

# Congressional Search

By Holden Huntzinger

## Introduction

Congressional records, including the text and relevant metadata of proposed and revised bills from both chambers, are a matter of public record; the Library of Congress helpfully provides well-curated access to this information. The search engine they provide for users to query these records, though, does not perform very well. This is obvious from the first-hand experience of using it, but it's also borne out by the numbers – a BM25 baseline retrieval model significantly outperforms the existing Congress.gov search function implementation. A better search engine for Congressional bills would aid researchers and legislators in identifying relevant bills based on issues of concern; of more importance to the wider public, it would increase legislative transparency. The approach in this paper, which uses a machine learning-based 'learn to rank' approach that includes features based on not just the full text of the bills but also their titles and human-generated summaries, not only beats the BM25 baseline but performs more than 3 times better than the Congress.gov search solution measured by normalized discounted cumulative gain. Higher-quality search, especially of such well-structured text in such a clearly delineated vertical domain, is entirely possible and relatively simple to implement with modern software. In addition to this written report, relevant code and data are available at [https://github.com/hhuntz/congressional\\_search](https://github.com/hhuntz/congressional_search).

## Data

Instead of training an information retrieval model on all historical proposed legislation (much of which is available via congress.gov) or on bills proposed in the last 5 congresses (which are available with significant metadata from the new US Congress API), I have chosen to start from a curated collection of bills. Researchers Anastassia Kornilova and Vlad Eidelman published a paper in 2019 demonstrating their work on models to programmatically summarize legislation<sup>1</sup>; with it, they published their corpus of more than 22,000 bills proposed in the US Congress between 1993 and 2018<sup>2</sup>. They removed bills that were exceptionally short or long as well as those identified programmatically as near-exact duplicates of other bills in the collection. The remaining corpus represents about 20% of the total bills introduced during the studied time period. It remains outside of the scope of this project, but future work might extract the work done on this training corpus to the entirety of congressional bills available in full text. My own attempt at gathering the entire corpus of available Congressional bill text took such

---

<sup>1</sup> Kornilova, Anastassia & Eidelman, Vlad. (2019). BillSum: A Corpus for Automatic Summarization of US Legislation.

<sup>2</sup> <https://github.com/FiscalNote/BillSum>

a spectacularly long time to run (think days), and so did not provide a reasonable solution for scraping full text, summaries, and titles of congressional bills. Once a corpus of bill text existed, the new Congressional API would make it relatively simple to regularly append newly proposed legislation to the existing corpus and modify the existing index appropriately.

In order to train models and measure performance, I manually annotated the reluctance of 2,353 total responses for 20 sample queries. Many of these sample queries are based on recent responses to the long-running Gallup poll of Americans that asks "What do you think is the most important problem facing the country today?"<sup>3</sup> I have intentionally diversified the grammar and syntax (i.e., some queries are single words, others are grammatically complete questions) of the sample queries to provide a reasonable range of expected user behavior. Despite this effort, it is important to note that each of these sample queries is relatively sensical and would elicit a substantial number of very relevant responses from a perfect search system – there are many potential queries, including nonsense ones, that would elicit no relevant documents.

In order to secure ‘ground truth’ labels for model training and evaluation, I manually annotated 2,353 responses (more than 100 each) for these queries. I used three baseline models (TF-IDF, BM25, and PL2) to retrieve documents based on the 20 sample queries. For each query, I took the union of the top 100 documents returned by the models and rated each document’s relevance to the query on a scale from one to five, with the most relevant documents receiving a score of 5. I did this ranking while keeping in mind the nature of this corpus and the likely motivations for search, i.e., to discover legislation related to the modern political issue denoted by the query.

In order to compare my model’s performance to the Congress.gov search function, I searched for each of the 20 sample queries on the site (using the site’s UI to limit results to the appropriate Congresses and order by relevance) and saved the results. Congress.gov helpfully provides a way to download search results as CSV files, and I programmatically combined these results into one CSV file, which is labeled ‘congress\_results.csv’ in the Github repository linked in the introduction.

