IE 663 - Information Retrieval & Web Search
Implementation of an Efficient Vector Space Retrieval System

Alexander Weiß
Nick Weber
Nicolas Wipfler
University of Mannheim
Schloss
Mannheim, 68131, Germany
{alweiss, nickwebe, nwipfler}@mail.uni-mannheim.de

Abstract

In this work we present the implementation of an information retrieval system based on the vector space model (VSM). We introduce various runtime performance improvements compared to vanilla VSM and discuss their impact on retrieval quality. We show that using a tiered index in combination with random projections results in the best tradeoff between runtime and retrieval performance. The improvement with the fastest runtime is the pre-clustering approach with random projections, but its retrieval performance is significantly worse than that of the tiered index. As expected, vanilla VSM has the best retrieval performance but lacks behind in runtime performance. Using word embeddings did not meet the expectations, as it did not improve retrieval quality.

1 Introduction

In times of big data and the Internet of Things, a very large amount of data is accumulated every day. For businesses and their desire for real time business intelligence, it is crucial to be capable of managing and analyzing this data efficiently. Beyond that, data volumes in the range of petabytes become common today in sciences such as astronomy.[1] In order to retrieve valuable knowledge from this data, information retrieval techniques must be both precise and fast. Especially in the world wide web a lot of data is produced in the form of text by billions of users of social networks, blogs or wikis. Search engines use various dedicated techniques in order to retrieve and rank documents for users according to their information need. One popular example of these techniques is the vector space model(Salton et al., 1975). In VSM, text documents are represented as vectors of terms and the similarity of documents is judged by computing the angle between their vector representations. Search engines make use of this by transforming user queries into this vector representation and computing the similarity between query and document vectors. After computing all similarities, the most relevant documents for the user's information need are retrieved and ranked. In this work we present the implementation and evaluation of an information retrieval system for medical research documents using VSM. Our work is based on the data set for medical information retrieval published by the Statistical NLP Group at the University of Heidelberg. The data set is used in their work (Boteva et al., 2016) to investigate learning to rank in the medical domain. In the following work we address potential performance improvements for the classic VSM and give a recommendation on what to use. The remainder of this work is organized as follows: Implementation details are discussed in Section 2. Section 3 presents the methodology and an evaluation of our results. In Section 4 we compare our work with related projects. Section 5 contains a summary and an outlook for future work.

2 Implementation

In this section we discuss the basic architecture and various implementation details of our system. In Section 2.1 the core system, an implementation of the traditional vector space model (in the following referred to as *vanilla* VSM), is presented. Besides providing the fundamental infrastructure for all further performance improvements, it also serves as a baseline for the evaluation. In sections 2.2 to 2.4 we discuss the performance im-

