PharmApp Suite

🧠 AI for Drug Discovery and Development 🧪

🧩 Topic: DrugBank interaction cross-validation

Okay, I understand. You're working with ChEMBL 35 data within an AIMLops structure and need to perform analysis using SQL and Python (RDKit and scikit-learn). You're encountering issues related to SQL syntax and scikit-learn version compatibility. I'll provide guidance and code snippets for your `Topic_CheMBL_35_83` analysis.

Here's a breakdown of the approach, code, and explanations:

1. Analysis Model (Analysis of Activity Data):

The core of this analysis revolves around retrieving and processing activity data from ChEMBL. This typically involves:

- **Data Extraction (SQL):** Querying the ChEMBL database to select relevant activity records based on specific criteria (e.g., target protein, activity type, activity value range).
- **Data Cleaning and Transformation (Python/RDKit):**
    - Removing or correcting erroneous data points.
    - Standardizing activity values (e.g., converting different units to a common unit like pIC50).
    - Generating molecular descriptors using RDKit.
- **Data Analysis (Python/scikit-learn):**
    - Exploring relationships between molecular descriptors and activity values using statistical methods.
    - Building predictive models (e.g., regression models) to predict activity based on molecular structure.
- **Visualization (Python/matplotlib/seaborn):**
    - Creating visualizations to highlight key trends and relationships in the data.

2. AIMLops Structure Adaptation:

I'll structure the code snippets with the AIMLops folder structure in mind, assuming a structure like this:

```
project_root/
├── data/          # CSV files from SQL queries
├── notebooks/     # Jupyter notebooks
│   └── Topic_CheMBL_35_83_1_data_extraction_and_cleaning.ipynb
│   └── Topic_CheMBL_35_83_2_descriptor_generation_and_analysis.ipynb
├── src/           # (Optional) Python modules for reusable code
├── models/        # (Optional) Saved models
└── README.md
```

3. Code (SQL and Python):

SQL (to extract data and save as CSV):

```sql
-- File: ../data/chembl35_activity_data.csv

SELECT
    md.chembl_id,
    act.standard_type,
    act.standard_value,
    act.standard_units,
    act.assay_chembl_id,
    cmpd.canonical_smiles
FROM
    activities act
JOIN
    molecule_dictionary md ON act.molregno = md.molregno
JOIN
    compound_structures cmpd ON md.molregno = cmpd.molregno
WHERE
    act.standard_type = 'IC50' -- Filter for IC50 values
    AND act.standard_units = 'nM' -- Filter for nM units
    AND act.standard_value IS NOT NULL
    AND act.standard_value > 0 -- Avoid zero or negative values
    AND act.standard_value::text ~ '^[0-9\.]+$' -- Ensure standard_value is numeric
(CAST TO TEXT THEN REGEX)
LIMIT 100; -- Limit to 100 rows for demonstration
```

Explanation:

- **`SELECT ... FROM ... JOIN ...`:** Standard SQL query structure to retrieve data from relevant tables (`activities`, `molecule_dictionary`, `compound_structures`).
- **`WHERE ...`:** Crucial filtering conditions:
    - `act.standard_type = 'IC50'` - Selects only IC50 activity measurements.
    - `act.standard_units = 'nM'` - Selects only values measured in nanomolar (nM).
    - `act.standard_value IS NOT NULL AND act.standard_value > 0` - Filters out null and non-positive values.
    - `act.standard_value::text ~ '^[0-9\.]+$'` - **This is the fix for your error (a):** The original `act.standard_value ~ '^[0-9\.]+$'` likely failed because the `standard_value` column is numeric, not text. This line converts the numeric column to text (`::text`) before applying the regular expression. This makes the regex matching work correctly and efficiently filters the column. The regex `^[0-9\.]+$` ensures the value contains only digits and periods. This prevents errors if the `standard_value` column contains non-numeric characters.
- **`LIMIT 100`:** Limits the output to 100 rows. Remove this line for a larger dataset (but be mindful of performance).

