PharmApp Suite

🧠 AI for Drug Discovery and Development 🍼

🧩 Topic: Drug synergy potential prediction using ChEMBL

Okay, I understand. As a Pharmaceutical Research and Development Specialist, I can guide you through your Chembl 35 data analysis project using RDKit and address the errors you've encountered. We'll focus on retrieving a manageable dataset (100 rows) for demonstration and analysis. We will utilize a simplified ML regression model.

1. Analysis Model Overview

Our analysis will focus on exploring bioactivity data from ChEMBL 35 and building a simple predictive model. Here's a breakdown:

- **Data Extraction (SQL):**
  - Retrieve activity data (e.g., IC50 values) and corresponding molecule information (e.g., SMILES strings) from the ChEMBL 35 database. We'll filter to get a small subset (100 rows) for demonstration.
- **Data Preprocessing (Python/RDKit):**
  - Clean and prepare the data. This includes:
    - Handling missing values.
    - Converting IC50 values to pIC50 (a more convenient scale for machine learning).
    - Converting SMILES strings to RDKit Mol objects.
    - Generating molecular descriptors (features) using RDKit. These descriptors are numerical representations of molecular properties that can be used in machine learning.
- **Model Building (Python/Scikit-learn):**
  - Select a simple regression model (e.g., Linear Regression, Ridge Regression) to predict pIC50 values based on the molecular descriptors.
  - Split the data into training and testing sets.
  - Train the model on the training data.
  - Evaluate the model's performance on the testing data using metrics like Mean Squared Error (MSE) or R-squared.

2. Code (SQL & Python)

Here's the code, addressing your specific requirements and errors.

2.1 SQL Code (`Topic_CheMBL_35_98_SQL.sql`)

```sql
-- Connect to the chembl_35 database

-- Extract data: molecule structures and bioactivity.  Limiting to 100 for
demonstration.
SELECT
    md.chembl_id,
    cs.canonical_smiles,
    act.standard_type,
    act.standard_value,
    act.standard_units
FROM
    molecule_dictionary md
JOIN
```

```
    compound_structures cs ON md.molregno = cs.molregno
JOIN
    activities act ON md.molregno = act.molregno
WHERE
    act.standard_type = 'IC50'  -- Focus on IC50 values
    AND act.standard_units = 'nM' -- Ensure units are consistent
    --AND act.standard_value ~ '^[0-9\.]+$'  -- Filter for valid numeric values (see
explanation below)
    AND cs.canonical_smiles IS NOT NULL
    AND act.standard_value IS NOT NULL
LIMIT 100;

-- Save the query results as a CSV file.  This is done through pgAdmin's export
functionality.
```

Explanation of the SQL Code and the Error (ERROR: operator does not exist: numeric ~ unknown):

- The original SQL query included `AND act.standard_value ~ '^[0-9\.]+$'`. This was intended to filter out `standard_value` entries that were not purely numeric. However, the error message "operator does not exist: numeric ~ unknown" indicates that PostgreSQL can't directly use regular expressions (~) to match against a `numeric` column. The column `standard_value` is likely defined as `numeric` in your database.
- **Solution:** Remove the line `AND act.standard_value ~ '^[0-9\.]+$'` . Instead, handle invalid values in the Python code after importing the CSV file. This is a more robust approach, allowing you to handle potential errors gracefully. We will use `pd.to_numeric` with `errors='coerce'` to handle this in the python code.
- The SQL query is designed to extract data from the `molecule_dictionary`, `compound_structures`, and `activities` tables in ChEMBL. It joins these tables based on their shared identifiers (`molregno`). It filters the data to include only IC50 values measured in nM and ensures both the SMILES string and the IC50 value are not null. Finally, it limits the output to 100 rows.

