Clustering and machine learning

- Molecular similarity
- MCSS
- Clustering
- Machine learning: QSAR
- Validation

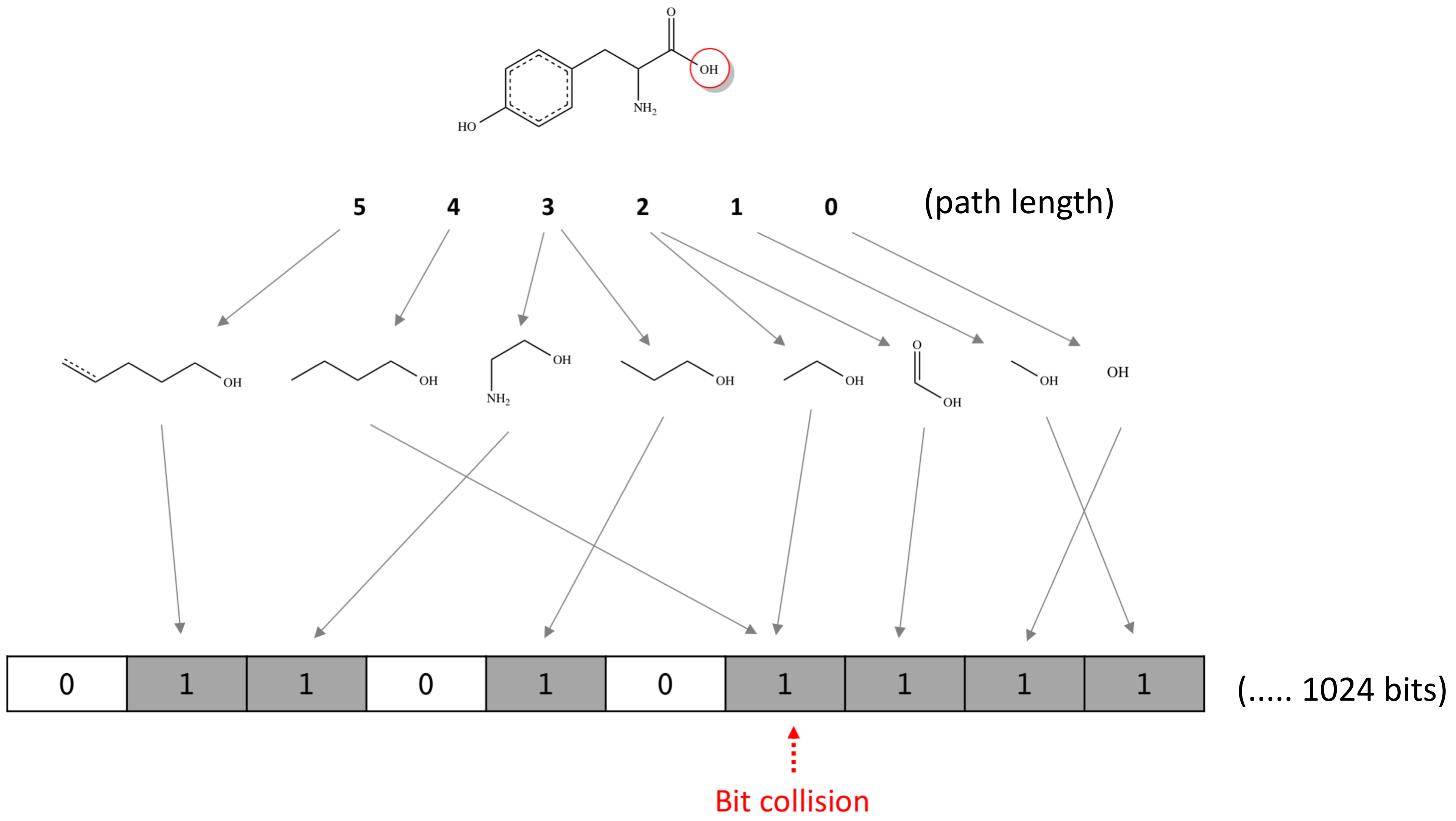
Molecular similarity

- Molecular fingerprints
  - Linear path-based
  - Circular path-based
  - Substructure-based
- Calculating similarity
  - Tanimoto
  - Tversky

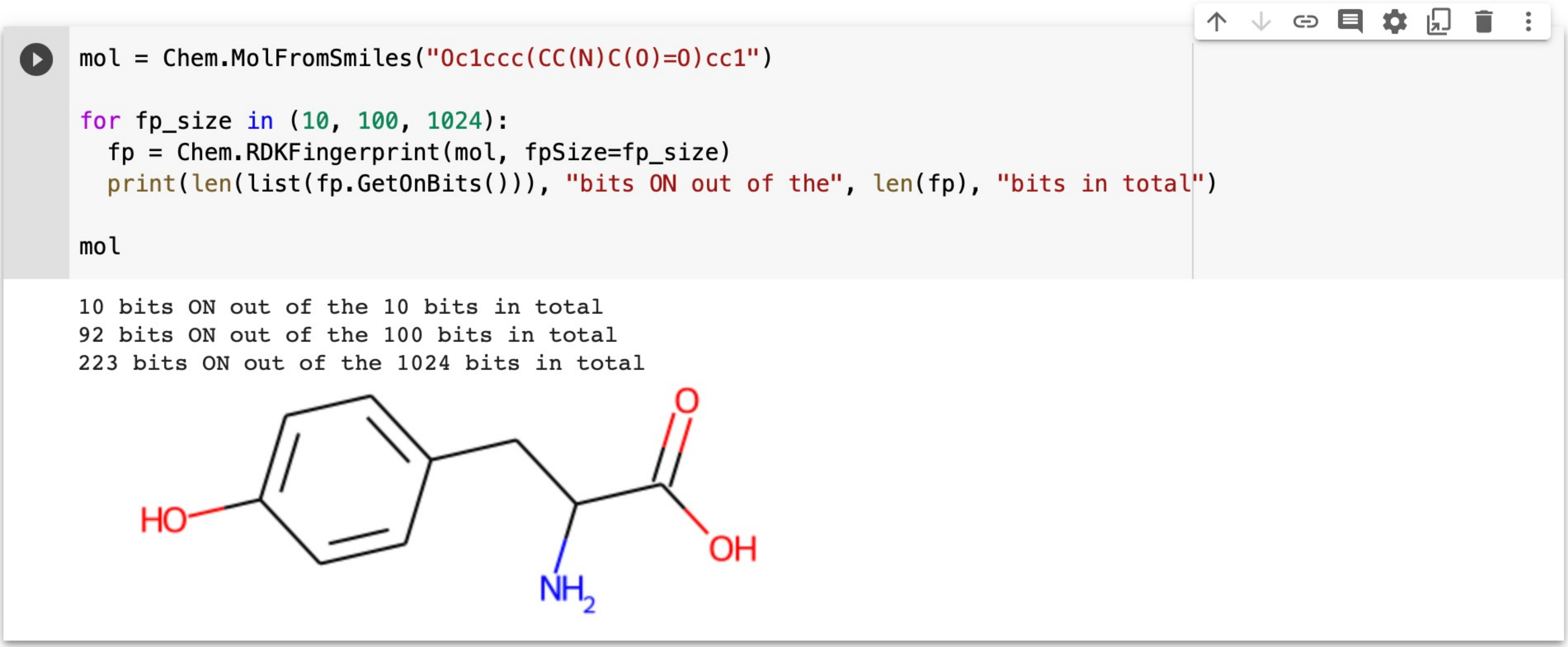
Fingerprints (FP’s)

- Bitwise representation of a molecule
- Each bit reflects the presence or absence of certain chemical features in the molecule
- Typically there are 166, 1024 of 2048 bits to represent a single molecule
- FP’s depend on many user-definable settings and the underlying algorithm
  - Comparison is only valid when calculated in a similar way!

Linear path-based FP's (Daylight)

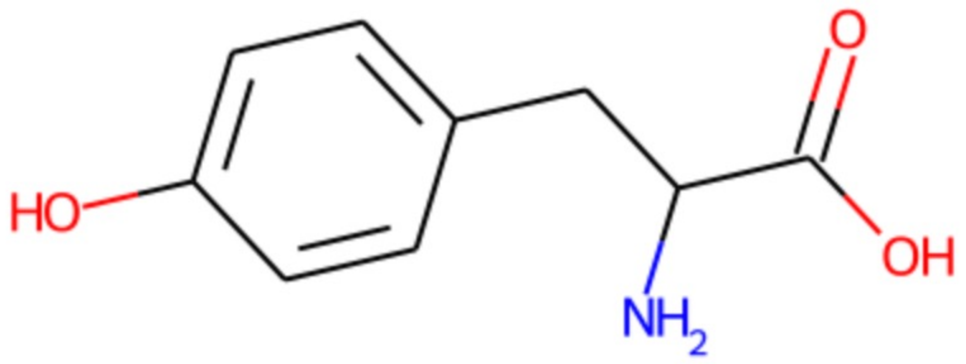


FP size

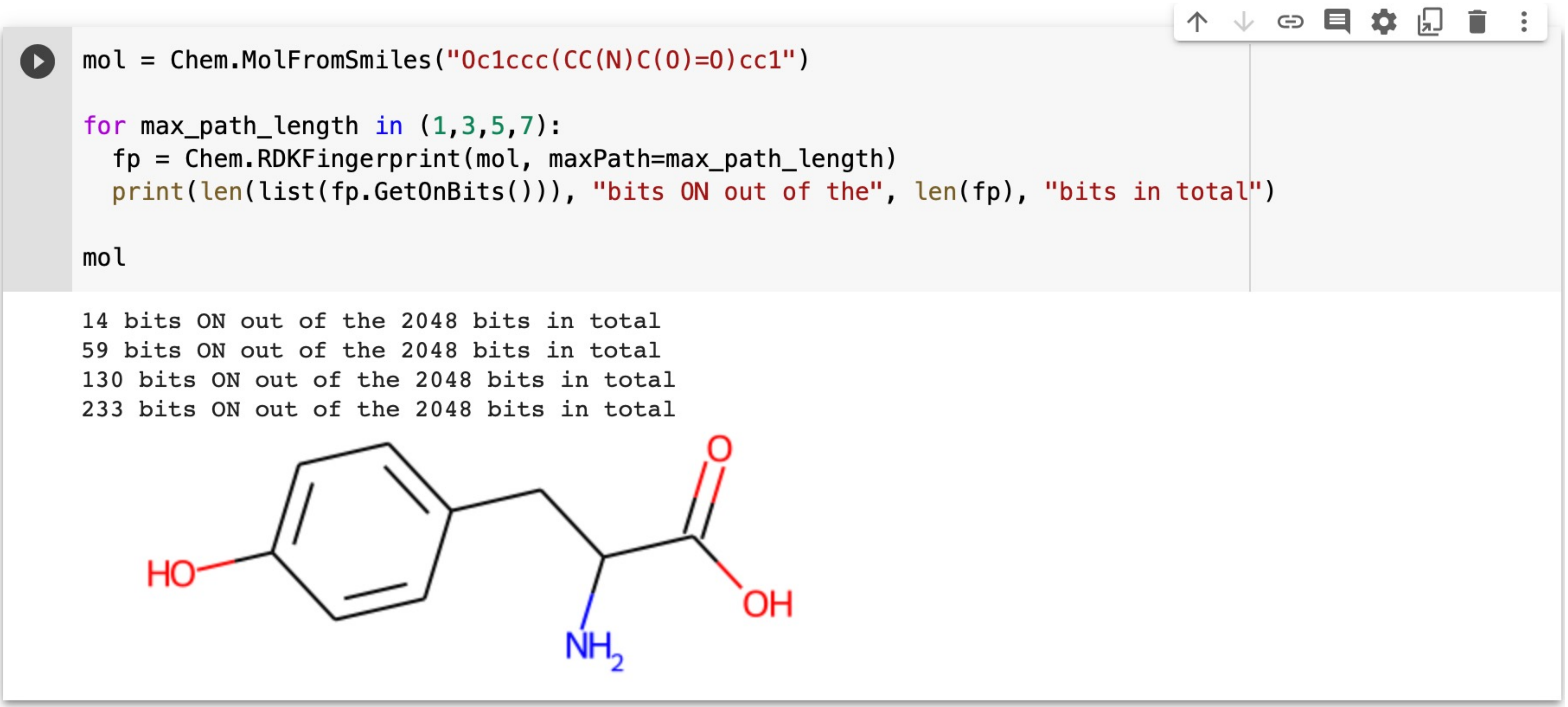


```python
mol = Chem.MolFromSmiles("Oc1ccc(CC(N)C(O)=O)cc1")

for fp_size in (10, 100, 1024):
  fp = Chem.RDKFingerprint(mol, fpSize=fp_size)
  print(len(list(fp.GetOnBits())), "bits ON out of the", len(fp), "bits in total")

mol
```

```
10 bits ON out of the 10 bits in total
92 bits ON out of the 100 bits in total
223 bits ON out of the 1024 bits in total
```

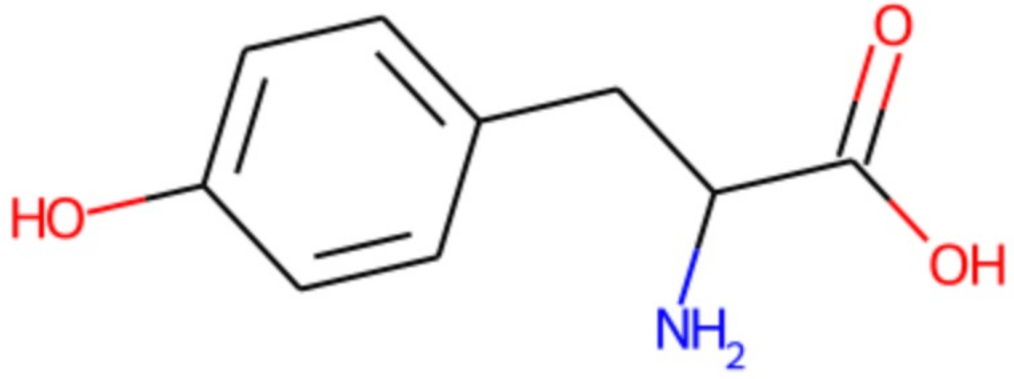


FP path length

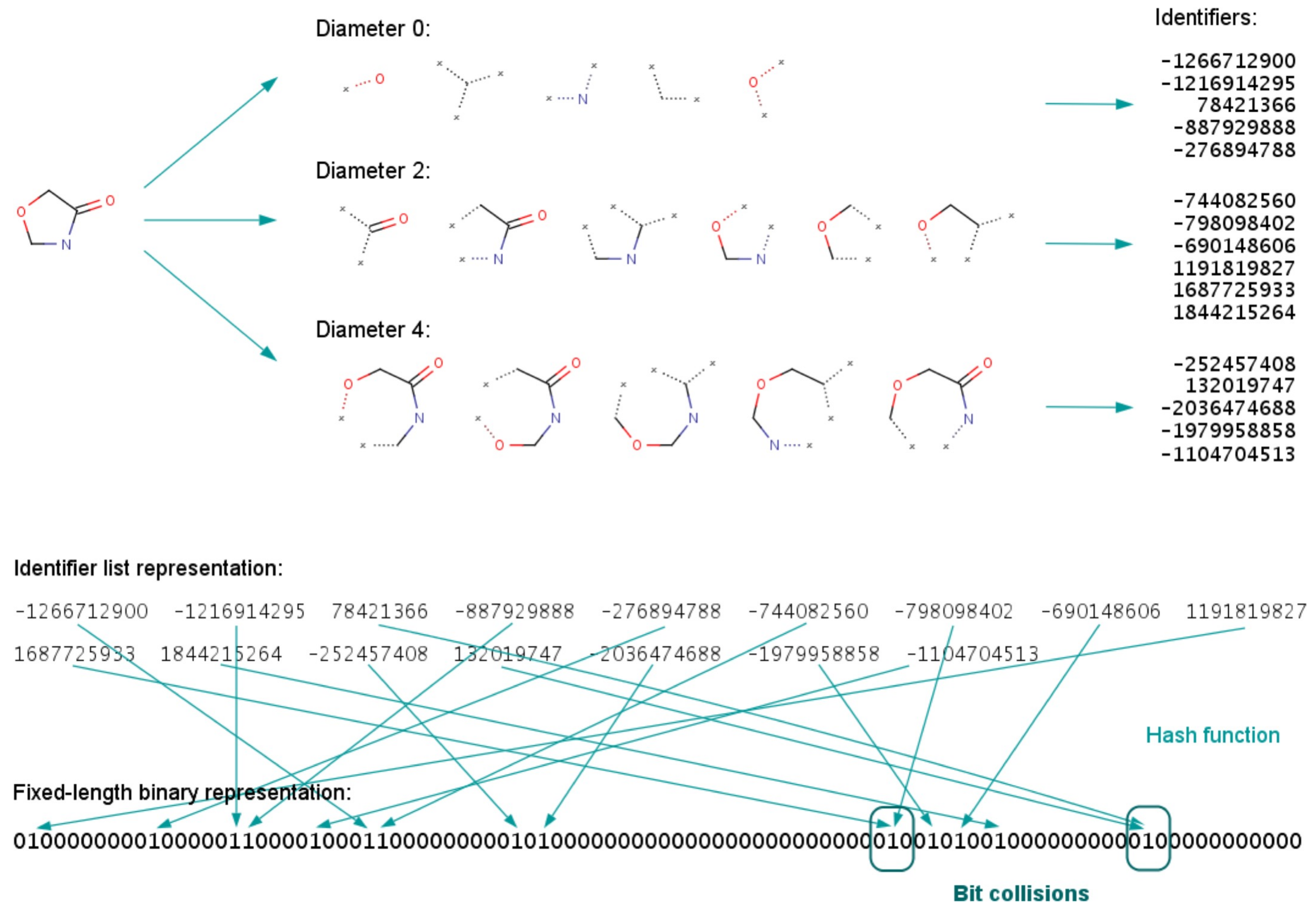


```python
mol = Chem.MolFromSmiles("Oc1ccc(CC(N)C(O)=O)cc1")

for max_path_length in (1,3,5,7):
  fp = Chem.RDKFingerprint(mol, maxPath=max_path_length)
  print(len(list(fp.GetOnBits())), "bits ON out of the", len(fp), "bits in total")

mol
```

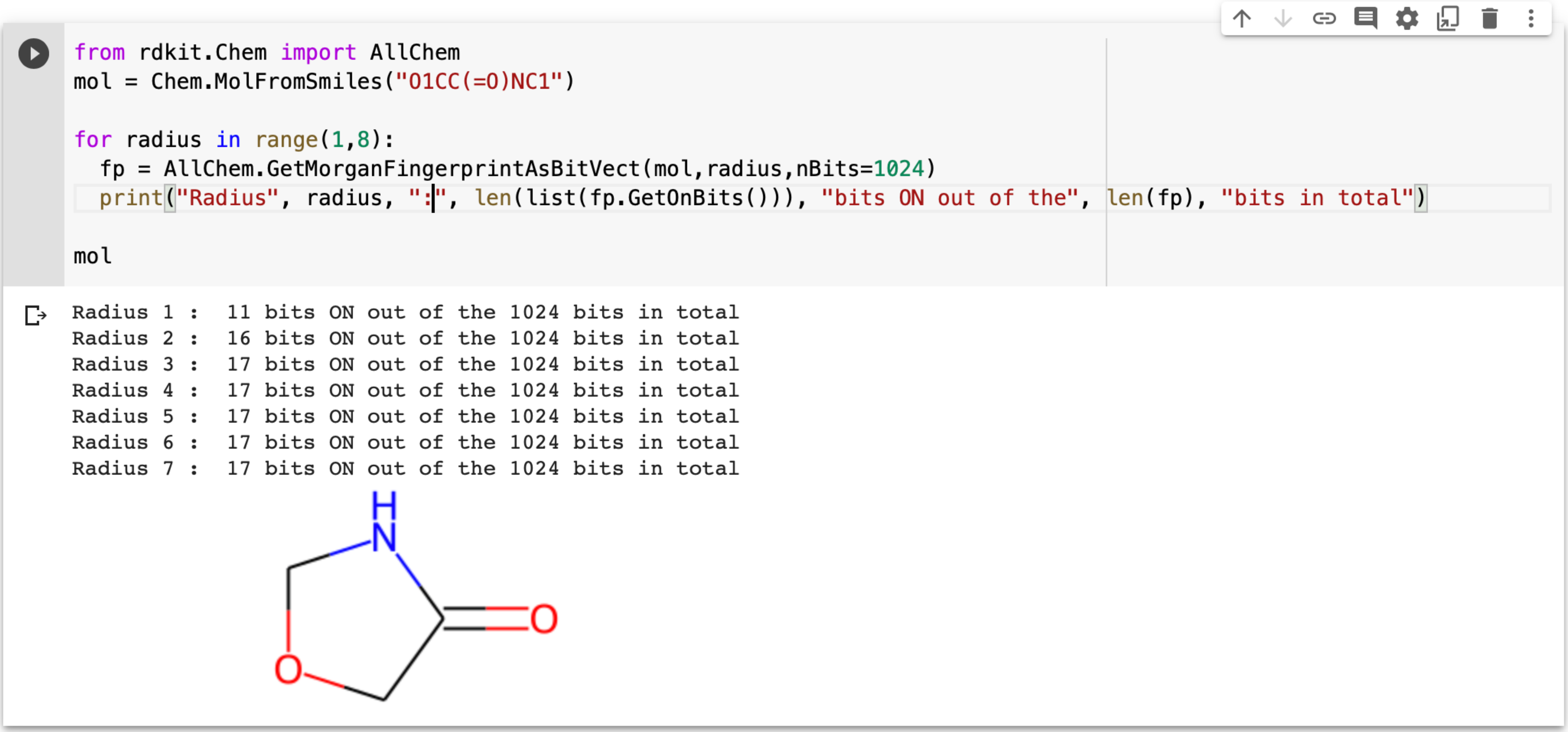
```
14 bits ON out of the 2048 bits in total
59 bits ON out of the 2048 bits in total
130 bits ON out of the 2048 bits in total
233 bits ON out of the 2048 bits in total
```



Circular path-based fingerprints (Morgan)

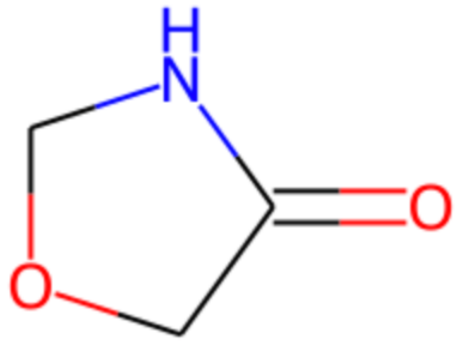


Specifying a diameter (radius)

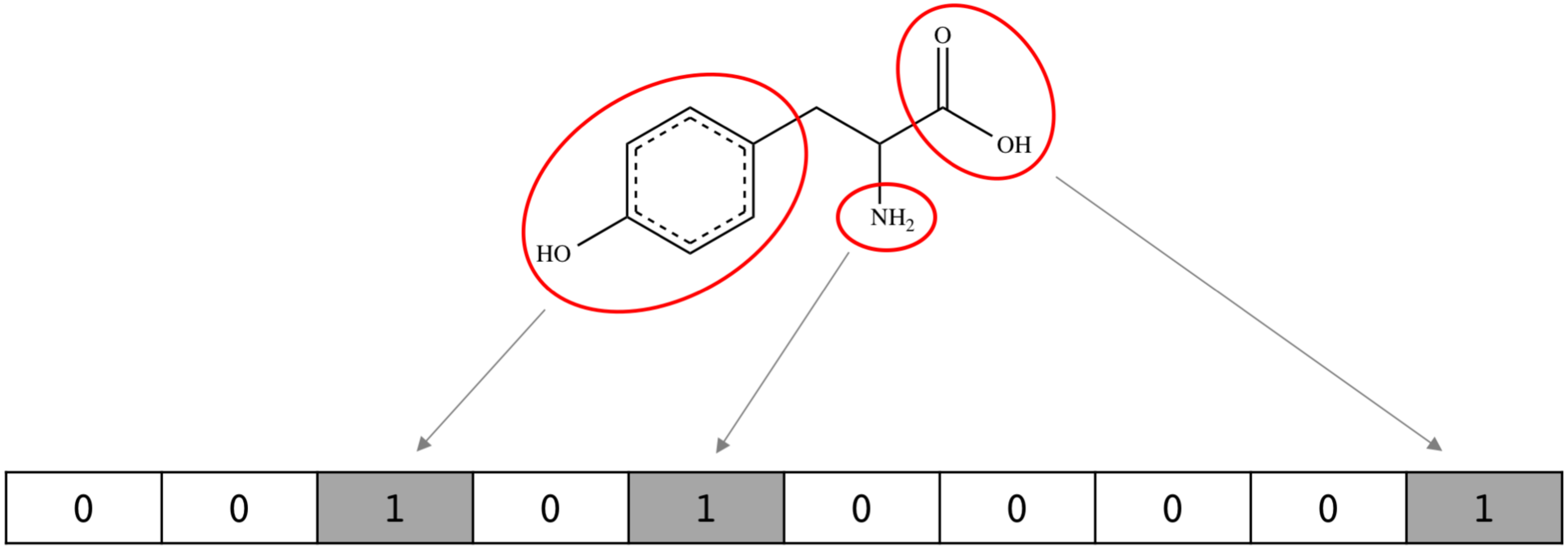


```
from rdkit.Chem import AllChem
mol = Chem.MolFromSmiles("O1CC(=O)NC1")

for radius in range(1,8):
  fp = AllChem.GetMorganFingerprintAsBitVect(mol,radius,nBits=1024)
  print("Radius", radius, ":", len(list(fp.GetOnBits())), "bits ON out of the", len(fp), "bits in total")

mol
```

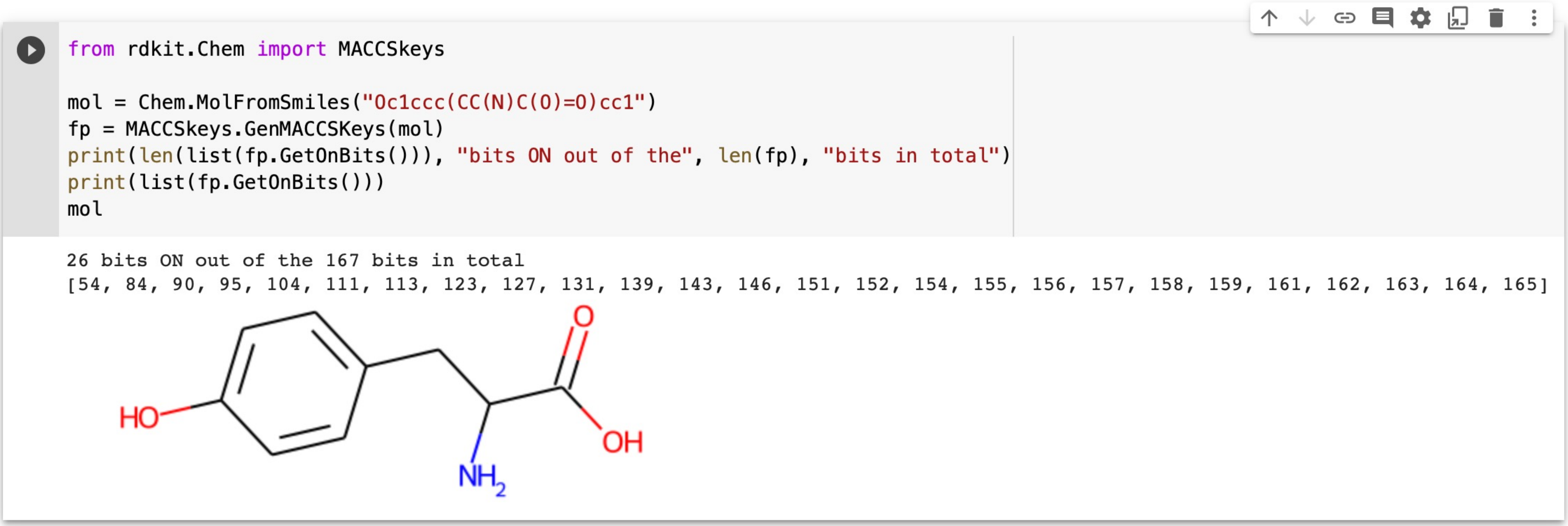
```
Radius 1 :  11 bits ON out of the 1024 bits in total
Radius 2 :  16 bits ON out of the 1024 bits in total
Radius 3 :  17 bits ON out of the 1024 bits in total
Radius 4 :  17 bits ON out of the 1024 bits in total
Radius 5 :  17 bits ON out of the 1024 bits in total
Radius 6 :  17 bits ON out of the 1024 bits in total
Radius 7 :  17 bits ON out of the 1024 bits in total
```



