Elasticsearch Learning to Rank Documentation

Doug Turnbull, Erik Berhardson, David Causse, Daniel Worley

Feb 24, 2020

Contents

1 Get started **3**

2 Installing **5**

3 HEEELP! **7**

4 Contents **9**

4.1 Core Concepts 9
4.2 How does the plugin fit in? 12
4.3 Working with Features 13
4.4 Feature Engineering 18
4.5 Logging Feature Scores 19
4.6 Uploading A Trained Model 26
4.7 Searching with LTR 30
4.8 On XPack Support (Security) 32
4.9 Advanced Functionality 34

5 Indices and tables **39**

Learning to Rank applies machine learning to relevance ranking. The Elasticsearch Learning to Rank plugin (Elasticsearch LTR) gives you tools to train and use ranking models in Elasticsearch. This plugin powers search at places like Wikimedia Foundation and Snagajob.

CHAPTER 1

Get started

- Want a quickstart? Check out the demo.
- Brand new to learning to rank? head to *Core Concepts*.
- Otherwise, start with *How does the plugin fit in?*

CHAPTER 2

Installing

Pre-built versions can be found here. Want a build for an ES version? Follow the instructions in the README for building or create an issue. Once you've found a version compatible with your Elasticsearch, you'd run a command such as:

```
./bin/elasticsearch-plugin install \
http://es-learn-to-rank.labs.o19s.com/ltr-1.1.0-es6.5.4.zip
```

(It's expected you'll confirm some security exceptions, you can pass -b to elasticsearch-plugin to automatically install)

Are you using x-pack security in your cluster? we got you covered, check *On XPack Support (Security)* for specific configuration details.

CHAPTER 3

HEEELP!

The plugin and guide was built by the search relevance consultants at OpenSource Connections in partnership with the Wikimedia Foundation and Snagajob Engineering. Please contact OpenSource Connections or create an issue if you have any questions or feedback.

CHAPTER 4

Contents

4.1 Core Concepts

Welcome! You're here if you're interested in adding machine learning ranking capabilities to your Elasticsearch system. This guidebook is intended for Elasticsearch developers and data scientists.

4.1.1 What is Learning to Rank?

Learning to Rank (LTR) applies machine learning to search relevance ranking. How does relevance ranking differ from other machine learning problems? Regression is one classic machine learning problem. In *regression*, you're attempting to predict a variable (such as a stock price) as a function of known information (such as number of company employees, the company's revenue, etc). In these cases, you're building a function, say *f*, that can take what's known (*numEmployees*, *revenue*), and have *f* output an approximate stock price.

Classification is another machine learning problem. With classification, our function *f*, would classify our company into several categories. For example, profitable or not profitable. Or perhaps whether or not the company is evading taxes.

In Learning to Rank, the function *f* we want to learn does not make a direct prediction. Rather it's used for ranking documents. We want a function *f* that comes as close as possible to our user's sense of the ideal ordering of documents dependent on a query. The value output by *f* itself has no meaning (it's not a stock price or a category). It's more a prediction of a users' sense of the relative usefulnes of a document given a query.

Here, we'll briefly walk through the 10,000 meter view of Learning to Rank. For more information, we recommend blog articles How is Search Different From Other Machine Learning Problems? and What is Learning to Rank?.

4.1.2 Judgments: expression of the ideal ordering

Judgment lists, sometimes referred to as "golden sets" grade individual search results for a keyword search. For example, our demo uses TheMovieDB. When users search for "Rambo" we can indicate which movies ought to come back for "Rambo" based on our user's expectations of search.

For example, we know these movies are very relevant:

- First Blood
- Rambo

We know these sequels are fairly relevant, but not exactly relevant:

- Rambo III
- Rambo First Blood, Part II

Some movies that star Sylvester Stallone are only tangentially relevant:

- Rocky
- Cobra

And of course many movies are not even close:

- Bambi
- First Daughter

Judgment lists apply "grades" to documents for a keyword, this helps establish the ideal ordering for a given keyword. For example, if we grade documents from 0-4, where 4 is exactly relevant. The above would turn into the judgment list:

```
grade,keywords,movie
4,Rambo,First Blood     # Exactly Relevant
4,Rambo,Rambo
3,Rambo,Rambo III       # Fairly Relevant
3,Rambo,Rambo First Blood Part II
2,Rambo,Rocky           # Tangentially Relevant
2,Rambo,Cobra
0,Rambo,Bambi           # Not even close...
0,Rambo,First Daughter
```

A search system that approximates this ordering for the search query "Rambo", and all our other test queries, can said to be performing well. Metrics such as NDCG and ERR evaluate a query's actual ordering vs the ideal judgment list.

Our ranking function f needs to rank search results as close as possible to our judgment lists. We want to maximize quality metrics such as ERR or NDCG over the broadest number of queries in our training set. When we do this, with accurate judgments, we work to return results listings that will be maximally useful to users.

4.1.3 Features: the raw material of relevance

Above in the example of a stock market predictor, our ranking function f used variables such as the number of employees, revenue, etc to arrive at a predicted stock price. These are *features* of the company. Here our ranking function must do the same: using features that describe the document, the query, or some relationship between the document and the query (such as query keyword's TF*IDF score in a field).

Features for movies, for example, might include:

- Whether/how much the search keywords match the title field (let's call this *titleScore*)
- Whether/how much the search keywords match the description field (*descScore*)
- The popularity of the movie (*popularity*)
- The rating of the movie (*rating*)
- How many keywords are used during search? (*numKeywords*)

Our ranking function then becomes `f(titleScore, descScore, popularity, rating, numKeywords)`. We hope whatever method we use to create a ranking function can utilize these features to maximize the likelihood of search results being useful for users. For example, it seems intuitive in the "Rambo" use case that *titleScore* matters quite a bit. But one top movie "First Blood" probably only mentions the keyword Rambo in the description. So in this case *descScore* comes into play. Also *popularity*/*rating* might help determine which movies are "sequels" and which are the originals. We might learn this feature doesn't work well in this regard, and introduce a new feature *isSequel* that our ranking function could use to make better ranking decisions.

Selecting and experimenting with features is a core piece of learning to rank. Good judgments with poor features that don't help predict patterns in the predicted grades and won't create a good search experience. Just like any other machine learning problem: garbage in-garbage out!

