Introduction:

In this sample project, the corpus selected is "state of the union" and the following writeup is dependent on that. Please note that the results shown in the source code may not correspond to those used in this evaluation, since they could come from different runs.

Experiments:

Experiment 1: Precision and Recall Tradeoff Analysis

Methodology:

1. Vary the number of results from 1 to 20 in increments of 1.
2. For each value, measure both the precision and recall
3. Plot a precision-recall curve.

Implementation:

```
def iterateNumResults():
    results = {}
    for n in range(1, 21):
        metrics = evaluate(n_results=n)
        results[n] = 'Number of results: ' + str(n) + '\n' + (f"Precision:
{metrics['precision_mean']:.4f} ± {metrics['precision_std']:.4f}") + '\n' +
(f"Recall: {metrics['recall_mean']:.4f} ± {metrics['recall_std']:.4f}") + '\n'

    return results
```

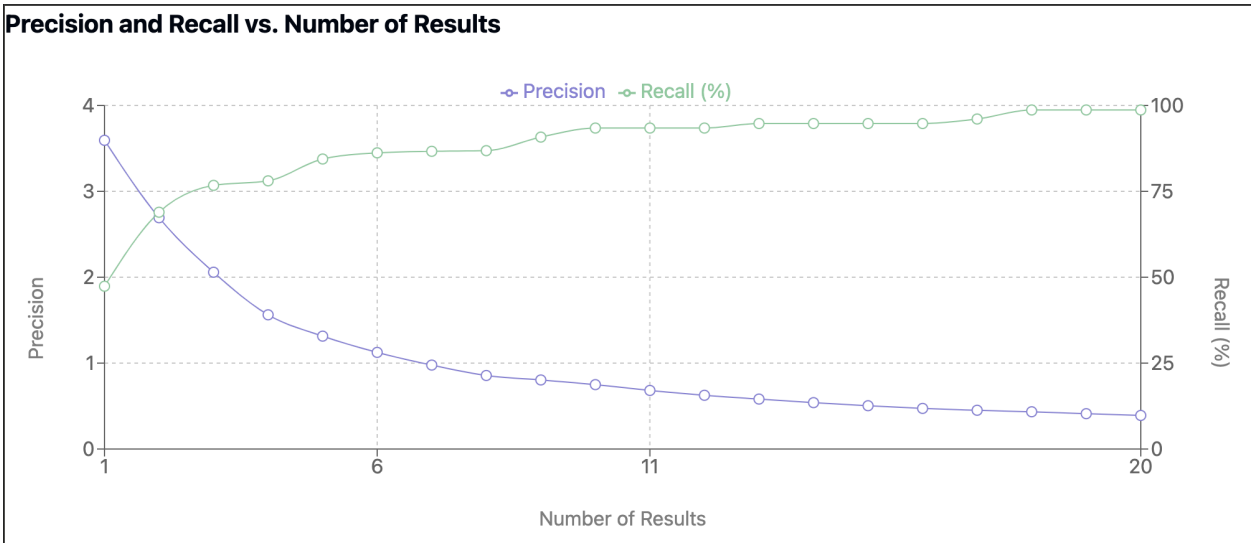
Results:

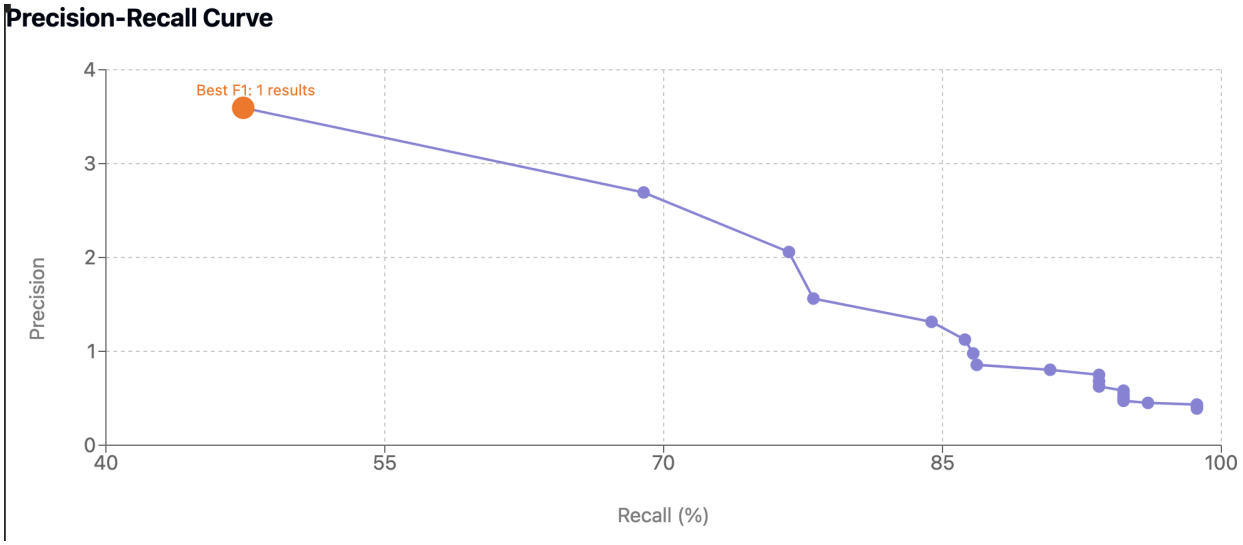
Raw-Data Table:
The following data was obtained by running the script above and tabulating the resulting output.

Number of Results:	Precision:	Recall:
1	3.5941 ± 4.6029	47.3871 ± 48.4871
2	2.6925 ± 2.5449	68.9252 ± 45.1868
3	2.0583 ± 1.9035	76.7390 ± 41.4831
4	1.5624 ± 1.4222	78.0548 ± 40.6048
5	1.3142 ± 1.0899	84.4130 ± 35.1043

6	1.1250 ± 0.9031	86.2054 ± 33.8142
7	0.9773 ± 0.7944	86.6520 ± 33.7693
8	0.8560 ± 0.6935	86.8421 ± 33.8032
9	0.8040 ± 0.6007	90.7895 ± 28.9174
10	0.7496 ± 0.5289	93.4211 ± 24.7914
11	0.6820 ± 0.4823	93.4211 ± 24.7914
12	0.6256 ± 0.4435	93.4211 ± 24.7914
13	0.5815 ± 0.4050	94.7368 ± 22.3297
14	0.5399 ± 0.3760	94.7368 ± 22.3297
15	0.5041 ± 0.3514	94.7368 ± 22.3297
16	0.4728 ± 0.3300	94.7368 ± 22.3297
17	0.4513 ± 0.3067	96.0526 ± 19.4719
18	0.4337 ± 0.2832	98.6842 ± 11.3951
19	0.4114 ± 0.2694	98.6842 ± 11.3951
20	0.3909 ± 0.2558	98.6842 ± 11.3951

Processed Data Graphs:





Analysis Insights:

- Precision decreases steadily as the number of results increases
- Recall increases rapidly at first, then plateaus around 98.7% after 18 results
- Optimal-score achieved at 1 results (highlighted in orange)
- After 10 results, recall gains are minimal while precision continues to drop

These data show typical precision-recall tradeoff behavior.

Experiment 2: Chunking Parameter Optimization

Methodology:

1. Cayley table containing chunk sizes (128, 256, 512, 1024) and overlaps (0, 50, 100, 200)
2. For each combination, evaluate with a fixed number of results (in this test 10)
3. Generate a heatmap of F1 scores (harmonic mean of precision and recall)

