stackOverflow Join Writeup

Russell Lee

8/25/17

Contents

Abstract

Approaching the task of joining a dataset with Wikipedia articles currently utilizes a feature vector with several string similarity measures on both abstracts and titles. After comparing different classifiers for stackOverflow data, the random forest classifier performs best with mean precision of 89% and recall of 86% on holdout sets. Precision on a random sample of unlabeled sets was very reasonable at 86%.

Attempts at joining NLM data appear extremely successful on labeled data, with precision and recall reaching 99%. However, verification on unlabeled data show that many classifiers produce incorrect pairs with precision under 10%.

Introduction

The task at hand is to join together stackOverflow tags with appropriate Wikipedia articles. A small portion of stackOverflow tags already have direct hyperlinks to Wikipedia articles within the short abstract of the tag (This was done on the 7th pass). We then approach the task with a machine learning approach by considering the high-precision linked pairs to be positive labels, and classify a (tag,Wikipedia) pair as linked or nonlinked based on features of the pair. Considering all possible combinations of pairs is clearly too much, so we only consider candidate pairs from an initial pass of soft tf-idf matching.

The same join can be done between National Library of Medicine (NLM) articles and Wikipedia articles, with candidate pairs also generated with soft tf-idf matching.

Tutorial

First we should preprocess the dbpedia articles. This is done by executing scripts/buildWikiData.py.

```
python buildWikiData.py
```

This only needs to be executed once, since the results are cached to dataCached/wikiDict_outfile and loaded from cache for all further uses.

Preprocessing the domain data should be done next, and depends on the dataset of interest. In this example we will do a join with the stackoverflow dataset. Relevant data will be in stackoverflowdata/. To run preprocessing on the domain data, we will need to execute buildAbstracts.py. This script takes in 4 command line arguments : the folder where domain data is stored, filename of full domain data, filename of high precision pairs, and filename of candidate pairs. For the stackoverflow domain, we would run the following:

```
python buildAbstracts.py stackoverflowdata/ tagdata.tsv annotatedMatches.gp
firstpass_joins.txt
```

For the NLM data domain, we would run the following:

```
python buildAbstracts.py NLMdata/ nlm-data.tsv labels.tsv candidatePairs.txt
```

Building the feature vector, classification, and prediction are all done with instances of a join object. Here is function which will do all of the above using logistic regression:

```
def classifyAndPredict(insampleData,outsampleData,folderName,componentList):
    # Declare instance of a join object with input arguments
    easyJoin = myJoin.join(insampleData,outsampleData,folderName)
    easyJoin.setComponentList(componentList)
    # Build feature vector
    easyJoin.buildInsampleFV()
    easyJoin.buildOutsampleFVReduced(0.01)
    # Classify and predict with logistic regression
    easyJoin.classify()
    easyJoin.classifyNIterations()
    easyJoin.predict()
```

To call this function, we must declare instances of insample data and outsample data. This is done by loading the cached results of buildAbstracts.py:

```
# Load preprocessed data from cache
SOInsampleFile = 'stackoverflowdata/dataCached/insample_abstracts_outfile'
SOOutsampleFile = 'stackoverflowdata/dataCached/outSample_abstracts_outfile'
SOInsampleData = pickle.load(open(SOInsampleFile,'rb'))
SOOutsampleData = pickle.load(open(SOOutsampleFile,'rb'))
```

We also must declare an instance of a list of FVComponent objects. For this example, the features will be cosine similarity on abstracts, cosine similarity on 1st sentence of abstracts, cosine measure between abstracts, and edit distance of titles:

```
# Instantiate list of FVComponent objects
csAbstract = FVC.CosSim('CSAbs',TfidfVectorizer( ngram_range = ( 1, 3 ),
             sublinear_tf = True ),False)
csSentence = FVC.CosSim('CSSent',TfidfVectorizer( ngram_range = ( 1, 3 ),
             sublinear_tf = True ),True)
cosM = FVC.stringMatchExcerpts('CosMeasure',sm.Cosine(),
             sm.WhitespaceTokenizer(return_set = True))
LVDist = FVC.stringMatchTitles('LVDist',sm.Levenshtein())
FVCList = [csAbstract,csSentence,cosM,LVDist]
```

With this proper setup, we can call our function for classifying and predicting with logistic regression:

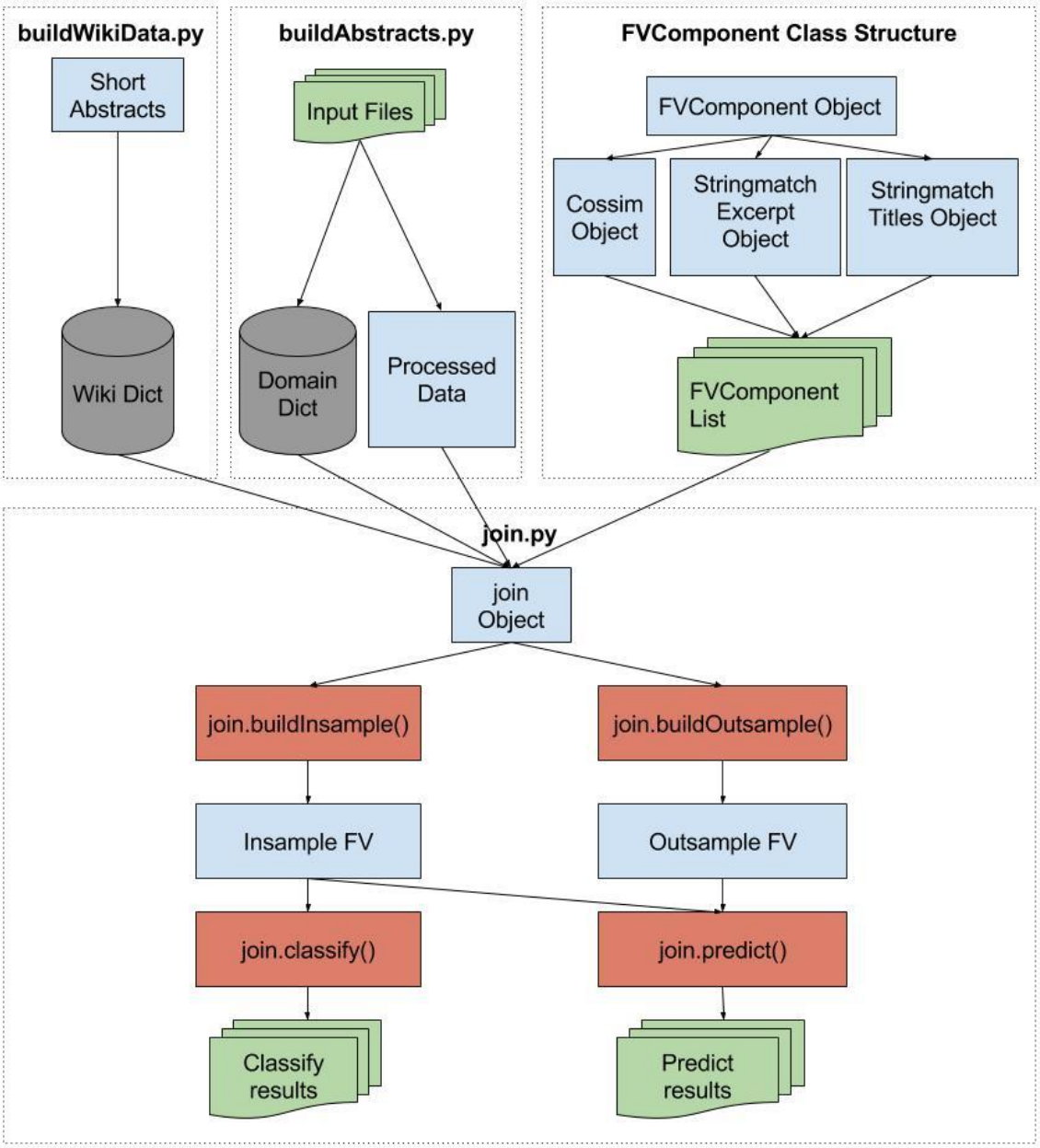
```
classifyAndPredict(SOInsampleData,SOOutsampleData,'stackoverflowdata/',FVCList)
```

If we wish to run this on a different domain, then all that needs to be changed is the insample and outsample data. Suppose that we want to see how the same feature vector performs on the NLM dataset:

```
# Load preprocessed data from cache
nlmInsampleFile = 'NLMdata/dataCached/insample_abstracts_outfile'
nlmOutsampleFile = 'NLMdata/dataCached/outSample_abstracts_outfile'
nlmInsampleData = pickle.load(open(nlmInsampleFile,'rb'))
nlmOutsampleData = pickle.load(open(nlmOutsampleFile,'rb'))

classifyAndPredict(nlmInsampleData,nlmOutsampleData,'NLMdata/',FVCList)
```

System Architecture Overview



The script buildWikiData.py uses dbpedia data to create a dictionary of dbpedia articles. Similarly, buildAbstracts.py takes input files from the chosen domain to create a dictionary for the domain and create other processed data. FVComponent is a class which generates features from processed data, and it has three subclasses. The wiki dictionary, domain dictionary, processed data, and a list of FVComponent instances are taken as inputs for a join object. The join object can generate a feature vector on labeled (insample) or unlabeled (outsample) pairs. Classification only requires the insample feature vector to have been generated, while prediction requires both insample and outsample feature vectors to have been generated.

Implementation Details

Learners

The learners used were the available learners from scikit-learn. Scikit-learn has consistent functions for prediction, classification, extracting precision and recall across its learners.

Features

The full feature vector uses the following similarity measures:

- Tf-idf cosine similarity on full abstracts
- Tf-idf cosine similarity on first sentence of each abstract
- Jaccard measure
- Qgram Jaccard measure
- Dice score
- Qgram Dice score
- Cosine Measure
- Qgram Cosine Measure
- Edit distance of titles
- Smith-Waterman measure of titles
- Needleman-Wunsch measure of titles
- Jaro-Winkler measure of titles

Tf-idf cosine similarity was chosen because it is a common text similarity measure with many tutorials available on Google. The remaining measures were chosen since they were part of the package py_stringmatching, a convenient package for implementing many string matching functions in python.

Datasets

stackoverflow

The stackoverflow dataset is made up of stackoverflow tags and their relevant metadata. Each tag has a numerical ID, and the two important metadata fields used for learning are a title for the tag and a longer text description of the tag. The file tagdata.tsv contains 24,926 entries of tags with titles and descriptions.

