# BANNER Competition

Solution descriptions

Last time compiled: May 24, 2017

## 1 *Romka*

Lucky to me, I took first place in my room just by tuning some coefficients in POSTagger training. I did not even store them in git repository because it was just to play around with BANNER system and the result were very similar to the default solution.

## 2 112atn323

I am sorry that I could attempt few ideas because I could not manually distinguish which are the disease words. One of my good attempts was to create a disease dictionary from the training data. I listed disease words and not disease words from the training data using the frequency of each word as a disease or not a disease. After the annotation by crf, I found the words which are diseases in the dictionary and the words which are not diseases in it.

At first, I annotated the documents which are not annotated by the expert. Then, I created the dictionary from the training data by the expert and the annotated data. The testing data was basically annotated by the training model created only with the training data by the expert. At last I found words which are listed as diseases in the created dictionary and found words which are listed as not diseases in it.

## 3 a9108tc

Missing

## 4 all_random

### 4.1 Algorithm:

Basically, my approach is based on the file `WorkOverview.txt` which states that the best result comes from a simple trust voting. So, I modified the file `Aggregator.java` (`crowd_words/src/org/scripps/crowdwords/Aggregator.jav` to use the two files: `ncbi_train_bioc.xml` and `ncbitrain_e11_bioc.xml` for calculating the F-score of each annotator, then use voting to create a new data set from file `newpubmed_e12_13.xml`.

Several voting scenarios are tried: i) Simple voting with total trust ii) Simple voting with a threshold: Annotators are considered if their F-scores are greater than a threshold. iii) Simple voting with different thresholds, for examples: If an annotation S is voted by a high F-score( $> 90\%$) annotator, it will have a high score. If an annotation S is voted by a low F-score ($<75\%$) annotator, its score will decrease.

Within each scenario, a set of annotations which have score be greater than a threshold is added to the expert data for training.

The default expert data results in 809K. Unfortunately, every new set of data results in a decrease in the provisional scores. These decreases are related to the amount of the new samples. The more samples are added, the more result is decreased. But sometimes, there is a small decrease with a large number of added samples.

In my opinion, there are some special cases these submits could contain a good annotation, comparing with the default one. Then, I tried to mixed different parts of different submits to improve the results. The strategy I used here is, within a voting scenario, the intersect between these submits is quite stable but these intersects change a lot between different scenarios. Therefore, a union between these intersects could provide a good solution.

### 4.2 Code description:

- The `Aggregator.java` is added with several functions, starting with "getAnnotationWithAllTrust_"

- The `TestTrustVoting.java` is used for generating the new data set (these dataset are on my local machine, it contains a large number of files).
- The script `training_data/combineData.py` produces the combined data set between the generated data above and the default expert1_bioc.xml
- The script `combineResult/combineResults.py` uses for comparing the similarity/differences between two submit files. It also produces a union, intersect and subtract between the two submit files.
- The BANNER is trained with the default config (just replace the data file name of the `banner_source/config/banner_b`

## 5 aswmtjdsj

First, the IOBEW is better than IOB tagging scheme. And this is what I submitted to achieve a slightly higher score than the sample.

Second, I believe using word embedding could improve the performance with a large step. You may refer to the paper "Turian, J., Ratinov, L., & Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning", for figuring out what I tried to do.

However, I don't know how to add real-valued feature in mallet. So I didn't try to submit this version of code in the competition and only add a branch to the git repo, which was sent to the BANNER guys, about the work done so far.

## 6 Breusov

After cloning repo and building the application I've received first solution. Base solution gave me 792xxx score.

The first step was to find where the main logic is placed. I found class CRFTagger and it contains main part of learning. I've tried to optimize the training with several options. And reached `813799` on submit. You can see commit with this result (I already sent my local git repository)

Here the changes:

`crfTrainer.train(instances, 50, new double[] {0.2, 0.4, 0.6, 0.8, 1.0});`

I tried several combinations, but this sequence give the higher and most stable result.

Next step was to add mturk to use their data. It was many exceptions because of some data were corrupted or something. I fixed processPassage() to collect data from passage as all-or-nothing. Using only mturk1 or mturk2 give very low quality 43xxxx or 32xxxx, maximum 61xxxx So i change loading data to load several data-files. (after trying to combine expert and mturk together in one file)

Because some sentences can duplicates in different files I changed storing process: - if there is no sentence equals to this then add it - otherwise find it and add only mention to found sentence.

The next was `getTokenLabels()`. I refactored code and create new method, so all reduced to labels = extractLabels(format, mentionTypes, sameType); where we choose label depending on size of tokenMentions()

When using experts we have only one mention (it looks like this), but on mturk file each line mentioned several times, mostly non-experts have same opinion, but sometimes not. Most work was done in this direction (see next paragraph) I think that improving this part can give very high results (and maybe TC member pfr concentrated on it), but it worked very long for getting file for submit(15 min), so it was difficult to check all ideas. I think I received higher score the other participants in my group because this part gave stable improvement.

Ok, I did/tried next (probably in this order): - choose only mention that repeated more than others - choose mention only if there is no other different mentions, otherwise TagPosition.O.name() - choose mention only if repeats at least N times and there is no other - if there are different mention or if their amount are less then 10 - return TagPosition.O.name() It give some improvement of score, but less that I think it should.

The local tester sometimes gives very high result 733xxx - it is high for local, but submit has too little tests, so I really don't know which one will be the best on more tests.

Also, this idea give me some boost. Because expert has higher quality than non-expert, I added his mention to sentence N=10 times... And it gives higher result. I tried to add counter (how much times each mention was mentioned), because don't want to duplicate mentions, but it didn't work... maybe there are implicit dependencies I didn't found...

Some days later I return to train process again. I look at sources of library and found more classes for training. The best result was when I used CRFTrainerByL1LabelLikelihood(model) - something near final score.

I think my result can be higher if I could test program with more tests than default implementation on local. The result from 5 test (there ere 5 on local?) are very unpredictable and unstable...

With combination of all written above and several tests I receive my best result. My best score was 817078.28

P.S. I highlighted all steps that give me my final result. And my sources I already sent you.

## 7 brev

1. At first I tried to modify some parameters in GRFTagger.java.

   crfTrainer.train(instances, 10, new double[] { 0.2, 0.5, 0.8 });
   crfTrainer.train(instances);
   Some results follows (expert1):
   50 – 0.2 , 0.5 , 0.8 532xxx
   150 – 0.5 , 0.8 25-16-9 666666 810xxx
   500 – 0.6 25-16-9 666666 810xxx
   500 – 0.4 26-16-8 684xxx 808xxx
   500 – 0.5 26-17-8 675xxx 814584
   10 – 0.5 26-16-8 684xxx 813213
   I tried order=2, CRFTrainerByL1LabelLikelihood instead of CRFTrainerByLabelLikelihood.

