Etapa 1: Representação de Entrada e Pré-processamento

Relação com o Algoritmo: Esta etapa corresponde à caixa "(1) Input representation & preprocessing" da Figura 1. O objetivo é carregar todos os dados brutos: informações sobre fármacos, alvos e suas afinidades de ligação.

Adaptação para o Dataset ChEMBL:

- **Valores de Afinidade:** O dataset Davis usava a constante de dissociação (Kd), onde um valor menor significa uma ligação mais forte. Isso exigia a aplicação da Equação (1) $pKd = -log10(Kd/1e9)$ para inverter e normalizar a escala. Como o dataset ChEMBL usa o pchembl_value essa transformação não é necessária, pois o valor já tem o significado desejado (maior = mais forte).
- **Transformação em Peso de Aresta:** A **Equação (2)** ($f(z)i = e - \alpha zi$) no artigo original servia para converter os valores de afinidade em um peso de aresta normalizado (entre 0 e 1). No novo script, essa conversão é feita usando o `MinMaxScaler` do Scikit-learn. O objetivo final é o mesmo: criar um grafo ponderado onde os pesos das arestas representam a força da ligação.

Etapa 2: Técnicas de Embedding e Construção do Grafo

- **Relação com o Algoritmo:** Esta etapa corresponde à caixa "(2) Embedding techniques & graph construction". O script utiliza os embeddings carregados para construir as matrizes de similaridade que formarão parte da rede heterogênea.
- **Adaptação e Equações:**
  - O script implementa diretamente a Equação (4) ($CosSim(vi, vj)$) ao usar a função cosine_similarity sobre os vetores de embedding para criar as matrizes DD_Sim_matrix e TT_Sim_matrix. Isso corresponde à criação do grafo "G2" do artigo, que é baseado em embeddings.
  - A construção do grafo de afinidade (affinity_graph_weights_train) é feita ao preencher uma matriz com os pesos de aresta normalizados (obtidos na Etapa 1).

Etapa 3: Extração de Características (Modelo Híbrido)

- **Relação com o Algoritmo:** Esta é uma implementação direta da caixa "(3) Three feature extraction models", focando no Affinity2Vec_Hybrid model, que combina as características de meta-caminho e de embedding.
- **Adaptação e Equações:**

- Características de Meta-Caminho (Pscores): O script calcula os 12 scores de meta-caminho (6 de soma, 6 de máximo) chamando as funções em OptimizedpathScores_functions.py. Essas funções implementam a lógica das **Equações (6), (7) e (8)**: calcular o score de um caminho como o produto dos pesos das arestas e depois agregar os scores de todos os caminhos de um mesmo tipo usando SumScore e MaxScore.

Equação (6):

Esta equação define o score de um único caminho como o produto dos pesos de suas arestas.

Relação com o Código: Esta equação não é implementada diretamente como uma função separada. Em vez disso, o cálculo do produto é um resultado implícito da multiplicação de matrizes. Por exemplo, ao calcular o score para o caminho D-D-T, o código multiplica a matriz de similaridade Dsim pela matriz de afinidade DT. O produto $Dsim[i, j] * DT[j, k]$ representa o score de um caminho específico do fármaco i para o alvo k passando pelo fármaco intermediário j.

Equação (7):

Esta equação agrega os scores de todos os caminhos possíveis entre um par de nós (fármaco, alvo) para um determinado tipo de meta-caminho (ex: D-D-T).

- **Relação com o Código (Original):** No arquivo pathScores_functions.py, o SumScore é calculado em duas etapas na função metaPath_Dsim_DT:

1. `m = np.einsum('ij,jk->ijk', Dsim, DT`: Cria um tensor 3D (m) onde cada elemento $m[i, j, k]$ contém o produto $Dsim[i, j] * DT[j, k]$.
2. `sumM = np.sum((m[:, :, None] ), axis = 1)` : Soma todos os produtos ao longo do eixo intermediário j, resultando exatamente no SumScore.

- **Relação com o Código (Otimizado):** No arquivo OptimizedpathScores_functions.py, o cálculo é mais direto e eficiente:
- `sumM = np.dot(Dsim, DT)`: A multiplicação de matrizes padrão (np.dot) já calcula a soma dos produtos por definição. O resultado sumM[i, k] é diretamente o SumScore de todos os caminhos D-D-T entre o fármaco i e o alvo k.

Equação (8):

Esta equação encontra o caminho de maior pontuação entre um par de nós para um determinado tipo de meta-caminho.

- **Relação com o Código (Original):** Assim como no SumScore, o script original primeiro cria o tensor 3D com todos os produtos de score:

1. `m = np.einsum('ij,jk->ijk', Dsim, DT)`
2. `maxM = np.max((m[:, :, None]), axis = 1)` : Em seguida, ele encontra o valor máximo ao longo do eixo intermediário j, resultando no MaxScore. Desvantagem: Criar o tensor 3D consome uma quantidade enorme de memória.

- **Relação com o Código (Otimizado):** A versão otimizada evita o tensor 3D para economizar memória, usando um loop e broadcasting:
  maxM = np.zeros((num_drugs, num_targets), dtype=np.float32)
  for i in range(num_drugs):
  # Broadcasting multiplica a linha 'i' de Dsim por toda a matriz DT
  temp_matrix = Dsim[i, :][:, np.newaxis] * DT
  # Encontra o máximo ao longo do eixo intermediário 'j' (axis=0)
  maxM[i, :] = np.max(temp_matrix, axis=0) Este loop calcula os scores para um fármaco de partida de cada vez, encontra o máximo e o armazena, sem nunca precisar manter todos os scores na memória simultaneamente.

Etapa 4: Modelo de Regressão e Avaliação:

Relação com o Algoritmo: Corresponde à caixa "(4) ML regression model". O script utiliza as características híbridas geradas para treinar um modelo de machine learning e prever a afinidade.

- **Adaptação e Equações:**
  - O regressor XGBoost é configurado com o objetivo de minimizar o erro quadrático (objective='reg:squarederror'), o que é uma implementação direta do princípio da **Equação (9)** (MSE).
- Modificações em Relação ao Affinity2Vec_Davis.py:
  - O modelo de regressão (XGBoost) é o mesmo.
  - A principal diferença é a metodologia de avaliação. O script Davis usava um loop de validação cruzada com folds pré-definidos. O novo script usa uma única divisão de treino/teste, também pré-definida nos arquivos train_val_idx.txt e test_idx.txt, tornando o experimento mais direto e focado em uma única execução.