Elements Of Data Science - F2021

Week 13: Databases and Review

12/13/2021

TODOs

- Quiz 13, **Due Sunday December 19th, 11:59pm ET**
- Final
    - Release Monday December 13th, 11:59pm ET
    - **Due Wednesday December 15th, 11:59pm ET**
    - Have 24hrs after starting exam to finish
    - 30-40 questions (fill in the blank/multiple choice/short answer)
    - Online via Gradescope
    - Open-book, open-note, open-python
    - Questions asked/answered **privately** via Ed
    - NOTE: we won't be answering questions specific to the exam in office hours, only via Ed

Today

- Relational DBs and SQL
- Connecting to databases with sqlalchemy and pandas
- Review for the final

Questions re Logistics?

Environment Setup

Environment Setup

In [1]:
```
import numpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from mlxtend.plotting import plot_decision_regions

sns.set_style('darkgrid')
%matplotlib inline
```

Accessing Databases with Python

- databases vs flat-files
- Relational Databases and SQL
- NoSQL databases

Flat Files

Company Details

E_ID	Name	Department	Dept_ID	Manager_Name
101	Anoop	Accounts	AC-10	Mr Gagan Thakral
201	Anurag	Accounts	AC-10	Mr Gagan Thakral
301	Rakesh	Accounts	AC-10	Mr Gagan Thakral
401	Saurav	Accounts	AC-10	Mr Gagan Thakral

- eg: csv, json, etc
- Pros
    - Ease of access
    - Simple to transport
- Cons
    - May include redundant information
    - Slow to search
    - No integity checks

Relational Databases

- Data stored in **tables** (rows/columns)
- Table columns have well defined datatype requirements
- Complex **indexes** can be set up over often used data/searches
- Row level security, separate from the operating system
- Related data is stored in separate tables, referenced by **keys**

- Many commonly used Relational Databases
    - sqlite (small footprint db, might already have it installed)
    - Mysql
    - PostgreSQL
    - Microsoft SQL Server
    - Oracle

Database Normalization

- Organize data in accordance with **normal forms**

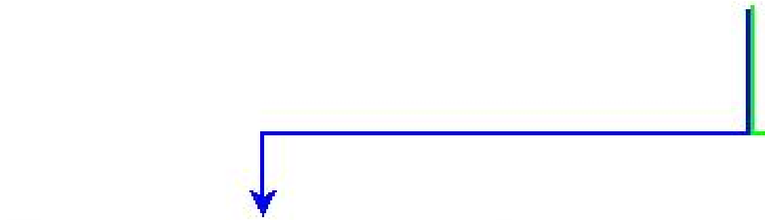
- Rules designed to:
    - reduce data redundancy
    - improve data integrity

- Rules like:
    - Has Primary Key
    - No repeating groups
    - Cells have single values
    - No partial dependencies on keys (use whole key)
    - ...

Database Normalization

Company Details

E_ID	Name	Department	Dept_ID	Manager_Name
101	**Anoop**	**Accounts**	**AC-10**	**Mr Gagan Thakral**
201	**Anurag**	**Accounts**	**AC-10**	**Mr Gagan Thakral**
301	**Rakesh**	**Accounts**	**AC-10**	**Mr Gagan Thakral**
401	**Saurav**	**Accounts**	**AC-10**	**Mr Gagan Thakral**



E_ID	Name	Dept_ID
101	**Anoop**	**AC-10**
201	**Anurag**	**AC-10**
301	**Rakesh**	**AC-10**
401	**Saurav**	**AC-10**

Employee_Details

Dept_ID	Department	Manager_Name
A-10	**Accounts**	**Mr Gagan Thakral**

Department_Details

From https://www.minigranth.com/dbms-tutorial/database-normalization-dbms/

De-Normalization

- But we want a single table/dataframe!
- Very often need to **denormalize**
- .. using joins! (see more later)

Structured Query Language (SQL)

- (Semi) standard language for querying, transforming and returning data
- Notable characteristics:
    - generally case independent
    - white-space is ignored
    - strings denoted with single quotes
    - comments start with double-dash "--"

```
SELECT
    client_id
    ,lastname
FROM
    company_db.bi.clients --usually database.schema.table
WHERE
    lastname LIKE 'Gi%'   --only include rows with lastname starting with Gi
LIMIT 10
```

Small but Powerful DB: SQLite3

- likely already have it installed
- many programs use it to store configurations, history, etc
- good place to play around with sql

```
bgibson@civet:~$ sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Accessing Relational DBs: `sqlalchemy`

- flexible library for accessing a variety of sql dbs
- can use to query through pandas itself to retrieve a dataframe

Accessing Relational DBs: `sqlalchemy`

- flexible library for accessing a variety of sql dbs
- can use to query through pandas itself to retrieve a dataframe

In [2]:
```
import sqlalchemy

