Gaussian mixture model (GMM) as a density estimator, classifier, and generative model on two-moons and MNIST datasets

James Cook

June 12, 2024

Abstract

We implement and optimize algorithms using Gaussian mixture models (GMM) to perform density estimation (i.e. estimating the probability density function) and classification of the two-moons and MNIST datasets. We use methods such as Bayesian information criterion (BIC) and F1 error to find the number of Gaussian components in our GMM which most accurately represents and classifies the datasets. Finally, we use our trained GMMs to generate new data that resembles the data in our original datasets. We also plot and analyze various graphical interpretations of the data and models to visually interpret their predictive performance, and compare the performance of our trained GMM to the performance of other classification methods, namely gradient boosted decision trees and random decision forests.

1 GMM as a density estimator of the two-moons dataset

In this section, we will use Gaussian mixture models in an unsupervised context, namely to estimate the probability density function (PDF) of the two-moons dataset. The Gaussian mixture model and all datasets used in this report are developed by **scikit-learn**, a free and open-source machine learning library for the Python programming language.

1.1 Introduction to GMM

A Gaussian mixture model is a probabilistic "soft clustering" model which assumes a dataset is generated from a mixture of a finite number of Gaussian distributions with unknown parameters. This is less restrictive than, for instance, k-means clustering: in k-means, one partitions ("hard clusters") n observations into k clusters (corresponding to, for example, k Gaussian distributions) such that each observation is assigned to the cluster with the nearest mean, that is, the cluster whose centroid is of minimal distance (defined by e.g. the L2 norm) from the observation. Further, the Gaussians used in k-means are typically circular (corresponding to clusters which are spherical), and of the same size (having the same standard deviation). By contrast, a *Gaussian mixture model* "soft clusters" the data, that is, the model determines the *probability* that a particular data point belongs to a particular class. Furthermore, the Gaussian components in a GMM can be of very different sizes and shapes; unlike k-means, which (as it says in the name) only cares about the distance from a data point to the *mean* of a Gaussian component, the Gaussian mixture model additionally incorporates information about the *covariance* structure of the data.

A Gaussian mixture model works as follows:

As stated, we assume that a dataset is generated from a mixture of k multivariate Gaussian distributions, a.k.a. Gaussian components, with means $\mu_\ell \in \mathbb{R}^{d\times 1}$ and covariance matrices $\Sigma_\ell \in \mathbb{R}^{d\times d}$, of the form

$$\mathcal{N}(x_i, \mu_\ell, \Sigma_\ell) = \frac{1}{(2\pi)^{d/2}\det(\Sigma_\ell)^{1/2}} \exp(-\frac{1}{2}(x_i-\mu_\ell)^T\Sigma_\ell^{-1}(x_i-\mu_\ell))$$

for $1 \leq \ell \leq k$ and where d is the dimension of the data points. Furthermore, each of these Gaussian distributions also has a *weight* π_ℓ (i.e. the probability that a randomly selected data point from the dataset will belong to component ℓ). In particular, the weights must be such that the resulting mixture of Gaussian distributions is itself a valid probability density function, that is, integrating under it equals 1. (One can imagine

that some divine being with infinite computational power is squishing and stretching multivariate Gaussians around to generate the probability function of our dataset, and we are trying to predict/approximate those Gaussians and their parameters.)

A GMM then uses the following iterative algorithm known as *expectation maximization* (EM):

1. Make initial guesses for each $\mu_\ell, \Sigma_\ell, \pi_\ell$. (Our GMM will run k-means on the dataset to make these initial guesses, although random guesses would also work.)

2. For each data point x_i, for each Gaussian component $\mathcal{N}(x_i, \mu_\ell, \Sigma_\ell)$, calculate the probability

$$\begin{aligned} r_{i,\ell} = P(y = \ell \mid X = x_i) &= \frac{P(X = x_i \mid y = \ell)P(Y = \ell)}{\sum_{j=1}^{k} P(X = x_i \mid y = k)} \\ &= \frac{\mathcal{N}(x_i \mid \mu_\ell, \Sigma_\ell)\pi_\ell}{\sum_{j=1}^{k} \mathcal{N}(x_i \mid \mu_j, \Sigma_j)\pi_j}. \end{aligned}$$

   This probability is called the *responsibility of component ℓ for x_i* and is derived from Bayes's theorem. It is the probability that the data point x_i belongs to component ℓ.

3. Based on the responsibilities calculated in the previous step, update the estimates of the model's parameters as follows (where n is the number of data points):

$$\pi_\ell = \frac{\sum_{i=1}^{n} r_{i,\ell}}{n}, \quad \mu_\ell = \frac{\sum_{i=1}^{n} r_{i,\ell} x_i}{\sum_{i=1}^{n} r_{i,k}}, \quad \Sigma_\ell = \frac{\sum_{i=1}^{n} r_{i,k}(x_i - \mu_\ell)^2}{\sum_{i=1}^{n} r_{i,k}}.$$

4. Repeat steps 2 and 3 until the model parameters converge, i.e., only negligibly change from one iteration to the next, or until a specified number of iterations have completed.

According to scikit-learn, GMM is the fastest algorithm for learning mixture models, and, as this algorithm maximizes only the likelihood, it will not bias the means towards zero, nor will it bias the cluster sizes to have specific structures that might or might not apply to the dataset.[1] As we will see, GMMs are highly effective for both density estimation and classification of this dataset. Furthermore, as we will also see, once we train a GMM, it can generate new data which closely resembles the data it was trained on.

1.2 Estimating the PDF of the two-moons dataset using GMM

The scikit-learn *two-moons dataset* consists of a procedurally generated set

$$\{X, y\} = \{\{X_i\}, \{y_i\}\} = \{\{x_{1,i}, x_{2,i}\}, \{y_i\}\}$$

where $X_i = \{x_{1,i}, x_{2,i}\}$ is a *two-dimensional feature*/data point, and $y_i \in \{0, 1\}$ is the *class* corresponding to that data point.

This dataset is generated by the function `make_moons`, which accepts the following parameters:

- `n_samples`: integer number of data points to be generated;
- `shuffle`: boolean value indicating whether to shuffle the samples;
- `noise`: floating point number which is the standard deviation of the Gaussian noise to be added to the data;
- `random_state`: integer value which determines random number generation for the shuffling and Gaussian noise of the dataset.

We will train and test the model for multiple values of `n_samples` and `noise`, and graph the results. Here, `shuffle` will be true (as it is by default), since we wish for our model to be able to meaningfully interpret new data which may be unsorted or of a random nature, hence we will always shuffle the samples on which our GMMs will be trained. Finally, passing a particular integer to `random_state` will allow the reader to reproduce all exact datasets which are used in this report; we pass `42`.

To begin, we generate 500 samples of data, with the standard deviation of the Gaussian noise added to the data as $\sigma = 0.05$. We graph this dataset (*Fig. 1.2.1*), and observe that the x_1, x_2 values appear as two curved shapes resembling moons. Though not used in this section, the class y_i tells us which moon the feature X_i is part of: if $y_i = 0$, then X_i is part of the top moon; if $y_i = 1$, X_i is part of the bottom moon.

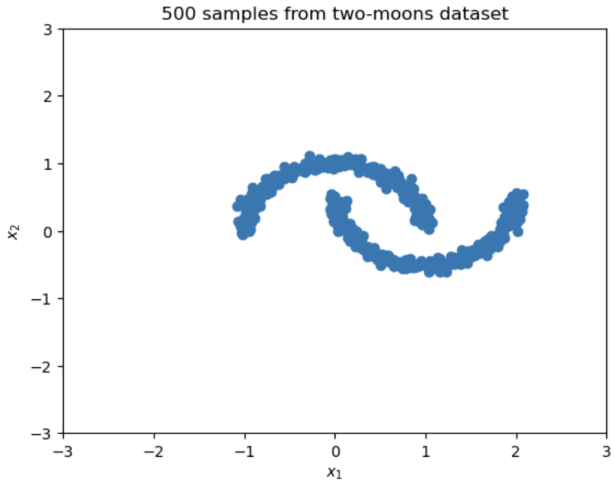


Fig. 1.2.1: Graphing the generated two-moons dataset.

