Abstract

This blog will delve into the world of chemical analysis, exploring how distributed technologies like Spark can analyze large datasets of molecules, compounds, and other chemical elements using open-source libraries such as RDKit. This blog will provide a walkthrough demo that shows users how to acquire large open datasets like ChEMBL and execute operations such as substructure and similarity analysis in a parallel and distributed fashion. Users will be able to scale the computing resources according to the size of the datasets.

We are using Databricks as the underlying PaaS, but the operations we will cover can be used with any other platform that supports pySpark 3.x.

ChEMBL

The dataset we are using throughout this blog is from the well-known open-source dataset ChEMBL.

ChEMBL is a manually curated database of bioactive molecules with drug-like properties. It brings together chemical, bioactivity, and genomic data to aid the translation of genomic information into effective new drugs.

Onboarding

We will use ChEMBL Database downloads, which include SQLite, MySQL, and PostgreSQL versions of the ChEMBL database. Visit this page to download the latest release and start a database of your choice where you can import this database.

Disclaimer: This blog will not dive deeply into how to import the ChEMBL database into a local instance, but you can refer to the following pages that cover how to import a database dump file: - Postgres: https://www.postgresql.org/docs/current/backup-dump.html - MySQL: https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html

For this guide, we are using Postgres as a bridge between the latest ChEMBL export and Databricks Delta-lake (which we will be using as a source going forward).

Once the data is on your desired Database, make sure the Spark cluster you use has a connection to it.

The following script will create a DataFrame out of the `compound_structures` table included in ChEMBL, which we will later use to import into Delta:

```
from pyspark.sql.functions import expr
compound_structures_df = spark.read \
    .format("jdbc") \
    .option("url", jdbc_url) \
    .option("user", jdbc_username) \
    .option("password", jdbc_password) \
    .option("numPartitions",250000) \
    .option("dbtable", "chembl.compound_structures") \
    .load() \
    .select("molregno", "canonical_smiles")
```

RDKit

It is important to give a small background about some of the operations on chemical compounds that the following blog will cover which are calculating the bits of fingerprints or the Tanimoto similarity;

There are multiple algorithms to calculate fingerprints provided by each available chemical library, in this blog we will use the open-source library named RDKit; which provides an eponymous fingerprint algorithm that starts by labelling all atoms with their atomic number/aromaticity and all edges with their bond type and then exhaustively enumerates the (multi-)set of all sub-graphs within a certain size limit (1-7 atoms by default). It can be accessed as sparse count vectors and dense (folded) bit vectors.

We define the fingerprint calculation method as a Spark UDF, this function takes an input string parameter that represents the SMILES representation of the compound, examples

Compound name	String representation
C Methane	CH4
CC Ethane	CH3CH3
C=C Ethene	CH2CH2
C#C Ethyne	CHCH
COC Dimethyl ether	CH3OCH3
CCO Ethanol	CH3CH2OH
C#N Hydrogen Cyanide	HCN

The fingerprint vector provided by RDKit is represented as a bit vector that we will encode as a base64 string which is easier to manipulate in a Dataframe, e.g.

```
from rdkit import Chem
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

@udf(returnType=StringType())
def get_smiles_fingerprint_base64(smiles):
    try:
        mol = Chem.MolFromSmiles(smiles)
        fpgen = Chem.AllChem.GetRDKitFPGenerator()
        fp = fpgen.GetFingerprint(mol)
        return str(fp.ToBase64())
    except:
        return ""
```

Due to the computer calculations conducted using the functions above, it is usually recommended to calculate the fingerprints in a Data preparation phase and stored them as part of the dataset, doing this will enable faster querying times in a posterior phase.

```
 compound_structures_df
    .withColumn("fingerprints", expr("get_smiles_fingerprint_base64(canonical_smiles)")) \
    .write \
    .mode("overwrite") \
    .saveAsTable("chembl.compound_structures_with_fingerprints")
```

Now we can query through compounds using the Tanimoto similarity algorithm which states that A and B are sets of fingerprint "bits" within the fingerprints of molecule A and molecule B. AB is defined as the set of common bits of fingerprints of both molecule A and B. The resulting Tanimoto coefficient (or T(A,B)) ranges from 0, when the fingerprints have no bits in common, to 1, when the fingerprints are identical.