For more on the art of creating features for search, check out the book Relevant Search by Doug Turnbull and John Berryman.

4.1.4 Logging features: completing the training set

With a set of features we want to use, we need to annotate the judgment list above with values of each feature. This data will be used once training commences.

In other words, we need to transfer:

```
grade,keywords,movie
4,Rambo,First Blood
4,Rambo,Rambo
3,Rambo,Rambo III
...
```

into:

```
grade,keywords,movie,titleScore,descScore,popularity,...
4,Rambo,First Blood,0.0,21.5,100,...
4,Rambo,Rambo,42.5,21.5,95,...
3,Rambo,Rambo III,53.1,40.1,50,...
```

(here titleScore is the relevance score of "Rambo" for title field in document "First Blood", and so on)

Many learning to rank models are familiar with a file format introduced by SVM Rank, an early learning to rank method. Queries are given ids, and the actual document identifier can be removed for the training process. Features in this file format are labeled with ordinals starting at 1. For the above example, we'd have the file format:

```
4   qid:1   1:0.0   2:21.5  3:100,...
4   qid:1   1:42.5  2:21.5  3:95,...
3   qid:1   1:53.1  2:40.1  3:50,...
...
```

In actual systems, you might log these values after the fact, gathering them to annotate a judgment list with feature values. In others the judgment list might come from user analytics, so it may be logged as the user interacts with the search application. More on this when we cover in *Logging Feature Scores*.

With judgments and features in place, the next decision is to arrive at the ranking function. There's a number of models available for ranking, with their own intricate pros and cons. Each one attempts to use the features to minimize the error in the ranking function. Each has its own notion of what "error" means in a ranking system. (for more read this blog article)

Generally speaking there's a couple of families of models:

- Tree-based models (LambdaMART, MART, Random Forests): These models tend to be most accurate in general. They're large and complex models that can be fairly expensive to train. RankLib and xgboost both focus on tree-based models.
- SVM based models (SVMRank): Less accurate, but cheap to train. See SVMRank.
- Linear models: Performing a basic linear regression over the judgment list. Tends to not be useful outside of toy examples. See this blog article

As with any technology, model selection can be as much about what a team has experience with, not just with what performs best.

4.1.5 Testing: is our model any good?

Our judgment lists can't cover every user query our model will encounter out in the wild. So it's important to throw our model curveballs, to see how well it can "think for itself." Or as machine learning folks say: can the model generalize beyond the training data? A model that cannot generalize beyond training data is *overfit* to the training data, and not as useful.

To avoid overfitting, you hide some of your judgment lists from the training process. You then use these to test your model. This side data set is known as the "test set." When evaluating models you'll hear about statistics such as "test NDCG" vs "training NDCG." The former reflects how your model will perform against scenarios it hasn't seen before. You hope as you train, your test search quality metrics continue to reflect high quality search. Further: after you deploy a model, you'll want to try out newer/more recent judgment lists to see if your model might be overfit to seasonal/temporal situations.

4.1.6 Real World Concerns

Now that you're oriented, the rest of this guide builds on this context to point out how to use the learning to rank plugin. But before we move on, we want to point out some crucial decisions everyone encounters in building learning to rank systems. We invite you to watch a talk with Doug Turnbull and Jason Kowalewski where the painful lessons of real learning to rank systems are brought out.

- How do you get accurate judgment lists that reflect your users real sense of search quality?
- What metrics best measure whether search results are useful to users?
- What infrastructure do you need to collect and log user behavior and features?
- How will you detect when/whether your model needs to be retrained?
- How will you A/B test your model vs your current solution? What KPIs will determine success in your search system.

Of course, please don't hesitate to seek out the services of *OpenSource Connection <http://opensourceconnections.com/services>*, sponsors of this plugin, as we work with organizations to explore these issues.

Next up, see how exactly this plugin's functionality fits into a learning to rank system: *How does the plugin fit in?*.

4.2 How does the plugin fit in?

In *Core Concepts* we mentioned a couple of activities you undertake when implementing learning to rank:

1. Judgment List Development
2. Feature Engineering

3. Logging features into the judgment list to create a training set
4. Training and testing models
5. Deploying and using models when searching

How does Elasticsearch LTR fit into this process?

4.2.1 What the plugin does

This plugin gives you building blocks to develop and use learning to rank models. It lets you develop query-dependent features and store them in Elasticsearch. After storing a set of features, you can log them for documents returned in search results to aid in offline model development.

Then other tools take over. With a logged set of features for documents, you join data with your judgment lists you've developed on your own. You've now got a training set you can use to test/train ranking models. Using of a tool like Ranklib or XGboost, you'll hopefully arrive at a satisfactory model.

With a ranking model, you turn back to the plugin. You upload the model and give it a name. The model is associated with the set of features used to generate the training data. You can then search with the model, using a custom Elasticsearch Query DSL primitive that executes the model. Hopefully this lets you deliver better search to users!

4.2.2 What the plugin is NOT

The plugin does not help with judgment list creation. This is work you must do and can be very domain specific. Wikimedia Foundation wrote a great article on how they arrive at judgment lists for people searching articles. Other domains such as e-commerce might be more conversion focused. Yet others might involve human relevance judges – either experts at your company or mechanical turk.

The plugin does not train or test models. This also happens offline in tools appropriate to the task. Instead the plugin uses models generated by XGboost and Ranklib libraries. Training and testing models is CPU intensive task that, involving data scientist supervision and offline testing. Most organizations want some data science supervision on model development. And you would not want this running in your production Elasticsearch cluster!

The rest of this guide is dedicated to walking you through how the plugin works to get you there. Continue on to *Working with Features*.

4.3 Working with Features

In *Core Concepts* , we mentioned the main roles you undertake building a learning to rank system. In *How does the plugin fit in?* we discussed at a high level what this plugin does to help you use Elasticsearch as a learning to rank system.

This section covers the functionality built into the Elasticsearch LTR plugin to build & upload features with the plugin.

4.3.1 What is a feature in Elasticsearch LTR?

Elasticsearch LTR features correspond to Elasticsearch queries. The score of an Elasticsearch query, when run using the user's search terms (and other parameters), are the values you use in your training set.