High-precision pairs were generated by selecting tags with directly links to Wikipedia in their descriptions. annotatedMatches.gp has 2,191 such pairs.

Candidate pairs of were generated from soft tf-idf matching, and firstpass_joins has 80,235 such pairs. When training the classifier, only pairs with tag IDs found in the set of 2,191 tag IDs from annotatedMatches.gp can be verified as correct or incorrect. Out of the 80,235 candidate pairs, only 9,083 pairs were able to be considered labeled pairs usable for training, while 67,930 pairs were considered unlabeled pairs. The missing 3225 pairs had tag or dbpedia entries without descriptions.

NLM

The NLM dataset is made up of NLM articles and their relevant metadata. Each article has a numerical ID, a summary of the article, and two different titles. The file nlm-data.tsv contains 40,732 entries of NLM articles.

labels.tsv has 16,741 high-precision pairs.

Candidate pairs were generated from soft tf-idf matching as well. The candidatePairs.tsv file is huge, with over 900 million such pairs. Doing training from all such pairs is excessive, so experiments done on the NLM dataset used 60,435 randomly selected labeled pairs.

Experiments

The full feature vector uses the following similarity measures:

- Tf-idf similarity on full abstracts
- Tf-idf on first sentence of each abstract
- Jaccard measure
- Qgram Jaccard measure
- Dice score
- Qgram Dice score
- Cosine Measure
- Qgram Cosine Measure
- Edit distance of titles
- Smith-Waterman measure of titles
- Needleman-Wunsch measure of titles
- Jaro-Winkler measure of titles

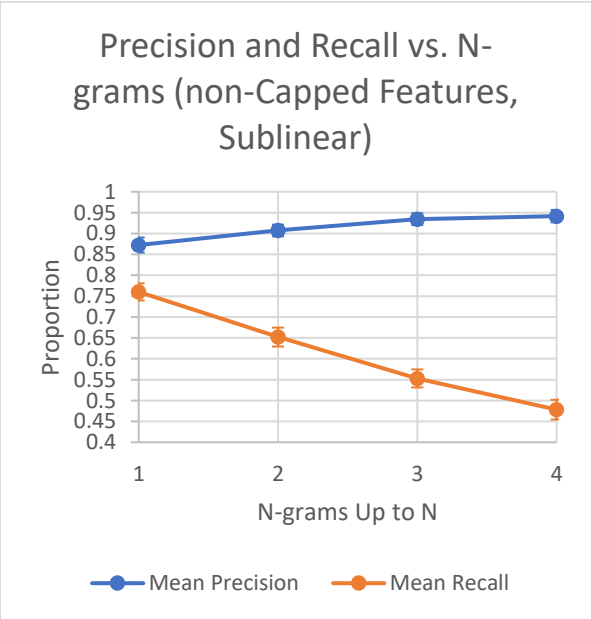
Although some of these measures have tunable parameters, we only tested parameters for tf-idf cosine similarity using logistic regression. Default parameters from py_stringmatching were used when computing the rest of the similarity measures.

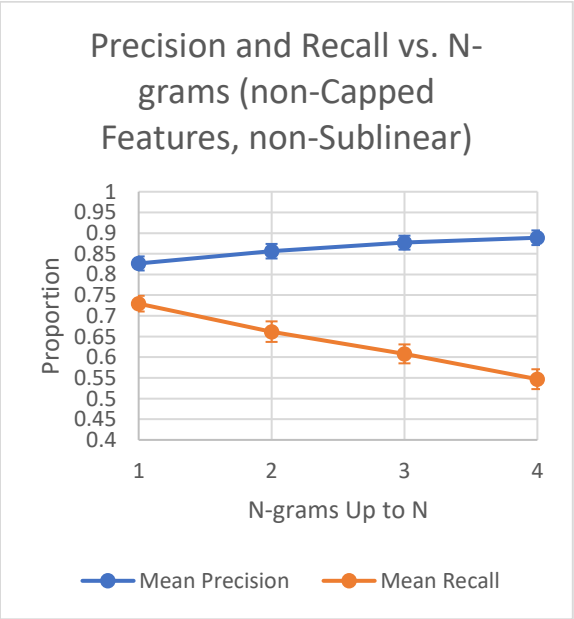
Cosine Similarity on Abstract

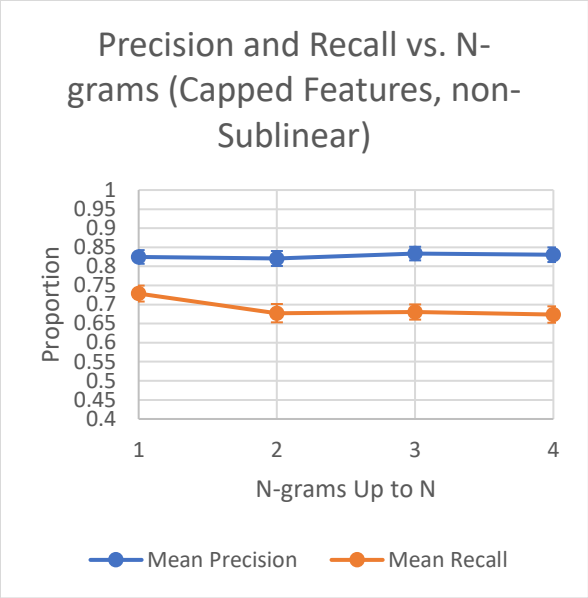
The selected parameters for tf-idf that were tested were as follows:

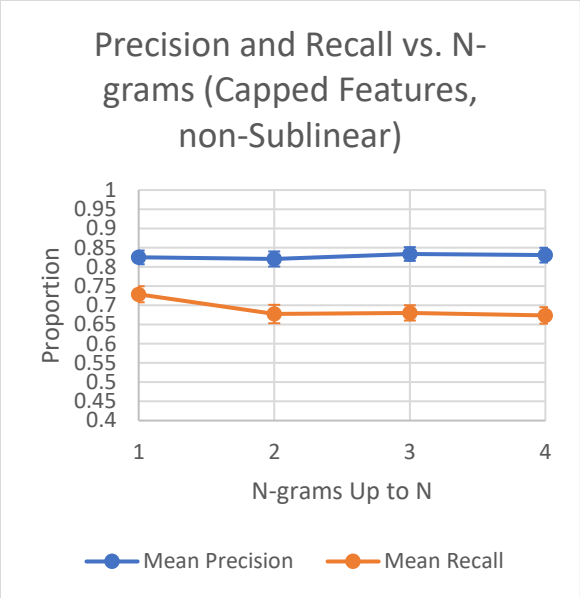
- **N-grams up to N**: range of n-grams up to some value.
- **Capped features**: can only consider the top 40,000 features generated from the ngrams, or consider unlimited features.
- **Sublinearity**: if used, then replace term frequency with 1+log(term frequency)

We tested N-grams up to 4, and tried all 4 variations of capped features/sublinearity.





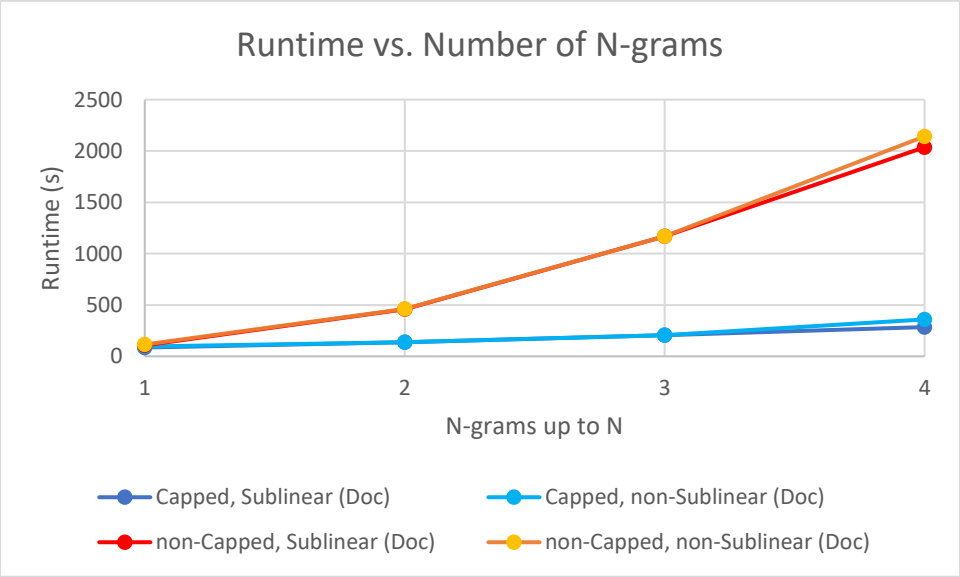




Removing the cap on the features allows mean precision to rise noticeably with the range of the n-grams increasing. The highest mean precision was 94% with no cap, sublinear, (1,4) grams compared to 88% precision with cap, sublinear, (1,4) grams. However, removing the cap also results in mean recall to drop noticeably with the range of the n-grams increasing. This suggests that as the range of n-grams increases, the number of pairs classified as positive decreases. As expected, removing the cap on features also results in a significant increase in runtime since the computation of cosine similarity is being done on a much larger feature vector.

Applying sublinear term frequency appears to lead to a rise in mean precision. However, its effects on mean recall are not clear. Furthermore, it appears to marginally reduce runtime.

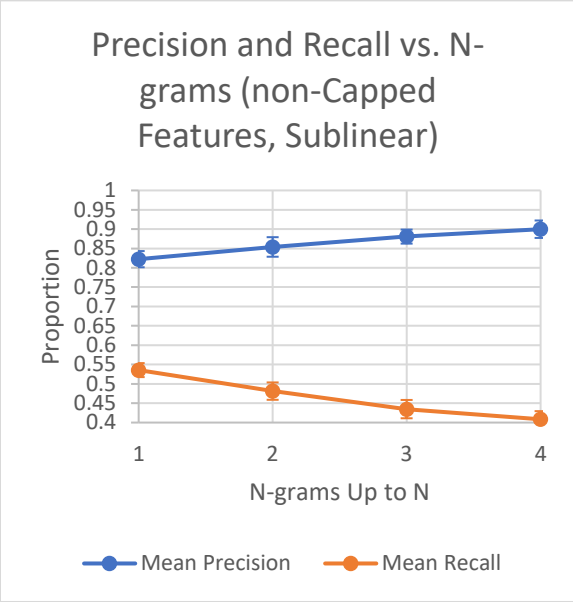
Overall, tf-idf cosine similarity is the most useful features for both precision and recall. The set of parameters of no cap, sublinear, and unigram offers a high mean precision of 87% and mean recall of 76% with a very modest runtime of 107s. Increasing the range of n-grams to (1,4) raises precision to 94% at a great cost of recall down to 47% and runtime up to 2036s. Thus we choose the no cap, sublinear, unigram model as the best model for low runtime, high recall, and relatively high precision.

