Search Engine & Web Mining HW#4

Name: Yitong Zhou ID: yitongz

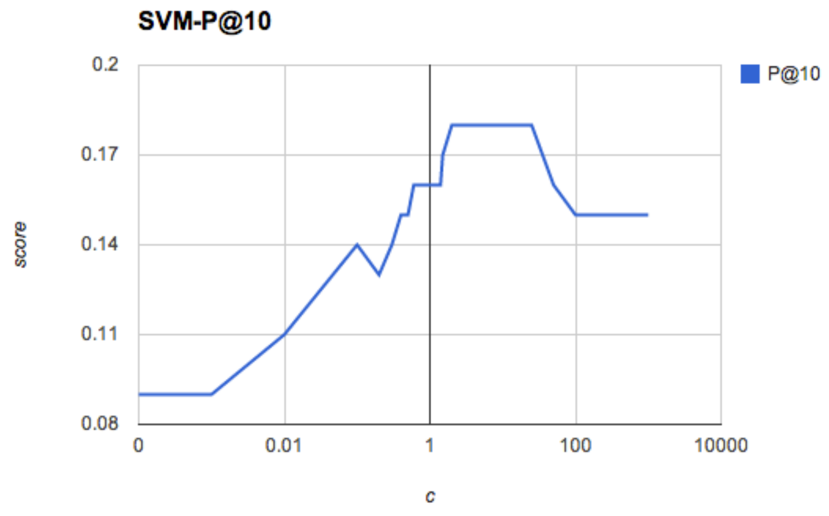
1 Experiment Result & Analysis

1.1 SVM Result:

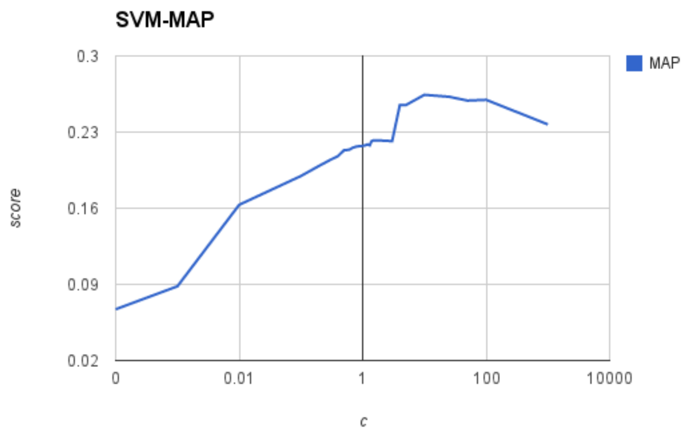
The original test data result is as below:

c	P@10	MAP	NDCG@10	Time	c	P@10	MAP	NDCG@10	Time
0	0.18	0.26149632	0.336131202	31	1.1	0.16	0.217830536	0.284522811	39
0.0001	0.09	0.067098654	0.122624833	578	1.2	0.16	0.218657176	0.285024508	32
0.001	0.09	0.088203405	0.139316831	540	1.3	0.16	0.217840969	0.284522811	45
0.01	0.11	0.163146417	0.193052188	376	1.4	0.16	0.221749234	0.294013619	42
0.1	0.14	0.189626974	0.248680232	194	1.5	0.17	0.222433451	0.300509902	54
0.2	0.13	0.199142216	0.242814358	166	2	0.18	0.222212673	0.306238902	33
0.3	0.14	0.204489535	0.25167943	93	3	0.18	0.221693076	0.307752558	27
0.4	0.15	0.208004418	0.268771991	59	4	0.18	0.254851349	0.32620607	35
0.5	0.15	0.213345532	0.273714813	115	5	0.18	0.254811804	0.32620607	34
0.6	0.16	0.213697518	0.282198447	97	10	0.18	0.264278679	0.338322389	31
0.7	0.16	0.215745387	0.283342139	61	25	0.18	0.262490349	0.340446254	33
0.8	0.16	0.216739288	0.28411752	92	50	0.16	0.258971301	0.318559033	34
0.9	0.16	0.217111976	0.28411752	102	100	0.15	0.259671342	0.312777835	34
1	0.16	0.217277868	0.28411752	80	1000	0.15	0.237050521	0.306826466	152