Obvious features might include traditional search queries, like a simple "match" query on title:

```
{
    "query": {
        "match": {
            "title": "{{keywords}}"
        }
    }
}
```

Of course, properties of documents such as popularity can also be a feature. Function score queries can help access these values. For example, to access the average user rating of a movie:

```
{
    "query": {
        "function_score": {
            "functions": {
                "field": "vote_average"
            },
            "query": {
                "match_all": {}
            }
        }
    }
}
```

One could also imagine a query based on the user's location:

```
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "200km",
                    "pin.location" : {
                        "lat" : {{users_lat}},
                        "lon" : {{users_lon}}
                    }
                }
            }
        }
    }
}
```

Similar to how you would develop queries like these to manually improve search relevance, the ranking function `f` you're training also combines these queries mathematically to arrive at a relevance score.

4.3.2 Features are Mustache Templated Elasticsearch Queries

You'll notice the `{{keywords}}`, `{{users_lat}}`, and `{{users_lon}}` above. This syntax is the mustache templating system used in other parts of Elasticsearch. This lets you inject various query or user-specific variables into the search template. Perhaps information about the user for personalization? Or the location of the searcher's phone?

For now, we'll simply focus on typical keyword searches.

4.3.3 Uploading and Naming Features

Elasticsearch LTR gives you an interface for creating and manipulating features. Once created, then you can have access to a set of feature for logging. Logged features when combined with your judgment list, can be trained into a model. Finally, that model can then be uploaded to Elasticsearch LTR and executed as a search.

Let's look how to work with sets of features.

4.3.4 Initialize the default feature store

A *feature store* corresponds to an Elasticsearch index used to store metadata about the features and models. Typically, one feature store corresponds to a major search site/implementation. For example, wikipedia vs wikitravel

For most use cases, you can simply get by with the single, default feature store and never think about feature stores ever again. This needs to be initialized the first time you use Elasticsearch Learning to Rank:

```
PUT _ltr
```

You can restart from scratch by deleting the default feature store:

```
DELETE _ltr
```

(WARNING this will blow everything away, use with caution!)

In the rest of this guide, we'll work with the default feature store.

4.3.5 Features and feature sets

Feature sets are where the action really happens in Elasticsearch LTR.

A *feature set* is a set of features that has been grouped together for logging & model evaluation. You'll refer to feature sets when you want to log multiple feature values for offline training. You'll also create a model from a feature set, copying the feature set into model.

4.3.6 Create a feature set

You can create a feature set simply by using a POST. To create it, you give a feature set a name and optionally a list of features:

```
POST _ltr/_featureset/more_movie_features
{
   "featureset": {
        "features": [
            {
                "name": "title_query",
                "params": [
                    "keywords"
                ],
                "template_language": "mustache",
                "template": {
                    "match": {
                        "title": "{{keywords}}"
                    }
                }
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "name": "title_query_boost",
                "params": [
                    "some_multiplier"
                ],
                "template_language": "derived_expressions",
                "template": "title_query * some_multiplier"
            },
            {
                "name": "custom_title_query_boost",
                "params": [
                    "some_multiplier"
                ],
                "template_language": "script_feature",
                "template": {
                    "lang": "painless",
                    "source": "params.feature_vector.get('title_query') *␣
↪(long)params.some_multiplier",
                    "params": {
                        "some_multiplier": "some_multiplier"
                    }
                }
            }
        ]
    }
}
```

4.3.7 Feature set CRUD

Fetching a feature set works as you'd expect:

```
GET _ltr/_featureset/more_movie_features
```

You can list all your feature sets:

```
GET _ltr/_featureset
```

Or filter by prefix in case you have many feature sets:

```
GET _ltr/_featureset?prefix=mor
```

You can also delete a featureset to start over:

```
DELETE _ltr/_featurset/more_movie_features
```

4.3.8 Validating features

When adding features, we recommend sanity checking that the features work as expected. Adding a "validation" block to your feature creation let's Elasticsearch LTR run the query before adding it. If you don't run this validation, you may find out only much later that the query, while valid JSON, was a malformed Elasticsearch query. You can imagine, batching dozens of features to log, only to have one of them fail in production can be quite annoying!

To run validation, you simply specify test parameters and a test index to run:

```
"validation": {
   "params": {
       "keywords": "rambo"
   },
   "index": "tmdb"
},
```

Place this alongside the feature set. You'll see below we have a malformed `match` query. The example below should return an error that validation failed. An indicator you should take a closer look at the query:

```
{
   "validation": {
     "params": {
         "keywords": "rambo"
     },
     "index": "tmdb"
    },
    "featureset": {
        "features": [
            {
                "name": "title_query",
                "params": [
                    "keywords"
                ],
                "template_language": "mustache",
                "template": {
                    "mooch": {
                        "title": "{{keywords}}"
                    }
                }
            }
        ]
    }
}
```

4.3.9 Adding to an existing feature set

Of course you may not know upfront what features could be useful. You may wish to append a new feature later for logging and model evaluation. For example, creating the *user_rating* feature, we could create it using the feature set append API, like below:

```
POST /_ltr/_featureset/my_featureset/_addfeatures
{
    "features": [{
        "name": "user_rating",
        "params": [],
        "template_language": "mustache",
        "template" : {
            "function_score": {
                "functions": {
                    "field": "vote_average"
                },
                "query": {
                    "match_all": {}
                }
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }]
}
```

4.3.10 Feature Names are Unique

Because some model training libraries refer to features by name, Elasticsearch LTR enforces unique names for each features. In the example above, we could not add a new *user_rating* feature without creating an error.

4.3.11 Feature Sets are Lists

You'll notice we *appended* to the feature set. Feature sets perhaps ought to be really called "lists." Each feature has an ordinal (its place in the list) in addition to a name. Some LTR training applications, such as Ranklib, refer to a feature by ordinal (the "1st" feature, the "2nd" feature). Others more conveniently refer to the name. So you may need both/either. You'll see that when features are logged, they give you a list of features back to preserve the ordinal.

4.3.12 But wait there's more

Feature engineering is a complex part of Elasticsearch Learning to Rank, and additional features (such as features that can be derived from other features) are listed in *Advanced Functionality*.

Next-up, we'll talk about some specific use cases you'll run into when *Feature Engineering*.