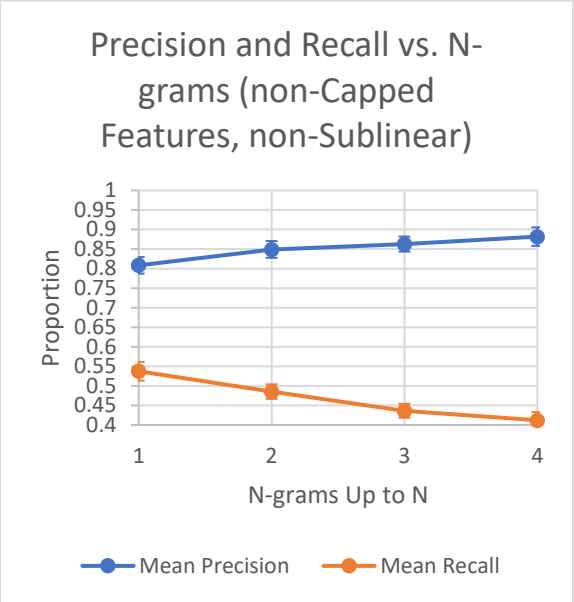


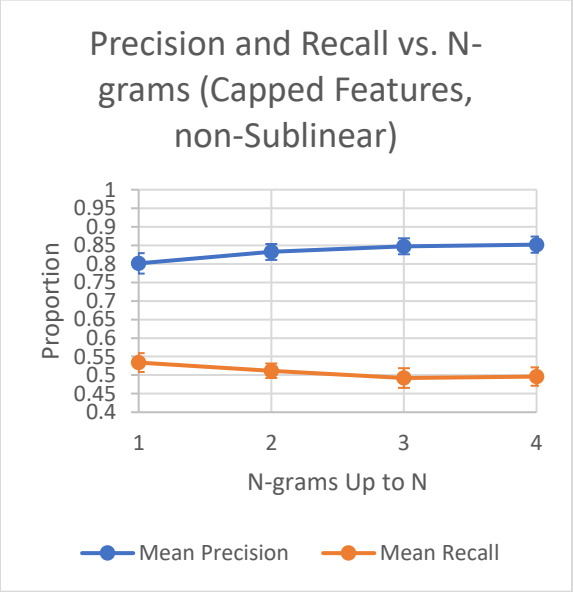
No Cap, Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.872358	0.018025	0.76018	0.020492	107.2684
2	0.907204	0.014049	0.652018	0.022663	458.9752
3	0.934356	0.014075	0.552938	0.021576	1168.418
4	0.941607	0.014866	0.478177	0.023608	2036.498
No Cap, No Sublinear					
N-grams	Precision		Recall		
	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
	0.826652	0.016995	0.729409	0.018939	115.6121
	0.856244	0.017617	0.661742	0.024816	461.155
	0.876882	0.016855	0.607993	0.022882	1167.647
	0.888809	0.017753	0.546811	0.023986	2142.432
Cap, Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
	0.869067	0.012778	0.762471	0.018861	85.80067
	0.8775	0.016666	0.683267	0.024669	136.296
	0.887772	0.014567	0.68292	0.022624	207.1462
	0.884583	0.016362	0.676105	0.024315	284.7822
Cap, No Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.824696	0.017907	0.728674	0.020891	95.71225
2	0.820648	0.019594	0.677304	0.024009	137.619
3	0.833538	0.0179	0.680314	0.01991	204.5634
4	0.83056	0.019164	0.673725	0.021421	359.3236

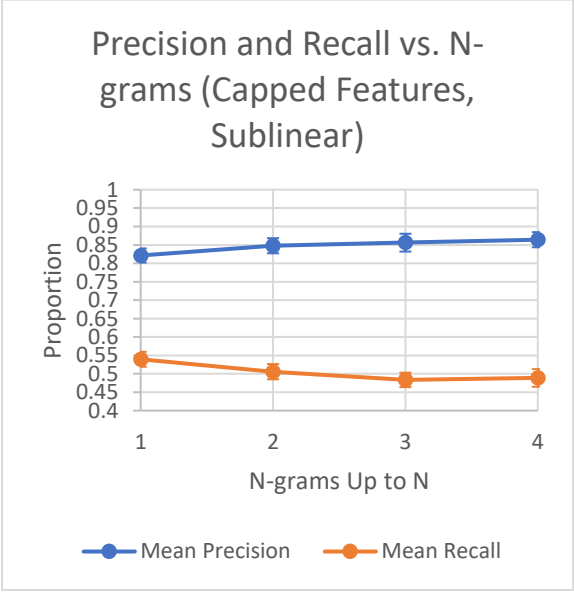
Cosine Similarity on 1st Sentence

The same parameters are tested as cosine similarity on the whole abstract.





