Project Summary

This project focuses on predicting chemical toxicity using the Tox21 dataset.
Each molecule is represented by its SMILES string and tested against 12 biological assays related to Nuclear Receptor (NR) and Stress Response (SR) pathways.

The workflow includes:

- Loading and inspecting the Tox21 dataset
- Handling missing assay values and class imbalance
- Generating molecular descriptors and fingerprints from SMILES using RDKit
- Exploring descriptor distributions before and after normalization
- Training multiple machine learning models for each toxicity assay
- Selecting the best model per assay based on cross-validated F1 score

The final outcome is a set of optimized binary classification models that can predict toxic or non-toxic behavior of molecules for individual biological targets.

In [123…

```
# -- General --
import joblib
import os

# -- EDA and Preprocessing --
import pandas as pd
import numpy as np

# -- Visualization --
import seaborn as sns
import matplotlib.pyplot as plt

# -- RDKit --
from rdkit import Chem
from rdkit.Chem import Draw, rdMolDescriptors, Lipinski, Descriptors, Crippen
```

```
from rdkit import DataStructs
from rdkit.Chem.rdFingerprintGenerator import GetMorganGenerator

# -- Machine Learning --
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.base import clone
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression
from lightgbm import LGBMClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline

# -- Preprocessing --
from sklearn.preprocessing import PowerTransformer, StandardScaler

# -- Metrics --
from sklearn.metrics import classification_report, confusion_matrix, f1_score
```

1. Loading Data

In [124…

```
# Loading Data
df_tox = pd.read_csv('../data/tox21.csv')
```

In [125…

```
print(f'The rows in the dataset are {df_tox.shape[0]} and the columns are {df_tox.shape[1]}!\n')
print(f'The columns in the dataset are:')

for no, col in enumerate(df_tox.columns, start=1):
    print(f'{no}. {col}')
```

```
The rows in the dataset are 7831 and the columns are 14!

The columns in the dataset are:
1. NR-AR
2. NR-AR-LBD
3. NR-AhR
4. NR-Aromatase
5. NR-ER
6. NR-ER-LBD
7. NR-PPAR-gamma
8. SR-ARE
9. SR-ATAD5
10. SR-HSE
11. SR-MMP
12. SR-p53
13. mol_id
14. smiles
```

Tox21 Dataset Column Definitions

Nuclear Receptor (NR) Pathways

- **`NR-AR`** : Tests if a chemical affects the Androgen (male hormone) receptor.
- **`NR-AR-LBD`** : Tests if a chemical binds to the active site of the Androgen receptor.
- **`NR-AhR`** : Tests if a chemical acts like toxic environmental pollutants.
- **`NR-Aromatase`** : Tests if a chemical blocks estrogen formation.
- **`NR-ER`** : Tests if a chemical affects the Estrogen (female hormone) receptor.
- **`NR-ER-LBD`** : Tests if a chemical binds to the active site of the Estrogen receptor.
- **`NR-PPAR-gamma`** : Tests if a chemical affects fat and metabolism control.

Stress Response (SR) Pathways

- **`SR-ARE`** : Tests if a chemical causes oxidative (cell) stress.
- **`SR-ATAD5`** : Tests if a chemical damages DNA during replication.
- **`SR-HSE`** : Tests if a chemical causes protein stress or misfolding.
- **`SR-MMP`** : Tests if a chemical damages mitochondria (cell energy).

- `SR-p53` : Tests if a chemical triggers DNA damage response or cell death.

🔎 Key Observations

- The dataset contains 7,831 molecules, each identified by a unique `mol_id` and a corresponding `smiles` string.
- There are 12 toxicity assay columns, covering Nuclear Receptor (NR) and Stress Response (SR) pathways.
- The dataset structure is suitable for multi-target binary classification, where each assay is treated as an independent prediction task.

2. Data Inspection

In [126…
```
df_tox.shape
```

Out[126…
```
(7831, 14)
```

In [127…
```
df_tox.head()
```

Out[127…

	NR-AR	NR-AR-LBD	NR-AhR	NR-Aromatase	NR-ER	NR-ER-LBD	NR-PPAR-gamma	SR-ARE	SR-ATAD5	SR-HSE	SR-MMP	SR-p53	mol_id	
0	0.0	0.0	1.0	NaN	NaN	0.0	0.0	1.0	0.0	0.0	0.0	0.0	TOX3021	CCOc1ccc2nc(S
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0	NaN	0.0	0.0	TOX3020	CCN1C(=O)N
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	NaN	0.0	NaN	NaN	TOX3024	CC[C@]1(O)CC[C@H]2[C@@H]3CCC4=CCCC
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0	NaN	0.0	0.0	TOX3027	CCCN(CC)C(CC)C(=
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	TOX20800	CC(O)(P(=O

In [128…
```
# Reordering columns
cols = df_tox.columns.tolist()
cols = ['mol_id', 'smiles'] + [c for c in cols if c != 'smiles' and c != 'mol_id']
df_tox = df_tox[cols]
```

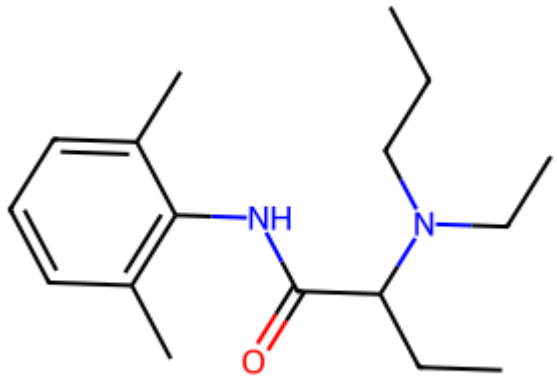
In [129…
```
df_tox.head()
```

Out[129…

	mol_id	smiles	NR-AR	NR-AR-LBD	NR-AhR	NR-Aromatase	NR-ER	NR-ER-LBD	NR-PPAR-gamma	SR-ARE	SR-ATAD5	S H
0	TOX3021	CCOc1ccc2nc(S(N)(=O)=O)sc2c1	0.0	0.0	1.0	NaN	NaN	0.0	0.0	1.0	0.0	
1	TOX3020	CCN1C(=O)NC(c2ccccc2)C1=O	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0	N
2	TOX3024	CC[C@]1(O)CC[C@H]2[C@@H]3CCC4=CCCC[C@@H]4[C@H]...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	NaN	
3	TOX3027	CCCN(CC)C(CC)C(=O)Nc1c(C)cccc1C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0	N
4	TOX20800	CC(O)(P(=O)(O)O)P(=O)(O)O	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

