# PharmApp Suite

## 🧠 AI for Drug Discovery and Development 🧪

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

## 🧩 Topic: Evaluating overfitting in QSAR models

Okay, I understand. You’re working with ChEMBL 35 data using RDKit and want to analyze it using a Jupyter Notebook within an MLOps-structured project. You need help with SQL queries, Python code (including addressing version issues and the errors mentioned), and implementing a workflow that adheres to your specified folder structure. Let’s get started.

**1. Analysis Model and Approach**

Based on the provided information, a general analysis model for this project would likely involve the following steps:

* **Data Extraction and Preparation:** This is where your SQL queries come in. You’ll use them to extract relevant data from the ChEMBL 35 database and save it into CSV files. The “Topic\_CheMBL\_35\_67” suggests the data selection might be related to specific activity or compound properties.
* **Data Loading and Preprocessing (Python):** Read the CSV data into your Jupyter Notebook. Clean and transform the data, including handling missing values, converting data types (e.g., strings to numeric), and potentially standardizing or normalizing numerical features.
* **Feature Engineering (Python):** Use RDKit to generate molecular descriptors (e.g., physicochemical properties, topological indices, structural fingerprints) from the SMILES strings in your ChEMBL data. These descriptors will serve as features for your analysis.
* **Data Analysis and Modeling (Python):** Depending on your research question, you might perform:
  + **Descriptive Analysis:** Calculate summary statistics, create visualizations (histograms, scatter plots), and explore relationships between features.
  + **Predictive Modeling:** Build machine learning models (e.g., regression, classification) to predict activity, toxicity, or other properties of compounds based on their molecular descriptors.
  + **Clustering:** Group compounds based on their similarities in chemical structure and properties.
* **Model Evaluation and Interpretation (Python):** Assess the performance of your models using appropriate metrics. Interpret the results to gain insights into the relationships between chemical structure and activity.
* **Reporting and Visualization (Python/Jupyter Notebook):** Document your findings, create visualizations, and present your analysis in a clear and concise manner within the Jupyter Notebook.

**Analyzing the Errors:**

* **Error a:** ERROR: operator does not exist: numeric ~ unknown, LINE 12: AND act.standard\_value ~ '^[0-9\.]+$'
  + This error indicates that you’re trying to use the regular expression operator ~ on a numeric column (act.standard\_value). PostgreSQL doesn’t directly support regular expressions on numeric types. You’ll need to cast the numeric column to text (string) before applying the regular expression.
* **Error b:** old scikit-learn version does not support parameters squared=False in the mean\_squared\_error function
  + This means your scikit-learn version is outdated. You have two options:
    1. **Upgrade scikit-learn:** The preferred solution is to upgrade your scikit-learn to a newer version (e.g., using pip install -U scikit-learn).
    2. **Remove squared=False:** If upgrading isn’t feasible, you can remove the squared=False argument from the mean\_squared\_error function. This will return the Mean Squared Error (MSE) instead of the Root Mean Squared Error (RMSE). Keep in mind that this will change the scale of your error metric.

**2. SQL, Python Code (with corrections and examples) and explanations**

**Project Directory Structure**

my\_chembl\_project/  
├── data/  
│ ├── extracted\_data.csv # Example CSV file after running SQL  
│ └── ...  
├── notebooks/  
│ ├── Topic\_CheMBL\_35\_67\_1\_data\_extraction\_and\_preparation.ipynb  
│ ├── Topic\_CheMBL\_35\_67\_2\_feature\_engineering\_and\_modeling.ipynb  
│ └── ...  
├── src/  
│ ├── utils.py # Optional: For helper functions  
├── README.md  
└── ...