2. After that I tried to add files mturk1 or/and mturk2. Results are the same in average.

   (expert1-mturk1)
   40 – 0.5 27-15-7 710xxx 805xxx

3. Then I loaded expert1 and tried to add some mentions from mturk1 or/and mturk2. Modifications are

   made in BioCDataset.java.
   Find the number of annotators:
   String na = document.getInfon("n_annotators");
   numturk = Integer.parseInt(na);
   I add mentions if there are exactly numturk identical mentions in the sentence. The sentence may contain some such groups. I used the method check to select necessary groups of mentions.
   I used many variants and modifications of programs GRFTagger.java and BioCDataset.java.
   My best provisional score was 815325.
   private boolean check(Sentence s) {
   }

## 8 chengweichi

1. We need combine three xml file (expert, mturk1, mturk2) , so we can get good training file(xml).

2. Test1 from TestCombineAll.java //test1 796034.71 //remove //get mturk2 annotations annos2.removeAll(annos); annos2.removeAll(gold_annos); //get mturk1 annotations annos.removeAll(annos2); annos.removeAll(gold_annos); //get gold annotations
   gold_annos.removeAll(annos); gold_annos.removeAll(annos2); //add //get gold annotaions add mturk1 annotations add mturk2 annotations annos.addAll(annos2); annos.addAll(gold_annos); //retain //get result annos.retainAll(gold_annos); //test1 796034.71

3.Test2 from TestCombineAll.java //test2 809723.67 //remove //get mturk2 annotations annos2.removeAll(annos); annos2.removeAll(gold_annos); //add //get mturk1 annotations add mturk2 annotations annos.addAll(annos2); //retain //get result annos.retainAll(gold_annos); //test2 809723.67 4. Test1 and test2 can't get best score ,because test1 and test2 does not use intersection. I think many day,I read removeAll addAll retainAll usage,finally I think retainAll will lose some annotations ,so we try do not use retainAll ,so we will get my best score without retainAll ,so I write some code

5.code from TestCombineAll.java //get mturk1 intersection mturk2 annotations boolean need_mturk_intersec_mturk2_anno int n_mturk_annos2_intersec = 0; List mturk_intersec_mturk2_annos = new ArrayList(); if(need_mturk_intersec_mturk2_annos) { List keep_annos = new ArrayList(); Map> mturk_doc_annos2 = ac.listtomap(mturk_annos2); Set mturk_ids2 = mturk_doc_annos2.keySet(); for(Annotation mturk_anno : mturk_annos ) { if(mturk_ids2.contains(mturk_anno.getDocumen { keep_annos.add(mturk_anno); } } n_mturk_annos2_intersec = keep_annos.size(); System.out.println("n_mturk_annos2_are"+ mturk_intersec_mturk2_annos = keep_annos; } else System.out.println("n_mturk_annos2_are_not_intersection"); //get mturk1 intersection mturk2 annotations

6. I rewrite intersection function ,so we can use this function get mturk1 annotations intersection mturk2 annotations. Code: public static List intersec(List a,List b) { AnnotationComparison ac = new AnnotationComparison();
   Aggregator agg = new Aggregator(); Aggregator agg2 = new Aggregator(); Map> a_annos = agg.getAnnotationAtAllEx Map> b_annos = agg2.getAnnotationAtAllExp(b); int annos_intersec=0; List keep_annos = new ArrayList(); Map> a_doc_annos = ac.listtomap(a); Set a_ids = a_doc_annos.keySet(); for(Annotation b_anno : b ){ if(a_ids.contains(b_anno.getDocument_id())){ keep_annos.add(b_anno); } } annos_intersec =keep_annos.size(); System.out.println("intersection size is" + annos_intersec); return keep_annos; }

7. How to use this function? //get expert annotations intersection mturk1 annotation List a_b= intersec(gold_annos,annos); //get mturk1 annotations intersection mturk2 annotation List b_c= intersec(annos,annos2); //get expert annotations intersection mturk2 annotation List a_c=intersec(gold_annos,annos2);

8 .we can use mturk_intersec_mturk2_annos(mturk1 intersection mturk2 annotations) get good training file(xml) now.

9. code from TestCombineAll.java //best 811804.61 //remove //get mturk2 annotations from mturk annotations intersection mturk2 annotations mturk_intersec_mturk2_annos.removeAll(annos); mturk_intersec_mturk2_annos. System.out.println(" mturk2 annotations from mturk1 annotations intersection mturk2 annotation are "+mturk_intersec_mturk2_annos.size()); //get mturk1 annotations System.out.println("Original mturk1 anootations are "+annos.size()); annos.removeAll(mturk_intersec_mturk2_annos); annos.removeAll(gold_annos); System.out.println("modified mturk1 anootations are "+annos.size()); //add //get gold annotaions add mturk annotations add mturk2 annotations annos.addAll(mturk_intersec_mturk2_annos); annos.addAll(gold_annos)

```
//best 811804.61
```

9.We can get my best score 811804.61 now.

## 9 cxz2004

Missing

## 10 dimkadimon

My final approach was basic. I ran all the different voting systems from crowdwords on `ncbitrain_e11_bioc.xml`. The best one seemed to be ExperienceVoting with threshold 4.69, giving an F-score of 87.27 (see crowdwords/src/org/scripps/crowdwords/results_e11.txt). I then ran TestExperienceVoting on newpubmed_e12_13_bioc.xml and selected the file with the corresponding threshold. I combined these mturk2 labels with expert labels by concatenating the two files. I trained on this combination with CRFTrainerByLabelLikelihood with 20 iterations and {0.2,0.5,0.8} subsets.

I tried many things that didn't work so well (on provisional data):

- CRF order=2
- CRFTrainerByL1LabelLikelihood worked better when training on pure expert data, but was worse for the combined data
- CRFTrainerByStochasticGradient
- Iterations=50 sometimes gave better results, but other times didn't converge to a good solution
- IOBEW tag format was worse, despite claims in http://psb.stanford.edu/psb-online/proceedings/psb08/leaman.pdf

## 11 EgorLakomkin

I have added only 2 things essentially: Brown clusters features and Bag-Of-Words features for each token in the sequence. These 2 classes are in /src/banner/features/ The rest of the code is the original Banner Brown clusters I have calculated over the PubMed, around 26mln sentences from the abstracts divided into 1000 classes with this tool https://github.com/percyliang/brown-cluster I have added them also in the folder.

Now I see that I have used brown clusters not the best way - I should have added not only the full class, but also prefixes 4, 6,8,10, etc.

## 12 EvbCFfp1XB

I used 4 approaches. 1. Majority voting 2. Random Forest 3. Ignore Mturk(my best submission) 4. Simple voting

1. Majority voting In TopCoder marathon matches, many coders use Random Forest. It's majority voting of decision trees. I thought majority voting of BANNERs works well for this problem. Each BANNER trains with all expert document and subset of mturk1 and subset of mturk2. This approach doesn't reach the target. This approach corresponds to submission12~26. This approach corresponds to directories "banner_source/src/ensemble","banner_source/src/ensemble2","banner_source/src/ensemble3" and "banner_source/src/ensemble4" in my repository.
2. Random Forest I thought Random Forest works well for this problem. But F-score is ~200,000. I think BANNER is too good. I didn't submit this approach. This approach corresponds to directory "src/next" in my repository.

3. Ignore Mturk(my best submission) In approach 1, when I use a lot of mturk's documents, I get low scores. I thought that BANNER might be better than mturk. This approach has 2 steps. step1. BANNER trains with all expert document. And BANNER tests for all mturk2 document that is removed all annotations. step2. BANNER trains with all expert document and result of step1. This approach reaches the target. This approach corresponds to submission29. This approach corresponds to directories "banner_source/src/submission29" in my repository.

