QSS20: Modern Statistical Computing

Unit 13: SQL

Goals for today

- SQL: ways of interacting with a database and starting connection
- Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- Subqueries, aggregations, and joins: one table
- Subqueries, aggregations and joins: two tables

Goals for today

- **Recap**
- SQL: ways of interacting with a database and starting connection
- Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- Subqueries, aggregations, and joins: one table
- Subqueries, aggregations and joins: two tables

Recap of supervised machine learning

What do you remember?

Recap of supervised ML: Steps with code, optimized

1. Preprocess the data: make DTM, do train/test split

```
vec = CountVectorizer(); dtm_sparse = vec.fit_transform(texts)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size)
```

2. Initialize model and parameters to search

```
dt = DecisionTreeClassifier(min_samples_split=5, min_samples_leaf=10)
param_grid = {'min_samples_split': [2,10], 'min_samples_leaf': [2,10]}
```

3. Using training data, select optimal model via gridsearch

```
grid_dt = GridSearchCV(dt, param_grid, cv=3)
grid_dt.fit(X_train, y_train)
best_idx = np.argmax(grid_dt.cv_results_["mean_train_score"]) # get best
grid_dt.cv_results_["params"][best_idx] # best params
```

4. Re-train optimal model with full training data, get predictions

```
best_dt.fit(X_train, y_train)
y_pred = best_dt.predict(X_test)
y_predprob = best_dt.predict_proba(X_test)
```

5. Interpret model to identify important features

```
feat_imp = pd.DataFrame({'feature_imp': best_dt.feature_importances_,
                'feature_name': [col for col in X_train.columns]})
```