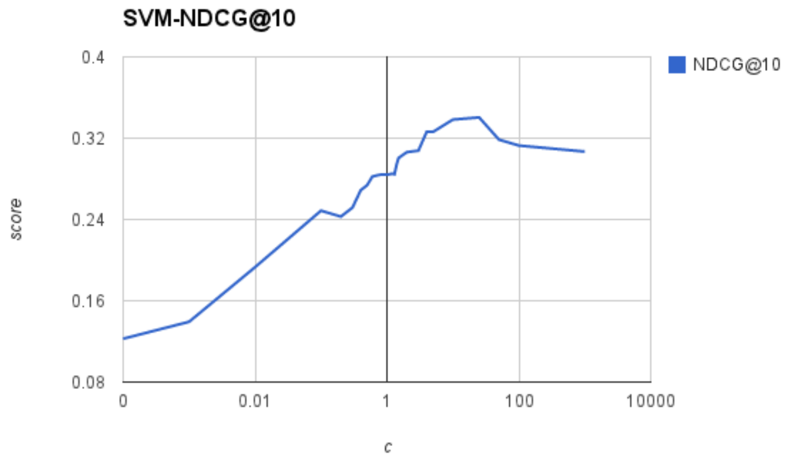
Plots:



Graph 1-1: SVM scores of P@10



Graph 1-2: SVM scores of MAP



Graph 1-3: SVM scores of NDCG@10

Analysis and Discussion:

1) Best Performance with c around 10

The experiment results indicates that with the value of c increasing, the P@10, MAP and NDCG@10 will all increase gradually, they all reach the maximum value with a c value of 10 around, the score begins to jump.

2) A singleton point when c=0

It is also interesting to notice that with a very small c very near to 0, the performance score will be quite bad. (With a MAP of 0.067 when c is 0.0001). And it is quite strange that that when c is exactly 0, it can reach quite good performance, this may be resulted from the implementation of the SVM_Light, which may totally do not use regularized term when c is exactly 0.

3) General change trend

For P@10, MAP and NDCG@10, they typically increase with c's value increase. But the increase is also a little flexible with some middle c values suddenly decrease or increase the performance.

4) Other points

- P@10 is a quite discrete score with strong jump or drop. The disadvantage of this pattern is that we cannot rely on P@10 to recognize the best performance point. However, the

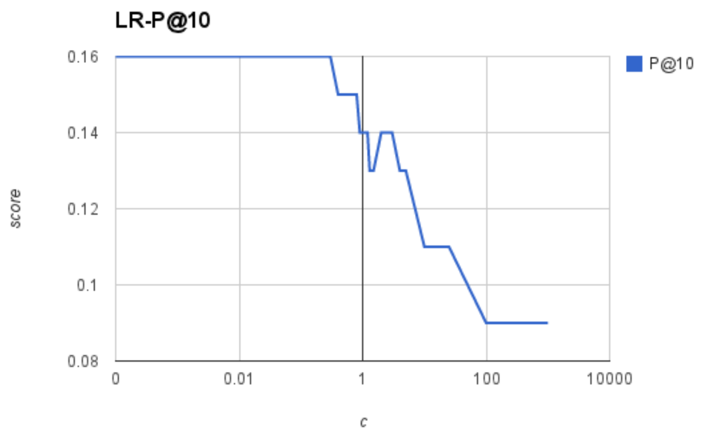
P@10 also stays unchangeable during the “Plateau Time”, so it can be very effectively used to identify the best performance c range.

1.2 Logistic Regression Result:

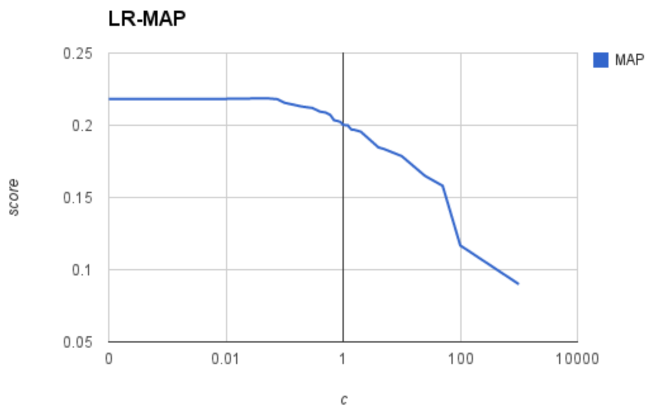
The original experiment result is as below:

c	P@10	MAP	NDCG@10	Time	c	P@10	MAP	NDCG@10	Time
0	0.16	0.218146485167437	0.28370420656185	437	1.1	0.14	0.200069091692988	0.257307080741629	190
0.0001	0.16	0.218149657957588	0.28370420656185	439	1.2	0.14	0.200017471559643	0.257307080741629	183
0.001	0.16	0.218148470026964	0.28370420656185	432	1.3	0.13	0.198214605799651	0.24225558095843	175
0.01	0.16	0.218295142288774	0.28370420656185	427	1.4	0.13	0.19688571347948	0.241310696218308	167
0.025	0.16	0.218518304186034	0.28370420656185	426	1.5	0.13	0.196994727460763	0.241310696218308	160
0.05	0.16	0.21868897349306	0.284287433360506	413	2	0.14	0.195579004811195	0.243913383326021	141
0.075	0.16	0.218073697387595	0.283852113583818	389	3	0.14	0.189224797388778	0.248131570809865	115
0.1	0.16	0.215581523155044	0.282368360466672	376	4	0.13	0.184710782181998	0.228721399768129	99
0.2	0.16	0.212922528926237	0.280865682483251	332	5	0.13	0.183542452205855	0.227441680080697	86
0.3	0.16	0.211955392212819	0.280089270983756	309	10	0.11	0.178676970500374	0.20737203290783	57
0.4	0.15	0.209474184299167	0.270338471759824	284	25	0.11	0.164953054772106	0.195643523152428	31
0.5	0.15	0.208849534589755	0.270338471759824	262	50	0.1	0.158078773706637	0.182080789974469	18
0.6	0.15	0.207272682858746	0.269755244961169	247	100	0.09	0.116798289336946	0.162609810904486	11
0.7	0.15	0.203503849162793	0.265567328808751	233	1000	0.09	0.089965757259056	0.144069215514955	3
0.8	0.15	0.202965969856237	0.265227268380339	221					
0.9	0.14	0.202297515259027	0.259498268283571	211					
1	0.14	0.200291638557993	0.257307080741629	202					

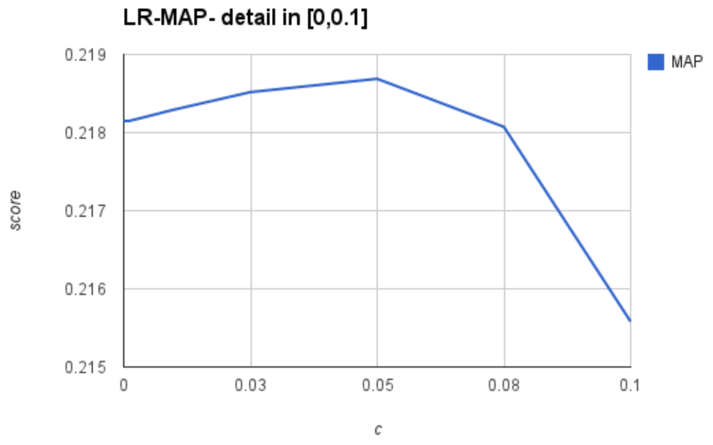
Plots:



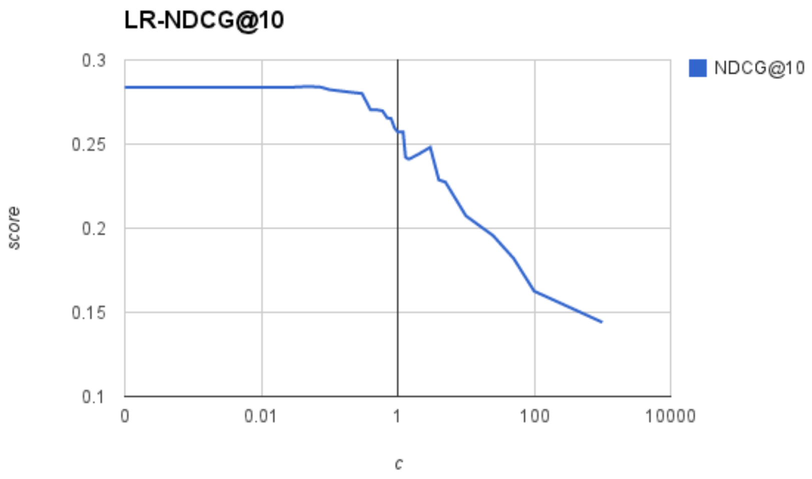
Graph 1-4: LR scores of P@10



Graph 1-5: LR scores of MAP



Graph 1-6: LR scores of MAP in detail of [0, 0.1]



