Part 1: Profiling

First, we save the code as a .py file and make the following change to prevent it from being killed due to a high number of processes:

```python
df = pd.concat([us_df, ca_df], ignore_index=True).sample(1000, random_state=42).reset_index(drop=True)
```

Then, we run the command below to analyze the time per function call:

python3 -m cProfile -s tottime ECS_CA4_P3.py > cProfile_stats.txt

Open

cProfile_stats.txt
~/Desktop/ECS/CA4/P3

```
[EMBEDDING][INFO]: Loading model all-MiniLM-L6-v2...
[EMBEDDING][INFO]: Embedding column 'title'...
[EMBEDDING][INFO]: Embedding column 'tags'...
[EMBEDDING][INFO]: Embedding column 'description'...
Preprocessing complete. Final shape: (1000, 1181)
         2362319229 function calls (2358581797 primitive calls) in 1044.928 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1182000  479.215    0.000  857.512    0.001 managers.py:958(fast_xs)
1400364159  214.585    0.000  234.423    0.000 blocks.py:1319(iget)
677174536   50.289    0.000   50.289    0.000 blocks.py:268(mgr_locs)
  1182000   47.838    0.000   47.839    0.000 managers.py:984(<listcomp>)
     3552   44.780    0.013   44.780    0.013 {built-in method torch._C._nn.linear}
        1   16.498   16.498 1044.941 1044.941 ECS_CA4_P3.py:1(<module>)
  1183159   10.807    0.000   32.248    0.000 cast.py:1442(find_common_type)
  1183927    8.651    0.000    8.651    0.000 {built-in method fromkeys}
      576    6.752    0.012    6.752    0.012 {built-in method torch._C._nn.scaled_dot_product_attention}
44564836/44561698    6.618    0.000    9.846    0.000 {built-in method builtins.isinstance}
  1182007    6.319    0.000  883.650    0.001 frame.py:3988(_ixs)
  1180000    5.877    0.000   10.255    0.000 datetimes.py:547(_box_func)
  2321159    4.674    0.000    6.957    0.000 managers.py:1012(iget)
  2320000    4.578    0.000   38.979    0.000 indexing.py:2529(__setitem__)
  1163155    4.272    0.000    4.472    0.000 cast.py:1401(np_find_common_type)
  3505332    4.030    0.000    4.473    0.000 base.py:3784(get_loc)
  1182039    3.896    0.000    5.676    0.000 generic.py:6255(__finalize__)
  1183152    3.740    0.000  898.087    0.001 indexing.py:1719(_getitem_axis)
       21    3.657    0.174    3.657    0.174 {method 'read' of '_ssl._SSLSocket' objects}
15771253/12193957    3.467    0.000    5.311    0.000 {built-in method builtins.len}
  2320000    3.450    0.000   30.047    0.000 frame.py:4545(_set_value)
  2322312    3.382    0.000    4.432    0.000 cast.py:1778(np_can_hold_element)
  1190222    3.360    0.000    3.360    0.000 {built-in method numpy.empty}
  1180005    3.269    0.000    3.269    0.000 {method 'view' of 'numpy.generic' objects}
      576    3.235    0.006    3.235    0.006 {built-in method torch._C._nn.gelu}
  3547242    3.173    0.000    3.184    0.000 generic.py:6320(__setattr__)
  1180004    3.106    0.000    3.221    0.000 utils.py:419(check_array_indexer)
  1180004    3.027    0.000   17.422    0.000 _mixins.py:278(__getitem__)
  1183000    2.656    0.000   12.201    0.000 series.py:1104(__getitem__)
  1183152    2.442    0.000  902.865    0.001 indexing.py:1176(__getitem__)
  2321152    2.404    0.000   22.459    0.000 managers.py:1298(column_setitem)
  2320000    2.353    0.000    9.168    0.000 base.py:341(setitem_inplace)
  1184351    2.286    0.000    3.964    0.000 blocks.py:2789(new_block)
  1183234    2.275    0.000    2.949    0.000 generic.py:278(__init__)
  1180004    2.224    0.000   19.839    0.000 datetimelike.py:390(__getitem__)
  2320000    2.212    0.000   43.243    0.000 indexing.py:2577(__setitem__)
2402361/2399878    2.107    0.000    5.814    0.000 {built-in method builtins.any}
  1183000    2.005    0.000    7.041    0.000 series.py:1229(_get_value)
  4684000    1.928    0.000    2.874    0.000 managers.py:291(arrays)
  2320000    1.924    0.000   11.833    0.000 managers.py:2021(setitem_inplace)
  7009635    1.912    0.000    2.698    0.000 common.py:372(apply_if_callable)
  6960000    1.825    0.000    3.271    0.000 indexing.py:2531(<genexpr>)
  4684000    1.712    0.000    4.586    0.000 base.py:332(array)
  1182007    1.683    0.000    8.181    0.000 frame.py:678(_constructor_sliced_from_mgr)
  2369617    1.645    0.000    2.681    0.000 indexing.py:2765(check_dict_or_set_indexers)
  7110465    1.548    0.000    2.561    0.000 cast.py:1472(<genexpr>)
  4757517    1.462    0.000    1.902    0.000 generic.py:37(_check)
```

2.36 billion function calls happened in 1045 seconds. managers.py:958(fast_xs) (used in DataFrame access, especially .iloc rows) took 479.215 seconds to run for 1182000 times and is the most time-consuming function.

Then, we run the command below to analyze the time per line:

kernprof -l -v ECS_CA4_P3.py

```
Preprocessing complete. Final shape: (1000, 1181)
Wrote profile results to ECS_CA4_P3.py.lprof
Timer unit: 1e-06 s

Total time: 1866.92 s
File: ECS_CA4_P3.py
Function: run_pipeline at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           @profile
     9                                           def run_pipeline():
    10         1          2.0      2.0      0.0      TEXTUAL_COLUMNS = ["title", "tags", "description"]
    11         1          1.0      1.0      0.0      EMBEDDING_MODEL = "all-MiniLM-L6-v2"
    12         1          1.0      1.0      0.0      EMBEDDING_DIM = 384
    13         1          1.0      1.0      0.0      OUTPUT_DIR = "tmp/embeddings/"
    14         1         26.0     26.0      0.0      os.makedirs(OUTPUT_DIR, exist_ok=True)
    15
    16         1     396619.0 396619.0      0.0      us_df = pd.read_csv("USvideos.csv")
    17         1       1264.0   1264.0      0.0      us_df["country"] = "US"
    18
    19         1     486403.0 486403.0      0.0      ca_df = pd.read_csv("CAvideos.csv")
    20         1        604.0    604.0      0.0      ca_df["country"] = "CA"
    21
    22         1       8895.0   8895.0      0.0      df = pd.concat([us_df, ca_df], ignore_index=True).sample(1000, random_state=42).reset_index(drop=True)
    23
    24         1         64.0     64.0      0.0      print(f"[EMBEDDING][INFO]: Loading model {EMBEDDING_MODEL}...")
    25         1    4592619.0 4592619.0      0.2       model = SentenceTransformer(EMBEDDING_MODEL)
    26
    27         1          4.0      4.0      0.0      def clean_tags(text):
    28                                                   return " ".join(tag.replace('"', '') for tag in str(text).split('|'))
    29
    30         4         11.0      2.8      0.0      for col in TEXTUAL_COLUMNS:
    31         3        247.0     82.3      0.0          print(f"[EMBEDDING][INFO]: Embedding column '{col}'...")
    32         3          6.0      2.0      0.0          if col == "tags":
    33         1       8191.0   8191.0      0.0              text_data = df[col].fillna("").apply(clean_tags).tolist()
    34                                                   else:
    35         2       2449.0   1224.5      0.0              text_data = df[col].fillna("").astype(str).tolist()
    36
    37         3   54372205.0 18124068.3      2.9           emb = model.encode(text_data, show_progress_bar=True, batch_size=32)
    38         3       1770.0    590.0      0.0          emb_df = pd.DataFrame(emb, columns=[f"{col}_emb_{i}" for i in range(emb.shape[1])])
    39         3       6551.0   2183.7      0.0          df = pd.concat([df.reset_index(drop=True), emb_df], axis=1)
    40
    41         1          2.0      2.0      0.0      def count_tags_loop(tag_str):
    42                                                   if pd.isna(tag_str):
    43                                                       return 0
    44                                                   count = 0
    45                                                   for tag in tag_str.split("|"):
    46                                                       if tag.strip() != "":
    47                                                           count += 1
    48                                                   return count
    49
    50         1          1.0      1.0      0.0      tag_counts = []
```

```
51      1001      1453.0      1.5      0.0      for i in range(len(df)):
52      1000    898718.0    898.7      0.0          tag_counts.append(count_tags_loop(df.iloc[i]["tags"]))
53         1       893.0    893.0      0.0      df["tag_count"] = tag_counts
54
55         1         2.0      2.0      0.0      publish_dates = []
56         1         1.0      1.0      0.0      publish_hours = []
57      1001      1101.0      1.1      0.0      for i in range(len(df)):
58      1000      1032.0      1.0      0.0          try:
59      1000    922827.0    922.8      0.0              dt = datetime.strptime(df.iloc[i]["publish_time"], "%Y-%m-%dT%H:%M:%S.%fZ")
60      1000      1482.0      1.5      0.0              publish_dates.append(dt)
61      1000      1331.0      1.3      0.0              publish_hours.append(dt.hour)
62                                                  except Exception:
63                                                      publish_dates.append(pd.NaT)
64                                                      publish_hours.append(np.nan)
65
66         1      3991.0   3991.0      0.0      df["publish_time"] = publish_dates
67         1       571.0    571.0      0.0      df["publish_hour"] = publish_hours
68
69         4         5.0      1.2      0.0      for col in TEXTUAL_COLUMNS:
70         3       157.0     52.3      0.0          if col in df.columns:
71         3      1153.0    384.3      0.0              del df[col]
72
73         1         1.0      1.0      0.0      engagement_rates = []
74         1         1.0      1.0      0.0      ratios = []
75      1001      1126.0      1.1      0.0      for i in range(len(df)):
76      1000    893254.0    893.3      0.0          row = df.iloc[i]
77      1000     10914.0     10.9      0.0          views = row["views"]
78      1000      6724.0      6.7      0.0          likes = row["likes"]
79      1000      6212.0      6.2      0.0          dislikes = row["dislikes"]
80      1000      6211.0      6.2      0.0          comments = row["comment_count"]
81      1000      2003.0      2.0      0.0          engagement_rates.append((likes + dislikes + comments) / (views + 1))
82      1000      1449.0      1.4      0.0          ratios.append(likes / (dislikes + 1))
83         1       754.0    754.0      0.0      df["engagement_rate"] = engagement_rates
84         1       678.0    678.0      0.0      df["like_dislike_ratio"] = ratios
85
86         1      1611.0   1611.0      0.0      unique_cats = sorted(df["category_id"].dropna().unique())
87         1         2.0      2.0      0.0      one_hot = []
88      1001      1207.0      1.2      0.0      for i in range(len(df)):
89      1000      1117.0      1.1      0.0          row = []
90     18000     27150.0      1.5      0.0          for cat in unique_cats:
91     17000  15600288.0    917.7      0.8              row.append(1 if df.iloc[i]["category_id"] == cat else 0)
92      1000      1406.0      1.4      0.0          one_hot.append(row)
93
94         1      2863.0   2863.0      0.0      cat_df = pd.DataFrame(one_hot, columns=[f"cat_{int(c)}" for c in unique_cats])
95         1      3698.0   3698.0      0.0      df = pd.concat([df.reset_index(drop=True), cat_df], axis=1)
96         1      1473.0   1473.0      0.0      df = df.drop(columns=["category_id"])
97
98         1         2.0      2.0      0.0      bool_cols = ["comments_disabled", "ratings_disabled", "video_error_or_removed"]
99         4         8.0      2.0      0.0      for col in bool_cols:
100        3      2061.0    687.0      0.0          df[col] = [int(val) for val in df[col]]
101        1      1352.0   1352.0      0.0      df = df.drop(columns=bool_cols)
102
103        1         2.0      2.0      0.0      seen_rows = set()
104        1         1.0      1.0      0.0      deduped_rows = []
```

```
105      1001        1562.0       1.6      0.0      for i in range(len(df)):
106      1000      981842.0     981.8      0.1          row_tuple = tuple(df.iloc[i].values)
107      1000       19336.0      19.3      0.0          if row_tuple not in seen_rows:
108      1000       17629.0      17.6      0.0              seen_rows.add(row_tuple)
109      1000      953327.0     953.3      0.1              deduped_rows.append(df.iloc[i])
110         1      172910.0  172910.0      0.0      df = pd.DataFrame(deduped_rows).reset_index(drop=True)
111
112         1           3.0       3.0      0.0      numeric_attributes = [
113                                                     "views", "publish_hour", "likes", "dislikes", "comment_count",
114                                                     "engagement_rate", "like_dislike_ratio", "tag_count"
115                                                 ]
116         1         235.0     235.0      0.0      numeric_attributes += [col for col in df.columns if "_emb_" in col]
117
118      1161        2563.0       2.2      0.0      for col in numeric_attributes:
119      1160        8043.0       6.9      0.0          transformed = []
120   1161160     1841835.0       1.6      0.1          for i in range(len(df)):
121   1160000  1730356167.0    1491.7     92.7              transformed.append(np.log1p(df.iloc[i][col]))
122      1160      659873.0     568.9      0.0          df[col] = transformed
123
124         1          49.0      49.0      0.0      minmax_scaler = MinMaxScaler()
125         1       76337.0   76337.0      0.0      scaled_minmax = minmax_scaler.fit_transform(df[numeric_attributes])
126      1161        1980.0       1.7      0.0      for j, col in enumerate(numeric_attributes):
127   1161160     1270199.0       1.1      0.1          for i in range(len(df)):
128   1160000    25483031.0      22.0      1.4              df.at[i, col] = np.float32(scaled_minmax[i][j])
129
130         1          10.0      10.0      0.0      standard_scaler = StandardScaler()
131         1       72600.0   72600.0      0.0      scaled_standard = standard_scaler.fit_transform(df[numeric_attributes])
132      1161        1930.0       1.7      0.0      for j, col in enumerate(numeric_attributes):
133   1161160     1268012.0       1.1      0.1          for i in range(len(df)):
134   1160000    25421565.0      21.9      1.4              df.at[i, col] = np.float32(scaled_standard[i][j])
135
136         1       23007.0   23007.0      0.0      df = df.drop(columns=["likes", "dislikes"])
137         1         103.0     103.0      0.0      print("Preprocessing complete. Final shape:", df.shape)
```