Python Code (Jupyter Notebook):

```python
# File: notebooks/Topic_CheMBL_35_83_1_data_extraction_and_cleaning.ipynb

import pandas as pd
import numpy as np
import os
import math

# Define base path
base_path = os.path.abspath(os.path.join(os.getcwd(), "..")) # Assuming notebooks are
one level down from project root
data_path = os.path.join(base_path, "data")

# Load the data
csv_file = os.path.join(data_path, "chembl35_activity_data.csv")
try:
    df = pd.read_csv(csv_file)
    print(f"Data loaded successfully from {csv_file}")
except FileNotFoundError:
    print(f"Error: File not found at {csv_file}.  Make sure you ran the SQL query and
saved the CSV.")
    exit()

# Data Cleaning and Transformation
def calculate_pIC50(standard_value):
    """Calculates pIC50 values from IC50 in nM."""
    if pd.isna(standard_value) or not isinstance(standard_value, (int, float)):
        return np.nan  # Handle missing or invalid values

    if standard_value <= 0:
        return np.nan  # Handle non-positive values

    pIC50 = -np.log10(standard_value * 1e-9)  # Convert nM to M and calculate pIC50
    return pIC50

# Apply the function to create a pIC50 column
df['pIC50'] = df['standard_value'].apply(calculate_pIC50)

# Remove rows with NaN pIC50 values (optional, but good practice)
df = df.dropna(subset=['pIC50'])

# Display the first few rows of the cleaned data
print(df.head())

# Save the cleaned data (optional)
cleaned_csv_file = os.path.join(data_path, "chembl35_activity_data_cleaned.csv")
df.to_csv(cleaned_csv_file, index=False)
print(f"Cleaned data saved to {cleaned_csv_file}")
```

```python
# File: notebooks/Topic_CheMBL_35_83_2_descriptor_generation_and_analysis.ipynb

import pandas as pd
import numpy as np
import os
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

# Define base path
base_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
data_path = os.path.join(base_path, "data")

# Load the cleaned data
cleaned_csv_file = os.path.join(data_path, "chembl35_activity_data_cleaned.csv")
try:
    df = pd.read_csv(cleaned_csv_file)
    print(f"Cleaned data loaded successfully from {cleaned_csv_file}")
except FileNotFoundError:
    print(f"Error: File not found at {cleaned_csv_file}. Make sure you ran the first
notebook.")
    exit()

# RDKit Descriptor Generation
def generate_descriptors(smiles):
    """Generates RDKit descriptors from SMILES string."""
    try:
        mol = Chem.MolFromSmiles(smiles)
        if mol is None:
            return None  # Handle invalid SMILES
        descriptors = {}
        for name, function in Descriptors.descList:
            try:
                descriptors[name] = function(mol)
            except:
                descriptors[name] = np.nan # Handle descriptor calculation failures
        return descriptors
    except Exception as e:
        print(f"Error generating descriptors for SMILES: {smiles}, Error: {e}")
        return None

# Apply the descriptor generation to each molecule
df['descriptors'] = df['canonical_smiles'].apply(generate_descriptors)

# Handle rows where descriptor generation failed.  Remove rows with None values.
df = df.dropna(subset=['descriptors'])

# Convert descriptors to a DataFrame (important for scikit-learn)
descriptor_df = pd.DataFrame(list(df['descriptors']))

# Merge descriptors with activity data
df = pd.concat([df, descriptor_df], axis=1)

# Drop the original 'descriptors' column (optional)
df = df.drop('descriptors', axis=1)

# Data Preprocessing for Machine Learning
# Select features (descriptors) and target (pIC50)
X = df.select_dtypes(include=np.number).drop(['standard_value', 'pIC50'], axis=1)
#Drop non descriptor columns
y = df['pIC50']

# Handle missing values (important!)
X = X.replace([np.inf, -np.inf], np.nan)  # Replace infinities with NaN
X = X.dropna(axis=1, how='any') # Drop columns with remaining NaN

# 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)

# Feature Scaling (important for linear models)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Model Training (Linear Regression)
model = LinearRegression()
model.fit(X_train, y_train)

# Model Evaluation
y_pred = model.predict(X_test)

# Check scikit-learn version
import sklearn
print(f"Scikit-learn version: {sklearn.__version__}")
```

```
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 (Predict pIC50 for a specific molecule)
def predict_pic50(smiles, model, scaler, feature_names):
    """Predicts pIC50 for a given SMILES string."""
    descriptors = generate_descriptors(smiles)
    if descriptors is None:
        return None

    #Convert to DF
    descriptor_df = pd.DataFrame([descriptors])

    # Select only the features the model was trained on.
    descriptor_df = descriptor_df[feature_names]
    descriptor_df = descriptor_df.fillna(0)  # Handle potential missing values after
feature selection

    # Scale the features
    scaled_descriptors = scaler.transform(descriptor_df)

    # Make the prediction
    predicted_pic50 = model.predict(scaled_descriptors)[0]
    return predicted_pic50

# Get feature names from the training data used in the model.
feature_names = list(X.columns) #Use the filtered X that has no NaNs

# Example SMILES string
example_smiles = 'CC(=O)Oc1ccccc1C(=O)O' # Aspirin

predicted_pic50 = predict_pic50(example_smiles, model, scaler, feature_names)

if predicted_pic50 is not None:
    print(f"Predicted pIC50 for {example_smiles}: {predicted_pic50}")
else:
    print(f"Could not predict pIC50 for {example_smiles}")
```