Graph 1-7: LR scores of NDCG@10

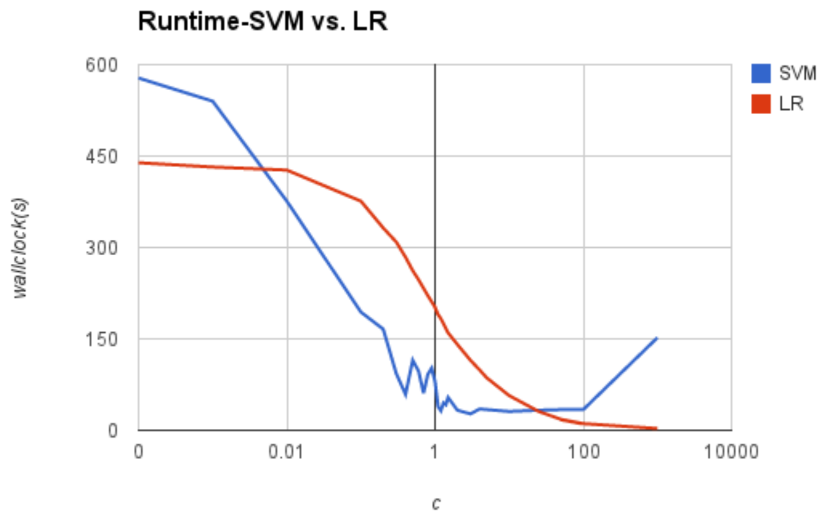
Analysis & Discussion:

1) General change trend:
For P@10, MAP and NDCG@10, when c is increasing, they both maintain at a high score plateau when c is smaller than 1. And when c is close to 1, they all begin to drop quickly. The best performance tends to be reached at a very small c value.

2) Over-fit effect:
According to graph 1-6, there is an over-fit effect when c is very close to 0. That is to say the performance score will first increase and then decrease a little bit. This indicates that instead of gaining the best performance at c=0, the LR model is more likely to gain the best performance at a very small c value, in this case the best c is around 0.05. This may be resulted from that when c=0, the model could be over trained making it perfect for training data but not the best fit for the future testing data.

3) A second-time increase trend when c is around 10
For both MAP and NDCG@10, the scores tend to increase again around c equals to two. Such effects are even amplified when I added my new features (check part 2).

1.3 Runtime Comparison



Graph 1-8: Runtime comparison between SVM and LR* models
* LR model sets learning rate=0.0001 and threshold=0.0005

Analysis:

- The time cost of SVM and LR given the same c are barely at the level.
- The Logistic Regression model has a more stable time cost variation with c changing from 0 to 1000
- They both indicates that with c increasing, the run time cost can be drastically decreasing.
- When the c is larger than 100, the SVM model also gains a time cost increase effect
- When c is too small or too large the time cost of LR tends to be quite stable.
- The SVM model can achieve the best performance (c around 10) with lowest time cost, which is very impressing.

2 New Added Feature

2.1 Strategy

I barely develop the following strategy and stick to them to create improving features:

- **No-linear:** directly add features together with different weight can be hardly effectively, because in the SVM training model, I think the different linear combinations are already considered into the model, so it would be useless to do linear combinations unless we do a general centering and normalization for all parameters. Oh the other hand, log function changes too slowly while power function changes too fast. So the best fit could be multiply different features together.
- **Amplify effect:** if one feature is really more significant than others, I may try to use some stable representing features to multiply with it to amplify the features effect
- **Cross check:** sometimes a relevant document can still perform badly in some features, but by multiplying a certain set of feature family, this kind of mistake can be avoided because we are actually crossing checking the same feature.
- **Heuristic parameter tuning:** even though the feature is really effective in improving results, we still need to carefully tune the parameters to make it perform better. For example the simple tuition of a*b may result in some improvement, but by understand the range of each feature, we can use try to normalize them a little bit before multiplying so that a better feature is possible to be generated.