We now train a GMM on this dataset. As stated, we will use scikit-learn's Gaussian mixture model, `GaussianMixture`. Some parameters of `GaussianMixture` which are particularly relevant to us include:

- `n_components`: integer indicating the number of Gaussian components in the GMM;
- `covariance_type`: string specifying the type of covariance parameters to use. Defaults to `full`, in which case each component has its own covariance matrix. This is what we want! We will be fitting Gaussian components (i.e. bell curves) to some strange data, and giving each component its own covariance matrix allows for more degrees of freedom in accurately representing the data using a mixture of Gaussian distributions.
- `init_params`: string specifying the method (either k-means, random from data, or true random) used to initialize the weights, means, and covariances. By default, the GMM will initialize using k-means. This can be costly if the dataset is extremely large; in our case, the datasets are of a tractible size for us to compute, so we choose k-means to get a good initial estimate. If computational cost is a concern, this can be changed to random values.
- `tol`: floating point number $\varepsilon > 0$ representing the convergence threshold. That is, the loop of EM will exit when the lower bound average gain is below this threshold. Defaults to 10^{-3}.
- `max_iter`: integer number of EM iterations to perform if the lower bound average gain never went below ε. Defaults to 100. The default values for `tol` and `max_iter` worked well in experiments.

Remark: the only parameter we specify in our code is `n_components`. We will see that the default values for all other parameters result in a highly effective model.

We now fit our Gaussian mixture model (that is, estimate model parameters using the EM algorithm described in 1.1) by training it on $\{X_i\}$ from our two-moons dataset. First, for purposes of demonstration, we will train a model which uses only 2 Gaussian components. We then plot the level sets of each component in our trained model on top of the dataset (*Fig. 1.2.2*), and also graph the probability surface of the mixture of these Gaussian components in 3 dimensions (*Fig. 1.2.3*), using `matplotlib`. Remark: in case it is not clear, for *all* 3D plots in this section, both the z-axis of the graph and the color represent the PDF of the GMM.

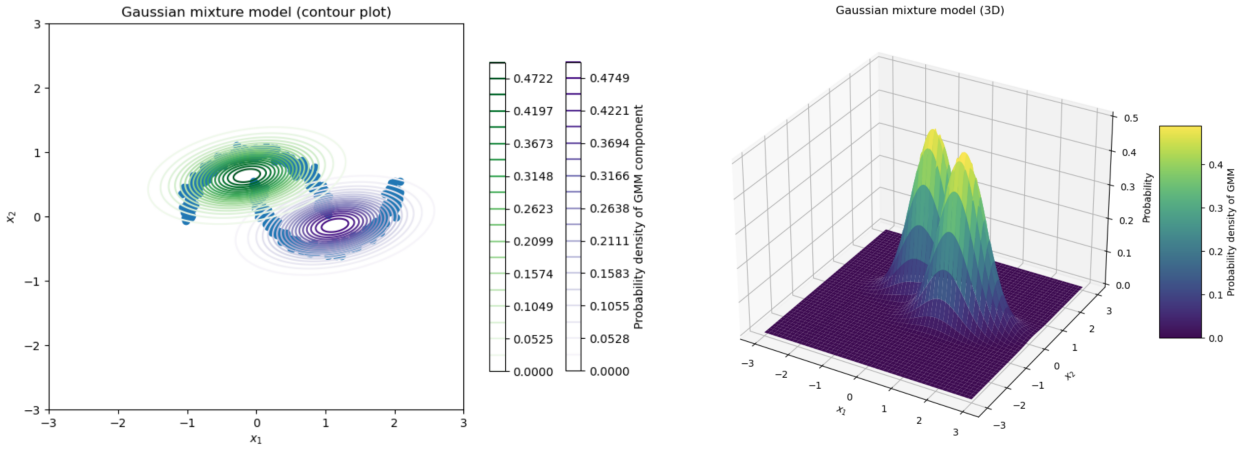


Fig. 1.2.2 (left): Level sets of our 2-component Gaussian mixture model plotted on top of the dataset. Fig. 1.2.3 (right): 3D graph of the probability surface of our Gaussian mixture.

As we can see, the two Gaussian components do not really capture the two shapes accurately. This is exactly as we would expect, intuitively: looking at the two moons, we cannot really encode this geometric object accurately using only two bells (i.e. components). However, if we increase the number of bells, we could place smaller bells roughly equal distances apart along the curvature of each moon, and in doing so better capture the geometry of the two moons. To demonstrate this idea, we first train models (on the same dataset) using 4- and 6-component GMMs, and, as before, graph their level sets and probability surfaces (*Fig. 1.2.4: 4-component model; Fig 1.2.5: 6-component model*). On visual inspection we can already see some improvement in approximating the features:

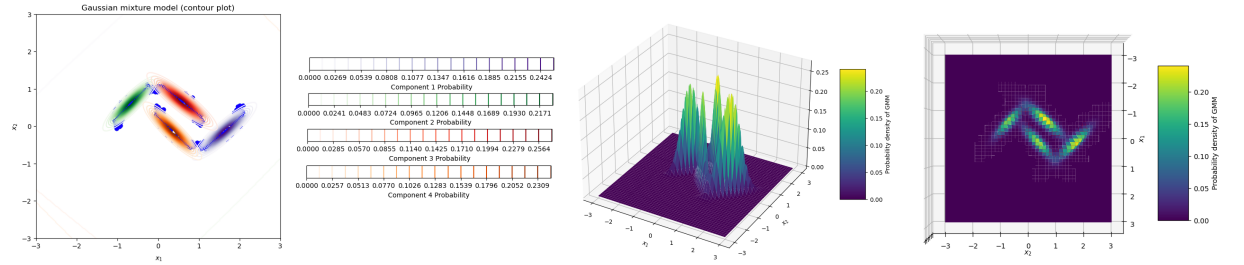


Fig. 1.2.4: Contour plots of each component (graphed on top of dataset) and 3d graphs of probability surface of 4-component GMM.

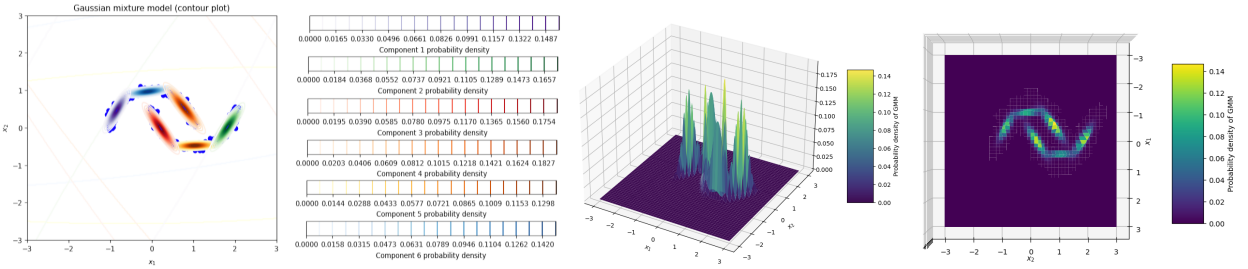


Fig. 1.2.5: Contour plots of each component (graphed on top of dataset) and 3d graphs of probability surface of 6-component GMM.