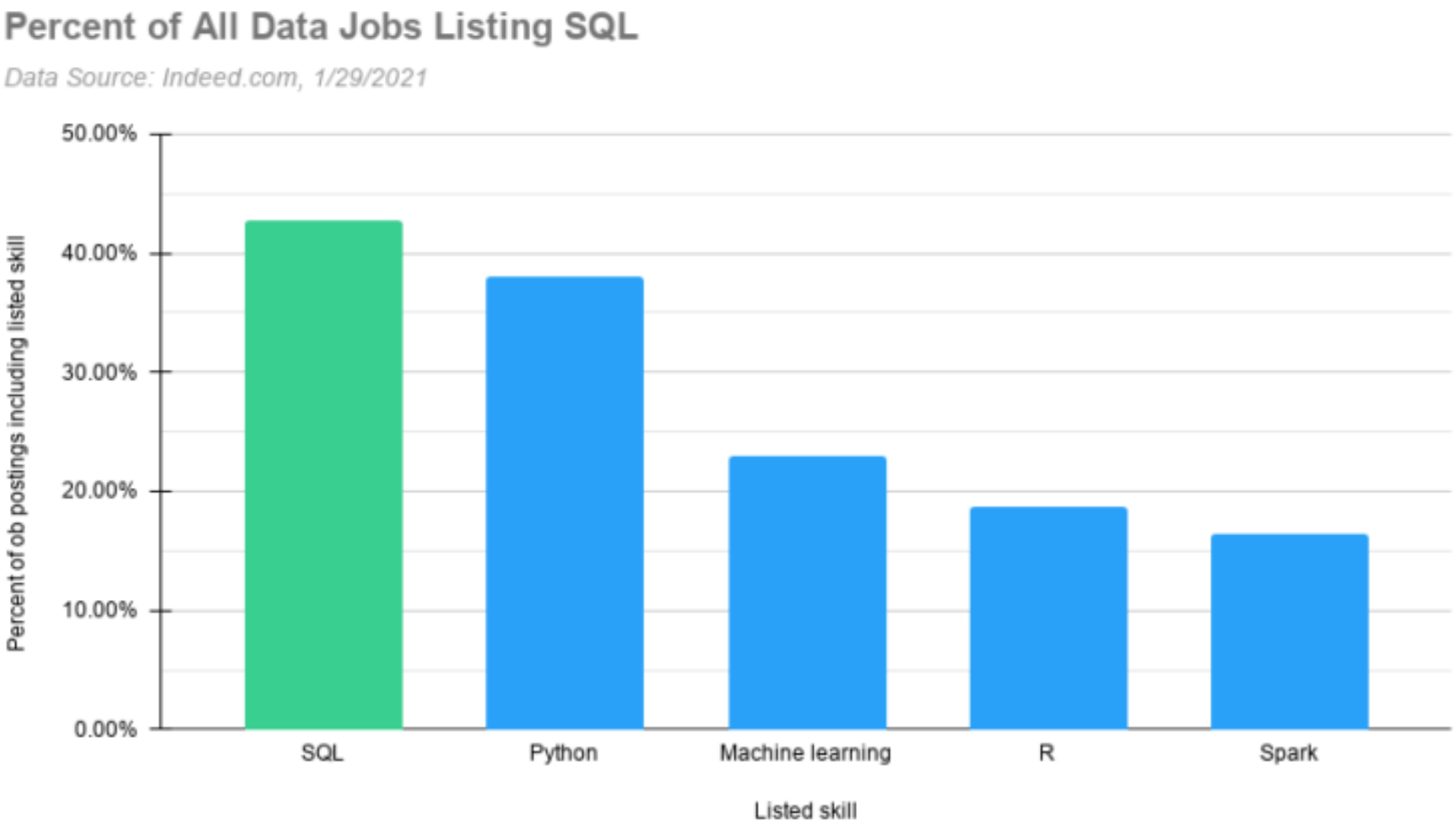
Where we are

- Recap
- **SQL: ways of interacting with a database and starting connection**
- Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- Subqueries, aggregations, and joins: one table
- Subqueries, aggregations and joins: two tables

What is SQL and why might it be useful?

- **S**tructured**Q**uery**L**anguage
- While relatively uncommon in academia, many companies / governments expect data scientists to be able to write SQL queries
- Various database engines used across data warehouses/companies: Amazon Redshift; MySQL; postgreSQL; Microsoft SQL server; SQLite
- Nearly identical syntax but some small differences on the margins; here, we're using a MySQL database since it's what Dartmouth Research Computing hosts
  - Also, around half of professional developers use MySQL—more than any other database–says StackOverflow

SQL a top skill for data science jobs



Source: Dataquest

When to use SQL

Likely scenarios:

- You walk into an internship/job and they say, "Here's how to authenticate to our database, it's in SQL. You can use that, right?" (very common)
- You have a huge dataset on which to run complex queries → can put in SQL and work from your laptop (requires little memory)
- You want to access a secured, external dataset without downloading → can put in SQL and work from your laptop (like an API)

Other tips:

- SQL like condensed version of Pandas; similar but diff. query syntax
- Usually easy to install Python MySQL connector (database setup less easy)—hopefully you've done this already
- Use creds `.yaml` file to access connector, as we did with APIs—or copy into code (less secure)

The way we will interact with database: connecting via another scripting language and sending queries through the connection

1. Use an R or Python package that helps you connect with a specific type of database (`Python`: SQLalchemy; MySQL connector; pyodbc; etc.; similar ones in `R`)
2. Establish a connection between your local computer and the database
3. Write a SQL query
4. Execute the query
5. Pull the result and work with the result in that language

Working example: two tables from Chicago felony prosecution datasets used in psets 1-2

Desc.	Table	Main cols	Database
Initiations	caseinit	CASE_ID; CASE_PARTICIPANT_ID; RACE; GENDER; UPDATED_OFFENSE_CATEGORY; is_in_diversion	sentencing
Diversions	divert	CASE_ID; CASE_PARTICIPANT_ID; RACE; DIVERSION_PROGRAM; OFFENSE_CATEGORY	sentencing

Overview of diversion programs

DC: Drug Treatment Court. Twenty-four months of treatment-based probation focusing on connecting defendants with housing and employment opportunities. (*Post-Plea*)

DDPP: Drug Deferred Prosecution Program. Links low-level, non-violent drug offenders to community-based services and includes a formal substance abuse assessment. (*Pre-Plea*)

DS: Drug School. Four 2-and-a-half-hour lessons provided by licensed treatment providers with a focus on substance abuse and education, not treatment. (Ended in 2017) (*Post-Plea*)

RJCC: Restorative Justice Community Court. Community court located in North Lawndale that practices restorative justice, a system of criminal justice which focuses on the rehabilitation of offenders through reconciliation with victims and the community at large. For a case to be eligible for RJCC, the victim of the crime must agree to participate in the process. (*Pre-Plea*)

MHC: Mental Health Treatment Court. Twenty-four months of intensive probation focusing on treatment, housing, psychiatric stability, and employment services. (*Post-Plea*)

