MITx 6.419x Data Analysis
Statistical Modeling and Computation in Applications

Analysis 2
Genomics and High Dimensional Data

Jonathan Chang (JonathanChang6d41)

March 30, 2021

Contents

Note that the code presented are curated snippets and not the whole thing. They're not essential to understanding the analysis, but they do provide a bit of context. Skip them if you wish.

Problem 2.
Larger Unlabeled Subset

We are told that there are three types of cells in the brain: *excitatory neurons*, *inhibitory neurons*, and *non-neuronal*. However, our data contains 45768 genes per each of 2169 cells; that is, the data set is (2169×45768), too large to directly gain any intuition about it. Perhaps we wish to present our knowledge in a presentation, or convey a research idea to secure a grant. Or perhaps we want to convince ourselves of this fact. Gene names and the raw numbers might be useful to domain experts, but not anyone else. In contrast, everyone understands a good visualization. Since genomics data are densely clustered with some distant outliers, we apply the normalization $\log_2(X+1)$ to spread out small data points, yet tame large ones.

Part 1.
Visualization

It may not be a good idea to search parameters that produce a visualization to prove some predetermined idea, in the same vein as p-hacking. But it may not be as harmful if we are already certain the fact is true. In this case, we use PCA to create embeddings of a lower dimension in order to improve the results of three dimensionality reduction methods: PCA, MDS, and T-SNE.

It was easy to tell from initial trials that the visual clusters appear to be of different variance, and often in long shapes in one direction. Therefore, k-Means would be a poor clustering algorithm since it only measures Euclidean distance between clusters, so it would not be likely to group long shapes or different sized groups. DBSCAN is appropriate for these considerations, but it would be hard to figure out an appropriate ε. The decision was then made to use the Gaussian mixture model (GMM) for its ability to capture differing variances.

In order to achieve *some* consistency, the clusters were sorted by the number of points they contain, such that red would always represent the smallest cluster, then green, with blue the largest.

```
pcs = [10, 50, 100, 250, 500]
algorithms = [ **PCA** , MDS, TSNE]
clust_colors = ['coral', 'yellowgreen',
  'skyblue']
# Triangle, plus, star markers
markers = [(3, 0, 0), '+', (5, 2, 45)]

for pc in pcs:
  args = {'n_components': pc,
    'random_state': 0}
  X_pc = PCA(**args).fit_transform(X)
  args[n_components] = 2
  for alg in algorithms:
    X_2d = alg(**args).fit_transform(X)
    labels = cluster_labels(X_2d, 3)
    plot_graphs(X_2d, labels)

def cluster_labels(X, n_clusters):
  # Label clusters with GMM.
  args = {n_components: n_clusters,
    random_state: 0}
  gmm = GaussianMixture(**args)
  return gmm.fit_predict(X)

def plot_graphs(X, labels):
  X_clust, colors = [], []
  for i in range(3):
    X_clust.append(X[labels == i, :])
    color = clust_colors[i]
    colors.append(color * len(X_clust[i]))
    # Plot/save scatter plot with pyplot
    visualize(X_clust, colors, markers)
```

Listing 1: Various PCs, colored with GMM.

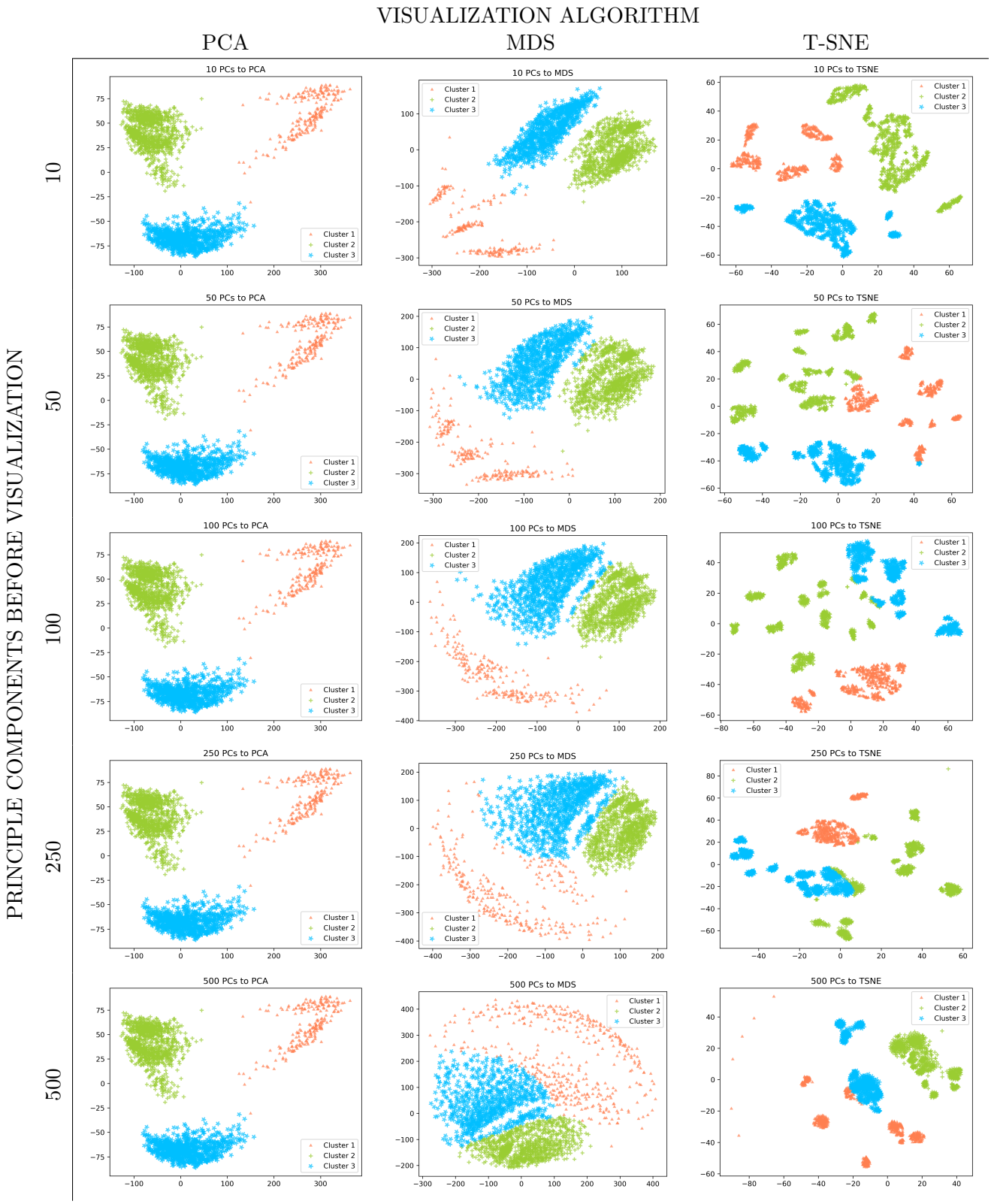


Figure 2.1: Clusters are colored with GMM over 50 PCs to PCA.

Since PCA is relatively stable with good separation between the three clusters, the GMM clustering on that graph, with PC = 50, will be used for further examination. In retrospect, if the end goal was logistic regression, it would have been better to use an intermediate set with more principle components to capture more variance. But I think this method is visually illustrative, and does decently well.

I was already able to tell using prior graphs that the clusters, after three, are visually indistinct with MDS, and variably separated with T-SNE. By looking at the clusters separately, I posited that the separation would improve, so I divide the data set into three representing each cluster.

```
def split_data_into_clusters(X, labels):
  # Use GMM labels of 3 clusters to
  # split data.
  return [
    X[labels == 0, :],
    X[labels == 1, :],
    X[labels == 2, :]
  ]
```

Listing 2: Data set is split into three clusters.

