8.3: Python Assignment

Machine Learning Basics

Downloadable Files

lecture08_machine-learning.ipynb

- Download the ipynb file and run your Jupyter notebook.
  - You can use the notebook you created in section 1.5 or the Jupyter hub at LibreText: https://jupyter.libretexts.org (see your instructor if you do not have access to the hub).
  - This page is an html version of the above .ipynb file.
    - If you have questions on this assignment you should use this web page and the hypothes.is annotation to post a question (or comment) to the 2019OLCCStu class group. Contact your instructor if you do not know how to access the 2019OLCCStu group within the hypothes.is system.

Required Modules

- **pandas**
- numpy
- time
- requests
- io
- rdkit
- sklearn

Objectives

- Build binary classification models that predict activity/inactivity of small molecules against human aromatase using supervised learning methods.
- Evaluate the performance of the developed models using performance measures.

Import bioactivity data from PubChem

In this notebook, we will develop a prediction model for small molecule's activity against human aromatase (https://pubchem.ncbi.nlm.nih.gov/protein/EAW77416), which is encoded by the CYP19A1 gene (https://pubchem.ncbi.nlm.nih.gov/gene/1588). The model will predict the activity of a molecule based on the structure of the molecule (represented with molecular fingerprints).

For model development, we will use the Tox21 bioassay data for human aromatase, archived in PubChem (https://pubchem.ncbi.nlm.nih.gov/bioassay/743139). The bioactivity data presented on this page can be downloaded by clicking the "Download" button available on this page and then read the data into a data frame. Alternatively, you can directly load the data into a data frame as shown in the cell below.

In [1]:

```
import pandas as pd
import numpy as np

url = 'https://pubchem.ncbi.nlm.nih.gov/assay/pcget.cgi?query=download&record_type=da
df_raw = pd.read_csv(url)
```

In [2]:

```
df_raw.head(7)
```

Out[2]:

	PUBCHEM_RESULT_TAG	PUBCHEM_SID	PUBCHEM_CID	PUBCHEM_ACTIVITY_OUTCOME	PUBCHEM_ACTIVITY_SCORE	PUBCHEM_ACTIVITY_URL	PUBCHEM_ASSAYDATA_COMMENT	Activity Summary	Antagonist Activity	Antagonist Potency (uM)	Antagonist Efficacy (%)	Viability Activity	Viability Potency (uM)	Viability Efficacy (%)	Sample Source
0	RESULT_TYPE	NaN	NaN	NaN	NaN	NaN	NaN	STRING	STRING	FLOAT	FLOAT	STRING	FLOAT	FLOAT	STRING
1	RESULT_DESCR	NaN	NaN	NaN	NaN	NaN	NaN	Type of compound activity based on both the ar...	Type of compound activity in the aromatase ant...	The concentration of sample yielding half-maxi...	Percent inhibition of aromatase.	Type of compound activity in the cell viabilit...	The concentration of sample yielding half-maxi...	Percent inhibition of cell viability.	Where sample was obtained.
2	RESULT_UNIT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	MICROMOLAR	PERCENT	NaN	MICROMOLAR	PERCENT	NaN
3	1	144203552.0	12850184.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
4	2	144203553.0	89753.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
5	3	144203554.0	9403.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
6	4	144203555.0	132187791.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI

Note: Lines 0-2 provide the descriptions for each column (data type, descriptions, units, etc). These rows need be removed.

In [3]:

```
df_raw = df_raw[3:]
df_raw.head(5)
```

Out[3]:

	PUBCHEM_RESULT_TAG	PUBCHEM_SID	PUBCHEM_CID	PUBCHEM_ACTIVITY_OUTCOME	PUBCHEM_ACTIVITY_SCORE	PUBCHEM_ACTIVITY_URL	PUBCHEM_ASSAYDATA_COMMENT	Activity Summary	Antagonist Activity	Antagonist Potency (uM)	Antagonist Efficacy (%)	Viability Activity	Viability Potency (uM)	Viability Efficacy (%)	Sample Source
3	1	144203552.0	12850184.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
4	2	144203553.0	89753.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
5	3	144203554.0	9403.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
6	4	144203555.0	132187779.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
7	5	144203556.0	142766.0	Inconclusive	25.0	NaN	NaN	inconclusive antagonist (cytotoxic)	active antagonist	15.5454	-115.803	active antagonist	14.9601	-76.8218	NCI

The column names in this data frame contain white spaces and special characters. For simplicity, let's rename the columns (no spaces or special characters except for the "_" character.)

In [4]:

```
df_raw.columns
```

Out[4]:

```
Index(['PUBCHEM_RESULT_TAG', 'PUBCHEM_SID', 'PUBCHEM_CID',
       'PUBCHEM_ACTIVITY_OUTCOME', 'PUBCHEM_ACTIVITY_SCORE',
       'PUBCHEM_ACTIVITY_URL', 'PUBCHEM_ASSAYDATA_COMMENT', 'Activity Summary',
       'Antagonist Activity', 'Antagonist Potency (uM)',
       'Antagonist Efficacy (%)', 'Viability Activity',
       'Viability Potency (uM)', 'Viability Efficacy (%)', 'Sample Source'],
      dtype='object')
```

