PharmApp Suite

🧠 AI for Drug Discovery and Development 🧪

🧩 Topic: Unsupervised clustering of ChEMBL compounds

Okay, I understand. You're working with ChEMBL 35 data, using PostgreSQL with RDKit in a Jupyter notebook environment, and you're encountering specific errors. You want a guided analysis, code examples (SQL and Python), and solutions to your errors, all limited to 100 rows for manageable processing. Let's get this done.

Analysis Model & Guiding Principles

Our analysis will focus on a targeted investigation of ChEMBL data based on your `Topic_CheMBL_35_68` topic (which you haven't specified the actual content of, so I'll assume a general exploration for illustration). We will:

1. **Data Retrieval (SQL):** Extract relevant data from ChEMBL 35 using SQL queries. We'll address the error you're encountering during data filtering. We'll limit the result set to 100 rows.
2. **Data Loading and Preparation (Python):** Load the extracted CSV data into a Pandas DataFrame in the Jupyter Notebook.
3. **SMILES Processing (RDKit):** Use RDKit to process the SMILES strings, calculating molecular descriptors.
4. **Exploratory Data Analysis (EDA):** Perform basic EDA, including visualization and summary statistics, to understand the distribution of molecular descriptors and activity data.
5. **Error Handling:** Solve the errors that you have encountered.
6. **Example Use Cases:** Illustrate 5 example applications.

Error Analysis and Solutions

- **Error a:** `ERROR: operator does not exist: numeric ~ unknown, LINE 12: AND act.standard_value ~ '^[0-9\.]+$'`

  This error arises because you're trying to use a regular expression operator (~) with a numeric column (`act.standard_value`). PostgreSQL requires explicit casting when using regular expressions with numeric columns. We'll fix this in the SQL query by casting the `standard_value` column to a string.

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

  This error indicates that you're using an older version of scikit-learn. You have two solutions:

  - **Upgrade scikit-learn:** The recommended solution is to upgrade scikit-learn to a version that supports `squared=False`. You can do this within your Jupyter Notebook using `!pip install --upgrade scikit-learn`.
  - **Modify the Code (Workaround):** If upgrading isn't feasible, you can calculate the Root Mean Squared Error (RMSE) manually by taking the square root of the Mean Squared Error (MSE).

Code (SQL and Python)

Let's assume `Topic_CheMBL_35_68` involves investigating compounds active against a specific target (e.g., a kinase). We'll need to determine the `target_chembl_id` for that kinase. You can find this ID by browsing the ChEMBL database (either online or through SQL queries against the `target_dictionary`

table). I'll use a placeholder `CHEMBL205` for demonstration (it's a common kinase target). **Replace this with the actual target ID for your topic.**

```sql
-- SQL Query (save as ../data/chembl_35_topic_68_data.csv)
-- Connect to your PostgreSQL database (pgAdmin) and run this query.

-- Extract relevant data from ChEMBL 35, limited to 100 rows.
-- Focusing on a specific target (replace CHEMBL205 with your target ID).
SELECT
    act.molregno,
    act.standard_value,
    act.standard_units,
    act.standard_type,
    md.chembl_id,
    cs.canonical_smiles
FROM
    activities act
JOIN
    molecule_dictionary md ON act.molregno = md.molregno
JOIN
    compound_structures cs ON md.molregno = cs.molregno
WHERE
    act.target_chembl_id = 'CHEMBL205'  -- Replace with your target ID
    AND act.standard_type = 'IC50'       -- Example: Focusing on IC50 values
    AND act.standard_units = 'nM'        -- Example: Focusing on nM values
    AND act.standard_value IS NOT NULL  -- Exclude null values
    AND act.standard_value::text ~ '^[0-9\.]+$'  -- Corrected: Cast to text for regex
LIMIT 100;
```

Explanation:

