Entity Resolution

Contributors

Timotheos Palaiologos Georgios Desipris

Abstract

Given a set of camera model descriptions , and a subset of their known correlations, we infer all the new possible similar and non similar pairs by adding them into cliques. Inductive reasoning is used to further expand our training set. By calculating the TF-IDF values of each concluded pair we train a linear regression model so we can predict the similarity of unknown pairs. The model goes through a retraining process by adding newly predicted pairs to the training set if they do not exceed a dynamic threshold. The starting set is split into 60% for training , 20% for testing and 20% for the final validation where the performance of our model is measured

File parsing

Folder structure

A folder path is passed via an argument. For the program to work properly the folder structure should be of the following format :

Folder containing the entire data set / Various folders with the name of each company / the corresponding item files of said company's camera models. It should be noted that the camera model files should be of .json format.

Json files

Like it was previously noted, a camera's description should be contained within a json file with the appropriate syntax. In this project we haven't implemented a fully functional json parser since the data set was predefined and there was no need for proper management of special cases (eg arrays within arrays) , while it would be preferred to further future proof this procedure , time was spent on more important aspects of the project.

Our parser can read the following formats :

A string within quotes (") followed by a colon (:) yet another string or an array of comma (,) separated strings and another comma at the end of the line. The left string of each line corresponds to a characteristic that is described by the right string's value. It should be noted at this point that said strings can contain various spaces , special characters , numbers and are not restricted to continuous characters only. The first line should always contain the word title at the left and the actual title of said item at the right.

W files

A W file is read containing the correlations between the json files. The format of said files goes as followed :

company1//id company2//id2 {0,1} (1 for similar , 0 for non similar)

The first line always contains the column names left_spec_id,right_spec_id,label

Pre-processing

Each json file has to be properly pre-processed before its TF-IDF values are calculated. That is so the values of our vectors are more precise and descriptive of the item's characteristics.

It is not that obvious if the title of each column (left value in each line) should be kept or stripped from the possible word list .

Let's think of the following example. If a json file containing the columns (hasX,hasY) and their corresponding values (yes,no) has its column titles removed , the meaning of "yes" and "no" is lost. Another item containing the column (hasZ) with the value (yes) would result in a faulty similarity between said items. Just because the word "yes" appears in both of their descriptions doesn't necessarily mean the context in which the value was used is the same . Therefore we can conclude that we would require a more sophisticated approach , such as word embeddings , to properly pre-process our data without losing the essence , grammar , meaning of each word.

Due to the scope of our project we use more mainstream methods. We do the bare minimum , by doing basic cleanup procedures to the various strings while splitting them into substrings when needed (eg "word1 word2" would be 2 words in our word list).

A small description of the mentioned procedures will be provided :

- Make all the strings lowercase
- Remove UNICODE strings (strings such as "word\u0001" appear quite often)
- Remove extra white spaces
- Remove punctuation
- Remove big numbers
- Remove STOPWORDS (STOPWORDS are commonly used words that provide little to no information , such as "is" "are" "the")

In the future it would be worth trying techniques such as lemmatization or/and stemming.

Calculating the IDF Dictionary

Now that we have all the preprocessed items, it is time to create the IDF Dictionary for our model. The IDF Dictionary is gonna be used for the calculation of the TFIDF values in the next step.

We already have a list with all our items and for everyone, a list with its remaining words. Our dictionary has the form of a Hash Table, so we can access each word's IDF value in (hopefully) O(1) time. To create this dictionary, we need to count how many times each unique word appears in our items' data. This means that for every item, we need to update our IDF Dictionary until there are no more items left. So, we go through our list of items, which is of O(N) time, and for every unique word of that item (again O(N)), we increment our dictionary's counter for that word. To make sure that the counter is incremented only once for each word of the item's data, we use a Hash Table to rid us of any duplicates. To get access to these counters and increment them each time, we use the Hash Table (dictionary) so the complexity is at best O(1).

After our last step, we have a dictionary that contains a counter for every unique word. We now compute the IDF value of every word based on these "sums". That means that the IDF value would be equal to log(number of items / sum).

Now we need to reduce the size of this dictionary, since it is inevitably going to be really large and we do not have enough computing power to handle it in the training step. The way we do this is we compute the TF-IDF values for every item. Then we sum all the TF-IDF values for each word and keep just the ones with the highest value(in our case 1000). The calculation of the TF-IDF values is gonna be described in the next chapter.

Creating TF-IDF Vectors

Now it's time to create the TF-IDF Vectors for our items. We already did this once before, but for every unique word there was. Since there is no need for every one of those words now, we will only compute it for the N words we kept after we trimmed our IDF Dictionary(in our case N = 1000). TF stands for term frequency and, logically, what we do is count how many times every word appears in our item's data. This is that word's frequency for that specific item. When we have computed these "sums", we multiply them with the IDF value of the word itself and so we have a TF-IDF value that corresponds to one word of a specific item. This means that if we had 1000 words in the dictionary and 100 items in total, we would have 100 vectors of size 1000, or a total of 100 * 1000 = 100,000 TF-IDF values.

The complexity of this task is effectively O(N*M) where N would be the number of words in the IDF Dictionary and M would be the number of our items. Why is that? The complexity would be a lot higher if we did not use a Hash Table to represent each vector. This is done by having the words as keys and an array for the TF-IDF values as the value of the Hash Table. This eliminates any complexity each time we have to increment the counter for a word. This means that when we need to do so, we just need to get the array for that specific word from the Hash Table, which would be at best O(1) complexity.

