Measurement and Analysis of NLTK for Natural Language Processing Applications

Sean Donohoe and Kyle Wiese

Abstract

This paper reports part of speech tagging accuracies for the Natural Language Toolkit's TnT tagger when subjected to different training files, training file sizes, CPU's, and memory organizations. An analysis of variance is conducted to examine if CPU or memory organization have a bigger impact on tagger run time, and a custom hidden markov model implementation is used to compare to the TnT tagger.

Keywords

ANOVA, TnT, MASC, Brown, Penn Treebank, Quick Tagger

Introduction

Data science is becoming a hot field in today's tech industry. From advertisements to semi-sentient products, understanding how to incorporate aspects of our world into smarter technology is a highly sought out skill. Of these aspects natural, or human, language processing is a vital cornerstone of a successful information extraction or processing application. Though it is in high demand, natural language processing is still a highly experimental field, with only highly educated individuals being capable of meeting its cutting edge tech standards.

In an attempt to make the incorporation of natural language processing technology a feasible goal in the context of today's tech industry, a variety of natural language processing API's and toolkits have been developed for the "average joe." Among these, the Natural Language Toolkit (NLTK) is a well-renowned, all-encompassing, toolkit used by a variety of enthusiasts and businesses. Even though it is used by a wide range of individuals, how good is NLTK actually? The API is based in Python, which can be upwards of 400 times slower than C -- which seems nonoptimal for real time language processing applications.

This investigation attempts to evaluate how "good" NLTK is by assessing the speed and accuracy of its functionality for the basis of most natural language applications -- part of speech tagging. In short, part of speech tagging is a good evaluation metric because it is relatively simple and is foundational to applications which require any sort of text parsing. At its most basic level, a part of speech tagger can be implemented as a bigram hidden markov model -- a concept that can be explained to an individual who is only slightly versed in statistics in a few hours. Evaluating the hidden markov model implementation is important, since most programmers will try to stick with what's familiar, or easier. We should note, though, that we are not concerned with individuals trying to quickly use NLTK's functionality for generic processing -- any individual trying to adapt a natural language model to a specific application will need to dive deeper into NLTK's classes in order to focus a tagger's functionality on a specific set of texts.

First, the experimental design is outlined. After discussing the motivations behind testing various aspects of NLTK, we will move onto a brief ANOVA to observe how much memory and cpu affect tagger run times. Finally, we discuss how NLTK's TnT class fares against our own quick tagger implementation, and how much of an impact tag specificity and training file size impact accuracy.

Experimental Design

In order to evaluate the differences in taggers, we first had to gain access to multiple different taggers and corpora. To achieve the necessary resources, we used NLTK to download the TnT tagger (Trigrams 'n tags), Brown corpus, MASC, and a partial Penn Treebank corpus. These specific corpora were chosen due to their inherent differences, which in turn would allow for a greater understanding into how specific corpora affect the performance of the taggers. In the interest of incorporating more variety into the taggers used, we wrote and implemented our own version of a tagger, referenced as the "quick tagger" in this paper, which uses a bigram approach instead of trigrams.

Once the correct corpora and taggers were downloaded and working, a python script was used to train and evaluate the two different taggers on a specified percentage of the different corpora. The TnT tagger was evaluated using the MASC corpus as well as the Brown corpus while the quick tagger was evaluated using the Brown Corpus. The results from this script include the time it takes a tagger to evaluate the test set as well as the accuracies returned from the evaluation.

Concerning the analysis of differences in the corpora, a second script was used to evaluate how different granularities (number of different POS tags seen in the corpus) and training sizes of the corpora (Penn, Brown, and MASC) would affect the training time as well as the accuracy of the TnT tagger. This script resulted in training times and accuracies recorded by increasing training file size in steps of 5% of the size of the penn tree bank corpus.

In order to observe the differences that hardware makes on the performance of POS tagging, the scripts that were detailed above were used on multiple machine configurations (described in more detail in the Setup section).

Finally, analysis was done on the measurements to determine which configuration factors played the biggest roles as well as which tagger was more accurate/more time efficient.

