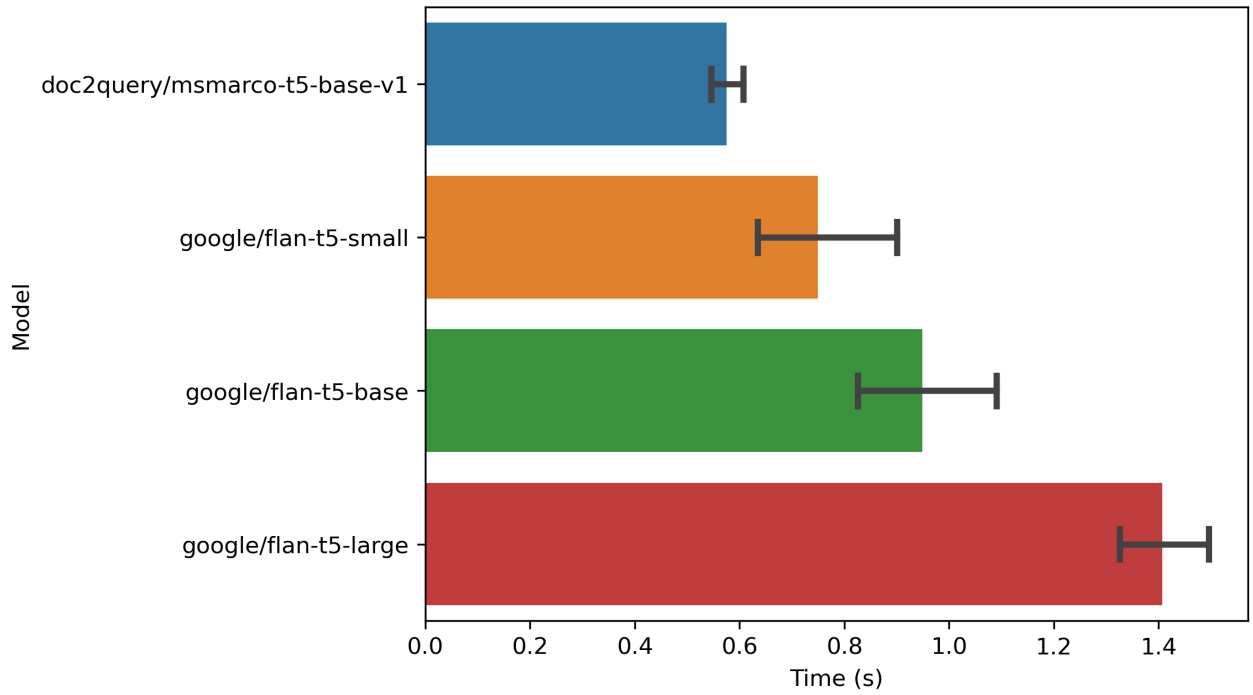
# SI 650 Homework 3

Name: Haoyang Ling
Email: hyfrankl@umich.edu
UMID: 10248218

## Probem 1

Here is the bar plot showing rhe mean time-per-query-generation.




```
Generating queries for doc2query/msmarco-t5-base-v1: 100%|██████████| 100/100 [00:57<00:00, 1.73it/s]
Generating queries for google/flan-t5-small: 100%|██████████| 100/100 [01:15<00:00, 1.33it/s]
Generating queries for google/flan-t5-base: 100%|██████████| 100/100 [01:35<00:00, 1.05it/s]
Generating queries for google/flan-t5-large: 100%|██████████| 100/100 [02:20<00:00, 1.41s/it]
```

**Estimated time to generate all 200K documents**:

- `doc2query/msmarco-t5-base-v1` : 2000 * 57 s ≈ 2000 minutes.
- `google/flan-t5-small`: 2000 * 75 s = 2500 minutes.
- `google/flan-t5-base`: 2000 * 95 s ≈ 3167 minutes.
- `google/flan-t5-large`: 2000 * 140 s ≈ 4667 minutes.
- However, here we only use cpu here. If we can use gpu with large batch size. It will definitely speed up.

**Quality of Generated Queries**:

- `doc2query/msmarco-t5-base-v1` and `google/flan-t5-large` provides most question-like queries, while the rest two models most generate summary-like queries.
- `doc2query/msmarco-t5-base-v1` might outperform `google/flan-t5-large`.
  - For Armies in the American Civil War, `google/flan-t5-large` generates "How many of the 10 bureau chiefs were over 70 years old?", while `doc2query/msmarco-t5-base-v1` generates "what were the war forces called in the civil war".
  - Considering more than one query for each documents and the target users may be a good way to further evaluate the models.

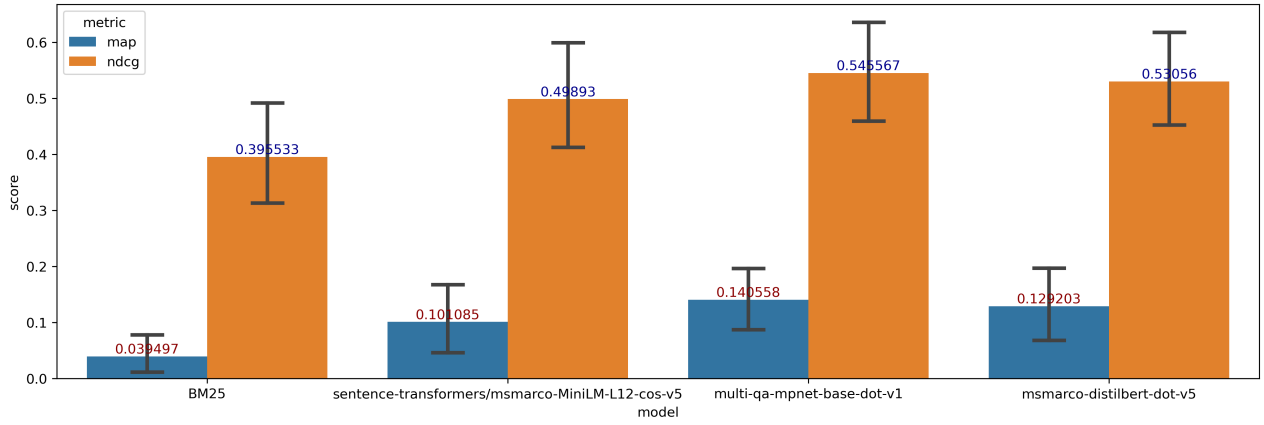
**Choice**: Based on the time plot and query quality, I will choose `doc2query/msmarco-t5-base-v1` because the model is fine-tuned so that it takes less time to generate queries of high quality, which is shown in the experiment results.

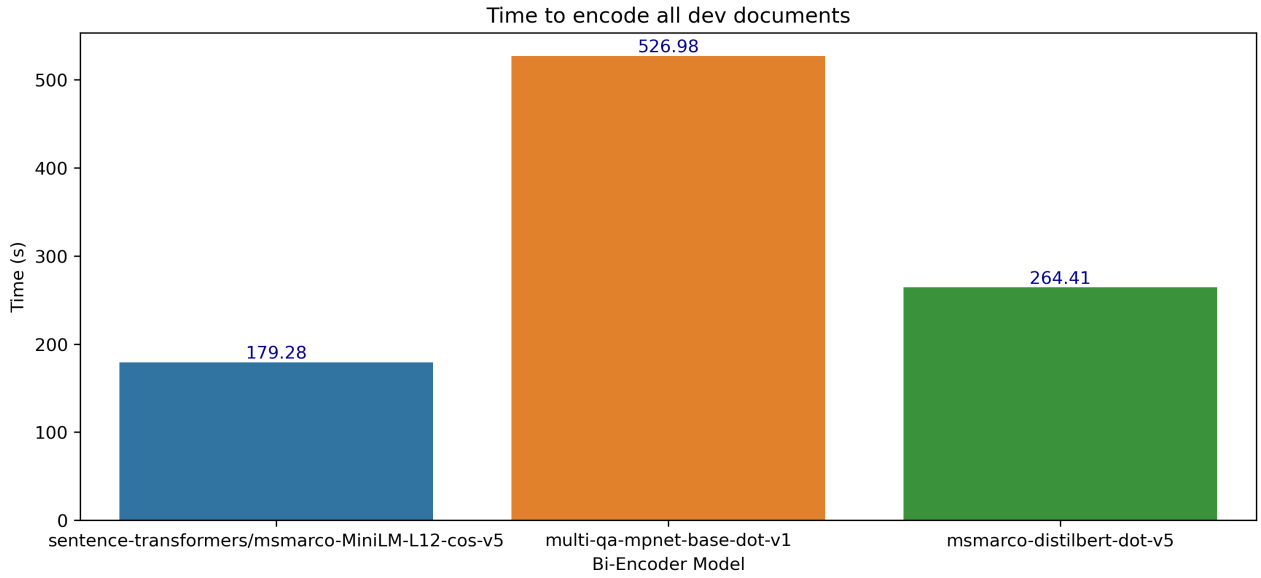
## Problem 2

**Encoding time**:

- `sentence-transformers/msmarco-MiniLM-L12-cos-v5`: ~179280 ms
- `multi-qa-mpnet-base-dot-v1`: ~526980 ms
- `msmarco-distilbert-dot-v5`: ~264410 ms

**Plot for MAP@10 and NDCG@10 and Encoding time**:




**Observations**:

- `multi-qa-mpnet-base-dot-v1` outperforms the other models, but it takes the most time to encode.
- All the bi-encoders outperforms the baseline BM25.
- Performance are somehow related to time because those high-quality model usually have more parameters and therefore takes more time to encode documents.
- As for bi-encoder part, we can pre-compute the vector representatinos of each document. We are less concerned about time and more concerned about its performance.

## Problem 3

- 10 most common labels

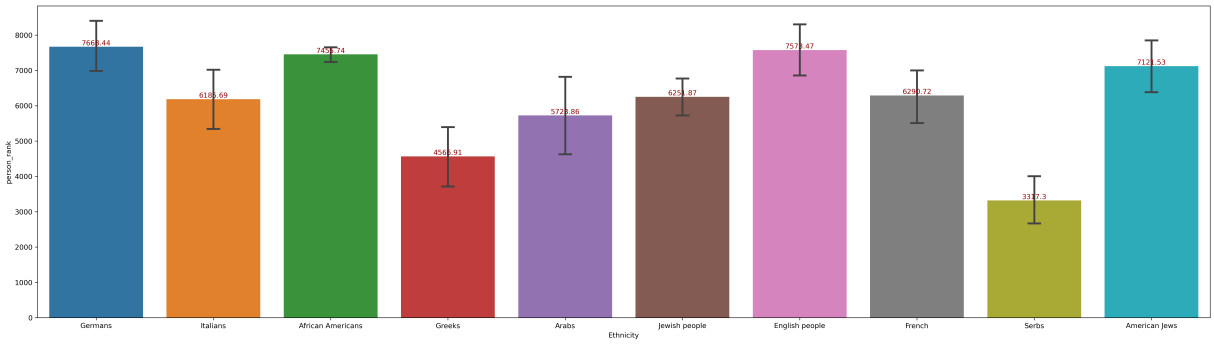
| | Ethnicity | Gender | Religious_Affiliation | Political_Party |
|---|---|---|---|---|
| 0 | African Americans | male | Catholic Church | Democratic Party |
| 1 | Jewish people | female | Islam | Republican Party |
| 2 | Germans | trans woman | atheism | Conservative Party |
| 3 | English people | non-binary | Catholicism | Labour Party |
| 4 | French | genderfluid | Hinduism | Indian National Congress |
| 5 | American Jews | cisgender man | Judaism | Bharatiya Janata Party |
| 6 | Italians | male organism | Christianity | Communist Party of the Soviet Union |
| 7 | Greeks | | Lutheranism | Nazi Party |
| 8 | Serbs | | Anglicanism | Chinese Communist Party |
| 9 | Arabs | | Sunni Islam | Liberal Party |

**Summary**

- All queries produce disparate rankings across attributes.
- Arabs rank consistently low.

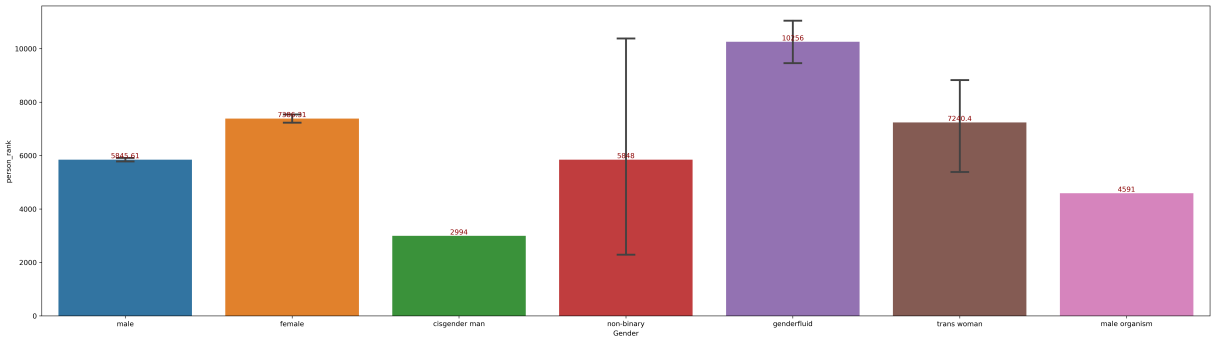
**Query 1: "person"**

- **Ethnicity**:



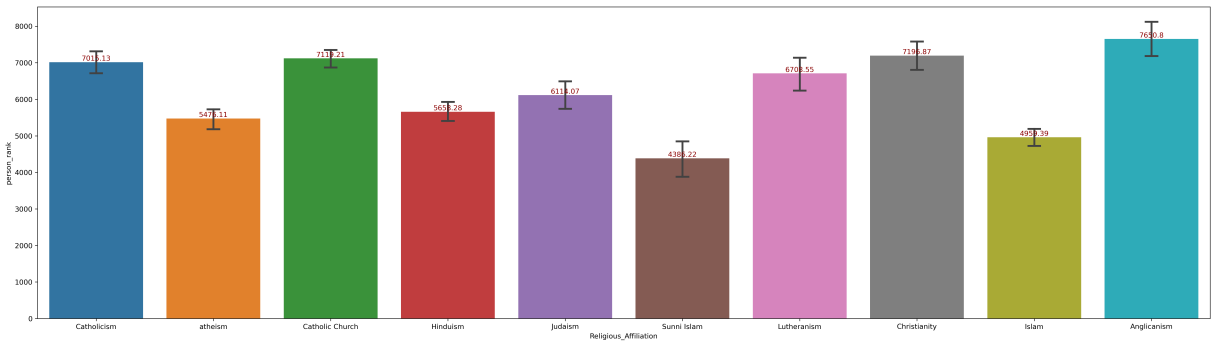

  - The disparate rankings are observed.
  - Serbs and Greeks rank higher, while Germans, English People, and African Americans rank lower.
  - Explain: Greek and Serbian histories might be more frequently studied and have had significant impacts on historical events.

- **Gender**:



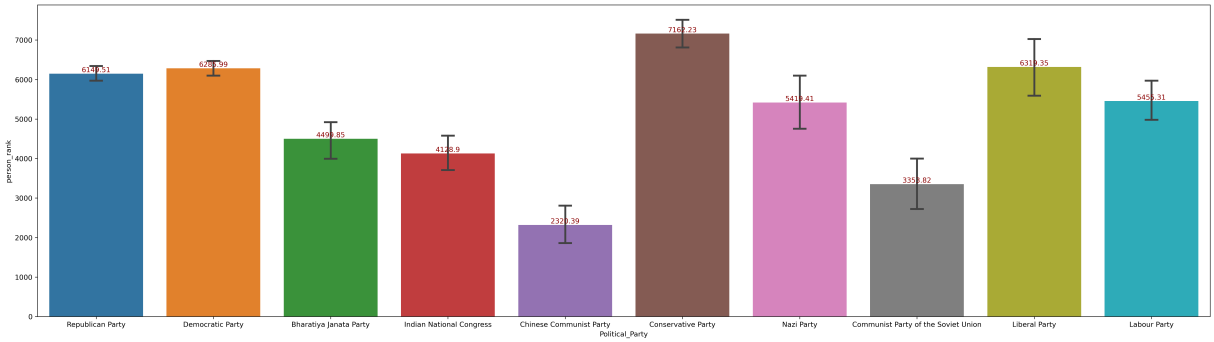

  - The disparate rankings are observed.
  - Gender-fluid, female, and trans woman get lower rank or less attention, while cisgender man dand male organism get higher rank or more attention.
  - Explain: Males, especially cisgender males, have historically been more documented, for example, Patrilineal society.

- **Religious**:




  - The disparate rankings are observed.
  - Sunni Islam and Islam ranks higher, while Anglicanism, Catholocism, and Catholic Church ranks lower.
  - Explain: Islam is one of the largest religions with a long history and obtain worldwide attention.

