-3,3,-3,3)
    plt.figure(figsize=(8,8), frameon=False)
    plt.axis(ax_lims)
    reg = LogisticRegression(C=best_c, tol=1e-10)
    reg.fit(X_r,Y_r)
    plotX,plotY,Z = poly_mat(reg,X_r,16,ax_lims)
    plt.contourf(plotX,plotY,Z,[-1e16,0,1e16], colors = ('b', 'r'),alpha=0.5)
    plt.contour(plotX,plotY,Z,[0], colors = ('k'))
    plt.plot(X_r[Y_r>0,0],X_r[Y_r>0,1],'or')
    plt.plot(X_r[Y_r<=0,0],X_r[Y_r<=0,1],'ob')
    plt.plot(X_t[Y_t>0,0],X_t[Y_t>0,1],'xr',mew=2)
    plt.plot(X_t[Y_t<=0,0],X_t[Y_t<=0,1],'xb',mew=2)
    plt.savefig('final_plot.png', dpi=300)
    plt.close()
```

Chapter 4

Support Vector Machines

Introduction to Support Vector Machines. Identifying support vectors. Soft margins and regularization. Optimizing kernel parameters with cross-validation. Comparing classifiers.

Download the `T4data.txt` and the `T4aux.py` file to your working folder. The first file contains a synthetic data set for the exercises. The second file contains an auxiliary plotting function. This exercise covers lectures 9, 10 (SVM) and the end of lecture 7 (comparing classifiers).

4.1 Exercise 1: Support Vector Machine and Support Vectors

In this exercise you will train a Support Vector Machine to classify the data set and examine the support vectors. Load the data and standardize it. Since this exercise will not require any selection or error estimation, do not split the data into a training and test sets. Use all data for training the `SVC` classifier. Note that, before standardizing, you will need to separate the features from the class column. The class is the last column on the data file. Fit a linear SVM to your data. To do this, use the `sklearn.svm.SVC` class with a linear kernel (with the `kernel='linear'` argument). Check the documentation for the details on the parameters and how to use the classifier[1]. After fitting, plot the data, discriminant and margins using the `plot_svm` function provided in the `T4aux.py` module. The arguments for this function are the data matrix (with the 1 and -1 classes in the last column; do not separate the features from the classes), the pre-trained SVM classifier object, the file name for the image and the value of C. Compare the results using C values of 0.1, 1 and 10. This should be the result: The `SVC` object, after fitting the classifier, includes the following useful attributers:

- `support_`, with the indexes of the support vectors in the data used for fitting and
- `dual_coef_`, with the product of the alpha values by class labels (-1 and 1) for the support vectors (the coefficients for the dual problem).

Using these attributes you can obtain the alpha values from the absolute value of the `dual_coef_` values. With this information from the support vectors and the plots, answer the following:

1. How many support vectors are there for each class (-1 and 1), for each value of C?

[1] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

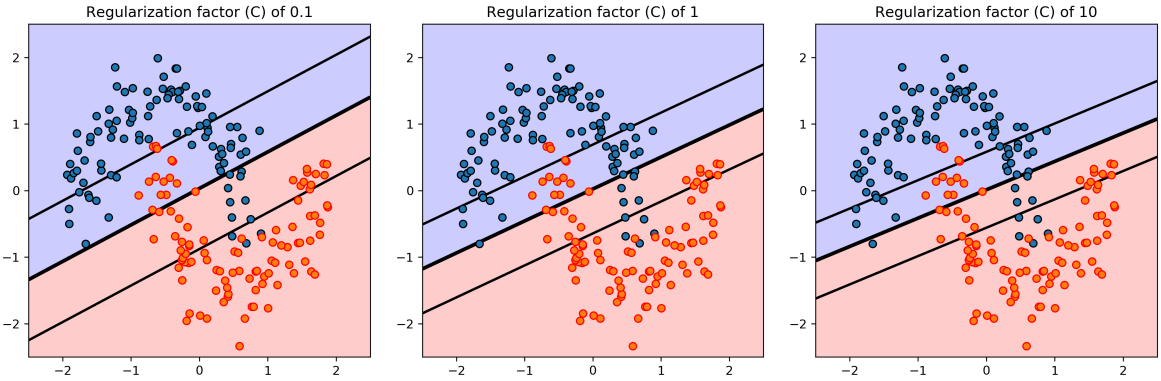


Figure 4.1: In these images, the thick black line is the discriminant and the thin lines are the margins.

2. How many of those support vectors correspond to examples at the margin or examples that violate the margin constraint?

3. Explain the variation of the margins with the value of C.

4. Explain the variation in the number of support vectors that violate the margin constraint as you vary the C value.