Environment

CPU's Used

- Intel Atom D510 @ 1.66 GHz
- Intel Core 2 6400 @ 2.13GHz

Operating System

- Ubuntu 14.04 32-bit

Memory

- Kingston DDR2 2GB RAM (arrangement of multiple sticks)

Design of Experiment

Percent of Training File Used	**CPU Speed**	**Memory Size**
-1	-1	-1
1	-1	-1
-1	-1	1
1	-1	1
-1	1	-1
1	1	-1
-1	1	1
1	1	1

Reasoning:

After a preliminary run using the NLTK provided TnT tagger, the amount of time required was substantial which in turn lead us to restrict our experimental design to a 2^3 design, which can be observed in the sign table shown above. Since our goal was to determine NLTK's performance on different platforms, we chose a 1.66 GHz processor and a 2G memory configuration in order to reflect a possible mobile natural language application. The 2.13 GHz processor and 4G memory configurations were meant to represent a higher scaled computing environment. One more aspect to our design was to use an in house server configuration rather than a service like S3 since it allowed us to easily configure the different environments required by our design. With respect to the "Percent of Training File Used" we chose 10% and 90% since the amount of

training data given should increase the accuracy and would allow for insight into the relation of how much it affect the accuracy.

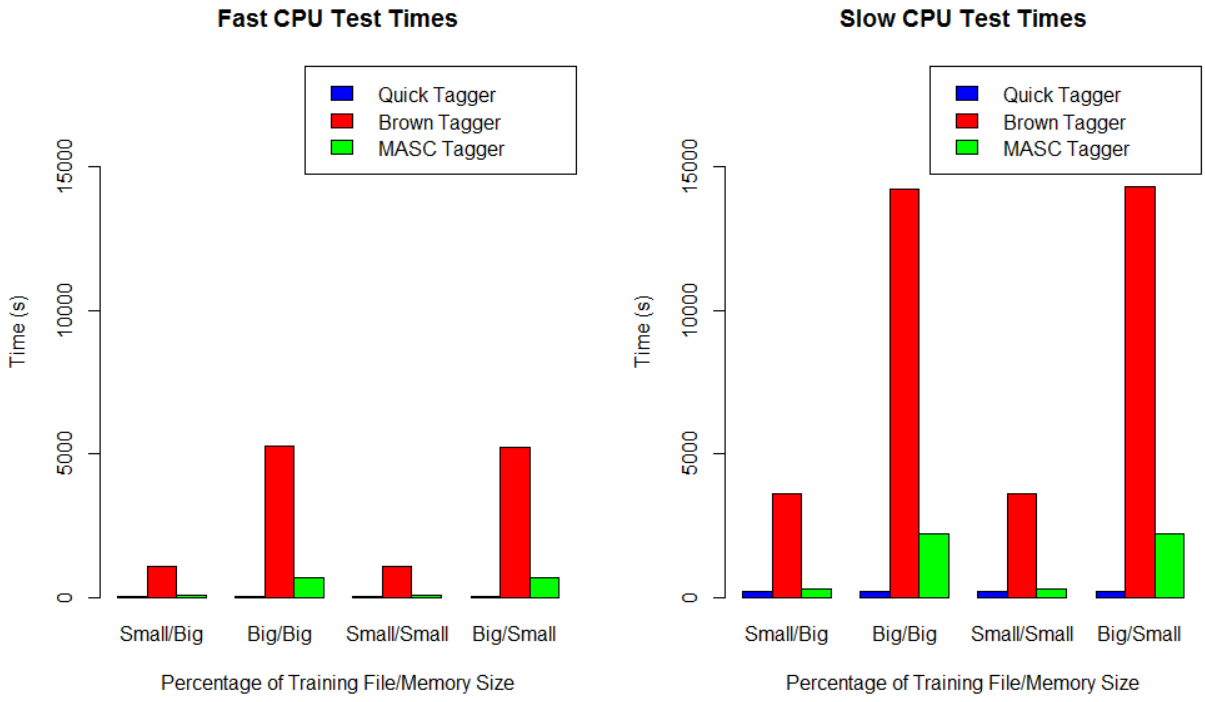
Results and Evaluation

Analysis of Run Times:

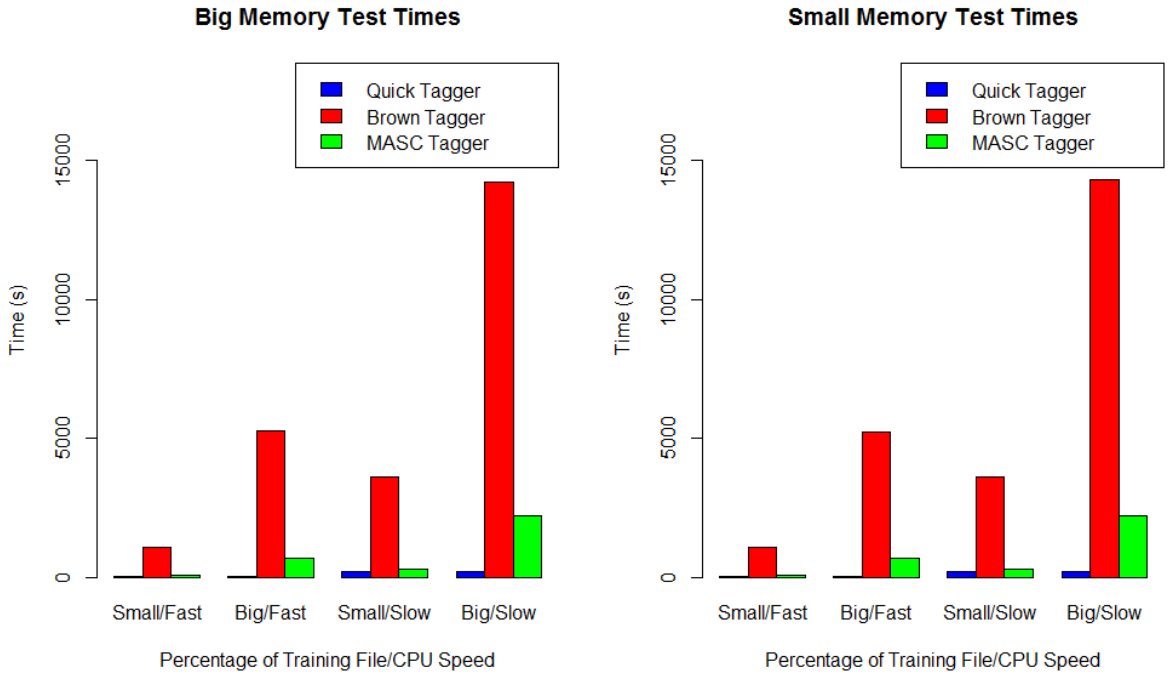
From the results observed from the ANOVA with respect to time, we can conclude that if the quick tagger is being used the CPU is the most significant factor in time needed to complete a task. Therefore, if an organization is interested in time utilization, they should use the quick tagger along with investing in more powerful CPU's. However, with respect to the TnT tagger, training size was observed to be the most significant factor. As a result, if an organization was interested in time utilization, was set on using the NLTK provided TnT tagger, and accuracy was not of significance they should focus on using a smaller training set. This conclusion can be explained by how our implementation of the quick tagger differs from how the TnT tagger functions. Even though both the quick tagger and TnT tagger utilize the Viterbi Algorithm, in our implementation of the quick tagger, we decided to not include the smoothing aspect as well as only using the context seen in the training data in order to cut down on the time required to run the program. Since these alterations in the quick tagger reduced the time taken to run the Viterbi algorithm significantly, the amount of training data used does not affect the runtime as much as the full Viterbi implementation used by the TnT tagger; therefore, the CPU becomes the most significant factor with respect to runtime for the quick tagger. On a similar note, since the TnT tagger does full smoothing and full accounting for unseen words, the TnT tagger runtime depends heavily on the size of the training data provided, and therefore decreases the significance with respect to the CPU.