To determine the optimal number of Gaussian components for our GMM, we test various values (k components for k from 2 to 40) using the method of *Bayesian information criterion* (BIC), defined by:

$$\mathrm{BIC} = k\ln(n) - 2\ln(L)$$

where k is the number of components, n is the number of data points, and L is the maximized value of the likelihood function of a particular model M, i.e., $L = p(x \mid \Theta, M)$ where Θ are the set of parameter

values which maximize the likelihood function and x is the observed data. In particular, observe that as this likelihood L *increases*, the BIC value *decreases*; that is, models with a *lower* BIC value are performing *better*. The scikit-learn `GaussianMixture` object features a BIC function which takes as input a dataset, and returns the BIC value of the current model (the model that `.bic(`X`)` is being called on, from which it gets k, n, and M) on that input dataset. We compute and plot BIC for $k \in \{2, 3, ..., 29\}$ components, and observe that the BIC value is lowest for $k = 9$ (*Fig. 1.2.6*):

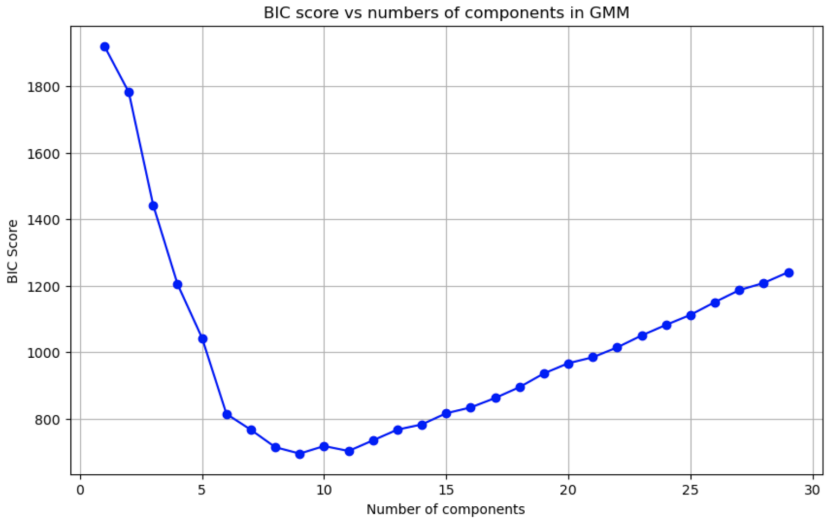


Fig 1.2.6: Plotting BIC value vs. number of Gaussian components in GMM.

This implies that $k = 9$ is the optimal number of Gaussian components in our GMM for a 500-sample dataset with the current level of noise. (Remark: it is interesting that in the optimal case, each moon does not get an equal number of Gaussians, likewise in the second-best case of 11 Gaussians.) We fit the dataset onto a 9-component GMM and see that the geometry of the dataset is captured *much* more accurately. We plot the level sets and probability surfaces of the 9-component GMM (*Fig.'s 1.2.7, 1.2.8, 1.2.9*). The success of the model is particularly evident in *Fig 1.2.9 (see next page)*.

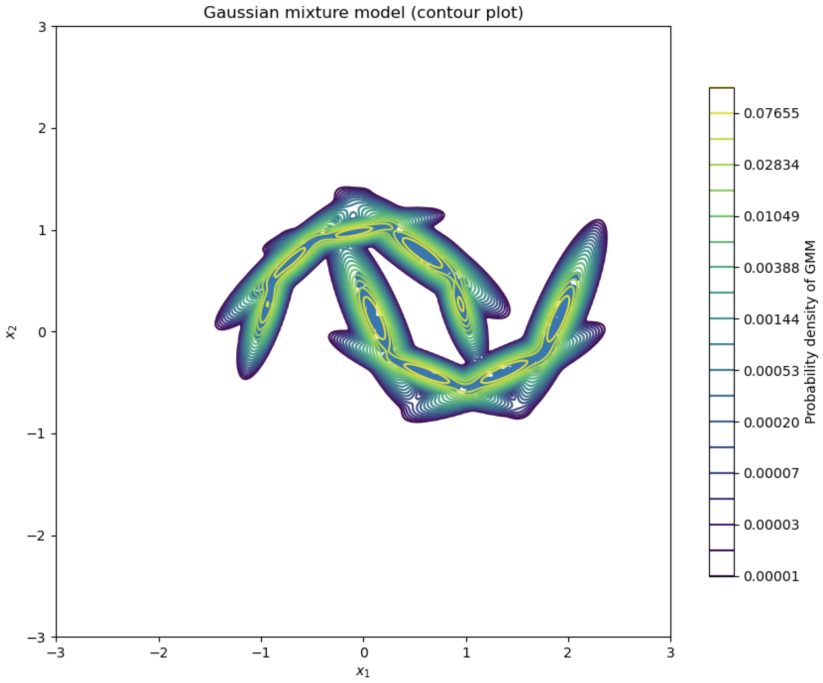


Fig. 1.2.7: 9-component GMM level sets.

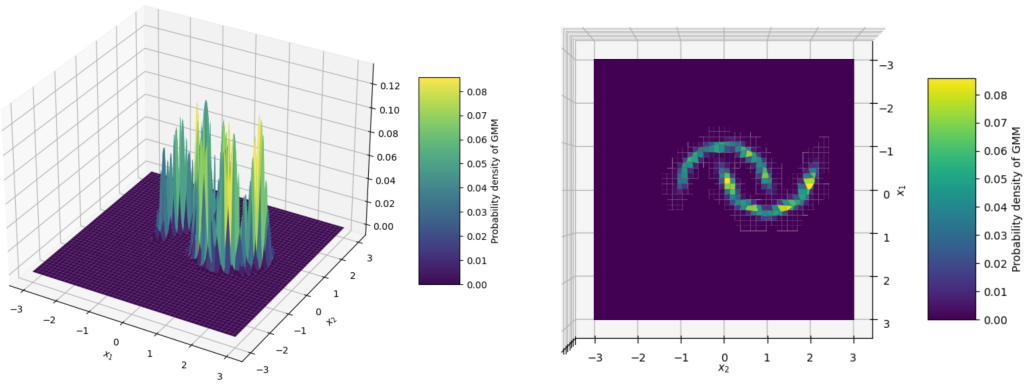


Fig. 1.2.8 (left), Fig. 1.2.9 (right): 9-component GMM 3D probability surface.

2 GMM as classification algorithm on two-moons dataset

We now use the labels provided in the two-moons dataset to use GMM as a classifier.

2.1 Classifying the two-moons dataset using GMMs

The two-moons dataset comes with a set of labels corresponding to which of the two moons a generated data point is part of. We graph this data below (*Fig. 2.1.1*). The axes correspond to the two dimensions of the features of the dataset, and the color corresponds to the label. Blue is class 0 and red is class 1.

We will now try to classify these data points using GMMs. As described in (1.1), we will use expectation maximization (EM) to classify the data points as belonging to either class 0 or 1. We will split the dataset into a train set and a test set with a test size of 20%. We want the size of the training set to be as large as possible, while still having enough data in the test set for us to test whether the model is working. If our GMM could predict the class of 1000 random samples with perfect accuracy, it would be worth it to compromise 80% of the dataset to training. Using EM, we will classify such that the point $X_i = (x_{i,1}, x_{i,2})$ belongs to the class $\text{argmax}_{\ell \in \{0,1\}} r_{i,\ell}$, where $r_{i,\ell} = P(y = \ell \mid X = x_i)$ is the responsibility of component ℓ for data point x_i. Then we will graph the results (*Fig. 2.1.2*), coloring based on predicted class (blue is 0 and red is 1.) We also plot the confusion matrix of the GMM.

