以下は、アプローチ1（レイアウト構造をグラフ化した深層学習手法）の概念実装例である。
ここでは以下を想定する：

- **目的**：
  書類のレイアウト（要素間の位置関係）をグラフとして表現し、そのグラフから埋め込みベクトルを得ることで、類似したレイアウトを持つ書類同士を類似ベクトルとしてマッピングする。

- **必要なデータ**：
  - 各書類ごとに、レイアウト要素(例：テキストブロック、画像ボックス、表領域など)が検出済みであること。
  - 各要素は少なくとも次の情報を持つ想定：
    - 要素の種類（"text_block"、"image"、"table"など）
    - 要素の座標情報（bounding box: x, y, width, height）
  - 書類全体から要素間の位置的関係を元にグラフを構築する手順が可能であること。
    例えば、要素同士が近接している場合にエッジを張る、同じ水平・垂直軸上に並んでいる場合にエッジを張るなど、ヒューリスティックは様々。ここでは近接距離が一定以下であればエッジを張る簡易な例とする。

- **処理フロー**：
  1. 書類のレイアウト要素情報（JSONなど）をロードする。
  2. 要素ごとにノードを生成し、ノード特徴量（座標・要素種別など）を埋め込みベクトルに変換。
  3. 要素間を繋ぐエッジ（例：ユークリッド距離が近いノード同士）を生成。
  4. PyTorch Geometric等のフレームワークでGNN（Graph Neural Network、ここではGraph Convolutional Networkの例）を定義し、グラフから文書レベルの埋め込み（プーリング）を計算。
  5. 得られた埋め込みを使って、類似文書検索を行う。（ここでは類似検索までは行わず、埋め込みを出力するところまで）