In [130…
```
# Example  of Visualization of a molecule from DataSet
mol = Chem.MolFromSmiles("CCCN(CC)C(CC)C(=O)Nc1c(C)cccc1C")
Draw.MolToImage(mol)
```

Out[130…



In [131…
```
df_tox.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7831 entries, 0 to 7830
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   mol_id         7831 non-null   object 
 1   smiles         7831 non-null   object 
 2   NR-AR          7265 non-null   float64
 3   NR-AR-LBD      6758 non-null   float64
 4   NR-AhR         6549 non-null   float64
 5   NR-Aromatase   5821 non-null   float64
 6   NR-ER          6193 non-null   float64
 7   NR-ER-LBD      6955 non-null   float64
 8   NR-PPAR-gamma  6450 non-null   float64
 9   SR-ARE         5832 non-null   float64
 10  SR-ATAD5       7072 non-null   float64
 11  SR-HSE         6467 non-null   float64
 12  SR-MMP         5810 non-null   float64
 13  SR-p53         6774 non-null   float64
dtypes: float64(12), object(2)
memory usage: 856.6+ KB
```

In [132…
```
df_tox.columns
```

Out[132…
```
Index(['mol_id', 'smiles', 'NR-AR', 'NR-AR-LBD', 'NR-AhR', 'NR-Aromatase',
       'NR-ER', 'NR-ER-LBD', 'NR-PPAR-gamma', 'SR-ARE', 'SR-ATAD5', 'SR-HSE',
       'SR-MMP', 'SR-p53'],
      dtype='object')
```

In [133…
```
target_columns = ['NR-AR', 'NR-AR-LBD', 'NR-AhR', 'NR-Aromatase',
                  'NR-ER', 'NR-ER-LBD', 'NR-PPAR-gamma', 'SR-ARE',
                  'SR-ATAD5', 'SR-HSE', 'SR-MMP', 'SR-p53']

summary = pd.DataFrame({
    "null_count": df_tox[target_columns].isna().sum(),
    "non_null_count": df_tox[target_columns].notna().sum(),
    "null_%": (df_tox[target_columns].isna().mean() * 100).round(2),
```

```
    # class counts (ignore NaNs automatically)
    "count_0": (df_tox[target_columns] == 0).sum(),
    "count_1": (df_tox[target_columns] == 1).sum(),

    # percentage of positive class
    "%_ones": ((df_tox[target_columns] == 1).sum()
               / df_tox[target_columns].notna().sum()
               * 100).round(2)

}).sort_values("null_count", ascending=False)

summary
```

Out[133...

	null_count	non_null_count	null_%	count_0	count_1	%_ones
SR-MMP	2021	5810	25.81	4892	918	15.80
NR-Aromatase	2010	5821	25.67	5521	300	5.15
SR-ARE	1999	5832	25.53	4890	942	16.15
NR-ER	1638	6193	20.92	5400	793	12.80
NR-PPAR-gamma	1381	6450	17.64	6264	186	2.88
SR-HSE	1364	6467	17.42	6095	372	5.75
NR-AhR	1282	6549	16.37	5781	768	11.73
NR-AR-LBD	1073	6758	13.70	6521	237	3.51
SR-p53	1057	6774	13.50	6351	423	6.24
NR-ER-LBD	876	6955	11.19	6605	350	5.03
SR-ATAD5	759	7072	9.69	6808	264	3.73
NR-AR	566	7265	7.23	6956	309	4.25

In [134...

```
df_tox.columns
```

Out[134…
```
Index(['mol_id', 'smiles', 'NR-AR', 'NR-AR-LBD', 'NR-AhR', 'NR-Aromatase',
       'NR-ER', 'NR-ER-LBD', 'NR-PPAR-gamma', 'SR-ARE', 'SR-ATAD5', 'SR-HSE',
       'SR-MMP', 'SR-p53'],
      dtype='object')
```

🔎 Key Observations

- Each assay is a binary label where:
  - `0` indicates non-toxic (inactive)
  - `1` indicates toxic (active)
- All assay columns contain missing values, indicating that not every molecule was tested against every biological target.
- The percentage of missing values varies across assays, with some targets having more than 25% missing data.
- The class distribution is highly imbalanced:
  - Most molecules are labeled as non-toxic (0)
  - Toxic (1) samples are comparatively rare
- This imbalance highlights the need for appropriate evaluation metrics (F1 score) and balanced learning strategies during modeling.

3. Feature Engineering

In [135…
```python
def calculate_descriptors_safe(smiles):
    """
    Calculates multiple molecular descriptors from a SMILES string.
    Returns a pandas Series so it can add multiple columns at once.
    Output schema is IDENTICAL to the original function.
    """
    morgan_gen = GetMorganGenerator(radius=3, fpSize=2048)
    try:
        # 1. Safely create and sanitize molecule
        mol = Chem.MolFromSmiles(smiles, sanitize=True)
        if mol is None:
            raise ValueError("Invalid SMILES")

        # 2. Remove explicit hydrogens (fixes H-without-neighbors warning)
        mol = Chem.RemoveHs(mol)
```

```
        # 3. Calculate descriptors (same as before)
        return pd.Series({
            'tpsa': rdMolDescriptors.CalcTPSA(mol),
            'mw':   Descriptors.MolWt(mol),
            'hbd':  Descriptors.NumHDonors(mol),
            'hba':  Descriptors.NumHAcceptors(mol),
            'rot':  Descriptors.NumRotatableBonds(mol),
            'logp': Crippen.MolLogP(mol),
            'mr':   Crippen.MolMR(mol),
            'hac':  Lipinski.HeavyAtomCount(mol),
            'fp':   morgan_gen.GetFingerprint(mol),  # 🔥 NO deprecation warning
            'ring_count': rdMolDescriptors.CalcNumRings(mol)
        })

    except Exception:
        # 4. Fail safely (same NaN structure as original)
        return pd.Series({
            'tpsa': np.nan,
            'mw': np.nan,
            'hbd': np.nan,
            'hba': np.nan,
            'rot': np.nan,
            'logp': np.nan,
            'mr': np.nan,
            'hac': np.nan,
            'fp': np.nan,
            'ring_count': np.nan
        })

# Apply the function (Creating all columns in one go)
# This returns a new DataFrame with the descriptor columns
descriptor_cols = df_tox['smiles'].apply(calculate_descriptors_safe)

# Join these new columns back to your original DataFrame
df_tox = pd.concat([df_tox, descriptor_cols], axis=1)

# Check for failures (just checking one column is enough since they all fail together)
print(f"Failed rows: {df_tox['tpsa'].isna().sum()}")
```

```
[15:16:10] WARNING: not removing hydrogen atom without neighbors
[15:16:10] WARNING: not removing hydrogen atom without neighbors
[15:16:11] Explicit valence for atom # 8 Al, 6, is greater than permitted
[15:16:12] Explicit valence for atom # 3 Al, 6, is greater than permitted
[15:16:12] Explicit valence for atom # 4 Al, 6, is greater than permitted
[15:16:13] Explicit valence for atom # 4 Al, 6, is greater than permitted
[15:16:14] Explicit valence for atom # 9 Al, 6, is greater than permitted
[15:16:15] Explicit valence for atom # 5 Al, 6, is greater than permitted
[15:16:16] Explicit valence for atom # 16 Al, 6, is greater than permitted
[15:16:17] Explicit valence for atom # 20 Al, 6, is greater than permitted
Failed rows: 8
```

