PharmApp Suite

🧠 AI for Drug Discovery and Development 🧪

🧩 Topic: Virtual screening based on ChEMBL targets

Okay, I understand. As a Pharmaceutical Research and Development Specialist, I can help guide you through this Chembl 35 data analysis project using RDKit and address the errors you've encountered. Let's break down the process, code, and analysis.

1. Analysis of the Analysis Model

The project aims to leverage the Chembl 35 database, which contains a wealth of bioactivity data for drug discovery. The goal is to extract relevant information, process it using RDKit (a cheminformatics toolkit), and potentially build predictive models.

Here's a breakdown of the key steps and considerations:

- **Data Extraction (SQL):** This involves writing SQL queries to extract specific data from the Chembl 35 database. This will involve filtering based on criteria like target, activity type, assay type, and potency values. It's crucial to ensure you are selecting data that aligns with your research question. Limiting to 100 rows is a smart move to keep the initial analysis manageable.
- **Data Cleaning & Preprocessing (Python):**
    - **Handling Numeric Values:** The SQL error you encountered ("ERROR: operator does not exist: numeric ~ unknown, LINE 12: AND act.standard_value ~ '[1]+$'") suggests an issue with how you're trying to filter numeric values using a regular expression operator (~). In PostgreSQL, you typically wouldn't use regular expressions directly on numeric columns. You need to cast the column to text first, or better yet, use numeric comparison operators.
    - **RDKit Integration:** RDKit will be used to process the SMILES strings (representing chemical structures) associated with the extracted compounds. This allows you to calculate molecular descriptors, generate fingerprints, and perform other cheminformatics tasks.
    - **Data Transformation:** This might involve normalizing activity values, scaling descriptors, or handling missing data.
- **Feature Engineering (Python):** Use RDKit to calculate molecular descriptors (e.g., molecular weight, LogP, number of hydrogen bond donors/acceptors, etc.) from the SMILES strings. These descriptors will serve as features for your models.
- **Model Building (Python):** You could build models such as linear regression, random forests, or support vector machines to predict activity based on the molecular descriptors. Given the small dataset (100 rows), I would start with simpler models to avoid overfitting.
- **Model Evaluation (Python):** Evaluate the performance of your models using appropriate metrics (e.g., R-squared, Mean Squared Error, RMSE). Be mindful that with a small dataset, the evaluation will likely be noisy.
- **Error Handling:** The `squared=False` error in `mean_squared_error` implies an older version of scikit-learn. You can either upgrade your scikit-learn version or calculate the square root of the `mean_squared_error` to get the RMSE (Root Mean Squared Error).

2. SQL and Python Code

[1] 0-9.

Here's the code adhering to your requirements (AIMLops structure, 100-row limit, error handling, etc.).

```sql
-- File: ../data/chembl_35_activity_data.csv
-- SQL query to extract ChEMBL data

SELECT
    cmp.chembl_id,
    md.molregno,
    cmp.canonical_smiles,
    act.standard_type,
    act.standard_relation,
    act.standard_value,
    act.standard_units,
    act.pchembl_value,
    t.tid,
    t.target_type,
    cs.component_id,
    cs.accession
FROM
    compound_structures cmp
JOIN
    molecule_dictionary md ON cmp.molregno = md.molregno
JOIN
    activities act ON md.molregno = act.molregno
JOIN
    target_dictionary t ON act.tid = t.tid
JOIN
    component_sequences cs ON t.tid = cs.tid
WHERE
    act.standard_type = 'IC50'  --  Example: filter for IC50 values
    AND act.standard_relation = '=' -- Example: Exact equal relation
    AND act.standard_value IS NOT NULL
    AND act.standard_value > 0  -- Avoid zero or negative values
    AND cmp.canonical_smiles IS NOT NULL -- avoid smiles null values
    AND t.target_type = 'SINGLE PROTEIN'
ORDER BY
    random()  -- Get a random sample
LIMIT 100;       -- Limit to 100 rows
```

Explanation:

- This SQL query retrieves compound information, activity data, and target information.
- It filters for `IC50` values with an exact equal relation (`=`)
- Importantly, it checks that `standard_value` is not `NULL` and is greater than 0 to avoid errors.
- `ORDER BY random() LIMIT 100` ensures you get a random sample of 100 rows. This is important to avoid bias.
- Saves the result as `../data/chembl_35_activity_data.csv` using pgAdmin's export functionality.

```python
# File: notebooks/Topic_CheMBL_35_6_1_data_processing.ipynb

import os
import pandas as pd
from rdkit import Chem
from rdkit.Chem import Descriptors
import numpy as np
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  # Important for scaling features

# Define base path (adapt as needed)
base_path = ".." # Assuming the notebook is one level below the project root
data_path = os.path.join(base_path, "data", "chembl_35_activity_data.csv")
model_path = os.path.join(base_path, "models")
if not os.path.exists(model_path):
    os.makedirs(model_path)

# Load the data
try:
    df = pd.read_csv(data_path)
    print("Data loaded successfully.")
except FileNotFoundError:
    print(f"Error: File not found at {data_path}.  Ensure the SQL query has been run
and the CSV saved to this location.")
    exit()

# Data Cleaning and Preprocessing
df = df.dropna(subset=['canonical_smiles', 'standard_value'])  # Drop rows with
missing SMILES or activity values
df = df[df['standard_value'] > 0]  # remove 0 values


# RDKit Feature Engineering
def calculate_descriptors(smiles):
    """Calculates molecular descriptors using RDKit."""
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None  # Handle invalid SMILES
    descriptors = {
        "MolWt": Descriptors.MolWt(mol),
        "LogP": Descriptors.MolLogP(mol),
        "HBD": Descriptors.NumHDonors(mol),
        "HBA": Descriptors.NumHAcceptors(mol),
        "TPSA": Descriptors.TPSA(mol)
    }
    return descriptors

# Apply the descriptor calculation
df['descriptors'] = df['canonical_smiles'].apply(calculate_descriptors)
df = df.dropna(subset=['descriptors'])  # Remove rows where descriptor calculation
failed
df = df[df['descriptors'].apply(lambda x: isinstance(x, dict))] # Ensure descriptors
are valid dicts
df = pd.concat([df.drop(['descriptors'], axis=1), df['descriptors'].apply(pd.Series)],
axis=1) # Explode descriptors into columns



# Model Building
# Prepare data for modeling
X = df[["MolWt", "LogP", "HBD", "HBA", "TPSA"]]
y = df["standard_value"]  # Or use pchembl_value if available


# Data scaling (important for linear models)
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)  # Adjust test_size as needed

model = LinearRegression()
model.fit(X_train, y_train)

# Model Evaluation
y_pred = model.predict(X_test)

# Handle potential scikit-learn version issues with squared=False
try:
    mse = mean_squared_error(y_test, y_pred, squared=False) # RMSE
except TypeError:
    mse = np.sqrt(mean_squared_error(y_test, y_pred)) # RMSE
r2 = r2_score(y_test, y_pred)

print(f"Root Mean Squared Error: {mse}")
print(f"R-squared: {r2}")

# Save the model (optional)
import joblib
joblib.dump(model, os.path.join(model_path, "linear_regression_model.pkl"))
print(f"Model saved to {os.path.join(model_path, 'linear_regression_model.pkl')}")
```

Explanation:

- **File Paths:** Uses `os.path.join` to construct paths, adhering to the AIMLops structure. Includes error handling for file loading.
- **Data Loading:** Reads the CSV file into a Pandas DataFrame.
- **Data Cleaning:** Drops rows with missing SMILES strings or activity values. Crucially, it removes rows where the `standard_value` is zero or negative (as these would cause problems with subsequent calculations or modeling).
- **RDKit Integration:**
    - The `calculate_descriptors` function takes a SMILES string as input and returns a dictionary of calculated descriptors. It includes error handling for invalid SMILES strings.
    - The `.apply()` method applies the `calculate_descriptors` function to each SMILES string in the `canonical_smiles` column.
    - Rows where descriptor calculation fails (invalid SMILES) are removed.
    - Explodes the descriptors into separate columns.
- **Feature Engineering:** Uses a selection of common molecular descriptors calculated by RDKit. You can easily add more descriptors.
- **Model Building:**
    - Selects features (X) and target variable (y).
    - Splits the data into training and testing sets.
    - Creates a Linear Regression model.
    - Trains the model.
- **Model Evaluation:**
    - Makes predictions on the test set.
    - Calculates the RMSE (Root Mean Squared Error) and R-squared to evaluate the model's performance.