In [5]:

```
col_names_map = {'PUBCHEM_RESULT_TAG' : 'pc_result_tag',
                 'PUBCHEM_SID' : 'sid',
```

```
                  'PUBCHEM_CID' : 'cid',
                  'PUBCHEM_ACTIVITY_OUTCOME' : 'activity_outcome',
                  'PUBCHEM_ACTIVITY_SCORE' : 'activity_score',
                  'PUBCHEM_ACTIVITY_URL' : 'activity_url',
                  'PUBCHEM_ASSAYDATA_COMMENT' : 'assay_data_comment',
                  'Activity Summary' : 'activity_summary',
                  'Antagonist Activity' : 'antagonist_activity',
                  'Antagonist Potency (uM)' : 'antagonist_potency',
                  'Antagonist Efficacy (%)' : 'antagonist_efficacy',
                  'Viability Activity' : 'viability_activity',
                  'Viability Potency (uM)' : 'viability_potency',
                  'Viability Efficacy (%)' : 'viability_efficacy',
                  'Sample Source' : 'sample_source' }
```

In [6]:

```
df_raw = df_raw.rename(columns = col_names_map)
df_raw.columns
```

Out[6]:

```
Index(['pc_result_tag', 'sid', 'cid', 'activity_outcome', 'activity_score',
       'activity_url', 'assay_data_comment', 'activity_summary',
       'antagonist_activity', 'antagonist_potency', 'antagonist_efficacy',
       'viability_activity', 'viability_potency', 'viability_efficacy',
       'sample_source'],
      dtype='object')
```

Check the number of compounds for each activity group

First, we need to understand what our data look like. Especially, we are interested in the activity class of the tested compounds because we are developing a model that classifies small molecules according to their activities against the target. This information is available in the "**activity_outcome**" and "**activity_summary**" columns.

In [7]:

```
df_raw.groupby(['activity_outcome']).count()
```

Out[7]:

	pc_result_tag	sid	cid	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
activity_outcome														
Active	379	379	378	379	0	0	379	379	378	379	379	115	359	379
Inactive	7562	7562	7466	7562	0	0	7562	7562	0	7562	7562	324	7449	7562

	pc_result_tag	sid	cid	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
activity_outcome														
Inconclusive	2545	2545	2493	2545	0	0	2545	2545	2111	2136	2545	1206	2450	2545

Based on the data in the **activity_outcome** column, there are 379 actives, 7562 inactives, and 2545 inconclusives.

In [8]:

```
df_raw.groupby(['activity_outcome','activity_summary']).count()
```

Out[8]:

		pc_result_tag	sid	cid	activity_score	activity_url	assay_data_comment	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
activity_outcome	activity_summary													
Active	active antagonist	379	379	378	379	0	0	379	378	379	379	115	359	379
Inactive	inactive	7562	7562	7466	7562	0	0	7562	0	7562	7562	324	7449	7562
Inconclusive	active agonist	612	612	571	612	0	0	612	612	612	612	60	590	612
	inconclusive	44	44	44	44	0	0	44	0	0	44	19	42	44
	inconclusive agonist	414	414	409	414	0	0	414	212	223	414	12	397	414
	inconclusive agonist (cytotoxic)	59	59	59	59	0	0	59	41	45	59	59	59	59

		pc_result_tag	sid	cid	activity_score	activity_url	assay_data_comment	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
activity_outcome	activity_summary													
	inconclusive antagonist	367	367	364	367	0	0	367	227	230	367	8	313	367
	inconclusive antagonist (cytotoxic)	1049	1049	1046	1049	0	0	1049	1019	1026	1049	1048	1049	1049

Now, we can see that, in the **activity_summary** column, the inconclusive compounds are further classified into subclasses, which include:

- **active agonist**
- inconclusive
- inconclusive agonist
- inconclusive antagonist
- inconclusive agonist (cytotoxic)
- inconclusive antagonist (cytotoxic)

As implied in the title of this assay record (https://pubchem.ncbi.nlm.nih.gov/bioassay/743139), this assay aims to identify **aromatase inhibitors**. Therefore, all **active antagonists** (in the activity summary column) were declared to be **active** compounds (in the activity outcome column).

On the other hand, the assay also identified 612 **active agonists** (in the activity summary column), and they are declared to be **inconclusive** (in the activity outcome column).

With that said, "inactive" compounds in this assay means those which are neither active agonists nor active antagonist.

It is important to understand that the criteria used for determining whether a compound is active or not in a given assay are selected by the data source who submitted that assay data to PubChem. For the purpose of this assignment (which aims to develop a binary classifier that tells if a molecule is active or inactive against the target), we should clarify what we mean by "active" and "inactive".

- **active** : any compounds that can change (either increase or decrease) the activity of the target. This is equivalent to either **active antagonists** or **active agonists** in the activity summary column.
- **inactive** : any compounds that do not change the activity of the target. This is equivalent to **inactive** compounds in the activity summary column.

Select active/inactive compounds for model building

Now we want to select only the active and inactive compounds from the data frame (that is, active agonists, active antagonists, and inactives based on the "activity summary" column).

In [9]:

```
df = df_raw[ (df_raw['activity_summary'] == 'active agonist' ) |
             (df_raw['activity_summary'] == 'active antagonist' ) |
```

```
            (df_raw['activity_summary'] == 'inactive' ) ]

len(df)
```

Out[9]:

```
8553
```

In [10]:

```
print(len(df['sid'].unique()))
print(len(df['cid'].unique()))
```

```
8553
6864
```

Note that the number of CIDs is not the same as the number of SIDs. There are two important potential reasons for this observation.

First, not all substances (SIDs) in PubChem have associated compounds (CIDs) because some substances failed during structure standardization. [Remember that, in PubChem, substances are depositor-provided structures and compounds are unique structures extracted from substances through structure standardization.] Because our model will use structural information of molecules to predict their bioactivity, we need to remove substances without associated CIDs (i.e., no standardized structures).

