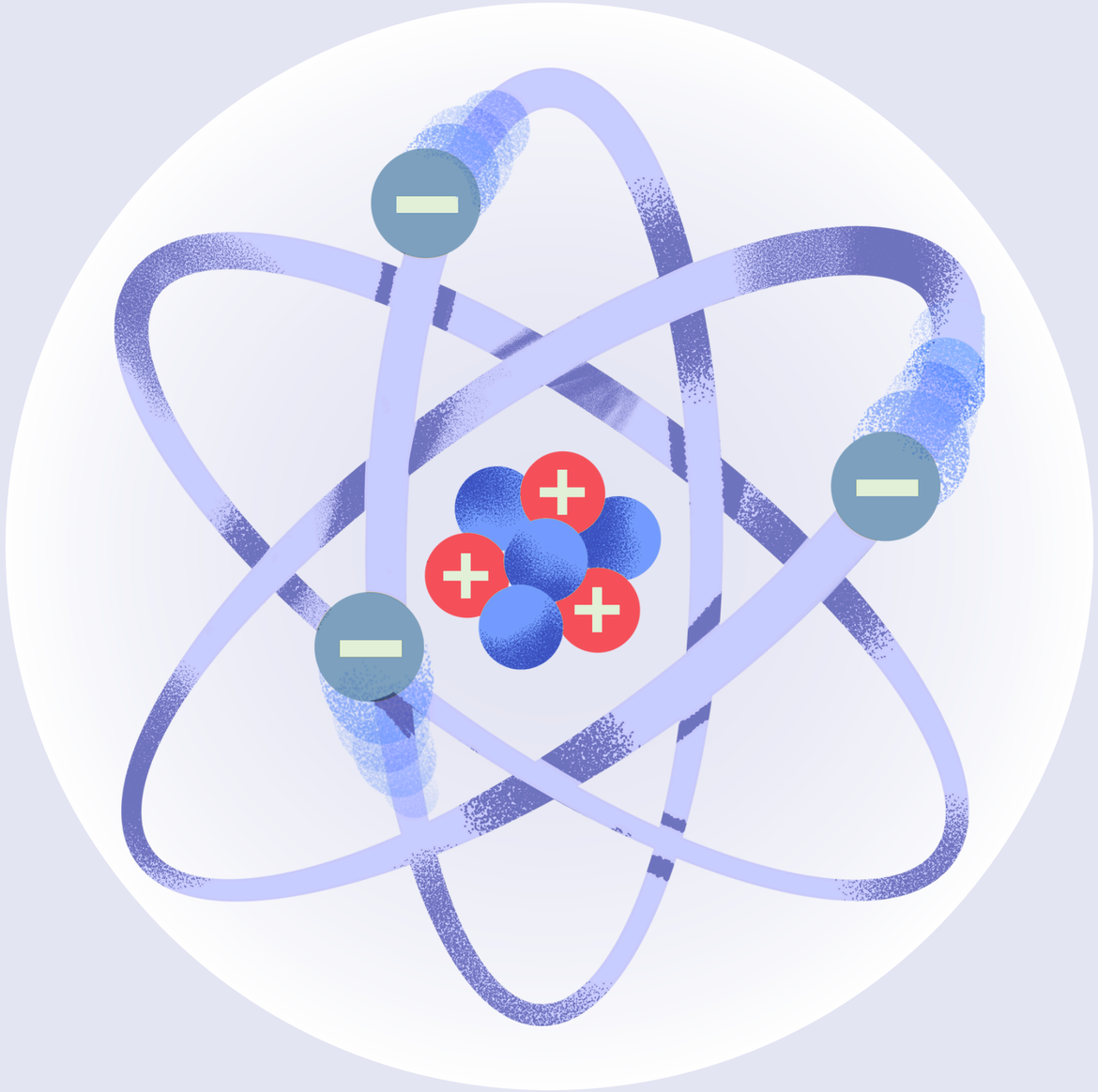
PREDICTING MOLECULAR MUTAGENICITY USING KNN FOR QSPR MODELING