Line 121 (transformed.append(np.log1p(df.iloc[i][col]))), which took 1730.36 seconds to run, is the most time-consuming one. This line accounts for 92.7% of the total execution time.

Then, we run the command below to analyze the memory usage per line:

kernprof -l -v ECS_CA4_P3.py

```
Line #    Mem usage    Increment   Line Contents
================================================
     8  651.629 MiB  651.629 MiB   @profile
     9                             def run_pipeline():
    10  651.629 MiB    0.000 MiB       TEXTUAL_COLUMNS = ["title", "tags", "description"]
    11  651.629 MiB    0.000 MiB       EMBEDDING_MODEL = "all-MiniLM-L6-v2"
    12  651.629 MiB    0.000 MiB       EMBEDDING_DIM = 384
    13  651.629 MiB    0.000 MiB       OUTPUT_DIR = "tmp/embeddings/"
    14  651.629 MiB    0.000 MiB       os.makedirs(OUTPUT_DIR, exist_ok=True)
    15
    16  687.438 MiB   35.809 MiB       us_df = pd.read_csv("USvideos.csv")
    17  687.562 MiB    0.125 MiB       us_df["country"] = "US"
    18
    19  730.297 MiB   42.734 MiB       ca_df = pd.read_csv("CAvideos.csv")
    20  730.297 MiB    0.000 MiB       ca_df["country"] = "CA"
    21
    22  731.797 MiB    1.500 MiB       df = pd.concat([us_df, ca_df], ignore_index=True).sample(1000, random_state=42).reset_index(drop=True)
    23
    24  731.797 MiB    0.000 MiB       print(f"[EMBEDDING][INFO]: Loading model {EMBEDDING_MODEL}...")
    25  756.902 MiB   25.105 MiB       model = SentenceTransformer(EMBEDDING_MODEL)
    26
    27  945.816 MiB    0.000 MiB       def clean_tags(text):
    28  945.816 MiB    0.000 MiB           return " ".join(tag.replace('"', '') for tag in str(text).split('|'))
    29
    30 1001.566 MiB    0.000 MiB       for col in TEXTUAL_COLUMNS:
    31  982.613 MiB    0.000 MiB           print(f"[EMBEDDING][INFO]: Embedding column '{col}'...")
    32  982.613 MiB    0.000 MiB           if col == "tags":
    33  945.816 MiB    0.000 MiB               text_data = df[col].fillna("").apply(clean_tags).tolist()
    34                                     else:
    35  982.738 MiB    0.125 MiB               text_data = df[col].fillna("").astype(str).tolist()
    36
    37 1001.566 MiB  244.539 MiB           emb = model.encode(text_data, show_progress_bar=True, batch_size=32)
    38 1001.566 MiB    0.000 MiB           emb_df = pd.DataFrame(emb, columns=[f"{col}_emb_{i}" for i in range(emb.shape[1])])
    39 1001.566 MiB    0.000 MiB           df = pd.concat([df.reset_index(drop=True), emb_df], axis=1)
    40
    41 1001.566 MiB    0.000 MiB       def count_tags_loop(tag_str):
    42 1001.566 MiB    0.000 MiB           if pd.isna(tag_str):
    43                                         return 0
    44 1001.566 MiB    0.000 MiB           count = 0
    45 1001.566 MiB    0.000 MiB           for tag in tag_str.split("|"):
    46 1001.566 MiB    0.000 MiB               if tag.strip() != "":
    47 1001.566 MiB    0.000 MiB                   count += 1
    48 1001.566 MiB    0.000 MiB           return count
    49
    50 1001.566 MiB    0.000 MiB       tag_counts = []
    51 1001.566 MiB    0.000 MiB       for i in range(len(df)):
    52 1001.566 MiB    0.000 MiB           tag_counts.append(count_tags_loop(df.iloc[i]["tags"]))
    53 1001.566 MiB    0.000 MiB       df["tag_count"] = tag_counts
    54
    55 1001.566 MiB    0.000 MiB       publish_dates = []
    56 1001.566 MiB    0.000 MiB       publish_hours = []
    57 1001.566 MiB    0.000 MiB       for i in range(len(df)):
    58 1001.566 MiB    0.000 MiB           try:
    59 1001.566 MiB    0.000 MiB               dt = datetime.strptime(df.iloc[i]["publish_time"], "%Y-%m-%dT%H:%M:%S.%fZ")
    60 1001.566 MiB    0.000 MiB               publish_dates.append(dt)
    61 1001.566 MiB    0.000 MiB               publish_hours.append(dt.hour)
```

```
 62                                          except Exception:
 63                                              publish_dates.append(pd.NaT)
 64                                              publish_hours.append(np.nan)
 65
 66 1003.066 MiB    1.500 MiB       df["publish_time"] = publish_dates
 67 1003.066 MiB    0.000 MiB       df["publish_hour"] = publish_hours
 68
 69 1003.191 MiB    0.000 MiB       for col in TEXTUAL_COLUMNS:
 70 1003.191 MiB    0.000 MiB           if col in df.columns:
 71 1003.191 MiB    0.125 MiB               del df[col]
 72
 73 1003.191 MiB    0.000 MiB       engagement_rates = []
 74 1003.191 MiB    0.000 MiB       ratios = []
 75 1003.316 MiB    0.000 MiB       for i in range(len(df)):
 76 1003.316 MiB    0.125 MiB           row = df.iloc[i]
 77 1003.316 MiB    0.000 MiB           views = row["views"]
 78 1003.316 MiB    0.000 MiB           likes = row["likes"]
 79 1003.316 MiB    0.000 MiB           dislikes = row["dislikes"]
 80 1003.316 MiB    0.000 MiB           comments = row["comment_count"]
 81 1003.316 MiB    0.000 MiB           engagement_rates.append((likes + dislikes + comments) / (views + 1))
 82 1003.316 MiB    0.000 MiB           ratios.append(likes / (dislikes + 1))
 83 1003.316 MiB    0.000 MiB       df["engagement_rate"] = engagement_rates
 84 1003.316 MiB    0.000 MiB       df["like_dislike_ratio"] = ratios
 85
 86 1003.691 MiB    0.375 MiB       unique_cats = sorted(df["category_id"].dropna().unique())
 87 1003.691 MiB    0.000 MiB       one_hot = []
 88 1003.691 MiB    0.000 MiB       for i in range(len(df)):
 89 1003.691 MiB    0.000 MiB           row = []
 90 1003.691 MiB    0.000 MiB           for cat in unique_cats:
 91 1003.691 MiB    0.000 MiB               row.append(1 if df.iloc[i]["category_id"] == cat else 0)
 92 1003.691 MiB    0.000 MiB           one_hot.append(row)
 93
 94 1003.691 MiB    0.000 MiB       cat_df = pd.DataFrame(one_hot, columns=[f"cat_{int(c)}" for c in unique_cats])
 95 1003.691 MiB    0.000 MiB       df = pd.concat([df.reset_index(drop=True), cat_df], axis=1)
 96 1003.816 MiB    0.125 MiB       df = df.drop(columns=["category_id"])
 97
 98 1003.816 MiB    0.000 MiB       bool_cols = ["comments_disabled", "ratings_disabled", "video_error_or_removed"]
 99 1003.816 MiB    0.000 MiB       for col in bool_cols:
100 1003.816 MiB    0.000 MiB           df[col] = [int(val) for val in df[col]]
101 1003.816 MiB    0.000 MiB       df = df.drop(columns=bool_cols)
102
103 1003.816 MiB    0.000 MiB       seen_rows = set()
104 1003.816 MiB    0.000 MiB       deduped_rows = []
105 1076.566 MiB    0.000 MiB       for i in range(len(df)):
106 1076.566 MiB   36.375 MiB           row_tuple = tuple(df.iloc[i].values)
107 1076.566 MiB    0.125 MiB           if row_tuple not in seen_rows:
108 1076.566 MiB    0.000 MiB               seen_rows.add(row_tuple)
109 1076.566 MiB   36.250 MiB               deduped_rows.append(df.iloc[i])
110 1077.316 MiB    0.750 MiB       df = pd.DataFrame(deduped_rows).reset_index(drop=True)
111
112 1077.316 MiB    0.000 MiB       numeric_attributes = [
113                                     "views", "publish_hour", "likes", "dislikes", "comment_count",
114                                     "engagement_rate", "like_dislike_ratio", "tag_count"
115                                 ]
116 1077.316 MiB    0.000 MiB       numeric_attributes += [col for col in df.columns if "_emb_" in col]
```

```
117
118 1078.566 MiB    0.000 MiB       for col in numeric_attributes:
119 1078.566 MiB    0.000 MiB           transformed = []
120 1078.566 MiB    0.000 MiB           for i in range(len(df)):
121 1078.566 MiB    1.125 MiB               transformed.append(np.log1p(df.iloc[i][col]))
122 1078.566 MiB    0.125 MiB           df[col] = transformed
123
124 1078.566 MiB    0.000 MiB       minmax_scaler = MinMaxScaler()
125 1080.191 MiB    1.625 MiB       scaled_minmax = minmax_scaler.fit_transform(df[numeric_attributes])
126 1080.316 MiB    0.000 MiB       for j, col in enumerate(numeric_attributes):
127 1080.316 MiB    0.000 MiB           for i in range(len(df)):
128 1080.316 MiB    0.125 MiB               df.at[i, col] = np.float32(scaled_minmax[i][j])
129
130 1080.316 MiB    0.000 MiB       standard_scaler = StandardScaler()
131 1080.316 MiB    0.000 MiB       scaled_standard = standard_scaler.fit_transform(df[numeric_attributes])
132 1080.316 MiB    0.000 MiB       for j, col in enumerate(numeric_attributes):
133 1080.316 MiB    0.000 MiB           for i in range(len(df)):
134 1080.316 MiB    0.000 MiB               df.at[i, col] = np.float32(scaled_standard[i][j])
135
136 1080.316 MiB    0.000 MiB       df = df.drop(columns=["likes", "dislikes"])
137 1080.316 MiB    0.000 MiB       print("Preprocessing complete. Final shape:", df.shape)
```