[1] Astronomy in the Big Data era

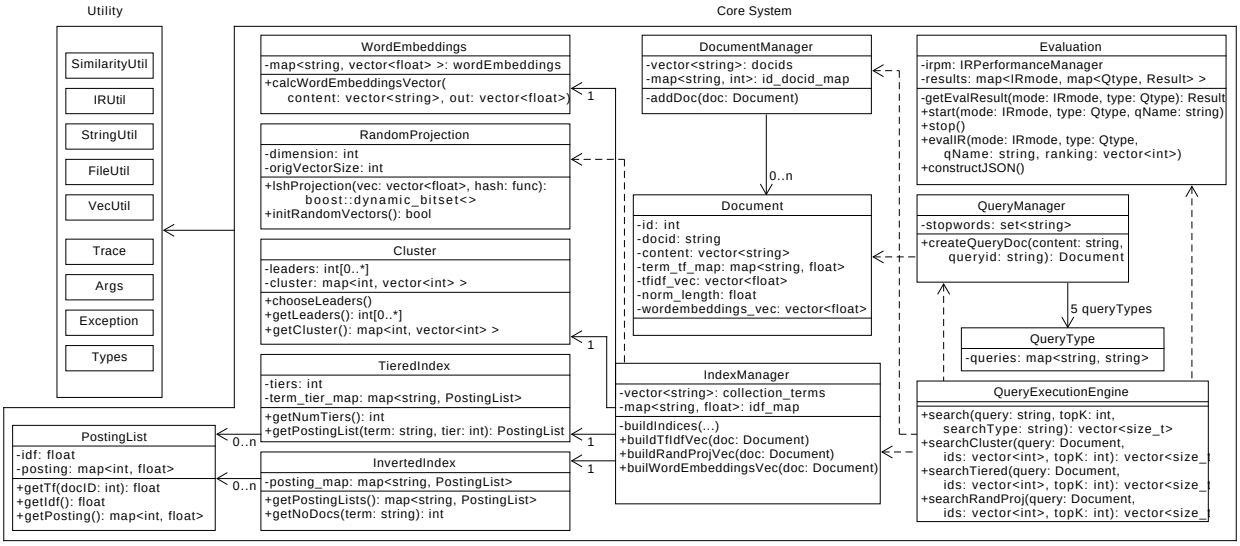

Figure 1: Structure of the Core System

provements *Tiered Index*, *Pre-Clustering* and *Random Projections* respectively. In order not to depend on term matching only, we tried to utilize word embeddings as described in Section 2.5. Last but not least, in Section 2.6 we discuss implementation details of the evaluation. In order to satisfy the goal of good runtime performance, we decided to implement the system in C++. For further implementation details you can visit the project on GitHub.

2.1 Core System

The basic architecture of the system is depicted in Figure 1. The five classes *Document*, *DocumentManager (DM)*, *QueryManager (QM)*, *IndexManager (IM)* and *QueryExecutionEngine (QEE)* implement the core logic of the system. Objects of the class *Document* are an in-memory representation of text documents and queries. This makes it easy to process the content of a document and is way faster than reading the data from disk each time. Internally, documents are uniquely identified by an integer ID. The original data collection consisted of multiple text documents stored within a single text file separated by line breaks. Each line (i.e., each document) in the collection is associated with a string of the form MED-X (for queries PLAIN-X), where X is some integer. This string serves as an "external" ID and also uniquely identifies a document. On system start the text file is parsed and each line is stored as a separate document object. The singleton class *DocumentManager* maintains all document objects in a hash table. Documents in the hash table are accessed by using their document ID as key. Similar functionality but for queries is implemented in the *QueryManager*. The *Index Manager* maintains several index structures to speed up retrieval. Since a term-document matrix is very sparse, the *IM* maintains an inverted index to store term incidences efficiently. Each corpus term is now represented by a posting list. It contains all the document IDs in which the term appears in ascending order. The *IM* also creates, initializes and manages the performance improvements introduced in the following sections. As already indicated by its name, the class *QueryExecutionEngine* is responsible for executing queries and finding the most similar documents to return.

2.2 Tiered Index

Taking the idea of the inverted index further, a more sophisticated index structure divides its posting lists hierarchically into tiers of decreasing importance. The organization of posting lists in different tiers gives the index structure its name, tiered index. When retrieving documents the tiered index lookup is done by merging the term postings of the top tier. In case this results in too few hits, it continues to merge lists of lower tiers. Since relevant documents are retrieved first, this enables a faster lookup and reduces runtime in case only few results (i.e., a low top-k) are necessary. It is configurable how many tiers are used

but this number must be in the interval $[2, \infty[$. The tiered index is built from the basic term postings. Document IDs are sorted descending by term frequency and are then assigned to the different tiers, whereas the resulting tiers are again sorted ascending by document ID. In the following an example is presented with $tiers = 3$ and the term posting: "Frodo" $\rightarrow [1, 2, 5, 8, 9, 11, 99]$. Suppose the descending sort by term frequency results in the list $[5, 1, 99, 2, 8, 9, 11]$. Finally, the tiered term posting for "Frodo" looks like the following: "Frodo" $\rightarrow$ [T0: $[1, 5]$, T1: $[2, 99]$, T2: $[8, 9, 11]$].