Second, some compounds are associated with more than one substances. In the context of this assay, it means that a compound may be tested multiple times in different samples (which are designated as different substances). It is not uncommon that different samples of the same chemical may result in conflicting activities (e.g., active agonist in one sample but inactive in another sample). In this practice, we remove such compounds with conflicting activities.

Drop substances without associated CIDs.

First, check if there are subtances without associated CIDs.

In [11]:

```
df.isna().sum()
```

Out[11]:

```
pc_result_tag            0
sid                      0
cid                    138
activity_outcome         0
activity_score           0
activity_url          8553
assay_data_comment    8553
activity_summary         0
antagonist_activity      0
antagonist_potency    7563
antagonist_efficacy      0
viability_activity       0
viability_potency     8054
```

```
viability_efficacy      155
sample_source             0
dtype: int64
```

There are 138 records whose "cid" column is NULL, and we want to remove those records.

In [12]:

```
df = df.dropna( subset=['cid'] )
len(df)
```

Out[12]:

```
8415
```

In [13]:

```
print(len(df['sid'].unique()))
print(len(df['cid'].unique()))
```

```
8415
6863
```

In [14]:

```
df.isna().sum()   # Check if the NULL values disappeared in the "cid" column
```

Out[14]:

```
pc_result_tag             0
sid                       0
cid                       0
activity_outcome          0
activity_score            0
activity_url           8415
assay_data_comment     8415
activity_summary          0
antagonist_activity       0
antagonist_potency     7467
antagonist_efficacy       0
viability_activity        0
viability_potency      7919
viability_efficacy      154
sample_source             0
dtype: int64
```

Remove CIDs with conflicting activities

Now identify compounds with conflicting activities and remove them.

In [15]:

```
cid_conflict = []
idx_conflict = []

for mycid in df['cid'].unique() :

    outcomes = df[ df.cid == mycid ].activity_summary.unique()

    if len(outcomes) > 1 :

        idx_tmp = df.index[ df.cid == mycid ].tolist()
        idx_conflict.extend(idx_tmp)
        cid_conflict.append(mycid)

print("#", len(cid_conflict), "CIDs with conflicting activities [associated with", le
```

```
# 65 CIDs with conflicting activities [associated with 146 rows (SIDs).]
```

In [16]:

```
df.loc[idx_conflict,:].head(10)
```

Out[16]:

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
8	6	144203557.0	16043.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
5956	5954	144209507.0	16043.0	Active	43.0	NaN	NaN	active antagonist	active antagonist	54.4827	-73.4024	inconclusive antagonist	NaN	NaN	Sigma Aldrich
6850	6848	144210401.0	16043.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	SIGMA
52	50	144203601.0	443939.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI
6130	6128	144209681.0	443939.0	Active	61.0	NaN	NaN	active antagonist	active antagonist	1.65519	-115.932	active antagonist	12.1763	-120.598	Toronto Research

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
66	64	144203615.0	2170.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	BIOMOL
9118	9116	144212669.0	2170.0	Active	50.0	NaN	NaN	active antagonist	active antagonist	16.5803	-115.202	inconclusive antagonist	61.1306	-80.7706	SIGMA
106	104	144203655.0	2554.0	Inconclusive	20.0	NaN	NaN	active agonist	active agonist	2.87255	73.7025	inactive	NaN	0	Sigma Aldrich
5920	5918	144209471.0	2554.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	SIGMA
6964	6962	144210515.0	2554.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	SIGMA

In [17]:

```
df = df.drop(idx_conflict)
```

In [18]:

```
df.groupby('activity_summary').count()
```

Out[18]:

activity_summary	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
active agonist	537	537	537	537	537	0	0	537	537	537	537	58	517	537
active antagonist	343	343	343	343	343	0	0	343	342	343	343	108	326	343
inactive	7389	7389	7389	7389	7389	0	0	7389	0	7389	7389	318	7278	7389

In [19]:

```
print(len(df['sid'].unique()))
print(len(df['cid'].unique()))
```

```
8269
6798
```

Remove redundant data

The above code cells [in 3-(2)] do not remove compounds tested multiple times if the testing results are consistent [e.g., active agonist in all samples (substances)]. The rows corresponding to these compounds are redundant, so we want remove them except for only one row for each compound.

In [20]:

```
df = df.drop_duplicates(subset='cid')  # remove duplicate rows except for the first o
print(len(df['sid'].unique()))
print(len(df['cid'].unique()))
```

```
6798
6798
```

Adding "numeric" activity classes

In general, the inputs and outputs to machine learning algorithms need to have numerical forms.
In this practice, the input (molecular structure) will be represented with binary fingerprints, which already have numerical forms (0 or 1). However, the output (activity) is currently in a string format (e.g., 'active agonist', 'active antagonist'). Therefore, we want to add an additional, 'activity' column, which contains numeric codes representing the active and inactive compounds:

- 1 for actives (either active agonists or active antagonists)
- 0 for inactives

Note that we are merging the two classes "active agonist" and "active antagonist", because we are going to build a binary classifer that distinguish actives from inactives.

In [21]:

```
df['activity'] = [ 0 if x == 'inactive' else 1 for x in df['activity_summary'] ]
```

Check if the new column 'activity' is added to (the end of) the data frame.

In [22]:

```
df.head(3)
```

Out[22]:

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source	activity

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source	activity
3	1	144203552.0	12850184.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI	0
4	2	144203553.0	89753.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI	0
5	3	144203554.0	9403.0	Inactive	0.0	NaN	NaN	inactive	inactive	NaN	0	inactive	NaN	0	NCI	0

Double-check the count of active/inactive compounds.

