B.Sc. Software Development – Artificial Intelligence (2020) Supplementary Assignment

A Language Detection Neural Network with Vector Hashing

Justin Servis – 10023036@gmit.ie

Brief Application Overview

- Simple UI with numbered options for dataset and network creation, building, training, saving and loading.
- Options to enter parameters for network creation, e.g. vector length, max epochs, max training time

Quick User Guide

- The application has my final network parameters hardcoded. These parameters can be changed but it is recommended to run the application defaults once for best results.
- I have split the menu into several subsections for extra clarity. I recommend running this in a tall terminal window because of this.
- From the main menu, simply select options numbered 1 to 4 in order to build the dataset, build the network, train the network and validate the network against the training set, respectively.
- A single file or folder of files can be specified using options 5 and 6. Classification of all files in a directory will be attempted, regardless of file type.
- A pre-processed vector file can be loaded. Obviously, this file needs to match the vector size of the current network to work correctly.
- The menu does not account for every possible scenario, it was not my focus and the parameters were added later.
- For setting the parameters, please bear in mind that:
    - Vector length should be set before creating a new dataset and before building/training a network.
    - Max Epochs, Max Time and Min Error should be set after the network has been built but before it is trained.
    - If you wish to train within a certain number of epochs, set epochs to the desired value and Max Time and Min Error to some unattainable values. This rule applies for all three of these parameters.

Neural Network Creation

Vector Hashing and N-Gram Creation

From the outset, I thought that this would be one of the biggest factors in creating an accurate and successful network. I created a class for splitting strings into n-grams. I added functionality which allowed me to specify a maximum and minimum length for the resulting n-grams, e.g. min=2, max=3 would result in all 2-grams and all 3-grams contained in the string. Each n-gram is returned as a hash of its string value and vector hashing is then performed on each n-gram. All vectors are normalised to between 0 and 1 - although I may experiment with this further down the line, I don't see it as being a major factor in performance of the network. Before starting, my initial thoughts are that the vector must be of a reasonably large size for the network to be accurate.

I did not process the source data in any other way at this point - no punctuation or special characters were removed, and letter case was left alone. I reasoned that while these intricacies may not be quite as important as a language's vocabulary, they do still have a bearing on its composition and structure. While this was possibly the simplest approach to have taken for this portion of the assignment, I fully intended to revisit and improve this processing at a later stage.

I decided at this point to get the ball rolling and create a network with the aim of determining whether the vector size would show any real impact on even the simplest of networks. I would begin

with a vector size of 50 and keep doubling this value until I reached 1600 at which point I would review and evaluate the data.

Logging Network Data

I created a small class for keeping track of a network's performance and a CSV writer to go along with it. Each time a network completed, the performance information would be appended to a CSV file where I could review it at a later stage.

I recorded the following data for each network:

- Vector Size
- Min and Max n-gram size used to create vectors
- Epoch data:
    - Epoch number
    - Error
    - Duration
    -

Creating the Network

I began by creating a very simple Encog network. My reasoning was that I should first try to find out if the vector size had any noticeable impact on a very simple network. Using the GPR rule of thumb for my initial number of neurons in hidden layers, I created a network with a single hidden layer with a Rectified Linear Unit activation function. For processing the source data, I only used 2-grams. I will be referring to this GPR value for each network throughout the rest of this document.

The entire network now looked like this:

Input Layer: (n=vectorLength)

One Hidden Layer: ReLU(n=GPR)

Output Layer: Softmax (n=235)

I have chosen SoftMax for the output layer since it is well known as an activation function suited to multiple classification.

I then ran this network with a range of vector lengths. Beginning with a vector length of 50 (v=50), I ran multiple networks within a loop, doubling the vector length each time. I set each network to run for as many epochs as it could start within 200 seconds.

Results - First Runs

I won't include the Epoch vs Epoch Error for each length here since there are far too many epochs for the lower vector values, but it showed that each of the networks up to 800 all tended to reach a minimum after a certain number of epochs. 800 managed 9 epochs while 1600 managed only 3 and both started off with a lower error than all the other networks. Given more time, the vector 1600

network may perform very well. It would stand to reason that reducing the number of neurons in the hidden layer would allow for faster epochs for all networks.