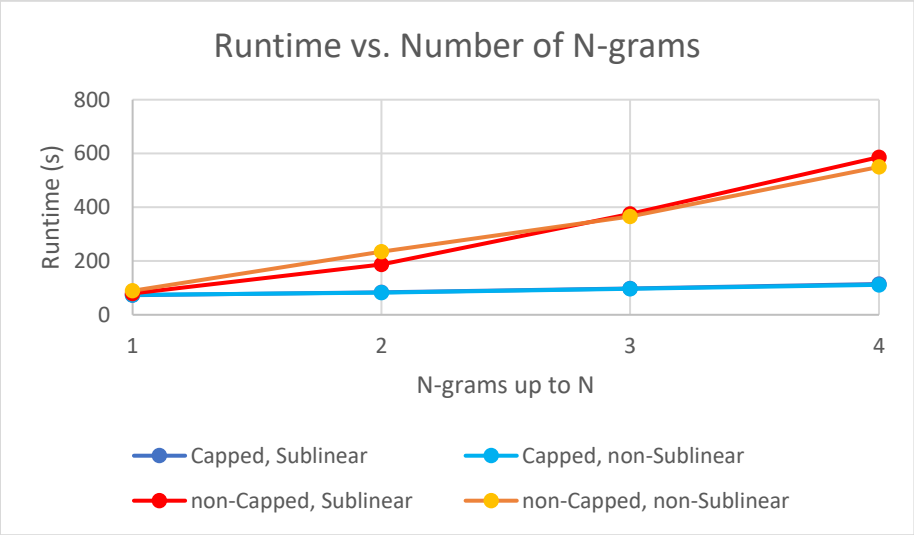




Unsurprisingly, the effects of the parameters of tf-idf are similar to the case before. Removing the cap allows for higher precision at the cost of lower recall, applying sublinear term frequency allows for slightly higher precision, and precision generally increases with a higher range of n-grams. Since tf-idf is only being computed on the first sentence, runtime is noticeably lower, especially when the number of features is not capped.

Compared to tf-idf on the whole abstract, only looking at the first sentence has slightly lower precision and noticeably lower recall. The difference in runtime is largest when running tf-idf on the whole abstract takes a long time. However, since one of the best full abstract models (no cap, sublinear, unigram) has a runtime of only 107s, the disadvantages of only looking at the first sentence are not worth the slightly lower runtime. The best model for looking at only the first sentence would be no cap, sublinear, unigram.

No Cap, Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.822317	0.020909	0.535606	0.018104	79.25415
2	0.853894	0.02527	0.48129	0.022412	186.9094
3	0.880698	0.018124	0.434778	0.023663	374.8583
4	0.899759	0.022535	0.40846	0.021029	586.3389
No Cap, No Sublinear					
N-grams	Precision		Recall		
	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.808389	0.021127	0.537438	0.023939	89.34612
2	0.84908	0.021436	0.485471	0.018578	235.1827
3	0.862771	0.019188	0.436357	0.017644	366.2122
4	0.881864	0.023582	0.411994	0.020758	549.9136
Cap, Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.821304	0.019136	0.53944	0.02028	72.86085
2	0.847775	0.020506	0.505738	0.020616	83.73405
3	0.856111	0.024175	0.483492	0.019625	98.13806
4	0.86411	0.020482	0.48874	0.023879	114.4896
Cap, No Sublinear					
	Precision		Recall		
N-grams	Mean Precision	Std Precision	Mean Recall	Std Recall	Runtime
1	0.801334	0.027557	0.533902	0.025389	74.05619
2	0.832127	0.021475	0.511851	0.019516	82.38722
3	0.847392	0.021533	0.492214	0.026466	96.13876
4	0.851772	0.02184	0.496152	0.024786	111.3003



Evaluating Full Feature Vector on stackoverflow Data

Previously with bag of category phrases, a classification model with logistic regression had an average precision of 89% and recall of 84% over 50 iterations. However, it failed to have acceptable precision on unlabeled pairs due to overfitting.

Running logistic regression on the new feature vector resulted in retaining a high mean precision of 90%, but recall went down to 69%. This time, the learner was able to generalize well to unlabeled data. When examining a random sample of 3,400 unlabeled pairs, 79 out of 94 positive predictions were correct. This is a reasonable precision of 84%. Ultimately, we want the learner to be able to identify positive pairs on unlabeled data with as high precision possible.

Evaluating Classifiers on stackoverflow Data

Although high precision on unlabeled data is our primary goal, a higher recall would also result in procuring more correct positive pairs. With the recall of the update feature vector dropping to 69%, we compare performance of logistic regression alongside several other classifiers over 50 iterations:

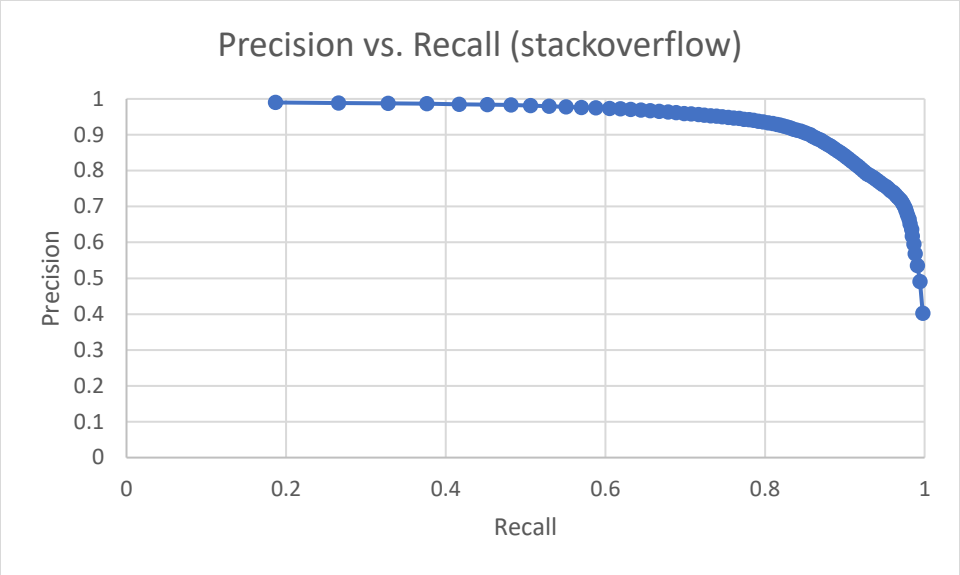
Model	Precision	Recall	Runtime
Logistic Regression	0.8978402	0.698939	3.67924
Random Forest (200 Trees)	0.8936411	0.836144	12.63331
KNN	0.8829762	0.68372	6.763958
Ada Boosted	0.8764025	0.84952	28.04469
Quadratic Discriminant Analysis	0.8708719	0.732219	0.621658
Gaussian Naïve Bayes	0.8685235	0.741248	0.574214
Decision Tree	0.8245824	0.824552	8.006232

Logistic Regression maintains the highest mean precision out of all classifiers being considered, but precision for classifiers within the range of 87% - 89% is still quite comparable. The random forest classifier notably maintains a precision of 89% while also have recall of 83%.

Verification of the random forest classifier on unlabeled data indicates reasonable results, with 123 out of 142 predicted positive pairs being correct, for a precision of 86%. This was in fact the same random sample of 3,400 pairs from logistic regression, so the higher recall also seems to hold true for unlabeled pairs. The random forest classifier results in more than a 50% increase of correct pairs on unlabeled data in this instance.

Precision vs. Recall

The standard threshold for classifying a candidate pair as positive or negative is 0.5 over regression score ranges from 0.0 to 1.0. Varying the threshold level can yield different levels of precision recall – raising the threshold typically raises precision at the expense of recall. Below are the results of 100 different threshold levels from 0.01 to 0.99 on the random forest classifier:



Remembering that the typical threshold level of 0.5 yields precision and recall of 89% and 83% respectively, we see that pushing for 95% precision could drop recall down to 60%. On the other hand, pushing for recall of 90% or greater reduces precision to 80%.

Evaluating Classifiers on NLM Data

We now consider the NLM dataset and seek to join (articles?) from the NLM database to dbpedia. There is ample training data for high-precision pairs, with 13,700 unique NLM articles paired with dbpedia articles. There is also a vast number of candidate pairs, with over 900 million NLM-dbpedia pairs generated candidate pairs. For the following experiments, 54,000 randomly selected candidate pairs were added to training data.

We ran the same machine learning pipeline with the same full feature vector over several different classifiers.

Model	Precision	Recall	Runtime
Random Forest (200 Trees)	0.9938263	0.988107	46.84583
Decision Tree	0.9794517	0.989665	24.9956
KNN	0.9720693	0.975459	56.58399
Ada Boosted	0.9585409	0.940027	159.2209
Logistic Regression	0.941447	0.874755	26.54442
Quadratic Discriminant Analysis	0.9150785	0.880926	5.272168
Gaussian Naïve Bayes	0.9066497	0.872167	4.612466

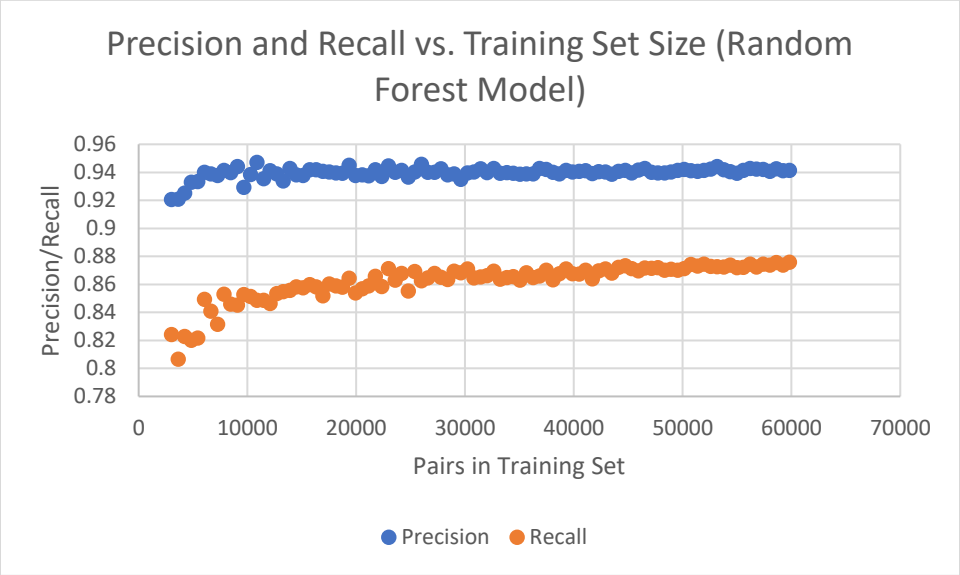
It appears that the classification is extremely successful on nearly all classifiers. The random forest classifier has the highest recall and precision of 99% and 98.8% respectively. Using a decision tree results inn precision only dropping down to 97.9% and recall staying about the same, with only half the runtime. Faster classifiers such as QDA and Gaussian Naïve Bayes have lower precision at about 90% and recall at about 88%, but run 10 times faster than the random forest classifier. Runtime may be an issue of interest when incorporating the full 900 million candidate pairs.

