# Classifying Art using Probability

Ian Chang

November 2020

## 1 Introduction

Humans have produced art for thousands of years in all sorts of forms. However, given the vast geographic differences and chronological differences, I was interested in seeing how works of art from different time periods and different countries differ statistically. If a significant difference exists, then it might be possible to create a classifier to categorize works of art. A classifier could help museum curators, art historians, or art auction houses verify and classify different works. Basically, I wanted to answer two guiding questions: does time and location statistically effect art? and is it possible to create a classifier that can place art into a specific time and location? Obviously, this is a broad problem, so I needed to find a scope to make it feasible. I decided to examine 4 of the most famous 'movements' or 'eras' of art: the Dutch Golden Age, Italian Renaissance, French Impressionism, and the English Victorian Age. The Dutch Golden Age is the name used to refer to paintings created in the Netherlands between 1600 and 1800. Famous painters of the Dutch Golden Age include Rembrandt and Vermeer. The majority of paintings created during the Dutch Golden Age were portraits of the merchant class, but landscapes and seascapes were also popular. The Italian Renaissance lasted from roughly 1400 to 1600. Famous Italian Renaissance painters include Raphael, Titian, Da Vinci, and Michelangelo. The vast majority of the paintings were religious works commissioned by the Catholic Church. French Impressionism was a famous art movement in France that lasted from roughly 1800 to 1900. Impressionism is characterized by having less detail and trying more to capture the essence of a painting. They were also painted on white or light backgrounds. Famous French Impressionists include Monet, Pissaro, and Degas. Finally, English Victorian Age painting occurred in England from around 1800 to 1900. Famous Victorian Age painters include Turner, Gainsborough, and Constable. In the next section, I'll explain more why I chose these specific eras. I wanted to see whether paintings from those different eras had attributes that were unique and predicated on being from those eras. I then wanted to see if I could build a classifier using the techniques we studied in class to sort paintings into those four eras.

## 2 Dataset

Finding good data was very difficult. High quality online images for historical art exists for only a fraction of the world's works, and of those that do exist, very few are open access and available for anyone to download and use. The Metropolitan Museum of Art in New York has one of the highest quality open access programs of any art museum with over 30,000 open access works. To make searching easier as well as ensuring all of my training data came from a uniform quality of image, I decided to use the Met's website to find all of my training data ($https://www.metmuseum.org/about-the-met/policies-and-documents/open-access$). However, even with 30,000 open access works the Met draws from all time periods and locations, which means that finding a collection from a specific time and place numerous enough to use as training data severely restricts what paintings you can use. In addition, many of the works are sculpture, furniture, or other non-painting items. Using the search function on the Met's website, I determined that the Dutch Golden Age, Italian Renaissance, French Impressionism, and Victorian Age were the collections that had the highest quantity of usable paintings. The wide time range I had to use for some of these eras was driven by the scarcity of available data, and the Met's websites search functionality. I used the websites filter functionality to filter by time period (which sometimes only allowed you to narrow to blocks of 200 years), country, and to make sure I was looking at Canvas paintings. Even choosing these periods, I was limited by the availability to only 40 images per era. I manually downloaded 40 images from each period. In order to remove bias in the data I just used the first 40 to appear in the search (since it appeared to be randomly sorted), however, I did manually filter out paintings that had a large frame or were non-rectangular in shape and thus had a large white space. Those pixels would have thrown off the data I extracted the images. 40 images from 4 periods meant I had a total training data set of 160 images, which is smaller than I would have liked, but was the best that was available.