In [136…

```
# Filter for rows where TPSA is NaN (empty)
invalid_rows = df_tox[df_tox['hbd'].isna()].index

# Display just the SMILES strings that failed
print("Invalid SMILES strings:")
print(invalid_rows)

df_tox = df_tox.drop(index=invalid_rows).reset_index(drop=True)
```

```
Invalid SMILES strings:
Index([1322, 2290, 2297, 3558, 4565, 4649, 5538, 6723], dtype='int64')
```

In [137…

```
# Filter for rows where TPSA is NaN (empty)
invalid_rows = df_tox[df_tox['hbd'].isna()].index

# Display just the SMILES strings that failed
print("Invalid SMILES strings:")
print(invalid_rows)
```

```
Invalid SMILES strings:
Index([], dtype='int64')
```

In [138…

```
df_tox.shape
```

Out[138… (7823, 24)

In [139…

```
df_tox.head(1)
```

Out[139...

	mol_id	smiles	NR-AR	NR-AR-LBD	NR-AhR	NR-Aromatase	NR-ER	NR-ER-LBD	NR-PPAR-gamma	SR-ARE	...	tpsa	mw	hbd	hba	rot	logp	mr
0	TOX3021	CCOc1ccc2nc(S(N)(=O)=O)sc2c1	0.0	0.0	1.0	NaN	NaN	0.0	0.0	1.0	...	82.28	258.324	1.0	5.0	3.0	1.3424	62.1622

