LINUX VOICE TUTORIAL

DATA ANALYSIS USING PYTHON AND MYSQL

BEN EVERARD

Graphing data makes it easier to understand, and graphing lots of data is easy with a script and a database.

WHY DO THIS?

- Pull out the information that's pertinent to you from a swarming mass of numbers.
- Improve your Python and SQL skills.
- Get your computer to draw pretty pictures that make you seem smart to friends, family and co-workers.

In recent years, governments around the world have been opening up their information archives to the public, and now there's more data available than ever before. However, the raw data is hard to digest, and it's often analysed by people with an agenda, whether that's newspapers trying to make a story sound exciting to sell more copies, or a company trying to make their product look better than the competition. It's hard to know whether data is being properly represented, so the solution is to dive in and analyse the figures for yourself. Let's take a look at how to do this using UK house prices.

You can get a complete list of every house sold in the UK along with its location, type (eg terrace, semi-detached) and price from **data.gov.uk**. The data goes back to 1994, and is licensed under the Open Government Licence, which is allows us to manipulate the data and publish it – so that's what we'll do.

Spreadsheets, such as *LibreOffice*'s *Calc*, can easily handle small data sets. However, this data set is too big and needs something a little more capable. We're going to use Python and *MySQL*, though you could use most programming languages and most databases for the task.

The data comes in a CSV file, which is a text file containing the values separated by commas. These are usually used with spreadsheets, but are also fairly easy to upload into databases. Databases enable us much better access to the data from programming environments, and can also handle much larger data sets than spreadsheets.

First you need to grab the software we'll be using. That's *MySQL* (both a client and server), and two Python modules (*MySQLDB* and *Matplotlib*). These are all quite common, and should be in your package manager. To get them in Debian-based systems, use:

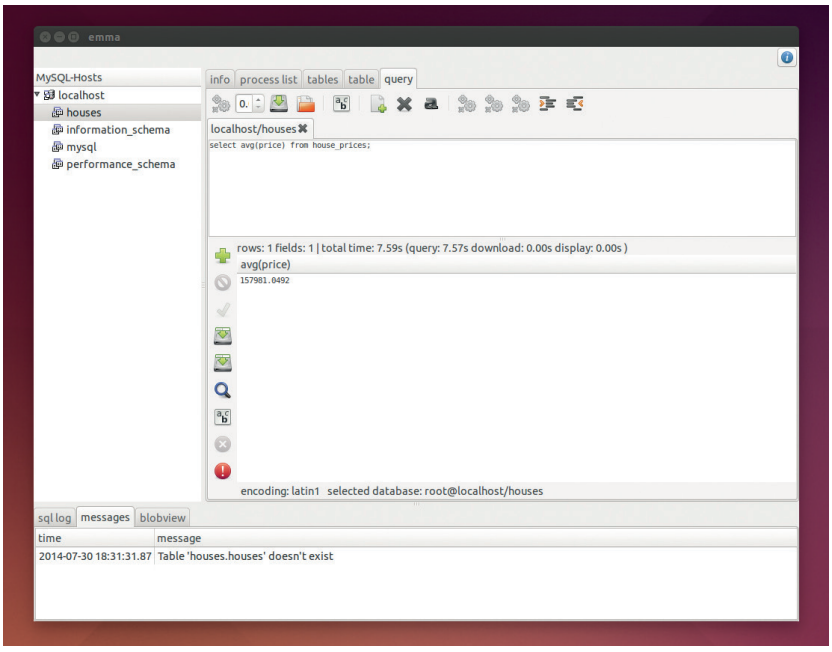
```
sudo apt-get install mysql-client mysql-server python-mysqldb
python-matplotlib
```

If your package manager hasn't asked you to set up a root password for *MySQL*, you can do that now with:

```
sudo mysqladmin -u root -p password newpass
```

Replace **newpass** with a password of your choice.

If you're using SQL for more than a few basic queries, there are some SQL clients (such as *Emma*, shown here) that can make your life a little easier.



Get the data

Now you've got the software, you just need to grab the data. The easy way to do this is to download our database dump from **www.linuxvoice.com/house-price-analysis**.