4.4 Feature Engineering

You've seen how to add features to feature sets. We want to show you how to address common feature engineering tasks that come up when developing a learning to rank solution.

4.4.1 Getting Raw Term Statistics

Many learning to rank solutions use raw term statistics in training. For example, the total term frequency for a term, the document frequency, and other statistics. Luckily, Elasticsearch LTR comes with a query primitive, `match_explorer`, that extracts these statistics for you for a set of terms. In it's simplest form, `match_explorer` you specify a statistic you're interested in and a match you'd like to explore. For example:

```
POST tmdb/_search
{
    "query": {
        "match_explorer": {
            "type": "max_raw_df",
            "query": {
                "match": {
                    "title": "rambo rocky"
                }
            }
        }
    }
}
```

This query returns the highest document frequency between the two terms.

A large number of statistics are available. The `type` parameter can be prepended with the operation to be performed across terms for the statistic `max`, `min`, `sum`, and `stddev`.

The statistics available include:

- `raw_df` – the direct document frequency for a term. So if rambo occurs in 3 movie titles, this is 3.
- `classic_idf` – the IDF calculation of the classic similarity `log((NUM_DOCS+1)/(raw_df+1)) + 1`.
- `raw_ttf` – the total term frequency for the term across the index. So if rambo is mentioned a total of 100 times in the overview field, this would be 100.

Putting the operation and the statistic together, you can see some examples. To get stddev of classic_idf, you would write `stddev_classic_idf`. To get the minimum total term frequency, you'd write `min_raw_ttf`.

Finally a special stat exists for just counting the number of search terms. That stat is `unique_terms_count`.

4.4.2 Document-specific features

Another common case in learning to rank is features such as popularity or recency, tied only to the document. Elasticsearch's `function_score` query has the functionality you need to pull this data out. You already saw an example when adding features in the last section:

```
{
    "query": {
        "function_score": {
            "functions": [{
                "field_value_factor": {
                    "field": "vote_average",
                    "missing": 0
                }
            }],
            "query": {
                "match_all": {}
            }
        }
    }
}
```

The score for this query corresponds to the value of the `vote_average` field.

4.4.3 Your index may drift

If you have an index that updates regularly, trends that held true today, may not hold true tomorrow! On an e-commerce store, sandals might be very popular in the summer, but impossible to find in the winter. Features that drive purchases for one time period, may not hold true for another. It's always a good idea to monitor your model's performance regularly, retrain as needed.

Next up, we discuss the all-important task of logging features in *Logging Feature Scores*.

4.5 Logging Feature Scores

To train a model, you need to log feature values. This is a major component of the learning to rank plugin: as users search, we log feature values from our feature sets so we can then train. Then we can discover models that work well

to predict relevance with that set of features.

4.5.1 Sltr Query

The `sltr` query is the primary way features are run and models are evaluated. When logging, we'll just use an `sltr` query for executing every feature-query to retrieve the scores of features.

For the sake of discussing logging, let's say we created a feature set like so that works with the TMDB data set from the demo:

```
PUT _ltr/_featureset/more_movie_features
{
    "name": "more_movie_features",
    "features": [
        {
            "name": "body_query",
            "params": [
                "keywords"
                ],
            "template": {
                "match": {
                    "overview": "{{keywords}}"
                }
            }
        },
        {
            "name": "title_query",
            "params": [
                "keywords"
            ],
            "template": {
                "match": {
                    "title": "{{keywords}}"
                }
            }
        }
    ]
}
```

Next, let's see how to log this feature set in a couple common use cases.

4.5.2 Joining feature values with a judgment list

Let's assume, in the simplest case, we have a judgment list already. We simply want to join feature values for each keyword/document pair to form a complete training set. For example, assume we have experts in our company, and they've arrived at this judgment list:

```
grade,keywords,docId
4,rambo,7555
3,rambo,1370
3,rambo,1369
4,rocky,4241
```

We want to get feature values for all documents that have judgment for each search term, one search term at a time. If we start with "rambo", we can create a filter for the ids associated with the "rambo" search:

```
{
    "filter": [
        {"terms": {
                "_id": ["7555", "1370", "1369"]
        }}
    ]
}
```

We also need to point Elasticsearch LTR at the features to log. To do this we use the *sltr* Elasticsearch query, included with Elasticsearch LTR. We construct this query such that it:

- Has a `_name` (the Elasticsearch named queries feature) to refer to it
- Refers to the featureset we created above `more_movie_features`
- Passes our search keywords "rambo" and whatever other parameters our features need

```
{
    "sltr": {
        "_name": "logged_featureset",
        "featureset": "more_movie_features",
        "params": {
            "keywords": "rambo"
        }
    }
}
```

Note: In *Searching with LTR* you'll see us use *sltr* for executing a model. Here we're just using it as a hook to point Elasticsearch LTR at the feature set we want to log.

You might be thinking, wait if we inject `sltr` query into the Elasticsearch query, won't it influence the score? The sneaky trick is to inject it as a filter. As a filter that doesn't actually filter anything, but injects our feature-logging only `sltr` query into our Elasticsearch query:

```
{"query": {
      "bool": {
              "filter": [
              {
                  "terms": {
                      "_id": ["7555", "1370", "1369"]

                  }
              },
              {
                  "sltr": {
                      "_name": "logged_featureset",
                      "featureset": "more_movie_features",
                      "params": {
                          "keywords": "rambo"
                      }
              }}

          ]
  }
}}
```

Running this, you'll see the three hits you'd expect. The next step is to turn on feature logging, referring to the `sltr` query we want to log.

This is what the logging extension gives you. It finds an Elasticsearch *sltr* query, pulls runs the feature set's queries, scores each document, then returns those as computed fields on each document:

```
"ext": {
    "ltr_log": {
        "log_specs": {
            "name": "log_entry1",
            "named_query": "logged_featureset"
        }
    }
}
```

This log extension comes with several arguments:

- `name`: The name of this log entry to fetch from each document
- `named_query` the named query which corresponds to an *sltr* query
- `rescore_index`: if `sltr` is in a rescore phase, this is the index of the query in the rescore list
- `missing_as_zero`: produce a 0 for missing features (when the feature does not match) (defaults to 'false')

Note: Either `named_query` or `rescore_index` must be set so that logging can locate an *sltr* query for logging either in the normal query phase or during rescoring.

