# PharmApp Suite

## 🧠 AI for Drug Discovery and Development 🧪

Copyright 2025 | RnD Pharma Plus | www.nghiencuuthuoc.com | Zalo: +84888999311 |

## 🧩 Topic: Oncology-focused drug trend in ChEMBL

Okay, I understand. You’re working on a ChEMBL 35 data analysis project using RDKit and need guidance on how to structure your code, run queries, handle potential errors, and create examples, following an AIMLops template and addressing specific SQL and Python issues. Let’s break this down step by step.

**1. Analysis of the Analysis Model**

Given that you’re using ChEMBL data and RDKit, the overall goal is likely to explore relationships between chemical structures and biological activity. Here’s a general model framework:

* **Data Extraction & Preparation (SQL):** Retrieve relevant data from ChEMBL, focusing on activity data (IC50, Ki, etc.) and associated chemical structures (SMILES). Clean the data, handle missing values, and standardize units. Limiting to 100 rows for initial development is smart.
* **Feature Engineering (Python/RDKit):** Use RDKit to generate molecular descriptors (e.g., molecular weight, LogP, number of hydrogen bond donors/acceptors, topological polar surface area (TPSA), etc.) from the SMILES strings. These descriptors become the features for your analysis.
* **Exploratory Data Analysis (EDA) (Python):** Explore the data, looking for correlations between descriptors and activity values. This can involve scatter plots, histograms, and other visualizations.
* **Modeling (Python/Scikit-learn):** Build a predictive model to relate the molecular descriptors to the activity values. Common choices include:
  + **Linear Regression:** Simple, but can be a good starting point.
  + **Random Forest:** Robust and can capture non-linear relationships.
  + **Support Vector Machines (SVM):** Can be effective with high-dimensional data.
  + **Neural Networks:** More complex, but can learn intricate patterns.
* **Evaluation (Python):** Evaluate the model’s performance using appropriate metrics such as:
  + **Mean Squared Error (MSE):** Measures the average squared difference between predicted and actual values.
  + **R-squared:** Represents the proportion of variance in the dependent variable that is predictable from the independent variables.
  + **Root Mean Squared Error (RMSE):** The square root of MSE, easier to interpret.
  + **Area Under the ROC Curve (AUC):** For classification problems (e.g., active/inactive).
* **Interpretation (Python):** Analyze the model to understand which descriptors are most important for predicting activity.

**2. SQL and Python Code**

Here’s the code, incorporating the error handling and row limit.

**SQL (../data/chembl\_35\_52\_data.sql):**

-- Extracting 100 rows of data for ChEMBL\_35\_52 analysis.  
-- Focuses on activity data and associated SMILES strings.  
  
SELECT  
 act.molregno,  
 act.standard\_value,  
 act.standard\_units,  
 act.standard\_type,  
 md.chembl\_id,  
 md.canonical\_smiles  
FROM  
 activities act  
JOIN  
 molecule\_dictionary md ON act.molregno = md.molregno  
WHERE  
 act.standard\_type = 'IC50' -- Focus on IC50 values  
 AND act.standard\_units = 'nM' -- Standardize units to nM  
 AND act.standard\_value IS NOT NULL  
 AND md.canonical\_smiles IS NOT NULL  
 AND act.standard\_value::text ~ '^[0-9\.]+$' -- Corrected: Use ::text for regex matching  
LIMIT 100;  
  
-- Save the result to a CSV file using pgAdmin's export feature.

**Explanation:**

* **SELECT ... FROM activities act JOIN molecule\_dictionary md ...:** This selects the necessary columns (activity data and SMILES strings) from the activities and molecule\_dictionary tables, joining them on molregno.
* **WHERE ...:** This filters the data to include only:
  + IC50 values (act.standard\_type = 'IC50').
  + Values in nM (act.standard\_units = 'nM').
  + Non-null values for both activity and SMILES.
  + **AND act.standard\_value::text ~ '^[0-9\.]+$':** This is the corrected version of your regex. The error operator does not exist: numeric ~ unknown arises because the ~ operator (regex match) in PostgreSQL typically works with text data. By casting act.standard\_value to text (act.standard\_value::text), you resolve this issue. The regex ^[0-9\.]+$ ensures that the standard\_value contains only digits and periods.
* **LIMIT 100:** Limits the output to 100 rows.

**Important:** After running this SQL query in pgAdmin, use pgAdmin’s export feature (usually right-click on the results and select “Copy” or “Save as CSV”) to save the results to a CSV file named chembl\_35\_52\_data.csv in your ../data/ directory.