However, verification of the near-perfect results on unlabeled data was disappointing. Many classifiers had a significant proportion positive pairs which were nonsensical, such as (oxycodone and acetaminophen, Buy Jupiter and Other Stories). Several classifiers are predicting along the lines of only 5 out of 80 unlabeled pairs correctly.

The random forest classifier had some indication of working reasonably on unlabeled pairs. It only predicted 8 positive pairs, and it predicted 3 of those correctly. The 5 misses were reasonable misses and not outlandish misses found in other predictions. However, predicting 3 out of 8 is still nowhere near the 99% on labeled data.

Precision/Recall vs. Training Set Size

Since the 54,000 candidate pairs chosen was only a small portion of the available NLM data, we wanted to see the effects of using varying training set sizes on precision and recall. From the training set of around 60,000 pairs, bootstrapped precision and recall were evaluated at points ranging from 5% to 100% of the training set.



Precision seems to stop rising and remain stable at around 20,000 pairs, while this occurs for recall somewhere between 30,000 to 40,000 pairs. This graph seems to indicate that adding additional candidate pairs to the training set would not really affect the holdout precision and recall. However, we should keep in mind that the poor performance on unlabeled data could indicate something is wrong with the feature vector entirely with respect to NLM data.

Future Work

1. Figure out why prediction on unlabeled NLM data is so poor. Could be a bug when writing lines to csv, or a problem with the actual feature vector.
2. Use both titles for NLM data.
3. Run experiments to determine if feature vector could be reduced to fewer components
4. Allow FVComponent class to support string distance libraries beyond py_stringmatching
5. Add an embeddings-based distance feature
6. Consider varying size of corpus for tf/idf denominator

Appendix

File Formats

Domain data files such as nlm-data.tsv and tagdata.tsv should be tab-separated values in the following format:

entity ID	**Text description of entity**	**additional data (optional, can be multiple fields)**
entity ID	**Text description of entity**	**additional data**
entity ID	**Text description of entity**	**additional data**

The entity ID should be a unique identifier and the same identifier used in the high-precision pairs file. A dictionary will be created with entityID: [excerpt text, additional data1, additional data2,…] as key-value pairs. It is presumed that the first additional data field will be a string description of the entity. All other fields are optional, but will not be used or processed by default unless code is edited.

High precision pairs files such as annotatedMatches.gp and labels.tsv should tab-separated values in the following format:

entity ID	**<http://dbpedia.org/resource/link1>**
entity ID	**<http://dbpedia.org/resource/link2>**
entity ID	**<http://dbpedia.org/resource/link3>**

The dbpedia link must be contained within the caret signs as shown above. There is no need to worry about proper capitalization since all dbpedia links will be converted to lowercase.

Candidate pairs files such as candidatePairs.txt and firstpass_joins.txt should be tab-separated values in the following format:

score	**<http://dbpedia.org/resource/link1>**	**entityID**
score	**<http://dbpedia.org/resource/link2>**	**entityID**
score	**<http://dbpedia.org/resource/link3>**	**entityID**

The score is whatever score was used to generate the candidate pairs. Although the score is not actually being used or processed, it may be useful when the number of candidate pairs is exceedingly large (NLM candidate pairs has over 900 million rows), and a subset of candidate pairs is selectively chosen based on this score.

Candidate pairs are considered part of the training samples if the entity ID is in the set of entity IDs contained in the high-precision pairs. All other candidate pairs are considered out of sample and unlabeled, since there is no immediate way to verify if the candidate pair against the high-precision pairs.

Discarded Ideas

Throughout this project, we went through several ideas which did not make it into the final version. These ideas are listed below:

Bag of n-grams

The bag of words, or bag of n-grams model was initially considered for inclusion in the feature vector. However, including each word or n-gram in the corpus causes the feature vector to be both sparse and high-dimensional. This caused runtimes to significantly slow. Furthermore, the performance of n-grams was poor, with n-grams through n=4 having precision below 70%.

Wikipedia Categories

Wikipedia automatically categorizes articles, so we gathered category data on our dbpedia entities with a webscraper. However, we did not have corresponding category data from stackoverflow. Initial inclusion of

features related to categories (such as a bag of category words) proved to be helpful for boosting recall, but upon further consideration we realized that these features were overfitting. This is because the training set is primarily composed of software tags, so the learner performed poorly on correctly joining non-software tags in the unlabeled set. Wikipedia category data may be of use in the future if corresponding data of a domain can be extracted successfully.

Active Learning

Active learning was considered when performance was unsatisfactory after removing category features. However, precision hardly improved with user input data. For example, a test trail of 20 user inputs only yielded an increase in precision of approximately 0.003%. This is possibly because there is already plenty of training data for positive and negative labels. Active learning can be considered in some domain where acquiring initial labels is difficult.