ASSIGNMENT 4

November 26, 2019

Bar Kadosh (bk497), Ben Kadosh (bk499)
CS5785: Applied Machine Learning
Instructor: Nathan Kallus, Teaching Assistants: Xiaojie Mao, Yichun Hu

Contents

0.1 Summary

Our general approach to the coding sections of this assignment was to cleanse and transform the data as needed to make it easy to use and provide the desired insights. Below we describe the approach in more detail for each problem.

For exercise 1, the focus was on better understanding MDS and different types of clustering through the analysis of similarity/dissimilarity of different populations. We used several extensions of sklearn to find the optimal k for clustering, as well as to implement k-medoids. Ultimately, the goal was to cluster the relevant data and label the different clusters for each exercise. We then evaluated each clustering to identify any changes and to analyze and better understand what information may be lost in each exercise, as well as analyze why each approach provides the results that it did. Further documentation can be found in the supporting jupyter notebook.

For exercise 2, the focus was on better understanding Random Forests and how to implement and fine-tune them. We worked through the process of implementing Random Forests with sklearn to predict the pixel values for an image (the Mona Lisa). We analyzed how major parameters for a Random Forest (tree depth and number of trees) can influence model accuracy (image prediction in this case) and how fine-tuning the model with these parameters will influence the model complexity. Further, we were better able to understand the concept of bagging and reducing variance through adding more trees to the forest. We analyzed decision rules for a tree and identified upper bounds for the patch color count for the outputs with differing numbers of trees and altered depths of trees.

For exercise 3, the focus was on better understanding SVM and how to implement the algorithm and fine-tune its parameters in sklearn. We first cleansed the data and encoded where necessary so we could fit the data with a support vector classifier. We were able to better understand how support vectors work and the corresponding coefficients returned in the implementation. Furthermore, we worked with different kernel implementations and further practiced cross validation by fine-tuning the C and gamma parameters. We plotted and displayed the different accuracy scores for each of the three kernels with different optimal C and gamma values and analyzed the performance of each.

For exercise 4, the focus was to better understand how neural networks work at a high level. We analyzed the fully connected layers, as well as the input and output layers. We assessed the number of neurons in each layer as well as the activation functions used. Ultimately, we experimented with the learning rate and the number of layers (both by adding and removing) to see how they impacted performance in predicting the image of a cat.

For the written exercises, all of our logic and explanations can be found in this write-up in the "written" section.

0.2 Problem 1: Multidimensional scaling for genetic population differences

In this exercise, we will look at a dataset of 42 human geographic populations collected by Cavalli-Sforza et al. in The History and Geography of Human Genes, 1994. There are many ways to measure genetic similarity between populations. This work uses the modified Nei's distance, which compares allele frequencies at specific locations within the human genome. Other ways to measure genetic similarity include the edit distance between DNA sequences or Euclidean distance of gene expressions. This dataset contains comparisons between every pair of populations in the study as a distance matrix. Here, $D_{i,j}$ is the dissimilarity between population i and j. Higher scores indicate more dissimilarity. Since this distance is not based on an underlying vector representation, we cannot directly use traditional techniques like classification or clustering to analyze it. Our first step will be to embed this distance matrix into an m-dimensional vector space.

(a) First, use multidimensional scaling (MDS) to coerce D into a 2-dimensional vector representation. You can use the functions in sklearn.manifold for this.

i. MDS will attempt to output a set of points $x \in \mathbb{R}^{42,m}$ such that the Euclidean distance between every pair of points approximates the Nei's distance between these populations, or $||x_i - x_i||_2 \approx D_{i,j}$. What assumptions are being made? Under what circumstances could this fail? How could we measure how much information is being lost? Please explain.

> One of the major underlying assumptions of MDS is that variance best explains information – meaning, the components that account for the most variance are also the ones that best explain and provide insights into the data. Further, as is referenced in Quantum Machine Learning by Peter Wittek, "As principal component analysis and multidimensional scaling are based on the eigendecomposition of the data matrix X, they can only deal with flat Euclidean structures, and they fail to discover the curved or nonlinear structures of the input data (Lin and Zha, 2008)." There is a given assumption about the structure and linearity of the data. This could also

lead it to fail when the structure is non-linear. In addition, if few dimensions do not approximate the data well, then MDS will fail to be useful. In terms of measuring how much information is lost, we can use a stress/goodness of fit metric. The stress metric evaluates the differences between the actual and predicted distance values. The smaller the stress value, the better the MDS is at capturing the dimensions that best predict the actual distances. The formula for stress is listed below.

$$\text{stress} = \sqrt{\frac{\sum(d_{ij} - \hat{d_{ij}})^2}{\sum(d_{ij})^2}}$$

Link 1: https://www.sciencedirect.com/topics/computer-science/multidimensional-scaling
Link 2: https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Multidimensional_Scaling.pdf

ii. One way of increasing the quality of the output is by increasing the dimensionality of the MDS result. How many dimensions are necessary to capture most of the variation in the data? There are several ways of making this judgment: for instance, you might sample some quality measure between x and D while varying m, or you might count the nonzero singular values of D, or inspect the singular values of x at some high dimension like m = 20. Briefly explain your method and justify why it makes sense.

We chose to use the stress/goodness of fit metric mentioned above. It is returned as one of the attributes in sklearn's MDS implementation, where they define stress as "The final value of the stress (sum of squared distance of the disparities and the distances for all constrained points)." With this, we ran MDS with different dimension counts ranging from 1 to 42 and plotted the stress values for each. As we can see, the majority of the variation of the data can be captured with just 4 to 5 dimensions. After 5 dimensions, the stress levels plateau, meaning that we can get similar value out of just 5 dimensions instead of all 42 dimensions.

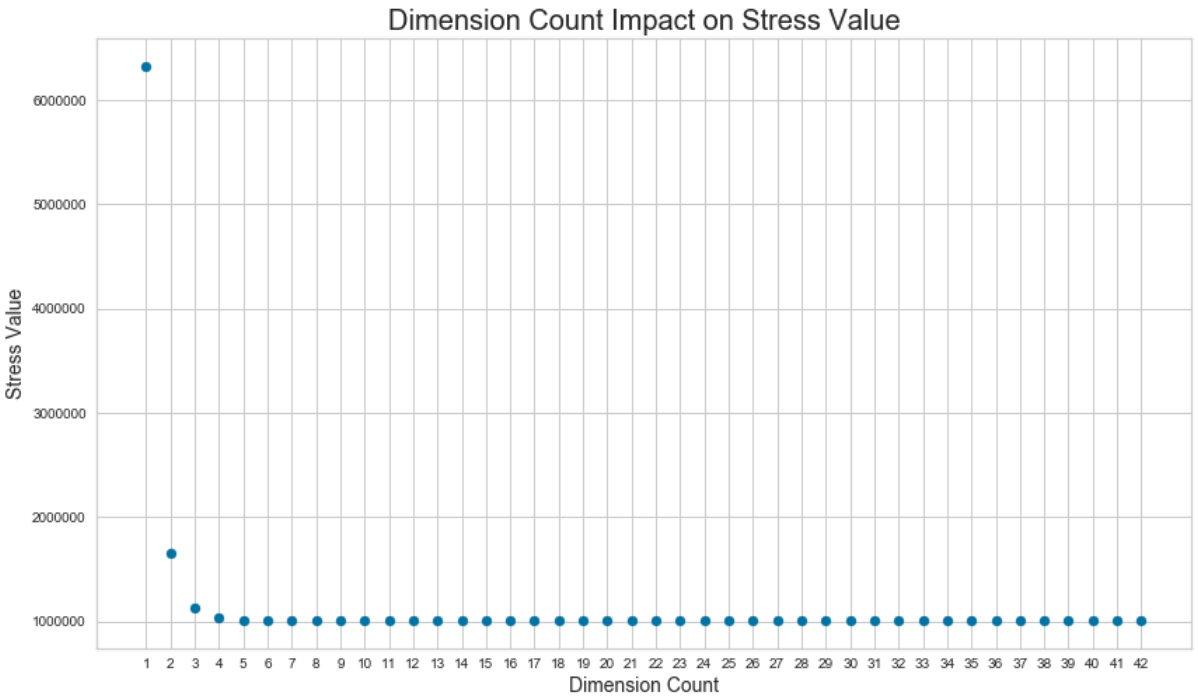


Figure 1: Optimal Dimension Count

iii. Use MDS to embed the distance matrix into only two dimensions and show the resulting scatterplot. Label each point with the name of its population.

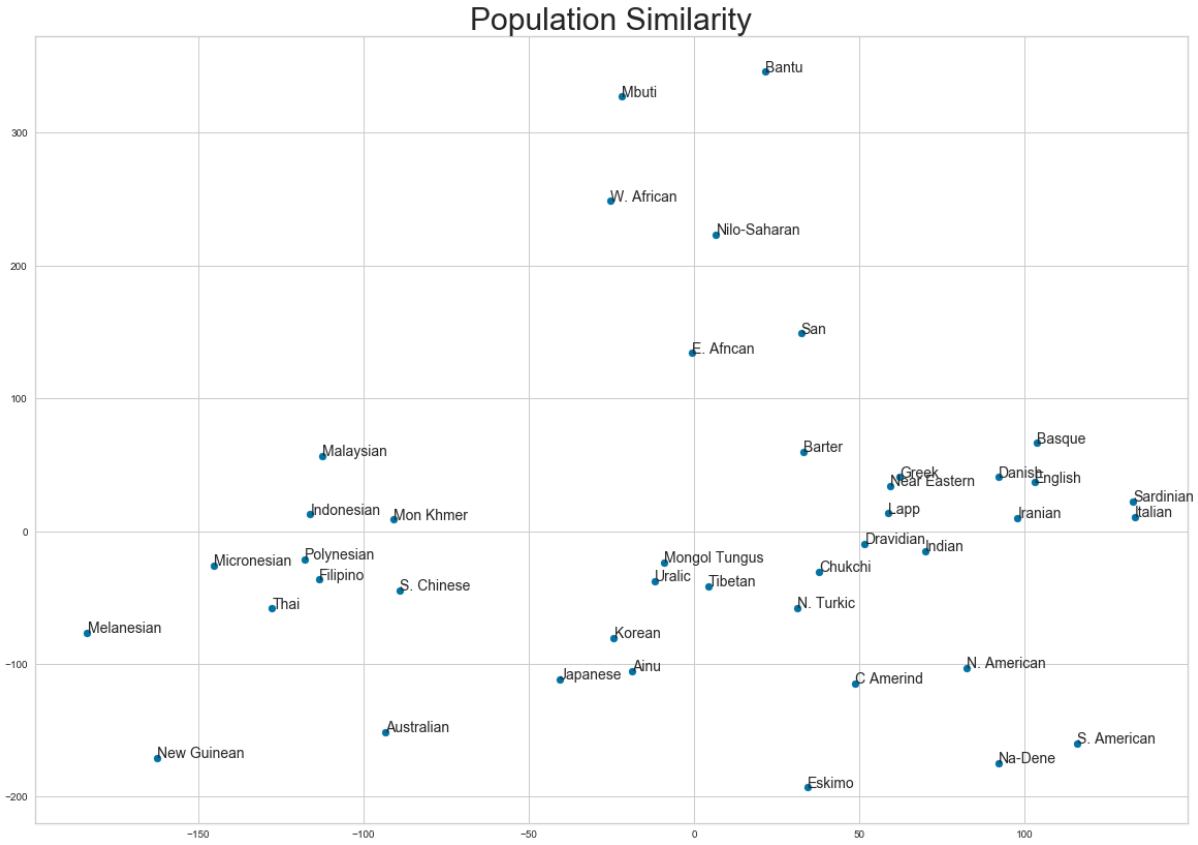


Figure 2: Population Similarity

(b) k-means on 2D embedding. Select an appropriate k and run k-means on the scatterplot. Show the resulting clusters. Since we are working with only two dimensions, this clustering is likely to lose a lot of highdimensional structure. Do you agree with the resulting clustering? What information seems to be lost?

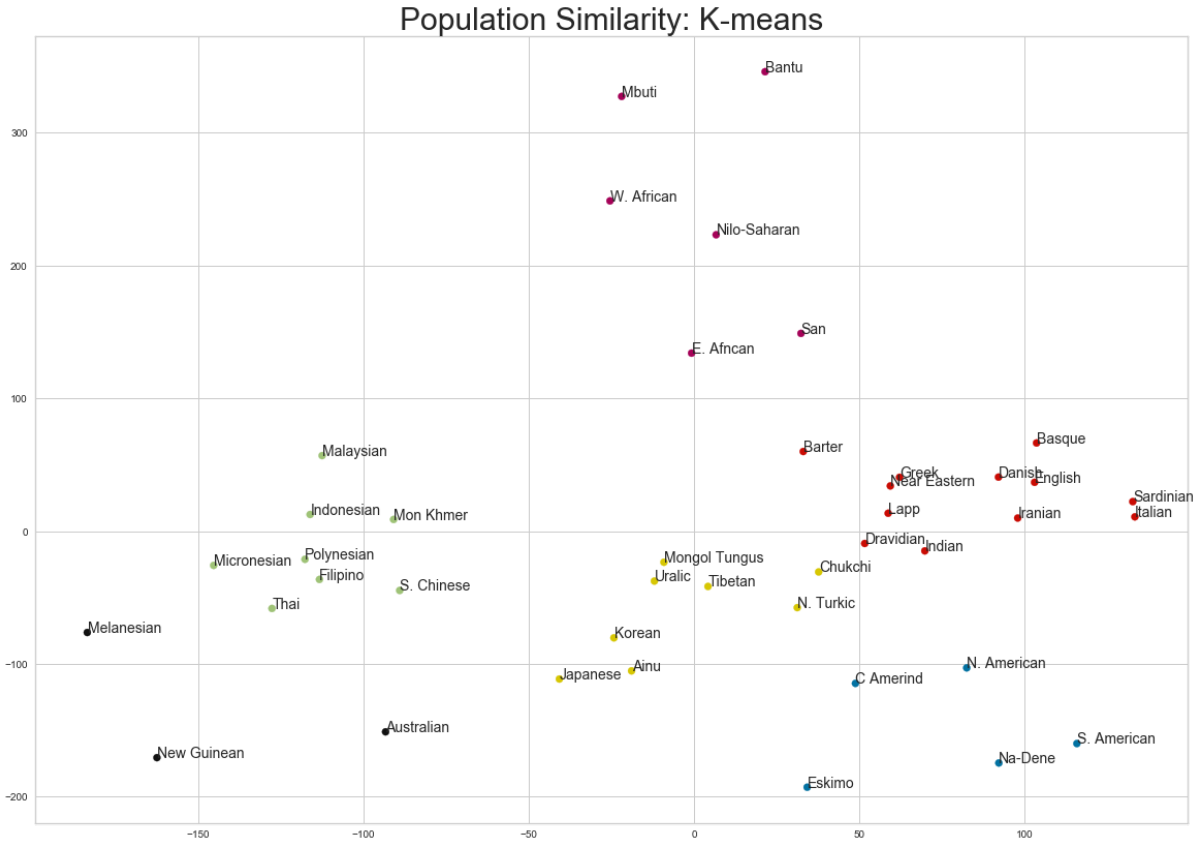


Figure 3: Population Similarity: K-means clustering

In order to find the optimal k value for clustering, we used KElbowVisualizer from the Yellowbrick library, which is an extension of Scikit-Learn. When we ran the KElbowVisualizer, we found that the optimal number of clusters, k, was 6. Once we found the optimal k, we proceeded to cluster the 2D embedding of the Distance Matrix, D. We then color coded the labels for each of the clusters and re-graphed the scatter plot.