ANOVA

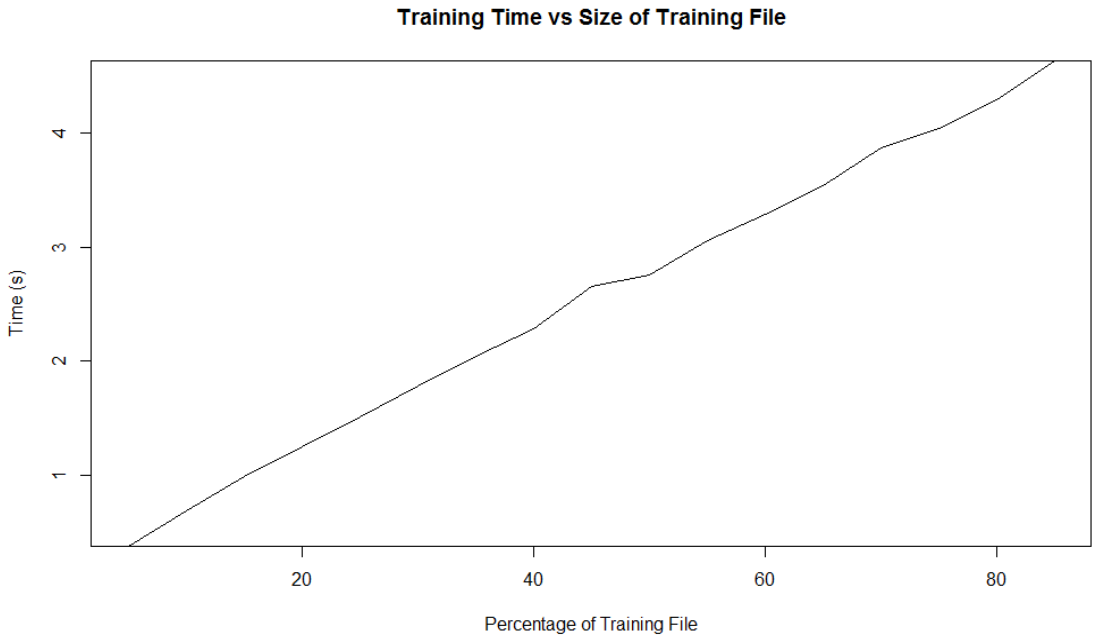
File Size (qt, b, m)	CPU	Memory	CPU:File Size
0.02124196%	99.97231%	0.0001621978%	0.003954701%
55.56295%	33.70003%	0.0002703555%	10.73463%
57.72646%	26.6996%	0.00000423088%	15.5725%



The above graph supports what was seen by the ANOVA in that the runtime for the quick tagger remained constant between large and small training file sizes and was only observed to be affected by the speed of the CPU. With respect to the TnT tagger trained on the Brown and MASC corpora, the training sizes as well as the CPU speeds were observed to make noticeable impacts on the performance.



The above graphs also support the ANOVA results in that they show that the memory has little to no impact whatsoever on the runtimes seen. The side by side graphs above demonstrate this in that there are no observable differences between the 2G and 4G memory configurations with respect to the runtime results.

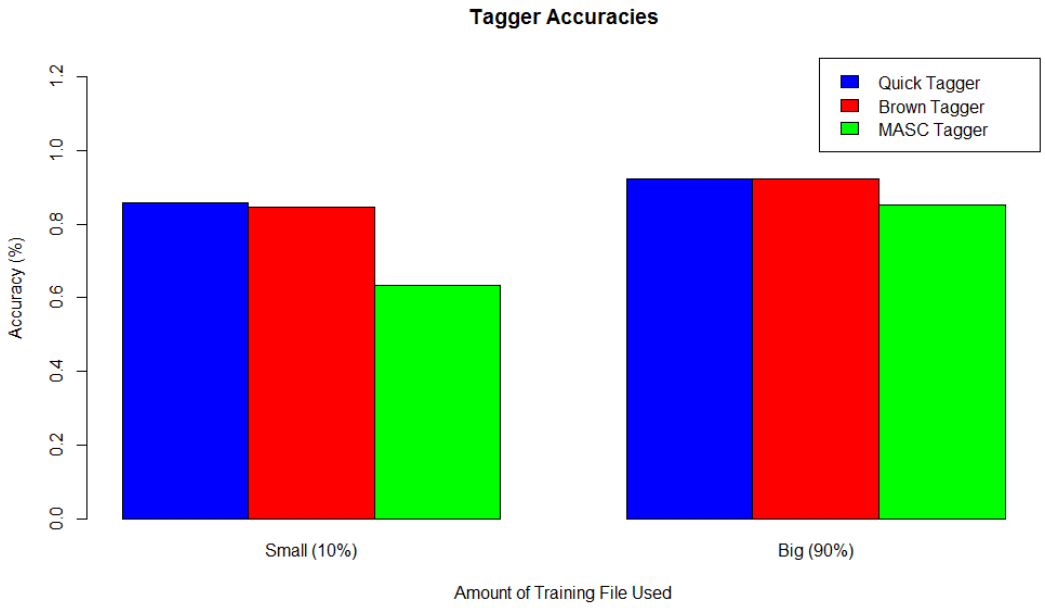


The above graph was obtained from the training times recorded by the granularity script described in the DOE section above. As can be seen, the graph describes a linear trend between the amount of training data provided to the TnT tagger and the time it takes to train.