## 3 Code Structure and Youtube Link

All the code is available at https://github.com/iannchang/art_classifier as well as turned in on Gradescope. The code consists of 5 files: main, cv_utils, nb_classifier, sr_classifier, and utils. Main calls methods from the other files to the program. nb_classifier and sr_classifier create Naive Bayes and Softmax Regression classifiers and contain fit, predict, and helper methods for thoes classifiers. Cv_utils contains wrapper methods for OpenCV that help control importing images and processing them. Utils contains miscellaneous functions that help tie all of the OpenCV functions together, calculate the conditional expectations, process data into training features and training labels, and inference using the Naive Bayes and Softmax Regression Classifiers. I do a walkthrough of my code in my video, which is the youtube link below

https://youtu.be/P0TFDwOySY4

# 4 Getting Data using OpenCV

I decided to use OpenCV to extract numerical data from the paintings to use as feature variables. I extracted six variables for each painting using OpenCV: average red value, average green value, average blue value, hue (color) , saturation (greyness), and value (brightness). Each pixel in each paintings has a r,g,b,h,s, and v value associated with it, and I took the average for the entire painting. r,g,b,s, and v are continuous random variables in the range [0,255] and h is a continuous random variable in the range [0,179]. When you import an image using opencv, it is basically treated like a matrix of pixels so I was able to calculate the average r,g,b,h,s, and v variables simply by looping through the matrix. This is how I calculated average rgb values. The process for hsv was similar except I had to convert the pixel matrix to hsv format first.

```
def find_rgb(image):
    counter = 0
    r = 0
    g = 0
    b = 0
    for row in image:
        for pixel in row:
            r += pixel[2]
            g += pixel[1]
            b += pixel[0]
            counter+=1
    average_rgb = [r/counter, g/counter, b/counter]
    return average_rgb
```

Thus, I was able to flatten each painting down into an array of six variables. Each of these painting arrays had an associated era name (dutch, english, french, or italian). I dumped all the data into a dictionary where the keys were the names of the art eras and the values were a list of the feature variable arrays. Thus, for example, the key dutch would have an associated value of a list containing 40 arrays each of which represented a painting. This would allow me to determine whether these variables were independent of the era or not.

# 5 Determining Independence

In order to calculate where the feature variables were independent of the art period, I used the conditional expectation. For example, let us look at the expectation for the average red value given that the painting is from the dutch golden age

$$E[r|era = dutch]$$

If r and art era are independent, then we would expect that

$$E[r] = E[r|era = dutch]$$

If they are not equal, we can assume that r is dependent on era. I therefore wrote a python function that took in the dictionary and calculated the conditional expectation for all six variables r,g,b,h,s, and v conditional on the era.

```
def test__independence(all_eras):
#ex: calculate the E[r | dutch] and see if it is different from E[r].
#If they are equal then we can see that they are independent
#note: ranges are [0,255] for rgb. [0,179] for h, [0,255] for s, and [0,255] for v
    era_averages = {}
    overall_totals = np.zeros(6)
    overall_painting_counter = 0
    for era in all_eras:
        totals = np.zeros(6)
        painting_counter = 0
        for painting in all_eras[era]:
            i = 0
            for value in painting:
                totals[i] += value
                overall_totals[i]+=value
                i+=1
            painting_counter+=1
        overall_painting_counter+=painting_counter
        averages = totals/painting_counter
        era_averages[era] = averages
    overall_average = overall_totals/overall_painting_counter
    var_names = ["r", "g", "b", "h", "s", "v"]
    for era in era_averages:
        counter = 0
        for expectation in era_averages[era]:
            print("E["+var_names[counter]+"|"+era + "] = " + str(expectation))
            counter+=1
    counter = 0
    for average in overall_average:
        print("E[" + var_names[counter] + "] = " + str(average))
        counter+=1
```

As the code shows, using the dictionary I described in the previous part, I could simply calculate the expectation for each feature variable given the era using averages. I then calculated the expectation for each feature variable for the total dataset. Results are shown below:

| | |
|---|---|
| $E[r\|dutch] = 59.525389689046584$ | $E[r\|english] = 95.97500613080473$ |
| $E[g\|dutch] = 50.94424490430606$ | $E[g\|english] = 86.11491905754502$ |
| $E[b\|dutch] = 38.186085980290635$ | $E[b\|english] = 70.70127467808587$ |
| $E[h\|dutch] = 24.735844386349033$ | $E[h\|english] = 31.86314638245661$ |
| $E[s\|dutch] = 99.98510215207102$ | $E[s\|english] = 93.709816192879$ |
| $E[v\|dutch] = 60.42927925696582$ | $E[v\|english] = 99.82074811876609$ |

