



University
of Glasgow



IR From Bag-of-words to BERT and Beyond through Practical Experiments

An ECIR 2021 tutorial with
PyTerrier and OpenNIR

Sean MacAvaney*
Craig Macdonald*
Nicola Tonello*

(*Alphabetical ordering)



Part 2 -*Modern* Retrieval Architectures

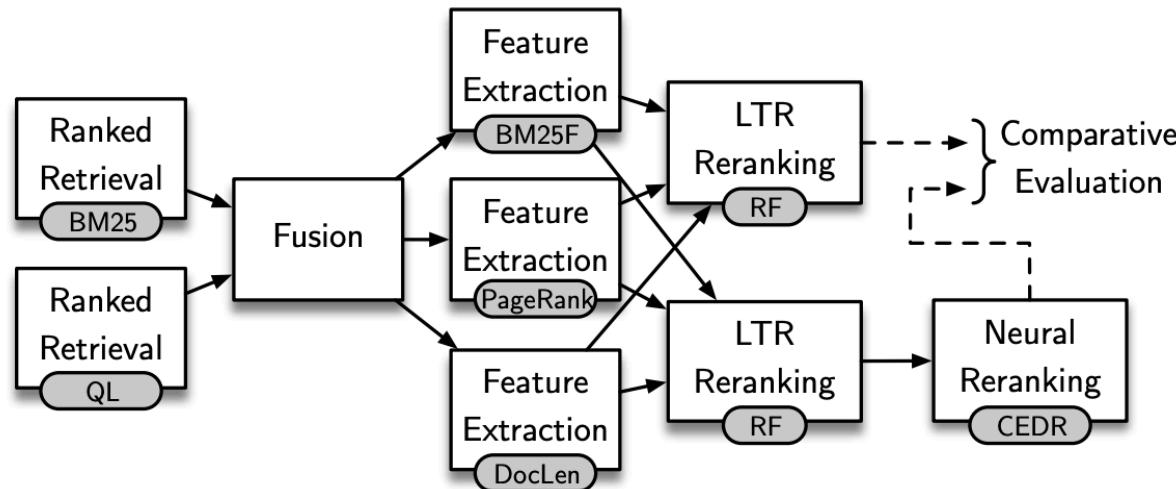
PyTerrier data model and operators, towards re-rankers and learning-to-rank

Bridging to Modern Retrieval Architectures

In Part 1, we saw how to perform an experiment in a declarative manner, and perform simple retrievals

Actually, in IR, we have many retrieval and re-ranking components

Working with and combining result files to allow comparing complex ranking *pipelines* is tricky:



In Part 2, we generalise the architecture that has been seen into the PyTerrier data model and operators, to make it easy to build complex ranking pipelines

Modern Ranking Pipelines

By using PyTerrier transformers and operators, we show that can easily express complex ranking pipelines in a **conceptual & declarative way**

Furthermore, we review **modern techniques that fit into such ranking pipelines – such as...**

- Query rewriting & pseudo-relevance feedback
- Learning to Rank

... and demonstrate how they can be implemented and extended within the PyTerrier framework

Finally, we also discuss the usefulness of PyTerrier for teaching IR

ILOs for this Session

- ILO 2A. Understand modern retrieval architectures, such as re-rankers and ranking pipelines incorporating learning-to-rank strategies.
- ILO 2B. Understand how different ranking and re-ranking operations can be composed to make more complex ranking models, in a declarative manner and leveraging a specific data model.
- ILO 2C. Perform retrieval experiments within a Python notebook, including use of different features and learning-to-rank.
- ILO 2D. Understand the benefits of PyTerrier's declarative nature for teaching IR

Outline of Part 2



Part 2A: PyTerrier data model & transformers

Part 2B: Learning to Rank

Part 2C: Usefulness of PyTerrier for Teaching IR

Part 2D: Wrap up and move to practical session



University
of Glasgow



UNIVERSITÀ DI PISA



Part 2A

PYTERRIER DATA MODEL AND TRANSFORMERS

An extensible data model for IR

Let Q and D represent sets of queries and documents.
Each row is a tuple, with *identifiers* and other optional attributes

	<u>qid</u>	query	...
$Q =$			

	<u>docno</u>	(text)	(url)	...
$D =$				

We define other derived types and attributes – e.g.

$R \subset Q \times D =$
(ranked documents)

	<u>qid</u>	query	<u>docno</u>	score	rank	...

$RA \subset Q \times D =$
(relevance assessment)

	<u>qid</u>	<u>docno</u>	label

Also R_f , for LTR [Part 2B]

(All of these are Pandas dataframes, i.e. relations)

Transformers

We build our declarative IR language on generic function objects called *transformers*.

- Each transformer takes as input a dataframe in one of Q, D, R, R_f specifications, and returns a new dataframe in one of those specifications

$$\text{Transformer}().\text{transform}(\Lambda) \mapsto \Lambda$$

where $\Lambda \in \{Q, D, R, R_f\}$

You have already seen one transformer, **BatchRetrieve**

However, this definition is **general** enough to allow expressing **many** common IR operations as transformers, and the **composition** of transformers into *pipelines*

Transformer Examples (1)

Retrieval:

$$R' = \text{Retrieve}()(Q).$$

- There are many ways to instantiate `Retrieve()`, e.g. with different weighting models.

```
firstpass = pt.BatchRetrieve(index, wmodel="BM25")
```

Query Rewriting:

$$Q' = \text{Rewrite}(Q).$$

- E.g. Sequential Dependence Model, adds #1 and #uw8 operators to the query

```
pt.rewrite.SequentialDependence()
```

- No documents returned

Transformer Examples (2)

PRF Query Expansion $Q = \text{Expand}(R)$

- Defined on queries & documents, returns queries, i.e. $R \mapsto Q$

```
rm3 = pt.rewrite.RM3(index)
bo1 = pt.rewrite.Bo1QueryExpansion(index)
```

- Output needs to applied to a retrieve transformer:

$$R' = \text{Retrieve}(\text{Expand}(\text{Retrieve}(Q)))$$

Re-Ranking

$$R' = \text{Rerank}(R)$$

- After retrieving a *candidate set* (e.g. BM25), we can add more features, query dependent or query independent

$$R' = \text{Rerank}(\text{Retrieve}(R))$$

Transformers → Operators

We recognise seven families of transformers

Input	Output	Transformer
D	D'	Document expansion
D	\emptyset	Document indexing
Q	Q'	Query rewriting
Q	R	Retrieval
R	Q'	Query expansion
R	R'	Re-ranking
R	R_f	Feature extraction

These can be combined as retrieval **pipelines** in different ways,
e.g. PRF

$$R' = \text{Retrieve}(\text{Expand}(\text{Retrieve}(Q)))$$

But such a notation is **klunky**. Python allows **operator overloading**

- This permits interesting possibilities to make more expressive *pipelines* of transformers



Transformer Operators

PyTerrier defines a suite of operators on transformers

Op.	Name	Description
>>	<i>then</i> <i>(composition)</i>	Pass the output from one transformer to the next transformer
+	<i>linear combine</i>	Sum the query-document scores of the two retrieved results lists
*	<i>scalar product</i>	Multiply the query-document scores of a retrieved results list by a scalar
**	<i>feature union</i>	Combine two retrieved results lists as features
	<i>set union</i>	Make the set union of documents from the two retrieved results lists
&	<i>set intersection</i>	Make the set intersection of the two retrieved results lists
%	<i>rank cutoff</i>	Shorten a retrieved results list to the first K elements
^	<i>concatenate</i>	Add the retrieved results list from one transformer to the bottom of the other

The ICTIR 2020 paper provides relational algebra semantics for each operator

- Let's look at a few...

