NANYANG TECHNOLOGICAL UNIVERSITY

NLP for English Grammar Pattern-based Recommendation

Revised Version (16.11.2020)

Ron Kow Kheng Hui

School of Computer Science and Engineering

2020

NANYANG TECHNOLOGICAL UNIVERSITY

MSAI Master Project MSAI/19/018

NLP for English Grammar Pattern-based Recommendation

Submitted by
Ron Kow Kheng Hui
under the supervision of
Assoc Prof Hui Siu Cheung

School of Computer Science and Engineering
2020

Abstract

Ranking is a crucial problem in information retrieval and in many tasks in natural language processing, such as question answering, in which candidate answers are ranked. In document retrieval, a search engine typically ranks documents by relevance based on textual similarity. This project addresses the problem of ranking by grammatical similarity, using machine learning methods known as Learning To Rank.

In this project, we develop a search system which recommends multiple-choice grammar questions for grammar learners. The search system is part of a grammar practice website we develop. In our system, users enter a sentence as a query. The user has the option of indicating a word or phrase in the sentence to be the focus of the query. A ranking model will match the query with grammar questions by their grammatical properties, and return a ranked list of grammar questions. If focus words are indicated, the system returns grammar questions with answers matching the focus words. The system uses Apache Solr as the search server.

We train two models using the LambdaMART algorithm. Our data consist of sentences covering five grammar topics and a curated list of terms in each topic. We use parts of speech tags and grammar production rules as model features to represent the grammatical properties of the sentences.

Our first model takes a sentence only as query and has five textual and grammatical features. Our second model takes a substring of a sentence and the focus words as query. This model is trained using 28 textual and grammatical features. The final model has 25 features.

Our two models significantly outperform the baseline model which uses only BM25 for textual similarity to rank results. In absolute terms, the first model outperforms the baseline by 46.3% on the MAP metric. The second model outperforms the baseline by 43.1% on the NDCG@5 metric and 42.3% on the NDCG@10 metric.

Acknowledgements

First and foremost, I would like to thank my project supervisor, Prof Hui Siu Cheung, for his guidance, encouragement, and kindness.

While working on this project, I learned a lot from the superb courses taught by Prof Sun Aixin and Dr Sourav Sen Gupta. I am also grateful to Dr Lek Hsiang Hui and Dr Zhao Jin from the National University of Singapore whose courses I attended in the past two years. Dr Zhao has been a great mentor. I am truly thankful for his support.

Most of all, I wish to thank my wife for her love, support, and understanding.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
BM25	Best Match 25
DCG@k	Discounted Cumulative Gain at Rank k
HTML	Hypertext Markup Language
IR	Information Retrieval
JSON	JavaScript Object Notation
LETOR	Learning To Rank Benchmark Dataset
LTR	Learning To Rank
MAP	Mean Average Precision
MART	Multiple Additive Regression Trees
ERR@k	Expected Reciprocal Rank at Rank k
NDCG@k	Normalized Discounted Cumulative Gain at Rank k
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
P@k	Precision at Rank k
POS	Part of Speech
REST	REpresentational State Transfer
RR@k	Reciprocal Rank at Rank k
SQL	Structured Query Language
SVM	Support Vector Machine
tf-idf	Term Frequency-Inverse Document Frequency
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

Chapter 1

Introduction

Ranking is a crucial problem in information retrieval. Ranking is also a component of many tasks in natural language processing, such as question answering, document summarization, machine translation, and grammar error correction [30]. For example, in grammar error correction, a ranking model is used to re-rank the candidate sentences output by the correction model [51].

In document retrieval, retrieved documents are ranked by relevance to the query. Relevance is typically based on textual similarity. This project addresses the problem of matching and ranking by grammatical similarity, using machine learning methods known as Learning To Rank.

1.1 Motivation

Consider a search system capable of matching grammar similarity. Given a sentence as query input, the ranking model in the system detects the input sentence's grammatical properties and matches them with the grammatical properties of other sentences. The system outputs a list of sentences ranked by grammatical similarity.

Our goal is to develop such a ranking model and build such a search system for English grammar learners. Given a query of only a grammar question and its correct answer, we want to develop a model that can detect the grammatical properties of the query, and implicitly, the likely grammar topic. We want to deploy the model in a grammar search system to recommend grammar questions.

It must be noted that this is quite different from a *recommender system* in the wider context. In many commercial recommender systems, a user's stored preferences and past behavior is used to recommend products (e.g., Amazon), movies (e.g., Netflix), or daily news (e.g., Yahoo News), for example. But ranking is a key component in any recommender system because users expect the

top few recommendations to be the most relevant to their needs.

In this project, we do not take into account a grammar learner's learning history. We only consider the question and answer in a query, with no other filtering options, so that we can build a model capable of detecting grammar.

1.2 Objectives

In this project we will:

- review related work in information retrieval, natural language processing, and in particular, Learning To Rank,
- develop ranking models for a grammar question search system,
- develop a grammar practice website with a grammar question search system, using the enterprise search software Apache Solr.

1.3 Organization of the Report

The rest of this report is organized into four chapters.

- Chapter 2 provides a concise overview of related work and a review of recent research. We will also describe the features of Apache Solr.
- Chapter 3 describes our methods, modelling process, and experimental results.
- Chapter 4 describes the development of the grammar practice website and the grammar question search system.
- Chapter 5 concludes the paper with an overall summary and a discussion of possible future development and enhancement.

Chapter 2

Related Work

This chapter provides an overview of past work related to the project and a review of recent research on Learning To Rank for document retrieval.

2.1 Definitions

First, we define some terms in information retrieval (IR).

Information retrieval is the process of searching for *documents* that satisfy an information need, from a large collection of documents [34]. An IR system is a computer system storing a collection of documents. A document refers to a unit of text information in the system, such as an article, a book, or a web page.

The backend of a search system is a server which contains an *inverted index*, or simply an index. The index contains a dictionary of all the terms in all documents, and maps each term to a list of document ids. This list is called a *postings list*.

Since users typically search for information from an IR system, we will call an IR system a *search system*. When a user inputs a *query* into a search system, the system outputs an ordered list of documents that are ranked by *relevance* to the query. The query could be some text entered, or selected options provided by the system to filter the search.

Essentially, relevance measures the degree of similarity between query and document. In a basic search system, relevance is based on *textual similarity*. Basically, query terms are matched with document terms and a weight is assigned to each term in the query. The sum of weights gives the *relevance score* for the query-document pair. A widely used scoring function, or *ranking function*, is BM25 [43].

Accuracy of rankings may be improved by using machine learning methods known as Learning To Rank (LTR), in which a *ranking model* is trained to rank the documents. In a search system where a ranking model is deployed, the model typically re-ranks the original top N results ranked by BM25.

In the following sections, we will describe BM25 and Learning To Rank in more detail.

2.2 BM25

In a basic search system, the relevance of a document d with respect to a query q is determined by the common terms in the query-document pair (q, d). We use a weighting function to compute the weight w_i of each term t_i in the query. If t_i is not found in d, $w_i = 0$. The relevance score of (q, d) is the sum of the weights of all query terms:

$$\text{score}(q, d) = \sum_{i=1}^{n} w_i \tag{2.1}$$

where query q contains the terms $t_1, ..., t_n$.

BM25 is a term weighting function. For a term t, BM25 is:

$$w_i = idf_{t_i} \cdot \frac{(k_1 + 1) \cdot tf_{t_i,d}}{k_1 \left(1 - b + b \cdot \frac{L_d}{L_{\text{mean}}}\right) + tf_{t_i,d}} \tag{2.2}$$

where

$tf_{t_i,d}$ is the term frequency of term t_i in document d,

idf_{t_i} is the inverse-document frequency of term t_i,

L_d is the length (i.e., the number of terms) of document d,

L_{mean} is the mean length of all documents in the collection,

k_1 and b are tuning parameters.

Combining (2.1) and (2.2) gives the function commonly called the BM25 ranking function:

$$\text{BM25}(q, d) = \sum_{i=1}^{n} idf_{t_i} \cdot \frac{(k_1 + 1) \cdot tf_{t_i,d}}{k_1 \left(1 - b + b \cdot \frac{L_d}{L_{\text{mean}}}\right) + tf_{t_i,d}}$$

The two parameters, k_1 and b, control the behavior of the function. The parameter k_1 controls the influence of the term frequency. If $k_1 = 0$, $tf_{t_i,d}$ cancels out and has no effect on the score. As k_1 increases, the influence of $tf_{t_i,d}$ increases. The parameter b penalizes longer documents, and is

used to eliminate the advantage that longer documents have over shorter documents, since longer documents have more terms and are thus likely to have higher term frequencies. In Apache Lucene [1], the default values are $k_1 = 1.2$, $b = 0.75$.

Other versions of BM25 have been proposed [49]. In Lucene, idf_{t_i} is computed as:

$$idf_{t_i} = \ln\left(\frac{N - df_{t_i} + 0.5}{df_{t_i} + 0.5} + 1\right)$$

where

N is the number of documents in the collection,

df_{t_i} is the number of documents containing the term t.

The addition of 1 to $\frac{N - df_{t_i} + 0.5}{df_{t_i} + 0.5}$ ensures that idf_{t_i} is positive for all $df_{t_i} \leq N$. BM25 is the default ranking function in Solr.

2.3 Evaluation Metrics

We describe the following evaluation metrics for ranked retrieval:

- P@k Precision at Rank k
- MAP@k Mean Average Precision at Rank k
- NDCG@k Normalized Discounted Cumulative Gain at Rank k

MAP, NDCG@3, NDCG@5, and NDCG@10 are the four most widely used metrics [48].

2.3.1 Precision at Rank *k*

Precision is the fraction of the retrieved documents that are relevant. By this definition, precision can be used to evaluate unranked retrievals.

For ranked retrieval, we define *Precision at Rank k* as the fraction of the top k documents that are relevant. For a query and an ordered list of top k documents $D_j = \{d_1, d_2, ..., d_k\}$, precision at rank k is:

$$\text{P@}k = \frac{\sum_{j=1}^{k} y_j}{k}$$

where y_j is the binary relevance label (i.e., $y_j = 0$ or $y_j = 1$) of document j.

2.3.2 Mean Average Precision

Mean Average Precision (MAP) is used to evaluate retrievals with binary relevance. First, we define Average Precision. Given a ranked list of retrieved documents, consider only the relevant documents. Each relevant document has a P@k value depending on its rank. We take the sum of the P@k values for all the relevant documents in the list. Then we compute the average over the number of relevant documents. For a *single* query, the formula can again be expressed using binary relevance labels y_j:

$$\text{Average precision} = \frac{\sum_{D_j} P@j \times y_j}{\sum_{D_j} y_j}$$

where

$D_j = \{d_1, d_2, ...\}$ is the ordered list of all retrieved documents,

$P@j$ is the precision at rank j,

y_j is the binary relevance of document j.

MAP is the mean of average precision values over *all* queries.

2.3.3 Normalized Discounted Cumulative Gain at Rank *k*

For non-binary relevance labels, we can use Normalized Discounted Cumulative Gain at Rank k (NDCG@k) [25]. First, we define Discounted Cumulative Gain at Rank k (DCG@k), which is given by:

$$\text{DCG@}k = \sum_{j=1}^{k} \frac{2^{y_j} - 1}{\log(j+1)}$$

where y_j is the relevance value of document j. Relevance values are ordinal. A larger value indicates greater relevance.

The numerator gives more weight to documents with larger relevance values y_j. The denominator penalizes documents lower in rankings (i.e., larger values of j).

NDCG@k normalizes the DCG@k value by using the DCG@k value of the *ideal ranking*, that is, the ground truth rankings, for which DCG@k is maximum. Thus, NDCG@k is given by:

$$\text{NDCG@}k = \frac{\text{DCG@}k}{\max \text{DCG@}k}$$

2.4 Learning To Rank

2.4.1 Definitions and Notation

Learning To Rank (LTR) may be defined as machine learning methods for building ranking models [30]. LTR is a supervised learning problem. The modelling methods and process are somewhat similar to conventional supervised learning for classification or regression problems. However, there are major differences.

There are many possible variations in the methods, beyond the scope of this report. The methods we describe here pertain to the methods we use for this project. Table 2.1 shows the notation we will use.

Table 2.1: Notation for Learning To Rank

Notation	**Description**
$Q_i = \{q_1, q_2, ..., q_m\}$	Set of queries
$D_j = \{d_1, d_2, ..., d_n\}$	Set of documents
(q_i, d_j)	A query-document pair
y_{ij}	Relevance label for the query-document (q_i, d_j)
$X_k = \{x_1, x_2, ..., x_p\}$	Set of features
$\mathbf{x}_{ij}$	Vector of feature values for the query-document (q_i, d_j)

2.4.2 Query-Document Relevance

Consider a query q_i and a document d_j. For each query-document pair (q_i, d_j), we judge the relevance of d_j with respect to q_i and assign a relevance label y_{ij}.

For example, we could define four ordinal relevance classes 0, 1, 2, and 3, with 0 representing no relevance and 3 representing greatest relevance. $y_{11} = 0$ means there is no relevance between query q_1 and document d_1. If $y_{12} = 3$, $y_{13} = 2$ and $y_{14} = 1$, then document d_2 is more relevant to query q_1 than document d_3, and document d_3 is more relevant to query q_1 than document d_4.

The relevance labels determine the ground truth labels. The relationship between relevance and ground truths depends on the type of learning algorithm. For example, for pointwise algorithms (explained in Section 2.4.5), the relevance labels are the ground truths labels. However, there are other ways to define the ground truth.

Hence, determining relevance correctly is very important. But it not an easy task. Using human judges to determine relevance is costly and subjective. In practice, other methods of determining

relevance depend on the nature of the ranking problem. For example, for web search, click-through data could be used [30].

2.4.3 Feature Engineering

The performance of a ranking model is very much affected by the choice of features. This is again problem-specific. For example, in web search, PageRank is widely used as a feature [30].

A feature could be dependent on both query and document, such as the BM25 score for textual similarity between query and document. It could be dependent on only the document, such as the number of words in the document. It could also be dependent on the query only, such as the sum of the *idf* values of all terms in the query.

2.4.4 LETOR Datasets

LETOR is a group of benchmark research datasets for LTR [41]. In LETOR, a document is divided into five fields: body, anchor, title, URL, and the whole document (i.e., union of body, anchor, title, and URL). For every field, the researchers computed a set of features. For example, for the document title, they computed the sum of term frequencies *tf*, the sum of inverse document frequencies *idf*, *tf-idf*, BM25, length of title, and language model measures. In total, there are 64 features in LETOR 3.0 and 46 features in LETOR 4.0. LTR algorithms require data to be in the LETOR dataset format:

```
relevance_label query_id feature1:value feature2:value... # document_id
```

where the relevance label is an integer and feature values are floating point numbers. An example with five features is shown below. There are two queries, with ids 7 and 8. Each query is paired with three documents, with ids 1, 2, and 3. The document id is appended as a comment at the end of each row.

```
2 qid:7 1:10.45931  2:4.230861 3:6.8068245 4:6.1970816 5:1.342564 # docid:1
0 qid:7 1:8.805974  2:3.730667 3:0.0       4:4.4450418 5:0.0      # docid:2
1 qid:7 1:12.005561 2:1.333564 3:4.9068347 4:0.1970812 5:6.645275 # docid:3
3 qid:8 1:0.405436  2:4.635867 3:7.8065246 4:4.1940311 5:7.675943 # docid:1
1 qid:8 1:6.705385  2:8.230765 3:0.6068545 4:0.0       5:6.690284 # docid:2
2 qid:8 1:11.605631 2:3.535866 3:4.4064647 4:4.1770616 5:0.0      # docid:3
```

2.4.5 Ranking Algorithms

A LTR algorithm, or *ranking algorithm*, takes data instances of query-document pairs as input and outputs a *ranking model*. The first notable ranking algorithm was RankSVM, proposed by Herbrich

et al. (1999) [24]. Since then, many ranking algorithms have been proposed. Tax et al. (2015) presented a benchmark comparison of 87 algorithms [48].

Most ranking algorithms can be classified into three learning approaches: pointwise approach, pairwise approach, and listwise approach [16] [33] [30]. The algorithms can also be classified into their underlying algorithms, such as support vector machines, boosted decision trees, and neural networks. Table 2.2 shows a sample of ranking algorithms classified into learning approaches and underlying algorithms.

Table 2.2: Sample of Learning To Rank algorithms

	SVM	**Boosted Trees**	**Neural Networks**
Pointwise	OC SVM (2003) [45]	McRank (2008) [31]	
Pairwise	RankSVM (1999) [24]	LambdaMART (2010) [52] RankBoost (2003) [19]	LambdaRank (2007) [14] RankNet (2005) [13]
Listwise	SVM Map (2007) [54]	AdaRank (2007) [53]	ListNet (2007) [16]

We now describe very briefly the differences between the three learning approaches in terms of the model input and the ground truths the model learns.

The pointwise approach maps the feature vector $\mathbf{x}_{ij}$ to the relevance label y_{ij} for one query-document pair (q_i, d_j). The relevance labels are the ground truths. If y_{ij} is a class label, then it is a classification problem. If y_{ij} is a real number, then it is a regression problem. In other words, the model takes the feature vector as input and learns to predict the relevance value. We then rank the documents by the predicted relevance.

In the pairwise approach, the model takes the feature vectors of two query-document pairs for the same query as input, such as (q_1, d_1) and (q_1, d_2), and learns to predict which of the two documents is the more relevant one. We then rank the documents according to the pairwise rankings. In practice, if relevance labels are ordinal classes 0, 1, and 2, the pairwise algorithm converts the labels into pairwise ground truths. For example, given the following dataset:

```
2 qid:7 1:10.45931  2:4.230861 3:6.8068245 4:6.1970816 5:1.342564 # docid:1
0 qid:7 1:8.805974  2:3.730667 3:0.0       4:4.4450418 5:0.0      # docid:2
1 qid:7 1:12.005561 2:1.333564 3:4.9068347 4:0.1970812 5:6.645275 # docid:3
```

The algorithm generates the following pairwise ground truths:

```
docid:1 > docid:2
docid:1 > docid:3
docid:3 > docid:2
```

In the listwise approach, the model takes the entire group of documents for the same query as input. Some listwise algorithms learn to predict the relevance values of each query-document pair in the group, which is similar to the pointwise approach. Other listwise algorithms learn to predict the rankings of documents in the group [33].