Finally the full request:

```
POST tmdb/_search
{
    "query": {
        "bool": {
            "filter": [
                {
                    "terms": {
                        "_id": ["7555", "1370", "1369"]
                    }
                },
                {
                    "sltr": {
                        "_name": "logged_featureset",
                        "featureset": "more_movie_features",
                        "params": {
                            "keywords": "rambo"
                        }
                }}
            ]
        }
    },
    "ext": {
        "ltr_log": {
            "log_specs": {
                "name": "log_entry1",
                "named_query": "logged_featureset"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

And now each document contains a log entry:

```
{
    "_index": "tmdb",
    "_type": "movie",
    "_id": "1370",
    "_score": 20.291,
    "_source": {
        ...
    },
    "fields": {
        "_ltrlog": [
            {
                "log_entry1": [
                    {"name": "title_query"
                     "value": 9.510193},
                    {"name": "body_query
                     "value": 10.7808075}
                ]
            }
        ]
    },
    "matched_queries": [
        "logged_featureset"
    ]
}
```

Now you can join your judgment list with feature values to produce a training set! For the line that corresponds to document 1370 for keywords "Rambo" we can now add:

```
4   qid:1   1:9.510193  2:10.7808075
```

Rinse and repeat for all your queries.

Note: For large judgment lists, batch up logging for multiple queries, use Elasticsearch's bulk search capabilities.

4.5.3 Logging values for a live feature set

Let's say you're running in production with a model being executed in an `sltr` query. We'll get more into model execution in *Searching with LTR*. But for our purposes, a sneak peak, a live model might look something like:

```
POST tmdb/_search
{
    "query": {
        "match": {
            "_all": "rambo"
        }
    },
    "rescore": {
        "query": {
```

(continues on next page)

(continued from previous page)

```
            "rescore_query": {
                "sltr": {
                    "params": {
                        "keywords": "rambo"
                    },
                    "model": "my_model"
                }
            }
        }
    }
}
```

Simply applying the correct logging spec to refer to the `sltr` query does the trick to let us log feature values for our query:

```
"ext": {
    "ltr_log": {
        "log_specs": {
            "name": "log_entry1",
            "rescore_index": 0
        }
    }
}
```

This will log features to the Elasticsearch response, giving you an ability to retrain a model with the same featureset later.

4.5.4 Modifying an existing feature set and logging

Feature sets can be appended to. As mentioned in *Working with Features*, you saw if you want to incorporate a new feature, such as `user_rating`, we can append that query to our featureset `more_movie_features`:

```
PUT _ltr/_feature/user_rating/_addfeatures
{
    "features": [
        "name": "user_rating",
        "params": [],
        "template_language": "mustache",
        "template" : {
            "function_score": {
                "functions": {
                    "field": "vote_average"
                },
                "query": {
                    "match_all": {}
                }
            }
        }
    ]
}
```

Then finally, when we log as the examples above, we'll have our new feature in our output:

```
{"log_entry1": [
    {"name": "title_query"
```

(continues on next page)

(continued from previous page)

```
    "value": 9.510193},
    {"name": "body_query
    "value": 10.7808075},
    {"name": "user_rating",
    "value": 7.8}
]}
```

4.5.5 Logging values for a proposed feature set

You might create a completely new feature set for experimental purposes. For example, let's say you create a brand new feature set, *other_movie_features*:

We can log *other_movie_features* alongside a live production *more_movie_features* by simply appending it as another filter, just like the first example above:

```
POST tmdb/_search
{
"query": {
    "bool": {
        "filter": [
            { "sltr": {
                "_name": "logged_featureset",
                "featureset": "other_movie_features",
                "params": {
                    "keywords": "rambo"
                }
            }},
            {"match": {
                "_all": "rambo"
            }}
        ]
    }
},
"rescore": {
    "query": {
        "rescore_query": {
            "sltr": {
                "params": {
                    "keywords": "rambo"
                },
                "model": "my_model"
            }
        }
    }
}
}
```

Continue with as many feature sets as you care to log!

4.5.6 'Logging' serves multiple purposes

With the tour done, it's worth point out real-life feature logging scenarios to think through.

First, you might develop judgment lists from user analytics. You want to have the exact value of a feature at the precise time a user interaction happened. If they clicked, you want to know the recency, title score, and every other value at

that exact moment. This way you can study later what correlated with relevance when training. To do this, you may build a large comprehensive feature set for later experimentation.

Second, you may simply want to keep your models up to date with a shifting index. Trends come and go, and models lose their effectiveness. You may have A/B testing in place, or monitoring business metrics, and you notice gradual degredation in model performance. In these cases, "logging" is used to retrain a model you're already relatively confident in.

Third, there's the "logging" that happens in model development. You may have a judgment list, but want to iterate heavily with a local copy of Elasticsearch. You're heavily, experimenting with new features, scrapping and adding to feature sets. You of course are a bit out of sync with the live index, but you do your best to keep up. Once you've arrived at a set of model parameters that you're happy with, you can train with production data and confirm the performance is still satisfactory.

Next up, let's briefly talk about training a model in *Uploading A Trained Model* in tools outside Elasticsearch LTR.

4.6 Uploading A Trained Model

Training models occurs outside Elasticsearch LTR. You use the plugin to log features (as mentioned in *Logging Feature Scores*). Then with whichever technology you choose, you train a ranking model. You upload a model to Elasticsearch LTR in the available serialization formats (ranklib, xgboost, and others).

4.6.1 Ranklib Example

We provide two demos for training a model. A fully-gledged Ranklib Demo uses Ranklib to train a model from Elasticsearch queries. You can see how features are logged and how models are trained . In particular, you'll note that logging create a ranklib consumable judgment file that looks like:

```
4   qid:1   1:9.8376875     2:12.318446 # 7555       rambo
3   qid:1   1:10.7808075    2:9.510193 # 1370        rambo
3   qid:1   1:10.7808075    2:6.8449354 # 1369       rambo
3   qid:1   1:10.7808075    2:0.0 # 1368     rambo
```

Here for query id 1 (Rambo) we've logged features 1 (a title TF*IDF score) and feature 2 (a description TF*IDF score) for a set of documents. In train.py you'll see how we call Ranklib to train one of it's supported models on this line:

```
cmd = "java -jar RankLib-2.8.jar -ranker %s -train %s -save %s -frate 1.0" %␣
↪(whichModel, judgmentsWithFeaturesFile, modelOutput)
```

Our "judgmentsWithFeatureFile" is the input to RankLib. Other parameters are passed, which you can read about in Ranklib's documentation.

Ranklib will output a model in it's own serialization format. For example a LambdaMART model is an ensemble of regression trees. It looks like:

```
## LambdaMART
## No. of trees = 1000
## No. of leaves = 10
## No. of threshold candidates = 256
## Learning rate = 0.1
## Stop early = 100

<ensemble>
    <tree id="1" weight="0.1">
```

(continues on next page)

(continued from previous page)

```
        <split>
            <feature> 2 </feature>
            ...
```

Notice how each tree examines the value of features, makes a decision based on the value of a feature, then ultimately outputs the relevance score. You'll note features are referred to by ordinal, starting by "1" with Ranklib (this corresponds to the 0th feature in your feature set). Ranklib does not use feature names when training.

4.6.2 XGBoost Example

There's also an example of how to train a model using XGBoost. Examining this demo, you'll see the difference in how Ranklib is executed vs XGBoost. XGBoost will output a serialization format for gradient boosted decision tree that looks like:

```
[  { "nodeid": 0, "depth": 0, "split": "tmdb_multi", "split_condition": 11.2009, "yes
↪": 1, "no": 2, "missing": 1, "children": [
    { "nodeid": 1, "depth": 1, "split": "tmdb_title", "split_condition": 2.20631, "yes
↪": 3, "no": 4, "missing": 3, "children": [
    { "nodeid": 3, "leaf": -0.03125 },
    ...
```

4.6.3 XGBoost Parameters

Additional parameters can optionally be passed for an XGBoost model. This can be done by specifying the definition as an object, with the decision trees as the 'splits' field. See the example below.

Currently supported parameters:

objective - Defines the model learning objective as specified in the XGBoost documentation. This parameter can transform the final model prediction. Using logistic objectives applies a sigmoid normalization.

Currently supported values: 'binary:logistic', 'binary:logitraw', 'rank:pairwise', 'reg:linear', 'reg:logistic'

4.6.4 Simple linear models

Many types of models simply output linear weights of each feature such as linear SVM. The LTR model supports simple linear weights for each features, such as those learned from an SVM model or linear regression:

```
{
    "title_query" : 0.3,
    "body_query" : 0.5,
    "recency" : 0.1
}
```

4.6.5 Uploading a model

Once you have a model, you'll want to use it for search. You'll need to upload it to Elasticsearch LTR. Models are uploaded specifying the following arguments

- The feature set that was trained against
- The type of model (such as ranklib or xgboost)

- The model contents

Uploading a Ranklib model trained against `more_movie_features` looks like:

```
POST _ltr/_featureset/more_movie_features/_createmodel
{
    "model": {
        "name": "my_ranklib_model",
        "model": {
            "type": "model/ranklib",
            "definition": "## LambdaMART\n
                           ## No. of trees = 1000
                           ## No. of leaves = 10
                           ## No. of threshold candidates = 256
                           ## Learning rate = 0.1
                           ## Stop early = 100

                           <ensemble>
                               <tree id="1" weight="0.1">
                                   <split>
                                       <feature> 2 </feature>
                                       ...
                       "
        }
    }
}
```

Or an xgboost model:

```
POST _ltr/_featureset/more_movie_features/_createmodel
{
    "model": {
        "name": "my_xgboost_model",
        "model": {
            "type": "model/xgboost+json",
            "definition": "[  { \"nodeid\": 0, \"depth\": 0, \"split\": \"tmdb_multi\
↪", \"split_condition\": 11.2009, \"yes\": 1, \"no\": 2, \"missing\": 1, \"children\
↪": [
                                { \"nodeid\": 1, \"depth\": 1, \"split\": \"tmdb_
↪title\", \"split_condition\": 2.20631, \"yes\": 3, \"no\": 4, \"missing\": 3, \
↪"children\": [
                                  { \"nodeid\": 3, \"leaf\": -0.03125 },
                                ..."
        }
    }
}
```

Or an xgboost model with parameters:

```
POST _ltr/_featureset/more_movie_features/_createmodel
{
    "model": {
        "name": "my_xgboost_model",
        "model": {
            "type": "model/xgboost+json",
            "definition": "{
                           \"objective\": \"reg:logistic\",
                           \"splits\": [  { \"nodeid\": 0, \"depth\": 0, \"split\":␣
↪\"tmdb_multi\", \"split_condition\": 11.2009, \"yes\": 1, \"no\": 2, \"missing\": 1,
↪ \"children\": [
```

(continues on next page)

(continued from previous page)

```
                                                    { \"nodeid\": 1, \"depth\": 1, \"split\
↪": \"tmdb_title\", \"split_condition\": 2.20631, \"yes\": 3, \"no\": 4, \"missing\
↪": 3, \"children\": [
                                                      { \"nodeid\": 3, \"leaf\": -0.03125 },
                                                    ...
                                                ]
                                }"
        }
    }
}
```

Or a simple linear model:

```
POST _ltr/_featureset/more_movie_features/_createmodel
{
    "model": {
        "name": "my_linear_model",
        "model": {
            "type": "model/linear",
            "definition": "
                                {
                                    \"title_query\" : 0.3,
                                    \"body_query\" : 0.5,
                                    \"recency\" : 0.1
                                }
                            "
        }
    }
}
```

4.6.6 Models aren't "owned by" featuresets

Though models are created in reference to a feature set, it's important to note after creation models are *top level* entities. For example, to fetch a model back, you use GET:

```
GET _ltr/_model/my_linear_model
```

Similarly, to delete:

```
DELETE _ltr/_model/my_linear_model
```

This of course means model names are globally unique across all feature sets.

The associated features are *copied into* the model. This is for your safety: modifying the feature set or deleting the feature set after model creation doesn't have an impact on a model in production. For example, if we delete the feature set above:

```
DELETE _ltr/_featureset/more_movie_features
```

We can still access and search with "my_linear_model". The following still accesses the model and it's associated features:

```
GET _ltr/_model/my_linear_model
```

You can expect a response that includes the features used to create the model (compare this with the more_movie_features in *Logging Feature Scores*):

```
{
"_index": ".ltrstore",
"_type": "store",
"_id": "model-my_linear_model",
"_version": 1,
"found": true,
"_source": {
    "name": "my_linear_model",
    "type": "model",
    "model": {
        "name": "my_linear_model",
        "feature_set": {
            "name": "more_movie_features",
            "features": [
            {
                "name": "body_query",
                "params": [
                    "keywords"
                    ],
                 "template": {
                    "match": {
                        "overview": "{{keywords}}"
                    }
                }
            },
            {
                "name": "title_query",
                "params": [
                    "keywords"
                ],
                "template": {
                    "match": {
                        "title": "{{keywords}}"
                    }
                }
            }
    ]}}}
```

With a model uploaded to Elasticsearch, you're ready to search! Head to *Searching with LTR* to see put model into action.

4.7 Searching with LTR

Now that you have a model, what can you do with it? As you saw in *Logging Feature Scores*, the Elasticsearch LTR plugin comes with the *sltr* query. This query is also what you use to execute models:

```
POST tmdb/_search
{
    "query": {
        "sltr": {
                "params": {
                    "keywords": "rambo"
                },
                "model": "my_model"
            }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Warning: you almost certainly don't want to run *sltr* this way :)

4.7.1 Rescore top N with *sltr*

In reality you would never want to use the `sltr` query this way. Why? This model executes on *every result in your index*. These models are CPU intensive. You'll quickly make your Elasticsearch cluster crawl with the query above.

More often, you'll execute your model on the top N of a baseline relevance query. You can do this using Elasticsearch's built in rescore functionality:

```
POST tmdb/_search
{
    "query": {
        "match": {
            "_all": "rambo"
        }
    },
    "rescore": {
        "window_size": 1000,
        "query": {
            "rescore_query": {
                "sltr": {
                    "params": {
                        "keywords": "rambo"
                    },
                    "model": "my_model"
                }
            }
        }
    }
}
```

Here we execute a query that limits the result set to documents that match "rambo". All the documents are scored based on Elasticsearch's default similarity (BM25). On top of those already reasonably relevant results we apply our model over the top 1000.

Viola!

4.7.2 Scoring on a subset of features with *sltr* (added in 1.0.1-es6.2.4)

Sometimes you might want to execute your query on a subset of the features rather than use all the ones specified in the model. In this case the features not specified in `active_features` list will not be scored upon. They will be marked as missing. You only need to specify the `params` applicable to the `active_features`. If you request a feature name that is not a part of the feature set assigned to that model the query will throw an error.

```
POST tmdb/_search
{
    "query": {
        "match": {
```

(continues on next page)

(continued from previous page)

```
            "_all": "rambo"
        }
    },
    "rescore": {
        "window_size": 1000,
        "query": {
            "rescore_query": {
                "sltr": {
                    "params": {
                        "keywords": "rambo"
                    },
                    "model": "my_model",
                    "active_features": ["title_query"]
                }
            }
        }
    }
}
```

Here we apply our model over the top 1000 results but only for the selected features which in this case is title_query

4.7.3 Models! Filters! Even more!

One advantage of having `sltr` as just another Elasticsearch query is you can mix/match it with business logic and other. We won't dive into these examples here, but we want to invite you to think creatively about scenarios, such as

- Filtering out results based on business rules, using Elasticsearch filters before applying the model
- Chaining multiple rescores, perhaps with increasingly sophisticated models
- Rescoring once for relevance (with *sltr*), and a second time for business concerns
- Forcing "bad" but relevant content out of the rescore window by downboosting it in the baseline query

4.8 On XPack Support (Security)

X-Pack is the collection of extensions provided by elastic to enhance the capabilities of the Elastic Stack with things such as reporting, monitoring and also security. If you installed x-pack your cluster will now be protected with the security module, this will also be like this if you are using Elasticsearch through the Elastic Cloud solution.

4.8.1 Setup roles and users.

After installing the plugin, the first thing you will have to do is configure the necessary users and roles to access let the LTR plugin operate.

We recommend you to create two separate roles, one for administrative task such as creating the models, updating the feature sets, etc .. and one to run the queries.

For this configuration, we supose you already have identified, and created two users, one for running queries and one for doing administrative tasks. If you need help to create the users, we recommend you to check the x-pack api documentation for user management.

To create two roles, you can do it with these commands:

```
POST /_xpack/security/role/ltr_admin
{
  "cluster": [ "ltr" ],
  "indices": [
    {
        "names": [ ".ltrstore*" ],
        "privileges": [ "all" ],
    }
  ]
}

POST /_xpack/security/role/ltr_query
{
    "cluster": [ "ltr" ],
    "indices": [
        {
            "names": [ ".ltrstore*" ],
            "privileges": [ "read" ],
        }
    ]
}
```

the first one will allow the users to perform all the operations while the last one will only allow read operations.

Once the roles are defined, the last step will be to attach this roles to existing users, for this documentation we will suppose two users, ltr_admin and ltr_user. The commands to set the roles are:

```
POST /_xpack/security/role_mapping/ltr_admins
{
    "roles": [ "ltr_admin" ],
    "rules": {
        "field" : { "username" : [ "ltr_admin01", "ltr_admin02" ] }
    },
    "metadata" : {
        "version" : 1
    }
}

POST /_xpack/security/role_mapping/ltr_users
{
    "roles": [ "ltr_query" ],
    "rules": {
        "field" : { "username" : [ "ltr_user01", "ltr_user02" ] }
    },
    "metadata" : {
        "version" : 1
    }
}
```

After this two steps, your plugin will be fully functional in your x-pack protected cluster.

For more in deep information on how to define roles, we recommend you to check the elastic x-pack api documentation.

4.8.2 Considerations

The read access to models via the sltr query is not strictly gated by x-pack. The access will only be checked if the model needs to be loaded, however If the model is already in the cache for that node no checks will be performed.

This will generally not have a major security impact, however is important to take into account in case is important for your use case.

4.9 Advanced Functionality

This section documents some additional functionality you may find useful after you're comfortable with the primary capabilities of Elasticsearch LTR.