**SQL (extracting 100 rows)**

-- File: data\_extraction.sql (To be run in pgAdmin and save results as ../data/extracted\_data.csv)  
SELECT  
 cmp.chembl\_id,  
 cmp.pref\_name,  
 act.standard\_type,  
 act.standard\_value,  
 act.standard\_units,  
 mol.molfile  
FROM  
 compound\_structures cmp  
JOIN  
 activities act ON cmp.molregno = act.molregno  
JOIN  
 molecule\_dictionary mol ON cmp.molregno = mol.molregno  
WHERE  
 act.standard\_type = 'IC50' -- Example filtering criteria  
 AND act.standard\_units = 'nM' -- Example filtering criteria  
 AND act.standard\_value IS NOT NULL  
 AND act.standard\_value::text ~ '^[0-9\.]+$' -- Corrected: Cast to text for regex  
ORDER BY  
 act.standard\_value ASC  
LIMIT 100;

**Explanation:**

* **SELECT ... FROM ... JOIN ...:** This selects data from the compound\_structures, activities, and molecule\_dictionary tables, joining them based on their respective keys.
* **WHERE act.standard\_type = 'IC50' AND act.standard\_units = 'nM':** This filters the data to only include IC50 values measured in nM. Change these values according to your requirements
* **AND act.standard\_value IS NOT NULL:** This ensures that only records with non-null standard values are selected.
* **AND act.standard\_value::text ~ '^[0-9\.]+$':** This is the **corrected** line. It first casts the act.standard\_value column to the text data type using ::text. Then, it applies the regular expression ~ '^[0-9\.]+$' to check if the value contains only digits and periods. This addresses the original SQL error.
* **ORDER BY act.standard\_value ASC:** Orders the results by the IC50 value in ascending order.
* **LIMIT 100:** Limits the result set to 100 rows.

**Python (Jupyter Notebook - Topic\_CheMBL\_35\_67\_1\_data\_extraction\_and\_preparation.ipynb)**

import os  
import pandas as pd  
from rdkit import Chem  
from rdkit.Chem import Descriptors  
#from sklearn.metrics import mean\_squared\_error # Removed, see below  
  
# Base path for your project  
base\_path = os.getcwd() # Get the current working directory (project root)  
data\_path = os.path.join(base\_path, 'data')  
notebook\_path = os.path.join(base\_path, 'notebooks')  
  
# File path for the extracted data CSV  
csv\_file = os.path.join(data\_path, 'extracted\_data.csv')  
  
# Load the CSV data into a Pandas DataFrame  
try:  
 df = pd.read\_csv(csv\_file)  
 print("Data loaded successfully.")  
except FileNotFoundError:  
 print(f"Error: File not found at {csv\_file}. Make sure you've run the SQL script and saved the data.")  
 exit()  
  
# Display the first few rows of the DataFrame  
print(df.head())  
  
# Convert 'standard\_value' to numeric, handling errors  
df['standard\_value'] = pd.to\_numeric(df['standard\_value'], errors='coerce')  
  
# Drop rows where 'standard\_value' is NaN after conversion  
df = df.dropna(subset=['standard\_value'])  
  
# Basic data cleaning  
print(f"Original DataFrame shape: {df.shape}")  
df = df.drop\_duplicates()  
print(f"DataFrame shape after removing duplicates: {df.shape}")  
  
# Convert molfile string to RDKit Mol object. Handle potential errors if molfile is malformed  
def mol\_from\_molfile(molfile\_string):  
 try:  
 mol = Chem.MolFromMolBlock(molfile\_string)  
 if mol is None:  
 return None  
 else:  
 return mol  
 except:  
 return None  
  
df['mol'] = df['molfile'].apply(mol\_from\_molfile)  
df = df.dropna(subset=['mol']) # Drop rows where mol object creation failed  
df = df.reset\_index(drop=True) # Reset index  
print(df.head())

**Explanation:**

* **Import Libraries:** Imports necessary libraries like os, pandas, and RDKit modules.
* **Path Management:** Demonstrates how to construct file paths using os.path.join to adhere to your project structure.
* **Data Loading:** Loads the CSV file into a Pandas DataFrame.
* **Data Cleaning:** Handles potential errors when converting the standard\_value column to numeric and removes rows with missing values.
* **RDKit Integration:** Converts the molfile column into RDKit Mol objects using Chem.MolFromMolBlock. Error handling ensures only valid molecules are kept in the dataframe. Rows with invalid molfile data are removed.