Initial memory usage is 651 MiB and Peak memory usage is 1080 MiB.

Then, we run the command below to analyze the real-time CPU usage:

py-spy top -- python3 ECS_CA4_P3.py

```
Collecting samples from 'python3 ECS_CA4_P3.py' (python v3.10.12)
Total Samples 6100
GIL: 1.00%, Active: 99.00%, Threads: 2

  %Own   %Total  OwnTime  TotalTime  Function (filename)
 68.00%  68.00%   38.71s    38.71s   forward (torch/nn/modules/linear.py)
 22.00%  97.00%    6.40s    49.69s   forward (transformers/models/bert/modeling_bert.py)
  3.00%   3.00%    3.00s     3.00s   forward (transformers/activations.py)
  0.00%   0.00%   0.860s    0.950s   read (pandas/io/parsers/c_parser_wrapper.py)
  0.00%   0.00%   0.850s     4.54s   _call_with_frames_removed (<frozen importlib._bootstrap>)
  0.00%   0.00%   0.650s    0.650s   get_data (<frozen importlib._bootstrap_external>)
  4.00%   4.00%   0.630s    0.630s   layer_norm (torch/nn/functional.py)
  0.00%   0.00%   0.550s    0.550s   _compile_bytecode (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.490s    0.820s   create_import_structure_from_path (transformers/utils/import_utils.py)
  0.00%  46.00%   0.380s    30.24s   apply_chunking_to_forward (transformers/pytorch_utils.py)
  0.00%   0.00%   0.270s    0.270s   fetch__all__ (transformers/utils/import_utils.py)
  0.00%   0.00%   0.260s    0.260s   embedding (torch/nn/functional.py)
  0.00%   0.00%   0.230s    0.230s   read (ssl.py)
  0.00%   0.00%   0.170s    0.180s   _path_hooks (<frozen importlib._bootstrap_external>)
  1.00%   1.00%   0.160s    0.280s   _batch_encode_plus (transformers/tokenization_utils_fast.py)
  0.00%   0.00%   0.130s    0.130s   _create_fn (dataclasses.py)
  1.00%   1.00%   0.120s    0.120s   forward (sentence_transformers/models/Pooling.py)
  0.00%   0.00%   0.110s    0.110s   _path_stat (<frozen importlib._bootstrap_external>)
  0.00%  98.00%   0.090s    49.81s   _call_impl (torch/nn/modules/module.py)
  0.00%   0.00%   0.080s    0.080s   __setattr__ (enum.py)
  0.00%   0.00%   0.070s    0.070s   impl (torch/library.py)
  0.00%   0.00%   0.070s    0.070s   as_tensor (transformers/tokenization_utils_base.py)
  0.00%   0.00%   0.060s    0.070s   _joinrealpath (posixpath.py)
  0.00%   0.00%   0.060s    0.060s   decode (codecs.py)
  0.00%   0.00%   0.060s    0.060s   __init__ (ctypes/__init__.py)
  0.00%   0.00%   0.050s    0.060s   dedent (textwrap.py)
  0.00%   0.00%   0.050s    0.050s   <listcomp> (sentence_transformers/SentenceTransformer.py)
  0.00%   0.00%   0.050s    0.060s   __init__ (inspect.py)
  0.00%   0.00%   0.050s    0.060s   ssl_wrap_socket (urllib3/util/ssl_.py)
  0.00%   0.00%   0.050s    0.060s   _has_script_object_arg (torch/_ops.py)
  0.00%   0.00%   0.040s    0.040s   _convert_encoding (transformers/tokenization_utils_fast.py)
  0.00%   0.00%   0.040s    0.040s   transpose_for_scores (transformers/models/bert/modeling_bert.py)
  0.00%   0.00%   0.040s    0.050s   _merge_blocks (pandas/core/internals/managers.py)
  0.00%   0.00%   0.030s    0.030s   join (posixpath.py)
  0.00%   0.00%   0.030s    0.110s   _signature_from_callable (inspect.py)
  0.00%   0.00%   0.030s    0.030s   __getattr__ (torch/nn/modules/module.py)
```