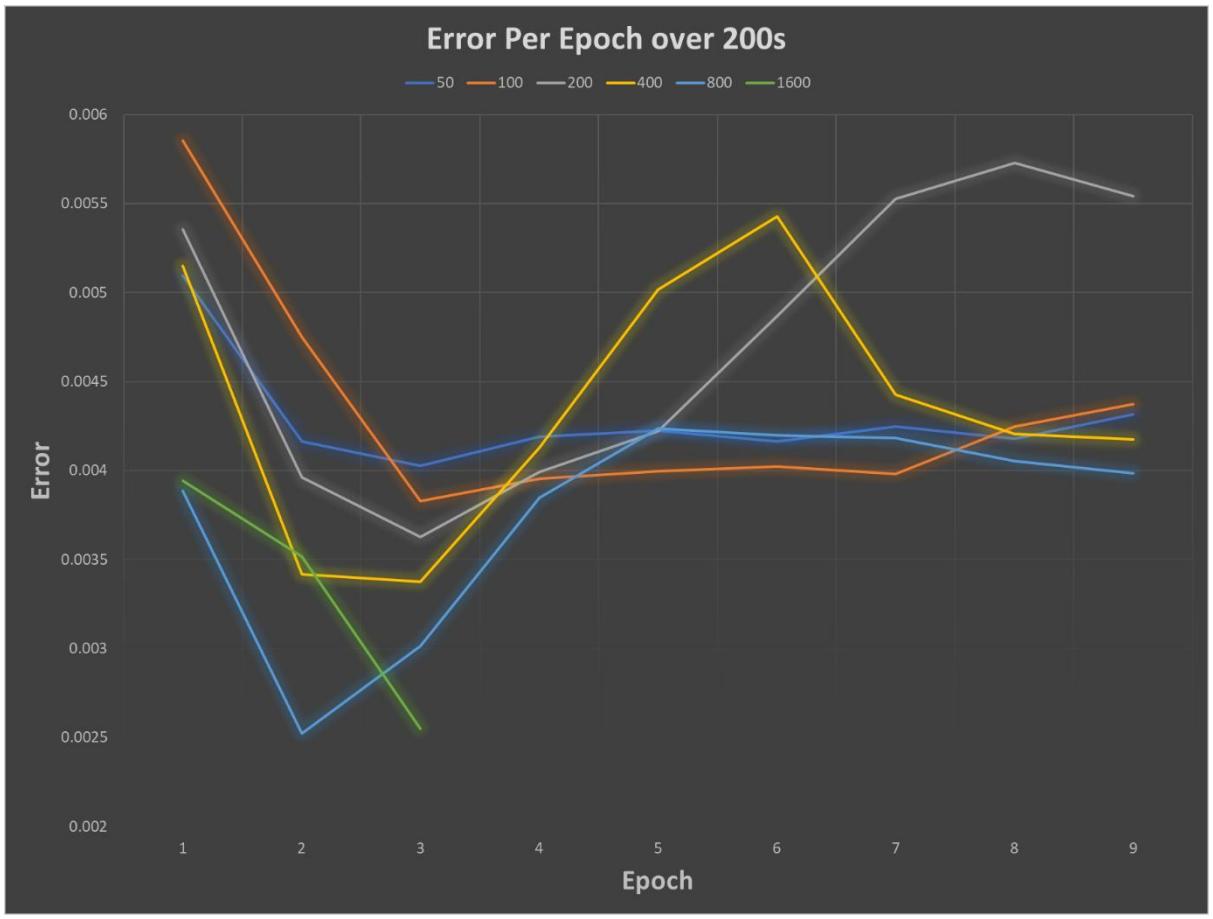


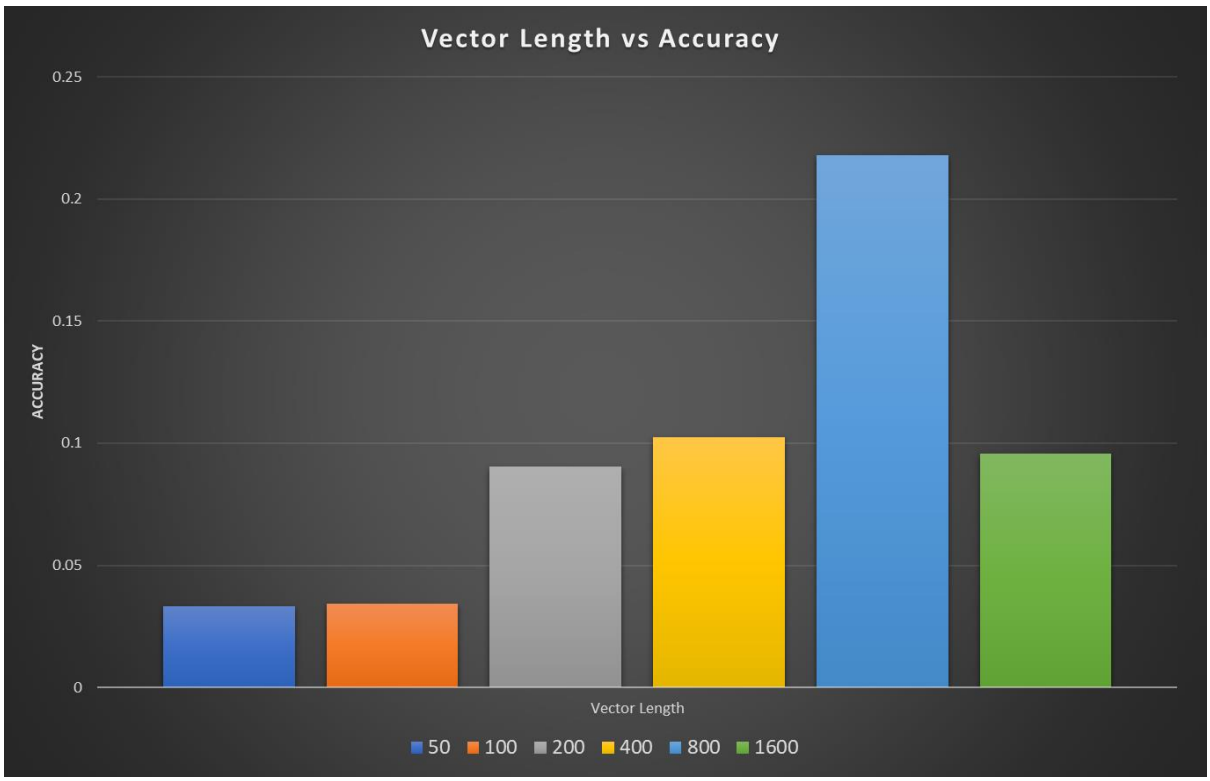
Figure 1- Vector Length vs Accuracy over 200 seconds with one ReLU activated layer of n=GPR neurons

Fig 1. is a chart of this run of networks which plots the error of each network per epoch over the first 9 epochs. It should be noted that both the 50 and 100 networks reached a minimum of ~0.0044 after several epochs and the error never rose thereafter.

Adjusting Hidden Neurons

As I mentioned above, I reasoned that allowing the larger networks to iterate over more epochs may help them reach a lower error. To achieve this, I expected that lowering the number of neurons relative to input size in the hidden layer would help. I reran all the above networks, this time using n=(GPR * 0.5) neurons in the hidden layer. I changed nothing else in the configuration.

This was interesting, by reducing the number of neurons and hence increasing the number of epochs as expected, the vector of 800 is now the best performing network, albeit at just over half of the accuracy of the previous configuration. This can be seen in Fig. 2.



At this point, I decide to revert to the previous configuration and take another approach. I still had not changed any n-gram sizes and had not altered the network itself in any way, aside from the number of neurons in the hidden layer. Having set out initially to get a broad idea of how the input vector may affect the network, I think I have somewhat had at least a glimpse of this so far.

Changing Vector Hashing

While doing ongoing research into an appropriate topology, I decide to take another look at how the source text is being processed and the n-grams are being created. While it would certainly be worthwhile to look at the size of the n-grams themselves, I think it may be more valuable at this stage to further diversify the relatively small amount of source data at hand.

I decide to implement further processing and set about it like so:

- Keep the original simple n-gram process - retain all punctuation and uppercase characters.
- Add processing of the text in all lower-case.
- Split the text into words and n-gram these.