Operator Example (1)

Composition (aka “then”) >>

- Consider a pipeline of retrieval, PRF and a ranking, expressed as follows

$$R' = \text{Retrieve}(\text{Expand}(\text{Retrieve}(Q)))$$

- Let binary operator $>>$ denote that the output of one transformer should be used as input to another, defined as:

$$\Lambda' = (T_1 >> T_2)(\Lambda) :=$$

$$\Lambda_1 = T_1(\Lambda) \quad \text{where } \Lambda \in \{Q, D, R, R_f\}$$

$$\Lambda' = T_2(\Lambda_1).$$

- So we can express the transformer pipeline as follows

Pipe = Retrieve >> Expand >> Rerank

$$R' = \text{Pipe}(Q)$$

Lets look at some retrieval use cases

1. Linear Combinations

```
linear = 2 * pt.BatchRetrieve(index, wmodel="BM25")  
+ 3 * pt.BatchRetrieve(index, wmodel="DPH")
```

Final Output Type: R

Output Types: R

2. Sequential Dependence Model (proximity)

```
sdm = pt.rewrite.SequentialDependence() >> bm25
```

Input Type: Q

qid	query
0	1 chemical reactions

Output Type: Q

Output Type: R

qid	query
0	1 #combine:0=0.1:wmodel=org.terrier.matching.models.dependence.pBiL(#1(chemical reactions)) #combine:0=0.1:wmodel=org.terrier.matching.models.dependence.pBiL(#uw8(chemical reactions)) #combine:0=0.1:wmodel=org.terrier.matching.models.dependence.pBiL(#uw12(chemical reactions))

Lets look at some retrieval use cases

1. Linear Combinations

```
linear = 2 * pt.BatchRetrieve(index, wmodel="BM25")  
+ 3 * pt.BatchRetrieve(index, wmodel="DPH")
```

Final Output Type: R

Output Types: R

2. Sequential Dependence Model (proximity)

```
sdm = pt.rewrite.SequentialDependence() >> bm25
```

Input Type: Q

Output Type: Q

Output Type: R

3. Pseudo-relevance feedback (QE)

```
sdm = bm25 >> pt.rewrite.RM3(index) >> bm25
```

Output Type: R

Output Type: Q

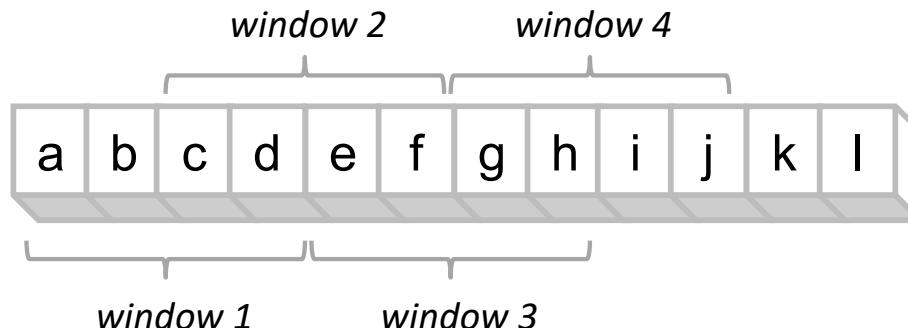
Output Type: R

Retrieval Use Case: Passaging

Some long documents may only contain a relevant passage about the query

- In such cases, **passage-based retrieval** can benefit effectiveness
- It's also useful for neural models that do not scale to long documents

For instance, applying a **sliding window over the text of the document to get passages, and scoring each passage individually**



Window size 4
Window stride 2

- Some settings benefit from prepending the title to each passage

Retrieval Use Case: Passaging

Lets try this out:

```
Bm25maxp = pt.BatchRetrieve(index, wmodel="BM25", metadata=["docno", "text"])
    >> pt.text.sliding("text", length=150, stride=75)
    >> pt.text.scorer(wmodel="BM25")
    >> pt.text.max_passage()
```

sliding("text", 2,1)

BR output (R w/ text)

qid	docno	text
q1	d1	a b c d

output

qid	docno	text
q1	d1%p1	a b
q1	d1%p2	b c
q1	d1%p3	c d

textscorer output (R)

qid	docno	rank	score
q1	d1%p5	0	5.0
q1	d2%p4	1	4.0
q1	d1%p3	2	3.0
q1	d1%p1	3	1.0

max_passage output

qid	docno	rank	score
q1	d1	0	5.0
q1	d2	1	4.0

(Some attributes omitted for brevity, e.g. query)

Of course, we could change `pt.text.scorer()` to be a more advanced approach, e.g. BERT-based neural re-ranker [Part 3]



Applying passaging during retrieval can be expensive

- Can it be applied at indexing time?
- There are also other techniques that we will see in Part 4 (e.g. DeepCT, doc2query) that can be applied at indexing time

We have recently integrated the `>>` operator so that transformers can be used during indexing

Hence, a MaxPassage retrieval can also be achieved as follows

```
indexer = pt.text.sliding("text", length=150, stride=75)
          >> pt.IterDictIndexer()
indexref = indexer.index(dataset.get_corpus_iter())
MaxP = pt.BatchRetrieve(indexref, wmodel="BM25")
        >> pt.text.max_passage()
```

Note that while transformers operate on a dataframes, we might not fit an entire corpus into a dataframe

- Hence index pipelines perform transparent batching of documents

Transformer Implementations (1/2)

All transformers implement a base class called TransformerBase

- This has all the methods to support operator overloading, e.g. `__add__()`, `__rshift__()`

However, creating your own transformer is easy using `pt.apply` functions. These create a new transformer from your own function

- For instance, to rewrite a query – `pt.apply.query()`

```
# this will remove pre-defined stopwords from the query
stops=set(["and", "the"])

# a naieve function to remove stopwords
def _remove_stops(q):
    terms = q["query"].split(" ")
    terms = [t for t in terms if not t in stops ]
    return " ".join(terms)

# a query rewriting transformer applying _remove_stops
p1 = pt.apply.query(_remove_stops) >> pt.BatchRetrieve(index, wmodel="DPH")
```

- `pt.apply.query()` returns a transformer, which applies `_remove_stops` to each query

Transformer Implementations

(2/2): *pt.apply.doc_score()*

pt.apply.doc_score() uses a function that returns a score

```
# this transformer will subtract the number of character in the query  
# from the score of each retrieved document
```

```
p = pt.BatchRetrieve(index, wmodel="DPH") >>  
    pt.apply.doc_score(lambda doc : doc["score"] - len(doc["query"]))
```

In this way, ANY function that can return a number can be used for retrieval

- i.e. any (SOTA) approach that can be expressed as a function of the **text of the document** and the **text of the query** can be used as a re-ranker in PyTerrier

pt.apply.doc_score() and pt.apply.query() have known semantics

- There are also advanced “generic” apply functions that can apply an arbitrary change of a dataframe as a transformer

Summary: PyTerrier Standard Transformers

Retrieval

- `pt.BatchRetrieve()`

Query Rewriting/Expansion

- `pt.rewrite.SequentialDependence()`
- `pt.rewrite.RM3()`
- `pt.rewrite.Bo1QueryExpansion()`
- `pt.rewrite.KLQueryExpansion()`

Text

- `pt.text.get_text()`
- `pt.text.scorer()`
- `pt.text.sliding()`
- `pt.text.max_passage()` `first_passage()` `mean_passage()` `kmaxavg_passage()`

In Part 2B, we'll see learning-to-rank transformers

In Parts 3 & 4, we'll see more add-on transformers for neural re-ranking and end-to-end

Summary: PyTerrier Transformer Operators

Op.	Name	Description
>>	<i>then</i> (composition)	Pass the output from one transformer to the next transformer
+	<i>linear combine</i>	Sum the query-document scores of the two retrieved results lists
*	<i>scalar product</i>	Multiply the query-document scores of a retrieved results list by a scalar
**	<i>feature union</i>	Combine two retrieved results lists as features <i>(Part 2B)</i>
	<i>set union</i>	Make the set union of documents from the two retrieved results lists
&	<i>set intersection</i>	Make the set intersection of the two retrieved results lists
%	<i>rank cutoff</i>	Shorten a retrieved results list to the first K elements
^	<i>concatenate</i>	Add the retrieved results list from one transformer to the bottom of the other

Reflection

Transformers build on PyTerrier's extensible data model

- defined on IR-level objects: queries and documents
- One could imagine data model columns for, e.g., ConvQA

Operators, defined with clear semantics, allow different transformers to be easily combined

- The structure of the pipeline implementations are close to their conceptual design
- This is all declarative in nature - we aren't **writing code** that iterates over any queries or documents (no for loops ☺!)

New techniques can be used and evaluated within a unified framework

- `pt.apply.doc_score()` can be easily applied to integrate new state-of-the-art approaches

Reasoning on Transformer Pipelines (1/2)

Validation

(Coming Soon!)

