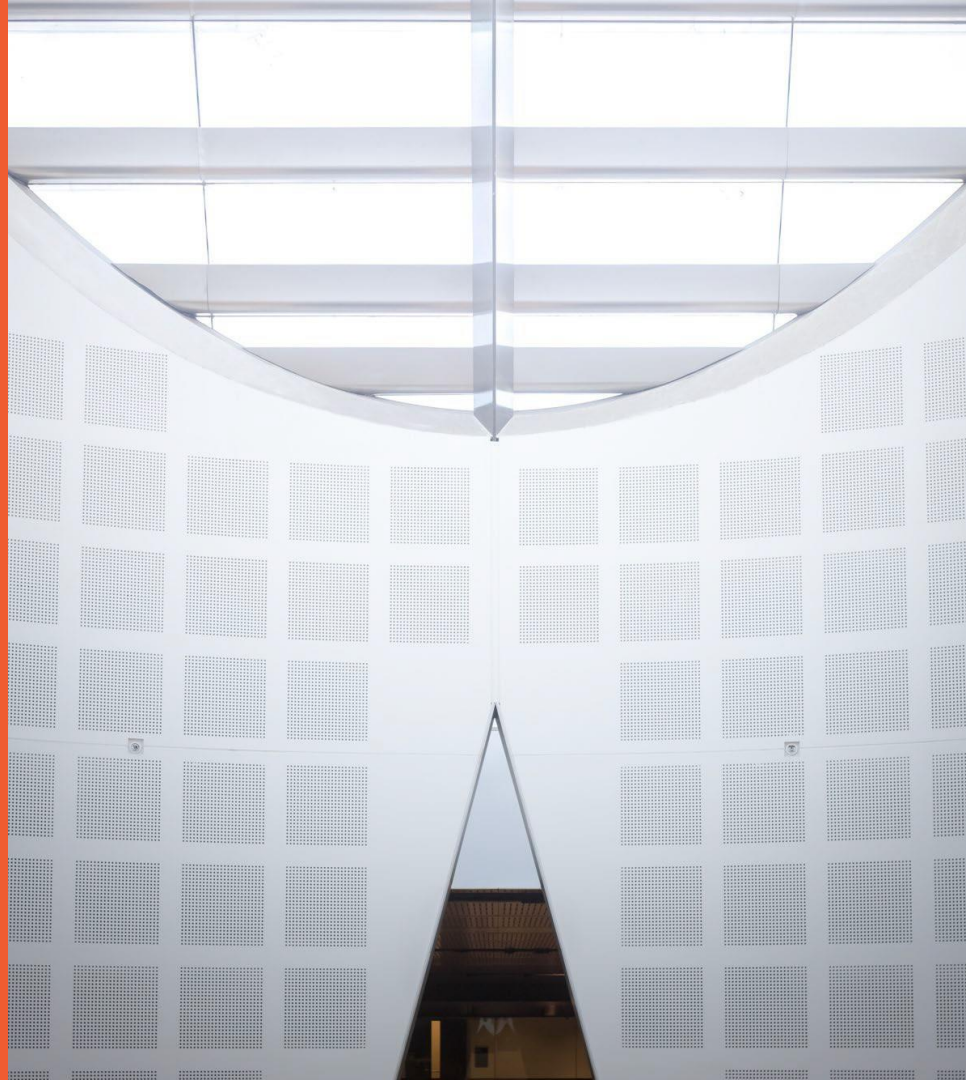
COMP5310: Principles of Data Science

W5: Querying and Summarising Data with SQL

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY



Overview of Week 5

Last Week: ETL with Python and SQL

Last Week's Objective

Use Python and PostgreSQL to extract, clean, transform and store data.

Lecture

- DB Access from Python
- Data cleaning and preprocessing
- Data Modeling and DB Creation
- Data Loading/Storage

Readings

- Data Science from Scratch: Ch 9 + 10

Exercises

- Python / Jupyter to load data
- psycopg2
- PostgreSQL to store data

Today: Querying and Summarising Data

Objective

To be able to extract a data set from a database, as well as to leverage on the SQL capabilities for in-database data summarisation and analysis.