4.2 Exercise 2: Kernel trick

Load the data (`T4data.txt`), standardize it and split it one half for training and one half for test. Keep the test set for the final exercise and use the training set train SVM classifiers with C=1 and using the following kernels:

1. Polynomial, $K(x, y) = (\gamma x^T y + r)^d$, with degree 3, gamma of 0.5 and r of 0 (the r value is set in the `coef0` parameter and is 0 by default).

2. Sigmoid, $K(x, y) = tanh(\gamma x^T y + r)$, with gamma of 0.5 and r of -2.

3. Gaussian RBF, $K(x, y) = exp(-\gamma ||x - y||^2)$, with gamma of 0.5.

Modify the plot function provided to plot a black cross (`'xk'`) on every support vector that violates the margin constraint and a black dot (`'.k'`) for every support vector in the margin. The images should look like Figure 4.2. Explain why the number of support vectors on the margin differ between the different kernels.

Exercise 2, optional

Using a value of 0 for the r parameter in the polynomial kernel is not a good idea in this case. Repeat the exercise above for the polynomial kernel but set $r = 1$ using the `coef0` argumment of the `SVC` class.

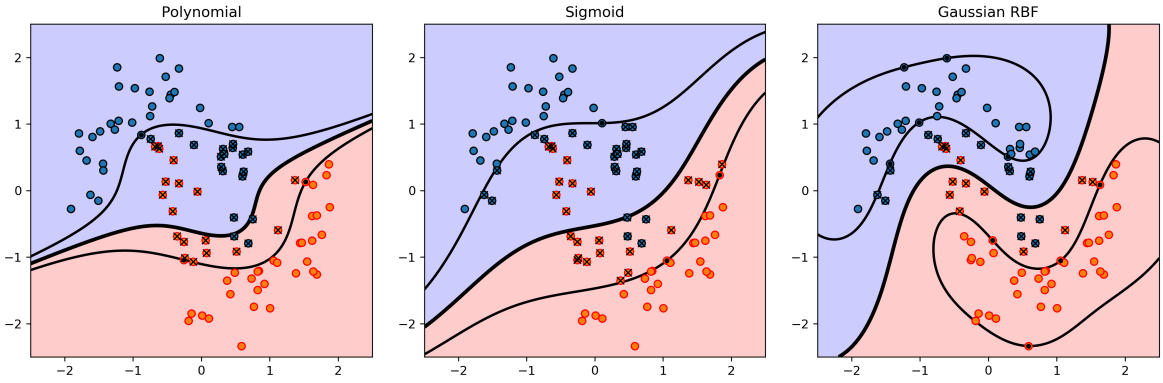


Figure 4.2: Comparing three different kernels. The thick black line is the discriminant, the thin lines are the margins; support vectors are marked with crosses (inside margins) or dots (on the margins)

4.3 Exercise 3: Optimize kernel parameters and compare classifiers

Use the training set from the previous exercise to optimize, with cross validation, the regularization parameter C for each of the three kernels above. Keep the remaining parameters constant, changing only the value of C, starting from 0.1 and doubling at each generation until greater than 10000. Plot the training and validation errors against the logarithm of C for each kernel and then train the SVM, for each kernel, with the best C. The result should look something like Figure 4.3. After optimizing the regularization for each kernel, use the test set to determine which classifiers perform significantly differently. Use the approximate normal and the McNemar tests as explained in Lecture 7 (and chapter 7 of the lecture notes). You can use the `predict` method of the SVM to obtain the predicted labels for the test set.

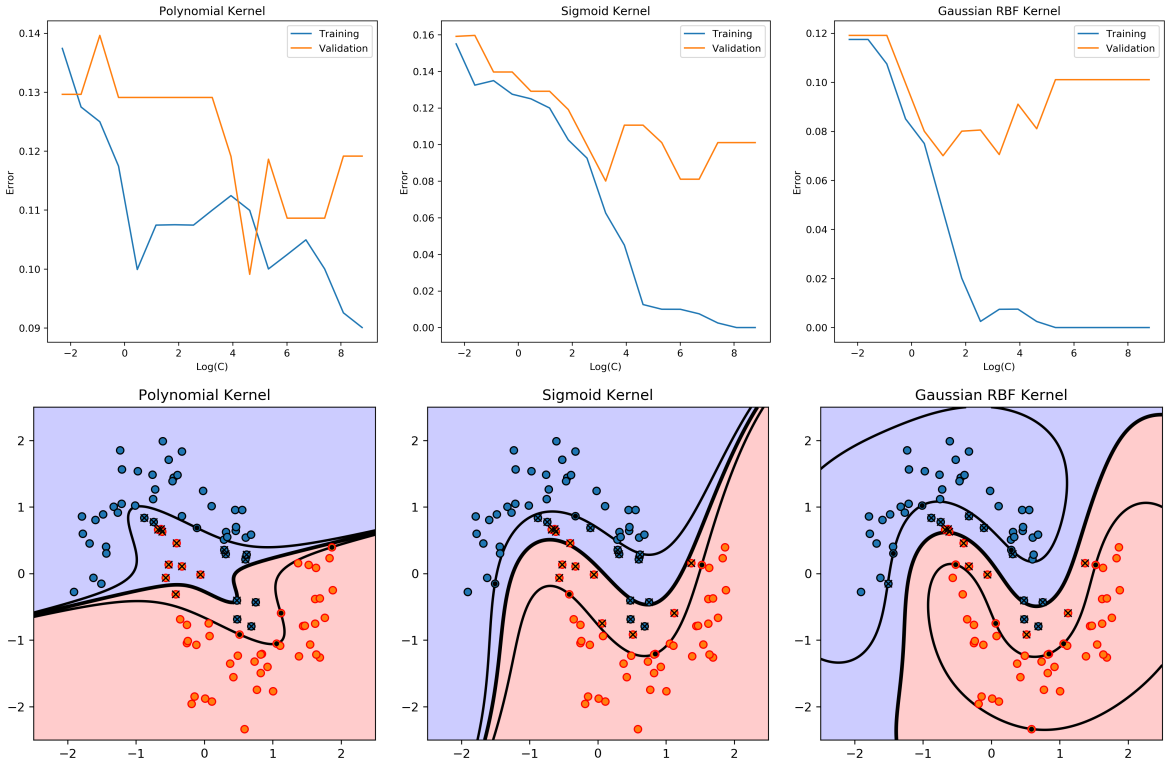


Figure 4.3: Optimizing the regularization parameter C for each kernel

Chapter 5

Bias, Variance and Multiclass Classification

Computing Bias and Variance with Bootstrapping. Classification with more than two classes.

Download the `T5-data.txt` and the `T5_plot.py` file to your working folder. The first file contains a synthetic data set for a univariate regression and the second file contains auxiliary plotting functions for multiclass classification.

5.1 Exercise 1: Bias and Variance for polynomial regression

Load the data in the `T5-data.txt` file. This is a set of 200 (x, y) values, separated by a space, so you can use the `numpy.loadtxt('T5-data.txt')` function with default parameters. Shuffle the data and split it into two sets of 100 points, for training and testing.

Create 500 replicas of the training set. To do this, create a 3D matrix with 500 layers, 100 lines and 2 columns, and fill each layer with a random sample of the training set. To obtain 100 points at random from the training set you can do something like this:

```
for sample in range(500):
    ix = np.random.randint(100,size=100)
    replicas[sample,:] = training_data[ix,:]
```

So, for each of the 500 replicas, we will generate a vector of 100 integer values, each chosen at random between 0 and 99 (the `randint` function always returns values less than the first argument). Then we copy into the corresponding replica the rows of the training data matrix corresponding to the randomly generated indeces.

After creating the 500 replicas, create a function that receives the degree of a polynomial, the matrix with all the replicas and the test set and returns the Bias and Variance computed on the test set. To do this, first create a matrix of 500 rows and 100 columns and fill it with the predicted values for the test set that are obtained by polynomials trained on each of the replicas. Something like this:

```
for ix in range(replicas.shape[0]):
    coefs = np.polyfit(replicas[ix,:,0],replicas[ix,:,1],degree)
    predictions[ix,:]=np.polyval(coefs,test[:,0])
```

Note that this should be in a function to make it easier to loop through different degrees of the polynomial and plot the bias and variance values.

After obtaining all predictions, compute the bias, which is the mean squared error between the average prediction and the true value. This can be computed in this manner:

```
mean_predictions = np.mean(predictions,axis=0)
bias = np.mean((test[:,1]-mean_predictions)**2)
```

The variance of the predictions can be computed as the mean of the difference between the mean of the squares of the predictions and the square of the means of the predictions:

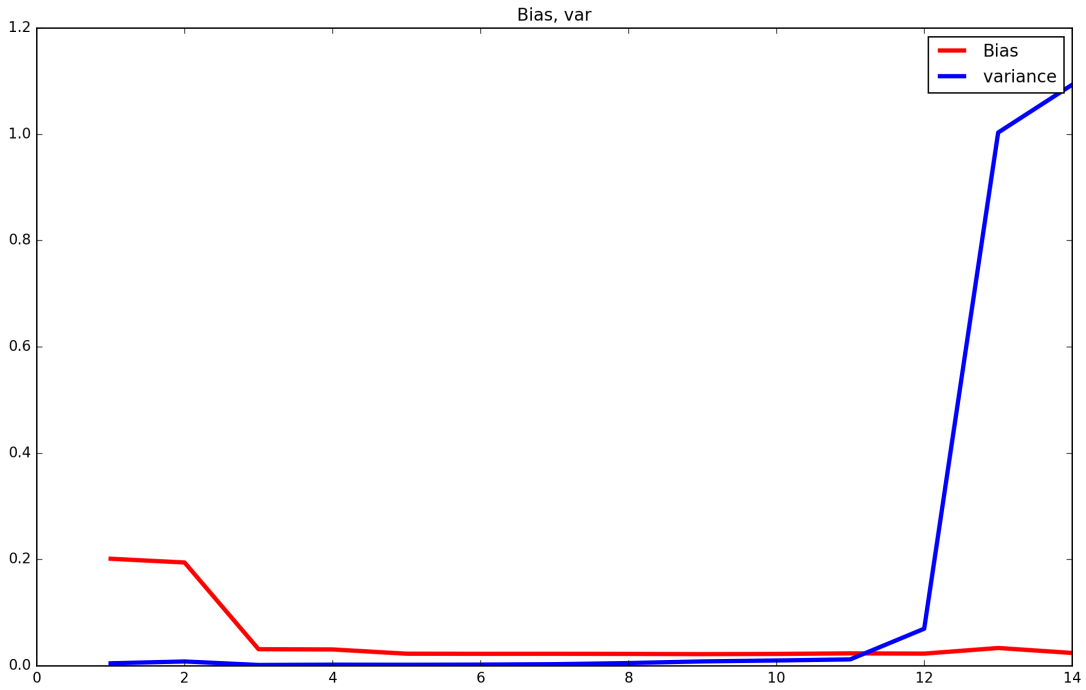
$$E\left((y-\bar{y})^2\right) = \bar{y^2} - \bar{y}^2$$

This can be done by first computing the square of the predictions, then the mean of the squared predictions and subtracting the means of the predictions, squared:

```
pred_square = predictions**2
mean_square = np.mean(pred_square,axis=0)
var = np.mean(mean_square-mean_predictions**2)
```

Write a function that computed the bias and variance given the predictions of the text values for hypotheses trained over all replicas. Make sure you understand these formulas (consult lecture 11 as needed).

Finally, loop through polynomial degrees from 1 through 14 and plot the bias and variance computed. The result should be something like this



5.2 Multiclass classification

From the `T5_plot.py` import the `plot_logregs` function. This is an auxiliary function for plotting the combination of three logistic regression classifiers you will train for a one-vs-rest classification of

the Iris data set. To load the Iris data set, import the datasets module from the ScikitLearn library and create your data matrices with only the sepal length and width attributes:

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data[:,[0,1]]
Y = iris.target
```

This way, the `X` variable will be a matrix with the two selected attributes and the `Y` variable will contain the class labels for the three variants of flowers (*Setosa*, *Versicolor* and *Virginica*).

Create a list of three `LogisticRegression` classifiers with low regularization (use a large value for C) and train each classifier so that class 0 corresponds to one of the three flower variants and class 1 corresponds to the other two. This is easy to do with a loop over the three class labels (0, 1, and 2), and in each iteration creating a class vector with values of 0 for all examples with the class corresponding to the iteration and values of 1 for examples with a different class. This should give you 3 logistic regression classifiers trained to distinguish each class from the remainder.