In [23]:

```
df.groupby('activity_summary').count()
```

Out[23]:

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source	activity
activity_summary															
active agonist	451	451	451	451	451	0	0	451	451	451	451	44	432	451	451
active antagonist	291	291	291	291	291	0	0	291	290	291	291	88	275	291	291
inactive	6056	6056	6056	6056	6056	0	0	6056	0	6056	6056	269	5970	6056	6056

In [24]:

```
df.groupby('activity').count()
```

Out[24]:

	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source

activity	pc_result_tag	sid	cid	activity_outcome	activity_score	activity_url	assay_data_comment	activity_summary	antagonist_activity	antagonist_potency	antagonist_efficacy	viability_activity	viability_potency	viability_efficacy	sample_source
activity															
0	6056	6056	6056	6056	6056	0	0	6056	6056	0	6056	6056	269	5970	6056
1	742	742	742	742	742	0	0	742	742	741	742	742	132	707	742

Create a smaller data frame that only contains CIDs and activities.

Let's create a smaller data frame that only contains CIDs and activities. This data frame will be merged with a data frame containing molecular fingerprint information.

In [25]:

```
df_activity = df[['cid','activity']]
```

In [26]:

```
df_activity.head(5)
```

Out[26]:

	cid	activity
3	12850184.0	0
4	89753.0	0
5	9403.0	0
6	13218779.0	0
12	637566.0	0

Download structure information for each compound from PubChem

Now we want to get structure information of the compounds from PubChem (in isomeric SMILES).

In [27]:

```
cids = df.cid.astype(int).tolist()
```

In [28]:

```
chunk_size = 200
num_cids = len(cids)

if num_cids % chunk_size == 0 :
    num_chunks = int( num_cids / chunk_size )
else :
    num_chunks = int( num_cids / chunk_size ) + 1
```

```
print("# CIDs = ", num_cids)
print("# CID Chunks = ", num_chunks, "(chunked by ", chunk_size, ")")
```

```
# CIDs =  6798
# CID Chunks =  34 (chunked by  200 )
```

In [29]:

```
import time
import requests
from io import StringIO

df_smiles = pd.DataFrame()
list_dfs = []  # temporary list of data frames

for i in range(0, num_chunks) :

    idx1 = chunk_size * i
    idx2 = chunk_size * (i + 1)
    cidstr = ",".join( str(x) for x in cids[idx1:idx2] )

    url = ('https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/cid/' + cidstr + '/pro
    res = requests.get(url)
    data = pd.read_csv( StringIO(res.text), header=None, names=['smiles'] )
    list_dfs.append(data)

    time.sleep(0.2)

    if ( i % 5 == 0 ) :
        print("Processing Chunk ", i)

#    if ( i == 2 ) : break  #- for debugging

df_smiles = pd.concat(list_dfs,ignore_index=True)
df_smiles[ 'cid' ] = cids
df_smiles.head(5)
```

```
Processing Chunk  0
Processing Chunk  5
Processing Chunk  10
Processing Chunk  15
Processing Chunk  20
Processing Chunk  25
Processing Chunk  30
```

Out[29]:

	smiles	cid
0	C(C(=O)[C@H]([C@@H]([C@H](C(=O)[O-])O)O)O)O.C(...	12850184
1	C([C@H]([C@H]([C@@H]([C@H](C(=O)[O-])O)O)O)O)O...	89753
2	C[C@]12CC[C@H]3[C@H]([C@@H]1CC[C@@H]2OC(=O)CCC...	9403
3	C[C@@]12CC[C@@H](C1(C)C)C[C@H]2OC(=O)CSC#N	13218779
4	CC(=CCC/C(=C/CO)/C)C	637566

In [30]:

```
len(df_smiles)
```

Out[30]:

```
6798
```

In [31]:

```
df_smiles = df_smiles[['cid','smiles']]
df_smiles.head(5)
```

Out[31]:

	cid	smiles
0	12850184	C(C(=O)[C@H]([C@@H]([C@H](C(=O)[O-])O)O)O)O.C(...
1	89753	C([C@H]([C@H]([C@@H]([C@H](C(=O)[O-])O)O)O)O)O...
2	9403	C[C@]12CC[C@H]3[C@H]([C@@H]1CC[C@@H]2OC(=O)CCC...
3	13218779	C[C@@]12CC[C@@H](C1(C)C)C[C@H]2OC(=O)CSC#N
4	637566	CC(=CCC/C(=C/CO)/C)C

Generate MACCS keys from SMILES.

In [32]:

```
from rdkit import Chem
from rdkit.Chem import MACCSkeys
```

In [33]:

```
fps=dict()

for idx, row in df_smiles.iterrows() :

    mol = Chem.MolFromSmiles(row.smiles)

    if mol == None :
        print("Can't generate MOL object:", "CID", row.cid, row.smiles)
    else:
        fps[row.cid] = [row.cid] + list(MACCSkeys.GenMACCSKeys(mol).ToBitString())
```

```
Can't generate MOL object: CID 28145 [NH4+].[NH4+].F[Si-2](F)(F)(F)(F)F
Can't generate MOL object: CID 28127 F[Si-2](F)(F)(F)(F)F.[Na+].[Na+]
```

In [34]:

```
# Generate column names
fpbitnames = []

fpbitnames.append('cid')

for i in range(0,167):   # from MACCS000 to MACCS166
    fpbitnames.append( "maccs" + str(i).zfill(3) )

df_fps = pd.DataFrame.from_dict(fps, orient='index', columns=fpbitnames)
```