First, for sake of demonstration, we train and test our GMM using only two Gaussian components.

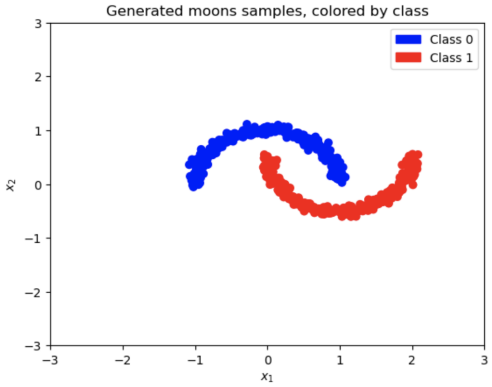


Fig. 2.1.1: Generated moons samples, colored by class.

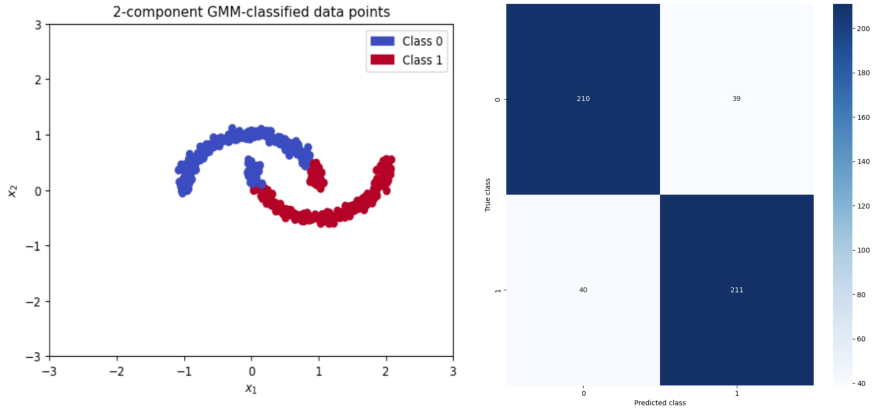


Fig. 2.1.2: Predicted classes (left) and confusion matrix (right) of $k = 2$ component GMM.

We observe that while using only 2 Gaussian components, we do not obtain an especially accurate classification of the data. As discussed in section 1.2, this is as we should expect given the complex shape of the data. But, if we use were to use more than 2 components in our GMM, then we would have more components than classes (of which there are 2). We account for this with the following "majority rule": we calculate the density of each Gaussian component for each sample by class, calculate the sum of those densities for each class, and then assign the component to the class with the higher of the two sums (indicating more points of that class lie within a smaller standard deviation of that component). Then, our GMM will simply classify each data point according to which class the component it lies in corresponds to (according to the majority rule).

In order to decide how many components our GMM should use in order to best classify the dataset, we employ a simple *accuracy metric*, one of the most straightforward metrics in machine learning, in which we simply compute the ratio of the number of accurate predictions to the number of total predictions. The accuracy method works well to determine the success of a model performing binary classification into classes of roughly equal size (the two-moons data admits such a classification); we test this accuracy using a cross-validation procedure known as *stratified K-folds cross-validation*. In K-folds cross validation, the dataset is partitioned into K subsets (i.e. "folds") without regard for the distribution of classes within each subset, in which case it is possible that the distribution of classes in the folds might not accurately represent the overall class distribution. Since we wish to ensure that the class distribution within our training and validation (test) sets are consistent with the overall class distribution of the dataset, we use *stratified K-folds* cross-validation, in which the dataset is partitioned into K subsets while maintaining the same class distribution in each subset as in the whole dataset. In addition to the accuracy metric, we double-check using another measure of predictive performance known as the *F1 score*, which is the harmonic mean between the *precision* (ratio of correctly predicted positives to total predicted positives, i.e., $\frac{\#\text{ true positives}}{\#\text{ true positives} + \#\text{ false positives}}$) and the *recall* ($\frac{\#\text{ true positives}}{\#\text{ true positives} + \#\text{ false negatives}}$) of our model. In machine learning, there is often a tradeoff between precision and recall, so a high F1 score (the harmonic mean of both values) implies a good balance between the two, and therefore a well-performing model.

We compute and graph the accuracy and F1 scores relative to different numbers k of components in our GMM (*Fig. 2.1.3*).

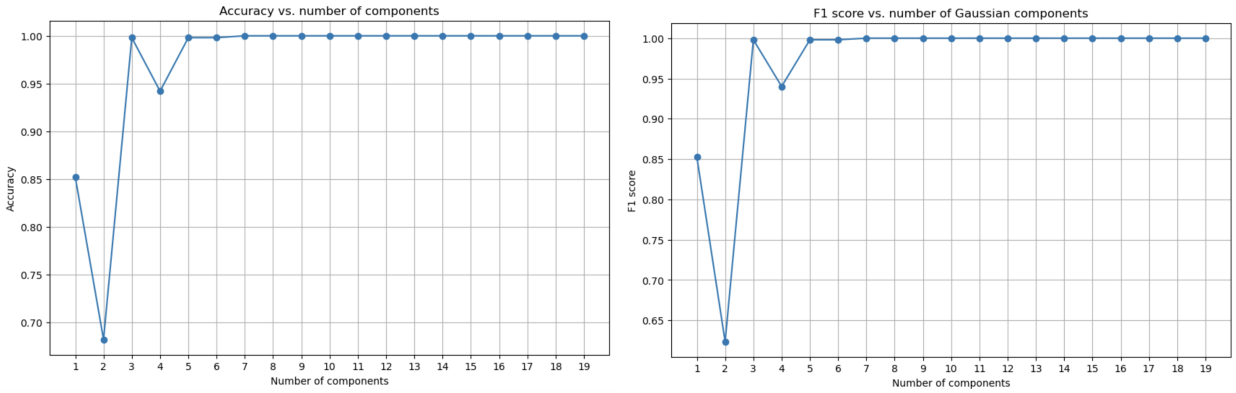


Fig. 2.1.3: Graphing the accuracy and F1 scores relative to different numbers of components in GMM.

Taking $k = 9$, we get a model that predicted 500 samples with 100% accuracy. We graph our predicted classes in the case of 500 samples and the level sets of the corresponding GMM (*Fig. 2.1.4, see next page*), and also compute the confusion matrix of said GMM after it predicted the classes of the dataset (*Fig. 2.1.5, see next page*).

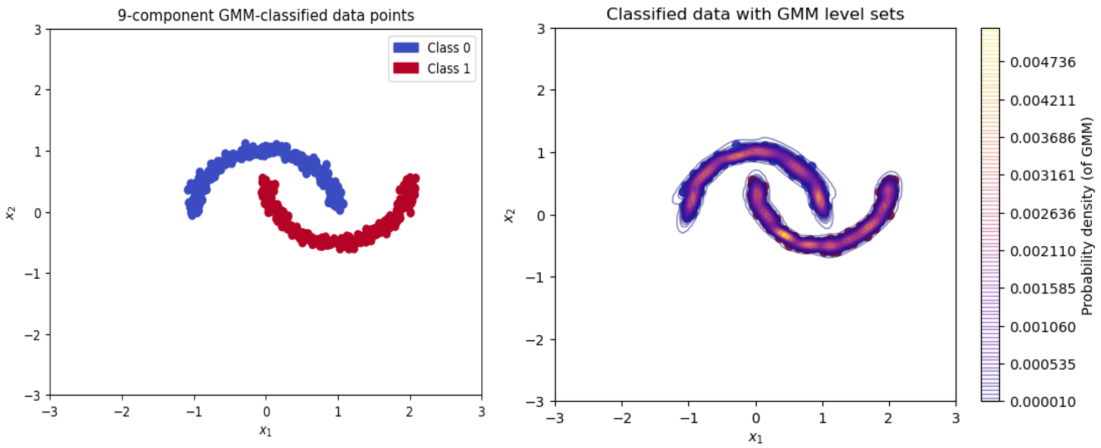


Fig. 2.1.4: Optimized-GMM-predicted classes (left) and level sets of the 9 Gaussian components (right).