`pipe.validate()`

- Lots of transformers, lots of operators. Is my pipeline expression valid? Do I have the correct columns?
- We can know the desired input & actual output columns of each transformer
 - E.g. BatchRetrieve needs qid and query columns, sliding() needs text column etc
- Similarly, we know actual inputs and desired outputs of an entire pipeline
 - E.g. We know that queries have qid and query columns
 - E.g. We know that we need qid, docno, score to evaluate a ranking
- Therefore, it is possible to *validate* (type check) a pipeline before execution – check if each transformer stage emits the necessary columns for the next stage
 - Q + Q would throw an error – we cannot make a linear combination of Q
- This would be transparent to most users; Some “generic” `pt.apply` operations would need to specify their input and output columns to ensure validation

Reasoning on Transformer Pipelines (2/2)

Optimisation

pipe = pipe.compile()

- Pipelines with identical semantics can be written in different ways
- In our ICTIR 2020 paper, we showed that pipelines can be rewritten into equivalent but faster equivalent pipelines as a form of graph rewriting

Parallelisation

(Coming Soon!)

- Transformers can be parallelized using different processes, or even to different machines
- This is because the primary keys of each transformer is known, so we can partition a Q or R dataframe by qid; a D dataframe by docno



University
of Glasgow



UNIVERSITÀ DI PISA



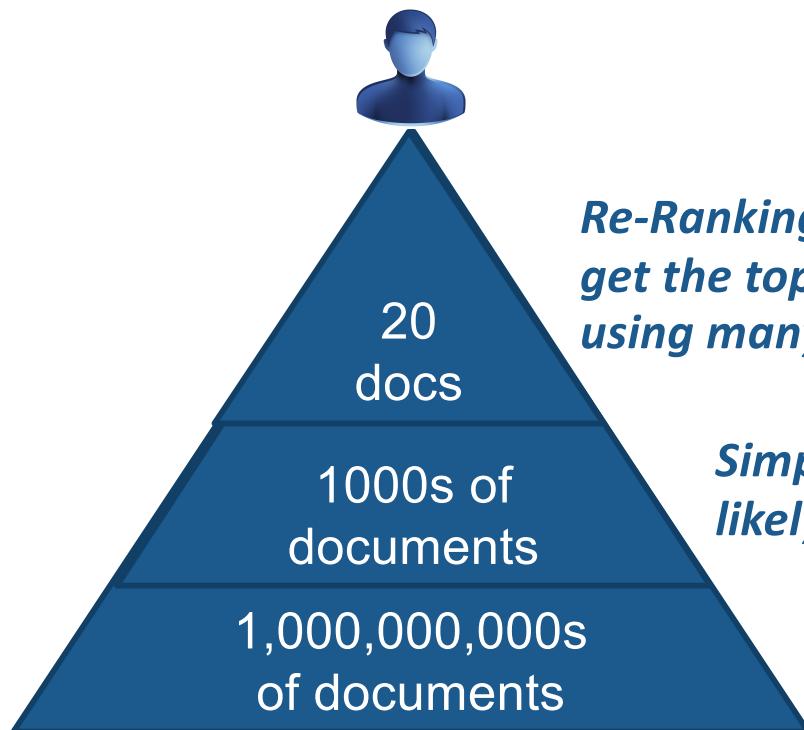
Part 2B

LEARNING TO RANK

Ranking Cascades

In many search scenarios, the ranking process can be seen as a series of cascades [1]

- Rank some documents
- Pass top-ranked onto next cascade for refined re-ranking



Re-Ranking: Try really hard to get the top of the ranking correct, using many signals (features)



LEARNING TO RANK Features & Models

Simple Ranking: Identify a set most likely to contain relevant documents



e.g. BM25

Boolean: Do query terms occur?



e.g. AND/OR /phrase

How to choose term weighting models?

- Term weighting models have different **assumptions** about how relevant documents should be retrieved

Also:

- **Proximity-models:** close co-occurrences matter more
- **Priors:** documents with particular lengths or URL/inlink distributions matter more
- **Field-based models:** term occurrences in different fields matter differently (e.g. BM25F, PL2F)
- ***Query Features:*** Long queries, difficult queries, query type

How to combine all these easily and appropriately?

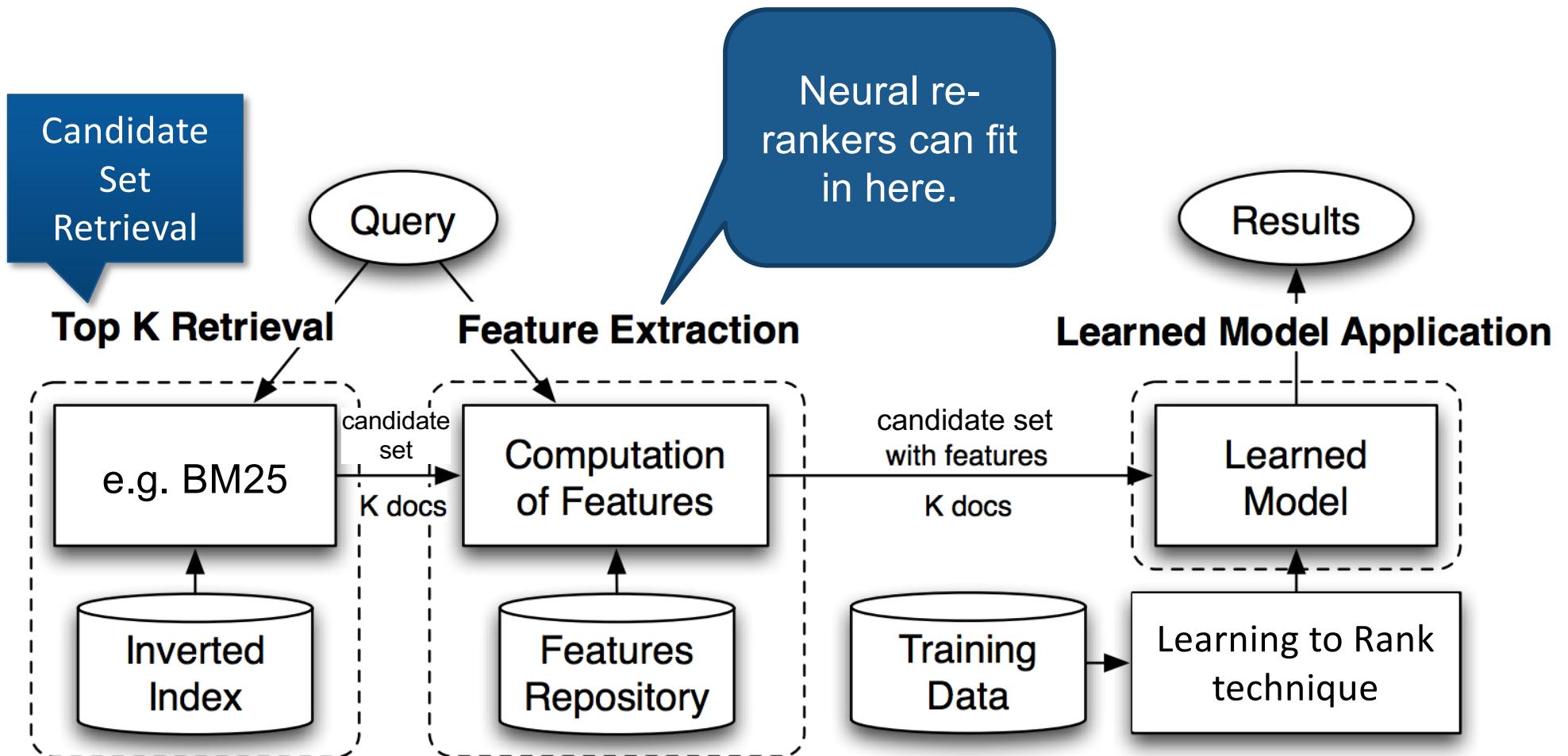
Learning to Rank definition

Learning to rank describes the application of tailored machine learning techniques, applied to re-rank a candidate set of retrieved documents

- This requires hand-engineered features (incl. weighting models) as relevance signals
- LTR techniques should focus on getting the top-ranked documents ranked correctly, using adapted loss functions:
 - Pointwise: e.g. regression – just tries to predict document label
 - Pairwise: tries to get pairs of documents with differing label correctly ranked
 - Listwise: tries to get most relevant document at top of ranking

Different model forms are popular: linear, trees, neural nets

LTR, Schematically...