4. Simple voting When I run programs of directory "crowd_words/src", I see that simple voting works well. BANNER trains with all expert document and subset of mturk1 and subset of mturk2 that is removed annotations by simple voting 2˜7. This approach doesn't reach the target. This approach corresponds to submission30˜42. This approach corresponds to directories "banner_source/src/submission30",...,"banner_source/src in my repository.

## 13 FCoskun

Actually I just downloaded the source code from the link given in the problem statement and tried to edit here and there. I always got the out of memory error so I changed the crfTrainer.train(instances, 10, new double[] { 0.2, 0.5, 0.8 }); line to crfTrainer.train(instances, 10, new double[] { 0.3, 0.5, 0.7 }); in CRFTagger.java class. It worked. And that's how I got the score. Nothing else.

## 14 Fed

How algorithm works: I have ended up with version, that combines several outputs from trained classifier. MATLAB script merge.m contains code that reads several solutions and produces new solution, which contains all annotations that occur at least twice in the solutions.

I have compared performance by running 5 by 2 cross-validation on provided data, both for training of classifier and for combining of turker's annotations.

Solutions used for merging: BannerAnnotate-mturk-trust.java - use trust voting for turker labels BannerAnnotate-mturk-k5.java - simple voting with threshold k=5 BannerAnnotate-mturk-inf.java - use Bayesian inference (see below) BannerAnnotate-baseline-mturk.java - simple voting with threshold k=6 BannerAnnotate-baseline.java - training only with expert labels

In Bayesian inference I have treated each turker as giving correct annotation with fixed probability (individual for each turker). I did estimation of this probability based on annotations with expert labels. Then I computed most probable annotations based on these probabilities.

Other things that I have tried: I have tried to compute correlations between each pair of turkers to incorporate this information into Bayesian inference. This could possibly give cue on how much information adds each label. However I have failed to get working algorithm using this idea.

## 15 fugusuki

I would send algorithm description statement by this email. My repository (https://bitbucket.org/fugusuki/banner/) is already shared to NTL_CIL_Harvard account, but if sharing code in other way is needed, please tell me that.

### 15.1 Concept of my algorithm

Like bagging method, my algorithm randomly generates many training sets from the given training set and creates predictors ("taggers" in this problem) using training sets. After filtering the taggers by some criteria, each remaining tagger generates result. Results were aggregated with voting to create final result.

This algorithm may have following advantages. (I don't know and didn't tested which one actually effects.) * Freedom of selecting threshold for final vote enables me to create final result that has the best balance of precision and recall. * Voting by many taggers may reduce variance and improve stability. * The filtering of taggers may improve precision.

## 15.2 What I tried and why

First, after cloning the code from repository I run the BANNER program using expert training file only to get example result. I thought there was something unbalanced in precision vs recall. Then I decided to use my algorithm, that is, the most simple one among the algorithms that enable me to change the size of final result.

To implement my algorithm, mainly I modified following codes. * crowd_words/src/org/scripps/crowdwords/TestAggregat - main - executeVotingExperiment * banner_source/src/banner/eval/BANNER.java - train (some codes are extracted to createTagger method) * banner_source/src/BANNER_BioC.java - main - BANNER_BioC - processPassage

## 15.3 How my code works

### 15.3.1 1. crowd_words/src/org/scripps/crowdwords/TestAggregation

In main method, repeatedly executeVotingExperiment method is called with some parameters(percentage of documents used, percentage of annotator used, threshold of voting) changing. In executeVotingExperiment method, documents and annotators are sampled from "data/mturk/newpubmed_e12_13_bioc.xml" with some parameters. Simple vote process by sampled annotators is applied to sampled documents and create new training xml file. ##2. banner_source/src/banner/eval/BANNER In main method, the train method is called. In train method, the training xml files created by TestAggregation.java are read. Each file is concatenated with the training set that is randomly sampled from "config/banner_bioc.xml" and used for creating tagger by createTagger method. In createTagger method, tagger is created by CRFTagger.train method and tested with out of bag sample. Then each tagger is saved as a file whose filename contains the score(all of precision, recall, f-score) of the out of bag test. ##3. banner_source/src/BANNER_BioC In main method, the BANNER_BioC instance is created and then processFile method is called. In BANNER_BioC constructor, called with parameters(which score is used as criterion of filtering, minimum score value), tagger files whose score meets the condition of the parameters are loaded and a list of taggers is created. In processFile method, for each sentence in testing file is processed by processPassage method and submission file is created. In processPassage method, called with parameter(threshold of vote), each tagger in list of taggers makes annotations for the passage and returns result by using simple vote process to the annotations.

## 15.4 Determination of parameters

- Values of parameters for TestAggregation are determined by my intuition and a little trial and error.
- Values of parameters for BANNER_BioC are determined by trial and error (commented out part in main method of BANNER_BioC).

## 15.5 Steps for using my program

1. Compile and run crowd_words/src/org/scripps/crowdwords/TestAggregation with no parameters. Then training files will be created at crowd_words/train.
2. Move training files to banner_source/train.
3. Compile and run banner_source/src/banner/eval/BANNER with no parameters. (In this step, my program stopped twice because of battery problem of my PC. So, the difference of results may be little, if you want to completely repeat my operation please stop the program after 33rd (lexicographical order

of filename) training and delete first 33 training files. Then modify line 60 of the source to 'Random rnd = new Random(1);' and again compile it and run. After first 35 training, stop the program and goto next step.)

4. Compile and run banner_source/src/BANNER_BioC with no parameters.

5. BannerAnnotate.java will be created at banner_source folder.

## 16 goldfinch

I submitted values obtained by procedure described in problem statement (baseline score). I have spent several hours trying to prepare good training files from MTurk data - ranking annotators based on their performance compared with experts, but it did not seem to give better results than baseline. So I did not change my initial submission.

## 17 haipt81

Below are the description of my solution:

- Combining the two dataset: ncbitrain_e11_bioc.xml and newpubmed_e12_13_bioc.xml for training: I modified BioCDataset and BANNER.java to allow loading the second dataset and combining it with the first one. (I first tried cross validation with random-selected sentences on the combined dataset but the result was not good.)
- I used crowdworks/Aggregator.js (with some minor modifications) to find out the most reliable annotators among the mturkers. mturkers are sorted by the accuracy of their annotations against the gold standard e11_bioc.xml, from highest score to lowest score.
- Starting with the top mturker, I find all the documents in newpubmed_e12_13_bioc.xml that are annotated by him, and associated those documents with that mturker (i.e that mturker is the best choice for those documents).
- I then repeat process with the next mturker and the documents that are not yet annotated (by the top mturker), and so on.
- When the top 50 mturkers are used, we covered most of the documents in data set newpubmed_e12_13_bioc.xml
- Back to BioCDataset, when loading the mturks' annotated dataset newpubmed_e12_13_bioc.xml, each of the document is annotated by the corresponding mturker that was associated with it in the process above.
- The combined dataset is trained, which produced the final result.

Please let me know if this description is good enough.