Plot the Iris data and the classifications of your 3 classifier using the one-vs-rest algorithm. This is implemented on the `plot_logregs`, which simply requires the list of trained classifiers as argument and creates a `iris-logistic.png` file with the result. Look at the `plot_logregs` function to see how this is done. Note in particular this part:

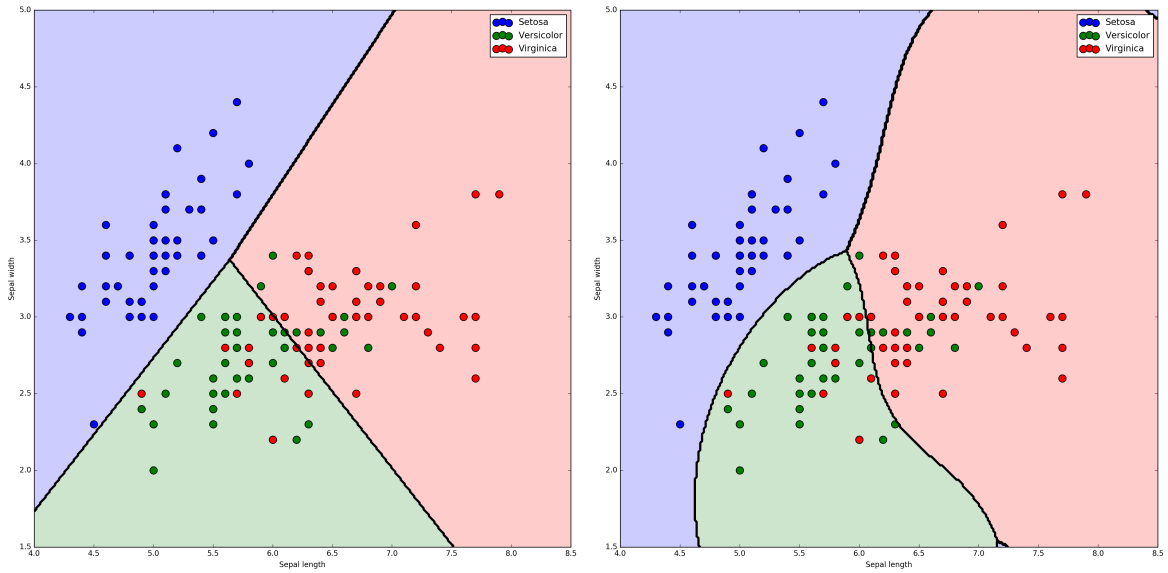
```
def plot_logregs(logregs,file_name='iris-logistic.png'):
    ...
    max_z = np.max(Zs,axis=0)
    Z = np.zeros(Zs[0,:,:].shape)
    for ix in range(3):
        tmp = Zs[ix,:,:]
        Z[tmp==max_z] = ix
```

Noting that the Z values are the probabilities of the points being in class 0, try to understand how the one-vs-rest classification was implemented here.