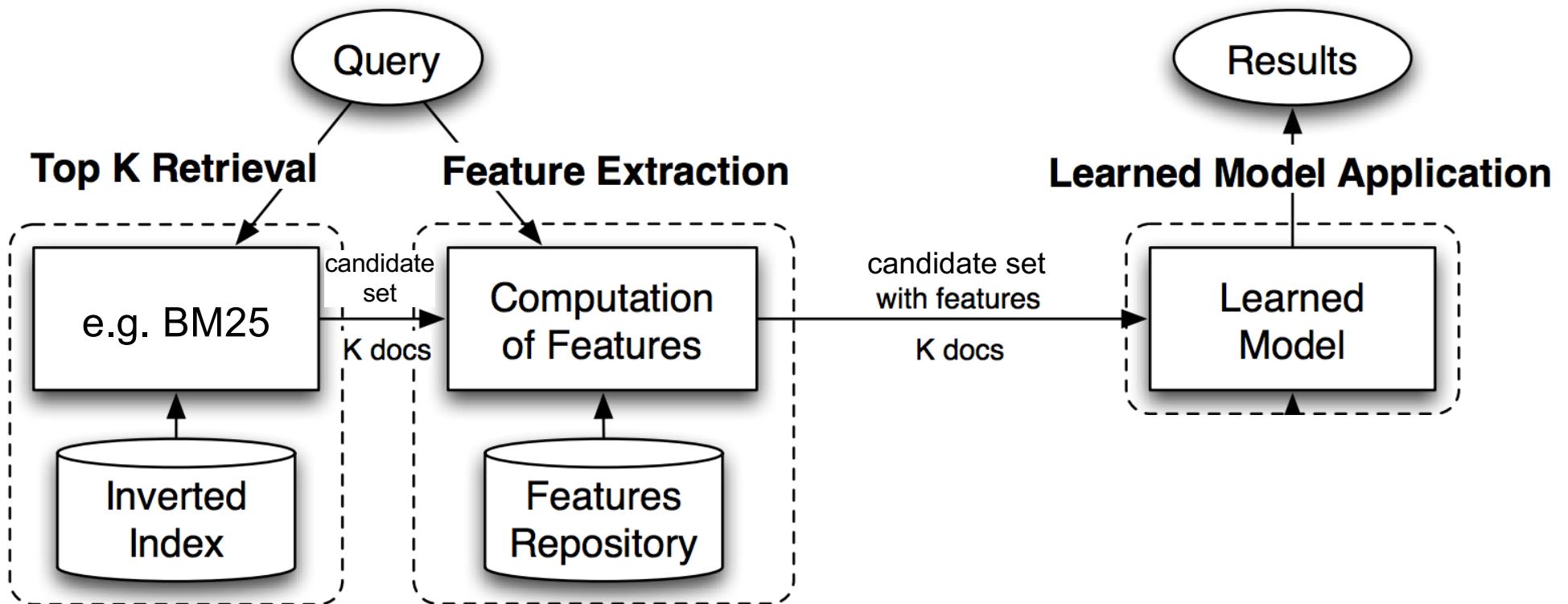


Types of Features

Typically, commercial search engines use hundreds of features for ranking documents, usually categorised as follows:

Name	Varies depending on...		Examples
	Query	Document	
Query Dependent Features	✓	✓	Weighting models, e.g. BM25, PL2 Proximity models, e.g. Markov Random Fields Field-based weighting models, e.g. PL2F Neural re-rankers [Part 3]
Query Independent Features	X	✓	PageRank, number of inlinks Spamminess
Query Features	✓	X	Query length, Query performance predictors Presence of entities

Matching LTR concepts to PyTerrier...



In PyTerrier:

```
(  
pt.BatchRetrieve()  
**  
pt.BatchRetrieve() % K >> pt.apply.doc_score() >> pt.ltr.apply_learned_model()  
**  
;  
)
```

“Featured” Data Model

Retrieved documents with features, R_f :

<u>qid</u>	query	<u>docno</u>	score	rank	<i>features</i>

- The features column contains, for each document, an array of additional feature scores.
- These can be used for learning and applying LTR models
- Applying an LTR model would override the original score of the document, causing a re-ranking

Feature Union Operator

Feature Union (**)

- Say we want to take multiple retrieval approaches as different features. We use $**$ to denote, that given a set of input documents, use these transformers to obtain additional feature scores

- Formally: $Rf = (T_1 \ ** \ T_2)(R) :=$

$$R_1 = T_1(R), \quad R_2 = T_2(R)$$

$$Rf = (R_1 \bowtie R_2)[[f_1, f_2] \rightarrow f]$$

- E.g.

BatchRetrieve("BM25") >>

(BatchRetrieve("Tf") ** BatchRetrieve("PL2"))

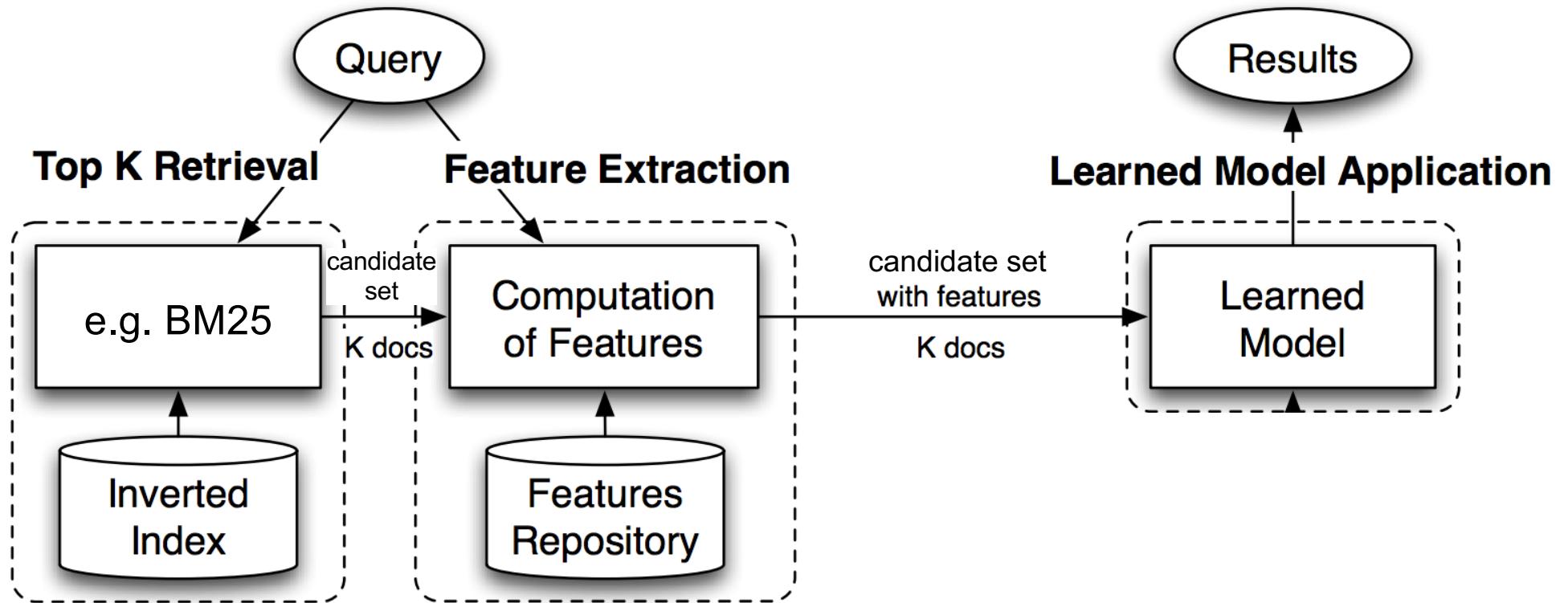
R

	qid	docno	score
1	q1	d5	(bm25 score)

R_f

	qid	docno	score	features
1	q1	d5	(bm25 score)	[tf score, pl2 score]

Matching LTR concepts to PyTerrier...