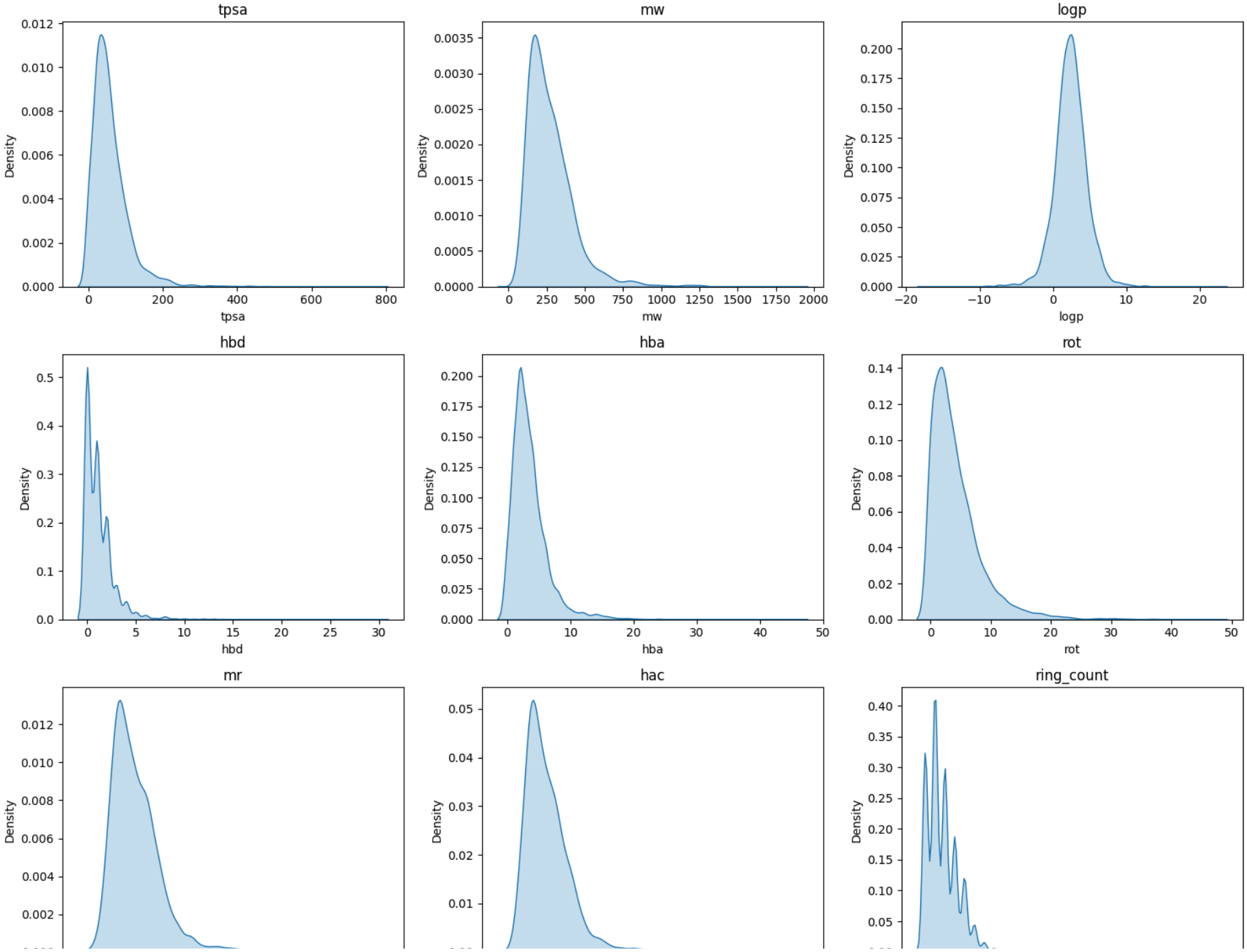
1 rows × 24 columns

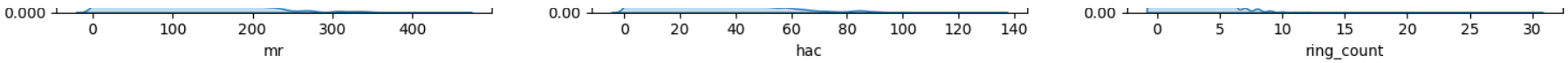
🔎 Key Observations

- SMILES strings were converted into numerical features because machine learning models cannot directly interpret chemical strings.
- Molecular descriptors such as TPSA, molecular weight, hydrogen bond counts, and rotatable bonds capture physicochemical properties of molecules.
- Morgan fingerprints (2048 bits) encode structural sub-patterns and substructures present in each molecule.
- A small number of invalid SMILES failed RDKit processing and were safely removed to maintain data quality.
- The combination of scalar descriptors and fingerprints provides both global chemical properties and detailed structural information for toxicity prediction.

4. Descriptor Distribution Analysis

In [ ]:
```
cols = ['tpsa', 'mw', 'logp', 'hbd', 'hba', 'rot', 'mr', 'hac', 'ring_count']

fig, axes = plt.subplots(3, 3, figsize=(15, 12))
axes = axes.flatten()

for i, col in enumerate(cols):
    sns.kdeplot(data=df_tox, x=col, fill=True, ax=axes[i])
    axes[i].set_title(col)

# Remove empty subplots if cols < 9
for j in range(len(cols), 9):
    fig.delaxes(axes[j])

plt.tight_layout()
# plt.savefig(r'..\plots\1.Descriptor_without_normalization.png')
plt.show()
```


tpsa
Density
tpsa
mw
Density
mw
logp
Density
logp
hbd
Density
hbd
hba
Density
hba
rot
Density
rot
mr
Density
hac
Density
ring_count
Density

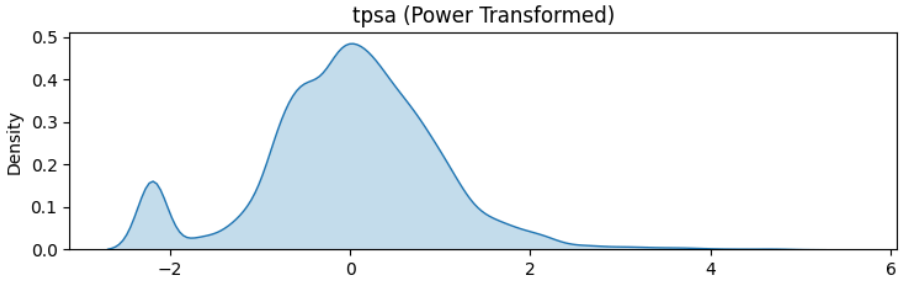


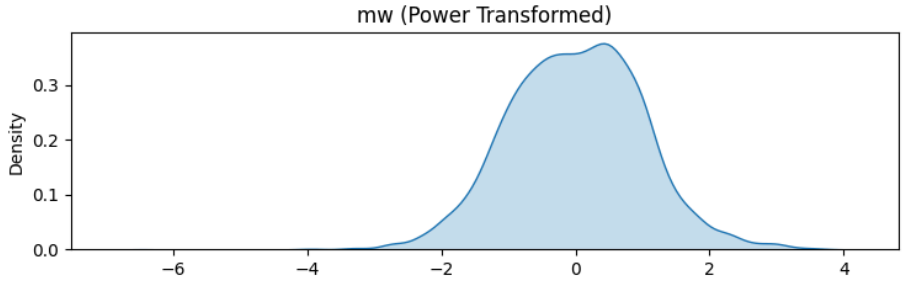
🔎 Key Observations

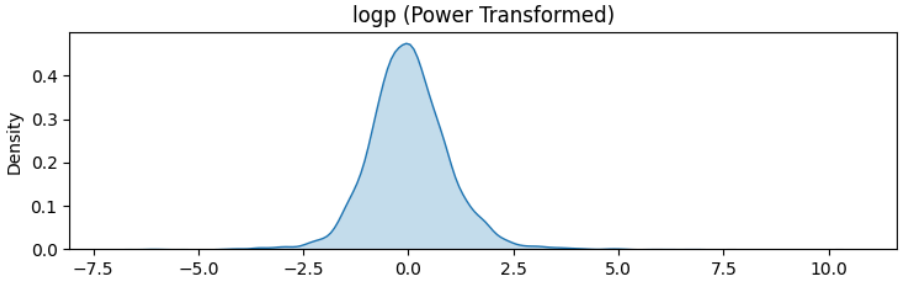
- Most molecular descriptors show right-skewed distributions, which is common for chemical property data.
- Properties such as molecular weight, TPSA, and rotatable bonds span a wide range of values.
- The presence of skewness suggests that direct use of raw values may negatively impact certain machine learning models.
- Visual inspection confirms the need for normalization to stabilize variance and improve model learning.