## 18 haveri

Missing

## 19 Infinity

My description is: First I was trying to add a few MTURK-annotated abstracts to the training data and train the model on it. The result was never satisfying, though I even tried to manually extract some very consistent and non-contradictory annotations from MTURK. I could barely exceed the 800000 score on training data with additional MTURK abstracts. Thus I came up to the decision that the banner learning algorithm (didn't dive into it) is not somewhat cool and robust. Though I did not implement any better algorithm, I started removing abstracts from expert-annotated training data. Having removed manually two random abstracted, I received the score 804000. After that I've removed 3 more abstracts and got score 807000. After that I started randomly removing abstracts from expert-annotated dataset and has reached to the score 812000-815000 on part of training data. It has turned out to make almost 816000 on pre-system tests and at most 812000 on system tests.

After that someone in my room has placed first, so I gave up upset, though I believe that removal a few more random expert abstracts from training dataset would result in a 818000+ score.

Summary: The contest statement included a hope to learn to make use of redundant and noisy MTURK annotations, but my experience had shown that learning algorithm, whatever it is based on, is simply not OK, because it would constantly reduce it's model quality after adding additional data and it would increase quality if the data is removed, even if it is high-quality and consistent expert annotations.

## 20 jthread

Unfortunately I spent only very little time on the contest, and it's quite a surprise that I managed to win anything. I tried to use the crowd_words tool to process the mturk data and use that for training, but that only gave worse results because the expert annotations were not included. My idea was to merge the expert annotations with some of the best annotations from the crowd_words tool coming from the mturks, but due to lack of time I was not able to get that working.

My end submission only uses the expert annotations with slightly different settings for the banner tool, which gave me a score just above the 'baseline' of simply running banner as is. In particular, in CRFTagger I found some hardcoded configuration settings regarding data subsets; I changed the subsets from { 0.2, 0.5, 0.8 } to { 0.1, 0.2, 0.5, 0.7, 0.9 } and set the number of convergations to 70 to obtain my final score. Hence, it seems that banner with the expert annotations could also be improved a bit further even without the use of the mturks.

## 21 kubapb

I managed to preper my solution. So I made small manual and included it into zip.

To sum up: My best solution was simple adding "_4.xml" to expert1_bioc.xml as I said before it's very primitive and I think it was just coincidence it score so good.

My main solution was simply machine learning with python scikit learn, and this is inside zip atachment.

## 22 Lapro

Workflow: 1. Run crowd_words/src/org/scripps/crowdwords/GenerateCombineData.java which is a source code I added to the crowd_words project. This will generate a combined training dataset using both expert and mturk data. This code is not well written. There are a few hardcoded variables so if you want to use, you need to change them in the code. Set "useUnkownTurkers = true; soft = false; K = 4" and goto your output directory and get combo-K4-trust0.50-hard-allturks.xml 2. Use this combined data as training (the one I use is named "combo-K4+trust0.50+hard+allturks.xml") and run banner crf training. This will get the final model I got, although it seems not the optimal one that I have tried.

Major modifications: 1. Data combination strategy: 1.1. I firstly tried the simple voting strategy implemented in crowd_words and found when K=4 (at least 4 turkers for each annotation) works best. 1.2. Then I tried to combine this K=4 turker data with expert data for training, which gives better performance. (let's call this combo-K=4 data) 1.3 Afterwards, I come up with the idea to see how each turker performs on the annotation (what is the recall of their annotation?). So I use the code to compute the "trust" value (which is also implemented) of each annotator. And I found some of them have very low trust value like 0.1 and 0.2. So I believe simply ignoring those turkers might help with cleaning the data. I tried different parameters and a threshold of 0.5 on combo-K=4 data performs best on the training set. And this parameter also gives best provisional scores of mine. 2. Postprocessing: I also focused on doing post processing after getting the results because I think the CRF model has not much thing to do with. I added a few files into the project. 1.1. banner.postprocessing.FirstPostProcessor.java: This post process will be applied before any other post processors. Firstly, if we found any mention with "benign" or "malignant" immediately before them, we should add those words into this mention. Secondly, if we found a mention ended with some adjectives like "associated" or "like", etc., we should remove those adjectives from the mention. Also, if a mention is connected to a dash "-" immediately before it, we should trace starting from the dash and include the word in front of it into the mention because we don't want to break any dash-connected words. 1.2. banner.postprocessing.RemoveAbbrevPostProcessor.java: This should be applied after all other post processors done. Firstly, I filter some cases of densely non-disease cases like mentions ended with "gene", "virus" and "protein". I remove the two-sides punctuation from the mentions. Remove those mentions with "-adj." ended. Another thing is when we found a mention is wrapped by brackets, we see the immediately previous word if it is included in any mention. If so, this is an abbrev. so we keep it. Otherwise, we need to remove this abbrev.

Minor fix: 1. The data set contains the "?-xxx" terms which seems to be caused by greek alphabets. So I modified the code to ignore these terms. 2. There are some complaints about finding overlapping mentions when I tried the code on murk data. So I look into the code and revised the Set to ArrayList of "overlapping" variable in Sentence.java (getTokenLabels).

Other things I tried but not working: 1. I tried the stanford tokenizer which makes the results worse. I did not dig into this for long. 2. I tried 2nd order CRF which also gives worse results. 3. I tried to combine the data using soft trust scores instead of ignoring those turkers below a threshold. That does not work.

## 23 lionelc

1. Tuning on training approach: in the original code, class banner.tagging.CRFTagger, it shows three rounds of warm-up training by using 20%, 50% and 80% of randomly-chosen data subset with 10 iteration by default. After trying different data. After a few trials, I figured out the following strategy works better: using 20%, 40%, 60% and 80% with 30, 50, 80 and 120 iterations. And then, the training iterations are changed to 500 to ensure sufficient convergence.

2. Change on Mallet library: by default, the iterative training, as in Mallet library (2.0.6 used as downloaded from http://mallet.cs.umass.edu/dist/mallet-2.0.6.tar.gz ) has an early termination with the condition of lower-than-threshold progress in any iteration. As it may not help the training out of local minima, I removed this early termination condition in both src/cc/mallet/optimize/OrthantWiseLimitedMemoryBFGS.j and src/cc/mallet/optimize/LimitedMemoryBFGS.java . Besides, while L1 regularization would avoid overfitting, I also changed src/cc/mallet/fst/CRFTrainerByL1LabelLikelihood.java by giving a weight to SPARSE_PRIOR. With several trials, SPARSE_PRIOR = 0.045 seems to be a good one. Then the mallet library is re-compiled to a jar to replace the original one included in BANNER code.

Note: as I see in the forum, many other contestants also said they se CRFTrainerByL1LabelLikelihood , but that's not really L1 regularization because SPARSE_PRIOR is default to 0.0 (for the L1 term). That change also makes a difference though, brought by the difference between OrthantWiseLimitedMemoryBFGS and LimitedMemoryBFGS. I tested all such combinations and found a non-zero weight of SPARSE_PRIOR (0.045) makes the score on provisional data better.

Unfortunately, in the contest, I didn't get to use mturk data but just stuck with the default training data set.

