Due: 5 Nov 2020

We used Python (`numpy`, `pandas`, and `matplotlib`) and Jupyter notebooks to process the data throughout. In Tyler's code, all the numerical algorithms can be found in the file `methods.py`. Tyler's code can be found at this link (click here) in the code folder.

Kusals' code is uploaded to https://github.com/thisiskusal/AMSC808NProject2Code; the README details how his code is organized.

1. **Dataset 1: movie rankings. Task: NMF.** Select a complete submatrix of the movie ranking matrix. Choose a reasonable number of clusters k and use the k-means algorithm to cluster the users. Try to interpret the result. For the same k, compute the NMF $A \approx WH$ where W has k columns using

- Projected gradient descent
- Lee and Seung's scheme
- Start with projected gradient descent, then continue with Lee-Seung.

Plot the Frobenius norm squared vs iteration number for each solver. Which one do you find to be the most efficient?

Solution. To select a complete submatrix, we first selected "good" columns as columns with at least one-fifth of their entries filled. Out of this, we got a submatrix by selecting the rows which are full in the "good" columns. This left us with an 8×16 data matrix.

We coded the k-means algorithm using the method described in class and searched for 3 clusters in the data. However, since the means were initialized randomly, the exact nature of the clustering changes every time it is run (even at high tolerances). Since k-means is known to be highly susceptible to different initializations, this behavior makes sense. That fact in conjunction with not having other information about the data points makes it difficult to interpret these clusters.

Continuing with $k = 3$, we ran projected gradient descent, the Lee-Seung scheme, and a blend of the two. In terms of parameters, we fixed the maximum number of iterations at 1000 and used a step size of $\alpha = 0.01$ for projected gradient descent. For the blended run, we initially used a 20/80 split: projected gradient descent ran for the first 20% of the total iterations, and then Lee-Seung finished the job. Results from a preliminary run can be found in Figure 1. Note that the blended method significantly outperforms both projected gradient descent and Lee-Seung.

To conclude this problem we experimented with finding the optimal split for the blended method. Figure 2 shows the Frobenius norm at the final iteration for a blended run with the indicated split averaged over 10 runs at that split. From it, we see that over 1000 iterations, the best performance is achieved by about a 20/80 split.

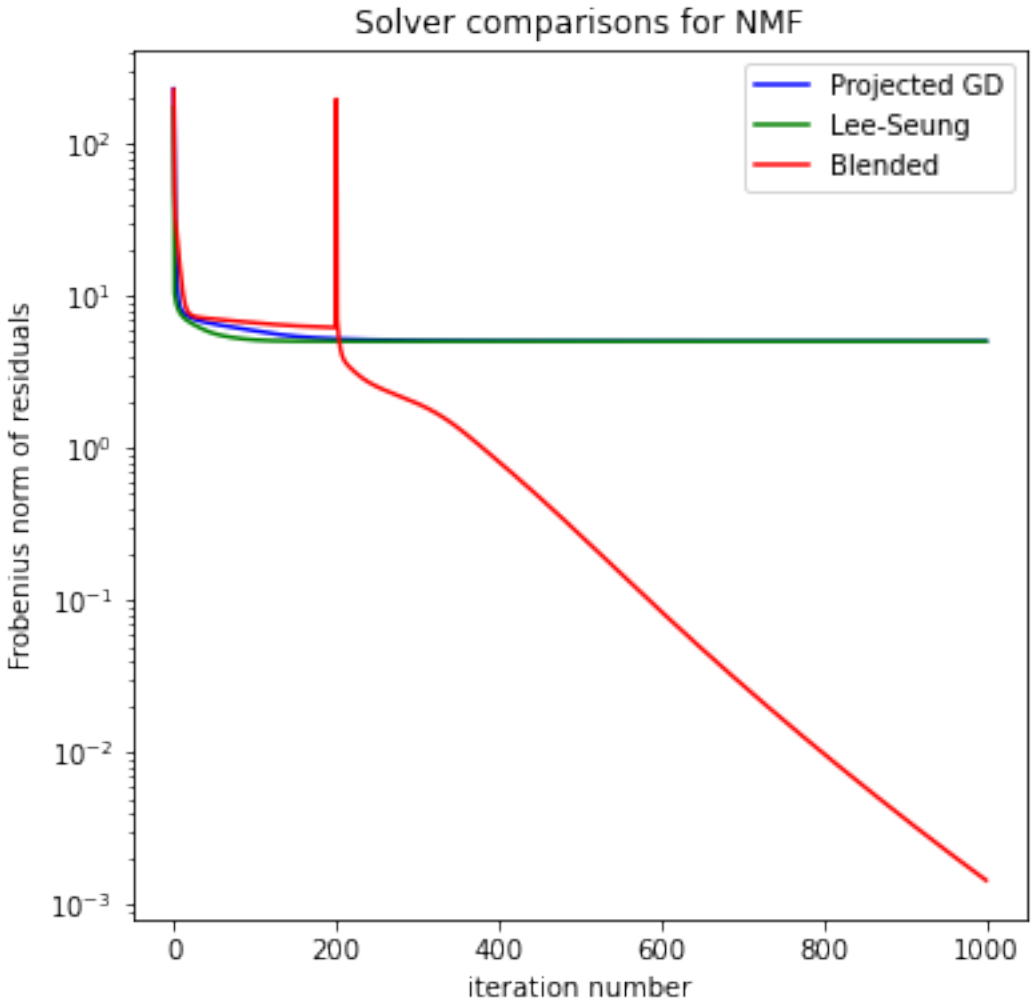


Figure 1: Various nonnegative matrix factorizations and their performance.