Substructure-based FP's: MACCS

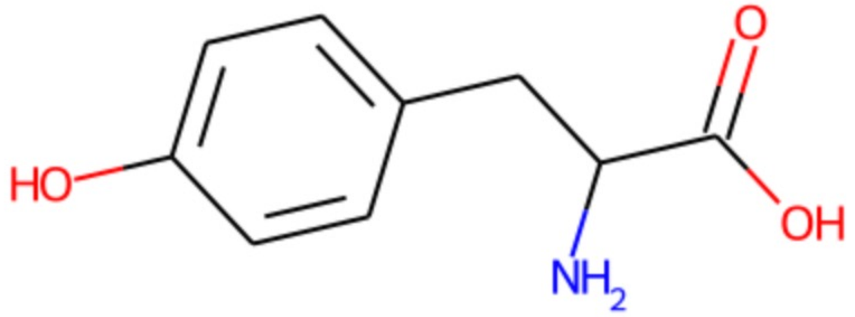


Only 167 bits (= 167 substructures)



```python
from rdkit.Chem import MACCSkeys

mol = Chem.MolFromSmiles("Oc1ccc(CC(N)C(O)=O)cc1")
fp = MACCSkeys.GenMACCSKeys(mol)
print(len(list(fp.GetOnBits())), "bits ON out of the", len(fp), "bits in total")
print(list(fp.GetOnBits()))
mol
```

```
26 bits ON out of the 167 bits in total
[54, 84, 90, 95, 104, 111, 113, 123, 127, 131, 139, 143, 146, 151, 152, 154, 155, 156, 157, 158, 159, 161, 162, 163, 164, 165]
```

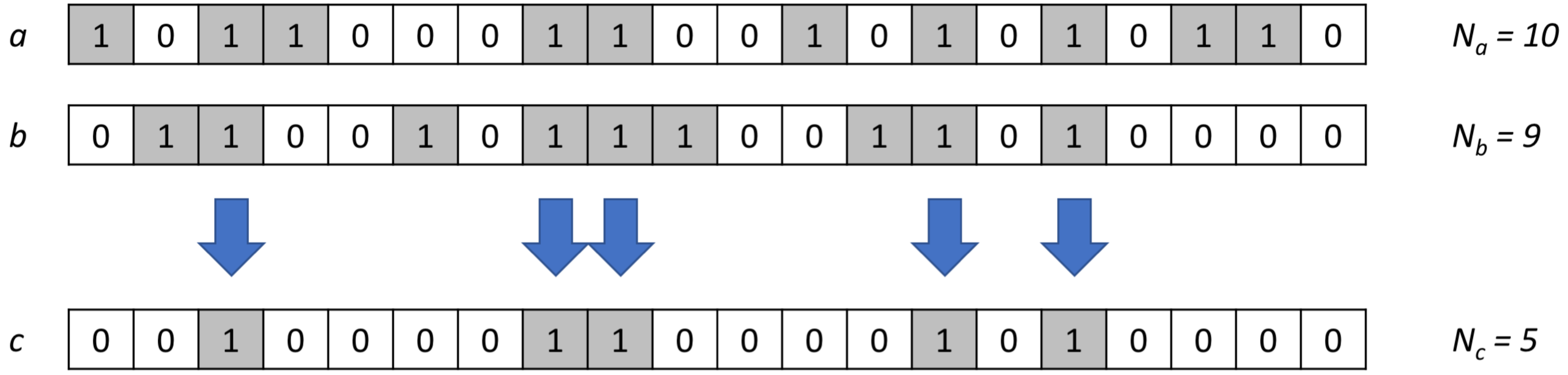


Molecular similarity

- Molecular fingerprints
  - Linear path-based
  - Circular path-based
  - Substructure-based

- Calculating similarity
  - Tanimoto
  - Tversky

Tanimoto index



$$T(a,b) = \frac{N_c}{N_a + N_b - N_c}$$

Tanimoto index

A	1	0	1	1	1	0	0	0
B	1	1	0	1	1	0	1	0

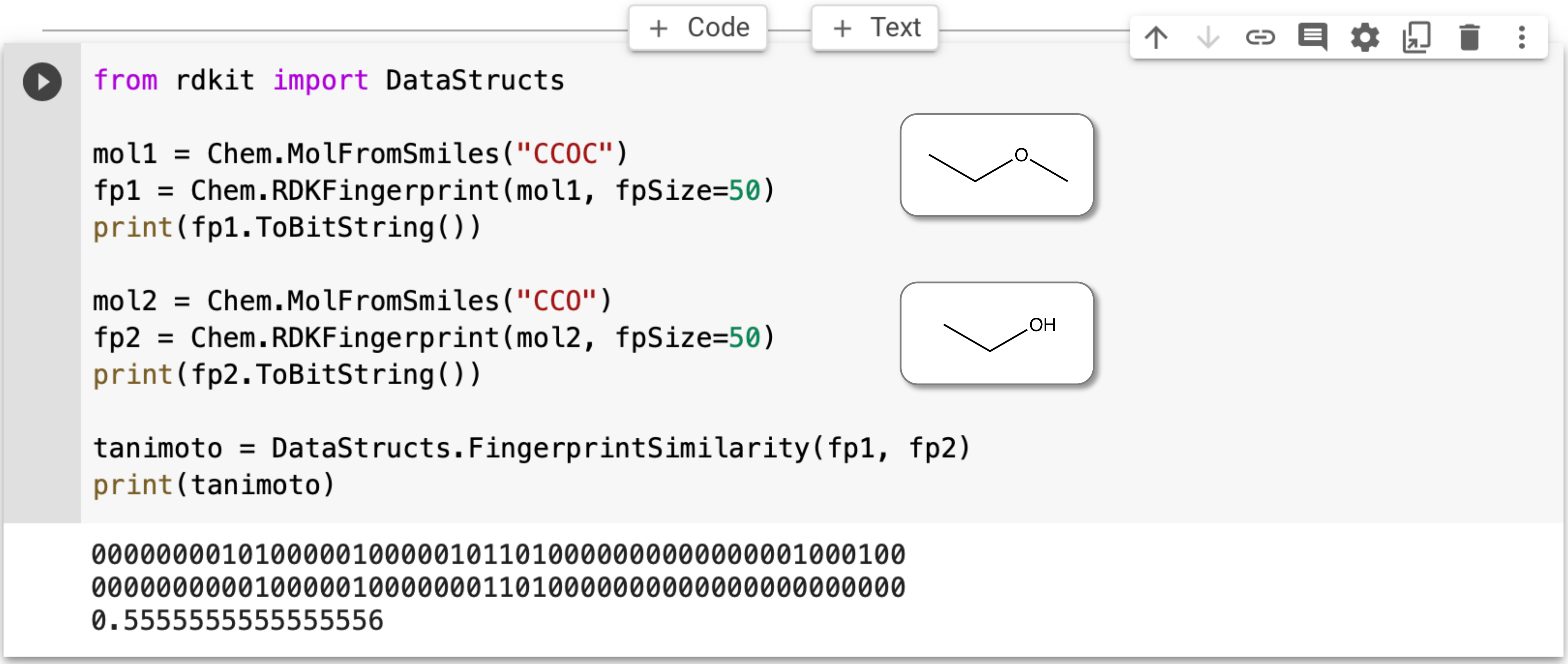
N_A = 4	*onlyA* = 1
N_B = 5	*onlyB* = 2

N_C = 3

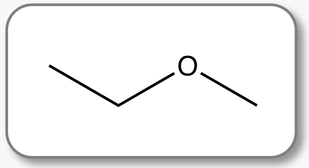
bothAB = 3

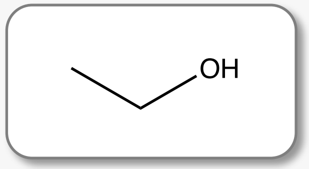
$$T(a,b) = \frac{N_C}{N_A + N_B - N_C} = \frac{bothAB}{onlyA + onlyB + bothAB}$$

1 = identical
0 = totally different



```python
from rdkit import DataStructs

mol1 = Chem.MolFromSmiles("CCOC")
fp1 = Chem.RDKFingerprint(mol1, fpSize=50)
print(fp1.ToBitString())

mol2 = Chem.MolFromSmiles("CCO")
fp2 = Chem.RDKFingerprint(mol2, fpSize=50)
print(fp2.ToBitString())

tanimoto = DataStructs.FingerprintSimilarity(fp1, fp2)
print(tanimoto)
```





```
00000000101000001000001011010000000000000000001000100
00000000001000001000000011010000000000000000000000
0.5555555555555556
```

Tversky index

A	1	0	1	1	1	0	0	0
B	1	1	0	1	1	0	1	0

$N_A = 4$ $\quad$ *onlyA* = 1

$N_B = 5$ $\quad$ *onlyB* = 2

$N_C = 3$

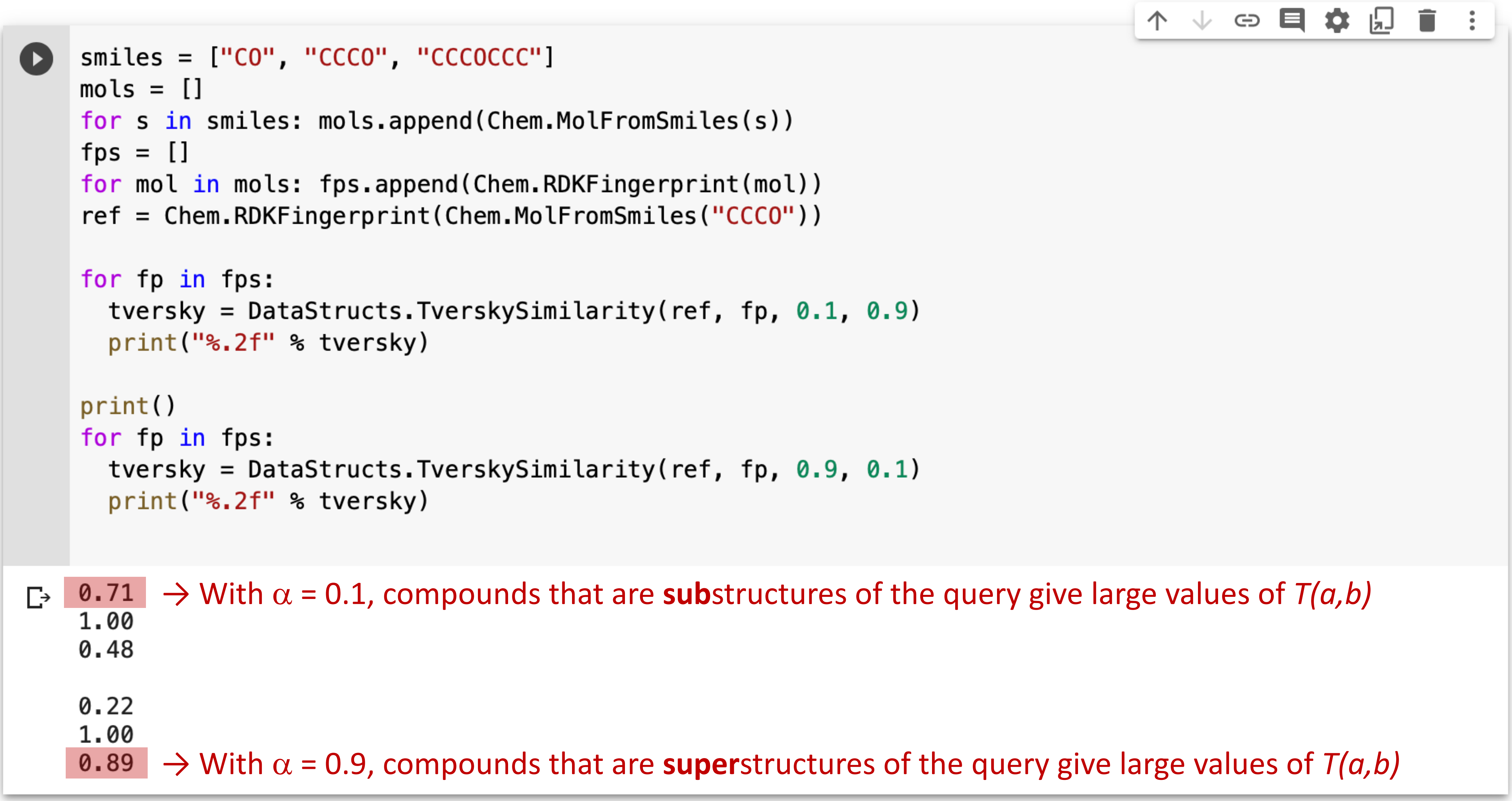
bothAB = 3

$$T(a,b) = \frac{bothAB}{\alpha * onlyA + \beta * onlyB + bothAB}$$

1 = identical
0 = totally different

The factor α weights the contribution of the first ‘reference’ molecule. The larger α becomes, the more weight is put on the bit setting of the reference molecule.

Tversky is asymmetric (α and β)



```
smiles = ["CO", "CCCO", "CCCOCCC"]
mols = []
for s in smiles: mols.append(Chem.MolFromSmiles(s))
fps = []
for mol in mols: fps.append(Chem.RDKFingerprint(mol))
ref = Chem.RDKFingerprint(Chem.MolFromSmiles("CCCO"))

for fp in fps:
  tversky = DataStructs.TverskySimilarity(ref, fp, 0.1, 0.9)
  print("%.2f" % tversky)

print()
for fp in fps:
  tversky = DataStructs.TverskySimilarity(ref, fp, 0.9, 0.1)
  print("%.2f" % tversky)
```

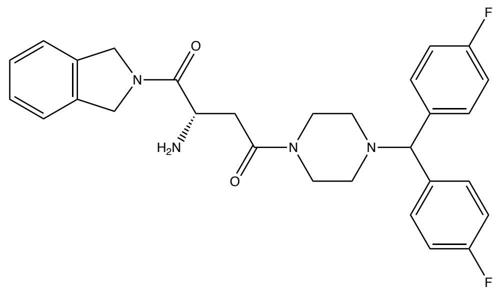
```
0.71
1.00
0.48

0.22
1.00
0.89
```

→ With α = 0.1, compounds that are **sub**structures of the query give large values of *T(a,b)*

→ With α = 0.9, compounds that are **super**structures of the query give large values of *T(a,b)*

Case study: similarity search

- In-house biological screen on DPP8 with 10,000 compounds revealed one hit:

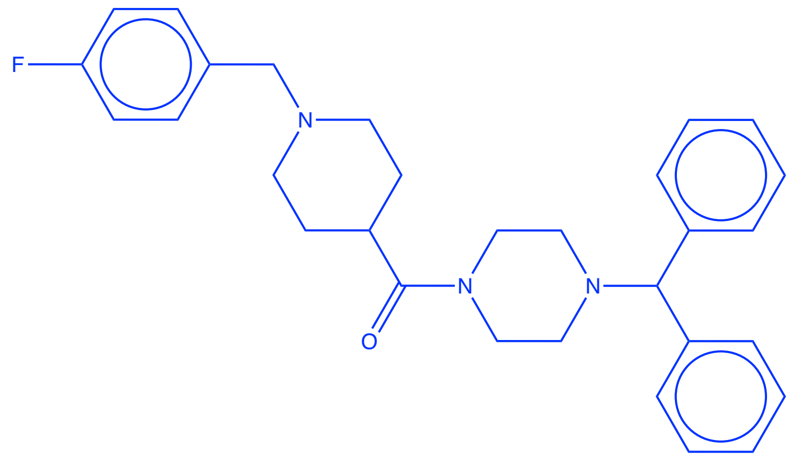


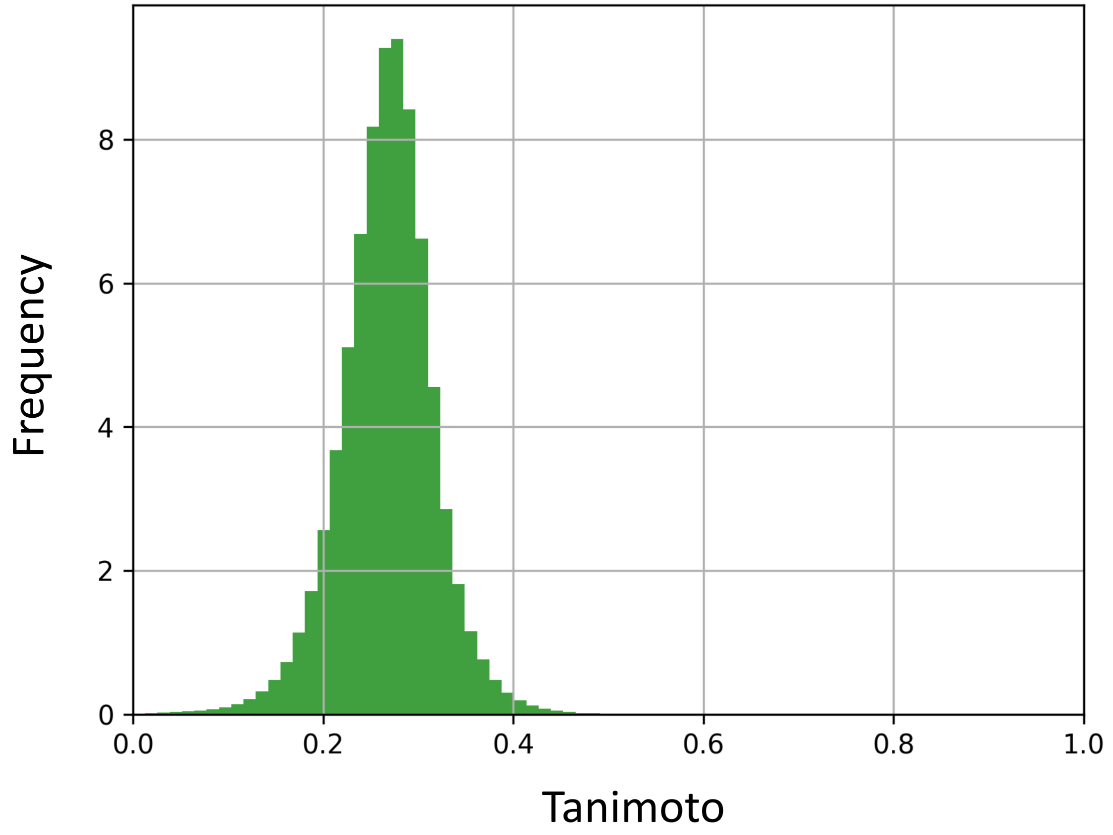
- Similarity search with this compound on a virtual database of compounds (>6M) revealed several compounds that could be purchased and tested *in vitro*

Case study: similarity search results

Top-10 of the most similar compounds:

```
O=C(C1CCN(Cc2ccc(F)cc2)CC1)N1CCN(C(c2ccccc2)c2ccccc2)CC1
O=C(O)CCC(=O)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
O=C(CCN1C(=O)CCC1=O)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
O=C(C1CC1)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
O=C([C@H]1CCCN1)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
CN1CCC(C(=O)N2CCN(C(c3ccc(F)cc3)c3ccc(F)cc3)CC2)C1
CN1CCNC(=O)C1CC(=O)N1CCN(C(c2ccccc2)c2ccccc2)CC1
CCC(=O)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
CC(=O)N1CCC(C(=O)N2CCN(C(c3ccc(F)cc3)c3ccc(F)cc3)CC2)CC1
O=C(CN1CCCC1=O)N1CCN(C(c2ccc(F)cc2)c2ccc(F)cc2)CC1
```



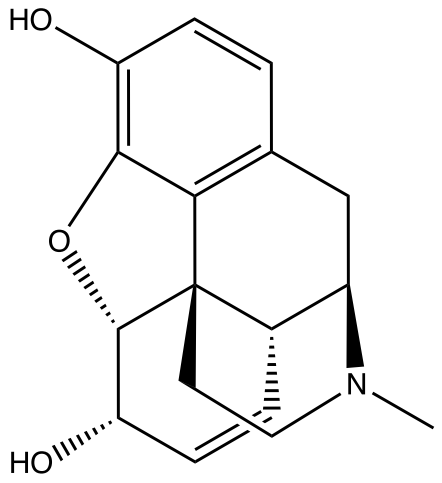


Clustering and machine learning

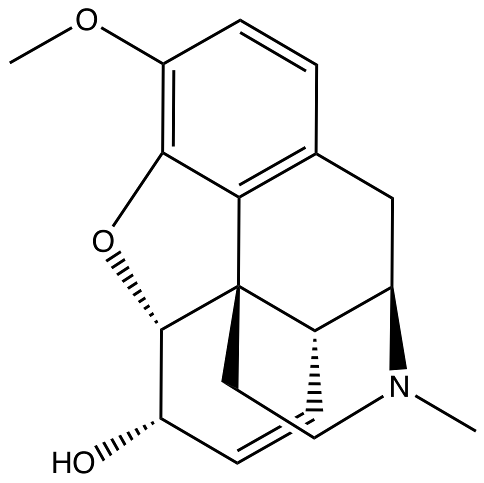
- Molecular similarity
- MCSS
- Clustering
- Machine learning: QSAR
- Validation

Maximum common substructure (MCSS)

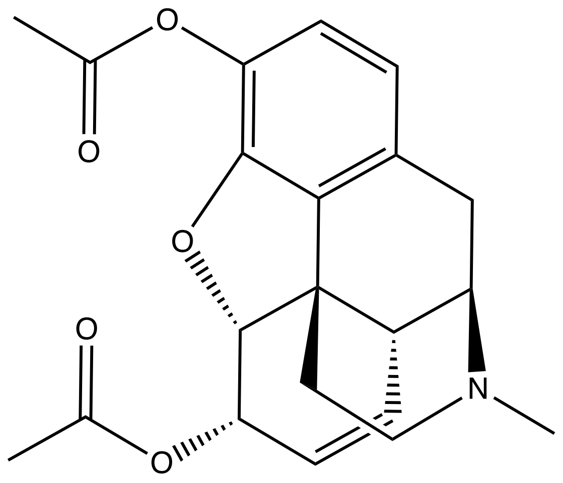
morphine



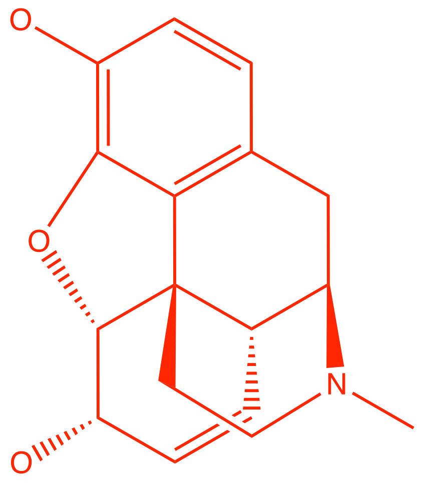
codeine



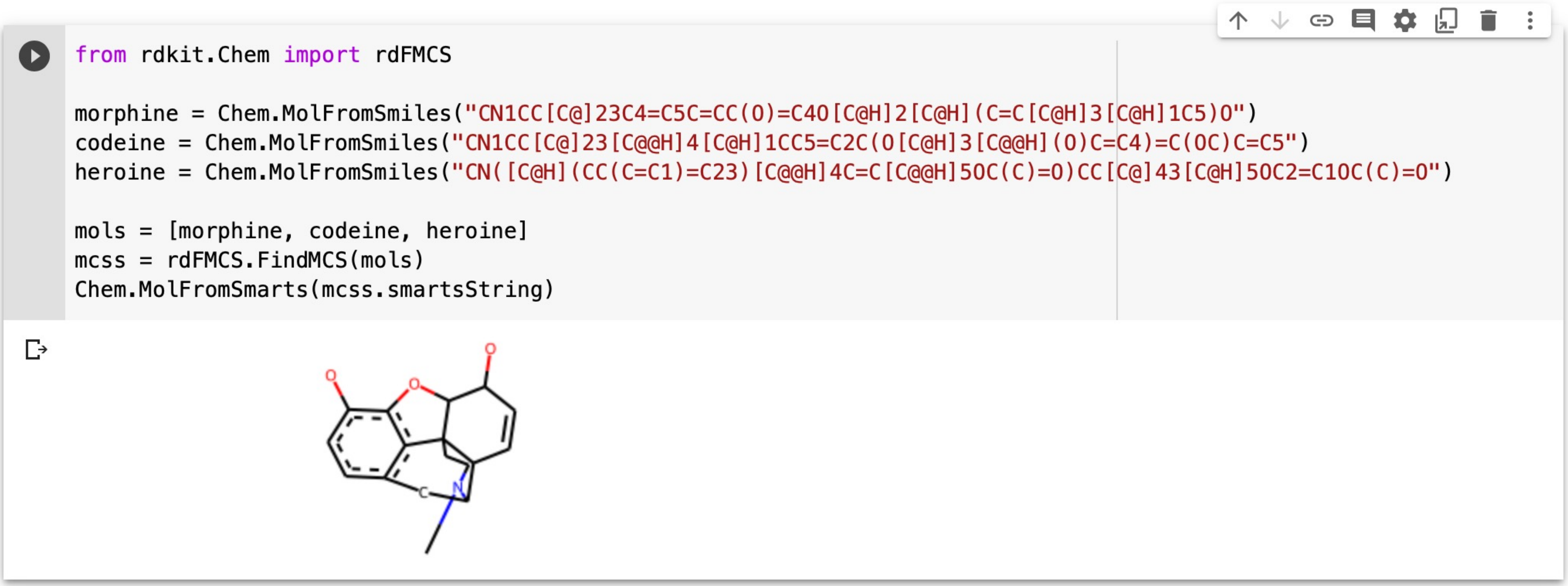
heroine



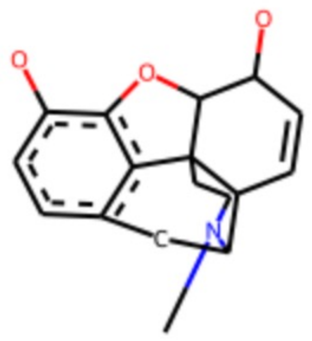
MCSS



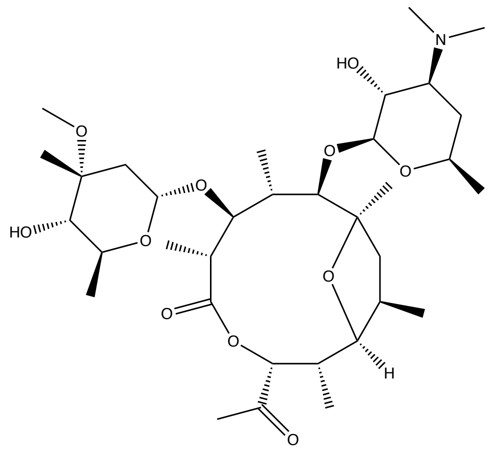
MCSS: RDKit code

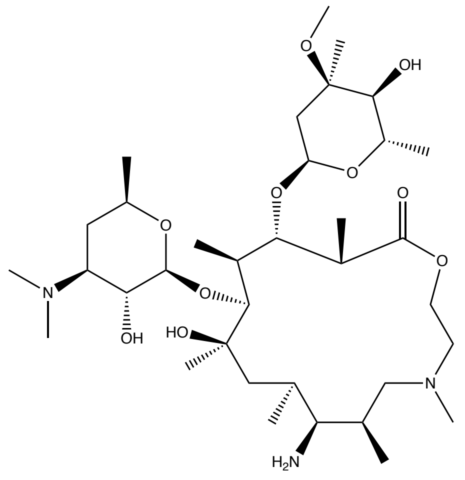


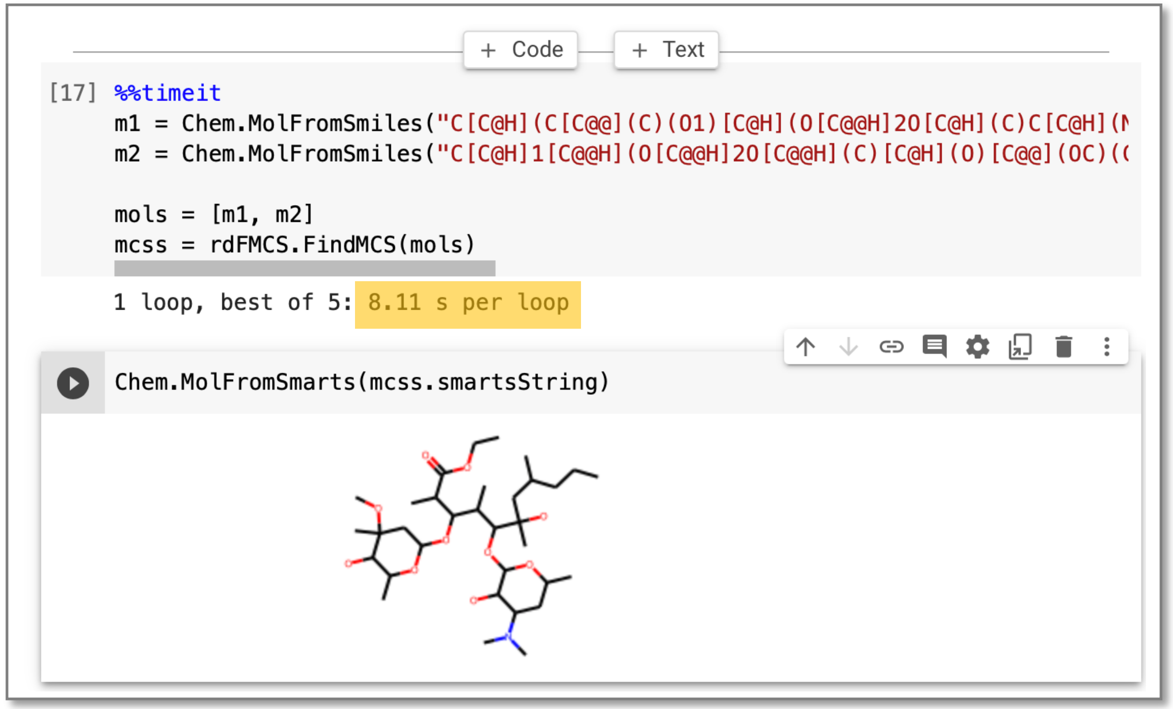
```python
from rdkit.Chem import rdFMCS

morphine = Chem.MolFromSmiles("CN1CC[C@]23C4=C5C=CC(O)=C4O[C@H]2[C@H](C=C[C@H]3[C@H]1C5)O")
codeine = Chem.MolFromSmiles("CN1CC[C@]23[C@@H]4[C@H]1CC5=C2C(O[C@H]3[C@@H](O)C=C4)=C(OC)C=C5")
heroine = Chem.MolFromSmiles("CN([C@H](CC(C=C1)=C23)[C@@H]4C=C[C@@H]5OC(C)=O)CC[C@]43[C@H]5OC2=C1OC(C)=O")

mols = [morphine, codeine, heroine]
mcss = rdFMCS.FindMCS(mols)
Chem.MolFromSmarts(mcss.smartsString)
```