Among the three approaches, pairwise and listwise approaches tend to perform better than pointwise approaches [30]. The pointwise approach is somewhat flawed because the model does not learn to rank one document relative to another.

2.4.6 LambdaMART

LambdaMART [52] replaces the neural network used in the algorithm LambdaRank [14] (which is based on RankNet [13]) with the boosted regression tree MART (Multiple Additive Regression Trees) [20].

LambdaRank can be formulated as a pairwise or listwise algorithm [30]. Instead of defining a loss function, LambdaRank only defines the gradient of the loss. This gradient function is called the Lambda function. The parameters of the neural network are updated by gradient descent. LambdaRank directly optimizes the NDCG metric. The mathematical details of LambdaRank, MART, and LambdaMART are presented in the review paper by Burges (2010) [15].

2.5 Grammar Representation

2.5.1 Parts of Speech

In linguistics, words are classified into parts of speech (POS). A word's part of speech depends on its definition and also its context in the sentence containing the word. For example, consider these two sentences tagged by the NLTK POS tagger [8]:

```
I need a break from work.      PRP VBP DT NN IN NN
I need to break the ice.       PRP VBP TO VB DT NN
```

The word "break" is a noun (`NN`) in the first sentence and a verb (`VB`) in the second sentence. NLTK uses the 36 POS tags from the Penn Treebank project [35]. Table 2.3 shows the list of 36 POS tags.

Table 2.3: Penn Treebank parts of speech tags

	Tag	Description	Examples
1	CC	conjunction, coordinating	and, or, whether, versus, &
2	CD	numeral, cardinal	two, 2, forty-two, one-tenth, 0.5
3	DT	determiner	the, an, another, any, both
4	EX	existential there	there
5	FW	foreign word	oui, hund
6	IN	preposition or conjunction, subordinating	on, into, by, among
7	JJ	adjective or numeral, ordinal	first, third, ill-mannered
8	JJR	adjective, comparative	braver, calmer, cheaper
9	JJS	adjective, superlative	bravest, calmest, cheapest
10	LS	list item marker	a, b, SP-44001
11	MD	modal auxiliary	can, cannot, could, might, must
12	NN	noun, common, singular or mass	cabbage, casino, afghan, humor
13	NNP	noun, proper, singular	Motown, Liverpool, John
14	NNPS	noun, proper, plural	Americans, Andes
15	NNS	noun, common, plural	undergraduates, products
16	PDT	pre-determiner	all, both, quite, such
17	POS	genitive marker	', 's
18	PRP	pronoun, personal	him, her, herself, it, them
19	PRP$	pronoun, possessive	his, mine, my, our, ours, their
20	RB	adverb	madly, occasionally, swiftly
21	RBR	adverb, comparative	further, gloomier, greater, harder
22	RBS	adverb, superlative	best, biggest, farthest, first, worst
23	RP	particle	about, across, along, before, behind
24	SYM	symbol	%, &,), *, U.S.
25	TO	"to" as preposition or infinitive marker	to
26	UH	interjection	Gosh, Wow, Hey, Oops
27	VB	verb, base form	ask, assume, begin
28	VBD	verb, past tense	pleaded, halted, aimed, wore
29	VBG	verb, present participle or gerund	judging, wincing
30	VBN	verb, past participle	used, experimented, imitated
31	VBP	verb, present tense, not 3rd person singular	appear, twist, terminate
32	VBZ	verb, present tense, 3rd person singular	mixes, slaps, speaks
33	WDT	WH-determiner	that, what, whatever, which
34	WP	WH-pronoun	that, what, whatever, which, whom
35	WP$	WH-pronoun, possessive	whose
36	WRB	WH-adverb	how, however, whenever, where, whereby

2.5.2 Constituency Grammar and Production Rules

A sentence can be broken up into phrases, or more formally, into *constituents*. A constituent is a group of words behaving as a single unit [26]. For example, consider the words "on 1st September" in the following sentences:

```
On 1st September, I will be flying to London.
I will be flying to London on 1st September.
I will be flying on 1st September to London.
```

We have moved "on 1st September" within a sentence to obtain another grammatically correct sentence with the same meaning. Thus, "on 1st September" is a constituent of each sentence. We see that "I will be flying" and "to London" are also constituents. If we separate the words within "on 1st September" or "I will be flying", we would not be able to form a grammatically correct sentence. For example:

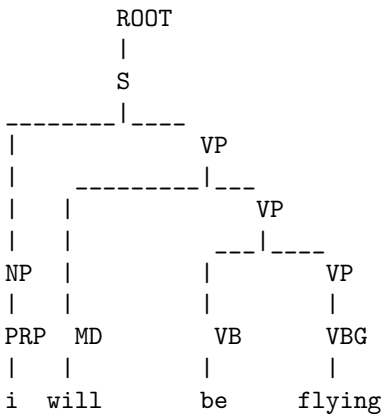
```
On 1st, I will be flying September to London.
On 1st September, I will be to London flying.
```

To model the constituency structure of a sentence, we use the method known as *context-free grammar* (CFG). CFG consists of a set of rules, called *production rules*, or simply productions. For example, the structure of "I will be flying" can be expressed using these productions:

```
S -> NP VP
VP -> MD VP
VP -> VB VP
NP -> PRP
VP -> VBG
PRP -> 'I'
MD -> 'will'
VB -> 'be'
VBG -> 'flying'
```

`S -> NP VP` expresses that a sentence (`S`) can be composed of a noun phrase (`NP`) followed by a verb phrase (`VP`). `VP -> MD VP` expresses that a verb phrase can be composed of a modal auxiliary (`MD`) followed by another verb phrase. `VP -> VB VP` expresses that a verb phrase can also be composed of a base form of a verb (`VB`) followed by another verb phrase. `NP -> PRP` and `VP -> VBG` express that a noun can be a pronoun (`PRP`), and a verb can be in present participle form (`VBG`). The last four rules map the part of speech to each word in the sentence.

When we connect these productions, we form the complete sentence "I will be flying". The entire structure is best visualized using a *constituency parse tree*:



2.6 Apache Solr

Apache Solr [2] is an open source enterprise search software built on Apache Lucene [1], the Java search library. It is one of the most widely used enterprise search software. In this section, we give a brief overview of the components of Solr used in this project.

2.6.1 Definitions

Solr stores data as *documents*, and each document is composed of *fields*. Every field has its own properties, such as the *field type*, which is the data type of the field.

Documents and fields are analogous to records and attributes (i.e., columns) in a database table. But a field in Solr can be defined to contain many different attributes. For instance, in our system, we define a field in Solr called "text" in which we put every attribute (question, answer, wrong answer choices, topic) in our questionbank table in the database. This allows Solr to quickly search for query matches within one field.

Whenever we add new data to Solr, we specify the field name, so that Solr will know how to index the data. When we modify existing data in Solr, we will need to tell Solr to re-index the data. All fields are defined in a *schema*, which is a XML file in Solr.

2.6.2 Schema and Configuration

There are two XML files in Solr for which we may need to modify: the schema file `schema.xml` and the configuration file `solrconfig.xml`.

Solr stores information about all fields and field types in `schema.xml`. For example, we define the field `ans` with the field type `text_general`, to store answers of grammar questions:

```
<field name="ans" type="text_general"/>
```

`text_general` is one of many field types Solr has defined in the schema:

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100"
multiValued="true">
   <analyzer type="index">
       <tokenizer class="solr.StandardTokenizerFactory"/>
       <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
       <filter class="solr.LowerCaseFilterFactory"/>
   </analyzer>
   <analyzer type="query">
       <tokenizer class="solr.StandardTokenizerFactory"/>
       <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
       <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
       format="solr" ignoreCase="false" expand="true"
       tokenizerFactory="solr.WhitespaceTokenizerFactory"/>
       <filter class="solr.LowerCaseFilterFactory"/>
   </analyzer>
</fieldType>
```

The file `solrconfig.xml` is generally left untouched, but it needs to be appended with information when additional functionalities are required. To enable LTR in Solr, we add information for the LTR plugin to this file.

2.6.3 Solr Administrative Interface

Solr is a web application. It runs as a standalone server and provides an administrative user interface (Solr Admin UI) on the web. We can access the Solr admin UI at http://localhost:8983/solr during development, on default port 8983.

Through the Solr Admin UI, we can view logs, information about our data, and details of our Solr configuration. We can view files such as the schema and configuration files, and JSON files in which we define LTR features and models. We can also run queries, re-index data and delete data. Figure 2.1 shows a screenshot of the Solr admin UI for our system.

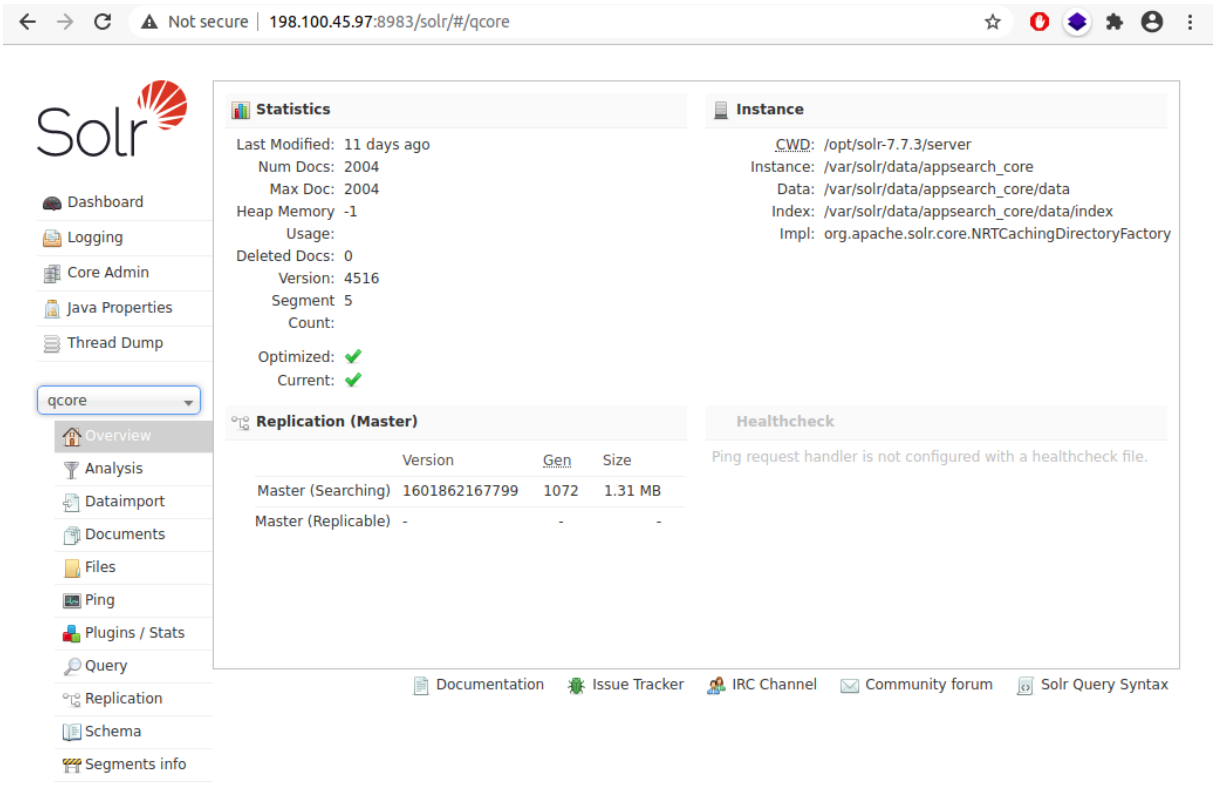


Figure 2.1: Screenshot of the Solr administrative interface

2.6.4 REST API

While Solr provides many different APIs, including a Java API which does not require a HTTP connection, its REST (REpresentational State Transfer) API is the easiest way to communicate with Solr. We can send data to the Solr server using HTTP methods such as GET, POST and PUT, with payloads formatted in JSON. For example, we can format our feature and model definitions in JSON and send it to Solr using a POST request. We can retrieve data from Solr using GET requests.

2.6.5 Query Parsers

Every search query to Solr is processed by a request handler which calls a *query parser*. Solr provides many different query parsers. When we code a query, we specify the query parser and write the code in that parser's syntax. A query contains the query string along with parameters to filter the search. For example, we can specify in the query which field to search. The query parser

takes in the terms in the query string, and the parameters, and processes the query.

The three main query parsers are the Standard query parser (also known as the Lucene query parser), the DisMax query parser, and the Extended DisMax query parser. In addition, there are more than twenty other query parsers, including the LTR query parser which we use in our system. Unfortunately, every query parser has its own peculiar syntax, so unless there are special needs, it is easiest to stick to one of the three main parsers.

Among the three main query parsers, the Standard query parser's syntax enables greater precision in searches. The DisMax query parser is designed to provide a user friendly experience to searching, similar to performing a Google search. It is more tolerant of syntax errors. The Extended DisMax query parser is an improved version of the DisMax query parser.

2.6.6 Learning To Rank

Solr supports LTR. In fact, Solr makes it easier to implement LTR because it simplifies the task of data preparation by computing feature values for the datasets. In Solr documentation, this is called feature extraction. We define a list of features and Solr extracts feature *values* based on the feature specifications.

Using the extracted feature values, we create our training, validation and testing datasets. After training and selecting the best model, we upload the model to Solr. Only some quick configurations are needed for Solr to be LTR-ready. This process is explained in detail in Section 3.4.

2.6.7 Re-ranking

In search software with LTR support, ranking models are typically used to *re-rank* queries. This means that the search system first ranks the results by a default ranking function before the ranking model is used to improve the original rankings.

In Solr, the default ranking function for textual matching is BM25. Solr first computes the BM25 scores of all documents and ranks them. After that, the ranking model re-ranks the top N results, where N is a pre-defined parameter.

Any document outside the top N will retain its original ranking and will not move up in rank even if it is more relevant to the query than any of the top N documents. This seems to suggest that N should be set as large as possible. But re-ranking a large set of documents can be computationally costly. In Solr, the default value of N is 200.

2.7 Recent Research

Previous work similar to this project is very limited. This project is in part inspired by the work of Fang et al. (2018) [18]. The researchers built a similar grammar question recommender system using a SVM model trained on textual, grammatical, and conceptual features.

Other research on ranking covers many areas in the intersection of IR and NLP. There is much research interest in deep neural models for ranking. Since 2014, the term "deep ranking" has been used in literature. The emphatic success of deep learning methods in NLP, and the release of the TF-Ranking TensorFlow library in 2018, [40] has no doubt helped to accelerate research in this area.

For the purpose of future work on this project, we did a literature search of deep ranking models and other research related to our work. In Table 2.4, we present a list of selected research from 2017 onwards.

Table 2.4: Selected research on neural ranking models since 2017

Year	Paper Title	Reference
2020	Deep Attentive Ranking Networks for Learning to Order Sentences	Kumar et al. [29]
2020	Listwise Learning To Rank with Deep Q-networks	Sharma [44]
2020	DeText: A Deep Text Ranking Framework with BERT	Guo et al. [23]
2020	Ranking Multiple Choice Question Distractors Using Semantically Informed Neural Networks	Sinha et al. [46]
2019	Neural Learning To Rank Using TensorFlow Ranking: A Hands-On Tutorial	Pasumarthi et al. [39]
2019	Listwise Neural Ranking Models	Rahimi et al. [42]
2019	DeepRank: Adapting Neural Tensor Networks for Ranking the Recommendations	Kabil et al. [27]
2019	Deep Generative Ranking for Personalized Recommendation	Liu et al. [32]
2019	A Deep Look into Neural Ranking Models for Information Retrieval	Guo et al. [22]
2019	RankQA: Neural Question Answering with Answer Re-Ranking	Kratzwald et al. [28]
2018	A New Deep Neural Network Based Learning To Rank Method for Information Retrieval	Fu et al. [21]
2018	Deep Neural Network for Learning To Rank Query-Text Pairs	Song et al. [47]

Year	Paper Title	Reference
2018	Learning a Deep Listwise Context Model for Ranking Refinement	Ai et al. [12]
2018	Deep Query Ranking for Question Answering Over Knowledge Bases	Zafar et al. [55]
2018	Deep Neural Network-Based Models for Ranking Question-Answering Pairs in Community Question Answering Systems	Nguyen et al. [37]
2018	Who's Better? Who's Best? Pairwise Deep Ranking for Skill Determination	Doughty et al. [17]
2018	Deep Relevance Ranking Using Enhanced Document-Query Interactions	McDonald et al. [36]
2018	Neural Ranking Models with Multiple Document Fields	Zamani et al. [56]
2017	DeepRank: A New Deep Architecture for Relevance Ranking in Information Retrieval	Pang et al. [38]
2017	An Attention-Based Deep Net for Learning To Rank	Wang et al. [50]

2.8 Summary

In this chapter, we provided a concise overview of BM25, the evaluation metrics for ranking, and LTR. We explained how grammar structure can be represented using parts of speech tags and production rules. We gave an overview of the components of Solr used in this project. Lastly, we provided a review of recent research on LTR.

Chapter 3

Development of Ranking Models

This chapter describes the process of developing our ranking models.

3.1 Modelling Process

The entire process can be broken down into these steps:

1. Raw data preparation: Create sentences (questions and answers)
2. Raw data preparation: Split raw data into queries and documents
3. Feature engineering: Define features to model grammatical similarity
4. Feature engineering: Define functions to score similarity
5. Feature extraction: Create query and document feature datasets from the raw data
6. Upload document feature data to Solr
7. Extract feature values in Solr for query-document pairs
8. Dataset preparation: Define criteria to measure query-document relevance
9. Dataset preparation: Create training, validation and testing datasets in LETOR format
10. Model training and testing: Experiment with algorithms and features
11. Upload selected models to Solr
12. Run queries in Solr to see actual results

Figure 3.1 illustrates this process.

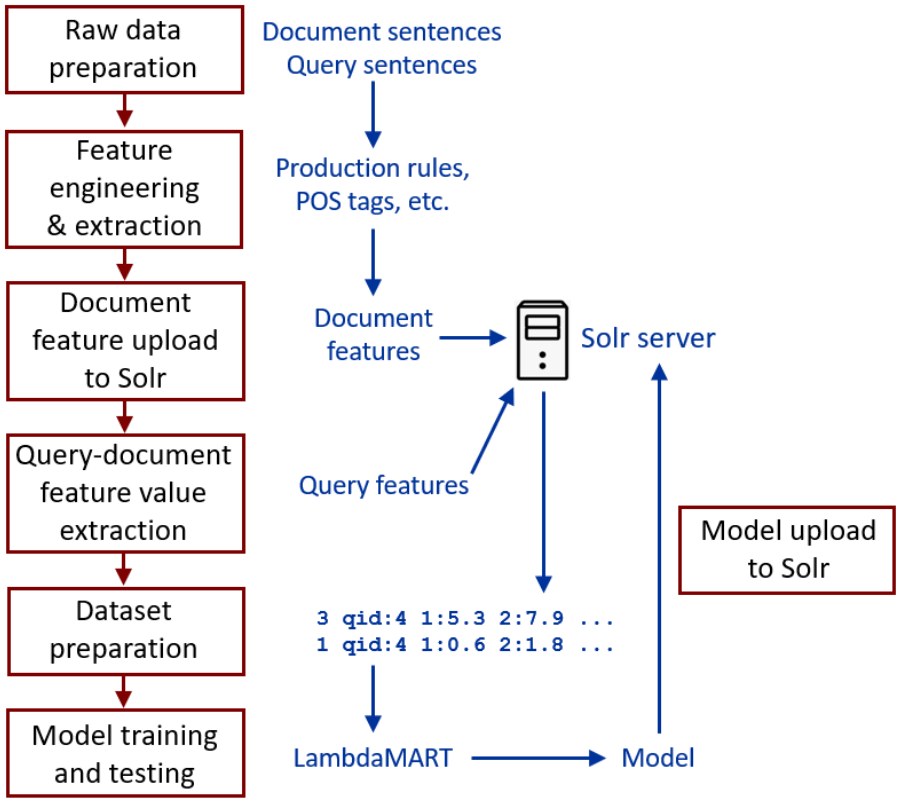


Figure 3.1: Modelling process

3.2 Raw Data

The creation and *curation* of raw data is one of the most important parts of the process. The quality of the data determines how well a model learns. The scope of English grammar is very wide. Thus we need to decide which grammar topic to include in our data, and select the set of terms in each topic. We describe the data preparation process in this section.

3.2.1 Definitions

The raw data contains 1154 data instances. A data instance has a question, the correct answer, up to three wrong answer choices, and the grammar topic. The question is a sentence with a word or phrase missing, to be filled with the correct answer.

An example of a data instance is shown below. In the database, an asterisk * in the question indicates the missing answer.

Question: `I will be in Tokyo * two weeks.`

Answer: `for`

Choice 1: `at`

Choice 2: `to`

Choice 3: `on`

Topic: `prepositions`

Most questions are one sentence long but a small number are two sentences long. However, the answer is always contained within a single sentence. In the following examples, the answer is underlined:

One sentence : `The Olympics team is aiming for a gold medal next year.`

Two sentences: `We are going to the movies. Do you want to come along?`

In the modelling process, the wrong answer choices (i.e., choice 1, choice 2 and choice 3) are not used. We extract features only from the question and the answer. The topic is not a model feature, a modelling decision which we will explain in Section 3.3.6. But the topic is required to compute the relevance of a document given a query when we prepare the modelling datasets.

Regardless of whether a question is one sentence or two sentences long, we shall call a question and correct answer data instance a *sentence*. We will use the term *grammar question* to refer to a data instance consisting of the question, the correct answer, and the wrong answer choices.

Among the four major grammar classes [4] – nouns, verbs, adverbs, adjectives – there are many sub-classes, such as uncountable nouns and phrasal verbs. Some sub-classes such as pronouns can be further classified into different classes. We will call any class or sub-class a *topic*. We will call a word (e.g., the preposition "about") or phrase (e.g., the preposition "up to") a *term*.

3.2.2 Creation of Sentences

The raw sentences are adapted and modified from examples in many sources, most notably the online Cambridge dictionary [4]. *Syntax* refers to the way words are arranged together. To enable our models to learn grammar, we have created sentences that are similar in sentence syntactic structure but different in words. For example, for the preposition "by":

`I go to school (by) bus.`

`He travels to work (by) train.`

Many sentences in the data are repeated. The difference is just in the answer. For example:

`I (am doing) my work now.` (verb tenses)

`I am doing (my) work now.` (pronouns)

3.2.3 Grammar Topics and Terms

Every natural language has a finite number of words. Every topic has a finite number of terms, though the number of terms can be very large in a major topic. Some terms can be classified into multiple topics. For example, "after" can be used as a preposition or a conjunction. Among commonly used nouns, verbs and adjectives, there are many terms that belong to multiple topics. For example, "increase" is used as a noun or a verb, and "blind" can be an noun or an adjective. Clearly, topic labelling in English grammar is not a trivial task.

Since we have a limited number of raw sentences, choices need to be made with regards to the topics and the terms in each chosen topic. Although the models learn from a limited number of topics and terms, the goal is to develop models that will generalize to detect the grammatical similarity of any two sentences. We have chosen the following five topics because they are major sub-classes of nouns and verbs and they form the basic building blocks of sentence construction.

- Prepositions
- Conjunctions
- Phrasal verbs
- Verb tenses
- Pronouns

It is virtually impossible to form any sentence without using at least one term from any of these five topics. The following example contains words from all five topics:

`I could get by with a small budget last time but times have changed.`

The sentence contains the preposition "with", the conjunction "but", the phrasal verb "get by", the pronoun "I", and the present perfect verb tense "have changed".

Among the five topics, the first four are somewhat inter-related, in the sense that there are common words among the four topics. Some words (e.g., after, since) can be used as both prepositions and conjunctions. Phrasal verbs and verb tenses (e.g., call for, will call, has called) contain common verbs. Phrasal verbs are formed by joining a verb and a preposition. Pronouns is the only topic whose terms are not found in the other four topics.

Considering the large number of terms in each topic, it is important to select terms that are commonly used so that the models can generalize well. Again, due to the limited data size, we can only select a small subset of terms among the commonly used terms. For instance, there are more than one hundred commonly used prepositions and hundreds of commonly used phrasal verbs. Our data contain only 30 prepositions and 42 phrasal verbs. Table 3.1 shows the statistics of the raw data.

Table 3.1: Raw data statistics

Topic	**Number of terms/tenses**	**Number of sentences**
Prepositions	30 terms	284
Conjunctions	16 terms	140
Phrasal verbs	42 terms	294
Verb tenses	10 tenses	240
Pronouns	28 terms	196

3.2.4 Prepositions

Prepositions show relationships between people, places or things. They often show relationships in space (e.g., she stood *behind* me) or time (e.g., the day *before* yesterday). We have selected 30 prepositions, as shown in Table 3.2.

Table 3.2: List of prepositions

Single-word terms		
about	across	after
at	against	among
before	behind	beyond
during	for	from
in	into	of
on	onto	over
off	through	with
throughout	to	by
under	within	without
Multi-word terms		
according to	on behalf of	up to

3.2.5 Conjunctions

Conjunctions connect words, phrases and clauses (groups of words containing a verb and a subject) in a sentence. The distinction between a conjunction and a preposition is subtle, and some words can be used as a conjunction and as a preposition. Consider the word "after":

`She became more independent (after) her mum passed away.` (conjunction)

`Her mum passed away (after) Christmas last year.` (preposition)

We have selected 16 conjunctions, as shown in Table 3.3.

Table 3.3: List of conjunctions

Terms			
after	because	before	but
if	once	yet	since
so	or	unless	until
when	whenever	whether	while

3.2.6 Phrasal Verbs

Phrasal verbs are phrases in which the first word is a verb. The verb is usually followed by one or more prepositions. We have selected 42 phrasal verbs among which multiple terms share the same verb (e.g., *break* down, *break* into), as shown in Table 3.4.

Table 3.4: List of phrasal verbs

Terms			
break down	break into	break out	break up
come about	come along	come across	
come down to	come off	come up	
call for	call off	call on	
go back on	go off	go on	
go through	go with		
get along	get by	get over	get through
give in	give up		
look after	look back	look down	look forward
look into	look up	look over	
let down	let off	let up	
put across	put forward	put off	
turn down	turn in	turn out	
take after	take off		

3.2.7 Verb Tenses

Of the 12 major verb tenses, we have selected all eight present and past tenses but only two of the four future tenses. Table 3.5 shows the 10 types of verb tenses.

Table 3.5: List of verb tenses

Tense	**Example**
Simple present	She plays the piano every morning.
Present continuous	She is playing the piano now.
Present perfect	She has played this piece many times.
Present perfect continuous	She has been playing the piano since morning.
Simple past	He played basketball yesterday.
Past continuous	He was playing basketball when it started to rain.
Past perfect	He had played for his country only once before.
Past perfect continuous	He had been playing basketball for an hour before it started to rain.
Simple future	He will play in the first team.
Future continuous	He will be playing in the next game.

3.2.8 Pronouns

A pronoun is a word that refers to a particular noun. Pronouns can be classified into six classes: personal (e.g., you, she), possessive (e.g., your, hers), reflexive (e.g., yourself, herself), indefinite (e.g., someone, anyone, nothing, everywhere), relative (e.g., ...which..., ...whom...), and interrogative (e.g., Which..., Whom...).

We have selected only four classes of pronouns. Among personal pronouns, “it” is excluded. Among reflexive pronouns, the plural form of “yourself” (yourselves) is excluded. Table 3.6 shows the 28 selected pronouns.

Table 3.6: List of pronouns

Pronoun type	**Terms**					
Personal (subject)	I	you	he	she	we	they
Personal (object)	me	you	him	her	us	them
Possessive	my mine	your yours	his his	her hers	our ours	their theirs
Reflexive	myself	youself	himself	herself	ourselves	themselves

3.2.9 Query and Document Datasets

We split the 1154 raw sentences into three distinct datasets: 850 are selected for the document set, 152 for the training query set, and the remaining 152 for the combined validation and testing query set. Queries for validation and testing will be randomly drawn from the combined validation and testing set. There will be 75 queries for validation and 77 queries for testing.

The 152 sentences in each query dataset are manually selected to ensure that there is at least one sentence for every term in the five topics. The distribution of the number of sentences in each dataset are shown in Tables 3.7 and 3.8.

Table 3.7: Distribution of the number of sentences in each dataset

	Dataset		
Topic	**Document**	**Query (train)**	**Query (validation, test)**
Prepositions	200	42	42
Conjunctions	100	20	20
Phrasal verbs	210	42	42
Verb tenses	200	20	20
Pronouns	140	28	28
Total	**850**	**152**	**152**

Table 3.8: Distribution of the number of sentences per term in each dataset

	Dataset		
Topic	**Document**	**Query (train)**	**Query (validation, test)**
Prepositions	5 or 10	1 or 2	1 or 2
Conjunctions	5 or 10	1 or 2	1 or 2
Phrasal verbs	5	1	1
Verb tenses	20 per tense	2 per tense	2 per tense
Pronouns	5	1	1

3.3 Feature Engineering

In this section, we define grammatical similarity, identify the features needed to represent grammatical similarity, and define the similarity scoring function for each feature. To select the best features for a grammar learning model, we need to consider the complexity of the English language. A discussion of English grammar is necessary to understand the motivation behind our choices of features.

3.3.1 Textual and Grammatical Similarity

We see from the sets of selected terms that among prepositions, conjunctions, phrasal verbs, and verb tenses, common words exist among the terms in two different topics. For example:

`break` (simple present verb tense)

`break into` (phrasal verb)

`into` (preposition)

If the words "break into" are entered as a query in typical search system, it would return documents containing "break" or "into". Depending on the search options available in the search system, the user could search for only documents containing the entire phrase "break into". An advanced search system could also allow wildcard searches. For example, the user could search for documents containing the phrase "break * down", so that the system can return phrases such as "break it down" or "break them down". Thus there are many ways to write a query if only textual similarity is required.

Problems arise if we need to search for text with particular grammatical properties. For example, a user might want to search for sentences using "break" as a noun but not as a verb. Before we define grammatical properties and sentence structure more precisely, consider the following three sentences, in which the first sentence is the query and the other two sentences are documents in the collection:

Query: `I go to school by bus every morning.`

Document 1: `He travels to work by train every afternoon.`

Document 2: `I see the school bus passing by every morning.`

Document 1 has three words (to, by, every) in common with the query. Intuitively, we can also see that they have the same sentence syntax. Document 2 and the query have six words in common (I, school, bus, by, every, morning).

If only textual similarity is considered, BM25 will rank document 2 higher than document 1. But for a grammar search system, we would want document 1 to rank higher by virtue of grammatical similarity with the query. Clearly, ranking by textual similarity alone is not good enough if we also need grammatical similarity. We shall define grammatical similarity between two sentences in terms of:

- The part of speech (POS) of individual words or phrases.
- The *syntax* of a sentence, that is, how the words in the sentence are arranged together.

Therefore, a grammar search system needs to be able to detect the parts of speech and tenses of individual words and phrases. For example:

`I am taking a break from work.` (noun)

`Be careful not to break the glass.` (simple present verb tense)

`I have broken the glass.` (present perfect verb tense)

A grammar search system also needs to be able to detect the syntactic structure of a sentence, and match its structure with that of another sentence, even when there are very few or no common words between two sentences. For example:

```
I am taking my daughter to school today.
She is driving her husband to office tonight.
```

3.3.2 Parts of Speech

For each sentence, we compute the set of POS tags as a feature. We also form bigrams and trigrams of POS tags and define each set of n-grams as a feature. Phrases with the same syntax and parts of speech will have identical n-grams of POS tags. The following example illustrates this:

```
I am taking my daughter to school today.      PRP VBP VBG PRP$ NN IN NN NN
She is driving her husband to office today.   PRP VBZ VBG PRP$ NN IN NN NN
```

We concatenate POS tags with underscores to form sets of bigrams and trigrams, as shown in Table 3.9.

Table 3.9: Bigrams and trigrams of POS tags of two sentences

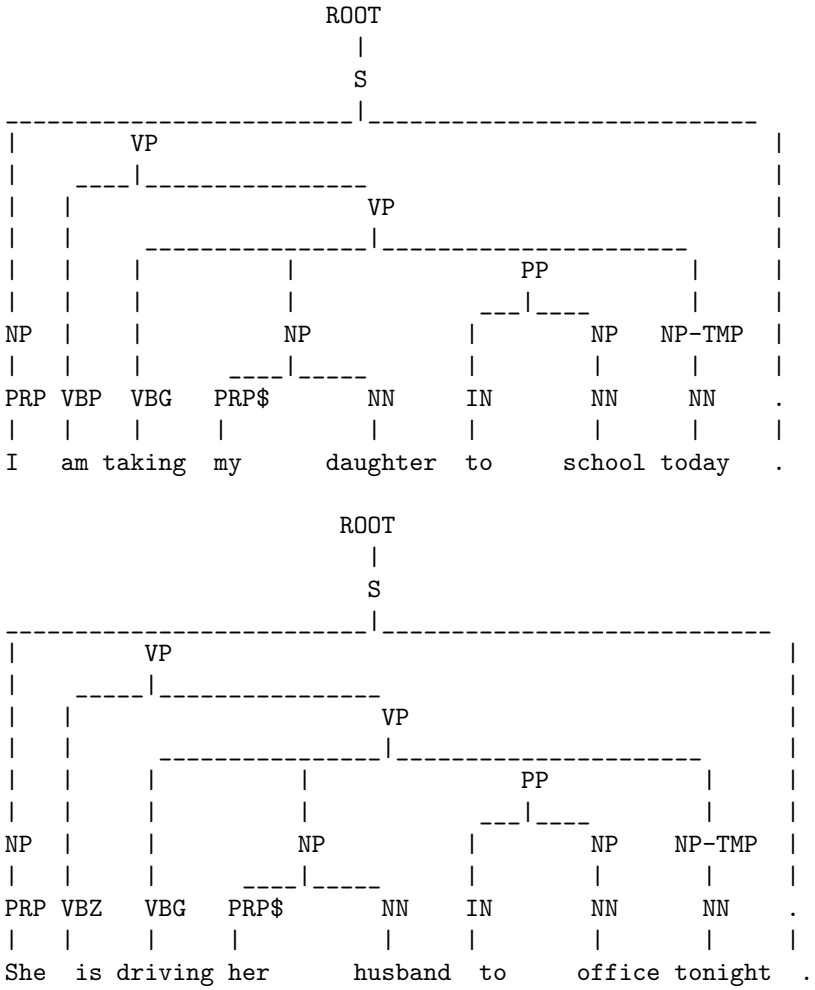
I am taking my daughter to school today.	**She is driving her husband to office today.**
PRP_VBP	PRP_VBZ
VBP_VBG	VBZ_VBG
VBG_PRP$	VBG_PRP$
PRP$_NN	PRP$_NN
NN_IN	NN_IN
IN_NN	IN_NN
NN_NN	NN_NN
PRP_VBP_VBG	PRP_VBZ_VBG
VBP_VBG_PRP$	VBZ_VBG_PRP$
VBG_PRP$_NN	VBG_PRP$_NN
PRP$_NN_IN	PRP$_NN_IN
NN_IN_NN	NN_IN_NN
IN_NN_NN	IN_NN_NN

Except for the verbs "am" (VBP, verb, non-3rd person singular present) and "is" (VBZ, verb, 3rd person singular present), all the other words in the two sentences have the same parts of speech. Among the bigrams, six are identical. Among the trigrams, four are identical. Identical bigrams and trigrams indicate that parts of the two sentences have the same syntax.

For each sentence, we extract the sets of POS unigrams, bigrams and trigrams as features. Given two sets of POS unigrams, bigrams or trigrams, we use BM25 to compute the similarity score.

3.3.3 Constituency Parse Trees and Production Rules

While parts of speech of individual words and phrases can be easily represented with n-grams of POS tags, sentence syntactic structure can be represented using constituency parse trees. Consider the sentences from previous example again. Their constituency parse trees are shown as follows:



Excluding the leaf nodes and the POS tags for “am” and “is”, the two parse trees have identical structure. Equivalently, we can compare the grammar production rules corresponding to the trees, as shown in Table 3.10:

Table 3.10: Production rules of two sentences

I am taking my daughter to school today.	**She is driving her husband to office tonight.**
ROOT -> S	ROOT -> S
S -> NP VP .	S -> NP VP .
NP -> PRP	NP -> PRP
PRP -> 'I'	PRP -> 'She'
VP -> VBP VP	VP -> VBZ VP
VBP -> 'am'	VBZ -> 'is'
VP -> VBG NP PP NP-TMP	VP -> VBG NP PP NP-TMP
VBG -> 'taking'	VBG -> 'driving'
NP -> PRP$ NN	NP -> PRP$ NN
PRP$ -> 'my'	PRP$ -> 'her'
NN -> 'daughter'	NN -> 'husband'
PP -> IN NP	PP -> IN NP
IN -> 'to'	IN -> 'to'
NP -> NN	NP -> NN
NN -> 'school'	NN -> 'office'
NP-TMP -> NN	NP-TMP -> NN
NN -> 'today'	NN -> 'tonight'
. -> '.'	. -> '.'

If we exclude all the productions with leaf nodes, then the two sets of productions are again almost identical. Since all parse trees will have the production `ROOT -> S` at the root node, we will exclude `ROOT -> S` as well. We replace each arrow with an underscore and use the reduced sets of productions to compare the syntactic structure of two sentences, as shown in Table 3.11.

Table 3.11: Reduced sets of productions rules for two sentences

I am taking my daughter to school today.	**She is driving her husband to office tonight.**
S_NP_VP_.	S_NP_VP_.
NP_PRP	NP_PRP
VP_VBP_VP	VP_VBZ_VP
VP_VBG_NP_PP_NP-TMP	VP_VBG_NP_PP_NP-TMP
NP_PRP$_NN	NP_PRP$_NN
PP_IN_NP	PP_IN_NP
NP_NN	NP_NN
NP-TMP_NN	NP-TMP_NN

For each sentence, we extract this set of productions as a feature. Given two sets of productions, we use BM25 to compute the similarity score.

3.3.4 Answer as a Model Feature

In our system, users enter a sentence as a query. The user has the option of indicating a word or phrase in the sentence to be focus of the query. The system matches the focus word or phrase with the answer of a grammar question. We shall call the focus word or phrase the *indicated answer* in the query.

Intuitively, the indicated answer would be the most important part of the query and should be extracted as a feature. This begs the question: is the answer the *only* necessary feature? If the system returns all grammar questions whose answers match the indicated answer, would that give the perfect result? The answer is that in some cases, matching the answer is good enough. But in many cases, textual matching of the answer is inadequate for a grammar search system.

First, textual matching does not work well with verbs. Verbs in different tenses have different spellings. Consider a query with the phrasal verb "break down" indicated as the answer. A grammar search system should also be able to match this with other verb tenses of the phrasal verb, such as the past tense (*broke* down) and future tense (*broken* down).

Consider a query containing the answer "have gone". The system must be able to detect that this is the present perfect verb tense and return other sentences with the same verb tense (e.g., has gone, have done, has done).

Second, as discussed earlier, textual matching does not detect parts of speech. If the query contain the answer "break", textual matching cannot tell if the word is used as a noun or verb in the query. Sometimes the different possible parts of speech of the same word can be subtle. In Section 3.2.5, we gave the example of the word "after", which can be a conjunction or a preposition, depending on the syntax of the sentence.

Third, in grammar learning, it is important for a learner to learn the subtle differences in the meaning and usage of similar words or phrases. That is the benefit of multiple choice questions. By offering wrong answer choices that seem correct, grammar learners learn the differences between similar words or phrases. For example, the prepositions "through" and "throughout" are used differently. A good model would be able to mimic a multiple choice question, by returning grammar questions containing answers that are different from the query, but are almost similar in meaning or usage.

3.3.5 Grammatical Properties of the Answer

We extract grammatical features from the answer, by considering the position of a word in a multi-word answer, and also by considering the words to the left and right of the answer. We observe the following properties:

The first word of the answer is a predictor of the topic. For verb tenses, the first word of the answer is all you need to predict whether it is a present (e.g., is, has), past (e.g., was, had) or future (e.g., will) tense, and whether it is a continuous (e.g., is, was) or perfect (e.g., has, had) tense. For a two-word answer, if the first word is a verb, the answer is likely to be a phrasal verb.

The last word of the answer is a predictor of the topic. For a multi-word answer, if the last word of the answer is a verb, the answer is likely to be a verb tense. The part of speech of the verb is a good predictor of the tense (e.g., "has eaten `VBZ VBN`" is present perfect tense, "is eating `VBZ VBG`" is present continuous tense). If the last word is a preposition, the answer is likely to be a phrasal verb.

The length of the answer is a predictor of the topic. Almost all pronouns are single words, while verb tenses have two or three words depending on the tense. Most phrasal verbs have two or three words, but very often, phrasal verbs are used with a noun separating them (e.g., drop *me* off, break *it* down).

The part of speech of the word before or after the answer may be related to the answer. The word before a verb tense is almost always a noun, and very often a personal pronoun (e.g., *I* have eaten, *He* will do). Therefore, the next word after a personal pronoun is likely to be a verb.

Some answers tend to be at the start or end of a sentence. Reflexive pronouns tend to be at the end of a sentence (e.g., Please look after *yourself*.). Some prepositions and conjunctions are often at the start of a sentence (e.g., If..., According to...).

3.3.6 Topic as a Model Feature

In our raw data, we classify and label each sentence into one of five topics. But we will not use the topic as a model feature. These topic labels only serve to help determine the relevance of each document with respect to a query. Each document-query data instance will be labelled with a relevance value. Relevance labelling is explained in Section 3.4.4.

Whether it is done by humans or by an intelligent system, grammar topic labelling is a non-trivial task requiring a thorough understanding of the taxonomy of English grammar. Rather

than attempting to label the data accurately with a precisely defined set of topics, it is less costly (and possibly more effective) to train a general model to learn to detect the topic by learning the grammatical properties of the query.

In our search system, we do not provide users with the option of entering the topic as a query input. In reality, most grammar learners would not be able to accurately specify the topic. Imagine a student Jenny coming across the phrase "on behalf of" in the sentence "I attended the meeting on behalf of my boss." She wants to find more examples of sentences containing the preposition "on behalf of". But she would not know that it is a preposition if she had not learned this phrase before.

By entering the query "I attended the meeting (on behalf of) my boss.", in which the indicated answer is enclosed in parentheses, a good model would be able to detect that the answer is a preposition.

3.3.7 Fields

In the LETOR 3.0 and LETOR 4.0 datasets, the researchers divided a document into fields, such as the body and the title. For each field, the researchers computed a set of features, as described in Section 2.4.4. We use a similar approach by dividing a sentence into fields. Consider the sentence containing the underlined answer:

`The company is expanding fast and has opened an office in India.`

We define five fields:

Sentence: `The company is expanding fast and has opened an office in India.`

Substring: `is expanding fast and has opened an office in India`

Before: `is expanding fast and`

After: `an office in India`

Answer: `has opened`

Substring consists of four words before the answer (the field "Before"), the answer itself (the field "Answer"), and four words after the answer (the field "After"). We decided on four words because extracting a substring of this length tends to result in a meaningful shorter sentence.

3.3.8 Feature Definitions

Within each field, we define features. Each feature in a query will be matched with the same feature in a document. Given a query and document, a similarity score will be computed for each feature. We use BM25 as to score most of the features. Some features have a binary score: 1 if the query feature and document feature match, 0 of there is no match. All features and the scoring function for each feature are shown in Tables 3.12, 3.13, 3.14, 3.15 and 3.16. There are a total of 33 features.

Table 3.12: Features for the field: Sentence

	Feature	**Example**	**Score**
1	Words	`The company is expanding fast and has opened an office in India`	BM25
2	POS tags	`DT NN VBZ VBG RB CC VBZ VBN DT NN IN NNP`	BM25
3	POS bigrams	`DT_NN NN_VBZ VBZ_VBG VBG_RB RB_CC CC_VBZ VBZ_VBN VBN_DT DT_NN NN_IN IN_NNP`	BM25
4	POS trigrams	`DT_NN_VBZ NN_VBZ_VBG VBZ_VBG_RB VBG_RB_CC RB_CC_VBZ CC_VBZ_VBN VBZ_VBN_DT VBN_DT_NN DT_NN_IN NN_IN_NNP`	BM25
5	Productions	`S_NP_VP_. NP_DT_NN VP_VP_CC_VP VP_VBZ_VP VP_VBG_ADVP ADVP_RB VP_VBZ_VP VP_VBN_NP NP_NP_PP NP_DT_NN PP_IN_NP NP_NNP`	BM25

Table 3.13: Features for the field: Substring

	Feature	**Example**	**Score**
1	Words	`is expanding fast and has opened an office in India`	BM25
2	POS tags	`POS VBZ VBG RB CC VBZ VBN DT NN IN NNP`	BM25
3	POS bigrams	`VBZ_VBG VBG_RB RB_CC CC_VBZ VBZ_VBN VBN_DT DT_NN NN_IN IN_NNP`	BM25
4	POS trigrams	`VBZ_VBG_RB VBG_RB_CC RB_CC_VBZ CC_VBZ_VBN VBZ_VBN_DT VBN_DT_NN DT_NN_IN NN_IN_NNP`	BM25
5	Productions	`SINV_VP_NP VP_VP_CC_VP VP_VBZ_VP VP_VBG_ADVP ADVP_RB VP_VBZ_VP VP_VBN NP_NP_PP NP_DT_NN PP_IN_NP NP_NNP`	BM25

Table 3.14: Features for the field: Before

	Feature	Example	Score
1	Words	`is expanding fast and`	BM25
2	Last word	`and`	1 if match, 0 otherwise
3	POS last word	`CC`	1 if match, 0 otherwise
4	POS tags	`VBZ VBG RB CC`	BM25
5	POS bigrams	`VBZ_VBG VBG_RB RB_CC`	BM25
6	POS trigrams	`VBZ_VBG_RB VBG_RB_CC`	BM25
7	Productions	`FRAG_VP VP_VBZ_VP VP_VBG_ADVP_ADVP ADVP_RB ADVP_CC`	BM25

Table 3.15: Features for the field: After

	Feature	Example	Score
1	Words	`an office in India`	BM25
2	First word	`an`	1 if match, 0 otherwise
3	POS first word	`DT`	1 if match, 0 otherwise
4	POS tags	`DT NN IN NNP`	BM25
5	POS bigrams	`DT_NN NN_IN IN_NNP`	BM25
6	POS trigrams	`DT_NN_IN NN_IN_NNP`	BM25
7	Productions	`NP_NP_PP NP_DT_NN PP_IN_NP NP_NNP`	BM25

Table 3.16: Features for the field: Answer

	Feature	Example	Score
1	Words	`has opened`	BM25
2	First word	`has`	1 if match, 0 otherwise
3	Last word	`opened`	1 if match, 0 otherwise
4	POS first word	`VBZ`	1 if match, 0 otherwise
5	POS last word	`VBN`	1 if match, 0 otherwise
6	POS concatenated	`VBZ_VBN`	1 if match, 0 otherwise
7	Is first word of sentence	`False`	1 if both True, 0 otherwise
8	Is last word of sentence	`False`	1 if both True, 0 otherwise
9	Length (number of words)	`2`	1 if match, 0 otherwise

3.4 Feature Extraction

We now describe the process of feature extraction, and of creating the training, validation and testing datasets. The process is outlined as follows:

1. Extract all 33 features from each sentence.
2. Upload data, including features, for all 850 documents to Solr.
3. Upload feature definitions to Solr.
4. Upload linear model definitions to Solr.
5. Send queries (including features) one by one to Solr and compute query-document feature values for top 50 documents.
6. Compute relevance for each query-document.

3.4.1 Document Upload to Solr

The first step of the process is to extract all 33 features from all 1154 sentences. These are saved in separate CSV files for documents, training queries, and validation/testing queries. Next, we upload the 850 documents to Solr. Figure 3.2 shows the data in Solr for one document instance.

3.4.2 Feature and Model Stores

In step 3 of the process, we define features in JSON format and upload the information to Solr, which will save the information in a *feature store* file named `_schema_feature-store.json`. The feature definition specifies the way we want Solr to compute the feature value. We specify the Solr query parser and the field in Solr to match the query with.

In step 4, we define a linear model in JSON format and upload the information to Solr, which will save the information in a *model store* file named `_schema_model-store.json`. This is a "dummy" untrained model which Solr will use to compute feature values. The features of a model are defined in a feature store.

For our experiments, we define two different linear models. Linear Model 1 has the five features from the Sentence field. Linear Model 2 has the remaining 28 features from the other four fields (Substring, Before, After, Answer).

```
{
  "id":"1",
  "qb_question":["The Olympics team is aiming * a gold medal next year."],
  "qb_answer":["for"],
  "qb_choice1":["about"],
  "qb_choice2":["with"],
  "qb_choice3":["after"],
  "qb_topic_id":[1],
  "qa":["The Olympics team is aiming for a gold medal next year."],
  "qa_pos":["DT NNP NN VBZ VBG IN DT NN NN IN NN"],
  "qa_pos_bigram":["DT_NNP NNP_NN NN_VBZ VBZ_VBG VBG_IN IN_DT DT_NN NN_NN NN_IN IN_NN"],
  "qa_pos_trigram":["DT_NNP_NN NNP_NN_VBZ NN_VBZ_VBG VBZ_VBG_IN VBG_IN_DT IN_DT_NN DT_NN_NN NN_NN_IN NN_IN_NN"],
  "qa_parse_tree":["S_NP_VP_. NP_DT_NNPS_NN VP_VBZ_VP VP_VBG_PP_NP-TMP PP_IN_NP NP_DT_NN_NN NP-TMP_JJ_NN"],
  "ss":["Olympics team is aiming for a gold medal next"],
  "ss_pos":["NNP NN VBZ VBG IN DT NN NN IN"],
  "ss_pos_bigram":["NNP_NN NN_VBZ VBZ_VBG VBG_IN IN_DT DT_NN NN_NN NN_IN"],
  "ss_pos_trigram":["NNP_NN_VBZ NN_VBZ_VBG VBZ_VBG_IN VBG_IN_DT IN_DT_NN DT_NN_NN NN_NN_IN"],
  "ss_parse_tree":["S_NP_VP NP_NNP_NN VP_VBZ_VP VP_VBG_PP_ADVP PP_IN_NP NP_DT_NN_NN ADVP_RB"],
  "before":["Olympics team is aiming"],
  "before_last":["aiming"],
  "before_last_pos":["VBG"],
  "before_pos":["NNP NN VBZ VBG"],
  "before_pos_bigram":["NNP_NN NN_VBZ VBZ_VBG"],
  "before_pos_trigram":["NNP_NN_VBZ NN_VBZ_VBG"],
  "before_parse_tree":["S_NP_VP NP_NNP_NN VP_VBZ_VP VP_VBG"],
  "after":["a gold medal next"],
  "after_first":["a"],
  "after_first_pos":["DT"],
  "after_pos":["DT NN NN IN"],
  "after_pos_bigram":["DT_NN NN_NN NN_IN"],
  "after_pos_trigram":["DT_NN_NN NN_NN_IN"],
  "after_parse_tree":["S_NP_ADVP NP_DT_NN_NN ADVP_RB"],
  "ans":["for"],
  "ans_first":["for"],
  "ans_last":["for"],
  "ans_pos":["IN"],
  "ans_first_pos":["IN"],
  "ans_last_pos":["IN"],
  "ans_is_first":["x"],
  "ans_is_last":["x"],
  "ans_length":[1.0],
  "_version_":1681690925745569792},
{
```


Figure 3.2: Features data in Solr for one document

We define the models this way because of the two types of query input which we will experiment with. The query input can either be a complete sentence (Model 1) or a sentence and the indicated answer (Model 2). The information in the model store for Linear Model 1 is shown below.

```
ms1 = {
    "store" : "feature_store1",
    "name" : "linear_model1",
    "class" : "org.apache.solr.ltr.model.LinearModel",
    "features" : [
        { "name": "qa_original_score" },
        { "name": "qa_pos" },
        { "name": "qa_pos_bigram" },
        { "name": "qa_pos_trigram" },
        { "name": "qa_parse_tree" },
    ],
    "params" : {
        "weights" : {
            "qa_original_score": 1.0,
            "qa_pos": 0.0,
            "qa_pos_bigram": 0.0,
            "qa_pos_trigram": 0.0,
            "qa_parse_tree": 0.0,
        }
    }
}
```

The first feature `qa_original_score` is the entire sentence. The score computed for this feature will be the default BM25 score for textual matching, that is, the original BM25 score if no ranking model is used.

We set the weight for the feature `qa_original_score` to be 1 and the weights for the other features to be 0 so that Solr will rank the results by this BM25 score only. We will call the scores the *original scores* and the rankings from this model the *original rankings*.

The information in the feature store for Linear Model 1 is shown on the next page. The parameters for each feature include the query parser (`dismax`, which uses BM25 by default) and the field (query field `qf`) to search. Note that we need not define the parameters for the first feature because we have defined its class to be the `OriginalScoreFeature`.

```
fs1 = [
    {
        "store": "feature_store1",
        "name": "qa_original_score",
        "class": "org.apache.solr.ltr.feature.OriginalScoreFeature",
        "params": {}
    },
```

```
    {
        "store": "feature_store1",
        "name": "qa_pos",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=qa_pos}${q2}" }
    },
    {
        "store": "feature_store1",
        "name": "qa_pos_bigram",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=qa_pos_bigram}${q3}" }
    },
    {
        "store": "feature_store1",
        "name": "qa_pos_trigram",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=qa_pos_trigram}${q4}" }
    },
    {
        "store": "feature_store1",
        "name": "qa_parse_tree",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=qa_parse_tree}${q5}" }
    },
]
```

The definitions in the model store and feature store for Linear Model 2 are given in Appendices B and C respectively.

3.4.3 Feature Extraction in Solr

We send each query instance from the query feature datasets to Solr via its REST API. For Linear Model 1, the data we send to Solr is shown below:

```
payload = {
    "q": f1,
    "defType": "edismax",
    "qf": "qa",
    "rq": f'{{!ltr model=linear_model1 \
        efi.q2="{f2}" \
        efi.q3="{f3}" \
        efi.q4="{f4}" \
        efi.q5="{f5}"}}',
    "fl": "id,[features]",
    "rows": 50
}
```

In each payload, we pass the query string (the variable `f1`) and the other query features (`f2`, `f3`, `f4`, `f5`), and specify the parameters for the first feature `qa_original_score` (query parser

`edismax`, query field `qa`). We also specify the model to be used (`linear_model1`), the required output (document id and feature values), and the top N documents to output ($N = 50$).

Solr will compute and return the five feature values for this query with respect to each of the top 50 documents. Essentially, we are performing a query and asking Solr to return the top 50 documents, plus all the feature values. The results will have the original rankings.

Since we have 152 training queries and each query is paired with 50 documents, there are 7600 query-document instances in the training dataset. There are 75 validation queries and 77 testing queries. Hence there are 3750 query-document instances in the validation dataset and 3850 query-document instances in the testing dataset.

We will be using the algorithms in the RankLib library [9]. Every algorithm requires datasets in the LETOR dataset format described in Section 2.4.4.