Lecture

- Data Gathering reprise
- SQL querying
- Summarising data with SQL
- Statistic functions support in SQL

Readings

- Data Science from Scratch, Ch 23

Exercises

- Data Loading
- SQL Querying
- Python DB Querying
- Data Summarization using SQL

TODO in W5

- Summarize datasets and discuss recommendations [Group]

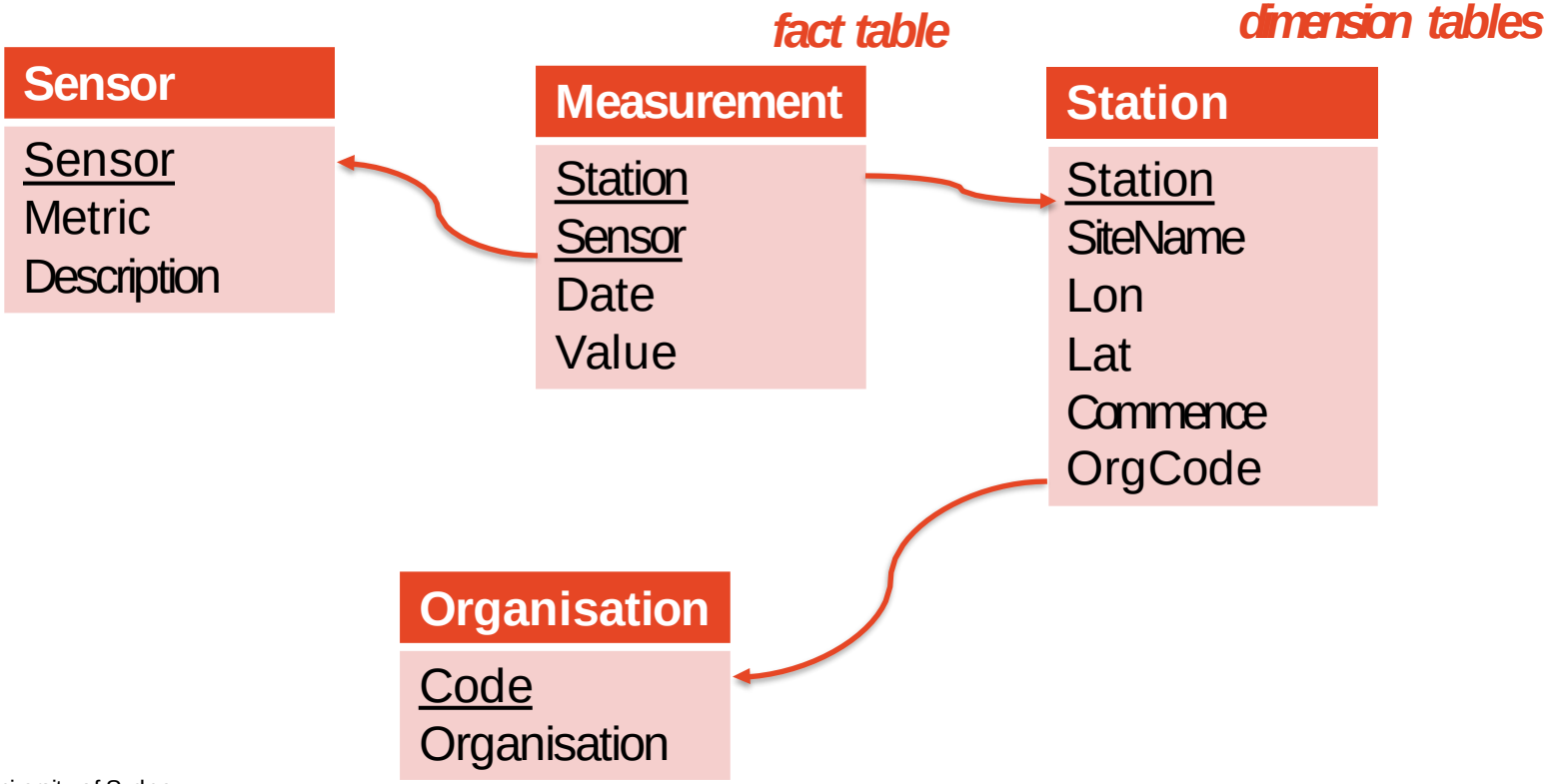
Data Loading / Storage

Data Storage

- We continue this week where we left off last time, with loading and storing data in a **relational database**
  - Last week focus on ETL: Extract-Transform-Load
  - This week take a clean database as given and concentrate on exploring and querying it