**Python Code (notebooks/Topic\_CheMBL\_35\_52\_1\_data\_preparation.ipynb):**

import os  
import pandas as pd  
from rdkit import Chem  
from rdkit.Chem import Descriptors  
import numpy as np  
  
# Define base path for AIMLops structure  
base\_path = ".." # Assuming the notebook is one level down from the base directory  
data\_path = os.path.join(base\_path, "data", "chembl\_35\_52\_data.csv")  
  
# Load data  
try:  
 df = pd.read\_csv(data\_path)  
except FileNotFoundError:  
 print(f"Error: File not found at {data\_path}. Make sure you've saved the CSV from pgAdmin.")  
 exit()  
  
print(f"Loaded {len(df)} rows from {data\_path}")  
print(df.head())  
  
# Function to calculate molecular descriptors using RDKit  
def calculate\_descriptors(smiles):  
 try:  
 mol = Chem.MolFromSmiles(smiles)  
 if mol is None:  
 return None  
 descriptors = {}  
 descriptors['MolWt'] = Descriptors.MolWt(mol)  
 descriptors['LogP'] = Descriptors.MolLogP(mol)  
 descriptors['HBD'] = Descriptors.NumHDonors(mol)  
 descriptors['HBA'] = Descriptors.NumHAcceptors(mol)  
 descriptors['TPSA'] = Descriptors.TPSA(mol)  
 return descriptors  
 except:  
 return None  
  
# Apply descriptor calculation  
df['descriptors'] = df['canonical\_smiles'].apply(calculate\_descriptors)  
df = df.dropna(subset=['descriptors']) # Drop rows where descriptor calculation failed  
df = df.join(pd.DataFrame(df.pop('descriptors').values.tolist(), index=df.index)) # Expand the descriptor column  
  
print(df.head())  
  
# Basic statistics  
print(df[['standard\_value', 'MolWt', 'LogP', 'HBD', 'HBA', 'TPSA']].describe())  
  
# Save the processed data (optional)  
processed\_data\_path = os.path.join(base\_path, "data", "chembl\_35\_52\_processed.csv")  
df.to\_csv(processed\_data\_path, index=False)  
print(f"Processed data saved to {processed\_data\_path}")

**Python Code (notebooks/Topic\_CheMBL\_35\_52\_2\_modeling.ipynb):**

import os  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import LinearRegression  
from sklearn.metrics import mean\_squared\_error, r2\_score  
from sklearn.preprocessing import StandardScaler  
  
# Define base path  
base\_path = ".."  
processed\_data\_path = os.path.join(base\_path, "data", "chembl\_35\_52\_processed.csv")  
  
# Load the processed data  
try:  
 df = pd.read\_csv(processed\_data\_path)  
except FileNotFoundError:  
 print(f"Error: File not found at {processed\_data\_path}. Make sure you've run the data preparation notebook.")  
 exit()  
  
# Prepare data for modeling  
df = df.dropna(subset=['standard\_value', 'MolWt', 'LogP', 'HBD', 'HBA', 'TPSA']) # Remove rows with missing values (important!)  
X = df[['MolWt', 'LogP', 'HBD', 'HBA', 'TPSA']]  
y = df['standard\_value']  
  
# Data scaling  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
  
# Split data into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)  
  
# Train a linear regression model  
model = LinearRegression()  
model.fit(X\_train, y\_train)  
  
# Make predictions  
y\_pred = model.predict(X\_test)  
  
# Evaluate the model  
mse = mean\_squared\_error(y\_test, y\_pred)  
r2 = r2\_score(y\_test, y\_pred)  
  
print(f"Mean Squared Error: {mse}")  
print(f"R-squared: {r2}")  
  
# Example prediction  
example\_index = 0  
example\_features = X\_test[example\_index]  
predicted\_value = model.predict(X\_test[example\_index].reshape(1, -1))[0]  
actual\_value = y\_test.iloc[example\_index]  
  
print(f"Example: Predicted IC50: {predicted\_value:.2f}, Actual IC50: {actual\_value:.2f}")