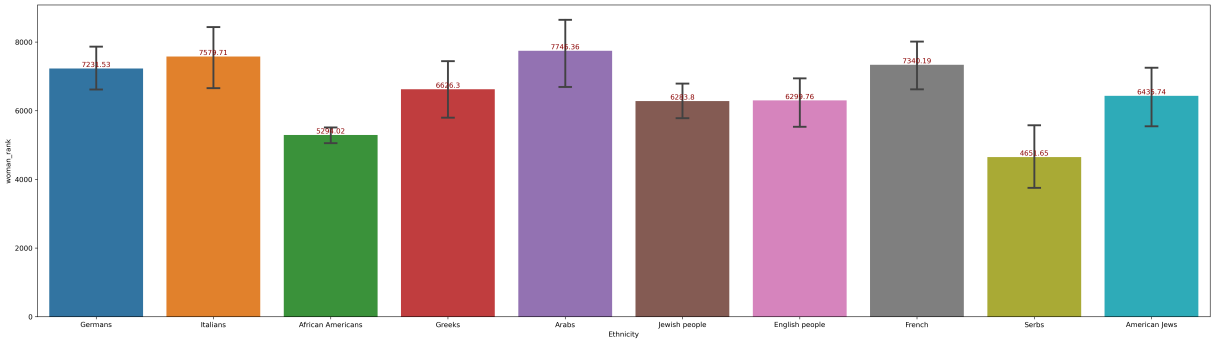
- **Political**:




  - The disparate rankings are observed.
  - Chinese Communist Party ranks higher, while Conservative Party ranks lower.
  - Explain: China's superpower and Chinese Communist Party plays an important role in the international relationships.

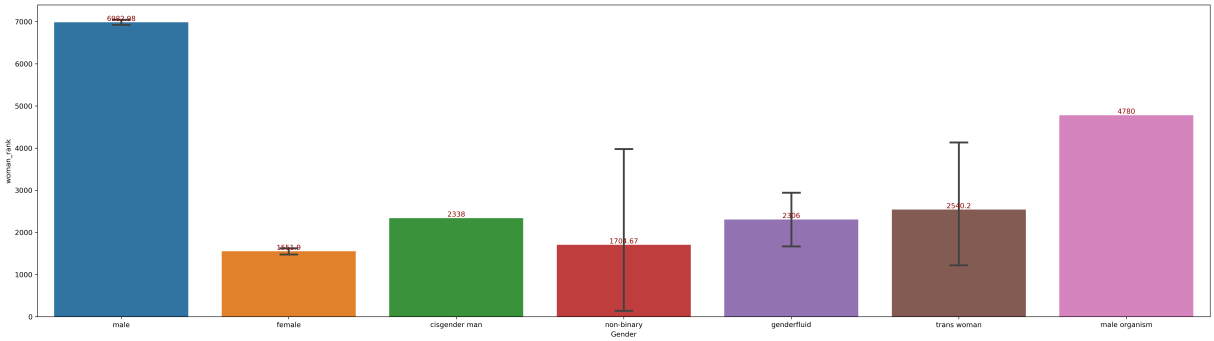
**Query 2: "woman"**

- **Ethnicity**:



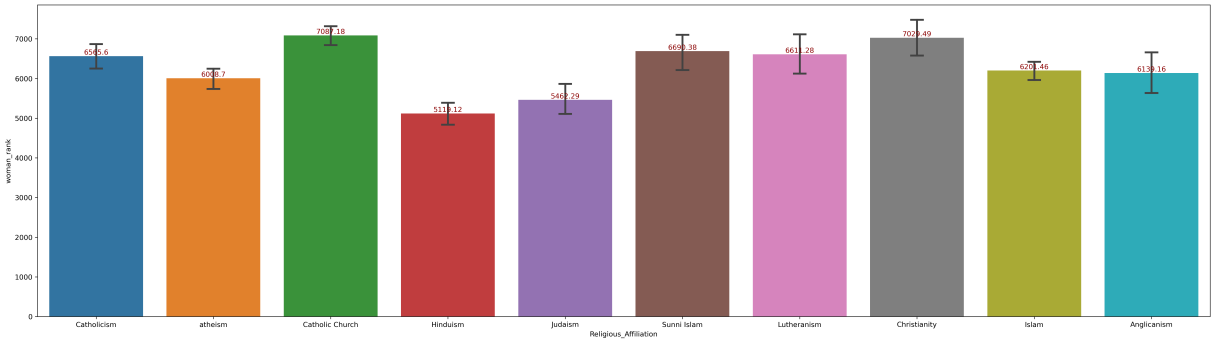

  - The disparate rankings are observed.
  - Serbs and African Americans rank higher, while Arabs and Italians rank lower.
  - Explain: Serbian and African American women get much attention. People may like dating with Serbian woman as the first Bing search result is "Serbian Women: What Makes Them Perfect Girlfriends?". Besides, societies pay much attention to influential African American women.

- **Gender**:




  - The disparate rankings are observed.
  - Female and non-binary gender ranks higher, while male ranks lower.
  - Explain: it is because the query is "woman" so it would retrieve more female than male.

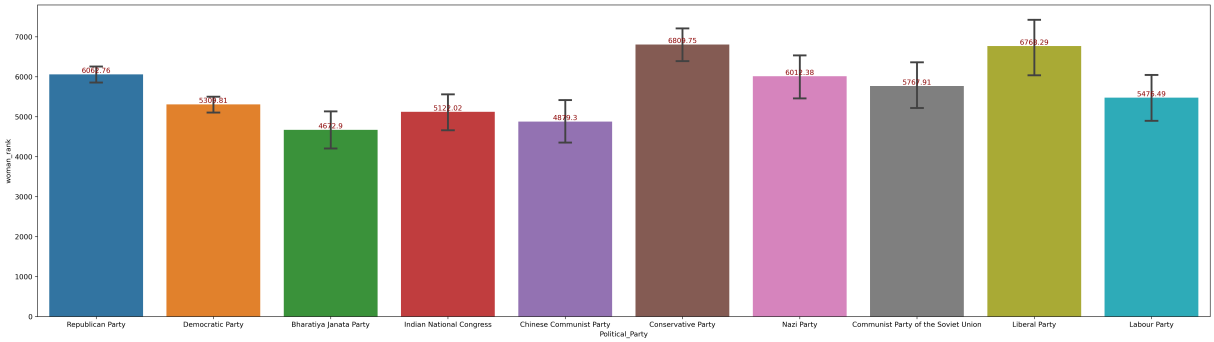
- **Religious**:




  - The disparate rankings are observed.

- Hinduism ranks higher, while Catholic Church ranks lower.
- Explain: Hinduism has diverse and rich cultural practices involving women, while Catholic Church doesn't.

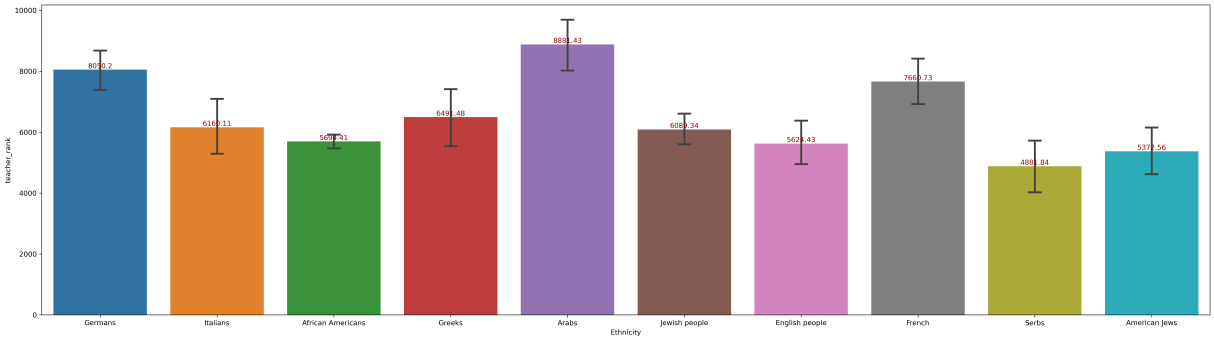
- **Political**:




- The disparate rankings are observed.
- Bharatiya janata Party and Chinese Communist Party ranks higher, while Conservation Party and Liberal Party ranks lower.
- Explain: Women in the BJP and Chinese Communist Party may play critical political roles.

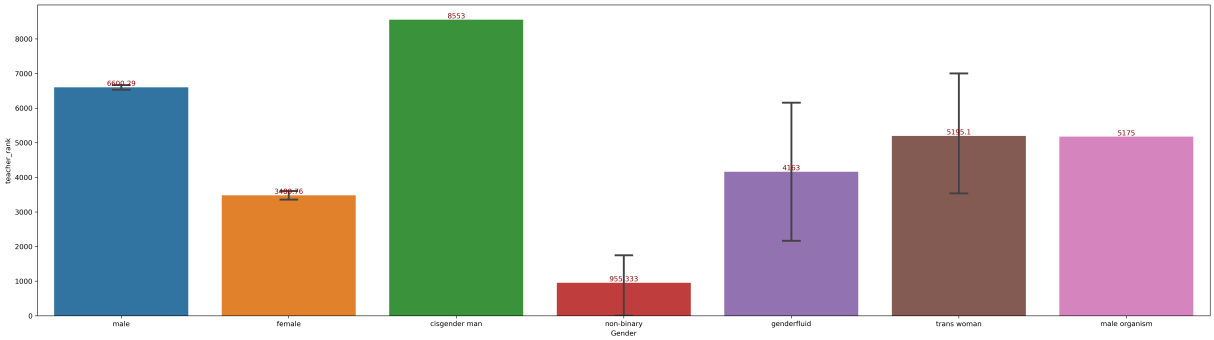
Query 3: "teacher"

- **Ethnicity**:



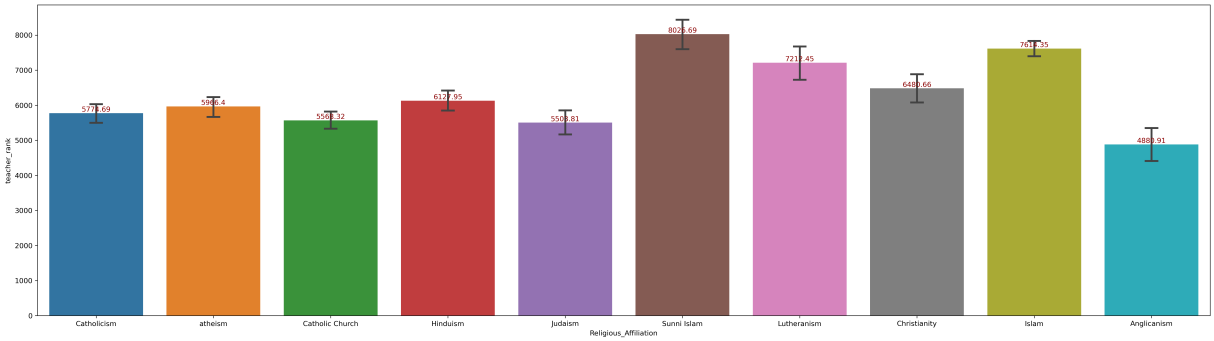

- The disparate rankings are observed.
- Serbs ranks higher, while Arabs ranks lower.
- Explain: Serbian culture might place a significant emphasis on the role of teachers, while Arab countries has less academic freedom.

- **Gender**:



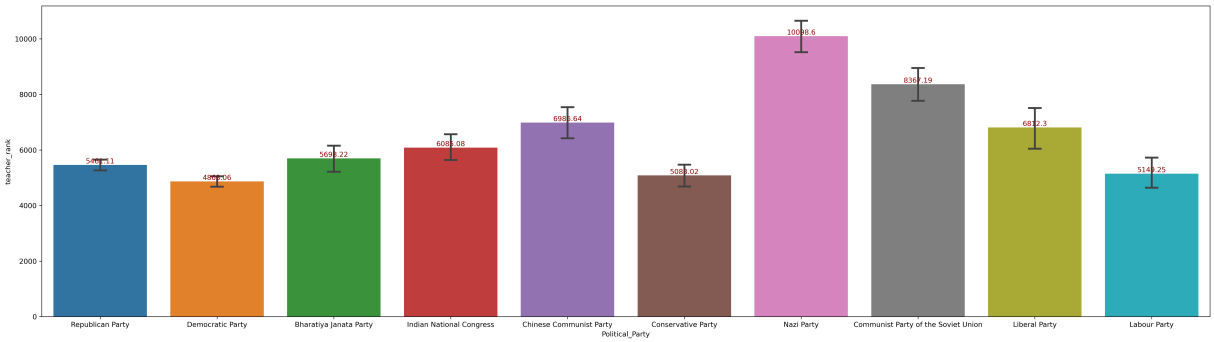

- The disparate rankings are observed.
- Non-binary gender and female ranks higher, while cisgender man ranker lower.
- Explain: Teaching is often stereotypically associated with females, which might lead to higher relevance and ranking for pages related to female and non-binary gender teachers because they tend to be more patient than male.

- **Religious**:



- The disparate rankings are observed but less noticable than other figures.
- Anglicanism ranks higher, while Sunni Islam ranks lower.
- Explain: Anglicanism might has a strong historical connection with education.

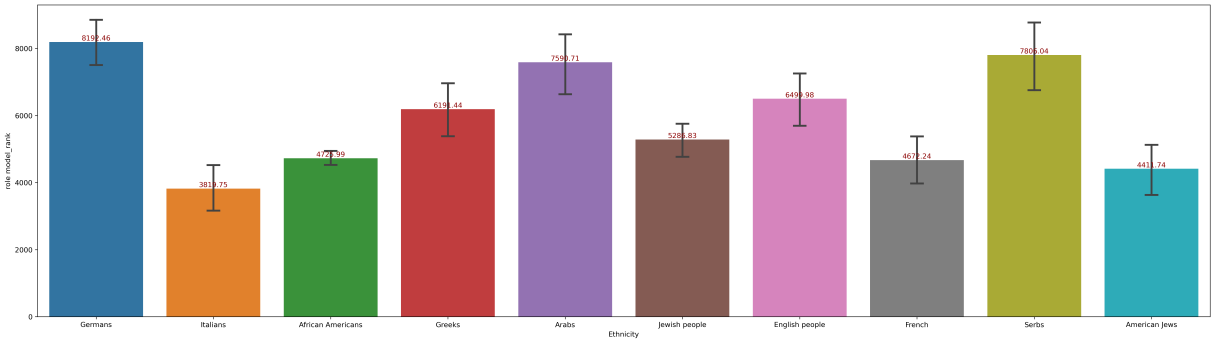
- **Political**:




- The disparate rankings are observed.
- Conservation Party ranks higher, while Nazi Party ranks lower.
- Explain: The Nazi Party has a negative historical connotation related to education -- propaganda.

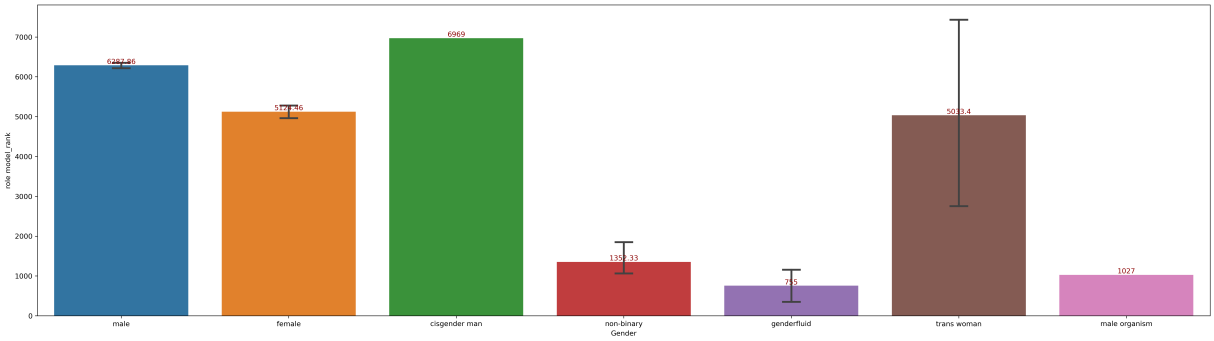
**Query 4: "role model"**

- **Ethnicity**:



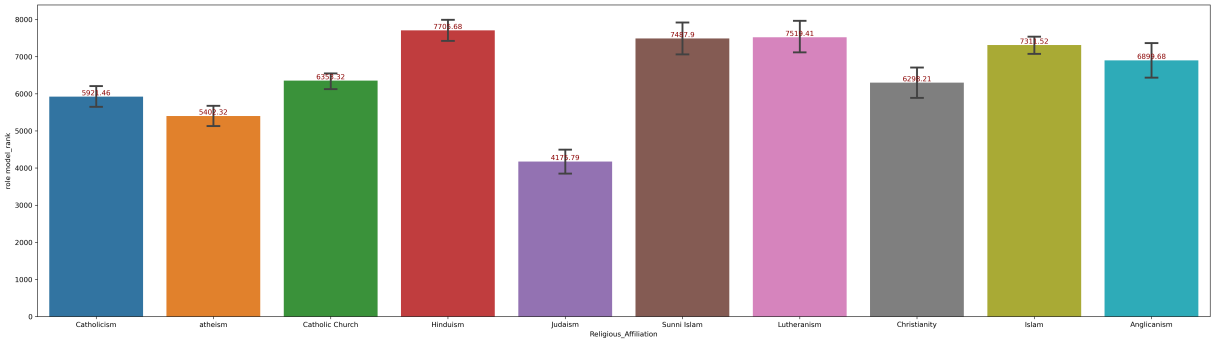

- The disparate rankings are observed.
- Italian ranks higher, while Germans ranks lower.
- Explain: there are many Italian artists and famous people that fit in with role model, while Germans are less likely.