Now let's explain why these values matter at all. They act as a metric for word importance. If a word appears in a lot of items' data, it will have a higher IDF value, which means it is pretty much irrelevant, since being in every item's data, is pretty much like being in no data at all!

On the other hand, the TF value indicates how important a word is in the item's data.

Thus, when paired together, we have a TF-IDF value which indicates how important a word is to an item in our collection of items.

Clique group

Our initial training test does not suffice for the training of our binary classifier. We can infer new pairs by using inductive reasoning on the already known correlations. The logic goes as followed :

- If A is similar to B and B is similar to C then A must be similar to C.
- If A is similar to B and B is not similar to C then A is not similar to C.

In every other case we do not have enough information to decide whether or not two items are similar or non similar.

Implementation

To implement the mentioned behaviour we create cliques of items within a struct called CliqueGroup. While we could create a complete graph structure to store this information , that would lead to a slower solution since we are only interested in the groupings of the items within the graph.

We start with 1 clique per item , ignoring the W file at the initialization step. Each item can be found via a Hash map , when provided with its ID. When adding a positive correlation all we have to do is merge the cliques of said items (joining the 2 lists). The merging is always done in a bottom up order. To keep track of all the cliques a list of cliques is stored. It is important when 2 cliques are merged for the appropriate node from the cliques list to be removed.

Our hashmap points to the items of the cliques and not the cliques directly. This way we avoid updating all the old cliques pointers that were lost when the merging occured.

Since we don’t want our Item struct to contain Clique pointers, we created an intermediate layer. Our Cliques contain pointers to said structs , we can have the same functionality without ruining the abstraction of our cliques.

So in short our hashmap points to the intermediate level called ItemCliquePair , an ICP points to the clique in which said item belongs to and the item struct itself. A clique is a list of ICP pointers. It should be noticed that there is a bidirectional relationship between cliques and icps.

Each clique also stores a list of non-similar ICPs. This is useful for when a non similar pair is introduced. At this point we have to be careful. If A clique contains 2 non similar pairs that

end up being in the same clique , after a future merge , our list will have duplicate information. This is problematic because when we extract the newly obtained pairs from our cliques we will produce duplicates if not handled appropriately.

There are 3 possible solutions to this problem

1. Check for duplicates at the extraction phase.
2. Check for duplicates at the insertion phase of the non similar ICP
3. Remove all the duplicates at the end

For solution (1) , while it will work , over time we will produce even more duplicates with each retraining making this procedure worse (more on retraining in upcoming chapters)

For solution (2) , while we don't add duplicate information at the insertion of a new ICP , we don't solve the case where 2 cliques may merge in the future so we will still have duplicates! It is also more time consuming than method (3)

For solution (3) , we remove the duplicates when it matters , while providing the best time complexity of all the 3 options. This procedure is assisted by the use of a hashmap making the removal of duplicates O(N). So we can expect a complexity of O(M*N) when we have M cliques.

Pair extraction

All is left to do after the formations of the cliques , is to extract all the newly created pairs. This step is pretty straight forward , we go through each clique and form all the possible pairs that correlate positively or negatively with said clique. Hash maps are yet again used to make sure we don't add a pair twice (A-B and B-A is considered the same pair). To make the procedure easier we firstly calculate all the similar pairs and then join them with the non similar ones. The list of pairs is then shuffled since we don't want our model to train on only 1 class in each batch. (More on the training method in the upcoming chapter)

Logistic Regression Model

After the calculation of the tf-idf values for each item we want to train a logistic regression model with all the possible pairs we can extract from the clique groups. Given a set dictionary size we know each vector should be 2 times larger than said value , where the first half will contain the values of the first item and the latter of the second. Like it was previously mentioned , when creating the IDF dictionary we only use the 1000 most frequent words to limit the size of each vector, lessening memory usage and completion time. So in our specific case our training vectors would be of size 2000.

Memory concern

Our training set is quite large , we can expect about ~300k pairs from the clique groups when using the large W file. Even by having a vector length of 2000 there is a memory concern to be considered. We have 300k * 2000 possible values and each value in our program is stored as a double. Let's say a double is 8 bytes in most systems , we can expect 300k * 2000 * 8 bytes for the storage of our training set. This would mean that our program requires about 4.8gb of memory ram for only the training set!

Solution

To solve this problem we came to the realization that calculating the tf-idf values of each possible pair is both a waste of time and memory. We only require the vectors of the items in isolations, then we can produce all the possible pairs by referencing their index in the table. To make the explanation more understandable let's provide an example. Instead of calculating a vector of size 2000 of the following format [item1 1000 | item2 1000] we only need to calculate [item1 1000] and [item2 1000] once separately! If another item is paired with item1 or item2 its value will not be re-calculated. This way we only need to produce all the tf-idf vectors per item not per pair. Let's recalculate the memory usage and see what we have gained. We have about ~30k items (json files) in our data set , so our tf-idf values will now be 30k * 1000 * 8 bytes = 240mb. We also need to create an array of indexes since we don't have a way to form the pairs yet! Each index is stored as an int , we got 2 indexes per pair , considering an int is 4 bytes in most system , that makes the index array 2*4*300k = 2mb. Adding it all together we have gone from 4.8gb to only 242mb if we use arrays for the storage of our training set.

Training

After producing all the tf-idf vectors and the helper arrays mentioned above we are finally ready to train our model. For this step 3 variations of the logistic regression model were attempted , full batch , mini batch , stochastic descent. It should be noted that for all 3 , gradient descent was used for the actual training of the model.

Full batch training produces the best results , since the gradient is calculated based on all the data set in each step , but is also the slowest. Our data set is way too large so it would require quite a lot of time to work properly.