The code I used (including changes in both BANNER and mallet) is in https://github.com/lionelc/banner_topcoder

## 24 lipsum

Index: 1. Algorithm description 2. What I tried 3. how to reproduce submitted code

1. Algorithm description:

My submittion is mostly based on original BANNER. What I changed were: 1. Add new features (in src/banner/tagging/FeatureSet.java): * END_WITH_'S feature : whether words are end with 's. ex) ABC's * CAPSs feature : whether words are matched [A-Z]+s regexp. ex) ABCs * END_WITH_SYMBOL feature : whether words are end with symbol character. ex) ABC', ABC- * token length feature : implemented in src/my/TokenLengthFeature.java * SINGLECAP feature : whether words are single CAPS character. ex) A, B, C * common medical word prefixes and postfixes features: such as angi-, gastro-, -um, -thy, etc The list of prefixes and postifxes are taken from [http://en.wikipedia.org/wiki/List_of_medical_roots,_suffixes_and_prefixes] .

```
2. Remove some features to harness overfitting (in src/banner/tagging/FeatureSet.java):
    * LowerCaseTokenText feature
    * UPPER-LOWER feature
    * LOWER-UPPER feature
    * MIXEDCAPS feature

3. replace original tokenizer with MyTokenizer. (src/my/MyTokenizer.java)
    The differences between original one and my implementation are:
        * The word are not split at "-", "'", "_" characters.
            ex) split "A-B" into "A-B" instead of "A", "-", "B"
        * Treat continuous symbols as one token.
            ex) split "AB --" into "A", "B", "--" instead of "A", "B", "-","-"
        * Ignore adverbs in token list:
            The list of ignoring words are in nlpdata/ignore_list.txt file.
            This file is roughly the same as nlpdata/lemmatiser/adv.exc file.

    These changes are for shortening the distance between nouns and verbs.
    By ignoreing adverbs and including some common symbols in words,
    nouns and verbs will be more closely each other,
    so I thought these changes made n-grams more useful and meaningful.

4. Ignore exception caused by my modifications
(in src/banner/types/Sentence.java, src/banner/postprocessing/AbbreviationPostProcessor.java):
    BANNER requires that mention boundaries match token boundaries.
    But because I changed the tokenizer, mention boundaries may not match token boundaries,
    and exceptions will be throwed from BANNER.
    I ignored these exceptions and these mentions.

I trained the model using expert annotated data only.
I did not use mturk annotated data for final submission.

These changes are totally based on trial and error.
I tried many things, and this setting were selected as final submission
```

`because it recorded highest provisional score. That’s all.`

---

2. What I tried: * Use newer version of mallet and use other available model trainers(in src/banner/tagging/CRFTagger.java): CRFTrainerByL1LabelLikelihood, CRFTrainerByThreadedLabelLikelihood or CRFTrainerByEntropyRegularization instead of CRFTrainerByLabelLikelihood * Change crfOrder to 2 or 3(using addStatesForThreeQuaterLabelsConnectedAsIn) instead of 1(in config/banner_bioc.xml, CRFTagger.java). * Ignore common words such like “a”, “an”, “the”, “do”, “it” in token list. * Ignore adjectives in token list. * Ignore prepositions in token list. * Ignore some single symbol characters such like “,”, “””, “”’. * Use 3,4 n-grams instead of 2, 3. ( TokenTextCharNGrams(“CHARNGRAM=”, new int[] { 3, 4 }, true) in FeatureSet.java) * Add PostProcessor to remove apparently incorrect results: If result words contain verb, it is apparently not diesease name so remove this result. * Train the model using mturk annotated data. * Train the model using all (expert + mturk) annotation merged data. * Try combinations of these settings.

---

3. To reproduce my submission: requirements: * java SDK 1.6+ * Apache ant

    1. unzip attached file and move to “banner_source” directory.
    2. To train model : type “ant train”
    3. To evaluate test case : type “ant eval”

    Generated BannerAnnotate.java in “banner_source” directory is submitted code.

note: According to this [http://apps.topcoder.com/forums/?module=Thread&threadID=850084&start=0] post match thread, if java version, CPU architectures(32-bits or 64-bits) or OS differ, generated result may differ. My environments is: OS: windows 7 32bits java: 1.8.0_31 memory: 4GB

## 25 logico14

I achieved max score simply by altering the BANNER code. I noticed that the training part could be implemented better, so I changed this: ” CRFTrainerByLabelLikelihood crfTrainer = new CRFTrainerByLabelLikelihood(model); crfTrainer.train(instances, 10, new double[] { 0.2, 0.5, 0.8 }); crfTrainer.train(instances); return new CRFTagger(model, featureSet, order); ” to this: ” CRFTrainerByL1LabelLikelihood crfTrainer = new CRFTrainerByL1LabelLikelihood(model); CRFTrainerByStochasticGradient crfTrainer1 = new CRFTrainerByStochasticGradient(model, instances); CRFTrainerByLabelLikelihood crfTrainer2 = new CRFTrainerByLabelLikelihood(model); crfTrainer1.train(instances,600); crfTrainer2.train(instances, 20, new double[] { 0.2, 0.4, 0.6, 0.8 }); crfTrainer.train(instances, 20, new double[] { 0.2, 0.5, 0.8 }); // Train to convergence crfTrainer2.train(instances); crfTrainer.train(instances); return new CRFTagger(model, featureSet, order); ” Code is from banner_source/src/banner/tagging/CRFTagger.java. The reasoning behind the change is simply due to the fact that, with more training data, the code should behave better.

I have also made some minor changes in other files. For instance, I added the following code at the beginning of (banner_source/src/banner/tagging/dictionary/DictionaryTagger.java) public void add(String text, Collection types) this: ” if (text.length() == 1) return; List tokens = process(text); if (tokens.size() <= 20) add(tokens, types); ” I intuitively considered this to be better.

I would be very grateful if I were to win the prize due to large amounts of work and effort spent on studying and improving the BANNER source code.

## 26 maniek

Missing

## 27 megaterik

Oh hey, I am not exactly sure, because I am 10000km away from the keyboard, but I believe that I might have changed the way that Turk(aggregation of five, probably) and expert are chosen for the main train set when there is a collision: pick first or last(second). I'll think more on that matter. The second and probably most likely hypothesis is that first one is trained on just experts and the second one on aggregation with k=5 or something

## 28 miguel_clean

Missing

## 29 Mimino

because of the lack of time during the competition, I managed to submit only the sample solution which was provided, i.e. just running the commands:

java -cp 'lib/*' *banner.eval.BANNER train config/banner_bioc.xml java -cp 'lib/*'* BANNER_BioC config/banner_bioc.xml data/test_file.xml out.xml

As discussed in the forums, this solution could give different results, depending on the system and/or java version which was used. My system parameters were:

System: Debian GNU/Linux 7.6 Java: 1.6.0_32

## 30 mln

My approach was to execute crawd_words on ncbitrain_e11_bioc.xml and see which agreement level give best results. Several files were created, one for each crowd_agreement. The 6th file had the highest F-score of 87.27. Then I tried to further improve this by classifying the MTurk annotations in two groups just by looking on the annotation text. I trained Logistic Regression on this file and corresponding expert data.

Then I run crawd_words on the newpubmed_e12_13_bioc.xml and taken the 6th file and applied the trained logistic regression. Then merged the resulting file with the expert data and this was used as source for BANNER.

I tried to split the labels on syllables and add these as features. The diseases are somehow more complex words and probably have different frequencies of syllables. These should be captured by n-grams but I tried anyway. It had better result on my cross-validation data but not on the testing set. I tried to add more n-grams. It had some effect on my cross-validation data and no effect on the testing set. I tried CRF order=2 It had some effect on my cross-validation data and no effect on testing set.