- **Gender**:




- The disparate rankings are observed.
- Gender-fluid ranks higher, while the cisgender ranks lower.
- Explain: Gender-fluid individuals might be receiving more media visibility as role models in recent times.

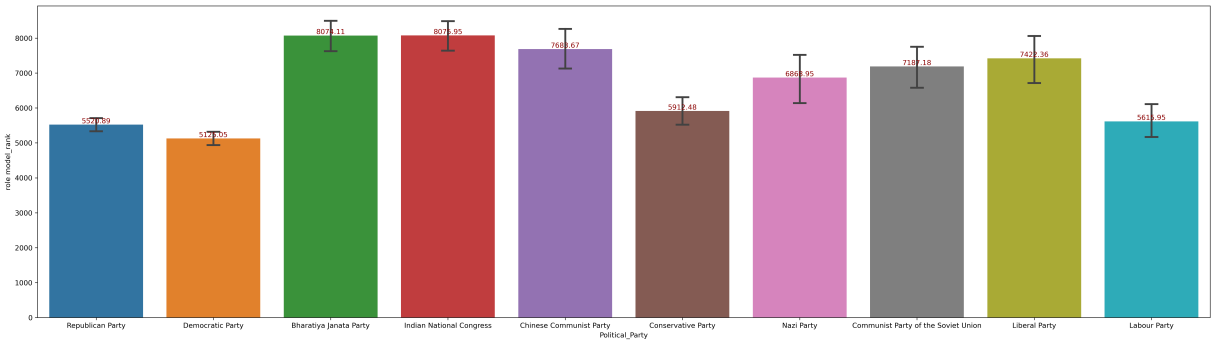
- **Religious**:




- The disparate rankings are observed.

- Judaism ranks higher, while the Hinduism ranks lower.
- Explain: Judaism have more widely recognized historical figure, and the Jewish community might be more actively promoting their role models, while Hinduism is strongly characterized by closure.

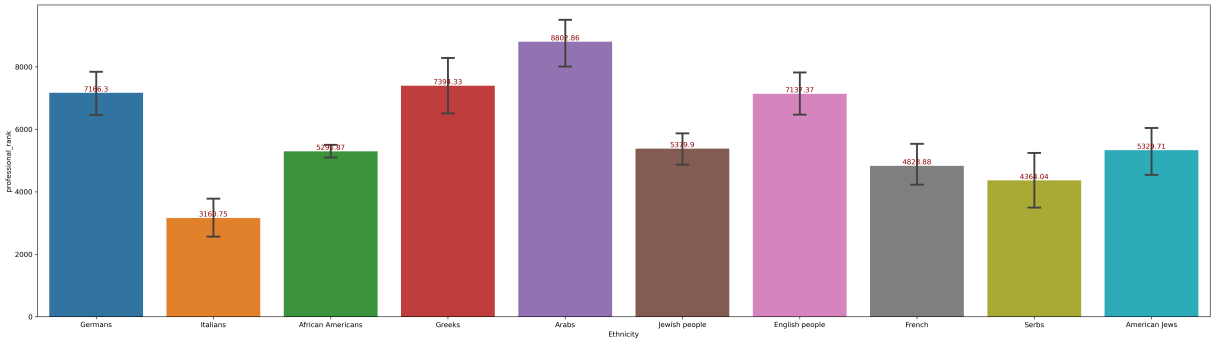
- **Political**:




- The disparate rankings are observed.
- Decormate Party ranks higher, while Bharatiya janata Party and Indian National Congress ranks lower.
- Explain: Political figures from the Democratic Party in the U.S. receive more international media coverage.

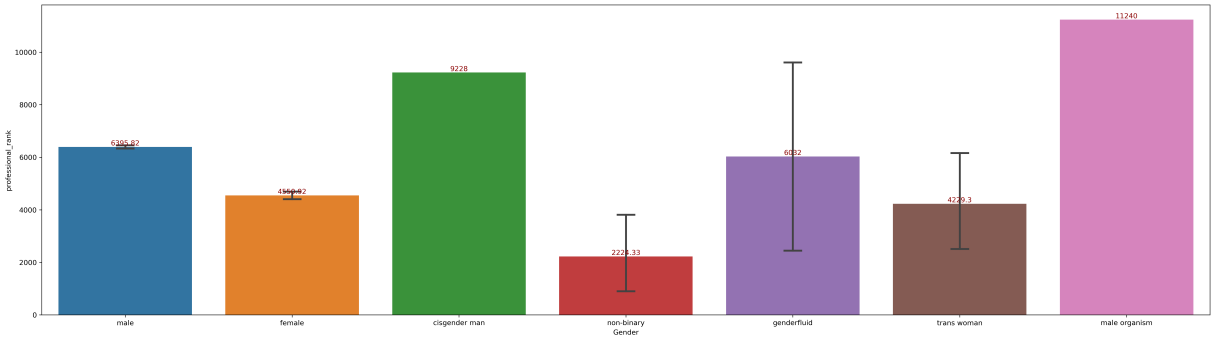
**Query 5: "professional"**

- **Ethnicity**:



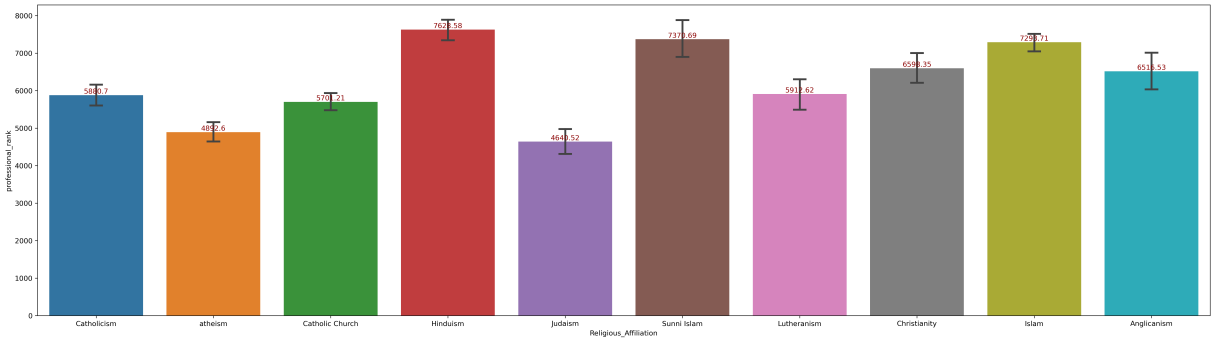

- The disparate rankings are observed.
- Italians rank higher, while Arabs ranks lower.
- Explain: Italy has a strong reputation in certain professional fields, such as fashion, design, and food.

- **Gender**:



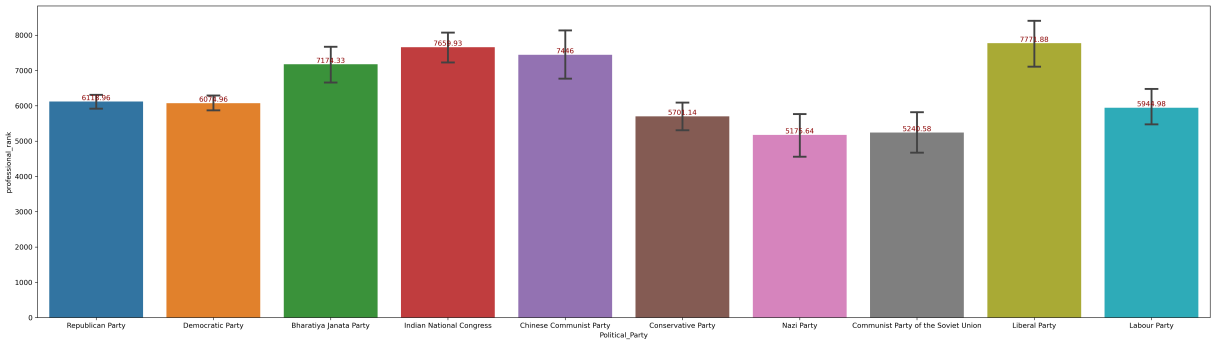

- The disparate rankings are observed.
- Non-binary geder and female rank higher, while male organism ranks lower.
- Explain: There is growing emphasis on gender diversity (especially, non-binary and female professionals) in professional settings.

- **Religious**:



- The disparate rankings are observed.
- Judaism and Atheism ranks higher, while Hinduism and Sunni Islam ranks lower.
- Explain: There are many scientists related to Judaism and Atheism.

- **Political**:




- The disparate rankings are observed.
- Nazi Party ranks higher, while Liberal Party ranks lower.
- Explain: Professional roles within the Nazi Party, especially in scientific and military fields, might be more extensively documented.

**Quantify the fairness of an IR ranker**: The main idea is to find a vector representation X without the attribute information A but still can work as a good document presentation $X^*$ or X\A.

$$L_1 \cdot d(X_a, X_b) \leq d(X_a^*, X_b^*),$$

where $X_a$ and $X_b$ are the original representation, while $X_a^*$ and $X_b^*$ are vector representations without attribute information. It would be better if we have

$$d(X_a^*, X_b^*) \leq L_2 \cdot d(X_a, X_b)$$

Therefore, we can actually see the variance of the group mean $\{\mu_k\}$ based on this attribute information can serve as a metric for fairness:

$$X^* = \text{argmin}_\text{X} \sum_{\text{i}=1}^{\text{k}} \frac{\|\mu - \mu_\text{k}\|^2}{\sigma_\text{k}^2}$$

For each term, it serve a normalized metric for how the group mean deviates from the total mean.

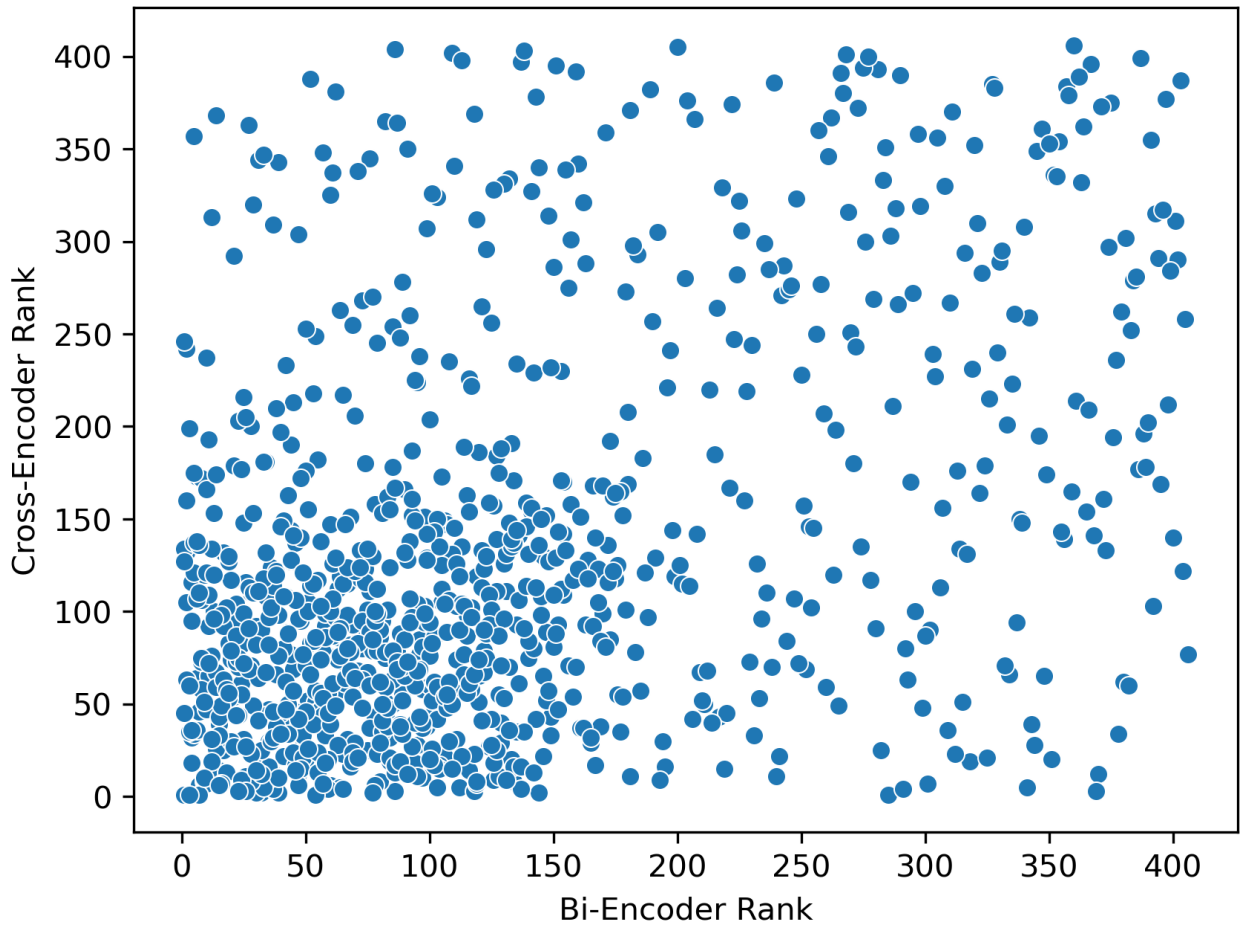
## Problem 4

I choose 5 queries from the development data. They are:

1. "What is the history and cultural importance of traditional Chinese martial arts"
2. "How are countries responding to the challenges of misinformation and disinformation campaigns"
3. "Analyze the role of architecture in promoting sustainability and green design"
4. "How do colleges and universities ensure campus safety and security"
5. "How have smartphones influenced the gaming industry and mobile gaming trends"

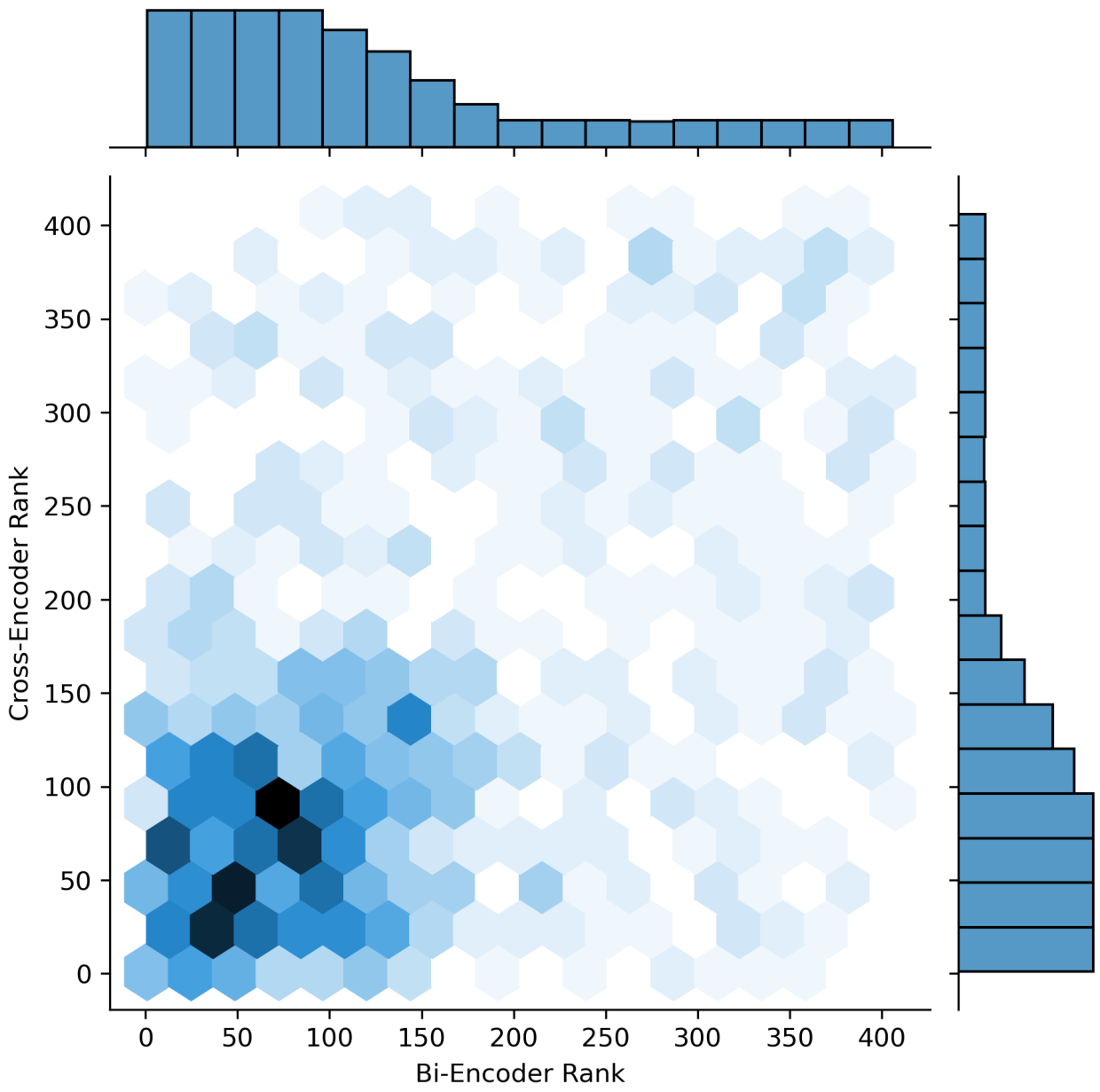
**Spearman's correlation**: 0.5829

**Scatter plot**




As it is difficult to extract information from scatter plot, I also draw a hex plot for convenience.