BY SUHANI JAIN



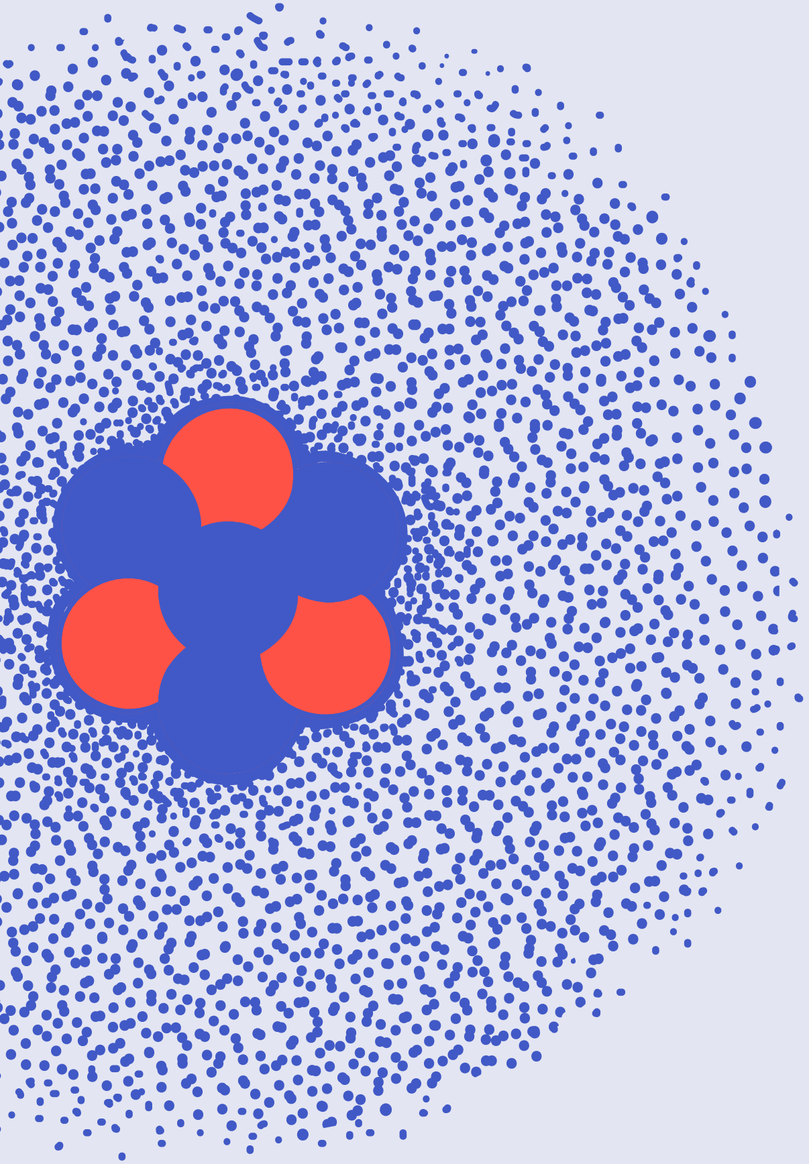


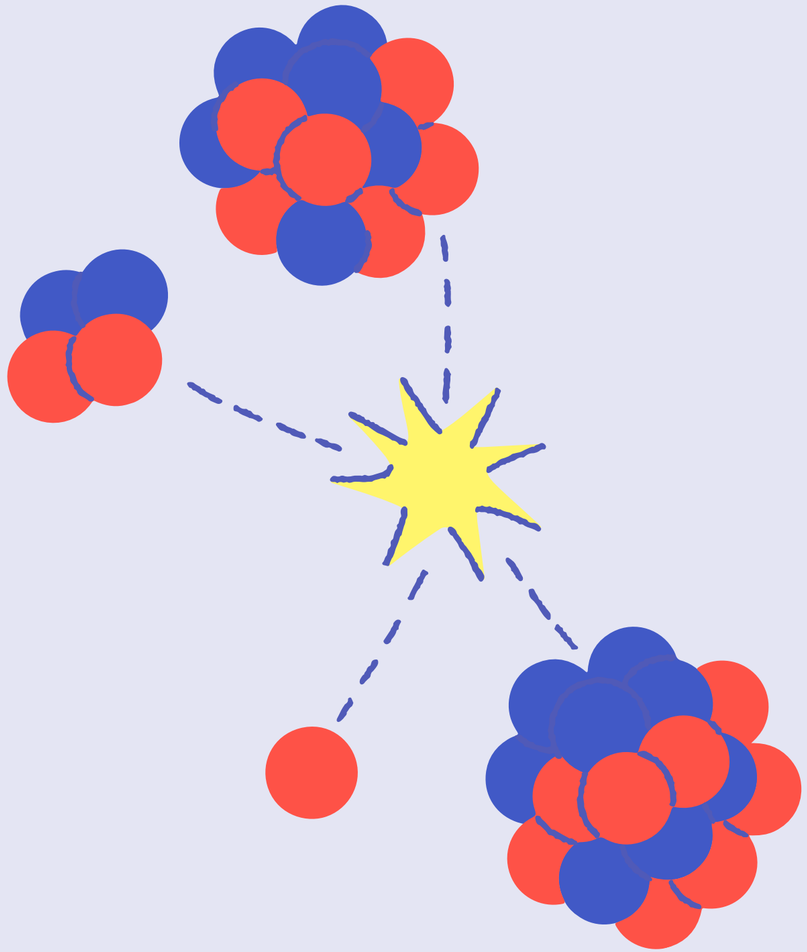
TABLE OF CONTENTS:

WHAT IS MUTAGENICITY?

Mutagenicity refers to the ability of a substance to cause genetic mutations by altering DNA sequences. It is a critical factor in evaluating chemical safety, particularly in drug discovery, environmental science, and industrial applications.

Why is it Important?

- Mutagenic compounds can lead to cancer, birth defects, and hereditary diseases.
- Regulatory agencies like FDA, EPA, and ECHA require mutagenicity testing for new chemicals.
- Machine learning can help predict mutagenicity efficiently, reducing the need for expensive laboratory tests.



INTRODUCTION TO THE PROBLEM

Objective:

To develop a k-Nearest Neighbors (kNN) model to predict whether a molecule is mutagenic (1) or non-mutagenic (0) based on its molecular descriptors.

Why This Matters?

- Mutagenicity is crucial for chemical safety in drug discovery and industrial applications.
- Machine learning models, such as kNN, can help predict mutagenicity based on molecular structure.

DATASET OVERVIEW

Key Features & Columns:

- **MolWt (Molecular Weight):** Represents the total weight of the molecule.
- **Total Polar Surface Area (TPSA):** Measures the polarity of a molecule, affecting its ability to interact with biological systems.
- **BalabanJ Index:** A topological index used to describe molecular structure.
- **Experimental Value (Target Variable):** Binary label indicating if a molecule is mutagenic (1) or non-mutagenic (0).
- **SMILES:** A string representation of the molecular structure (removed for model training).
- **CAS (Chemical Abstracts Service) Registry Number:** A unique identifier for the molecule (removed for analysis).
- **Status & Predicted Value:** Columns that were unnecessary for model building and were dropped.

DATA PREPROCESSING:

- **Checked for missing values.**
- **Dropped unnecessary columns (e.g., Status, CAS, SMILES, etc.).**
- **Split data into 80% training and 20% testing.**
- **Standardized features using StandardScaler.**

```
# check for null values
df.isnull().sum()
```

✓ 0.0s

```
Unnamed: 0              0
Id                      0
CAS                     0
SMILES                  0
Status                  0
Experimental value      0
Predicted value         0
NumValenceElectrons     0
qed                     0
TPSA                    0
MolMR                   0
BalabanJ                0
BertzCT                 0
MolWt                   0
MolLogP                 0
dtype: int64
```

```
# spliting the data (80% train and 20% test)
x_train , x_test , y_train , y_test = train_test_split(x , y , test_size= 0.2 , random_state= 42)
```

✓ 0.0s

```
# normalize the x variable
ss = StandardScaler()
x_train = ss.fit_transform(x_train)
x_test = ss.transform(x_test)
```

✓ 0.0s

EXPLORATORY DATA ANALYSIS (EDA)

Key Insights:

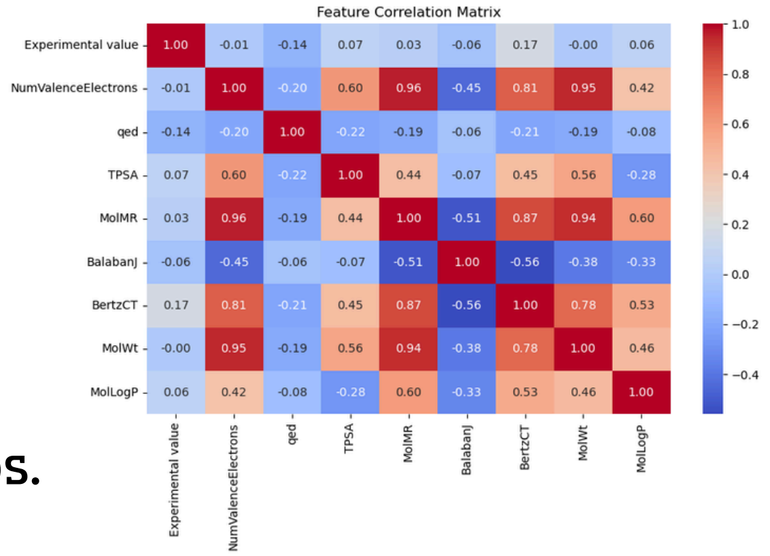
- **Check for Imbalance data:**

  Mutagenic: ~ 56.40%

  Non-mutagenic: ~ 43.60%

- **Correlation Matrix:**

  Heatmap visualization of feature relationships.



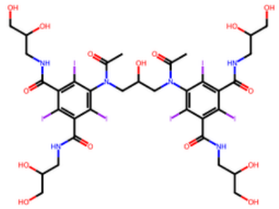
- **Molecular Structure Visualization:**

  Displayed chemical structure of molecules with highest MolWt.

```
# Print the percentage of mutagenic and non-mutagenic molecules
y = df['Experimental value'].to_numpy()
perc_mutegenic = y.sum()/len(y)*100
perc_non_mutegenic = (1-y.sum()/len(y))*100
print(f"Percentage of mutagenic molecules: {perc_mutegenic:.2f}%")
print(f"Percentage of non-mutagenic molecules: {perc_non_mutegenic:.2f}%")
```

✓ 0.0s

```
Percentage of mutagenic molecules: 56.40%
Percentage of non-mutagenic molecules: 43.60%
```



MODEL DEVELOPMENT

K-Nearest Neighbors (KNN) Algorithm:

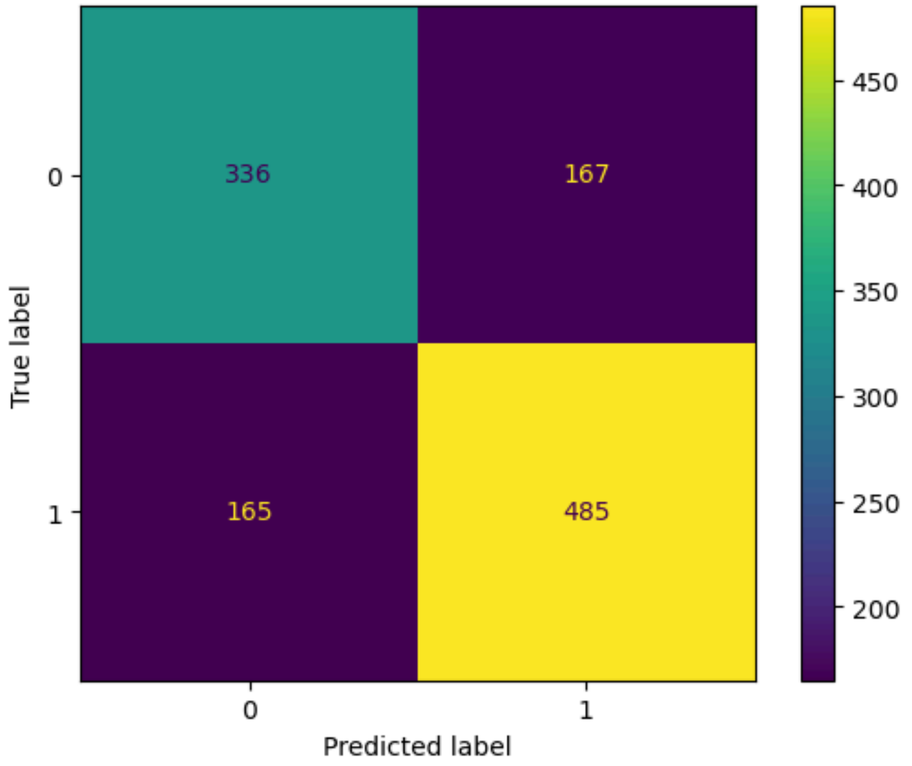
- A simple, non-parametric method that classifies molecules based on their nearest neighbors.

```python
# simple knn model
knn_simple = KNeighborsClassifier()
knn_simple.fit(x_train , y_train)
y_pred = knn_simple.predict(x_test)

# evaluate the simple knn model
print(f"Accuracy : {accuracy_score(y_test , y_pred)}")
print(f"Precision_Score : {precision_score(y_test , y_pred)}")
print(f"F1_Score : {f1_score(y_test , y_pred)}")
print(f"Recall-score : {recall_score(y_test , y_pred)}")
```