## **Related Work**

The most related work I have found is from Burgess (who happens to have gotten a PhD from UM in computer science) et al.<sup>4</sup> Their goal is to algorithmically discover semantically similar text across state legislature bills to explore the diffusion of specific bill language and draw out networks of political influence. Because of the vast

---

<sup>3</sup> Gallup, Inc. “Most Important Problem.” Gallup News, October 31, 2022. <https://news.gallup.com/poll/1675/Most-Important-Problem.aspx>.

<sup>4</sup> Burgess, Matthew, Eugenia Giraudy, Julian Katz-Samuels, Joe Walsh, Derek Willis, Lauren Haynes, and Rayid Ghani. “The Legislative Influence Detector.” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. <https://doi.org/10.1145/2939672.2939697>.

size of their corpus, they choose to first use a search function to narrow down the text options. They represent documents by taking all n-grams of length three to five and ranking them by TF-IDF. They then turn the top 25 results into a query and use Elasticsearch to grab the top 300 n-grams returned. They then run an algorithm to find the similarity between the individual documents and then uncover shared passages.

Andrew O. Ballard<sup>5</sup> used Doc2Vec to represent Congressional bills and then pairs these representations with the results of floor votes in order to generate predictions of what each member would think of every proposed bill. He then uses these predictions to gauge differences in agenda power (i.e., a majority party's ability to determine which bills are voted on and passed) between the House of Representatives and the Senate.

The fundamental goal of a search engine is to make documents more readily available based on stated need. For another way to make bills more accessible to the public, Aktolga et al. have used similarity measures and visualization techniques to highlight sections of bills that seem out of place in their context.<sup>6</sup> To find sections that are dissimilar to the rest of a given bill, the authors use human-generated keywords as queries and rank sections with BM25 scores; they then take the sections most related to the keywords and then compare, via several different dissimilarity algorithms, each other section to the combined most related sections. Particularly divergent sections are then highlighted for readers in their publicly-accessible visualization tool.

Yet another example of information retrieval involves using text representations and metadata to predict whether a bill will make it out of its committee.<sup>7</sup> For context, bills are first discussed by a pre-established, topical committee of members; the committee can then collectively choose whether to refer the vote to the larger chamber for a vote. These decisions are relatively opaque to the public and generally organized through individual conversations and back-channels. The interesting thing about this particular study is that the researchers combine both metadata (the member who proposed the bill, the committee that will review it, etc.) with representations of text. They try a few different text models, but the most relevant here involves predicting how individual members of a committee would vote on a bill (were it up for a floor vote) by casting each bill as a TF-IDF vector and comparing it to bills on which public votes have happened via cosine similarity.

Perhaps the most common computational project using Congressional bill text is developing systems for classification. Hillard et al., for example, compared the performance of several models (including a naive bayes classifier and an SVM) on

---

<sup>5</sup> Andrew O. Ballard, James M. Curry. Minority Party Capacity in Congress, *American Political Science Review* 115, no.44 (May 2021): 1388–1405.

<sup>6</sup> Aktolga, Elif, Irene Ros, Yannick Assogba, and Joan DiMicco. "Many Bills: Visualizing the Anatomy of Congressional Legislation." *Scalable Integration of Analytics and Visualization*, Papers from the 2011 AAAI Workshop, August 7, 2011.

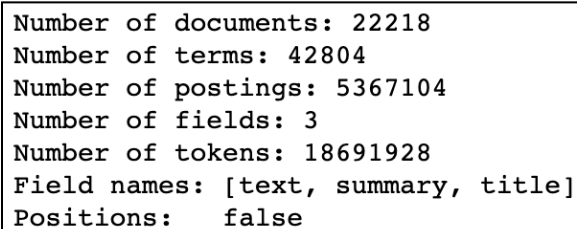
<sup>7</sup> Yano, Tae & Smith, Noah & Wilkerson, John. (2012). Textual Predictors of Bill Survival in Congressional Committees. 793-802.

human-generated bill content classifications; they land on a relatively successful ensemble model that they hope will reduce the need for manual classification in systems such as the now-defunct Legislative Indexing Vocabulary of Congress' (also now-defunct) THOMAS website.<sup>8</sup>

## Methodology

This work was done in Python using several packages to extend capability; chief among them was PyTerrier,<sup>9</sup> which offers high-level information retrieval capability. This software package was integral to this work, which relies heavily on its built-in models and integration with Scikit-learn<sup>10</sup> machine learning models.