In evaluating the resulting scatter plot, the clustering seems to be fairly reasonable. The populations have a geographic correlation so far as we can see, which we would expect. For example, Korean and Japanese are in the same cluster, as are Thai and Filipino. It also seems that African groups are generally clustered together, as are North/South American groups. However, it does appear that some information regarding distance and origin seems to have been lost. For example, Greek and Indian populations are currently clustered together and appear to be more similar than Greek and Italian populations. It seems to defy expectations in the sense that proximity of nations/regions (which we assume correlates with genetic proximity) may not be fully captured with a 2D MDS result. However, this is just an assumption and it is possible that Indians and Greeks may in fact be more related than Italians and Greeks (though we are speculative).

(c) Comparing hierarchical clustering with K-Means. Use hierarchical clustering to cluster the original distance matrix. We suggest using the functions in scipy.cluster.hierarchy for this, as this library can plot the resulting graph structure. Show the resulting tree as a dendrogram, labeling the x axis with the categorical population names and the y axis with the Nei's distance between clusters. (It's also possible to do this with sklearn, but traversing the resulting structure is much harder). To turn the resulting tree into a flat clustering of points, cut off the dendrogram at a certain distance by merging all subclusters within this distance together. This can be done with the scipy.cluster.hierarchy.fcluster function. Select a distance cutoff that roughly corresponds with your chosen k earlier, balancing the number of points in each cluster with the number of resulting clusters. Visualize the resulting clustering by coloring the corresponding points on your 2D MDS embedding. How does this clustering compare with the k-means clustering you computed earlier?

In order to construct the Dendrogram and flattend clustering, we imported dendrogram, linkage, and fcluster from scipy.cluster.hierarchy. We first plotted the Hierarchical Clustering Dendrogram on the entire data matrix, D. Once we had the dendrogram, we were able to flatten the data and cluster it using the fcluster function. We used the same number of clusters, 6, as calculated above for k-means. The graphs and analysis are displayed below.

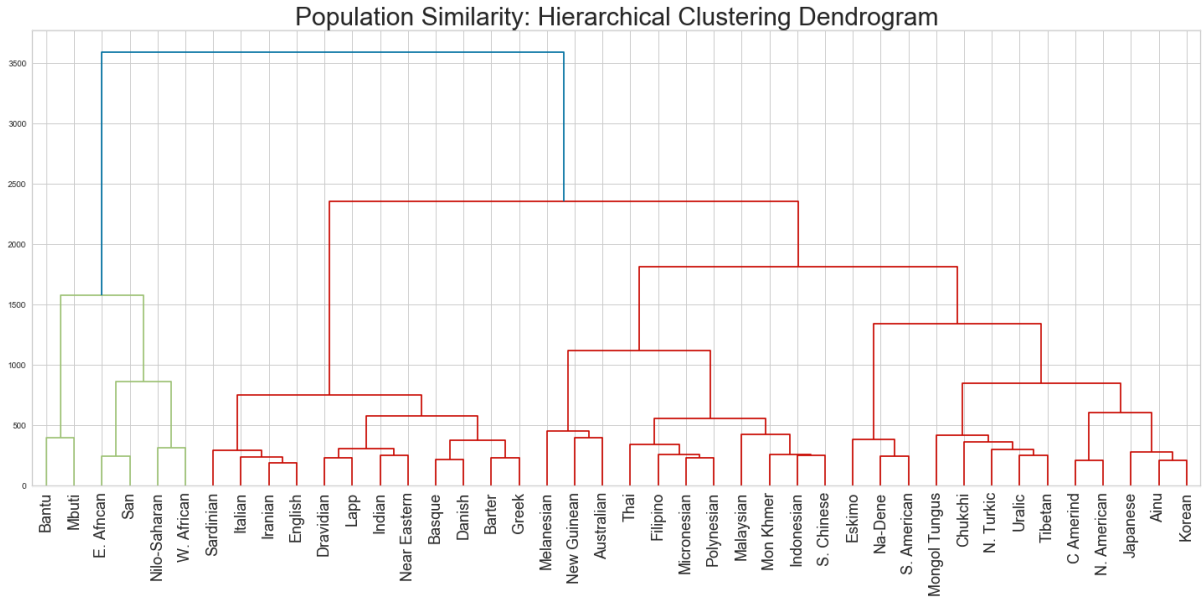


Figure 4: Population Similarity: Hierarchical clustering

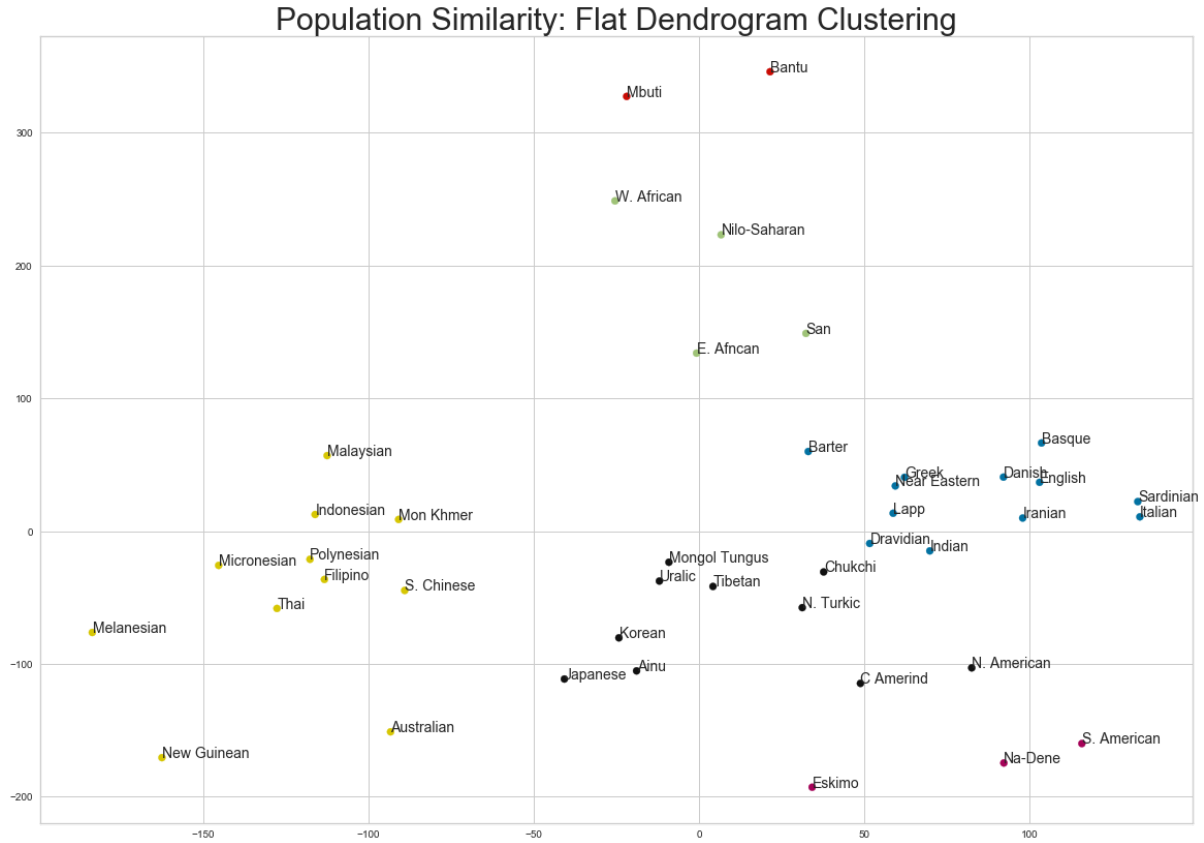


Figure 5: Population Similarity: Flat clustering

The K-means clustering and flat clustering graphs seem somewhat similar; however, they do share some noticeable differences. Though Melanesian, New Guinean, and Australian were previously their own cluster, they have now been clumped together with Thai, Filipino, and others. In addition, Mbuti and Bantu have split off to form a new cluster. Similarly, C. Amerind and N. American have also moved to a new cluster. So, while most cluster labels generally remained the same, there has certainly been some movement.

We suspect that the flat clustering would be more accurate simply because the clustering occurred on the entire matrix instead of a 2D embedding of MDS, which would surely lose some information.

(d) Compare k-medoids with k-means. Repeat the above experiment, but applying k-medoid clustering on the original distance matrix. Show the resulting clusters on a 2D scatterplot. Are there any significant differences between the clustering chosen by k-medoids compared to k-means?

For the k-medoids implementation, we downloaded sklearn_extra, an extension of scikit-learn and used the KMedoids implementation from the cluster library. Again, we used the same number of clusters, 6, as was previously determined with the KElbowVisualizer. The scatterplot and analysis are displayed below.

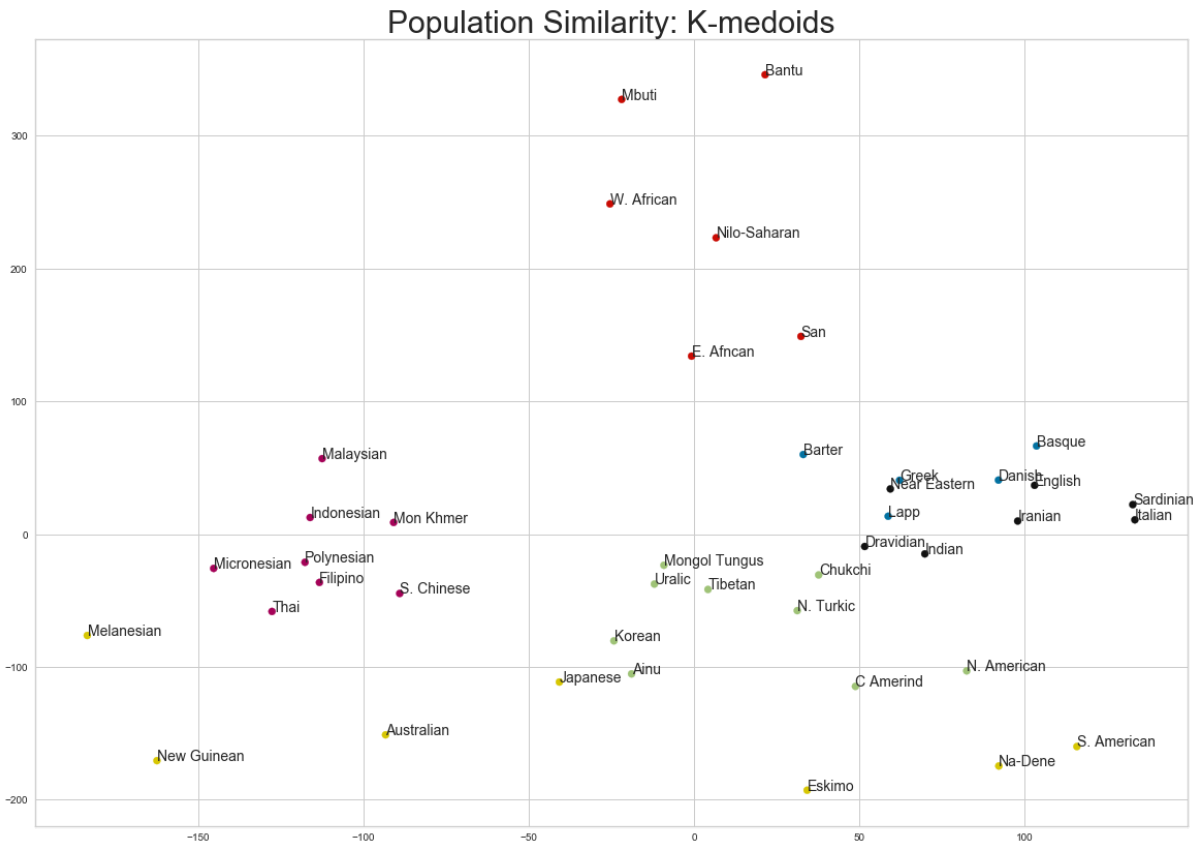


Figure 6: Population Similarity: K-medoids clustering

There are several notable changes in the clustering with K-medoids. What immediately stuck out was that the data points in the bottom portion of the Y-axis all appear to cluster together, almost disregarding the X-axis influence on similarity. Almost all labels below -100 on the Y-axis were clustered together as yellow. In addition, South American getting clustered with Australian and Melanesian seems to make less logical sense than the prior plots.

The other interesting distinction noted is that there was more differentiation into two clusters for the populations representing the geographic region of Europe + India + Iran. Now, Greek, Basque, and Danish are not clustered with Italian, Indian, and Iranian. The rearrangement of clustering highlights that K-medoids will noticeably impact the resulting clustering.

0.3 Problem 2: Random Forests For Image Approximation

In this question, you will use random forest regression to approximate an image by learning a function, f: $\mathbb{R}^2 \rightarrow \mathbb{R}$, that takes image (x, y) coordinates as input and outputs pixel brightness. This way, the function learns to approximate areas of the image that it has not seen before.

(a) Start with an image of the Mona Lisa. If you don't like the Mona Lisa, pick another interesting image of your choice.

(b) Preprocessing the input. To build your "training set," uniformly sample 5,000 random (x, y) coordinate locations. Note if you use an image other than the Mona Lisa image linked above you may need to sample a different number of coordinates, so that you sample approximately the same percentage of the total number of pixels in the image (around 1 %).

- What other preprocessing steps are necessary for random forests inputs? Describe them, implement them, and justify your decisions. In particular, do you need to perform mean subtraction, standardization, or unit-normalization?

  As was discussed with Xiaojie and Professor Kallus, Random Forests do not require pre-processing. As is further explained by Professor Kilian Weinberger of Cornell University in the following lecture from around 11 minutes to 11:45 minutes (link below), "If you want to use most machine learning algorithms you want to make sure that feature values scale between 0 - 1, [for most machine learning algorithms] you have to preprocess the features very carefully... with Random Forests you don't have to do any of that." Therefore, you do not need to perform mean subtraction, standardization, or unit-normalization.

  Link: https://www.youtube.com/watch?v=4EOCQJgqAOY

(c) Preprocessing the output. Sample pixel values at each of the given coordinate locations. Each pixel contains red, green, and blue intensity values, so decide how you want to handle this. There are several options available to you:

- Convert the image to grayscale
- Regress all three values at once, so your function maps (x, y) coordinates to (r, g ,b) values: f: $\mathbb{R}^2 \rightarrow \mathbb{R}^3$
- Learn a different function for each channel, $f_{Red} : \mathbb{R}^2 \rightarrow \mathbb{R}$, and likewise for f_{Green}, f_{Blue} .

Note that you may need to rescale the pixel intensities to lie between 0.0 and 1.0. (The default for pixel values may be between 0 and 255, but your image library may have different defaults.)
What other preprocessing steps are necessary for random regression forest outputs? Describe them, implement them, and justify your decisions.

For the output preprocessing, we converted the image to grayscale in approximating the image. In order to do so, we used the following formula per guidance from some online searches:

grayscale_val = 0.2989(r) + 0.5870(g) + 0.1140(b)

Other than that, as we noted above, random forests don't require much tuning or preprocessing, so no further preprocessing for the output was done.