2.3 Pre Clustering

The cluster is initialized in two steps by the *IM*. First, the random leaders are selected. By this, $\sqrt{N}$ (where N is the number of documents in the collection) random documents are chosen as leaders. As each leader is itself a document and represents a cluster, a cluster can be identified by its leader's ID. After leader selection, each document in the collection is compared with each leader and assigned to the cluster of the most similar leader. This cluster is implemented as a hash table. Key of the hash table is the leader's ID and the associated value (i.e., one cluster) is an array of document IDs. When retrieving documents with the clustered index, the *QEE* compares a query with each leader document. The ID of the most similar leader is used to access the cluster's hash table and retrieve document IDs in the respective cluster. These IDs can then directly be used to access and retrieve the documents managed by the *DM* (c.f. Section 2.1). If the number of documents in one cluster is lower than the requested number of results, the next most similar cluster and its associated documents are retrieved, respectively.

2.4 Random Projections

In order to improve the runtime performance of the system, we additionally implemented *Random Projections* as part of the family of *Locality Sensitive Hashing Methods*. To accomplish this, on system start a *dimension* variable is set, indicating to what dimension (d) the TF-IDF (size n) vectors are reduced. Afterwards, $d \cdot n$ dimensional vectors containing normal distributed floating point numbers are generated, which are then used to project the TF-IDF vectors into the d-dimensional space. In order to decide whether a position in the random projection vector is assigned the value 0 or 1, we are using the following hash function h

$$h(x) = \begin{cases} 1 & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where x corresponds to the scalar product of the TF-IDF vector and random projection vector. The random projection vectors are built for every indexed document as well as for all queries. This allows the system to switch flexibly between document retrieval with random projections and other retrieval modes at runtime, without the need to be restarted. However, the dimension d of the vectors must be set on system start and can not be changed afterwards.

2.5 Word Embeddings

So far, we merely discussed improvements to the VSM itself. One main drawback of VSM is that in general it just compares the term overlap between a query and the documents being searched. Thus, documents which may be semantically related to a query may receive a low relevance score according to cosine similarity, if they have few or no words in common. In order to resolve this issue, we incorporated word embeddings into the TF-IDF vectors of our documents and queries. As it is very time consuming to build reliable word embeddings, we used the *GloVe - Global Vectors for Word Representation* project from the Stanford University (Pennington et al., 2014). This project provides several word embedding models for free usage. We used a model based on *Wikipedia* and *Gigaword-5* (a comprehensive archive of newswire text data) which provides 400k words with 300-dimensional word embedding vectors. In order to incorporate the semantic meaning of a document into its basic TF-IDF vector, we simply iterate over all terms of a document and retrieve the respective word embedding vector. Once all vectors are retrieved, they are averaged and stored as the "document word embedding" which is subsequently appended to the raw TF-IDF vector. It is used when invoking a search mode which uses word embeddings. The impact of extending the system with word embeddings is discussed in Section 3.3.

2.6 Evaluation Implementation

In order to evaluate performance of the models (i.e., the retrieval modes in Table 1), the class *Evaluation* is used. It provides functionalities for measuring runtime and retrieval performance of

executed queries. By this, we are comparing the impact of the different models. To assess runtime performance, we measured the time it took for each query to execute. This runtime measurement is subsequently stored and associated with the current used retrieval model. To evaluate retrieval performance, the produced ranking is compared with the query relevance scores of the gold standard in order to compute evaluation metrics such as *MAP* and *nDCG* (cf. Section 3). The respective retrieval performance measurement is also stored and associated with the current used retrieval model. After all queries have been executed, a structured JSON file is created from all measurements. We parse this file with external tools and use it in the following section to evaluate the system.

3 Evaluation

In this section we discuss how the various performance improvement techniques for VSM influence runtime and retrieval performance. Section 3.1 introduces our procedure of measuring performance. We take a look at the runtime of vanilla VSM in comparison to its improvements in Section 3.2. Finally, we discuss the impact of these improvements on the retrieval performance in Section 3.3.

3.1 Methodology

Table 1 shows the different combinations of performance improvements. In the following sections, we first asses the models in their basic form (i.e., vanilla VSM, tiered index and pre-clustering) without any modifications. Next, we combine the models with random projections in order to reduce dimensions of the document vector representations. Lastly, we discuss the influence of word embeddings on the respective performance measurement.