First, the corpus of bills was indexed with PyTerrier, which makes this process surprisingly simple. The default stemmer, lemmatizer, and English-language tokenizer was used because limited testing with other PyTerrier-implemented stemmers and lemmatizers did not significantly affect final model performance. Basic information about this index is shown in the screenshot image below.

A screenshot of a terminal window showing the output of a command to display index statistics. The text is as follows:

```
Number of documents: 22218
Number of terms: 42804
Number of postings: 5367104
Number of fields: 3
Number of tokens: 18691928
Field names: [text, summary, title]
Positions: false
```

Next, basic models were implemented in Py-Terrier in order to provide a baseline against which to compare new models. Three baseline models will be used for comparison: untuned BM25, untuned TF-IDF, and the search engine implemented at Congress.gov. These models are compared with machine learning-based 'learn to rank' (LTR) models below via MAP scores, normalized discounted cumulative gain (nDCG), nDCG of the first 5 ranked results, nDCG of the first 10 ranked results, and mean response time. Each is tested with a portion of the manually annotated results for the sample queries discussed above that was reserved for this purpose. The table below

---

<sup>8</sup> Hillard, Dustin, Stephen Purpura, and John Wilkerson. "Computer-Assisted Topic Classification for Mixed-Methods Social Science Research." *Journal of Information Technology & Politics* 4, no. 4 (2008): 31–46. <https://doi.org/10.1080/19331680801975367>.

<sup>9</sup> Macdonald, Craig, Nicola Tonellotto, Sean MacAvaney, and Iadh Ounis. "Pyterrier: Declarative Experimentation in Python from BM25 to Dense Retrieval." *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021. <https://doi.org/10.1145/3459637.3482013>.

<sup>10</sup> Fabian Pedregosa, , Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research* 12, no.85 (2011): 2825-2830.

shows initial scores for these baseline models (the third row is for the Congress.gov results).

	name	map	nDCG	nDCG@5	nDCG@10
0	BR(TF_IDF)	0.959585	0.909306	0.769908	0.744234
1	BR(BM25)	0.956574	0.907808	0.772988	0.748468
2	Unnamed: 0 docno qid rank sco...	0.160404	0.308643	0.371456	0.341436

The Congress.gov results perform very poorly, even compared to these simple models. Even after careful scrutiny, it is not clear how any of this error may have been introduced by the measurement process. The way in which results were gathered is described above in the Data section. Congress.gov results were joined to the corpus on bill number, which was standardized in the corpus as a combination of the chamber, type, and number of each bill. Each sample query except for the last, ‘corporate corruption greed’, which retrieved no results from Congress.gov, retrieved a significant (median: 97) number of results that were successfully joined with the corpus. Scoring for this search function did not change significantly whether results were forcibly ranked with descending scores whereby the first 20% of results were ranked 5 for relevance, the next 20% 4, the next 20% 3, etcetera or universally scored 5 for relevance if they were returned. The latter approach was taken because it offered very marginal improvement. It was surprising that the government solution was so poor, but the first-hand experience of using it does bear out this inadequacy.

The process of feature selection for LTR modeling here was relatively limited as one of the first tested models performed surprisingly well – this is likely because of many factors, including the limited use cases expected (and so tested for) and because of the well-structured, tightly-vertical nature of this corpus. Additional hyperparameter tuning for the most successful model is discussed in the following section. It seems intuitive that searching a well-organized corpus with a relatively small set of likely use cases would make searching relatively easy; this work confirms that expectation. Perhaps someone should tell the Federal Government they are lucky to have such a simple information retrieval problem.