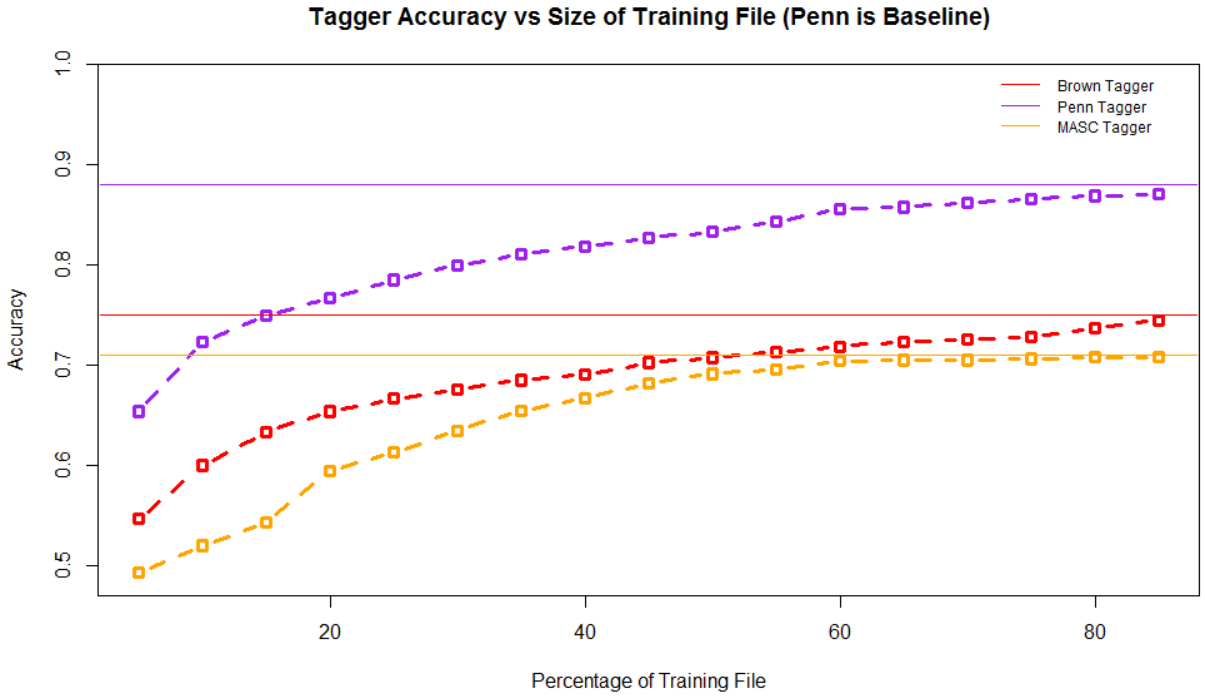
The conclusion we came to as to why the memory didn't end up affecting the runtime was due to Python's memory conservation techniques, such as lazy retrieval of data through yield statements. Therefore, if the file size bottleneck in the TnT tagger were to be removed, the CPU would be the most significant factor in the amount of runtime seen. As a result, if a company or organization were interested in increasing their runtime performance, they should invest in a system that does not have a training file size bottle neck as well as investing in more powerful CPU's.

Analysis of Accuracy

The accuracies of the TnT taggers for the Brown and MASC corpora, and the quick tagger for the Brown corpus, were obtained by changing the amount of the respective corpora used for training data. As described in the experimental setup, it was expected that the accuracy improves with increasing training data size. The goal of this measurement was to analyze how much the accuracy improves among the different corpora as the portion of training data was increased. The plot below summarizes the effect of training file size on accuracy for the three taggers.



Though there is an apparent difference in accuracy for the small and large training file sizes, this difference is surprisingly small. Perhaps even more surprising is that the quick tagger was never less accurate than the brown tagger. This would indicate that the overhead imposed by smoothing and trigram methods does not offer much of a performance boost, and that simpler, in this case, is better. The MASC accuracy is to be expected -- as indicated by the plot below, we see that a higher granularity in POS tags improves the accuracy of a tagger, and MASC has the same number of tags as the brown corpus, but with less sentence variety.