Repeat this exercise using the `OneVsRestClassifier` class provided in the `sklearn.multiclass` module. You can create a one-vs-rest classifier object by providing one classifier object to use for each classification. For example, to use support vector machines with a Gaussian kernel with a γ of 0.7 and a C value of 10 for regularization:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
ovr = OneVsRestClassifier(SVC(kernel='rbf', gamma=0.7, C=10))
```

Import the `plot_ovr` function from the `T5_plot` module and use it to plot your classification using this one-vs-rest classifier, after training it with the Iris data set, to output the result to the `iris-ovr.png` file. The two plots should look like this:



5.3 Questions

After these exercises, you should be able to answer the following questions:

- How do we create replicas with bootstrapping?
- What are the Bias and Variance?
- How do we compute the Bias and Variance for a regression with a squared error loss function?
- What is the one-vs-rest multiclass classification algorithm?

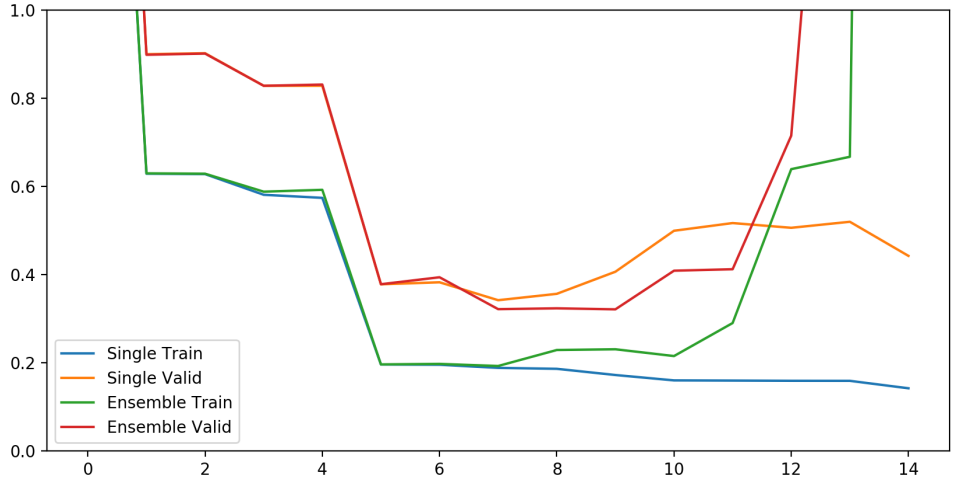
Chapter 6

AdaBoost and Visualization

Bootstrap Aggregating with polynomial regression. Adaptive Boosting with decision tree stumps (stumping). Visualizing multidimensional data.

6.1 Exercise 1: Bagging

The files `T6-train.txt` and `T6-valid.txt` have examples for, respectively, the training and validation sets for a regression problem. Compare single polynomial models of degree varying from 0 through 14 with ensembles of 200 polynomials, of the same degrees, obtained by bootstrap aggregating (bagging). Use the training set for fitting the single polynomials and for creating the replicas for the ensembles and then measure the error on the training and validation sets. The results should be approximately like this:



To compute the training error for the ensemble, average all predictions for the training set and compare the mean predictions with the target values. Do the same for the validation set. The error shown on the plot is the mean quadratic error.

Note that the advantage of using an ensemble of polynomials is small. This is because polynomial regression is fairly stable and thus the different hypotheses are strongly correlated. Bagging works better with less stable models, like regression trees.

6.2 Exercise 2: AdaBoost

Download the `T6-data.txt` to your working folder. The file contains the synthetic data set shown in lecture 12 on the AdaBoost example. For this exercise, you will reproduce the AdaBoost example and create a boosted classifier using decision trees with a depth of 1 (decision stumps).

Start by loading and standardizing the data. Use all the data for training. The class label is the third column; the features are the first two columns. Initialize a weight vector with the same weight of $1/N$ for all data points, where N is the number of points in the data set. Now iterate through 20 cycles of boosting.

For each cycle, fit a `DecisionTreeClassifier` with `max_depth=1` (from the `sklearn.tree` module) to your data using the weights vector in the `sample_weight` field. Computed the weighted error by adding the weights of the examples that were incorrectly classified by this classifier:

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_m^n I(y_m(x^n) \neq t^n)}{\sum_{n=1}^{N} w_m^n}$$

Compute the α value for the classifier:

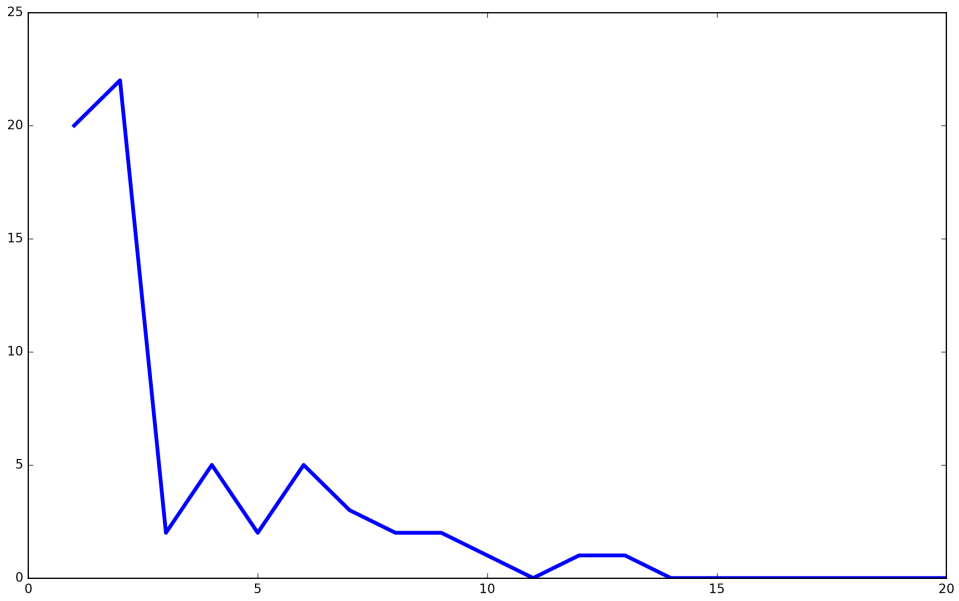
$$\alpha_m = \ln \frac{1 - \epsilon_m}{\epsilon_m}$$

and finally update the weights for the next cycle:

$$w_{m+1}^n = w_m^n \exp\left(\alpha_m I(y_m(x^n) \neq t^n)\right)$$

Note: keep the weights normalised; after computing the new weights divide them by the sum of the weights. This helps simplify the error computation by making $\sum_{n=1}^{N} w_m^n$ equal to 1.

At each cycle, compute the number of misclassified examples of the boosted classifier so far. The boosted classifier is the weighted sum of the individual classifiers, each weighted by its α value. Note that the class labels are -1 and 1, so if the weighted sum is below zero you can consider the example to be classified in class -1, and in class 1 if otherwise. If you plot the errors of the boosted classifier as a function of the number of cycles, it should look like this:



6.3 Tutorial: Visualizing multidimensional data

Follow the examples on visualisation in Lecture 15. Use the `pandas` library and load the `iris.data` csv file as shown in the Lecture slides. Try different ways of visualizing the data. Note that saving the images to files is optional.

- Plot the stacked histograms. Try different values for `alpha`, which controls the opacity of the bars. To do this use the `plot` method of the `DataFrame` object specifying `kind='hist'`.
- Plot the separated histograms, using the `hist` method of the `DataFrame` object.
- Create a box plot for the four features using the `plot` method of the `DataFrame` object specifying `kind='box'`.