In [35]:

```
df_fps.head(5)
```

Out[35]:

	cid	maccs000	maccs001	maccs002	maccs003	maccs004	maccs005	maccs006	maccs007	maccs008	...	maccs157	maccs158	maccs159	maccs160	maccs161	maccs162	maccs163	maccs164	maccs165	maccs166
12850184	12850184	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
89753	89753	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
9403	9403	0	0	0	0	0	0	0	0	0	...	1	0	1	1	0	1	1	1	1	0
13218779	13218779	0	0	0	0	0	0	0	0	0	...	1	0	1	1	1	0	1	1	1	0

	cid	mac cs000	mac cs001	mac cs002	mac cs003	mac cs004	mac cs005	mac cs006	mac cs007	mac cs008	...	mac cs157	mac cs158	mac cs159	mac cs160	mac cs161	mac cs162	mac cs163	mac cs164	mac cs165	mac cs166
637566	637566	0	0	0	0	0	0	0	0	0	...	1	0	0	1	0	0	0	1	0	0

5 rows × 168 columns

Merge activity data and fingerprint information

In [36]:

```
df_activity.head(3)
```

Out[36]:

	cid	activity
3	12850184.0	0
4	89753.0	0
5	9403.0	0

In [37]:

```
df_fps.head(3)
```

Out[37]:

	cid	mac cs000	mac cs001	mac cs002	mac cs003	mac cs004	mac cs005	mac cs006	mac cs007	mac cs008	...	mac cs157	mac cs158	mac cs159	mac cs160	mac cs161	mac cs162	mac cs163	mac cs164	mac cs165	mac cs166
12850184	12850184	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
89753	89753	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
9403	9403	0	0	0	0	0	0	0	0	0	...	1	0	1	1	0	1	1	1	1	0

3 rows × 168 columns

In [38]:

```
df_data = df_activity.join(df_fps.set_index('cid'), on='cid')
```

In Section 5, there were two CIDs for which the MACCS keys could not be generated. They need to be removed from **df_data**.

In [39]:

```
df_data[df_data.isna().any(axis=1)]
```

Out[39]:

	cid	activity	macs000	macs001	macs002	macs003	macs004	macs005	macs006	macs007	...	macs157	macs158	macs159	macs160	macs161	macs162	macs163	macs164	macs165	macs166
2293	28145.0	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9077	28127.0	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

2 rows × 169 columns

In [40]:

```
df_data = df_data.dropna()
len(df_data)
```

Out[40]:

```
6796
```

Save df_data in CSV for future use.

In [41]:

```
df_data.to_csv('df_data.csv')
```

Preparation for model building

Loading the data into X and y.

In [42]:

```
df_data.head(3)
```

Out[42]:

	cid	activity	macs000	macs001	macs002	macs003	macs004	macs005	macs006	macs007	...	macs157	macs158	macs159	macs160	macs161	macs162	macs163	macs164	macs165	macs166
3	12850184.0	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
4	89753.0	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
5	9403.0	0	0	0	0	0	0	0	0	0	...	1	0	1	1	0	1	1	1	1	0

3 rows × 169 columns

In [43]:

```
X = df_data.iloc[:,2:]
y = df_data['activity'].values
```

In [44]:

```
X.head(3)
```

Out[44]:

	macs0000	macs0001	macs0002	macs0003	macs0004	macs0005	macs0006	macs0007	macs0008	macs0009	...	macs1157	macs1158	macs1159	macs1160	macs1161	macs1162	macs1163	macs1164	macs1165	macs1166
3	0	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
4	0	0	0	0	0	0	0	0	0	0	...	1	0	1	0	0	0	0	1	0	1
5	0	0	0	0	0	0	0	0	0	0	...	1	0	1	1	0	1	1	1	1	0

3 rows × 167 columns

In [45]:

```
print(len(y))    # Number of all compounds
y.sum()          # Number of actives
```

```
6796
```

Out[45]:

```
742
```

Remove zero-variance features

Some features in X are not helpful in distinguishing actives from inactives, because they are set ON for all compounds or OFF for all compounds. Such features need to be removed because they would consume more computational resources without improving the model.

In [46]:

```
from sklearn.feature_selection import VarianceThreshold
```

In [47]:

```
X.shape  #- Before removal
```

Out[47]:

```
(6796, 167)
```

In [48]:

```
sel = VarianceThreshold()
X=sel.fit_transform(X)
X.shape  #- After removal
```

Out[48]:

```
(6796, 163)
```

In this case, four features had zero variances. Note that one of them is the first bit (maccs000) of the MACCS keys, which is added as a "dummy" to name each of bits 1~166 as maccs001, maccs002, ... maccs166.

Train-Test-Split (a 9:1 ratio)

Now split the data set into a training set (90%) and test set (10%). The training set will be used to train the model. The developed model will be tested against the test set.