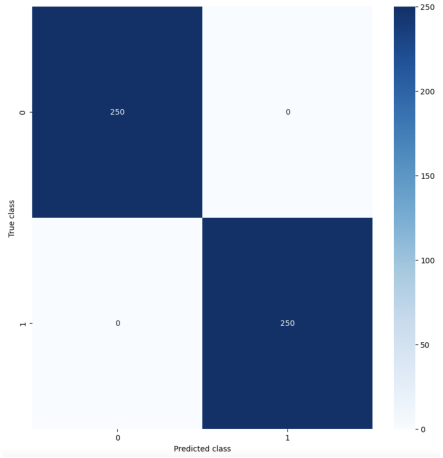


Fig. 2.1.5: Confusion matrix of the GMM (after optimizing for number of components) on the test data.

2.2 Generating new data points using our trained GMM

Now that we have the trained Gaussian mixture model (with the optimal number of components, $k = 9$), we are able to generate samples from it. This is because it is relatively simple to generate points from a Gaussian distribution, and therefore from a GMM. We will generate 500 data points from our trained GMM, graph these data points, and observe that they approximately reproduce the two moons (*Fig 2.2.1*), except for a scarce few outliers. The red/blue dots are the original dataset (colored by class), and the orange dots are our generated samples.

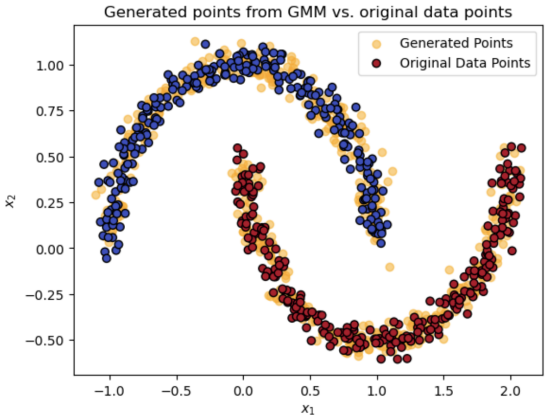


Fig. 2.2.1: Data points generated by our GMM (orange) compared to the true dataset (red and blue).

2.3 A remark on the number of samples and the Gaussian noise added to them

In the experiments and calculations above, we generated a two-moons dataset containing $n = 500$ samples, i.e., our final model (optimized for number of Gaussian components) was trained on 400 samples and tested on 100. Furthermore, a Gaussian noise with a standard deviation of $\sigma = 0.05$ was added to the original dataset. Fixing $\sigma = 0.05$, no significant difference was observed when using much more or fewer samples without changing the train/test split ratio of 4:1; specifically, our trained and optimized GMMs classified the test data (of $n/5$ samples) with 100% accuracy in the case of $n = 5000$, 500, 250, 100, and 50 samples, even in the case of changing the RNG seed. In the attached code, n and σ (`noise`) can be modified and tested to the reader's liking in the first (non-import) line of code, that is, the calling of `make_moons`.

Interestingly, if we make σ large, i.e. when our dataset is *extremely* noisy, our model (optimized for a given n, σ) is still able to classify, with decent success, a dataset for which a binary classification may be indiscernable to the eye. For example: when the average observation is a full standard deviation away from the mean, $\sigma = 1$, and we generate $n = 500$ samples, the dataset looks like one indistinguishable cloud (*Fig. 2.3.1*), and our GMM classifies it with roughly 80% accuracy (*Fig 2.3.2*):

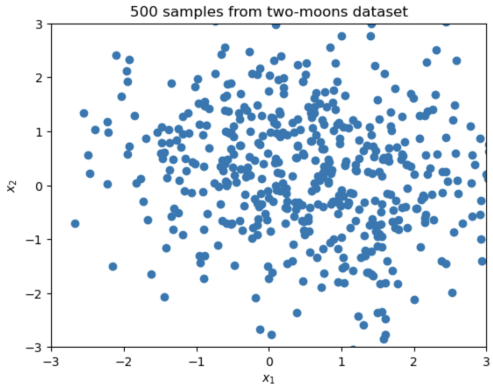


Fig. 2.3.1: Dataset with Gaussian noise added of standard deviation $\sigma = 1$.

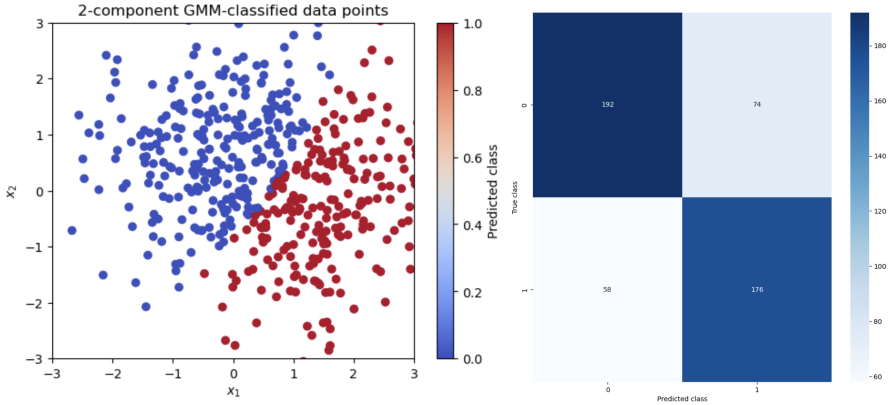


Fig. 2.3.2: Classification by optimized GMM and its confusion matrix, for $\sigma = 1$.

Of course, it does so by placing two huge ellipses next to each other at the optimal place, but it is still impressive. Furthermore, when the dataset is still very noisy, but not as extreme, for instance choosing $\sigma = 0.3$, the GMM (F1-optimized with 6 Gaussian components) classifies remarkably well (*Fig. 2.3.3, 2.3.4 see next page*).

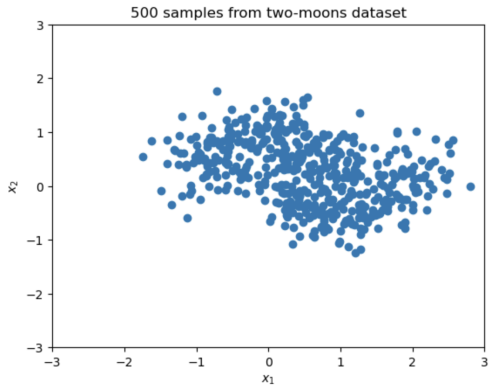


Fig. 2.3.3: Dataset with Gaussian noise added of standard deviation $\sigma = 0.3$.

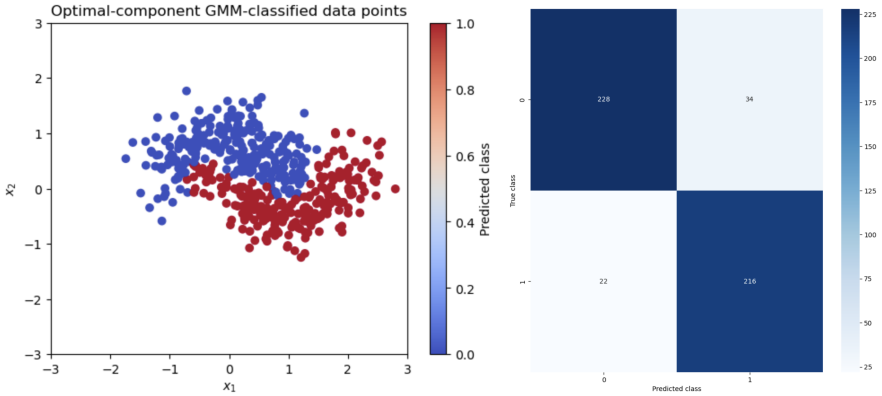


Fig. 2.3.4: Classification by optimized GMM and its confusion matrix, for $\sigma = 0.3$.