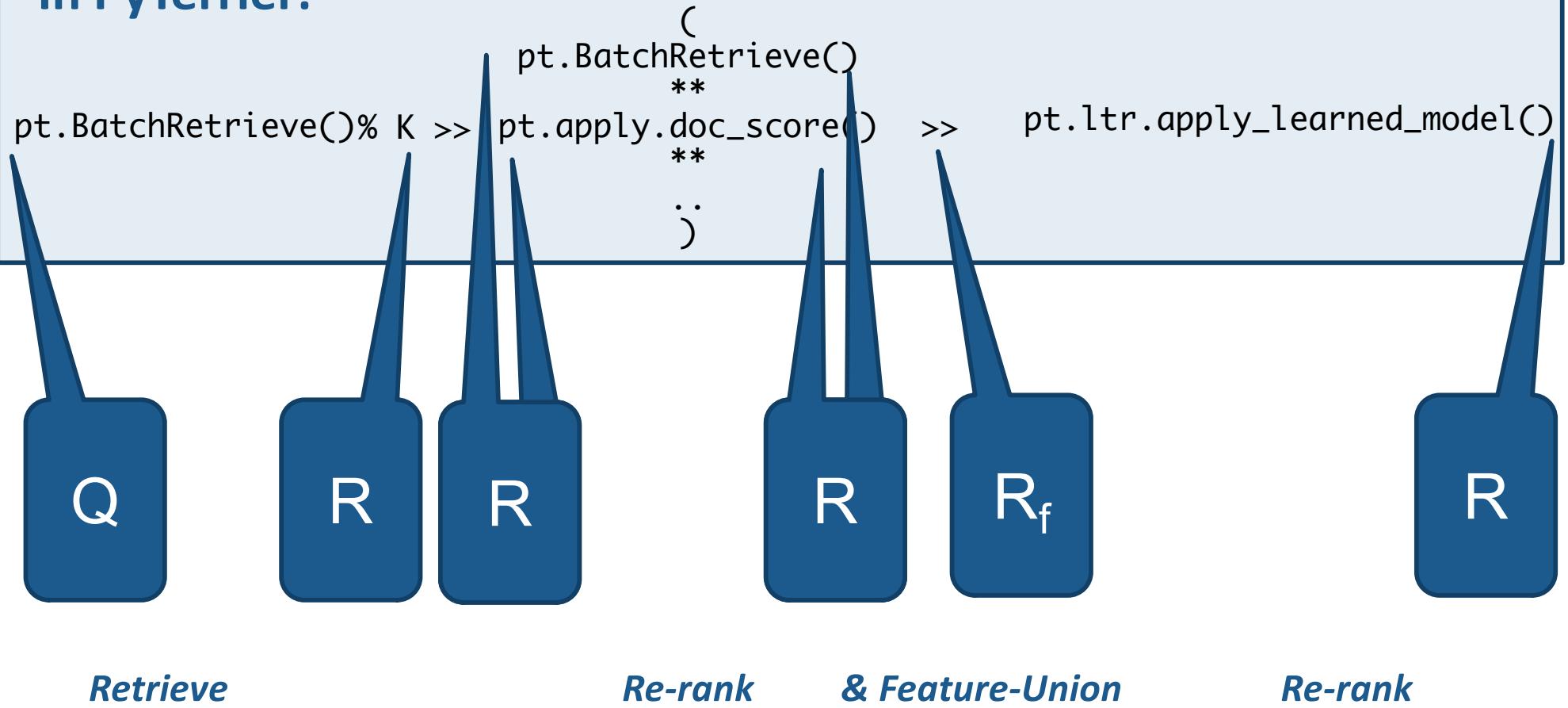


In PyTerrier:

```
(  
pt.BatchRetrieve()  
**  
pt.BatchRetrieve() % K >> pt.apply.doc_score() >> pt.ltr.apply_learned_model()  
**  
;)
```

Data Model Transformations

In PyTerrier:



Query-Dependent Feature Extraction



BatchRetrieve("BM25") >> (BatchRetrieve("Tf") ** BatchRetrieve("PL2"))

Here we are using BatchRetrieve as a **re-ranker**

- Feature union is only defined on an input set of documents
- RHS BatchRetrieves will re-score the BM25 retrieved documents using the other weighting models

We might deploy a number of query dependent features

- Such as additional weighting models, e.g. fields/proximity, that are calculated based on information in the inverted index
- Each BatchRetrieve causes another access to the inverted index posting lists

FeatureBatchRetrieve is an alternative for Terrier

```
pt.FeatureBatchRetrieve(index, wmodel="BM25",
                        features=[“WMODEL:Tf”, “WMODEL:PL2”])
```

- Multiple features scored in a single pass of the inverted index
- PyTerrier can perform this optimisation when .compile() is called



University
of Glasgow



UNIVERSITÀ DI PISA



TYPES OF MODELS FOR LTR

Types of Learned Models (1)

Linear Model

Linear Models (the most intuitive to comprehend)

- Many learning to rank techniques generate a linear combination of feature values:

$$score(d, Q) = \sum_f w_f \cdot value_f(d)$$

- Linear Models make some assumptions:
 - **Feature Usage**: They assume that the same features are needed by all queries
 - **Model Form**: The model is only a linear combination of feature values.
 - Contrast this with genetic algorithms, which can learn functional combinations of features, by randomly introducing operators (e.g. try divide feature a by feature b), but are unpractical to learn

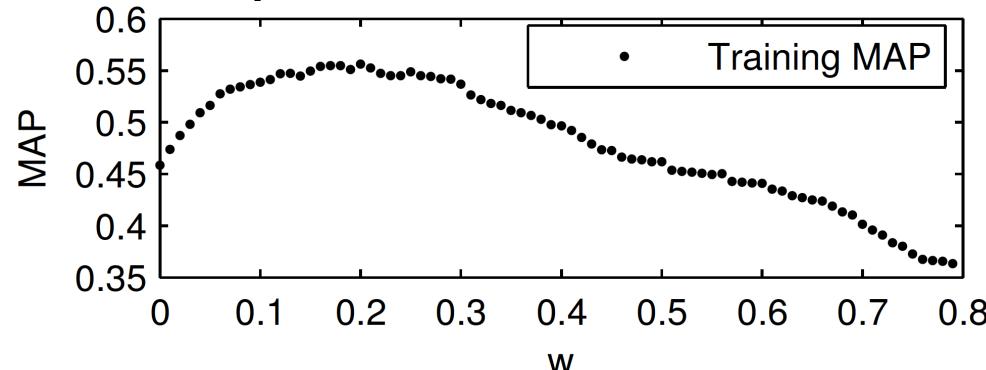
It is difficult to find w_f values that maximise the performance of an IR evaluation metric, as they are non-smooth and non-differentiable

- Typically, techniques such as simulated annealing or stochastic gradient descent are used to empirically obtain w_f values

Difficulty in Training

IR evaluation measures are not continuous, and hence are not differentiable wrt. a feature weight

- i.e. they are not smooth as we vary a feature weight w



Why?

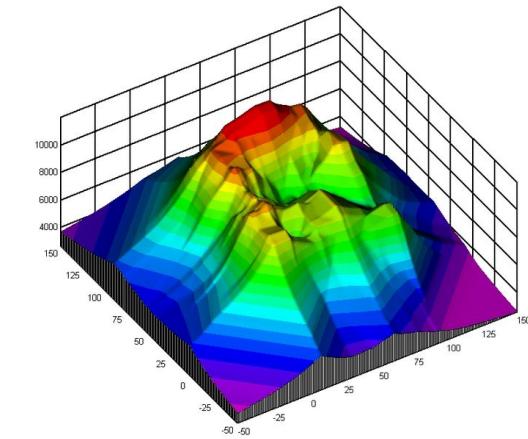


Figure 2: Tuning w , for \log PageRank combination.