Before evaluating the performance differences between these models, we determined the best parameter setting for each model (i.e., the number of tiers and dimensions). We set the parameter top-k to 20, which is the requested number of results for each query. For the best performing number of tiers, we found that an increase of up to 100 tiers improves runtime performance significantly with no negative impact on retrieval performance. This is why we chose 100 tiers as the default setting for the tiered index. Therefore, approximately 35 document IDs are stored in each tier. A low number of dimensions (i.e., 500) had a positive impact on runtime performance but on the other hand resulted in bad retrieval performance. Increasing the number of dimensions from 500 to $5,000$ improved the *nDCG* (cf. Section 3.3) significantly but also increased the runtime by merely 7% as depicted in the following table.

Vanilla	**Random Proj.**	**Word Emb.**
VSM	VSM_R	VSM_W
TieredIndex	TieredIndex_R	TieredIndex_W
Clustering	Clustering_R	Clustering_W

Table 1: Different combinations of performance improvement techniques

Dimensions	500	$5,000$	$10,000$
Runtime	2.8ms	3ms(+7%)	4ms(+33%)
nDCG	0.28	0.43(+54%)	0.49(+14%)

Table 2: Average query runtime and *nDCG*

A further increase of dimensions to $10,000$ had a much higher negative impact on the runtime while the retrieval performance was increasing only a little. Tests for the best number of dimensions in an interval of $[500, 10000]$ revealed that $5,000$ dimensions for the random projection results in a good tradeoff between runtime and retrieval performance.

After investigating the best settings for random projections and tiered indexes, we started to evaluate the performance of the models. Before we are going into more detail on this subject in Section 3.2 and 3.3, we will present our approach of gathering measurement data in the following. Since we have nine different models to evaluate (cf. Table 1) and five different query types (cf. Table 3), there are already $9 * 5 = 45$ combinations to evaluate. For each of these combinations we evaluate all the queries. In total this yields $9 * 9,935 = 89,415$ evaluation results which we store in a JSON file for further processing with external tools. We observed that the different query types generally do not differ in their performance, thus we aggregated the evaluation results and grouped them on the models under investigation. Thereby we reduced the data to just 9 results, one for each model. These results are discussed in the following sections.