Sometimes very long calculation times







```
[17] %%timeit
     m1 = Chem.MolFromSmiles("C[C@H](C[C@@](C)(O1)[C@H](O[C@@H]2O[C@H](C)C[C@H](
     m2 = Chem.MolFromSmiles("C[C@H]1[C@@H](O[C@@H]2O[C@@H](C)[C@H](O)[C@@](OC)(

     mols = [m1, m2]
     mcss = rdFMCS.FindMCS(mols)

     1 loop, best of 5: 8.11 s per loop
```

```
Chem.MolFromSmarts(mcss.smartsString)
```

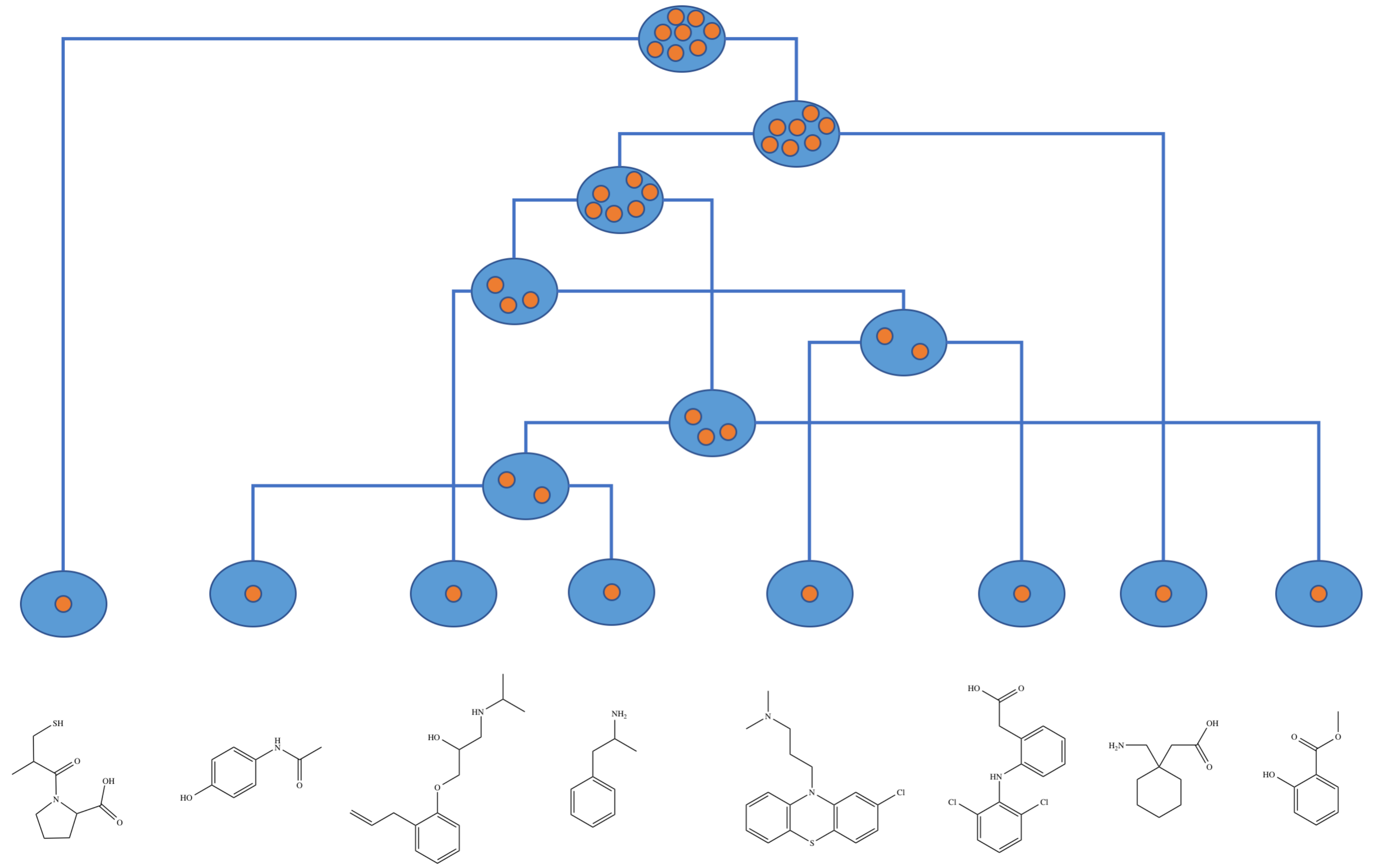
Clustering and machine learning

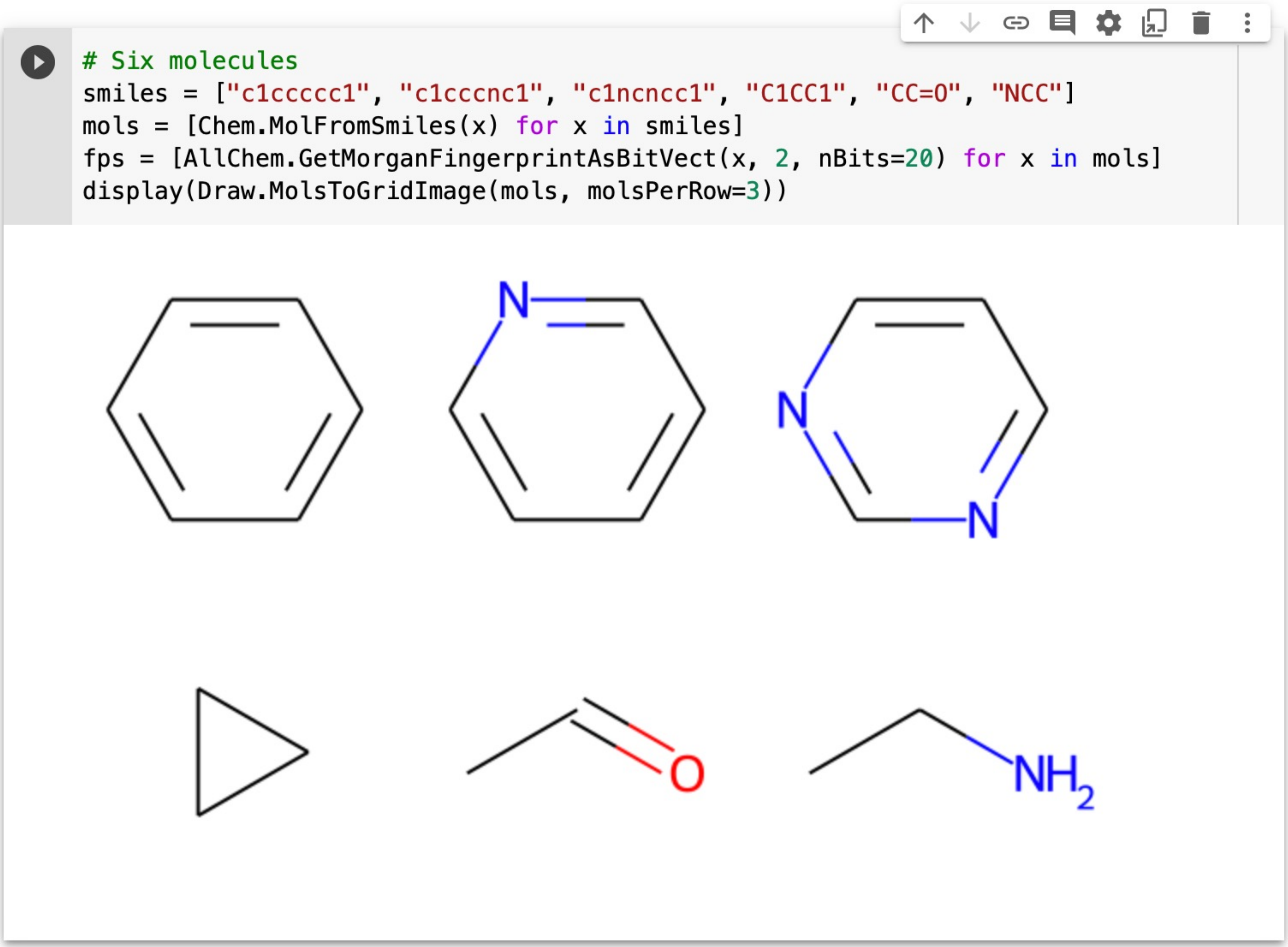
- Molecular similarity
- MCSS
- Clustering
- Machine learning: QSAR
- Validation

Clustering

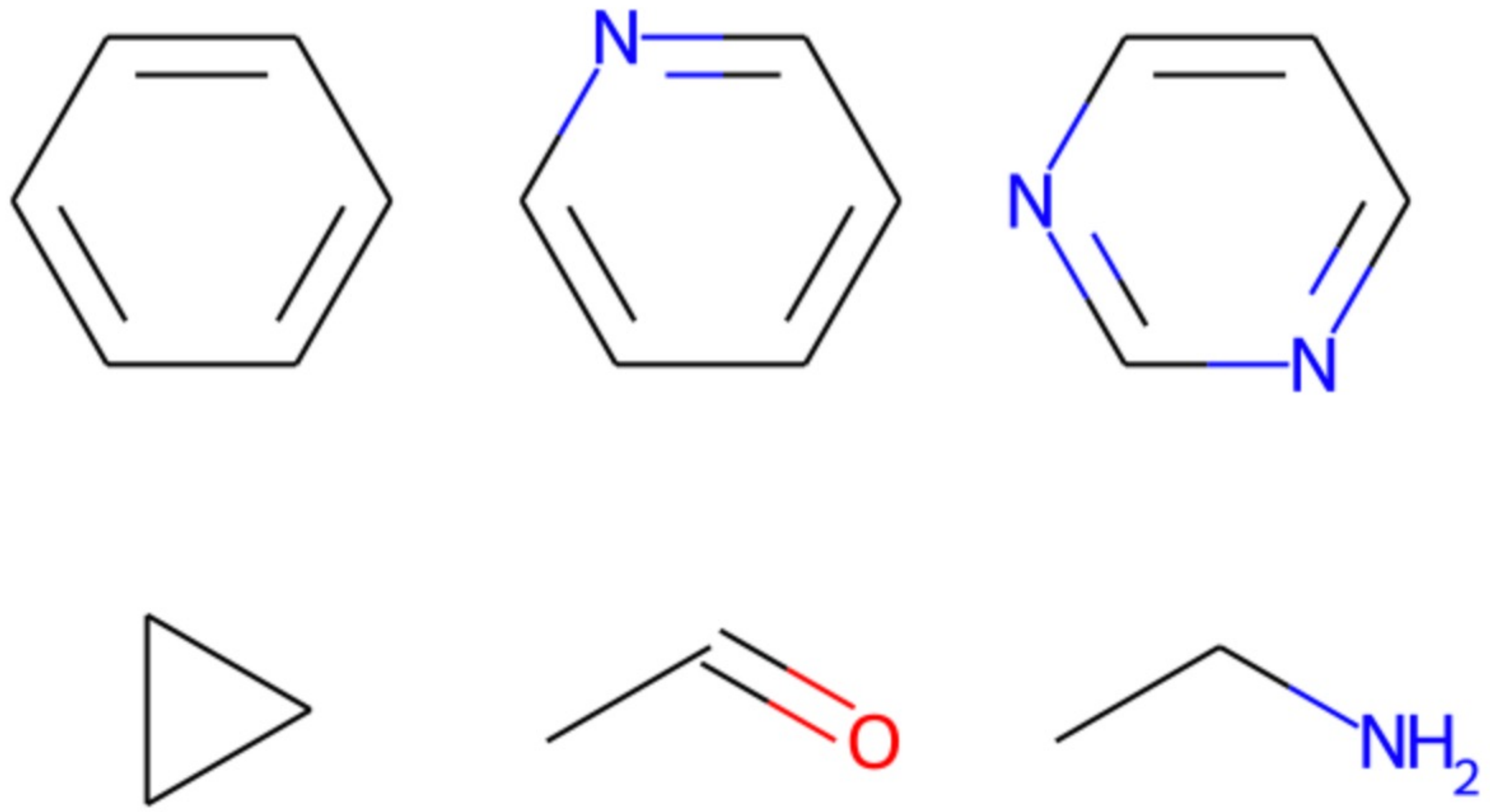
- Hierarchical clustering
- Non-hierarchical clustering

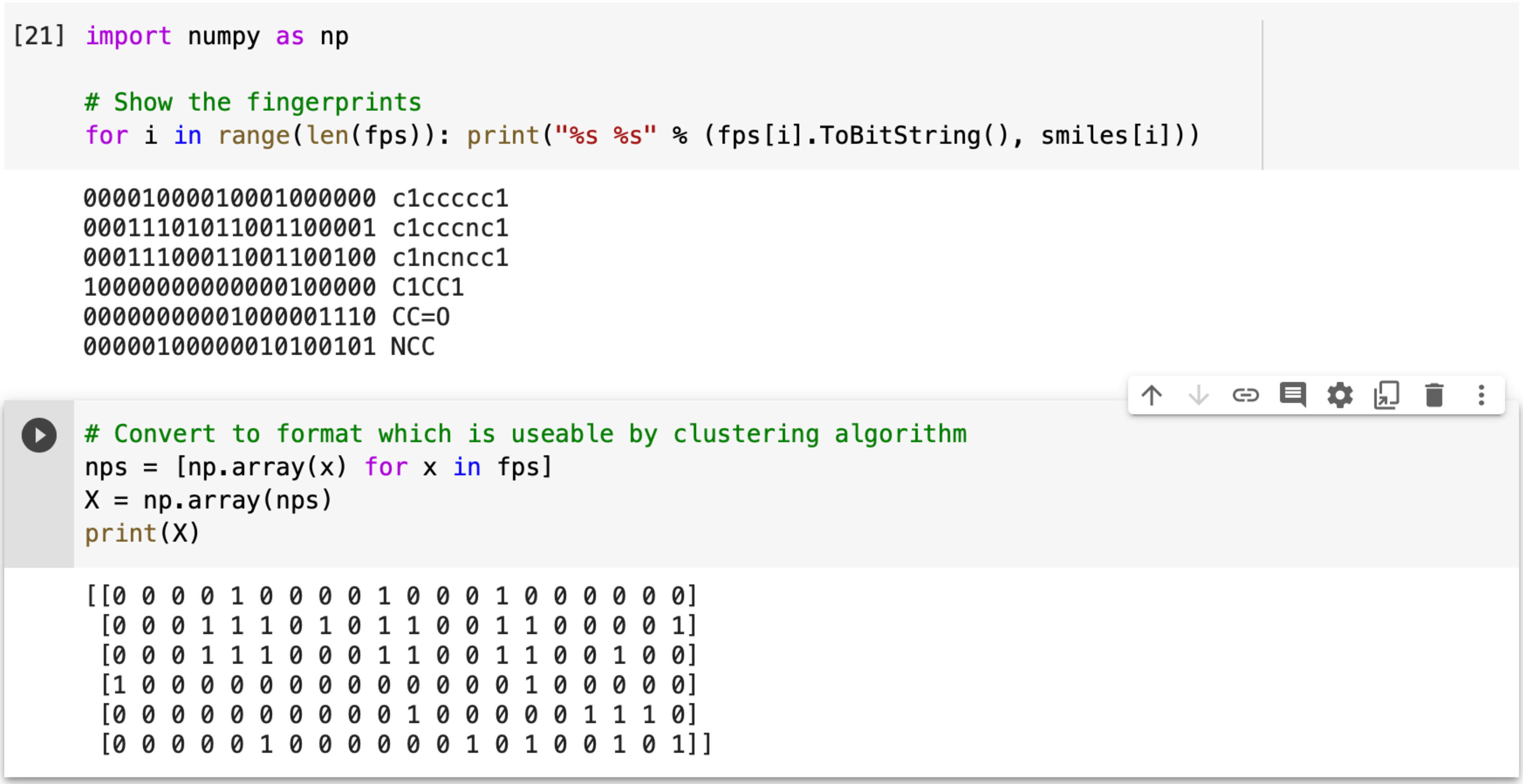
Hierarchical clustering





```python
# Six molecules
smiles = ["c1ccccc1", "c1cccnc1", "c1ncncc1", "C1CC1", "CC=O", "NCC"]
mols = [Chem.MolFromSmiles(x) for x in smiles]
fps = [AllChem.GetMorganFingerprintAsBitVect(x, 2, nBits=20) for x in mols]
display(Draw.MolsToGridImage(mols, molsPerRow=3))
```



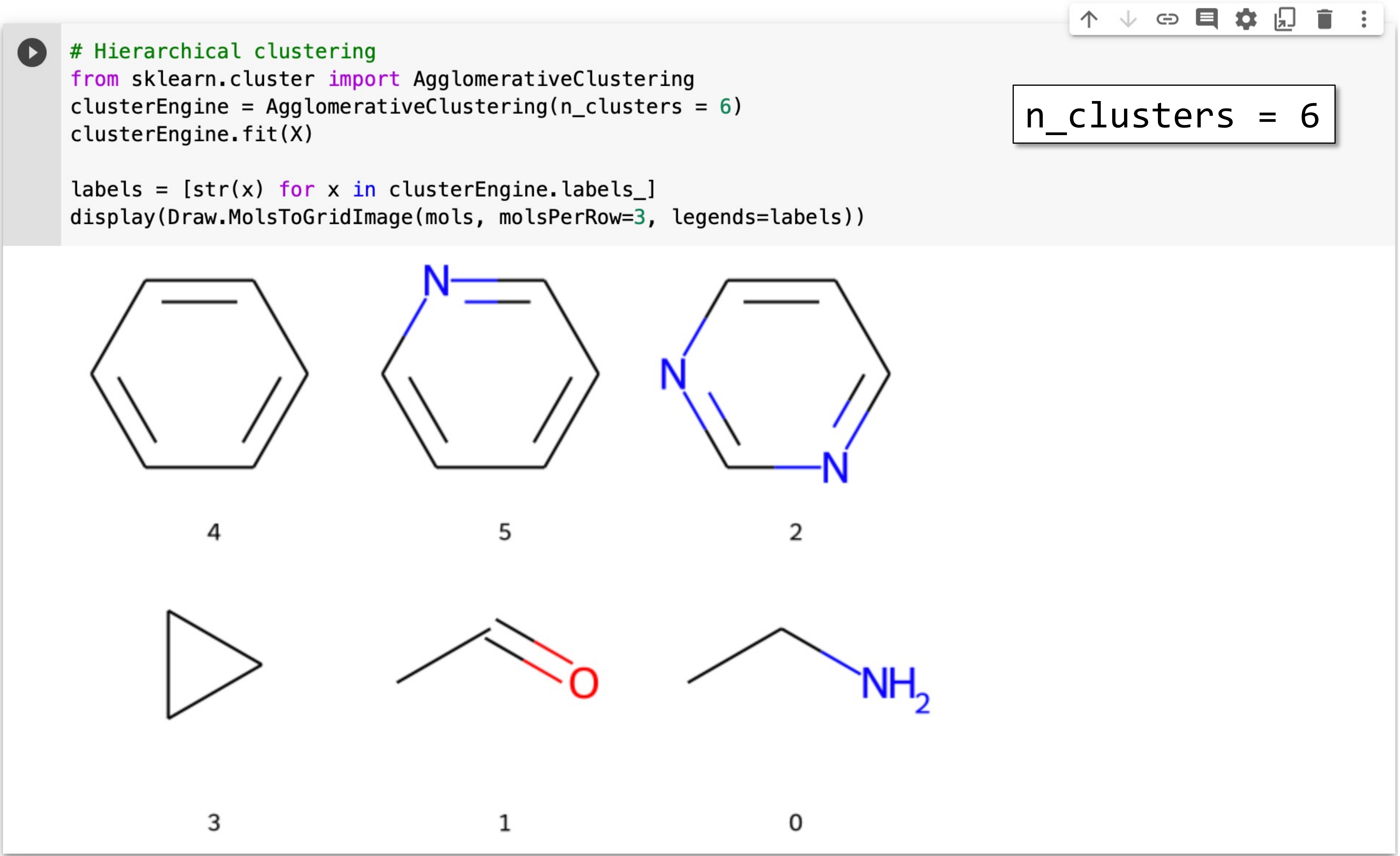


```
[21] import numpy as np

     # Show the fingerprints
     for i in range(len(fps)): print("%s %s" % (fps[i].ToBitString(), smiles[i]))
```

```
     00001000010001000000 c1ccccc1
     00011101011001100001 c1cccnc1
     00011100011001100100 c1ncncc1
     10000000000000100000 C1CC1
     00000000001000001110 CC=O
     00000100000010100101 NCC
```

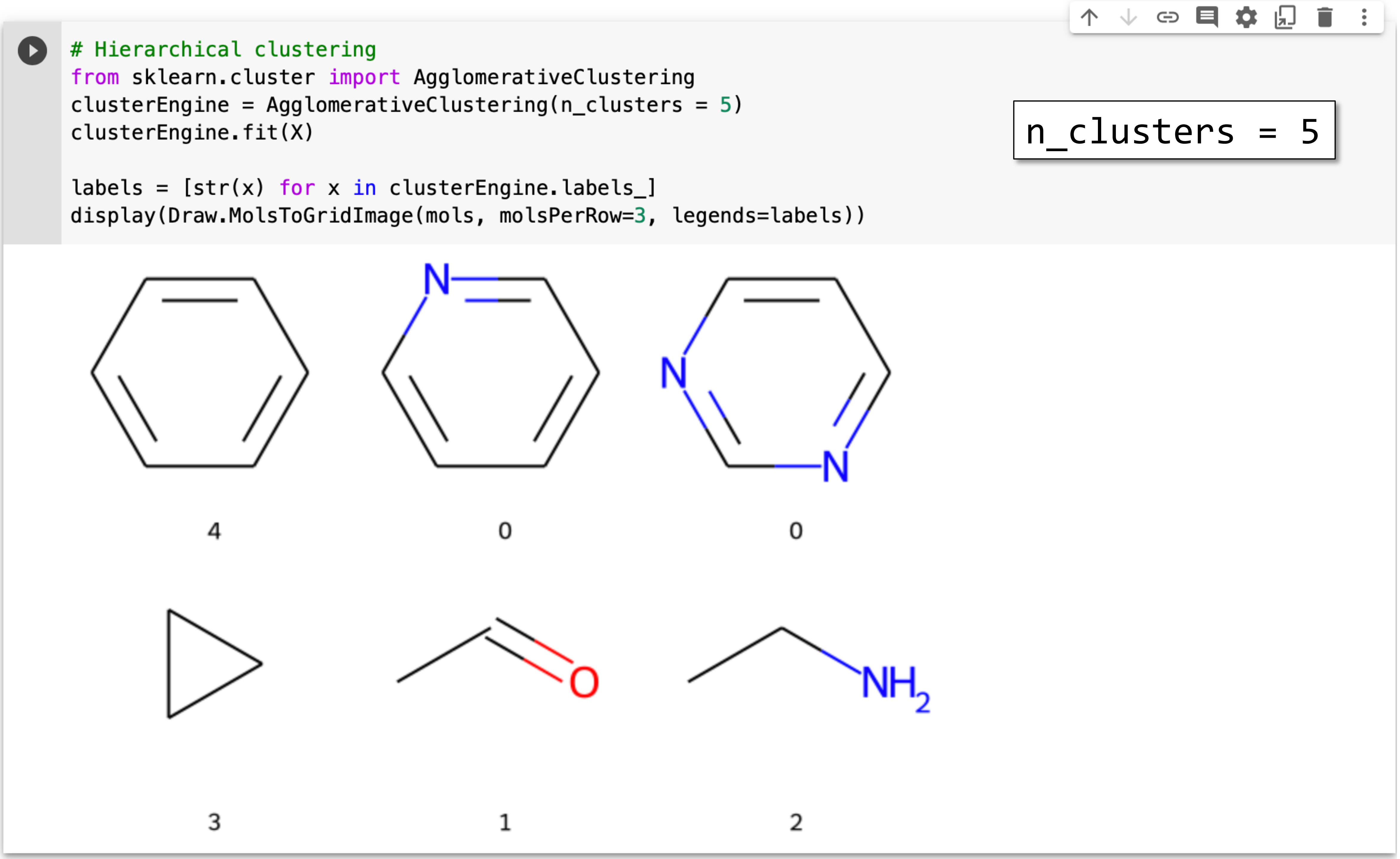
```
# Convert to format which is useable by clustering algorithm
nps = [np.array(x) for x in fps]
X = np.array(nps)
print(X)
```

```
[[0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 0 0 0 0 1]
 [0 0 0 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1]]
```