Metrics don't respond to changes in the scores of documents, but only when a swap between the ordering of two documents takes place

- We need advanced gradient descent/hill-climbing/simulated annealing
- Many of the advances in learning to rank try to address this

Figures from: (a) Craswell et al., SIGIR 2005 (b) Hongning Wang

(Greedy) Automatic Feature Selection (AFS)

Consider a linear model:

$$score(d, Q) = \sum_f w_f \cdot value_f(d)$$

AFS: Select a $w_f > 0$ for the feature f that best improves an evaluation measure M

	f_1	f_2	f_3	f_4
d_1	0.2	0.1	0.5	0.8
d_2	0.5	0.7	0.0	0.2
d_3	0.3	0.2	0.5	0.0

Step 1: Select the most effective single feature according to M

f1	0.30
f2	0.35
f3	0.05
f4	0.10

→ $(f_2, 1)$

w_1	w_2	w_3	w_4

Current Best $M = 0$

(Greedy) Automatic Feature Selection (AFS)

Consider a linear model:

$$score(d, Q) = \sum_f w_f \cdot value_f(d)$$

AFS: Select a $w_f > 0$ for the feature f that best improves an evaluation measure M

	f_1	f_2	f_3	f_4
d_1	0.2	0.1	0.5	0.8
d_2	0.5	0.7	0.0	0.2
d_3	0.3	0.2	0.5	0.0

Step 2: Select the feature and corresponding w_f that most improves M (e.g. using sim. annealing)

f2 + ? x f1	
f2 + ? x f3	
f2 + ? x f4	

w_1	w_2	w_3	w_4
	1		

Current Best $M = 0.35$

(Greedy) Automatic Feature Selection (AFS)

Consider a linear model:

$$score(d, Q) = \sum_f w_f \cdot value_f(d)$$

AFS: Select a $w_f > 0$ for the feature f that best improves an evaluation measure M

	f_1	f_2	f_3	f_4
d_1	0.2	0.1	0.5	0.8
d_2	0.5	0.7	0.0	0.2
d_3	0.3	0.2	0.5	0.0

Step 2: Select the feature and corresponding w_f that most improves M (e.g. using sim. annealing)

$f_2 + 1.3 \times f_1$	0.36
$f_2 + 0.3 \times f_3$	0.40
$f_2 + 0.9 \times f_4$	0.34

→ $(f_3, 0.3)$

w_1	w_2	w_3	w_4
	1	0.3	

Current Best $M = 0.40$

(Greedy) Automatic Feature Selection (AFS)

Consider a linear model:

$$score(d, Q) = \sum_f w_f \cdot value_f(d)$$

AFS: Select a $w_f > 0$ for the feature f that best improves an evaluation measure M

	f_1	f_2	f_3	f_4
d_1	0.2	0.1	0.5	0.8
d_2	0.5	0.7	0.0	0.2
d_3	0.3	0.2	0.5	0.0

Step 2: Select the feature and corresponding w_f that most improves M (e.g. using sim. annealing)

$f_2 + 1.3 \times f_1$	0.36
$f_2 + 0.3 \times f_3$	0.40
$f_2 + 0.9 \times f_4$	0.34

→ $(f_3, 0.3)$

Step 3: Goto step 2, until we observe no more improvements in M , or no more features

w_1	w_2	w_3	w_4
	1	0.3	

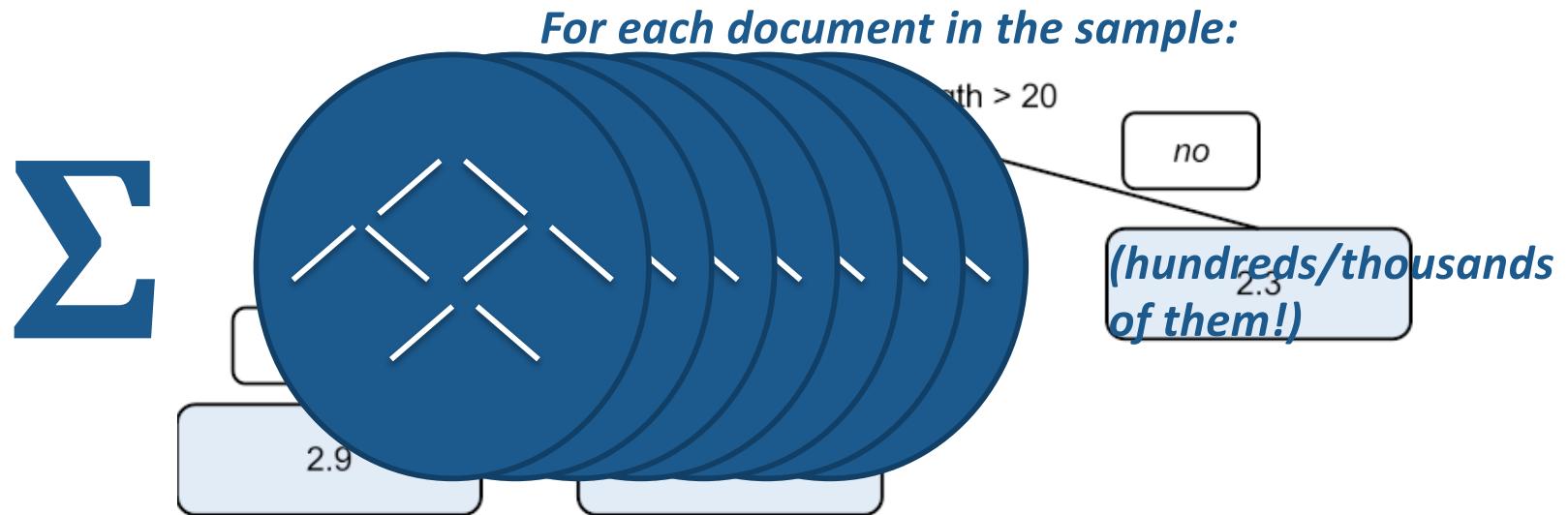
Current Best $M = 0.40$

Type of Learned Models (2)

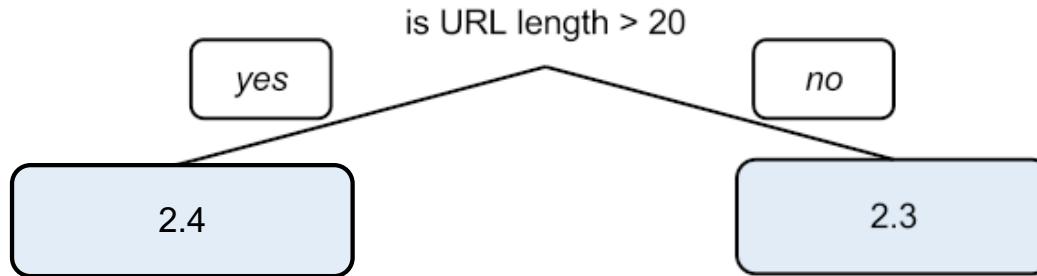
Regression Trees



- A *regression tree* is series of decisions, leading to a partial score output
- The outcome of the learner is a “forest” of many such trees, used to calculate the final score of a document for a query
- Their ability to customise branches makes them more effective than linear models
- Regression trees are **pointwise** in nature, but several major search engines have created adapted regression tree techniques that are **listwise**
 - E.g. Microsoft’s **LambdaMART** is at the heart of the Bing search engine



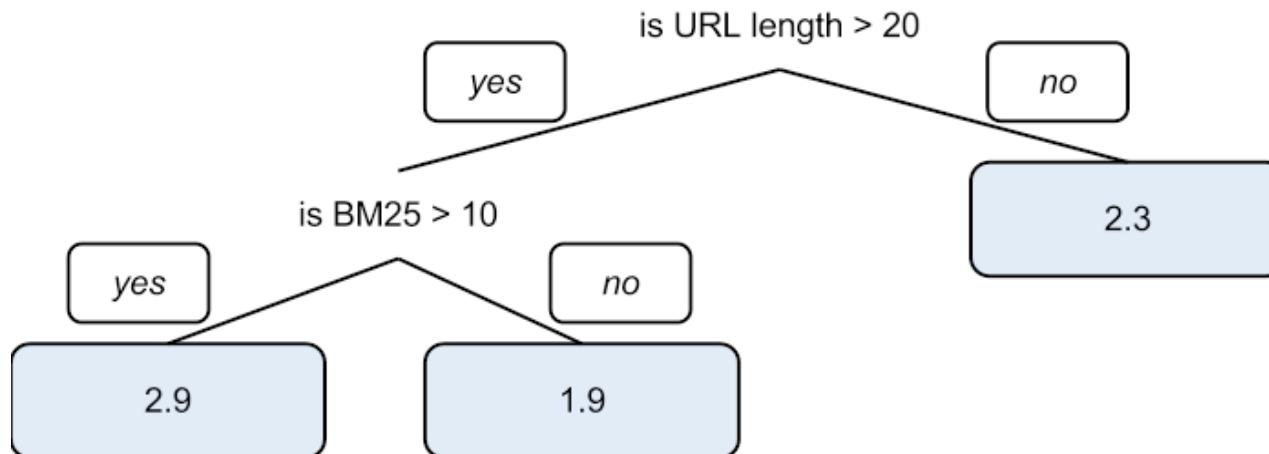
Growing a Tree



	f_1	f_2	f_3	f_4	L
d_1	0.2	0.1	0.5	0.8	1
d_2	0.5	0.7	0.0	0.2	2
d_3	0.3	0.2	0.5	0.0	0

$$\text{Loss} = 1.4$$

Q: Can we add another decision point?