This is an xzipped SQL file, so you can load it with:

```
unxz house_prices.sql.xz
mysql -u root -p < houseprices.xz
```

This will create a database called **houses**, and a table within it called **house_prices** that contains all the information we're going to work with.

That's the easy way. The hard way (which you'll need to do if you want to load data other than UK house prices), is to download the raw CSV files and load them into *MySQL*. This isn't too hard, but it can be a little fiddly.

First you need to get the CSV files. The ones we've been using are from **data.gov.uk**. However, there are loads of sources of open data you may wish to use (see the boxout over the page for more details). CSV files are often created with Windows encoding rather than Unix. There's a utility called **dos2unix** that can fix this, which you use with:

```
dos2unix <filename>
```

MySQL is really designed as a server tool, not a desktop one. This means that it has a few security features that you may not expect. One such feature is that by default, it won't usually load local files. You can get around that by starting the client with the **--in-file** flag:

```
mysql --u root -p --in-file
```

This will drop you into the *MySQL* commandline. First you need to create a new database to use:

```
create database houses;
```

```
use houses;
```

Now you need to create a new table to store the data. This has to have the same layout as the CSV files that you want to upload. For example:

```
create table house_prices (id varchar(50), price int, date
datetime, postcode varchar(10), type varchar(1), newbuild
varchar(1), leasefree varchar(1), address1 varchar(50), address2
varchar(50), address3 varchar(50), address4 varchar(50),
address5 varchar(50), address6 varchar(50), address7
varchar(50), dontknow varchar(1));
```

With all this set up, you can load the files with the following SQL statement:

```
load data local infile "file_name.csv" into table house_prices
fields terminated by ',' enclosed by '"';
```

The UK house price data comes in separate files for each year. You can use the **cat** command to join them together into one big file, or import them individually (which makes it easier to identify problems).

Getting started with SQL

Now you've got everything in the database, you can use SQL to pull out the information you want.

The basic usage of SQL to pull information out of a database is in the form:

```
select <something> from <table> where <condition>;
```

This is quite simple, but it enables you to get almost anything you need from the data store, and gives you a quick way of getting data (although complicated queries on large bodies of data can be slow).

For example, to get all of the price and house numbers for a particular postcode, you can use:

```
select price, address1 from house_prices where postcode = "XX1
1XX";
```

where **XX1 1XX** is the postcode. As well as getting specific bits of data, you can aggregate it using functions such as **avg()**, which returns the average.

For example, the following line returns the average price for houses in Bristol:

```
select avg(price) from house_prices where address6 =
"BRISTOL";
```

You'll see a few more SQL techniques as we go through the article, but they all follow this same basic process. If you're unsure of anything, *MySQL* has excellent documentation at **dev.mysql.com/doc**.

Drawing pictures with Python

SQL is great for pulling out bits of information, but it's not great at combining and comparing it. That's were Python comes in. We're going to use it to compare and graph the information we pull out of *MySQL* to make everything easy to understand.

In this case, our Python program will be acting as a glue between a module that access the database and a module that outputs graphs. Let's first look at *MySQLdb*, which we'll use to access the database.

Using the *MySQLdb* module is a fairly straightforward process. You have to connect to the database, and then create a cursor object. This cursor can then be used to execute queries and fetch the results. Take a look at the following example, which prints out the average house price in the data set.