In [49]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, shuffle=True, random_state=3100, stratify=y, test_size=0.1

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
print(y_train.sum(), y_test.sum())
```

```
(6116, 163) (680, 163) (6116,) (680,)
668 74
```

Balance the training set through downsampling

Check the dimension of the training data set.

In [50]:

```
print(len(y_train))
print(len(X_train))
print(len(X_train[0]))
```

```
6116
6116
163
```

Check the number of actives and inactives compound.

In [51]:

```
print("# inactives : ", len(y_train) - y_train.sum())
print("# actives   : ", y_train.sum())
```

```
# inactives :  5448
# actives   :  668
```

The data set is highly imbalanced [the inactive to active ratio is 8.16 (=5448 / 668)]. To address this issue, let's downsample the majority class (inactive compounds) to balance the data set.

In [52]:

```
# Indicies of each class' observations
idx_inactives = np.where( y_train == 0 )[0]
idx_actives   = np.where( y_train == 1 )[0]

# Number of observations in each class
num_inactives = len(idx_inactives)
num_actives   = len(idx_actives)

# Randomly sample from inactives without replacement
np.random.seed(0)
idx_inactives_downsampled = np.random.choice(idx_inactives, size=num_actives, replace=
# Join together downsampled inactives with actives
X_train = np.vstack((X_train[idx_inactives_downsampled], X_train[idx_actives]))
y_train = np.hstack((y_train[idx_inactives_downsampled], y_train[idx_actives]))
```

It is noteworthy that **np.vstack** is used for X_train and **np.hstack** is used for Y_train. The direction of stacking is different because X_train is a 2-D array and y_train is a 1-D array.

Confirm that the downsampled data set has the correct dimension and active/inactive counts.

In [53]:

```
print("# inactives : ", len(y_train) - y_train.sum())
print("# actives   : ", y_train.sum())
```

```
# inactives :  668
# actives   :  668
```

In [54]:

```
print(len(y_train))
print(len(X_train))
print(len(X_train[0]))
```

```
1336
1336
163
```

Build a model using the training set.

Now we are ready to build predictive models using machine learning algorithms available in the scikit-learn library (https://scikit-learn.org/). This notebook will use Naive Bayes and decisiont tree, because they are relatively fast and simple.

In [55]:

```
from sklearn.naive_bayes import BernoulliNB        #-- Naive Bayes
from sklearn.tree import DecisionTreeClassifier    #-- Decision Tree
```

In [56]:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
```

Naive Bayes

In [57]:

```
clf = BernoulliNB()            # set up the NB classification model
```

In [58]:

```
clf.fit( X_train ,y_train )    # Train the model by fitting it to the data.
```

Out[58]:

```
BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
```

In [59]:

```
y_true, y_pred = y_train, clf.predict( X_train )    # Apply the model to predict the
```

In [60]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
               #  [FN, TP]]
```

```
[[462 206]
 [199 469]]
```

In [61]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )    # TP / (FN + TP)
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )    # TN / (TN + FP )
bacc = (sens + spec) / 2
```

```
y_score = clf.predict_proba( X_train )[:, 1]
auc = roc_auc_score( y_true, y_score )
```

In [62]:

```
print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.6968562874251497
#-- Balanced Accuracy =  0.6968562874251496
#-- Sensitivity       =  0.7020958083832335
#-- Specificity       =  0.6916167664670658
#-- AUC-ROC           =  0.7496985818781599
```

When applied to predict the activity of the training compounds, the NB classifier resulted in the accuracy of 0.70 and AUC-ROC of 0.75. However, the real performance of the model should be evaluated with the test set data, which are not used for model training.

In [63]:

```
y_true, y_pred = y_test, clf.predict(X_test)    #-- Apply the model to predict the te
```

In [64]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
               #  [FN, TP]]
```

```
[[412 194]
 [ 28  46]]
```

In [65]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )
bacc = (sens + spec) / 2

y_score = clf.predict_proba( X_test )[:, 1]
auc = roc_auc_score( y_true, y_score )

print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
```

```
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.6735294117647059
#-- Balanced Accuracy =  0.6507448042101507
#-- Sensitivity       =  0.6216216216216216
#-- Specificity       =  0.6798679867986799
#-- AUC-ROC           =  0.724099099099099
```

For the test set, the accuracy is 0.67 and the AUC-ROC is 0.72. These values are somewhat smaller (by 0.03) than those for the training set. Also note that the accuracy is no longer the same as the balanced accruacy (which is the average of the sensitivity and specificity).

Some additional performance information may be obtained using **classification_report()**.

In [66]:

```
print( classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.68      0.79       606
           1       0.19      0.62      0.29        74

    accuracy                           0.67       680
   macro avg       0.56      0.65      0.54       680
weighted avg       0.86      0.67      0.73       680
```

Decision Tree

In [67]:

```
clf = DecisionTreeClassifier( random_state=0 )    # set up the DT classification mode
```

In [68]:

```
clf.fit( X_train ,y_train )    # Train the model by fitting it to the data (using the
```

Out[68]:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=0, splitter='best')
```

In [69]:

```
y_true, y_pred = y_train, clf.predict( X_train )    # Apply the model to predict the
```

In [70]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
               #  [FN, TP]]
```

```
[[663   5]
 [  3 665]]
```

In [71]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )    # TP / (FN + TP)
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )    # TN / (TN + FP )
bacc = (sens + spec) / 2

y_score = clf.predict_proba( X_train )[:, 1]
auc = roc_auc_score( y_true, y_score )
```

In [72]:

```
print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.9940119760479041
#-- Balanced Accuracy =  0.9940119760479043
#-- Sensitivity       =  0.9955089820359282
#-- Specificity       =  0.9925149700598802
#-- AUC-ROC           =  0.9998890691670551
```

When applied to predict the activity of the **training** compounds, the DT classifier resulted in very high scores (>0.99) for all five performance measures considered here. However, it does **not** necessarily mean that the model will perform very well for the **test** set compounds. Let's apply the model to the test set.

In [73]:

```
y_true, y_pred = y_test, clf.predict(X_test)    #-- Apply the model to predict the te
```

In [74]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
```

```
#  [FN, TP]]
```

```
[[422 184]
 [ 32  42]]
```