**Explanation:**

* **os.path.join(base\_path, ...):** Uses os.path.join to construct file paths, making the code more robust to different directory structures. This is essential for AIMLops.
* **try...except FileNotFoundError:** Handles the case where the CSV file is not found, providing a more informative error message.
* **RDKit Descriptor Calculation:** The calculate\_descriptors function calculates several important molecular descriptors using RDKit. It also includes error handling (try...except) in case RDKit encounters issues with a particular SMILES string. If mol is None the descriptor can’t be calculated.
* **Data Cleaning:** df = df.dropna(subset=['descriptors']) removes any rows where descriptor calculation failed. This is crucial because RDKit might not be able to process every SMILES string.
* **Modeling:** Uses sklearn.linear\_model.LinearRegression for a basic regression model. You can easily replace this with other models like RandomForestRegressor or SVR.
* **mean\_squared\_error(y\_test, y\_pred):** Calculates the mean squared error.
* **train\_test\_split:** Splits the data into training and testing sets to evaluate the model’s performance on unseen data.
* **StandardScaler:** Scales the features to have zero mean and unit variance, which can improve the performance of some machine learning algorithms.
* **Example Prediction:** Added an example prediction to show how to use the model to predict the IC50 value for a single compound.

**Addressing the squared=False Error:**

The error old scikit-learn version does not support parameters squared=False in the mean\_squared\_error function indicates that you are using an older version of scikit-learn. The squared=False parameter was added in a later version to allow for the direct calculation of Root Mean Squared Error (RMSE).

**Solution:**

1. **Upgrade Scikit-learn:** The best solution is to upgrade your scikit-learn version. In your Jupyter Notebook or terminal, run:

* pip install -U scikit-learn
* or
* conda update scikit-learn
* Then restart your Jupyter Notebook kernel to ensure the updated version is loaded.

1. **Calculate RMSE Manually (if upgrading is not possible):** If you absolutely cannot upgrade scikit-learn (e.g., due to environment constraints), you can calculate RMSE manually:

* from sklearn.metrics import mean\_squared\_error  
  import numpy as np  
    
  mse = mean\_squared\_error(y\_test, y\_pred)  
  rmse = np.sqrt(mse) # Calculate RMSE manually  
  print(f"Root Mean Squared Error: {rmse}")

**3. AIMLops Structure Considerations**

Your folder structure is a good starting point for AIMLops. Here’s a more detailed suggestion:

Topic\_CheMBL\_35\_52/  
├── data/ # Raw data (CSV from SQL) and processed data  
│ ├── chembl\_35\_52\_data.csv  
│ └── chembl\_35\_52\_processed.csv  
├── notebooks/ # Jupyter notebooks for exploration, analysis, and modeling  
│ ├── Topic\_CheMBL\_35\_52\_1\_data\_preparation.ipynb  
│ └── Topic\_CheMBL\_35\_52\_2\_modeling.ipynb  
├── models/ # Saved model files (e.g., pickled scikit-learn models)  
│ └── linear\_regression\_model.pkl  
├── scripts/ # Python scripts for reusable functions or pipelines  
│ └── feature\_engineering.py  
├── reports/ # Generated reports (e.g., HTML, PDF)  
│ └── model\_performance\_report.html  
├── sql/ # SQL scripts  
│ └── chembl\_35\_52\_data.sql  
├── README.md # Project description and instructions  
└── requirements.txt # List of Python dependencies

**Key AIMLops Principles Reflected in this Structure:**

* **Reproducibility:** The requirements.txt file ensures that anyone can recreate your environment and run your code. The SQL scripts are also version-controlled.
* **Modularity:** Breaking the code into notebooks and scripts promotes modularity and reusability.
* **Data Versioning (Implicit):** While you’re not explicitly versioning the data files, you *are* versioning the SQL script that generates them. This provides a degree of traceability. For true data versioning, consider tools like DVC (Data Version Control).
* **Experiment Tracking:** Although not explicitly shown here, you could use tools like MLflow to track different model runs, parameters, and metrics. This helps you manage and compare experiments.

**4. Five Examples**

Here are five examples of things you can do with this framework:

1. **Different Regression Models:** Replace the LinearRegression model with RandomForestRegressor or SVR (Support Vector Regression) and compare their performance. Experiment with hyperparameters for each model.

* from sklearn.ensemble import RandomForestRegressor  
    
  model = RandomForestRegressor(n\_estimators=100, random\_state=42) # Example hyperparameters  
  model.fit(X\_train, y\_train)  
  y\_pred = model.predict(X\_test)  
  mse = mean\_squared\_error(y\_test, y\_pred)  
  r2 = r2\_score(y\_test, y\_pred)  
  print(f"Random Forest - Mean Squared Error: {mse}")  
  print(f"Random Forest - R-squared: {r2}")

1. **Feature Selection:** Use feature selection techniques (e.g., SelectKBest, Recursive Feature Elimination) to identify the most important descriptors for predicting activity. This can simplify your model and improve its interpretability.