In the example screenshot, total samples are 6100. GIL (Global Interpreter Lock) is 1.00% that means 99% of time is spent in native code not blocked by Python itself. Active is 99.00% that means the program is CPU-active almost all the time. Threads is 2 that means some operations use background threads.

Part 2: Optimization

1. *.iloc[i]* is extremely slow and memory-intensive.

```
tag_counts = []
for i in range(len(df)):
    tag_counts.append(count_tags_loop(df.iloc[i]["tags"]))
df["tag_count"] = tag_counts
```

⬇️

```
df["tag_count"] = df["tags"].fillna("").apply(lambda x: len([t for t in
str(x).split("|") if t.strip()]))
```

2. *.iloc[i]* is extremely slow and memory-intensive.

```
for i in range(len(df)):
    row = df.iloc[i]
    views = row["views"]
    likes = row["likes"]
    dislikes = row["dislikes"]
    comments = row["comment_count"]
    engagement_rates.append((likes + dislikes + comments) / (views + 1))
    ratios.append(likes / (dislikes + 1))
```

⬇️

```
df["engagement_rate"] = (df["likes"] + df["dislikes"] + df["comment_count"]) /
(df["views"] + 1)
df["like_dislike_ratio"] = df["likes"] / (df["dislikes"] + 1)
```

3. The code manually loops over rows and categories, creating one-hot encodings with conditionals.

```
unique_cats = sorted(df["category_id"].dropna().unique())
one_hot = []
for i in range(len(df)):
    row = []
    for cat in unique_cats:
        row.append(1 if df.iloc[i]["category_id"] == cat else 0)
    one_hot.append(row)
cat_df = pd.DataFrame(one_hot, columns=[f"cat_{int(c)}" for c in unique_cats])
df = pd.concat([df.reset_index(drop=True), cat_df], axis=1)
```

⬇️

```
cat_df = pd.get_dummies(df["category_id"], prefix="cat")
df = pd.concat([df, cat_df], axis=1)
```

cProfile:

```
[EMBEDDING][INFO]: Loading model all-MiniLM-L6-v2...
[EMBEDDING][INFO]: Embedding column 'title'...
[EMBEDDING][INFO]: Embedding column 'tags'...
[EMBEDDING][INFO]: Embedding column 'description'...
Preprocessing complete. Final shape: (1000, 1181)
         11148287 function calls (10944247 primitive calls) in 68.517 seconds


   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3552   43.161    0.012   43.161    0.012 {built-in method torch._C._nn.linear}
      576    6.508    0.011    6.508    0.011 {built-in method torch._C._nn.scaled_dot_product_attention}
      576    3.459    0.006    3.459    0.006 {built-in method torch._C._nn.gelu}
       21    3.438    0.164    3.438    0.164 {method 'read' of '_ssl._SSLSocket' objects}
     1248    0.783    0.001    0.783    0.001 {built-in method torch.layer_norm}
        2    0.766    0.383    0.836    0.418 c_parser_wrapper.py:222(read)
       96    0.570    0.006    0.570    0.006 {method 'encode_batch' of 'tokenizers.Tokenizer' objects}
      576    0.563    0.001   34.167    0.059 pytorch_utils.py:175(apply_chunking_to_forward)
      576    0.409    0.001   18.715    0.032 modeling_bert.py:506(forward)
  192/191    0.379    0.002    0.381    0.002 {built-in method _imp.create_dynamic}
      576    0.343    0.001   14.799    0.026 modeling_bert.py:519(forward)
      576    0.334    0.001    4.377    0.008 modeling_bert.py:433(forward)
     3696    0.284    0.000    0.590    0.000 import_utils.py:2309(fetch__all__)
     4097    0.262    0.000    0.262    0.000 {built-in method marshal.loads}
  2940820    0.215    0.000    0.215    0.000 {method 'startswith' of 'str' objects}
      288    0.187    0.001    0.187    0.001 {built-in method torch.embedding}
   7103/1    0.182    0.000   68.529   68.529 {built-in method builtins.exec}
     5884    0.174    0.000    0.181    0.000 generic.py:6255(__finalize__)
     3937    0.148    0.000    0.170    0.000 {method 'read' of '_io.TextIOWrapper' objects}
 1064/319    0.142    0.000    1.034    0.003 import_utils.py:2351(create_import_structure_from_path)
11136/288    0.133    0.000   56.686    0.197 module.py:1755(_call_impl)
    13920    0.131    0.000    0.131    0.000 {method 'splitlines' of 'str' objects}
     2973    0.118    0.000    0.189    0.000 SentenceTransformer.py:1999(<listcomp>)
    36447    0.116    0.000    0.117    0.000 {built-in method posix.stat}
1674969/1641076    0.108    0.000    0.116    0.000 {built-in method builtins.len}
    15253    0.099    0.000    0.103    0.000 <frozen importlib._bootstrap>:100(acquire)
        1    0.099    0.099    0.099    0.099 {built-in method _socket.getaddrinfo}
        1    0.099    0.099    0.099    0.099 {method 'connect' of '_socket.socket' objects}
    15253    0.097    0.000    0.112    0.000 <frozen importlib._bootstrap>:179(_get_module_lock)
       96    0.092    0.001    0.347    0.004 modeling_bert.py:149(forward)
     4114    0.090    0.000    0.090    0.000 {method 'read' of '_io.BufferedReader' objects}
      371    0.088    0.000    0.088    0.000 {built-in method torch.tensor}
        1    0.088    0.088    0.088    0.088 {method 'do_handshake' of '_ssl._SSLSocket' objects}
7854/7742    0.086    0.000    0.380    0.000 {built-in method builtins.__build_class__}
834758/824491    0.085    0.000    0.112    0.000 {built-in method builtins.isinstance}
     5037    0.082    0.000    0.082    0.000 {built-in method _codecs.utf_8_decode}
     1382    0.068    0.000    0.068    0.000 {method 'to' of 'torch._C.TensorBase' objects}
     6633    0.065    0.000    0.077    0.000 functools.py:35(update_wrapper)
     4099    0.064    0.000    0.064    0.000 {built-in method io.open_code}
       96    0.053    0.001    0.137    0.001 Pooling.py:135(forward)
 7293/666    0.052    0.000    4.973    0.007 <frozen importlib._bootstrap>:1022(_find_and_load)
        3    0.050    0.017   57.747   19.249 SentenceTransformer.py:826(encode)
217017/216897    0.050    0.000    0.153    0.000 {built-in method builtins.getattr}
        1    0.048    0.048    0.048    0.048 {method 'load_verify_locations' of '_ssl._SSLContext' objects}
       96    0.046    0.000   56.050    0.584 modeling_bert.py:619(forward)
```