Stochastic descent is the fastest method , it only uses a random vector per iteration but produces worse results for the exact same reason.

Mini batch training is the middle ground. We split the training set into batches , so the training should be slower than stochastic descent but with better results and faster than full batch training but with slightly worse results. It should be noted that if the batch size is 1 then we get stochastic descent whereas if the batch size is bigger or equal to our entire training set , we get full batch training.

For this project mini batch training is used as the final implementation , since it is more versatile , can produce all of the 3 variations and provides the most opportunities for parallelism (more on that in later chapters).

After N batches , an average gradient vector is calculated before the update of the weights.

Hyperparameters and termination

The learning rate is chosen via an argument. The model does a predefined amount of epochs but can be prematurely terminated if the distance between the previously set W vector and the new one is insignificant.

Noteworthy optimizations

By taking a closer look at the produced X array we can notice that 95% of the values are zeroes. Zeroes don’t affect the training process since the corresponding coefficients of the Wi variables will always result in a value of zero. Skipping said values will yield better speeds. When calculating the linear function of our model , we don't need to do 2000 iterations but only ~100 on average which is a huge boost.

To achieve this we use sparse arrays , each element is a tuple containing a non zero value and its corresponding actual position in the vector (as if it were an array). When calculating the linear function we go through each tuple and simulate the calculation of the same sum as if we were to skip the zero values. It should be noted that the order in which the values are calculated is not necessarily the same since the sparse array is implemented via a hash map and the key value pair list is not ordered in any way.

Eg : we might calculate f(x) = x1w1 + x99w99 + x5w5 + … + x3w3

It should be also noted at this point , that since we use an array of indices for each pair we calculate 2 sub sums for each item then add them together. The calculation is again the same as if the values were a continuous array , we only have to be careful when accessing the appropriate weight values. The sparse array of each item contains positions bounded to [0,999] so when calculating the sub-sum of the second item of the pair , we just have to add 1000 to each index to find the correct Wi variable.

When calculating the partial derivatives , if Xij is zero we skip the entire calculation since the rest of the formula is also zeroed out. We are able to find the Xij values in O(1) from the sparse array.

Order of items within a pair

The order in which the items appear within a pair matters a lot. It might seem logical that if A is equal to B then B must be equal to A , but that is not the case for our model if we don't handle that case appropriately.

Let's think of an example to explain why that is. Item A will have a vector Va whereas B will have a Vector Vb. If Vb is the second vector in our pair then the corresponding sub-sum would be the following :

Vb[0]*W[1000] + Vb[1]*W[1001] + … + Vb[999]*W[1999] + Vb[1000]W[2000]

Let's now inverse the order of the items inside the pair (making Vb the first item and Va the second). The new sub-sum now is the following :

Vb[0]*W[0] + Vb[1]*W[0] + … + Vb[999]*W[0] + Vb[1000]W[0]

We notice the W values are all offset by 1000 making the final computation completely different. This means that our model will not have the same prediction for the pair A-B and B-A even though they are the same identical pair!

To solve this problem we follow a very basic principle , the items within the pairs must be always ordered. This way even if we try to predict the pair B-A and the model has been trained with the pair A-B , the actual vector will still correspond to the A-B vector that has been used in the training process. Since in our implementation we produce the pairs as a set of indices (as mentioned in the previous section “memory concern”) all we have to do is sort the indices. (Since we only have 2 values we don't need a proper sorting algorithm but just a basic comparison between the two)

Possible ideas for even less memory consumption

Since we are using the mini batch training approach it would probably be wise to calculate each batch’s indices on the go to limit the active ram usage at any given time. It’s not clear if the performance hit is worth taking , further research should be done on the matter for proper measuring.

Data set imbalance

It is obtained that the given W files have an imbalance between similar and non similar pairs. The problem is that by following the clique procedure that was described in the previous sections we further reinforce that imbalance. Big cliques are created and the formed pairs get exponentially bigger. For each newly added item in a clique of N items we get at least (N-1) new similar pairs (a new item is similar with every already placed item in said clique). New negative connections are also created , making new non similar pairs , but they don't seem to solve the data imbalance since they are far less.

This is problematic since our model can be heavily biased towards similar pairs and not generalize well with non similar pairs.

Dynamic Retraining

After the basic training of our model is done we go through a retraining procedure. The idea is that we want to further expand our training set by adding accurate predictions as if they were intiality in the training set. Then we repeat the same process until we reach better accuracy. We will expand our explanation on this topic in the upcoming section.

The algorithm

1. Produce some random pairs
2. Initialize a training set based on the pairs extracted from the cliques
3. Train a model with said training set
4. Predict the random pairs with our model
5. Accept the ones that have an accuracy above a set threshold for the current iteration
6. Add the accepted pairs into the cliques
7. Increase the threshold for the next iteration
8. Repeat (2) for X iterations

Let's go into further detail for some of the steps.

For step (2) , we want to produce some new random pairs that we know nothing about , to enhance our data set. We got 30k items so all the possible combinations are C(30k,2) ~= 450m. It would not be possible to add them all to our cliques , so a subset will do. It should be noted that even a small amount of pairs can drastically increase the size of a clique so adding a few pairs can exponentially increase time! The number of pairs to be added should be big enough for the retraining process to have an effect and small enough for it to be computable. (In our project , when the pairs become increasingly big , a safety guard stops the production of all the possible combinations from within the cliques to avoid crashes)

For steps (2),(7) , we want the threshold to be small enough so the model doesn't add wrong predictions to the training set.

For step (6) , we should be aware of 2 cases.

An accepted pair may be contradictory to a clique's data (Predicts 2 items are similar when they are not). In that case the pair is thrown away.