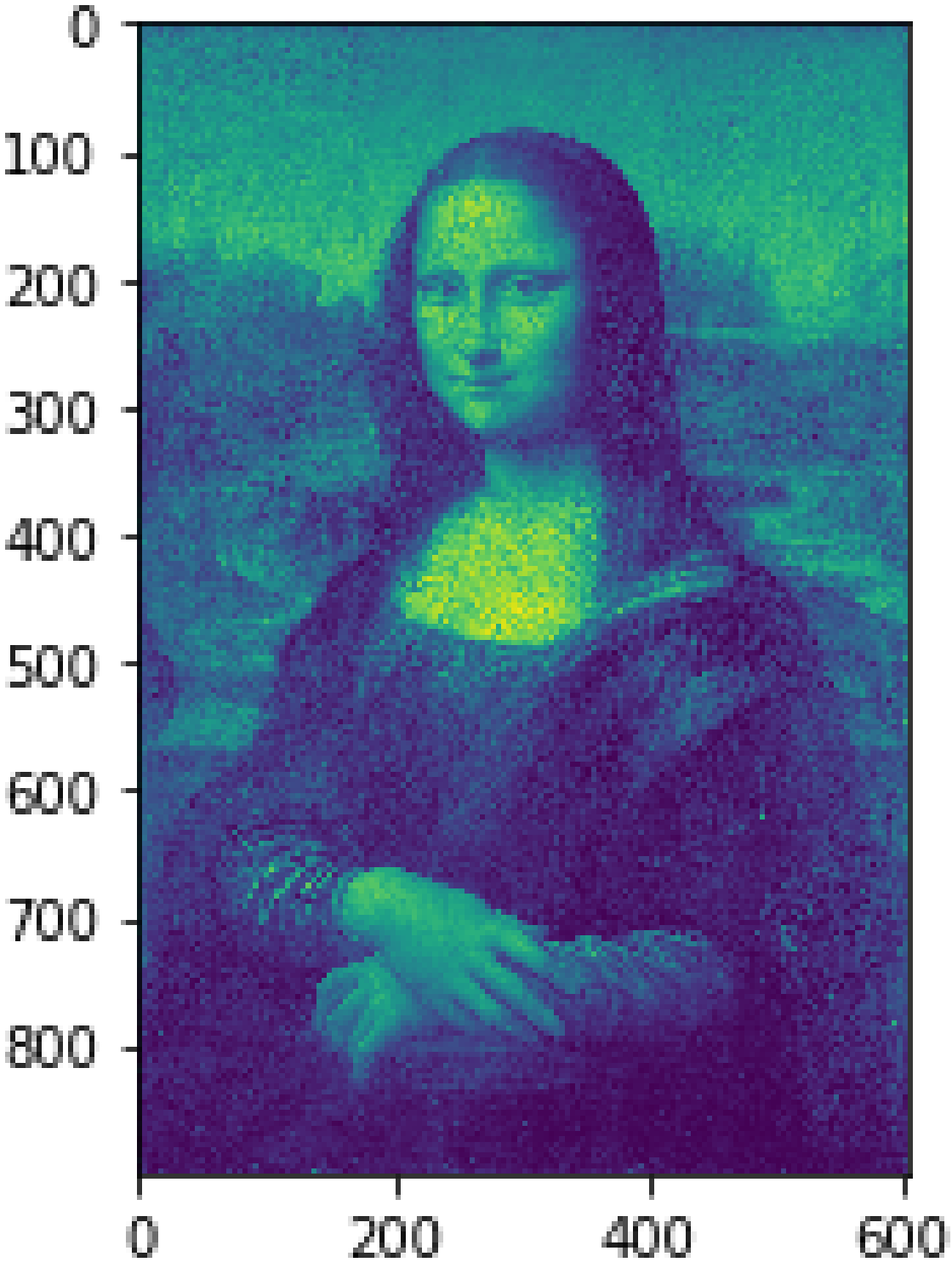


Figure 7: Mona Lisa Grayscale

(d) To build the final image, for each pixel of the output, feed the pixel coordinate through the random forest and color the resulting pixel with the output prediction. You can then use imshow to view the result. (If you are using grayscale, try imshow(Y, cmap='gray') to avoid fake-coloring). You may use any implementation of random forests, but you should understand the implementation and you must cite your sources.

For the random forest implementation we used the RandomForestRegressor regression model from sklearn.ensemble. The two main parameters which we experimented with fine tuning were n_estimators (which sets the number of trees in the forest) and max_depth (which sets the maximum depth of the tree). In addition, we used the returned estimators_ attribute to better display the decision rules below. We further describe how the implementation works and how fine tuning the parameters affects the results below.

(e) Experimentation.

i. Repeat the experiment for a random forest containing a single decision tree, but with depths 1, 2, 3, 5, 10, and 15. How does depth impact the result? Describe in detail why.

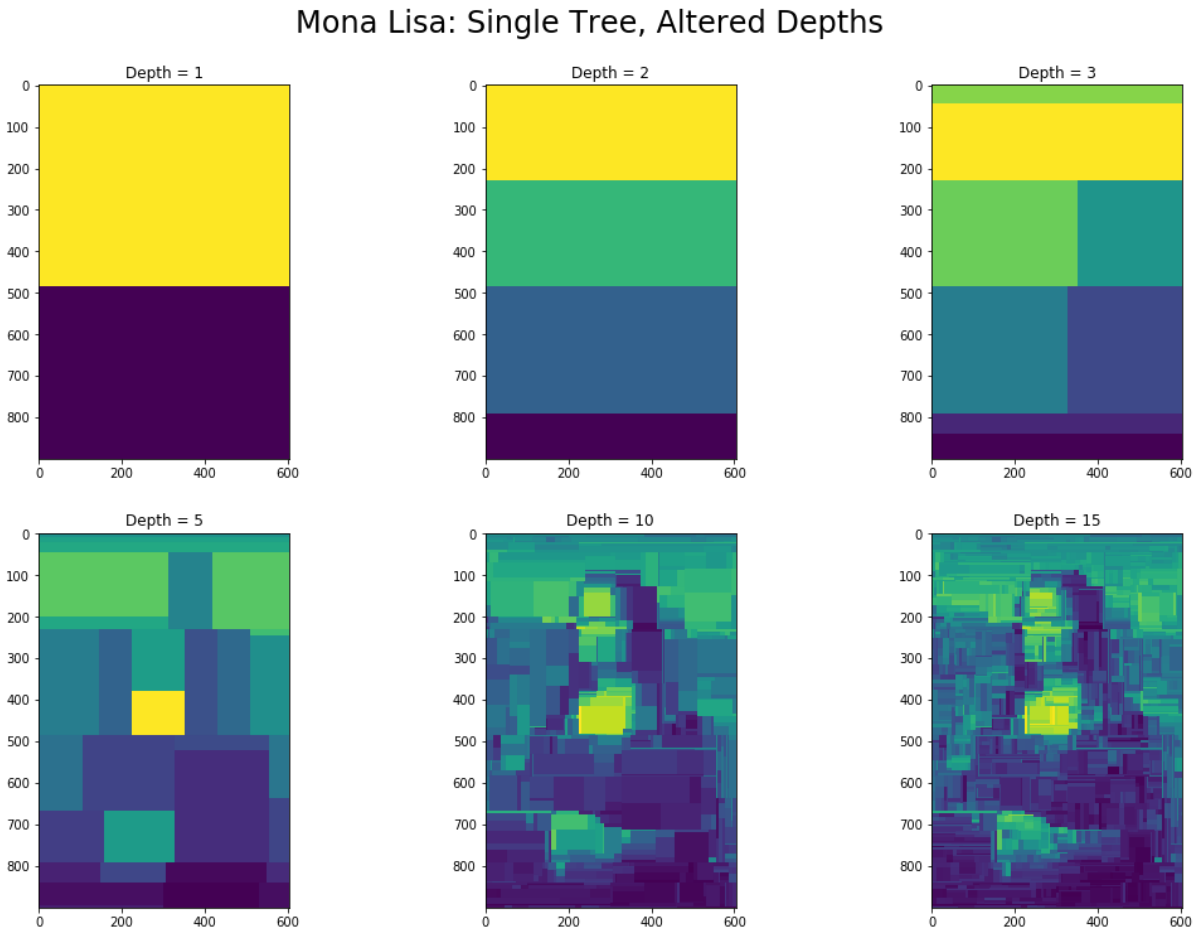


Figure 8: Mona Lisa: Altered Tree Depth

Managing the depth of a tree can be viewed as similar to managing the bias-variance trade off. As the tree becomes deeper, it becomes more and more complex, meaning the variance of the model increases and with it comes overfitting. Like we have studied, there is an optimal balance where total error is at a minimum. What that means is that some depth is of course desired, but too much depth can lead to overfitting and not enough depth will lead to underfitting. What we can see in this example is that as we increase the depth of the tree, it isn't until we reach a depth level of 10 that we can even make out a rough shape of the image. At depth 15, we get even better and more defined detail. This means that even at depth 15, we have not yet overfit the model. What we can see (as we did additional checks with depth 20, 50, 100, and 200) is that past a depth of 20, the image prediction doesn't really improve and we are simply overfitting the model at that point. Because there is too much depth and variance, we cannot accurately predict new test data well. To fix this, we can introduce the concept of bagging and add more trees to the forest (discussed further below).

ii. Repeat the experiment for a random forest of depth 7, but with number of trees equal to 1, 3, 5, 10, and 100. How does the number of trees impact the result? Describe in detail why.

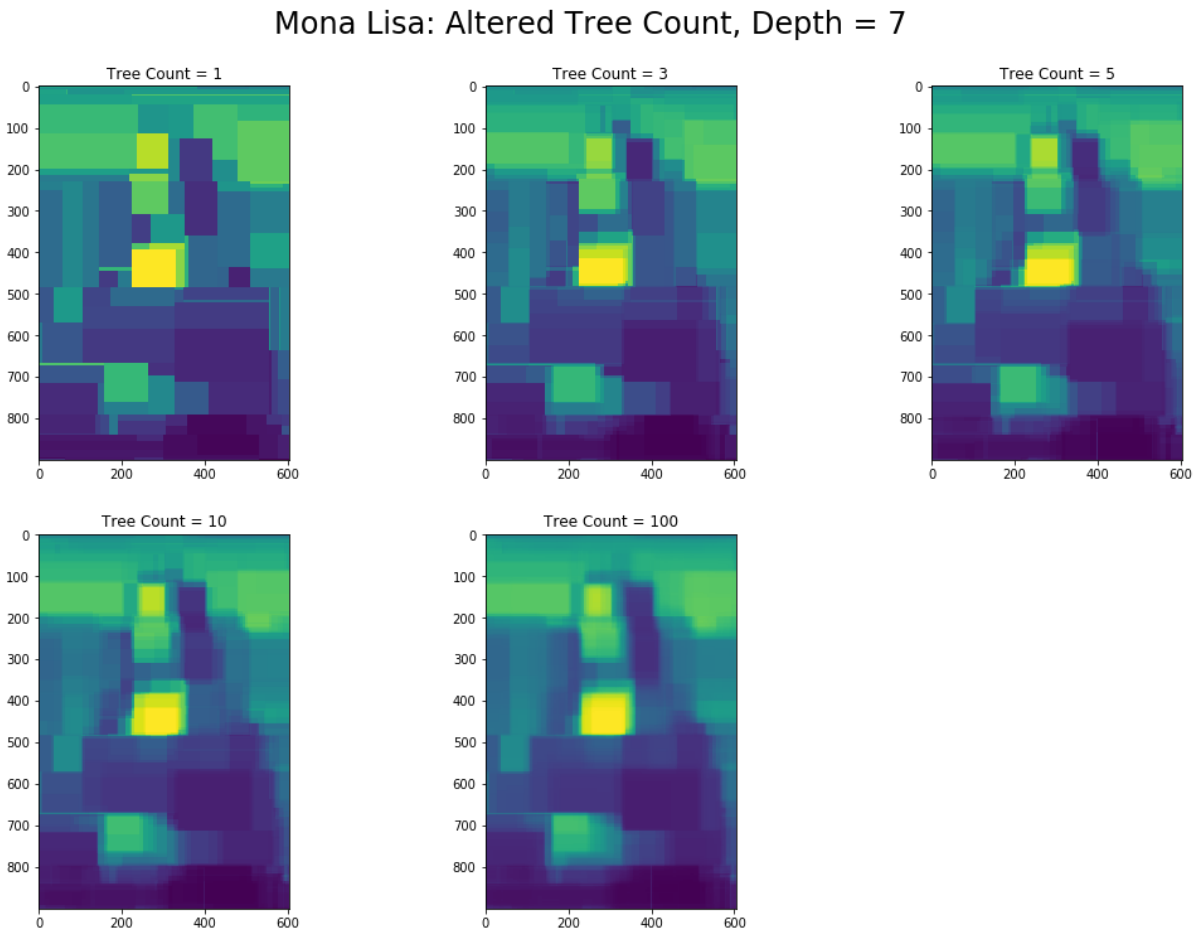


Figure 9: Mona Lisa: Altered Tree Count

While the depth of the tree will influence how much variance the model has and how overfit it is, the number of trees in a forest allows us to harness the power of bagging. Bagging allows us to sample features from the training data at random and with replacement. By using bagging, we are able to reduce the variance error in the model. As the towardsdatascience article (linked below) states, "bagging uses the insight that a suitably large number of uncorrelated errors average out to zero. Bagging chooses multiple random samples of observations from the training data, with replacement, constructing a tree from each one. Since each tree learns from different data, they are fairly uncorrelated from one another."

Furthermore, while adding trees to the Random Forest model will reduce variance error and improve performance of the model, there are limitations and trade offs to adding trees, namely that adding trees makes running the model more computationally intensive, and after a certain point we experience diminishing returns to model performance. Lastly, it is worth noting that adding trees will reduce variance, but it will not reduce the bias in the underlying data.

Link: https://towardsdatascience.com/random-forests-and-the-bias-variance-tradeoff-3b77fee339b4

iii. As a simple baseline, repeat the experiment using a k-NN regressor, for k = 1. This means that every pixel in the output will equal the nearest pixel from the "training set." Compare and contrast the outlook: why does this look the way it does?

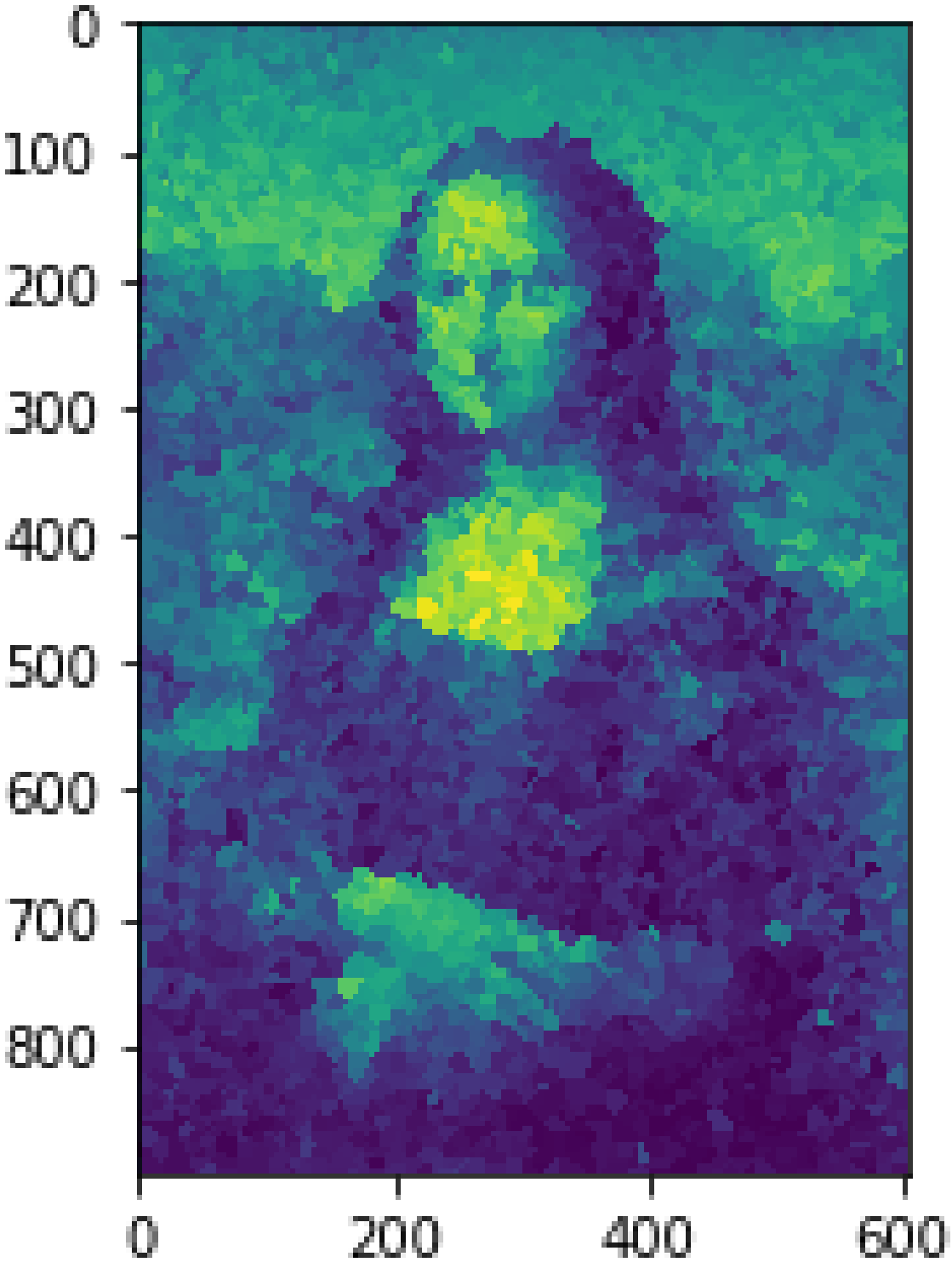


Figure 10: Mona Lisa: kNN Regressor Output

Using a kNN regressor results in an output more representative of the original image with better approximation. As a further step in our analysis, which will also be addressed in part F, we explored why this is the case in the bottom portion of our Jupyter notebook. Several takeaways from the analysis were as follows:

1. Testing with 1 tree and altered depths, we saw that depending on the depth of the tree, the resulting number of leaf nodes was the possible number of different pixel values to output. With 1 tree, this is directly proportional to 2^n where n is the depth. Therefore, with a depth of 6, we could have 64 possible leaf nodes or pixel values to output for any given x,y coordinate.