RDKit provides this similarity function by calling it like this;

```
from rdkit import DataStructs

def similarity(fp1, fp2, roundTo=2):
    try:
        return round(DataStructs.TanimotoSimilarity(fp1, fp2) * 100, roundTo)
    except Exception as e:
        return 0
```

This function requires 2 fingerprints in order to calculate the distance between vectors, but we can create handy functions that will take SMILES strings as inputs as well. E.g.

```
from rdkit import Chem
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType

@udf(returnType=FloatType())
def similarity_pct(smiles1, smiles2, roundTo=2):
    try:
        fpgen = Chem.AllChem.GetRDKitFPGenerator()

        m1 = Chem.MolFromSmiles(smiles1)
        fp1 = fpgen.GetFingerprint(m1)
        m2 = Chem.MolFromSmiles(smiles2)
        fp2 = fpgen.GetFingerprint(m2)

        return similarity(fp1, fp2, roundTo)
    except Exception as e:
        return 0
```

Now we can use these functions to do more interesting queries. For example, we can search our entire dataset and get all molecules that surpass a particular similar threshold to a given compound (benzene: c1ccccc1)

```
from pyspark.sql import functions as F

molecules = spark.read.table("chembl.compound_structures_with_fingerprints")
molecules \
    .withColumn("similarity_pct", similarity_pct(lit('c1ccccc1'), F.col("canonical_smiles"))
    .filter("similarity_pct >= 80")
```

We can even declare these UDFs so they are accessible in a SQL context

```
spark.udf.register("similarity_pct", similarity_pct)
```

```
SELECT
  molregno,
  canonical_smiles,
  similarity_pct('c1ccccc1', canonical_smiles) as similarity
FROM chembl.compound_structures
WHERE similarity_pct('c1ccccc1', canonical_smiles) >= 80
```

Substructure searching

Substructure searching, the process of finding a particular pattern (subgraph) in a molecule (graph), is one of the most important tasks for computers in chemistry. It is used in virtually every application that employs a digital representation of a molecule, including depiction (to highlight a particular functional group), drug design (searching a database for similar structures and activity), analytical chemistry (looking for previously-characterized structures and comparing their data to that of an unknown), and a host of other problems.

This function will return True or False if a compound is included as a substructure on a list of molecules.

```
from pyspark.sql.types import BooleanType
from pyspark.sql import functions as F
from rdkit import Chem

@udf(returnType=BooleanType())
```

```python
def has_substruct(pattern, smiles):
    try:
        m = Chem.MolFromSmiles(smiles)
        substruct = m.HasSubstructMatch(Chem.MolFromSmiles(pattern))
        return substruct
    except Exception as e:
        print(e)
        return False

molecules = spark.read.table("chembl.compound_structures_with_fingerprints")

molecules \
  .withColumn("has_substruct", has_substruct(F.lit('c1ccccc1'), F.col("canonical_smiles")))
```

Similarly, we can use this function on a SQL context.

```python
spark.udf.register("has_substruct", has_substruct)
```

```sql
SELECT
  molregno,
  canonical_smiles
FROM chembl.compound_structures
WHERE has_substruct('c1ccccc1', canonical_smiles)
```

Arthor

As we did with rdkit and since there are other libraries available that we can integrate with, the next examples are using Arthor as chemistry computation toolkit Two of the main features of Arthor is substructure searching and similarity searching as we did with rdkit. When working with Arthor we can either load an Arthor database using a binary file (.SubDb or .SimDb files) or creating databases on the fly using pandas series. We will use the latter to integrate with Spark.

To do this, Spark provides pandas UDFs that we can leverage to create the Arthor databases on the fly. The following code will use a smiles array and create an Arthor database;

```python
import arthor
import numpy as np

smiles_arr = [
    "Cc1ccc(C(c2c[nH]c3ccc(C(=O)NNS(=O)(=O)c4cccc([N+](=O)[O-])c4)cc23)c2c[nH]c3ccc(C(=O)NNS
    "N=C(N)NCCC[C@H](NC(=O)[C@H](Cc1ccc(O)cc1)NC(=O)OCC1c2ccccc2-c2ccccc21)C(=O)c1nc2ccccc2s
    "CC(C)(C)/[N+]([O-])=C/c1ccc(F)c(F)c1",
    "CCCCC(=O)N/N=C/c1c(C(=O)OCC)[nH]c2cc(Cl)ccc12",
    "CC(C)[C@H](NC(=O)[C@@H](NC(=O)[C@@H](Cc1ccccc1)Oc1cc(Cl)cc(Cl)c1)c1ccccc1)C(=O)C(F)(F)(
    "O=C(CCC(=O)N1CCN(CCCN2c3ccccc3Sc3ccccc32)CC1)OCC#CCOc1c[n+]([O-])on1",
    "C=CC(=O)Nc1cc(Nc2nccc(-c3cn(C)c4ccccc34)n2)c(OC)cc1N1CC[S+]([O-])CC1",
    "CCC(=O)O",
]

smiles = np.array(smiles_arr)

df = smiles.to_frame(name='smiles')
smiles = arthor.SubDb.from_smiles(df['smiles'])
```

Once we build the database using Pandas series as input, these classes will have a `.search()` method which takes a SMILES or SMARTS string respectively and returns a ResultSet object.