## 31 moshu

In package "banner_source" I made changes only the file "CRFTagger.java" in folder " \ src banner tagging ". The original file is a set of preliminary tests in the 'train' method followed by convergence (see lines 126-132 of the original file"CRFTagger.java").

I found that I get a better score if regime change prior tests. The idea came to me from shell-sort is a sort of bubble-sort Pitch (from big step to step lower). So I tried variants for line 129 above. In the end I chose submission below:

After the competition I downloaded the file with the results and I extracted my results. Unlike the original table we introduced new columns (see file "rezult_moshu.xlsx" attached): columnF: (Prov1+Prov2)/2; columnG: (Prov1+Prov2+System)/3 columnH: The source file appropriately columnI: Equivalent to 129 line mentioned above lines of source files columnJ: percentage (please note that we performed with different amounts of training data set)

## 32 neo.subrata

Approach 1: - 1. Train the solution with the training data. Store the valid annotate and invalid annotate 2. Split the passage into sentence. 3. Check if the sentence contains any known annotate irrespective of case(upper/lower) 4. If any known annotate found then check if it is actually a word or a part of word (e.g. – 'had' contains 'ad' and annotate contains 'AD'. To ignore these cases check if it has any letter in front of the annotate. 5. If the word length is too small then match the case to validate the annotate. 6. If the above validation passed then add it to possible_annotate list 7. Remove any annotate that is present in the invalid_annotate and not in UPPER CASE 8. Print the annotate in output.

Approach 2:- 1. Use a data dictionary to store verb. 2. Split the passage into sentences. 3. Find the first occurrence of verb in the sentence. Match the word removing any special character/-ing/-ed/-s/es etc. Split the sentence based on the verb. Please refer to the function find_verb and comments for more details on finding the verb. 4. Loop through the each word of the first part of the sentence. Try to match the word if it is a known word/adjective by comparing with the known word & adjective (remove (-ly/-ion/-ed etc.) 5. Check if it has multiple numeric values /special characters 6. If step 4 & 5 passes then add to annotate.

## 33 nkshn

### 33.1 My approach

I used BANNER 28 times with 95% all annotations or docs of expert annotation training file which are randomly choosen every time. Then I combined their result and I choosed annotations that were annotated by more than half times(in this case,15 times). It may be called 'bagging'. I didn't tuning hyperparameters(in this case, they are 28, 95 and 15). I had no time to tune them.

### 33.2 I tried but not went well

First, I tried to use both of expert and MTurk annotation file. However, when I trained Banner with expert and MTurk annotation file, the provisional result is less than trained with only expert annotation file. When I choose annotations that many annotator annotated from MTurk annotation file and trained BANNER, the result is same. So, I quit to use MTurk file.

## 34 orlovan

1. The banner system was used as a black box: default configuration, no changes in the source code (except the SentenceBreaker class to handle ? symbols in annotations, see below).

2. No external library is used to work with XML. parse() is an ad-hoc parser for the given document format. In the beginning all 3 given XML-files are parsed with it, and a hash-table documents is constructed as a result. For each document (identified by document id) it contains all annotations from all the given files (Document::annotations vector). At any moment later expert annotations can be distinguished by annotator_id == 6 (from the given expert1_bioc.xml file).

3. For each annotation (identified by document, offset, and length) two sets are constructed: voters_for_annotation — all the annotator_id-s which have listed the annotation; voters_against_annotation — all the annotator_id-s which have listed at least one annotation for the document but haven't listed this particular annotation.

4. Based on the information given by all sets of voters_for_annotation and voters_against_annotation then there goes iterative process of computing confusion matrix for each annotator and assigning weights to annotators, and voting for annotations in different ways (see below).

5. After it is settled down which annotations are considered "correct" there is possible postprocessing of chosen annotations (see below).

6. A new XML-file containing all the documents and all the chosen annotations is created with output_documents() function. For each document each annotation (identified by offset and length) is output only once. For each annotation annotator_id is set to be 1.

Weighting annotators and voting for annotations

The main cycle of weighting and voting proceeds as follows: 1. Each annotator is assigned a weight based on the similarity of his voting with the expert upon all the documents they both (the annotator and the expert) have listed at least one annotation. 2. If an annotator is weighted lower then some threshold he is considered totally incompetent and all his annotations are deleted from all the documents. 3. Using the annotator weights each annotation in each document is evaluated for whether it is considered "correct" or not. 4. Reassign each annotator a weight based on the similarity of his voting with the expert AND with the "correct" annotations for the documents not voted by the experts. 5. Repeat from the step 2 until weights converge (it is usually happens pretty fast — about 70 cycles maximum, depending on the exact scheme for similarity and voting).

I tried different ways to compute how good is each annotator comparing to a set of "correct" annotations, and also different ways to decide whether an annotation is "correct" depending on the weights of annotators voted for and against it (see voters_for_annotation and voters_against_annotation sets above).

Trying different schemes I aimed to maximise the provisional score. Better approach could be divide given expert annotations into training and testing subsets, and try to maximise the score on testing. But I didn't try that.

It seems the promising ways to compute annotator weights are: 1) Matthews correlation coefficient (http://en.wikipedia.org/wiki/Matthews_correlation_coefficient) upon all the documents an annotator participated in. 2) Average value of F1-score for each document an annotator participated in.

Good ways to decide whether an annotation is "correct" seem to be: 1) Compare {Sum of pow_base^annotator_weight} for annotators from voters_for_annotation and voters_against_annotation. pow_base needs to be fitted. 2) Compare {Sum of annotator_weight, where annotator_weight is from maximum N weights} for annotators from voters_for_annotation and voters_against_annotation. N could be 3, 4, 5... needs to be fitted. 3) Compare just {Average of all annotator_weights} for annotators from voters_for_annotation and voters_against_annotation.

After the cycle of weighting and voting completes and we have a set of chosen ("correct") annotations, it could be a good idea to run the weighting and voting again from the start, but with different similarity/voting parameters because it can reveal slightly different set of good annotations. Then annotation sets can be merged (care should be taken not to add intersecting annotations).

It seems the best system score of my submissions was achieved with the following parameters: - Run cycle of weighting and voting two times: 1) Use MCC for weights, 0.5 for weight threshold, {Sum of 50000^annotator_weight} for votes. 2) Average of F1-score for weights, no threshold, {Sum of 4^annotator_weight} for votes.

Postprocessing of chosen annotations

There are two steps of postprocessing of chosen annotations.

First, for each annotation present in the input files I check whether there exists textually identical chosen annotation in the same document. If there is one, and the former annotation doesn't intersect with any other annotation, then it is added to the set of chosen annotations. This step is done in the cycle right in the body of the main() function.

I'm not exactly sure about the impact of this step, but I'm pretty confident it is positive. It's hard to imagine situation, given the small size of documents, when the same phrase means disease once, and something other when mentioned another time.

Second, I've noticed that there is a number of adjectives which can go before the disease name (such as "acute", "benign", "chronic", "sporadic", ...) about which there seems to be no agreement between annotators, to include them or not.

I tried the following: manually compiled a short list of such adjectives (global adjectives constant) and extended each of the chosen annotations with all the adjectives from the list which go just before the annotation in the document text. This is done in Document::extend_chosen_annotations() function.

This step seemed to have positive effect on the provisional score, but (as it turns out) negative, if any, on the system tests.