Water Database Schema

Four tables as shown below including foreign-key relationships



Querying Data with SQL

SELECT Statement

- SELECT: retrieves data (rows) from one or more tables that fulfill a search condition
- Clauses of the SELECT statement:
  - **SELECT** Lists the attributes (and expressions) that should be returned from the query
  - **FROM** Indicate the table(s) from which data will be obtained
  - **WHERE** Indicate the conditions to include a tuple in the result
  - **GROUP BY** Indicate the categorization of tuples
  - **HAVING** Indicate the conditions to include a category
  - **ORDER BY** Sorts the result according to specified criteria
- The result of an SQL query is a relation

More SELECT Statement Options

SQL Statement	Meaning
SELECT COUNT(*) FROM *T*	count how many tuples are stored in table *T*
SELECT * FROM *T*	list the content of table *T*
SELECT * FROM *T* LIMIT *n*	only list *n* tuples from a table
SELECT * FROM *T* ORDER BY *a*	order the result by attribute *a* (in ascending order; add DESC for descending order)

Single-Table SELECT Statement

- The *workhorse* command: **SELECT – FROM – WHERE**
  - Example 1: Which station commence after 1900-1-1?

```
SELECT siteName, commence, orgcode
  FROM station
 WHERE commence > '1900-1-1';
```

  - Example 2: How many measurements we have done?

```
SELECT COUNT(*) FROM Measurement;
```

  - Example 3: List top five measurements ordered by date in descending order.

```
SELECT * FROM Measurement
   ORDER BY date DESC limit 5;
```

- Note: SQL is case-insensitive and additional spaces + newlines are ignored; use this to format a query for better readability.

SQL Data Types

- Integers
  - Standard integer arithmetic and comparisons available
- Floats, Numeric
  - Floating point numbers with many mathematical operators and functions
- Strings (CHAR, VARCHAR)
  - SQL string literals must be enclosed in single quotes ('like this')
  - CHAR: fixed length; VARCHAR: variable length strings up-to max length
  - String comparison is case-sensitive
  - Pattern matching with LIKE operator and % placeholders
  - String concatenation: || (eg. 'hello ' || 'there')
- Date, Timestamp

Comparison Operations

- Comparison operators in SQL: **=** , **>** , **>=** , **<** , **<=** , **!=** , **<>**, **BETWEEN**
- Comparison results can be combined using logical connectives: **and**, **or**, **not**
- Example 1:

```
SELECT *
  FROM TelescopeConfig
 WHERE     ( mindec BETWEEN -90 AND -50 )
       AND
           ( maxdec >= -45 )
       AND
            ( tele_array = 'H168' );
```

- Example 2:

```
SELECT *
  FROM TelescopeConfig
 WHERE tele_array LIKE 'H%';
```

Date and Time in SQL

SQL Type	Example	Description
DATE	'2012-03-26'	a date (some systems incl. time)
TIME	'16:12:05'	a time, often down to nanoseconds
TIMESTAMP	'2012-03-26 16:12:05'	Time at a certain date: SQL Server: DATETIME
INTERVAL	'5 DAY'	a time duration

- Comparisons
  - Normal time-order comparisons with '=', '>', <', '<=', '>=', …
- Constants
  - CURRENT_DATE db system's current date
  - CURRENT_TIME db system's current timestamp
- Example:

```
SELECT *
  FROM Epoch
 WHERE startDate < CURRENT_DATE;
```

Date and Time in SQL (cont'd)

- Database systems support a variety of date/time related ops
  - Unfortunately not very standardized – a lot of slight differences
- Main Operations
  - **EXTRACT(** *component* **FROM** *date* **)**
    - e.g. EXTRACT(year FROM startDate)
  - **DATE** *string* (Oracle syntax: TO_DATE(*string,template*))
    - e.g. DATE '2012-03-01'
    - Some systems allow templates on how to interpret *string*
    - Oracle syntax: TO_DATE('01-03-2012', 'DD-Mon-YYYY')
  - **+/- INTERVAL**:
    - e.g. '2012-04-01' + INTERVAL '36 HOUR'

NULL Values

- Tuples can have missing values for some attributes, denoted by **NULL**
  - Integral part of SQL to handle missing / unknown information
  - **null** signifies that a value *does not exist*, it does *not mean "0" or "blank"*!
- The predicate **is null** or **is not null** can be used to check for nulls
  - e.g. Find measurements with an unknown intensity error value.

```
SELECT gid, band, epoch
  FROM Measurement
 WHERE intensity IS NULL
```