The plot of accuracy versus training file size shown above **did not** involve our quick tagger implementation, and instead used a TnT tagger which was trained on a subset of the penn treebank corpus included with NLTK. The sizes of the training files were based on the size of the penn corpus. The penn corpus utilizes 36 tags, while the Brown and MASC corpora use 227 and 20, respectively. The penn tagset therefore has a higher detail in labeling sentences than the MASC tagset, but coarser detail than the Brown tagset. This plot shows that as the granularity increases, so does the accuracy of the tagger -- to a point. The Brown corpus, which has the finest tag granularity, is actually *worse* than the penn tagset in accuracy. This is most likely due

to a higher granularity in tags requiring more training data in order to accurately "learn" which tags are applied where.

The plot above also shows how an asymptotic accuracy is reached for an increasing training file percentage. We should note at this point that the size of the penn corpus was relatively small (only a few thousand sentences), thus we can imagine these accuracies more quickly reaching their respective asymptotes more quickly in a plot of taggers which were based on the entire penn corpus. For this data, however, we can interestingly observe that it may not make sense to use a training file size past a certain percentage; such as going beyond 80% for the penn corpus-based tagger. If not much accuracy can be gained per percent increase in training size, after some point, it doesn't make sense for an individual to sacrifice run time performance for such a minor gain in accuracy.

Discussion of Results

This investigation explored how suitable NLTK is to a variety of potential applications -- from small scale mobile processing, to systems which might be involved in high performance cluster computing. As discussed earlier a fundamental, and critical, functionality in natural language processing is part of speech tagging. In order to evaluate NLTK, the accuracy of one of its basic taggers, the TnT tagger, was evaluated by training TnT on the MASC and Brown corpora. Varying portions of these corpora were used as training data in order to gain a better insight into how much data is necessary for a tagger to make accurate predictions. NLTK was also evaluated by how fast its classes could run by training and evaluating the taggers on systems with different CPU-memory organizations. To gain a better insight into how NLTK's TnT tagger is optimized, the taggers were compared to a custom bigram, hidden markov model, tagger implemented in Python, NLTK's language. In evaluating these systems, the pros and cons of using python as a language of choice were weighed with the observed accuracies and run times in order to gain a better insight into how suitable NLTK is for the "Average Joe's" of natural language processing.

The NLTK library is based solely off of a python base, which as mentioned in the introduction, is up to 400x slower than it could have been if it were based in a language like C. However, even though Python is slower, we found that NLTK is meant to encompass a wide variety of NLP

applications as well as making it easy to access and use. As a result of this goal, we came to conclude that due to the scope to which NLTK is meant to achieve a language like Python has an advantage over other quicker languages that tend to require more knowledge and skill to use as well as having some better memory allocation properties. Through the use of ANOVA, we described how the main hardware bottleneck was the CPU and that memory had little to no overall impact on the overall performance of either tagger. Concerning the results gained through the accuracy and runtime results, we found that our quick tagger implementation significantly outperformed the TnT tagger in mainly time but also accuracy. This finding points out that even though NLTK includes a large variety of tools for NLP, there is still a lot of work and analysis that needs to be done in the field in order to increase the overall productivity and accuracy of its applications.

The results obtained from the accuracy experiments showed that the accuracy of a tagger depended more on the type of training file being used, not necessarily the size of the training set. Though the size of the training sets did impact the accuracy of each tagger, we surprisingly saw that there was not a significant difference in accuracy for small and large training files. This was further reflected in the tag set granularity analysis, where we could see accuracy reaching various horizontal asymptotes for each tagger. Furthermore, these results indicated a “goldilocks” scenario of sorts for tag set granularity -- the MASC and Brown-based taggers were significantly outperformed by the Penn-based tagger, indicating a tag set which is not too fine, nor too coarse, in its part of speech tags is optimal.

Overall, this experiment showed that though Python is easy to use and memory efficient, NLTK still has ways to improve its base tagger classes. As mentioned earlier, for an individual who is not concerned with a custom implementation of a tagger, NLTK provides a suite of tools which are suitable for such applications. For individuals who want a tailored application, it is necessary to turn to NLTK’s base classes for taggers, specifically the TnT tagger, which was drastically outperformed by our simpler tagger implementation. The results of the experiment showed that NLTK has the added benefit of Python’s conservative memory techniques, which eliminate amount of memory as a significant performance factor -- focusing instead on CPU and training file size. The results also showed, however, that the more complex smoothing and trigram

methods involved in TnT actually detract from performance, as our unsmoothed bigram implementation never performed worse. Because of this, it is reasonable to conclude that NLTK does offer a critical foundation for natural language processing to individuals who are just being exposed to the field, but for those who wish to create high-performance, custom, applications there is still some work to be done. Perhaps as NLTK expands as the NLP enthusiast's toolkit of choice, it could focus on porting its implementations to faster languages, like C and Java, and offer a wider variety of less-complex, faster, taggers.