$$\begin{array}{|c|c|}
\hline
E[r|italian] = 87.6303628900454 & E[r|french] = 98.18106044178163 \\
E[g|italian] = 72.45830984593529 & E[g|french] = 89.00373970041542 \\
E[b|italian] = 54.76331460464585 & E[b|french] = 69.60685279045282 \\
E[h|italian] = 35.776986561332365 & E[h|french] = 32.55238035206773 \\
E[s|italian] = 114.37470711815197 & E[s|french] = 95.97260139901147 \\
E[v|italian] = 90.27693747145459 & E[v|french] = 102.00551275203058 \\
\hline
\end{array}$$

$$E[r] = 85.32795478791964$$
$$E[g] = 74.63030337705044$$
$$E[b] = 58.314382013368835$$
$$E[h] = 31.232089420551432$$
$$E[s] = 101.01055671552834$$
$$E[v] = 88.13311939980424$$

Given the small data set, it would be reasonable to assume perhaps a +/- 5 tolerance for qualifying whether the expectation as a whole is equal to the conditional expectations. An inspection of the data shows that except possibly for hue, all of the feature variables are not independent for at least one of the eras. Thus, we have determined that these variables are not independent and can thus be used as feature variables for a classifier. In addition, this data provides us interesting inferences about the paintings themselves. For example using $E[v|dutch] = 59.53$ compared to the total $E[v] = 88.13$ we can see that Dutch paintings are much darker than the dataset as a whole. In addition, because of the low rgb values compared to the entire dataset we can see that Dutch paintings are a lot less colorful also. We can also see that both French and English paintings tend to be a lot brighter than the whole and also a lot more colorful. In addition, by comparing the conditional expectations for English and French paintings we can see that they are very closely related. This foreshadows problems later on. In addition, comparing Italian paintings and the total expectation we can see that except for saturation, all of the variables seem to be independent of era. This also foreshadows problems that occured later on.

# 6 Classifying using Naive Bayes

Given our results from the previous section, we can begin creating a classifier using the Naive Bayes algorithm. First, we must process our data into training features and training labels matrices from the dict format it was in for calculating conditional expectation.

```
def process_training_data(all_eras):
train_features = []
train_labels = []
discrete_labels_dict = {"dutch":0, "english":1, "french":2, "italian":3}
for era in all_eras:
    for painting in all_eras[era]:
        train_features.append(painting)
```

```
            train_labels.append(discrete_labels_dict[era])
    train_features = np.asarray(train_features)
    train_labels = np.asarray(train_labels)
    return (train_features, train_labels)
```

Next, we must bucket our feature variables in order to make them discrete. I accomplished this by dividing each continuous variable and then using a floor function. I bucketed them into 15 buckets of range 17.

```
def bucket_data(test_features, training_features):
    #bucket continuous features variables to make them discrete.
    row_counter = 0
    for features in test_features:
        column_counter = 0
        for feature in features:
            test_features[row_counter][column_counter] = math.floor(feature/17)
            column_counter+=1
        row_counter+=1
    row_counter = 0
    for features in training_features:
        column_counter = 0
        for feature in features:
            training_features[row_counter][column_counter] = math.floor(feature/17)
            column_counter+=1
        row_counter+=1
    return (test_features, training_features)
```

We want to use Naive Bayes to calculate the probability for each era, given our continuous feature variables r,g,b,h,s,and,v which from now we'll call $X_0...X_5$. Let us call our label variable $E$ where $E$ is discrete and can equal $0, 1, 2, 3$ where 0: Dutch, 1: English, 2: French, 3: Italian. We want to calculate:

$$P(E|X)$$

We use Bayes Theorem to show that:

$$P(E|X) = \frac{P(E)P(X|E)}{PX}$$

We could then use the fact that the numerator is the joint probability and then use the chain rule to calculate them all. However, that is not sensible computationally. Thus, we make the Naive Bayes assumption, which is that the feature variables $X_0...X_5$ are independent of each other given the era condition. Thus, all we need to do is calculate $P(E)$ as well as the $P(X_i|E)$ for each feature variable. Since we have 160 paintings in our training data set and six feature variables, our Training Features matrix is a 160 x 6 Matrix. Our Training Labels array is an array of length 160 where each value corresponds to the era for the corresponding row in the training features matrix. These values are either 0,1,2,3 depending on which era the painting came from. We use the fit function to create counts of the frequency of each label as well as the frequency of combinations of feature variable values and labels.

```
def fit(self, train_features, train_labels):
        self.label_counts[0] = 0
        self.label_counts[1] = 0
        self.label_counts[2] = 0
        self.label_counts[3] = 0
        for label in train_labels:
            self.label_counts[label]+=1
        row_counter = 0
        for row in train_features:
            column_counter = 0
            for column in row:
                tuple = (column_counter, train_features[row_counter][column_counter], \
                    train_labels[row_counter])
                frequency = self.feature_counts.get(tuple, 0)
                frequency+=1
                self.feature_counts[tuple] = frequency
                column_counter+=1
            row_counter+=1
```

We then use those counts in our predict function. I downloaded a further 5 images from the Met's website for each era. That gives a total test dataset of 20 images. Thus, test features is a 20 x 6 matrix. I bucketed the test feature variables using the same process as the training feature variables. Calculating the $P(E)$ was trivial using the label counts array from the fit function. Since my dataset was evenly divided between the four eras, the probability for each ended up being .25. I stored these in a list. I then had to calculate the $P(X_i|E)$ for each feature variable. For each feature variable in each row of the test features matrix, I then had to calculate 4 values. Let's say that $x_i$ was the value of the feature variable in the test matrix,I then had to calculate $P(X_i = x_i|E = 0), P(X_i = x_i|E = 1), P(X_i = x_i|E = 2), P(X_i = x_i|E = 3)$. I was able to calculate those values by

$$\frac{number\ of\ training\ examples\ where\ X_i{=}x_i\ and\ E{=}e_j}{number\ of\ training\ examples\ where\ E{=}e_j}$$

I did this using the label counts and feature counts I calculated in fit. At this point, I also had a toggle where I could either add in 1 in the numerator and 2 in the denominator to use Laplace smoothing. Thus, for each painting in the test features I then had a 2d array where each feature variable had an associated four probabilities for $E = 0, 1, 2, 3$. I then was able to multiply those all out, ie: for $E = 0$: I calculated

$$P(X_1 = x_1|E = 0)P(X_2 = x_2|E = 0)P(X_2 = x_2|E = 0)P(X_3 = x_3|E = 0)P(X_4 = x_4|E = 0)P(X_5 = x_5|E = 0)$$

After doing that for all 4 values for E, I took the max of those four calculated probabilities and the associated value for E was my prediction of the era that that painting came from.

```
def predict(self, test_features):
        preds = np.zeros(test_features.shape[0], dtype=np.uint8)
        laplace_num = 1
        laplace_den = 2
        if self.use_mle:
            laplace_num = 0
            laplace_den = 0
        num_row = 0
        total_label_count = 0
        for count in self.label_counts:
            total_label_count+=self.label_counts[count]
        p = []
        for count in self.label_counts:
            p.append(self.label_counts[count]/total_label_count)
        for row in test_features:
            num_feature = 0
            feature_prob = [[]]
            for feature in row:
                feature_prob.append([])
                for i in range(len(self.label_counts)):
                    num = self.feature_counts.get((num_feature, \
                        test_features[num_row][num_feature], i),0) + laplace_num
                    den = self.label_counts[i] + laplace_den
                    feature_prob[num_feature].append(num/den)
                num_feature+=1

            row_prob = [1]*len(self.label_counts)
            for i in range(len(feature_prob)-1):
                for j in range(len(self.label_counts)):
                    row_prob[j] = row_prob[j] * feature_prob[i][j]

            for i in range(len(self.label_counts)):
                row_prob[i] = row_prob[i] * p[i]
            row_prob = np.asarray(row_prob)
            preds[num_row] = np.argmax(row_prob)
            num_row+=1
        return preds
```

The predictions ended up being decent. Here are the predictions Naive Bayes MAP made compared to the actual test labels.