The order in which we insert the accepted pairs into the cliques matters. An accepted pair can create a scenario where the first case occurs for an upcoming accepted pair. In this case which pair should be accepted and which thrown away? The answer is the one with the highest accuracy should be prioritized. Since we don't know what the correct prediction should be , we measure the smallest distance from 0 or 1 accordingly.

To achieve this in our project , we simply sort the accepted pairs based on said distance before going on step (6). This way we always add the pairs with the highest accuracy first and discard the ones that produce conflicts in the cliques.

Concern on the training bias

The model is being trained by its own predictions , meaning it’s relying on its own bias on whether or not something is true (since it can't know if something is factually true). If our starting model is biased towards a specific class , then we can expect our model to reinforce that idea in each iteration which may or may not be problematic.

In our case , like it was previously mentioned , our data set is heavily biased towards the class 0 (non similar pairs) so we can expect our model to be way more biased on said class , at the end of the retraining process.

In short this means that our model will have very good predictions for items that might be non similar and worse predictions for items that may be similar.

The results of our training will be addressed in the upcoming sections.

Multithreading

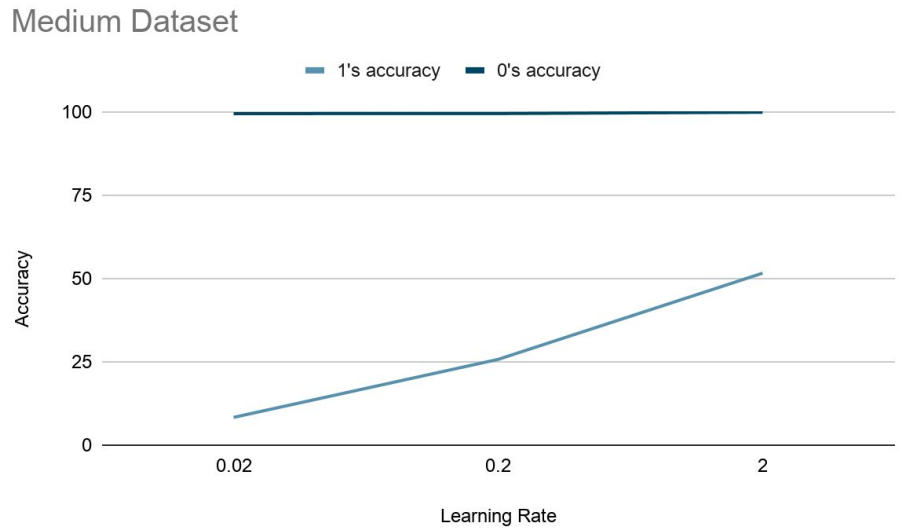
To make the training process faster , multithreading is used when possible. A thread pool handles job requests. Each thread finishes a task and outputs its result in an appropriate queue. After a set amount of jobs are complete , the main thread is synchronized with the job handlers and the results can be safely obtained.

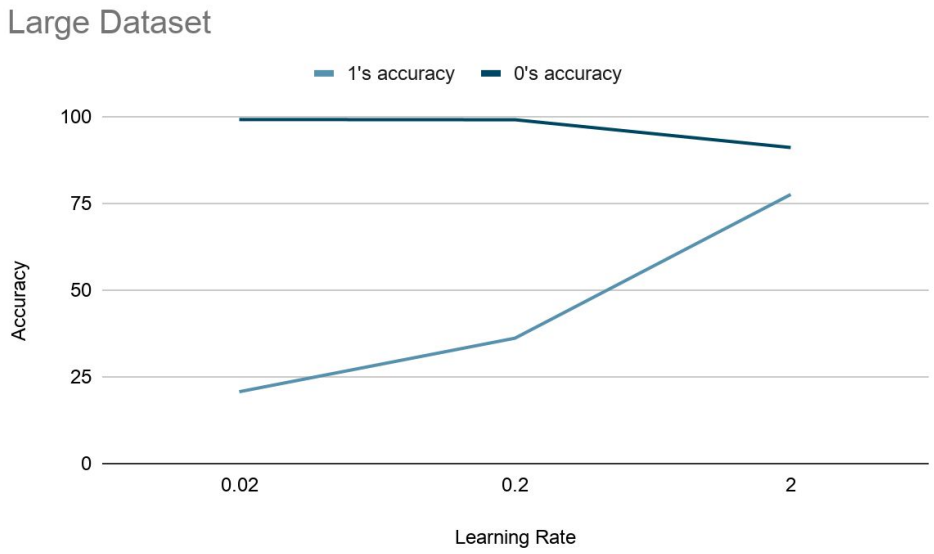
In this project we have 1 job per batch , meaning we can compute N batches at once. After all the sub-gradients are computed the main thread can calculate the average gradient vector that is going to be used for altering the weights of the current iteration.