4.9.1 Reusable Features

In *Working with Features* we demonstrated creating feature sets by uploading a list of features. Instead of repeating common features in every feature set, you may want to keep a library of features around.

For example, perhaps a query on the title field is important to many of your feature sets, you can use the feature API to create a title query:

```
POST _ltr/_feature/titleSearch
{
    "feature":
    {
        "params": [
        "keywords"
        ],
        "template": {
        "match": {
            "title": "{{keywords}}"
        }
        }
    }
}
```

As you'd expect, normal CRUD operations apply. You can DELETE a feature:

```
DELETE _ltr/_feature/titleSearch
```

And fetch an individual feature:

```
GET _ltr/_feature/titleSearch
```

Or look at all your features, optionally filtered by name prefix:

```
GET /_ltr/_feature?prefix=t
```

You can create or update a feature set, you can refer to the titleSearch feature:

```
POST /_ltr/_featureset/my_featureset/_addfeatures/titleSearch
```

This will place titleSearch at the next ordinal position under "my_feature_set"

4.9.2 Derived Features

Features that build on top of other features are called derived features. These can be expressed as lucene expressions. They are recognized by `"template_language": "derived_expression"`. Besides these can also take in query time variables of type Number as explained in *Create a feature set*.

Script Features

These are essentially *Derived Features*, having access to the `feature_vector` but could be native or painless elasticsearch scripts rather than lucene expressions. `"template_language": "script_feature"` allows LTR to identify the templated script as a regular elasticsearch script e.g. native, painless, etc.

The custom script has access to the feature_vector via the java Map interface as explained in *Create a feature set*.

(WARNING script features can cause the performance of your Elasticsearch cluster to degrade, if possible avoid using these for feature generation if you require your queries to be highly performant)

Script Features Parameters

Script features are essentially native/painless scripts and can accept parameters as per the elasticsearch script documentation. We can override parameter values and names to scripts within LTR scripts. Priority for parameterization in increasing order is as follows

- parameter name, value passed in directly to source script but not in params in ltr script. These cannot be configured at query time.
- parameter name passed in to sltr query and to source script, so the script parameter values can be overridden at query time.
- ltr script parameter name to native script parameter name indirection. This allows ltr parameter name to be different from the underlying script parameter name. This allows same native script to be reused as different features within LTR by specifying different parameter names at query time:

```
POST _ltr/_featureset/more_movie_features
{
   "featureset": {
        "features": [
            {
                "name": "title_query",
                "params": [
                    "keywords"
                ],
                "template_language": "mustache",
                "template": {
                    "match": {
                        "title": "{{keywords}}"
                    }
                }
            },
            {
                "name": "custom_title_query_boost",
                "params": [
                    "some_multiplier",
                    "ltr_param_foo"
                ],
                "template_language": "script_feature",
                "template": {
                    "lang": "painless",
                    "source": "(long)params.default_param * params.feature_vector.
↪get('title_query') * (long)params.some_multiplier * (long) params.param_foo",
                    "params": {
                        "default_param" : 10.0,
                        "some_multiplier": "some_multiplier",
                        "extra_script_params": {"ltr_param_foo": "param_foo"}
```

(continues on next page)

(continued from previous page)

```
                        }
                    }
                }
            ]
        }
}
```

4.9.3 Multiple Feature Stores

We defined a feature store in *Working with Features*. A feature store corresponds to an independent LTR system: features, feature sets, models backed by a single index and cache. A feature store corresponds roughly to a single search problem, often tied to a single application. For example wikipedia might be backed by one feature store, but wiktionary would be backed by another. There's nothing that would be shared between the two.

Should your Elasticsearch cluster back multiple properties, you can use all the capabilities of this guide on named feature stores, simply by:

```
PUT _ltr/wikipedia
```

Then the same API in this guide applies to this feature store, for example to create a feature set:

```
POST _ltr/wikipedia/_featureset/attempt_1
{
   "featureset": {
        "features": [
            {
                "name": "title_query",
                "params": [
                    "keywords"
                ],
                "template_language": "mustache",
                "template": {
                    "match": {
                        "title": "{{keywords}}"
                    }
                }
            }
        ]
   }
}
```

And of course you can delete a featureset:

```
DELETE _ltr/wikipedia/_featureset/attempt_1
```

4.9.4 Model Caching

The plugin uses an internal cache for compiled models.

Clear the cache for a feature store to force models to be recompiled:

```
POST /_ltr/_clearcache
```

Get cluster wide cache statistics for this store:

```
GET /_ltr/_cachestats
```

Characteristics of the internal cache can be controlled with these node settings:

```
# limit cache usage to 12 megabytes (defaults to 10mb or max_heap/10 if lower)
ltr.caches.max_mem: 12mb
# Evict cache entries 10 minutes after insertion (defaults to 1hour, set to 0 to␣
↪disable)
ltr.caches.expire_after_write: 10m
# Evict cache entries 10 minutes after access (defaults to 1hour, set to 0 to disable)
ltr.caches.expire_after_read: 10m
```

4.9.5 Extra Logging

As described in *Logging Feature Scores*, it is possible to use the logging extension to return the feature values with each document. For native scripts, it is also possible to return extra arbitrary information with the logged features.

For native scripts, the parameter `extra_logging` is injected into the script parameters. The parameter value is a Supplier <Map>, which provides a non-null `Map<String,Object>` **only** during the logging fetch phase. Any values added to this Map will be returned with the logged features:

```
@Override
public double runAsDouble() {
...
    Map<String,Object> extraLoggingMap = ((Supplier<Map<String,Object>>) getParams().
↪get("extra_logging")).get();
    if (extraLoggingMap != null) {
        extraLoggingMap.put("extra_float", 10.0f);
        extraLoggingMap.put("extra_string", "additional_info");
    }
...
}
```

If (and only if) the extra logging Map is accessed, it will be returned as an additional entry with the logged features:

```
{
    "log_entry1": [
        {
            "name": "title_query"
            "value": 9.510193
        },
        {
            "name": "body_query"
            "value": 10.7808075
        },
        {
            "name": "user_rating",
            "value": 7.8
        },
        {
            "name": "extra_logging",
            "value": {
                "extra_float": 10.0,
                "extra_string": "additional_info"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

CHAPTER 5

Indices and tables

- genindex
- search