2. When using a kNN regressor, we do not limit the number of possible decision splits or possible pixel values. Rather, it is a byproduct of how many different pixel values we draw from the random 5,000 draw of training sample x,y coordinates. In our case, we drew 4,507 random pixel values. Ultimately, we had a noticeably larger pool of pixel values to choose from with kNN, and thus were outputting a better defined image.

3. Once we added more trees to the mix (10 trees for examples), the number of possible pixel values jumped up drastically. Possible values exceeded 10,000, more than what we had with kNN – so, why wasn't the image clearer?

4. To better test this, we also evaluated the variance in the pixel values in kNN and the Random Forest. What we found was that adding trees reduced variance, as expected, and that while there were noticeably more unique pixel values, many of them were extremly close to one another, enough to say that we weren't really adding more defined pixel values. And so, with kNN there was actually more variance in the pixel values we could use to approximate the image, and for that reason we see a better result with kNN.

iv. (Optional) Experiment with different pruning strategies of your choice.

(f) Analysis.

i. What is the decision rule at each split point? Write down the 1-line formula for the split point at the root node for one the trained decision trees inside the forest. Feel free to define any variables you need.

Generally speaking the decision rule at each split point in a tree would look similar to the following:

$$\begin{cases} leftnode & x < threshold \\ rightnode & x \geq threshold \end{cases}$$

For this exercise, we knew that we could use the 'estimators_' attribute from the RandomForestRegressor object to identify the decision rules at each split point. In order to more effectively extract and evaluate the data, we sought out guidance on stackoverflow where we found a really helpful function defined in a post. We leveraged that and cited it in our Jupyter Notebook. To test the decision rule split, we fit the data with a RandomForestRegresor with two trees each of depth five.

The formula for the split point at the root node of the first tree was extracted from the Jupyter notebook and the output is listed below:

TREE: 0
0 NODE: if feature[1] $<$ 476.5 then next=1 else next=32

*The syntax above is specific to the function and can be interpreted as: if the feature[1], which is the y coordinate, is less than 484.5 (the threshold for the decision split), then go to the next node (in this case node 1); otherwise, go to node 32.

ii. Why does the resulting image look like the way it does? What shape are the patches of color, and how are they arranged?

The resulting images look the way they do for several reasons. To better explain why, we will analyze the image when the Random Forest is comprised of one tree with altering depths and when the Random Forest has several trees with a fixed depth.

As we saw in experimentation part i. in part e., all we could see are rectangles with different colors when using depths of values 1, 2, and 3. As we explained, the possible pixel values would equal 2^n, where n is depth. At depth 1, we see two colors, at depth 2 we see 4 colors, and at depth 3 we see 8 colors. The images look as they do because the depth influences how many possible pixel values can be used, which will affect the definition and clarity of the image. The shapes/colors are arranged to correspond more or less to the vertical and then horizontal color layout in the image. To clarify, with depth 1 we see yellow and then purple, as the top of the image is brighter where her face is. Similarly for depth 2, we see yellow and then green in the top half, and then in the bottom half we see blue and then purple, which better correspond to the lower portions of the image where her dress is, which is darker. Ultimately, the patches are rectangular and will take a pixel value corresponding to one of the values defined by the decision trees, so the more depth and number of trees there are, the more unique pixel values are available to use and depict the image.

iii. Straightforward: How many patches of color may be in the resulting image if the forest contains a single decision tree? Define any variables you need and provide an upper bound.

As we noted above, if there is a single decision tree, the patches of color will correspond to how many leaf nodes the tree has. This will be limited by the tree depth. The number of color patches possible will be proportional to 2^x where x is the depth of the tree. Depth = 1,2,3,4,5,6 corresponds to an upper bound of 2,4,8,16,32,64 color patches respectively.

iv. Tricky: How many patches of color might be in the resulting image if the forest contains n decision trees? Define any variables you need and provide an upper bound.

In this question, it is important to realize that each tree will randomly sample different features. As a result, each tree will be different from the next and have different decision split rules, and thus different color patches defined as possible outputs. Building on the above, for each tree there is a limitation of 2^x leaf nodes. We must then apply the same limitations to each tree in combination when forming the forest.

Therefore, the upper bound on the number of patches of color if the forest contains n decision trees is 2^{xn} where x is the tree depth and n is the number of trees in the forest.

0.4 Problem 3: SVM Classification

This problem involves the OJ data set.

(a) Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations. Report the class fractions in your training and test sets.

Training class fractions:
CH = 494
MM = 306
CH fraction = $\frac{494}{800} = 0.6175$
MM fraction = $\frac{306}{800} = 0.3825$

Test class fractions:
CH = 159
MM = 111
CH fraction = $\frac{159}{270} = 0.588$
MM fraction = $\frac{111}{270} = 0.411$

(b) Fit a support vector classifier to the training data using C = 0.01, with Purchase as the response and the other variables as predictors. Describe the classifier. What are the learned coefficients? (You can use sklearn to help with this.)

The classifier uses the SVC implementation from sklearn.svm. The classifier uses a C of 0.01 with a linear kernel.

The Learned Coefficients for each of the features is listed below:

Column Header	Coefficient
WeekofPurchase	-0.0099
StoreID	-0.1276
PriceCH	0.0489
PriceMM	-0.0780
DiscCH	0.0036
DiscMM	0.1524
SpecialCH	-0.0628
SpecialMM	0.1856
LoyalCH	-1.0212
SalePriceMM	-0.2304
SalePriceCH	0.0454
PriceDiff	-0.2758
Store7	-0.0145
PctDiscMM	0.0704
PctDiscCH	0.0016
ListPriceDiff	-0.1270
STORE	-0.0262

Figure 11: Learned Coefficients

(c) What are the training and test error rates?

In order to determine the training and test error rates, we first split the data as the instructions noted above. We then used a linear kernel with C = 0.01 to fit the training data. We then predicted the outputs for the test data. Once we had fit a classifier on the training data, we ran the score function on the training and test data to output the accuracy scores below. The error rates would be 1 - accuracy score. The Linear Kernel Results are:

Training Accuracy Score: 0.7600
Test Accuracy Score: 0.7222

(d) Use cross validation to select an optimal cost (C) and report a CV error plot. Consider values in the range 0.01 to 10. What C is the best using the one-stderr rule?

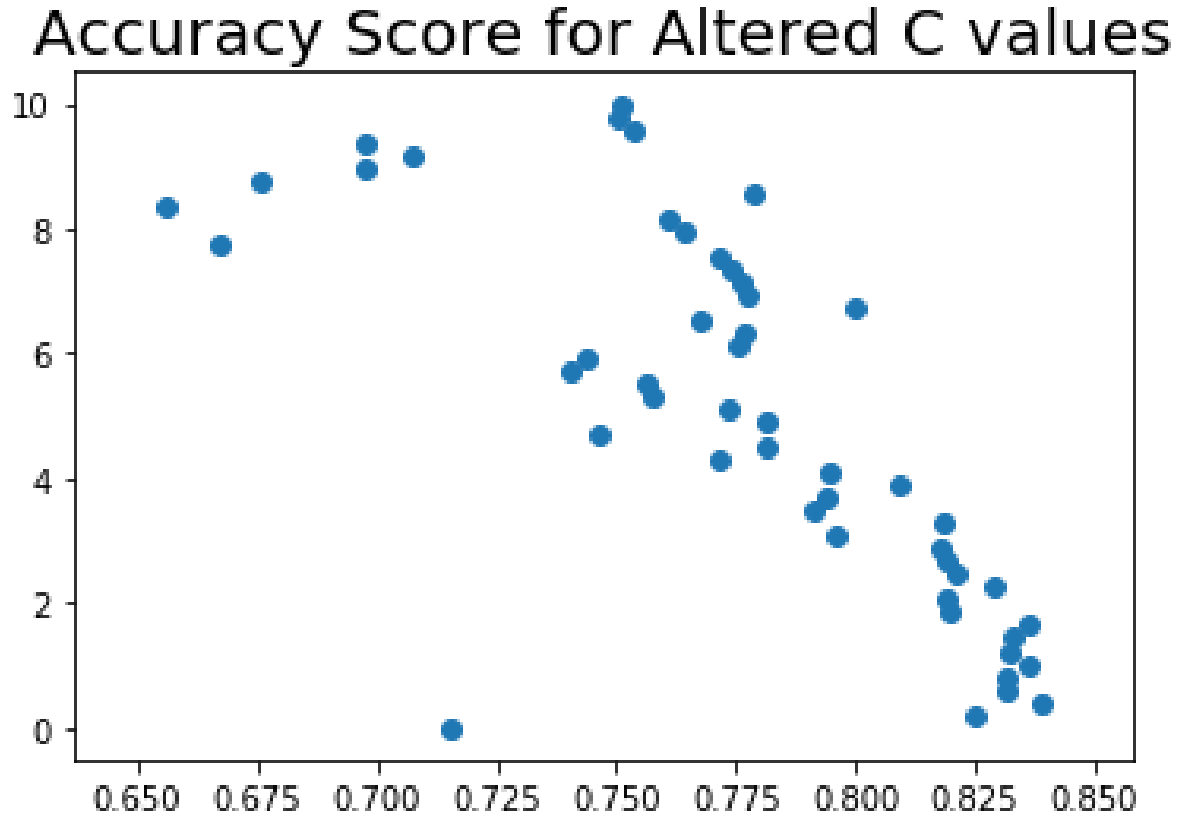


Figure 12: CV Error Plot

Using cross validation, we get the CV error plot above. The optimal C value that we determined was 0.4178.

(e) Compute the training and test error rates using this new value for cost.

Using the same process as in part (c), we use the C value we determined through cross validation. The Linear Kernel results with the cross-validated C (C = 0.4178) are:

Training Accuracy Score: 0.8363
Test Accuracy Score: 0.8037

(f) For a support vector classifier with a radial basis function (RBF) kernel $k(x,x') = e^{-\gamma||x-x'||^2}$, why should we consider a smaller γ value as corresponding to a simpler model? (Note that sometimes we write the RBF kernel as $k(x,x') = e^{-||x-x'||^2/\sigma^2}$ using $\sigma = 1/\sqrt{\gamma}$.)

As Scikit-Learn's RBF SVM Parameters Documentation page states, "The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors." Mathematically, we can present this as:

$$\gamma = \frac{1}{\text{radius of influence}}$$

We can rearrange this equation to present it as follows:

$$\text{radius of influence} = \frac{1}{\gamma}$$

From this equation, we can see that as gamma increases, the radius of influence decreases. If gamma is sufficiently large, the radius of influence essentially becomes 0, meaning the radius of influence of the support vectors only captures the support vectors themselves, leading to overfitting. As we have learned, overfitting a model is analogous to increasing the complexity of the model. As a result, a sufficiently small gamma will have the opposite effect on the model as it will underfit the model and be very simple.

The graph displayed below helps highlight this relationship between overfitting and increasing complexity.

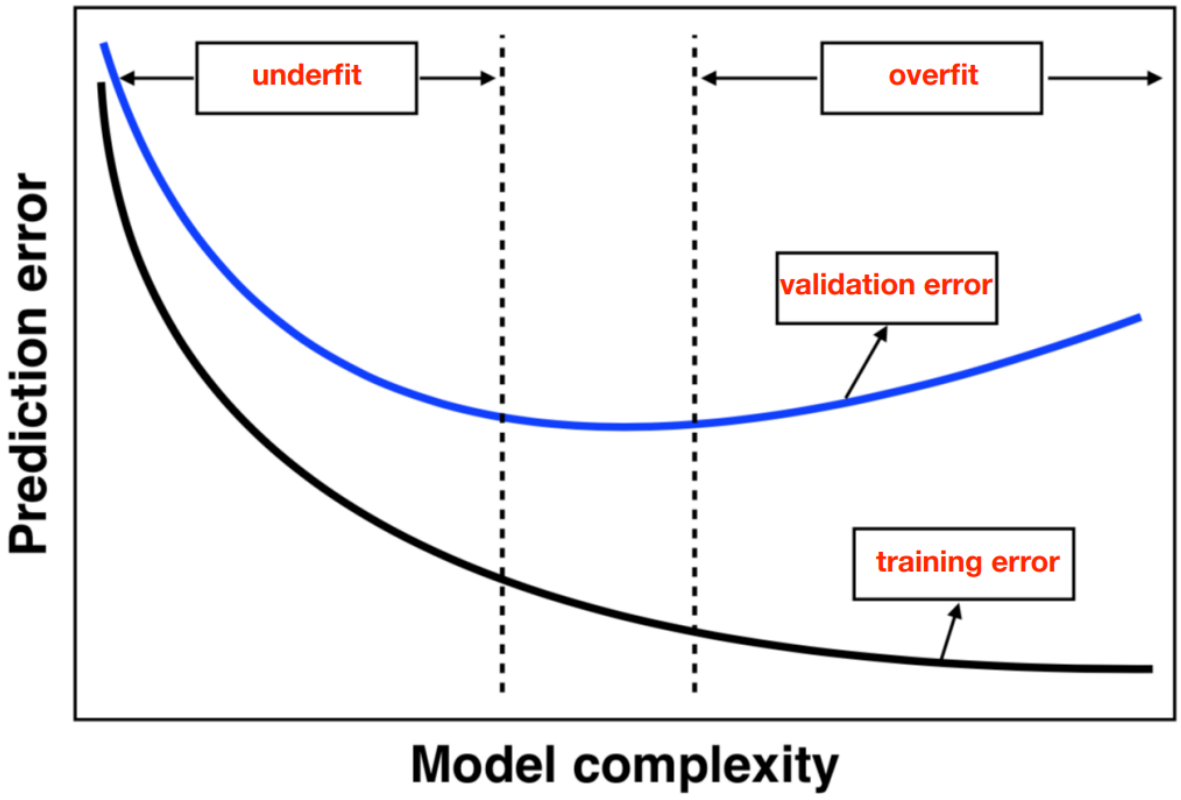


Figure 13: Model Complexity

Furthermore, as Sklearn's documentation notes: "When gamma is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model." Thus, we can conclude that a smaller γ value corresponds to a simpler model.