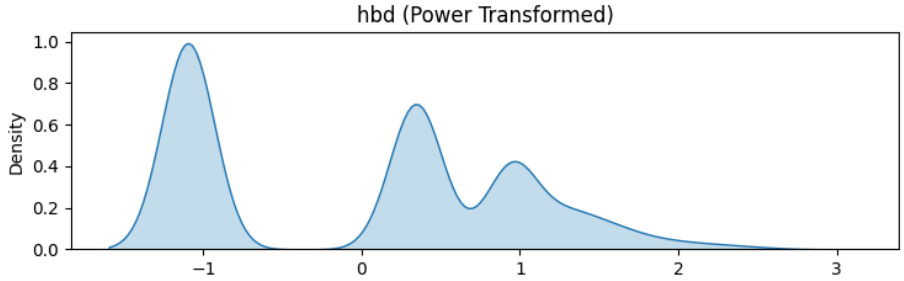
5. Power Transformation & Normalization

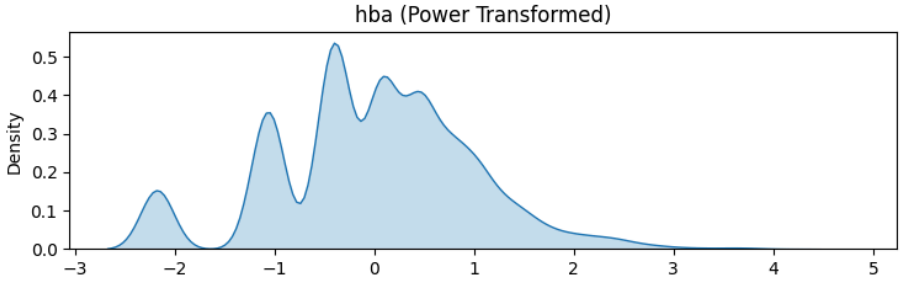
In [ ]:
```
n_rows = (len(cols) + 1) // 2
n_cols = 2

fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, 12))
axes = axes.flatten()

for i, col in enumerate(cols):
    pt = PowerTransformer()
    transformed = pt.fit_transform(df_tox[[col]]).ravel()

    sns.kdeplot(x=transformed, fill=True, ax=axes[i])
    axes[i].set_title(f"{col} (Power Transformed)")

# remove unused axes
for j in range(len(cols), len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout()
# plt.savefig(r'..\plots\2.Descriptor_with_normalization.png')
plt.show()
```

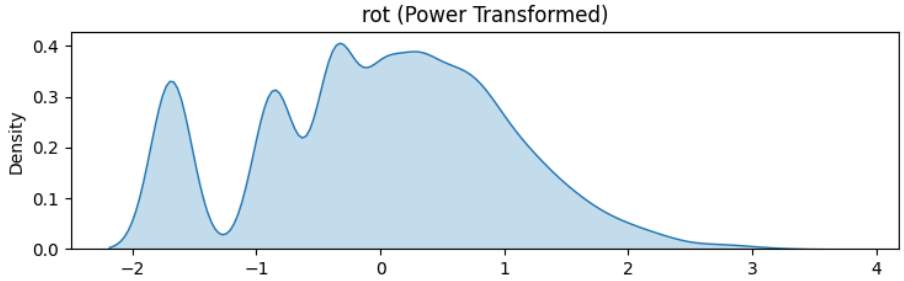

tpsa (Power Transformed)
Density

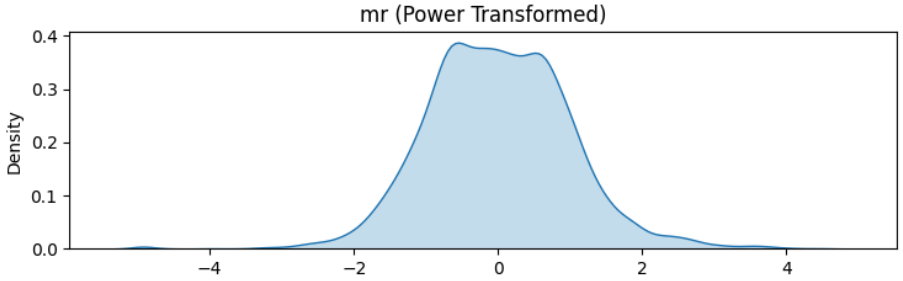

mw (Power Transformed)
Density

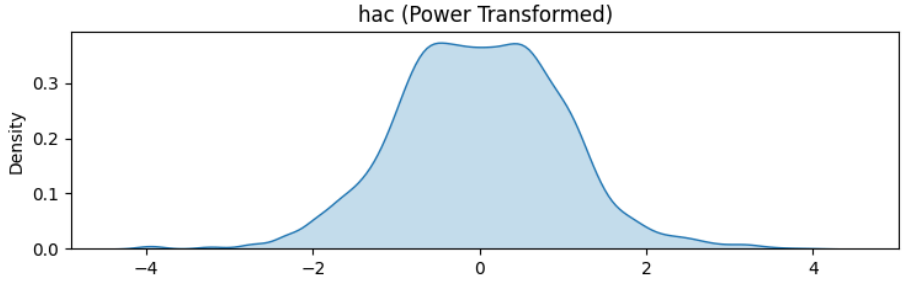

logp (Power Transformed)
Density

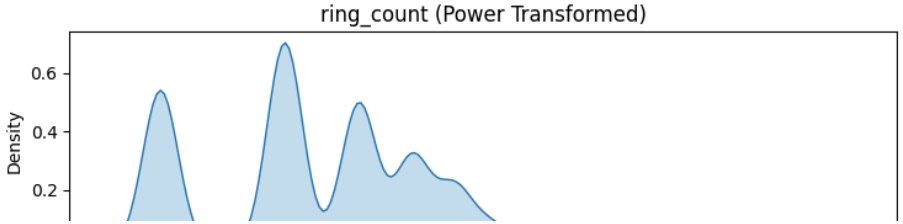

hbd (Power Transformed)
Density


hba (Power Transformed)
Density


rot (Power Transformed)
Density


mr (Power Transformed)
Density


hac (Power Transformed)
Density


ring_count (Power Transformed)
Density

In [142…
```python
def normalize(df):
    pt = PowerTransformer(method="yeo-johnson")
    to_normal = ['hac', 'mw', 'mr', 'hba', 'tpsa', 'rot']
    df[to_normal] = pt.fit_transform(df[to_normal])
    return df

df_tox_normalize = normalize(df_tox.copy())
```

In [143…
```python
# Converting fingerprint to numpy array
def to_array(fp):
    arr = np.zeros(fp.GetNumBits(), dtype=np.uint8)
    DataStructs.ConvertToNumpyArray(fp, arr)
    return arr

fp_array = df_tox['fp'].apply(to_array)

df_tox['fp'] = fp_array
df_tox_normalize['fp'] = fp_array
```

🔎 Key Observations

- PowerTransformer (Yeo-Johnson) normalization was applied to selected molecular descriptors.
- After transformation, feature distributions become more symmetric and closer to a normal shape.
- Normalization is especially beneficial for distance-based and linear models such as Logistic Regression and KNN.
- Both normalized and non-normalized feature sets were evaluated to compare model performance.