VC: Veterans Treatment Court. Twenty-four months of probation focusing on employment, housing, and any necessary treatment. (*Post Plea*)

Source: Cook County SAO data documentation

Where we are

- Recap
- SQL: ways of interacting with a database and starting connection
- **Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions**
- Subqueries, aggregations, and joins: one table
- Subqueries, aggregations and joins: two tables

Notebook to follow along

https://github.com/jhaber-zz/QSS20_public/blob/main/activities/solutions/09_SQL_examplecode.ipynb

Preliminary step: load credentials and establish a connection

```
## import mysql connector
import mysql.connector

## load creds
creds = load_creds("../11_db_cred.yaml")

## use username, pwd, host, port, etc
## to establish a connection to the database
cnx = mysql.connector.connect(
    user=creds['practice_database']['user'],
    password=creds['practice_database']['password'],
    port=creds['practice_database']['port'],
    database= creds['practice_database']['database'],
    host = creds['practice_database']['host'])
```

Basic syntax of a SQL query

- Select specific columns and rows that meet condition:

```
select col1, col2
from tablename
where somecondition holds
```

- Select all columns and rows that meet condition:

```
select *
from tablename
where somecondition holds
```

Examining structure of data: selecting first 10 rows from case initiations table

```
## define a query
sample_case_q = """
select *
from caseinit
limit 10
"""

## feed read sql query the query and my db connection
read_sample_d = pd.read_sql_query(sample_case_q, cnx)
```

Breaking things down:

- `select *`: select all columns
- `from caseinit`: which table in database to pull from (if our database was more complicated, might be structured as something like sentencing_schema.caseinit that would indicate the case initiations table in the sentencing schema)
- Feed the (1) query and (2) database connection to pandas `read_sql_query`

Columns available to select from

```
Index(['CASE_ID', 'CASE_PARTICIPANT_ID', 'RECEIVED_DATE', 'OFFENSE_CATEGORY',
       'PRIMARY_CHARGE_FLAG', 'CHARGE_ID', 'CHARGE_VERSION_ID',
       'CHARGE_OFFENSE_TITLE', 'CHARGE_COUNT', 'CHAPTER', 'ACT', 'SECTION',
       'CLASS', 'AOIC', 'EVENT', 'EVENT_DATE', 'FINDING_NO_PROBABLE_CAUSE',
       'ARRAIGNMENT_DATE', 'BOND_DATE_INITIAL', 'BOND_DATE_CURRENT',
       'BOND_TYPE_INITIAL', 'BOND_TYPE_CURRENT', 'BOND_AMOUNT_INITIAL',
       'BOND_AMOUNT_CURRENT', 'BOND_ELECTRONIC_MONITOR_FLAG_INITIAL',
       'BOND_ELECTROINIC_MONITOR_FLAG_CURRENT', 'AGE_AT_INCIDENT', 'RACE',
       'GENDER', 'INCIDENT_CITY', 'INCIDENT_BEGIN_DATE', 'INCIDENT_END_DATE',
       'LAW_ENFORCEMENT_AGENCY', 'LAW_ENFORCEMENT_UNIT', 'ARREST_DATE',
       'FELONY_REVIEW_DATE', 'FELONY_REVIEW_RESULT',
       'UPDATED_OFFENSE_CATEGORY', 'is_in_diversion'],
      dtype='object')
```

Columns: selecting specific columns with no transformations/additions

```
1 select CASE_ID, CASE_PARTICIPANT_ID
2 from caseinit
```

What this does: selects those two identifiers from the case initiations table

Rows: filtering to specific rows using where

```
1 select CASE_ID, CASE_PARTICIPANT_ID,
2 AGE_AT_INCIDENT
3 from caseinit
4 where AGE_AT_INCIDENT > 40
```

Other logical operators:

- Equals: =
- Not equals: <>

Rows: filtering to specific rows using in or like

- Specify categories:

```
select CASE_ID, CASE_PARTICIPANT_ID,
RACE
from caseinit
where RACE in ("Black", "HISPANIC")
```

- If contains Black anywhere in RACE string

```
select CASE_ID, CASE_PARTICIPANT_ID,
RACE
from caseinit
where RACE like '%Black%'
```

Columns: creating new columns based on conditions

`CASE, WHEN, ELSE` syntax works similar to `np.where` and `np.select`

```
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
    THEN 'Same offense'
    ELSE 'Diff offense'
END as charge_update
from caseinit
```

What if we want to create a new col and then filter using that same columns as part of the same query?