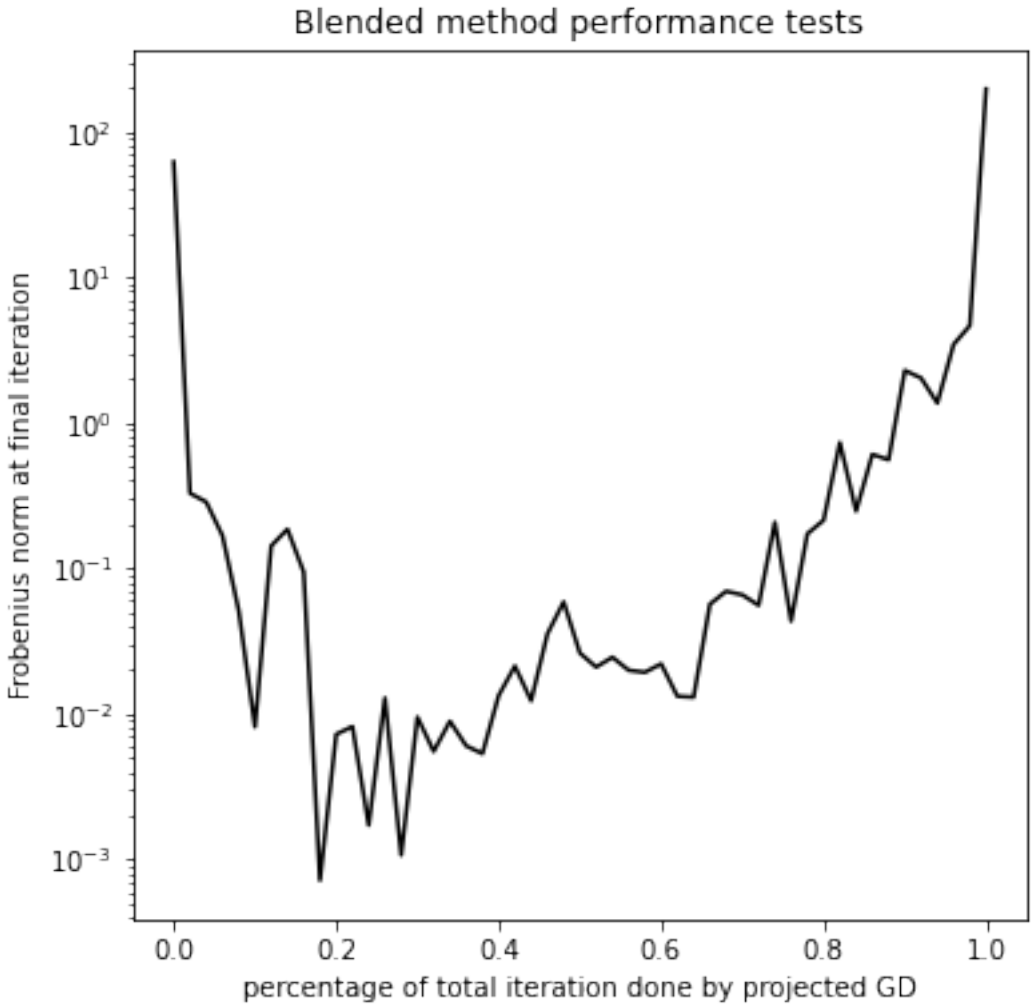


Figure 2: Final iteration Frobenius norms for various splits.

2. **Dataset 1: movie rankings. Task: matrix completion.**

Solution. This analysis primarily looks at how the regularization parameter λ effects matrix completion. As such, we first state the hyperparameters used in our experiments. For Low-Rank Factorization (LRF), the method takes as hyperparameters both λ and k, the latter being the permitted rank of the matrix decomposition. To reduce this space to just being dependent on λ, we fix k at values 2, 4, 6, ..., 20 (the original matrix has 20 columns). For the Nuclear-Norm Trick Factorization (NNT), the method solely takes as hyperparameter λ, so we did not have this issues. Asides from this, we used a constant convergence tolerance of 10^{-12} - if at some iteration our loss changed by less than this amount, then we declare the method converged.

The metrics we used to evaluate these methods were total time till convergences, number of epochs, method loss (which were defined different for LRF and NNT), and Frobenius distance to A ($\|A - M\|_F$). To estimate the values of these metrics with respect to each matrix completion method, we ran each algorithm 30 times and took the mean value over these runs.

In selecting a range of λ's to test over, we first found that large λ values would cause the regularization terms to crush the Frobenius distance term, leading to extremely high Frobenius distance to A. As such, we limited our λ testing to 0.5, 1, 1.5, ... 6.0.

With this setup, we find various results, with most of them being quite interpretable. Speed of convergence (in both time and number of epochs) would increase with λ, as the iterations would not need to find as close a solution to A. Method loss increased with λ as it is directly by it; more interestingly, the Frobenius score would also increase, as again there was not as much weight on weight on $\|A - M\|_F$ with higher λ.