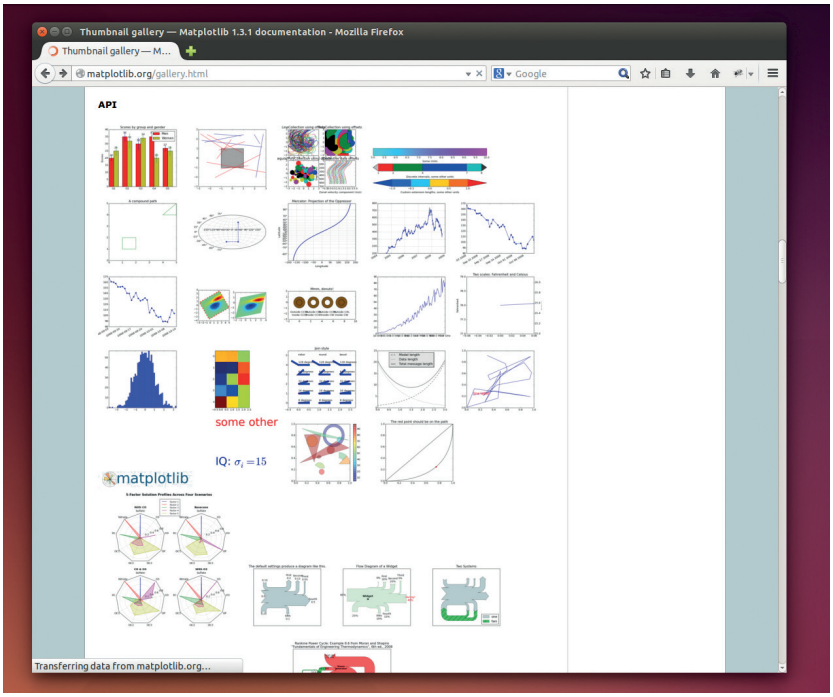
```
import MySQLdb

db = MySQLdb.connect(host="localhost", user="root",
passwd="xxxx", db="houses")

cur = db.cursor()
cur.execute("select avg(price) from house_prices;")

result = cur.fetchone()
print str(result[0])
```

You'll need to change the password and possibly user in the **connect** command, depending on how your database is configured.

Once the connection to the database is set up, you can call **execute()** with a string containing an SQL query, and then get the result with **fetchone()**. This returns a tuple containing an entry for each column returned by the SQL (in this case, there's just one). If you expect the query to return more than one result, you can loop through them with:

```
for row in cur.fetchall():
        #do what you need to here
```

Since you just need to pass a string to **cur.execute()**, you can build this up with the usual Python tools. For example, if you want to get the average



The *MatPlotLib* project maintains a gallery of different chart types, and examples of how to use them at **http://matplotlib.org/gallery.html**.

MariaDB

We decided to do this tutorial using *MySQL*, because it's probably still the most widely used database for Linux. However, we know that a lot of people aren't happy with Oracle's handling of the project, and so may wish to use *MariaDB* instead, a fork of *MySQL* led by the original creator of *MySQL*, Michael "Monty" Widenius.

It should be completely compatible with *MySQL*, and so if you'd rather use this database, you should be able to follow along with this tutorial without any problems.

Big data and NoSQL

Big data is one of the industry's current buzzwords. Like most tech buzzwords, there aren't any hard-and-fast rules to define it, but loosely speaking, it refers to any chunk of data that's too big to process on an ordinary computer, meaning you need some special setup to handle it efficiently. That could be a high-powered server, or a cluster of servers.

It is possible to use SQL databases to handle huge data sets, but specialist tools have sprung up to make it easier, and one common type is the so-called NoSQL variety of database. These are databases that don't use tables to hold structured information; instead they hold all the data in one non-structured mass. This means that for some processes, they can be quicker than SQL databases, and it can be easier to share the load across many machines. They tend to process data using the **map-reduce** method, which goes through each item in turn and maps it to a value. These values can then be combined (or reduced) to a result.

The data set we've used here is 19 million items big. We've certainly heard people calling much more mundane analyses than this big data, but in our view, it doesn't qualify. *MySQL* handles the task perfectly well, and it's a technology that's far more useful in most circumstances than NoSQL.

However, if you happen to be in the job market at the moment, NoSQL is one of the hottest skills around (according to **www.indeed.com/jobtrends**, *MongoDB* – a NoSQL database – is the second hottest skill to have after HTML5).