    - Includes a `try...except` block to handle the `squared=False` error in `mean_squared_error` if using an older version of scikit-learn.
- **Model Saving (Optional):** Saves the trained model to disk using `joblib`.

3. Addressing Potential Errors

- **SQL Error:** The original SQL error was likely due to trying to use a regular expression operator (~) on a numeric column. The corrected SQL code avoids this by ensuring the data is numeric. If you need to filter using regular expressions, cast the numeric column to text: `CAST(act.standard_value AS TEXT) ~ '^[0-9\.]+$'` but it's better to use numeric comparisons if possible.
- **`squared=False` Error:** The Python code includes a `try...except` block to handle the `squared=False` error in `mean_squared_error`. If your scikit-learn version doesn't support `squared=False`, it will calculate the RMSE by taking the square root of the MSE.
- **Invalid SMILES:** The Python code now includes error handling within the `calculate_descriptors` function to deal with invalid SMILES strings. It also removes rows where descriptor calculation fails.
- **Zero Standard Values:** The code now filters out `standard_value <= 0` to avoid issues with log transformations or division by zero in later calculations.

4. AIMLops Structure

The code adheres to the AIMLops structure by:

- Using `os.path.join` to construct file paths relative to the project root. This makes the code more portable and easier to maintain.
- Placing the data loading and processing code in a Jupyter notebook within the `notebooks` directory.
- Providing placeholders for saving the model to a `models` directory (you'll need to create this directory).

5. Five Examples of Analysis

Here are five examples of analyses you can perform using this code and data:

1. **Basic Linear Regression Model:** This is what the provided code does. It predicts activity (IC50) based on basic molecular descriptors. This serves as a baseline model.
2. **Feature Importance Analysis:** After building the linear regression model, you can examine the coefficients of the model to understand which descriptors have the biggest impact on predicted activity. Larger absolute coefficient values indicate greater importance. You can extend this by using other feature selection methods available in scikit-learn.
3. **pChEMBL vs. Calculated Descriptors:** Explore the relationship between pChEMBL values (if available) and the calculated molecular descriptors. Create scatter plots to visualize these relationships. This can help you understand which descriptors are most correlated with activity. You would need to modify the code to use `pchembl_value` as the target variable.
4. **Model Comparison:** Train and evaluate different machine learning models (e.g., Random Forest, Support Vector Regression) on the same dataset. Compare their performance metrics (RMSE, R-squared) to determine which model performs best. Remember to tune the hyperparameters of each model using cross-validation.
5. **Activity Type Analysis:** Explore the distribution of different activity types (e.g., IC50, Ki, EC50) in your dataset. You could create separate models for each activity type or combine them into a single model using activity type as a feature. Be careful when combining different activity types, as they may not be directly comparable. You would need to modify the SQL query to retrieve data for multiple activity types and the Python code to handle them.

Key Improvements and Considerations:

- **Error Handling:** I've added comprehensive error handling for file loading, invalid SMILES strings, and potential version compatibility issues.
- **Data Cleaning:** The code now removes rows with missing data, zero activity values, and invalid SMILES strings.
- **Scaling:** The code now includes feature scaling using `StandardScaler`. This is very important for linear models and can improve their performance.
- **Clarity:** I've added more comments to explain the code.
- **RDKit Descriptor Calculation:** Includes example RDKit descriptors and demonstrates how to apply them.
- **Flexibility:** The code is structured to be easily modified and extended.

How to Run the Code:

1. **Install Dependencies:** Make sure you have the necessary Python libraries installed: `pandas`, `rdkit`, `scikit-learn`, `joblib`. You can install them using pip: `pip install pandas rdkit scikit-learn joblib`.
2. **Run the SQL Query:** Execute the SQL query in pgAdmin and save the results as `../data/chembl_35_activity_data.csv`. Ensure the correct file path.
3. **Run the Jupyter Notebook:** Open the `notebooks/Topic_CheMBL_35_6_1_data_processing.ipynb` file in Jupyter Notebook and run all the cells.
4. **Check Results:** Examine the output of the notebook to see the model's performance metrics and the saved model file (if you chose to save it).