# sqlite sqlalchemy relative path syntax: 'sqlite:///[path to database file]'
engine = sqlalchemy.create_engine('sqlite:///../data/example_business.sqlite')

# read all records from the table sales
sql = """
SELECT
    *
FROM
    clients
"""

pd.read_sql(sql,engine)
```

Out[2]:

	client_id	firstname	lastname	home_address_id
0	102	Mikel	Rouse	1002
1	103	Laura	Gibson	1003
2	104	None	Reeves	1003
3	105	Scott	Payseur	1004

SQL: SELECT

SQL: SELECT

In [3]:
```
sql="""
SELECT
    client_id
    ,lastname
FROM
    clients
"""

pd.read_sql(sql,engine)
```

Out[3]:

	client_id	lastname
0	102	Rouse
1	103	Gibson
2	104	Reeves
3	105	Payseur

SQL: * (wildcard)

SQL: * (wildcard)

In [4]:
```
sql="""
SELECT
    *
FROM
    clients
"""
clients = pd.read_sql(sql,engine)
clients
```

Out[4]:

	client_id	firstname	lastname	home_address_id
0	102	Mikel	Rouse	1002
1	103	Laura	Gibson	1003
2	104	None	Reeves	1003
3	105	Scott	Payseur	1004

SQL: * (wildcard)

In [4]:
```
sql="""
SELECT
    *
FROM
    clients
"""
clients = pd.read_sql(sql,engine)
clients
```

Out[4]:

	client_id	firstname	lastname	home_address_id
0	102	Mikel	Rouse	1002
1	103	Laura	Gibson	1003
2	104	None	Reeves	1003
3	105	Scott	Payseur	1004

In [5]:
```
sql="""
SELECT
    *
FROM
    addresses
"""
addresses = pd.read_sql(sql,engine)
addresses
```

Out[5]:

	address_id	address
0	1002	1 First Ave.
1	1003	2 Second Ave.
2	1005	3 Third Ave.

SQL: LIMIT

SQL: LIMIT

In [6]:
```
sql="""
SELECT
    *
FROM
    clients
LIMIT 2
"""
pd.read_sql(sql,engine)
```

Out[6]:

	client_id	firstname	lastname	home_address_id
0	102	Mikel	Rouse	1002
1	103	Laura	Gibson	1003

SQL: WHERE

SQL: WHERE

In [7]:
```
sql = """
SELECT
    *
FROM
    clients
WHERE home_address_id = 1003
"""

pd.read_sql(sql,engine)
```

Out[7]:

	client_id	firstname	lastname	home_address_id
0	103	Laura	Gibson	1003
1	104	None	Reeves	1003

SQL: LIKE and %

SQL: LIKE and %

In [8]:
```
sql = """
SELECT
    *
FROM
    clients
WHERE home_address_id = 1003 AND lastname LIKE 'Gi%'
"""

pd.read_sql(sql,engine)
```

Out[8]:

	client_id	firstname	lastname	home_address_id
0	103	Laura	Gibson	1003

SQL: AS alias

SQL: AS alias

In [9]:
```
sql="""
SELECT
    client_id AS CID
    ,lastname AS Lastname
FROM
    clients AS ca
"""

pd.read_sql(sql,engine)
```

Out[9]:

	CID	Lastname
0	102	Rouse
1	103	Gibson
2	104	Reeves
3	105	Payseur

SQL: (INNER) JOIN

SQL: (INNER) JOIN

In [10]:
```
sql="""
SELECT
    c.firstname
    ,a.address
FROM clients AS c
JOIN addresses AS a ON c.home_address_id = a.address_id
WHERE c.firstname IS NOT NULL
"""

pd.read_sql(sql,engine)
```

Out[10]:

	firstname	address
0	Mikel	1 First Ave.
1	Laura	2 Second Ave.

SQL: LEFT JOIN

SQL: LEFT JOIN

In [11]:
```
sql="""
SELECT
    c.firstname,a.address
FROM clients AS c
LEFT JOIN addresses AS a ON c.home_address_id = a.address_id
WHERE c.firstname IS NOT NULL
"""