- We join the `activities`, `molecule_dictionary`, and `compound_structures` tables to get activity data, molecule IDs, and SMILES strings.
- We filter by `target_chembl_id` and `standard_type` (adjust as needed).
- `AND act.standard_value::text ~ '^[0-9\.]+$'` is the corrected line. It casts the numeric `standard_value` to text before applying the regular expression.
- `LIMIT 100` ensures that we only retrieve 100 rows.
- Save the output of this query as `../data/chembl_35_topic_68_data.csv` in the correct directory within your AIMLops structure.

```python
# Python Code (Jupyter Notebook:
Topic_CheMBL_35_68_1_data_loading_and_processing.ipynb)

import pandas as pd
import os
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

# Define base path (assuming your AIMLops root is one level above)
base_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
data_path = os.path.join(base_path, "data", "chembl_35_topic_68_data.csv")

# Load the CSV data
try:
    df = pd.read_csv(data_path)
    print("Data loaded successfully.")
```

```
except FileNotFoundError:
    print(f"Error: File not found at {data_path}.  Make sure you ran the SQL query and
saved the data correctly.")
    exit()

# Data Cleaning and Preparation
df = df.dropna(subset=['canonical_smiles', 'standard_value']) # Drop rows with missing
SMILES or activity values
df['standard_value'] = pd.to_numeric(df['standard_value'], errors='coerce') # Convert
to numeric, coerce errors to NaN
df = df.dropna(subset=['standard_value']) # Drop any remaining rows with invalid
standard_value
df = df[df['standard_value'] > 0]  #Filter out the non-positive standard_value
# RDKit Molecular Descriptor Calculation
def calculate_descriptors(smiles):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None  # Handle invalid SMILES
    descriptors = {}
    descriptors["MolecularWeight"] = Descriptors.MolWt(mol)
    descriptors["LogP"] = Descriptors.MolLogP(mol)
    descriptors["HBD"] = Descriptors.NumHDonors(mol)
    descriptors["HBA"] = Descriptors.NumHAcceptors(mol)
    return descriptors

# Apply the descriptor calculation
df['descriptors'] = df['canonical_smiles'].apply(calculate_descriptors)
df = df.dropna(subset=['descriptors']) #Drop the Nones
# Convert descriptor dictionary to columns
df = pd.concat([df.drop(['descriptors'], axis=1), df['descriptors'].apply(pd.Series)],
axis=1)

#Convert IC50 to pIC50
df['pIC50'] = -np.log10(df['standard_value'] * 1e-9) # Convert nM to M and then to
pIC50

# Basic EDA
print(df.describe()) # Summary statistics
print(df.head()) # Show the first few rows

# Simple Linear Regression (Example)
X = df[['MolecularWeight', 'LogP', 'HBD', 'HBA']]
y = df['pIC50']

# Handle missing values (imputation or removal)
X = X.fillna(X.mean())  # Simple imputation with the mean

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calculate metrics
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)  #Manually calculate RMSE
r2 = r2_score(y_test, y_pred)
```

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

Explanation:

- **Import Libraries:** Imports necessary libraries.
- **Path Handling:** Uses `os.path.join` for robust path construction, crucial for your AIMLops structure.
- **Data Loading:** Loads the CSV file into a Pandas DataFrame. Includes error handling for file not found.
- **SMILES Processing:** Uses RDKit to parse SMILES strings and calculate a few basic molecular descriptors (Molecular Weight, LogP, Hydrogen Bond Donors/Acceptors). Includes handling for invalid SMILES.
- **EDA:** Prints summary statistics (`describe()`) and the first few rows (`head()`) for initial data exploration.
- **Linear Regression (Example):** Demonstrates a simple linear regression model using the calculated descriptors to predict pIC50 values. This is just a basic example; you'll likely want to explore more sophisticated modeling techniques.
- **Error Handling:** Addresses the `squared=False` error by manually calculating the RMSE.
- **NaN Handling:** Addresses any NaN value.