Again, I use GMM on each cluster, and for each visualization method in 2D. To find the optimal number of clusters, I employ the Bayesian information criterion (BIC), which is a somewhat more sophisticated elbow method in that it penalizes increasing clusters, k, against the score. For N samples and D features,

$$\text{BIC}^{1} = k\frac{D(D+1)}{2}\ln(N) - 2N\ln(\hat{L}) \qquad (2.1)$$

where $\hat{L}$ is the average likehood of points being assigned to their clusters. Intuitively, as the likelihood improves, BIC decreases. But as k increases, BIC increases. So we are looking for the *lowest* BIC score, which is plotted for good measure.

```
def plot_optimal_clusters(X, color):
  scores = search_over_clusters(X)
  # Plot scores with pyplot.
  plot_scores(scores['bic'],
    scores['components'], color)
  return get_optimal_n_clusters(scores)

def search_over_clusters(X):
  # Range of clusters to search.
  min_clust, max_clust = (3, 18)
  clusters = range(min_clust, max_clust)
  scores, bic = [], []
  for n in clusters:
    args = {'n_components': n,
      'random_state': 33}
    gmm = GaussianMixture(**args).fit(X)
    scores.append(gmm.score(X))
    bic.append(gmm.bic(X))
    return {
      'scores': scores,
      'bic': bic,
      'components': components}

def get_optimal_n_clusters(scores):
  min_index = np.argmin(scores['bic'])
  components = scores['components']
  return components[min_index]
```

Listing 3: Find optimal number of sub-clusters.

With the optimal number of clusters for each visualization method, we aim to find the one with the most visual separation, and the highest resolution. For this, T-SNE should be most suitable. The relevant labels and their corresponding data points are saved for the upcoming parts.

```
all_labels = []
for n, X in enumerate(X_clust):
  for alg in algorithms:
    X_pc = PCA(**args).fit_transform(X)
    X_2d = reduce_X_to_2d(X_pc, alg)
    k = plot_optimal_clusters(X_2d, color)
    labels = cluster_labels(X_2d, k)
    visualize(X_2d, colors, markers[n])
    if alg_name == 'TSNE':
      all_labels.append(labels)
```

Listing 4: Save sub-cluster labels per cluster.

[1] scikit-learn's implementation of BIC from the GMM class.

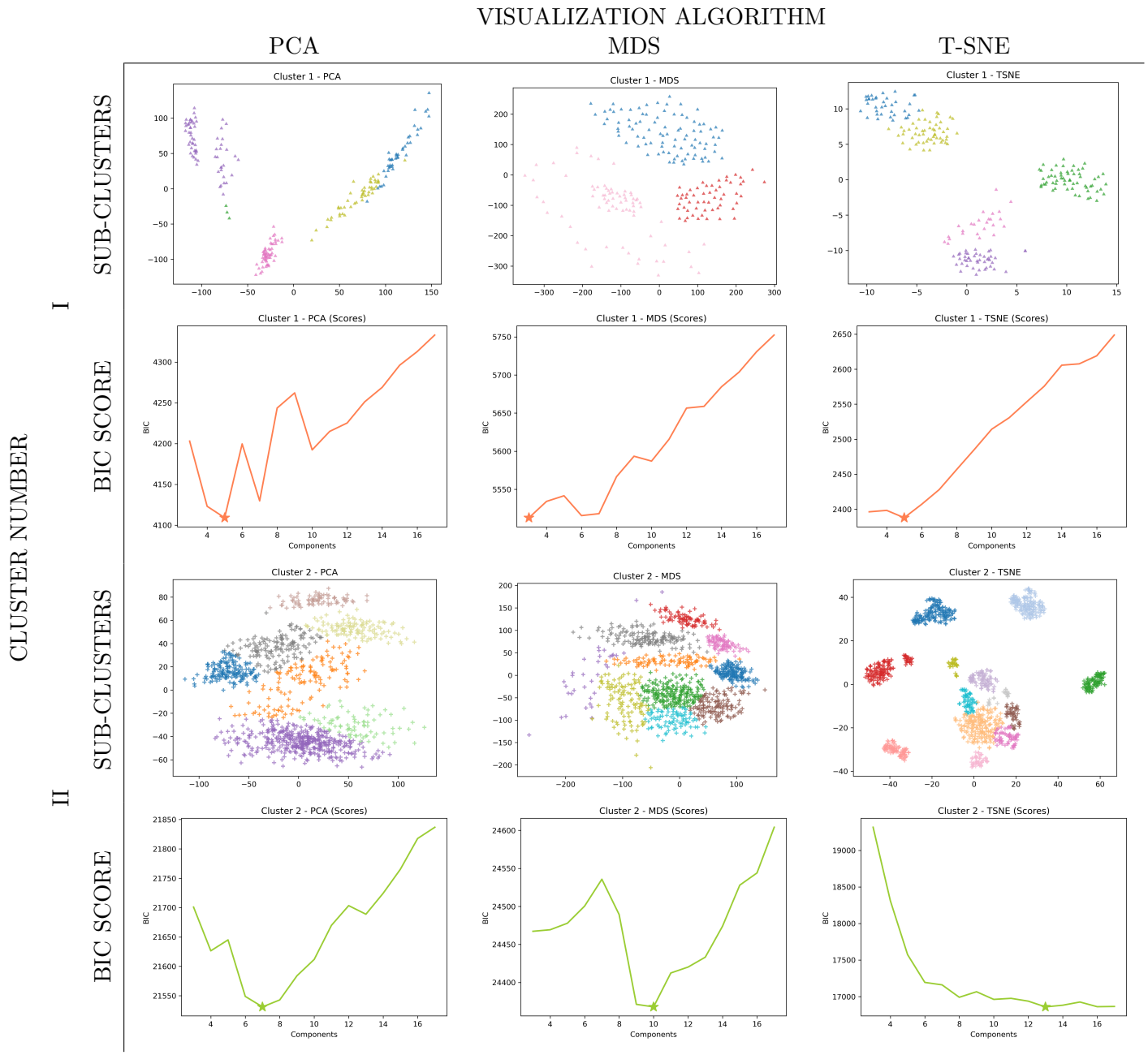


Figure 2.2: Sub-clusters within clusters 1 and 2. T-SNE is most illustrative.

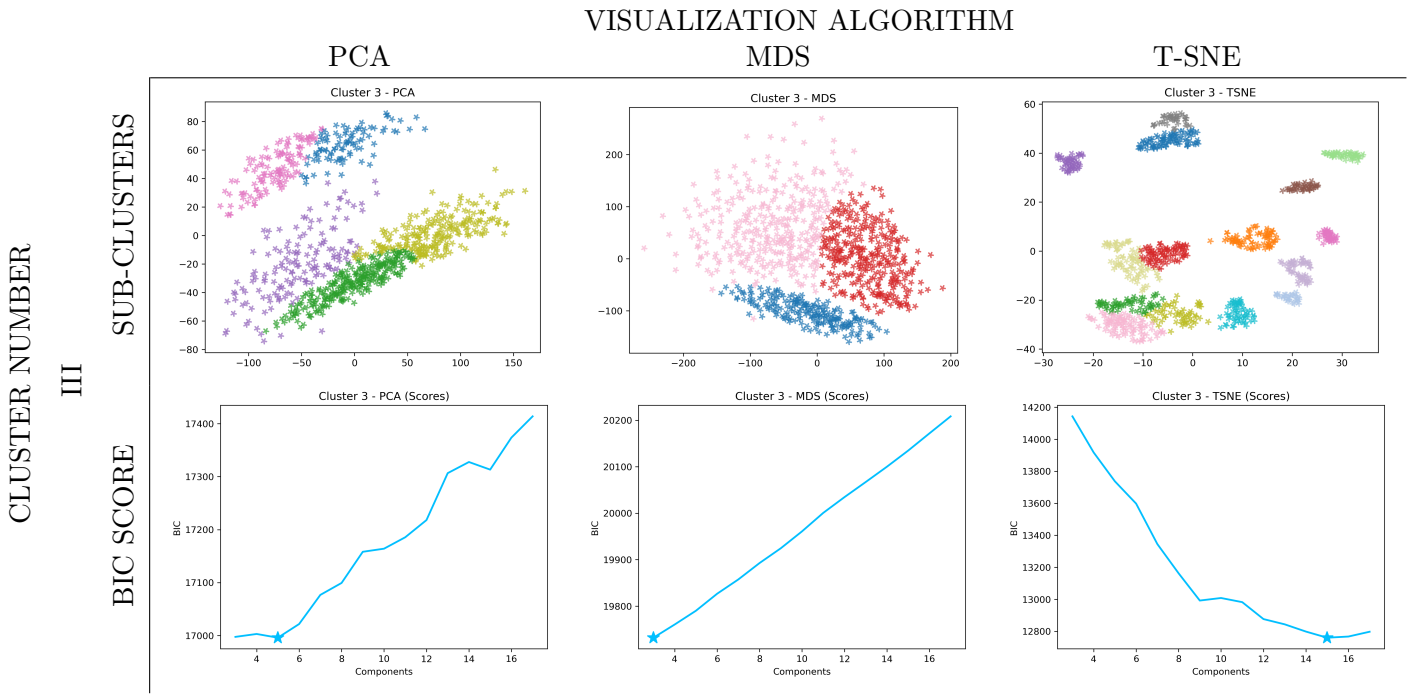


Figure 2.3: Sub-clusters within cluster 3. T-SNE is most illustrative (continued).