```
relevance_label query_id feature1:value feature2:value... # document_id
```

At this point, we have all the data we need for our two experimental models (Model 1 and Model 2) except the relevance label. An example of a query-document instance in a dataset for Model 1 is shown below. The final step in the preparation of the datasets is to prepend the relevance label to each query-document instance.

```
qid:851 1:10.217693 2:3.4914756 3:6.1261463 4:4.1029177 5:3.8547819 # docid:554
```

3.4.4 Relevance Labels

Every query-document instance needs to be judged for relevance and assigned a relevance value. Typically, this value is a class label and the ground truth for that data instance. For each model, we will define a relevance criteria based on selected features.

For Model 1 which has five features, the first feature is textual and the other four features are grammatical as shown in Table 3.12. We define a document to be relevant to a query if all BM25 scores for the four grammatical features are at least 3.0. Otherwise the document is not relevant to the query. This criteria ensures that there is *some* grammatical similarity for all four features. The criteria is shown in Table 3.17.

Table 3.17: Relevance criteria for Model 1

Relevance	Criteria
1	BM25 >= 3.0 for POS tags, POS bigrams, POS trigrams, Productions
0	Otherwise

An example of six sentences from our data and their relevance labels is shown in Table 3.18. The rank in the first column is the original ranking. The first three sentences clearly have phrases (I am waiting, I am taking, I am going) with identical parts of speech. The last three sentences are ranked quite highly but fails in at least one of our four criteria for relevance.

Table 3.18: Example of sentences and their relevance labels (Model 1)

Rank	**Relevance**	**Query:** I am waiting for the train.
1	1	I am taking the train to London tomorrow.
12	1	I am very tired so I am going to take a nap.
15	1	I am going into the room to check.
5	0	He has been waiting for the bus in the past one hour but it is not here yet.
7	0	He looks after my cat whenever I am away for work.
8	0	I am satisfied, but I am not entirely happy.

For Model 2, since the answer is indicated, we can define a more objective criteria. We define four ordinal relevance labels 0, 1, 2, and 3, with 0 representing no relevance and 3 representing greatest relevance. We use the features of the Answer field (Table 3.16) to define the criteria shown in Table 3.19.

Table 3.19: Relevance criteria for Model 2

Relevance	**Criteria**
3	Criteria for label 1 and label 2, and First word = 1, Last word = 1
2	Criteria for label 1, and POS first word = 1
1	POS last word = 1, Length = 1, Topic id = 1
0	Otherwise

What this criteria says is that for a document to be minimally relevant (label 1) to a query, they must be from the same topic, their answers must have the same number of words, and the POS tags of the last words of the answers must be identical.

To be judged as more relevant (label 2), the POS tags of the first words of the answers must also be identical. To be judged as most relevant (label 3), the first words and last words of the answers must also be identical.

We compute and prepend the relevance label to each row in the datasets. Sample data from the training datasets of Models 1 and 2 are shown in Appendices D and E.

An example of four sentences from our data and their relevance labels is shown in Table 3.20.

Only the first two sentences have the same verb tense as the query. Although the third sentence has a different verb tense, it is judged to be minimally relevant because the last word, "playing", has the same part of speech as "thinking". The fourth sentence is judged to be irrelevant because it is of a different topic (phrasal verbs).

Table 3.20: Example of sentences and their relevance labels (Model 2)

Rank	**Relevance**	**Query:** They were thinking of buying a new house before the recession.
1	3	We were thinking of moving to New Zealand before the pandemic began.
19	2	The naughty boys were making fun of the poor girl the other day.
13	1	She is playing a new game on the computer in her room right now.
3	0	When we looked into buying a house, we decided it was better to rent.

3.5 Ranking Algorithms

The RankLib library [9] is a Java LTR library providing the eight ranking algorithms shown in Table 3.21.

Table 3.21: Algorithms in RankLib library

Algorithm	**Approach**
MART (Multiple Additive Regression Trees)	Pointwise
Random Forests	Pointwise
LambdaMART	Pairwise
RankBoost	Pairwise
RankNet	Pairwise
Coordinate Ascent	Listwise
AdaRank	Listwise
ListNet	Listwise

RankLib provides the evaluation metrics MAP, NDCG@k, DCG@k, P@k, RR@k, ERR@k. If not specified, RankLib uses the default metric ERR@10.

3.6 Experiments

Our goal is to build the best ranking models for our search system. In our system, users enter a sentence as a query, with the option of indicating a word or phrase in the sentence to be focus of the query. The system will match the answers of grammar questions with the focus words.

We will build Model 1 for queries with no focus words indicated, and Model 2 for queries in which focus words are indicated. We will select a suitable algorithm to train our models, and experiment with different features.

3.6.1 Algorithms

We train and evaluate Model 2 (with 28 features) using all eight algorithms and the NDCG@10 metric to see the behavior of each algorithm and the output we would obtain. Table 3.22 shows the results in this preliminary experiment.

Table 3.22: Test results for different algorithms using Model 2

Algorithm	**NDCG@10**
MART (Multiple Additive Regression Trees)	0.9703
Random Forests	0.9718
LambdaMART	0.9710
RankBoost	0.9663
RankNet	0.7539
Coordinate Ascent	0.9642
ListNet	0.6998
AdaRank	0.8948

We see that the two neural network algorithms ListNet and RankNet performed poorly. This is likely to be due to the small size of our training dataset. The decision tree algorithms, MART, Random Forests, and LambdaMART, performed well. Research has shown that pairwise and list-wise approaches usually outperform the pointwise approach [30]. For this reason, and the fact that LambdaMART is designed to optimize the NDCG metric, we use LambdaMART for all subsequent experiments.

3.6.2 Model 1

For Model 1, we use only features from the Sentence field. Our baseline model is the model with only one feature: the words in the entire sentence, ranked by BM25 for textual similarity.

We train models with different combinations of the remaining four features. Since the relevance labels are binary, we evaluate them using the MAP metric. Table 3.23 shows the results. The last column shows the absolute change from the baseline result.

Table 3.23: Test results for models with different features (Model 1)

Model	**# of Features**	**MAP**	**Change**
Baseline: Sentence (Words)	1	0.3910	-
Sentence (Words, POS tags)	2	0.5411	+0.1501
Sentence (Words, POS bigrams)	2	0.6863	+0.2953
Sentence (Words, POS trigrams)	2	0.7234	+0.3324
Sentence (Words, Productions)	2	0.5908	+0.1998
Sentence (Words, POS bigrams, POS trigrams)	3	0.7329	+0.3419
Sentence (Words, POS tags, POS bigrams, POS trigrams)	4	0.7675	+0.3765
Model 1	**5**	**0.8549**	**+0.4639**

The results show that adding each additional feature to the baseline improved the MAP score. The complete Model 1 with five features improved the MAP score by 46.3% over the baseline.

We can generate model statistics from RankLib. The statistics for Model 1 are shown below. It shows the number of internal nodes on the regression tree for each feature. The feature frequencies give an indication of which feature is important. If the frequency of a feature is very small, we may consider discarding it.

```
Feature frequencies :
Feature[1] :     291
Feature[2] :     232
Feature[3] :     226
Feature[4] :     385
Feature[5] :     261

Total Features Used: 5

Min frequency    :     226.00
Max frequency    :     385.00
Median frequency :     261.00
Avg frequency    :     279.00
Variance         :    4180.50
STD              :      64.66
```

3.6.3 Model 2

For Model 2, our baseline model is the model with only one feature: the words in the substring, ranked by BM25 for textual similarity.

We wish to determine if there are any features that will not improve on the baseline rankings and should be dropped. Since we have classified the 28 features by field, we train and evaluate models with features from different fields. Table 3.24 shows the results.

Table 3.24: Test results for models with features from different fields (Model 2)

Model	**# of Features**	**NDCG@10**	**Change**
Baseline: Substring (Words)	1	0.5479	-
Field: Substring	5	0.6561	+0.1082
Substring (Words) + Field: Before	8	0.6548	+0.1069
Substring (Words) + Field: After	8	0.6557	+0.1078
Substring (Words) + Fields: Before, After	15	0.7375	+0.1896
Fields: Substring, Before, After	19	0.8205	+0.2726
Substring (Words) + Field: Answer	10	0.9620	+0.4141
Fields: Substring, Answer	14	0.9674	+0.4195
Substring (Words) + Fields: Answer, Before, After	24	0.9680	+0.4201

Performance appears to improve as we add more features. When the Answer field is added, the NDCG@10 score increases significantly. This is expected because relevance labels were determined using features from the Answer field.

Next, we test the influence of POS tags and production rules as features. For this experiment, we do not include any features from the Answer field. Table 3.25 shows the results.

Table 3.25: Influence of POS tags and production rules as features (Model 2)

Model	**# of Features**	**NDCG@10**	**Change**
Baseline: Substring (Words)	1	0.5479	-
Substring (Words, Productions) + Before (Productions) + After (Productions)	4	0.6290	+0.0811
Substring (Words, POS tags) + Before (POS tags) + After (POS tags)	4	0.6617	+0.1138
Substring (Words, POS tags, POS bigrams, POS trigrams) + Before (POS tags, POS bigrams, POS trigrams) + After (POS tags, POS bigrams, POS trigrams)	10	0.7397	+0.1918

In absolute terms, the results show that adding production features to the baseline improved the NDCG@10 score by 8.1%. Adding POS tags of individual words as features improved the NDCG@10 score by 11.3% over the baseline. When POS bigrams and trigrams are also added as features, the NDCG@10 score improved by 19.1% over the baseline.

For our last experiment, we test the influence of the word before and the word after the answer, including the POS tags of these words. The result in Table 3.26 shows that the NDCG@10 scored improved by 8.3% over the baseline.

Table 3.26: Influence of words before and after the answer as features (Model 2)

Model	**# of Features**	**NDCG@10**	**Change**
Baseline: Substring (Words)	1	0.5479	-
Substring (Words) + Before (Last word, POS last word) + After (First word, POS first word)	5	0.6314	+0.0835

Since every combination of fields and features show some improvement over the baseline model, we will use all 28 features for Model 2, and evaluate it using NDCG@1, NDCG@3, NDCG@5, and NDCG@10. Table 3.27 shows the results.

Table 3.27: Final test results for Model 2

Model	**# of Features**	**NDCG@1**	**NDCG@3**	**NDCG@5**	**NDCG@10**
Baseline	1	0.6456	0.5697	0.5402	0.5479
Model 2	**28**	**0.9654**	**0.9682**	**0.9714**	**0.9710**
Change	-	+0.3198	+0.3985	+0.4312	+0.4231

The results show that Model 2 outperforms the baseline model significantly on all metrics. In absolute terms, it outperforms the baseline by 43.1% on NDCG@5 and 42.3% on NDCG@10. The statistics of the model are shown below.

```
Feature frequencies :
Feature[1] :      415
Feature[2] :      394
Feature[3] :      180
Feature[4] :      118
Feature[5] :      297
Feature[6] :      147
Feature[9] :      329
Feature[10] :      33
Feature[11] :       8
Feature[12] :     411
```

```
Feature[13] :     103
Feature[14] :      83
Feature[15] :      86
Feature[16] :     110
Feature[17] :      60
Feature[18] :       2
Feature[19] :     183
Feature[20] :     546
Feature[21] :     100
Feature[22] :      88
Feature[23] :     300
Feature[24] :     164
Feature[25] :     158
Feature[26] :      86
Feature[28] :     126

Total Features Used: 25

Min frequency    :       2.00
Max frequency    :     546.00
Median frequency :     126.00
Avg frequency    :     181.08
Variance         :   20718.66
STD              :     143.94
```

We see that features 7 (Before (Last word)), 8 (Before (POS last word)), and 27 (Answer (Is last word of sentence)) are not used at all. We could discard these three features for our current model. However, we will use these features again in future work when we use more data and improve our models.

3.7 Model Upload to Solr

The results from RankLib show that our models are able to optimize the metrics. To see the actual performance of the models, we need to run some queries in Solr and see the output.

RankLib outputs the regression tree model in a text file. In order to upload the models to Solr, we need to convert the text files to JSON format. We then upload the JSON files to Solr via Solr's REST API. Figure 3.3 shows part of Model 1 in Solr.

```
{
  "name":"lambdamart_model1",
  "class":"org.apache.solr.ltr.model.MultipleAdditiveTreesModel",
  "store":"feature_store_all",
  "features":[
    {
      "name":"qa_original_score",
      "norm":{"class":"org.apache.solr.ltr.norm.IdentityNormalizer"}},
    {
      "name":"qa_pos",
      "norm":{"class":"org.apache.solr.ltr.norm.IdentityNormalizer"}},
    {
      "name":"qa_pos_bigram",
      "norm":{"class":"org.apache.solr.ltr.norm.IdentityNormalizer"}},
    {
      "name":"qa_pos_trigram",
      "norm":{"class":"org.apache.solr.ltr.norm.IdentityNormalizer"}},
    {
      "name":"qa_parse_tree",
      "norm":{"class":"org.apache.solr.ltr.norm.IdentityNormalizer"}}],
  "params":{"trees":[
      {
        "weight":"0.1",
        "root":{
          "feature":"qa_original_score",
          "threshold":"13.569935",
          "left":{
            "feature":"qa_original_score",
            "threshold":"9.567596",
            "left":{
              "feature":"qa_original_score",
              "threshold":"6.8601317",
              "left":{"value":"-1.3503507375717163"},
              "right":{"value":"-0.01865535043179989"}},
            "right":{
              "feature":"qa_pos_trigram",
              "threshold":"10.4560995",
              "left":{
                "feature":"qa_pos",
                "threshold":"10.885765",
                "left":{"value":"0.837558925151825"},
                "right":{"value":"1.8527597188949585"}},
              "right":{"value":"1.821455955505371"}}},
```


Figure 3.3: Part of Model 1 in Solr

3.7.1 Model 1

To test Model 1, we select a random query from the validation/testing query dataset and send the query to Solr. The top 10 results are shown below. Basic Solr search does not use any ranking models. But the results are quite similar to the results from the baseline model because they are both ranked by textual similarity only. Each row in the results is formatted as:

`document_id question (answer)(topic_id)`

The query input is the sentence only. The answer and topic id are displayed for reference.

```
Sentence: I can look after myself so you don't have to be concerned.
Answer: look after
Topic_id: 3

Basic Solr Search
Top 10 results for the query: I can look after myself so you don't have to be concerned.

392 She is very independent. She can * herself. (look after)(3)
393 I hope you will * my garden when I am gone. (look after)(3)
826 Be careful when you handle the knife. Don't cut *. (yourself)(5)
273 I know I should have waited for you * I was so hungry just now. (but)(2)
822 I finished the work all by * after three long months. (myself)(5)
233 You will not be able to cancel this contract * you have signed it. (once)(2)
60  Thank you for your consideration. I look forward to hearing * you. (from)(1)
692 This time next month, I * myself in Paris. (will be enjoying)(4)
259 You can come to me * you need help. (whenever)(2)
206 I need to talk to you * you are done with your work. (after)(2)

LTR Baseline Model
Top 10 results for the query: I can look after myself so you don't have to be concerned.

393 I hope you will * my garden when I am gone. (look after)(3)
392 She is very independent. She can * herself. (look after)(3)
60  Thank you for your consideration. I look forward to hearing * you. (from)(1)
273 I know I should have waited for you * I was so hungry just now. (but)(2)
692 This time next month, I * myself in Paris. (will be enjoying)(4)
822 I finished the work all by * after three long months. (myself)(5)
826 Be careful when you handle the knife. Don't cut *. (yourself)(5)
233 You will not be able to cancel this contract * you have signed it. (once)(2)
206 I need to talk to you * you are done with your work. (after)(2)
259 You can come to me * you need help. (whenever)(2)

LTR Model 1
Top 10 results for the query: I can look after myself so you don't have to be concerned.

10  I will be in Tokyo * two weeks. (for)(1)
209 Soon * we set off, the car began to make strange noises. (after)(2)
276 There was a nation-wide manhunt, * he was nowhere to be found. (but)(2)
278 They don't serve coffee, * they have tea. (but)(2)
292 I will regret it * I don't do this now. (if)(2)
```

```
291 We will miss the train * we don't hurry. (if)(2)
346 You cannot go to the party unless I *. (come along)(3)
384 You will have to * your rifles before the new gun laws take effect. (turn in)(3)
382 I am feeling sleepy so I will * now. (turn in)(3)
422 I hope things will start to * when this pandemic is over. (look up)(3)
```

Although the search results from Model 1 are different compared to the results from basic Solr search and the baseline model, it is hard to discern whether the sentences from the ranking model are grammatically similar to the query and is an improvement on the baseline results.

3.7.2 Model 2

To test Model 2, we use the same query sentence as before. But now, the answer is indicated and our search system will extract a substring consisting of four words before the answer, the answer itself, and four words after the answer. The query input is the substring and the answer only. The topic id is displayed for reference.