* from sklearn.feature\_selection import SelectKBest, f\_regression  
    
  selector = SelectKBest(score\_func=f\_regression, k=3) # Select top 3 features  
  X\_new = selector.fit\_transform(X\_scaled, y\_train)  
  print("Selected feature indices:", selector.get\_support(indices=True))  
  # Train a model using X\_new instead of X\_scaled

1. **Different Activity Types:** Modify the SQL query to extract data for a different activity type (e.g., Ki, EC50) instead of IC50. You’ll need to adjust the WHERE clause in the SQL query.

* WHERE  
   act.standard\_type = 'Ki' -- Changed to Ki  
   AND act.standard\_units = 'nM'  
   AND act.standard\_value IS NOT NULL  
   AND md.canonical\_smiles IS NOT NULL  
   AND act.standard\_value::text ~ '^[0-9\.]+$'  
  LIMIT 100;

1. **Substructure Search:** Use RDKit to perform a substructure search. For example, find all compounds that contain a specific chemical fragment.

* from rdkit import Chem  
  from rdkit.Chem import Draw  
    
  # Define a SMARTS pattern for the substructure  
  substructure\_smarts = 'c1ccccc1' # Example: Benzene ring  
  substructure = Chem.MolFromSmarts(substructure\_smarts)  
    
  # Find compounds containing the substructure  
  matches = df['canonical\_smiles'].apply(lambda smiles: Chem.MolFromSmiles(smiles).HasSubstructMatch(substructure) if Chem.MolFromSmiles(smiles) else False)  
  df\_with\_substructure = df[matches]  
  print(f"Number of compounds with substructure: {len(df\_with\_substructure)}")  
    
  # Visualize the first few compounds with the substructure  
  if not df\_with\_substructure.empty:  
   mols = [Chem.MolFromSmiles(smiles) for smiles in df\_with\_substructure['canonical\_smiles'].head(3)]  
   img = Draw.MolsToGridImage(mols, highlightAtomLists=[mol.GetSubstructMatch(substructure) for mol in mols], molsPerRow=3)  
   img # Display the image (in a Jupyter Notebook)

1. **Clustering:** Use clustering algorithms (e.g., k-means) to group compounds based on their molecular descriptors. This can help you identify clusters of compounds with similar activity.

* from sklearn.cluster import KMeans  
  import matplotlib.pyplot as plt  
    
  # Choose the number of clusters  
  n\_clusters = 5  
    
  # Apply k-means clustering  
  kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42, n\_init=10) # Added n\_init  
  df['cluster'] = kmeans.fit\_predict(X\_scaled)  
    
  # Visualize the clusters (using two descriptors for simplicity)  
  plt.scatter(df['MolWt'], df['LogP'], c=df['cluster'], cmap='viridis')  
  plt.xlabel('Molecular Weight')  
  plt.ylabel('LogP')  
  plt.title('Compound Clusters')  
  plt.show()

**Important Considerations:**

* **Data Quality:** Always be mindful of data quality. Check for inconsistencies, errors, and outliers. The standard\_value::text ~ '^[0-9\.]+$' filter is a good start, but you might need more sophisticated cleaning.
* **Model Interpretability:** While complex models can achieve higher accuracy, they can be harder to interpret. Consider using techniques like SHAP (SHapley Additive exPlanations) to understand the predictions of your models.
* **Domain Knowledge:** Leverage your domain knowledge to guide your analysis. Are there specific descriptors or structural features that are known to be important for the target you are studying?
* **Cross-Validation:** Use cross-validation techniques (e.g., k-fold cross-validation) to get a more robust estimate of your model’s performance.

**Two Languages (English and Simplified Chinese):**

**English:**

I have provided you with a comprehensive guide to analyzing ChEMBL 35 data using RDKit and scikit-learn, addressing your specific questions and concerns. Remember to adapt the code and examples to your specific research goals.

**Simplified Chinese (中文):**

我为你提供了一份全面的指南，用于使用 RDKit 和 scikit-learn 分析 ChEMBL 35 数据，解决了你的具体问题和顾虑。请记住根据你的具体研究目标调整代码和示例。

I hope this helps you move forward with your project! Let me know if you have any further questions.