- Consequence: Three-valued logic
  - The result of any arithmetic expression involving null is null
    - e.g. 5 + null returns null
  - However, (most) aggregate functions simply ignore nulls

NULL Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - e.g. 5 < *null* or *null* <> *null* or *null* = *null*

a	b	a = b	a AND b	a OR b	NOT a	a IS NULL
true	true	true	true	true	false	false
true	false	false	false	true	false	false
false	true	false	false	true	true	false
false	false	false	false	false	true	false
true	NULL	unknown	unknown	true	false	false
false	NULL	unknown	false	unknown	true	false
NULL	true	unknown	unknown	true	unknown	true
NULL	false	unknown	false	unknown	unknown	true
NULL	NULL	unknown	unknown	unknown	unknown	true

- Result of **where** clause predicate is treated as false if it evaluates to unknown
  - e.g: **select** sid **from** enrolled **where** grade = 'unknown'
    ignores all students without a grade so far

JOIN: Querying Multiple Tables

- Often data that is stored in multiple different relations must be combined
- We say that the relations are **joined**
  - **FROM** clause lists all relations involved in the query
  - join-predicates can be explicitly stated in the **where** clause; do not forget it!
- Examples:
  - Produces the cross-product ***Station*** x ***Sensor***

```
SELECT *
  FROM Station, Organisation;
```

  - Find the site name, commence date and organisation name of all stations :

```
SELECT sitename, commence, organisation
  FROM Station S, Organisation O
 WHERE S.orgcode = O.code;
```

SQL Join Operators

- SQL offers join operators to directly formulate the natural join, equi-join, and the theta join operations.
  - R **natural join** S
  - R [**inner**] **join** S **on** <join condition>
  - R [**inner**] **join** S **using** (<list of attributes>)
- These additional operations are typically used in the from clause
  - List all details of the first three measurements including Water data.

```
SELECT *
  FROM Measurement JOIN Sensor USING (sensor)
 LIMIT 3;
```

  - Find the site name, commence date and organisation name of all stations :

```
SELECT sitename, commence, organisation
  FROM Station JOIN Organisation
  ON orgcode = code;
```

SQL Aggregate Functions

SQL Aggregate Function	Meaning
COUNT(*attr*) ; COUNT(*)	Number of *Not-null-attr ; or* of all values
MIN(*attr*)	Minimum value of *attr*
MAX(*attr*)	Maximum value of *attr*
AVG(*attr*)	Average value of *attr* (arithmetic mean)
MODE() WITHIN GROUP (ORDER BY *attr*)	mode function over *attr*
PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY *attr*)	median of the *attr* values
…	…

Reprise: Accessing PostgreSQL from Python: psycopg2

- First, we need to import psycopg2, then connect to Postgresql
- Note: You need obviously to provide you own login name

```
def pgconnect():
    # please replace with your own details
    YOUR_DBNAME = 'your_dbName'
    YOUR_USERNAME = 'postgres'##or your created user
    YOUR_PW      = 'your_password'
    try:
        conn = psycopg2.connect(host='localhost',
                                database=YOUR_DBNAME,
                                user=YOUR_USERNAME,
                                password=YOUR_PW)
        print('connected')
    except Exception as e:
        print("unable to connect to the database")
        print(e)
    return conn
```

Querying PostgreSQL from Python

– How to execute an SQL statement on a given connection 'conn'

```
def pgquery( conn, sqlcmd, args, silent=False ):
    """ utility function to execute some SQL query statement
        it can take optional arguments (as a dictionary) to fill in for placeholder in the SQL
        will return the complete query result as return value - or in case of error: None
        error and transaction handling built-in (by using the 'with' clauses) """
    retval = None
    with conn:
        with conn.cursor() as cur:
            try:
                if args is None:
                    cur.execute(sqlcmd)
                else:
                    cur.execute(sqlcmd, args)
                retval = cur.fetchall() # we use fetchall() as we expect only _small_ query results
            except Exception as e:
                if not(silent):
                    print("db read error: ")
                    print(e)
    return retval
```

automatic commit rollback

executes SQL statement with or without arguments

error handling

Querying PostgreSQL from Python (cont'd)