If we try this query (created the charge_update column and then row filtering):

```
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
    THEN 'Same offense'
    ELSE 'Diff offense'
END as charge_update
from caseinit
where charge_update = 'Diff offense'
```

What if we want to create a new col and then filter using that same columns as part of the same query?

Get this SQL code error where it's telling us that it doesn't recognize the new column, because we can't simultaneously create a new col and filter:

```
DatabaseError: Execution failed on sql '
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY THEN 'Same offense'
    ELSE 'Diff offense'
END charge_update
from caseinit
where charge_update = 'Diff offense'
': 1054 (42S22): Unknown column 'charge_update' in 'where clause'
```

Another approach: direct row filtering using `where`–without the `case when`

```
1 select *
2 from caseinit
3 where OFFENSE_CATEGORY <> UPDATED_OFFENSE_CATEGORY
```

Where we are

- Recap
- SQL: ways of interacting with a database and starting connection
- Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- **Subqueries, aggregations, and joins: one table**
- Subqueries, aggregations and joins: two tables

Approach using subqueries: in words

1. Write a subquery to create the column indicating whether the charge has been updated (`charge_update`)
2. Use the output of that subquery
3. Then, in the main select column, we can select/do whatever we want with the `charge_update` column we created in the subquery

Approach using subqueries: in code

Create `charge_update` col, filter to changed offenses using inner join:

```
1  select *
2  from caseinit
3  inner join
4      (select CASE_ID as cid,
5      CASE_PARTICIPANT_ID as cpid,
6      CASE
7          WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
8          THEN 'Same offense'
9          ELSE 'Diff offense'
10     END as charge_update
11     from caseinit) as tmp
12     on tmp.cid = caseinit.case_ID and
13     tmp.cpid = caseinit.CASE_PARTICIPANT_ID
14
15 where charge_update = "Diff offense"
```

Breaking things down, we use parantheses to define a subquery where we:

- Use "as" to alias `CASE_ID` as cid, similar with cpid
- Execute our case when statement
- Alias the newly created table as `tmp` and join back w/ our main data

One more time: Subquery to create col, filter w/ inner join

```
select *
from caseinit
inner join
    (select CASE_ID as cid, CASE_PARTICIPANT_ID as cpid,
    CASE
        WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
        THEN 'Same offense' ELSE 'Diff offense'
    END as charge_update
    from caseinit) as tmp
    on tmp.cid = caseinit.case_ID and
    tmp.cpid = caseinit.CASE_PARTICIPANT_ID
where charge_update = "Diff offense"
```

Equivalent in pandas:

```
tmp = caseinit.copy()
tmp['charge_update'] = np.where(
    tmp.OFFENSE_CATEGORY == tmp.UPDATED_OFFENSE_CATEGORY,
    'Same offense', 'Diff offense')
merged = pd.merge(left=caseinit, right=tmp, how='inner',
    left_on=('CASE_ID', 'CASE_PARTICIPANT_ID'),
    right_on=('cid','cpid'))
merged = merged[merged.charge_update == 'Diff offense']
```

Subqueries are most powerful in the context of aggregations

General workflow:

1. Construct a subquery that does some transformation or aggregation of the table
2. Join the result to the main table
3. Do operations like row and column filtering in the outer part of the query that uses the output of the subquery

Example: disparities in who receives leniency through diversion

Want to:

1. Find the five most common offenses in the `caseinit` table
2. For those five most common offenses, find the percent of Black defendants whose cases are diverted and the percent of White defendants whose cases are diverted
3. Create a new column—`diff_diversion`—that's the White diversion rate for the offense minus the Black diversion rate

Rather than creating a complex query all at once, let's incrementally build the query

Step 1: finding five most common offenses

```
select UPDATED_OFFENSE_CATEGORY,
count(*) as count_offense
from caseinit
where RACE in ("Black", "White")
group by UPDATED_OFFENSE_CATEGORY
order by count_offense desc
limit 5
```

Breaking it down:

- Grouping by offense category
- Using `count(*)` to get the number of rows in that group
- Using `as` to call that column `count_offense`
- Order from highest to lowest count of rows; take top 5