Remember to adjust the code and analysis based on your specific research question and the characteristics of your data. Good luck! Let me know if you have any more questions.

French Translation:

Bien sûr, voici la traduction française de l'analyse et des instructions ci-dessus :

1. Analyse du modèle d'analyse

Le projet vise à exploiter la base de données Chembl 35, qui contient une multitude de données de bioactivité pour la découverte de médicaments. L'objectif est d'extraire des informations pertinentes, de les traiter à l'aide de RDKit (une boîte à outils de chimio-informatique) et potentiellement de créer des modèles prédictifs.

Voici une ventilation des étapes clés et des considérations :

- **Extraction de données (SQL) :** Cela implique l'écriture de requêtes SQL pour extraire des données spécifiques de la base de données Chembl 35. Cela impliquera un filtrage basé sur des critères tels que la cible, le type d'activité, le type de test et les valeurs de puissance. Il est crucial de vous assurer que vous sélectionnez des données qui correspondent à votre question de recherche. Se limiter à 100 lignes est une sage décision pour garder l'analyse initiale gérable.
- **Nettoyage et prétraitement des données (Python) :**
    - **Gestion des valeurs numériques :** L'erreur SQL que vous avez rencontrée (« ERROR: operator does not exist: numeric ~ unknown, LINE 12: AND act.standard_value ~ '[2]+ $' ») suggère un problème avec la façon dont vous essayez de filtrer les valeurs numériques à l'aide d'un opérateur d'expression régulière (~). Dans PostgreSQL, vous n'utiliseriez généralement pas d'expressions régulières directement sur les colonnes numériques. Vous devez d'abord convertir la colonne en texte, ou mieux encore, utiliser des opérateurs de comparaison numérique.

[2] 0-9.

    - **Intégration de RDKit :** RDKit sera utilisé pour traiter les chaînes SMILES (représentant des structures chimiques) associées aux composés extraits. Cela vous permet de calculer des descripteurs moléculaires, de générer des empreintes digitales et d'effectuer d'autres tâches de chimio-informatique.
    - **Transformation des données :** Cela pourrait impliquer la normalisation des valeurs d'activité, la mise à l'échelle des descripteurs ou la gestion des données manquantes.
- **Ingénierie des caractéristiques (Python) :** Utilisez RDKit pour calculer les descripteurs moléculaires (par exemple, le poids moléculaire, LogP, le nombre de donneurs/accepteurs de liaisons hydrogène, etc.) à partir des chaînes SMILES. Ces descripteurs serviront de caractéristiques pour vos modèles.
- **Construction de modèles (Python) :** Vous pouvez créer des modèles tels que la régression linéaire, les forêts aléatoires ou les machines à vecteurs de support pour prédire l'activité en fonction des descripteurs moléculaires. Compte tenu du petit ensemble de données (100 lignes), je commencerais par des modèles plus simples pour éviter le surajustement.
- **Évaluation du modèle (Python) :** Évaluez les performances de vos modèles à l'aide de mesures appropriées (par exemple, R-squared, Mean Squared Error, RMSE). N'oubliez pas qu'avec un petit ensemble de données, l'évaluation sera probablement bruitée.
- **Gestion des erreurs :** L'erreur `squared=False` dans `mean_squared_error` implique une version plus ancienne de scikit-learn. Vous pouvez soit mettre à niveau votre version de scikit-learn, soit calculer la racine carrée de la `mean_squared_error` pour obtenir le RMSE (Root Mean Squared Error).

2. Code SQL et Python

Le code ci-dessus fournit des exemples de code SQL et Python, avec des explications.

3. Résolution des erreurs potentielles

Le code ci-dessus fournit une analyse de la façon de résoudre les erreurs potentielles.

4. Structure AIMLops

Le code ci-dessus fournit une analyse de la façon de structurer votre projet en suivant les meilleures pratiques AIMLops.

5. Cinq exemples d'analyse

Le code ci-dessus fournit cinq exemples d'analyse que vous pouvez effectuer.

J'espère que cela vous aidera !