2.4 An interesting picture

During a failed attempt to graph the probability surfaces as in *Fig. 2.1.4* but with no noise added to the dataset, the author accidentally stumbled upon a beautiful picture (*Fig. 2.4.1*) that might call to mind the thought of our GMM dreaming. This image was generated by asking our GMM to classify points from the entire linear space of the graph, not just the dataset. Notice the strange semi-isometry and the stretched checkerboard-like shapes strewn about.

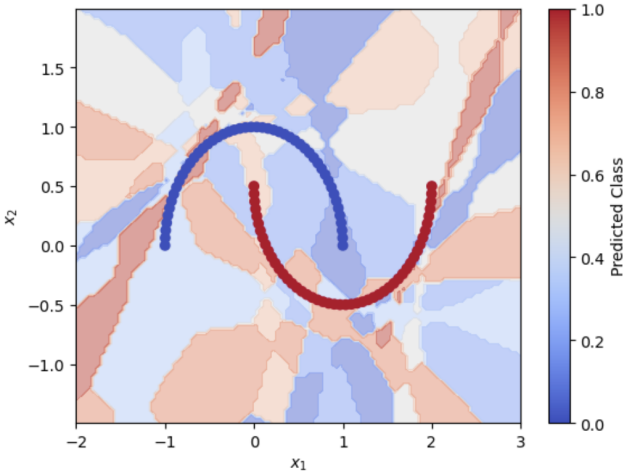


Fig. 2.4.1: What our GMM thinks the classes of random points in the graph are.

3 Unsupervised and supervised GMMs on the MNIST dataset

Now that we have demonstrated the use of GMMs to classify and estimate the probability density of the two-moons dataset, as well as generate new samples resembling the dataset, we will do all the same except we are now using the MNIST dataset. We will use the scikit-learn MNIST dataset, which contains 1,797 8x8 grayscale images of handwritten digits from 0 to 9.

3.1 Compressing the MNIST dataset using principal component analysis

Because the data points in this MNIST dataset are of a high dimension (8x8, that is, 64-dimensional), it may be slow and computationally intractible to work with the dataset as it is (not really, but just to illustrate the example of PCA.) We will compress the dataset using *principal component analysis* (PCA). In principal component analysis, we wish to reduce the dimension of the dataset while preserving as much of the dataset's "eigen-information" as possible; this is accomplished by computing the vectors which maximize the *projected variance* of the dataset, which in particular are the eigenvectors (principal components) corresponding to the largest eigenvalues of the correlation matrix of the dataset. The data points are then multiplied by the matrix which has these eigenvectors as columns, reducing the dimension of those data points to the number of eigenvectors we chose. (Typically in practice, as in this case, we normalize the dataset before performing PCA to prevent bias towards or against particular features and thereby avoid over- or under-predicting them.)

We experiment using various quantities of principal components (i.e. various values of integer d between 2 and 64, corresponding to the use of d eigenvectors) and inspect the dataset visually. Further, we observe a helpful parameter of the scikit-learn PCA method: if `n_components` (i.e. number of principal components) is passed a floating point number δ between 0 and 1, rather than an integer specifying the actual number of components, the method will instead keep only as many components as are necessary to preserve $(100 \times \delta)\%$ of the variance of the original dataset. For example, if we pass `0.95` to `n_components`, the scikit-learn PCA method will keep only as many components as is necessary to preserve 95% of the variance present in the original dataset. Recall that variance is how much the values of a particular data point deviate from the mean; that is, variance captures much of the essential shape and properties of the dataset. It turns out that we can preserve 95% of the variance in the MNIST dataset using $d = 29$ principal components.

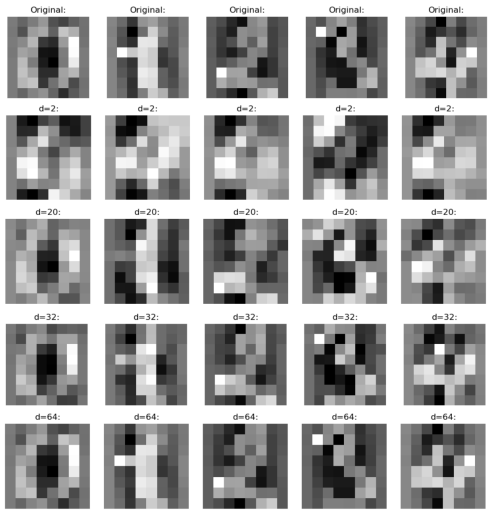


Fig. 3.1.1: The results of performing PCA on the normalized MNIST dataset using different numbers (d) of components.

We graph the results of performing PCA on the MNIST dataset using different numbers of components (d) and then scaling them up (via PCA inverse tranform) back to the original dimension (*Fig. 3.1.1*). It is clear that with very low d-values, we do not get a good approximation of the dataset. However, $d = 29$ components produces a highly accurate representation of the data while reducing its dimension by more than half, which will make training and testing a predictive model on the data faster and computationally cheaper. We opt for a 29-dimensional dataset on which to train and test our GMM.

As described, $d = 29$ is a good approximation of the dataset, as we will soon see from the success of the models we train on this dataset. Now that we have a compressed dataset, we will train several GMMs on it, firstly in a supervised context.

3.2 Using GMM(s) as a classifier on the compressed MNIST dataset

We train a k-component GMM on each class, using the labels included in the (PCA-compressed with $d = 29$ principal components) MNIST dataset (i.e., in a supervised learning context). We need at least as many Gaussian components as classes; we again use the BIC method (*Fig. 3.2.1, see next page*) to find the optimal number $k = 15$, accounting for our unique approach (namely training 10 different GMMs, one for each class).

If computational cost is a concern, the GMM trained on each class could have a different and lesser number of components, though this would involve computing a BIC for each component.

As BIC suggests, we train a 15-component GMM on each class of the MNIST dataset. We plot the means of the Gaussian components belonging to each class (*Fig. 3.2.2*), and see that they resemble the digits 0-9 (with kind of a funny-looking 1):

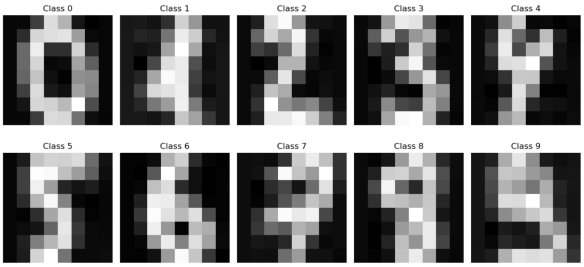


Fig. 3.2.2: The means of the GMMs belonging to each class (supervised case).

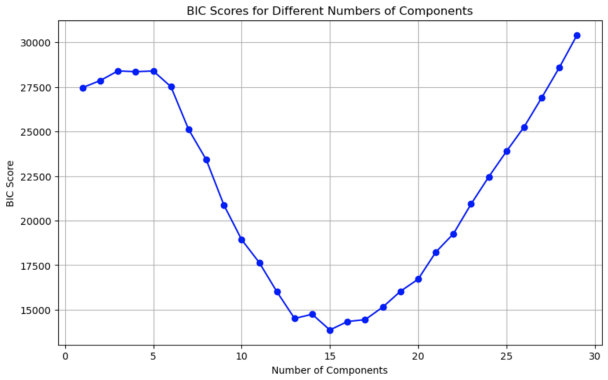


Fig. 3.2.1: BIC method to determine optimal number (k) of GMM components.