To maintain some sort of baseline, I reran the first network with this new processing for each vector length:

Input Layer: (n=vectorLength)

Hidden Layer: ReLU(n=gpr)

Output Layer: Softmax (n=235)

In Fig. 3 is a graph of the error per epoch for each vector length over 200 seconds. To keep the graph more concise I have trimmed some of the later data for the smaller networks since they reached a minimum error and never improved thereafter. From looking at this graph, the new processing strategy may have improved the accuracy of the vector 1600 network by ~33%. Once again, the lower vector lengths are performing poorly. At this stage, maintaining a larger vector size while reducing the number of neurons may yield further improvements. These results are great since this is my starter network in its initial state, having reverted from n=(GPR * 0.5) back to n=GPR for the number of neurons in the single hidden layer.

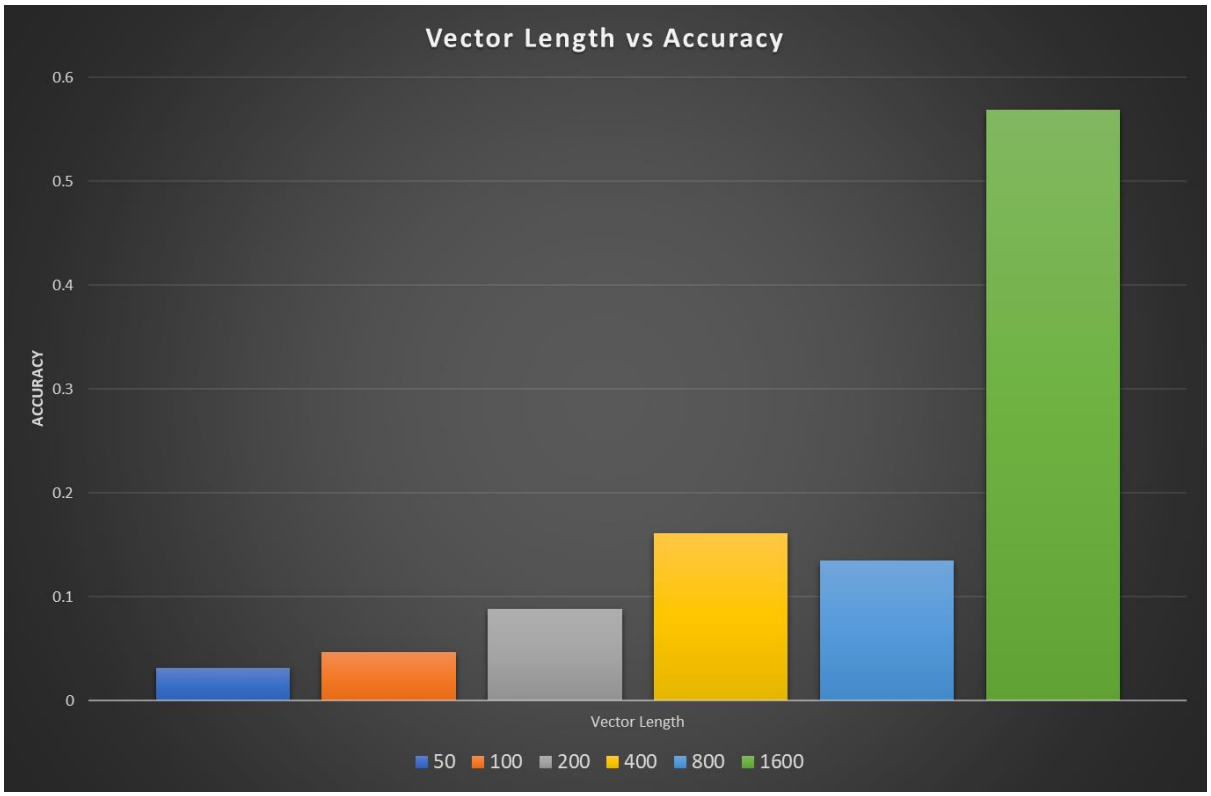


Figure 3 The final accuracy of each vector after 200 seconds

The network with vector length of 1600 achieves 59% accuracy with no adjustments to the topology whatsoever. It should be kept in mind that this accuracy is obtained against a set of data which the network has been trained with, so the real-world accuracy may be quite a bit less but it's all that there is to go on for now.

These results are very interesting and I'm not entirely sure how I proceed - I could test with different n-gram parameters and see what effect that has, or I could simply reduce the neurons in the hidden layer.