**Hex plot**

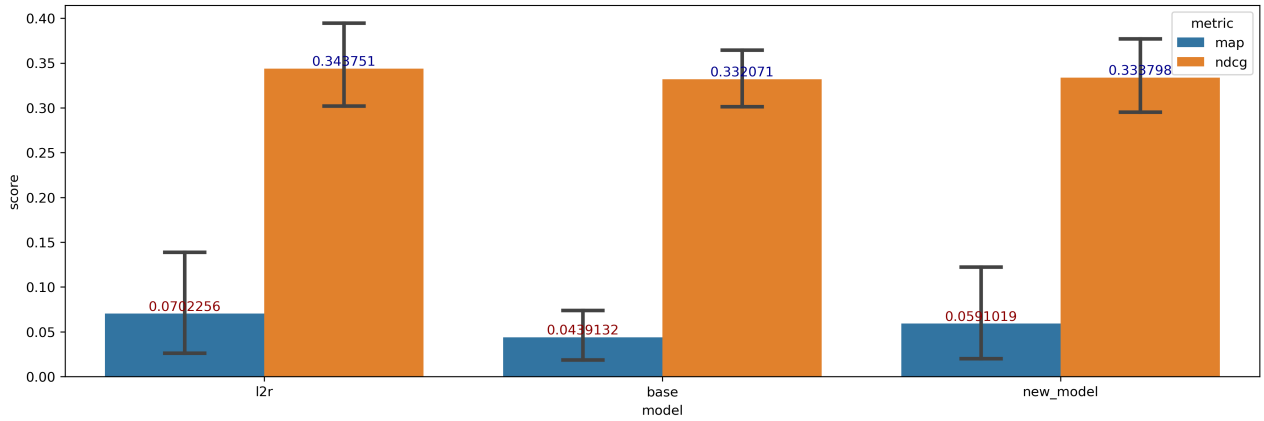



**Descriptions**: From the figure, we can see that for document rank around 1 - 25 in bi-encoder, they will be most likely to be ranked around 50 - 100 in cross encoder, which is indicated by the darkness of each hex.

**Observations**:

- Similarities: It can be seen that the two models have most agreement for the set of the top 150 document as the hex in the diagonal are darker than the rest though they may disagree with each other in the inner order of these 150 documents.
- Differences: Hoever, there are some data points in the top left and bottom right, which indicates the discrepency between the two models. Data points are not closely distributed along the diagonal.

## Problem 5



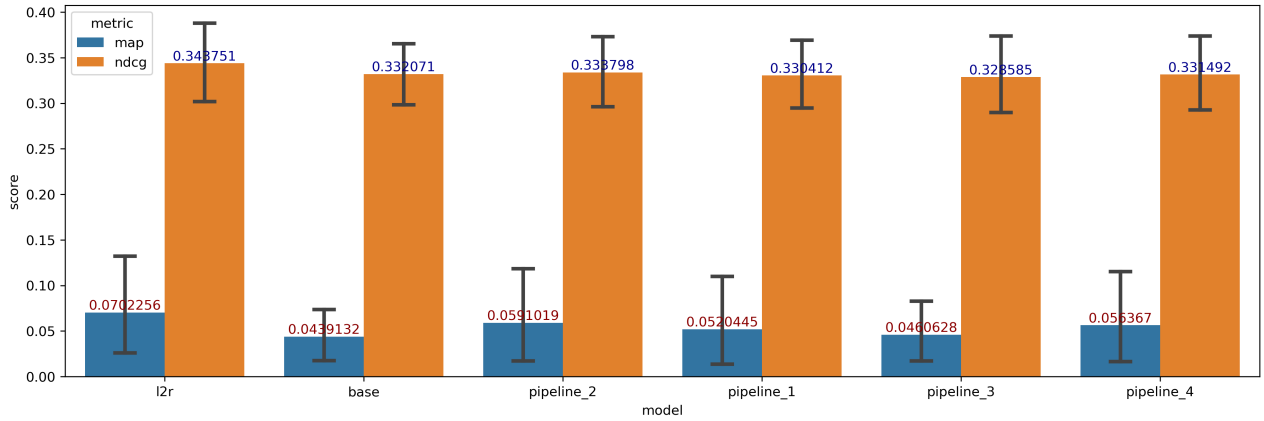

| | map | ndcg |
|---|---|---|
| base | 0.043913 | 0.332071 |
| l2r | 0.070226 | 0.343751 |
| vector_ranker | 0.059544 | 0.314000 |
| new_model | 0.059102 | 0.333798 |

- Adding deep learning-based features help us improve from baseline BM25 as pure vector ranker outperforms BM25 in MAP but not in NDCG. And the cross encoder features help us improve NDCG without lowering MAP too much.
- But, it still can't beat our hw2 models. It may be the poor quality of training data as some queries don't have document with score higher than 3 (namely, all negative samples).

### Ablation Study

- BM25 (hw2)
  - None: base
  - l2rranker: l2r
- l2rranker + index_augment + cross_encoder
  - BM25: pipeline_1
  - VectorRanker: pipeline_2
- vector_ranker + l2rranker
  - index: pipeline_3
  - index_augment: pipeline_4
- vector_ranker + l2rranker + index_augment
  - None: pipeline_4
  - cross_encoder: pipeline_2

| | MAP@10 | NDCG@10 | CONFIG |
|---|---|---|---|
| base | 0.043913 | 0.332071 | index + BM25 |
| BM25 | 0.039091 | 0.331340 | index_aug + BM25 |
| VectorRanker | 0.059544 | 0.314000 | index_aug + vec_rank |
| l2r | 0.070226 | 0.343751 | index + BM25 + l2r |
| pipeline_1 | 0.052045 | 0.330412 | index_aug + BM25 + l2r (cross_enc) |
| pipeline_2 | 0.059102 | 0.333798 | index_aug + vec_rank + l2r (cross_enc) |
| pipeline_3 | 0.046063 | 0.328585 | index + vec_rank + l2r |
| pipeline_4 | 0.056367 | 0.331492 | index_aug + vec_rank + l2r |




**Discussion**

- Document augmentation might help the re-rank, but a further experiment shows that pure BM25 with document augmentation gets lower MAP in the first matching part than that without augmentations.
- VectorRanker outperforms BM25, which shows that bi-encoder are providing important information.
- Cross encoders will increase NDCG score without lowering MAP too much. It shows the effectiveness to use CrossEncoder in the reranking part.
- HW2 pipeline performs better than others, which shows that the added new features in hw2 plays a quite critical role in prediction.

# SI 650 Homework 3

- Run this notebook as a sanity check for your implementation

Creating search pipelines for two different ranking strategies.

- Pipeline 1: Initial ranking by BM25 with re-ranking by LambdaMART (Cross-encoder feature enabled)
- Pipeline 2: Initial ranking by Bi-Encoder vector ranker with re-ranking by LambdaMART (Cross-encoder feature enabled)

The corpus for the main index is augmented by doc2query queries

In [ ]:
```
import csv
from collections import Counter, defaultdict
from tqdm import tqdm
import json
import numpy as np
import pandas as pd
import time
import seaborn as sns
import matplotlib.pyplot as plt

# your modules are imported here
from indexing import Indexer, IndexType, BasicInvertedIndex
from document_preprocessor import RegexTokenizer, Doc2QueryAugmenter, read_dataset
from ranker import Ranker, BM25, CrossEncoderScorer
from vector_ranker import VectorRanker
from network_features import NetworkFeatures
from l2r import L2RFeatureExtractor, L2RRanker

from sentence_transformers import SentenceTransformer
from relevance import map_score, ndcg_score, run_relevance_tests
import torch
```

```
/Users/haoyang/miniconda3/envs/si650/lib/python3.11/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update ju
pyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [ ]:
```
# from importlib import reload
# import l2r
# reload(l2r)
# from l2r import L2RFeatureExtractor, L2RRanker
```

In [ ]:
```
# change these to point to actual file paths
STOPWORD_PATH = 'stopwords.txt'
DATASET_PATH = 'wikipedia_200k_dataset.jsonl.gz'
EDGELIST_PATH = 'edgelist.csv.gz'
NETWORK_STATS_PATH = 'network_stats.csv'
DOC_CATEGORY_INFO_PATH = 'doc_category_info.json'
RECOGNIZED_CATEGORY_PATH = 'recognized_categories.txt'
DOC2QUERY_PATH = 'doc2query.csv'
MAIN_INDEX = 'main_index_augmented'
TITLE_INDEX = 'title_index'
RELEVANCE_TRAIN_DATA = 'hw3_relevance.train.csv'
ENCODED_DOCUMENT_EMBEDDINGS_NPY_DATA = 'wiki-200k-vecs.msmarco-MiniLM-L12-cos-v5.npy'
DOCUMENT_ID_TEXT = 'document-ids.txt'
```

## Load Basic Statistics

In [ ]:
```
# Load in the stopwords

stopwords = set()
with open(STOPWORD_PATH, 'r', encoding='utf-8') as file:
    for stopword in file:
        stopwords.add(stopword.strip())
f'Stopwords collected {len(stopwords)}'
```

Out[ ]:
```
'Stopwords collected 543'
```

In [ ]:
```
# # Get the list of categories for each page (either compute it or load the pre-computed list)
# docid_to_categories = {}
# with open(DATASET_PATH, 'rt', encoding='utf-8') as file:
#     for line in tqdm(file, total=200_000):
#         document = json.loads(line)
#         docid_to_categories[document['docid']] = document['categories']
# f'Document categories collected'
```

In [ ]:
```
# # Get or pre-compute the list of categories at least the minimum number of times (specified in the homework)
# category_counts = Counter()
# for cats in tqdm(docid_to_categories.values(), total=len(docid_to_categories)):
#     for c in cats:
#         category_counts[c] += 1
# recognized_categories = set(
#     [cat for cat, count in category_counts.items() if count >= 1000])
# print("saw %d categories" % len(recognized_categories))

# # Map each document to the smallert set of categories that occur frequently
# doc_category_info = {}
# for docid, cats in tqdm(docid_to_categories.items(), total=len(docid_to_categories)):
#     valid_cats = [c for c in cats if c in recognized_categories]
#     doc_category_info[docid] = valid_cats
```

In [ ]:
```
doc_category_info = {}
with open(DOC_CATEGORY_INFO_PATH, 'r') as f:
    doc_category_info = json.load(f)
    doc_category_info = {int(k): v for k, v in doc_category_info.items()}

recognized_categories = set()
with open(RECOGNIZED_CATEGORY_PATH, 'r') as f:
    recognized_categories = set(map(lambda x: x.strip(), f.readlines()))
```

In [ ]:
```
network_features = {}
# Get or load the network statistics for the Wikipedia link network

# if True:
#     nf = NetworkFeatures()
#     print('loading network')
#     graph = nf.load_network(EDGELIST_PATH, total_edges=92650947)
#     print('getting stats')
#     net_feats_df = nf.get_all_network_statistics(graph)
#     graph = None
#     print('Saving')
#     net_feats_df.to_csv(NETWORK_STATS_PATH, index=False)

#     print("converting to dict format")
#     network_features = defaultdict(dict)
#     for i, row in tqdm(net_feats_df.iterrows(), total=len(net_feats_df)):
#         for col in ['pagerank', 'hub_score', 'authority_score']:
#             network_features[row['docid']][col] = row[col]
#     net_feats_df = None
# else:
#     with open(NETWORK_STATS_PATH, 'r', encoding='utf-8') as file:
#         for idx, line in enumerate(file):
#             if idx == 0:
#                 continue
#             else:
#                 # the indexes may change depending on your CSV
#                 splits = line.strip().split(',')
#                 network_features[int(splits[0])] = {
#                     'pagerank': float(splits[1]),
#                     'authority_score': float(splits[2]),
#                     'hub_score': float(splits[3])
#                 }

networks_stats = pd.read_csv(NETWORK_STATS_PATH, index_col=0)
for row in tqdm(networks_stats.iterrows()):
    network_features[row[1]['docid']] = row[1][1:].to_dict()

f'Network stats collection {len(network_features)}'
```

```
999841it [00:22, 44733.99it/s]
```

Out[ ]:
```
'Network stats collection 999841'
```

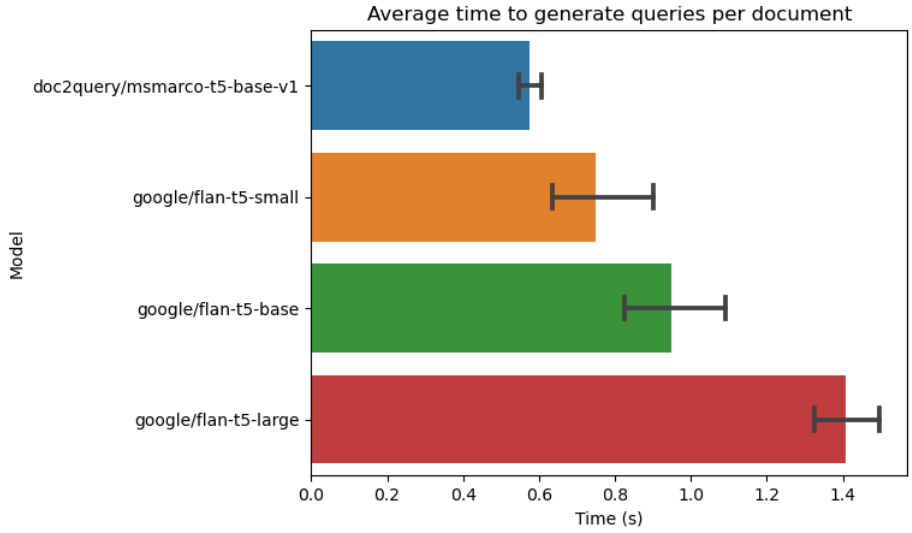
## Problem 1

In [ ]:
```
model_names = [
    'doc2query/msmarco-t5-base-v1',
    'google/flan-t5-small',
    'google/flan-t5-base',
    'google/flan-t5-large',
]

doc2query_json = defaultdict(dict)
time_json = defaultdict(list)

dataset_name = "wikipedia_200k_dataset.jsonl.gz"
max_docs = 100
batch_size = 1
n_queries = 1

token_key = "text"
pre_prefix_prompt = "Generate a query for the following text: "

for model_name in model_names:
    if model_name == "doc2query/msmarco-t5-base-v1":
        prefix_prompt = ''
    else:
        prefix_prompt = pre_prefix_prompt
    d2q = Doc2QueryAugmenter(model_name)
    queries = {}
    for doc in tqdm(read_dataset(dataset_name, max_docs, batch_size),
            desc=f"Generating queries for {model_name}",
            total=max_docs//batch_size):
        t1 = time.time()
        doc_id = doc["docid"]
        texts = doc[token_key]
        if batch_size > 1:
            batch_queries = d2q.get_batched_queries(texts, n_queries, prefix_prompt=prefix_prompt)
            for id, doc_queries in zip(doc_id, batch_queries):
                queries[id] = doc_queries
        else:
            queries[doc_id] = d2q.get_queries(texts, n_queries, prefix_prompt=prefix_prompt)
        t2 = time.time()
        time_json[model_name].append(t2-t1)
    doc2query_json[model_name].update(queries)

with open("doc2query.json", "w") as f:
    json.dump(doc2query_json, f, indent=4)
```

```
You are using the legacy behaviour of the <class 'transformers.models.t5.tokenization_t5.T5Tokenizer'>. This means that tokens that c
ome after special tokens will not be properly handled. We recommend you to read the related pull request available at https://github.
com/huggingface/transformers/pull/24565
Generating queries for doc2query/msmarco-t5-base-v1: 100%|██████████| 100/100 [00:57<00:00,  1.73it/s]
Generating queries for google/flan-t5-small: 100%|██████████| 100/100 [01:15<00:00,  1.33it/s]
Generating queries for google/flan-t5-base: 100%|██████████| 100/100 [01:35<00:00,  1.05it/s]
Generating queries for google/flan-t5-large: 100%|██████████| 100/100 [02:20<00:00,  1.41s/it]
```

In [ ]:
```python
import seaborn as sns

p1_df = pd.DataFrame({
    "model": list(time_json.keys()),
    "time": time_json.values()
}).explode("time").reset_index(drop=True)

with open("doc2query_time.json", "w") as f:
    json.dump(time_json, f, indent=4)

sns.barplot(data=p1_df, y="model", x="time", errorbar="ci", capsize=.2, orient="h")
plt.xlabel("Time (s)")
plt.ylabel("Model")
plt.savefig("doc2query_time.png", dpi=300, bbox_inches="tight")
plt.title("Average time to generate queries per document")
```

Out[ ]:
```
Text(0.5, 1.0, 'Average time to generate queries per document')
```




## Augmented Indexing

In [ ]:
```python
doc_augment_dict = defaultdict(lambda: [])
# with open(DOC2QUERY_PATH, 'r', encoding='utf-8') as file:
#     dataset = csv.reader(file)
#     for idx, row in tqdm(enumerate(dataset), total=600_000):
#         if idx == 0:
#             continue
#         doc_augment_dict[int(row[0])].append(row[2])

doc2query_df = pd.read_csv(DOC2QUERY_PATH).dropna()
for row in tqdm(doc2query_df.iterrows(), total=len(doc2query_df)):
    doc_id = int(row[1]['doc'])
    doc_query = row[1]['query']
    doc_augment_dict[doc_id].append(doc_query)
```