The LTR models were tested on information from a limited training set of 80% (16 of 20) sample queries – the other 20% were reserved for testing. These were scored via TF-IDF and subsequently re-scored by BM25. This combination resulted in the best results for untuned models (discussed below) when compared to the other possible iterations of combining TF-IDF, BM25, and PL2 scoring.

Additional features were tested in the form of TF-IDF and BM25 scores for queries based on the official bill titles and human-generated titles of each bill. Indexing these texts separately (with the same stemmer, lemmatizer, and tokenizer functions

used on the full text) meant that BM25 and TF-IDF scores could be calculated independently of the full text. The hypothesis here was that similarity between the query and the summary or the title might be significantly more important in determining relevance than similarity between the query and the full text of the bill.

## Evaluation and Results

Three untuned models, chosen based on their prevalence in similar applications, were compared: a random forest regressor, fastrank, and XGBRanker (XGBoost modified for ranking). Initial results, based on the measures explained above, can be seen in the table below.

	name	map	ndcg	ndcg_cut_10	mrt
0	BM25	0.956574	0.907808	0.748468	0.031439
1	TF-IDF	0.959585	0.909306	0.744234	0.043416
2	Congress.gov	0.160404	0.308643	0.341436	0.000000
3	Random Forest	0.960541	0.964854	0.913514	0.127402
4	Fastrank	0.955094	0.910001	0.753370	0.097183
5	XGB Ranker	0.855016	0.872096	0.644952	0.108437

After this first round of scoring, there was an obvious frontrunner. Though the learned models are all slightly slower than the scoring algorithms, only one outperformed the reigning BM25 baseline by all other measures: the random forest. It was a nice surprise to beat the existing metrics so quickly. The Congress.gov solution, of course, was quickly left in the dust.

Moving on with the Random Forest model seemed the obvious choice, but the other two were also trained for the following efforts at improving performance by selecting additional features. Indexing the title and summary of each bill separately offered very little performance gain but in every case at least doubled the response time. They also significantly increased the model's training time, which is negligible in this context and for this corpus but might matter in other deployment situations. Results can be seen in the table directly below:

	name	map	ndcg	ndcg_cut_10	mrt
0	BM25	0.956574	0.907808	0.748468	0.028336
1	TF-IDF	0.959585	0.909306	0.744234	0.023007
2	Congress.gov	0.160404	0.308643	0.341436	0.000000
3	Random Forest	0.960541	0.964854	0.913514	0.154369
4	Fastrank	0.955094	0.910001	0.753370	0.125927
5	RF Multi-index	0.959618	0.969221	0.937189	0.333866
6	Fastrank Multi-index	0.877736	0.912184	0.790205	0.286179
7	XGB Multi-index	0.848265	0.895405	0.733043	0.264605

The Random Forest model (RF) is improved significantly (by ~2.4%) when given the additional indexes; this turns out to be true whether the additional indexes are scored by BM25 or TFIDF (there's less than a 0.001 difference in final nDCG). The gain in overall nDCG is minimal, but that may be because the base model is already performing so well.

Tuning the hyperparameters of the RF slightly allows for an additional small boost in performance. The first hyperparameter to tune is the number of trees in the forest. The multi-index model identified above was trained with 100, 200, 300, 400, 500, and 600 trees and measured by nDCG, nDCG of the first 10 results, and mean response time. These changes make a noticeable difference in how long it takes to train the model (100 trees take ~4.3 seconds while 600 take ~12 seconds). Interestingly, using 200 trees successfully reduces both the model training time (wall time) to 6 seconds from 9.7 seconds as well as reducing response time to 0.268 down from 0.334 in the models above, which were compared with 400 trees. Though using fewer trees makes the model even faster to train, using 200 trees results in the highest overall nDCG (0.97) as well as the highest nDCG at the first 10 results (0.939). The RF *n\_jobs* parameter, which is the number of jobs to run in parallel, was also tuned: as expected, more parallelization results in faster response times without affecting other measures of success. It also reduces slightly (by about 15 seconds in this case) the time required to train the model.