6. Model Training & Evaluation

In [145…
```python
# Tox21 binary classification targets
targets = ['NR-AR', 'NR-AR-LBD', 'NR-AhR', 'NR-Aromatase',
           'NR-ER', 'NR-ER-LBD', 'NR-PPAR-gamma',
           'SR-ARE', 'SR-ATAD5', 'SR-HSE', 'SR-MMP', 'SR-p53'
```

```
            ]

# Scalar molecular descriptors
scalar_cols = ['tpsa', 'mw', 'hbd', 'hba', 'rot',
               'logp', 'mr', 'hac', 'ring_count'
                ]
```

In [146…

```
models_config = {
    "LR": make_pipeline(StandardScaler(), LogisticRegression(solver="lbfgs", class_weight="balanced", max_iter=500, n_jobs=-1)
    "RF": RandomForestClassifier(n_estimators=300, n_jobs=-1, random_state=24),
    "KNN": make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=5)),
    "XGB": XGBClassifier(n_estimators=300, max_depth=6, learning_rate=0.05, subsample=0.8, colsample_bytree=0.8, eval_metric="
    "LGBM_Bal": LGBMClassifier(class_weight="balanced", verbosity=-1, random_state=24),
    "LGBM_Unbal": LGBMClassifier(is_unbalance=True, verbosity=-1, random_state=24)
}
```

In [147…

```
def cross_val_f1(model, X, y, n_splits=5):
    """
    Returns mean F1 score for class 1 using Stratified K-Fold CV
    """
    skf = StratifiedKFold(
        n_splits=n_splits,
        shuffle=True,
        random_state=24
    )

    f1_scores = []

    for train_idx, val_idx in skf.split(X, y):
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        model_fold = clone(model)
        model_fold.fit(X_train, y_train)

        y_pred = model_fold.predict(X_val)
        f1 = f1_score(y_val, y_pred, pos_label=1, zero_division=0)

        f1_scores.append(f1)
```

```
    return np.mean(f1_scores)
```

In [ ]:

```
# Running ML models without normalizing the data

results = {t: {} for t in targets}
best_models_store = {}

for target in targets:
    print(f"\n>>> Target: {target}")

    # Remove missing labels
    df_sub = df_tox[df_tox[target].notna()]
    y = df_sub[target].values

    # Combine scalar + fingerprint features
    X_scalar = df_sub[scalar_cols].values
    X_fp = np.stack(df_sub["fp"].values)
    X = np.hstack([X_scalar, X_fp])

    best_f1 = -1
    best_model = None
    best_model_name = None

    for name, model in models_config.items():
        mean_f1 = cross_val_f1(model, X, y)
        results[target][name] = mean_f1

        print(f"{name} | CV F1: {mean_f1:.4f}")

        if mean_f1 > best_f1:
            best_f1 = mean_f1
            best_model = clone(model)
            best_model_name = name

    # Refit best model on full dataset
    best_model.fit(X, y)

    print(f"✅ Best model: {best_model_name} | F1: {best_f1:.4f}")
```

```
>>> Target: NR-AR
LR | CV F1: 0.4098
RF | CV F1: 0.5975
KNN | CV F1: 0.4695
XGB | CV F1: 0.5961
LGBM_Bal | CV F1: 0.5271
LGBM_Unbal | CV F1: 0.5222
✅ Best model: RF | F1: 0.5975

>>> Target: NR-AR-LBD
LR | CV F1: 0.5493
RF | CV F1: 0.6275
KNN | CV F1: 0.4682
XGB | CV F1: 0.6561
LGBM_Bal | CV F1: 0.6616
LGBM_Unbal | CV F1: 0.6603
✅ Best model: LGBM_Bal | F1: 0.6616

>>> Target: NR-AhR
LR | CV F1: 0.4433
RF | CV F1: 0.4016
KNN | CV F1: 0.1645
XGB | CV F1: 0.5233
LGBM_Bal | CV F1: 0.5671
LGBM_Unbal | CV F1: 0.5680
✅ Best model: LGBM_Unbal | F1: 0.5680

>>> Target: NR-Aromatase
LR | CV F1: 0.3208
RF | CV F1: 0.3089
KNN | CV F1: 0.1259
XGB | CV F1: 0.4004
LGBM_Bal | CV F1: 0.4067
LGBM_Unbal | CV F1: 0.3776
✅ Best model: LGBM_Bal | F1: 0.4067

>>> Target: NR-ER
LR | CV F1: 0.2763
RF | CV F1: 0.3227
KNN | CV F1: 0.1918
XGB | CV F1: 0.3718
```

```
LGBM_Bal | CV F1: 0.4062
LGBM_Unbal | CV F1: 0.4115
✅ Best model: LGBM_Unbal | F1: 0.4115

>>> Target: NR-ER-LBD
LR | CV F1: 0.3247
RF | CV F1: 0.3878
KNN | CV F1: 0.2528
XGB | CV F1: 0.4791
LGBM_Bal | CV F1: 0.4989
LGBM_Unbal | CV F1: 0.4851
✅ Best model: LGBM_Bal | F1: 0.4989

>>> Target: NR-PPAR-gamma
LR | CV F1: 0.2638
RF | CV F1: 0.0418
KNN | CV F1: 0.0974
XGB | CV F1: 0.2166
LGBM_Bal | CV F1: 0.3314
LGBM_Unbal | CV F1: 0.3050
✅ Best model: LGBM_Bal | F1: 0.3314

>>> Target: SR-ARE
LR | CV F1: 0.3551
RF | CV F1: 0.2924
KNN | CV F1: 0.1178
XGB | CV F1: 0.3996
LGBM_Bal | CV F1: 0.5068
LGBM_Unbal | CV F1: 0.5155
✅ Best model: LGBM_Unbal | F1: 0.5155

>>> Target: SR-ATAD5
LR | CV F1: 0.3079
RF | CV F1: 0.0984
KNN | CV F1: 0.1303
XGB | CV F1: 0.1894
LGBM_Bal | CV F1: 0.3564
LGBM_Unbal | CV F1: 0.3643
✅ Best model: LGBM_Unbal | F1: 0.3643

>>> Target: SR-HSE
```

```
LR | CV F1: 0.2429
RF | CV F1: 0.0982
KNN | CV F1: 0.1038
XGB | CV F1: 0.2195
LGBM_Bal | CV F1: 0.3467
LGBM_Unbal | CV F1: 0.3355
✅ Best model: LGBM_Bal | F1: 0.3467

>>> Target: SR-MMP
LR | CV F1: 0.5256
RF | CV F1: 0.5224
KNN | CV F1: 0.1806
XGB | CV F1: 0.6447
LGBM_Bal | CV F1: 0.6716
LGBM_Unbal | CV F1: 0.6642
✅ Best model: LGBM_Bal | F1: 0.6716

>>> Target: SR-p53
LR | CV F1: 0.2727
RF | CV F1: 0.1544
KNN | CV F1: 0.0765
XGB | CV F1: 0.2992
LGBM_Bal | CV F1: 0.3930
LGBM_Unbal | CV F1: 0.3771
✅ Best model: LGBM_Bal | F1: 0.3930
```