✓ 0.4s

```
Accuracy : 0.7120550737207 29
Precision_Score : 0.7438650306748467
F1_Score : 0.7450076804915514
Recall-score : 0.7461538461538462
```



HYPERPARAMETER TUNING

Finding the Best k:

- Used GridSearchCV to test values from 1 to 20.
- Optimized for F1-score (balanced precision & recall).
- Best k = 15.

Testing Different Distance Metrics:

- Euclidean, Manhattan, Minkowski.

Weighting Methods:

- Uniform (equal weighting)
- Distance (closer neighbors weigh more.

```python
# use GridSearchCV to search for the best 'k' based on cross-validation
param_grid = {'n_neighbors': np.arange(1, 21)} # check for 1 to 20
knn = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='f1')  # 5-fold cross-validation, optimize F1-score
knn.fit(x_train, y_train)

# get the best 'k' from GridSearchCV
best_k = knn.best_params_['n_neighbors']
print(f"Best k: {best_k}")

# train the model with best K
final_knn = KNeighborsClassifier(n_neighbors=best_k)
final_knn.fit(x_train, y_train)

# prediction
y_pred = final_knn.predict(x_test)
```

✓ 11.8s

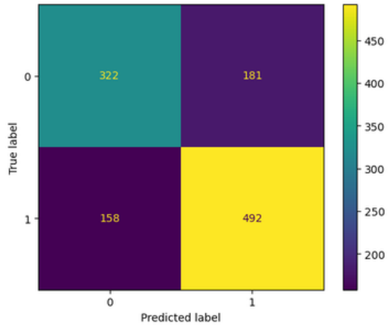
```
Best k: 15
```

```python
# evaluate the model
print(f"Accuracy : {accuracy_score(y_test , y_pred)}")
print(f"Precision_Score : {precision_score(y_test , y_pred)}")
print(f"F1_Score : {f1_score(y_test , y_pred)}")
print(f"Recall-score : {recall_score(y_test , y_pred)}")

# confusion matrix
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm).plot()
plt.show()
```

✓ 1.0s

```
Accuracy : 0.7059843885516045
Precision_Score : 0.7310549777117384
F1_Score : 0.7437641723356009
Recall-score : 0.7569230769230769
```




```python
param_grid = {
    'n_neighbors': np.arange(1, 15),
    'metric': ['euclidean', 'manhattan', 'minkowski'],
    'weights': ['uniform', 'distance']
}

knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='f1', n_jobs=-1)
grid_search.fit(x_train, y_train)

print("Best parameters:", grid_search.best_params_)
```

✓ 12.3s

```
Best parameters: {'metric': 'euclidean', 'n_neighbors': 14, 'weights': 'distance'}
```

```python
param_grid = {
    'n_neighbors': np.arange(1, 15),
    'metric': ['euclidean', 'manhattan', 'minkowski'],
    'weights': ['uniform', 'distance']
}

knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(x_train, y_train)

print("Best parameters:", grid_search.best_params_)
```

✓ 20.5s

```
Best parameters: {'metric': 'manhattan', 'n_neighbors': 12, 'weights': 'distance'}
```

MODEL EVALUATION

METRICS	VALUE
ACCURACY	73.46%
F1 SCORE	76.67%
PRECISION	75.98%
RECALL	77.38%

THANK YOU