Adjusting Hidden Neurons Again

Since I reduced the number of neurons before changing the vector hashing, it would be worthwhile to do it again and compare the results. I feel like I should probably stop testing with the lower vector lengths, but I will keep them in for now so that they can also be compared to the previous accuracies with the reduced number of neurons. So, I ran all 6 vector lengths again with half of the hidden

neurons previously used to see what effect it had. After adjusting the hidden neurons, the results were not good, and a different approach is needed at this point.

Assessment of Progress So Far

From the best two sets of results, show previously in Fig. 1 and Fig. 2, the extra processing of the source data is having a more positive effect on the larger vector size. This would make sense since there is a wider range of hashes being input into the vectors, so a larger set allows for more diverse input data. It looks like I should probably stick with a larger vector size for now and start to reduce the number of neurons in the hidden layer to see at what point is there either a negative impact on the minimum error or overall accuracy of the network.

The chart in Fig. 4 was created from all the data gathered thus far and highlights the positive impact of the increased string processing on a larger vector size and the negative impact on a smaller vector size.

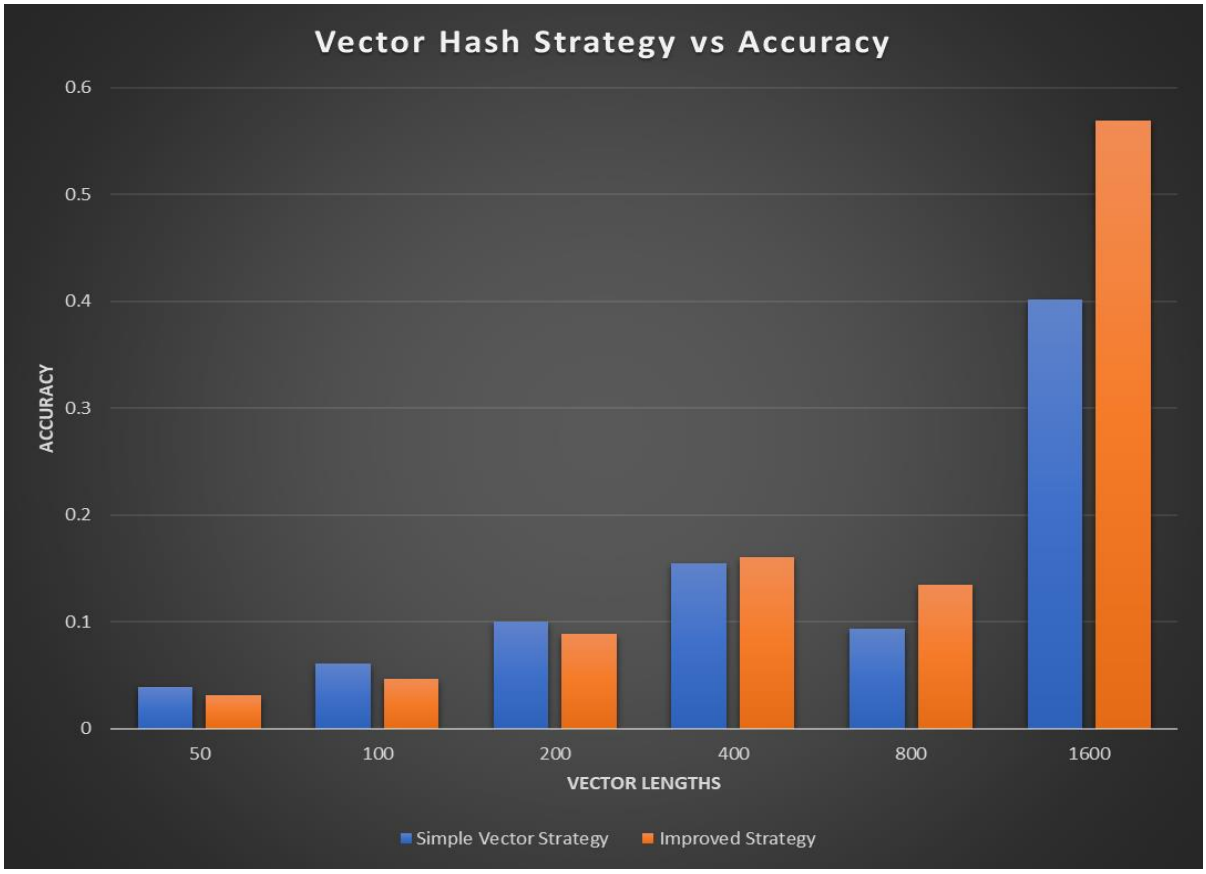


Figure 4. A comparison of the basic n-gram strategy vs the more diverse one.