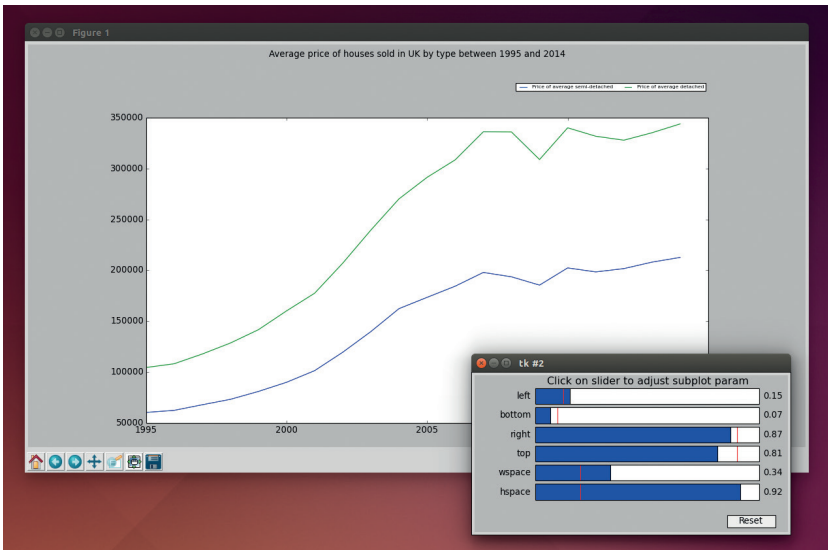
prices for a few different counties, you could use:

```
for county in ['GREATER MANCHESTER','GLOUCESTERSHIRE']:
        query = "select avg(price) from house_prices where address7 = '" + county + "';"
        cur.execute(query)
        result = cur.fetchone()
        print "Average house price in " + county + " : " + str(result[0])

```

Alternatively, you could see how the house prices have changed over the 20 years we have data for using the following. You'll need to include the previous code to connect to the database as well.

```
years = range(1995, 2015)
data = []

for year in years:
        query = 'select avg(price) from house_prices where data between '" + str(year) + '-01-01" and '" + str(year) + '-12-31";'
        cur.execute(query)
        result = cur.fetchone()
        print str(year) + " : " + str(result[0])
        data.append(int(result[0]))
```

If you're an SQL user, you'll probably notice that this could be done in a single query. We've done it this way to make the code a bit easier to follow.



You can change some parameters of the figure after it's created using the Configure Subplots button (second from the right).

This code stores the data in a list as well as printing it on a screen. This list (rounded to whole numbers), can be used to create graphs. One option is to output it to a file in CSV format. CSVs can be loaded into most spreadsheets (such as *LibreOffice Calc*), and from there you can generate any graphics you need. This can be a good way to experiment with different types of graph, because it enables you to quickly try various visualisations on the data. However, it's bad if you need to produce lots of graphs based on the data, because it requires quite a bit of manual intervention. For this, it's much easier to use the *MatPlotLib* module to automatically draw any charts you want.

Get Matplotlib

To use this, you'll need to import it. We'll pull it in with *pylab*, which provides some other functions as well as chart drawing. You'll need to add the following to the start of your program:

```
from pylab import *
```

The following two lines can then be added to the end of the previous program to plot the data, and show the chart:

```
plot(years, data)
show()
```

This is the most basic use of the plotting module, and it can do far more than this. Let's take a look at a slightly more complicated example. This time, we'll see how the average price of houses has changed for detached and semi-detached houses. First we need to pull the appropriate information from the database with the following code (this will also need the code to connect to the database):

```
def get_value(cur, query):
        cur.execute(query)
        row = cur.fetchone()
        return int(row[0])
val_of_semi = []
val_of_detatched = []
years = range(1995, 2015)
for year in years:
        query = 'select avg(price) from house_prices where data between '" + str(year) + '-01-01" and '" + str(year) + '-12-31" and type="S";'
        val_of_semi.append(get_value(cur, query))
```

```
        query = 'select avg(price) from house_prices where
data between "' + str(year) + '-01-01" and "' + str(year) + '-12-31"
and type="D";'
        val_of_detatched.append(get_value(cur, query))
```

Now you have two lists; you just need to put them in the plot. The following code does this:

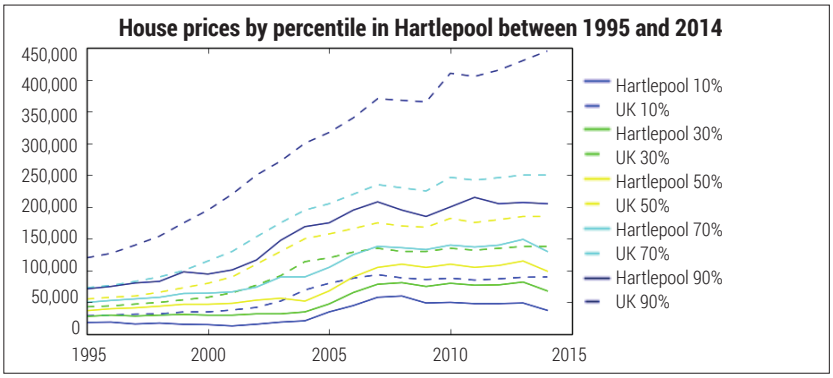
```
fig = figure()

fig.set_size_inches(10,4,forward=True)
ax = subplot(111)
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width, box.height*0.80])
semi_line = ax.plot(years, val_of_semi, label="Price of average
semi-detached")
detached_line = ax.plot(years, val_of_detached, label="Price of
average detached")

ax.legend(bbox_to_anchor=(0., 1.02, 1., .102), ncol=2,
prop={"size":7})
suptitle('Average price of houses sold in UK by type between
1995 and 2014')

show()
```

First, this code creates a figure, and resizes it to 1000 pixels by 400 pixels (it defaults to 100 pixels per inch). The parameter **forward=True** allows you to re-size the window.

Instead of just calling **plot()** like we did in the previous example, this time we create a subplot and shrink it down to 80% of its original height. This gives us space to put a title and legend above it.

The value returned by **plot()** is a line object that we can manipulate to alter the way the line will be displayed. Although we don't do it in this example, you can use this object to alter the way they're displayed.

For example the following (placed before **show()**) would make the lines red and green (by (r,g,b) values),



Hartlepool (among other towns and cities) hasn't seen the same rise in house prices as south-eastern England. See **www.linuxvoice.com/house-price-analysis** for the rest of our analysis.

and dashed (**linestyle "--"**).

```
setp(semi_line, "color", (1,0,0))
setp(detached_line, "color", (0,1,0))
setp((semi_line, detached_line), "linestyle", "--")
```

Other line styles are **"-"** (solid line), **":"** (dotted), and **"-."** (dash-dot). You can also use **setp** to change the alpha (transparency) settings. In fact, there is a mind-boggling set of different options you can set to make the graph look exactly how you want. If you want to create your own graphs, it's best to spend a little time perusing the set of examples at **http://matplotlib.org/gallery.html** to see what's available.

Once you've got everything for the subplot organised, you need to make sure your graph is labelled properly. Adding a title is easy, as you can see in the above call to **suptitle()**. Adding a legend is a bit more complex, because positioning in *Matplotlib* is something of a dark art.

If you want to save figures rather than just displaying them, you can use:

```
savefig('filename')
```

There are loads of ways you can drill down to almost any level of detail, and pull out whatever you want. Of course, this does require an ability to program, and the time to do it.

The end goal, of course, isn't to draw pretty pictures, but to get a better understanding of what the data means. In this case, we've been looking at how the prices of houses have changed over the past 20 years. We won't tell you exactly how to do this because it would defeat the point of this tutorial (which is to learn how to analyse the data for yourself), but we looked into how the house prices changed across different locations and different values of house.

You can see our results at **www.linuxvoice.com/house-price-analysis**. This challenges the view that house prices are rising in the UK. In fact, our analysis shows that in most places house prices are quite static, but that rapid rises in London are pushing the average price up across the UK, distorting the picture. Don't take our word for it though. Dive into the data and see what it tells you.

Data sources

There are loads of other sources of data that are crying out for analysis. Here are a few places to start looking:

- **Data.gov.uk** The official source of all UK government data (this is where the housing data for this article comes from).
- **www.data.gov** The US government's data sets.
- **bitly.com/bundles/bigmlcom/i** A bundle of links to the data websites for many governments from around the world.
- **data.worldbank.org** The world bank publishes financial data on the state of the world economy.
- **epp.eurostat.ec.europa.eu** Eurostat is the directorate general of the European Commission, and is responsible for compiling and publishing statistics about the European Union.
- **www.eea.europa.eu/data-and-maps** The European Environment Agancy publishes a lot of data about the state of Europe.
- **aws.amazon.com/datasets** A list of some of the most popular data sets from around the world.
- **www.reddit.com/r/datasets** A subreddit dedicated to seeking out data on all topics.

Ben Everard is the co-author of the best-selling *Learn Python With Raspberry Pi*, and is working on a best-selling follow-up called *Learning Computer Architecture With Raspberry Pi*.