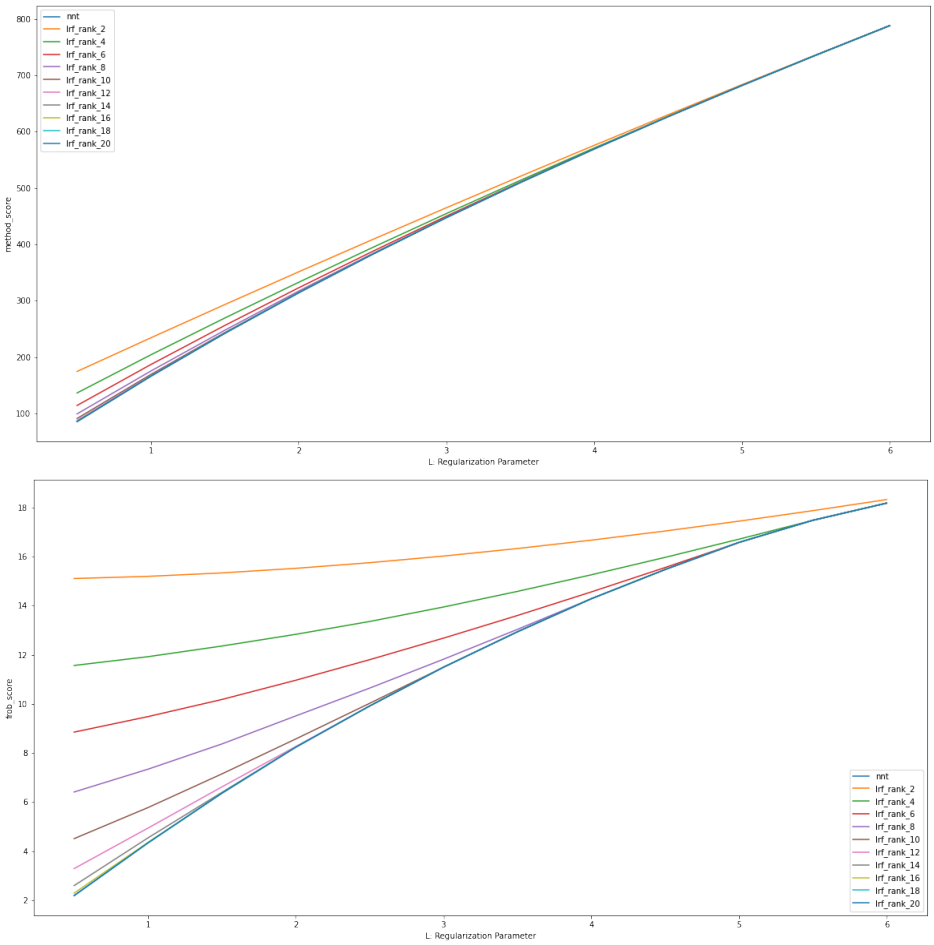


Figure 3: Method loss (Top) and non-regularized method loss (Bottom) for LRF and NNT. Higher is worse for both graphs.

Comparing NNT and LRF, both come with their respective trade-offs. In general, the computation for NNT is far more vectorizable, thus each epoch took significantly less time than an LRF epoch. This difference of epoch time would only increase with the matrix size, since LRF requires $2n$ unique least squares solution at each iteration. NNT also performed better with regards to Frobenius distance to A, which was again expected given that there are no hard constraints on the rank of the NNT solution. Finally, the total number of epochs for NNT was almost double that of LRF; this is probably due to NNT having a much larger parameter set (LRFs' parameter size scales with k), thus it had more values to learn.