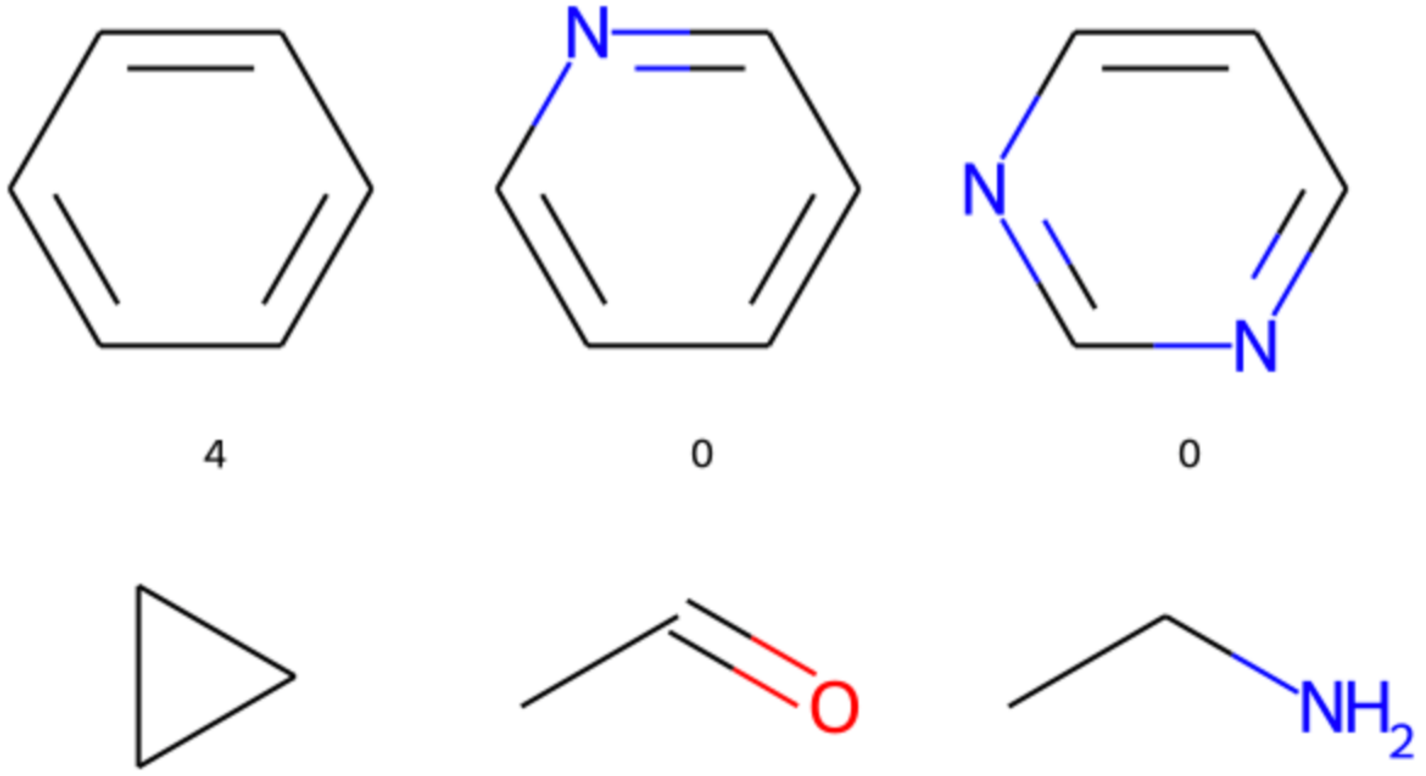
```
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 6)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

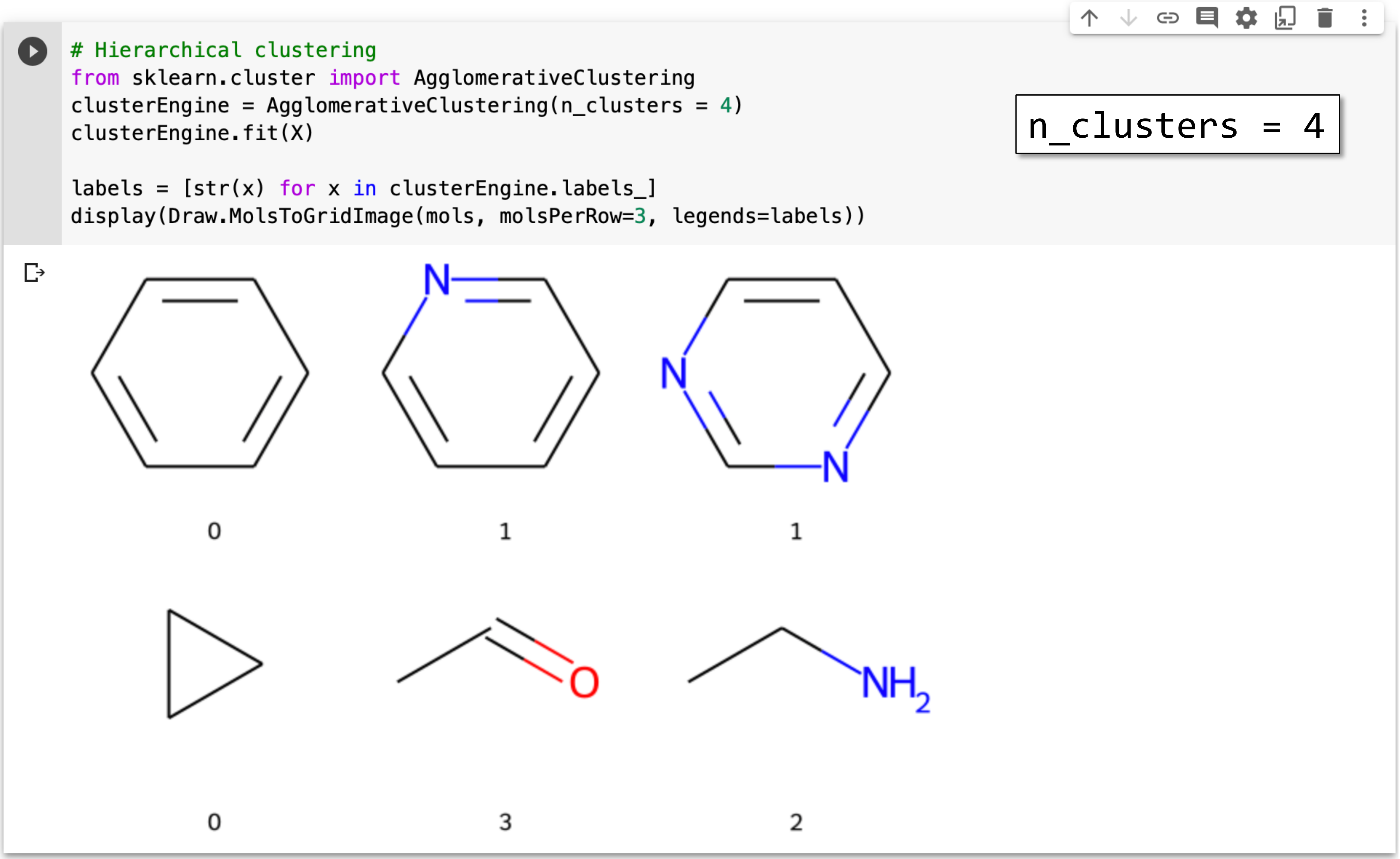
n_clusters = 6



```
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 5)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

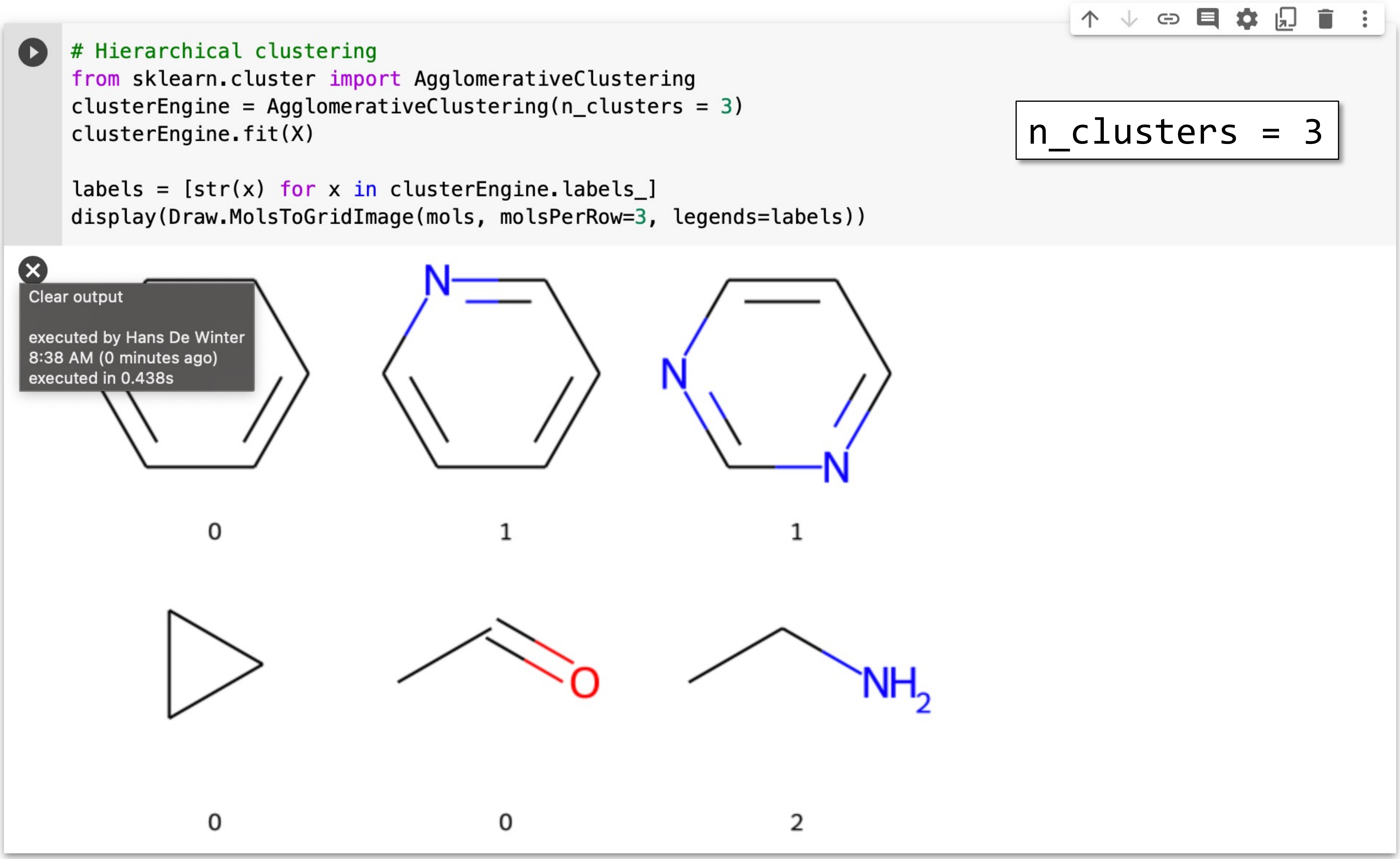
`n_clusters = 5`





```
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 4)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

n_clusters = 4

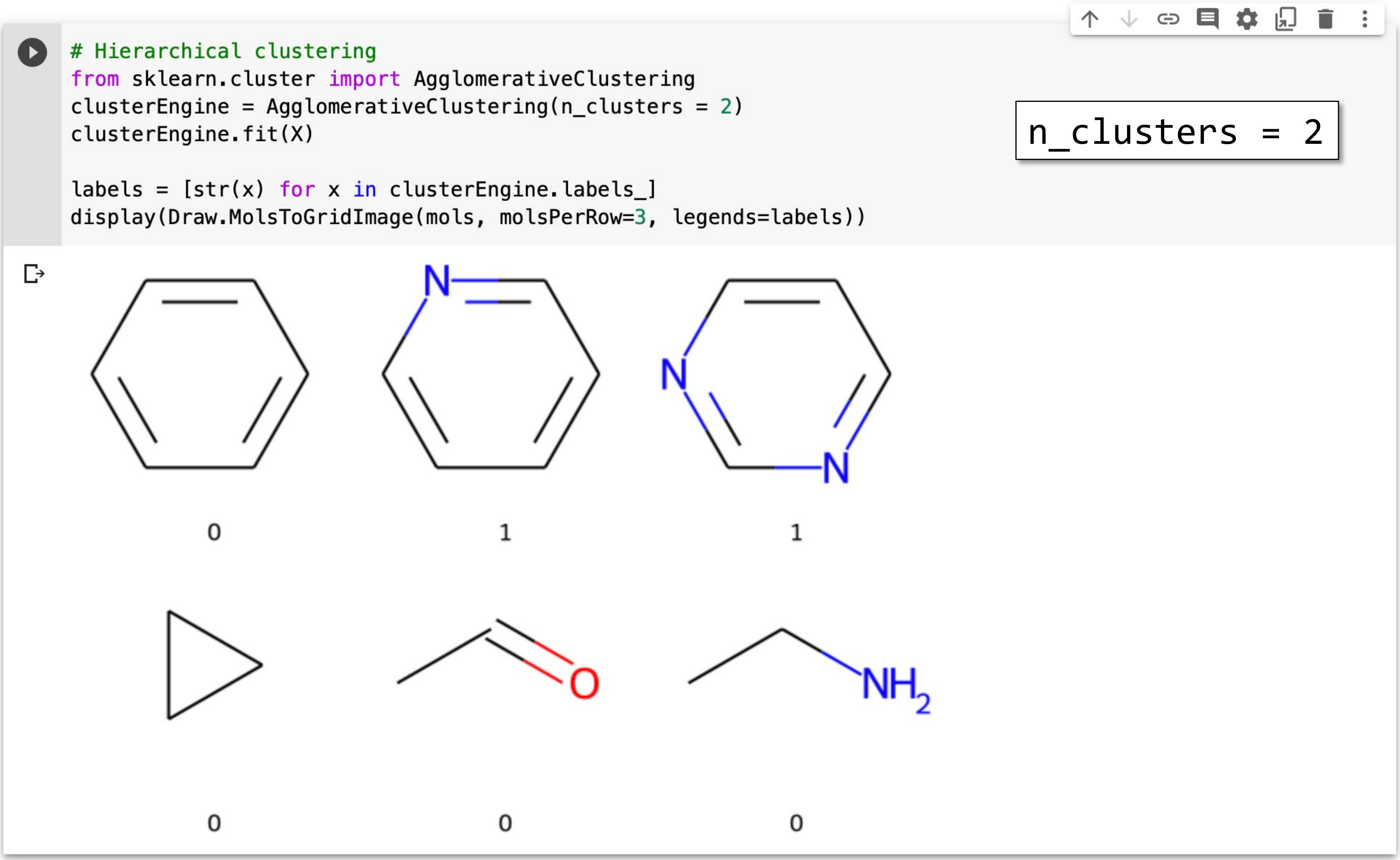


```python
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 3)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

n_clusters = 3

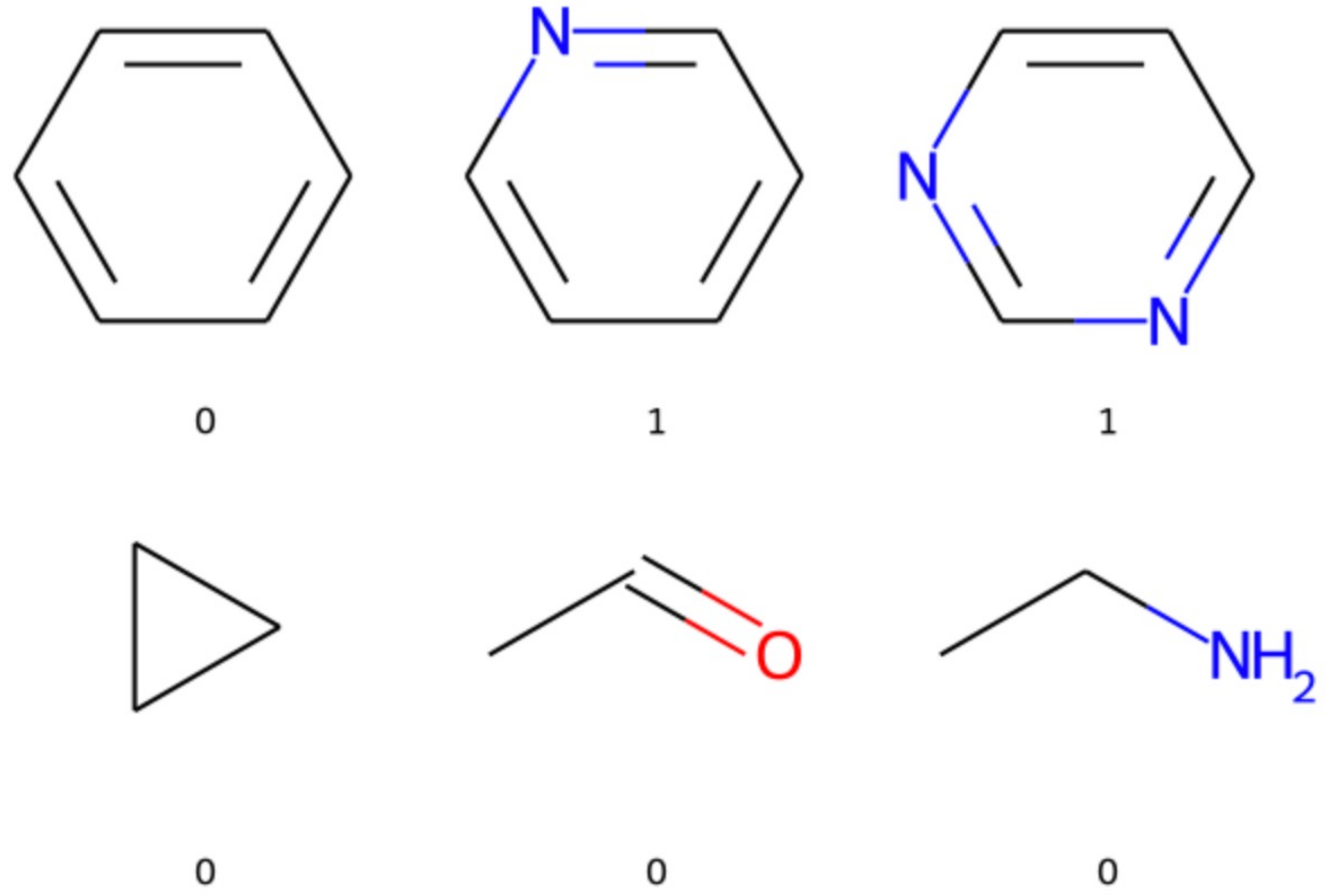
Clear output

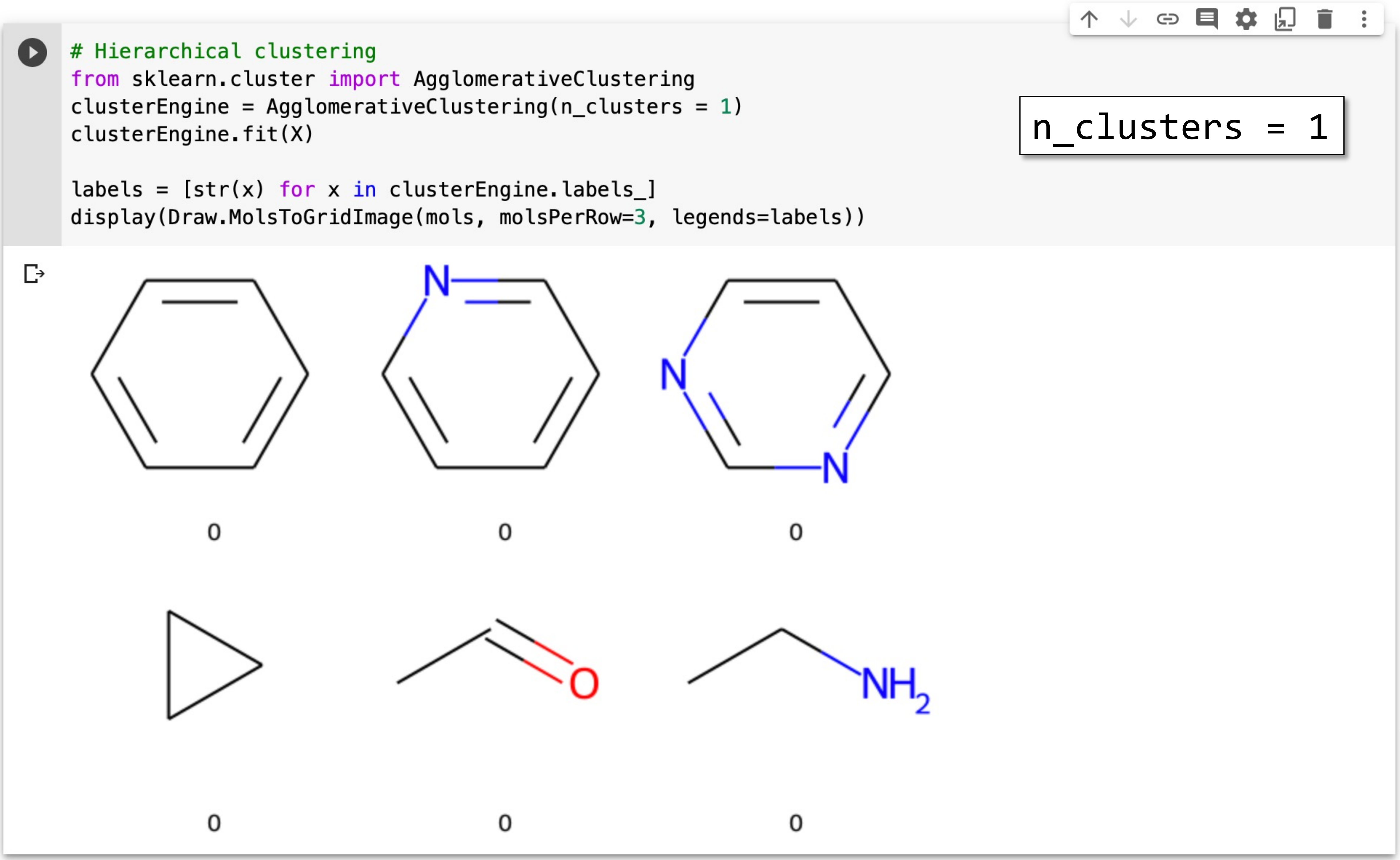
executed by Hans De Winter
8:38 AM (0 minutes ago)
executed in 0.438s



```
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 2)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

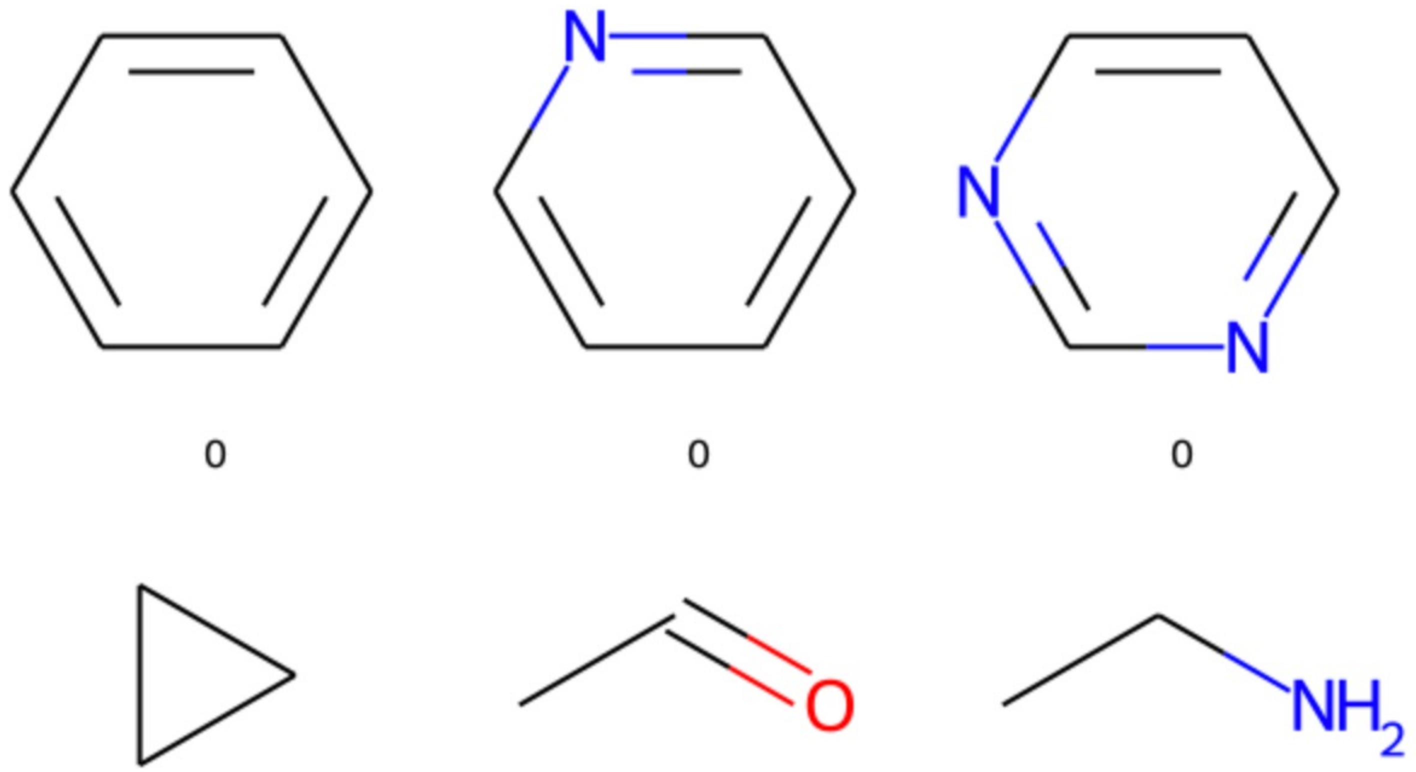
`n_clusters = 2`

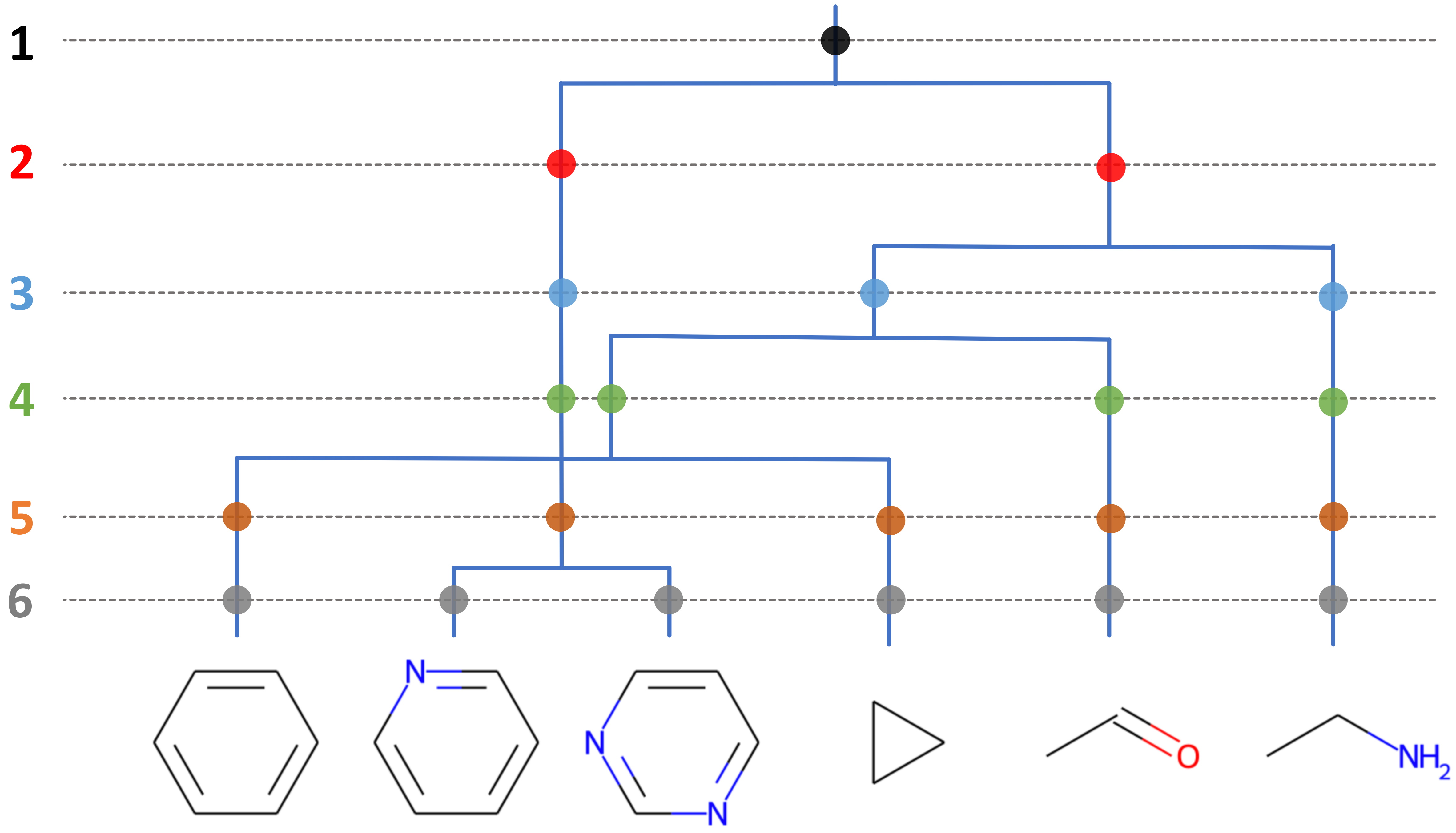




```
# Hierarchical clustering
from sklearn.cluster import AgglomerativeClustering
clusterEngine = AgglomerativeClustering(n_clusters = 1)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```

```
n_clusters = 1
```