Maybe more sensible approach would be to extend the annotations before voting. I tried this also (function Document::extend_annotations()), but it had negative effect on the provisional score, so I didn't experiment with this a lot.

OS and Java version dependency

Please note that the resulting XML-file can differ depending on the OS and the C++ compiler/standard library used (this is due to the fact that documents is a hash-table, and the order of traversing it differs with different systems).

Also I've noticed that the result of the banner training differs on different systems even when the input is exactly the same.

My systems producing different results are: - Intel Core i5, OS X Yosemite, Oracle java version "1.8.0_05" - Intel Core2 Duo E7500, Linux 3.2.0-4-amd64, Oracle java version "1.7.0_25"

I didn't investigate the reason for this difference.

The only change in the banner system: SentenceBreaker

Problem: annotations often contain '?' symbols not in the end of a sentence, but (obviously) substituted instead of some meaningful symbols used in the original text and lost upon the conversion to XML. SentenceBreaker on the other hand considers all '?' symbols as the end-of-sentence mark. This results in annotations contain multiple sentences, and banner isn't happy about that.

Solution: if we've got a sentence, and it isn't the first sentence in a document, and it starts with '-' symbol or a lower case letter then join this sentence with a previous one. Sure, this is rather hackish, but it works with the given document base.

Here is the patch:

diff –git a/banner_source/src/banner/util/SentenceBreaker.java b/banner_source/src/banner/util/SentenceBreaker.java index 1fa6832..bd60aec 100755 — a/banner_source/src/banner/util/SentenceBreaker.java +++ b/banner_source/src/banner/ @@ -31,7 +31,13 @@ public class SentenceBreaker { sentences.set(last, sentence); } else { depth += getParenDepth(sentence); - sentences.add(sentence); + int last = sentences.size() - 1; + if (last >= 0 && (sentence.charAt(0) == '-' || Character.isLowerCase(sentence.charAt(0)))) { + sentence = sentences.get(last) + sentence; + sentences.set(last, sentence); + } else { + sentences.add(sentence); + } } index = bi.current(); } pfr =============

Introduction

There are two distinct categories of improvements:

generating the best possible dataset from non-expert annotations

improvements to the BANNER algorithm itself, which could be applied to any dataset

-1 Exploiting non-expert annotations

–1.1 Annotator weight selection

We train a logistic regression based on the dataset annotated by both experts and non-experts,
in order to determine the coefficients giving the best prediction of expert ground truth based on
non-expert annotations.

We then force each coefficient to a minimum of 0.25, and use a default weight of 1 for annotators
who did not annotate expert-annotated abstracts. We make an exception for annotator number
1412, who showed outstanding agreement with the consensus despite not annotating any expert- annotated abstracts: we use a weight of 5 for this person's contributions, which is similar to the best
annotators.

The final prediction for each document is the weighted average of the available annotations.

We use IO encoding throughout the process, since consecutive entity mentions are very rare.

–1.2 Non-expert weighting

Even after this process, the non-expert consensus is more likely to contain mistakes than the expert labels, so we reduce the weight of documents that weren't annotated by experts so that their weight is only 25% relative to an expert-annotated document. When expert annotations are available, we ignore non-expert annotations.

–1.3 Soft thresholding

To better encode uncertainty resulting from the voting process, instead of a single 50% threshold, we use the average of three voting thresholds, at 38%, 50% and 62%.

-2 Improvements to the core BANNER algorithm

–2.1 Regularization

Regularization is important for avoiding overfit.

We use L1 + L2 regularization, with an L1 weight of 0.08 and an L2 prior of 0.35.

–2.2 Weight boosting

One can show that the optimal strategy for maximizing the F-score is to output the maximum- likelihood prediction of an adjusted model where the log-likelihood of an entity appearing is artificially increased by

B = log p0

1 p0

where p0 is the optimal probability threshold for deciding whether a candidate mention should be included. We can show that p0 = F1/2 where F1 is an estimation of the final F-score (we used a value of 0.85).

–2.3 Tagging format

We use IOBEW encoding with weight sharing.

–2.4 Abbreviation post-processing

We detect when a document introduces an abbreviation for an annotated mention. Unless we have a reason to think that this abbreviation is ambiguous or does not refer to a disease (based on the training data), we then annotate all instances of that abbreviation in the document.

-3 Implementation details

–3.1 Pre-requisites

The implementation requires:

Python 3
numpy
pandas
scikit-learn
lxml

–3.2 Generating the dataset

You need to run the Python scripts new/ba.py and new/crowd.py.

–3.3 BANNER configuration

BANNER is compiled and run in the usual way. The configuration file is config/crowd.xml.

–3.4 Post-processing

new/postprocess.py will analyze the BANNER Java submission output (BannerAnnotate.java) and output a post-processed C++ submission on standard output.

## 35 Psyho

All of my solutions were really simple.

There were 2 types of solutions:

1) Using BANNER with default parameters and training on expert data + some subset of mturk2 data based on majority voting. Each voting scheme was based on some algorithm for computing "value" for each annotator plus some simple thresholds. Computed annotator value was either fixed, based on experience (number of processed documents) or computed f-score on expert data. Threshold included: minimum number of annotators for documents (if below I skipped the document), minimum percentile

in terms of total value of all annotators (in other words, I dropped some percentage of least-reliably processed documents), minimum threshold for each annotation.

2) My last few submissions in last 2 hours (of the first 48h period) were simple majority votes using submissions generated from (1). I assumed that BANNER generates pretty unstable results. Based on my experience with other ML methods, I assumed that I should get better results by ensembling several weaker models (earlier submissions). Each such submission used several top-scoring submits (based on provisional scoring) and performed majority voting on the results.

My best scoring submit (on system tests) was based on (1), even though (2) performed really good on provisional tests. The most probably cause for this, is that some of the solutions I ensembled scored 20K lower than my best submits from (1). Still even with such weak models used, the results were very close behind the best submits from (1) and I believe that (2) was overally a very good idea.

## 36 rado42

1. I downloaded sources as directed by the Problem Statement:

git clone https://bitbucket.org/NTL_CIL_Harvard/banner

2. I made modification in file 'banner_source/src/banner/tagging/CRFTagger.java', at or near line 128: Changed '{0.2, 0.5, 0.8}' to '{0.25, 0.55, 0.85}'. The reason for the change: After multiple trials and tests modifying various parameters, this modification ended with the best score while using 'expert' training data set. The mod brought my provisional score from 7940... to 8160...

3. I made modifications in file 'banner_source/src/banner/eval/dataset/BioCDataset.java': Changed the 'throw' statements at lines 62, 66, 71,108 to 'continue'. Deleted 'throw' statement at line 140. The reasons for the change: The built software wasn't able to successfully parse the training data, more particularly, 'mturk2_bioc.xml'.