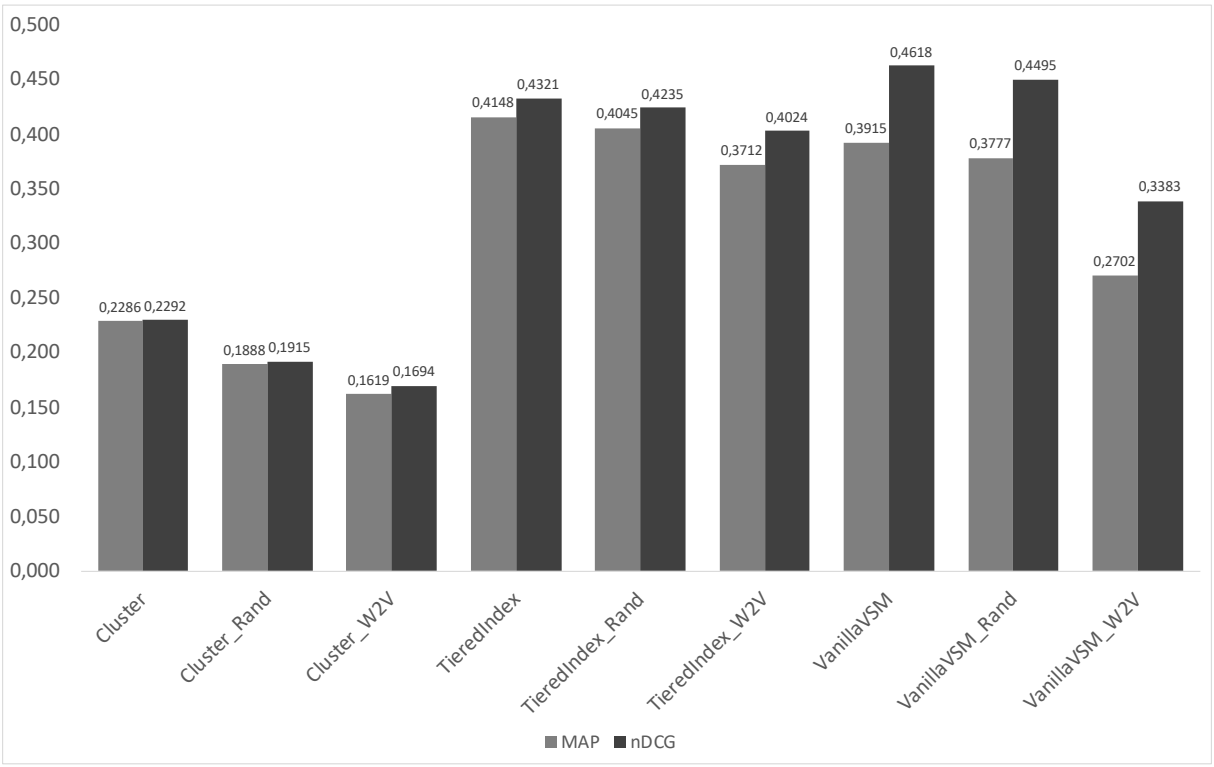

Figure 2: Retrieval performance for the various models

Query Types	# of Queries
all	3,237
titles	1,429
non-topic titles	3,237
video description	1,016
video titles	1,016
Total	9,935

Table 3: Query types and their number of queries

3.2 Runtime Performance

A crucial part of efficient vector space retrieval is the runtime performance. In this section we will discuss the runtime performance of the models in Table 1.

Runtime performance is measured as the average runtime of a query, i.e. the time it takes to execute the search for this query including document retrieval and ranking. Compared to its vanilla form, VSM with pre-clustering or tiered indexes executes a query significantly faster. This is outlined in Table 4. Numbers in this table are average runtimes per query. On average the tiered index yields a runtime of 8.2ms which is about 16 times faster than vanilla VSM. Pre-clustering achieves a runtime improvement by factor 8.5 compared to vanilla VSM. However, the use of random projections drastically improve the runtime of every retrieval technique and is therefore recommended. It is especially impressive that by the use of random projections, the runtime of vanilla VSM is improved by a factor of 32 and even by a factor of 77 for pre-clustering. Although the runtime of models with word embeddings were also measured, they are excluded from the runtime evaluation as we focus on runtime improvements and word embeddings were significantly slower than vanilla VSM. The following section addresses at what cost these runtime improvements come in terms of retrieval performance.

	w/o RP	with RP
Cluster	15.4ms	0.2ms
Tiered Index	8.2ms	0.3ms
Vanilla VSM	130.5ms	4ms

Table 4: Analysis of runtime improvement when using random projections (RP)

3.3 Retrieval Performance

For each model we used *Mean Average Precision (MAP)* and *Normalized Discounted Cumulative Gain (nDCG)* as a evaluation metric. The following formulas were used to calculate the respec-

$$MAP = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} P(R_{jk})$$

$$nDCG(k) = \frac{\sum_{i=1}^{k} \frac{2^{rel_i}-1}{log_2(i+1)}}{\sum_{i=1}^{|relevant|} \frac{2^{rel_i}-1}{log_2(i+1)}}$$

Figure 3: Formula to calculate *MAP* and *nDCG*

tive metric. For the *DCG* we chose a formula that produces a stronger emphasis for retrieving relevant documents. Both *MAP* and *nDCG* were calculated on the top-k documents and are therefore approximate values. As *MAP* values are mainly determined by results appearing at the top of the ranking, the impact of this approximation should be minimal (cf. Section 4). The same idea applies to the *nDCG* score. In Figure 2 we present our retrieval performance results for each model. First of all, we can observe that the pre-clustering approach is always the worst in terms of retrieval performance. As its runtime performance is similar to the tiered index, we do not recommend to use pre-clustering as VSM improvement. However, pre-clustering was straightforward and quick to implement and could therefore be an option in case retrieval performance is not an issue. The second observation is that the tiered index is very similar in retrieval performance compared to vanilla VSM. In the case of the basic tiered index and basic VSM (i.e., without random projections or word embeddings), performance in terms of *MAP* of the tiered index is higher. Reasons for this are discussed in more detail in the following section. As the tiered index has a way faster runtime, we recommend to use the tiered index with the presented settings in Section 3.1. It achieves the best possible runtime with a minimal retrieval performance loss. We did not discuss random projections regarding retrieval quality yet. In Section 3.2 we already discussed the runtime improvement of using random projections. It is very interesting to see the impact on retrieval performance in Table 5.