Step 2: adding row filtering to offenses in those top 5

```
select *
from caseinit
inner join(
    select UPDATED_OFFENSE_CATEGORY as tmp_oc,
    count(*) as count_offense
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY
    order by count_offense desc
    limit 5
    ) as top5
    on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc
```

Breaking it down:

- Put the query we wrote in previous step into a subquery
- The `inner join` means that the only rows from the `caseinit` table retained are ones where the `UPDATED_OFFENSE_CATEGORY` is in that top 5

One more time: Group by offense, add top 5 w/ inner join

```sql
select *
from caseinit
inner join(
    select UPDATED_OFFENSE_CATEGORY as tmp_oc,
    count(*) as count_offense
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY
    order by count_offense desc
    limit 5
    ) as top5
    on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc
```

Equivalent in pandas:

```python
top5 = caseinit[caseinit.RACE.isin(['Black', 'White'])].
    groupby('UPDATED_OFFENSE_CATEGORY').
    agg({'CASE_ID': 'nunique'}).reset_index().
    rename(columns = {'CASE_ID': 'count_offense',
        'UPDATED_OFFENSE_CATEGORY': 'tmp_oc'}).
    sort_values(by='count_offense', ascending=False)[:5]
merged = pd.merge(left=caseinit, right=top5, how='inner',
    left_on='UPDATED_OFFENSE_CATEGORY',
    right_on='tmp_oc')
```

Step 3: for each offense, get proportion diverted by race

```
select UPDATED_OFFENSE_CATEGORY, is_in_diversion , RACE,
count(*) as count_divert , count(*)/count_group as prop_divert
from caseinit
inner join (
    select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race,
    count(*) as count_group
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY, RACE
    ) as tmp on tmp.tmp_race = caseinit.RACE
    and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY
group by UPDATED_OFFENSE_CATEGORY, RACE,
is_in_diversion
```

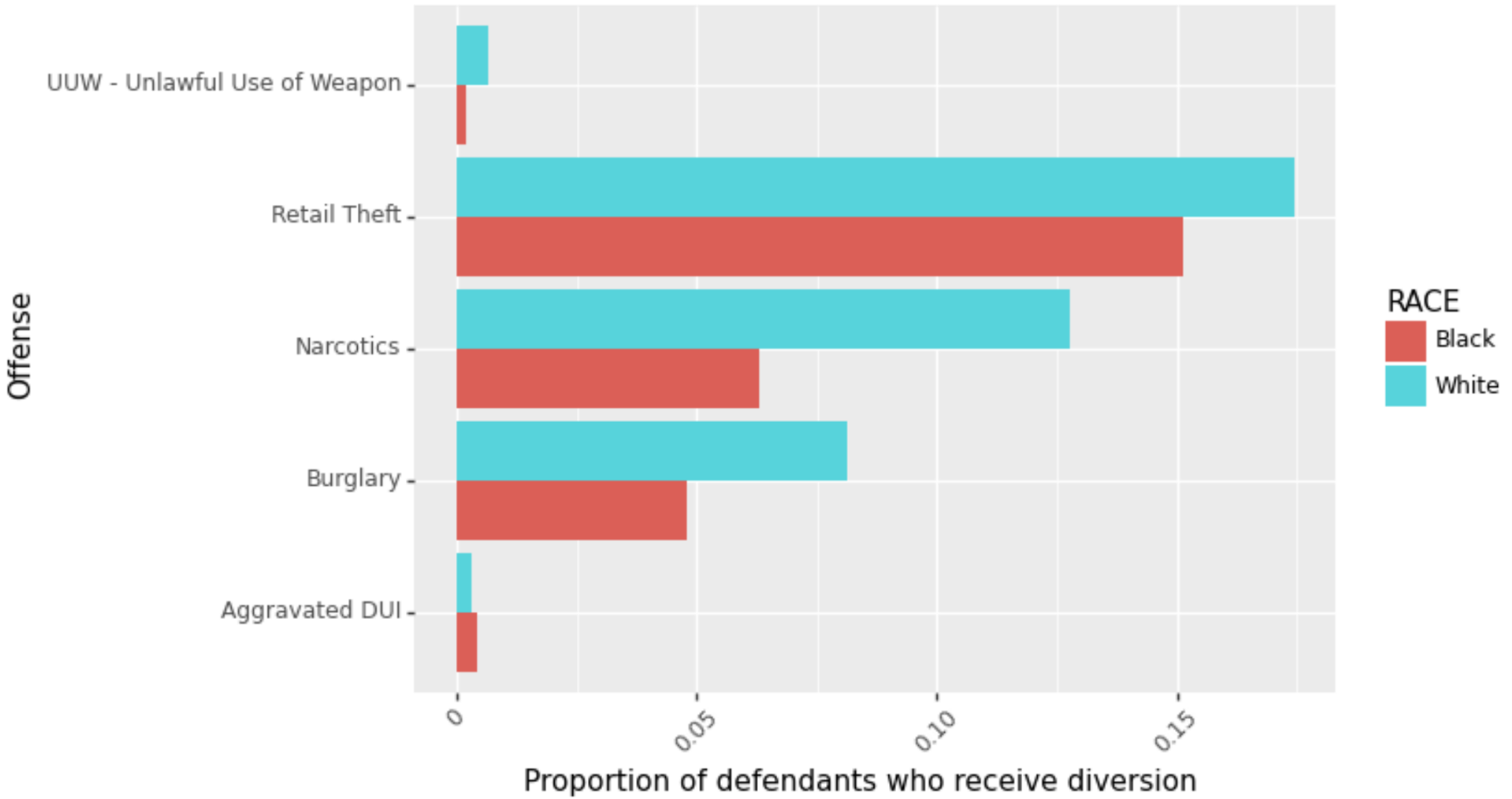
Logic:

- Filtering to Black and White defendants, group by race and crime to get the count of defendants in each race x crime combination (`count_group`)
- Merge retaining only defendants of those two races
- Group by race, crime, and diversion status to get count/proportion

Putting it together

```
select UPDATED_OFFENSE_CATEGORY, is_in_diversion , RACE,
count(*) as count_divert , count(*)/count_group as prop_divert
from caseinit
inner join (
    select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race,
    count(*) as count_group
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY, RACE
    ) as tmp on tmp.tmp_race = caseinit.RACE
    and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY
inner join (
    select UPDATED_OFFENSE_CATEGORY as tmp_oc_t5, count(*) as
    count_offense
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY
    order by count_offense desc
    limit 5
    ) as top5 on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc_t5
where is_in_diversion = 'True'
group by UPDATED_OFFENSE_CATEGORY, RACE,
is_in_diversion
```

After all that code, some disparities in narcotics



Activity 1: var creation and subquery practice

File: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/09_SQL_activity_blank.ipynb

1. Create a new column – in_chicago when pulling from the caseinit table that takes on the value of "YES" if INCIDENT_CITY = Chicago; "NO" otherwise (which represents incidents in Cook County suburbs outside the city limits); and pull the table. Use crosstabs to confirm that this worked
2. Repeat step 1 but also filter out blank strings (INCIDENT_CITY=="")
3. Use `where` to row filter to initiations in Chicago and use group by to find the count of cases diverted and not diverted (is_in_diversion); pull the table with those counts
4. Modify the query in step 3 to find the proportion of cases in chicago diverted (hint you made need to use case when in a subquery)
5. Modify the query in step 4 to find the proportion of cases in chicago versus cases not in chicago sent to diversion

Where we are

- Recap
- SQL: ways of interacting with a database and starting connection
- Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- Subqueries, aggregations, and joins: one table
- **Subqueries, aggregations and joins: two tables**

Merge diversions info with case initiations data to explore...

- How use of diversions differs across police departments (e.g., Chicago PD versus suburban PD)
- How bond/probation is related to diversion
- Age patterns (demographic var. available in caseinit but not in divert)

Left join of some cols from caseinit onto diversions: no aliasing

```
1 select divert.*,
2 AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,
3 INCIDENT_CITY
4 from divert
5 LEFT JOIN caseinit
6 ON divert.CASE_ID = caseinit.CASE_ID
7 AND divert.CASE_PARTICIPANT_ID = caseinit.
    CASE_PARTICIPANT_ID
```

Breaking it down:

- Selected all cols from divert using the syntax `tablename.*`
- Selected only age, law enforc. agency, and incident city from caseinit