2.2 Motivation

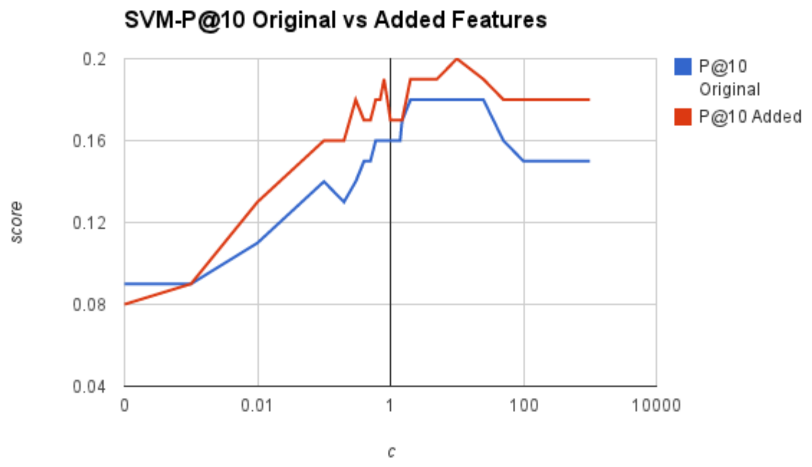
New Added Feature	Motivation
Idea: Hits Authority [6] * Sitemap based feature propagation [13] * tfidf of body [32] * Hyperlink base score propagation: weight out-link [40] **Formula:** =([6]+1) * ([13]+1) * ([32]+2.5) * (480*[40]+1.2) **Procedure:** I first multiply them together to see if they could improve the results. The first result is not very good. Then I checked the range of the variables and find that they are not within the similar range, so I multiply them with some parameters to change them into the same range. Then the performance is still not good enough. So I use two very stable parameters (here stable means the relevant documents always have a higher score in this feature) [6] [32] to amplify the effect of propagation. And finally I adjust the value of the parameters to gain slight improvement.	**Reason:** Indicated by Qin T. etc ‘s paper -- “A study of relevance propagation for web search. Proceedings of SIGIR 2005”. The measure of propagation (like [13] [40]) can be a very unique and different feature than any other traditional features like tf, idf, BM25 score. It can effectively amplify the relevance of a document. By multiply two propagation features and two stable traditional features, I am really trying to use the propagation to amplify the traditional relevant features. **Performance:** Improve MAP from 0.20 to 0.26
Idea: PageRank[14] * Topical Rank [36] * HostRank [8] **Formula:** = 100*((0.63* [14]+1) * ([36]+1)-0.999999) * [8] **Procedure:** I found that since [14] and [36] could very easily to be all 0, it is very hard to multiply them together while still makes those irrelevant documents not getting a score of 0. So I use a tricky part by adding them with 1, timing them and minus them by a number very close to 1. By doing so, not a document will get a 0 score, while the difference between relevant and irrelevant documents will still be quite good. And finally multiply them with [8] to make the relevance stronger shown in the score.	**Reason:** PageRank first raised by google can be the most well-know feature within search engine technology, even years before, I have already heard about this feature. I strongly believe it deserves a higher weight compared to other features. However, one rank feature is quite unstable, which means irrelevant documents sometimes can also gain higher score. But if we do cross check, by with multiplying topical rank and HostRank, the improved complex rank score can be far more reliable and stable. **Performance:** Improve MAP from 0.23 to 0.27

Idea: The square of Tf of Anchor [29]	**Reason:** Maybe similar to URL and out-links, the anchor can be a more objective measurement for relevance compared to title and body, so it deserves to have a higher impact.
Formula: = ([29]+0.7) * ([29]+0.7)	**Performance:** Improve MAP from 0.27 to 0.2989
Procedure: My guess is that multiplying several anchor features together can achieve a good result. But the fact is, many other features are really unstable, they can harm the performance instead of improving them. So I only used a square of [29], and it become very effective. Of course, we need to add a small number before square it, so even [29] is 0, the new score would also be a bigger number to make the scores closer and more uniformly distributed.	