In [ ]:
```
MODEL_DIR = "../models"
os.makedirs(MODEL_DIR, exist_ok=True)
```

In [150…
```
results_normalize = {t: {} for t in targets}
best_models_store_normalize = {}

for target in targets:
    print(f"\n>>> Target: {target}")

    # Remove missing labels
    df_sub = df_tox_normalize[df_tox_normalize[target].notna()]
    y = df_sub[target].values

    # Combine scalar + fingerprint features
```

```
    X_scalar = df_sub[scalar_cols].values
    X_fp = np.stack(df_sub["fp"].values)
    X = np.hstack([X_scalar, X_fp])

    best_f1 = -1
    best_model = None
    best_model_name = None

    for name, model in models_config.items():
        mean_f1 = cross_val_f1(model, X, y)
        results_normalize[target][name] = mean_f1

        print(f"{name} | CV F1: {mean_f1:.4f}")

        if mean_f1 > best_f1:
            best_f1 = mean_f1
            best_model = clone(model)
            best_model_name = name

    # Refit best model on full dataset
    best_model.fit(X, y)

    # Save model to disk
    model_path = f"{MODEL_DIR}/{target}_best_model.joblib"
    joblib.dump(best_model, model_path)

    best_models_store_normalize[target] = {
        "model_name": best_model_name,
        "model_path": model_path,
        "best_cv_f1": best_f1
    }

    print(f"💾 Saved model to: {model_path}")
    print(f"✅ Best model: {best_model_name} | F1: {best_f1:.4f}")
```

```
>>> Target: NR-AR
```

```
LR | CV F1: 0.4075
RF | CV F1: 0.5975
KNN | CV F1: 0.4695
XGB | CV F1: 0.5961
LGBM_Bal | CV F1: 0.5302
LGBM_Unbal | CV F1: 0.5253
💾 Saved model to: ../models/NR-AR_best_model.joblib
✅ Best model: RF | F1: 0.5975

>>> Target: NR-AR-LBD
LR | CV F1: 0.5634
RF | CV F1: 0.6275
KNN | CV F1: 0.4771
XGB | CV F1: 0.6561
LGBM_Bal | CV F1: 0.6666
LGBM_Unbal | CV F1: 0.6628
💾 Saved model to: ../models/NR-AR-LBD_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.6666

>>> Target: NR-AhR
LR | CV F1: 0.4414
RF | CV F1: 0.4030
KNN | CV F1: 0.1551
XGB | CV F1: 0.5233
LGBM_Bal | CV F1: 0.5741
LGBM_Unbal | CV F1: 0.5643
💾 Saved model to: ../models/NR-AhR_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.5741

>>> Target: NR-Aromatase
LR | CV F1: 0.3201
RF | CV F1: 0.3040
KNN | CV F1: 0.1259
XGB | CV F1: 0.4004
LGBM_Bal | CV F1: 0.4061
LGBM_Unbal | CV F1: 0.3908
💾 Saved model to: ../models/NR-Aromatase_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.4061

>>> Target: NR-ER
LR | CV F1: 0.2763
```

```
RF | CV F1: 0.3231
KNN | CV F1: 0.2033
XGB | CV F1: 0.3718
LGBM_Bal | CV F1: 0.4098
LGBM_Unbal | CV F1: 0.3974
💾 Saved model to: ../models/NR-ER_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.4098

>>> Target: NR-ER-LBD
LR | CV F1: 0.3316
RF | CV F1: 0.3878
KNN | CV F1: 0.2611
XGB | CV F1: 0.4791
LGBM_Bal | CV F1: 0.4970
LGBM_Unbal | CV F1: 0.4767
💾 Saved model to: ../models/NR-ER-LBD_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.4970

>>> Target: NR-PPAR-gamma
LR | CV F1: 0.2786
RF | CV F1: 0.0316
KNN | CV F1: 0.0974
XGB | CV F1: 0.2166
LGBM_Bal | CV F1: 0.3353
LGBM_Unbal | CV F1: 0.3007
💾 Saved model to: ../models/NR-PPAR-gamma_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.3353

>>> Target: SR-ARE
LR | CV F1: 0.3570
RF | CV F1: 0.2926
KNN | CV F1: 0.1247
XGB | CV F1: 0.3996
LGBM_Bal | CV F1: 0.5170
LGBM_Unbal | CV F1: 0.4998
💾 Saved model to: ../models/SR-ARE_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.5170

>>> Target: SR-ATAD5
LR | CV F1: 0.3125
RF | CV F1: 0.0984
```

```
KNN | CV F1: 0.1179
XGB | CV F1: 0.1894
LGBM_Bal | CV F1: 0.3806
LGBM_Unbal | CV F1: 0.3390
💾 Saved model to: ../models/SR-ATAD5_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.3806

>>> Target: SR-HSE
LR | CV F1: 0.2373
RF | CV F1: 0.0985
KNN | CV F1: 0.0804
XGB | CV F1: 0.2195
LGBM_Bal | CV F1: 0.3242
LGBM_Unbal | CV F1: 0.3343
💾 Saved model to: ../models/SR-HSE_best_model.joblib
✅ Best model: LGBM_Unbal | F1: 0.3343

>>> Target: SR-MMP
LR | CV F1: 0.5309
RF | CV F1: 0.5236
KNN | CV F1: 0.1892
XGB | CV F1: 0.6447
LGBM_Bal | CV F1: 0.6667
LGBM_Unbal | CV F1: 0.6687
💾 Saved model to: ../models/SR-MMP_best_model.joblib
✅ Best model: LGBM_Unbal | F1: 0.6687

>>> Target: SR-p53
LR | CV F1: 0.2851
RF | CV F1: 0.1544
KNN | CV F1: 0.0763
XGB | CV F1: 0.2992
LGBM_Bal | CV F1: 0.4021
LGBM_Unbal | CV F1: 0.3674
💾 Saved model to: ../models/SR-p53_best_model.joblib
✅ Best model: LGBM_Bal | F1: 0.4021
```