Test Labels: [0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3]
Predictions: [0 0 3 0 0 2 2 1 1 1 2 2 1 2 2 3 2 3 2 3]

And here are the predictions made by the Naive Bayes MLE.

Test Labels: [0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3]
Predictions: [0 0 3 0 0 1 2 1 1 1 2 2 2 2 2 3 2 3 2 3]

Thus, the Naive Bayes MAP correctly predicts the era of the painting 70% of the time while the Naive Bayes MLE correctly predicts the era of the painting 80% of the time. This is pretty decent considering that random choice would only correctly predict era 25% of the time. The reasoning behind the mistakes also seems pretty obvious considering the conditional independence. Naive Bayes MLE being more accurate then MAP is probably due to the prior having a large effect on the small dataset. Let us look at the Naive Bayes MLE predictions. We see that it correctly predicts all of the French images. We also see that it correctly predicts all of the English images except for one which it predicts as French. Out of all of the possible incorrect predictions, predicting that an English painting is French is the most correct. The conditional expectations for the feature variables are similar for all except the average green and blue values. Thus, to the classifier the French and Engish images are pretty similar. This actually makes a lot of sense, because the French Impressionism and English Victorian Age happened at the same time (1800-1900) and France and England are close geographically. In addition, it seems that the era it had the most trouble identifying was Italian. This also makes sense after looking at the conditional expectation given that the most of the feature variables for Italian paintings seemed to closely resemble the feature variables for the overall training dataset. Finally, looking at the one Dutch painting the classifier incorrectly identified I saw that this painting is not a portrait. However, the vast majority of the Dutch paintings in the training data were portraits. That shows that the classifier has a hard time predicting outliers, and also that a Dutch landscape may have more in common with an Italian landscape than it would a Dutch portrait. However, it also shows that using a Naive Bayes classifier to tell whether a painting is a portrait or not would probably be pretty successful. Overall, the Naive Bayes Classifier works decently but it struggles identifying outliers and classifying into eras that do not exhibit unique traits.

# 7 Classifying using Softmax Regression

Next, I tried to accomplish writing a classifier using Softmax Regression. Softmax regression is similar to logistic regression but it can be used for situations where the label variable is discrete and not just binary. In addition, because it can use continuous feature variables I thought this might be more accurate than Naive Bayes which I needed to bucket. Since this topic is not covered in the class, I won't go into too many details, but it uses gradient descent with the softmax function.

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$$

The fit function is very similar to how a fit function for a logistic regression would look. I inserted a column of ones for the bias weight. I also had problems with the weights being all very large negative values, which resulted in the softmax function for prediction outputing all zeros, so I fixed that problem by

normalizing the training features. I then was able to calculate the gradients using the softmax function and one hot encoding as seen below. One hot encoding is simply reformatting the training labels so instead of having a 1d array, you have a 2d array where the number of rows is the length of the training labels array and you have a 1 in the column whose index is the value of the label variable. The rest are zeros. After repeating the gradient calculating many times, I ended up with a set of weights to use in the predict function.

```
def softmax(vec):
    vec -= np.max(vec)
    return np.exp(vec) / np.sum(np.exp(vec))

def one_hot_encode(train_labels, num_labels):
  one_hot = np.zeros((len(train_labels), num_labels))
  one_hot[np.arange(len(train_labels)), train_labels] = 1
  return one_hot

def fit(self, train_features, train_labels, num_labels):
        train_features = np.insert(train_features, 0, 1, axis=1)
        train_features[:, 1:] = (train_features[:, 1:] - np.mean(train_features[:, 1:], \
            axis=0)) / np.std(train_features[:, 1:], axis=0)
        train_features.setflags(write=False)
        m = train_features.shape[1]
        np.random.seed(0)
        theta = np.random.random((train_features.shape[1],num_labels))
        for i in range(self.max_steps):
            thetatx = np.matmul(train_features, theta)
            gradient = 1/m * np.matmul(np.transpose(train_features), (softmax(thetatx) - \
                one_hot_encode(train_labels, num_labels)))
            theta = theta - self.learning_rate*gradient
        self.weights = theta
```