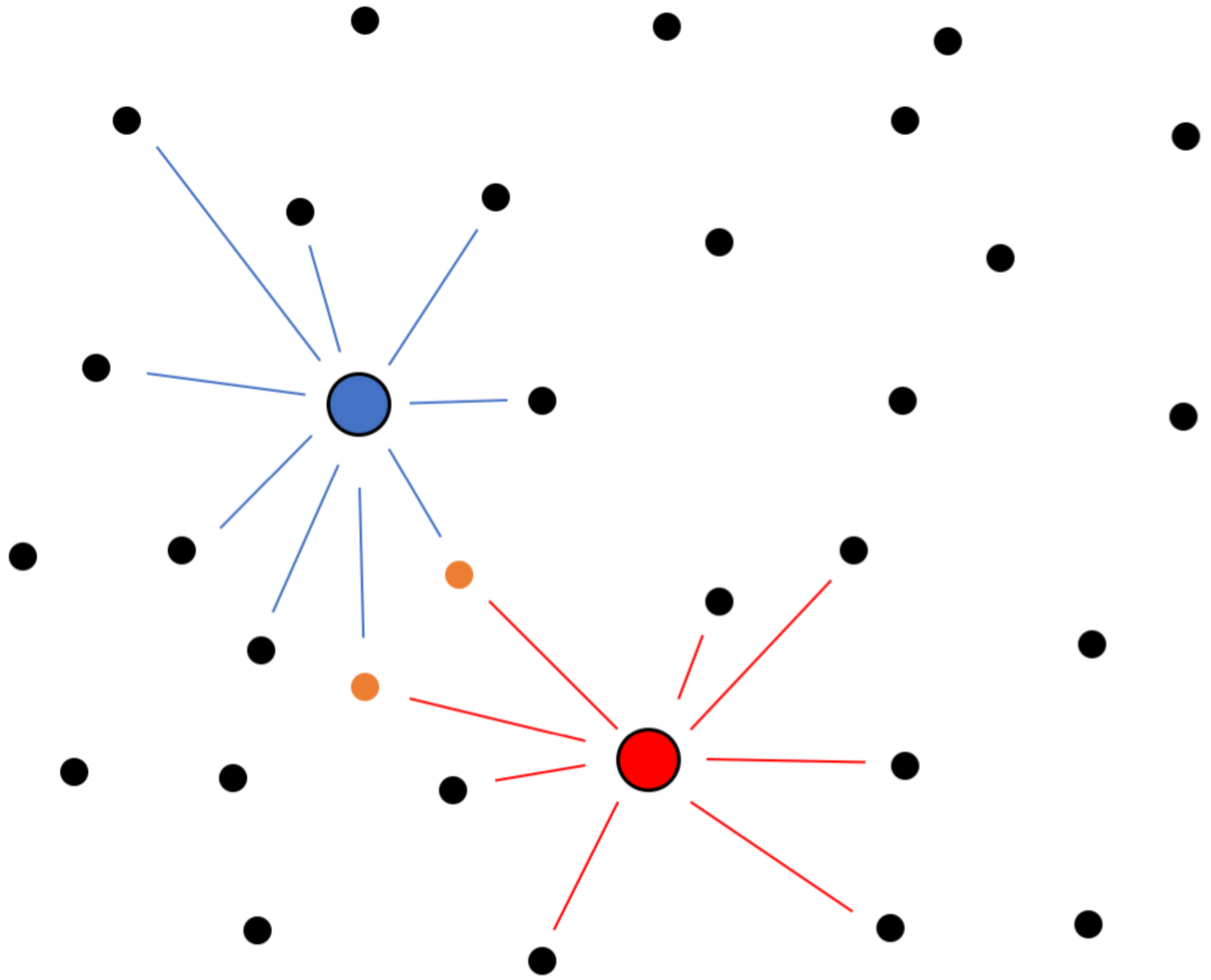


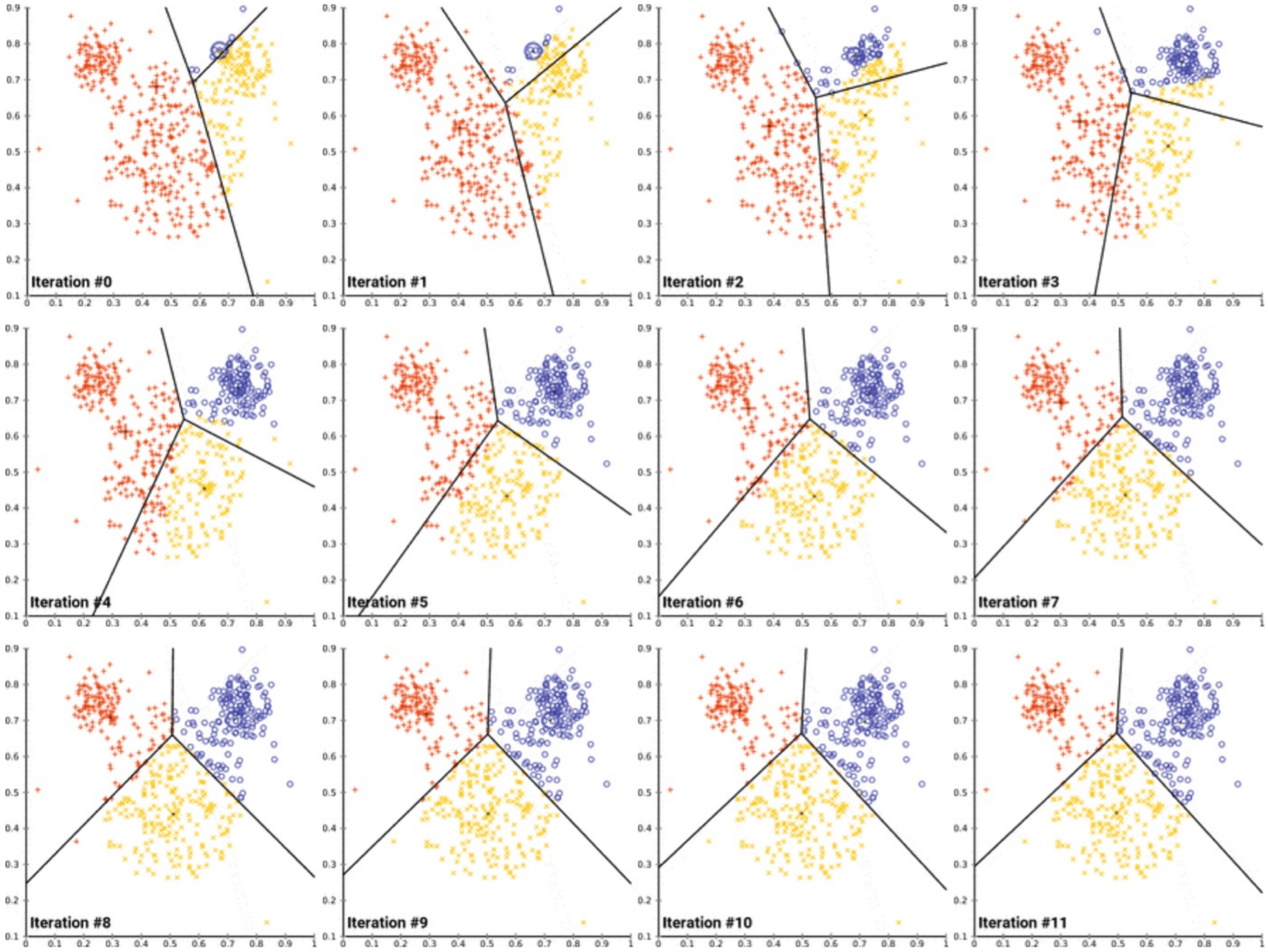

1
2
3
4
5
6
N
N
N
O
NH2

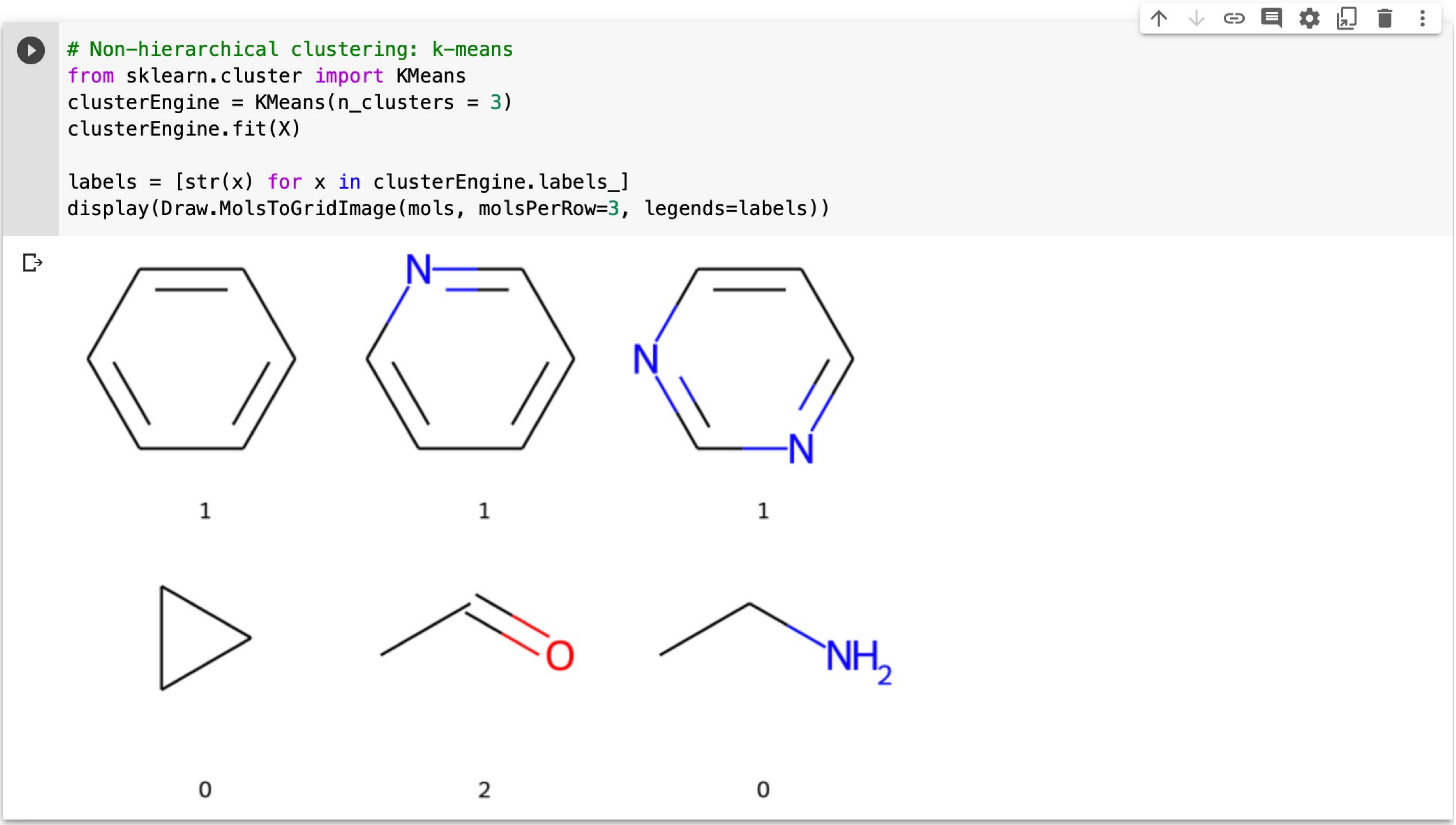
Clustering

- Hierarchical clustering
- Non-hierarchical clustering

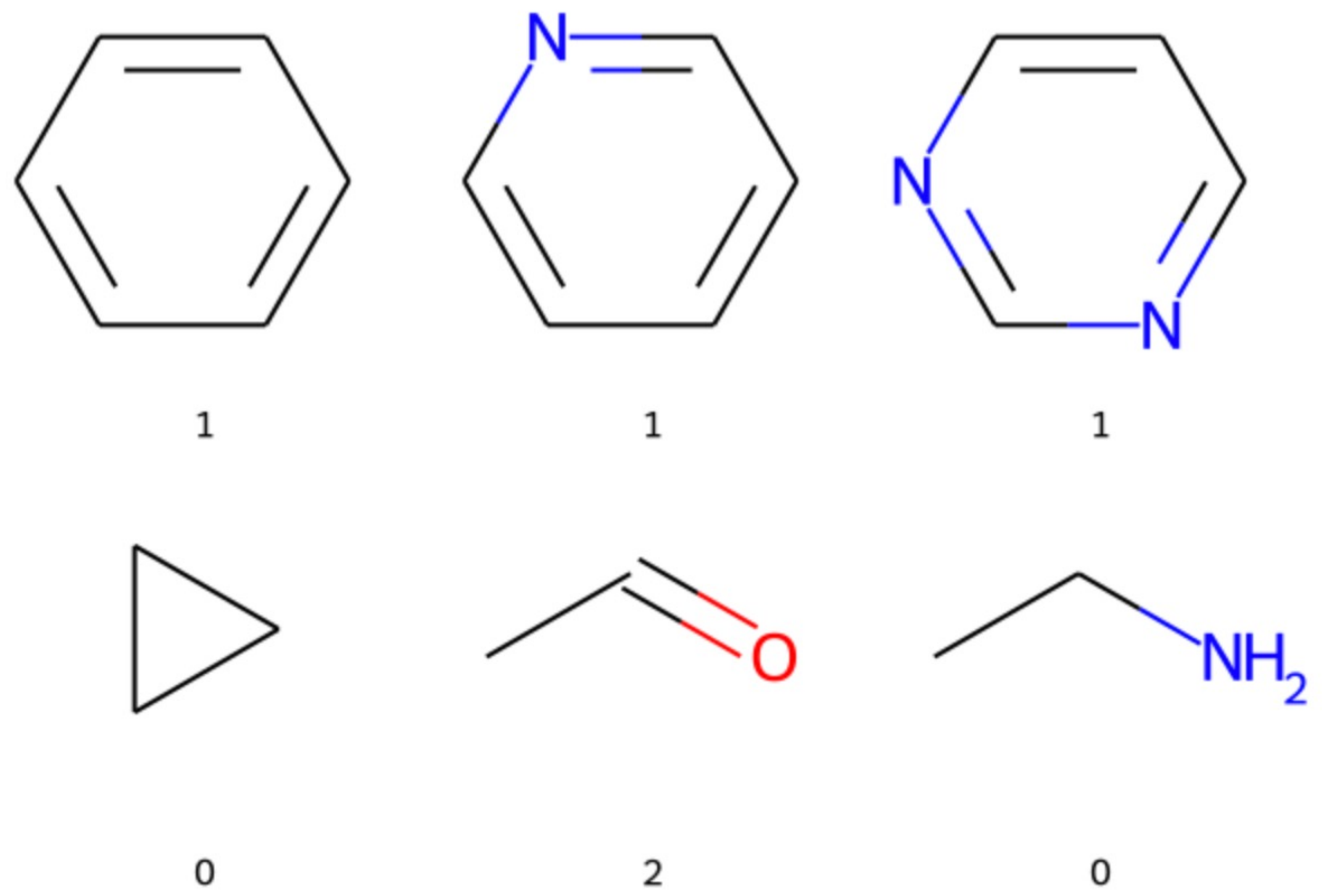
Non-hierarchical clustering




Iteration #0
Iteration #1
Iteration #2
Iteration #3
Iteration #4
Iteration #5
Iteration #6
Iteration #7
Iteration #8
Iteration #9
Iteration #10
Iteration #11



```python
# Non-hierarchical clustering: k-means
from sklearn.cluster import KMeans
clusterEngine = KMeans(n_clusters = 3)
clusterEngine.fit(X)

labels = [str(x) for x in clusterEngine.labels_]
display(Draw.MolsToGridImage(mols, molsPerRow=3, legends=labels))
```



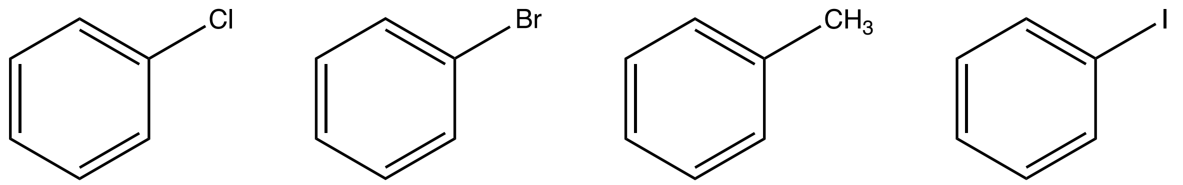
Clustering and machine learning

- Molecular similarity
- MCSS
- Clustering
- Machine learning: QSAR
- Validation

Quantitative Structure-Activity Relationship

- QSAR / QSPR
- Corwin Hansch (Ponoma College, California)
- Hansch equation:

$$Molecular\ property = f(atomic\ properties)$$



Data analytics

What has happened?

- Descriptive analytics
  - Mean
  - SD
  - Histograms
  - …

What will happen?

- Predictive analytics
  - This chapter

Two types of machine learning methods

- Supervised learning: each datapoint is labeled with a certain property
  - Classification
  - Regression
- Unsupervised learning: no labels
  - Clustering
  - Dimensionality reduction

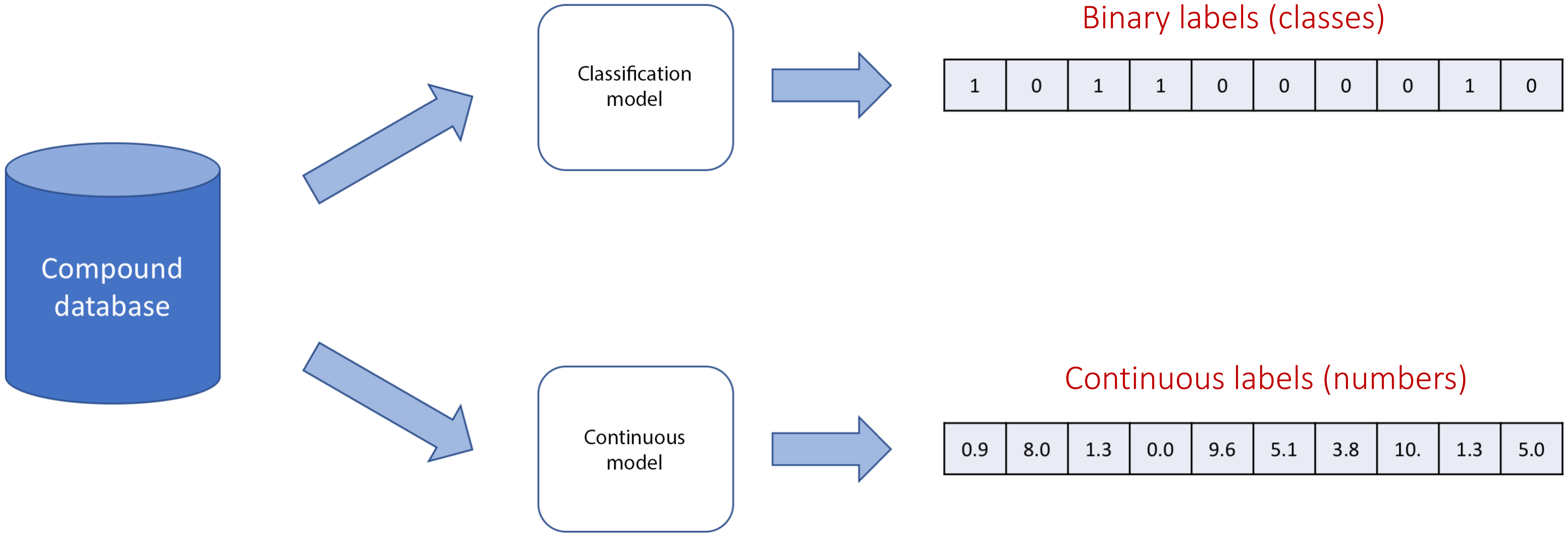
Supervised learning: labels

- **Classification**: predict a class
  - Active *versus* non-active
  - Soluble *versus* insoluble
  - QT-elongation *versus* safe
  - Belongs to class A/B/C/...

Binary labels (classes)

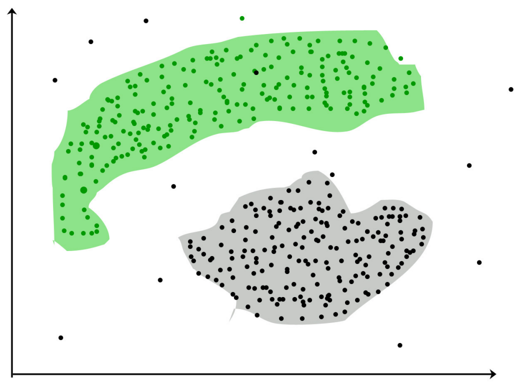
- **Regression**: predict a quantitative number
  - Probability of being active
  - Quantitative estimation of activity (e.g. IC_{50})
  - Predicted solubility in g/L
  - ...

Continuous labels (numbers)

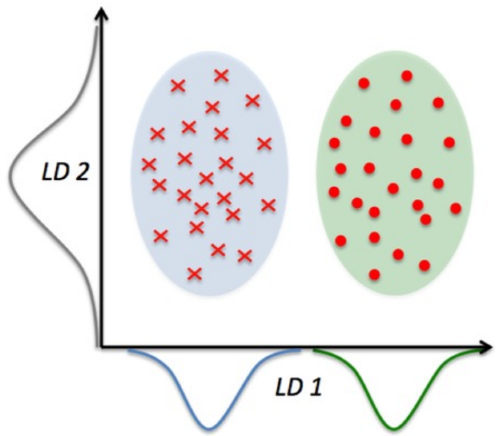

Compound database
Classification model
Continuous model
Binary labels (classes)
1 0 1 1 0 0 0 0 1 0
Continuous labels (numbers)
0.9 8.0 1.3 0.0 9.6 5.1 3.8 10. 1.3 5.0

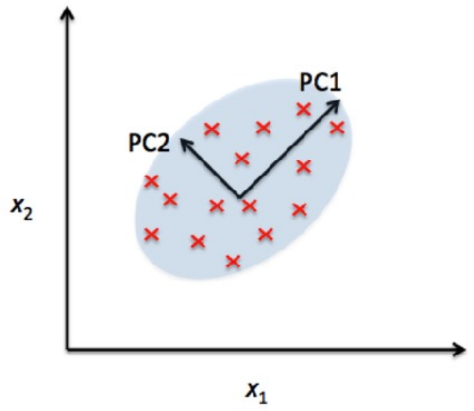
Unsupervised learning: no labels

- Clustering
  - Hierarchical
  - Non-hierarchical



- Dimensionality reduction
  - PCA
  - Feature selection





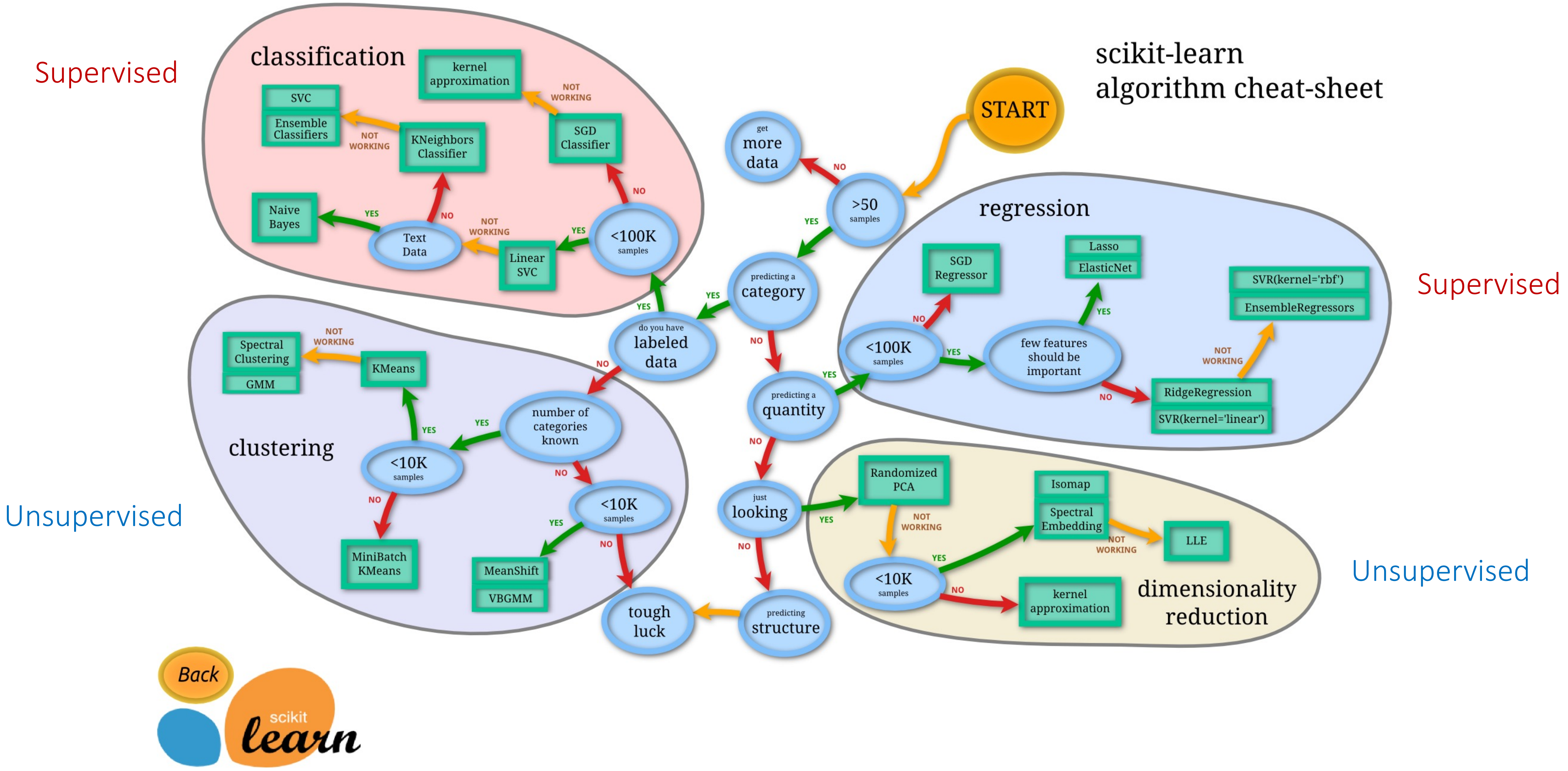
Bringing it all together

Type of labels	Model	Learning method
Continuous (numbers)	Regression	Supervised
Binary (classes)	Classification	
No labels	Clustering	Unsupervised
	Dimensionality reduction	

Many algorithms exist

Supervised

Unsupervised



Supervised

Unsupervised

Supervised learning: what is a model?

$$y = ax + b$$

The property that needs to be modeled. Can be a biological activity, or a physicochemical property.

Molecular description. Can be a molecular fingerprint, or some calculated properties like logP, MW, ...

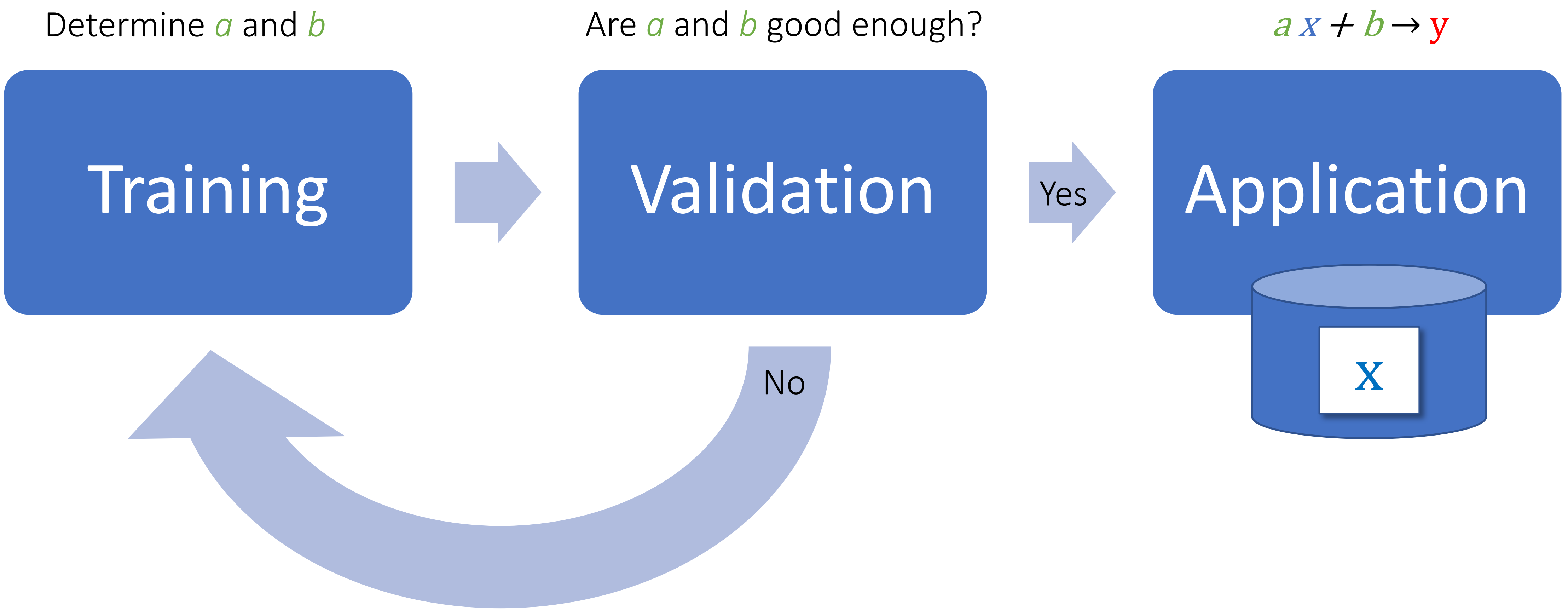
Model parameters. These parameters are determined during the training phase.

$$y = f(x)$$

f(x) can be anything:

- linear regression model
- random forest model
- neural network
- ...