Results and Observations

Learning rate

First off, we need to determine a good learning rate for our model. Below is a graph that shows the percentage for 0 and 1 value accuracies with 3 different learning rates at **24 epochs** and **512 batch size.**



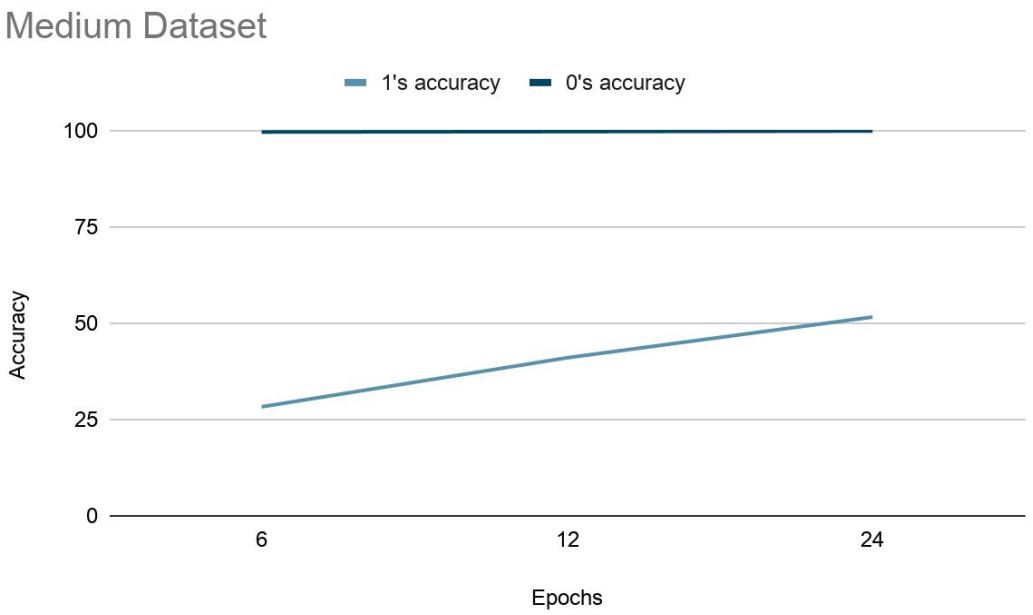


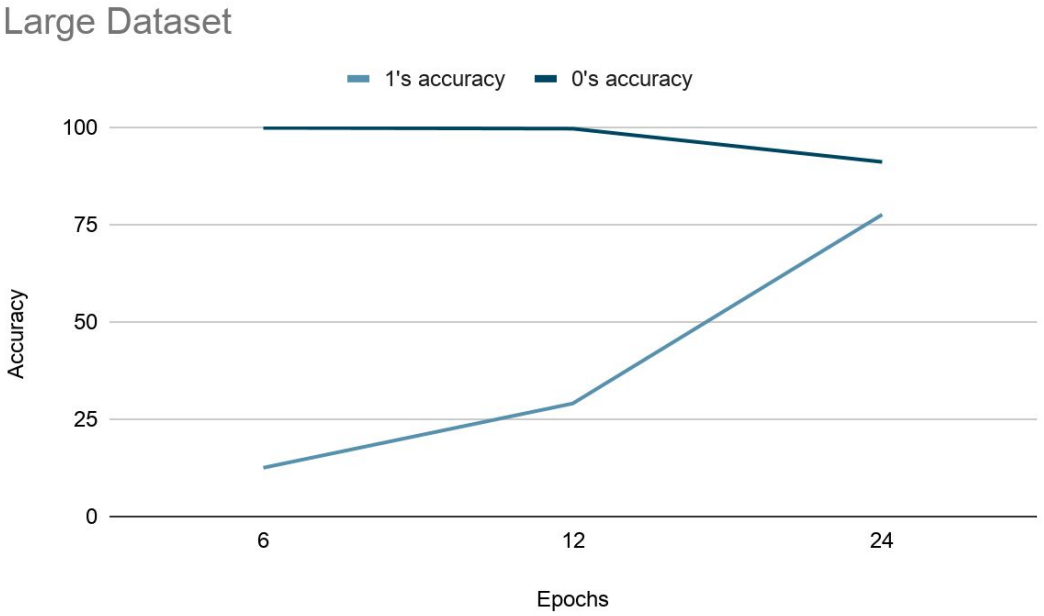
We can safely say that the best **learning rate** of the three is **2**, so we are going to work with that. With a bigger learning rate, the model seemed to not be able to converge!

We can also see that the model is trained better for the 0 values. This happens because our dataset contains more non-identical pairs than identical.

Epochs

Now it's time to compare the results while changing the **epochs**.
We will use: Learning rate: **2.0** Batch size: **512**