## Discussion

It was unexpected that this relatively simple LTR implementation would perform so very well, but the biggest takeaway here is this: high-quality search, especially of well-structured text with well-defined use case scenarios, is eminently possible and LTR models can make it relatively simple to implement. The tuned random forest model not only beat the (already quite well performing) BM25 baseline, but it actually scored more than three times better than the official Congress.gov solution.

The next step in this research would be to implement this search engine across the entirety of the larger corpus of bills introduced to the US Congress and to build out a graphical user interface to make it more usable. Because this corpus will continue to evolve as new bills are developed, the index will need to grow to incorporate new terms; relative to some other search domains (such as broad web search), this growth will be limited. Additionally, I would worry about overfitting based on the relatively small set of queries on which this model was trained and tested. This problem could be solved either with more manual annotation or with the incorporation of user feedback. Even limited feedback, such as re-scoring the first result on which a user clicks as a 5 for relevance based on the related query, would quickly add robustness to this model such that it would work more effectively on a wider range of queries.

It seems from the first-hand experience of using it that this new solution, albeit to a lesser extent than the Congress.gov solution, is more likely to return irrelevant or less relevant bills (false positives) than to fail to return particularly relevant ones (false negatives). This preference toward increased recall at the expense of precision is exactly the right choice to make in the domain of bill search. The goal is to return relatively long documents that will then be read by humans who have the capacity to quickly and effectively ignore those that seem irrelevant. The cost of returned an irrelevant document is rather slim, whereas the cost of not returning a relevant document is much higher. This is because many users will not be looking for a specific document (if they were, they would likely search using known metadata, like the bill's title or number) but for a more general topic. The need for a search engine in this case is not to deliver the most relevant bill first each and every time, but first to successfully retrieve all relevant bills and second not to return too many irrelevant ones.

## **Other Things I Tried**

I was sort of disappointed that my model worked so well so quickly – there wasn't much room for progress beyond that! That said, the other idea I experimented with was using additional metadata to gain additional performance. Bill data is well structured and comes with tons of metadata, so I wrote an additional python script to query the new US Congress API to grab more information on each bill in the corpus. I got this working and I wanted to find a way to use this metadata to generate more features, but I couldn't come up with a coherent hypothesis about what would work. Why would bills introduced by a specific member of Congress be more relevant to a query when there are bills introduced by nearly 1,000 members in this limited corpus? Similarly, why would a bill be more relevant based on the specific Congressional session it was introduced in? Or how many members cosponsored it? My point is, I couldn't find any potentially-relevant metadata features because none of them has anything to do with the semantic content of the bill.



I spent a significant amount of time trying to gather my corpus. I had hoped to use the Congressional API to get the full text and metadata of each bill, but eventually decided to use a pre-pruned corpus. I got a Python script working that could grab the text and data on each bill, but it was processing bills *very* slowly. I did the math and found that it would have taken nearly 6 days running on my laptop to process all of the bills introduced over the last 5 Congresses. I had hit my computational limit, and Great Lakes couldn't solve my problem because processing power wasn't the limiting reagent – API calls can only go so fast, it seems.

### **What I Would Have Done Differently or Next**

Well, I might have picked a harder problem.

More importantly, though, I would have liked to have found a better way to include a larger and more diverse subset (if not the entirety) of available bills. I wonder if artificially limiting the corpus to avoid the shortest bills, which may have made this task more complicated, gave me an unfair upper hand. Though I am glad I don't have to because it was not the most fun part of this work, I also would have annotated more features (or, in a context with actual users and a reasonable UI, incorporated user feedback).

That said, the direct next steps would be twofold: first, to apply this work to the entirety of introduced bills, and second, to build out a reasonable user interface. At this point, the search engine works – it works better than all the baseline measures, and it works *much* better than the existing Congress.gov solution. Implementing it would be the right next challenge.