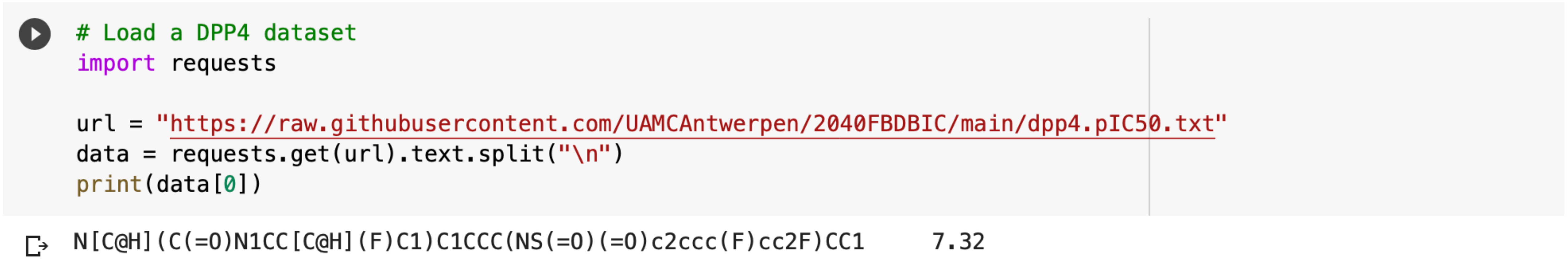
Model building phases



$$y = ax + b$$

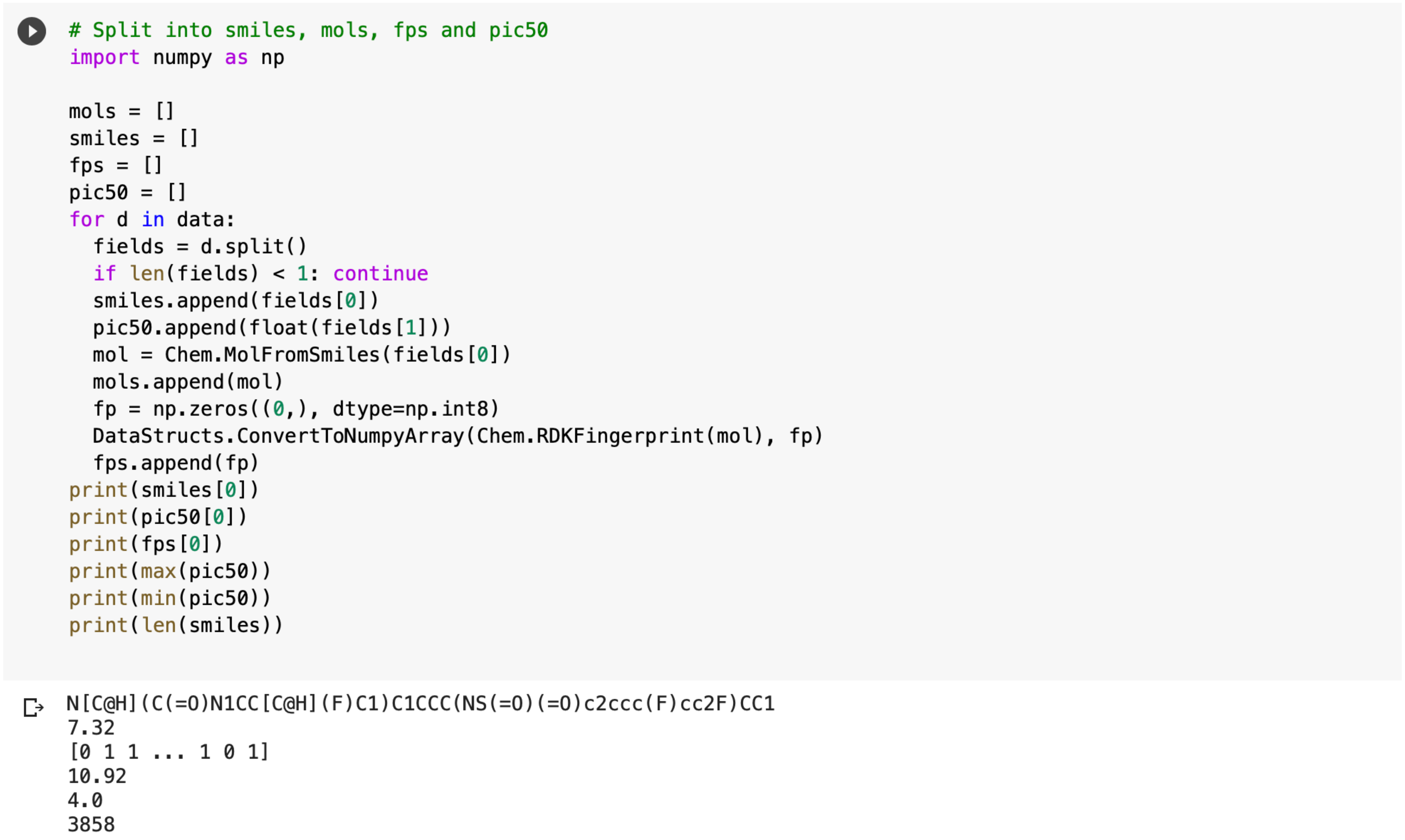
Linear regression

$$y = ax + b$$



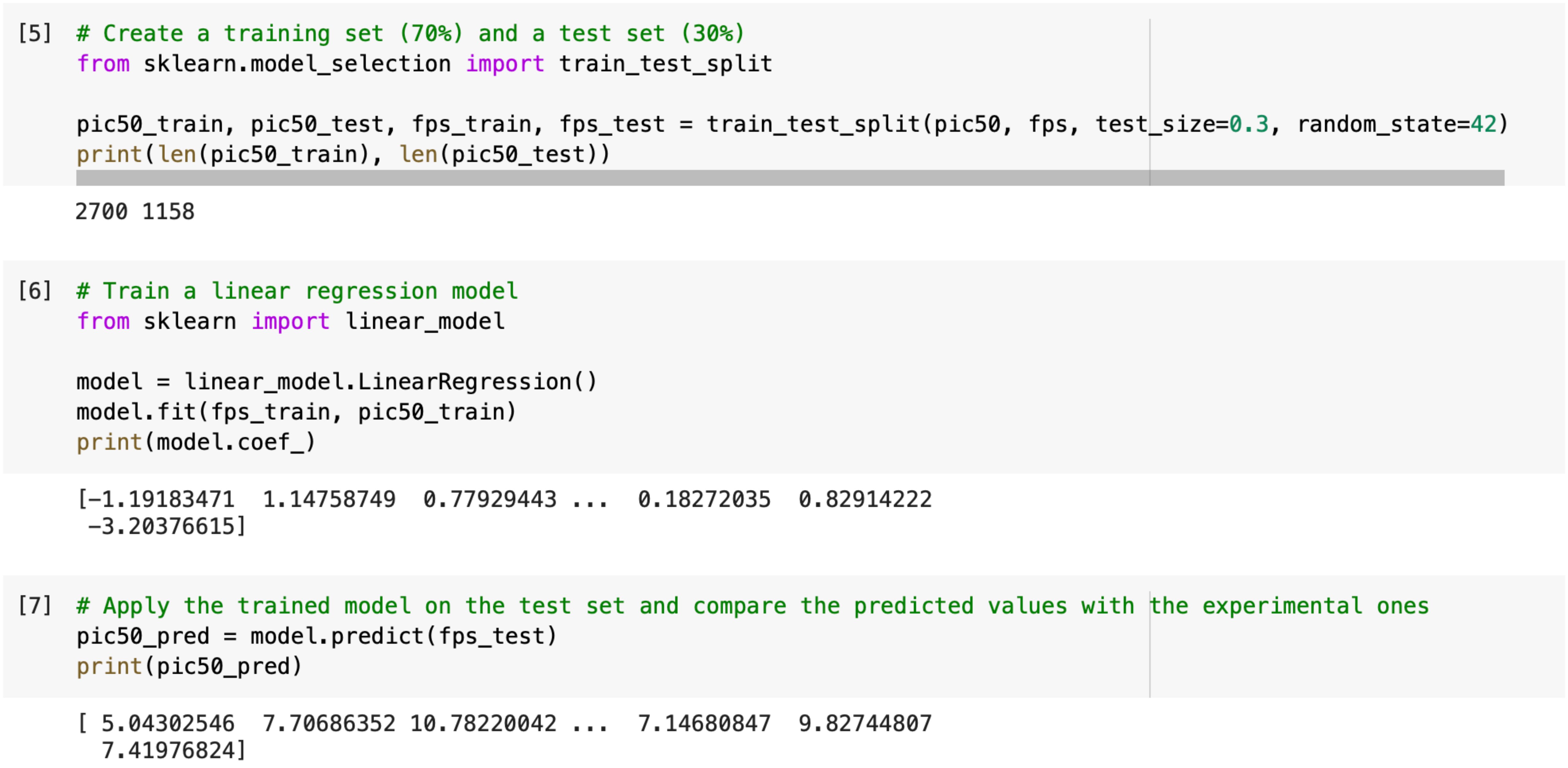
```
# Load a DPP4 dataset
import requests

url = "https://raw.githubusercontent.com/UAMCAntwerpen/2040FBDBIC/main/dpp4.pIC50.txt"
data = requests.get(url).text.split("\n")
print(data[0])
```

```
N[C@H](C(=O)N1CC[C@H](F)C1)C1CCC(NS(=O)(=O)c2ccc(F)cc2F)CC1     7.32
```



```python
# Split into smiles, mols, fps and pic50
import numpy as np

mols = []
smiles = []
fps = []
pic50 = []
for d in data:
  fields = d.split()
  if len(fields) < 1: continue
  smiles.append(fields[0])
  pic50.append(float(fields[1]))
  mol = Chem.MolFromSmiles(fields[0])
  mols.append(mol)
  fp = np.zeros((0,), dtype=np.int8)
  DataStructs.ConvertToNumpyArray(Chem.RDKFingerprint(mol), fp)
  fps.append(fp)
print(smiles[0])
print(pic50[0])
print(fps[0])
print(max(pic50))
print(min(pic50))
print(len(smiles))
```

```
N[C@H](C(=O)N1CC[C@H](F)C1)C1CCC(NS(=O)(=O)c2ccc(F)cc2F)CC1
7.32
[0 1 1 ... 1 0 1]
10.92
4.0
3858
```



```
[5] # Create a training set (70%) and a test set (30%)
    from sklearn.model_selection import train_test_split

    pic50_train, pic50_test, fps_train, fps_test = train_test_split(pic50, fps, test_size=0.3, random_state=42)
    print(len(pic50_train), len(pic50_test))
```

```
    2700 1158
```

```
[6] # Train a linear regression model
    from sklearn import linear_model

    model = linear_model.LinearRegression()
    model.fit(fps_train, pic50_train)
    print(model.coef_)
```

```
    [-1.19183471  1.14758749  0.77929443 ...  0.18272035  0.82914222
     -3.20376615]
```

```
[7] # Apply the trained model on the test set and compare the predicted values with the experimental ones
    pic50_pred = model.predict(fps_test)
    print(pic50_pred)
```

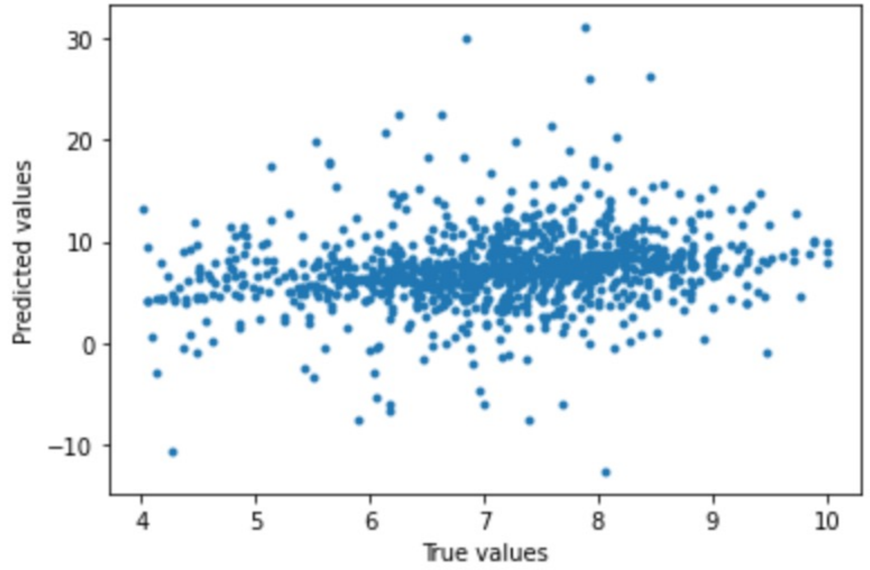
```
    [ 5.04302546  7.70686352 10.78220042 ...  7.14680847  9.82744807
      7.41976824]
```

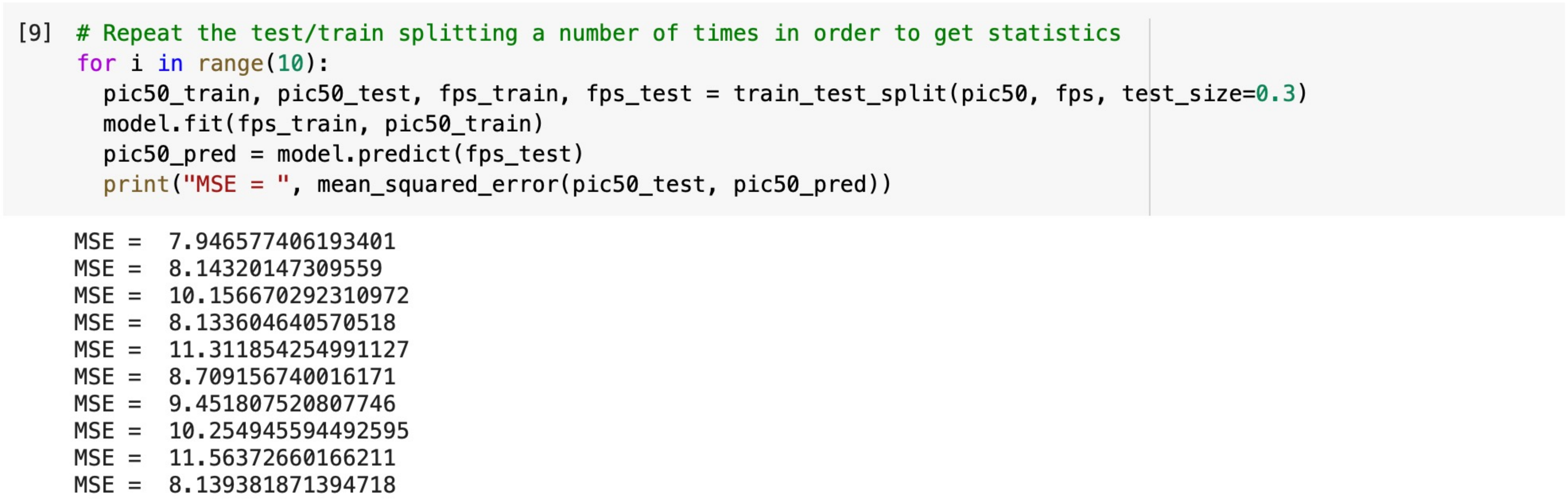
```
[7] # Apply the trained model on the test set and compare the predicted values with the experimental ones
    pic50_pred = model.predict(fps_test)
    print(pic50_pred)
```

```
[ 5.04302546  7.70686352 10.78220042 ...  7.14680847  9.82744807
  7.41976824]
```

```
[8] # Validate the model by calculating the MSE of the predictions when compared to the true values
    from sklearn.metrics import mean_squared_error
    import matplotlib.pyplot as plt

    print("MSE = ", mean_squared_error(pic50_test, pic50_pred))
    plt.plot(pic50_test, pic50_pred, '.')
    plt.xlabel("True values")
    plt.ylabel("Predicted values")
```

```
MSE =  13.059437179691 74
Text(0, 0.5, 'Predicted values')
```

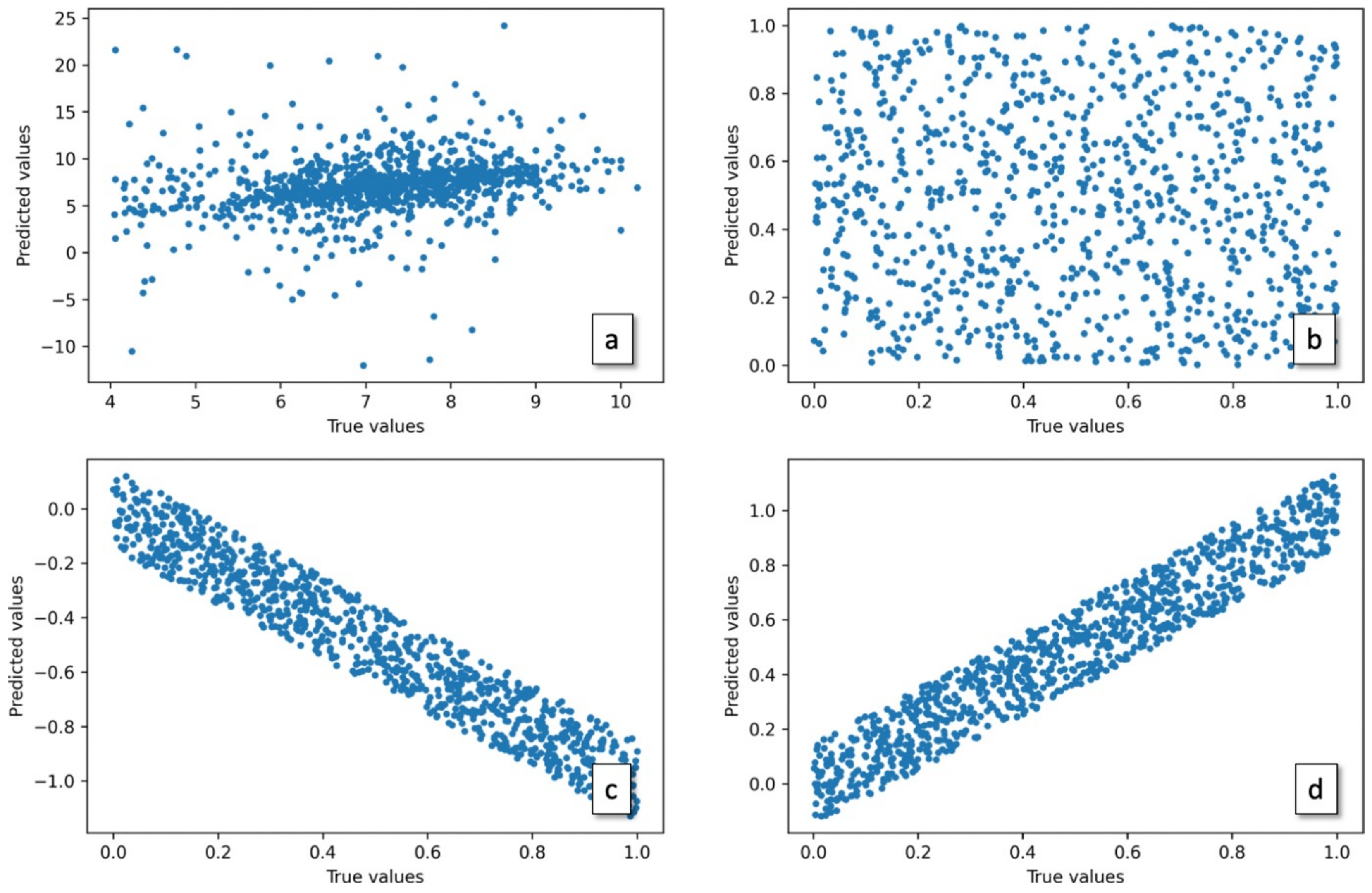




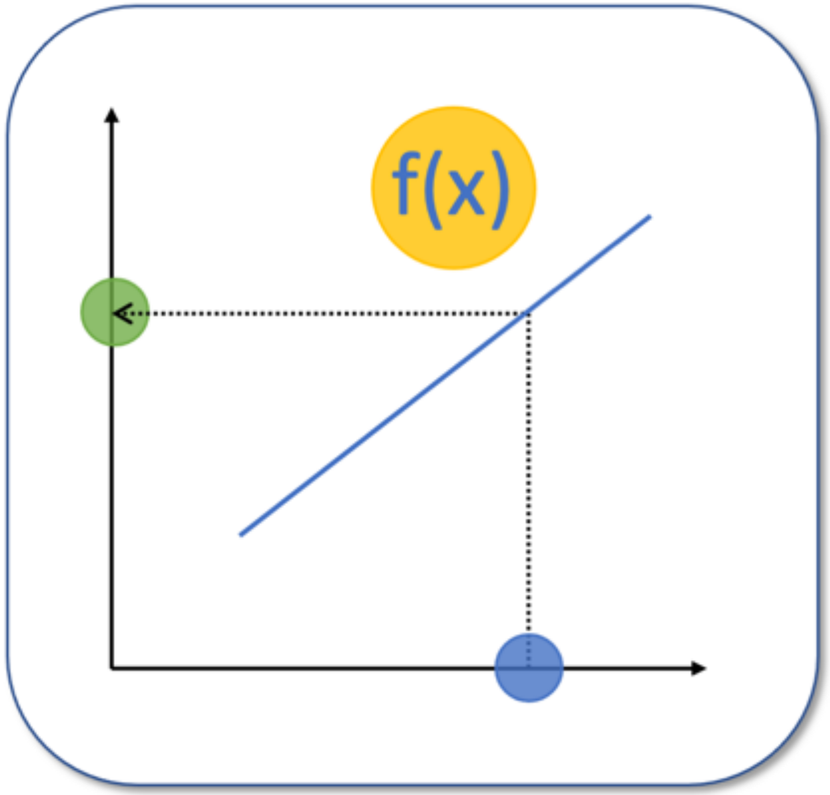
```
[9] # Repeat the test/train splitting a number of times in order to get statistics
    for i in range(10):
      pic50_train, pic50_test, fps_train, fps_test = train_test_split(pic50, fps, test_size=0.3)
      model.fit(fps_train, pic50_train)
      pic50_pred = model.predict(fps_test)
      print("MSE = ", mean_squared_error(pic50_test, pic50_pred))
```

```
MSE =  7.946577406193401
MSE =  8.14320147309559
MSE =  10.156670292310972
MSE =  8.133604640570518
MSE =  11.311854254991127
MSE =  8.709156740016171
MSE =  9.451807520807746
MSE =  10.254945594492595
MSE =  11.56372660166211
MSE =  8.139381871394718
```