We confirm that the T-SNE plots show the most visually distinct clusters, besides a few choices that might seem subjective. Using the lowest BIC scores, we find that the three main clusters have 5, 13, and 15 sub-clusters respectively for a total of **33**.

A hierarchical dendrogram could serve as a sanity check. In order to create one, a different clustering algorithm is used, the Agglomerative hierarchical method.

```
def agglomerative(X, linkage='ward'):
  args = {'linkage': linkage,
    'compute_distances': True}
  model = AgglomerativeClustering(**args)
  model = model.fit(X)
  children = float(model.children_)
  heights = float(model.distances_)
  counts = np.zeros(children.shape[0])
  linkage_matrix = np.column_stack([
    children, heights, counts])
  return linkage_matrix
```

Listing 5: Create linkage matrix with hierarchical clustering.

Note that Ward's method is the only one to account for cluster variance, so it's the most suitable for the same reason GMM was previously chosen.

We need to create a linkage matrix to pass into scipy's dendrogram function with each row corresponding to a split, as *(child 1, child 2, height, count)* (the count of samples in cluster is irrelevant for the plot).

```
ax = plt.figure().add_subplot()
create_dendrogram(X, ax)
# Formatting is skipped here...
plt.savefig(fname, dpi=300, format='png')

def create_dendrogram(X, ax):
  linkage_matrix = agglomerative(X)
  set_link_color_palette(cluster_colors)
  args = {'no_labels': True,
    color_threshold=3000, ax=ax}
  dn = dendrogram(linkage_matrix, 33,
    'lastp', **args)
```

Listing 6: Plot the dendrogram.

Keep in mind that the clustering algorithm is different from before, so the results differ.

But we would see that the proportion of sub-clusters within main clusters are relatively preserved, with one containing less than the other two. And the maximum height between levels demonstrate the three main clusters.

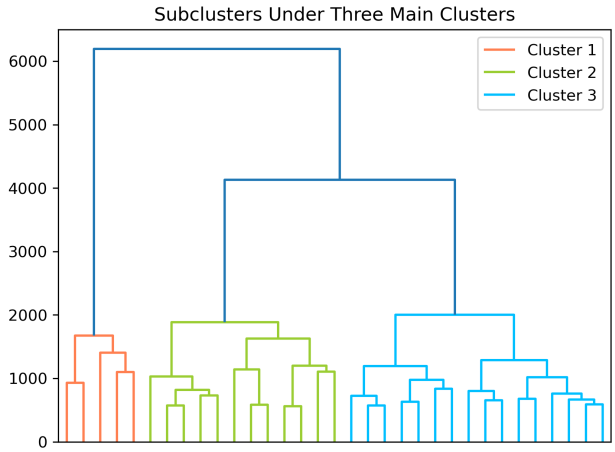


Figure 2.4: Dendrogram showing relative proportion of clusters.

Part 2.

Unsupervised Feature Selection

It was previously discussed that GMM clustering was due to the visually noticeable varying densities in the plots. The visualizations that maximized cluster separation were used, so it was PCA for the three main clusters, and T-SNE plots for sub-clusters. Using this method, **33** clusters were discovered. To be sure, we could apply the sub-cluster labels back to the original plots.

```
colors = convert_to_colors(subclust_labels)
clust_labels, _ = two_PCs(X, PCA, n=3)
clust_labels = sort_clustsize(clust_labels)
for alg in algorithms:
  _, X_2d = two_PCs(X, alg, n=3)
  clusts = three_clusts(X_2d, clust_labels)
  plot_sub_clusters(clusts, alg, colors)
```

Listing 7: Keep sub-clusters in their clusters.

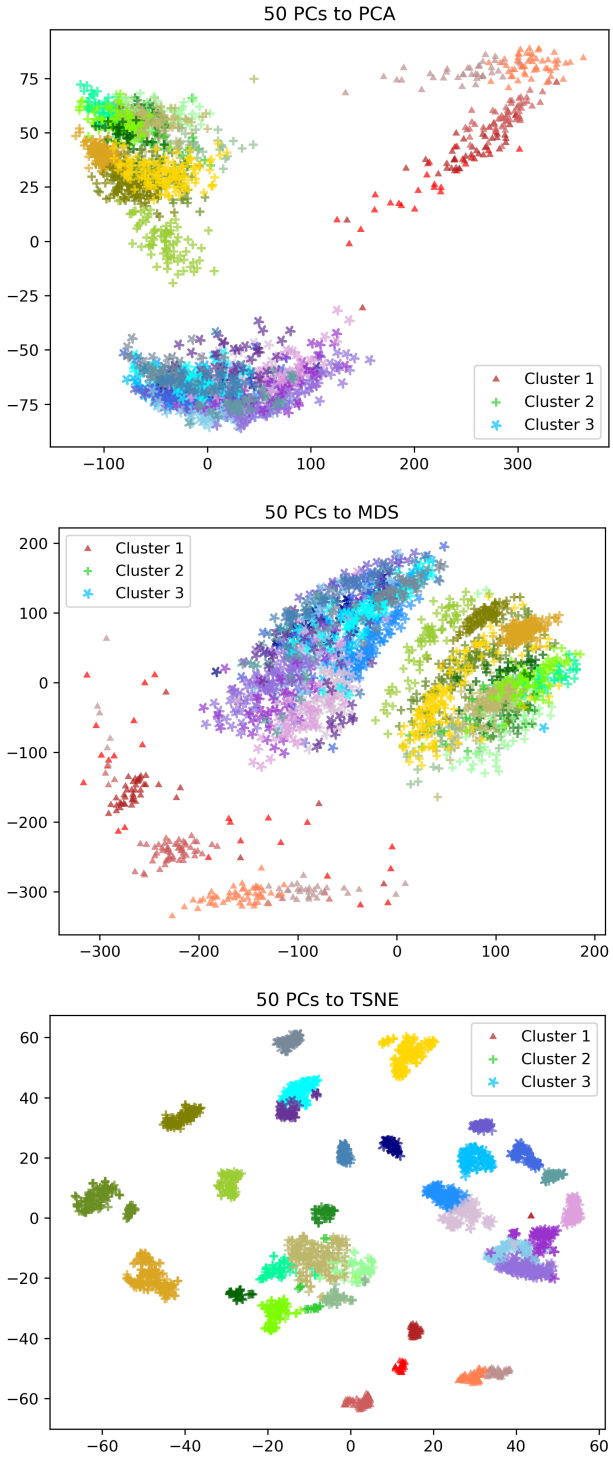


Figure 2.5: Sub-clusters in PCA, MDS, and T-SNE visualizations.

So we see that the clusters mostly make sense.

Next, the sub-clusters will serve as labels for regularized logistic regression. L2 regularization is susceptible to outliers since Euclidean distance squares each dimension before taking the square root. However, because we took $\log_2(X+1)$ of the data, outliers are significantly reeled in. The data set consists of floats (e.g. real numbers), which play well with Euclidean distance, whereas L1 regularization is more suited to categorical data (that we have none of). For these reasons, only L2 regularization is used.

To have a fair comparison to the evaluation set results, X is split into X_{train} and X_{test} (9:1 ratio), where 5-fold cross validation is run only on X_{train}. The test accuracy should be reported on first-seen data, whereas CV that keeps the best score over fits and doesn't generalize well, and the validation set can hardly be considered first-seen if CV takes an average.

```
from numpy.random import default_rng
def train_test_split(X, y, ratio, seed):
  rng = default_rng(seed)
  N_trn = int(ratio * len(y))
  args = {'size': N_trn, 'replace': False}
  train = rng.choice(len(y), **args)
  X_trn, X_tst = X[train, :], X[~train, :]
  y_trn, y_tst = y[train, :], y[~train, :]
  return X_trn, X_tst, y_trn, y_tst
```

Listing 8: Randomly split in train and test sets.