Before optimization, *managers.py:958(fast_xs)* and *blocks.py:1319(iget)* dominated CPU time due to inefficient DataFrame operations (e.g., iloc, .at[], manual loops).
Log transform, MinMaxScaler, and StandardScaler were applied with for loops and *.at[i, j]* style updates which were extremely slow and memory-inefficient.

After optimization, these high-cost pandas internal calls are gone or negligible because of using vectorized pandas operations, avoiding slow row-by-row *.iloc* loops.
Transformed with apply + vectorized operations, and scalers applied directly to the entire DataFrame slice.

line-profiler:

```
Preprocessing complete. Final shape: (1000, 1181)
Wrote profile results to ECS_CA4_P3_optimized.py.lprof
Timer unit: 1e-06 s

Total time: 68.027 s
File: ECS_CA4_P3_optimized.py
Function: run_pipeline at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           @profile
     9                                           def run_pipeline():
    10         1          3.0      3.0      0.0      TEXTUAL_COLUMNS = ["title", "tags", "description"]
    11         1          1.0      1.0      0.0      EMBEDDING_MODEL = "all-MiniLM-L6-v2"
    12         1          1.0      1.0      0.0      EMBEDDING_DIM = 384
    13         1          1.0      1.0      0.0      OUTPUT_DIR = "tmp/embeddings/"
    14         1         28.0     28.0      0.0      os.makedirs(OUTPUT_DIR, exist_ok=True)
    15
    16         1     438056.0 438056.0      0.6      us_df = pd.read_csv("USvideos.csv")
    17         1       1223.0   1223.0      0.0      us_df["country"] = "US"
    18         1     576184.0 576184.0      0.8      ca_df = pd.read_csv("CAvideos.csv")
    19         1        545.0    545.0      0.0      ca_df["country"] = "CA"
    20
    21         1       7002.0   7002.0      0.0      df = pd.concat([us_df, ca_df], ignore_index=True).sample(1000, random_state=42).reset_index(drop=True)
    22
    23         1        123.0    123.0      0.0      print(f"[EMBEDDING][INFO]: Loading model {EMBEDDING_MODEL}...")
    24         1    4505658.0 4505658.0      6.6      model = SentenceTransformer(EMBEDDING_MODEL)
    25
    26         1          3.0      3.0      0.0      def clean_tags(text):
    27                                                   return " ".join(tag.replace('"', '') for tag in str(text).split('|'))
    28
    29         4          8.0      2.0      0.0      for col in TEXTUAL_COLUMNS:
    30         3        219.0     73.0      0.0          print(f"[EMBEDDING][INFO]: Embedding column '{col}'...")
    31         3          4.0      1.3      0.0          if col == "tags":
    32         1       8997.0   8997.0      0.0              text_data = df[col].fillna("").apply(clean_tags).tolist()
    33                                                   else:
    34         2       3465.0   1732.5      0.0              text_data = df[col].fillna("").astype(str).tolist()
    35
    36         3   61489954.0 20496651.3     90.4          emb = model.encode(text_data, show_progress_bar=True, batch_size=32)
    37         3       1945.0    648.3      0.0          emb_df = pd.DataFrame(emb, columns=[f"{col}_emb_{i}" for i in range(emb.shape[1])])
    38         3       5943.0   1981.0      0.0          df = pd.concat([df.reset_index(drop=True), emb_df], axis=1)
    39
    40         1       5425.0   5425.0      0.0      df["tag_count"] = df["tags"].fillna("").apply(lambda x: len([t for t in str(x).split("|") if t.strip()]))
    41
    42         1      19194.0  19194.0      0.0      df["publish_time"] = pd.to_datetime(df["publish_time"], errors="coerce", utc=True)
    43         1       1431.0   1431.0      0.0      df["publish_hour"] = df["publish_time"].dt.hour
    44
    45         1       4958.0   4958.0      0.0      df.drop(columns=[col for col in TEXTUAL_COLUMNS if col in df.columns], inplace=True)
    46
    47         1       1894.0   1894.0      0.0      df["engagement_rate"] = (df["likes"] + df["dislikes"] + df["comment_count"]) / (df["views"] + 1)
    48         1        581.0    581.0      0.0      df["like_dislike_ratio"] = df["likes"] / (df["dislikes"] + 1)
    49
    50         1       1860.0   1860.0      0.0      cat_df = pd.get_dummies(df["category_id"], prefix="cat")
    51         1       2130.0   2130.0      0.0      df = pd.concat([df, cat_df], axis=1)
    52         1       6787.0   6787.0      0.0      df.drop(columns=["category_id"], inplace=True)
```

```
    53
    54         1          2.0      2.0      0.0      bool_cols = ["comments_disabled", "ratings_disabled", "video_error_or_removed"]
    55         1       1329.0   1329.0      0.0      df[bool_cols] = df[bool_cols].astype(int)
    56         1       1004.0   1004.0      0.0      df.drop(columns=bool_cols, inplace=True)
    57
    58         1     140663.0 140663.0      0.2      df = df.drop_duplicates().reset_index(drop=True)
    59
    60         2          7.0      3.5      0.0      numeric_attributes = [
    61                                                   "views", "publish_hour", "likes", "dislikes", "comment_count",
    62                                                   "engagement_rate", "like_dislike_ratio", "tag_count"
    63         1        202.0    202.0      0.0      ] + [col for col in df.columns if "_emb_" in col]
    64
    65         1     486082.0 486082.0      0.7      df[numeric_attributes] = df[numeric_attributes].apply(lambda col: np.log1p(col))
    66
    67         1          8.0      8.0      0.0      minmax_scaler = MinMaxScaler()
    68         1     138957.0 138957.0      0.2      df[numeric_attributes] = minmax_scaler.fit_transform(df[numeric_attributes])
    69
    70         1         10.0     10.0      0.0      standard_scaler = StandardScaler()
    71         1     147623.0 147623.0      0.2      df[numeric_attributes] = standard_scaler.fit_transform(df[numeric_attributes])
    72
    73         1      27450.0  27450.0      0.0      df.drop(columns=["likes", "dislikes"], inplace=True)
    74
    75         1         66.0     66.0      0.0      print("Preprocessing complete. Final shape:", df.shape)
```