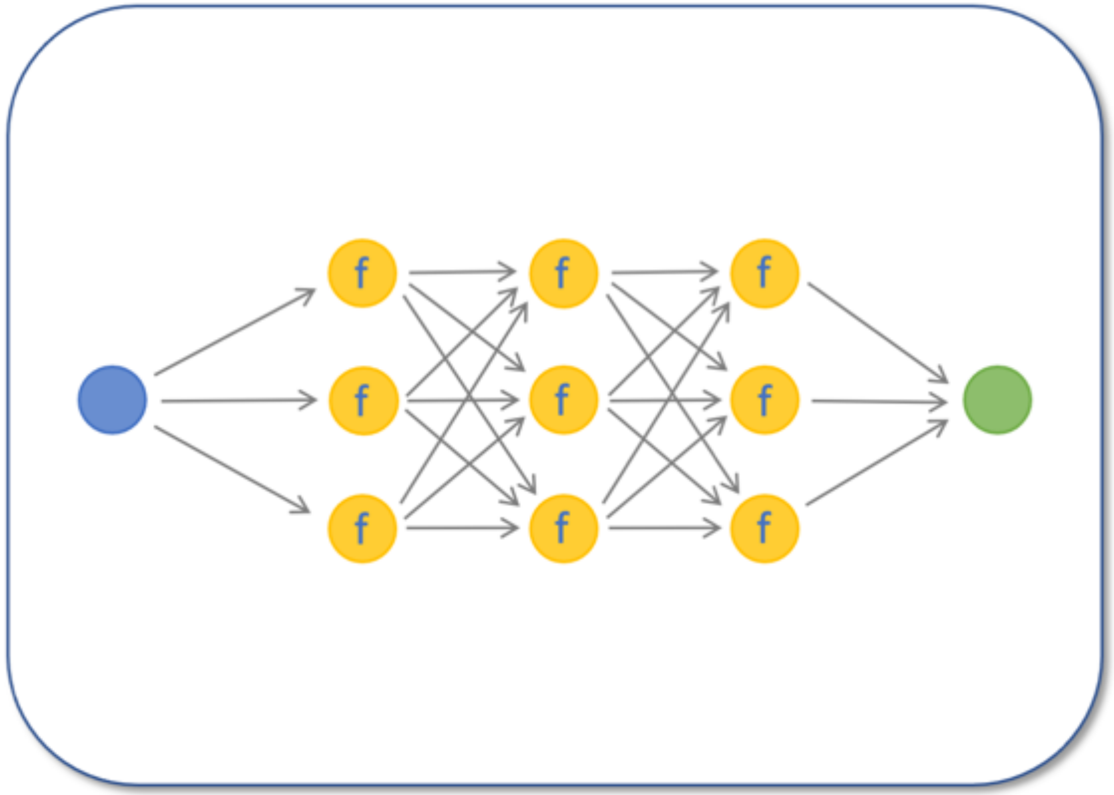
Correlation plots



Neural network

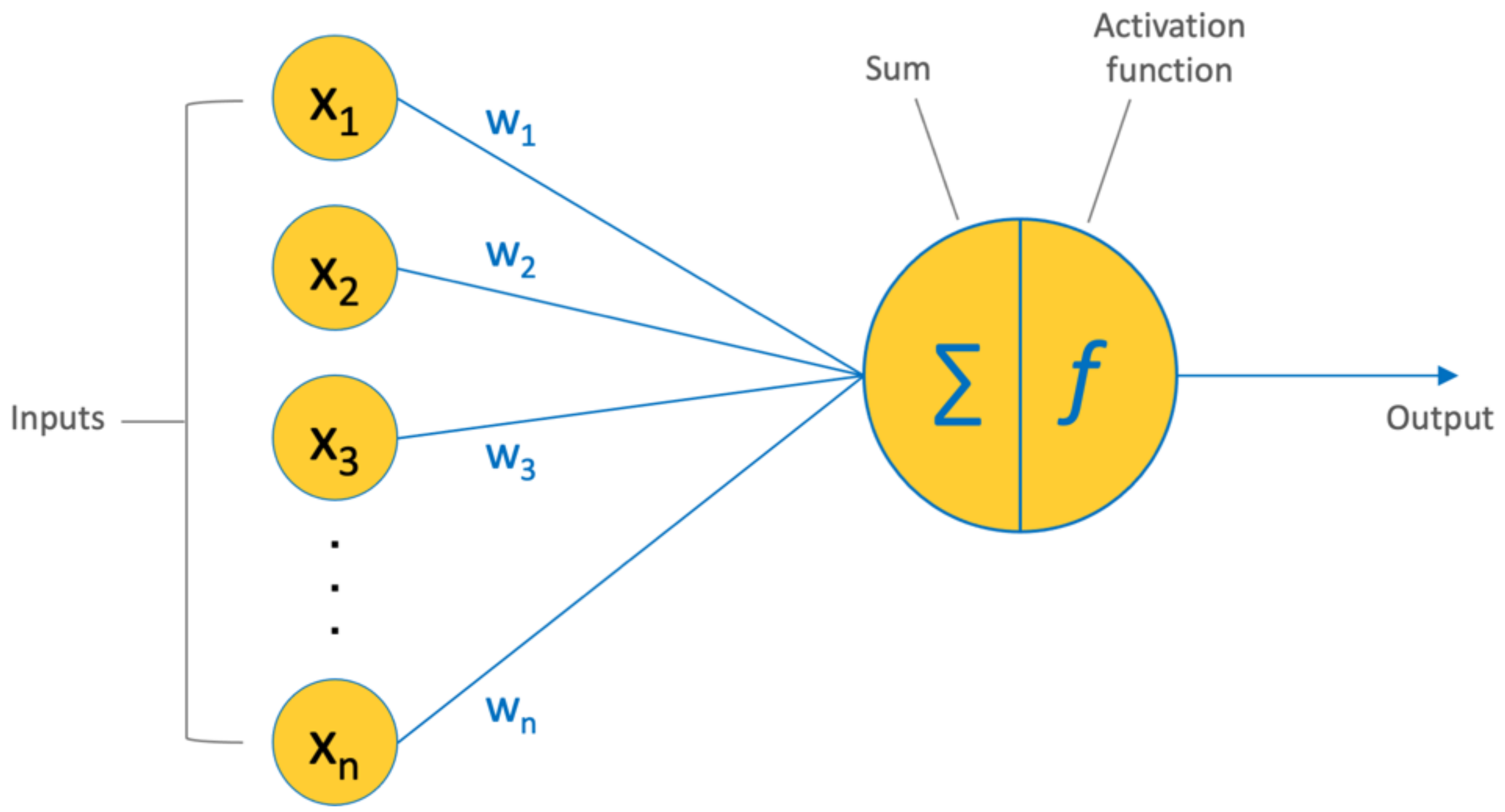


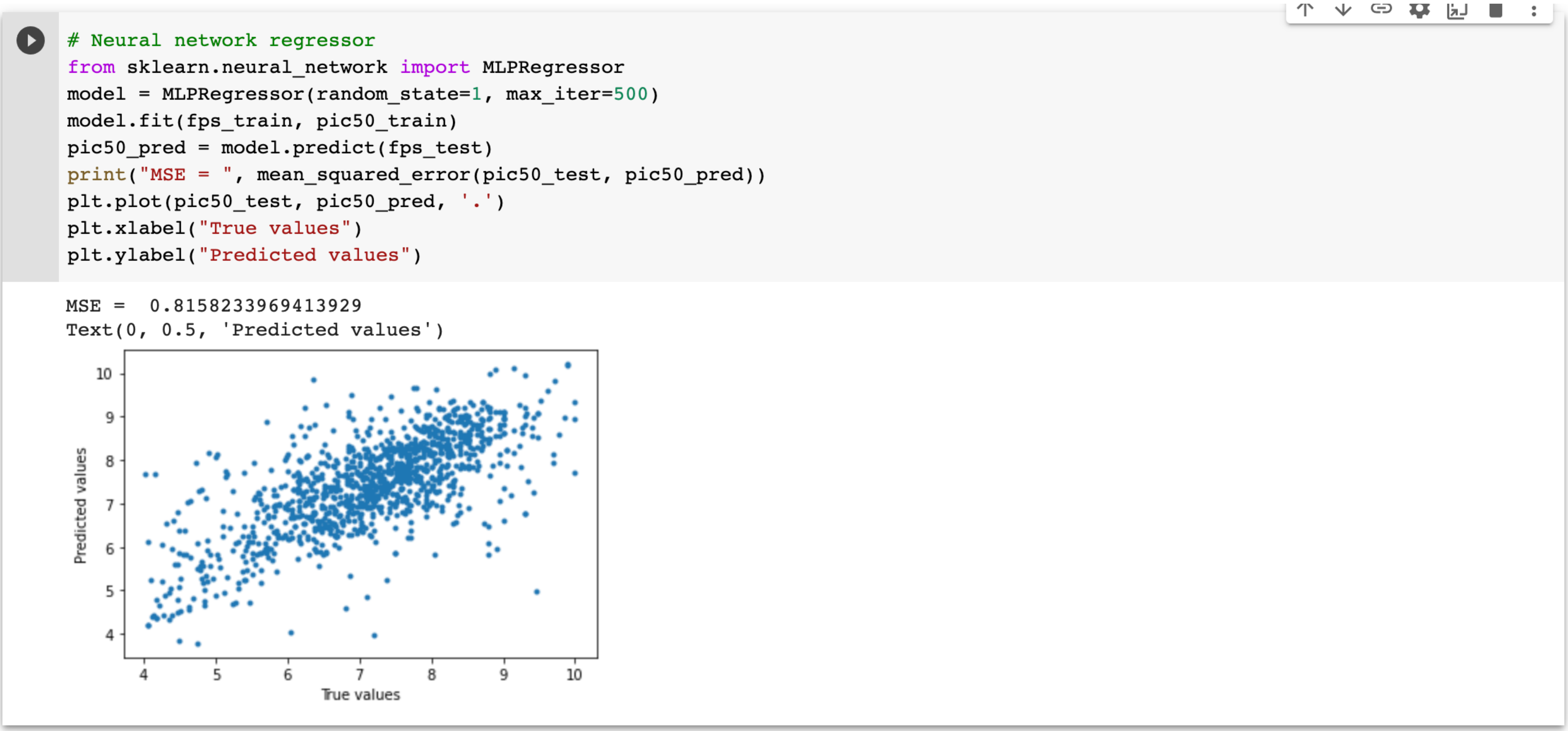
NEURON



NEURAL NETWORK

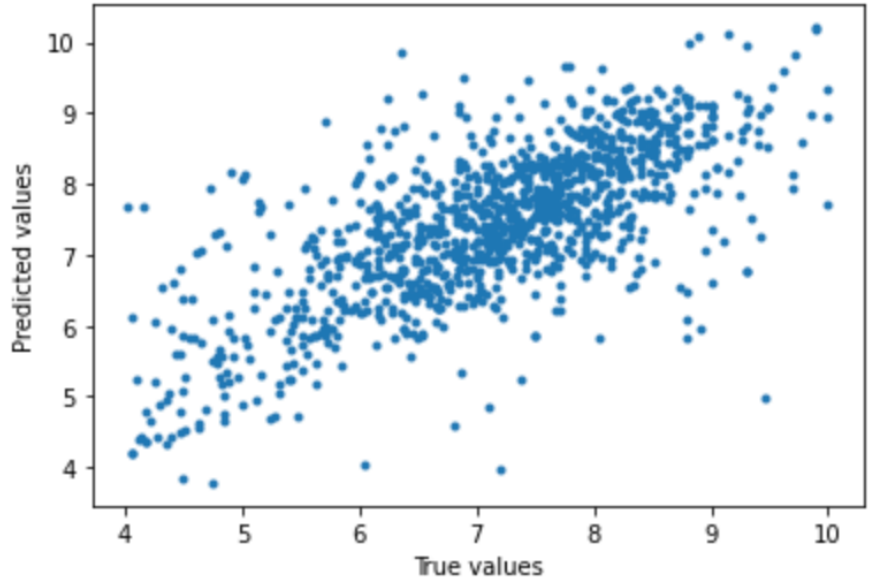
Percepron

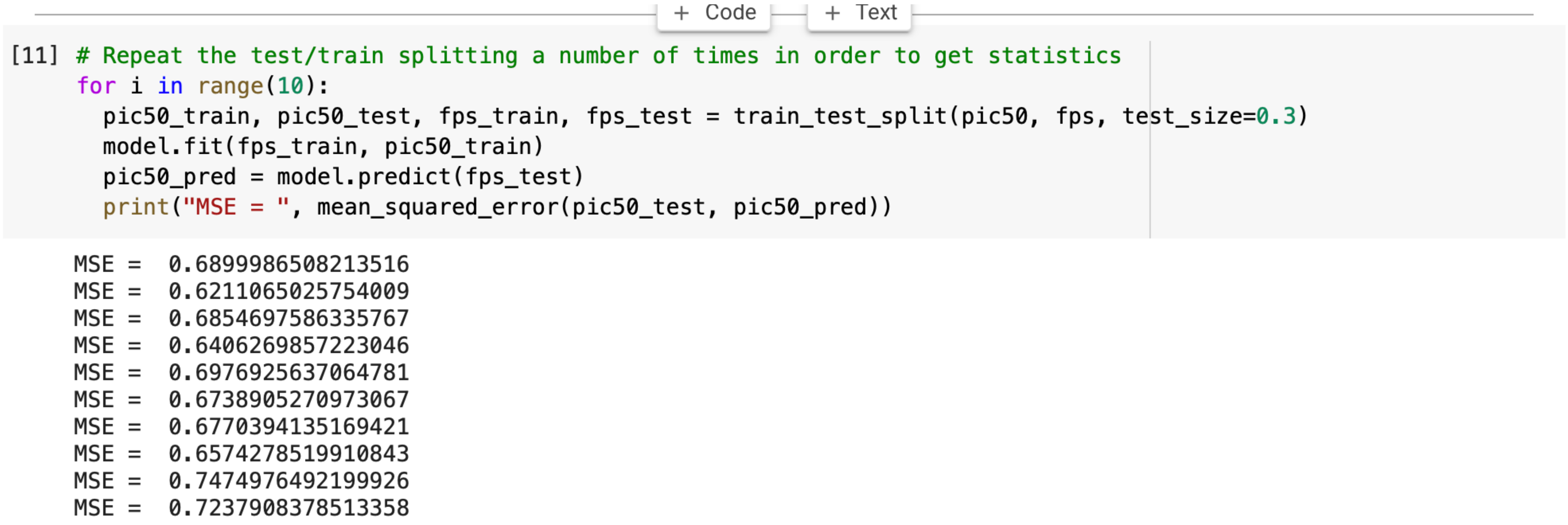

Inputs
x_1
x_2
x_3
x_n
w_1
w_2
w_3
w_n
Sum
Activation function
Σ
f
Output



```
# Neural network regressor
from sklearn.neural_network import MLPRegressor
model = MLPRegressor(random_state=1, max_iter=500)
model.fit(fps_train, pic50_train)
pic50_pred = model.predict(fps_test)
print("MSE = ", mean_squared_error(pic50_test, pic50_pred))
plt.plot(pic50_test, pic50_pred, '.')
plt.xlabel("True values")
plt.ylabel("Predicted values")
```

```
MSE =  0.8158233969413929
Text(0, 0.5, 'Predicted values')
```

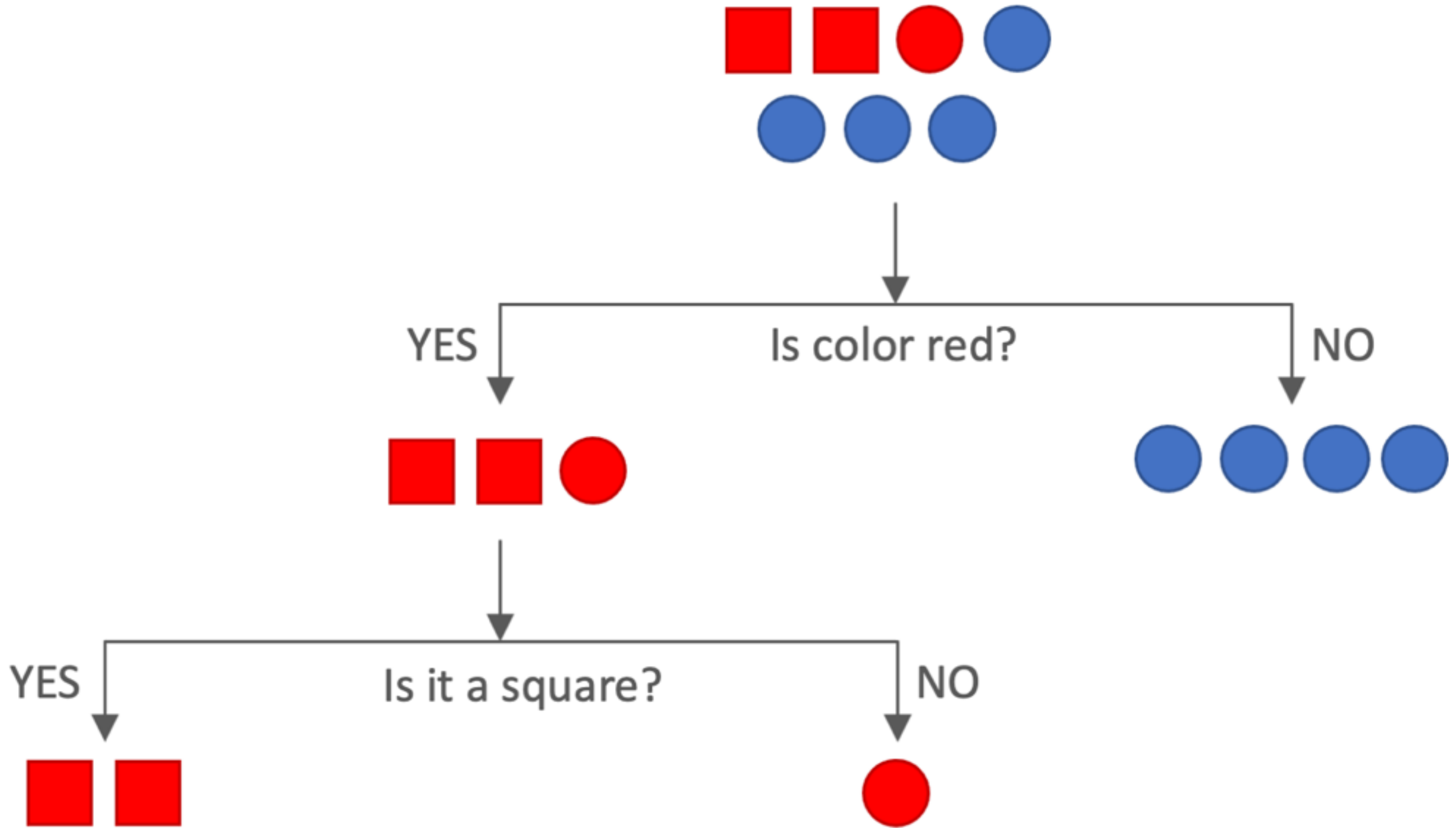


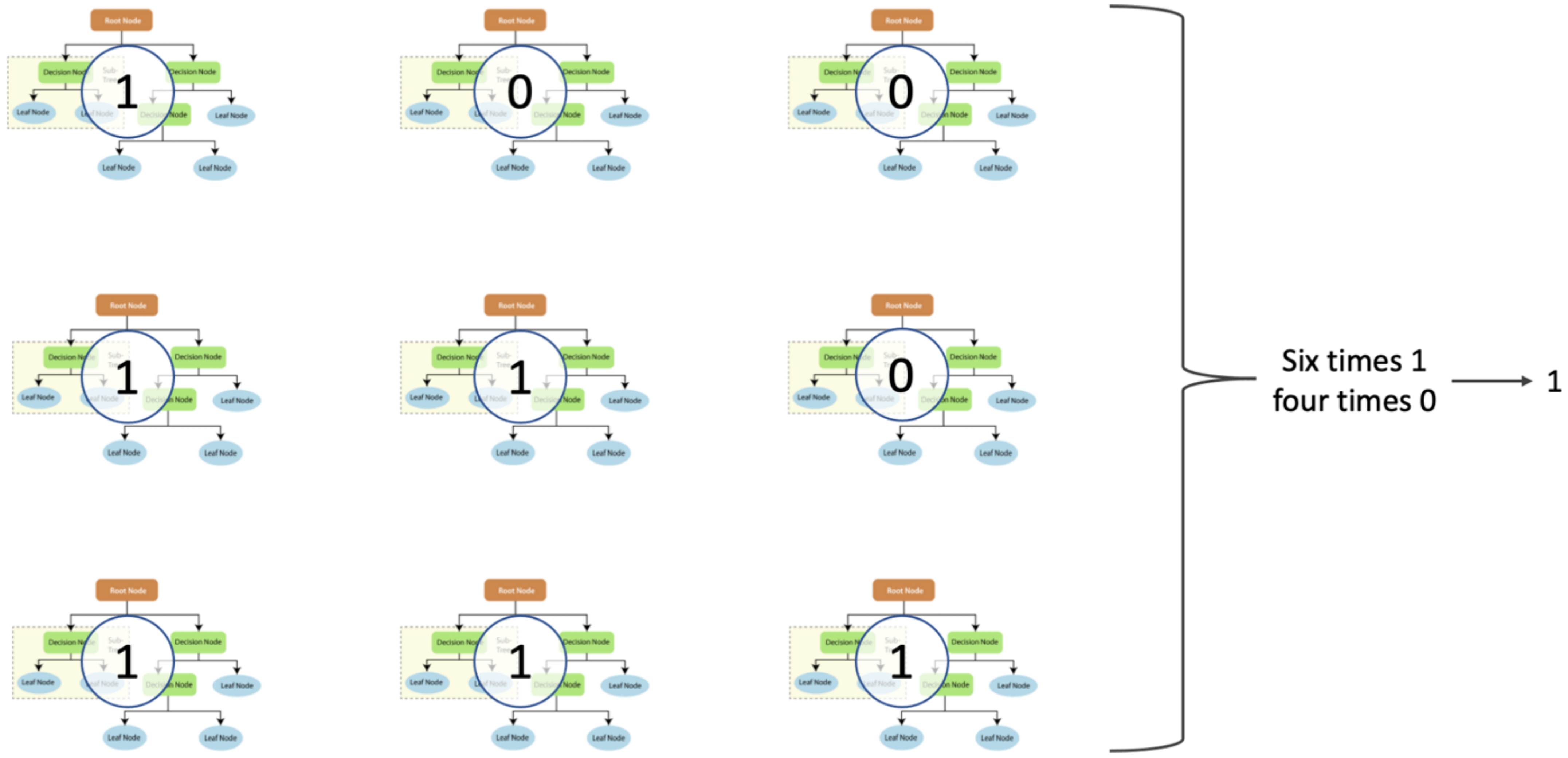


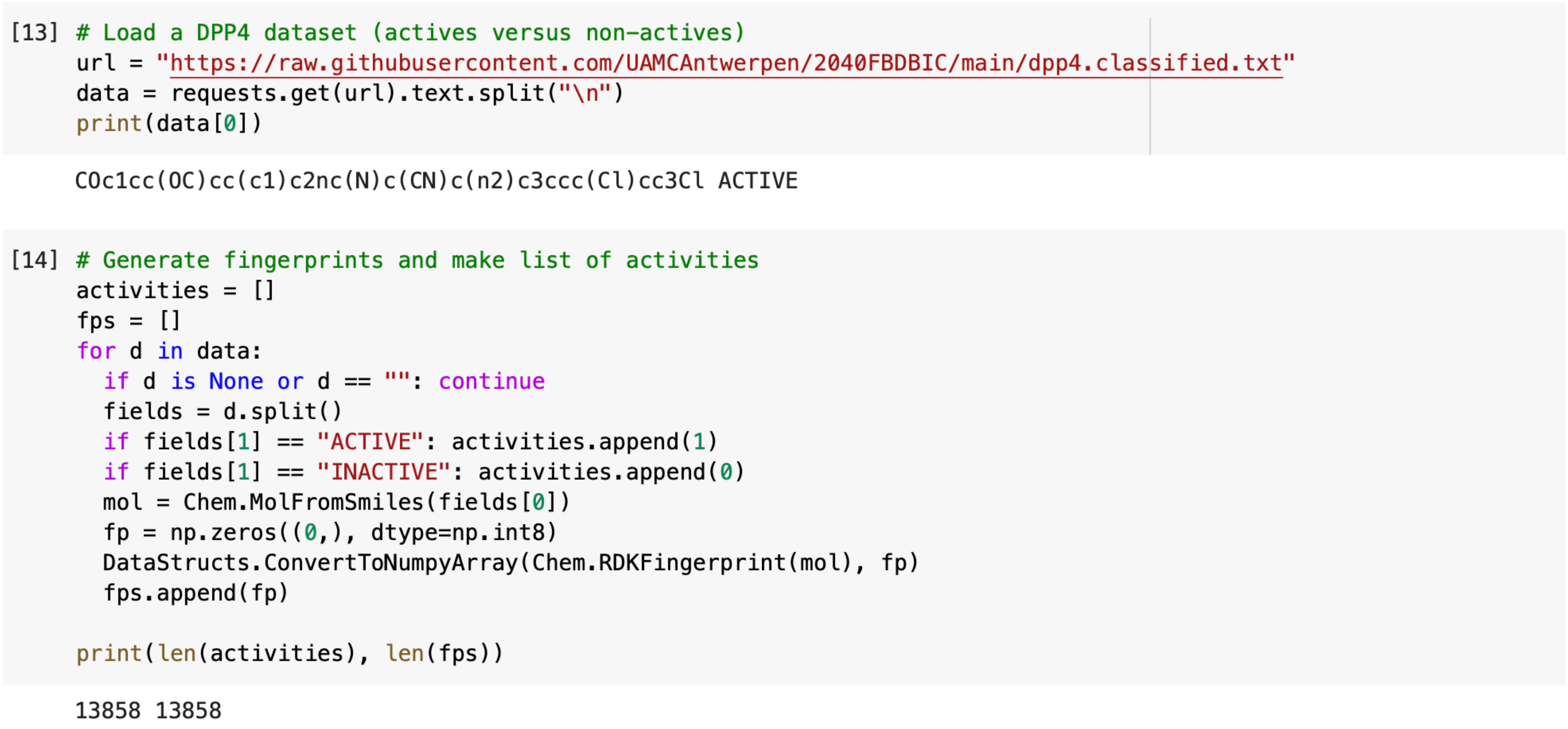
```
[11] # Repeat the test/train splitting a number of times in order to get statistics
     for i in range(10):
       pic50_train, pic50_test, fps_train, fps_test = train_test_split(pic50, fps, test_size=0.3)
       model.fit(fps_train, pic50_train)
       pic50_pred = model.predict(fps_test)
       print("MSE = ", mean_squared_error(pic50_test, pic50_pred))
```

```
MSE =  0.6899986508213516
MSE =  0.6211065025754009
MSE =  0.6854697586335767
MSE =  0.6406269857223046
MSE =  0.6976925637064781
MSE =  0.6738905270973067
MSE =  0.6770394135169421
MSE =  0.6574278519910843
MSE =  0.7474976492199926
MSE =  0.7237908378513358
```

Random forest classifier
(a forest of decision trees)




Root Node
Decision Node
Sub-Tree
Leaf Node
1
0
0
1
1
0
1
1
1
Six times 1
four times 0
1

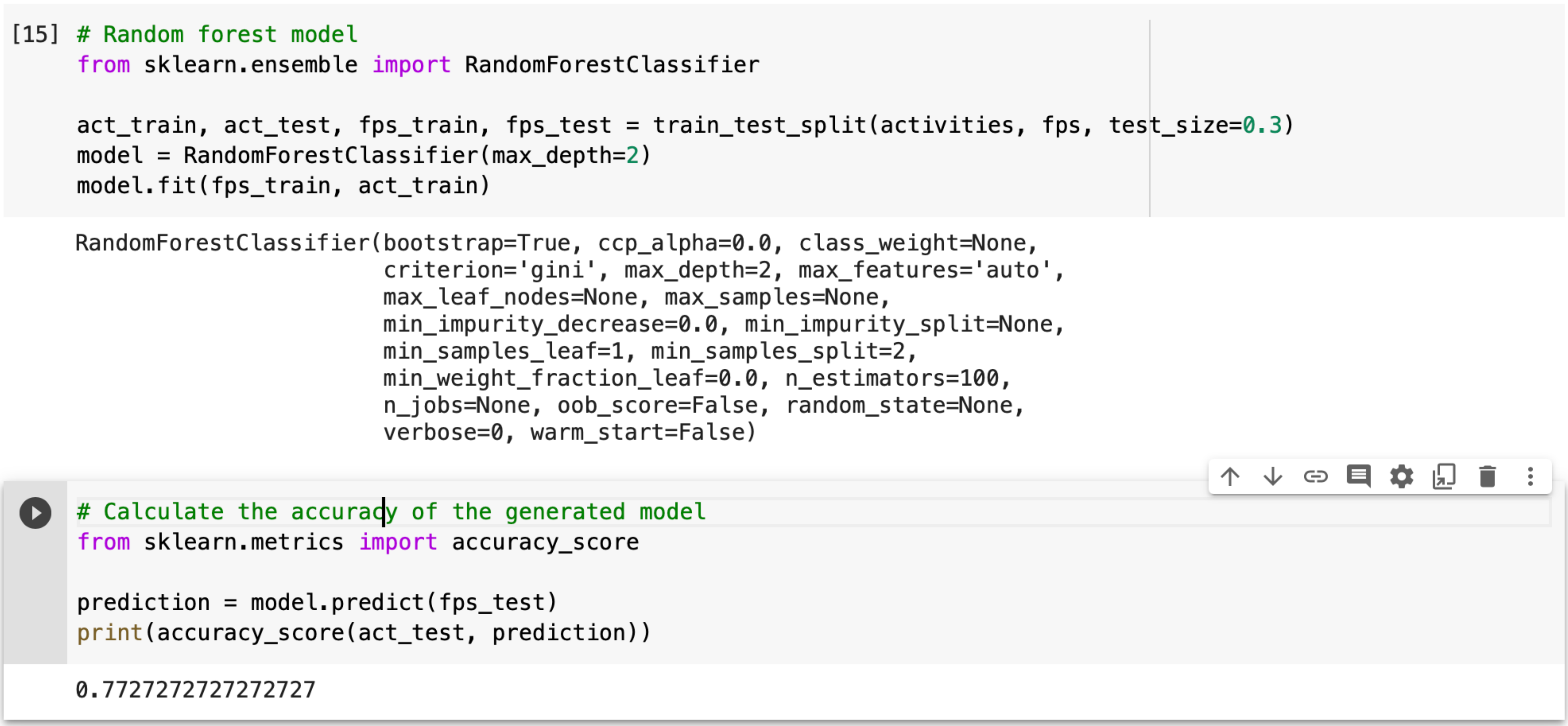


```
[13] # Load a DPP4 dataset (actives versus non-actives)
     url = "https://raw.githubusercontent.com/UAMCAntwerpen/2040FBDBIC/main/dpp4.classified.txt"
     data = requests.get(url).text.split("\n")
     print(data[0])
```

```
     COc1cc(OC)cc(c1)c2nc(N)c(CN)c(n2)c3ccc(Cl)cc3Cl ACTIVE
```

```
[14] # Generate fingerprints and make list of activities
     activities = []
     fps = []
     for d in data:
       if d is None or d == "": continue
       fields = d.split()
       if fields[1] == "ACTIVE": activities.append(1)
       if fields[1] == "INACTIVE": activities.append(0)
       mol = Chem.MolFromSmiles(fields[0])
       fp = np.zeros((0,), dtype=np.int8)
       DataStructs.ConvertToNumpyArray(Chem.RDKFingerprint(mol), fp)
       fps.append(fp)

     print(len(activities), len(fps))
```

```
     13858 13858
```

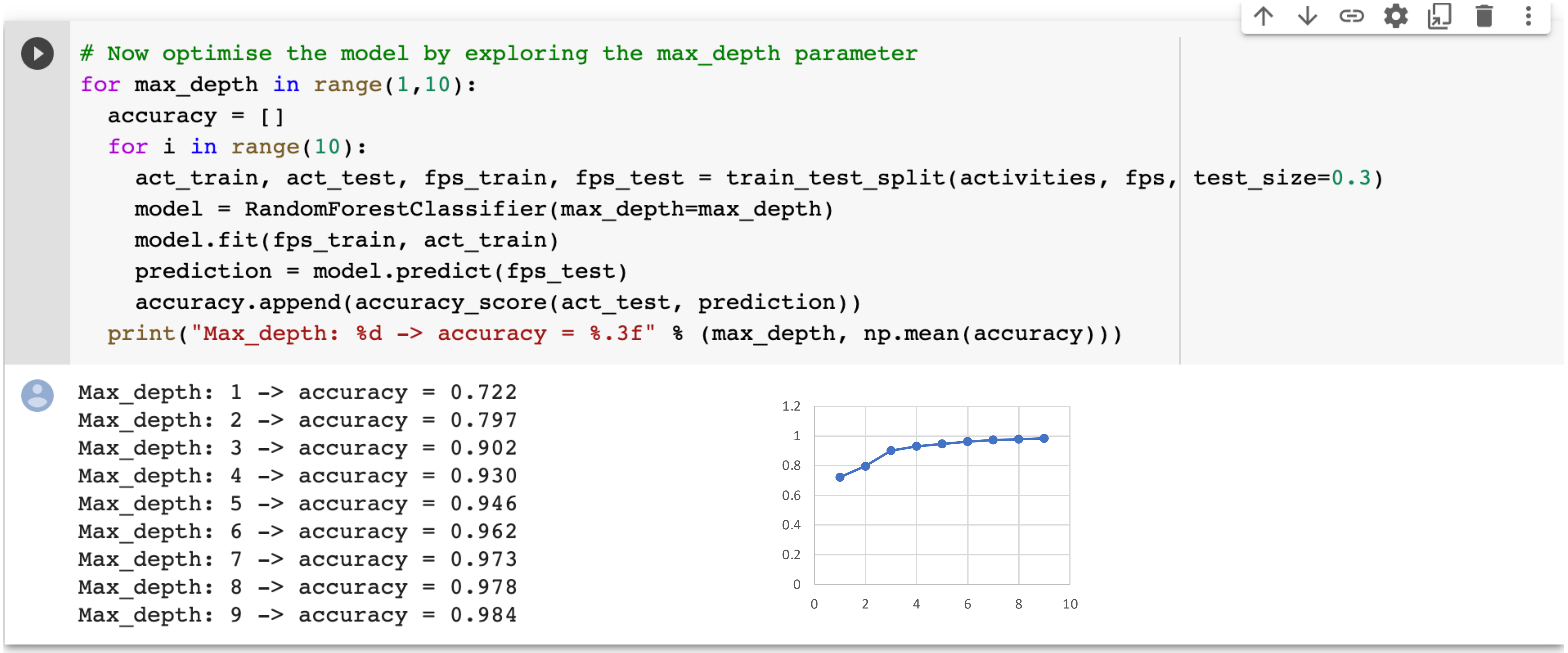


```
[15] # Random forest model
     from sklearn.ensemble import RandomForestClassifier

     act_train, act_test, fps_train, fps_test = train_test_split(activities, fps, test_size=0.3)
     model = RandomForestClassifier(max_depth=2)
     model.fit(fps_train, act_train)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=2, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

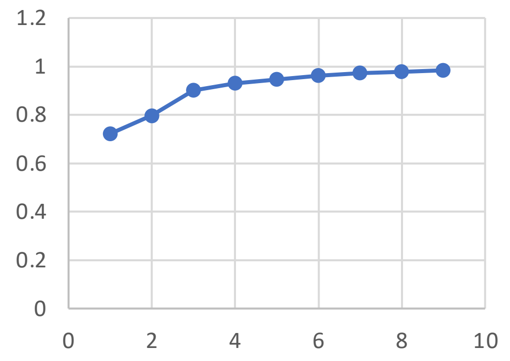
```
# Calculate the accuracy of the generated model
from sklearn.metrics import accuracy_score

prediction = model.predict(fps_test)
print(accuracy_score(act_test, prediction))
```