Because `LogisticRegressionCV` could take hours, the results are saved into a text, so it would still be there after coffee and dinner. ☺

```
args = {'random_state': 0,
  'max_iter': 100000, 'verbose': 1 }
clf = LogisticRegressionCV(**args).fit(
  X_trn, y_trn)
trn_score = clf.score(X_trn, y_trn)
tst_score = clf.score(X_tst, y_tst)
str = "Train: {0}\nTest: {1}\n".format(
  trn_score, tst_score)
with open("result.txt") as f:
  f.write(str)
```

Listing 9: Perform logistic regression and save.

The unsupervised accuracy scores are:

Train score: 1.0
Test score: 0.9400921658986175

Unsupervised results

To compare, we use the features corresponding to the top 100 most significant coefficients (1) of the unsupervised logistic regression model to filter an evaluation set, by taking the maximum number (the sum gave a slightly lower score). As a baseline, we also consider the evaluation data with 100 random features (2), and 100 features of most variance (3). A new logistic regression model is trained for all three, and their scores saved.

```
def top_100_features(X_trn, X_tst):
  coef = np.load('coef_cv.npy',
    allow_pickle=True)
  max_coef = np.amax(coef, axis=0)
  indices = np.argsort(max_coef)
  filter = lambda X: X[:, indices[-100:]]
  return filter(X_trn), filter(X_tst)

def random_100_features(X_trn, X_tst):
  n_feats = X_trn.shape[1]
  rng = default_rng(seed=0)
  args = {'size': 100, 'replace': False}
  indices = rng.choice(n_feats, **args)
  filter = lambda X: X[:, indices]
  return filter(X_trn), filter(X_tst)

def high_var_100_features(X_trn, X_tst):
  X_stack = np.row_stack([X_trn, X_tst])
  variances = np.var(X_stack, axis=0)
  indices = np.argsort(variances)
  filter = lambda X: X[:, indices[-100:]]
  return filter(X_trn), filter(X_tst)
```

Listing 10: Filter evaluation set to perform logistic regressions for comparison.

Train score: 0.9990714948932219
Test score: 0.9305054151624549

Top 100 most significant features results

Interestingly, the score is probably within some error margin of the unsupervised score. Our baseline accuracy scores are:

Train score: 0.7780872794800371
Test score: 0.4629963898916967

100 random features results

Train score: 1.0
Test score: 0.9287003610108303

Top 100 most variance features results

It isn't surprising that using random features performs poorly and doesn't generalize particularly well, considering a large number of features capture relatively low variance. It is also expected that using features with high variance performs well. It isn't so expected that the high variance score is similar to the unsupervised score. This would seem to suggest that we should just pick features with high variance, but it's also conceivable that if a high variance feature has similar variances when stratified by cluster label, then wouldn't be as revealing. That isn't the case here.

We compare the variance of the two groups with a histogram plot, and my preferred stacked bar plot. It turns out the significant features from the unsupervised model explain about half the variance as the features with most variance.

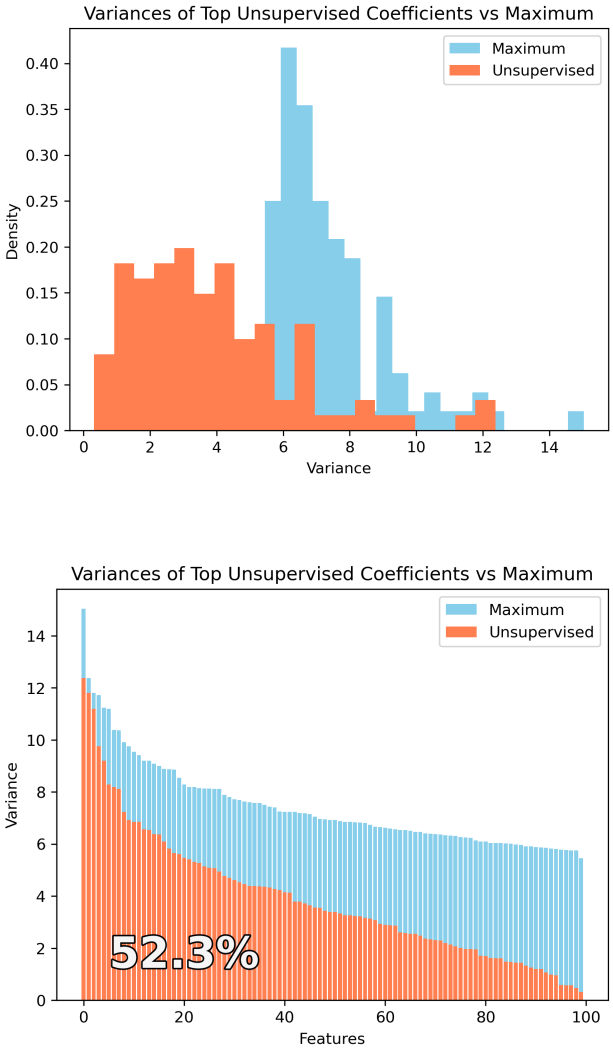


Figure 2.6: 100 features with most variance over most significant coefficients.

Problem 3.
Influence of Hyper-parameters

Part 1.
Principle Components

To motivate this section, first we use the smaller `p1` set to look at the effect of increasing principle components on which T-SNE is run. It's immediately noticeable that the amount and quality of clusters decrease as the number of principle components increase. There are five visual clusters in the plot with 10 PCs, each with distinct shape. By 150 PCs, each cluster takes on a Gaussian characteristic with the most variance across PC1. By 500 PCs, we note that there are three visual clusters left, and we might hastily believe that higher PCs expose truer relations, since we know that there are three main types of cells. That isn't necessarily the case.

I think the plot with 100 PCs is illustrative. If we zoom in on the bottom right corner, we see one visible point that juxtaposes all the remaining points in the top left. Since T-SNE moves points at a distance further away, it seems that this one outlier is all the rest of the points into a corner, making the clusters less distinguishable.

We might also wonder if the effect depends on a fixed amount of PCs or if the effect scales with the amount of samples.

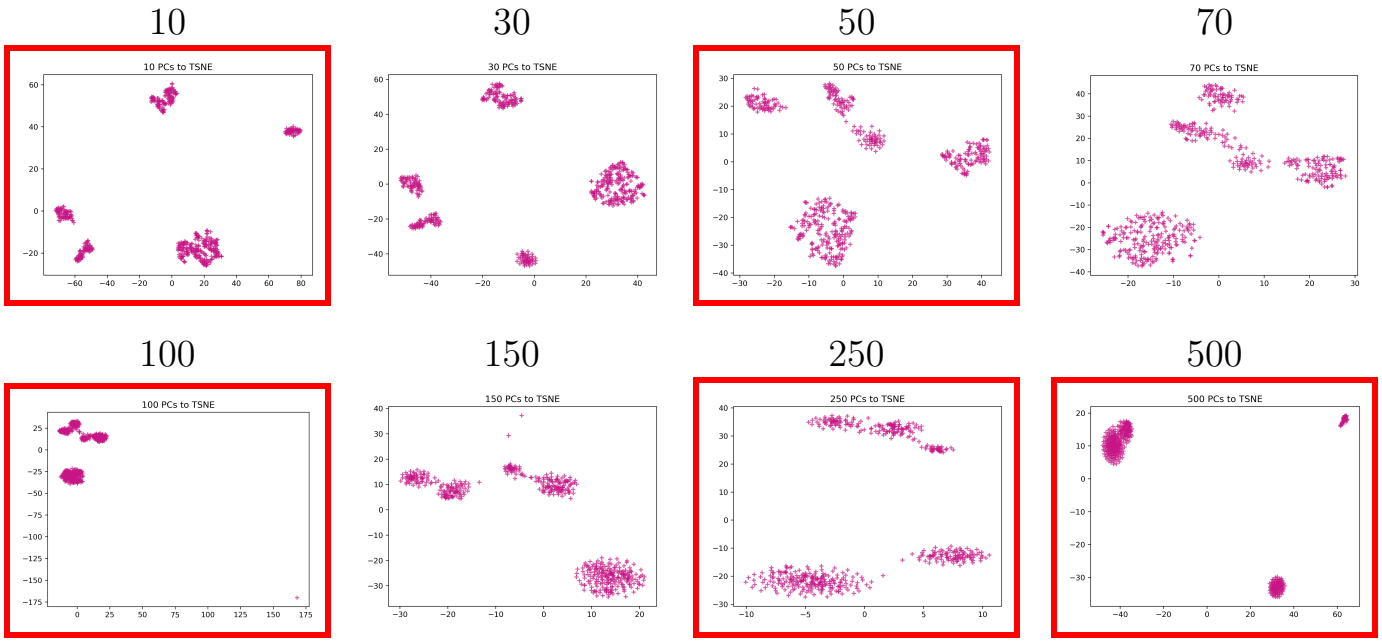


Figure 3.1: T-SNE using various PCs and the smaller `p1` set.