- Create Scatter Matrix plots using histograms and kernel density estimation for the diagonals.
- Create parallel coordinates and Andrew's curves plots representing the different classes in different colours.

To clarify any doubts about the details of these exercises, try consulting the Pandas visualization examples at this location: http://pandas.pydata.org/pandas-docs/version/0.18.1/visualization.html

6.4 Questions

After these exercises, you should be able to answer the following questions:

- What makes the training of the different hypotheses change from one to the next in bagging?
- What makes the training of the different hypotheses change from one to the next in boosting?
- What is the final result of the boosting training algorithm and how can you use it to classify new points?

- What are parallel coordinates plots and what are they useful for?
- What information is represented in a box plot?

Chapter 7

Selection, Extraction and Vector Quantization

Feature selection in supervised learning with f-scores. Feature extraction with Principal Component Analysis. Vector Quantization with K-Means clustering.

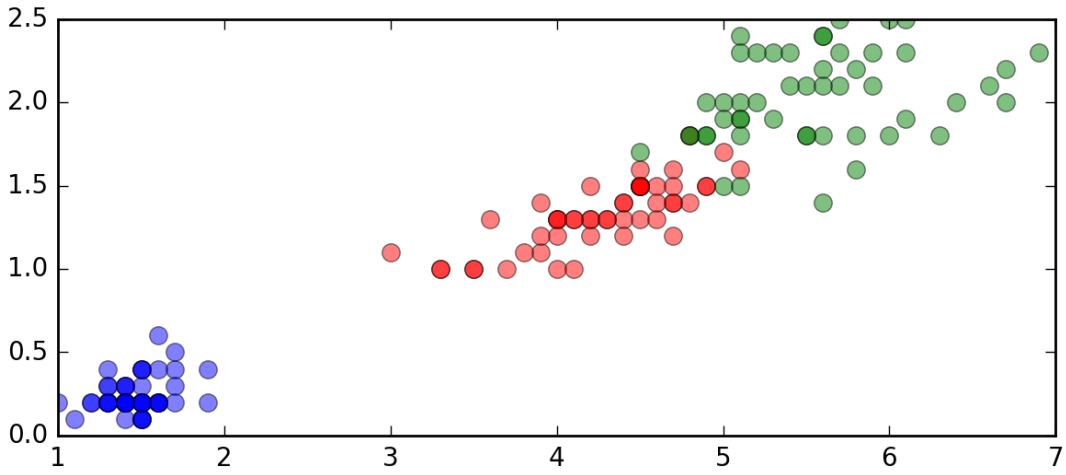
7.1 Exercise 1: Feature Selection

Load the Iris dataset and separate the features and class labels in two different variables. This data set is available from the `datasets` module:

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Now use the `f_classif` function from the `sklearn.feature_selection` the ANOVA F-value for each feature, considering the class labels. Consult Lecture 15. Do the exercise by examining the result (F-values and probabilities) of the `f_classif` function and manually selecting the best features. Optionally, you can repeat this exercise using the `SelectKBest` class, but do not skip the previous steps to make sure you understand how the features are selected.

Plot the data projected on the two selected features. The result should look like this. If you need help reproducing this plot, check the last section on this chapter for the plotting function used in this plot.

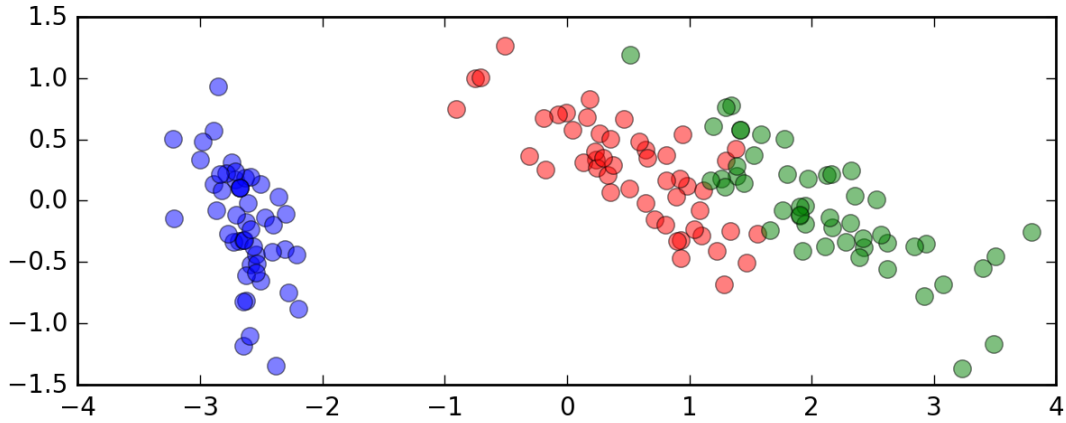


7.2 Exercise 2: Feature Extraction with PCA

Feature extraction is the process of computing features from the initial data. This can be a complex problem and is often very dependent on the data. It is used in image processing, sound processing, stream data such as event logs and so on. In this exercise we will look only at Principal Component Analysis, a generic method of dimensionality reduction which computes new features from linear combinations of existing features while maximizing the preserved variance in the data set. This can be used in unsupervised learning because it relies solely on the feature values.

Import the Iris dataset as before. Note that the Iris values are all lengths in millimetres and we want to find the axes of largest variance. Thus, in this case, we should not standardize or normalize the data because doing so will distort its shape. In unsupervised learning, we often need to be careful about how we transform the data because the shape of its distribution and the distances between the points may be important.

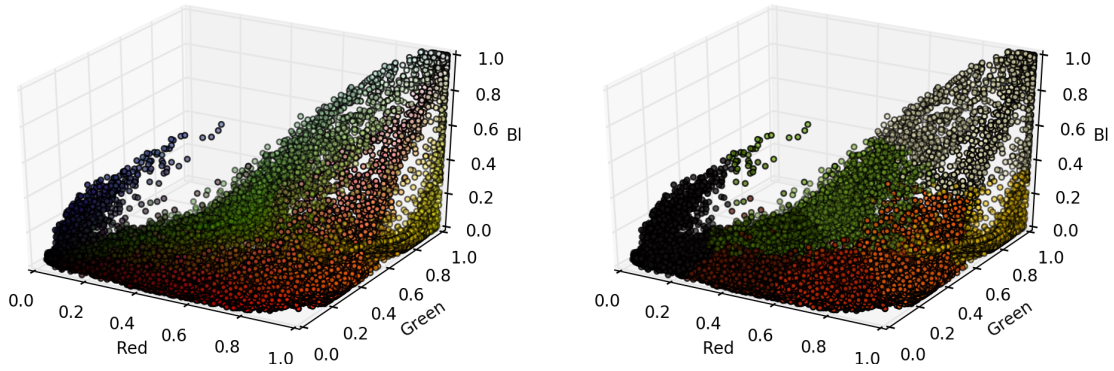
Use the `PCA` class from the `sklearn.decomposition` module to transform the Iris data into a new set of two features, corresponding to the two largest principal components. The result should look like this.



7.3 Exercise 3: Vector Quantization

Download the `vegetables.png` image file to your working folder[1]. Note that the image used in the example of Lecture 17 only has the red and green components, so all colour values could be represented easily in a 2D-plot with red and green as the axes. This exercise will use a more realistic image with all colour components, so the corresponding 3D plot is a little more complex. In this exercise we are going to compute k centroids to the 3D colour space with the K-Means clustering algorithm, and then convert each pixel into the closest centroid to compress the colour space reducing the number of different colours in the image. This is called vector quantization because we are converting vectors in a continuous space into a finite set of values (quantities).

The figures below show the original colour space and the compressed colour space with $k = 8$, with the point coordinates corresponding to the original colour.



In the Appendix section you can see the code for the 3D plot, but it is not necessary for you to do it. In this exercise we will just load the images, convert the colours and save the images.

We start by loading the image using the `imread` function from the `skimage.io` module. We will also need the `imsave` function to save the images computed at the end. These functions simplify the conversion between image files and Numpy matrices. Note that the image is coded with one byte per colour, with an integer range of 0-255. We will convert this into floating point values between 0-1 by dividing the matrix by 255.0.