– Example: Retrieving some data from the database

```
# connect to your database
conn = pgconnect()

# prepare SQL statement
query_stmt = "SELECT * FROM Sensor"

# execute query and print result
query_result = pgquery (conn, query_stmt, None)
print(query_stmt)
print(query_result)

# prepare another SQL statement including placeholders
query_stmt = "SELECT * FROM Measurement WHERE date=%(date)s"

# define the 'band' parameter, execute query+parameters, and print result
param = {'date' : '29/04/2005'}
query_result = pgquery (conn, query_stmt, param)
print(query_stmt)
print(query_result)

# cleanup
conn.close()
```

Example *range query*: query all rows of a table

parameter binding

Example *point query*: query a specific row

Exercise 1: Data Loading + Initial Exploration

- First part in Jupyter notebook
  - We need one file for this week's tutorial
    1. Jupyter notebook: "05_Summarising_Data_with_SQL.ipynb"
  - This file is available on Canvas
  - Upload it into your Jupyter area

  - Start the "05_Summarising_Data_with_SQL.ipynb" notebook and follow the first Exercise.
    - Make sure that you have loaded the example data into your postgresql databases from week-4 lab solution
    - Let the tutors know if there are any problems

Exercise 1: Querying Data with SQL

- Next part in Jupyter notebook
  - Use some SQL queries to explore the example data set

Summarising Data with SQL

THE UNIVERSITY OF SYDNEY

Summarising a Database with SQL

- SQL covered so far merely allows simple exploring and retrieving of a data set
- But we can do more with SQL:
  - Data categorization and aggregation
  - Complex filtering
  - Nested queries
  - Ranking
  - Etc.

- Basis of data summarisation is the GROUP BY clause

SQL Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Example: Find company and total amount of sales

Sales Table

company	amount
IBM	5500
DELL	4500
IBM	6500

```
SELECT Company, SUM(Amount)
  FROM Sales
```

company	amount
IBM	16500
DELL	16500
IBM	16500

```
SELECT Company, SUM(Amount)
  FROM Sales
 GROUP BY Company
```

company	amount
IBM	12000
DELL	4500

Queries with GROUP BY and HAVING

- In SQL, we can "partition" a relation into *groups* according to the value(s) of one or more attributes:

```
SELECT   [DISTINCT]   target-list
  FROM   relation-list
 WHERE   qualification
GROUP BY grouping-list
HAVING   group-qualification
```

- A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.
- Note: Attributes in **select** clause outside of aggregate functions must appear in the *grouping-list*
  - Intuitively, each answer tuple corresponds to a *group,* and these attributes must have a single value per group.

Example: Filtering Groups with HAVING Clause

- GROUP BY Example:
  - What was the average mark of each course?

```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
```

- HAVING clause: can further filter groups to fulfil a predicate
  - Example:

```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
  HAVING AVG(mark) > 10
```

  - Note: Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Evaluation Example

– Find the average marks of 6-credit point courses with more than 2 results

```
SELECT uos_code as unit_of_study, AVG(mark)
 FROM Assessment NATURAL JOIN UnitOfStudy
 WHERE credit_points = 6
GROUP BY uos_code
 HAVING COUNT(*) > 2
```

1. Assessment and UnitOfStudy are joined

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
ISYS3207	1002	10500	67	IS Project	4	10500
ISYS3207	1004	10505	80	IS Project	4	10505
SOFT3000	1001	10505	56	C Prog.	6	10505
INFO2120	1005	10500	63	DBS 1	4	10500
…	…	…	….	…	…	…

2. Tuples that fail the WHERE condition are discarded

Evaluation Example (cont'd)

3. remaining tuples are partitioned into groups by the value of attributes in the grouping-list.

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
~~SOFT3000~~	~~1001~~	~~10505~~	~~56~~	~~C Prog.~~	~~6~~	~~10505~~
INFO5990	1001	10505	67	IT Practice	6	10505
…	…	…	….	…	…	…

4. Groups which fail the HAVING condition are discarded.

5. ONE answer tuple is generated per group

uos_code	AVG(..)
COMP5138	56
INFO5990	40.5

Exercise 2: Summarising Data with SQL

- Next part in Jupyter notebook
  - About more advanced analytical SQL queries

- Your Task
  - Answer some summarisation questions about the given water data set with single SQL queries