The following is the same plots on the larger `p2_unsupervised` set. Notice that the amount and quality of visual clusters decrease still as PCs increase, but the effect continues to 2000 PCs. To put it in perspective, `p1` has 511 samples, and `p2_unsupervised` has 2169 samples. Where the prior set has three visual clusters at 500 PCs, now there are more, but only two main clusters at 2000 PCs. However, the points being pushed into the corner is still a trend and with greater magnitude. If we look closely, there is still just one outlier at 250 PCs, but at least 5 at 500 PCs, and about 30 by 2000 PCs.

And I think this is indicative of the **curse of dimensionality**. To state plainly, increasing features, d, magnifies Euclidean distance,

$$||x||_2 = \sqrt{\sum_i^D x_i^2} \tag{3.1}$$

so points separated in multiple dimensions appear much further away. In contrast, clusters separated in the most variant dimension might be similar in other dimensions, so they become concentrated.

For example, imagine two clusters in 3D (x, y, z) space centered at $(0, 1, 0)$ and $(0, -1, 0)$ respectively. If we look at the x-y plot, we see two clusters. But if we look at the x-z plot, they become identical. With every additional lower variance dimension, these two clusters seem closer and closer together.

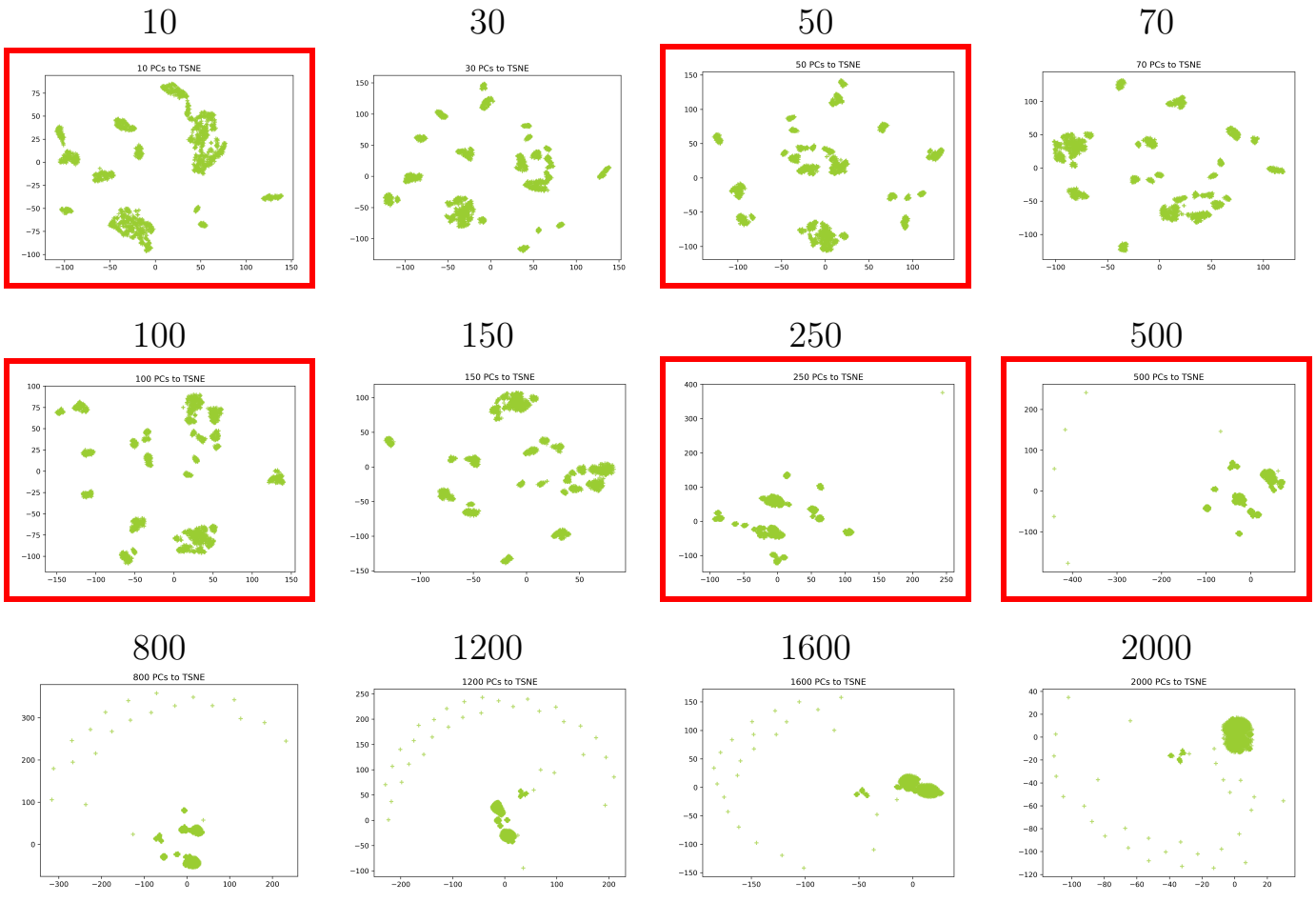


Figure 3.2: T-SNE using various PCs and `p2_unsupervised` set.

Effect of PCs on Clustering

We might try to see how the amount of PCs affect clustering algorithms (besides just observing them visually). To do this, `p2_unsupervised` is reduced to certain components. I run three clustering algorithms with the amount of clusters chosen automatically by a score mechanism:

1. Ward's agglomerative (hierarchical), with clusters determined by maximum cut height.
2. Gaussian mixture model, with clusters determined by minimum Bayesian information content (BIC) score.
3. K-means, with clusters determined by maximum silhouette score.

```python
def maximum_cut_height(linkage_matrix):
  dist = linkage_matrix[:, 2].flatten()
  layer_heights = []
  for i in range(1, len(dist)):
    height = np.abs(dist[i] - dist[i - 1])
    layers_heights.append(height)
  max_indx = np.argmin(layer_heights)
  current = dist[max_indx]
  lower = dist[max_indx + 1]
  cut_height = (current + lower) / 2
  return cut_height
```

Listing 11: Finding the maximum cut height from a dendrogram.