Example Use Cases (based on the data you're now processing):

1. **QSAR Modeling:** Develop Quantitative Structure-Activity Relationship (QSAR) models to predict the activity of new compounds against your target based on their molecular descriptors. This is an extension of the linear regression example.
2. **Virtual Screening:** Use the QSAR model to screen a large library of virtual compounds and identify promising candidates for further experimental testing.
3. **Lead Optimization:** Analyze the relationship between specific molecular descriptors and activity to guide the optimization of lead compounds. For example, if you find a strong correlation between LogP and activity, you might focus on modifying the molecule to improve its LogP.
4. **Descriptor Importance Analysis:** Determine which molecular descriptors are most important for predicting activity. This can provide insights into the structural features that are critical for binding to the target. You can do this with feature importance methods available in libraries like scikit-learn after training a model.
5. **Activity Landscape Analysis:** Visualize the activity landscape (e.g., using a scatter plot of two key descriptors, colored by activity) to identify regions of chemical space that are likely to contain highly active compounds.

Important Considerations and Next Steps:

- **Target Selection:** The most important step is to **replace `CHEMBL205` with the correct `target_chembl_id` for your specific research question.**
- **Descriptor Selection:** The descriptors I've included are just examples. Choose descriptors that are relevant to your target and hypothesis. RDKit provides a wide range of descriptors.
- **Model Selection:** Linear regression is a very basic model. Explore more sophisticated models like Random Forests, Support Vector Machines, or deep learning models.
- **Data Validation:** Always validate your models using appropriate techniques like cross-validation or a hold-out test set.
- **Logging:** Incorporate logging to track the execution of your code and identify potential errors.
- **Version Control:** Use Git for version control of your code.

Revised Code (incorporating suggested improvements)

```python
# Python Code (Jupyter Notebook:
Topic_CheMBL_35_68_1_data_loading_and_processing.ipynb)

import pandas as pd
import os
from rdkit import Chem
from rdkit.Chem import Descriptors
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import logging  # Import the logging module

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %
(message)s')

# Define base path (assuming your AIMLops root is one level above)
base_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
data_path = os.path.join(base_path, "data", "chembl_35_topic_68_data.csv")

# Load the CSV data
try:
    df = pd.read_csv(data_path)
    logging.info("Data loaded successfully.")
except FileNotFoundError:
    logging.error(f"Error: File not found at {data_path}.  Make sure you ran the SQL
query and saved the data correctly.")
    exit()

# Data Cleaning and Preparation
df = df.dropna(subset=['canonical_smiles', 'standard_value']) # Drop rows with missing
SMILES or activity values
df['standard_value'] = pd.to_numeric(df['standard_value'], errors='coerce') # Convert
to numeric, coerce errors to NaN
df = df.dropna(subset=['standard_value']) # Drop any remaining rows with invalid
standard_value
df = df[df['standard_value'] > 0]  #Filter out the non-positive standard_value
# RDKit Molecular Descriptor Calculation
def calculate_descriptors(smiles):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        logging.warning(f"Invalid SMILES encountered: {smiles}")
        return None  # Handle invalid SMILES
    descriptors = {}
    try:
        descriptors["MolecularWeight"] = Descriptors.MolWt(mol)
        descriptors["LogP"] = Descriptors.MolLogP(mol)
        descriptors["HBD"] = Descriptors.NumHDonors(mol)
        descriptors["HBA"] = Descriptors.NumHAcceptors(mol)
    except Exception as e:
        logging.error(f"Error calculating descriptors for SMILES {smiles}: {e}")
        return None
    return descriptors

# Apply the descriptor calculation
df['descriptors'] = df['canonical_smiles'].apply(calculate_descriptors)
df = df.dropna(subset=['descriptors']) #Drop the Nones
# Convert descriptor dictionary to columns
df = pd.concat([df.drop(['descriptors'], axis=1), df['descriptors'].apply(pd.Series)],
```

```
axis=1)

#Convert IC50 to pIC50
df['pIC50'] = -np.log10(df['standard_value'] * 1e-9) # Convert nM to M and then to
pIC50

# Basic EDA
logging.info(df.describe()) # Summary statistics
logging.info(df.head()) # Show the first few rows

# Simple Linear Regression (Example)
X = df[['MolecularWeight', 'LogP', 'HBD', 'HBA']]
y = df['pIC50']

# Handle missing values (imputation or removal)
X = X.fillna(X.mean())  # Simple imputation with the mean

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calculate metrics
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)  #Manually calculate RMSE
r2 = r2_score(y_test, y_pred)

logging.info(f"Mean Squared Error: {mse}")
logging.info(f"Root Mean Squared Error: {rmse}")
logging.info(f"R-squared: {r2}")

#Cross Validation
cv = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(model, X, y, cv=cv, scoring='r2')
logging.info(f"Cross-validation R^2 scores: {cv_scores}")
logging.info(f"Mean cross-validation R^2 score: {cv_scores.mean()}")
```