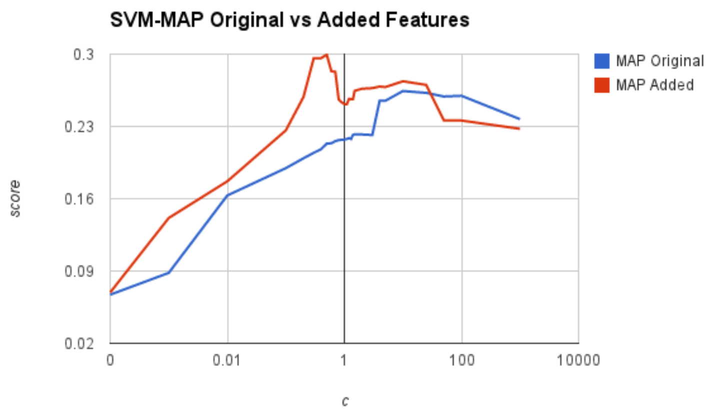
2.3 SVM Performance Improvement:

The highest scores of SVM:

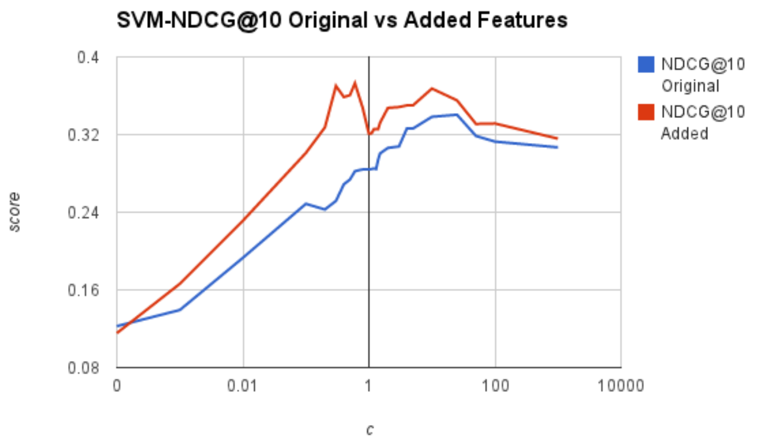
• **P@10:**	0.19	reached at c around 2~5
• **MAP:**	0.2994	reached at c around 0.5
• **NDCG@10:**	0.3729	reached at c around 0.6



Graph 2-1 SVM-P@10 Comparison



Graph 2-2 SVM-MAP Comparison

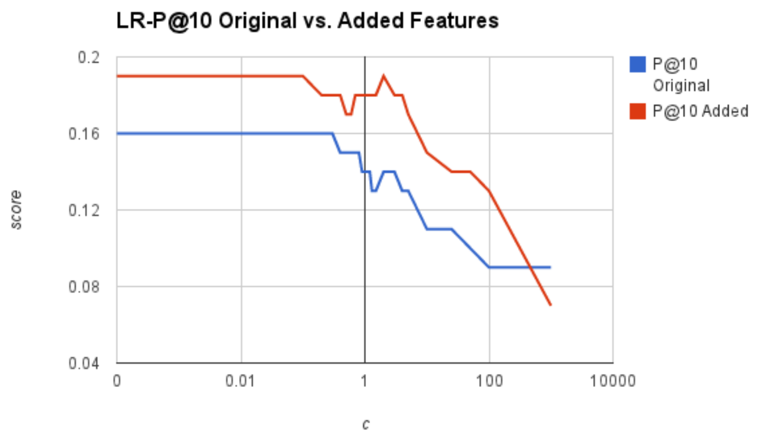


Graph 2-3 SVM-NDCG@10 Comparison

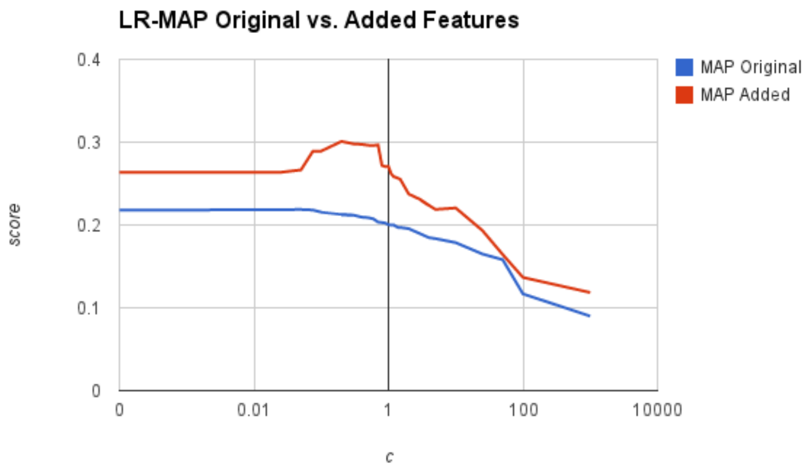
2.4 LR Performance Improvement:

The highest scores of LR:

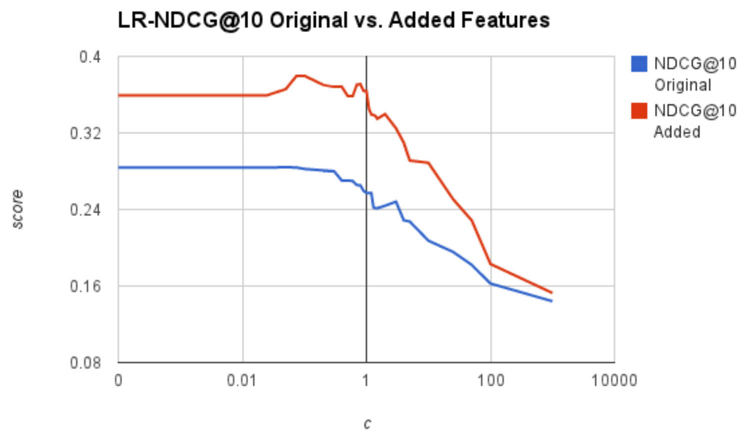
P@10:	0.19	reached at c around 0~0.1 and 2
MAP:	0.3008	reached at c around 0.2
NDCG@10:	0.3798	reached at c around 0.1



Graph 2-4 LR-P@10 Comparison



Graph 2-5 LR-MAP Comparison



Graph 2-6 LR-NDCG@10 Comparison

2.5 Analysis

- Unless the c values are extremely large, my three new added features can effective improve the P@10, MAP and NDCG@10 scores at all c values by approximately 25%~50%
- Graph 2-5 of LR MAP score comparison shows a more distinct over fit effect. When c is around 0.1, the MAP score is quite obvious to be larger than when c is 0 or 1.
- Even for different models, the improvement generated by my new feature is almost the same and quite reliable.

Also notice that after such improvement, the runtime of the same c value will be slightly larger because we are having more dimensions of w parameters, but basically, the runtime increase is quite controllable and within a limit.

3 Implementation Explanation

Since this time the coding model is really simple, so here I only briefly introduce my implementation.

3.1 Classes

Currently the LR and logistic regression is developed in Java and the class is as follows:

1. package com.yitongz.lr for Logistic Regression Model

• Main	The entrance
• Train	Read file and call for train and test
• DocVector	Consider every line of document as a vector
• DocElement	The class for each dimension of a DocVector, contains index and score
• GradientOperator	Do training until convergence
• PariWise	Make D+ vectors minus D- vectors
• Parameter	Store some static parameters

2. package com.yitongz.svm for Support Vector Machine Model

• Main	The entrance
• Svm	Read file, rewrite file and do testing
• DocVector	Consider every line of document as a vector
• DocElement	The class for each dimension of a DocVector, contains index and score
• PariWise	Make D+ vectors minus D- vectors

3. test.cpp a test program, written for automatically run a batch of c values and collect the scores into a result file

3.2 General Procedures

Logistic Regression:

1) First call the Train class, read the DATA.txt to gain configuration information
2) Instantiate an instance of GradientOperator
3) Then for each line of the training data, initialize a DocVector and add it into current DocVector List
4) Add customized scores to DocVector
5) Normalize the DocVector
6) Do D+ minus D-
7) After finishing read all the documents, put the DocVector list into GradientOperator
8) While loop until the convergence criteria is smaller than threshold
9) Back to Train class, call start_test() method
10) Read test data file
11) Add customized scores to DocVector
12) Normalize the DocVector
13) Calculate sigema function values
14) Output the scores to STDOUT

Support Vector Machine:

1) First call the Train class, read the DATA.txt to gain configuration information
2) Instantiate an instance of GradientOperator
3) Then for each line of the training data, initialize a DocVector and add it into current DocVector List
4) Add customized scores to DocVector
5) Normalize the DocVector
6) Do D+ minus D-
7) Write the DocVector list into temporary train data file
8) Call svm_light to learn
9) Wait until get result file
10) Read the test data file, for each line
11) Add customized scores to DocVector
12) Normalize the DocVector
13) Write the DocVector list into temporary test data file
14) Call svm_light to test the test data file
15) Read the result scores and output them to STDOUT