Gathering Data for Visualization

Data Gathering for Visualisation from SQL in Python

```python
import psycopg2.extras

def pgquery( conn, sqlcmd, args, silent=False, returntype='tuple'):
    """ utility function to execute some SQL query statement
        it can take optional arguments (as a dictionary) to fill in for placeholder in the SQL
        will return the complete query result as return value - or in case of error: None
        error and transaction handling built-in (by using the 'with' clauses) """
    retval = None
    with conn:
        cursortype = None if returntype != 'dict' else psycopg2.extras.RealDictCursor
        print(returntype)
        with conn.cursor(cursor_factory=cursortype) as cur:
            try:
                if args is None:
                    cur.execute(sqlcmd)
                else:
                    cur.execute(sqlcmd, args)
                retval = cur.fetchall() # we use fetchall() as we expect only _small_ query results
            except Exception as e:
                if e.pgcode != None and not(silent):
                    print("db read error: ")
                    print(e)
    return retval
```

specifies to return each result row as a dictionary (named key-value pairs)

```python
# connect to your database
conn = pgconnect()

# prepare SQL statement
query_stmt = """SELECT *
                FROM Sensor"""

# execute query and print result
query_result = pgquery (conn, query_stmt, None, returntype='dict')
print(query_result)

# close connection
conn.close()
```

Data Visualisation from SQL in Python

```python
import matplotlib.pyplot as plt
import numpy as np

def make_plot(data, x_key, y_key, title, xlabel=None, ylabel=None, bar_width=0.5, categorical=True):
    xlabel = xlabel or x_key
    ylabel = ylabel or y_key
    xs = [row[x_key] for row in data]
    ys = [row[y_key] for row in data]

    if categorical:
        plt.bar(range(len(data)), ys, width=bar_width)
        plt.xticks(np.arange(len(data))+bar_width/2., xs)
    else:
        plt.scatter(xs, ys)

    plt.title(title)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show()
```

```python
conn = pgconnect()

# prepare SQL statement
query_stmt ="""SELECT sensor, COUNT(*)
            FROM Measurement
               GROUP BY sensor;"""
# execute query and print result
query_result = pgquery (conn, query_stmt, None, returntype='dict')
print(query_result)
for r in query_result:
    print(r)
# cleanup
conn.close()

make_plot(
    query_result,
    x_key='sensor',
    y_key='count',
    title='Sensor Measurements',
    categorical=True)
```

Exercise 3: Data Visualisation of Query Results

- Next part in Jupyter notebook
  - Follow the steps to connect in Python to PostgreSQL
  - Query different parts of the water database using Python
  - Explore the difference between SQL results and CSV data imports
  - Run the given visualisation of a query result

- Your task:
  - SQL Data visualisation of other query results
  - Feel free to adapt any other plot functions from Week 3 following the given programming pattern

Review

References

- "Data Science From Scratch", Chapter 23
- Grok tutorial, SQL part (Section 16 onwards)
- PostgreSQL Online documentation
  - http://www.postgresql.org/docs/current/static/
  - http://www.postgresql.org/docs/current/static/functions-aggregate.html

Next Time

Next Lecture : Hypothesis Testing and Evaluation

Objective

Learn Python tools for exploring a new data set programmatically.

Lecture

- Questions: Do two populations differ? Which approach is better?
- Significance: pairwise t-tests, confidence intervals, non-normal data
- Evaluation: cluster evaluation, classifier evaluation, user evaluation

Readings

- Model evaluation (sklearn doco) http://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation
- Hypothesis testing (scipy lectures) http://www.scipy-lectures.org/packages/statistics/index.html#hypothesis-testing-comparing-two-groups

Exercises

- scipy: statistical tests
- sklearn: evaluation metrics

TODO in W6

- Project final report

Project Stage 1

Project Scope 1

Objective
Pick, clean and prepare a data set and perform simple analysis on it

Activities
- Choose a data set [Individual]
- Explore and summarise data set [Individual]
- Clean and prepare data [Individual]
- Perform simple analysis [Individual]
- Compare datasets and give recommendations [Group]

Output

report
Individual + group components

dataset
zip or tar.gz file.

Marking
5% of overall mark

Suggested Timeline for Project Stage 1

- W2: Identify possible topics
- W3: Identify and Explore possible data sets
- W4: Clean and prepare data, perform simple analysis using Python [Individual]
- Week 5: Summarize datasets and discuss recommendations [Group]
- W6: Report writing

Questions?