pd.read_sql(sql,engine)
```

Out[11]:

	firstname	address
0	Mikel	1 First Ave.
1	Laura	2 Second Ave.
2	Scott	None

SQL: RIGHT JOIN

SQL: RIGHT JOIN

In [12]:
```
# this will cause an error in pandas, right join not supported in sqlalchemy + sqlite3
sql="""
SELECT
    c.firstname,a.address
FROM clients AS c
RIGHT JOIN addresses AS a ON c.home_address_id = a.address_id
"""
#pd.read_sql(sql,engine)
```

SQL: RIGHT JOIN

In [12]:
```
# this will cause an error in pandas, right join not supported in sqlalchemy + sqlite3
sql="""
SELECT
    c.firstname,a.address
FROM clients AS c
RIGHT JOIN addresses AS a ON c.home_address_id = a.address_id
"""
#pd.read_sql(sql,engine)
```

In [13]:
```
pd.merge(clients,addresses,left_on='home_address_id',right_on='address_id',how='right')[['firstname','address']]
```

Out[13]:

	firstname	address
0	Mikel	1 First Ave.
1	Laura	2 Second Ave.
2	None	2 Second Ave.
3	NaN	3 Third Ave.

SQL: FULL OUTER JOIN

SQL: FULL OUTER JOIN

In [14]:
```
# this will cause an error in pandas, outer join not supported in sqlalchemy + sqlite3
sql="""
SELECT
    c.firstname,a.address
FROM clients AS c
OUTER JOIN addresses AS a ON c.home_address_id = a.address_id
"""
#pd.read_sql(sql,engine)
```

SQL: FULL OUTER JOIN

In [14]:
```
# this will cause an error in pandas, outer join not supported in sqlalchemy + sqlite3
sql="""
SELECT
    c.firstname,a.address
FROM clients AS c
OUTER JOIN addresses AS a ON c.home_address_id = a.address_id
"""
#pd.read_sql(sql,engine)
```

In [15]:
```
pd.merge(clients,addresses,left_on='home_address_id',right_on='address_id',how='outer')[['firstname','address']]
```

Out[15]:

	firstname	address
0	Mikel	1 First Ave.
1	Laura	2 Second Ave.
2	None	2 Second Ave.
3	Scott	NaN
4	NaN	3 Third Ave.

SQL: And Much More!

- Multiple Joins
- DISTINCT
- COUNT
- ORDER BY
- GROUP BY
- Operators (string concatenate operator is '||' in sqlite)
- Subqueries
- HAVING
- see Data Science From Scratch Ch. 23

pandasql

- allows for querying of pandas DataFrames using SQLite syntax
- good way to practice SQL without a database

pandasql

- allows for querying of pandas DataFrames using SQLite syntax
- good way to practice SQL without a database

In [16]:
```
from pandasql import PandaSQL

# set up an instance of PandaSQL to pass SQL commands to
pysqldf = PandaSQL()
```

pandasql

- allows for querying of pandas DataFrames using SQLite syntax
- good way to practice SQL without a database

In [16]:
```
from pandasql import PandaSQL

# set up an instance of PandaSQL to pass SQL commands to
pysqldf = PandaSQL()
```

In [17]:
```
sql = """
SELECT
    c.firstname,a.address
FROM clients AS c
JOIN addresses AS a ON c.home_address_id = a.address_id
"""
pysqldf(sql)
```

Out[17]:

	firstname	address
0	Mikel	1 First Ave.
1	Laura	2 Second Ave.
2	None	2 Second Ave.

NoSQL

- Anything that isn't traditional SQL/RDBMS
    - key-value (Redis,Berkely DB)
    - document store (MongoDB, DocumentDB)
    - wide column (Cassandra, HBase, DynamoDB)
    - graph (Neo4j)
- Rapidly growing field to fit needs
- Probably more as we speak

Example: Mongo

- records represented as documents (think json)
- very flexible structure
- great way to store semi-structure data
- a lot of processing needed to turn into feature vectors

- contains databases (db)
    - which contain collections (like tables)
        - which you then do finds on

Example: Mongo

- Need to have Mongo running on your local machine with a 'twitter_db' database

Example: Mongo

- Need to have Mongo running on your local machine with a 'twitter_db' database

In [18]:
```
import pymongo

# start up our client, defaults to the local machine
mdb = pymongo.MongoClient()

# get a connection to a database
db = mdb.twitter_db

# get a connection to a collection in that database
coll = db.twitter_collection
```

Example: Mongo

Example: Mongo

In [19]:

```
# get one record
coll.find_one()

example_output = """
{'_id': ObjectId('6073547ff41410932828e3cd'),
 'created_at': 'Sun Apr 11 19:56:25 +0000 2021',
 'id': 1381335345875279873,
 'id_str': '1381335345875279873',
 'text': 'RT @IainLJBrown: Artificial Intelligence and the Art of Culinary Presentation - Columbia University\n\nRead more here:
 'truncated': False,
 'entities': {'hashtags': [],
  'symbols': [],
  'user_mentions': [{'screen_name': 'IainLJBrown',
    'name': 'Iain Brown, PhD',
    'id': 467513287,
    'id_str': '467513287',
    'indices': [3, 15]}],
  'urls': []},
  ...
"""
```

Questions re Databases?

For SQL practice, check out SQL Murder Mystery (https://mystery.knightlab.com/)

Final Review