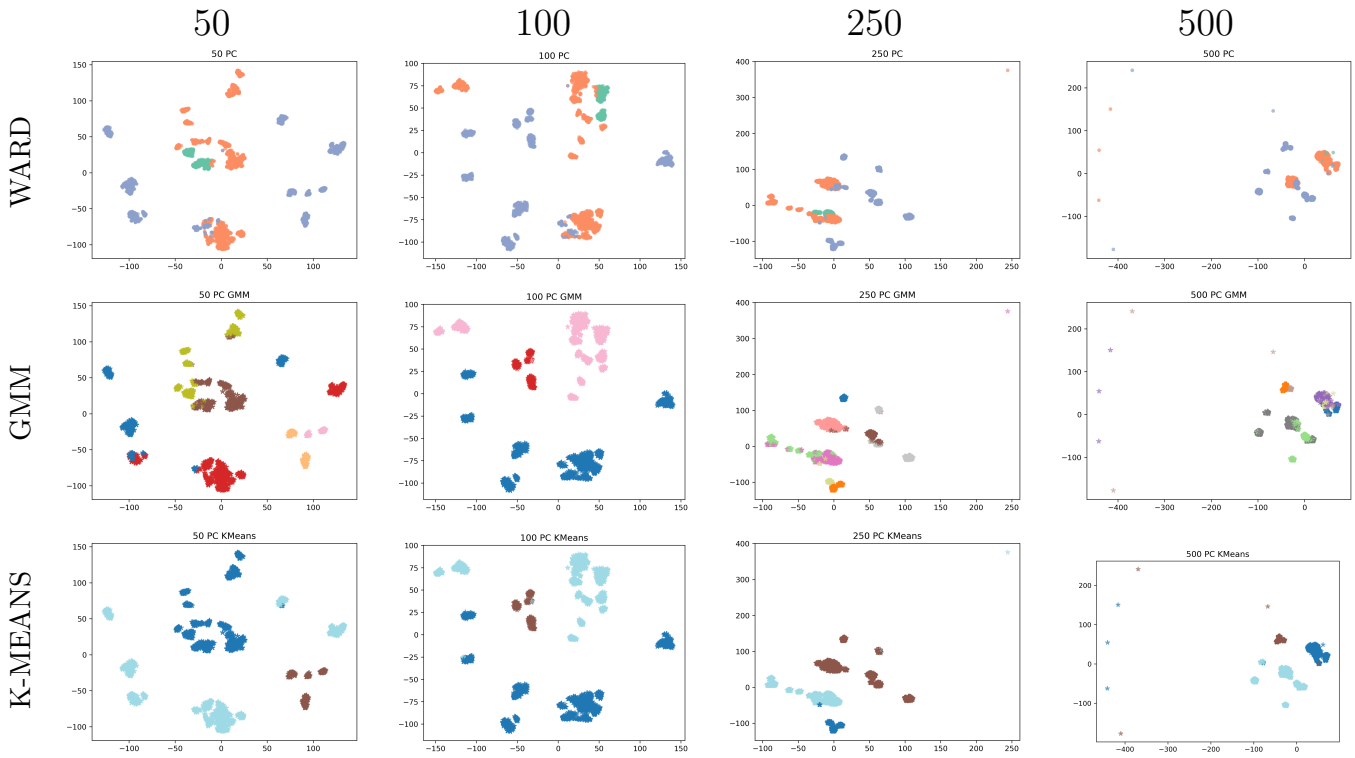


Figure 3.3: Optimal clusters remain constant whether 50 or 500 PCs.

Both Ward and K-means find three optimal clusters. GMM finds 10 or 18, depending on the amount of PCs. While it might look like these decrease as PCs increase, the clusters found by each method remains constant, even for GMM. The only difference is that the clusters appear much more compact, again, due to outliers. It's notable that these outliers are not reported to be

their own clusters by *any* of the methods; rather, they are a mix of the same cluster labels as in the condensed clumps. This would seem to demonstrate that they are an artifact due to the way T-SNE works in high dimensional space.

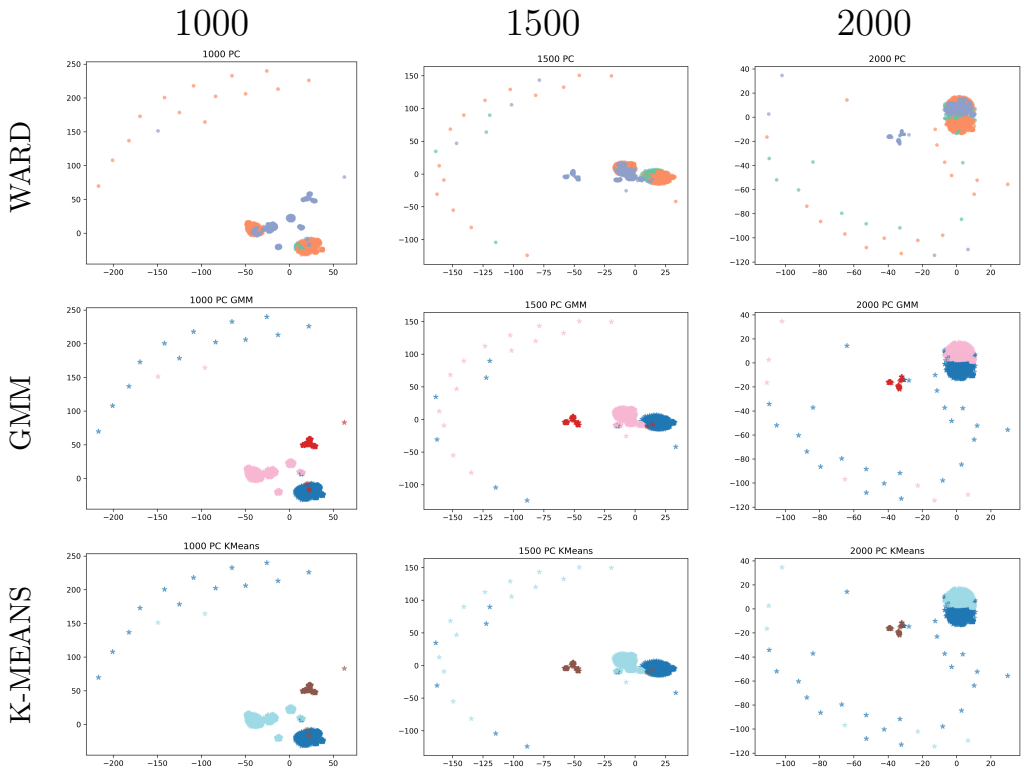
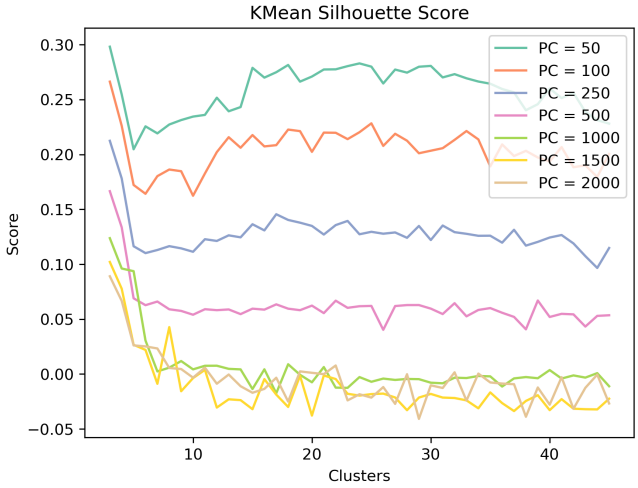


Figure 3.4: Optimal clusters remain constant even in high dimensions.

The following silhouette score plot illustrates that the peak scores remain relatively the same regardless of PCs, but increasing PCs almost decrease the score monotonically until a very high dimension. In other words, increasing PCs has no clear effect on clustering, except somewhat obscuring the clusters with noisier data.



Part 2.
Perplexity

Perplexity is perhaps one of the most perplexing parts of T-SNE. Adjusting it seems to render unpredictable effects. But besides that low perplexity (under 30) seems to dramatically increase training iterations, there is at least one more reason to tune it – perplexity counteracts *some* of the deleterious effects of high dimensionality. Specifically, it spreads out points in areas where they are too condensed. So while the artificial outliers remain, at least the clustering is more visible.

Recall that on the smaller `p1` set, T-SNE rendered 4-5 relatively evenly spaced clusters from 50 PCs. In the following plot, we use 50 PCs, recommended by the sklearn documentation, and vary perplexity. Since the default (indicated by the red box) is close to what we want, increasing it does not help us. Notice that the dense clumps at 30 perplexity spreads out until 250, in which the larger clumps on the bottom left implodes at 500. The other ones follow.