2.2 Python Code (`notebook/Topic_CheMBL_35_98_1_DataPrep_Modeling.ipynb`)

```
import os
import pandas as pd
import numpy as np
from rdkit import Chem
from rdkit.Chem import AllChem
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 math

# Define the base path for your project (adjust as needed)
base_path = "../data"
csv_file_path = os.path.join(base_path, "Topic_CheMBL_35_98_SQL.csv")

# 1. Data Loading and Preprocessing
try:
    df = pd.read_csv(csv_file_path)
    print("Data loaded successfully.")
except FileNotFoundError:
    print(f"Error: File not found at {csv_file_path}")
    exit()
```

```
# Handle missing values (important!)
df.dropna(subset=['canonical_smiles', 'standard_value'], inplace=True)

# Convert 'standard_value' to numeric, handling errors
df['standard_value'] = pd.to_numeric(df['standard_value'], errors='coerce')
df.dropna(subset=['standard_value'], inplace=True) # Drop rows where conversion failed

# Convert IC50 to pIC50
def ic50_to_pic50(ic50):
    """Converts IC50 (nM) to pIC50."""
    pIC50 = 9 - np.log10(ic50)
    return pIC50

df['pIC50'] = df['standard_value'].apply(ic50_to_pic50)

# RDKit Mol Objects and Descriptors
def smiles_to_descriptors(smiles):
    """Converts SMILES string to RDKit Mol object and calculates descriptors."""
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None, None  # Handle invalid SMILES
    # Add hydrogens to the molecule
    mol = Chem.AddHs(mol)
    # Calculate 3D coordinates
    AllChem.EmbedMolecule(mol, AllChem.ETKDGv2())
    descriptors = Descriptors.CalcMolDescriptors(mol)
    return mol, descriptors

df[['mol', 'descriptors']] = df['canonical_smiles'].apply(lambda x:
pd.Series(smiles_to_descriptors(x)))

# Drop rows with invalid SMILES
df.dropna(subset=['mol', 'descriptors'], inplace=True)

# Extract descriptors into a feature matrix
def extract_descriptor_values(descriptors):
    if descriptors is None:
        return None
    return list(descriptors.__dict__.values())

df['descriptor_values'] = df['descriptors'].apply(extract_descriptor_values)
df.dropna(subset=['descriptor_values'], inplace=True)  # Drop if descriptor
calculation failed

# Convert the list of descriptor values into a format suitable for scikit-learn
X = np.array(df['descriptor_values'].tolist())
y = df['pIC50'].values

# 2. Data Splitting and Scaling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scale the features using StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 3. Model Training
```

```
model = LinearRegression()  # Or try Ridge Regression
model.fit(X_train, y_train)

# 4. Model Evaluation
y_pred = model.predict(X_test)
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 predictions (first 5)
print("\nExample Predictions:")
for i in range(min(5, len(y_test))):
    print(f"Actual: {y_test[i]:.2f}, Predicted: {y_pred[i]:.2f}")
```

Explanation of the Python Code:

1. **Import Libraries:** Import necessary libraries like `pandas` for data manipulation, `RDKit` for cheminformatics tasks, and `scikit-learn` for machine learning.
2. **Load Data:** Load the CSV file you created from the SQL query using `pd.read_csv()`. Handle the `FileNotFoundError` exception.
3. **Data Cleaning:**
    - **Handle Missing Values:** Use `df.dropna()` to remove rows with missing values in the 'canonical_smiles' or 'standard_value' columns. This is *crucial*.
    - **Convert to Numeric:** Convert the `standard_value` column to numeric using `pd.to_numeric(errors='coerce')`. `errors='coerce'` will turn any non-numeric values into `NaN`, which can then be removed using `df.dropna()`. This resolves the original SQL error workaround.
4. **IC50 to pIC50 Conversion:** Converts IC50 values to pIC50 using the formula: pIC50 = 9 - log10(IC50).
5. **SMILES to RDKit Mol Objects and Descriptors:**
    - The `smiles_to_descriptors` function converts SMILES strings to RDKit Mol objects. It also calculates a set of molecular descriptors using `Descriptors.CalcMolDescriptors`. Crucially, it handles potential errors where a SMILES string might be invalid. It also adds Hydrogens to the molecule and calculates 3D coordinates before calculating descriptors - this is very important.
    - The code then applies this function to each SMILES string in the DataFrame and stores the Mol objects and descriptors.
    - It handles invalid SMILES by dropping rows where `mol` or `descriptors` are `None`.
6. **Feature Extraction:** Extracts the calculated descriptors into a numerical feature matrix `X`.
7. **Data Splitting:** Splits the data into training and testing sets using `train_test_split`.
8. **Scaling:** Scale the training and testing features using `StandardScaler`. This is *important* for many machine learning algorithms, especially those based on distance calculations.
9. **Model Training:** Creates a `LinearRegression` model and trains it on the training data. You can easily swap this out for another regression model like `Ridge` or `RandomForestRegressor`.
10. **Model Evaluation:** Predicts pIC50 values for the test set and evaluates the model's performance using Mean Squared Error (MSE) and R-squared.
11. **Example Predictions:** Prints a few example predictions to show how the model is performing.