Finally, we classify the MNIST dataset using our trained GMMs as follows: for each sample in the dataset, we compute the log likelihood of the sample under each GMM. The "class" of the GMM corresponding to the highest log likelihood is considered the predicted class for that sample. We graph the predictive performance of the (combined) trained model by means of a confusion matrix, and observe our "combined GMM" (Gaussian mixture *mixture* model, patent pending) classified the 1,797-sample dataset with 100% accuracy (*Fig. 3.2.3*).

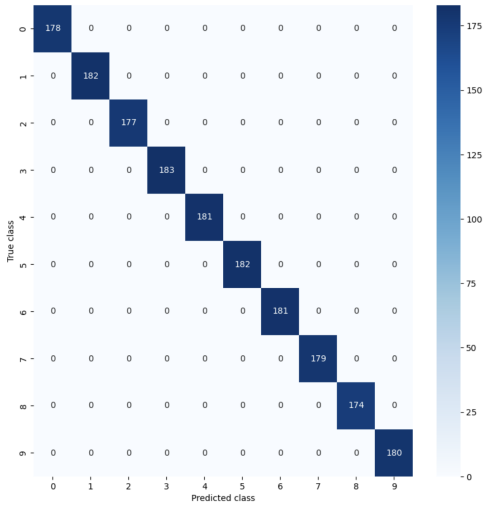


Fig. 3.2.3: The confusion matrix of our combined-GMM classifier.

3.3 Using GMM as a density estimator on the compressed MNIST dataset

We can also use a GMM in an unsupervised context on the PCA-compressed dataset, namely, to estimate the probability densities of its classes. As before, we train the model on our dataset, and then plot the means of each class (*Fig 3.3.1*). Notice that the class labels assigned by the GMM do not correspond to the true class labels, but the their means are nonetheless recognizable as digits. It is interesting to observe where the unsupervised model had trouble predicting the 10 classes in the dataset: in particular, the model had difficulty distinguishing between the digits 3 and 9, predicting that the two are one class, while predicting two different classes for the digit 6.

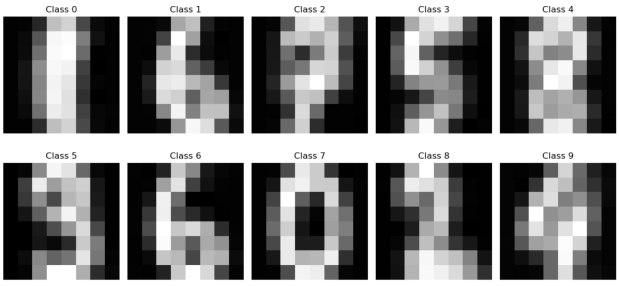


Fig. 3.3.1: The means of each class according to our unsupervised GMM as trained on 29-principal-component MNIST dataset. Notice the combined 3-9 in Class 5, and that Class 2 and Class 6 both look like the number 6.

We let the unsupervised model predict the classes of the dataset and compute the resulting confusion matrix (*Fig 3.3.2*) to evaluate its predictive performance. Note that the class labels assigned by the GMM do not correspond to the true class labels, so the confusion matrix is not as immediately interpretable as in the supervised case (in particular, it will not be diagonal). However, we can still see that the GMM is predicting an "unordered" classification, at least of some classes, with decent accuracy. In particular, it is interesting to observe which digits are more likely to be predicted incorrectly by the unsupervised GMM, although this is again hard to interpret reading the confusion matrix of the unsupervised model. Nonetheless, a careful eye can see that the model had the hardest time classifying what it thought was "class 5" (i.e. some combination of 3 and 9), and that it split half the samples with true class 6 into its two (erroneous) distinct classes for 6-shaped digits.

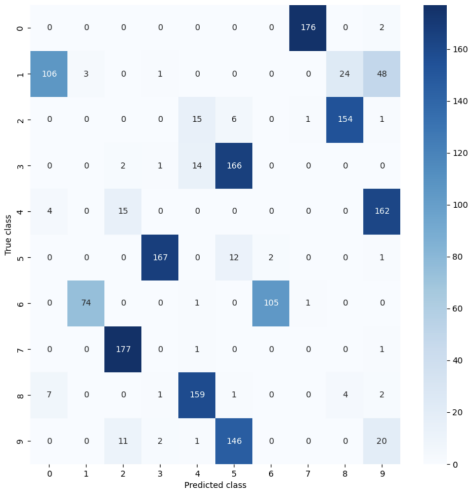
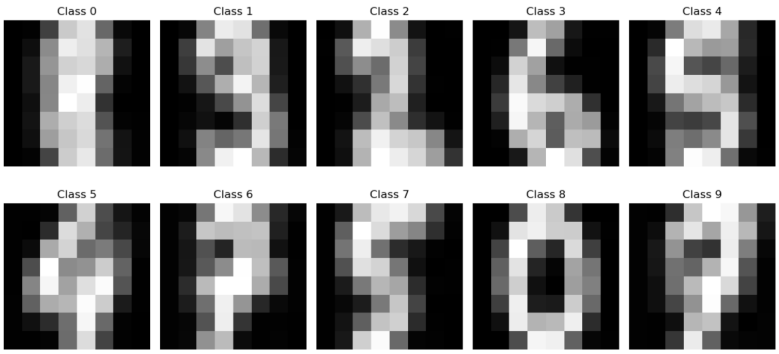


Fig. 3.3.2: The confusion matrix of our unsupervised GMM, trained on data with 29 principal components (means in fig. 3.3.1, see previous page).

It turns out that we took too few PCA components for our unsupervised model to handle. Currently we keep 29 principal components to retain 95% of the variance of the original dataset. But even if we retain, for example, 99% of the variance of the original dataset using 41 principal components, our unsupervised model doesn't classify data much better, although it does better predict the 10 means as the distinct digits 0-9, with some trouble distinguishing 5 from 8 and a little trouble between 7 and 9. We again plot the means of its predicted classes (*Fig. 3.3.3*), let it classify the dataset, and plot the resulting confusion matrix (*Fig. 3.3.4*).



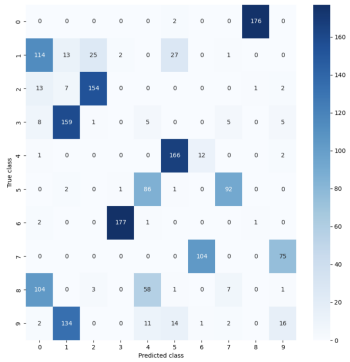


Fig. 3.3.3 (left): The means of each class according to our unsupervised GMM as trained on data with 41 principal components.

Fig. 3.3.4 (right): The confusion matrix obtained by classifying using that unsupervised GMM.