Activation of First Hidden Layer

It may be time to drop the smaller vector sizes and focus more on larger ones for now. This can be adjusted and tested again later once a better topology has been created. Trying to keep the topology as simple as possible, I attempted using a Sigmoid activation function on the hidden layer thinking that it was a good choice to start with since it is known to be especially useful for models where a

probability is the expected output. Since the probability of anything is in the range of 0 and 1, Sigmoid seems to fit. After several attempts of using Sigmoid on its own with very poor results, I then turned to the TanH activation function.

I began with a network like so:

Input Layer: (n=vectorLength)

Hidden Layer: TanH(n=gpr)

Output Layer: Softmax (n=235)

The accuracy achieved by this network was roughly equivalent to using the ReLU hidden layer (~60%) with the same number of neurons. I then began decreasing the number of neurons to see what the effect was on accuracy. I reduced by 50% each run of the network until accuracy dropped and then slowly added more back to find a sweet spot, my thinking being that perhaps I could squeeze just enough out of it by having it run for another epoch. I ended up with n=(GPR * 0.2) neurons for the hidden layer and could not increase the accuracy by any means. I tried to add a little dropout which should, in theory reduce the time taken per epoch and prevent the network from overfitting but this seemed to have very little if any effect on the performance of the network. Coupled with the fact that the Encog documentation does not seem to encompass every feature of the library, I started looking for other options for the last bit of tweaking.

Other Options

I looked at some of the other activation functions provided by the Encog library and had good success with the Elliott Symmetric function. From a bit of research, I found this description of the function:

"The advantage of this transfer function over other sigmoids is that it is fast to calculate on simple computing hardware as it does not require any exponential or trigonometric functions. Its disadvantage is that it only flattens out for large inputs, so its effect is not as local as other sigmoid functions. This might result in more training iterations, or require more neurons to achieve the same accuracy." (https://www.mathworks.com/help/deeplearning/ref/elliotsig.html)

Using just this function and with all other parameters kept the same, I could achieve ~97.7% in 175s. From here, I again tried to tweak some parameters slightly to break the 98% barrier but could not quite get there.

I created a set of 45 text files between 500 and 1300 words in a variety of different languages files for the network to classify, since up until now the accuracy figure was solely based off the data which it was trained with. This validation provided great insight into how the network was performing and it worked very well with a couple of small, negligible errors presenting themselves -. two incorrect languages which were very close in nature to the source languages.

I was convinced that the network was serving its purpose well and validated the network against the large WiLI text file. The network predicted close to 85% of the entries correctly which is fantastic considering the limited amount of data which it was trained with.

Final Adjustments

Knowing the network was performing very well but still shy of the 98% mark, I went back to my vector hashing approach and expanded on it even more. I added n-grams of each word along with

the hashes of each word, both in source case and after converting to upper case. I created n-grams from the text with all punctuation removed.

Rerunning the network again with the new vector hashing, it achieved a score of ~98.35% in 182 seconds. By then limiting the number of epochs the network was trained in and rerunning it, the following result was achieved:

Total Training Time: 172.370262s

Results Correct: 11517/11738

Accuracy 98.117226%

(CPU: Ryzen 1600, 12-threads @3.6Ghz)

Total Training Time: 82.970310s

Results Correct: 11560/11738

Accuracy: 98.483558%

(CPU: Ryzen 3700X, 16-threads @3.6Ghz)

Conclusion

While the network itself is structurally quite simple and straightforward with just a single hidden layer, trying to get the last few percent out of it was challenging. From early on, I think I had the right approach in thinking that one of the biggest factors for good accuracy was making the most of the source data and how it was processed into vectors. The network itself and its many iterations also threw some unexpected results at times and it is sometimes hard to reason exactly why. One thing that I have not covered in this document in any detail is the use of different sizes of n-grams. This is due to the relatively poor performance I observed for any combination of n-gram sizes greater than just 2-grams alone. This also had me thinking quite a bit as I first assumed that by incorporating perhaps 2 and 3-grams would yield better results. This was not the case for me at least.