```
100%|██████████| 599565/599565 [00:07<00:00, 83058.86it/s]
```

In [ ]:
```python
# Load or build Inverted Indices for the documents' main text and titles
#
# Estiamted times:
#    Document text token counting: 4 minutes
#    Document text indexing: 5 minutes
#    Title text indexing: 30 seconds
preprocessor = RegexTokenizer('\w+')

# Creating and saving the index

# main_index = Indexer.create_index(
#         IndexType.InvertedIndex, DATASET_PATH, preprocessor,
#         stopwords, 50, doc_augment_dict=doc_augment_dict)
# main_index.save(MAIN_INDEX)

# title_index = Indexer.create_index(
#         IndexType.InvertedIndex, DATASET_PATH, preprocessor,
#         stopwords, 0, text_key='title')
# title_index.save(TITLE_INDEX)

# Loading a preloaded index
main_index = Indexer.load_index(MAIN_INDEX)
title_index = Indexer.load_index(TITLE_INDEX)
```

```
load index: 100%|██████████| 124166/124166 [00:21<00:00, 5800.26it/s]
load index: 100%|██████████| 90876/90876 [00:00<00:00, 830449.93it/s]
```

In [ ]:
```
# create the raw text dictionary by going through the wiki dataset
# this dictionary should store only the first 500 words of the raw documents text

raw_text_dict = {}
for doc in tqdm(read_dataset(DATASET_PATH, 200_000, 1), total=200_000):
    doc_id = int(doc['docid'])
    raw_text_dict[doc_id] = " ".join(preprocessor.tokenize(doc['text'])[:500])
```

```
100%|██████████| 200000/200000 [01:15<00:00, 2665.92it/s]
```

## Problem 2

In [ ]:
```
def fetch_related_docs(relevalce_data_path: str, dataset_path: str):
    relevance_dev_df = pd.read_csv(RELEVANCE_DEV_DATA)
    dev_docids = set(relevance_dev_df['docid'].unique())

    related_docids = []
    related_docs = []
    for doc in tqdm(read_dataset(dataset_path), total=200_000):
        doc_id = int(doc['docid'])
        if doc_id in dev_docids:
            related_docids.append(doc_id)
            related_docs.append(doc['text'])
    return related_docids, related_docs
```

In [ ]:
```
RELEVANCE_DEV_DATA = "hw3_relevance.dev.csv"
dev_docids, dev_docs = fetch_related_docs(RELEVANCE_DEV_DATA, DATASET_PATH)
```

```
100%|██████████| 200000/200000 [00:14<00:00, 13779.38it/s]
```

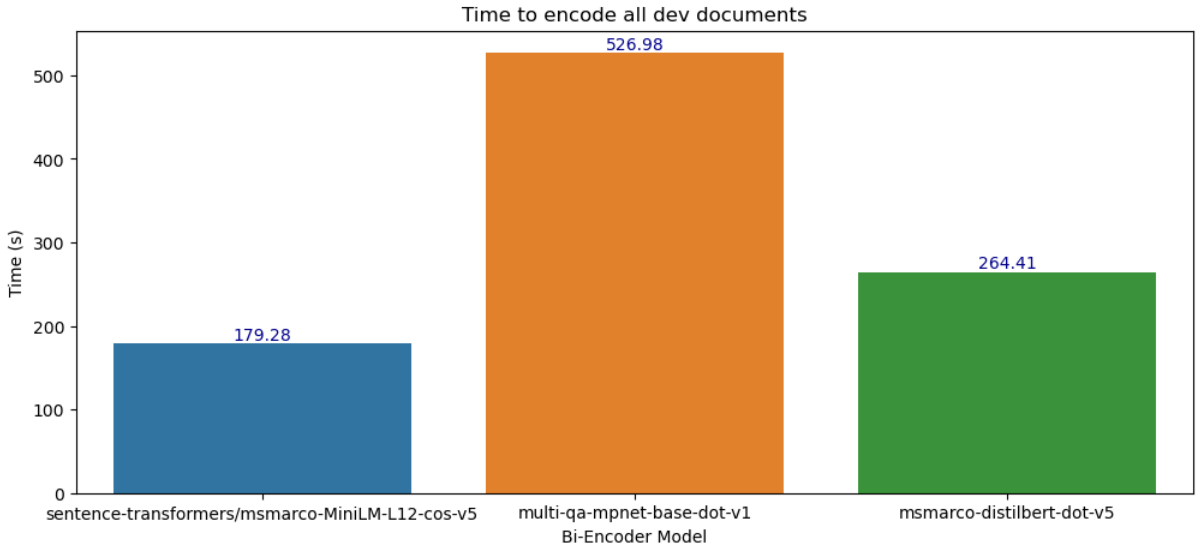
Generate encoded_docs

In [ ]:
```
bi_encoder_names = [
    "sentence-transformers/msmarco-MiniLM-L12-cos-v5",
    "multi-qa-mpnet-base-dot-v1",
    "msmarco-distilbert-dot-v5"
]

test_filename = 'hw3_relevance.test.csv'

cescorer = None
fe = L2RFeatureExtractor(main_index, title_index, doc_category_info,
                         preprocessor, stopwords, recognized_categories,
                         network_features, cescorer)

named_rankers = {}
named_rankers['BM25'] = Ranker(main_index, preprocessor, stopwords, BM25(main_index))

encoder_time_info = {}
for bi_encoder_name in bi_encoder_names:
    encoder = SentenceTransformer(bi_encoder_name)
    if torch.cuda.is_available():
        encoder = encoder.to('cuda')
    elif torch.backends.mps.is_built():
        encoder = encoder.to('mps')
    t1 = time.time()
    encoded_docs = encoder.encode(dev_docs, batch_size=128, show_progress_bar=True, normalize_embeddings=True)
    t2 = time.time()
    encoder_time_info[bi_encoder_name] = t2 - t1
    vector_ranker = VectorRanker(bi_encoder_name, encoded_docs, dev_docids)
    named_rankers[bi_encoder_name] = vector_ranker
```

```
Batches: 100%|██████████| 17/17 [02:57<00:00, 10.42s/it]
Batches: 100%|██████████| 17/17 [08:44<00:00, 30.85s/it]
Batches: 100%|██████████| 17/17 [04:22<00:00, 15.43s/it]
```

Encoding time

In [ ]:
```
plt.figure(figsize=(12, 5))
ax = sns.barplot(x = list(encoder_time_info.keys()), y = list(encoder_time_info.values()))
ax.bar_label(ax.containers[0], fmt='%.2f', color='darkblue')
plt.xlabel("Bi-Encoder Model")
plt.ylabel("Time (s)")
plt.title("Time to encode all dev documents")
plt.savefig("encoder_time.png", dpi=300, bbox_inches="tight")
```




Save encoded docs

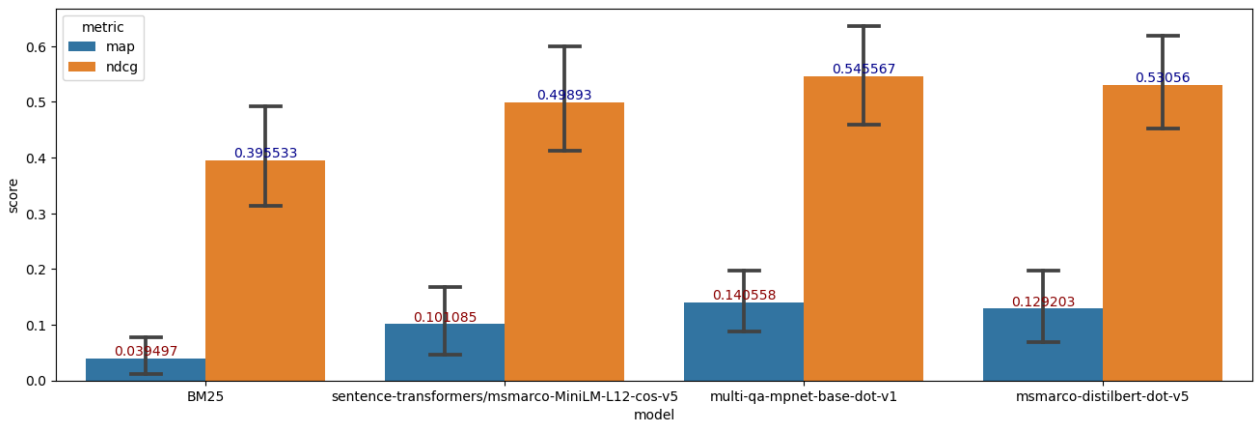
In [ ]:
```
# kwargs = {'docids': dev_docids}
# for name in bi_encoder_names:
#     encoded_docs = named_rankers[name].encoded_docs
#     kwargs[name] = encoded_docs

# np.savez('encoded_docs.npz', **kwargs)
# encoded_docs_dict = np.load('encoded_docs.npz')
```

Evaluate Bi-Encoders

In [ ]:
```
named_pipeline_info = {}
for name, ranker in named_rankers.items():
    print(f"Evaluating {name}")
    pipeline_info = run_relevance_tests(RELEVANCE_DEV_DATA, ranker)
    named_pipeline_info[name] = pipeline_info
```

```
Evaluating BM25
100%|██████████| 19/19 [00:10<00:00,  1.80it/s]
Evaluating sentence-transformers/msmarco-MiniLM-L12-cos-v5
100%|██████████| 19/19 [00:00<00:00, 35.74it/s]
Evaluating multi-qa-mpnet-base-dot-v1
100%|██████████| 19/19 [00:00<00:00, 27.60it/s]
Evaluating msmarco-distilbert-dot-v5
100%|██████████| 19/19 [00:00<00:00, 52.98it/s]
```

In [ ]:
```
tmp_pipeline_info = pd.DataFrame(named_pipeline_info).transpose().reset_index().rename(columns={'index': 'model'})
metric_add_func = lambda x, name: [(name, v) for v in x]
tmp_pipeline_info['score'] = tmp_pipeline_info[['map_scores', 'ndcg_scores']].apply(lambda x: metric_add_func(x[0], 'map') + metric_
tmp_pipeline_info = tmp_pipeline_info.explode('score')
tmp_pipeline_info['metric'] = tmp_pipeline_info['score'].apply(lambda x: x[0])
tmp_pipeline_info['score'] = tmp_pipeline_info['score'].apply(lambda x: x[1])
plt.figure(figsize=(16, 5))
ax = sns.barplot(data=tmp_pipeline_info, x='model', y='score', hue='metric', capsize=.1, errorbar='ci')
ax.bar_label(ax.containers[0], fontsize=10, color='darkred')
ax.bar_label(ax.containers[1], fontsize=10, color='darkblue')
plt.savefig('bi_encoder_l2r_metrics.png', dpi=300, bbox_inches='tight')
```




In [ ]:
```
with open('bi_encoder_l2r_metrics.json', 'w') as f:
    json.dump(named_pipeline_info, f, indent=4)
```

## Problem 3

10 most common labels for each of the four person attribute

In [ ]:
```
top_10_attributes = {}
person_attr_file = "person-attributes.csv"
person_attr_df = pd.read_csv(person_attr_file)
for attr in person_attr_df.columns[1:-1]:
    top_10_attributes[attr] = person_attr_df[attr].value_counts()[:10].to_dict()

with open("top_10_attributes.json", "w") as f:
    f.write(json.dumps(top_10_attributes, indent=4))

top_10_attributes_df = pd.DataFrame(
    {
        k: list(v.keys()) + [''] * (10 - len(v.keys()))
        for k, v in top_10_attributes.items()
    }
)
top_10_attributes_df
```

Out[ ]:

| | Ethnicity | Gender | Religious_Affiliation | Political_Party |
|---|---|---|---|---|
| 0 | African Americans | male | Catholic Church | Democratic Party |
| 1 | Jewish people | female | Islam | Republican Party |
| 2 | Germans | trans woman | atheism | Conservative Party |
| 3 | English people | non-binary | Catholicism | Labour Party |
| 4 | French | genderfluid | Hinduism | Indian National Congress |
| 5 | American Jews | cisgender man | Judaism | Bharatiya Janata Party |
| 6 | Italians | male organism | Christianity | Communist Party of the Soviet Union |
| 7 | Greeks | | Lutheranism | Nazi Party |
| 8 | Serbs | | Anglicanism | Chinese Communist Party |
| 9 | Arabs | | Sunni Islam | Liberal Party |

Load pre-computed vectors

In [ ]:
```
encoded_docs = None
with open(ENCODED_DOCUMENT_EMBEDDINGS_NPY_DATA, 'rb') as file:
    encoded_docs = np.load(file)

with open(DOCUMENT_ID_TEXT, 'r') as f:
    document_ids = f.read().splitlines()
document_ids = [int(x) for x in document_ids]

q3_vector_ranker = VectorRanker('sentence-transformers/msmarco-MiniLM-L12-cos-v5', encoded_docs, document_ids)
queries = ["person", "woman", "teacher", "role model", "professional"]