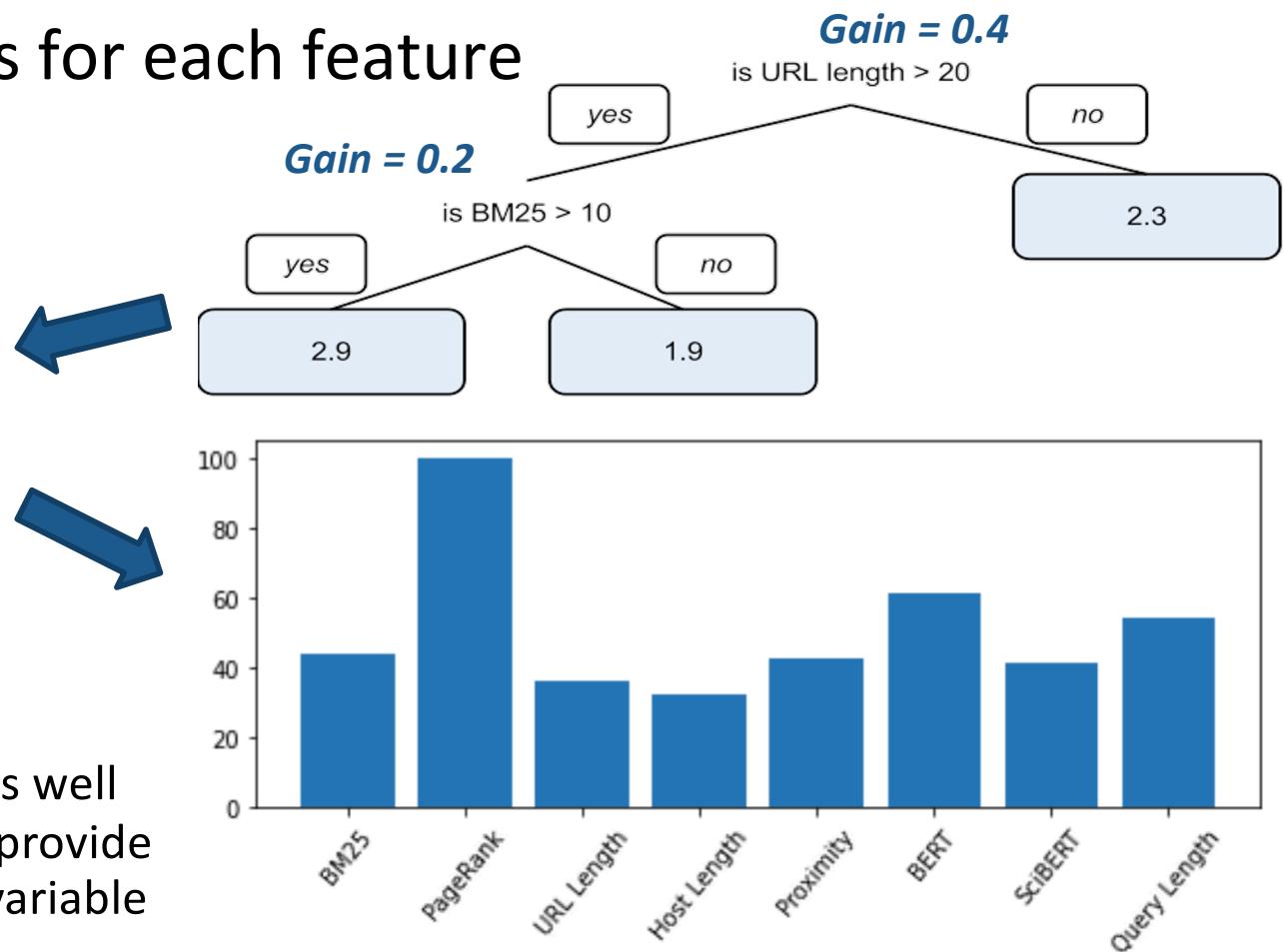
$$\text{Loss} = 1.2 \quad \Rightarrow \text{Gain} = 0.2$$

Feature Importance

Regression Trees have an in-built measure of feature importance

- By recording the “gain” at each node during training time
- Summing the gains for each feature

Feature	Norm. Total Gain
BM25	45
PageRank	100
URL Length	39
...	...



NB: sklearn's RandomForest, as well as XGBoost and LightGBM, all provide a `feature_importances_` variable

Learning and Applying a Learned Model

PyTerrier has transformers that accept standard learners

- Accessed via `pt.ltr.apply_learned_model()`
- E.g. sci-kit learn has a random forest (pointwise) learner

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=400)
rf_pipe = pipeline >> pt.ltr.apply_learned_model(rf)
rf_pipe.fit(train_topics, qrels)
```
- Also for XGBoost and LightGBM, which have LambdaMART (listwise) implementations

```
import xgboost as xgb
lmart = xgb.sklearn.XGBRanker(objective='rank:ndcg')
lmart_pipe = pipeline >> pt.ltr.apply_learned_model(lmart_x, form="ltr")
lmart_pipe.fit(train_topics, train_qrels,
               validation_topics, validation_qrels)
```

- And for FastRank, which has coordinate ascent (linear, listwise) and random forest (pointwise) learners

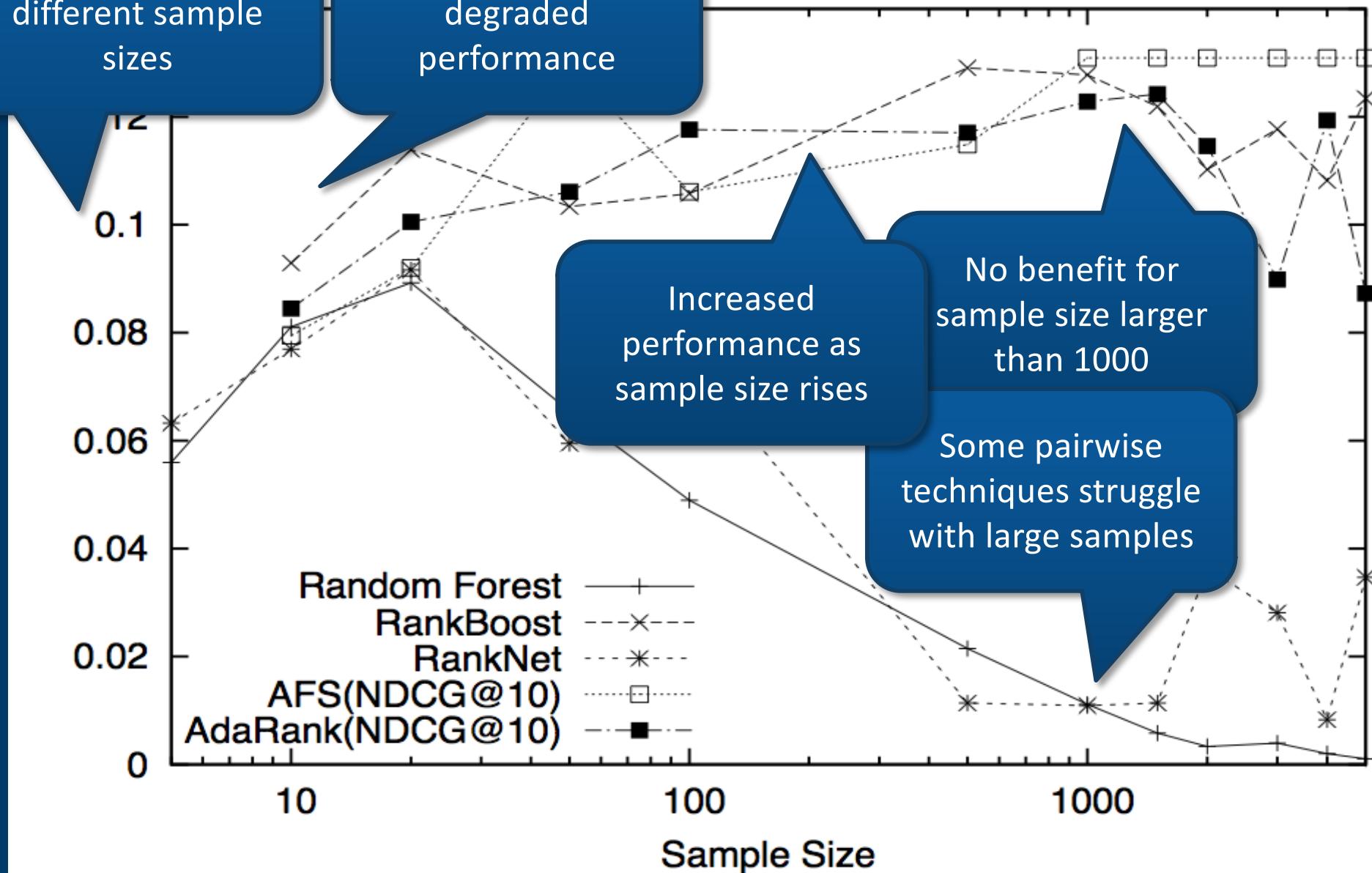
 <https://pyterrier.readthedocs.io/en/latest/ltr.html>



The Importance of Candidate Set Size

NDCG@20 for different sample sizes

Small sample size, degraded performance



Analysing a Model

Many tree-based learners provide a feature_importances_ variable

- This allows the researcher to see the importance of various features during *training*
- (How much each feature was used to reduce the loss during tree construction)

Alternatively, utility on the *test* set can be measured by ranking by each feature alone, or ablating (removing) a feature before learning

- E.g. `pt.ltr.ablate_features(int)` is a transformer that zeros out one or more features.
- This is useful when your pipeline is already defined.