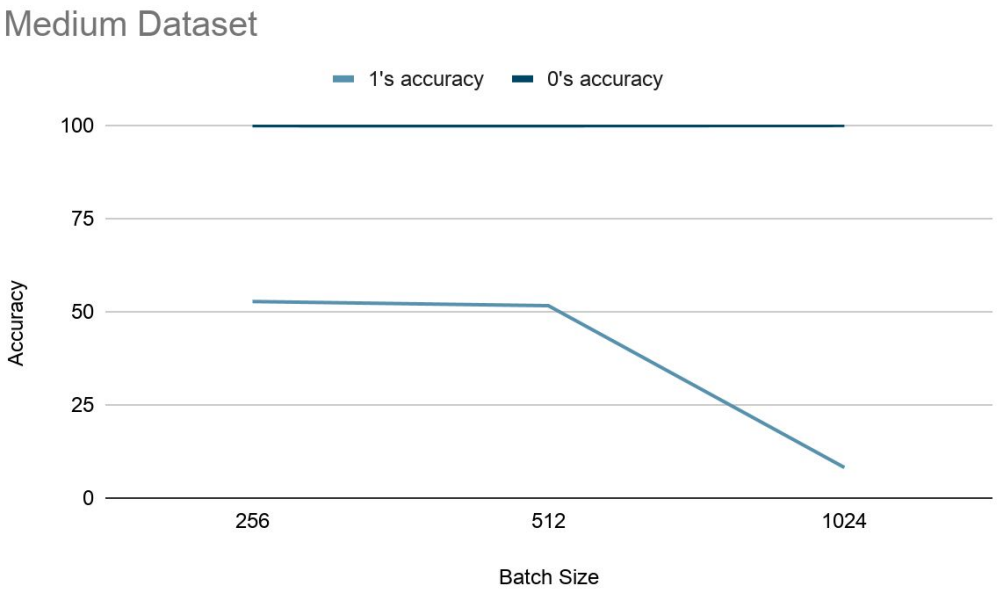


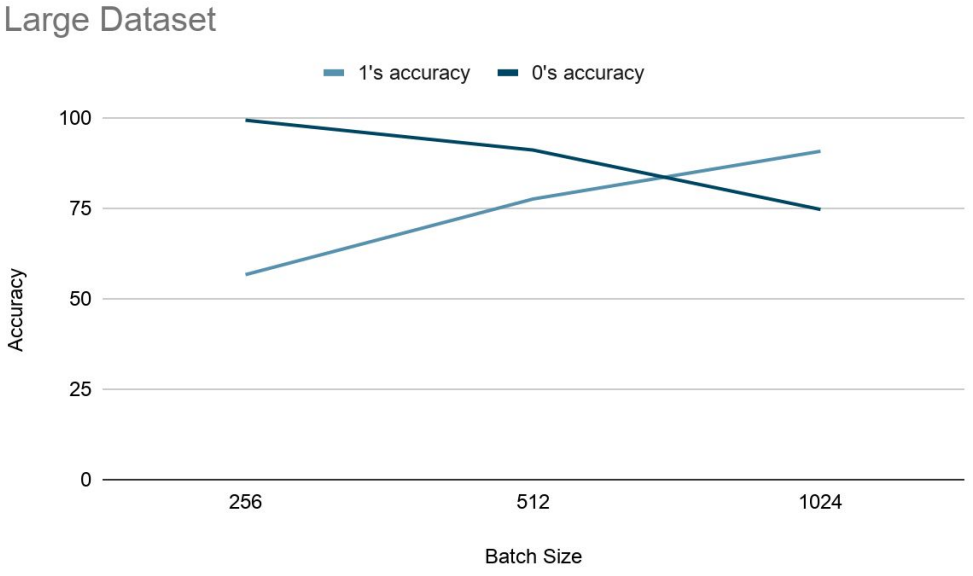


It is obvious that with more epochs, the model will get better results. We can also see the two lines starting to meet at a point around 80%.

Batch size

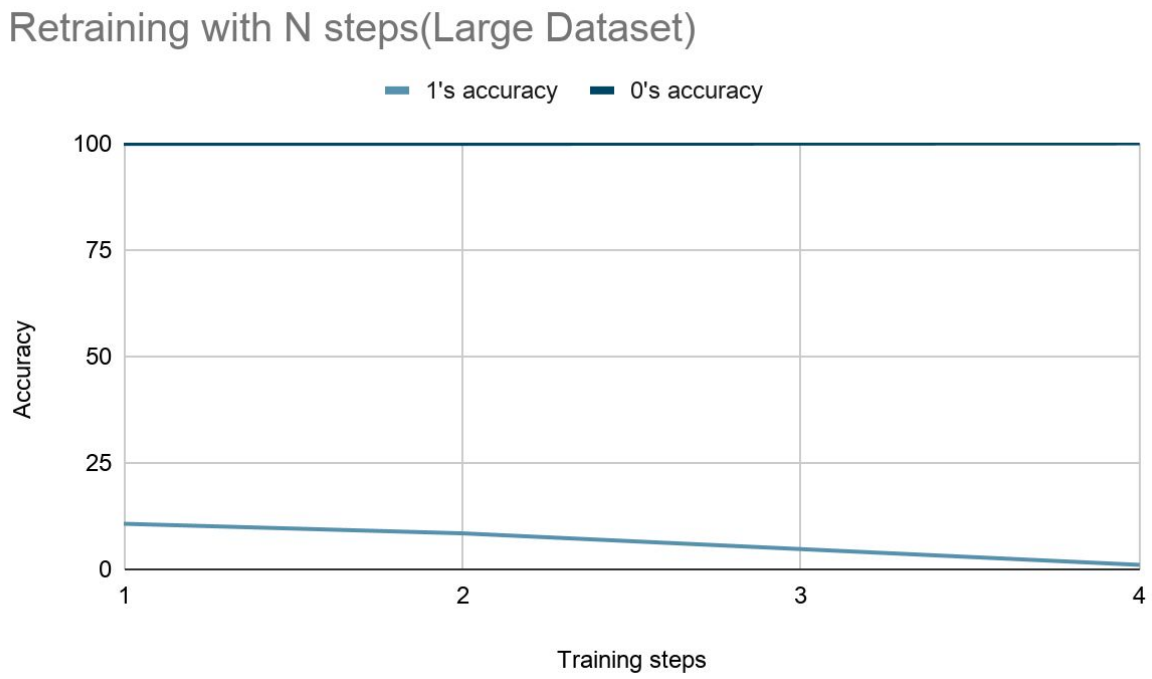
Now we will see the results while only changing the batch size. We will use the same learning rate(**2.0**) and the same number of epochs(**24**) as before.





Retraining

*In retraining we accept values that have a distance of 0.02 or less to the real value. This increments by 0.02 every training step.