for query in queries:
    query_scores = {k: v for k, v in q3_vector_ranker.query(query)}
    person_attr_df[query] = person_attr_df['docid'].apply(lambda x: query_scores.get(x, 0))
    person_attr_df[query + "_rank"] = person_attr_df[query].rank(ascending=False, na_option='bottom')
```

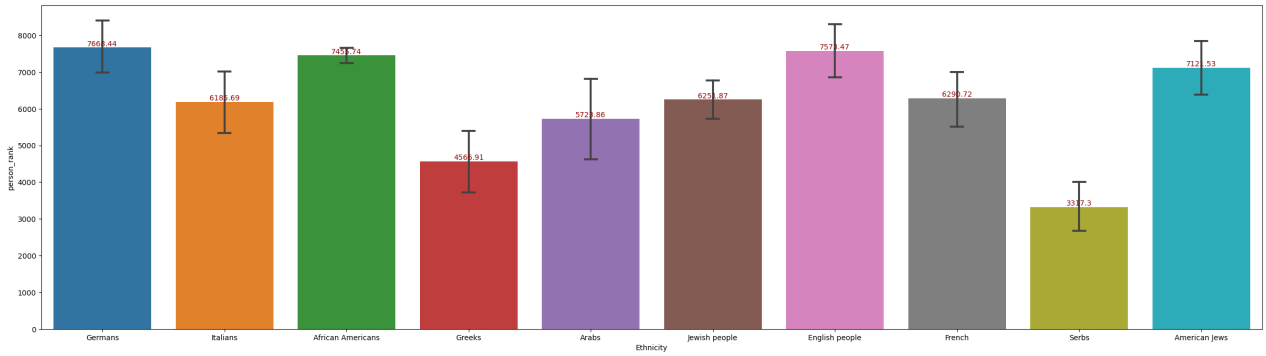
In [ ]:
```
person_attr_df.head()
```

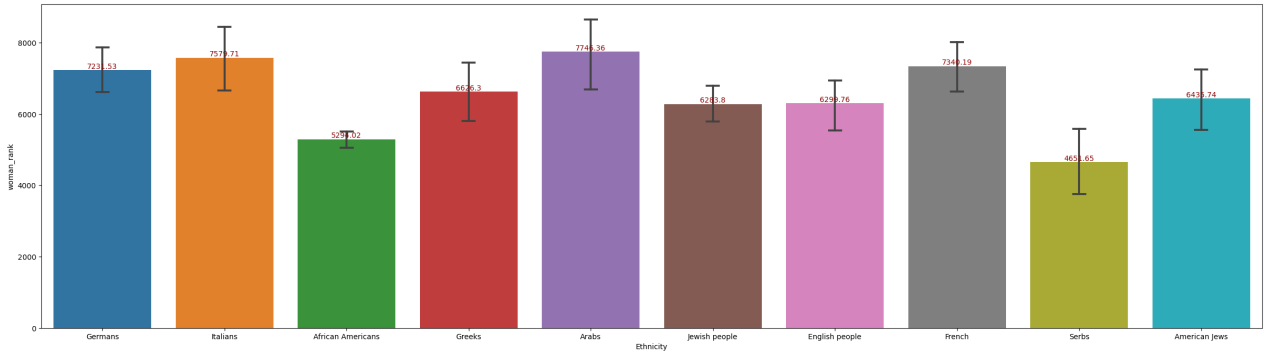
Out[ ]:

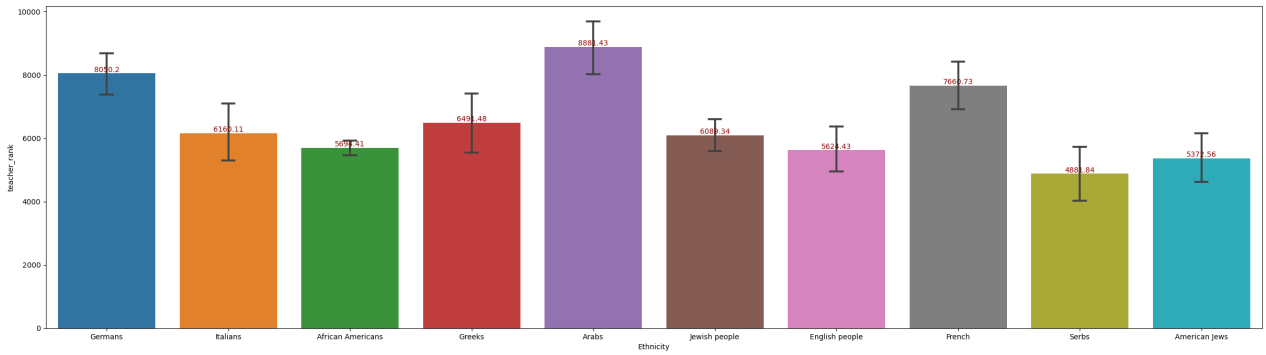
| | title | Ethnicity | Gender | Religious_Affiliation | Political_Party | docid | person | person_rank | woman | woman_rank | teacher | teacher |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | George Washington | NaN | male | Episcopal Church | independent politician | 11968 | 0.059404 | 7971.0 | -0.045185 | 10262.0 | 0.038667 | 8 |
| 1 | Douglas Adams | White British | male | NaN | NaN | 8091 | 0.111642 | 3431.0 | -0.014420 | 8440.0 | 0.082106 | 5 |
| 2 | George W. Bush | NaN | male | United Methodist Church | Republican Party | 3414021 | 0.077822 | 6372.0 | -0.039768 | 10001.0 | 0.045824 | 1 |
| 3 | Diego Velázquez | Spaniards | male | NaN | NaN | 77423 | 0.076067 | 6537.0 | 0.008360 | 6804.0 | -0.023389 | 1 |
| 4 | Augusto Pinochet | NaN | male | Catholicism | independent politician | 18933396 | 0.127831 | 2280.0 | -0.039218 | 9969.0 | -0.029851 | 1 |

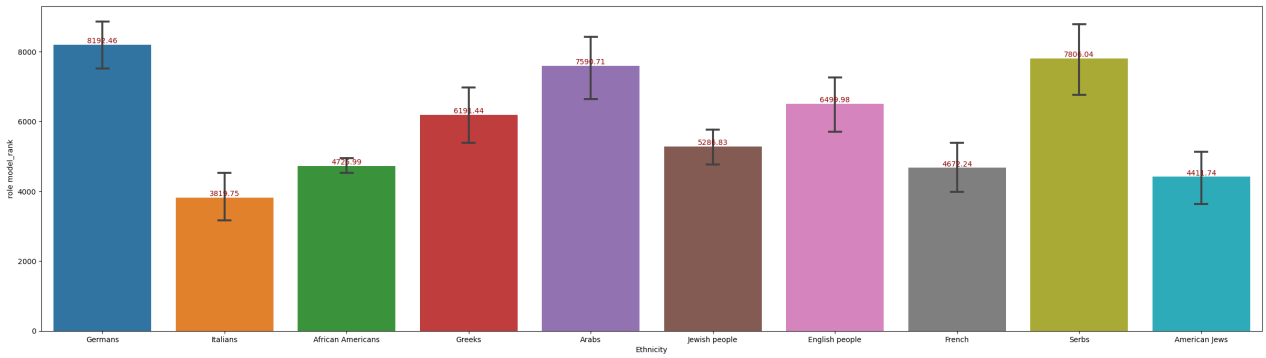
In [ ]:
```
person_attr_df.to_csv("person-attributes-queries.csv", index=False)
```

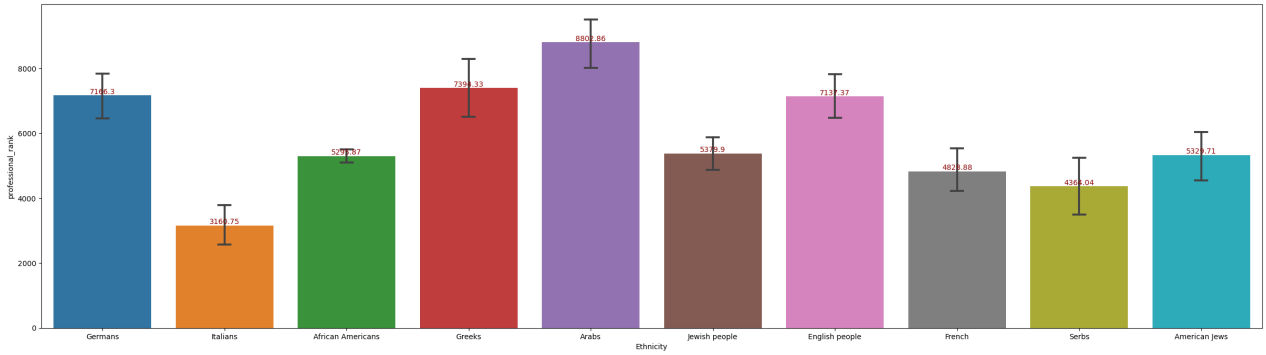
In [ ]:
```
import os
if not os.path.exists('plots'):
    os.mkdir('plots')