While using pre-clustering with random projections has a high loss in retrieval performance (-16% *nDCG*), the loss for vanilla VSM and tiered index is negligible. Therefore, we recommend to use vanilla VSM and tiered indexes in combination with random projections as the retrieval performance loss is minimal while the runtime improvements are significant (cf. Section 3.2).

	w/o RP	with RP
Cluster	0.229	0.192 (-16%)
Tiered Index	0.432	0.424 (-2%)
Vanilla VSM	0.462	0.450 (-3%)

Table 5: Analysis of *nDCG* loss when using random projections (RP)

In order to improve the shortcomings of VSM, namely the plain focus on term overlap and the lack of semantic features, we integrated word embeddings support for our system. As already mentioned, the word embeddings integration did not meet the expected results. Instead of improving retrieval performance due to the additional semantic features, retrieval quality got worse. In the following we discuss reasons for that.

The pre-trained word vectors are integrated into the TF-IDF vectors by taking the word embeddings vector of every term from the query or document. Subsequently, these vectors are averaged in order to obtain a single vector associated with the query or document, respectively. At the time this vector is computed the documents are already preprocessed. Hence, it happens that words with the same root are erroneously associated with the same word embeddings vector of this common root instead of their real word vectors. An example for this would be the final word vector of the terms "*house*" and "*houses*". Although both are originally associated with distinct word vectors, they are assigned the vector of their common root "*hous*". This is one major drawback of our approach, since it conceals the real pre-trained word vectors with a more general vector of the stemmed term. Apart from that, the used word embeddings are too broad as they are not domain specific. Thus, the word embeddings could not improve the performance. Some potential improvements are discussed in Section 5.1.

4 Related Work

In this section we compare our retrieval results with the related work of (Boteva et al., 2016). As they provide a *MAP* and *nDCG* measurement, this is a good fit for comparing results, since we also evaluate our system based on these metrics. The first column in Table 6, named "VSM (Boteva et

Query Types	VSM (Boteva et al.)		Vanilla VSM		Tiered Index	
	MAP	*nDCG*	*MAP*	*nDCG*	*MAP*	*nDCG*
all	0.136	0.393	0.421	0.537	**0.479**	**0.545**
titles	0.123	0.258	0.402	**0.470**	**0.426**	0.464
non-topic titles	0.097	0.285	0.373	**0.422**	**0.395**	0.394
video description	0.111	0.351	**0.386**	**0.453**	0.376	0.358
video titles	0.101	0.287	0.372	**0.424**	**0.396**	0.397

Table 6: Comparison of retrieval quality results. Bold entries indicate the best performance for a type

al.)", shows the results of the VSM approach presented in the work mentioned above. For comparison, in column two and three we present our results for vanilla VSM and tiered index. As the second column demonstrates, our vanilla VSM achieves the best *nDCG* scores. However, our tiered index approach results in better *MAP* scores. We try to give a reason for this in the following.

MAP does not take into account graded relevance scores of the documents but only considers the fact whether a document is relevant to the query at hand or not. In the tiered index, less documents are retrieved than in vanilla VSM. Therefore, we assume that some irrelevant documents are retrieved by vanilla VSM but not by the tiered index. If some of these irrelevant documents are ranked sufficiently high by vanilla VSM, this results in a decline of the *MAP* score.