Addressing Error 4b: old scikit-learn version does not support parameters squared=False in the mean_squared_error function

The parameter `squared=False` in `mean_squared_error` is used to return the Root Mean Squared Error (RMSE) instead of the MSE. Since your scikit-learn version doesn't support it, simply remove the

`squared=False` argument to get the MSE. The code above reflects this change: `mse = mean_squared_error(y_test, y_pred)`. If you need the RMSE, calculate it manually: `rmse = math.sqrt(mse)`.

3. Creating the AIMLops Standard folder tree

Create the following directory structure:

```
Topic_CheMBL_35_98/
├── data/
│   └── Topic_CheMBL_35_98_SQL.csv
├── notebooks/
│   └── Topic_CheMBL_35_98_1_DataPrep_Modeling.ipynb
└── models/
    └── (Placeholder for saved models - not used in this example)
```

- **`data/`:** This folder will contain the `Topic_CheMBL_35_98_SQL.csv` file generated from your SQL query.
- **`notebooks/`:** This folder will contain the Jupyter Notebook (`Topic_CheMBL_35_98_1_DataPrep_Modeling.ipynb`).
- **`models/`:** This folder is a placeholder for saving trained models. In this simplified example, we're not saving the model, but in a real project, you would use this folder. You can save the trained model using `pickle` or `joblib`.

4. Running the Code

1. **Execute the SQL Query:** Run the SQL query (`Topic_CheMBL_35_98_SQL.sql`) in pgAdmin connected to your ChEMBL 35 database (ip: 192.168.206.136, user: rd, pass: rd, database: chembl_35). Save the results as a CSV file named `Topic_CheMBL_35_98_SQL.csv` in the `data/` folder. **Important:** In pgAdmin, right-click on the query results and choose "Copy with Headers" to copy the data to your clipboard. Then, paste the contents of your clipboard into the file `Topic_CheMBL_35_98_SQL.csv` after creating the file. Make sure the headers from the SQL query are included in the file.
2. **Open the Jupyter Notebook:** Open the Jupyter Notebook (`Topic_CheMBL_35_98_1_DataPrep_Modeling.ipynb`) in the `notebooks/` folder.
3. **Run the Notebook:** Execute each cell in the notebook sequentially. The code will load the data, perform preprocessing, train the model, and evaluate its performance. Pay attention to the output of each cell to identify any errors or warnings.

5. Example Results (Output from the Python Code)

The output you see will depend on the specific data extracted from your ChEMBL 35 database. Here's an example of what the output might look like:

```
Data loaded successfully.
Mean Squared Error: 1.253456789012345
R-squared: 0.6789012345678901

Example Predictions:
Actual: 6.50, Predicted: 7.12
Actual: 8.20, Predicted: 7.85
Actual: 5.90, Predicted: 6.23
Actual: 7.10, Predicted: 6.98
Actual: 9.00, Predicted: 8.55
```

Key Takeaways and Next Steps

- **Error Handling:** The code includes error handling for file loading and SMILES conversion, making it more robust.