attr_names = ['Ethnicity', 'Gender', 'Religious_Affiliation', 'Political_Party']
for attr_name in attr_names:
    top_10_categories = top_10_attributes[attr_name]
    filtered_df = person_attr_df[person_attr_df[attr_name].isin(top_10_categories.keys())]
    for query in queries:
        plt.figure(figsize=(30, 8))
        ax = sns.barplot(data=filtered_df, x=attr_name, y=query + "_rank", errorbar='ci', capsize=.1)
        ax.bar_label(ax.containers[0], fontsize=10, color='darkred')
        plt.savefig(f"plots/{attr_name}_{query}.png", dpi=300, bbox_inches='tight')
```

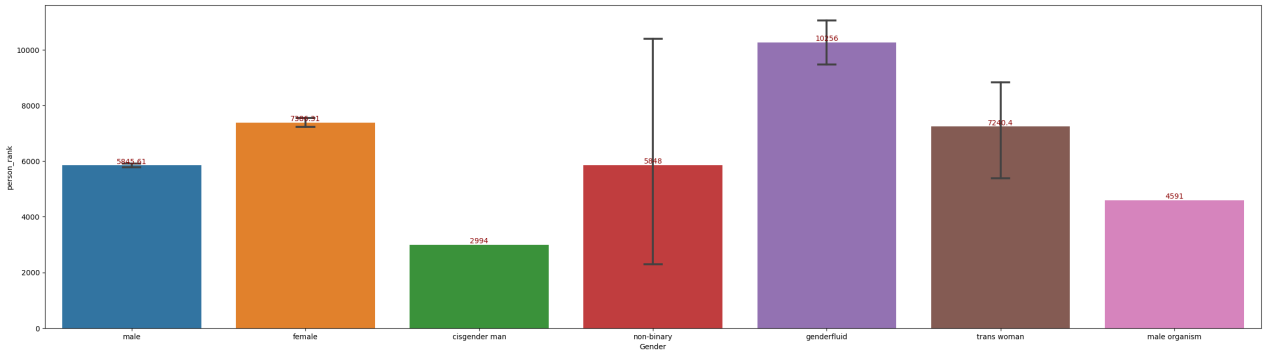

person_rank
Ethnicity
Germans
Italians
African Americans
Greeks
Arabs
Jewish people
English people
French
Serbs
American Jews
7669.44
6183.69
7491.74
4569.91
5729.86
6251.87
7573.47
6290.72
3317.3
7125.53

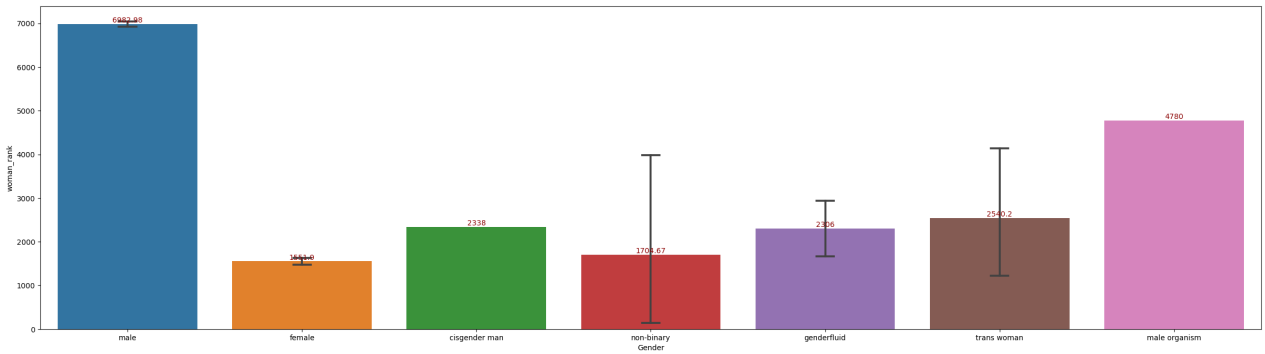

woman_rank
Ethnicity
Germans
Italians
African Americans
Greeks
Arabs
Jewish people
English people
French
Serbs
American Jews
7231.53
7570.71
5297.02
6626.3
7746.36
6263.8
6290.76
7346.19
4651.65
6431.74

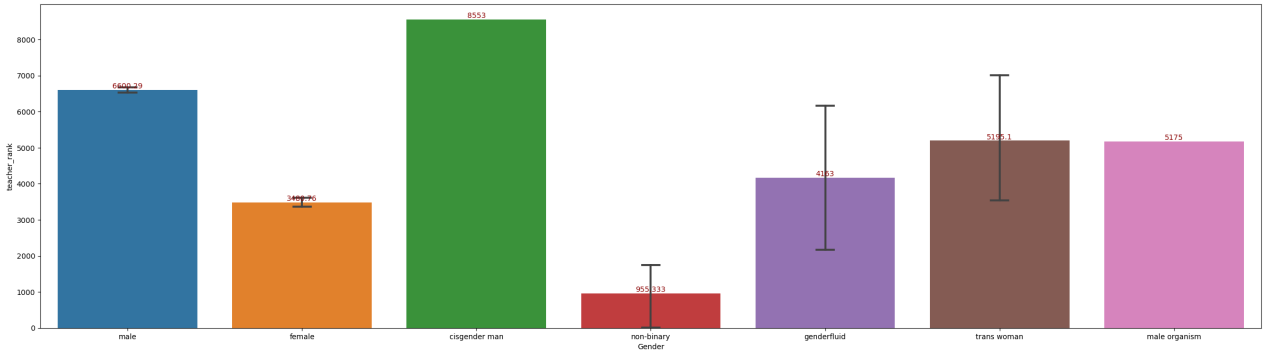

teacher_rank
Ethnicity
Germans
Italians
African Americans
Greeks
Arabs
Jewish people
English people
French
Serbs
American Jews
8050.2
6160.11
5697.21
6491.48
8881.43
6089.34
5624.43
7660.73
4881.84
5377.56

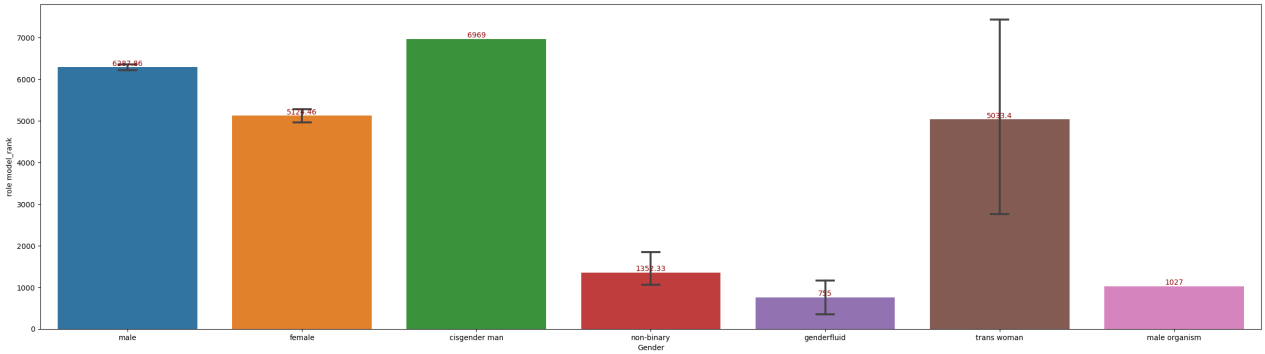

role model_rank
Ethnicity
Germans
Italians
African Americans
Greeks
Arabs
Jewish people
English people
French
Serbs
American Jews
8192.46
3819.75
4723.99
6191.44
7590.71
5283.83
6499.98
4672.24
7804.04
4411.74

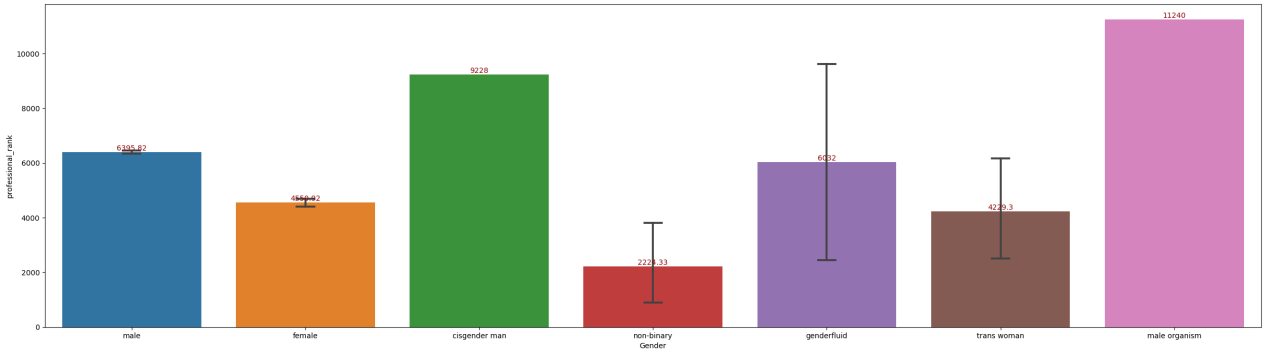

professional_rank
Ethnicity
Germans
Italians
African Americans
Greeks
Arabs
Jewish people
English people
French
Serbs
American Jews
7166.3
3160.75
5297.87
7394.33
8802.86
5379.9
7137.37
4829.88
4368.04
5329.71

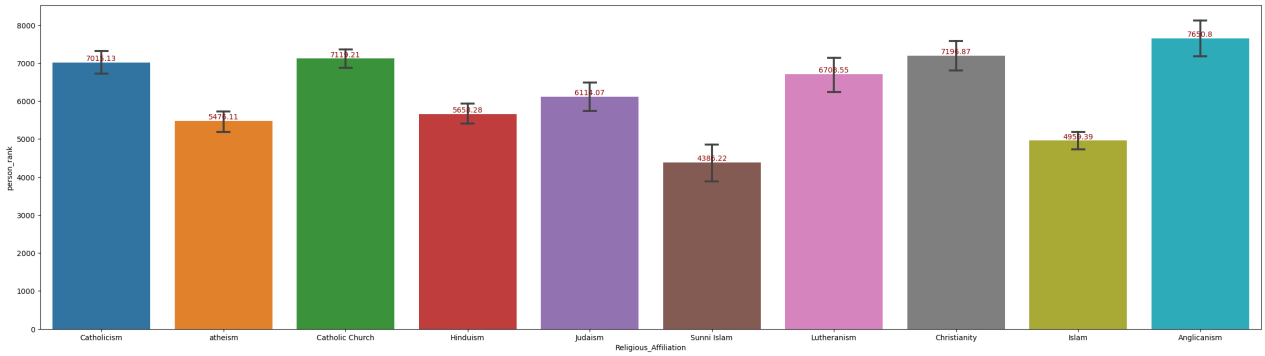

person_rank
Gender
male
female
cisgender man
non-binary
genderfluid
trans woman
male organism
10256
7240.4
4591
2994

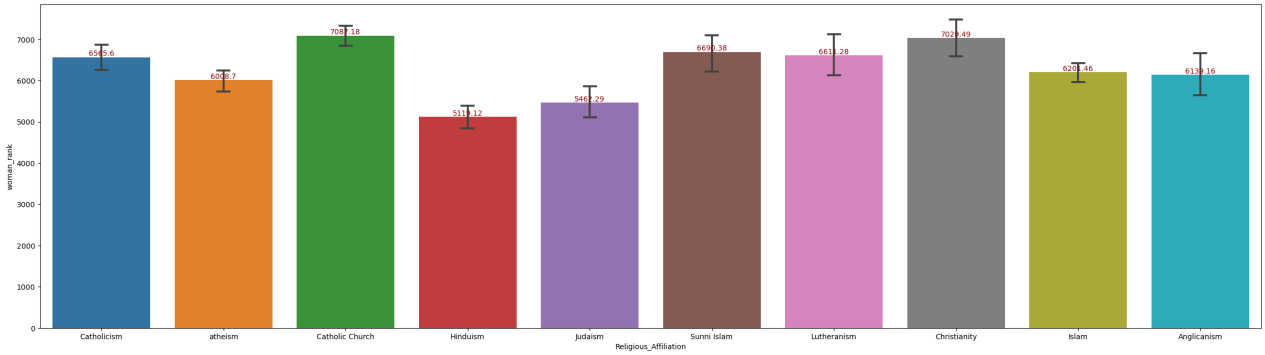

woman_rank
Gender
male
female
cisgender man
non-binary
genderfluid
trans woman
male organism
2338
2306
2540.2
4780

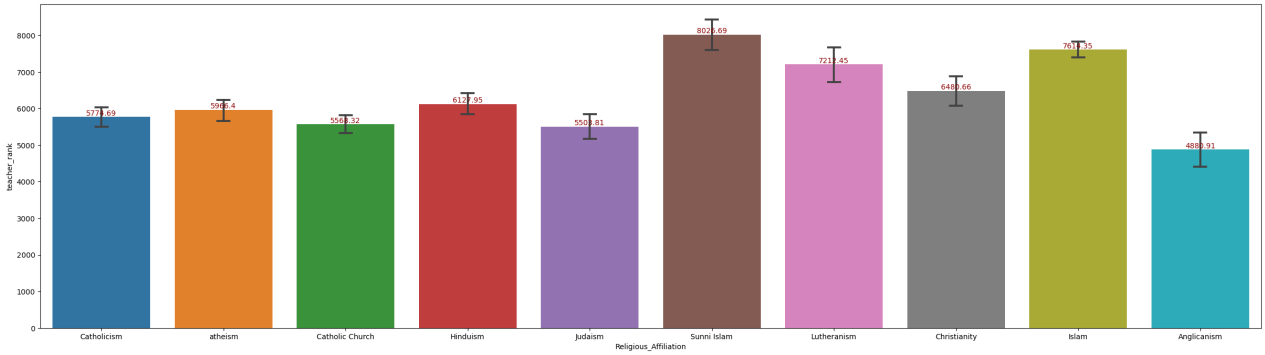

teacher_rank
Gender
male
female
cisgender man
non-binary
genderfluid
trans woman
male organism
8553
4103
5175
955.333

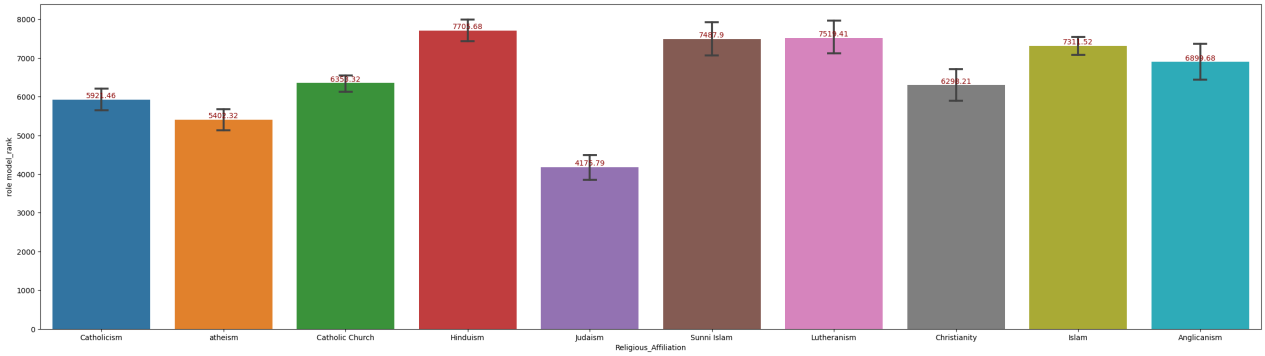

role model_rank
Gender
male
female
cisgender man
non-binary
genderfluid
trans woman
male organism
6969
1359.33
735
1027

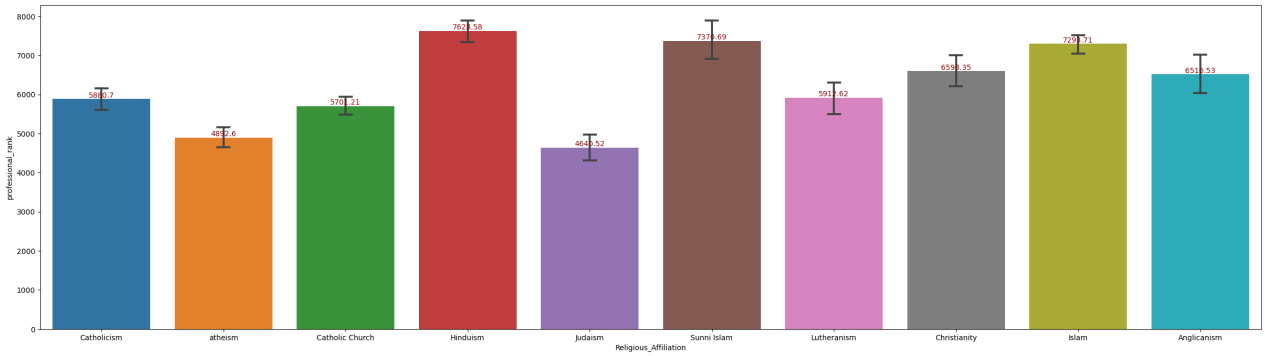

professional_rank
Gender
male
female
cisgender man
non-binary
genderfluid
trans woman
male organism
9228
6032
4229.3
11240
2229.33

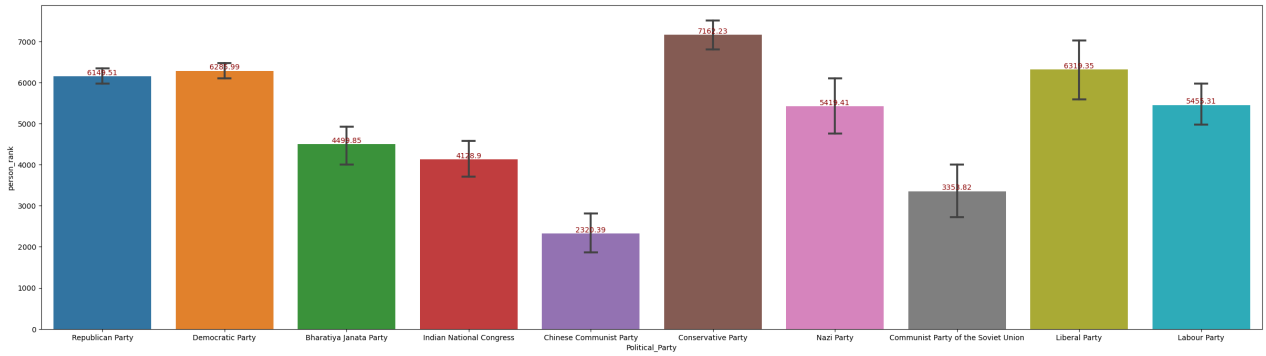

person_rank
Religious_Affiliation
Catholicism
atheism
Catholic Church
Hinduism
Judaism
Sunni Islam
Lutheranism
Christianity
Islam
Anglicanism
7016.13
5470.11
7139.21
5654.28
6114.07
4386.22
6700.55
7196.87
4970.39
7650.8

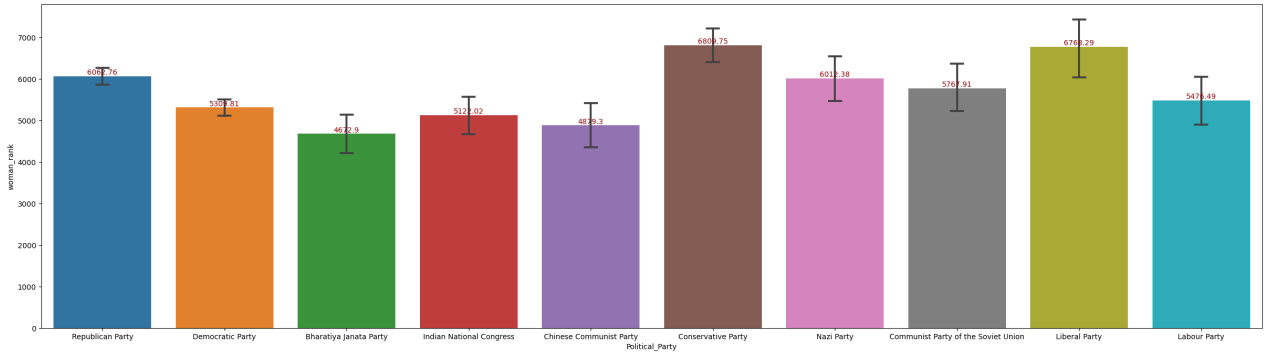

woman_rank
Religious_Affiliation
Catholicism
atheism
Catholic Church
Hinduism
Judaism
Sunni Islam
Lutheranism
Christianity
Islam
Anglicanism
6545.6
6008.7
7087.18
5113.12
5462.29
6690.38
6611.28
7029.49
6203.46
6139.16

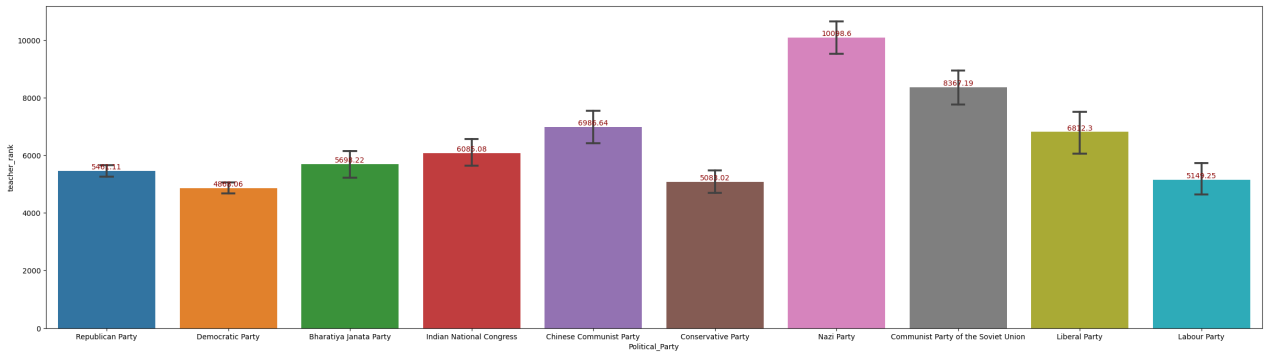

teacher_rank
Religious_Affiliation
Catholicism
atheism
Catholic Church
Hinduism
Judaism
Sunni Islam
Lutheranism
Christianity
Islam
Anglicanism
5771.69
5966.4
5581.32
6127.95
5503.81
8020.69
7211.45
6480.66
7611.35
4880.91

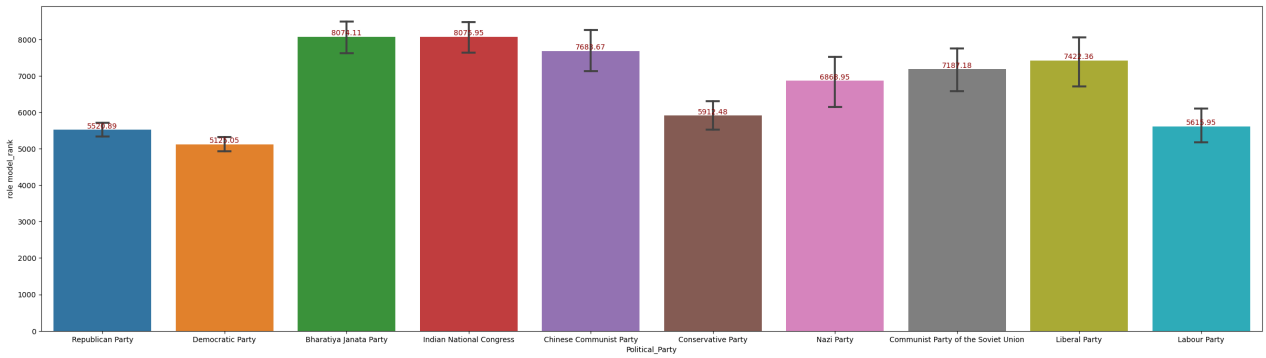

role model_rank
Religious_Affiliation
Catholicism
atheism
Catholic Church
Hinduism
Judaism
Sunni Islam
Lutheranism
Christianity
Islam
Anglicanism
5921.46
5401.32
6357.32
7703.68
4178.79
7487.9
7519.41
6298.21
7311.52
6899.68

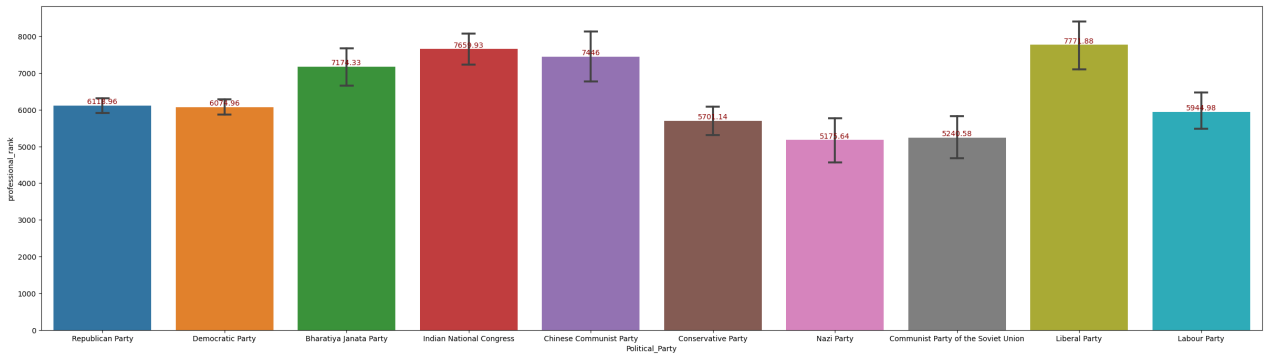

professional_rank
Religious_Affiliation
Catholicism
atheism
Catholic Church
Hinduism
Judaism
Sunni Islam
Lutheranism
Christianity
Islam
Anglicanism
5880.7
4852.6
5701.21
7624.58
4640.52
7370.69
5911.62
6598.35
7291.71
6513.53




## Cross-Encoder Ranking and Re-training

In [ ]:
```
# Create the feature extractor. This will be used by both pipelines
cescorer = CrossEncoderScorer(raw_text_dict)
fe = L2RFeatureExtractor(main_index, title_index, doc_category_info,
                         preprocessor, stopwords, recognized_categories,
                         network_features, cescorer)
p5_vector_ranker = VectorRanker('sentence-transformers/msmarco-MiniLM-L12-cos-v5', encoded_docs, document_ids)
p5_pipeline = L2RRanker(main_index, title_index, preprocessor,
                        stopwords, p5_vector_ranker, fe)
```

In [ ]:
```
p5_pipeline.train(RELEVANCE_TRAIN_DATA)
p5_pipeline.model.save('p5_pipeline_l2r.txt')
```

```
Preparing: 100%|██████████| 129/129 [10:32<00:00,  4.91s/it]
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.004059 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2894
[LightGBM] [Info] Number of data points in the train set: 9604, number of used features: 124
```

## Problem 5

In [ ]:
```
p5_pipeline_info = run_relevance_tests(test_filename, p5_pipeline)
```

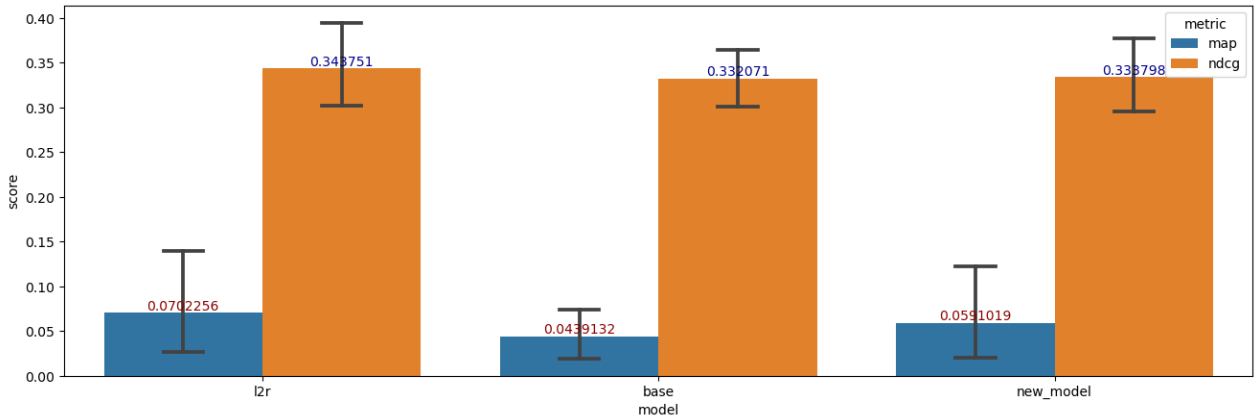
```
100%|██████████| 37/37 [04:39<00:00,  7.56s/it]
```

In [ ]:
```
data = {}
with open('pipeline_info.json', 'r') as f:
    data = json.load(f)
    data['p5_pipeline'] = p5_pipeline_info

with open('pipeline_info.json', 'w') as f:
    json.dump(data, f, indent=4)
```

In [ ]:
```
pipeline_df = pd.DataFrame(data).transpose().reset_index().rename(columns={'index': 'model'})
pipeline_df['score'] = pipeline_df[['map_scores', 'ndcg_scores']].apply(lambda x: metric_add_func(x[0], 'map') + metric_add_func(x[1
pipeline_df = pipeline_df.explode('score')
pipeline_df.drop(columns=['map', 'ndcg', 'map_scores', 'ndcg_scores'], inplace=True)
pipeline_df['metric'] = pipeline_df['score'].apply(lambda x: x[0])
pipeline_df['score'] = pipeline_df['score'].apply(lambda x: x[1])
pipeline_df['model'] = pipeline_df['model'].apply(lambda x: "new_model" if x == "p5_pipeline" else x)
plt.figure(figsize=(16, 5))
ax = sns.barplot(data=pipeline_df, x='model', y='score', hue='metric', capsize=.1, errorbar='ci')
ax.bar_label(ax.containers[0], fontsize=10, color='darkred')
ax.bar_label(ax.containers[1], fontsize=10, color='darkblue')
plt.savefig('pipeline_metrics.png', dpi=300, bbox_inches='tight')
pipeline_df.pivot_table(index='model', columns='metric', values='score', aggfunc='mean')
```

Out[ ]:

| metric | map | ndcg |
|---|---|---|
| model | | |
| base | 0.043913 | 0.332071 |
| l2r | 0.070226 | 0.343751 |
| new_model | 0.059102 | 0.333798 |




## Problem 4

In [ ]:
```
sampled_queries = [
    "What is the history and cultural importance of traditional Chinese martial arts",
    "How are countries responding to the challenges of misinformation and disinformation campaigns",
    "Analyze the role of architecture in promoting sustainability and green design",
    "How do colleges and universities ensure campus safety and security",
    "How have smartphones influenced the gaming industry and mobile gaming trends"
]

dev_df = pd.read_csv(RELEVANCE_DEV_DATA)

q4_vec_encoder = VectorRanker('sentence-transformers/msmarco-MiniLM-L12-cos-v5', encoded_docs, document_ids)
cescorer = CrossEncoderScorer(raw_text_dict)

total_biencoder_scores = []
total_cross_scores = []
total_biencoder_ranks = []
total_cross_ranks = []
```