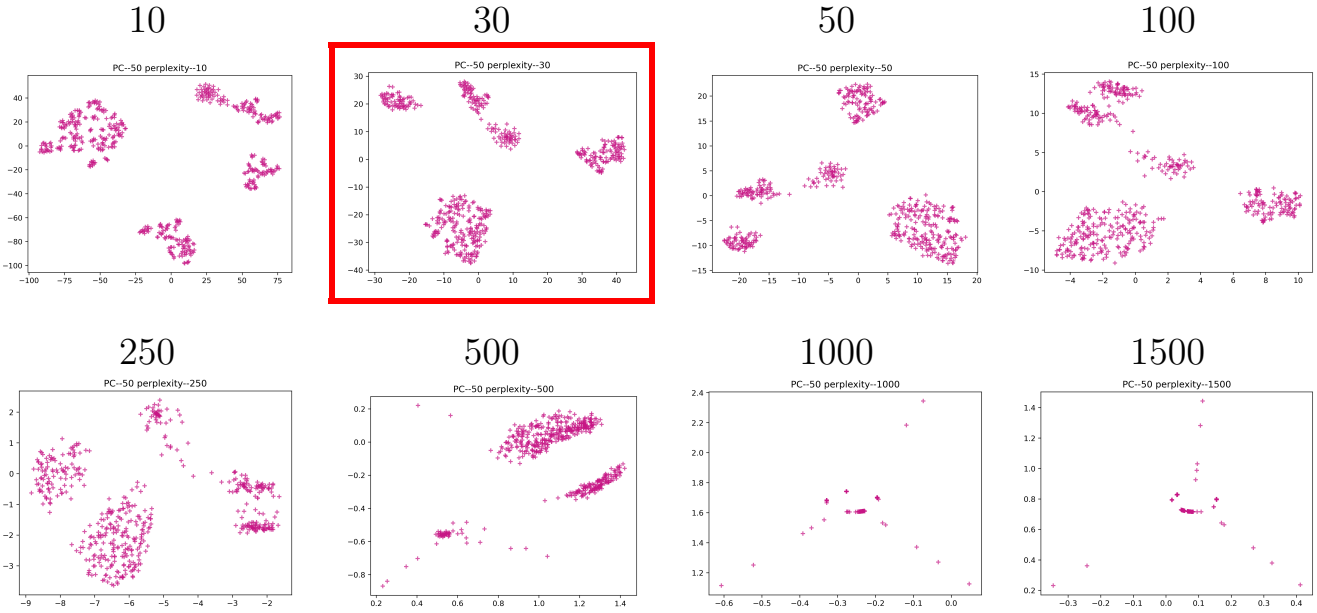


Figure 3.5: T-SNE using 50 PCs, varying perplexity and `p1` set.

We've also previously shown that by 500 PCs on the `p1` set, T-SNE produced three condensed clusters due to outliers. We show it again here in the red box, and show how increasing perplexity expands those clusters. Once again, increasing too much causes unpredictable spread.

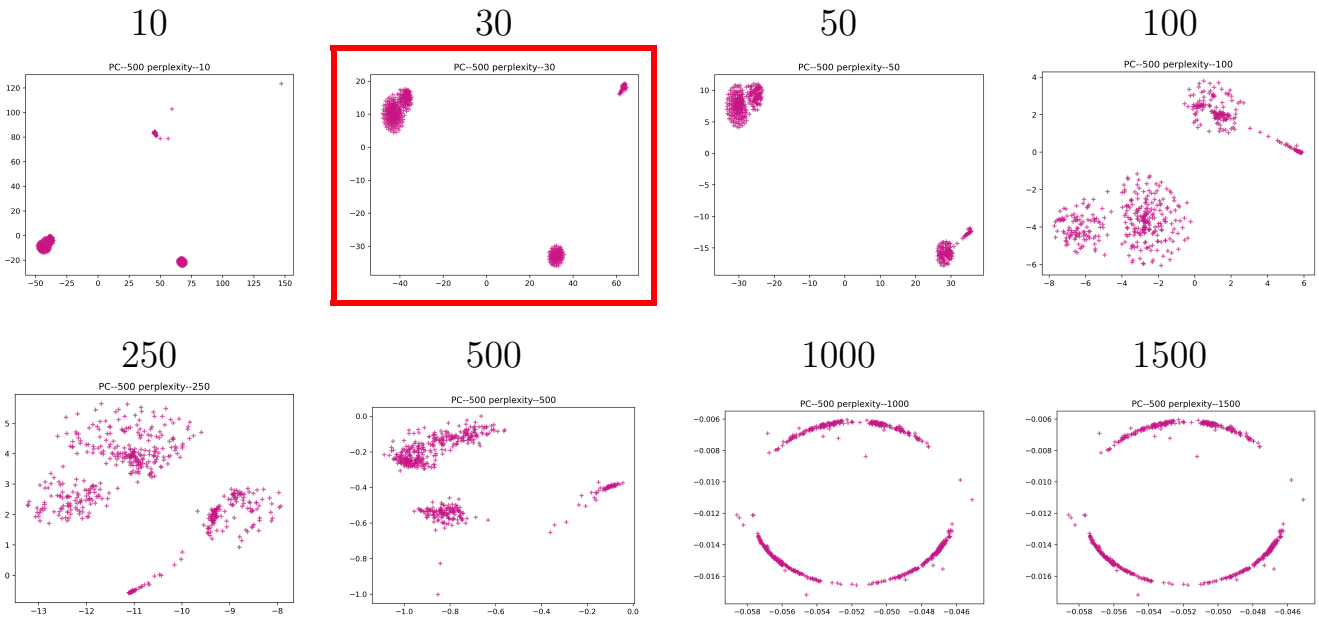


Figure 3.6: T-SNE using *500* PCs, varying perplexity and `p1` set.

On the larger `p2_unsupervised` set, we see the same story unfold. The condensed clusters spread out to a reasonable degree at 500 perplexity until they start to spread out too far. At higher settings, the clusters start to dissipate; however, not to the same degree that it does on the smaller set.

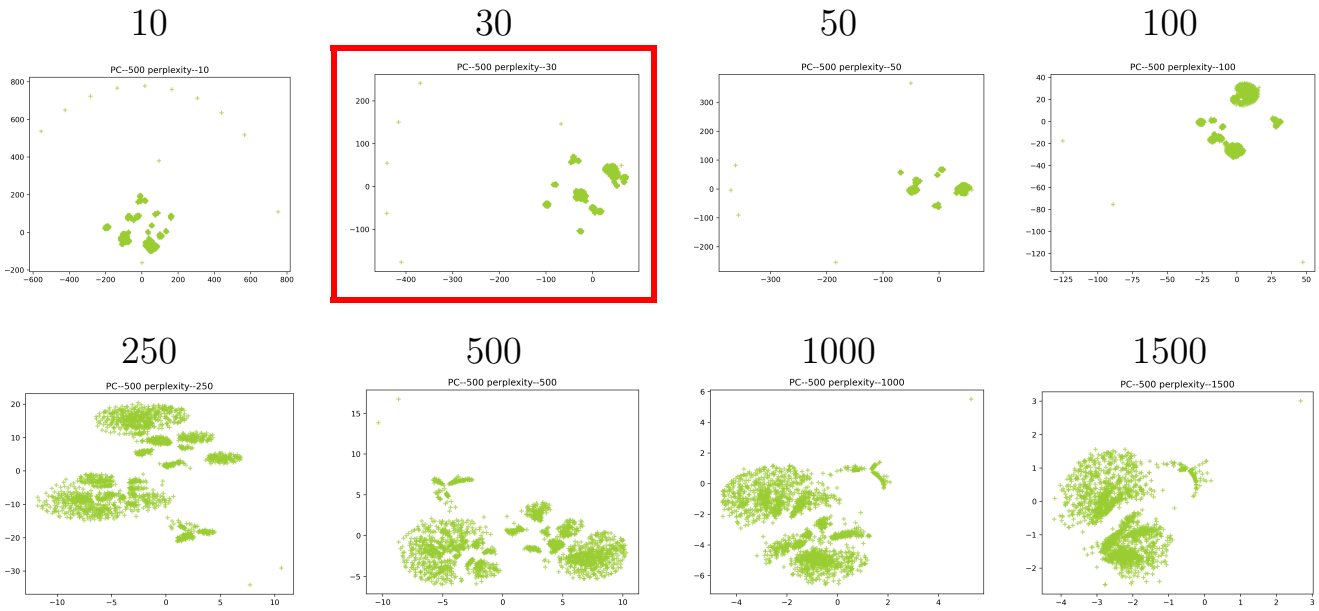


Figure 3.7: T-SNE using 500 PCs, varying perplexity and `p2_unsupervised` set.