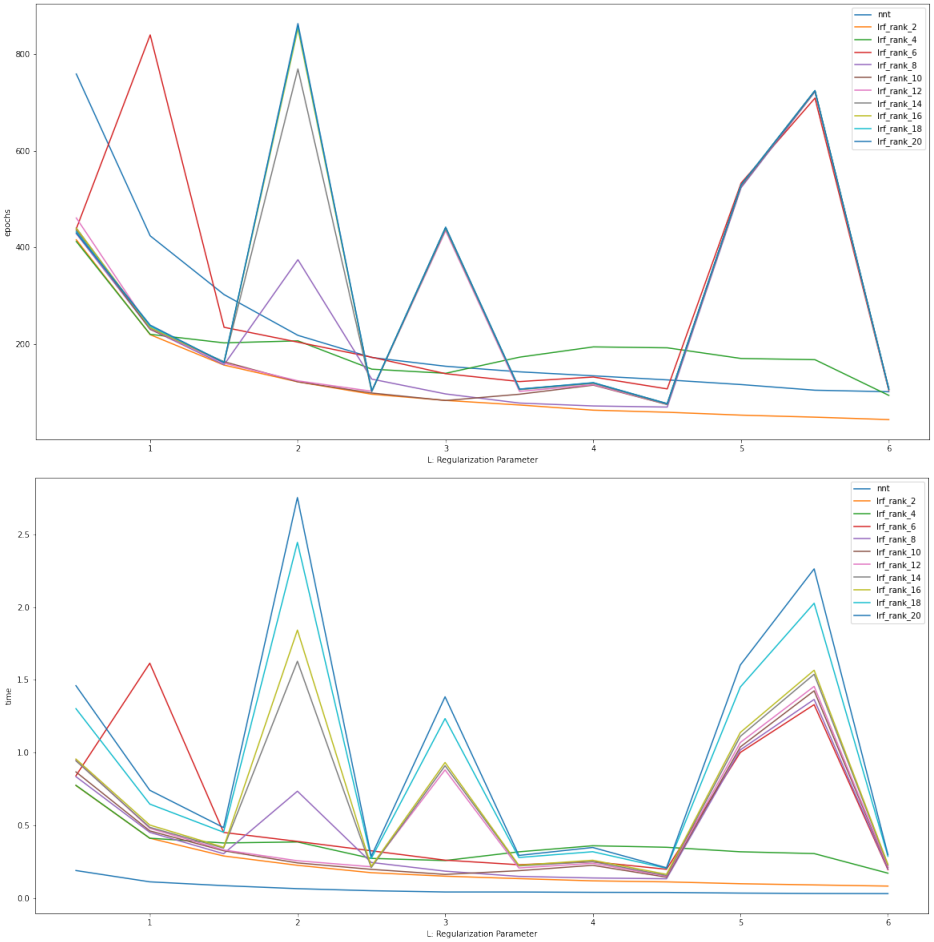


Figure 4: Number of epochs (Top) and time in seconds (Bottom) for LRF and NNT. Higher is worse for both graphs.

3. **Dataset 2: text documents. Task: CUR factorization.** Program the CUR algorithm as described in Mahoney and Drineas (2009). Run it for k from 2 to 10 and $c = r = ak$ for a from 1 to 8. Since it is a randomized algorithm, perform 100 runs for each combination of k and a. Plot the mean ratio $\|M - CUR\|_F / \|M - M_k\|_F$ versus k for each a. Also plot $\|M - M_k\|_F$ versus k for each a. Pick a reasonable combination of k and a.

Solution. We programmed the CUR algorithm as described in Mahoney and Drineas with the vectorization detailed in class. Following the instructions, we set up and ran a numerical experiment with 100 runs for each combination of k and a, computing average time for the individual combinations and the requested Frobenius norm metrics as we went.

Our error results can be found in Figure 4, where the requested ratios are plotted. From these, it can be seen that the ratio metric $\|M - CUR\|_F / \|M - M_k\|_F$ is monotonic in both k and a, with the optimal performance given by $(k, a) = (10, 8)$ as this is when the ratio is closest to 1. Due to the error bounds given in Mahoney and Drineas (2009), this ratio will never be perfectly 1, but it gets quite close with these optimal k and a values.

We know from the analysis that

$$\|M - CUR\|_F \leq (2 + \epsilon)\|M - M_k\|_F$$

and hence from this graph we can roughly bound $\epsilon > 0.2$.

Interestingly, the higher the a value, the faster on average the computation (Figure 5). This may be due to the fact that enforcing a very low rank factorization leads to numerical oddities.

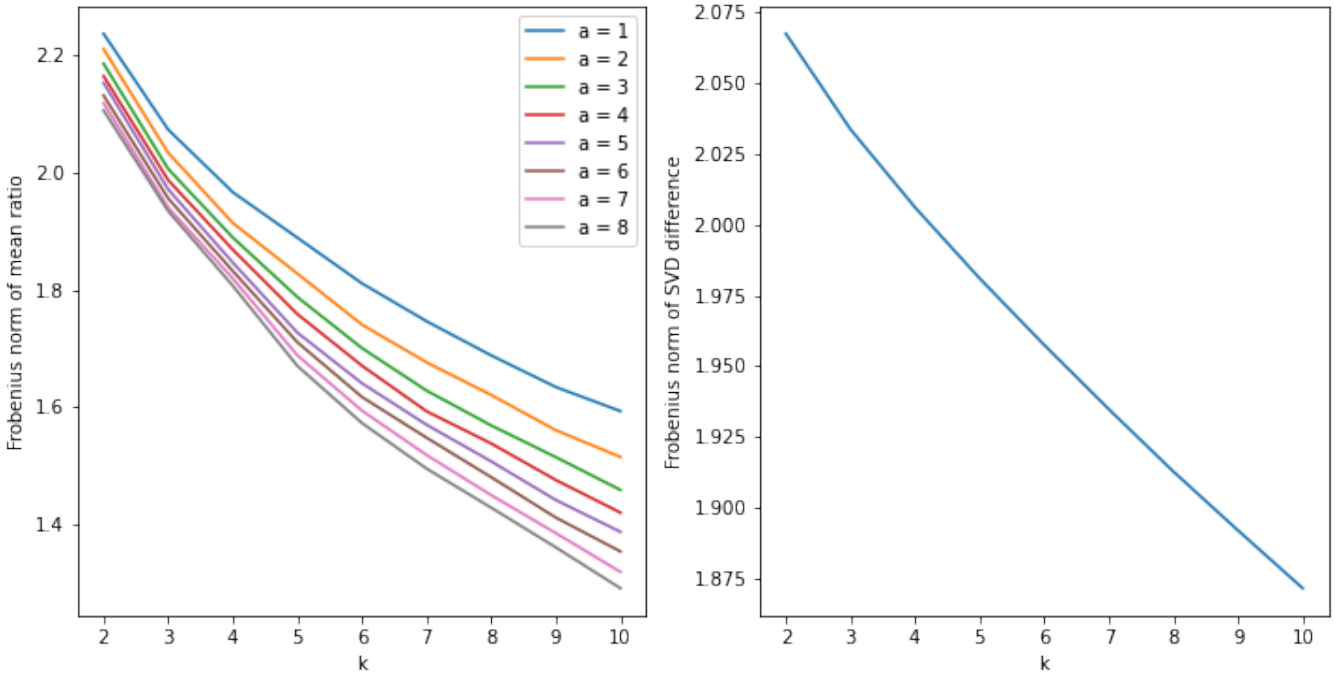


Figure 5: Depiction of the CUR algorithm's performance as compared to SVD. Left: Plot of the given ratio metric $\|M - CUR\|_F / \|M - M_k\|_F$ versus k for each a. Right: Error from the truncated SVD, $\|M - M_k\|_F$ versus k. Lower is better for both of these plots.