It did not become clear why the baseline TF-IDF retrieval system of (Boteva et al., 2016) performed significantly worse than the other systems in their work. In comparison to their results, our vanilla VSM and even tiered index approach has a considerably performance improvement, especially for the *MAP*. Unfortunately, their work does not go into much detail on this subject. As we evaluate retrieval performance on the top-k results only, this has an impact on the scoring metrics. However, this impact is usually minor as, for example, values of the *MAP* are determined mainly by the results at the top of the ranking anyway.

Unfortunately, we could not find any related work focusing specifically on runtime performance improvements for VSM. Most projects dedicated to improving VSM intend to improve its retrieval performance by using sophisticated techniques such as machine learning(Van Gysel et al., 2017; Dali et al., 2010). Since we were unable to improve retrieval performance of VSM by integrating word embeddings, related work on this topic is not discussed.

5 Conclusion

In this work we investigated the tradeoff between runtime and retrieval performance of an information retrieval system. Our system is based on the vector space model, using different retrieval speedup techniques. First of all an overview of the core system and our implementation of the VSM improvements was given. Therefore, we presented our engineering approach and introduced various implementation details of the system. Next, the evaluation of performance measurements for the models were discussed. We introduced our methodology and described the procedure of finding the best settings for the tiered index and random projections. The reason for aggregating the different query types into one measurement was discussed before taking a detailed look at the runtime performance. We showed that the tiered index and pre-clustering approach both significantly improve runtime performance over vanilla VSM. Combined with random projections this improvement is again drastically increased. When investigating the retrieval performance, we found that the tiered index hits the best possible tradeoff between runtime and quality of retrieval. While the pre-clustering approach is fast, its retrieval performance is unacceptable. One advantage of clustering is its straightforward implementation. However, this should not justify its utilization over the other techniques. We recommended to use the tiered index combined with random projections for a fast runtime with minimal retrieval quality loss. Improving VSM by incorporating latent semantic features with word embeddings did not meet our expectations as retrieval performance turned out to become worse. Last but not least we discussed retrieval results in our system in comparison to related work.

5.1 Future Work

There is still room for improvement of the system. Our main focus for future work are improvements regarding the general methodology of integrating word embeddings into the models. Since the semantic feature vectors, based on word embeddings, are built based on the preprocessed document collection, terms are assigned the word vectors of their morphosyntactic root. This produces word vectors which are too generic and do not reflect the correct semantic meaning of the term. In order to tackle this problem, the semantic word vectors of documents should be built before the document collection is stemmed during preprocessing. Moreover, as already addressed in Section3.3, a domain specific word embedding is likely to improve retrieval performance a lot. Thus, a dedicated word embedding with a focus on medical context should be built. Lastly, the construction of word vectors for documents can be adjusted to be computed as the weighted average of the word vectors based on the inverse document frequency values. This reflects the domain specific importance of the word vectors more accurately.

Another important aspect is the possible integration of machine learning methods for the sake of learning to rank. Supervised machine learning techniques offer the possibility to learn output labels from input vectors. This can be especially helpful in information retrieval for learning how to rank a set of documents based on a rich resource of features (i.e., the optimal ranking of documents for a given query). Training a classifier based on this ranking and other features enables the construction of a document ranking for a given query that performs better than traditional methods focusing mainly on term overlap.

References

[Boteva et al.2016] Vera Boteva, Demian Gholipour, Artem Sokolov, and Stefan Riezler. 2016. A full-text learning to rank dataset for medical information retrieval. In *European Conference on Information Retrieval*, pages 716–722. Springer.

[Dali et al.2010] Lorand Dali, Bla Fortuna, and Jan Rupnik. 2010. Learning to rank for personalized news article retrieval. In Tom Diethe, Nello Cristianini, and John Shawe-Taylor, editors, *Proceedings of the First Workshop on Applications of Pattern Analysis*, volume 11 of *Proceedings of Machine Learning Research*, pages 152–159, Cumberland Lodge, Windsor, UK, 01–03 Sep. PMLR.

[Pennington et al.2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Salton et al.1975] G. Salton, A. Wong, and C. S. Yang. 1975. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November.

[Van Gysel et al.2017] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2017. Neural vector spaces for unsupervised information retrieval. 08.