In the last exhibit for this section, we show the same larger set using 1200 PCs. Again, we see the clumps in the default setting spread out to a reasonable degree past 250 PCs, even decreasing some of those seeming outliers. At 2000 PCs, we see those isolated points start to increase again, due to the clumps dissipating outwards.

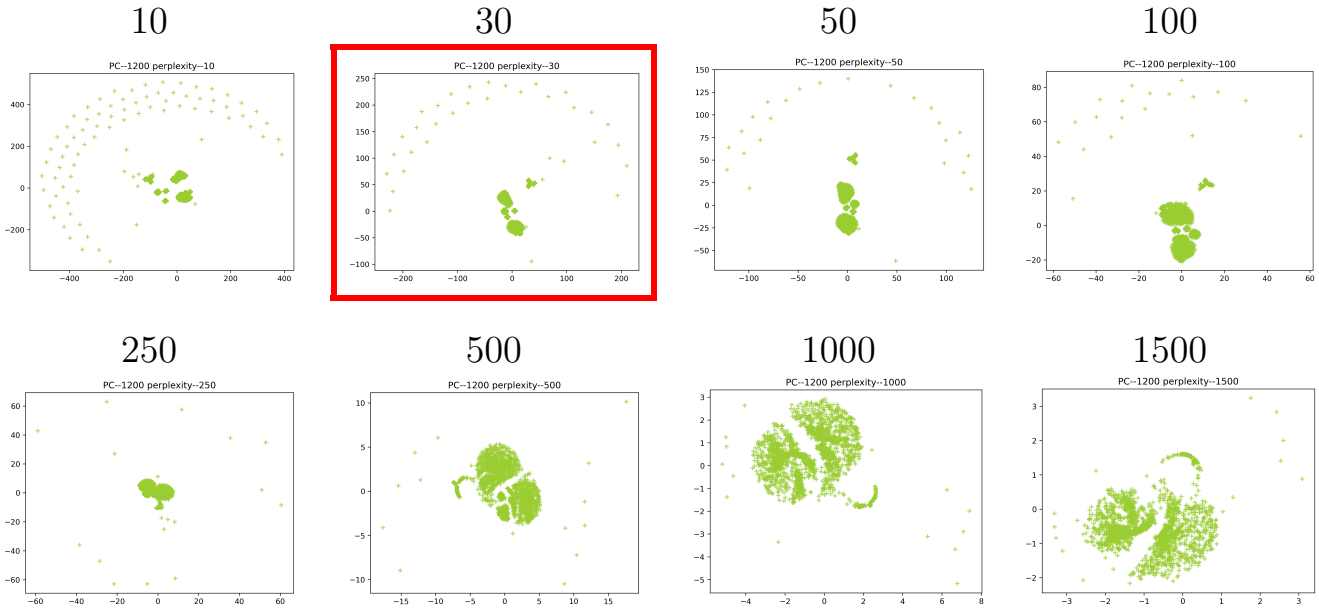


Figure 3.8: T-SNE using 1200 PCs, varying perplexity and `p2_unsupervised` set.

In summary, we could expect perplexity to expand condensed points while pulling in some of the distant points that appear as outliers on T-SNE plots from high dimensions, probably due to the curse of dimensionality. A very high amount of PCs always produces a plot with a centralized clump surrounded by a loose cloud of outliers, and perplexity seems to counteract that to some degree.

Based on the results, it seems that setting perplexity to something between 0.5-1.5 times PCs produces reasonable results.

Part 3.
Regularization

Logistic regression is defined as

$$\begin{aligned} h_\theta(x) &= \frac{1}{1+e^{-\theta^T x}} \\ \text{Cost}(h_\theta(x)^{(i)}, y^{(i)}) &= -[y^{(i)} \log(h_\theta(x)) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))] \end{aligned} \tag{3.2}$$

Therefore, regularized logistic regression is

$$\begin{aligned} J_{\text{L1}}(\theta) &= \frac{\texttt{C}}{D} \sum_i^D \text{Cost}(h_\theta(x)^{(i)}, y^{(i)}) + \frac{1}{D} \sum_j^N |\theta_j| \\ J_{\text{L2}}(\theta) &= \frac{\texttt{C}}{D} \sum_i^D \text{Cost}(h_\theta(x)^{(i)}, y^{(i)}) + \frac{1}{2D} \sum_j^N \theta_j^2 \end{aligned} \tag{3.3}$$

The generalized model is Elastic Net,

$$J_{\text{Elastic Net}}(\theta) = \frac{\texttt{C}}{D} \sum_i^D \text{Cost}(h_\theta(x)^{(i)}, y^{(i)}) + \frac{\texttt{l1\_ratio}}{D} \sum_j^N |\theta_j| + \frac{1-\texttt{l1\_ratio}}{2D} \sum_j^N \theta_j^2 \tag{3.4}$$

In this section, we examine the spectrum of regularization from L1 to L2 and see how they affect the results. Unfortunately, only the `saga` solver of `LogisticRegression` accepts Elastic Net penalty, and it's extremely slow with the full set of features. The antidote I found is to use `SGDClassifier` with `log` loss, which performs stochastic gradient descent on logistic regression. To combat the results being less stable, I use four random states *[0, 7, 8, 42]*, which I'm sure you'd agree are lucky numbers. The results are averaged over the random states.

As before, a logistic regression model is trained for each setting on the `p2_unsupervised` data set. We use maximum coefficients to select the top 100 features, and compare the sum of their variances against the sum of variances of the 100 maximally variant features of `p2_evaluation`. Finally, we use the selected features to run `LogisticRegressionCV` on the evaluation set to find out if accuracy scores correspond to the variance.

	RANDOM STATES				
L1 Ratio	0	7	8	42	Mean
1.00	0.5330	0.5652	0.5618	0.5132	0.5433
0.75	0.5179	0.5341	0.5625	0.5546	0.5398
0.50	0.5319	0.5289	0.5332	0.5415	0.5339
0.25	0.5237	0.5333	0.5255	0.5219	0.5302
0.00	0.5175	0.5237	0.5343	0.5010	0.5186

Figure 3.9: Ratios of explained variance of features corresponding to top 100 coefficients versus 100 features with maximum variance.

L1 Ratio	RANDOM STATES 0	7	8	42	Mean
1.00	0.9251	0.9296	0.9088	0.9233	0.9217
0.75	0.9260	0.9305	0.9224	0.9368	0.9289
0.50	0.9224	0.9106	0.9278	0.9251	0.9215
0.25	0.9106	0.9079	0.9350	0.9350	0.9210
0.00	0.9323	0.9395	0.9251	0.9350	0.9319

Figure 3.10: Test score accuracies using features corresponding to top 100 coefficients.

The cross validation tests 10 different magnitudes of regularization `C`, which will not be discussed here (although the most common best setting tested is $C \approx 0.05$). It also tests L1, L2, and elastic net (0.5L1+0.5L2), using the 100 features obtained from each of the settings above. The best setting is usually L2, with an occasional sprinkle of the others.

We see that increasing L1 over L2 increases the variance of the resulting top 100 features, slightly. Interestingly, the variance of the variances is also higher with L1 (7.366) than L2 (6.478). So the spread between most variant and least variant features are also higher.

However, this same trend doesn't seem to translate to test accuracy. The highest accuracy model is actually one with L2 regularization at nearly 0.94, even though L1 produces higher variance features on average. That means these small differences may just be the result of noise. It also means that feature variance is not necessarily proportional to the accuracy of the model. It may just be one indication out of many.

Finally, we compare the results to if we had no regularization at all.

	RANDOM STATES 0	7	8	42	Mean
Variance ratio	0.5529	0.5218	0.5554	0.5291	0.5398
Variance of variance	5.775	5.6957	5.5829	5.9972	5.7627
Test accuracy	0.9314	0.9260	0.9287	0.9224	0.9271

Figure 3.11: Variance statistics and test accuracy without regularization.

Surprisingly, this isn't that bad at all. The variance ratio (to maximum variance features) is in line with the range regularization produces, but the variance of variance is actually higher. This means having no regularization gets lower highs but higher lows, with roughly the same mean. Test accuracy is also similar to regularized results.

Perhaps what this really tells us is that whether the model over fits is not really relevant to selecting the top 100 features. After all, the act of limiting the features is in itself a form of regularization.

List of Figures

List of Listings