For example, from nested loops to *apply(np.log1p)*has a 30x speedup.

memory-profiler:

```
Preprocessing complete. Final shape: (1000, 1181)
Filename: ECS_CA4_P3_optimized.py

Line #    Mem usage    Increment   Line Contents
================================================
     8  649.922 MiB  649.922 MiB   @profile
     9                             def run_pipeline():
    10  649.922 MiB    0.000 MiB       TEXTUAL_COLUMNS = ["title", "tags", "description"]
    11  649.922 MiB    0.000 MiB       EMBEDDING_MODEL = "all-MiniLM-L6-v2"
    12  649.922 MiB    0.000 MiB       EMBEDDING_DIM = 384
    13  649.922 MiB    0.000 MiB       OUTPUT_DIR = "tmp/embeddings/"
    14  649.922 MiB    0.000 MiB       os.makedirs(OUTPUT_DIR, exist_ok=True)
    15
    16  685.488 MiB   35.566 MiB       us_df = pd.read_csv("USvideos.csv")
    17  685.738 MiB    0.250 MiB       us_df["country"] = "US"
    18  753.105 MiB   67.367 MiB       ca_df = pd.read_csv("CAvideos.csv")
    19  753.105 MiB    0.000 MiB       ca_df["country"] = "CA"
    20
    21  753.730 MiB    0.625 MiB       df = pd.concat([us_df, ca_df], ignore_index=True).sample(1000, random_state=42).reset_index(drop=True)
    22
    23  753.730 MiB    0.000 MiB       print(f"[EMBEDDING][INFO]: Loading model {EMBEDDING_MODEL}...")
    24  770.184 MiB   16.453 MiB       model = SentenceTransformer(EMBEDDING_MODEL)
    25
    26  914.297 MiB    0.000 MiB       def clean_tags(text):
    27  914.297 MiB    0.000 MiB           return " ".join(tag.replace('"', '') for tag in str(text).split('|'))
    28
    29  980.711 MiB  -24.027 MiB       for col in TEXTUAL_COLUMNS:
    30  980.711 MiB    0.000 MiB           print(f"[EMBEDDING][INFO]: Embedding column '{col}'...")
    31  980.711 MiB    0.000 MiB           if col == "tags":
    32  914.297 MiB    0.000 MiB               text_data = df[col].fillna("").apply(clean_tags).tolist()
    33                                     else:
    34  980.836 MiB    0.125 MiB               text_data = df[col].fillna("").astype(str).tolist()
    35
    36  980.711 MiB  186.250 MiB           emb = model.encode(text_data, show_progress_bar=True, batch_size=32)
    37  980.711 MiB -9298.582 MiB           emb_df = pd.DataFrame(emb, columns=[f"{col}_emb_{i}" for i in range(emb.shape[1])])
    38  980.711 MiB  -23.902 MiB           df = pd.concat([df.reset_index(drop=True), emb_df], axis=1)
    39
    40  956.684 MiB  -24.027 MiB       df["tag_count"] = df["tags"].fillna("").apply(lambda x: len([t for t in str(x).split("|") if t.strip()]))
    41
    42  958.934 MiB    2.250 MiB       df["publish_time"] = pd.to_datetime(df["publish_time"], errors="coerce", utc=True)
    43  959.059 MiB    0.125 MiB       df["publish_hour"] = df["publish_time"].dt.hour
    44
    45  959.434 MiB    0.375 MiB       df.drop(columns=[col for col in TEXTUAL_COLUMNS if col in df.columns], inplace=True)
    46
    47  959.559 MiB    0.125 MiB       df["engagement_rate"] = (df["likes"] + df["dislikes"] + df["comment_count"]) / (df["views"] + 1)
    48  959.559 MiB    0.000 MiB       df["like_dislike_ratio"] = df["likes"] / (df["dislikes"] + 1)
    49
    50  959.809 MiB    0.250 MiB       cat_df = pd.get_dummies(df["category_id"], prefix="cat")
    51  959.809 MiB    0.000 MiB       df = pd.concat([df, cat_df], axis=1)
    52  959.809 MiB    0.000 MiB       df.drop(columns=["category_id"], inplace=True)
    53
    54  959.809 MiB    0.000 MiB       bool_cols = ["comments_disabled", "ratings_disabled", "video_error_or_removed"]
    55  959.809 MiB    0.000 MiB       df[bool_cols] = df[bool_cols].astype(int)
    56  959.809 MiB    0.000 MiB       df.drop(columns=bool_cols, inplace=True)
    57
    58  961.059 MiB    1.250 MiB       df = df.drop_duplicates().reset_index(drop=True)
    59
    60  961.059 MiB    0.000 MiB       numeric_attributes = [
    61                                     "views", "publish_hour", "likes", "dislikes", "comment_count",
    62                                     "engagement_rate", "like_dislike_ratio", "tag_count"
    63  961.059 MiB    0.000 MiB       ] + [col for col in df.columns if "_emb_" in col]
    64
    65  961.559 MiB    0.500 MiB       df[numeric_attributes] = df[numeric_attributes].apply(lambda col: np.log1p(col))
    66
    67  961.559 MiB    0.000 MiB       minmax_scaler = MinMaxScaler()
    68  962.059 MiB    0.500 MiB       df[numeric_attributes] = minmax_scaler.fit_transform(df[numeric_attributes])
    69
    70  962.059 MiB    0.000 MiB       standard_scaler = StandardScaler()
    71  962.184 MiB    0.125 MiB       df[numeric_attributes] = standard_scaler.fit_transform(df[numeric_attributes])
    72
    73  962.184 MiB    0.000 MiB       df.drop(columns=["likes", "dislikes"], inplace=True)
    74
    75  962.184 MiB    0.000 MiB       print("Preprocessing complete. Final shape:", df.shape)
```

Before optimization, the use of *.iloc[i]* in multiple for-loops duplicated data temporarily, increasing memory overhead.

After optimization, memory-intensive manual deduplication and transformation loops were replaced with pandas-native drop_duplicates() and apply(lambda col: ...), reducing memory peaks.