3.4 Using the trained GMM to generate data new points

Now that we have trained GMMs on the MNIST dataset, we can generate and plot new samples from the GMMs and see that they closely resemble data from the MNIST dataset, but there is still a problem. The GMM is *memorizing* the test data, meaning the variances of the Gaussian components in the GMM are extremely low (their probability surfaces have very high peaks). This results in the fact that the GMM generates almost *exactly* the same data points multiple times. We generate and plot 20 samples from our GMM (*Fig. 3.4.1*) and observe the problem of memorization is pretty extreme. We attempted to reduce this problem through various means, namely

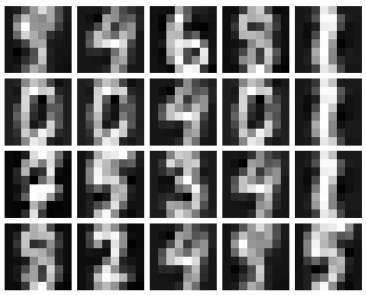


Fig. 3.4.1: Samples generated by the trained supervised GMM.

by trying different numbers of principal components, of Gaussian components, changing the properties of the covariances of those Gaussian components, etc., but we were not able to find a solution on the supervised model. Our unsupervised model, on the other hand, was capable of generating lots of unique samples, but this required a high dimensionality even after PCA (namely 41 principal components to preserve 99% of the variance of the original dataset in order to get good results), and even then, the generated samples are not perfect. Still, they are largely discernable as digits (*Fig. 3.4.2*), although not all of them would be suitable as training data.

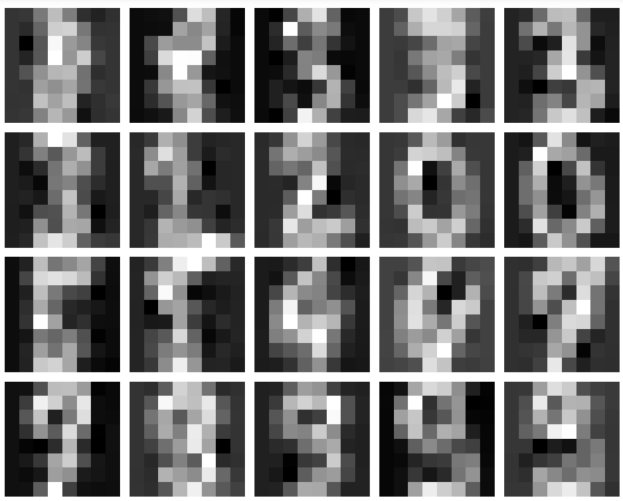


Fig. 3.4.2: Samples generated by our BIC-optimized, unsupervised GMM.

4 Comparing our GMM classifier's predictive performance with that of other classification methods

Now that we have seen the success of GMMs on the MNIST dataset, we will compare their predictive performance with that of several other classification methods, namely using gradient boosted decision trees and random decision forests. This will give us additional benchmarks to gauge how well other algorithms, albeit without their hyperparameters optimized as we did for GMM, perform on the same data.

4.1 Gradient boosted decision tree classification

We implement a gradient boosted decision tree classifier using scikit-learn's `GradientBoostingClassifier`. We initialize a simple model, in this case a constant model, and compute the *residual*, the difference between the actual class label and the prediction made by the model, for each sample in the dataset. We then train a decision tree, but instead of predicting the actual target value, it tries to predict the residuals from the previous step, thus modeling the errors of the current model. The predicted residuals from this new tree are then added to the existing predictions. This process repeats iteratively until the algorithm converges or a specified number of iterations have taken place. We fit `GradientBoostingClassifier` to the same MNIST dataset as before using 100 iterations with log-loss (same as in logistic regression) as the loss function to be optimized (we optimize by gradient descent, hence the name) and an 80/20 train/test split. We then compute the confusion matrix of the decision tree model (*Fig. 4.1.1*). Our supervised GMM from section 3 performed better on this dataset (*Fig. 3.2.3*).

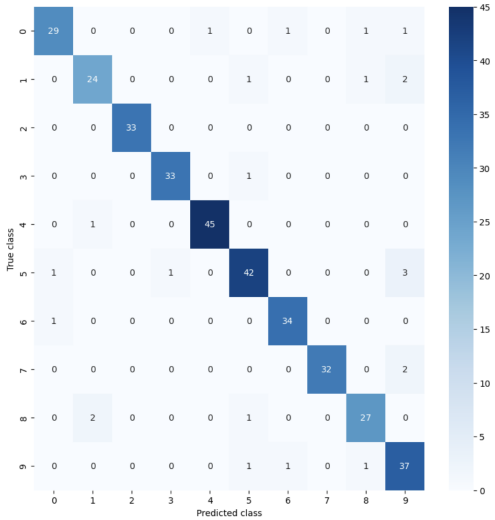


Fig. 4.1.1: Confusion matrix of the gradient boosted tree classifier.

4.2 Random decision forests

A random decision forest classifier is an algorithm in which many iterated decision trees, each trained on a sampled-with-replacement subset of the dataset, democratically vote to predict the class of an observation. We might expect it to be able to well classify the MNIST dataset, since random forest models are described as correcting for decision trees' habit to overfit to the dataset[2]. In this case, MNIST is a highly complex dataset of high dimensionality, even after using PCA to reduce. Furthermore, there are 10 classes, with a wide difference between samples within each class. Will a random forest classifier perform better than our GMM?

We implement a 100-tree random decision forest classifier using scikit-learn's `RandomForestClassifier`. We fit it to the dataset, train it on 80% of the data, test it, and plot the resulting confusion matrix. (*Fig. 4.2.1*). The random forest classifier performs very well, but again, our supervised GMM from section 3 performed better on this dataset (*Fig. 3.2.3*).

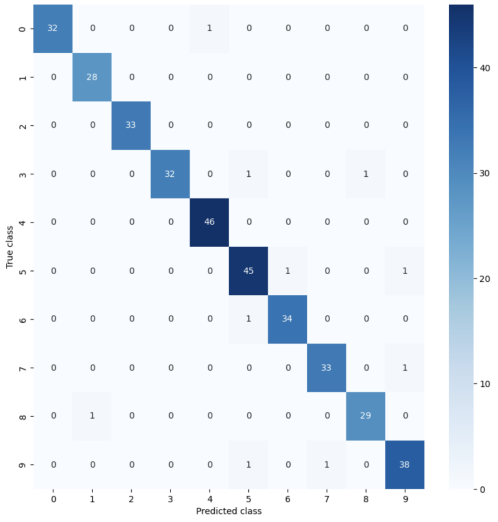


Fig. 4.2.1: Confusion matrix of the random decision forest classifier.

However, we did not spend a significant time tuning the many hyperparameters of a random forest for this model, for the scope of this report, so this is not representative of the possible predictive success of a highly optimized random decision forest classifier on the dataset. In particular, we only attempted to modify the number of trees, their maximum depth, and the minimal number of samples required to split a leaf node or comprise a leaf node. It is possible that through optimizing these and other hyperparameters, this random forest classifier might classify the dataset as accurately as our trained GMM does. It is impressive that the random forest classified the dataset so well with essentially no optimization of its hyperparameters from those assigned to it by default.

5 References

The following methods and libraries used in this report were developed by scikit-learn: `GaussianMixture`, `moons` (two-moons dataset), `load_digits` (MNIST dataset), `train_test_split`, `confusion_matrix`, `BIC`, `StandardScaler`, `PCA`, `accuracy_score`, `f1_score`, `StratifiedKFold`, `GradientBoostingClassifier`, and `RandomForestClassifier`. In each case, the author read the thorough documentation and source code of these methods to ensure not only their appropriate application, but to accurately interpret the data which these methods produced. Documentation and source code of scikit-learn methods can be accessed at https://scikit-learn.org/stable/.

[1] https://scikit-learn.org/stable/modules/mixture.html

[2] https://en.wikipedia.org/wiki/Random_forest

https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm

https://en.wikipedia.org/wiki/Cross-validation_(statistics)

https://en.wikipedia.org/wiki/Accuracy_and_precision

https://en.wikipedia.org/wiki/Precision_and_recall

https://en.wikipedia.org/wiki/Decision_tree_learning#Decision_tree_types

https://en.wikipedia.org/wiki/Gradient_boosting#Gradient_tree_boosting