What happens if we select cols available in both dataframes?

```
select divert.*,
AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,
INCIDENT_CITY, RACE
from divert
LEFT JOIN caseinit
ON divert.CASE_ID = caseinit.CASE_ID
AND divert.CASE_PARTICIPANT_ID = caseinit.
    CASE_PARTICIPANT_ID
```

Error:

```
IntegrityError: 1052 (23000): Column 'RACE' in field list is ambiguous
```

How to fix: aliasing the col

```
select divert.*,
AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,
INCIDENT_CITY, caseinit.RACE as caseinit_race
from divert
LEFT JOIN caseinit
ON divert.CASE_ID = caseinit.CASE_ID
AND divert.CASE_PARTICIPANT_ID = caseinit.
    CASE_PARTICIPANT_ID
```

Breaking it down:

- Use syntax `tablename.colname as something` to alias the RACE var from the case initiations table as something else so that we know which table it's from

Simplifying the query by aliasing the table names

```
select d.*,
AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,
INCIDENT_CITY, d.RACE as caseinit_race
from divert as d
LEFT JOIN caseinit as c
ON d.CASE_ID = d.CASE_ID
AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
```

Breaking it down:

- Rename caseinit as c
- Rename diversions as d

Other joins

- `INNER`, `OUTER`, `CROSS` (latter takes all rows from LHS data and repeats each for all rows of RHS data, and vice versa)
- Good discussion here: `https://www.guru99.com/joins.html`

Combining aggregation of one table and join

Goal: among the cases that are diverted, for each of the charges (`UPDATED_OFFENSE_CATEGORY`) in the case initiations, find the percentage of defendants with that charge going to each `DIVERSION_PROGRAM`

Step 1: find the count of offenses by diversion program

```
select count(*) as count_offenses_byprogram,
UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
from divert as d
INNER JOIN caseinit as c
ON d.CASE_ID = c.CASE_ID
AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
order by count_offenses_byprogram desc
```

Step 1: find the count of offenses by diversion program

```
select count(*) as count_offenses_byprogram,
UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
from divert as d
INNER JOIN caseinit as c
ON d.CASE_ID = c.CASE_ID
AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
order by count_offenses_byprogram desc
```

Breaking it down:

- Joining divert to caseinit
- Grouping by both offense and diversion program
- Aggregating using `count(*)`

Step 2: find the count of offenses in general

```
select count(*) as count_offenses_total,
UPDATED_OFFENSE_CATEGORY
from divert as d
INNER JOIN caseinit as c
ON d.CASE_ID = c.CASE_ID
AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
group by UPDATED_OFFENSE_CATEGORY
order by count_offenses_total desc
```

Step 3: combine into one query

```
select count(*) as count_offenses,
count_offenses_byprogram/count(*) as prop_offenses_inprogram,
UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
from caseinit
inner join (
    select
    count(*) as count_offenses_byprogram,
    UPDATED_OFFENSE_CATEGORY as ofc, DIVERSION_PROGRAM
    from divert as d
    INNER JOIN caseinit as c
    ON d.CASE_ID = c.CASE_ID
    AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
    group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM) as num
on num.ofc = caseinit.UPDATED_OFFENSE_CATEGORY
group by UPDATED_OFFENSE_CATEGORY
order by prop_offenses_inprogram desc
```

Breaking it down:

- Put the numerator into a subquery and do the proportions in the outer query.
- Add the second inner join to make sure that when we get the denominator from the outer part, we're restricting to defendants sent to diversion.

Activity 2: join and subquery practice

- Use the following crosswalk and the `CASE` command to create a new variable `DIVERSION_PROGRAM_TEXT` that spells out the diversion programs
  - DC: Drug Court
  - DPPP: Drug Deferred Prosecution
  - DS: Drug School
  - RJCC: Restorative Justice
  - MHC: Mental Health Court
  - VC: Veteran Court
- Build on the query from step 1 to filter to Narcotics as the 'UPDATED_OFFENSE_CATEGORY' and Black or White defendants (based on race in the diversions table) (hint: you'll need to join with the caseinit table based on case_id and case_participant_id, you can do a inner join to keep only those diverted). Select the case_id, case_participant_id, case, race, and diversion_program_text columns
- Built on the query from step 2 (and/or modify to just focus on drug school and drug court) to find the (1) rate of Black defendants sent to drug court, (2) rate of white defendants sent to drug court, (3) rate of Black defendants sent to drug school, and (4) rate of white defendants sent to drug school