Key improvements:

- **Logging:** Added comprehensive logging using the `logging` module to track data loading, descriptor calculation (handling invalid SMILES), and model performance. Errors are logged as `ERROR`, warnings as `WARNING`, and informational messages as `INFO`.
- **Error Handling in Descriptor Calculation:** Wrapped the descriptor calculation in a `try...except` block to handle potential errors during the process. Invalid SMILES strings are now logged as warnings.
- **Cross-Validation:** Implemented k-fold cross-validation using `cross_val_score` to get a more robust estimate of model performance. The `KFold` object allows you to control the shuffling and random state for reproducibility.
- **Clarity:** Improved variable names and comments for better readability.

This revised code provides a more robust and informative starting point for your ChEMBL 35 data analysis project. Remember to adapt the target ID, descriptor selection, and modeling techniques to your specific research question. Good luck!

Traducción al Español:

De acuerdo, entiendo. Estás trabajando con datos de ChEMBL 35, usando PostgreSQL con RDKit en un entorno de Jupyter notebook y te encuentras con errores específicos. Quieres un análisis guiado, ejemplos de código (SQL y Python) y soluciones a tus errores, todo limitado a 100 filas para un procesamiento manejable. Vamos a hacerlo.

Modelo de Análisis y Principios Guía

Nuestro análisis se centrará en una investigación específica de los datos de ChEMBL basada en tu tema `Topic_CheMBL_35_68` (del cual no has especificado el contenido real, por lo que asumiré una exploración general a modo de ilustración). Nosotros:

1. **Recuperación de Datos (SQL):** Extraeremos datos relevantes de ChEMBL 35 utilizando consultas SQL. Abordaremos el error que estás encontrando durante el filtrado de datos. Limitaremos el conjunto de resultados a 100 filas.
2. **Carga y Preparación de Datos (Python):** Cargaremos los datos CSV extraídos en un DataFrame de Pandas en el Jupyter Notebook.
3. **Procesamiento de SMILES (RDKit):** Utilizaremos RDKit para procesar las cadenas SMILES, calculando descriptores moleculares.
4. **Análisis Exploratorio de Datos (EDA):** Realizaremos un EDA básico, incluyendo visualización y estadísticas resumidas, para comprender la distribución de los descriptores moleculares y los datos de actividad.
5. **Manejo de Errores:** Resolveremos los errores que has encontrado.
6. **Ejemplos de Casos de Uso:** Ilustraremos 5 ejemplos de aplicaciones.

Análisis de Errores y Soluciones

- **Error a:** **`ERROR: operator does not exist: numeric ~ unknown, LINE 12: AND act.standard_value ~ '^[0-9\.]+$'`**

  Este error surge porque estás intentando usar un operador de expresión regular (~) con una columna numérica (`act.standard_value`). PostgreSQL requiere una conversión explícita al usar expresiones regulares con columnas numéricas. Lo solucionaremos en la consulta SQL convirtiendo la columna `standard_value` a una cadena de texto.

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

  Este error indica que estás utilizando una versión antigua de scikit-learn. Tienes dos soluciones:

  - **Actualizar scikit-learn:** La solución recomendada es actualizar scikit-learn a una versión que admita `squared=False`. Puedes hacerlo dentro de tu Jupyter Notebook usando `!pip install --upgrade scikit-learn`.
  - **Modificar el Código (Solución Temporal):** Si la actualización no es factible, puedes calcular el Error Cuadrático Medio Raíz (RMSE) manualmente tomando la raíz cuadrada del Error Cuadrático Medio (MSE).