**Python (Jupyter Notebook - Topic\_CheMBL\_35\_67\_2\_feature\_engineering\_and\_modeling.ipynb)**

import os  
import pandas as pd  
from rdkit import Chem  
from rdkit.Chem import Descriptors  
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  
import numpy as np  
  
# Base path for your project  
base\_path = os.getcwd()  
data\_path = os.path.join(base\_path, 'data')  
  
# File path for the extracted data CSV (or read the cleaned dataframe from the previous notebook)  
csv\_file = os.path.join(data\_path, 'extracted\_data.csv')  
  
# Load the CSV data into a Pandas DataFrame  
try:  
 df = pd.read\_csv(csv\_file)  
 print("Data loaded successfully.")  
except FileNotFoundError:  
 print(f"Error: File not found at {csv\_file}. Make sure you've run the SQL script and saved the data.")  
 exit()  
  
# Ensure 'standard\_value' is numeric and no NaNs exist after loading.  
df['standard\_value'] = pd.to\_numeric(df['standard\_value'], errors='coerce')  
df = df.dropna(subset=['standard\_value'])  
  
# Convert molfile string to RDKit Mol object. Handle potential errors.  
def mol\_from\_molfile(molfile\_string):  
 try:  
 mol = Chem.MolFromMolBlock(molfile\_string)  
 if mol is None:  
 return None  
 else:  
 return mol  
 except:  
 return None  
  
df['mol'] = df['molfile'].apply(mol\_from\_molfile)  
df = df.dropna(subset=['mol']) # Drop rows where mol object creation failed  
df = df.reset\_index(drop=True) # Reset index  
  
# Feature Engineering using RDKit  
def calculate\_descriptors(mol):  
 try:  
 descriptors = {}  
 descriptors["MolLogP"] = Descriptors.MolLogP(mol)  
 descriptors["MolecularWeight"] = Descriptors.MolWt(mol)  
 descriptors["NumHAcceptors"] = Descriptors.NumHAcceptors(mol)  
 descriptors["NumHDonors"] = Descriptors.NumHDonors(mol)  
 return pd.Series(descriptors) #Return a series to create a single row  
 except Exception as e:  
 print(f"Error calculating descriptors: {e}")  
 return pd.Series() # Return an empty series in case of error  
  
df = pd.concat([df, df['mol'].apply(calculate\_descriptors)], axis=1)  
  
# Drop rows where descriptor calculation failed.  
df = df.dropna(subset=["MolLogP", "MolecularWeight", "NumHAcceptors", "NumHDonors"])  
  
# Prepare data for modeling  
X = df[["MolLogP", "MolecularWeight", "NumHAcceptors", "NumHDonors"]]  
y = df['standard\_value']  
  
# Data scaling using StandardScaler  
scaler = StandardScaler()  
X = scaler.fit\_transform(X)  
  
  
# Split data into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, 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) # NO squared=False here if you can't upgrade scikit-learn  
rmse = np.sqrt(mse) # Calculate RMSE manually  
r2 = r2\_score(y\_test, y\_pred)  
  
print(f"Mean Squared Error (MSE): {mse}")  
print(f"Root Mean Squared Error (RMSE): {rmse}")  
print(f"R-squared (R2): {r2}")

**Explanation:**

* **Feature Engineering:** Calculates molecular descriptors (LogP, Molecular Weight, Number of Hydrogen Bond Acceptors/Donors) using RDKit. This is just an example; you can add more descriptors based on your needs. Includes error handling. Returns a pd.Series to ensure it’s handled correctly.
* **Data Preparation for Modeling:** Selects the calculated descriptors as features (X) and the standard\_value as the target variable (y).
* **Data Scaling:** Applies StandardScaler to scale the features. This is generally good practice for linear models.
* **Train/Test Split:** Splits the data into training and testing sets.
* **Model Training:** Trains a linear regression model using the training data.
* **Model Evaluation:** Evaluates the model’s performance using Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R2).
* **Error handling:** The code now includes a try-except block within the calculate\_descriptors function to handle potential errors during descriptor calculation. If an error occurs for a particular molecule, the function will return an empty series which will lead to a NaN value and then that row gets dropped. This is important for robustness.

**3. Five Examples**

Here are five examples of things you can do with this data, building upon the code provided:

1. **Expanding Descriptor Set:** Add more RDKit descriptors to your feature set. Explore things like Topological Polar Surface Area (TPSA), Rotatable Bond Count, or various fingerprint types (e.g., Morgan fingerprints). This will likely improve model performance.