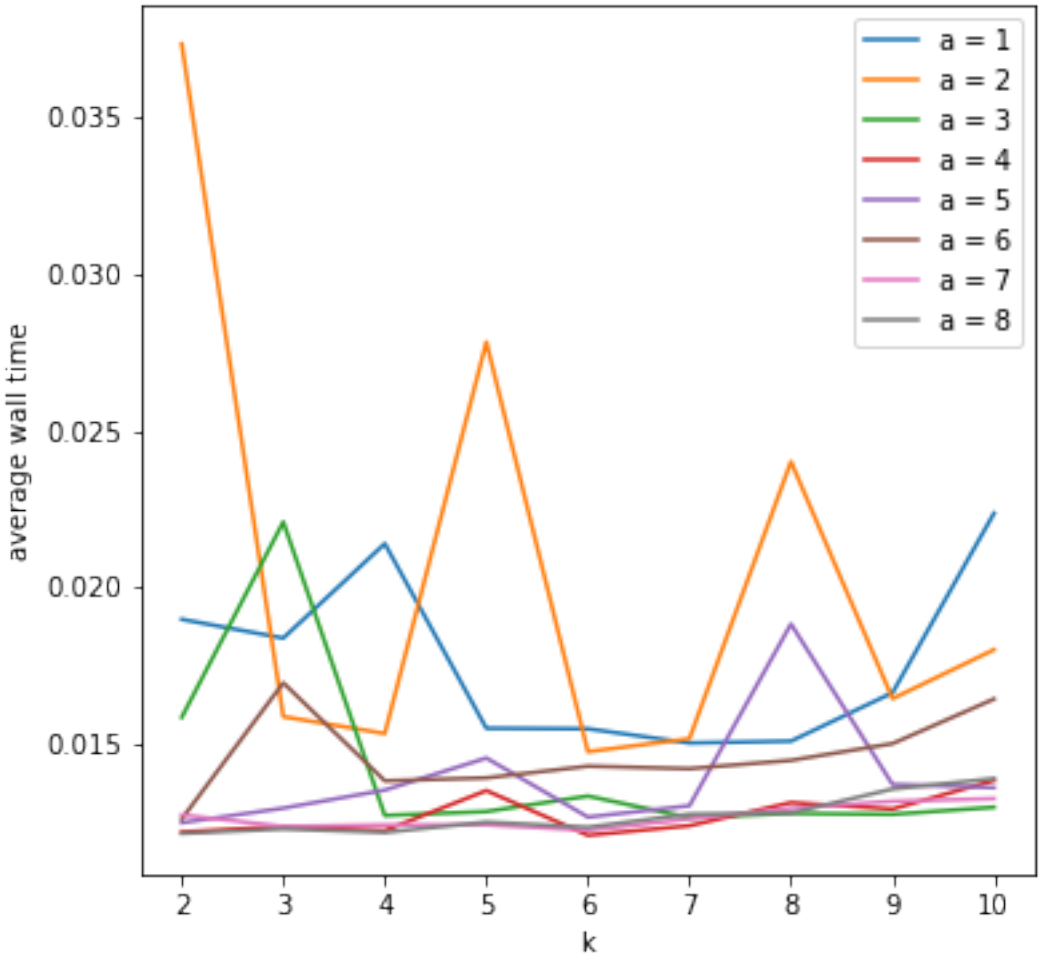


Figure 6: Average wall times for the various CUR parameters recorded from the same simulation as the previous figure.

4. **Dataset 2: text documents. Task: text categorization.**

Solution. To explore this task, we first drafted various column scoring metrics. Some were creating using only M as a parameter and others used both M and y for column selection. For each selection method, we then selected the top k scores and ran a PCA reduction onto 2 dimensions off those columns. To evaluate the strength of each 2 dimensional projection, we quantified a 'separability' metric by the raw accuracy of a fully trained Linear SVM. Obviously, there are other things we could have evaluated, such as what the margin of the Linear SVM was or how much total error (not just accuracy) the SVM had; for now, lets just focus on accuracy.

For more details on experimental setup, we tested 15 total column-scoring methods (8 of which used CUR factorization) and selected 2 to 299 columns from M before PCA. PCA was done after mean-normalizing the resulting data vector. The SVM used a linear kernal and did not allow hard margins; it was allowed to train for a maximum of 5000 iterations, but no run took that long to converge.

Though it would be neat to understand exactly what the principal components said about the texts, unfortunately PCA results aren't too interpretable. As a proxy, we instead just took all the words selected by the various column ranking methods and found the most common words between different runs.