```
Sentence: I can look after myself so you don't have to be concerned.
Substring: I can look after myself so you don't
Answer: look after
Topic_id: 3

Basic Solr Search
Top 10 results for the query: I can look after myself so you don't

392 She is very independent. She can * herself. (look after)(3)
393 I hope you will * my garden when I am gone. (look after)(3)
744 Can you show * where I can find a good supermarket? (me)(5)
414 I will * how we can improve the work processes. (look into)(3)
394 I * my neighbour's dog when he is away for business. (look after)(3)
210 I went home immediately * I met you. (after)(2)
822 I finished the work all by * after three long months. (myself)(5)
474 I am counting on you. Please don't *. (let me down)(3)
718 Don't worry, * will not be blamed for this mess your team members created. (you)(5)
273 I know I should have waited for you * I was so hungry just now. (but)(2)

LTR Baseline Model
Top 10 results for the query: I can look after myself so you don't

392 She is very independent. She can * herself. (look after)(3)
393 I hope you will * my garden when I am gone. (look after)(3)
744 Can you show * where I can find a good supermarket? (me)(5)
210 I went home immediately * I met you. (after)(2)
491 If you * so easily, you will never succeed. (give up)(3)
273 I know I should have waited for you * I was so hungry just now. (but)(2)
474 I am counting on you. Please don't *. (let me down)(3)
718 Don't worry, * will not be blamed for this mess your team members created. (you)(5)
```

```
822 I finished the work all by * after three long months. (myself)(5)
394 I * my neighbour's dog when he is away for business. (look after)(3)

LTR Model 2
Top 10 results for the query: I can look after myself so you don't

394 I * my neighbour's dog when he is away for business. (look after)(3)
393 I hope you will * my garden when I am gone. (look after)(3)
395 The nurses * the patients very well in this hospital. (look after)(3)
392 She is very independent. She can * herself. (look after)(3)
420 I was asked to * the report before sending it for approval. (look over)(3)
384 You will have to * your rifles before the new gun laws take effect. (turn in)(3)
414 I will * how we can improve the work processes. (look into)(3)
415 I hope the government will * the alleged fraud. (look into)(3)
413 If you really * it, you will find that this is the best offer you will get. (look into)(3)
314 He * in a rash after eating oranges. (broke out)(3)
```

Model 2 appears to give better top 10 sentences compared to the results from basic Solr search and the baseline model. This conclusion can be made by simply counting the number of sentences that are relevant to the query in terms of grammar topic and the answer.

3.8 Summary

We presented our methods and development process. We built two models, to be deployed in our Solr search system. We trained Model 1 with five features and Model 2 with 28 features using the LambdaMART algorithm. Both models outperformed the baseline models significantly.

Chapter 4

Grammar Practice Website

We developed a grammar practice website to implement our ranking models and demonstrate intelligent search. This chapter describes the development and components of the website.

The website provides different ways to search for grammar questions. The website also provides 1002 grammar questions for practice and comes with authentication and security features. The website is developed using the Django framework [5]. The Django Haystack library [7] provides the API to integrate Solr with Django.

4.1 Grammar Question Search

In this section, we describe the search features on the website.

4.1.1 Search Features

We provide basic search and intelligent search of grammar questions. The basic search page provides four ways to search for grammar questions:

- Basic Solr search (ranked)
- Search by topic (unranked, direct from database)
- Search by question text (unranked, direct from database, filtered by Django Q objects)
- Search by answer choice text (unranked, direct from database, filtered by Django Q objects)

The intelligent search page provides two ways to search for grammar questions:

- Intelligent Solr search by sentence
- Intelligent Solr search by sentence with indicated answer

For intelligent search, the original results are re-ranked by the ranking models. Figure 4.1 illustrates the search features.

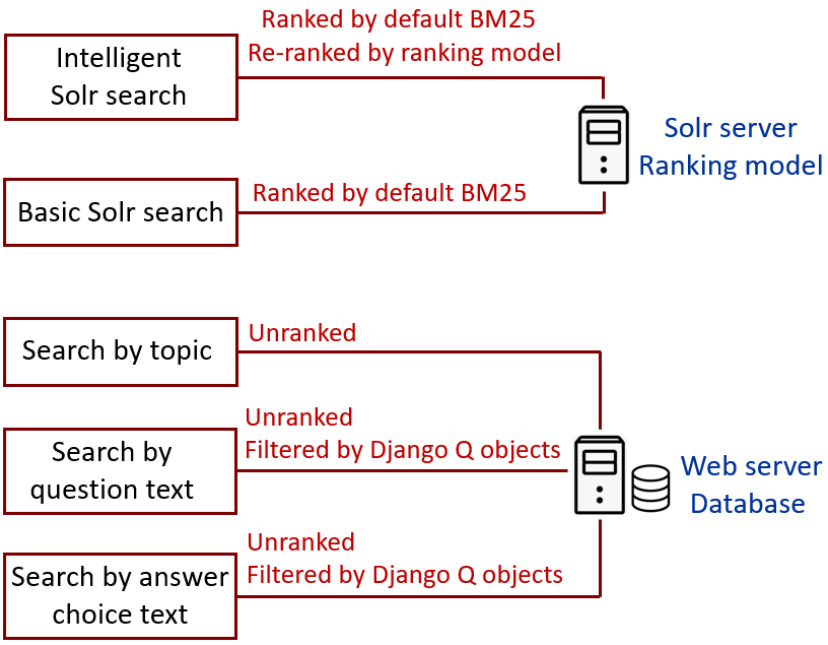


Figure 4.1: Solr search features

4.1.2 Solr Search Process

The index in the Solr server is created from two separate sets of data.

The first set of data contains 1002 grammar questions. Each data instance of a grammar question consists of the question, the correct answer, up to three wrong answer choices, and the topic. This set is uploaded to Solr from the database in the web server, via Haystack. Figure 4.2 shows fields in Solr created for the first set of data.

The second set of data contains the same 1002 grammar questions, and all the 33 features. This set is uploaded to Solr outside of Django, via Solr's REST API, and is shown in Figure 3.2.

```
{
  "id":"appsearch.modelquestionbank.1",
  "django_ct":"appsearch.modelquestionbank",
  "django_id":"1",
  "text":"The Olympics team is aiming * a gold medal next year.\nfor\nabout\nwith\nafter\nnPrepositions",
  "content_auto":"The Olympics team is aiming * a gold medal next year.\nfor\nabout\nwith\nafter\nnPrepositions",
  "_version_":1681612843681579008},
{
```


Figure 4.2: Fields in Solr created from basic data

For basic Solr search, the query terms are matched with the fields created from the first set of data. For intelligent Solr search, the query *features* are matched with the fields created from the second set of data.

For intelligent search, features are first extracted from the query in the Django backend. Then the query, along with all its features, are sent to Solr. Solr will match the query features with the fields created from the second set of data. Solr first ranks the results by default BM25, then re-ranks the top-N results. It then sends the ranked list of document ids back to the web server. Django then retrieves the grammar question for each document id from the database. Finally it sends the ranked list of grammar questions as Django data objects to the client's web browser for rendering. Figure 4.3 shows the process of sending a query to Solr and receiving the results.

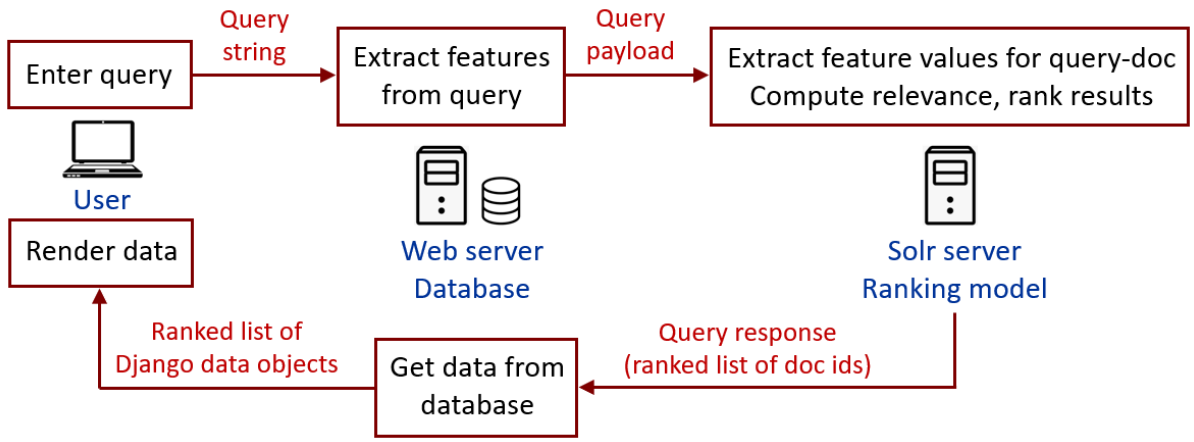


Figure 4.3: Solr search process

4.1.3 Basic Solr Search

The basic Solr search ranks documents by BM25. The search output depends on how Solr stores the data. For this search, all data (i.e., the question, the correct answer, the wrong answer choices, and the topic) in the database are stored in a single field called "text" in Solr. The properties of

this field is defined in Django via Haystack. All queries will be matched with this field. This means that if, for example, the query is "phrasal", Solr will return all grammar questions belonging to the topic phrasal verbs, and all other grammar questions containing the word "phrasal" in the question or answer choices.

An auto-complete feature in Solr allows incomplete spellings of words in a query, as long as the first two (or more) letters of the word are entered. For example, the query "prep" will return all grammar questions belonging to the topic prepositions, and all other grammar questions containing words starting with "prep" in the question and answer choices. The auto-complete feature is enabled by setting the field type of this field as `edge_ngram` . Instead of setting this field type in the Solr schema file, we can set this within Django via the Haystack API. Figure 4.4 shows a screenshot of the basic search page.

4.1.4 Django Q Objects

Besides basic Solr search, we have also implemented searches filtered by Django Q objects. These searches retrieve data directly from the database.

In a Solr search, individual terms in the query are matched with documents. However, Django Q objects treat query text as one contiguous string, and return sentences matching the entire query string, including any space between words. Our system reduces multiple spaces between words to a single space. Leading and trailing spaces are also removed.

Since this is simply string matching, the query "peat" will return grammar questions with words containing the string, such as "repeating". Results are not ranked; they are displayed according to the order of question ids in the database.

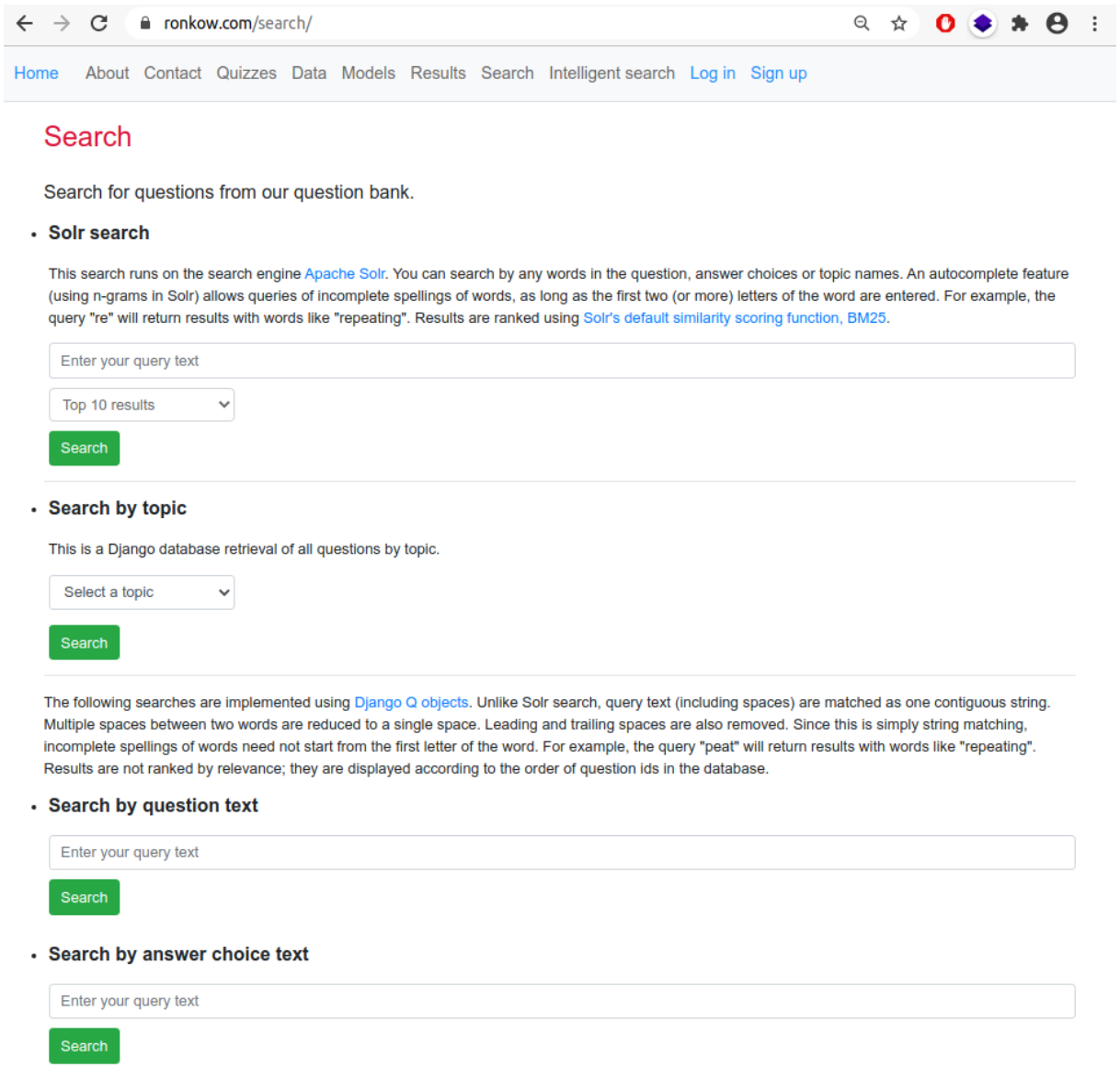


Figure 4.4: Screenshot of basic search page

4.1.5 Intelligent Solr Search

Using two different ranking models, we provide two ways to search for grammar questions:

- **Model 1:** Intelligent Solr search by sentence. The entire sentence entered is taken as the query.
- **Model 2:** Intelligent Solr search by sentence with answer indicated by enclosing a word or phrase in parentheses. The system extracts a substring as the query. The substring consists of four (or less) words before the indicated answer, the answer itself, and four (or less) words after the answer.

For Model 2, although the answer is part of the query input, we do not provide the user the option of specifying the grammar topic as a query input. As explained in Section 3.3.6, topic is not a feature in the model, and users who are English learners may not be able to select the correct topic for their query sentence.

On the intelligent search page, users may select the number of results to display, up to the top 50 results. On the search results page, we display, for the purpose of demonstration, both the original results (which is similar to basic Solr search) and the re-ranked results from the ranking model. To demonstrate re-ranking, the system re-ranks only the top 30 original results. Figure 4.5 shows a screenshot of the intelligent search page.

4.2 Development Tools

This section describes the frameworks, libraries and database used in the development of the website.

4.2.1 Major Tools

We use a Django backend and frontend, and a SQLite database. These components reside in a web server. Solr resides in a separate server.

The four major components required for the search system are:

- Django
- SQLite
- Apache Solr
- Haystack

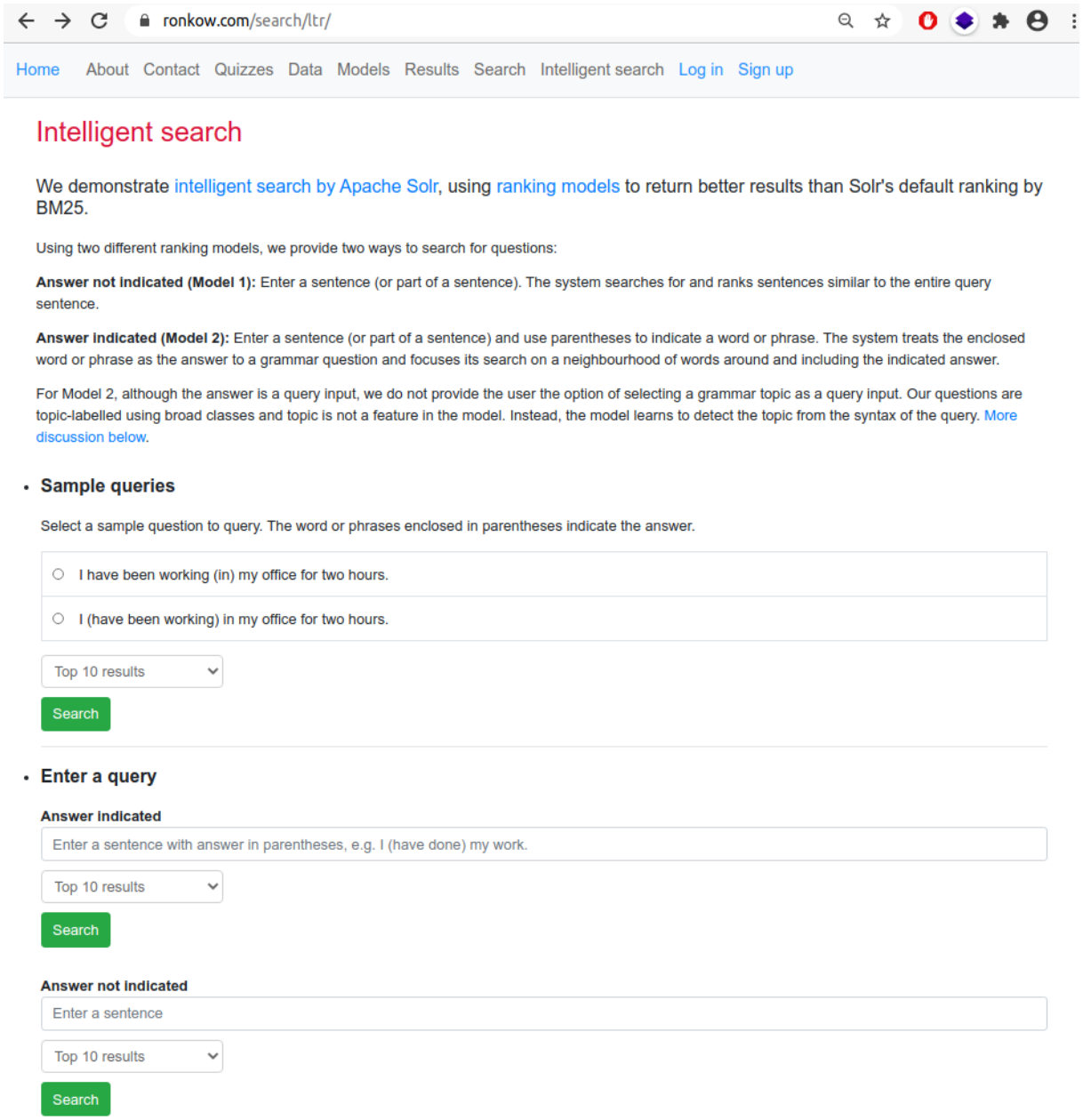


Figure 4.5: Screenshot of intelligent search page

In addition, these libraries are used for model building:

- RankLib
- Stanford CoreNLP
- Natural Language Toolkit (NLTK)

In the Django backend, CoreNLP and NLTK are used to extract features from a query before the query is sent to Solr. The CoreNLP server is accessed via a client in Stanford Stanza [11], the Python NLP library from Stanford.

Developing the complete website requires many other libraries and frameworks. The two major ones are:

- Bootstrap, the frontend design framework [3]
- Django-allauth, the user authentication library [6]

4.2.2 Django

Django is a frontend and backend web development framework. Backend development is in Python, and frontend development integrates the Django template language with HTML. Django is designed to make web development easy for anyone familiar with Python. Some features of Django would surprise web developers using Django for the first time.

For example, compared to some other web frameworks, Django makes it very easy to build and query a database, because there is no need to write any SQL code. Instead, we define a data model in scripts named `models.py`.

In our database, all the grammar questions that are indexed in Solr are stored in a single table. In `models.py`, we define this table in the class `ModelQuestionbank`:

```
class ModelQuestionbank(models.Model):
    qb_topic = models.ForeignKey('appquiz.ModelTopic', on_delete=models.CASCADE,
        related_name='modelqbtopic')
    qb_question = models.TextField(unique=True)
    qb_answer = models.CharField(max_length=50)
    qb_choice1 = models.CharField(max_length=50)
    qb_choice2 = models.CharField(max_length=50)
    qb_choice3 = models.CharField(max_length=50)
```

Under the hood, Django creates the database table. Instead of SQL statements, database queries are written using Django's database API. Database queries return Django data objects which are then passed to the HTML pages to be rendered.

4.2.3 SQLite

SQLite [10] is a basic relational database management system. Unlike other database systems such as MySQL and PostgreSQL which are built for enterprise data, SQLite is designed for local data storage in standalone applications and devices. It is suitable for web applications with medium traffic, in the region of a few hundred thousand hits per day.

Each database table is defined as a data model in Django scripts `models.py`. The data models and their equivalent SQL statements are shown in Appendix A. Figure 4.6 shows the entity relationship diagram of our simple database.

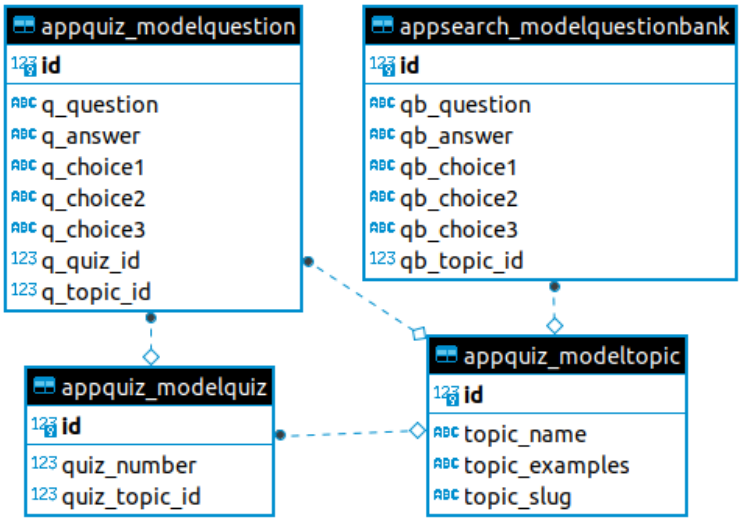


Figure 4.6: Entity relationship diagram of the database

4.2.4 Apache Solr and Django Haystack

The major features of Solr have been described in Section 2.6. We now describe the integration of Solr with Django. The Django Haystack library [7] provides the API to integrate popular search software with Django. Haystack supports Solr and Elasticsearch, both of which are built on Lucene. A few lines of configuration code connect the Django backend and the Solr server:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://my_userid:my_password@my_server:8983/solr/my_core',
    },
}
```

Haystack does the job of moving all data from the SQLite database to Solr for indexing. Furthermore, we can prepare a template file in Django to define a special "catch-all" field to be created in the Solr index. In this special field, simply called `text`, we include all the fields in our data (i.e., the question, the correct answer, wrong answer choices, and the topic). When we upload the data to Solr via Haystack, Solr will store the data in this field. For any query, this is the default field in the index which Solr will search.

However, Haystack provides no API for LTR. Thus, all LTR feature and model specifications are uploaded to Solr via Solr's REST API. We also need to upload all features data to Solr via Solr's REST API, as explained in Section 4.1.2.

4.3 Summary

In this chapter, we described the development and components of the grammar practice website and the grammar question search system in which we deployed our ranking models. We described the search process and the four main components required to develop the search features: the Django backend, SQLite database, Apache Solr, and Django Haystack.

Chapter 5

Conclusion

We conclude our report with an overall summary and a discussion on possible future development and enhancement.

5.1 Overall Summary

The main objective of the project was to use LTR methods to develop ranking models which are able to detect grammatical similarity between two sentences. We also developed a grammar practice website with a grammar question search system in which we deployed our models.

In this report, we provided a concise overview of related work which form the building blocks of the project: BM25, evaluation metrics for ranking, LTR, LambdaMART algorithm, parts of speech tagging, production rules and constituency parse trees, and Apache Solr. We also reviewed recent research on LTR.

We experimented with algorithms and different sets of model features. We trained two main models using LambdaMART. Model 1 takes in a sentence as query input and the model has five features. Model 2 takes in a substring from a sentence and an indicated answer as query input and the model has 28 features.

For Model 2, we divided the query input into four fields and extracted features from each field. The features include parts of speech tags and production rules to detect grammatical similarity. The final model from LambdaMART uses 25 features.

We defined a relevance criteria for each model to judge the relevance of a document with respect to a query. The criteria is based on matching selected query and document features.

Compared to the baseline model which rank results by textual similarity only, Model 1 outper-

formed the baseline by 46.3% in absolute terms on the MAP metric. In absolute terms, Model 2 outperformed the baseline by 43.1% on the NDCG@5 metric and 42.3% on the NDCG@10 metric.

When tested with actual queries, Model 2 appears to give better top 10 sentences compared to the results from basic Solr search and the baseline model. For Model 1, although the search results are different compared to the results from basic Solr search and the baseline model, it is hard to discern whether the sentences are grammatically similar to the query.

5.2 Future Work

We have thus far developed a ranking model using limited training data. We have also developed a basic grammar question search system which recommends grammar questions. There is much room for future work.

5.2.1 Additional Data

Our data covers a small subset of English grammar topics. It covers only a small set of terms in each of the five selected topics. Our training and testing datasets contain sentences with the same set of terms. Therefore our models are only empirically proven to work on these five sets of terms. Because we have chosen topics that are essential in any sentence construction, a little generalization is possible. But more data and a wider range of topics and terms are needed to build a better model.

5.2.2 Deep Neural Models

Compared to tree algorithms, experiments with neural network algorithms RankNet and ListNet produced disappointing NDCG@10 results. This is likely due to insufficient data.

In many NLP tasks, deep neural models have outperformed traditional machine learning models. Other than better results, the distinct advantage of deep learning methods over traditional machine learning methods is that time-consuming feature engineering is no longer necessary. The trade-off is that a large amount of data is necessary.

For this project, the challenge would be to develop neural models that can learn grammar with only raw sentences as training data. Section 2.7 highlighted recent research in the use of deep neural models for ranking. We expect to see research progress in this area in the next few years, and possibly more deep learning libraries for IR.

5.2.3 A Complete Recommender System

We have developed a simple grammar question search system with limited features to recommend grammar questions. A truly useful website for grammar learners would include a wider selection of topics and questions in the question bank. Question search would provide more filtering options. Users would be able to save their grammar practice history. The system would be able to recommend questions related to those that the learner had made mistakes in. Questions could be classified into different levels of difficulty to allow question recommendation tailored to suit the learner's ability.

5.3 Website and Source Files

The grammar practice website is accessible at:

https://ronkow.com/grammar/

The source files, including the data, models, and results, are accessible at:

https://github.com/ronkow/solr-learning-to-rank

Bibliography

[1] Apache lucene. https://lucene.apache.org/. 5, 13

[2] Apache solr. https://lucene.apache.org/solr/. 13

[3] Bootstrap. https://getbootstrap.com/. 60

[4] Cambridge dictionary. https://dictionary.cambridge.org/. 21

[5] Django. https://www.djangoproject.com/. 53

[6] Django-allauth. https://www.intenct.nl/projects/django-allauth/. 60

[7] Django haystack. http://haystacksearch.org/. 53, 61

[8] Natural language toolkit. https://www.nltk.org/. 10

[9] Ranklib. https://sourceforge.net/p/lemur/wiki/RankLib/. 41, 43

[10] SQLite. https://www.sqlite.org/index.html. 61

[11] Stanford stanza. https://stanfordnlp.github.io/stanza/. 60

[12] Qingyao Ai, Keping Bi, Jiafeng Guo, and W Bruce Croft. Learning a deep listwise context model for ranking refinement. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 135–144, 2018. 18

[13] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005. 9, 10

[14] Christopher J Burges, Robert Ragno, and Quoc V Le. Learning to rank with nonsmooth cost functions. In *Advances in neural information processing systems*, pages 193–200, 2007. 9, 10

[15] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010. 10

[16] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007. 9

[17] Hazel Doughty, Dima Damen, and Walterio Mayol-Cuevas. Who's better? who's best? pairwise deep ranking for skill determination. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6057–6066, 2018. 18

[18] Lanting Fang, Luu Anh Tuan, Siu Cheung Hui, and Lenan Wu. Syntactic based approach for grammar question retrieval. *Information Processing & Management*, 54(2):184–202, 2018. 17

[19] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003. 9

[20] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 10

[21] Mingsheng Fu, Hong Qu, Fan Li, and Yanjun Liu. A new deep neural network based learning to rank method for information retrieval. In *2018 IEEE International Conference on Information and Automation (ICIA)*, pages 1311–1316. IEEE, 2018. 17

[22] Jiafeng Guo, Yixing Fan, Liang Pang, Liu Yang, Qingyao Ai, Hamed Zamani, Chen Wu, W Bruce Croft, and Xueqi Cheng. A deep look into neural ranking models for information retrieval. *Information Processing & Management*, page 102067, 2019. 17

[23] Weiwei Guo, Xiaowei Liu, Sida Wang, Huiji Gao, Ananth Sankar, Zimeng Yang, Qi Guo, Liang Zhang, Bo Long, Bee-Chung Chen, et al. Detext: A deep text ranking framework with bert. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2509–2516, 2020. 17

[24] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. Support vector learning for ordinal regression. 1999. 9

[25] Kalervo Järvelin and Jaana Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *ACM SIGIR Forum*, volume 51, pages 243–250. ACM New York, NY, USA, 2017. 6

[26] Daniel Jurafsky and James H Martin. *Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing*. Upper Saddle River, NJ: Prentice Hall, 2008. 12

[27] Raaiha Humayun Kabir, Bisma Pervaiz, Tayyeba Muhammad Khan, Adnan Ul-Hasan, Raheel Nawaz, and Faisal Shafait. Deeprank: Adapting neural tensor networks for ranking the recommendations. In *Mediterranean Conference on Pattern Recognition and Artificial Intelligence*, pages 162–176. Springer, 2019. 17

[28] Bernhard Kratzwald, Anna Eigenmann, and Stefan Feuerriegel. Rankqa: Neural question answering with answer re-ranking. *arXiv preprint arXiv:1906.03008*, 2019. 17

[29] Pawan Kumar, Dhanajit Brahma, Harish Karnick, and Piyush Rai. Deep attentive ranking networks for learning to order sentences. In *AAAI*, pages 8115–8122, 2020. 17

[30] Hang Li. *Learning to rank for information retrieval and natural language processing*. Morgan & Claypool Publishers, 2014. 1, 7, 8, 9, 10, 44

[31] Ping Li, Qiang Wu, and Christopher J Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in neural information processing systems*, pages 897–904, 2008. 9

[32] Huafeng Liu, Jingxuan Wen, Liping Jing, and Jian Yu. Deep generative ranking for personalized recommendation. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 34–42, 2019. 17

[33] Tie-Yan Liu. *Learning to rank for information retrieval*. Springer Science & Business Media, 2011. 9, 10

[34] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008. 3

[35] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993. 10

[36] Ryan McDonald, Georgios-Ioannis Brokos, and Ion Androutsopoulos. Deep relevance ranking using enhanced document-query interactions. *arXiv preprint arXiv:1809.01682*, 2018. 18

[37] Van-Tu Nguyen and Anh-Cuong Le. Deep neural network-based models for ranking question-answering pairs in community question answering systems. In *International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making*, pages 179–190. Springer, 2018. 18

[38] Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Jingfang Xu, and Xueqi Cheng. Deeprank: A new deep architecture for relevance ranking in information retrieval. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 257–266, 2017. 18

[39] Rama Kumar Pasumarthi, Sebastian Bruch, Michael Bendersky, and Xuanhui Wang. Neural learning to rank using tensorflow ranking: A hands-on tutorial. In *Proceedings of the 2019 ACM SIGIR International Conference on Theory of Information Retrieval*, pages 253–254, 2019. 17

[40] Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. Tf-ranking: Scalable tensorflow library for learning-to-rank. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2970–2978, 2019. 17

[41] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010. 8

[42] Razieh Rahimi, Ali Montazeralghaem, and James Allan. Listwise neural ranking models. In *Proceedings of the 2019 ACM SIGIR International Conference on Theory of Information Retrieval*, pages 101–104, 2019. 17

[43] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *Nist Special Publication Sp*, 109:109, 1995. 3

[44] Abhishek Sharma. Listwise learning to rank with deep q-networks. *arXiv preprint arXiv:2002.07651*, 2020. 17

[45] Amnon Shashua and Anat Levin. Ranking with large margin principle: Two approaches. In *Advances in neural information processing systems*, pages 961–968, 2003. 9

[46] Manjira Sinha, Tirthankar Dasgupta, and Jatin Mandav. Ranking multiple choice question distractors using semantically informed neural networks. In *Proceedings of the 29th ACM*

International Conference on Information & Knowledge Management, pages 3329–3332, 2020. 17

[47] Baoyang Song. Deep neural network for learning to rank query-text pairs. *arXiv preprint arXiv:1802.08988*, 2018. 17

[48] Niek Tax, Sander Bockting, and Djoerd Hiemstra. A cross-benchmark comparison of 87 learning to rank methods. *Information processing & management*, 51(6):757–772, 2015. 5, 9

[49] Andrew Trotman, Antti Puurula, and Blake Burgess. Improvements to bm25 and language models examined. In *Proceedings of the 2014 Australasian Document Computing Symposium*, pages 58–65, 2014. 5

[50] Baiyang Wang and Diego Klabjan. An attention-based deep net for learning to rank. *arXiv preprint arXiv:1702.06106*, 2017. 18

[51] Yu Wang, Yuelin Wang, Jie Liu, and Zhuo Liu. A comprehensive survey of grammar error correction. *arXiv preprint arXiv:2005.06600*, 2020. 1

[52] Qiang Wu, Christopher JC Burges, Krysta M Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010. 9, 10

[53] Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, 2007. 9

[54] Yisong Yue, Thomas Finley, Filip Radlinski, and Thorsten Joachims. A support vector method for optimizing average precision. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 271–278, 2007. 9

[55] Hamid Zafar, Giulio Napolitano, and Jens Lehmann. Deep query ranking for question answering over knowledge bases. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 635–638. Springer, 2018. 18

[56] Hamed Zamani, Bhaskar Mitra, Xia Song, Nick Craswell, and Saurabh Tiwary. Neural ranking models with multiple document fields. In *Proceedings of the eleventh ACM international conference on web search and data mining*, pages 700–708, 2018. 18

Appendices

A Django Data Model Definitions and Equivalent SQL Statements

We define data models in scripts named `models.py`.

```
class ModelTopic(models.Model):
    topic_name = models.CharField(max_length=100, unique=True)
    topic_examples = models.TextField(null=True)
    topic_slug = models.SlugField()

    def __str__(self):
        return self.topic_name

class ModelQuiz(models.Model):
    QUIZ_NUMBER = ((1, 'Quiz 1'),(2, 'Quiz 2'),)
    quiz_topic = models.ForeignKey(ModelTopic, on_delete=models.CASCADE,
        related_name='modelquiztopic')
    quiz_number = models.IntegerField(choices=QUIZ_NUMBER)

    def __str__(self):
        return f'{self.quiz_number}:{self.quiz_topic}'

class ModelQuestionbank(models.Model):
    qb_question = models.TextField(unique=True)
    qb_answer = models.CharField(max_length=50)
    qb_choice1 = models.CharField(max_length=50)
    qb_choice2 = models.CharField(max_length=50)
    qb_choice3 = models.CharField(max_length=50)
    qb_topic = models.ForeignKey('appquiz.ModelTopic', on_delete=models.CASCADE,
        related_name='modelqbtopic')

    def __str__(self):
        return self.qb_question

class ModelQuestion(models.Model):
    q_question = models.TextField(unique=True)
    q_answer = models.CharField(max_length=50)
    q_choice1 = models.CharField(max_length=50)
    q_choice2 = models.CharField(max_length=50)
    q_choice3 = models.CharField(max_length=50)
    q_topic = models.ForeignKey(ModelTopic, on_delete=models.CASCADE,
        related_name='modelquestiontopic')
    q_quiz = models.ForeignKey(ModelQuiz, on_delete=models.CASCADE,
        related_name='modelquestionquiz')

    def __str__(self):
        return self.q_question
```