Sub-structure search

The following method will return a new pandas series with a boolean column that will show if a smiles string is a substructure of another smiles string used for querying.

```python
import arthor
import pandas as pd

def has_substruct(smiles: pd.Series, search: str) -> pd.Series:
    df = smiles.to_frame(name='smiles')
    db = arthor.SubDb.from_smiles(df['smiles'])
    res_df = db.search(search)
    rs_arr = res_df.to_array()
    df["hits"] = df.iloc[rs_arr]
    result = df["hits"].fillna(False).apply(lambda x: True if x else False)
    return result
```

Similarity search

As with the example above we construct with the following code, which will return a Dataframe with a numeric column named `score` which will represent the percentage of similarity 0-100

```python
import arthor
import pandas as pd

def similarity(smiles: pd.Series, search: str) -> pd.Series:
    df = smiles.to_frame(name='smiles')
    db = arthor.SimDb.from_smiles(df['smiles'])
    res_df = db.search(search).to_dataframe()
    subs = df.join(res_df.set_index('offset'))
    result = subs.fillna(0)["score"].map(lambda x: round(x*100, 2))
    return result
```

Building the similarity and sub-structure Spark UDFs

Finally, we create the following UDFs in order to use the functions above on Spark.

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("float")
def similarity_pct(pattern, smiles):
    return similarity(smiles, pattern[0])

@pandas_udf("boolean")
def has_substruct(pattern, smiles):
    return has_substruct(smiles, pattern[0])
```

The following code will read the source Chembl table that we were using at the beginning of this blog and add 2 columns representing the similarity percentage and the substructure evaluation of a source smiles string, in this case we use `c1ccccc1`

```
from pyspark.sql import functions as F

baseQuery = "c1ccccc1"

baseDF = spark.read.table("chembl.compound_structures")\
    .withColumn("similarity", similarity_pct(F.lit(baseQuery), F.col("smiles")))\
    .withColumn("substructure", has_substruct(F.lit(baseQuery), F.col("smiles")))
```

We can register the UDFs and do the same on a SQL context provided by Spark.

```
spark.udf.register("similarity_pct_arthor", similarity_pct)
spark.udf.register("has_substruct_arthor", has_substruct)
```

```
SELECT smiles,
       similarity_pct_arthor('c1ccccc1', smiles) as similarity,
       has_substruct_arthor('c1ccccc1', smiles) as substructure,
FROM chembl.compound_structures
```

Conclusions

As we can see with the above experiments, we can leverage libraries such as arthor an rdkit to run massive calculations in parallel using Spark. It is important to highlight how both libraries calculate similarities.

Rdkit calculates similarities against individual records atomically this means;

- Operations are heavier in compute than in memory
- A single operation can be poor in performance but it allows to parallelize operations and scale linearly.
- It benefits on a high amount of CPU than high levels of memory.

Arthor takes a chunk of data and loads it into memory via a pandas Series, this allows faster calculations against medium size chunks of data. This means…

- Its hard to scale linearly, since each executor needs to load data into memory
- It benefits more on a smaller amount of executors with high memory

References

- https://jcheminf.biomedcentral.com/articles/10.1186/s13321-015-0069-3
- https://github.com/gashawmg/Molecular-fingerprints
- https://greglandrum.github.io/rdkit-blog/posts/2023-01-18-fingerprint-generator-tutorial.html
- https://www.daylight.com/dayhtml/doc/theory/theory.smarts.html
- https://github.com/rdkit/rdkit-tutorials
- https://greglandrum.github.io/rdkit-blog/posts/2021-08-03-generalized-substructure-search.html
- https://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/latest/schema_documentation.html
- https://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/latest/
- https://chem.libretexts.org/Courses/University_of_Arkansas_Little_Rock/ChemInformatics_(2017)%3A
- https://www.sciencedirect.com/science/article/abs/pii/S135964462200349X#:~:text=Molecular%20fingerp
- https://www.featurebase.com/blog/tanimoto-and-chemical-similarity-in-featurebase#:~:text=Overview%20of%20Tanimoto&text=Thus%2C,more%20similar%20the%20molecules%
- https://nextmovesoftware.com/blog/2020/02/21/arthor-and-data-science-interoperability/