Explanation:

- **`Topic_CheMBL_35_83_1_data_extraction_and_cleaning.ipynb`**:
  - Loads the data extracted by the SQL query.
  - `calculate_pIC50`: Calculates pIC50 values from IC50 values. The function correctly handles potential errors (invalid inputs) and converts the units appropriately.
  - Saves the cleaned data for the next stage.
- **`Topic_CheMBL_35_83_2_descriptor_generation_and_analysis.ipynb`**:
  - Loads the cleaned data.
  - `generate_descriptors`: Generates RDKit descriptors for each molecule using the SMILES string. Critically, it includes comprehensive error handling, including checking if the SMILES is valid and handling descriptor calculation errors. This is very important for real-world datasets. The error handling in this function will prevent the notebook from crashing. Invalid SMILES are handled.
  - Creates a Pandas DataFrame from the descriptors.

    - Performs a train/test split.
    - **`StandardScaler`:** Scales the features (descriptors) using `StandardScaler`. This is very important for linear models like Linear Regression.
    - Trains a `LinearRegression` model.
    - Evaluates the model using MSE and R-squared.
    - Includes example prediction code using the trained model. The `predict_pic50` function correctly handles descriptor generation, scaling, and prediction. The `feature_names` are correctly extracted to align with training data.
- **Error Handling:** The code includes error handling for file loading, SMILES parsing, and descriptor calculation. This is crucial for robustness.
- **`sklearn.__version__`**: Prints the scikit-learn version.

Important Notes & Addressing Error (b):

- **scikit-learn version:** The error "old scikit-learn version does not support parameters `squared=False` in the `mean_squared_error` function" means you are using a very old version of scikit-learn. Update it immediately: `pip install -U scikit-learn`. The `squared` parameter was added in a relatively recent version. If you cannot update, you should remove `squared=False` from the `mean_squared_error` call, but this is *strongly* discouraged. Updating is the right solution.
- **Descriptor Selection:** The code generates *all* RDKit descriptors. In a real project, you would perform feature selection to choose the most relevant descriptors. This can improve model performance and reduce overfitting. Consider using techniques like:
    - Variance Thresholding: Remove descriptors with low variance.
    - Univariate Feature Selection: Select descriptors based on statistical tests.
    - Recursive Feature Elimination: Iteratively remove descriptors based on model performance.
- **Model Selection:** Linear Regression is a simple starting point. Experiment with other models (e.g., Random Forest, Support Vector Regression) to see if you can improve performance.
- **Cross-Validation:** Use cross-validation to get a more robust estimate of model performance.
- **Hyperparameter Tuning:** Optimize the hyperparameters of your chosen model using techniques like grid search or random search.
- **Data Exploration:** Spend time exploring the data to understand relationships and identify potential issues. Visualize the data using histograms, scatter plots, etc.
- **Units:** Be extremely careful with units. Ensure all activity values are in consistent units before calculating pIC50.

4. 5 Examples:

Here are 5 examples, demonstrating different aspects of the process.

Example 1: Basic Data Extraction & pIC50 Calculation:

This example focuses on extracting data from the ChEMBL database and calculating pIC50 values. It's a simplified version of the full notebook.

```
import pandas as pd
import numpy as np

# Simulate reading from CSV (replace with pd.read_csv)
data = {'standard_value': [100, 500, 1000, 200, None, -10], 'standard_units':
['nM']*6}
df = pd.DataFrame(data)

def calculate_pIC50(standard_value):
    """Calculates pIC50 values from IC50 in nM."""
    if pd.isna(standard_value) or not isinstance(standard_value, (int, float)):
        return np.nan  # Handle missing or invalid values

    if standard_value <= 0:
        return np.nan  # Handle non-positive values

    pIC50 = -np.log10(standard_value * 1e-9)  # Convert nM to M and calculate pIC50
    return pIC50

df['pIC50'] = df['standard_value'].apply(calculate_pIC50)
print(df)
```

Example 2: Filtering by Target:

This example demonstrates how to modify the SQL query to filter by a specific ChEMBL target ID. *Modify the target_chembl_id with a real ChEMBL Target ID.*

```
-- File: ../data/chembl35_activity_data_target.csv
-- REPLACE "CHEMBL205" with an actual ChEMBL target ID

SELECT
    md.chembl_id,
    act.standard_type,
    act.standard_value,
    act.standard_units,
    act.assay_chembl_id,
    cmpd.canonical_smiles
FROM
    activities act
JOIN
    molecule_dictionary md ON act.molregno = md.molregno
JOIN
    compound_structures cmpd ON md.molregno = cmpd.molregno
JOIN
    assays ass ON act.assay_id = ass.assay_id  -- Join with assays table
WHERE
    act.standard_type = 'IC50'
    AND act.standard_units = 'nM'
    AND act.standard_value IS NOT NULL
    AND act.standard_value > 0
    AND act.standard_value::text ~ '^[0-9\.]+$'
    AND ass.tid = (SELECT tid FROM target_dictionary WHERE chembl_id = 'CHEMBL205') --
Filter by target CHEMBL_ID
LIMIT 100;
```

Example 3: Checking for Invalid SMILES:

This adds extra logging to identify problematic SMILES strings.

```
import pandas as pd
import numpy as np
from rdkit import Chem

# Simulate reading from CSV
data = {'canonical_smiles': ['CC(=O)Oc1ccccc1C(=O)O', 'invalid SMILES', 'C1CCCCC1']}
df = pd.DataFrame(data)

def generate_descriptors(smiles):
    """Generates RDKit descriptors from SMILES string with detailed logging."""
    try:
        mol = Chem.MolFromSmiles(smiles)
        if mol is None:
            print(f"ERROR: Could not parse SMILES: {smiles}") #ADD LOGGING
            return None
        descriptors = {}
        for name, function in Descriptors.descList:
            try:
                descriptors[name] = function(mol)
            except:
                descriptors[name] = np.nan # Handle descriptor calculation failures
        return descriptors
    except Exception as e:
        print(f"Error generating descriptors for SMILES: {smiles}, Error: {e}")
        return None

df['descriptors'] = df['canonical_smiles'].apply(generate_descriptors)
print(df)
```

Example 4: Feature Selection (Variance Threshold):

This demonstrates a simple feature selection technique.

```
import pandas as pd
import numpy as np
from sklearn.feature_selection import VarianceThreshold

# Simulate descriptor data
data = {'feature1': [1, 1, 1, 1, 1], 'feature2': [1, 2, 3, 4, 5], 'feature3': [5, 4,
3, 2, 1]}
X = pd.DataFrame(data)

# Apply Variance Threshold
selector = VarianceThreshold(threshold=0.5)  # Adjust threshold as needed
selector.fit(X)

# Get selected features
selected_features = X.columns[selector.get_support()]
print("Selected Features:", selected_features)

# Transform the data to keep only selected features
X_transformed = selector.transform(X)
print(X_transformed)
```

Example 5: Saving the Model:

This example demonstrates how to save your trained model for later use.

```
import pandas as pd
import numpy as np
import os
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 pickle # Import for saving models

# Define base path
base_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
data_path = os.path.join(base_path, "data")
models_path = os.path.join(base_path, "models")  # Path for saving models
```

```python
os.makedirs(models_path, exist_ok=True)  # Ensure directory exists

# Load the cleaned data
cleaned_csv_file = os.path.join(data_path, "chembl35_activity_data_cleaned.csv")
try:
    df = pd.read_csv(cleaned_csv_file)
    print(f"Cleaned data loaded successfully from {cleaned_csv_file}")
except FileNotFoundError:
    print(f"Error: File not found at {cleaned_csv_file}. Make sure you ran the first
notebook.")
    exit()

# RDKit Descriptor Generation
def generate_descriptors(smiles):
    """Generates RDKit descriptors from SMILES string."""
    try:
        mol = Chem.MolFromSmiles(smiles)
        if mol is None:
            return None  # Handle invalid SMILES
        descriptors = {}
        for name, function in Descriptors.descList:
            try:
                descriptors[name] = function(mol)
            except:
                descriptors[name] = np.nan # Handle descriptor calculation failures
        return descriptors
    except Exception as e:
        print(f"Error generating descriptors for SMILES: {smiles}, Error: {e}")
        return None

# Apply the descriptor generation to each molecule
df['descriptors'] = df['canonical_smiles'].apply(generate_descriptors)

# Handle rows where descriptor generation failed.  Remove rows with None values.
df = df.dropna(subset=['descriptors'])

# Convert descriptors to a DataFrame (important for scikit-learn)
descriptor_df = pd.DataFrame(list(df['descriptors']))

# Merge descriptors with activity data
df = pd.concat([df, descriptor_df], axis=1)

# Drop the original 'descriptors' column (optional)
df = df.drop('descriptors', axis=1)

# Data Preprocessing for Machine Learning
# Select features (descriptors) and target (pIC50)
X = df.select_dtypes(include=np.number).drop(['standard_value', 'pIC50'], axis=1)
#Drop non descriptor columns
y = df['pIC50']

# Handle missing values (important!)
X = X.replace([np.inf, -np.inf], np.nan)  # Replace infinities with NaN
X = X.dropna(axis=1, how='any') # Drop columns with remaining NaN

# 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)
```

```
# Feature Scaling (important for linear models)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Model Training (Linear Regression)
model = LinearRegression()
model.fit(X_train, y_train)

# Save the model
model_filename = os.path.join(models_path, "linear_regression_model.pkl")
scaler_filename = os.path.join(models_path, "standard_scaler.pkl") #Saving scaler

#Saving the model
pickle.dump(model, open(model_filename, 'wb'))
pickle.dump(scaler, open(scaler_filename, 'wb'))

print(f"Model saved to {model_filename}")
print(f"Scaler saved to {scaler_filename}")

# Load the model (for demonstration)
loaded_model = pickle.load(open(model_filename, 'rb'))
loaded_scaler = pickle.load(open(scaler_filename, 'rb')) #Load scaler

# Example Prediction (Predict pIC50 for a specific molecule)
def predict_pic50(smiles, model, scaler, feature_names):
    """Predicts pIC50 for a given SMILES string."""
    descriptors = generate_descriptors(smiles)
    if descriptors is None:
        return None

    #Convert to DF
    descriptor_df = pd.DataFrame([descriptors])

    # Select only the features the model was trained on.
    descriptor_df = descriptor_df[feature_names]
    descriptor_df = descriptor_df.fillna(0)  # Handle potential missing values after
feature selection

    # Scale the features
    scaled_descriptors = scaler.transform(descriptor_df)

    # Make the prediction
    predicted_pic50 = model.predict(scaled_descriptors)[0]
    return predicted_pic50

# Get feature names from the training data used in the model.
feature_names = list(X.columns) #Use the filtered X that has no NaNs

# Example SMILES string
example_smiles = 'CC(=O)Oc1ccccc1C(=O)O' # Aspirin

predicted_pic50 = predict_pic50(example_smiles, loaded_model, loaded_scaler,
feature_names) #Use loaded models

if predicted_pic50 is not None:
    print(f"Predicted pIC50 for {example_smiles}: {predicted_pic50}")
else:
    print(f"Could not predict pIC50 for {example_smiles}")
```

Important Considerations for Real-World Use:

- **Data Size:** 100 rows is sufficient for demonstration, but you'll need significantly more data for robust models. Be prepared for longer run times. Consider using techniques like parallel processing to speed up descriptor calculations.
- **ChEMBL License:** Ensure you comply with the ChEMBL license agreement.
- **Database Connection:** For production environments, use a more secure method to store database credentials (e.g., environment variables). Avoid hardcoding passwords in your code.
- **Logging:** Implement comprehensive logging to track errors, warnings, and progress. This is essential for debugging and monitoring.
- **Reproducibility:** Use a consistent environment (e.g., Docker) to ensure reproducibility. Specify the versions of all libraries used in a `requirements.txt` file.

Summary (English):

This comprehensive analysis provides a foundation for working with ChEMBL 35 data for drug discovery. It covers data extraction, cleaning, descriptor generation, model building, and prediction. The code includes error handling, and best practices for data analysis and machine learning. Remember to adapt the code to your specific needs and to explore the data thoroughly. Be mindful of database credentials, error handling, and reproducibility.

Resumen (Español):

Este análisis completo proporciona una base para trabajar con datos de ChEMBL 35 para el descubrimiento de fármacos. Cubre la extracción de datos, la limpieza, la generación de descriptores, la construcción de modelos y la predicción. El código incluye el manejo de errores y las mejores prácticas para el análisis de datos y el aprendizaje automático. Recuerde adaptar el código a sus necesidades específicas y explorar los datos a fondo. Tenga en cuenta las credenciales de la base de datos, el manejo de errores y la reproducibilidad.

Remember to install the necessary libraries: `pip install pandas rdkit scikit-learn numpy`. Good luck with your project!