- **Data Cleaning:** Handling missing values and converting to numeric types are critical steps in data preprocessing.
- **Feature Engineering:** Molecular descriptors are important features for modeling bioactivity.
- **Model Selection:** Linear Regression is a simple starting point. Experiment with other regression models like Ridge Regression, Random Forest, or Gradient Boosting for potentially better performance.
- **Hyperparameter Tuning:** Tune the hyperparameters of your chosen model using techniques like cross-validation to optimize its performance.
- **Model Validation:** Use a separate validation set to assess the model's generalization ability.
- **Save the Model:** Use `pickle` or `joblib` to save the trained model so you can reuse it later without retraining.
- **Further Exploration:** Explore other molecular descriptors and feature selection techniques to improve the model's accuracy.

6. Five Examples of How to Extend the Analysis

Here are five examples of things you could do to further extend this analysis, along with brief code snippets to get you started:

1. **Try Different Molecular Descriptors:** Experiment with different sets of molecular descriptors.

```
from rdkit.Chem import Descriptors3D  # 3D descriptors
from rdkit.Chem import Lipinski  # Lipinski's Rule of Five descriptors

def smiles_to_extended_descriptors(smiles):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None, None
    mol = Chem.AddHs(mol)
    AllChem.EmbedMolecule(mol, AllChem.ETKDGv2())

    # Calculate a few more descriptors as an example
    mw = Descriptors.MolWt(mol)
    logp = Descriptors.MolLogP(mol)
    hba = Lipinski.NumHAcceptors(mol)
    hbd = Lipinski.NumHDonors(mol)
    tpsa = Descriptors3D.TPSA(mol)

    return mol, (mw, logp, hba, hbd, tpsa)

df[['mol', 'extended_descriptors']] = df['canonical_smiles'].apply(lambda x:
pd.Series(smiles_to_extended_descriptors(x)))

# Extract the new descriptors into a numpy array
def extract_extended_descriptor_values(descriptors):
  if descriptors is None:
      return None
  return list(descriptors)

df['extended_descriptor_values'] =
df['extended_descriptors'].apply(extract_extended_descriptor_values)

# Drop rows where extended_descriptor_values are null
df.dropna(subset=['extended_descriptor_values'], inplace=True)

# Convert the list of extended descriptor values into a format suitable for scikit-
learn
X_extended = np.array(df['extended_descriptor_values'].tolist())
```

2. **Use a Different Regression Model:** Try a non-linear model like `RandomForestRegressor` or `GradientBoostingRegressor`.

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=42) # Example parameters
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}")
```

3. **Implement Cross-Validation:** Use cross-validation to get a more robust estimate of the model's performance.

```
from sklearn.model_selection import cross_val_score, KFold

kf = KFold(n_splits=5, shuffle=True, random_state=42)  # 5-fold cross-validation
cv_scores = cross_val_score(model, X, y, cv=kf, scoring='neg_mean_squared_error')
cv_mse = -cv_scores.mean()  # Convert back to positive MSE
print(f"Cross-Validation Mean Squared Error: {cv_mse}")
```

4. **Visualize the Results:** Create scatter plots of predicted vs. actual pIC50 values.

```
import matplotlib.pyplot as plt

plt.scatter(y_test, y_pred)
plt.xlabel("Actual pIC50")
plt.ylabel("Predicted pIC50")
plt.title("Actual vs. Predicted pIC50 Values")
plt.show()
```

5. **Implement Feature Selection:** Use feature selection techniques to identify the most important descriptors.

```
from sklearn.feature_selection import SelectKBest, f_regression

# Use SelectKBest to select the 10 best features based on f_regression
selector = SelectKBest(score_func=f_regression, k=10)
X_new = selector.fit_transform(X, y)

# Get the indices of the selected features
selected_indices = selector.get_support(indices=True)

# Print the selected features
print("Selected feature indices:", selected_indices)
```

This is a starting point. The specific techniques and code will need to be adapted to your specific data and research questions. Remember to consult the RDKit and scikit-learn documentation for more information on the available functions and options. Remember the models folder should contain model files to ensure AIMLops structure Let me know if you have any more questions.