- **環境依存パッケージ**：
  - Python 3.9.7
  - `torch` および `torch-geometric`

    ```
    pip install torch==1.10.0  # 例
    pip install torch-geometric
    ```

  - その他標準ライブラリ

以下に、Jupyter Notebook上で実行可能なサンプルコードを提示する。
（実データは用意しにくいので、ランダムに生成したダミーデータを使用する。）

```
import json
import math
import random
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data, DataLoader
from torch_geometric.nn import global_mean_pool, GCNConv

# ===== ダミーデータ生成部 =====
# 以下はあくまでサンプルとして、5つのドキュメントをランダム生成する。
# 実際には、ここでユーザが用意したJSON等のレイアウト分析済みデータを用いる。

def generate_dummy_document(num_elements=10):
    # 各要素は、(x, y, w, h, type)を持つ想定
    # typeは簡易的に0=テキスト,1=画像,2=表 などのカテゴリIDで表現
    doc = []
    for _ in range(num_elements):
        x = random.uniform(0, 1000)
        y = random.uniform(0, 1000)
        w = random.uniform(50, 200)
        h = random.uniform(20, 100)
        elem_type = random.randint(0, 2)  # 3種類の要素
        doc.append({
            "x": x,
            "y": y,
            "w": w,
            "h": h,
            "type": elem_type
        })
    return doc

num_documents = 5
documents = [generate_dummy_document(num_elements=10) for _ in
range(num_documents)]

# ===== グラフ化処理 =====
# ノード特徴量は (x, y, w, h, type) を単純な数値ベクトル化する。
# エッジは要素間の中心座標距離が一定以下なら接続するものとする。

def build_graph_from_document(doc, distance_threshold=300.0):
    # doc: [{"x":..., "y":..., "w":..., "h":..., "type":...}, ...]
    # ノード特徴量行列、エッジindexを作成
    xs = []
    for elem in doc:
        x_center = elem["x"] + elem["w"]/2.0
        y_center = elem["y"] + elem["h"]/2.0
        elem_type = elem["type"]
        # 特徴量： (x_center, y_center, w, h, type)
        xs.append([x_center, y_center, elem["w"], elem["h"], float(elem_type)])
    xs = torch.tensor(xs, dtype=torch.float)

    # エッジ生成：距離がthreshold以下ならエッジ
    # O(N^2)ですがドキュメント内要素数が小さいことを仮定
    indices = []
    n = xs.size(0)
    for i in range(n):
```

```
            for j in range(i+1, n):
                dist = math.dist(xs[i,0:2].tolist(), xs[j,0:2].tolist())
                if dist < distance_threshold:
                    indices.append([i, j])
                    indices.append([j, i])
        if len(indices) > 0:
            edge_index = torch.tensor(indices, dtype=torch.long).t().contiguous()
        else:
            # 要素が全く接続されない場合にも対応
            edge_index = torch.empty((2,0), dtype=torch.long)

        # PyTorch GeometricのDataオブジェクトに格納
        data = Data(x=xs, edge_index=edge_index)
        return data

    graph_dataset = [build_graph_from_document(doc) for doc in documents]

    # ===== GNNモデル定義 =====
    class LayoutGNN(nn.Module):
        def __init__(self, in_channels=5, hidden_channels=64, out_channels=128):
            super(LayoutGNN, self).__init__()
            self.conv1 = GCNConv(in_channels, hidden_channels)
            self.conv2 = GCNConv(hidden_channels, hidden_channels)
            self.lin = nn.Linear(hidden_channels, out_channels)

        def forward(self, x, edge_index, batch):
            # x: [num_nodes, in_channels]
            # edge_index: [2, num_edges]
            # batch: [num_nodes] (どのノードがどのグラフ所属か)
            x = self.conv1(x, edge_index)
            x = F.relu(x)
            x = self.conv2(x, edge_index)
            x = F.relu(x)
            # グラフレベルの埋め込みを得るためpooling
            x = global_mean_pool(x, batch)  # [num_graphs, hidden_channels]
            x = self.lin(x)
            return x

    # 本来は大量の文書で学習するが、ここでは学習なしでforwardだけ行う。
    model = LayoutGNN()

    # DataLoaderを使う場合
    loader = DataLoader(graph_dataset, batch_size=2, shuffle=False)

    model.eval()
    with torch.no_grad():
        embeddings = []
        for batch in loader:
            # batchはDataLoaderで複数Graphを束ねたBatchオブジェクト
            out = model(batch.x, batch.edge_index, batch.batch)
            embeddings.append(out)
        embeddings = torch.cat(embeddings, dim=0)

    print("Document embeddings shape:", embeddings.shape)
```

```
print("Document embeddings:", embeddings)

# これで各文書（グラフ）を128次元の埋め込み空間にマッピングした。
# embeddings[i]がi番目のドキュメントのレイアウト埋め込みベクトルとなる。
# 類似度判定は例えばコサイン類似度で計算可能:
similarities = F.cosine_similarity(embeddings[0].unsqueeze(0), embeddings[1:],
dim=1)
print("Similarity between doc0 and others:", similarities)
```

データの詳細例

実際には、下記のようなJSON形式の入力データを想定できる。（例）

```
[
  {
    "x": 100.0,
    "y": 200.0,
    "w": 150.0,
    "h": 50.0,
    "type": "text_block"
  },
  {
    "x": 300.0,
    "y": 210.0,
    "w": 100.0,
    "h": 30.0,
    "type": "image"
  },
  ...
]
```

※上記の例では"type"は文字列だが、実装側ではカテゴリIDなど数値にマッピングする。
また、このJSONは事前にレイアウト解析（OCRや版面解析、表領域抽出ツールなど）によって得られた結果とする。

ポイント

- 実務では大量のドキュメントデータセットを用意し、これらのグラフデータに対してモデルを学習させる必要がある。
- 学習時には、同一レイアウトのドキュメントペアを正例、異なるレイアウトを負例としてコントラスト学習を行うなどの工夫でモデルを訓練できる。
- 埋め込みを得た後は、FAISSなどを使って大規模データから類似文書を高速検索することも可能。

上記はあくまで例示的なコードであり、実際にはデータ前処理、モデルチューニング、学習戦略設計などが必要となるが、基本的な骨子としてはこのような実装が考えられる。