The predict function was pretty simple to write, all I had to do was input the product of the test features matrix and the weights calculated in the fit function into the softmax function. The result of that would be a matrix where the number of rows was the number of the paintings in the data set and the value for each column was the probability that $E = 0, E = 1, E = 2, E = 3$. I could then simply just find the max probability in each row, and then my prediction would be the corresponding era.

```
    def predict(self, test_features):
        test_features = np.insert(test_features, 0, 1, axis=1)
        preds = np.zeros(test_features.shape[0])
        probabilities = softmax(np.matmul(test_features, self.weights))
        counter = 0
        for row in probabilities:
            preds[counter] = np.argmax(row)
```

```
            counter+=1
        return preds
```

The softmax regression surprisingly did worse than the Naive Bayes prediction.

Test Labels: [0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3]
Predictions: [0 0 0 0 0 0 2 2 3 3 2 2 2 2 2 2 2 2 2 3 2 2]

Even after playing around with the learning rate and number of steps a lot, the best I could get it to perform was 11/20 correct which at 55% correct is better than random selection but not nearly as good as either of Naive Bayes implementations. The Softmax algorithm seemed to want to either converge to predicting every painting to be English or Italian or every painting to be Dutch or French depending on the step count/learning rate.

# 8 Conclusion

Looking back on my two guiding questions. Does time and location statistically effect art? My calculations show that it clearly does. For example, we could see that Dutch paintings were much darker than the entire dataset. However, I think that types of art (landscape vs portrait etc) may have more in common across eras. A Dutch landscape may have more in common with an English landscape than it does a Dutch portrait. Regardless, it was still interesting to see trends for each era as a whole. For the second question, is it possible to build a classifier? I had mixed results. Both classifiers were able to predict at a rate much better than random choice, but both were also nowhere near perfect. I have a number of theories why. First, my dataset was rather small. Having such a small dataset means that outliers have a large effect on both the training and predicting. 160 images for training and 20 for testing was definitely not enough. However, given the scarcity and difficulty of assembling high quality datasets for art, this problem may be a hindrance to anyone trying to use historical art as training or testing data. This scarcity also meant I had to choose wide ranges for my label variables. Finally, I think my choice of feature variables may have been poor. First of all, they definitely fail the Naive Bayes assumption since rgb and h values are directly related. Also, finding the averages for the entire painting may have removed some specificity. Perhaps calculating based on individual pixels may have worked better, but that would have also introduced a major level of complexity. Only considering six feature variables is probably not enough for the complexity of the problem. It could also be that European art is simply too similar. However, I do think that with higher quality datasets and more defined buckets to place paintings into, that a machine learning classifier for art could work. Perhaps, sorting into individual artists could have success (ie: is a painting a Rembrandt or a Van Gogh). Overall, I do think that machine learning will be a viable tool for art historians in the future. The fact that I was able to achieve 80% accuracy using such a small training dataset is pretty good, and having more data and more feature

variables would definitely increase accuracy. It's very clear to humans viewing art and academics studying art that art from different eras have differences, so trying to quantify those differences is an interesting and important task. I am positive that machine learning and probabilistic analyses could provide a lot of insights for problems such as examining the movement of art trends throughout time, identifying unsigned paintings, finding cultural similarities, and so much more.

# 9 References


$https://www.nga.gov/content/dam/ngaweb/Education/learning-resources/teaching-packets/pdfs/dutch\_painting.pdf$

$https://mymodernmet.com/italian-renaissance-art-definition/$

$https://www.metmuseum.org/toah/hd/imml/hd_imml.htm$

$https://www.khanacademy.org/humanities/becoming-modern/romanticism/romanticism-in-england/a/romanticism-and-the-victorian-era$

$https://github.com/rickwierenga/MLFundamentals$

$https://www.metmuseum.org/about-the-met/policies-and-documents/open-access$