```
import numpy as np
from sklearn.cluster import KMeans
from mpl_toolkits.mplot3d import axes3d
from skimage.io import imsave,imread


img = imread("vegetables.png")
w,h,d = img.shape
cols = np.reshape(img/255.0, (w * h, d))
```

The reshape at the end places all the points into a matrix of colours with 3 columns. This is the matrix we will use for the K-Means quantization.

Now use the `KMeans` class from the `sklearn.cluster` module to compute 64 centroids and then convert all the colours to the nearest centroid. You can get the centroid positions from the `cluster_centers_` attribute of your K-Means object and the labels from the `predict` method. Look up the example on Lecture 17 and the documentation on the K-Means class. To convert the colours you just need to assign the nearest centroid values to the pixel colour values. For example:

[1] CC Jane Fresco, source https://en.wikipedia.org/wiki/File:Colorful_Photo_of_Vegetables.png

```
c_cols = np.zeros(cols.shape)
for ix in range(cols.shape[0]):
    c_cols[ix,:]=centroids[labels[ix]]
```

Finally, reshape the colours matrix into an image matrix and save it. For example:

```
final_img = np.reshape(c_cols,(w,h,3))
imsave('image.png',final_img)
```

Where the `w` and `h` variables store the width and the height of the original image (see the first code snippet in this section).

The figures below show the comparison between the original image (left), the 64 colours compressed image (centre) and the 8 colours compressed image (right)

7.4 Appendix: plotting code

This is the code for the plotting function used in the Iris exercises.

```
def plot_iris(X,y,file_name):
    plt.figure(figsize=(7,7))
    plt.plot(X[y==0,0], X[y==0,1],'o', markersize=7, color='blue', alpha=0.5)
    plt.plot(X[y==1,0], X[y==1,1],'o', markersize=7, color='red', alpha=0.5)
    plt.plot(X[y==2,0], X[y==2,1],'o', markersize=7, color='green', alpha=0.5)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.savefig(file_name,dpi=200,bbox_inches='tight')
    plt.close()
```

This is the code for plotting the 3D representation of the colour space in the vector quantization exercise. Note that this is a very large plot because each plot in the chart corresponds to a pixel in the image. It can take a few minutes to complete and requires up to 500MB of RAM during execution. The `cols` variable contains the colour matrix for the pixels.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(cols[:,0], cols[:,1], cols[:,2], c=cols,s=10)
ax.set_xlabel('Red')
ax.set_ylabel('Green')
ax.set_zlabel('Blue')
ax.set_xlim3d(0,1)
ax.set_ylim3d(0,1)
ax.set_zlim3d(0,1)
plt.savefig('T7-veg_rgb.png',dpi=200,bbox_inches='tight')
```

7.5 Questions

After these exercises, you should be able to answer the following questions:

- What is the purpose of feature selection?
- How do feature selection and feature extraction differ?
- What is the result of Principal Component Analysis?
- How is the result of the K-Means algorithm?
- How can we use K-Means for vector quantization?

Chapter 8

Comparing clustering algorithms

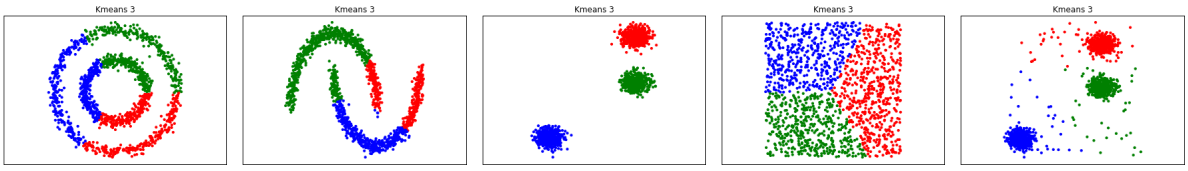
Plotting the results of different algorithms.