In [75]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )
bacc = (sens + spec) / 2

y_score = clf.predict_proba( X_test )[:, 1]
auc = roc_auc_score( y_true, y_score )

print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.6823529411764706
#-- Balanced Accuracy =  0.6319686022656319
#-- Sensitivity       =  0.5675675675675675
#-- Specificity       =  0.6963696369636964
#-- AUC-ROC           =  0.6298724467041299
```

When the DT model was applied to the test set, all performance measures were much worse than those for the training set. This is a typical example of **outfitting**.

Model building through cross-validation

In the above section, the models were developed using the default values for many optional hyperparamters, which cannot be learned by the training algorithm. For example, when building a decision tree model, one should specify how the tree should be deep, how many compounds should be allowed in a single leaf, what is the minimum number of compounds in a single leaf, etc.

The cells below demonstrate how to perform hyperparameter optimization through 10-fold cross-validation. In this example, five values for each of three hyperparameters used in decision tree are considered (max_depth, min_samples_split, and min_samples_leaf), resulting in a total of 125 combination of the parameter values (=5 x 5 x 5). For each combination, 10 models are generated (through 10-fold cross validation) and the average performance will be tracked. The goal is to find the parameter value combination that results in the highest average performance score (e.g., 'roc_auc') from the 10-fold cross validation.

In [76]:

```
from sklearn.model_selection import GridSearchCV
```

In [77]:

```
scores = [ 'roc_auc', 'balanced_accuracy' ]
```

In [78]:

```
ncvs = 10

max_depth_range         = np.linspace( 3, 7, num=5, dtype='int32' )
min_samples_split_range = np.linspace( 3, 7, num=5, dtype='int32' )
min_samples_leaf_range  = np.linspace( 2, 6, num=5, dtype='int32' )

param_grid = dict( max_depth=max_depth_range,
                   min_samples_split=min_samples_split_range,
                   min_samples_leaf=min_samples_leaf_range )

clf = GridSearchCV( DecisionTreeClassifier( random_state=0 ),
                    param_grid=param_grid, cv=ncvs, scoring=scores, refit='roc_auc',
                    return_train_score = True, iid=False)
```

In [79]:

```
clf.fit( X_train, y_train )
print("Best parameter set", clf.best_params_)
```

```
Best parameter set {'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 3}
```

If necessary, it is possible to look into the performance data for each parameter value combination (stored in **clf.cv*results***), as shown in the following cell.

In [80]:

```
means_1a = clf.cv_results_['mean_train_roc_auc']
stds_1a  = clf.cv_results_['std_train_roc_auc']

means_1b = clf.cv_results_['mean_test_roc_auc']
stds_1b  = clf.cv_results_['std_test_roc_auc']

means_2a = clf.cv_results_['mean_train_balanced_accuracy']
stds_2a  = clf.cv_results_['std_train_balanced_accuracy']

means_2b = clf.cv_results_['mean_test_balanced_accuracy']
stds_2b  = clf.cv_results_['std_test_balanced_accuracy']

iterobjs = zip( means_1a, stds_1a, means_1b, stds_1b,
                means_2a, stds_2a, means_2b, stds_2b, clf.cv_results_['params'] )

for m1a, s1a, m1b, s1b, m2a, s2a, m2b, s2b, params in iterobjs :

    print( "Grid %r : %0.4f %0.04f %0.4f %0.04f %0.4f %0.04f %0.4f %0.04f"
           % ( params, m1a, s1a, m1b, s1b, m2a, s2a, m2b, s2b))
```

```
Grid {'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 3} : 0.7714 0.0042 (
Grid {'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 4} : 0.7714 0.0042 (
Grid {'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 5} : 0.7714 0.0042 (
Grid {'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 6} : 0.7714 0.0042 (
Grid {'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 7} : 0.7714 0.0042 (
.
.
.

Grid {'max_depth': 7, 'min_samples_leaf': 6, 'min_samples_split': 4} : 0.8873 0.0072 (
Grid {'max_depth': 7, 'min_samples_leaf': 6, 'min_samples_split': 5} : 0.8873 0.0072 (
Grid {'max_depth': 7, 'min_samples_leaf': 6, 'min_samples_split': 6} : 0.8873 0.0072 (
Grid {'max_depth': 7, 'min_samples_leaf': 6, 'min_samples_split': 7} : 0.8873 0.0072 (
```

Uncomment the following cell to look into additional performance data stored in cv*result*.

In [81]:

```
#print(clf.cv_result_)
```

It is important to understand that each model built through 10-fold cross-validation during hyperparameter optimization uses only 90% of the compounds in the training set and the remaining 10% is used for testing that model. After all parameter value combinations are evaluated, the best parameter values are selected and used to rebuild a model from **all** compounds in the training set. **GridSearchCV()** takes care of this last step automatically. Therefore, there is no need to take an extra step to build a model using **cls.fit()** after hyperparameter optimization.

In [82]:

```
y_true, y_pred = y_train, clf.predict( X_train )    # Apply the model to predict the
```

In [83]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
               #  [FN, TP]]
```

```
[[543 125]
 [179 489]]
```

In [84]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )    # TP / (FN + TP)
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )    # TN / (TN + FP )
bacc = (sens + spec) / 2

y_score = clf.predict_proba( X_train )[:, 1]
auc = roc_auc_score( y_true, y_score )
```

In [85]:

```
print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.7724550898203593
#-- Balanced Accuracy =  0.7724550898203593
#-- Sensitivity       =  0.7320359281437125
#-- Specificity       =  0.812874251497006
#-- AUC-ROC           =  0.8336452095808383
```

Compare these performance data with those from section 8-(2) (for the training set). When the default values were used, the DT model gave >0.99 for all performance measures, but the current models (developed using hyperparameter optimization) have much lower values, ranging from 0.73 to 0.83. Again, however, what really matters is the performance against the test set, which contains the data not used for model training.

In [86]:

```
y_true, y_pred = y_test, clf.predict(X_test)    #-- Apply the model to predict the te
```

In [87]:

```
CMat = confusion_matrix( y_true, y_pred )    #-- generate confusion matrix
print(CMat)    # [[TN, FP],
               #  [FN, TP]]
```

```
[[457 149]
 [ 26  48]]
```

In [88]:

```
acc  = accuracy_score( y_true, y_pred )

sens = CMat[ 1 ][ 1 ] / ( CMat[ 1 ][ 0 ] + CMat[ 1 ][ 1 ] )
spec = CMat[ 0 ][ 0 ] / ( CMat[ 0 ][ 0 ] + CMat[ 0 ][ 1 ] )
bacc = (sens + spec) / 2

y_score = clf.predict_proba( X_test )[:, 1]
auc = roc_auc_score( y_true, y_score )

print("#-- Accuracy          = ", acc)
print("#-- Balanced Accuracy = ", bacc)
print("#-- Sensitivity       = ", sens)
print("#-- Specificity       = ", spec)
print("#-- AUC-ROC           = ", auc)
```

```
#-- Accuracy          =  0.7426470588235294
#-- Balanced Accuracy =  0.7013870305949514
#-- Sensitivity       =  0.6486486486486487
#-- Specificity       =  0.7541254125412541
#-- AUC-ROC           =  0.7496209080367496
```

Now we can see that the model from hyperparameter optimization gives better performance data against the test set, compared to the model developed using the default parameter values. Importantly, the model from hyperparameter optimization shows smaller differences in performance measures between the training and test sets, indicatiing that the issue of outffiting has been alleviated substantially.

Exercises

In this assignment, we will build predictive models using the same aromatase data.

step 1 Show the following information to make sure that the activity data in the **df_activity** data frame is still available.

- The first five lines of **df_activity**

In [89]:

```
# Write your code in this cell.
```

- The counts of active/inactive compounds in **df_activity**

In [90]:

```
# Write your code in this cell.
```

Step 2 Show the following information to make sure the structure data is still available.

- The first five lines of **df_smiles**

In [91]:

```
# Write your code in this cell.
```

- the number of rows of **df_smiles**

In [92]:

```
# Write your code in this cell.
```

Step 3 Generate the (ECFP-equivalent) circular fingerprints from the SMILES strings.

- Use RDKit to generate 1024-bit-long circular fingerprints.
- Set the radius of the circular fingerprint to 2.
- Store the fingerprints in a data_frame called **df_fps** (along with the CIDs).
- Print the dimension of **df_fps**.
- Show the first five lines of **df_fps**.

In [93]:

```
# Write your code in this cell
```

Step 4 Merge the **df_activity** and **df_fps** data frames into a data frame called **df_data**

- Join the two data frames using the CID column as keys.

- Remove the rows that have any NULL values (i.e., compounds for which the fingerprints couldn't be generated).
- Print the dimension of **df_data**.
- Show the first five lines of **df_data**.

In [94]:

```
# Write your code in this cell.
```

Step 5 Prepare input and output data for model building

- Load the fingerprint data into 2-D array (X) and the activity data into 1-D array (y).
- Show the dimension of X and y.

In [95]:

```
# Write your code in this cell.
```

- Remove zero-variance features from X (if any).

In [96]:

```
# Write your code in this cell.
```

- Split the data set into training and test sets (90% vs 10%) (using random_state=3100).
- Print the dimension of X and y for the training and test sets.

In [97]:

```
# Write your code in this cell.
```

- Balance the training data set through downsampling.
- Show the number of inactive/active compounds in the downsampled training set.

In [98]:

```
# Write your code in this cell.
```

Step 6 Building a Random Forest model using the balanced training data set.

- First read the followng documents about random forest (https://scikit-learn.org/stable/modules/ensemble.html#forest and https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier).
- Use 10-fold cross validation to select the best value for the "n_estimators" parameter that maximizes the **balanced accuracy**. Test 40 values from 5 to 200 with an increment of 5 (e.g., 5, 10, 15, 20, ..., 190, 195, 200).
- For parameters 'max_depth', 'min_samples_leaf', and 'min_samples_split', use the best values found in Section 9.
- For other parameters, use the default values.
- For each parameter value, print the mean balanced accuracies (for both training and test from cross validation).

In [99]:

```
# Write your code in this cell.
```

Step 7 Apply the developed RF model to predict the activity of the **training** set compounds.

- Report the confusion matrix.
- Report the accuracy, balanced accurayc, sensitivity, specificity, and auc-roc.

In [100]:

```
# Write your code in this cell.
```

Step 8 Apply the developed RF model to predict the activity of the **test** set compounds.

- Report the accuracy, balanced accurayc, sensitivity, specificity, and auc-roc.

In [101]:

```
# Write your code in this cell.
```

Step 9 Read a recent paper published in *Chem. Res. Toxicol.* (https://doi.org/10.1021/acs.chemrestox.7b00037) and answer the following questions (in no more than five sentences for each question).

- What different approaches did the paper take to develop prediction models (compared to those used in this notebook)?
- How different are the models reported in the paper from those constructed in this paper (in terms of the performance measures)?
- What would you do to develop models with improved performance?

Write your answers in this cell.

-
-
-

In [ ]:

```

```