In [155…

```
results_df = pd.DataFrame(results).T

numeric_cols = results_df.select_dtypes(include="number")
```

```
results_df["Best_Model"] = numeric_cols.idxmax(axis=1)
results_df["Best_CV_F1"] = numeric_cols.max(axis=1)

results_df.to_csv("results_without_normalize.csv", index=False)
display(results_df.sort_values("Best_CV_F1", ascending=False))
```

	LR	RF	KNN	XGB	LGBM_Bal	LGBM_Unbal	Best_Model	Best_CV_F1
SR-MMP	0.525636	0.522369	0.180621	0.644671	0.671556	0.664200	LGBM_Bal	0.671556
NR-AR-LBD	0.549349	0.627506	0.468190	0.656110	0.661583	0.660259	LGBM_Bal	0.661583
NR-AR	0.409784	0.597478	0.469451	0.596067	0.527136	0.522190	RF	0.597478
NR-AhR	0.443330	0.401628	0.164511	0.523340	0.567128	0.568015	LGBM_Unbal	0.568015
SR-ARE	0.355140	0.292361	0.117791	0.399605	0.506830	0.515540	LGBM_Unbal	0.515540
NR-ER-LBD	0.324697	0.387830	0.252772	0.479083	0.498854	0.485081	LGBM_Bal	0.498854
NR-ER	0.276254	0.322696	0.191848	0.371827	0.406170	0.411484	LGBM_Unbal	0.411484
NR-Aromatase	0.320820	0.308870	0.125869	0.400427	0.406678	0.377553	LGBM_Bal	0.406678
SR-p53	0.272741	0.154439	0.076511	0.299235	0.393049	0.377095	LGBM_Bal	0.393049
SR-ATAD5	0.307950	0.098393	0.130313	0.189361	0.356411	0.364271	LGBM_Unbal	0.364271
SR-HSE	0.242853	0.098244	0.103843	0.219453	0.346691	0.335546	LGBM_Bal	0.346691
NR-PPAR-gamma	0.263779	0.041835	0.097432	0.216648	0.331361	0.305023	LGBM_Bal	0.331361

In [156…

```
results_df_normalize = pd.DataFrame(results_normalize).T

numeric_cols = results_df_normalize.select_dtypes(include="number")

results_df_normalize["Best_Model"] = numeric_cols.idxmax(axis=1)
results_df_normalize["Best_CV_F1"] = numeric_cols.max(axis=1)

results_df_normalize.to_csv("results_with_normalize.csv", index=False)
display(results_df_normalize.sort_values("Best_CV_F1", ascending=False))
```

	LR	RF	KNN	XGB	LGBM_Bal	LGBM_Unbal	Best_Model	Best_CV_F1
SR-MMP	0.530936	0.523626	0.189183	0.644671	0.666689	0.668658	LGBM_Unbal	0.668658
NR-AR-LBD	0.563444	0.627506	0.477069	0.656110	0.666598	0.662834	LGBM_Bal	0.666598
NR-AR	0.407459	0.597478	0.469451	0.596067	0.530213	0.525340	RF	0.597478
NR-AhR	0.441444	0.402950	0.155087	0.523340	0.574051	0.564316	LGBM_Bal	0.574051
SR-ARE	0.357019	0.292604	0.124729	0.399605	0.517001	0.499784	LGBM_Bal	0.517001
NR-ER-LBD	0.331597	0.387830	0.261134	0.479083	0.497031	0.476729	LGBM_Bal	0.497031
NR-ER	0.276285	0.323051	0.203333	0.371827	0.409765	0.397445	LGBM_Bal	0.409765
NR-Aromatase	0.320114	0.303960	0.125869	0.400427	0.406057	0.390839	LGBM_Bal	0.406057
SR-p53	0.285116	0.154439	0.076316	0.299235	0.402117	0.367426	LGBM_Bal	0.402117
SR-ATAD5	0.312545	0.098393	0.117886	0.189361	0.380610	0.339005	LGBM_Bal	0.380610
NR-PPAR-gamma	0.278572	0.031579	0.097432	0.216648	0.335271	0.300735	LGBM_Bal	0.335271
SR-HSE	0.237291	0.098491	0.080366	0.219453	0.324187	0.334283	LGBM_Unbal	0.334283

🔎 Key Observations

- Each toxicity assay was treated as an independent binary classification problem.
- Multiple machine learning models were evaluated, including Logistic Regression, Random Forest, KNN, XGBoost, and LightGBM.
- Stratified K-Fold cross-validation was used to handle class imbalance during model evaluation.
- F1 score was selected as the primary metric to balance precision and recall for the minority (toxic) class.
- LightGBM (balanced or unbalanced) and Random Forest models consistently showed the best performance across most assays.
- For each assay, the model with the highest cross-validated F1 score was selected as the final best model.
- Normalized molecular descriptors generally improved or maintained model performance compared to raw features.
- The selected best models were retrained on the full available dataset for each assay.
- Final trained models were saved to disk for future prediction, reuse, or deployment workflows.

Project Conclusion

- In this project, a complete machine learning workflow to predict chemical toxicity using the `Tox21 dataset`. Starting from raw `SMILES` strings, molecular structures were converted into meaningful numerical representations using physicochemical descriptors and structural fingerprints. This allowed the models to capture both global chemical properties and detailed substructural information.
- Due to missing labels and strong class imbalance across toxicity assays, each biological target was modeled as an independent binary classification task and evaluated using stratified cross-validation. The `F1 score` was used as the primary metric to ensure balanced performance on the `minority toxic class`.
- Multiple machine learning algorithms were compared, and ensemble-based models, particularly `LightGBM` and `Random Forest`, consistently achieved the best performance across most assays. Feature normalization generally improved model stability without degrading performance. The final optimized models were retrained on all available data and saved for downstream reuse.

Overall, this study demonstrates that combining chemically meaningful descriptors with robust machine learning techniques provides an effective and scalable approach for in silico toxicity screening, supporting early-stage chemical safety assessment and drug discovery workflows.