Concerning the overarching implementations of part of speech taggers in natural language processing, independent of any APIs or languages, this investigation showed that the accuracy of a part of speech tagger largely depends on the type of training set used -- not necessarily the size. Though size does impact how well a tagger predicts tags, the accuracy boost gained from increasing file size varies greatly from training set to training set, and doesn't offer much of a boost in most cases. For those individuals who are concerned with getting that extra 1% in accuracy, the run time increase may well be worth increasing the training file size, but for others it might be necessary to think twice. This investigation also showed that training sets which are either too fine or too coarse are not good for tagging -- the optimal granularity for a tag set should be somewhere in between, and will drastically outperform the alternatives.

Looking back on the design of our experiment, there were some improvements that we found could be made in future related experiments. For instance, enforcing stricter, constant training file sizes as well as using cross fold evaluation would be a step towards increasing the consistency of results. One other improvement that could have been made would have been to evaluate our quick tagger on multiple different corpora instead of just the Brown corpus in order to gain a better understanding on how different corpora affect different implementations of taggers. Finally, after more research, we found that NLTK does provide a "Fast Tagger" which could have been utilized to increase the variety of taggers tested; however, this tagger has a very specific application and would not have been a good addition for considering custom tagging applications, which is what our measurements and analysis were more focused on.

References

http://www.nltk.org/

http://www.coli.uni-saarland.de/~thorsten/tnt/

GitHub Link to Quick Tagger Source Code:

https://github.com/kwiese/Quick-Tagger

Code Appendix

All analysis was done in the R language for this investigation

```
nltk_data = read.csv(file.choose(), header=TRUE)
g_data = read.csv(file.choose(), header=TRUE)
nltk_data$train_perc = factor(nltk_data$train_perc)
nltk_data$cpu = factor(nltk_data$cpu)
nltk_data$mem = factor(nltk_data$mem)


nltk_qt_m_t = lm(qt_t ~ train_perc*cpu*mem, data=nltk_data)
a = anova(nltk_qt_m_t)
sse = sum(a[[2]])
qt_acc_perc = a[[2]]/sse

nltk_b_m_t = lm(br_t ~ train_perc*cpu*mem, data=nltk_data)
a = anova(nltk_b_m_t)
sse = sum(a[[2]])
b_acc_perc = a[[2]]/sse

nltk_m_m_t = lm(masc_t ~ train_perc*cpu*mem, data=nltk_data)
a = anova(nltk_m_m_t)
sse = sum(a[[2]])
m_acc_perc = a[[2]]/sse

par(mfrow=c(1,2))
cpu_ft = t(data.frame(qt=nltk_data[1:4,7], br=nltk_data[1:4,8], m=nltk_data[1:4,9]))
barplot(cpu_ft, beside=TRUE,
names.arg=c("Small/Big","Big/Big","Small/Small","Big/Small"),col=c("blue","red","green"),leg
end=c("Quick Tagger","Brown Tagger","MASC
Tagger"),args.legend=list(x="topright"),ylim=c(0, 18500),main="Fast CPU Test
Times",ylab="Time (s)",xlab="Percentage of Training File/Memory Size")
cpu_st = t(data.frame(qt=nltk_data[5:8,7], br=nltk_data[5:8,8], m=nltk_data[5:8,9]))
```

```
barplot(cpu_st, beside=TRUE,
names.arg=c("Small/Big","Big/Big","Small/Small","Big/Small"),col=c("blue","red","green"),leg
end=c("Quick Tagger","Brown Tagger","MASC
Tagger"),args.legend=list(x="topright"),ylim=c(0, 18500),main="Slow CPU Test
Times",ylab="Time (s)",xlab="Percentage of Training File/Memory Size")

par(mfrow=c(1,2))
mem_ft = t(data.frame(qt=nltk_data[c(1,2,5,6),7], br=nltk_data[c(1,2,5,6),8],
m=nltk_data[c(1,2,5,6),9]))
barplot(mem_ft, beside=TRUE,
names.arg=c("Small/Fast","Big/Fast","Small/Slow","Big/Slow"),col=c("blue","red","green"),leg
end=c("Quick Tagger","Brown Tagger","MASC
Tagger"),args.legend=list(x="topright"),ylim=c(0, 18500),main="Big Memory Test
Times",ylab="Time (s)",xlab="Percentage of Training File/CPU Speed")
mem_st = t(data.frame(qt=nltk_data[c(3,4,7,8),7], br=nltk_data[c(3,4,7,8),8],
m=nltk_data[c(3,4,7,8),9]))
barplot(mem_st, beside=TRUE,
names.arg=c("Small/Fast","Big/Fast","Small/Slow","Big/Slow"),col=c("blue","red","green"),leg
end=c("Quick Tagger","Brown Tagger","MASC
Tagger"),args.legend=list(x="topright"),ylim=c(0, 18500),main="Small Memory Test
Times",ylab="Time (s)",xlab="Percentage of Training File/CPU Speed")

par(mfrow=c(1,1))
acc_data = t(nltk_data[c(1,2),4:6])
barplot(acc_data, beside=TRUE, names.arg=c("Small (10%)","Big
(90%)"),col=c("blue","red","green"),legend=c("Quick Tagger","Brown Tagger","MASC
Tagger"),args.legend=list(x="topright"),ylim=c(0, 1.25),main="Tagger
Accuracies",ylab="Accuracy (%)",xlab="Amount of Training File Used")

plot(g_data$percentage, g_data$brown, pch=0, lty=2, lwd=3, type="b", col="red",
ylim=c(0.47,1), main="Tagger Accuracy vs Size of Training File (Penn is Baseline)",
xlab="Percentage of Training File", ylab="Accuracy")
points(g_data$percentage, g_data$penn, pch=0, lty=2, lwd=3, type="b", col="purple")
points(g_data$percentage, g_data$masc, pch=0, lty=2, lwd=3, type="b", col="orange")
clip(2, 88, 0, 1)
abline(0.88,0,col="purple")
abline(0.75,0,col="red")
abline(0.71,0,col="orange")
legend('topright',c("Brown Tagger", "Penn Tagger", "MASC
Tagger"),lty=1,col=c('red','purple','orange'),bty='n',cex=0.75)

plot(g_data$percentage, g_data$training_time, type="l", main="Training Time vs Size of
Training File", xlab="Percentage of Training File", ylab="Time (s)")
```