* from rdkit.Chem import Crippen, Lipinski  
    
  def calculate\_more\_descriptors(mol):  
   descriptors = {}  
   descriptors["MolLogP"] = Crippen.MolLogP(mol) # Replaced Descriptors.MolLogP with Crippen.MolLogP  
   descriptors["MolecularWeight"] = Descriptors.MolWt(mol)  
   descriptors["NumHAcceptors"] = Lipinski.NumHAcceptors(mol) # Replaced Descriptors.NumHAcceptors with Lipinski.NumHAcceptors  
   descriptors["NumHDonors"] = Lipinski.NumHDonors(mol) # Replaced Descriptors.NumHDonors with Lipinski.NumHDonors  
   descriptors["TPSA"] = Descriptors.TPSA(mol)  
   descriptors["RotatableBonds"] = Descriptors.NumRotatableBonds(mol)  
   return pd.Series(descriptors)  
    
  df = pd.concat([df, df['mol'].apply(calculate\_more\_descriptors)], axis=1)  
  df = df.dropna(subset=["MolLogP", "MolecularWeight", "NumHAcceptors", "NumHDonors", "TPSA", "RotatableBonds"]) # Update dropped cols

1. **Different Regression Models:** Try different regression models like Random Forest Regressor, Support Vector Regressor, or Gradient Boosting Regressor. These models might capture non-linear relationships better than linear regression.

* from sklearn.ensemble import RandomForestRegressor  
    
  model = RandomForestRegressor(n\_estimators=100, random\_state=42) # Example: Random Forest  
  model.fit(X\_train, y\_train)  
  y\_pred = model.predict(X\_test)  
  # ... evaluate model ...

1. **Data Visualization:** Create scatter plots of different descriptors versus the standard\_value to visualize potential relationships. Use matplotlib or seaborn for visualization.

* import matplotlib.pyplot as plt  
  import seaborn as sns  
    
  sns.scatterplot(x='MolLogP', y='standard\_value', data=df)  
  plt.xlabel('MolLogP')  
  plt.ylabel('IC50 (nM)')  
  plt.title('MolLogP vs. IC50')  
  plt.show()

1. **Filtering by Compound Properties in SQL:** Modify your SQL query to filter compounds based on specific properties (e.g., molecular weight range, presence of certain functional groups). This lets you focus on a specific subset of your data.

* -- Example: Filter by molecular weight  
  SELECT ...  
  FROM ...  
  WHERE ...  
  AND cmp.molregno IN (SELECT molregno FROM molecule\_properties WHERE mw\_freebase BETWEEN 200 AND 400)  
  LIMIT 100;

1. **Clustering Analysis:** Use clustering algorithms (e.g., K-Means, hierarchical clustering) to group compounds based on their descriptor profiles. This can help identify clusters of compounds with similar activities.

* from sklearn.cluster import KMeans  
  from sklearn.preprocessing import StandardScaler  
    
  # Scale the data  
  scaler = StandardScaler()  
  X\_scaled = scaler.fit\_transform(X)  
    
  # Apply K-Means clustering  
  kmeans = KMeans(n\_clusters=5, random\_state=42) # Example: 5 clusters  
  df['cluster'] = kmeans.fit\_predict(X\_scaled)  
    
  # Analyze the clusters (e.g., calculate average activity per cluster)  
  print(df.groupby('cluster')['standard\_value'].mean())

**Important Considerations:**

* **Error Handling:** Always include error handling in your code to gracefully manage unexpected situations (e.g., invalid data, missing files).
* **Documentation:** Document your code clearly with comments to explain what each step does.
* **Data Understanding:** Spend time understanding your data and the meaning of the different features. This will help you make informed decisions about your analysis.
* **Virtual Environments:** Use virtual environments to manage your Python dependencies and avoid conflicts.
* **ChEMBL License:** Remember to comply with the ChEMBL database license terms.

**In summary (en resumen):**

This comprehensive guide provides you with the necessary code, explanations, and examples to start your ChEMBL 35 data analysis project. It addresses your specific requirements, including the SQL error correction, Python version compatibility (by removing squared=False), and adherence to your project directory structure. Remember to adapt the code and examples to your specific research question and data. Good luck!