Link: https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

(g) Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use cross validation (with the one-standard-error rule) to select an appropriate gamma.

Similarly to the above process, we used cross validation to find the optimal parameters to run SVC with the RBF kernel. The one added layer here is that we created a grid of C and γ values to be tested together to find the optimal combination to use. We tested 30 different C values with each of 30 different γ values. The RBF Kernel Results with Cross-validated C and Gamma values are:

Best C value from cross-validation: 9.6555
Best Gamma value from cross-validation: 0.0100
Training Accuracy Score: 0.8400
Test Accuracy Score: 0.7926

(h) Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set degree=2 and use cross validation again to choose an appropriate gamma.

Similarly to the RBF kernel, we used the C and γ grid to cross validate the optimal combination of parameter values to run the SVC with. The Polynomial Kernel results with the cross-validated C and γ are:

Best C value from cross-validation: 8.6221
Best Gamma value from cross-validation: 7.2441
Training Accuracy Score: 0.6887
Test Accuracy Score: 0.7000

(i) Overall, which approach seems to give the best results on this data?

Overall, with the cross-validated scores, it is clear that the polynomial kernel performed the worst of the three. The Radial and Linear kernels performed similarly, so it is hard to say which approach gives the best results on this data. While the radial kernel had a slightly higher training accuracy, the linear kernel had a slightly higher test accuracy. Furthermore, while we cross validated more combinations of C and γ (30 x 30) in the radial case, we did use less C values than the 50 we used with the linear kernel, so it is possible we could have found a more optimal combination for the radial kernel. However, even if we did, the increases would be small (based on the results we saw) and it would not result in finding a radial kernel that performs significantly better than the linear kernel for this data. Therefore, both the linear and radial kernels are preferred to the polynomial kernel on this data.

0.5 Problem 4: Approximating images with neural networks

(a) Describe the structure of the network. How many layers does this network have? What are the sizes of the layers? What activation function(s) are being used?

The network has 9 layers that get pushed on as layers in the code. The first layer is the input layer and the last layer is the loss output layer. The seven layers in between are the fully connected hidden layers. The 7 hidden layers all have a size of 20 neurons. The input layer has a size of 2 neurons which represent the x and y position values of a given pixel. The output layer has a size of 3 neurons which represent the R, G, and B values of the pixel at that position (meaning it predicts the color of the pixel at that x and y position). As can be seen in the code, the activation function being used in the hidden layers is ReLu.

(b) What does "Loss" mean here? What is the actual loss function? You may need to consult the source code (https://cs.stanford.edu/people/karpathy/convnetjs/docs.html).

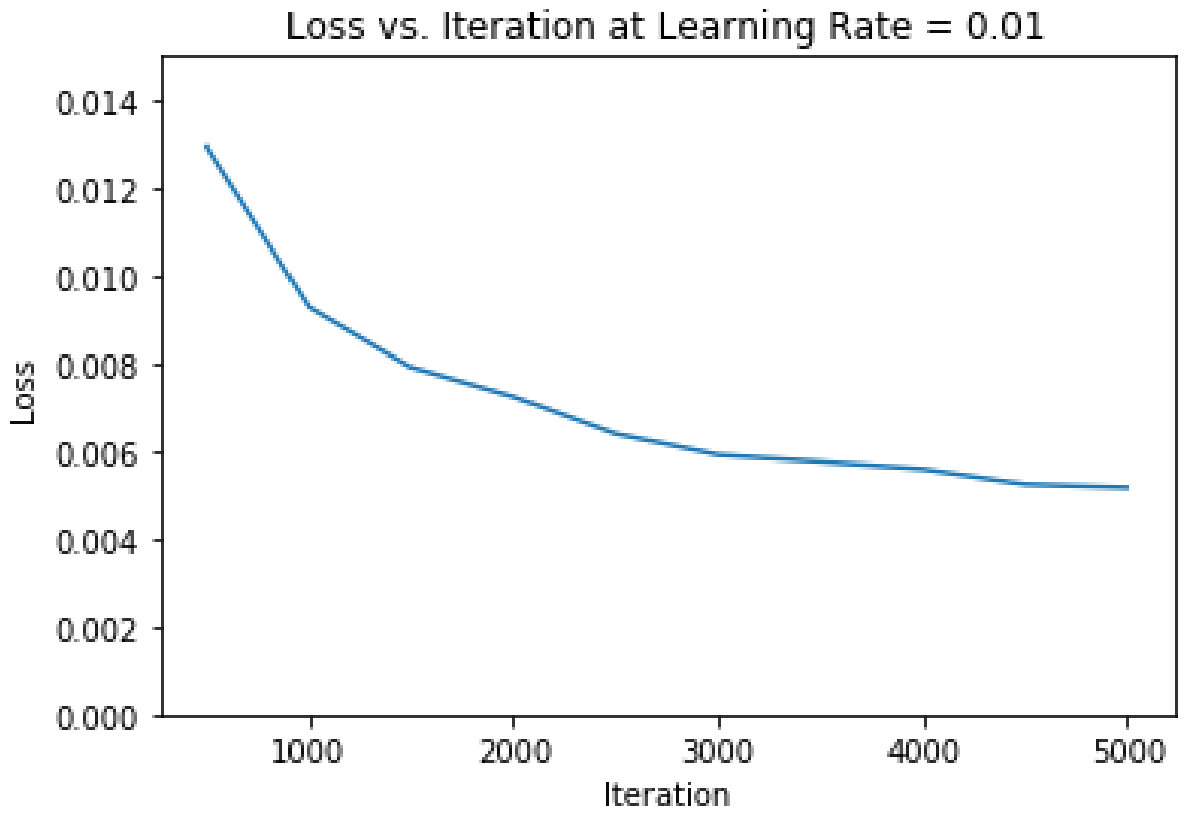
In the context of this exercise and approximating images with neural networks, "Loss" means the difference between the predicted image that is output and the original image.

To determine the loss function, we studied the code and documentation sections and noted the following:

- In the code itself, the line below tells us the output values from the neural network are determined based on regression:
  - layer_defs.push(type:'regression', num_neurons:3);

- We then searched through the documentation to find the following: "Loss layers: L2 Regression Layer...Create a regression layer which takes a list of targets and backprops the L2 Loss." As a result, we can determine that the loss function incorporated into the design of this neural network for regression problems is L2 loss, and since the ouput layer uses regression, we have L2 loss.

(c) Plot the loss over time, after letting it run for 5,000 iterations. You can do this by watching the training and manually writing down the loss every 500 or so iterations. How good does the network eventually get?



As can be seen, the loss value begins by dropping dramatically as it iterates. However, it eventually slows down around the 3000th iteration. Between 3000 and 5000 iterations, the rate of decreasing loss has significantly slowed down, with it eventually reaching .0052 loss at iteration = 5000. We allowed the iterations to continue running until the 10000th iteration, where it reached a loss value of 0.004425. This confirmed for us that the loss would and does continue to slightly decrease, but at a very slow and decreasing rate.

(d) Can you make the network converge to a lower loss function by lowering the learning rate every 1,000 iterations? Some learning rate schedules, for example, halve the learning rate every n iterations: does this technique let the network converge to a lower training loss?

The slider was a bit hard to control, but we changed the learning rate to the following values at each of the following iterations: iteration 1000: 0.00708, iteration 2000: 0.00501, iteration 3000: 0.00224, iteration 4000: 0.00112. Ultimately, at the 5000th iteration, the loss value had converged to a value of 0.003278. Not only is this loss value much lower than the value we reached at the 5000th iteration when we didn't alter the learning rate, but it is also lower than the value we reached at the 10000th iteration when we didn't alter the learning rate as well. This helps highlight that we certainly can lower the value the loss function reaches by lowering the learning rate every n iterations.

(e) Lesion study. The text box contains a small snippet of Javascript code that initializes the network. You can change the network structure by clicking the "Reload network" button, which simply evaluates the code. Let's perform some brain surgery: Try commenting out each layer, one by one. Report some results: How many layers can you drop before the accuracy drops below a useful value? How few hidden units can you get away with before quality drops noticeably?

We started by just commenting out the hidden layers one by one and measuring the loss value reached at 5000. The results are as follows:

- One hidden layer commented out: 0.005192
- Two hidden layers commented out: 0.005445
- Three hidden layers commented out: 0.005909
- Four hidden layers commented out: 0.007271
- Five hidden layers commented out: 0.010892
- Six hidden layers commented out: 0.026801
- Seven hidden layers commented out: 0.069588

We also recorded what the photo looked like for each of these cases at the 5000th iteration – the results are below:

Figure 14: Photo at 5000th iteration with 1 hidden layer commented out

Figure 15: Photo at 5000th iteration with 2 hidden layers commented out

Figure 16: Photo at 5000th iteration with 3 hidden layers commented out

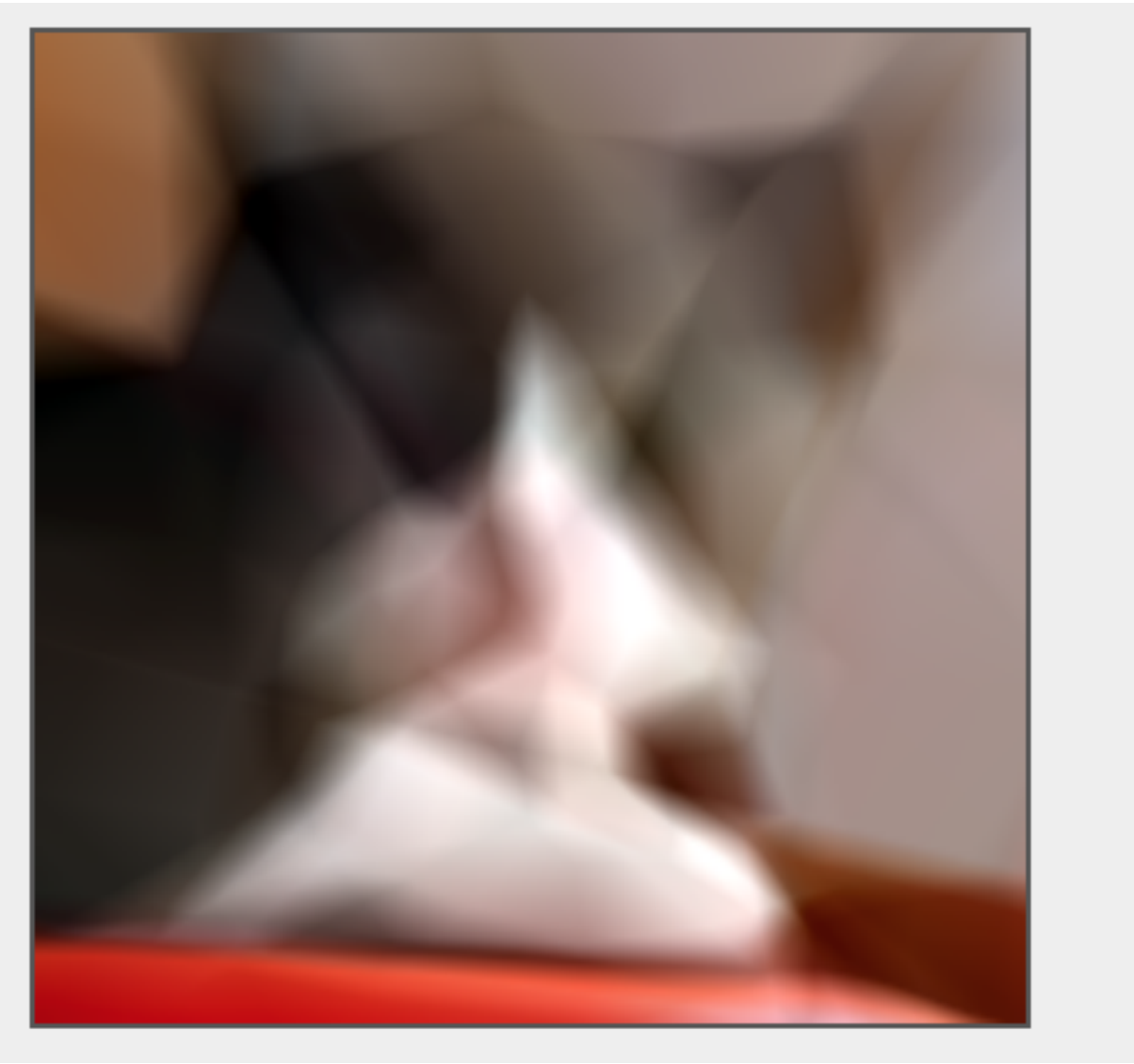


Figure 17: Photo at 5000th iteration with 4 hidden layers commented out

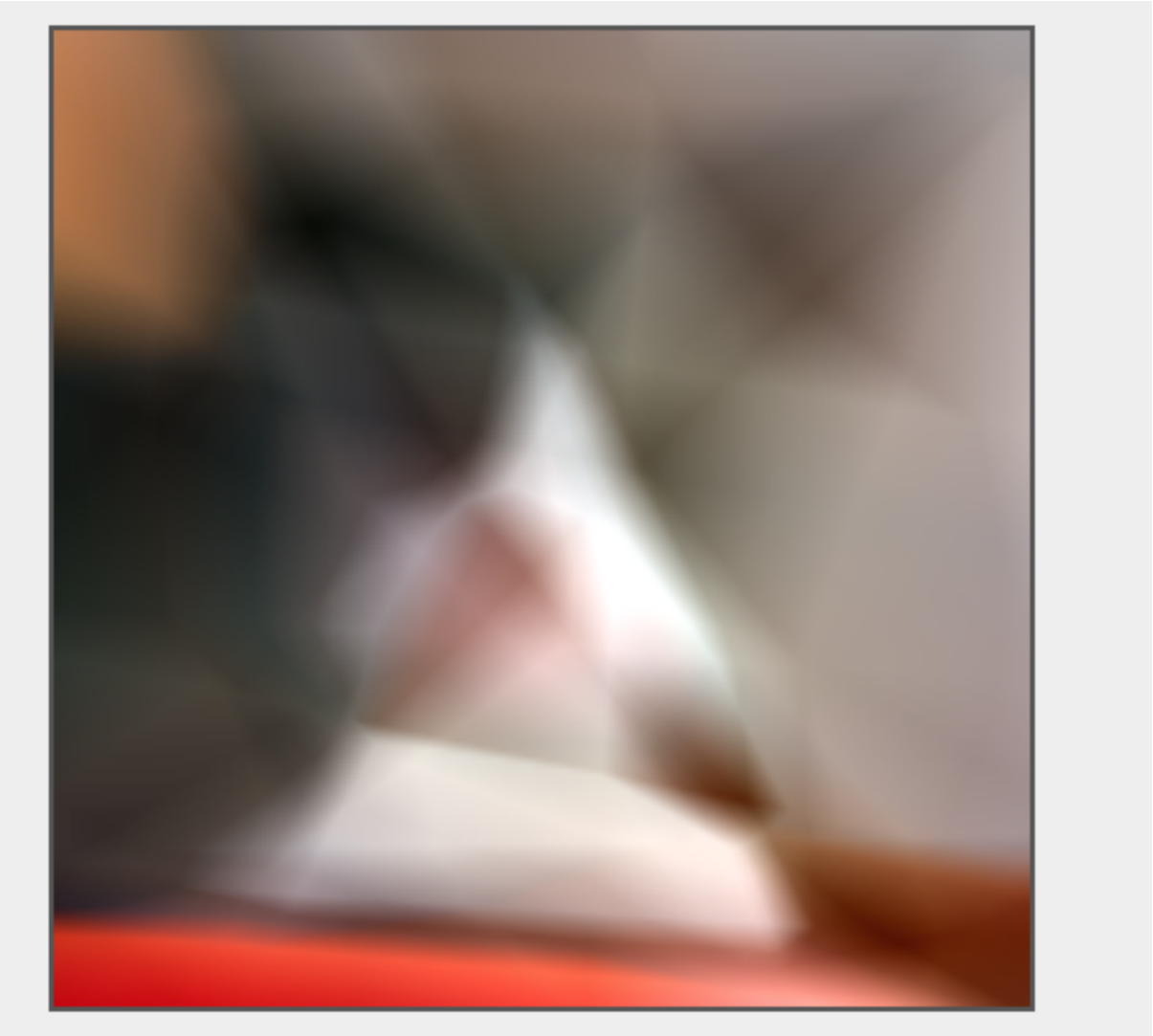


Figure 18: Photo at 5000th iteration with 5 hidden layers commented out

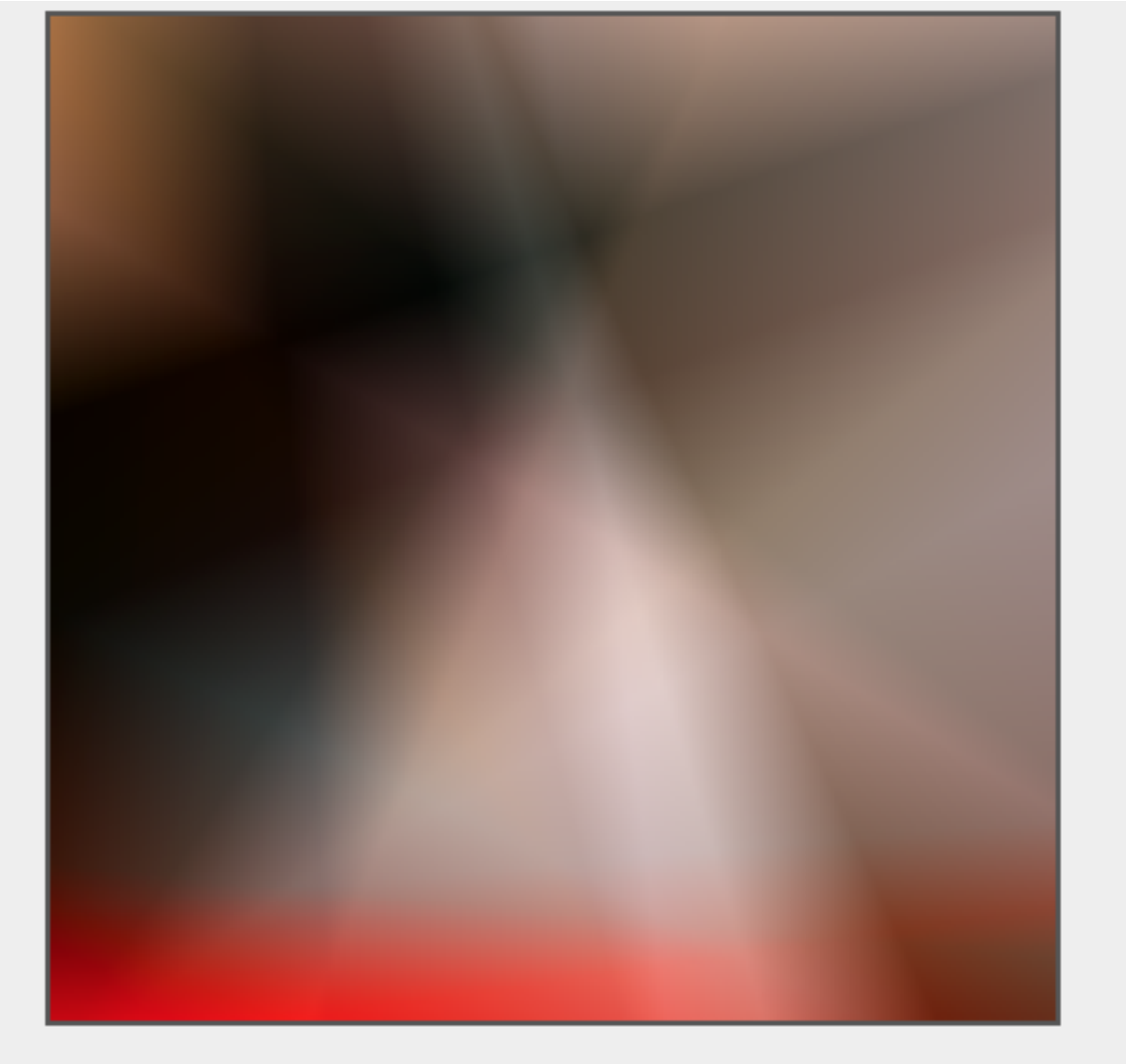


Figure 19: Photo at 5000th iteration with 6 hidden layers commented out

Figure 20: Photo at 5000th iteration with 7 hidden layers commented out

As can be seen, when we comment out the first hidden layer, the loss value stays essentially the same. When we comment out the second and third hidden layers, the loss value gets worse but slightly – the loss value is still acceptable. When we comment out the fourth hidden layer, the loss begins to get significantly worse, dropping from 0.005909 to 0.007271. However, it is still potentially acceptable. However, once we comment out the fifth hidden layer, the value goes from 0.007271 to 0.010892, getting almost 50% worse. By this point, we would argue that the loss value has reached an unacceptable level. Commenting out the sixth layer causes the loss value to go from 0.010892 to 0.026801, which is over a 200% increase in loss. Finally, commenting out the last layer causes the loss to go from 0.026801 to 0.069588, which is again over a 200% loss increase. The loss values are certainly unacceptable at these points as well.

We can also look at the photos that result from each case. With only one layer removed, the photo is identical to what it was with all 7 layers present. In the second and third photos, the quality slightly decreases, but it is still clear that the photo is of a cat. By the fourth photo, the quality has started to drop – you can still tell it is an animal, but the ears are not as pointed, for example, and you might mistake it for a dog instead of a cat. By the fifth photo (5 layers commented), quality has dropped significantly and it is not so clear what is in the photo. The sixth photo (six layers commented out) is even more unclear and blurry, and the last photo (all 7 layers commented out) has removed any appearance of an object – it is just a gradient of black to red colors.

(f) Try adding a few layers by copy+pasting lines in the network definition. Can you noticeably increase the accuracy of the network?

We tried the network out with 10, 14, and 25 layers. The results were as follows:

- 10 layers: 0.005698
- 14 layers: 0.005698
- 25 layers: 0.008070

We also recorded what the photo looked like for each of these cases at the 5000th iteration – the results are below:

Figure 21: Photo at 5000th iteration with 10 hidden layers

Figure 22: Photo at 5000th iteration with 14 hidden layers

Figure 23: Photo at 5000th iteration with 25 hidden layers

The results here are quite interesting. With 10 and 14 layers, we get similar loss values, though they are actually slightly worse than with 7 layers. With 25 layers, our loss value gets significantly worse, going to 0.008070. This leaves us with the impression that we can't noticeably increase the accuracy of the network with more layers and that adding more layers actually made it worse.

By looking at the photos, we see something interesting. Although the 10-layer loss is worse than the 7-layer, the 10-layer photo seems to be a bit sharper and has more defined edges, making it a bit clearer to us that the photo is of a cat. However, the photos end up being quite similar in appearance.

0.6 Problem 5: Written Exercises

1. Decision trees. Suppose we modify the tree-growing algorithm presented in class to use a new impurity function. Define f(r) = $\min\{r, 1-r\}$. Then we will define the impurity of a set of examples as:

$$I(y_1, ..., y_n) = f(p) \tag{1}$$

where p is the fraction of positive examples in $\{y_1, ..., y_n\}$. Let us call this the min-error impurity function.

As usual, for a split where p_1 positive and n_1 negative examples reach the left branch and p_2 positive and n_2 negative examples reach the right branch, the weighted impurity of the split will be:

$$(p_1 + n_1) \cdot f(\frac{p_1}{p_1 + n_1}) + (p_2 + n_2) \cdot f(\frac{p_2}{p_2 + n_2}) \tag{2}$$

(a) Suppose that each branch of this split is replaced by a leaf labeled with the more frequent class among the examples that reach that branch. Show that the number of training mistakes made by this truncated tree is exactly equal to the weighted impurity given above. Thus, using the min-error impurity is equivalent to growing the tree greedily to minimize training error.

> Let's suppose that for a given split, each branch is replaced by the leaf labeled with the more frequent class. Let's say that there are x items in class 1 and y items in class 2. If x is greater than y, we are going to classify all items as x. If we think about this logically, this means that we are correctly classifying x items, because the x items that are in class 1 are all getting labeled as being in class 1. However, we incorrectly classify y items because they are also being classified as being in class 1 even though they are in class 2. The logic here follows that the number of mistakes or items that we incorrectly classify is equivalent to the number of items in the class with the lower item count. Therefore, we can logically say that the fraction of mistakes for a leaf with x items in one class and y items in another class is $\frac{\min\{x,y\}}{x+y}$.

Again, using logic, we can apply this new connection we have identified to the following expression: $f(\frac{p_1}{p_1+n_1})$. As we know, the function f returns the minimum of its argument and 1 minus its argument, and in this case, those two options are $(\frac{p_1}{p_1+n_1})$ and $(\frac{n_1}{p_1+n_1})$. Since we want the smaller of these two expressions, we see that the f function performs the exact same operation as $\frac{\min\{x,y\}}{x+y}$ – it takes the min of the two values (in this case p_1 and n_1) and divides it by the sum of the two values. Therefore, based on our definition above, $f(\frac{p_1}{p_1+n_1})$ is the fraction of mistakes for the first leaf and $f(\frac{p_2}{p_2+n_2})$ is the fraction of mistakes for the second leaf.

Since these are just fractions, we can say that the total number of mistakes for each leaf is these functions multiplied by the number of items in each respective leaf: $(p_1+n_1)\cdot f(\frac{p_1}{p_1+n_1})$ for the first leaf and $(p_2+n_2)\cdot f(\frac{p_2}{p_2+n_2})$ for the second leaf. Therefore, the total number of mistakes for the truncated method is just the sum of these two values: $(p_1+n_1)\cdot f(\frac{p_1}{p_1+n_1}) + (p_2+n_2)\cdot f(\frac{p_2}{p_2+n_2})$. As we can see, this expression that represents the total number of training mistakes is perfectly equivalent to the weighted impurity given above.

(b) Suppose the dataset looks like the following. There are three $\{0,1\}$-valued attributes, and one $\{-,+\}$-valued class label y.

a_1	a_2	a_3	y
0	0	0	+
1	1	0	+
0	1	0	+
1	0	1	-
0	0	1	-
0	1	0	-
1	1	0	-
1	1	1	-
1	0	0	-
1	1	0	-

Which split will be chosen at the root when the Gini index impurity function is used? Which split will be chosen at the root when min-error impurity is used? Explain your answers.

In order to determine which split will be chosen at the root, we need to calculate the impurity resulting from choosing each attribute as the root node. The calculations for the Gini index impurity for each of the attributes is listed below. The general formula we use for calculating the Gini Impurity is as follows:

Gini Impurity for left/right branch = 1 - $(\frac{\text{pos}}{\text{total}})^2$ - $(\frac{\text{neg}}{\text{total}})^2$

Accounting for the differing counts in each of the branches, we need to weigh the impurities accordingly.

Weighted impurity = $(\frac{\text{left count}}{\text{total}})$ * (left impurity) + $(\frac{\text{right count}}{\text{total}})$ * (right impurity)

a_1 counts

- 0 and + = 2
- 0 and - = 2
- 1 and + = 1
- 1 and - = 5

Gini Impurity for 0 branch = 1 - $(\frac{2}{4})^2$ - $(\frac{2}{4})^2$ = 0.5

Gini Impurity for 1 branch = 1 - $(\frac{1}{6})^2$ - $(\frac{5}{6})^2$ = 0.277

Weighted impurity for a_1 = $(\frac{4}{10})$ *(0.5) + $(\frac{6}{10})$ * (.277) = 0.367

a_2 counts

- 0 and + = 1
- 0 and - = 3
- 1 and + = 2
- 1 and - = 4

Gini Impurity for 0 branch = 1 - $(\frac{1}{4})^2$ - $(\frac{3}{4})^2$ = 0.375

Gini Impurity for 1 branch = 1 - $(\frac{2}{6})^2$ - $(\frac{4}{6})^2$ = 0.444

Weighted impurity for a_2 = $(\frac{4}{10})$ *(0.375) + $(\frac{6}{10})$ * (0.444) = 0.4167

a_3 counts

- 0 and + = 3
- 0 and - = 4
- 1 and + = 0
- 1 and - = 3

Gini Impurity for 0 branch = 1 - $(\frac{3}{7})^2$ - $(\frac{4}{7})^2$ = 0.4898

Gini Impurity for 1 branch = 1 - $(\frac{0}{3})^2$ - $(\frac{3}{3})^2$ = 0.0

Weighted impurity for a_3 = $(\frac{7}{10})$ *(0.4898) + $(\frac{3}{10})$ * (0.0) = 0.343

We can now see that attribute a_3 has the smallest weighted impurity when using the gini index; therefore, a_3 would be chosen as the splitting attribute in the root node.

We will now calculate which split will be chosen at the root when min-error impurity is used. As we know, the formula for min-error impurity is as follows:

$(p_1 + n_1) \cdot f(\frac{p_1}{p_1+n_1}) + (p_2 + n_2) \cdot f(\frac{p_2}{p_2+n_2})$

The calculations for each of the three attributes is below (using the values defined above):

a_1

$= (2+2) \cdot f(\frac{2}{2+2}) + (1+5) \cdot f(\frac{1}{1+5})$

$= 4^* \frac{1}{2} + 6 * \frac{1}{6} = 3$

a_2

$= (1+3) \cdot f(\frac{1}{1+3}) + (2+4) \cdot f(\frac{2}{2+4})$

$= 4^* \frac{1}{4} + 6 * \frac{2}{6} = 3$

a_3
$= (3+4) \cdot f(\frac{3}{3+4}) + (0+3) \cdot f(\frac{0}{0+3})$
$= 7^* \frac{3}{7} + 3 * \frac{0}{3} = 3$

As we can see, the min-error impurity metric results in all three attributes having the same impurity, such that there is no one particular attribute that would be best to use for splitting at the root node.

(c) Under what general conditions on p_1, n_1, p_2, and n_2 will the weighted min-error impurity of the split be strictly smaller than the min-error impurity before making the split (i.e., of all the examples taken together)?