Código (SQL y Python)

Asumamos que `Topic_CheMBL_35_68` implica investigar compuestos activos contra un objetivo específico (por ejemplo, una quinasa). Necesitaremos determinar el `target_chembl_id` para esa quinasa. Puedes encontrar este ID navegando por la base de datos ChEMBL (ya sea en línea o a través de consultas SQL contra la tabla `target_dictionary`). Usaré un marcador de posición `CHEMBL205` para la demostración (es un objetivo de quinasa común). **Reemplaza esto con el ID de objetivo real para tu tema.**

```
-- Consulta SQL (guardar como ../data/chembl_35_topic_68_data.csv)
-- Conéctate a tu base de datos PostgreSQL (pgAdmin) y ejecuta esta consulta.
```

```sql
-- Extrae datos relevantes de ChEMBL 35, limitado a 100 filas.
-- Centrándose en un objetivo específico (reemplaza CHEMBL205 con tu ID de objetivo).
SELECT
    act.molregno,
    act.standard_value,
    act.standard_units,
    act.standard_type,
    md.chembl_id,
    cs.canonical_smiles
FROM
    activities act
JOIN
    molecule_dictionary md ON act.molregno = md.molregno
JOIN
    compound_structures cs ON md.molregno = cs.molregno
WHERE
    act.target_chembl_id = 'CHEMBL205'  -- Reemplaza con tu ID de objetivo
    AND act.standard_type = 'IC50'       -- Ejemplo: Centrándose en valores IC50
    AND act.standard_units = 'nM'        -- Ejemplo: Centrándose en valores nM
    AND act.standard_value IS NOT NULL  -- Excluye valores nulos
    AND act.standard_value::text ~ '^[0-9\.]+$'  -- Corregido: Convierte a texto para
la expresión regular
LIMIT 100;
```

Explicación:

- Unimos las tablas `activities`, `molecule_dictionary` y `compound_structures` para obtener datos de actividad, IDs de moléculas y cadenas SMILES.
- Filtramos por `target_chembl_id` y `standard_type` (ajusta según sea necesario).
- `AND act.standard_value::text ~ '^[0-9\.]+$'` es la línea corregida. Convierte el `standard_value` numérico a texto antes de aplicar la expresión regular.
- `LIMIT 100` asegura que solo recuperemos 100 filas.
- Guarda la salida de esta consulta como `../data/chembl_35_topic_68_data.csv` en el directorio correcto dentro de tu estructura AIMLops.

```python
# Código Python (Jupyter Notebook:
Topic_CheMBL_35_68_1_data_loading_and_processing.ipynb)

import pandas as pd
import os
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

# Define la ruta base (asumiendo que la raíz de tu AIMLops está un nivel por encima)
base_path = os.path.abspath(os.path.join(os.getcwd(), ".."))
data_path = os.path.join(base_path, "data", "chembl_35_topic_68_data.csv")

# Carga los datos CSV
try:
    df = pd.read_csv(data_path)
    print("Datos cargados con éxito.")
except FileNotFoundError:
    print(f"Error: No se encontró el archivo en {data_path}. Asegúrate de haber
ejecutado la consulta SQL y guardado los datos correctamente.")
    exit()
```

```
# Limpieza y Preparación de Datos
df = df.dropna(subset=['canonical_smiles', 'standard_value']) # Elimina las filas con
SMILES o valores de actividad faltantes
df['standard_value'] = pd.to_numeric(df['standard_value'], errors='coerce') #
Convierte a numérico, fuerza los errores a NaN
df = df.dropna(subset=['standard_value']) # Elimina cualquier fila restante con
standard_value no válido
df = df[df['standard_value'] > 0]  # Filtra los standard_value no positivos

# Cálculo de Descriptores Moleculares con RDKit
def calculate_descriptors(smiles):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None  # Maneja SMILES no válidos
    descriptors = {}
    descriptors["MolecularWeight"] = Descriptors.MolWt(mol)
    descriptors["LogP"] = Descriptors.MolLogP(mol)
    descriptors["HBD"] = Descriptors.NumHDonors(mol)
    descriptors["HBA"] = Descriptors.NumHAcceptors(mol)
    return descriptors

# Aplica el cálculo de descriptores
df['descriptors'] = df['canonical_smiles'].apply(calculate_descriptors)
df = df.dropna(subset=['descriptors']) #Elimina los Nones

# Convierte el diccionario de descriptores a columnas
df = pd.concat([df.drop(['descriptors'], axis=1), df['descriptors'].apply(pd.Series)],
axis=1)

# Convierte IC50 a pIC50
df['pIC50'] = -np.log10(df['standard_value'] * 1e-9) # Convierte nM a M y luego a
pIC50

# EDA Básico
print(df.describe()) # Estadísticas resumidas
print(df.head()) # Muestra las primeras filas

# Regresión Lineal Simple (Ejemplo)
X = df[['MolecularWeight', 'LogP', 'HBD', 'HBA']]
y = df['pIC50']

# Maneja valores faltantes (imputación o eliminación)
X = X.fillna(X.mean())  # Imputación simple con la media

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calcula las métricas
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)  # Calcula RMSE manualmente
r2 = r2_score(y_test, y_pred)

print(f"Error Cuadrático Medio: {mse}")
print(f"Error Cuadrático Medio Raíz: {rmse}")
print(f"R-cuadrado: {r2}")
```