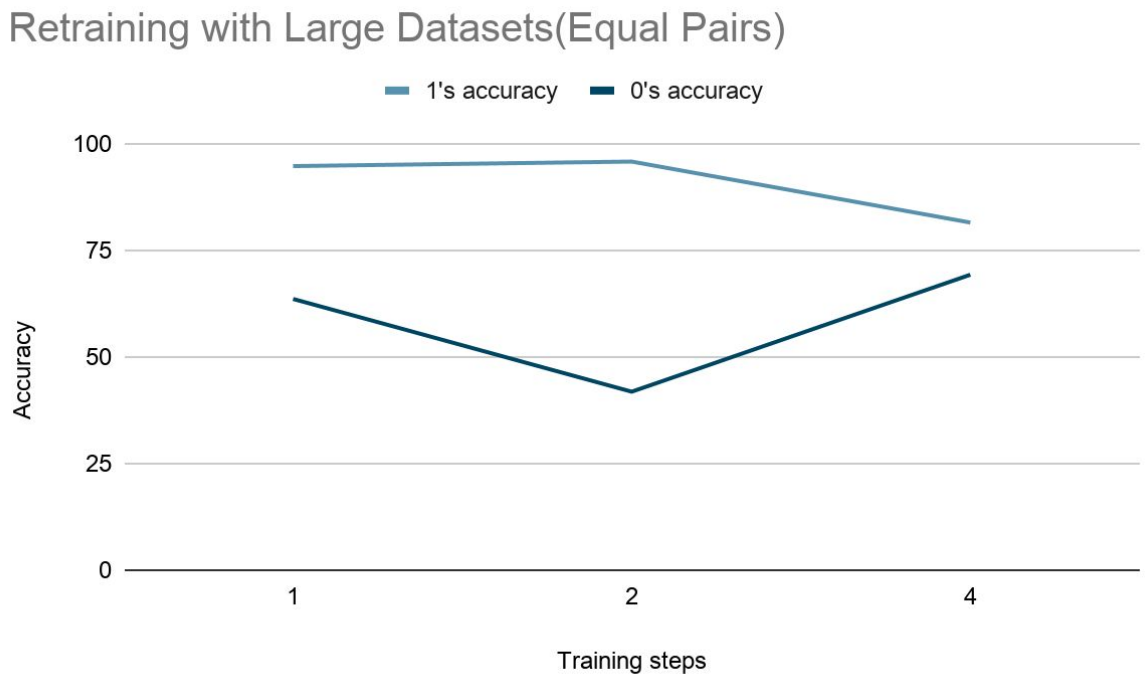
Here we can see that the bias that was mentioned above was reinforced and the 1's accuracy keeps dropping as the training steps become more.

We will test Retraining again in the next chapter with equal identical and non identical pairs.

Equal Pairs

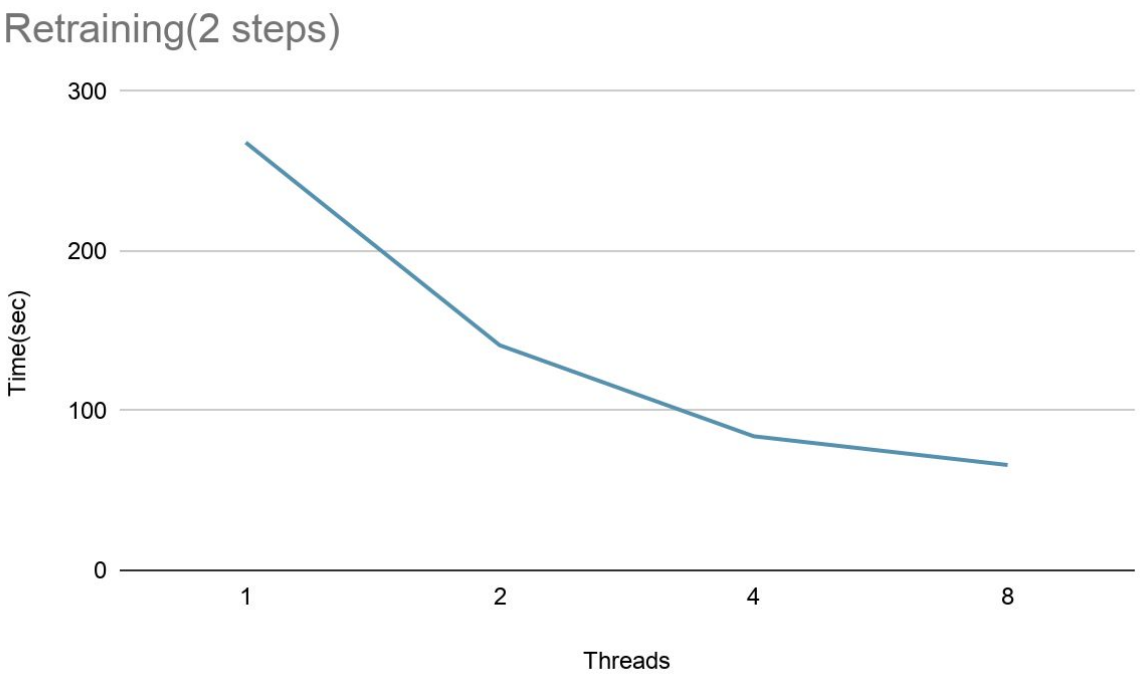
We have added a functionality with the flag -eq (1 for true, 0 for false) which makes sure that before training, the identical and non identical pairs that are given to the model are of equal length. The excess is discarded. With this flag, the accuracy of the 0's and the 1's are almost equivalent because the bias that was mentioned above no longer exists.