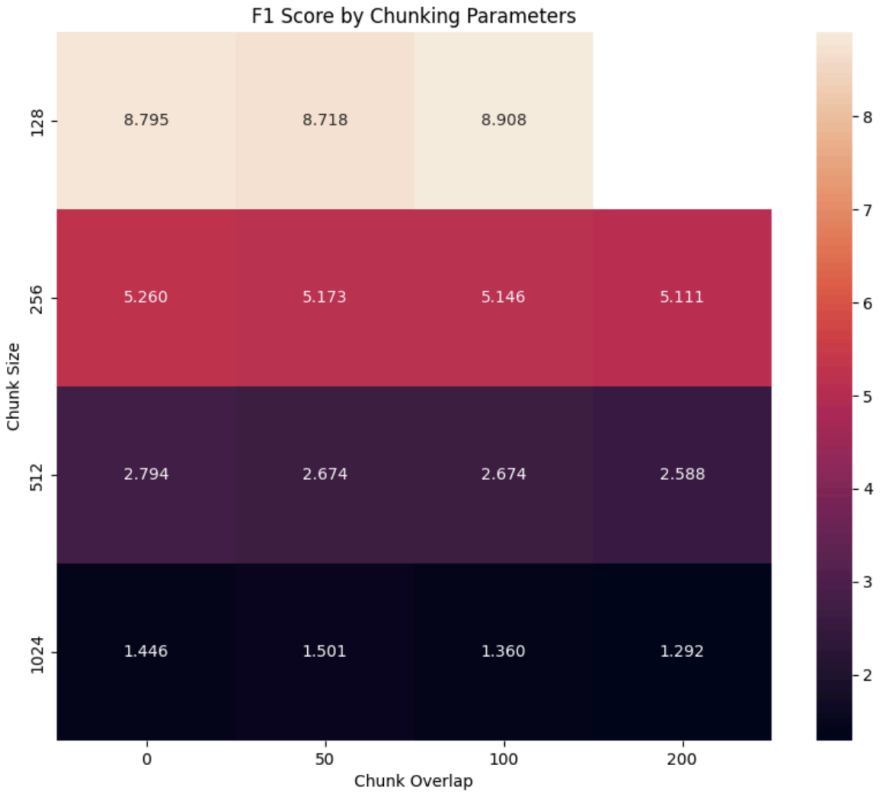
Justification of F1 score as a measure:
F1 score is the harmonic mean of precision and recall (2 × (precision × recall) ÷ (precision + recall)), which combines the two metrics into a single value which can be used for comparison (*"Introduction to Information Retrieval" by Manning, Raghavan, and Schütze*).

Implementation:

```
def optimalChunkAndOverlap():
    chunk_sizes = [128, 256, 512, 1024]
    overlaps = [0, 50, 100, 200]
```

```
grid_results = {}

for size in chunk_sizes:
    for overlap in overlaps:
        if overlap >= size:  # Skip invalid combinations
            continue

        chunker = FixedTokenChunker(chunk_size=size, chunk_overlap=overlap)
        metrics = evaluate(n_results=5, chunker=chunker)

        # Calculate F1 score
        p = metrics['precision_mean']
        r = metrics['recall_mean']
        f1 = 2 * (p * r) / (p + r) if (p + r) > 0 else 0

        grid_results[(size, overlap)] = f1
```

Processed Data Graph:



Analysis Insights:

The highest F1 scores (around 8.9) are achieved with the smallest chunk size (128 tokens). The best performer, for the tested combinations, appears to be a chunk size of 128 with an overlap of 100 tokens (F1 = 8.908). Moreover, there is a strong inverse relationship between chunk size

and performance (high F1 score). In particular, 128 tokens: ~8.7-8.9 (best), 256 tokens: ~5.1-5.3 (moderate), 512 tokens: ~2.6-2.8 (poor) and 1024 tokens: ~1.3-1.5 (worst). However, the performance drop is non-linear. Notice how the difference between 128 and 256 is dramatic (~3.5 points), whilst the difference between 512 and 1024 is comparatively smaller (~1.3 points). Additionally, chunk overlap seems to have a relatively smaller effect compared to chunk size. There's no clear pattern showing that higher overlap consistently improves performance.