Explicación:

- **Importa Librerías:** Importa las librerías necesarias.
- **Manejo de Rutas:** Utiliza `os.path.join` para una construcción robusta de rutas, crucial para tu estructura AIMLops.
- **Carga de Datos:** Carga el archivo CSV en un DataFrame de Pandas. Incluye manejo de errores para el caso de que el archivo no se encuentre.
- **Procesamiento de SMILES:** Utiliza RDKit para analizar cadenas SMILES y calcular algunos descriptores moleculares básicos (Peso Molecular, LogP, Donadores/Aceptores de Enlaces de Hidrógeno). Incluye manejo para SMILES no válidos.
- **EDA:** Imprime estadísticas resumidas (`describe()`) y las primeras filas (`head()`) para una exploración inicial de los datos.
- **Regresión Lineal (Ejemplo):** Demuestra un modelo de regresión lineal simple utilizando los descriptores calculados para predecir los valores de pIC50. Este es solo un ejemplo básico; es probable que desees explorar técnicas de modelado más sofisticadas.
- **Manejo de Errores:** Aborda el error `squared=False` calculando manualmente el RMSE.
- **Manejo de NaN:** Aborda cualquier valor NaN.

Ejemplos de Casos de Uso (basados en los datos que ahora estás procesando):

1. **Modelado QSAR:** Desarrolla modelos de Relación Cuantitativa Estructura-Actividad (QSAR) para predecir la actividad de nuevos compuestos contra tu objetivo basándote en sus descriptores moleculares. Esta es una extensión del ejemplo de regresión lineal.
2. **Cribado Virtual:** Utiliza el modelo QSAR para cribar una gran biblioteca de compuestos virtuales e identificar candidatos prometedores para pruebas experimentales adicionales.
3. **Optimización de Leads:** Analiza la relación entre descriptores moleculares específicos y la actividad para guiar la optimización de compuestos líderes. Por ejemplo, si encuentras una fuerte correlación entre LogP y la actividad, podrías centrarte en modificar la molécula para mejorar su LogP.
4. **Análisis de Importancia de Descriptores:** Determina qué descriptores moleculares son más importantes para predecir la actividad. Esto puede proporcionar información sobre las características estructurales que son críticas para la unión al objetivo. Puedes hacer esto con métodos de importancia de características disponibles en bibliotecas como scikit-learn después de entrenar un modelo.
5. **Análisis del Paisaje de Actividad:** Visualiza el paisaje de actividad (por ejemplo, utilizando un diagrama de dispersión de dos descriptores clave, coloreados por actividad) para identificar regiones del espacio químico que son propensas a contener compuestos altamente activos.