```
0.7727272727272727
```



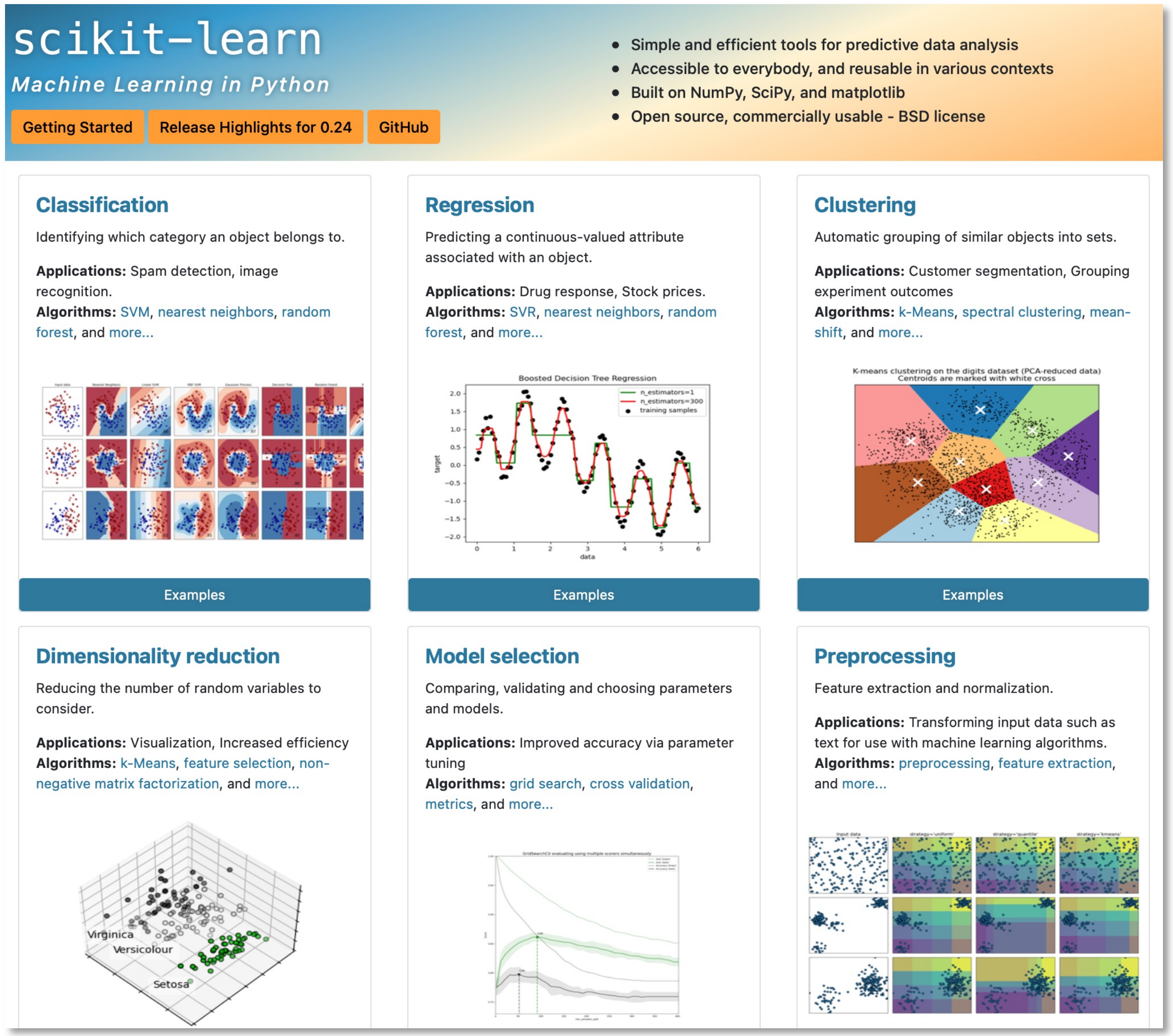
```python
# Now optimise the model by exploring the max_depth parameter
for max_depth in range(1,10):
  accuracy = []
  for i in range(10):
    act_train, act_test, fps_train, fps_test = train_test_split(activities, fps, test_size=0.3)
    model = RandomForestClassifier(max_depth=max_depth)
    model.fit(fps_train, act_train)
    prediction = model.predict(fps_test)
    accuracy.append(accuracy_score(act_test, prediction))
  print("Max_depth: %d -> accuracy = %.3f" % (max_depth, np.mean(accuracy)))
```

```
Max_depth: 1 -> accuracy = 0.722
Max_depth: 2 -> accuracy = 0.797
Max_depth: 3 -> accuracy = 0.902
Max_depth: 4 -> accuracy = 0.930
Max_depth: 5 -> accuracy = 0.946
Max_depth: 6 -> accuracy = 0.962
Max_depth: 7 -> accuracy = 0.973
Max_depth: 8 -> accuracy = 0.978
Max_depth: 9 -> accuracy = 0.984
```



scikit-learn: many useful models ready to use

https://scikit-learn.org/stable/index.html



scikit-learn

Machine Learning in Python

Getting Started | Release Highlights for 0.24 | GitHub

- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying which category an object belongs to.

Applications: Spam detection, image recognition.
Algorithms: SVM, nearest neighbors, random forest, and more...

Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, nearest neighbors, random forest, and more...

Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, and more...

Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: k-Means, feature selection, non-negative matrix factorization, and more...

Model selection

Comparing, validating and choosing parameters and models.

Applications: Improved accuracy via parameter tuning
Algorithms: grid search, cross validation, metrics, and more...

Preprocessing

Feature extraction and normalization.

Applications: Transforming input data such as text for use with machine learning algorithms.
Algorithms: preprocessing, feature extraction, and more...

Boosted Decision Tree Regression
n_estimators=1
n_estimators=300
training samples
K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross
Virginica
Versicolour
Setosa

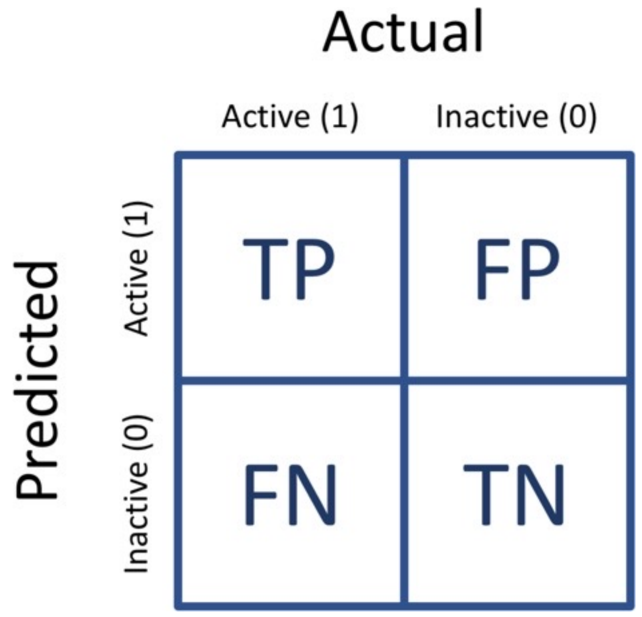
Supervised:
- Classification
- Regression

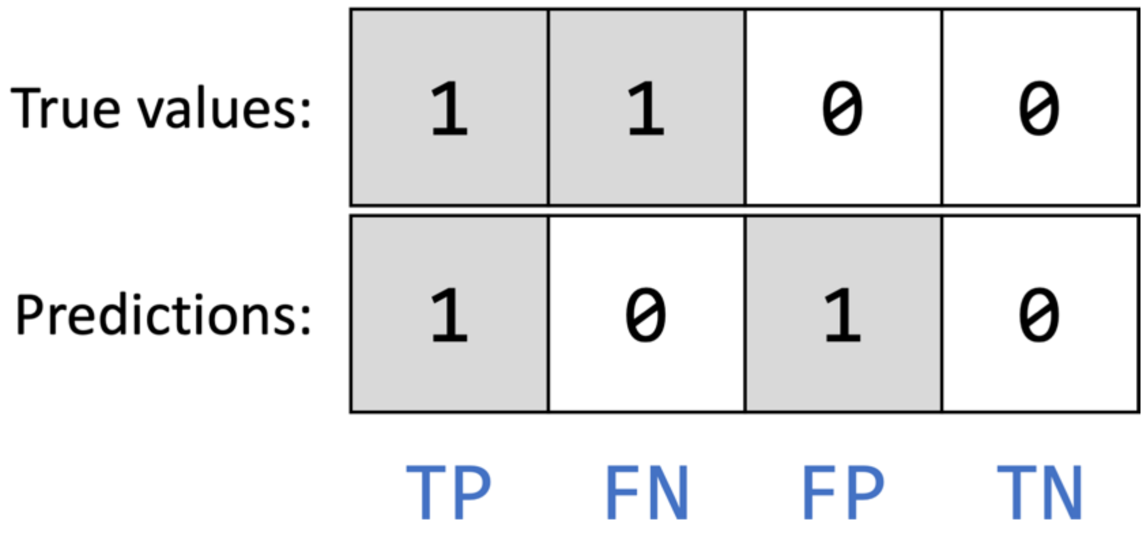
Unsupervised:
- Clustering
- Dimensionality reduction

Clustering and machine learning

- Molecular similarity
- MCSS
- Clustering
- Machine learning: QSAR
- Validation

Validation of classification models: The confusion matrix





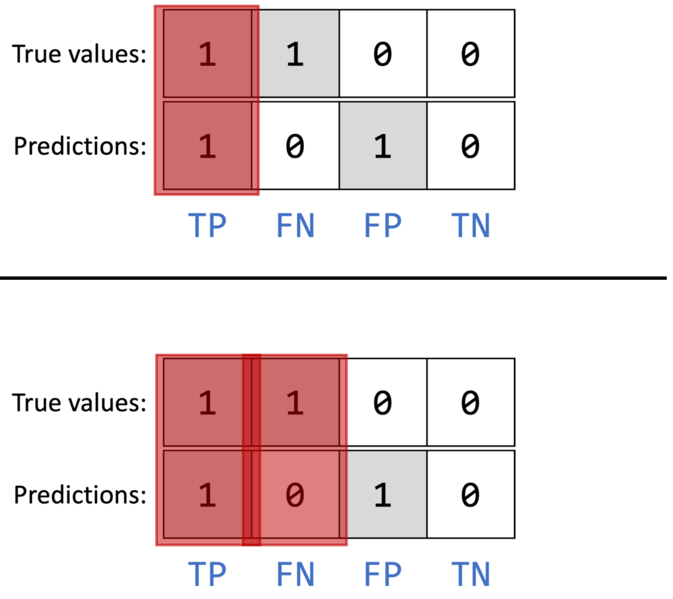
True positive rate (TPR)

- TPR = Sensitivity = recall

  Tells us which fraction of the true actives are actually predicted by the model to be active.

  <u>Issue</u>: since that the *FP*'s are not part of the equation, a model that predicts all compounds to be active (also those that are not) leads to a *TPR* of 1...

$$Sensitivity = recall = \frac{TP}{TP + FN} = TPR \quad =$$



True negative rate (TNR)

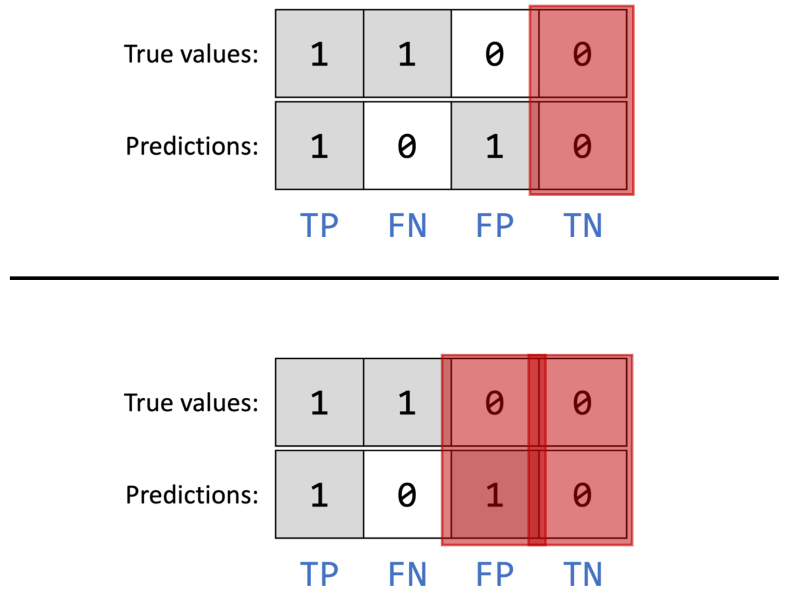
- TNR = Specificity

  Tells us which fraction of the true non-actives are actually predicted by the model to be non-active.

  Issue: since that the *FN*'s are not part of the equation, a model that predicts all compounds to be non-active (also those that are not) leads to a *TNR* of 1...

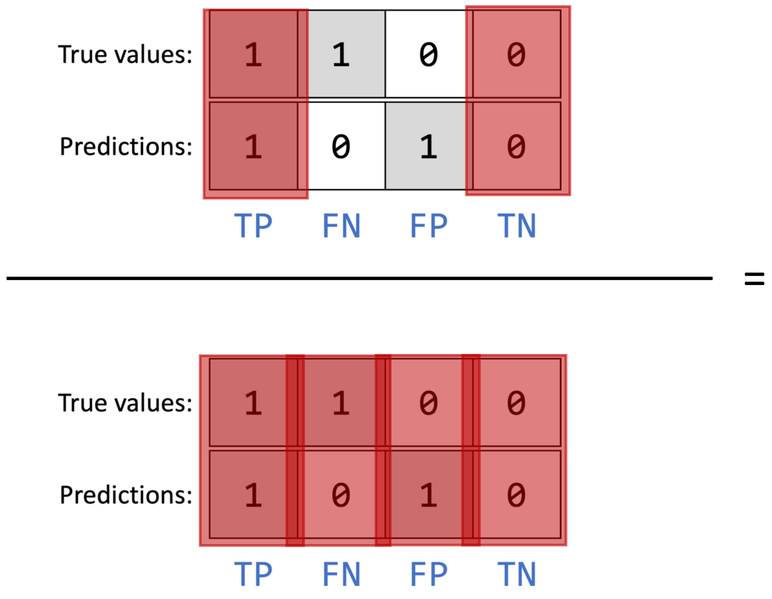
$$Specificity = \frac{TN}{TN + FP} = TNR$$

=



Accuracy

- Compromise between TPR and TNR
  Tells us which fraction of the predictions are indeed correct predictions



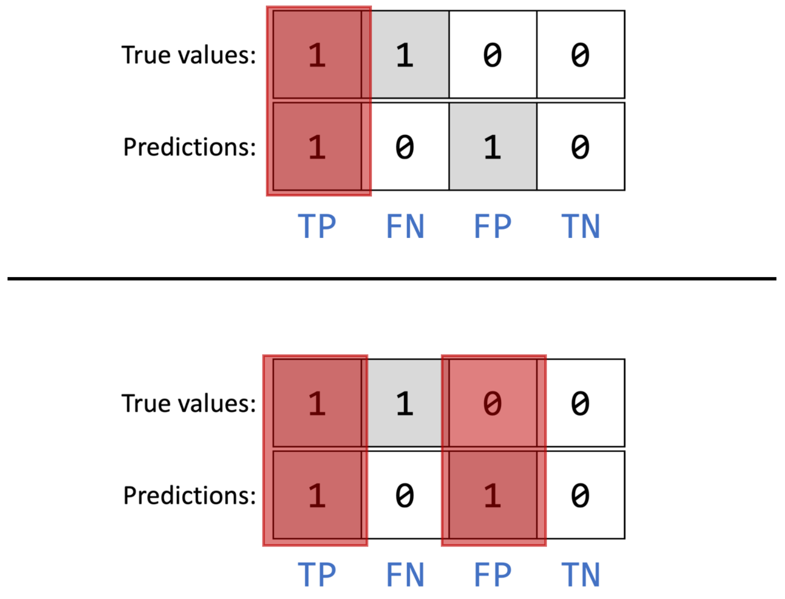
$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Precision

- Precision = Positive predictive value (PPV)

  Tells us which fraction of the predicted actives are actually real actives.

$$Precision = \frac{TP}{TP + FP} =$$



Other performance metrics

- False positive rate (FPR) = Fall-out

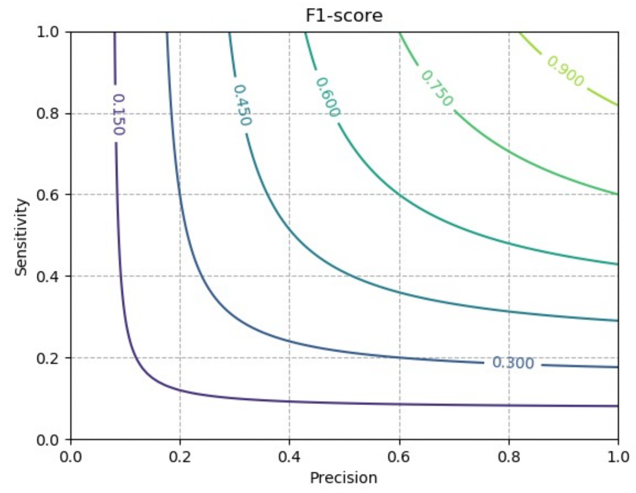
$$Fall\ out = \frac{FP}{FP + TN} = FPR$$

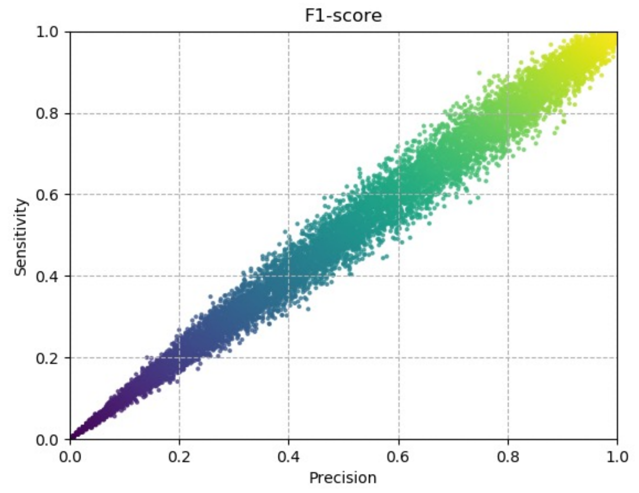
- False negative rate (FNR) = Miss rate

$$FNR = \frac{FN}{FN + TP}$$

- F1-score: harmonic mean of precision and sensitivity:

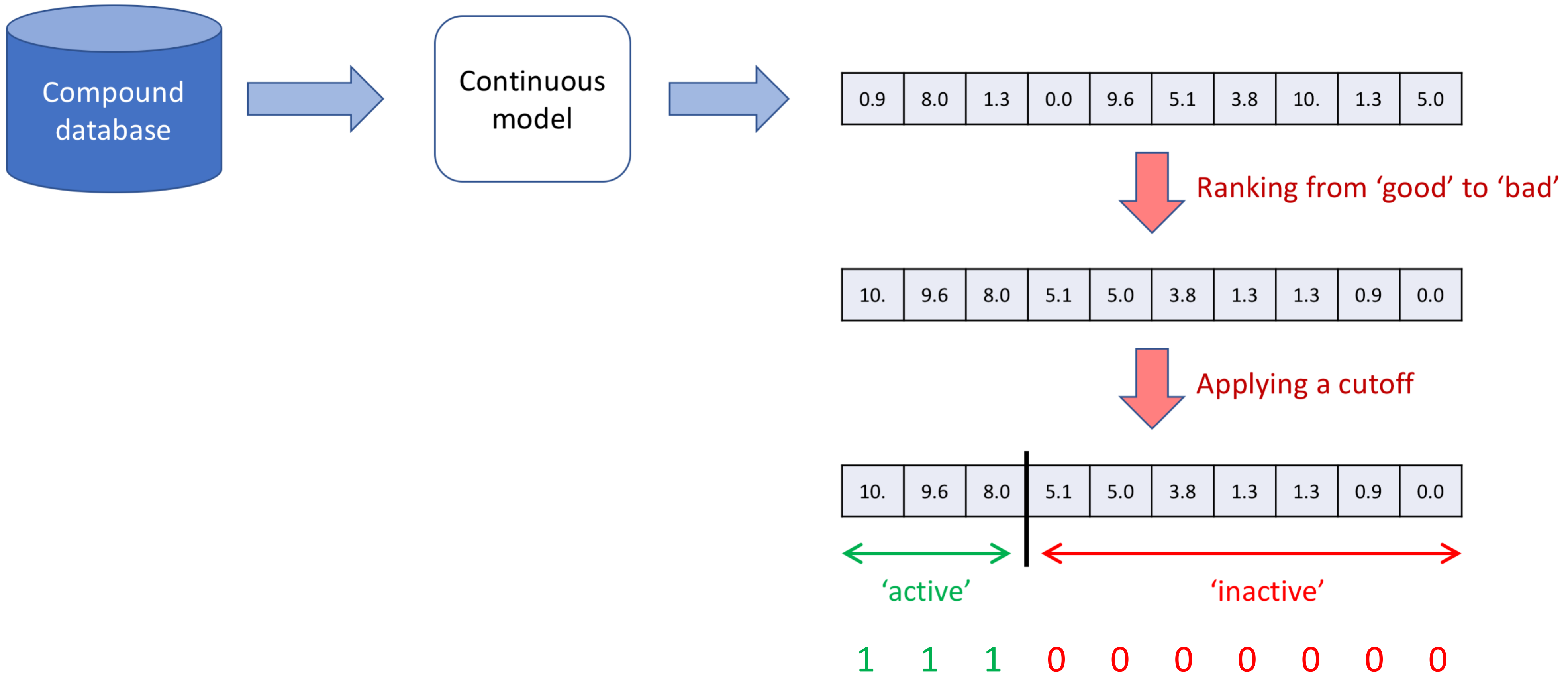
$$F1\ score = \frac{2 * precision * recall}{precision + recall}$$



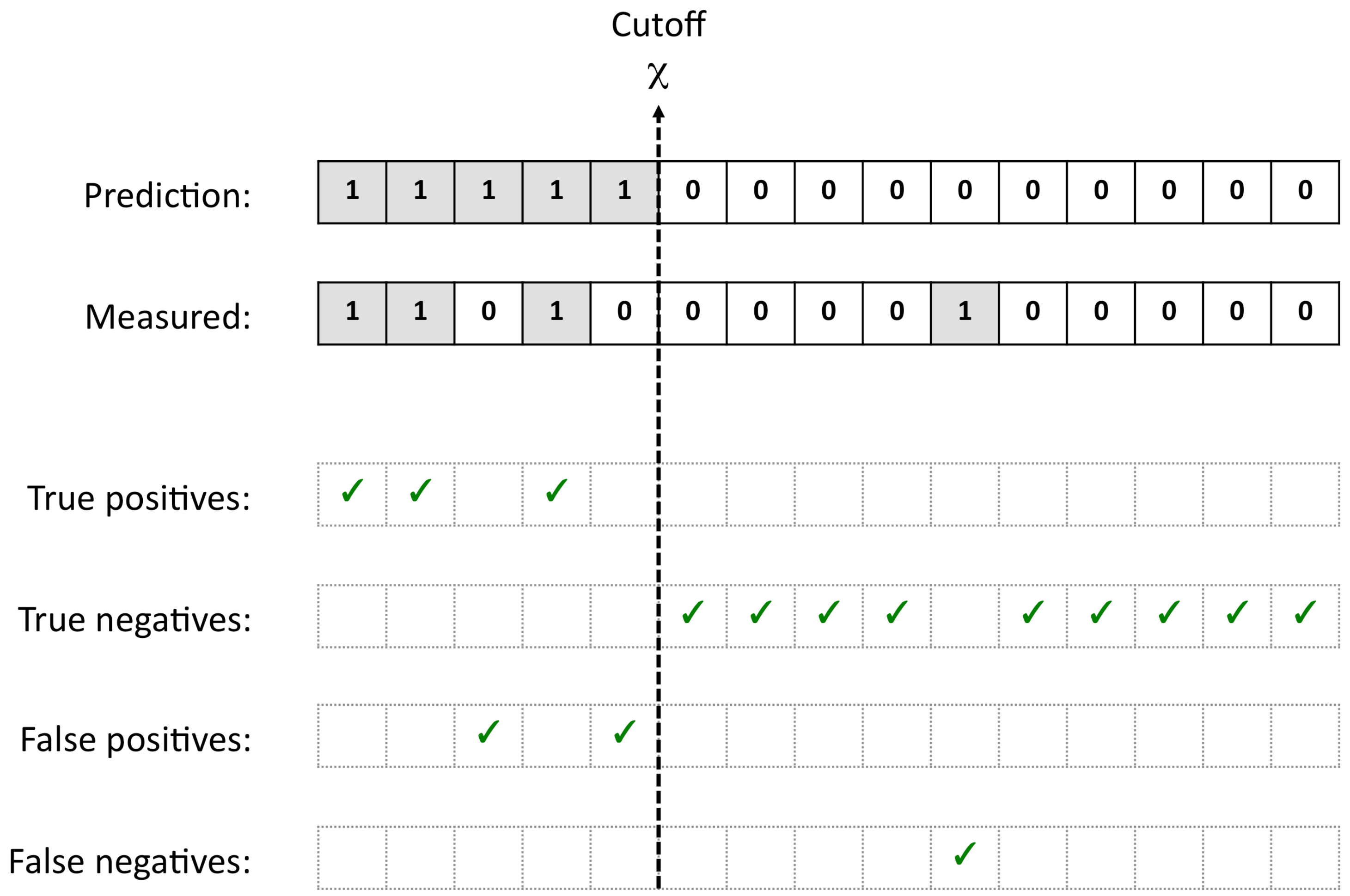


Validation of continous models: the cutoff value

- We can use a cutoff value to convert a ranking into a classification



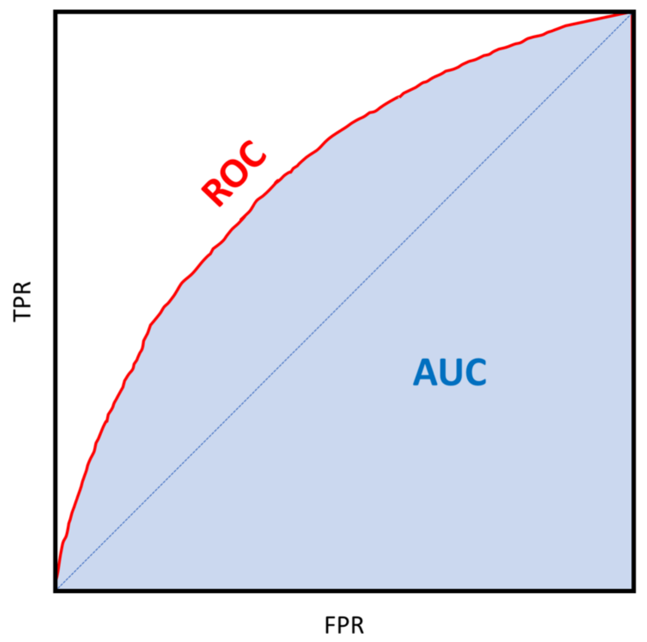
Given a specified cutoff value, one can use the same performance metrics as for the classification models



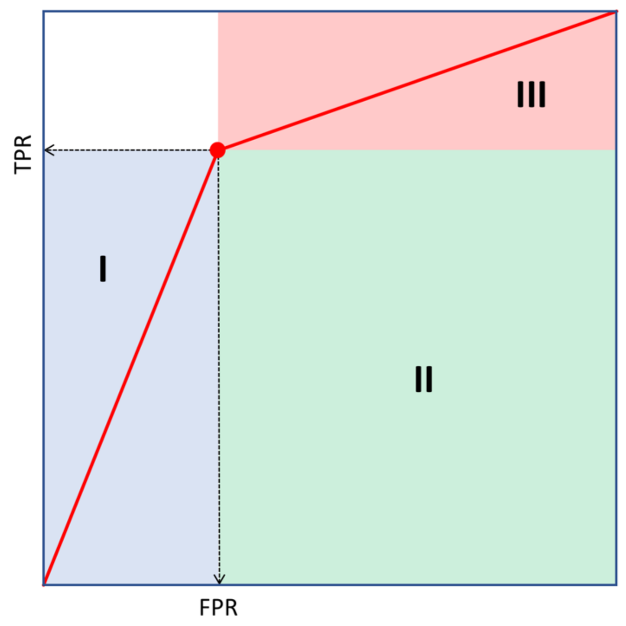
Prediction:	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
Measured:	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0
True positives:	✓	✓		✓											
True negatives:						✓	✓	✓	✓		✓	✓	✓	✓	✓
False positives:			✓		✓										
False negatives:										✓					

The AUC-ROC curve

- Performance metric that is often used for continuous models (but can also be used for classification models)



Continuous model: calculated by varying the applied cutoff value: many TPR-FPR pairs



Classification model: only a single TPR-FPR pair is available

EF and MSE

- Enrichment factor EF: measures by how much the model is able to 'enrich' the number of actives in the predicted set of actives when compared to how many actives there exist in the entire dataset:

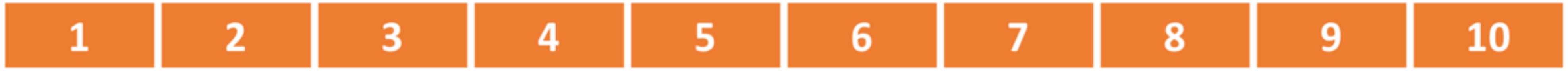
$$EF = \frac{TP(TP + TN + FN + FP)}{(TP + FP)(TP + FN)}$$

- Mean squared error MSE: measures the average squared difference between predictions and true values:

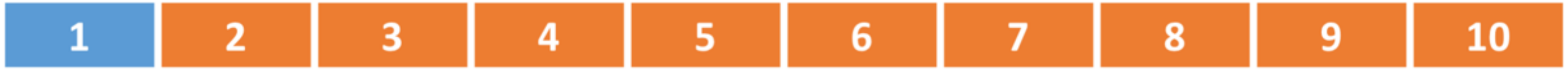
$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

k-fold cross-validation

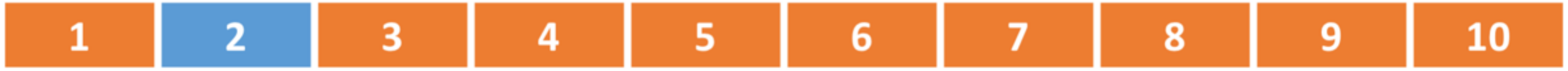
Step 1: Divide the dataset into *k* folds, here *k* is 10



Step 2: Use one fold for validating the model that has been built on all other folds



Step 3: Repeat the model building and validation for each of the data folds (10 times)



Step 4: Calculate the avarege of all of the *k* validation performance values