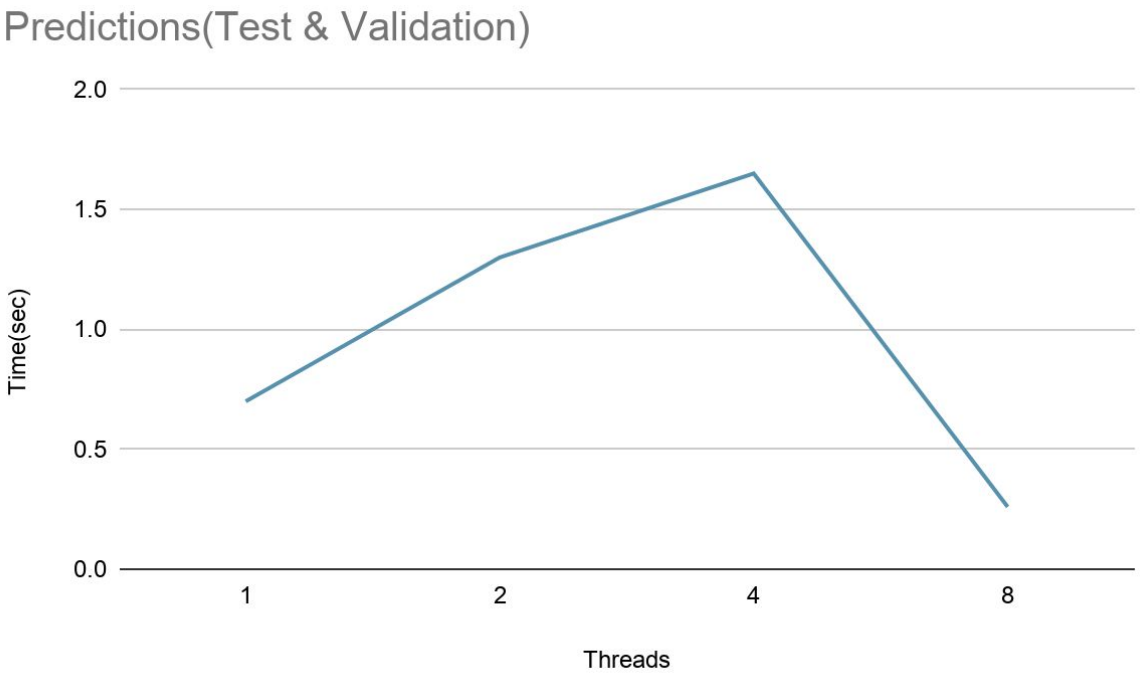
Let's see how this would differ from the above graph:



The accuracy this time is much more balanced and with the right tweaks, this model has tested for close to 90% accuracy for both values.

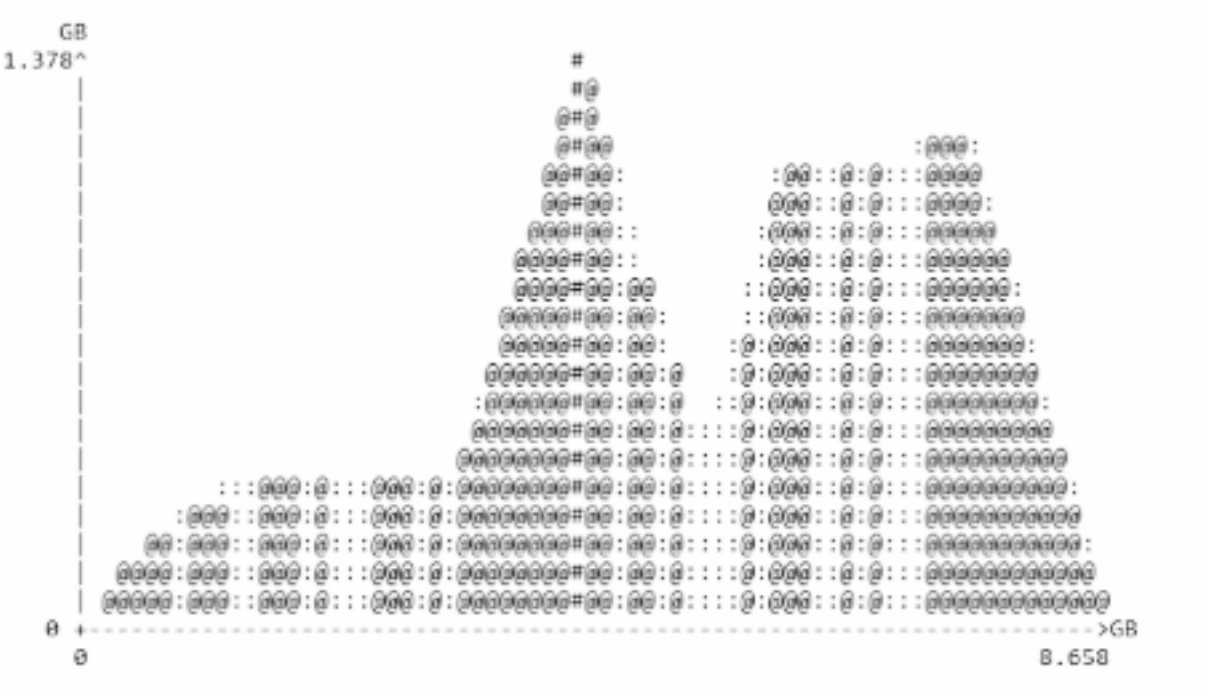
Multi-Threading





Memory Consumption

In this instance the program was run with the **medium** dataset and a dictionary size of **100**.



Note:
With the **large** dataset and a dictionary size of **1000**, the program does not exceed **3GB** of ram at any given moment.

Final Thoughts

While the training seems to be working fine, the proportion of identical to non identical pairs seems to cause a bias that seems to be resolved after a large number of epochs and with the right batch size. When using “equal pairs” it is clear that the bias does indeed exist since with much less data (fewer pairs), we manage to get good results with fewer epochs and a faster runtime (or more epochs and the same runtime since the Training Dataset would be smaller).