```
# learn a model for 3 features, removing one each time
for fid in range(numf):
    ablated = pipeline >> pt.ltr.ablate_features(fid) >> pt.ltr.apply_learned_model(RandomForestRegressor(n_estimators=400))
    ablated.fit(trainTopics, trainQrels, validTopics, validQrels)
    rankers.append(full)
```

- You will see another analysis in the Part 2 notebook

PyTerrier Learning to Rank Summary



UNIVERSITÀ DI PISA



University
of Glasgow

Key Operators: `>>`, `**`

Key Transformers:

- Retrieval: `pt.BatchRetrieve`
- Features: `pt.apply.doc_score()`, `pt.FeaturesBatchRetrieve`, `pt.BatchRetrieve`
- Learning to Rank: `pt.ltr.apply_learned_model()`
- Analyse: `pt.ltr.ablate_features()`, `pt.ltr.feature_to_score()`

LTR Integrations:

- sklearn: Regression (esp. Random Forests)
- LightGBM: Regression trees/LambdaMART
- xgBoost: Regression trees/LambdaMART
- FastRank: Coordinate Ascent (aka AFS, linear), Random Forests

<https://pyterrier.readthedocs.io/en/latest/ltr.html>



96

LTR Summary



Learning to rank is still an important method of integrating many features within a machine-learned search architecture

- Neural re-ranking models can also fit in this architecture
- Indeed, we found them to be the most important feature for LTR models learned on MSMARCO

PyTerrier aims to make it as easy as possible to express new features, and analyse the generated models

- We include integrations of various LTR libraries
- Tensorflow Ranking is one notable possible future integration, which blurs the lines between LTR and neural re-ranking

References

- [Burges2010] Christopher J.C. Burges. 2010. From RankNet to LambdaRank to LambdaMART: An Overview. Technical Report MSR-TR-2010-82.
- [Chen2016] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In Proceedings of SIGKDD.
- [Foley2019] FastRank alpha, John Foley <https://jjfoley.me/2019/10/11/fastrank-alpha.html>
- [Ke2017] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. Proceedings of NeurIPS.
- [Liu2009] Tie-Yan Liu (2009), "Learning to Rank for Information Retrieval", Foundations and Trends® in Information Retrieval: Vol. 3: No. 3, pp 225-331.
- [Macdonald2012] The Whens and Hows of Learning to Rank. Craig Macdonald, Rodrygo Santos and Iadh Ounis. Information Retrieval 16(5):584-628.
- [Macdonald2012a] On the Usefulness of Query Features for Learning to Rank. Craig Macdonald, Rodrygo Santos and Iadh Ounis. In Proceedings of CIKM 2012.
- [Macdonald2013] About Learning Models with Multiple Query Dependent Features. Craig Macdonald, Rodrygo L.T. Santos, Iadh Ounis and Ben He. *Transactions on Information Systems*. 31(3).
- [Metzler2005] Donald Metzler and W Bruce Croft. 2005. A Markov random field model for term dependencies. In Proceedings of SIGIR.
- [Tonellotto2018] Efficient Query Processing for Scalable Web Search. Nicola Tonellotto, Craig Macdonald and Iadh Ounis. In *Foundations and Trends® in Information Retrieval*. Vol. 12: No. 4-5, pp 319-500, 2018.



University
of Glasgow



UNIVERSITÀ DI PISA



Part 2C

USEFULNESS FOR TEACHING IR

Using PyTerrier for Teaching (during a Pandemic)



UNIVERSITÀ DI PISA



University
of Glasgow

We (Iadh Ounis & I) used PyTerrier for teaching IR this semester

- All exercises were conducted in Google Colab, and hence suitable for online teaching during the pandemic
- Made use of the Dataset API to provide index/topics/qrels (with authentication)
- Students prefer notebooks to commandline interfaces (c.f. Terrier)

Pedagogical appeal

- Full pipeline easily available – indexing, retrieval etc.
- Concepts taught in the course can be easily connected to Python (transformer) classes w/ clear inputs & outputs
 - e.g., a pseudo-relevance feedback component operates after an initial retrieval, and before another ranking component is deployed (final ranking of documents).

Practical Courseworks (1/2)

Our practical assessments balance ‘programming independence’ with easily markable assessments that scale to large classes

Coursework 1:

- Perform indexing & retrieval; PyTerrier operators
- Conduct experiments on several query sets, using several classical and probabilistic IR weighting models, e.g., TF-IDF, BM25, PL2, with and without PRF, including applying significance testing, interpolated precision-recall graphs, per-query performance analysis
- Implement a word2vec based QE technique and compare to classical

{ pt.DFIndexer()
pt.BatchRetrieve()

pt.BatchRetrieve()
Bo1QueryExpansion()
pt.Experiment()

pt.apply.query()

Most of the exercise has known correct answers can be easily automatically marked in a quiz

Practical Courseworks (2/2)

Coursework 2:

- Train a learning to rank model in a web search setting
 - Analyse feature importance
 - Define and implement features for URL length and query term proximity
 - Conduct experiments, perform significance testing & failure analysis
- } pt.ltr.*
} pt.apply.doc_score()
} pt.Experiment()

Overall feedback:

- PyTerrier builds on familiar technologies: Python, Pandas
- Easy to design and implement new techniques / features
- Students who embrace declarative thinking will write elegant clear code
- Providing template notebooks helped students to get going



University
of Glasgow



UNIVERSITÀ DI PISA



Part 2D

WRAPUP

Achievements in Part 2



We have **reviewed modern IR architectures**, namely...

- ...how existing IR techniques fit into the PyTerrier datamodel and transformers
- ...how transformer pipelines can be created using operators
- the learning-to-rank architecture, as well as common learners, and how to easily create features using LTR

Finally, we also discussed the usefulness of PyTerrier for teaching
In the practical session, you will have a chance to **experience** these
techniques in practice on TREC Covid test collection.

In Part 3, we will **review** contemporary IR re-ranking architectures based on neural networks, as well allowing you to **experience** these through PyTerrier



University
of Glasgow



UNIVERSITÀ DI PISA



QUESTIONS?

Practical Time



The tutorial Github repo has links to the notebook for **Part 2**

- <https://github.com/terrier-org/ecir2021tutorial>
- Press the  Open in Colab link for each notebook to start a Colab session

Timings:

- Practical – in breakout rooms – until 1300
- Lunch break 1300-1415
- Part 3 resumes at 1415

If you are leaving us here, please complete our feedback quiz <https://forms.office.com/r/2WbpLiQmWV>