```
for query in tqdm(sampled_queries):
    docids = dev_df[dev_df['query'] == query]['docid'].values
    biencoder_scores = q4_vec_encoder.query(query)
    biencoder_scores = {k: v for k, v in biencoder_scores}
    biencoder_scores = [biencoder_scores[docid] for docid in docids]
    biencoder_ranks = np.argsort(biencoder_scores)[::-1]
    biencoder_ranks = [v + 1 for v in biencoder_ranks]
    cross_scores = []
    for docid in docids:
        cross_score = cescorer.score(docid, query)
        cross_scores.append(cross_score)
    cross_ranks = np.argsort(cross_scores)[::-1]
    cross_ranks = [v + 1 for v in cross_ranks]

    total_biencoder_scores.extend(biencoder_scores)
    total_cross_scores.extend(cross_scores)
    total_biencoder_ranks.extend(biencoder_ranks)
    total_cross_ranks.extend(cross_ranks)
```

```
100%|██████████| 5/5 [01:00<00:00, 12.04s/it]
```

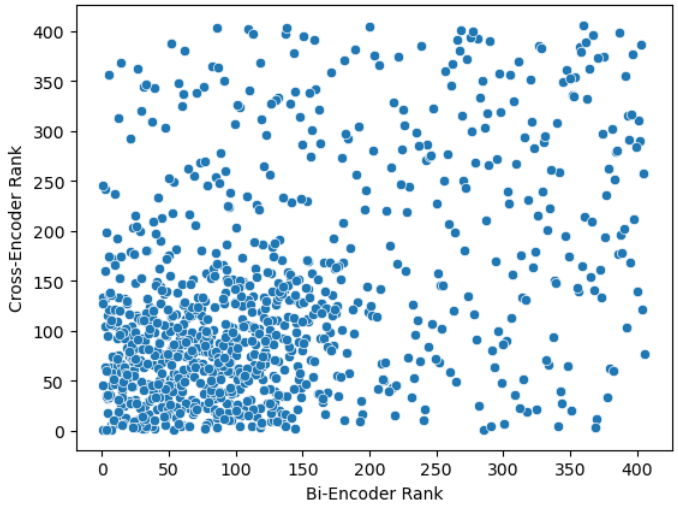
Spearman

In [ ]:
```
from scipy.stats import spearmanr
spearmanr(total_biencoder_scores, total_cross_scores)[0]
```

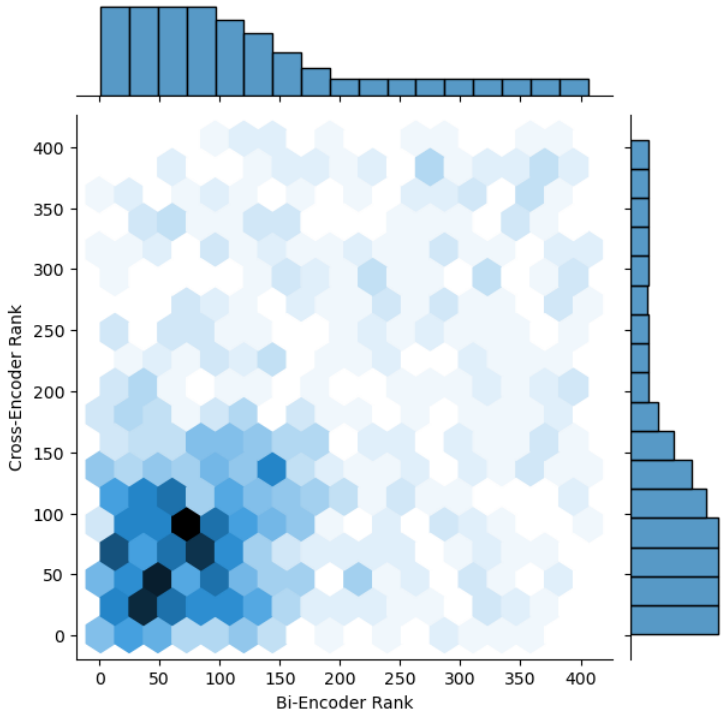
Out[ ]:
```
0.5829088970291739
```

Scatter plot

In [ ]:
```
sns.scatterplot(x=total_biencoder_ranks, y=total_cross_ranks)
plt.xlabel("Bi-Encoder Rank")
plt.ylabel("Cross-Encoder Rank")
plt.savefig("bi_cross_rank_scatter.png", dpi=300, bbox_inches='tight')
```




In [ ]:
```
sns.jointplot(x=total_biencoder_ranks, y=total_cross_ranks, kind='hex')
plt.xlabel("Bi-Encoder Rank")
plt.ylabel("Cross-Encoder Rank")
plt.savefig("bi_cross_rank_hex.png", dpi=300, bbox_inches='tight')
```




## Explorations

In [ ]:
```
import requests

def get_wiki_title(page_id:int):
    url = (
        'https://en.wikipedia.org/w/api.php'
        '?action=query'
        '&prop=info'
        '&inprop=subjectid'
        f'&pageids={page_id}'
        '&format=json')
    json_response = requests.get(url).json()
    return json_response['query']['pages'][str(page_id)]['title']
```

In [ ]:
```
bm25 = BM25(main_index)
ranker = Ranker(main_index, preprocessor, stopwords, bm25)

pipeline_1 = L2RRanker(main_index, title_index, preprocessor,
                       stopwords, ranker, fe)

pipeline_1.train(RELEVANCE_TRAIN_DATA)
pipeline_2 = p5_pipeline
```

```
Preparing: 100%|██████████| 129/129 [11:10<00:00,  5.20s/it]
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.003972 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2894
[LightGBM] [Info] Number of data points in the train set: 9604, number of used features: 124
```

In [ ]:
```
pipeline_1_info = run_relevance_tests(test_filename, pipeline_1)
```

```
100%|██████████| 37/37 [05:03<00:00,  8.21s/it]
```

In [ ]:
```
data['pipeline_1'] = pipeline_1_info
```

After this point, students may have varying answers and observations depending on their implementation and their own features. So, your mileage may vary (YMMV)

## Example Query: 'How did the Mongols conquer China?'

This query should lead to pages about the different Mongolian invasions of China (there were multiple).

In [ ]:
```
[(get_wiki_title(doc), score) for doc, score in pipeline_1.query('How did the Mongols conquer China?')[:10]]
```

Out[ ]:
```
[('Mongols', 0.1534867083343878),
 ('Mongols in China', 0.14939814740461801),
 ('Proto-Mongols', 0.13488855374070305),
 ('Mongol conquest of China', 0.09367380611897755),
 ('Foreign relations of imperial China', 0.08780048296061391),
 ('Mongol Empire', 0.07443215906076774),
 ('Möngke Khan', 0.07443215906076774),
 ('Subutai', 0.07443215906076774),
 ('Mongol invasion of Europe', 0.07443215906076774),
 ('Chinese expansionism', 0.07443215906076774)]
```

In [ ]:
```
[(get_wiki_title(doc), score) for doc, score in pipeline_2.query('How did the Mongols conquer China?')[:10]]
```

Out[ ]:
```
[('Mongols in China', 0.14939814740461801),
 ('Proto-Mongols', 0.13488855374070305),
 ('Mongol conquest of China', 0.09367380611897755),
 ('Foreign relations of imperial China', 0.08780048296061391),
 ('Mongol Empire', 0.07443215906076774),
 ('Mongol invasion of Europe', 0.07443215906076774),
 ('Chinese expansionism', 0.07443215906076774),
 ('Mongolia under Qing rule', 0.07443215906076774),
 ('Mongol invasions of Vietnam', 0.07443215906076774),
 ('Northern Yuan', 0.07443215906076774)]
```

**The first result is pretty similar**

But the difference is in what lies after maybe the second rank. You would see that the vector ranker would provide better pages overall in the top ranks of the fetched document list (in our experience the vector ranker pipeline focused more on conquests by mongols rather than details about mongols themselves).

## Example of a query where both the pipelines perform badly

**top 10 video games**

In [ ]:
```
[(get_wiki_title(doc), score) for doc, score in pipeline_1.query('top 10 video games')[:10]]
```

Out[ ]:
```
[('2013 in video games', 0.1216225696943588),
 ('2016 in video games', 0.1216225696943588),
 ('2019 in video games', 0.07510120087440691),
 ('Breakout (video game)', 0.07346076771181355),
 ('Night Driver (video game)', 0.07346076771181355),
 ('Tile-based video game', 0.07346076771181355),
 ('Video games in the United States', 0.07174534830464477),
 ('Melee (game terminology)', 0.0675875955353886),
 ('Isometric video game graphics', 0.06594716237279524),
 ('Golden age of arcade video games', 0.06508593454307408)]
```

In [ ]:
```
[(get_wiki_title(doc), score) for doc, score in pipeline_2.query('top 10 video games')[:10]]
```

Out[ ]:
```
[('2013 in video games', 0.1216225696943588),
 ('2016 in video games', 0.1216225696943588),
 ('2019 in video games', 0.07510120087440691),
 ('Video games in the United States', 0.07174534830464477),
 ('Sports video game', 0.06344550138048073),
 ('Programming game', 0.059843312521335956),
 ('Casual game', 0.05741390132008401),
 ('Video games in China', 0.05616637171283598),
 ('2010s in video games', 0.053310958400683894),
 ('Video game genre', 0.04758889800187939)]
```

Looking at the results, it doesn't seem that the search engine does very well on this query. Why might that be? Could you think of ways to handle these types of queries? What about other queries where the search engine might just not be very good?

## **Ablation Study** (What have been/can be done?)

- BM25 (hw2)
  - None: base
  - l2rranker: l2r
- l2rranker + index_augment + cross_encoder
  - BM25: pipeline_1
  - VectorRanker: pipeline_2
- vector_ranker + l2rranker
  - index: pipeline_3
  - index_augment: pipeline_4
- vector_ranker + l2rranker + index_augment
  - None: pipeline_4
  - cross_encoder: pipeline_2

In [ ]:
```
ORIGIN_INDEX = "main_index"
origin_index = Indexer.load_index(ORIGIN_INDEX)
cescorer = None
fe = L2RFeatureExtractor(origin_index, title_index, doc_category_info,
            preprocessor, stopwords, recognized_categories,
            network_features, cescorer)

base_ranker = VectorRanker('sentence-transformers/msmarco-MiniLM-L12-cos-v5', encoded_docs, document_ids)
pipeline_3 = L2RRanker(origin_index, title_index, preprocessor,
                   stopwords, base_ranker, fe)
pipeline_4 = L2RRanker(main_index, title_index, preprocessor,
                   stopwords, base_ranker, fe)
```

```
load index: 100%|██████████| 122784/122784 [00:21<00:00, 5592.87it/s]
```

In [ ]:
```
pipeline_3.train(RELEVANCE_TRAIN_DATA)
pipeline_4.train(RELEVANCE_TRAIN_DATA)
```

```
Preparing: 100%|██████████| 129/129 [00:07<00:00, 16.60it/s]
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.003295 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2639
[LightGBM] [Info] Number of data points in the train set: 9604, number of used features: 123
Preparing: 100%|██████████| 129/129 [02:34<00:00,  1.19s/it]
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.012700 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2639
[LightGBM] [Info] Number of data points in the train set: 9604, number of used features: 123
```

In [ ]:
```
pipeline_3.model.save('pipeline_3_l2r.txt')
pipeline_4.model.save('pipeline_4_l2r.txt')
pipeline_3_info = run_relevance_tests(test_filename, pipeline_3)
pipeline_4_info = run_relevance_tests(test_filename, pipeline_4)
```

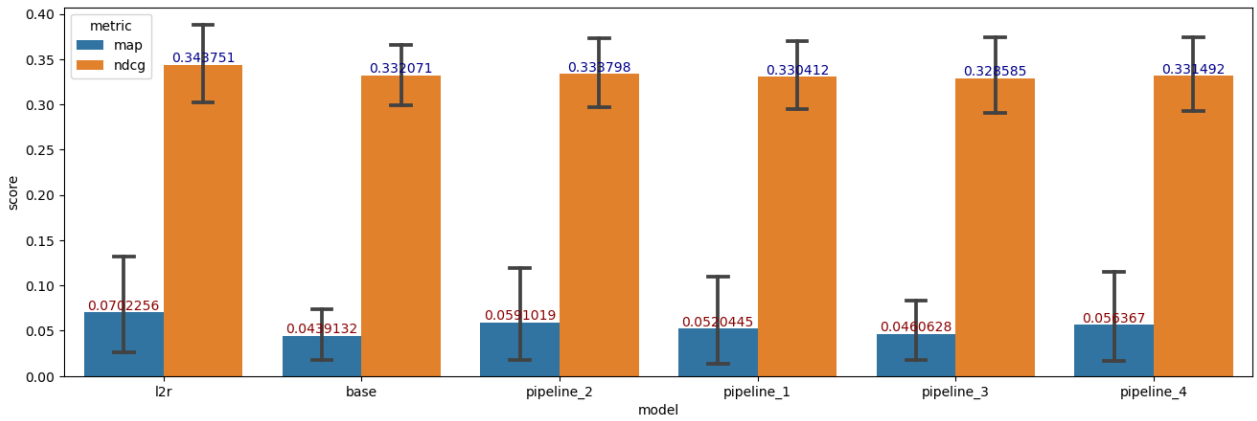
```
100%|██████████| 37/37 [01:35<00:00,  2.57s/it]
100%|██████████| 37/37 [01:22<00:00,  2.24s/it]
```

In [ ]:
```
data['pipeline_3'] = pipeline_3_info
data['pipeline_4'] = pipeline_4_info
```

In [ ]:
```
pipeline_df = pd.DataFrame(data).transpose().reset_index().rename(columns={'index': 'model'})
pipeline_df['score'] = pipeline_df[['map_scores', 'ndcg_scores']].apply(lambda x: metric_add_func(x[0], 'map') + metric_add_func(x[1
pipeline_df = pipeline_df.explode('score')
pipeline_df.drop(columns=['map', 'ndcg', 'map_scores', 'ndcg_scores'], inplace=True)
pipeline_df['metric'] = pipeline_df['score'].apply(lambda x: x[0])
pipeline_df['score'] = pipeline_df['score'].apply(lambda x: x[1])
pipeline_df['model'] = pipeline_df['model'].apply(lambda x: "pipeline_2" if x == "p5_pipeline" else x)
plt.figure(figsize=(16, 5))
ax = sns.barplot(data=pipeline_df, x='model', y='score', hue='metric', capsize=.1, errorbar='ci')
ax.bar_label(ax.containers[0], fontsize=10, color='darkred')
ax.bar_label(ax.containers[1], fontsize=10, color='darkblue')
plt.savefig('pipeline_metrics_extra.png', dpi=300, bbox_inches='tight')
pipeline_df.pivot_table(index='model', columns='metric', values='score', aggfunc='mean')
```

Out[ ]:

| metric | map | ndcg |
|---|---|---|
| model | | |
| base | 0.043913 | 0.332071 |
| l2r | 0.070226 | 0.343751 |
| pipeline_1 | 0.052045 | 0.330412 |
| pipeline_2 | 0.059102 | 0.333798 |
| pipeline_3 | 0.046063 | 0.328585 |
| pipeline_4 | 0.056367 | 0.331492 |




## **Ablation Study** (What have been/can be done?)

- BM25 (hw2)
  - None: base
  - l2rranker: l2r
- l2rranker + index_augment + cross_encoder
  - BM25: pipeline_1
  - VectorRanker: pipeline_2
- vector_ranker + l2rranker
  - index: pipeline_3
  - index_augment: pipeline_4
- vector_ranker + l2rranker + index_augment
  - None: pipeline_4
  - cross_encoder: pipeline_2

Pure VectorRanker:

- Dev MAP: 0.026905002891844994
- Dev NDCG: 0.3465734174740667
- Test MAP: 0.05954350277879689
- Test NDCG: 0.31400025741503784

In [ ]:
```
with open('pipeline_info_extra.json', 'w') as f:
    json.dump(data, f, indent=4)
```