Consideraciones Importantes y Próximos Pasos:

- **Selección del Objetivo:** El paso más importante es **reemplazar `CHEMBL205` con el `target_chembl_id` correcto para tu pregunta de investigación específica.**
- **Selección de Descriptores:** Los descriptores que he incluido son solo ejemplos. Elige descriptores que sean relevantes para tu objetivo e hipótesis. RDKit proporciona una amplia gama de descriptores.
- **Selección de Modelos:** La regresión lineal es un modelo muy básico. Explora modelos más sofisticados como Random Forests, Support Vector Machines o modelos de aprendizaje profundo.
- **Validación de Datos:** Valida siempre tus modelos utilizando técnicas apropiadas como la validación cruzada o un conjunto de pruebas de retención.
- **Registro:** Incorpora el registro para realizar un seguimiento de la ejecución de tu código e identificar posibles errores.
- **Control de Versiones:** Utiliza Git para el control de versiones de tu código.

Código Revisado (incorporando mejoras sugeridas)

```python # Código Python (Jupyter Notebook: Topic_CheMBL_35_68_1_data_loading_and_processing.ipynb)

import pandas as pd import os from rdkit import Chem from rdkit.Chem import Descriptors import numpy as np from sklearn.model_selection import train_test_split, cross_val_score, KFold from sklearn.linear_model import LinearRegression from sklearn.metrics import mean_squared_error, r2_score import logging # Importa el módulo de registro

## Configura el registro

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

## Define la ruta base (asumiendo que la raíz de tu AIMLops está un nivel por encima)

base_path = os.path.abspath(os.path.join(os.getcwd(), "..")) data_path = os.path.join(base_path, "data", "chembl_35_topic_68_data.csv")

## Carga los datos CSV

try: df = pd.read_csv(data_path) logging.info("Datos cargados con éxito.") except FileNotFoundError: logging.error(f"Error: No se encontró el archivo en {data_path}. Asegúrate de haber ejecutado la consulta SQL y guardado los datos correctamente.") exit()

## Limpieza y Preparación de Datos

df = df.dropna(subset=['canonical_smiles', 'standard_value']) # Elimina las filas con SMILES o valores de actividad faltantes df['standard_value'] = pd.to_numeric(df['standard_value'], errors='coerce') # Convierte a numérico, fuerza los errores a NaN df = df.dropna(subset=['standard_value']) # Elimina cualquier fila restante con standard_value no válido df = df[df['standard_value'] > 0] # Filtra los standard_value no positivos

## Cálculo de Descriptores Moleculares con RDKit

def calculate_descriptors(smiles): mol = Chem.MolFromSmiles(smiles) if mol is None: logging.warning(f"SMILES no válido encontrado: {smiles}") return None # Maneja SMILES no válidos descriptors = {} try: descriptors["MolecularWeight"] = Descriptors.MolWt(mol) descriptors["LogP"] = Descriptors.MolLogP(mol) descriptors["HBD"] = Descriptors.NumHDonors(mol) descriptors["HBA"] = Descriptors.NumHAcceptors(mol) except Exception as e: logging.error(f"Error al calcular los descriptores para SMILES {smiles}: {e}") return None return descriptors

## Aplica el cálculo de descriptores

df['descriptors'] = df['canonical_smiles'].apply(calculate_descriptors) df = df.dropna(subset=['descriptors']) #Elimina los Nones

## Convierte el diccionario de descriptores a columnas

df = pd.concat([df.drop(['descriptors'], axis=1), df['descriptors'].apply(pd.Series)], axis=1)

## Convierte IC50 a pIC50

df['pIC50'] = -np.log10(df['standard_value'] * 1e-9) # Convierte nM a M y luego a pIC50
```

EDA Básico

logging.info(df.describe()) # Estadísticas resumidas logging.info(df.head()) # Muestra las primeras filas

Regresión Lineal Simple (Ejemplo)

X = df[['MolecularWeight', 'LogP', 'HBD', 'HBA']] y = df['pIC50']

Maneja valores faltantes (imputación o eliminación)

X = X.fillna(X.mean()) # Imputación simple con la media

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = LinearRegression() model.fit(X_train, y_train) y_pred = model.predict(X_test)

Calcula las métricas

mse = mean_squared_error(y_test, y_pred) rmse = np.sqrt(mse) # Calcula RMSE manualmente r2 = r2_score(y_test, y_pred)

logging.info(f"Error Cuadrático Medio: {mse}") logging.info(f"Error