Equivalent SQL statements to create the tables in the database.

```
CREATE TABLE "appquiz_modeltopic" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "topic_name" varchar(100) NOT NULL UNIQUE,
    "topic_examples" text NULL,
    "topic_slug" varchar(50) NOT NULL
)

CREATE TABLE "appquiz_modelquiz" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "quiz_number" integer NOT NULL,
    "quiz_topic_id" integer NOT NULL REFERENCES
    "appquiz_modeltopic" ("id") DEFERRABLE INITIALLY DEFERRED
)

CREATE TABLE "appsearch_modelquestionbank" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "qb_question" text NOT NULL UNIQUE,
    "qb_answer" varchar(50) NOT NULL,
    "qb_choice1" varchar(50) NOT NULL,
    "qb_choice2" varchar(50) NOT NULL,
    "qb_choice3" varchar(50) NOT NULL,
    "qb_topic_id" integer NOT NULL REFERENCES "appquiz_modeltopic" ("id")
        DEFERRABLE INITIALLY DEFERRED
)

CREATE TABLE "appquiz_modelquestion" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "q_question" text NOT NULL UNIQUE,
    "q_answer" varchar(50)  NOT NULL,
    "q_choice1" varchar(50) NOT NULL,
    "q_choice2" varchar(50) NOT NULL,
    "q_choice3" varchar(50) NOT NULL,
    "q_quiz_id" integer  NOT NULL REFERENCES "appquiz_modelquiz" ("id")
        DEFERRABLE INITIALLY DEFERRED,
    "q_topic_id" integer NOT NULL REFERENCES "appquiz_modeltopic" ("id")
        DEFERRABLE INITIALLY DEFERRED
)
```

B Model Definition in Solr (Linear Model 2)

```
ms2 = {
    "store" : "feature_store2",
    "name" : "linear_model2",
    "class" : "org.apache.solr.ltr.model.LinearModel",
    "features" : [
        { "name": "ss_original_score" },
        { "name": "ss_pos" },
        { "name": "ss_pos_bigram" },
        { "name": "ss_pos_trigram" },
        { "name": "ss_parse_tree" },
        { "name": "before" },
        { "name": "before_last" },
        { "name": "before_last_pos" },
        { "name": "before_pos" },
        { "name": "before_pos_bigram" },
        { "name": "before_pos_trigram" },
        { "name": "before_parse_tree" },
        { "name": "after" },
        { "name": "after_first" },
        { "name": "after_first_pos" },
        { "name": "after_pos" },
        { "name": "after_pos_bigram" },
        { "name": "after_pos_trigram" },
        { "name": "after_parse_tree" },
        { "name": "ans" },
        { "name": "ans_first" },
        { "name": "ans_last" },
        { "name": "ans_pos" },
        { "name": "ans_first_pos" },
        { "name": "ans_last_pos" },
        { "name": "ans_is_first" },
        { "name": "ans_is_last" },
        { "name": "ans_length" },
    ],
    "params" : {
        "weights" : {
            "ss_original_score": 1.0,
            "ss_pos": 0.0,
            "ss_pos_bigram": 0.0,
            "ss_pos_trigram": 0.0,
            "ss_parse_tree": 0.0,
            "before": 0.0,
            "before_last": 0.0,
            "before_last_pos": 0.0,
            "before_pos": 0.0,
            "before_pos_bigram": 0.0,
            "before_pos_trigram": 0.0,
            "before_parse_tree": 0.0,
            "after": 0.0,
```

```
            "after_first": 0.0,
            "after_first_pos": 0.0,
            "after_pos": 0.0,
            "after_pos_bigram": 0.0,
            "after_pos_trigram": 0.0,
            "after_parse_tree": 0.0,
            "ans": 0.0,
            "ans_first": 0.0,
            "ans_last": 0.0,
            "ans_pos": 0.0,
            "ans_first_pos": 0.0,
            "ans_last_pos": 0.0,
            "ans_is_first": 0.0,
            "ans_is_last": 0.0,
            "ans_length": 0.0,
        }
    }
}
```

C Feature Definition in Solr (Model 2)

```
fs2 = [
    {
        "store": "feature_store2",
        "name": "ss_original_score",
        "class": "org.apache.solr.ltr.feature.OriginalScoreFeature",
        "params": {}
    },
    {
        "store": "feature_store2",
        "name": "ss_pos",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=ss_pos}${q2}" }
    },
    {
        "store": "feature_store2",
        "name": "ss_pos_bigram",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=ss_pos_bigram}${q3}" }
    },
    {
        "store": "feature_store2",
        "name": "ss_pos_trigram",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=ss_pos_trigram}${q4}" }
    },
    {
        "store": "feature_store2",
        "name": "ss_parse_tree",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=ss_parse_tree}${q5}" }
    },
    {
        "store": "feature_store2",
        "name": "before",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "q": "{!dismax qf=before}${q6}" }
    },
    {
        "store": "feature_store2",
        "name": "before_last",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=before_last}${q7}"] }
    },
    {
        "store": "feature_store2",
        "name": "before_last_pos",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=before_last_pos}${q8}"] }
    },
```

```
{
    "store": "feature_store2",
    "name": "before_pos",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=before_pos}${q9}" }
},
{
    "store": "feature_store2",
    "name": "before_pos_bigram",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=before_pos_bigram}${q10}" }
},
{
    "store": "feature_store2",
    "name": "before_pos_trigram",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=before_pos_trigram}${q11}" }
},
{
    "store": "feature_store2",
    "name": "before_parse_tree",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=before_parse_tree}${q12}" }
},
{
    "store": "feature_store2",
    "name": "after",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=after}${q13}" }
},
{
    "store": "feature_store2",
    "name": "after_first",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=after_first}${q14}"] }
},
{
    "store": "feature_store2",
    "name": "after_first_pos",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=after_first_pos}${q15}"] }
},
{
    "store": "feature_store2",
    "name": "after_pos",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=after_pos}${q16}" }
},
```

```
{
    "store": "feature_store2",
    "name": "after_pos_bigram",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=after_pos_bigram}${q17}" }
},
{
    "store": "feature_store2",
    "name": "after_pos_trigram",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=after_pos_trigram}${q18}" }
},
{
    "store": "feature_store2",
    "name": "after_parse_tree",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=after_parse_tree}${q19}" }
},
{
    "store": "feature_store2",
    "name": "ans",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "q": "{!dismax qf=ans}${q20}" }
},
{
    "store": "feature_store2",
    "name": "ans_first",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=ans_first}${q21}"] }
},
{
    "store": "feature_store2",
    "name": "ans_last",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=ans_last}${q22}"] }
},
{
    "store": "feature_store2",
    "name": "ans_pos",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=ans_pos}${q23}"] }
},
{
    "store": "feature_store2",
    "name": "ans_first_pos",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": { "fq": ["{!term f=ans_first_pos}${q24}"] }
},
```

```
    {
        "store": "feature_store2",
        "name": "ans_last_pos",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=ans_last_pos}${q25}"] }
    },
    {
        "store": "feature_store2",
        "name": "ans_is_first",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=ans_is_first}${q26}"] }
    },
    {
        "store": "feature_store2",
        "name": "ans_is_last",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=ans_is_last}${q27}"] }
    },
    {
        "store": "feature_store2",
        "name": "ans_length",
        "class": "org.apache.solr.ltr.feature.SolrFeature",
        "params": { "fq": ["{!term f=ans_length}${q28}"] }
    },
]
```

D Sample of Training Dataset (Model 1)

```
0 qid:851 1:5.059935 2:2.151426 3:1.9510254 4:1.143509 5:1.5566376 # docid:137
0 qid:851 1:5.053303 2:0.60843366 3:0.52210945 4:0.0 5:1.8449144 # docid:518
1 qid:851 1:5.040501 2:3.5248418 3:4.84146 4:3.209079 5:4.2115293 # docid:57
0 qid:851 1:5.040501 2:2.12029 3:2.8691504 4:1.1933943 5:1.9023052 # docid:33
0 qid:851 1:4.9899435 2:1.7796078 3:1.5263851 4:0.0 5:0.9472734 # docid:480
0 qid:851 1:4.9661746 2:2.3648534 3:1.821882 4:0.0 5:0.8461648 # docid:754
0 qid:851 1:4.9634566 2:1.6855067 3:0.0 4:0.0 5:0.88083315 # docid:65
0 qid:851 1:4.8534226 2:3.476744 3:2.8691504 4:1.1933943 5:4.635674 # docid:142
1 qid:851 1:4.8534226 2:3.3056526 3:5.3912125 4:3.5659509 5:3.171269 # docid:442
1 qid:851 1:4.8534226 2:3.3527045 3:5.3912125 4:3.5659509 5:3.5252767 # docid:821
1 qid:851 1:4.8189144 2:3.1648488 3:5.093672 4:3.7286108 5:3.467321 # docid:382
0 qid:851 1:4.8189144 2:2.1546311 3:1.9756243 4:0.0 5:1.16054 # docid:48
0 qid:851 1:4.7561555 2:1.7804202 3:1.5957444 4:0.0 5:0.5737468 # docid:223
0 qid:851 1:4.742285 2:2.2763608 3:3.4378977 4:1.3730977 5:1.8817657 # docid:40
0 qid:851 1:4.742285 2:2.4928927 3:2.6907158 4:0.0 5:0.1853511 # docid:263
0 qid:851 1:4.6737213 2:2.3195326 3:2.108052 4:0.0 5:0.9151481 # docid:512
0 qid:851 1:4.6737213 2:2.3195326 3:2.108052 4:0.0 5:0.9151481 # docid:785
0 qid:851 1:4.659534 2:2.0089843 3:1.905539 4:0.0 5:1.1529775 # docid:680
1 qid:851 1:4.5521617 2:3.5323648 3:6.3624477 4:3.906819 5:4.1280117 # docid:430
1 qid:851 1:4.5521617 2:3.6665218 3:9.595817 4:12.6099615 5:3.8055003 # docid:552
0 qid:851 1:4.510627 2:1.9575185 3:2.2768698 4:0.0 5:0.86480474 # docid:436
0 qid:851 1:4.510627 2:2.0931325 3:1.7758176 4:0.0 5:3.7886531 # docid:600
0 qid:851 1:4.510627 2:2.0931325 3:1.821882 4:0.0 5:3.017345 # docid:393
0 qid:851 1:4.510627 2:3.758675 3:1.821882 4:0.0 5:2.5912304 # docid:631
0 qid:851 1:4.510627 2:2.2069368 3:2.285994 4:0.0 5:0.48735827 # docid:795
0 qid:852 1:12.408153 2:2.0998564 3:6.132431 4:7.24146 5:4.700104 # docid:7
0 qid:852 1:12.007708 2:3.8993492 3:1.4221642 4:0.0 5:5.93133 # docid:243
0 qid:852 1:11.521721 2:3.4803436 3:5.469503 4:6.8014345 5:0.98569524 # docid:750
0 qid:852 1:10.558846 2:4.133375 3:1.7783868 4:0.0 5:7.1326222 # docid:245
0 qid:852 1:9.661868 2:0.8618351 3:2.7154598 4:0.0 5:1.7104149 # docid:742
0 qid:852 1:8.71882 2:1.1017758 3:0.5012309 4:0.0 5:2.5034924 # docid:46
0 qid:852 1:8.664849 2:4.396191 3:5.9397316 4:0.0 5:8.498297 # docid:539
0 qid:852 1:8.664849 2:4.396191 3:5.9397316 4:0.0 5:8.498297 # docid:762
0 qid:852 1:8.227466 2:2.9546523 3:5.806029 4:5.9925985 5:1.352556 # docid:719
0 qid:852 1:8.113874 2:4.658458 3:2.604295 4:0.0 5:4.8126626 # docid:280
0 qid:852 1:7.933031 2:2.8886724 3:5.8901215 4:3.770838 5:1.9831146 # docid:806
0 qid:852 1:7.7095804 2:3.8709106 3:1.2829962 4:0.0 5:4.993978 # docid:586
1 qid:852 1:7.6589427 2:3.1288488 3:5.469503 4:6.8014345 5:4.473129 # docid:321
0 qid:852 1:7.2553177 2:3.0845537 3:2.4263391 4:0.0 5:6.1597357 # docid:633
0 qid:852 1:7.1344137 2:3.2828927 3:0.0 4:0.0 5:5.0631933 # docid:737
0 qid:852 1:7.020779 2:3.576986 3:3.29223 4:0.0 5:3.781301 # docid:738
0 qid:852 1:6.884693 2:5.331807 3:4.159304 4:0.0 5:5.445375 # docid:441
0 qid:852 1:6.486387 2:2.7876792 3:1.5927768 4:0.0 5:5.158302 # docid:193
0 qid:852 1:6.3917713 2:1.1442199 3:0.70514715 4:0.0 5:1.5839087 # docid:525
0 qid:852 1:6.3917713 2:1.1442199 3:0.70514715 4:0.0 5:1.5839087 # docid:775
0 qid:852 1:6.3917713 2:3.5025487 3:3.6958408 4:0.0 5:1.0158817 # docid:814
```

E Sample of Training Dataset (Model 2)

```
0 qid:851 1:10.405931 2:4.230867 3:6.8068247 4:4.1970816 5:4.853129 6:2.2070153 7:0.0 8:0.0
9:0.7036993 10:0.0 11:0.0 12:0.6572361 13:5.3674583 14:1.0 15:1.0 16:1.1569695 17:1.0780797
18:0.0 19:1.4527704 20:0.0 21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:0.0 # docid:554
0 qid:851 1:9.783033 2:4.257559 3:4.876296 4:0.0 5:4.287144 6:1.4057359 7:0.0 8:0.0
9:2.5579796 10:2.4146028 11:0.0 12:1.5384526 13:0.0 14:0.0 15:0.0 16:0.0 17:0.0 18:0.0
19:0.0 20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0 28:1.0 # docid:236
3 qid:851 1:9.324368 2:4.7786293 3:11.464623 4:15.891939 5:9.098976 6:6.240824 7:1.0 8:1.0
9:6.0062943 10:8.046046 11:5.6499414 12:9.690812 13:1.4804273 14:1.0 15:1.0 16:1.4822142
17:1.5018107 18:0.0 19:4.314536 20:4.701566 21:1.0 22:1.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0
28:1.0 # docid:6
0 qid:851 1:8.860167 2:3.5016544 3:4.3270397 4:0.0 5:7.1936283 6:2.2070153 7:0.0 8:0.0
9:0.7036993 10:0.0 11:0.0 12:0.6572361 13:0.0 14:0.0 15:0.0 16:0.0 17:0.0 18:0.0 19:0.0 20:0.0
21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:0.0 # docid:632
0 qid:851 1:8.347566 2:2.8728871 3:4.5456753 4:4.754389 5:2.2510476 6:0.0 7:0.0 8:0.0
9:0.7036993 10:0.0 11:0.0 12:0.6572361 13:1.1559619 14:0.0 15:0.0 16:1.1569695 17:1.0780797
18:0.0 19:1.4527704 20:0.0 21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:0.0 # docid:633
0 qid:851 1:6.79864 2:2.8062017 3:2.7386246 4:0.0 5:1.1876667 6:0.0 7:0.0 8:0.0 9:0.0 10:0.0
11:0.0 12:0.0 13:0.0 14:0.0 15:0.0 16:0.0 17:0.0 18:0.0 19:0.0 20:0.0 21:0.0 22:0.0 23:1.0
24:1.0 25:1.0 26:0.0 27:0.0 28:1.0 # docid:284
2 qid:851 1:6.775952 2:3.3060672 3:4.535988 4:1.9107668 5:2.1581402 6:0.0 7:0.0 8:0.0 9:0.0
10:0.0 11:0.0 12:0.0 13:1.1559619 14:1.0 15:1.0 16:1.1569695 17:1.0780797 18:0.0 19:1.2069812
20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0 28:1.0 # docid:71
0 qid:851 1:6.766583 2:2.656864 3:0.8882673 4:0.0 5:0.56869006 6:1.8546312 7:0.0 8:0.0
9:0.5912134 10:0.0 11:0.0 12:0.5160939 13:5.3674583 14:1.0 15:1.0 16:1.1569695 17:1.0780797
18:0.0 19:1.7998906 20:0.0 21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:1.0 # docid:511
0 qid:851 1:6.685004 2:2.4078975 3:2.421287 4:0.0 5:0.70507956 6:5.520366 7:0.0 8:0.0
9:2.9105182 10:2.837012 11:0.0 12:0.5160939 13:0.0 14:0.0 15:0.0 16:0.0 17:0.0 18:0.0 19:0.0
20:0.0 21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:1.0 # docid:279
2 qid:851 1:6.026882 2:4.616071 3:3.969133 4:1.6344748 5:2.1448812 6:6.401779 7:0.0 8:0.0
9:3.3757627 10:3.4385498 11:0.0 12:1.723516 13:1.1559619 14:1.0 15:1.0 16:1.1569695
17:1.0780797 18:0.0 19:1.3185191 20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0
28:1.0 # docid:142
0 qid:851 1:5.876196 2:2.5187464 3:1.0264364 4:0.0 5:0.0 6:1.4057359 7:0.0 8:0.0 9:0.44799113
10:0.0 11:0.0 12:0.0 13:5.3674583 14:0.0 15:0.0 16:1.3320849 17:1.0780797 18:0.0 19:0.0 20:0.0
21:0.0 22:0.0 23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:0.0 # docid:710
2 qid:851 1:5.7111998 2:2.9784222 3:3.7357104 4:1.524272 5:5.387085 6:5.520366 7:0.0 8:0.0
9:2.9105182 10:2.837012 11:0.0 12:1.5384526 13:1.1559619 14:1.0 15:1.0 16:1.1569695 17:1.0780797
18:0.0 19:1.4527704 20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0 28:1.0 # docid:45
2 qid:851 1:5.7111998 2:4.2775846 3:10.116165 4:13.748943 5:5.0411496 6:5.520366 7:0.0 8:1.0
9:6.0062943 10:8.046046 11:5.6499414 12:9.690812 13:1.1559619 14:1.0 15:1.0 16:1.1569695
17:1.0780797 18:0.0 19:1.7998906 20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0
28:1.0 # docid:107
2 qid:851 1:5.426942 2:3.9937203 3:5.1644645 4:3.0260859 5:4.126788 6:4.852291 7:0.0 8:1.0
9:5.2787776 10:2.4146028 11:0.0 12:4.6766353 13:1.1559619 14:1.0 15:1.0 16:0.7158682 17:0.0
18:0.0 19:0.0 20:0.0 21:0.0 22:0.0 23:1.0 24:1.0 25:1.0 26:0.0 27:0.0 28:1.0 # docid:57
0 qid:851 1:5.426942 2:2.588736 3:2.321512 4:0.0 5:1.40591 6:0.0 7:0.0 8:0.0 9:0.0 10:0.0
11:0.0 12:0.0 13:0.0 14:0.0 15:0.0 16:0.44110143 17:0.0 18:0.0 19:0.0 20:0.0 21:0.0 22:0.0
23:0.0 24:0.0 25:0.0 26:0.0 27:0.0 28:1.0 # docid:785
```