As we know, the weighted min error impurity of the split is represented as:

$(p_1 + n_1) \cdot f(\frac{p_1}{p_1+n_1}) + (p_2 + n_2) \cdot f(\frac{p_2}{p_2+n_2})$

To represent the min-error impurity before making the split, we first note that our total pool of data is now $(p_1 + p_2 + n_1 + n_2)$, as this is the full data before it is split. We also know that the impurity function (fraction of positive data) f(p) for the un-split data is f($\frac{p_1+p_2}{p_1+p_2+n_1+n_2}$), as p_1 and p_2 are both part of the un-split data. Therefore, the min-error impurity before making the split is:

$(p_1 + p_2 + n_1 + n_2) \cdot$ f($\frac{p_1+p_2}{p_1+p_2+n_1+n_2}$)

And therefore, since we are interested in the case where the split impurity is less than the impurity before being split, we have the inequality:

$(p_1 + n_1) \cdot f(\frac{p_1}{p_1+n_1}) + (p_2 + n_2) \cdot f(\frac{p_2}{p_2+n_2}) < (p_1 + p_2 + n_1 + n_2) \cdot$ f($\frac{p_1+p_2}{p_1+p_2+n_1+n_2}$)

However, we can notice that the denominators in the f functions will be constant regardless; the only thing that will change is the numerator based on the minimum value. Therefore, we can simplify this expression by canceling out the denominators and noting that the numerator will be the minimum of two values:

$$\min\{p_1, n_1\} + \min\{p_2, n_2\} < \min\{(p_1 + p_2), (n_1 + n_2)\}$$

In this form, it is much more clear what general conditions on p_1, p_2, n_1, and n_2 would cause the inequality above to hold. The conditions are as follows:

(1) $p_1 < n_1$ and $n_2 < p_2$
(2) $n_1 < p_1$ and $p_2 < n_2$

These alternating cases hold and we can show so mathematically. Using the expression from above, $\min\{p_1, n_1\} + \min\{p_2, n_2\} < \min\{(p_1 + p_2), (n_1 + n_2)\}$, we will end up with the following two possibilities for case (1) and can cancel terms:

$p_1 + n_2 < p_1 + p_2$, which simplifies to $n_2 < p_2$
$p_1 + n_2 < n_1 + n_2$, which simplifies to $p_1 < n_1$

For case (1), we see that the two expressions simplify to the following inequalities: $n_2 < p_2$ and $p_1 < n_1$. And as we already defined, these two inequalities both hold for case (1), so case (1) is a condition that will cause our overall inequality (split impurity < unsplit impurity) to hold. We follow the same process for case (2):

$n_1 + p_2 < p_1 + p_2$, which simplifies to $n_1 < p_1$
$n_1 + p_2 < n_1 + n_2$, which simplifies to $p_2 < n_2$

Again, we see that the two expressions simplify to the following inequalities: $n_1 < p_1$ and $p_2 < n_2$. And as we already defined, these two inequalities both hold for case (2), so case (2) is a condition that will also cause our overall inequality (split impurity < unsplit impurity) to hold.

(d) What do your answers to the last two parts suggest about the suitability of min-error impurity for growing decision trees?

Part (b) suggests that using min-error impurity for growing decision trees is not a very suitable metric. By just inspecting the data, we can see right away that a_3 is the best feature to split on as one of its leafs has homogeneous values, meaning it has 0 impurity. However, our impurity calculations returned the same value of 3 for a_1, a_2, and a_3, meaning it wasn't able to provide a conclusive answer even when we could identify the right answer just by inspection. This is a good indication that this is a poor impurity function to base our splitting on, as it is not very successful.

Part (c) suggests that using min-error impurity is again not ideal because there are only certain conditions where we are actually guaranteed to reduce our impurity when splitting. Ultimately, we do not gain anything from performing a split if the impurity is not reduced. And with this impurity function, we are only guaranteed to reduce impurity under certain conditions, which is not ideal. We would prefer an impurity function that does not require certain conditions to be true when splitting in order for impurity to actually be reduced.

Therefore, the min-error impurity function is not a suitable one, and we found in part (b) that using the gini index helped us perform splits more effectively.

2. Bootstrap aggregation ("bagging"). Suppose we have a training set of N examples, and we use bagging to create a bootstrap replicate by drawing N samples with replacement to form a new training set. Because each sample is drawn with replacement, some examples may be included in this bootstrap replicate multiple times, and some examples will be omitted from it entirely. As a function of N, compute the expected fraction of the training set that does not appear at all in the bootstrap replicate. What is the limit of this expectation as $N \to \infty$?

Hint 1: You can express the number of examples not chosen by bootstrapping as the sum of n random variables X_i, where each of these variables equals 1 if the i'th sample was never chosen, or 0 otherwise.

Hint 2: Recall that expectation is linear.

Hint 3: You may want to use the fact that $\lim_{n\to\infty}(1 + x/n)^n = e^x$.

If we have N samples to choose from for any given iteration, the probability of choosing a specific value X_i is $\frac{1}{N}$. Therefore, the probability of not choosing that specific value on a given iteration is the complement, 1 - $\frac{1}{N}$. Since we will be choosing a sample with replacement N times, the probability of not choosing a given sample over those N samplings will be the probability of not choosing that value over one iteration multiplied together N times. In other words, it is $(1-\frac{1}{N})^N$. Since this is the probability of not choosing a given value over N samplings, we can logically extend this to say that the expected fraction of the training set that does not appear at all in the bootstrap replicate is equivalent to this probability.

The limit of this expectation can be calculated using the Hint 3 provided, which is that $\lim_{N\to\infty}(1+x/N)^N = e^x$. By rearranging our expectation from $(1-\frac{1}{N})^N$ to $(1+\frac{-1}{N})^N$, we now have our expectation in the same form as the expression in the limit in hint 3. By taking the limit of our expectation, we now have $\lim_{N\to\infty}(1+\frac{-1}{N})^N$. We know that this limit is equivalent to e^x and in our case/form, we can see that x = -1, as all other parts of the expression match the expression in hint 3. Therefore we can say that $\lim_{N\to\infty}(1+\frac{-1}{N})^N = e^{-1}$. Through a simple calculation, we can say that as we approach N = ∞, the expected fraction of the training set that does not appear in the bootstrap replicate is 0.368.

3. Maximum-Margin classifiers. Suppose we are given n = 7 observations in p = 2 dimensions. For each observation, there is an associated class label.

X_1	X_2	Y
3	4	Red
2	2	Red
4	4	Red
1	4	Red
2	1	Blue
4	3	Blue
4	1	Blue

(a) Sketch the observations and the maximum-margin separating hyperplane.

To start the process, we first plotted all of the points on a plot using python. We made an initial guess that (2,2) would be the red point closest to the hyperplane and (4,3) would be the blue point closest to the hyperplane. We then guessed two lines with slope 0.9 that went through those points (which would be equidistant on either side from the hyperplane), which produced the following:

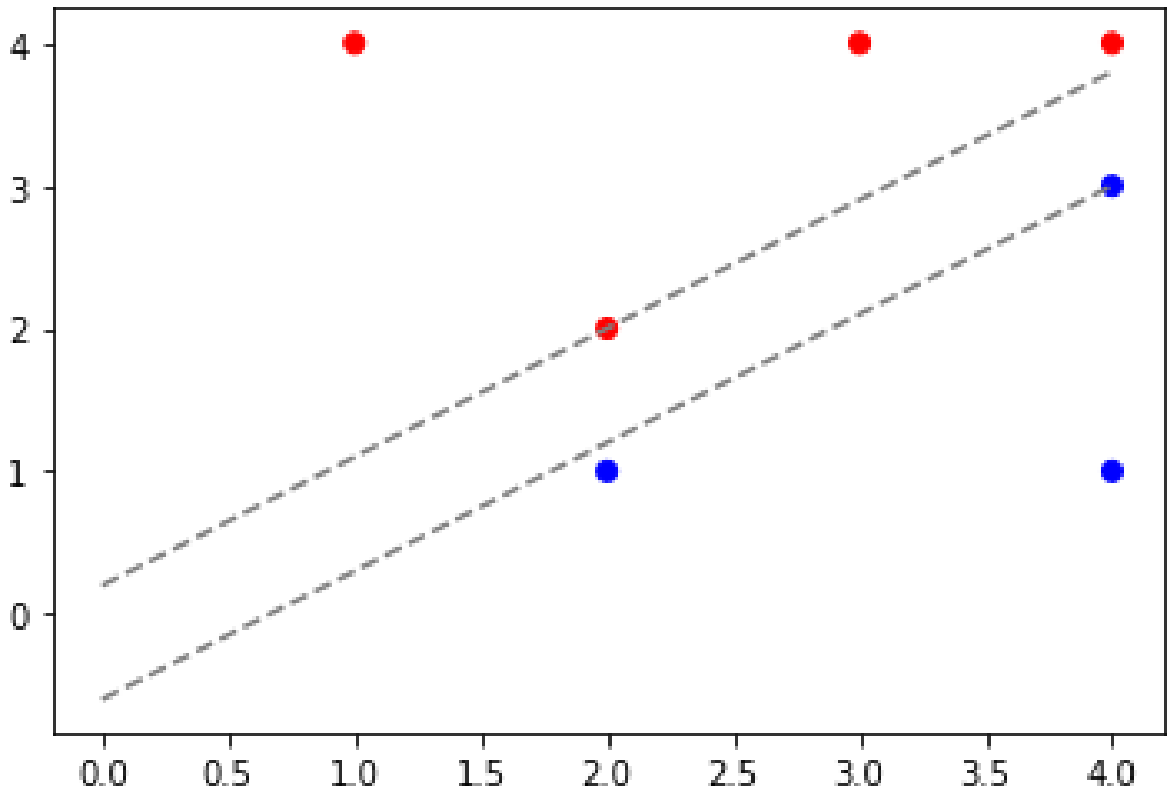


Figure 24: Observations with initial guess of lines of slope 0.9

As can be seen, these lines do properly split the data – no data points fall in between the two lines. We then wanted to guess two lines that went through the same points but with a smaller slope of 0.6. The result was as follows:

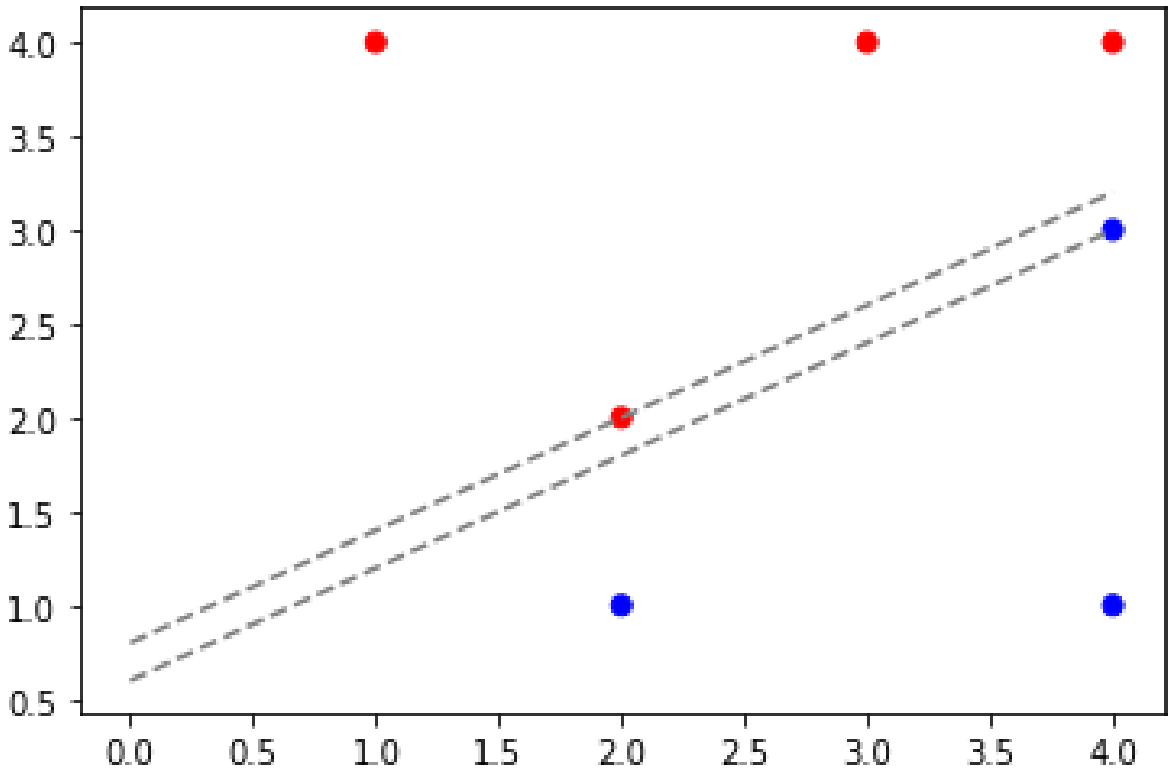


Figure 25: Observations with initial guess of lines of slope 0.6

As can be seen, the lines again properly split the data. For clarity, we plot both of these results on the same figure:

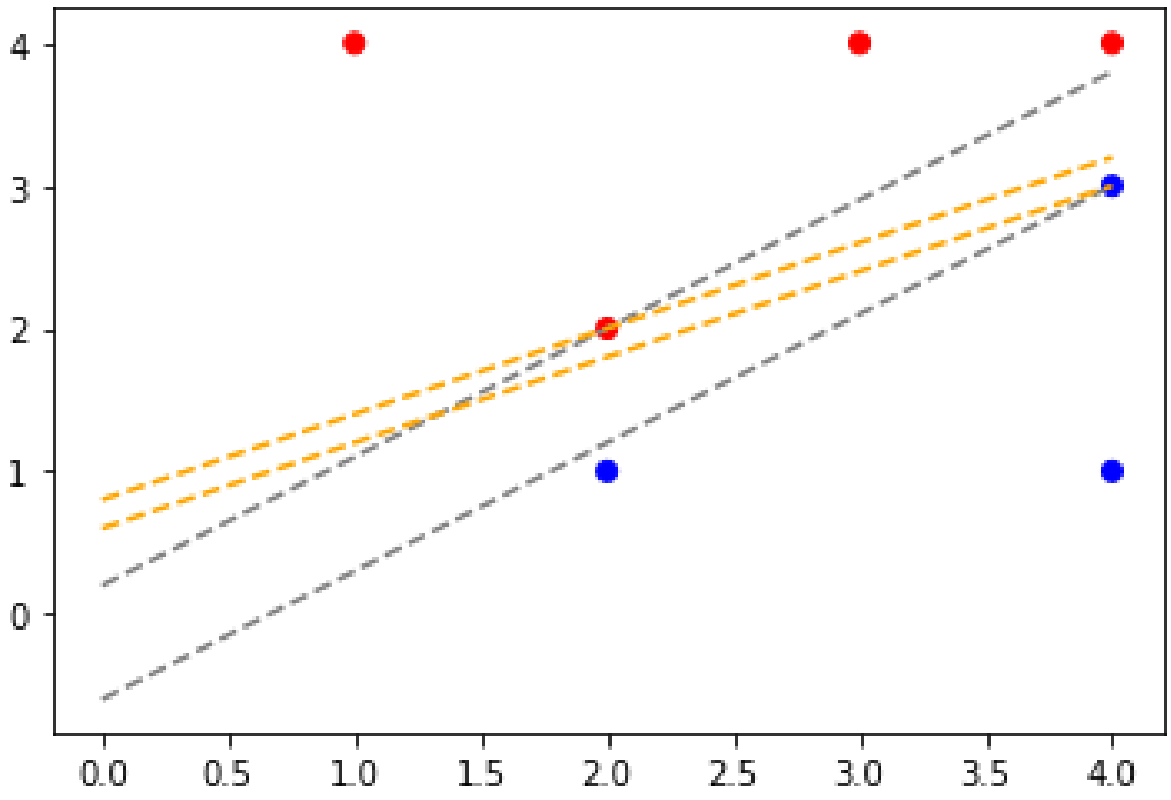


Figure 26: Observations with initial guesses of lines of slope 0.6 and 0.9

This led us to an important clue for this data – rotating the lines clockwise (in other words, decreasing the slope) was leading to smaller margins. This is evident in the plot – the space between the lines of slope 0.6 is much smaller than the space between the lines of slope 0.9. This indicated to us that to maximize the margin of our hyperplane, we had to increase the slope of these lines until they failed to properly split the data. We quickly realized that when the slope of each of these lines is 1.0 (and each still go through the initial two points we defined, respectively), they then also go through red point (4,4) and blue point (2,1), respectively. This means that increasing the slopes past 1.0 under the same conditions would cause (4,4) and (2,1) to fall in between our two lines, which would no longer make it the maximum margin classifier as another point from the class would be closer to the opposing class than the one being used as a support vector in defining the hyper-plane separating the two classes. Therefore, the two lines we want are both of slope 1 with one line going through the points (2,2) and (4,4) and the other line going through the points (2,1) and (4,3) – this maximizes the margin in between the two lines while still maintaining a proper split of the data.