4. I built the software following instructions in the problem statement. In order to succeed I had to do some modifications due to working on Windows

5. Using the built software, I 1 created 2 solutions:

a. 'Expert' solution by using 'expert' training data

b. 'Turk2' solution by using 'turk2' training data

6. Extracted the substantial 'initializer' data from these, by using this command: type BannerAnnotate.java | find "," > initializer.txt I named these two files 'mentions.txt' (for expert one) and 'additions.txt'

7. Wrote a "reconciler" (C++) program that essentially combines the 'expert' and 'turk2' solutions, using the following algorithm:

a. Everything happens separately for each 'paragraph' (indicated by 'id').

b. I kept all of the 'expert' Mentions unaltered.

c. I added some unaltered 'turk' Mentions using the following selection criteria:

Initially these were java files ready-to-be-submitted

i. Reject any Mention intersecting an 'expert' mention;

ii. Reject any Mention of length > 4;

iii. Reject any Mention if the expert Mentions in this paragraph/id/group do *not* contain at least one Mention of length equal to the proposed 'turk' Mention's length.

iv. The above parameters and algorithm were selected by 'brute force / trial-and- error' approach to maximize the provisional score to the final '8248. . . .' (apparently MAXED the systemtests)

v. I tried to farther filter 'turk' contributions by ignoring the ones that came from short paragraphs (< 300), which did marginally improve my provisional score (to the final solution) but apparently fatally broke my system score.

8. In retrospect, my solutions seems to rely on the following assumptions:

a. Original software was not fine-tuned with respect to some parameters.

b. Experts are much better at doing annotation, compared to 'turkeys'. Therefore, in case of any conflict,

c. Experts sometime miss some annotations (particularly short ones) even if they recognized them

9. My final score was 0.8% above the 'base' line so I hoped my solution was good enough to satisfy the stated 'minimal requirement'.

10. My changes (even if somewhat blind and magic) are not bound to particular data-set. The (provisional and system) sample sizes are reassuringly large. Thus I wasn't concerned with potential over-fitting.

11. I was never able to successfully parse 'turk1' training data. I expect that adding it to the 'additions.txt' set would have improved my score marginally.

12. I never looked at the actual training data contents. I only used the complete solutions created by the software that were using different training sets. The above algorithm may be improved if instead of using just the term length to verify the correctness of potentially-missed mention it uses the terms themselves.

13. Not having the actual training data contents I also completely missed the opportunity to do some 'integral' assessment (like e.g. annotations that are found in other paragraphs are more likely to be correct ones).

14. I did not make any git repository commits. I am also attaching the C++ script I developed that was performing the reconciliation. the expert is always correct. elsewhere in the same paragraph.

## 37 rrr_guru

to try solve this problem, I reuse existing code. Changes I made there are in "CRFTagger.java" file, lines 129-130. All parameters were chosen experimentally. I'm not familiar with any "GIT" software, so please be understanding checking my solution.

## 38 SebM

Missing

## 39 soybean

1. At the beginning, I read through the given solution code (BANNER), to understand how it works.
2. I have done a test mining project with professor (in school) before.It's using naive bayes. So I built a test classifier based on that for finding annotations.

- Unfortunately, the result is really bad.

3. Move back to CRF training. Focus on the optimization for CRFTrainerByLabelLikelihood

- Avoid all the punctuations except "(" and ")"
- Increase the numIterationsPerProportion
- Adjust the parameters in CRFTrainerByLabelLikelihood to get more annotations and better score.
- Final score: 805403.31

## 40 tanzaku

As you know, I only execute algorithm of BANNER :p I had tried to use NBestCRFTagger, but I cound't.
(And I never create git repository in the Harvard BANNER marathon match. . . )

## 41 Vetteru

I changed constants on banner/banner_source/src/banner/tagging/CRFTagger.java. I changed this code crfTrainer.train(instances, 10, new double[] { 0.3, 0.7, 0.9 }); There are 4 constants (10, 0.3, 0.7, 0.9). By changing manually the constants randomly I modified score.

## 42 WojciechMig

Missing

## 43 wsobolewski

Missing

## 44 Wtrei

Missing

## 45 wzyxp_123

Hello, my algorithm description is following(and there is a copy in attachment)

I have tried three parts of methods: 1). Add or adjust feature of CRF in Mallet(banner source). 2). Filter or classify MTurk data to increase the scale of training data. 3). Take union and intersection for different submission

Details: 1). I have force the preTagger. a). I have tried collecting all named entity in Expert dataset as a dictionary, and use it to build a dictionaryTagger as the preTagger. b). I first train a CRF tagger use the default settings, and treat it as the preTagger and train a new CRF again. Unfortunately, these two approach didn't perform better. 2). I force the expert data set, and two Mturk data sets (noted as Mturk_1 and Mturk_2 respectively). I treat expert data set and Mturk_1 data set as a TrainDataSet and train a classifier, to pick up the right named entities in Mturk_2. Then use the right ones in Mturk_2 to add into the set of expert data set for training CRF Tagger. I build one datapoint for each annotation and treat the annotator_ids of it as features of the datapoint, if the annotation appears in expert dataset, its label is 1, otherwise its label is 0. Then I apply libsvm to train a svm classifier to pick up right ones in Mturk_2. By using this method, I have collect double training data for CRF tagger, but the CRF training result is not good. I think the predicting accuracy of svm is not good enough, and the reason of it may be I haven't choosen good features and treating annotator_ids as features is not good enough. 3). I have tried the methods of voting to choose right annotation, which are implemented in crowd_words, including TestAggregation, TestCrowdAgreement, TestExperienceVoting and TestTrustVoting. I found some additional annotations obtained from these approaches in different F1 score( I think precision is more important here, because I want to add annotations who are mostly true, even though the number of new annotations is small) are useful and obtain an improve of the final CRF performance. Especially, the result labeled as 6.1439 obtained from TestExperienceVoting impressed me very much, and I noted the final result when it being added in train data set as Exper6. Then I think many submissions' intersection can improve the precision, and the unions can improve the recall. Therefore, I choose several my previous submissions and make them vote. I simple choose my 10th to 16th submission, and if one annotation appearing in these submissions more then 3 times, I keep it. However, the mechanism of intersection decrease the number of result set and make the recall down even though its precision is really good. Therefore, I try to take the union action for the intersection result and Exper6 result, and the union set have achieve my best F1 score: example results: F1=700000.0, precision =0.6086956521739131, recall=0.8235294117647058 provisional results: F1=821936.31 system results: F1=821124

All code is at: https://bitbucket.org/cloudssdut/banner_wzy

Regards, Zhuoyu (wzyxp_123)

## 46 xhae

Missing

## 47 yaric_om

### 47.1 The winning submission:

I've used CrowdAgreement algorithm provided as source code in order to combine first 'expert1_bioc' and 'mturk1_bioc' document's annotations (as its both about the same abstracts) based on maximal calculated consistency. From resulting document I've extracted annotations for dictionary. Than I've combined resulting document with 'mturk2_bioc' using CrowdAgreement algorithm too. After that I've used final document to train BANNER with CRFTrainerByL1LabelLikelihood, order = 1 and gold dictionary I've generated.

Unfortunately due to the huge drop in provisional test I've abandoned this solution. It seems drop was caused by provisional test bias for more than 15000.

I believe this solution can be further improved (by using different crowd algorithms, etc) if I can got correct tests results.

### 47.2 Other tries:

I've tried various combinations of crowd algorithms and BANNER settings, fixed several bugs in BANNER source code.

## 47.3 Abstract:

I my opinion with such huge bias of provisional test there was two feasible ways to win: 1.) By chance find appropriate solution and further improve it despite bad provisional tests results (intuition) 2.) Pure random moon shoots which seems was applied by majority of winners (good luck)