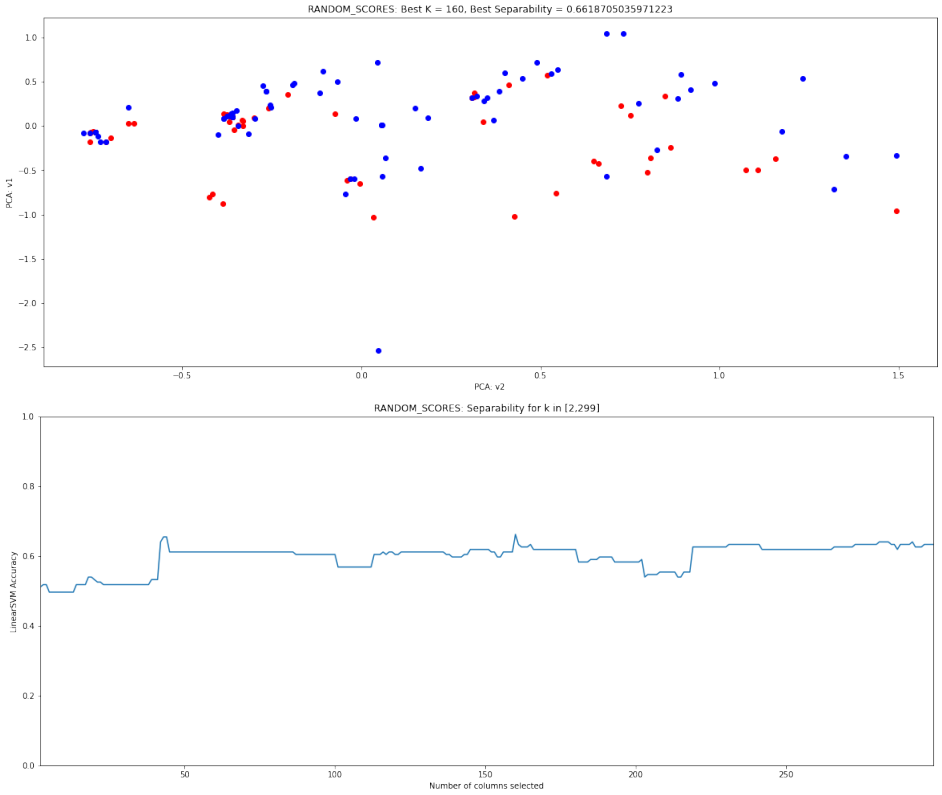


Figure 7: K-dependent (Top) and best (Bottom) performance of RANDOM_SCORES selection.

As a testing baseline, we tested against a random column selection (RANDOM_SCORES). In general, RANDOM_SCORES provided an accuracy of around 0.6 (regardless of k). This makes sense as a truly random binary classifier would provide 0.50 accuracy by default, but by chance RANDOM_SCORES could choose *some* useful words that.

In general, the column selection methods that took both M and y as parameters performed slightly better than those that only used M. This was of course expected, given that this first column selection process just has more information. Surprisingly though, the difference between these two classes of methods was extremely small - the first group performed at about 0.978 accuracy, while the second group ranged from 0.935 to 0.985 accuracy. The best performing metric was LEVERAGE_SCORES_10:

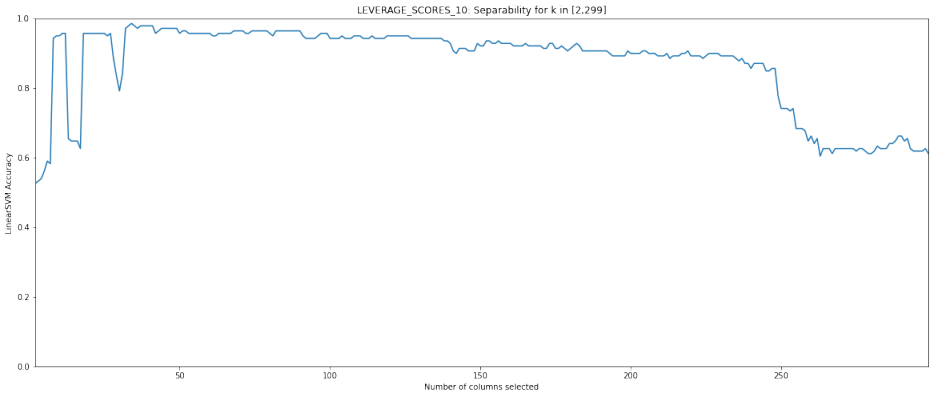


Figure 8: K-dependent performance of LEVERAGE_SCORES_10

Here we see that the method performed best when using between 25 and 250 columns. The best performance was generated via exactly 34 columns, to which we see the projected data below:

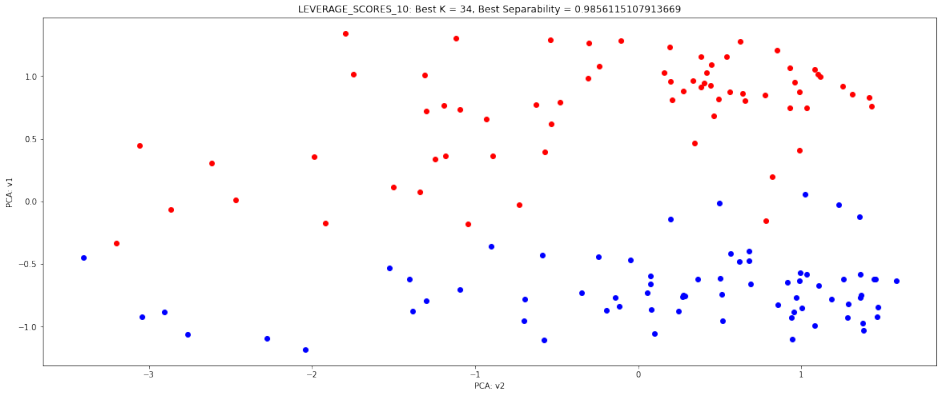
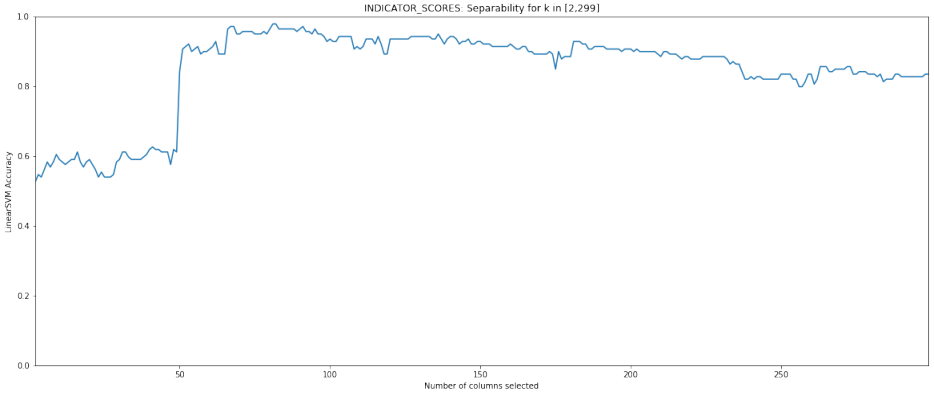


Figure 9: Best performance of LEVERAGE_SCORES_10

Interestingly enough, the INDICATOR_SCORES metric performed almost as well as these other metrics. This metric counts the number of documents that each word comes up in - aka the percentage of total documents where the word was present. The interesting part of this is that the computation involved in this metric is extremely low in comparison to the other well-performing metrics. Whereas the leverage scores requires SVD computations, and the stratified scores took partitioned SVD computations, the indicator score merely counts the number of non-zero entries in each matrix column. The results for this metric are shown below:



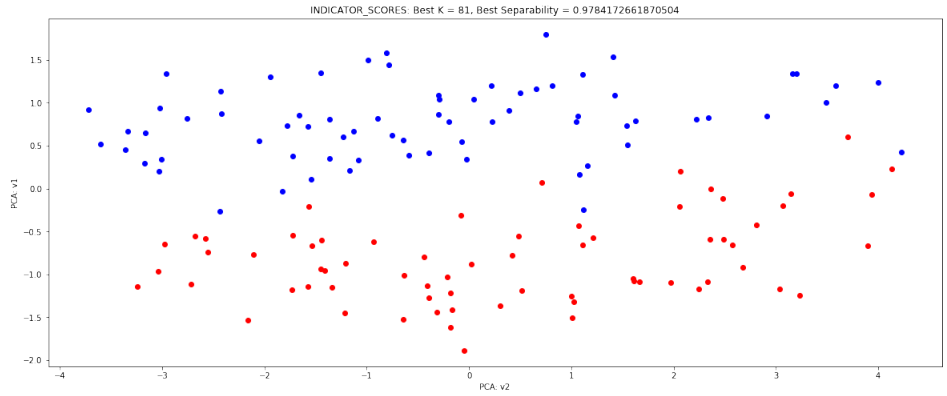


Figure 10: K-dependent (Top) and best (Bottom) performance of INDICATOR_SCORES selection.

An interesting stratified metric (one that used both M and y) was STRATIFY_LEVERAGE_SCORES. Basically, we created the leverage scores of only looking at the Indiana documents and only looking at the Florida documents, absolute value of these scores. The idea being that words that come up in both partitions would cancel each others leverage scores a lot, and words that mostly come up in a single partition (aka the word 'palm' mostly comes up in 'Florida'-labelled documents) would have a large difference. The results of this metric were fairly good, but it wasn't able to out-perform the simple INDICATOR_SCORES metric:

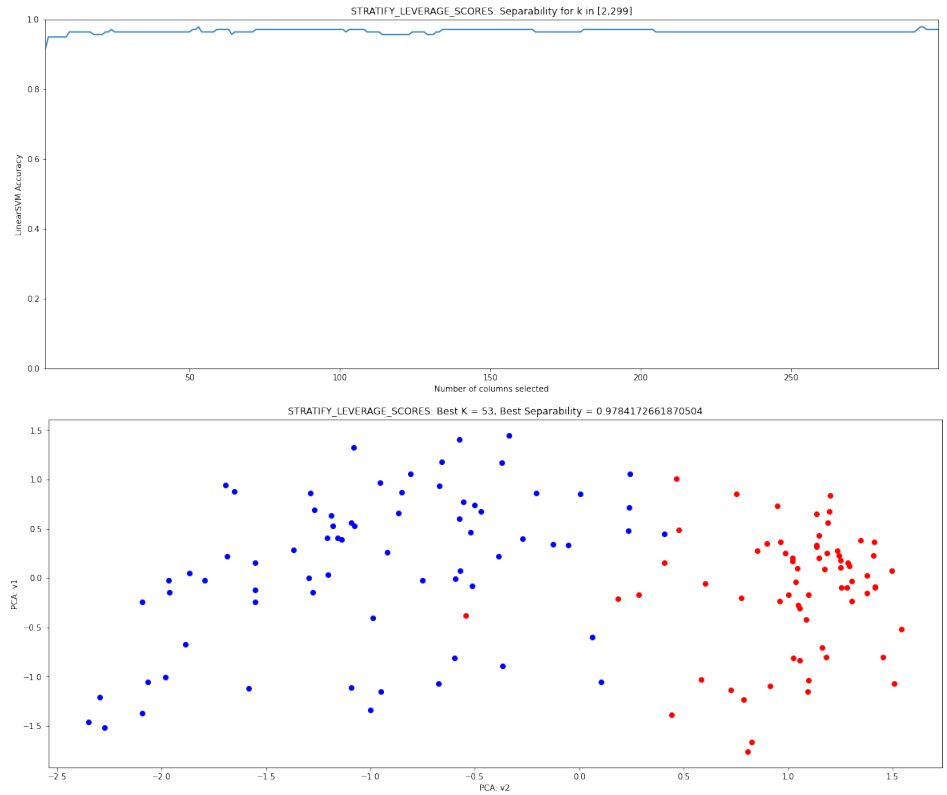


Figure 11: K-dependent (Top) and best (Bottom) performance of INDICATOR_SCORES selection.

Interestingly, when looking at the best performing k (number of columns) choice for each scoring metric, the only words that were used in all of these selections were 'indiana' and 'evansville'. Otherwise, other common words were 'florida', 'from', 'contact', 'and', 'all', 'you', for', 'will', and 'from'. We see that asides for three of these, these words offer no *intuitive* pointer to the document class. More study would be needed to understand this phenomenon.

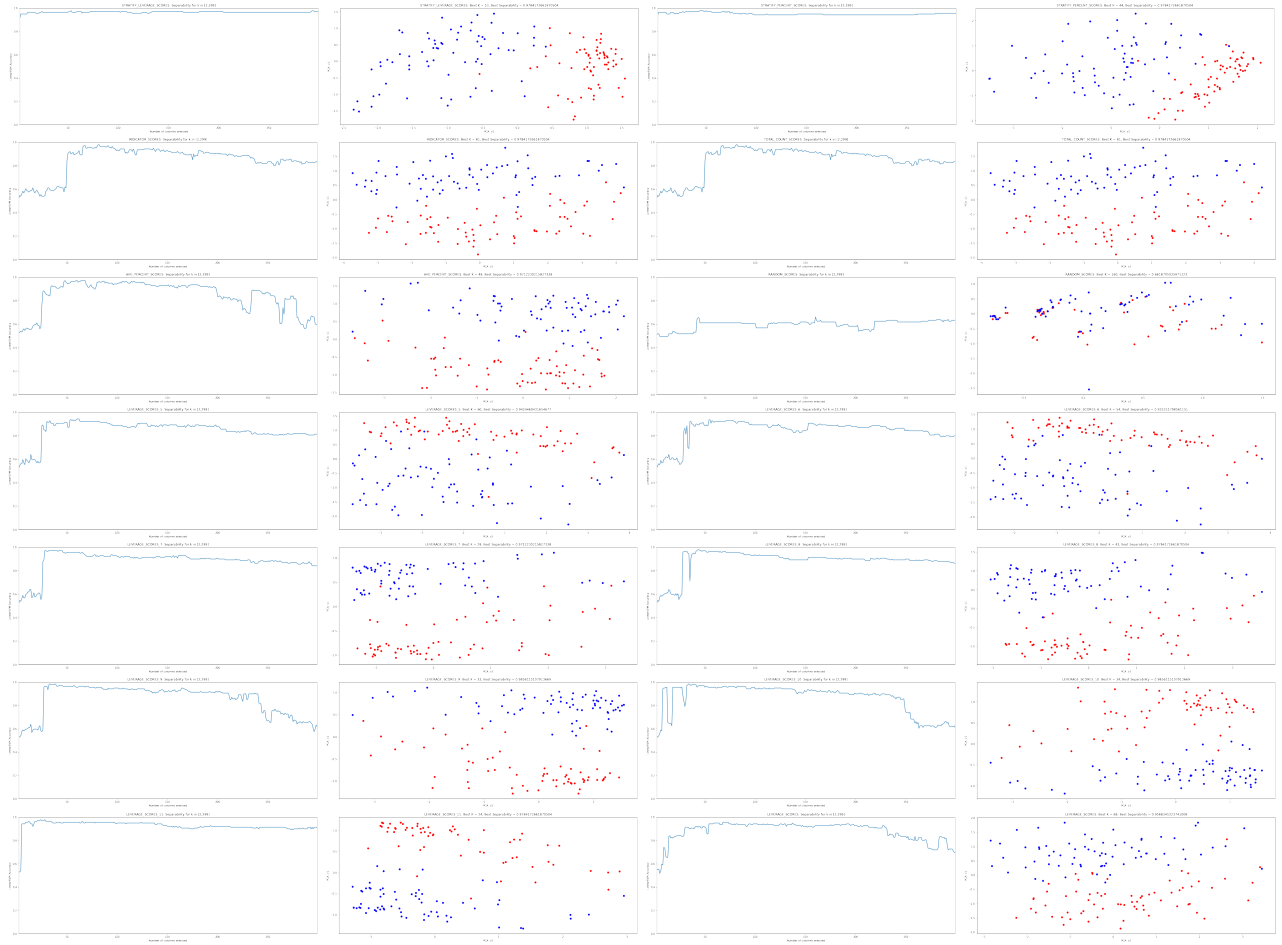


Figure 12: K-dependent and best performance graphs of tested column-selection criteria methods. Descriptions for each can be found in Kusals' code.