8.1 Exercise

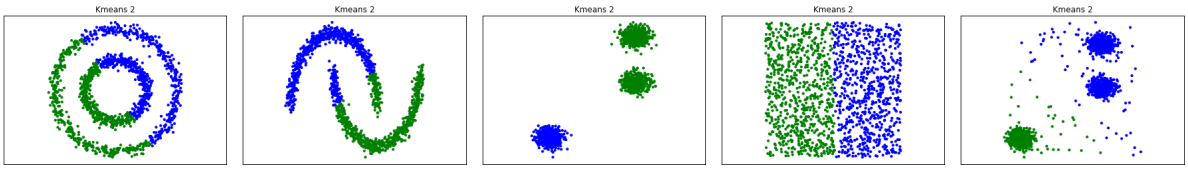
Start from the script provided (`t8_clusters.py`). The initial part of this script has the code to generate the different data sets you will be using to experiment with different clustering algorithms. The datasets are in the list `datasets`, and each dataset is a matrix with two features and 1500 examples.

Use the `plot_clusters` function to plot your labelled clusters for each algorithm. Try the following, using the default parameter values except where otherwise indicated. All clustering algorithms are in the `cluster` module, except the `GMM` (Gaussian Mixture Models), which is in the `mixture` module.

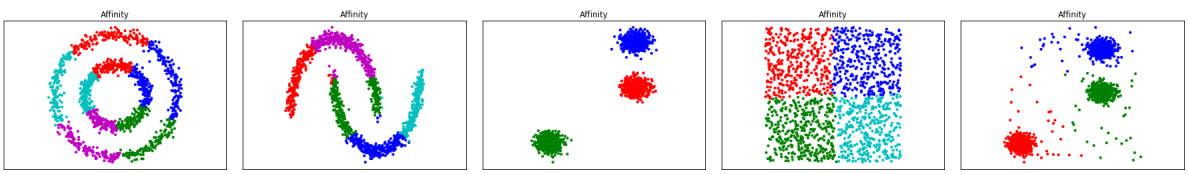
- `KMeans` with `n_clusters=3`



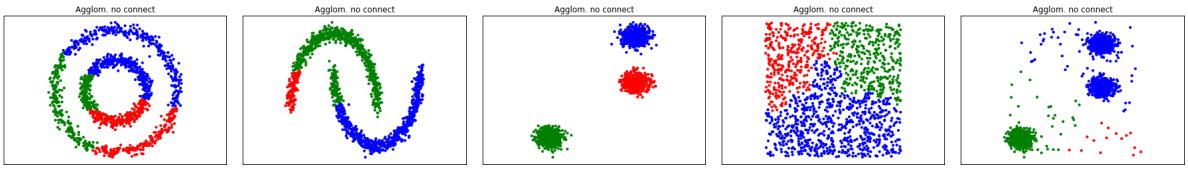
- `KMeans` with `n_clusters=2`



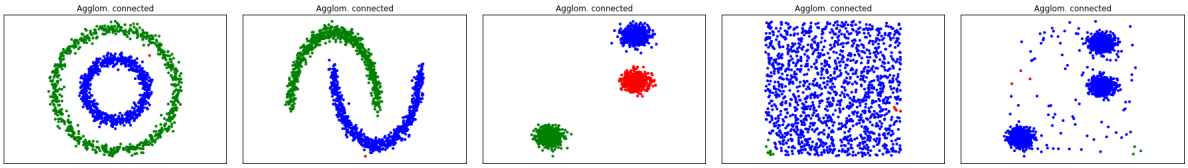
- `AffinityPropagation` with `damping=0.9` and `preference=-200`



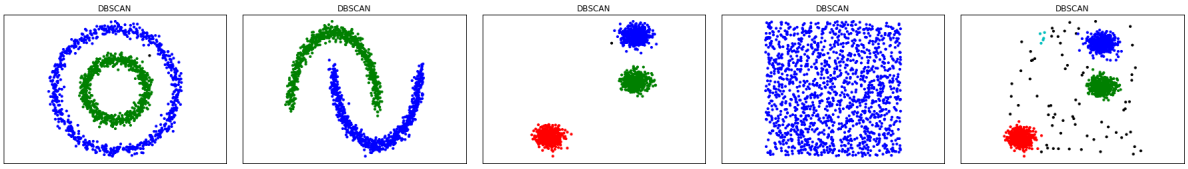
- `AgglomerativeClustering` with `linkage='average'` and `n_clusters=3`



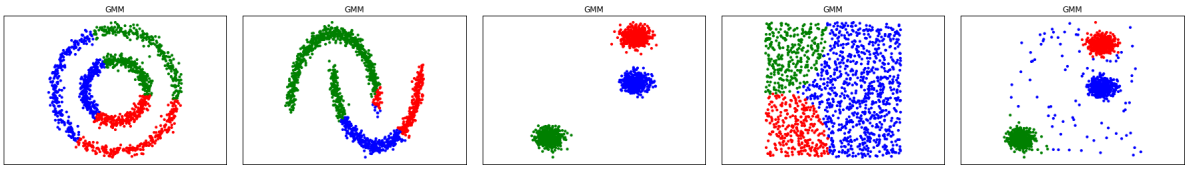
- `AgglomerativeClustering` with `linkage='average'` and `n_clusters=3`, plus `connectivity` set to a connectivity matrix connecting the 10 nearest neighbours. Use the `kneighbors_graph` to obtain the connectivity matrix.



- `DBSCAN` with `eps=0.2`.



- `GMM` (from the `mixture` module) with `n_components=3` and `covariance_type='spherical'`.



8.2 Questions

After these exercises, you should be able to answer the following questions:

- Can the K-Means algorithm adapt the number of clusters to the data?
- Can the affinity propagation algorithm adapt to the number of clusters in the data?
- What is the function of the connectivity matrix in the Agglomerative Clustering algorithm?
- How does DBSCAN deal with points in low density regions?

Bibliography

[1] Sanford Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.