Experiment 3: Embedding Model Comparison

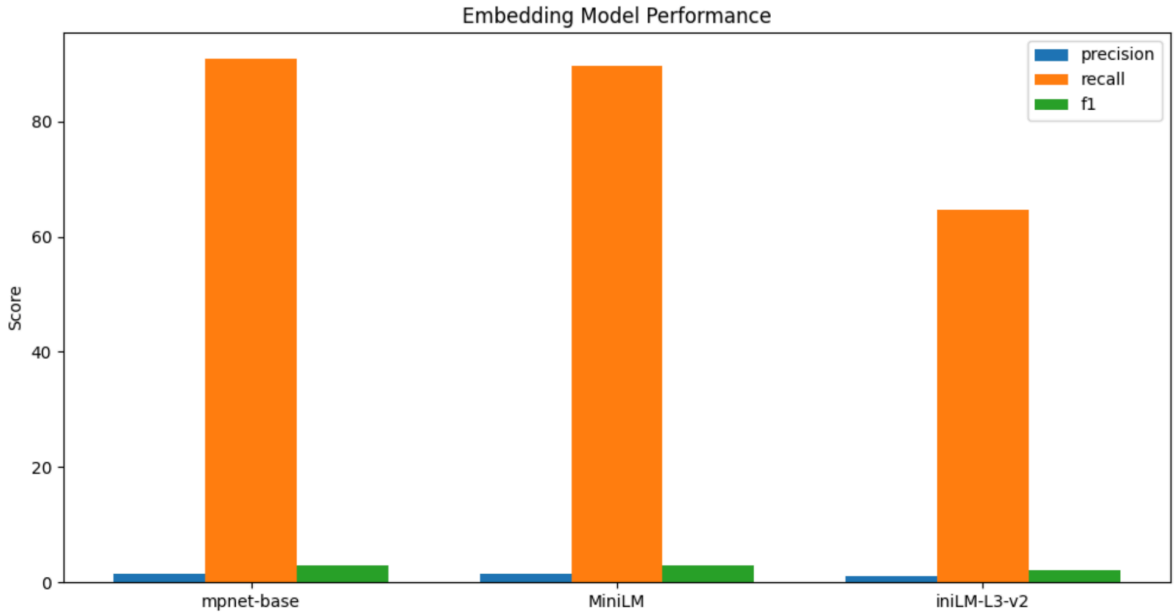
Methodology:

1. Use multiple embedding models with fixed chunking parameters
2. Compare precision and recall between the embedding models.

Implementation:

```
def compareEmbeddingFunctions():
    import chromadb.utils.embedding_functions as embedding_functions
    embedding_functions = {
        "mpnet-base":
embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-mpnet-
base-v2"),
        "MiniLM":
embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM
-L6-v2"),
        "iniLM-L3-v2":
embedding_functions.SentenceTransformerEmbeddingFunction(model_name="paraphrase
-MiniLM-L3-v2"),
    }

    model_results = {}
    for name, emb_func in embedding_functions.items():
        metrics = evaluate(
            chunker=FixedTokenChunker(
                chunk_size=512,
                chunk_overlap=50
            ),  # Use a fixed chunker configuration
            embedding_function=emb_func,
            n_results=5
        )

        model_results[name] = {
            'precision': metrics['precision_mean'],
            'recall': metrics['recall_mean'],
            'f1': 2 * (metrics['precision_mean'] * metrics['recall_mean']) /
                  (metrics['precision_mean'] + metrics['recall_mean'])
        }
```

Processed Data Graph:



Analysis Insights:

The embedding models reveal a consistent pattern of high recall (65-90%) coupled with relatively minimal precision (1-2%) across all tested architectures, suggesting that the fundamental limitation lies not in the embedding technology selection but rather in the chunking hyperparameters employed. Notice however that the iniLM-L3-v2 model does offer smaller recall, but the other metrics are close enough not to suggest it as an outlier. Despite architectural differences between mpnet-base, MiniLM, and MiniLM-L3-v2, their similar performance profiles indicate that optimizing chunk size, overlap ratios, and retrieval count would yield substantially greater improvements in the precision-recall balance than model substitution alone, which reaffirms the importance of proper chunking methods.

Extension:

In conjunction with the tests conducted, it is also possible to (1) evaluate if query modifications improve retrieval performance. For example, compare different query enhancement techniques and measure the impact on recall and precision. This would fall more under prompt engineering than chunking methodologies. One could also (2) access the impact of corpus size on metrics. For instance, create subsets of your corpus at different sizes (25%, 50%, 75%, 100%) and evaluate precision and recall on these subsets with fixed hyper-parameters.

Works Cited

Manning, Christopher D., et al. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

Smith, Brandon, and Anton Troynikov. "Evaluating Chunking Strategies for Retrieval." *Trychroma.com*, 2024, research.trychroma.com/evaluating-chunking#metrics. Accessed 28 Mar. 2025.