Py-spy:

```
Total Samples 6100
GIL: 1.00%, Active: 100.00%, Threads: 2

  %Own   %Total  OwnTime  TotalTime  Function (filename)
 69.00%  69.00%   39.21s    39.21s   forward (torch/nn/modules/linear.py)
 23.00% 100.00%    7.09s    51.14s   forward (transformers/models/bert/modeling_bert.py)
  5.00%   5.00%    2.81s     2.81s   forward (transformers/activations.py)
  0.00%   0.00%   0.830s    0.830s   layer_norm (torch/nn/functional.py)
  0.00%   0.00%   0.820s    0.850s   read (pandas/io/parsers/c_parser_wrapper.py)
  0.00%   0.00%   0.750s    0.750s   get_data (<frozen importlib._bootstrap_external>)
  2.00%  57.00%   0.570s    30.92s   apply_chunking_to_forward (transformers/pytorch_utils.py)
  0.00%   0.00%   0.420s    0.420s   _compile_bytecode (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.400s     3.51s   _call_with_frames_removed (<frozen importlib._bootstrap>)
  0.00%   0.00%   0.350s    0.600s   create_import_structure_from_path (transformers/utils/import_utils.py)
  0.00%   0.00%   0.220s    0.220s   fetch__all__ (transformers/utils/import_utils.py)
  0.00%   0.00%   0.200s    0.320s   _batch_encode_plus (transformers/tokenization_utils_fast.py)
  0.00%   0.00%   0.160s    0.160s   embedding (torch/nn/functional.py)
  0.00%   0.00%   0.130s    0.130s   _create_fn (dataclasses.py)
  0.00%   0.00%   0.120s    0.120s   _expand_mask (transformers/modeling_attn_mask_utils.py)
  0.00%   0.00%   0.110s    0.110s   read (ssl.py)
  1.00% 100.00%   0.100s    51.23s   _call_impl (torch/nn/modules/module.py)
  0.00%   0.00%   0.100s    0.100s   _path_hooks (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.090s    0.090s   __setattr__ (enum.py)
  0.00%   0.00%   0.080s    0.080s   inner (tqdm/utils.py)
  0.00%   0.00%   0.080s    0.080s   as_tensor (transformers/tokenization_utils_base.py)
  0.00%   0.00%   0.080s    0.080s   transpose_for_scores (transformers/models/bert/modeling_bert.py)
  0.00%   0.00%   0.080s    0.080s   _path_stat (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.070s    0.070s   forward (sentence_transformers/models/Pooling.py)
  0.00%   0.00%   0.060s    0.070s   _has_script_object_arg (torch/_ops.py)
  0.00%   0.00%   0.060s    0.060s   _joinrealpath (posixpath.py)
  0.00%   0.00%   0.050s    0.050s   exists (genericpath.py)
  0.00%   0.00%   0.050s    0.050s   forward (torch/nn/modules/activation.py)
  0.00%   0.00%   0.040s    0.040s   decode (codecs.py)
  0.00%   0.00%   0.040s    0.090s   __init__ (torch/_ops.py)
  0.00%   0.00%   0.040s    0.040s   ssl_wrap_socket (urllib3/util/ssl_.py)
  0.00%   0.00%   0.030s    0.030s   _fill_cache (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.030s    0.030s   docformat (scipy/_lib/doccer.py)
  0.00%   0.00%   0.030s    0.030s   <listcomp> (<frozen importlib._bootstrap_external>)
  0.00%   0.00%   0.030s    0.030s   dedent (textwrap.py)
  0.00%   0.00%   0.030s    0.030s   impl (torch/library.py)
  0.00%   0.00%   0.030s    0.030s   raw_decode (json/decoder.py)
  0.00%   0.00%   0.030s    0.030s   _prepare_class_assumptions (sympy/core/assumptions.py)
  0.00%   0.00%   0.030s    0.040s   __init__ (inspect.py)
  0.00%   0.00%   0.030s    0.030s   open (pathlib.py)
  0.00%   0.00%   0.030s    0.030s   isdir (genericpath.py)
  0.00%   0.00%   0.030s    0.030s   _get_packet (torch/_ops.py)
  0.00%   0.00%   0.030s    0.200s   _process_class (dataclasses.py)
  0.00%   0.00%   0.020s     3.87s   _find_and_load_unlocked (<frozen importlib._bootstrap>)
  0.00%   0.00%   0.020s    0.040s   <listcomp> (sentence_transformers/SentenceTransformer.py)
  0.00%   0.00%   0.020s    0.030s   set_truncation_and_padding (transformers/tokenization_utils_fast.py)
  0.00%   0.00%   0.020s    0.020s   vstack (numpy/_core/shape_base.py)
  0.00%   0.00%   0.020s    0.030s   __init__ (transformers/utils/import_utils.py)
  0.00%   0.00%   0.020s    0.020s   dropout (torch/nn/functional.py)
  0.00%   0.00%   0.020s    0.020s   __init__ (pandas/io/parsers/c_parser_wrapper.py)

Press Control-C to quit, or ? for help.
```

sentence encoding is expensive and stays the core bottleneck.

Part 3: compare three different methods for applying a condition to a column in a DataFrame

suppose we have a column called "views" and we want to create a new column "popularity" that contains "popular" if *views > 100,000* and "not popular" otherwise.

1. **Using pandas apply:**

df["popularity"] = df["views"].apply(lambda x: "popular" if x > 100000 else "not popular")

This is the slowest method because it loops over rows in Python. However, it is the most readable one.

2. **Using pandas map**

df["popularity"] = (df["views"] > 100000).map({True: "popular", False: "not popular"})

This is faster than apply but less readable.

3. **Using numpy where**

df["popularity"] = np.where(df["views"] > 100000, "popular", "not popular")

This is the fastest and most memory-efficient due to vectorized operations but the least readable.

Execution time, memory usage, CPU usage, and readability are the key metrics to compare.

Overall, if performance matters (especially on large datasets), prefer np.where. If clarity is more important for the reader or for debugging, apply may be a better fit.

Part 4: Questions

۱. پروفایلینگ در محیط Production باید با دقت بیشتری انجام شود تا کمترین تأثیر را روی عملکرد سیستم داشته باشد و اطلاعات حساس در معرض خطر قرار نگیرند. معمولاً از ابزارهای سبک مثل py-spy یا cProfile در حالت نمونه‌برداری استفاده می‌شود چون می‌توانند بدون توقف برنامه، آن را بررسی کنند. در این محیط، باید به تاثیر عملکردی، امنیت و دقت داده‌ها در شرایط واقعی توجه ویژه داشت.

۲. در برنامه‌های multithreaded، پروفایلینگ پیچیده‌تر است چون زمان اجرا و مصرف CPU بین چند نخ پخش می‌شود. ابزارهایی مثل py-spy یا perf برای این شرایط مناسب‌تر هستند چون می‌توانند عملکرد هر نخ را جداگانه بررسی کنند. برخلاف برنامه‌های تک‌ریسمانی، در اینجا باید به مسائل مربوط به رقابت بین نخ‌ها، جابه‌جایی بین نخ‌ها و محدودیت‌های GIL در پایتون توجه کرد تا تحلیل دقیقی به دست آورد.