Our maximum margin seperating hyperplane is therefore going to fall exactly in between these two lines with a parallel slope. That means that we know m = 1 in the equation y = mx + b, which in our case is X2 = mX1 + b. We also know the y intercept, b, should fall right in between the other two y-intercepts. We calculate b for the other two lines below:

Line 1: X2 = X1 + b. We plug in (2,2) from the line. Therefore, b = 0.

Line 2: X2 = X1 + b. We plug in (2,1) from the line. Therefore, b = -1.

Our b for our hyperplane line will therefore be right in between -1 and 0, which means b = -0.5. Therefore the equation for our hyperplane line will be X2 = X1 - 0.5.

Just for confirmation and ease of use, we used scikit learn do create a more attractive and easy to use representation of our data and hyperplane:

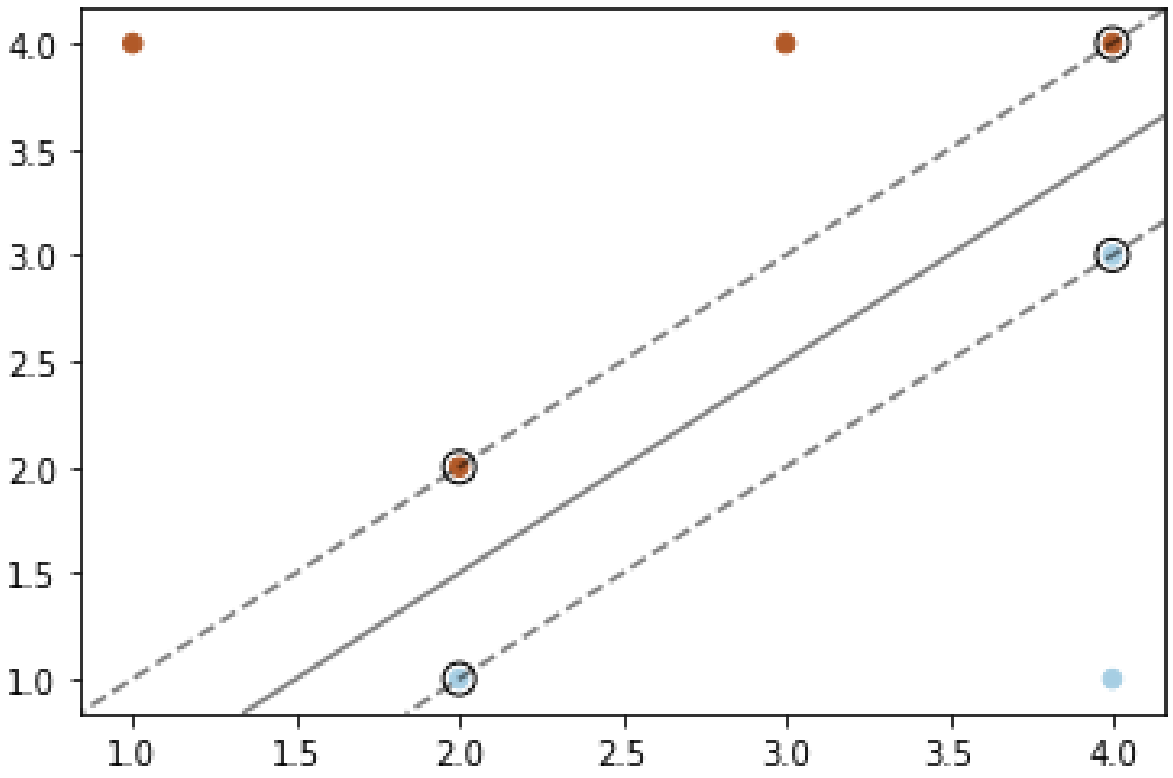


Figure 27: Observations and the Maximum-Margin Seperating Hyperplane

(b) Describe the classification rule for the maximal margin classifier. It should be something along the lines of "Classify as Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$, or classify as Blue otherwise." Provide the values for β_0, β_1, and β_2.

The equation of the maximum-margin hyperplane line that we drew and calculated in part (a) is X2 = X1 - 0.5 (see explanation above). This line can be rearranged and represented as X2 - X1 + 0.5 = 0. All the points above this line are red and all the points below this line are blue. Therefore, our classification rule can be: "Classify as red if X2 - X1 + 0.5 > 0, otherwise classify it as blue." This expression is of the form $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$, meaning that our $\beta_0 = 0.5$, $\beta_1 = -1$, and $\beta_2 = 1$.

(c) On your sketch, indicate the margin for the maximal margin hyperplane.

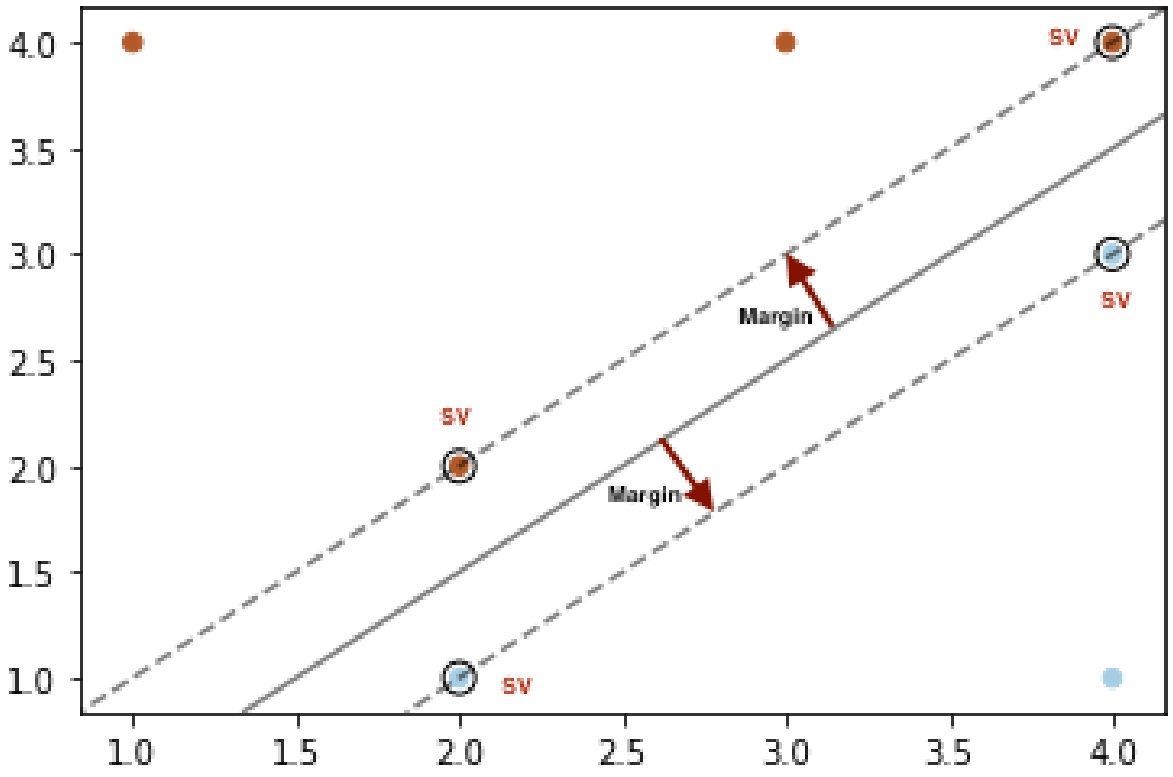


Figure 28: Maximum-Margin Seperating Hyperplane with Margin and Support Vectors

As can be seen in figure 28, the margin is labeled as the perpendicular distance from the maximum margin separating hyper plane to the nearest point of each class (in this case, the dotted line goes through the closest points, so drawing the margin line perpendicularly to the dotted line is the same as drawing it to the nearest point of each class).

(d) Indicate the support vectors for the maximal margin classifier.

As can be seen in figure 28 as well, the support vectors are all labeled with label "sv." These are the points that lie right on the edge of the margin and are the points that dictate what the maximum-margin separating hyperplane will be. The perpendicular distance from the hyperplane to any of these points will be equivalent and they are the points closest to the hyperplane.

(e) Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.

The seventh observation, point (4,1), is not a support vector. Ultimately, support vectors are the data points that dictate what the maximal margin hyperplane will be, as they are the points closest to such a plane (they form the closest boundaries of each class). (4,1) is much farther from the red-labeled points than the current support vectors for the blue class, (4,3) and (2,1). Therefore, as long as we move the seventh observation in such a way that keeps it below the line that goes through (4,3) and (2,1), those two points will still be the support vectors and dictate the hyperplane. This means that we can actually move the seventh observation around a lot without affecting the maximal margin hyperplane. Ultimately, moving the observation would only change the hyperplane if it moved above the line that goes through (4,3) and (2,1), as this point would then be the new support vector and the new limiting factor for the hyperplane (as it would be the point from class blue closest to class red points).

(f) Sketch a hyperplane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.

A hyperplane that is not the maximum margin hyperplane is one that we already started displaying in part (a). For reference, below are the two lines that we drew in part (a) that surround a hyperplane that does not have the maximum margin. We have added the hyperplane (colored green):

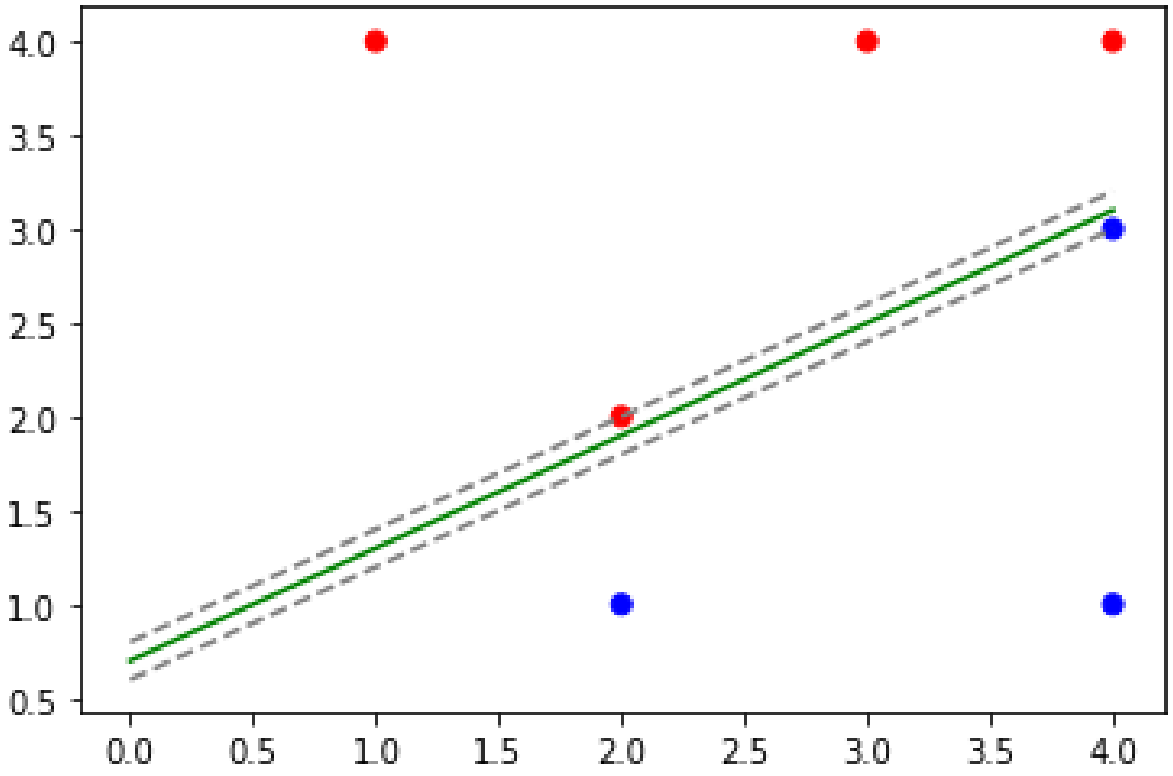


Figure 29: Hyperplane with sloipe 0.6 (margin not maximized)

As is evident, this hyperplane does properly split the blue and red points. However, its margin (distance to each of the gray lines) is significantly reduced relative to the margin we saw in part (c). This is because we chose lines with a smaller slope even though we could still increase the slope past 0.6 while still splitting the red and blue points with no mis classifications. As we note in our code, the equations of the two gray lines are X2 = 0.6X1 + 0.8 and X2 = 0.6X1 + 0.6. Therefore, the green hyperplane we have sketched will fall right in between these two lines (same slope, but y intercept in between the y intercepts of the two lines) with equation **X2 = 0.6X1 + 0.7**.

(g) Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.

We have added a point to our original datapoints and have plotted it below (the arrow points to the new point):

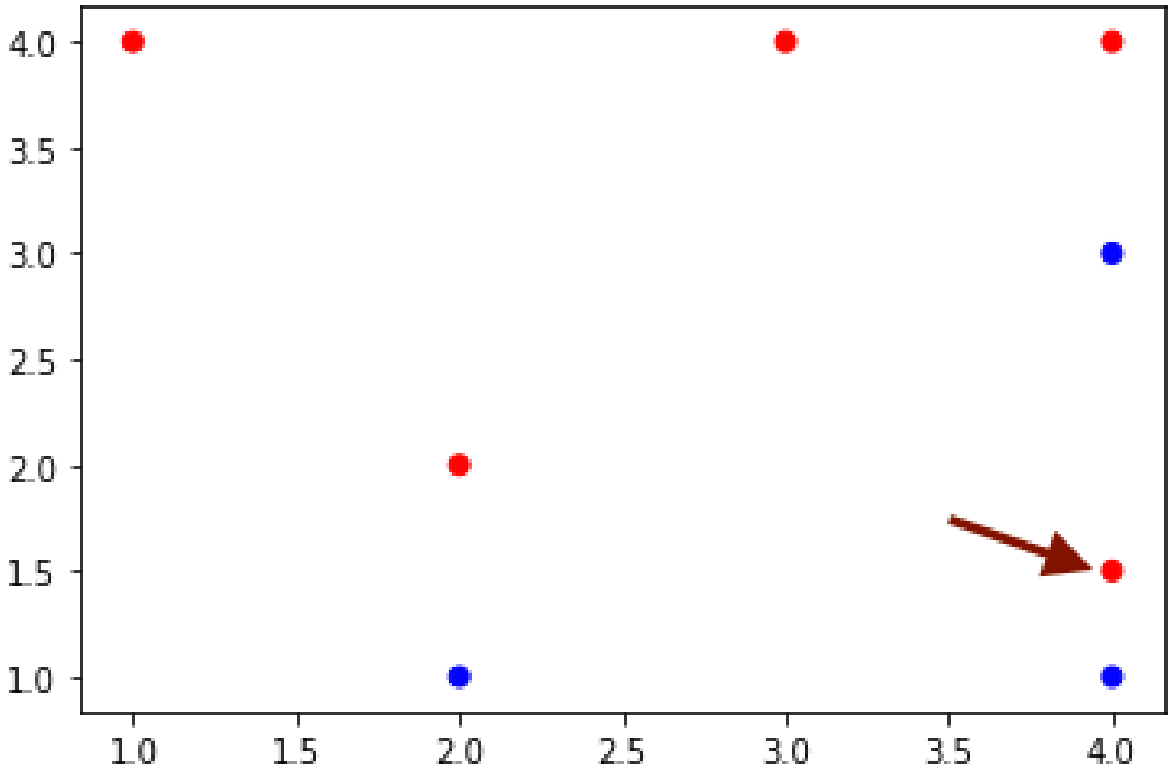


Figure 30: Dataset with a new point added

We added a red point at (4,1.5). Since our data is in 2 dimensions, our hyperplane is linear (a line). As is clear by the way this data is set up, a red data point falls within the general area that we think of as "the blue points." This makes it impossible to draw a linear line that can separate all of the red points from the blue points. Any line/hyperplane will inevitably lead to mixture of the different classes.