Result Appendix

Granularity Results

percentage	brown	penn	masc	training_time
5	0.546255507	0.653521679	0.492734765	0.37613821
10	0.599719664	0.722778105	0.519626979	0.682872057
15	0.633159792	0.749264273	0.542832357	0.98768115
20	0.653584301	0.766529331	0.594231186	1.251214027
25	0.666199439	0.784971552	0.612665365	1.512205124
30	0.675010012	0.79968609	0.634352635	1.783311844
35	0.684821786	0.811065333	0.65408805	2.045107841
40	0.690228274	0.818520698	0.666666667	2.28164196
45	0.702442932	0.827153227	0.681847755	2.659052134
50	0.707448939	0.832842849	0.690956409	2.759758949
55	0.712855427	0.843437316	0.695727608	3.059401989
60	0.718462155	0.85540514	0.70396877	3.290658951
65	0.723267922	0.857563273	0.704836261	3.541774035
70	0.725670805	0.862271925	0.704836261	3.874016047
75	0.728474169	0.865214832	0.706137497	4.040028095
80	0.737084501	0.868942515	0.707655606	4.292201042
85	0.744693632	0.870315872	0.707655606	4.636209965

TnT vs Quick Tagger Results

train_perc	cpu	mem	qt_acc	br_acc	masc_acc	qt_t	br_t	masc_t
0.1	1	1	0.858164131	0.845364648	0.63369438	42.15170813	1092.553493	66.480124
0.9	1	1	0.924299798	0.924320148	0.851643582	43.80678487	5260.683573	688.4652069
0.1	1	-1	0.858164131	0.845364648	0.63369438	42.1639359	1083.507329	65.98715401
0.9	1	-1	0.924299798	0.924320148	0.851643582	42.89849901	5222.456951	679.293505
0.1	-1	1	0.858164131	0.845364648	0.63369438	184.995254	3588.021027	273.1057649
0.9	-1	1	0.924299798	0.924320148	0.851643582	188.790025	14228.50364	2220.862996
0.1	-1	-1	0.858164131	0.845364648	0.63369438	185.8623269	3616.409827	272.3999159
0.9	-1	-1	0.924299798	0.924320148	0.851643582	188.0844271	14312.77754	2229.841858