Gaussian Mixture Models

A Deeper Analysis

We'll start by focusing on the scalability issues

We have established that KDE has trouble with:

- Large dimensional datasets
- Large number of training examples

Can you make a guess about the root of the problem?

A Deeper Analysis

We'll start by focusing on the scalability issues

We have established that KDE has trouble with:

- Large dimensional datasets
- Large number of training examples

Can you make a guess about the root of the problem?

KDE makes no attempt to "compress" the information from the training data:

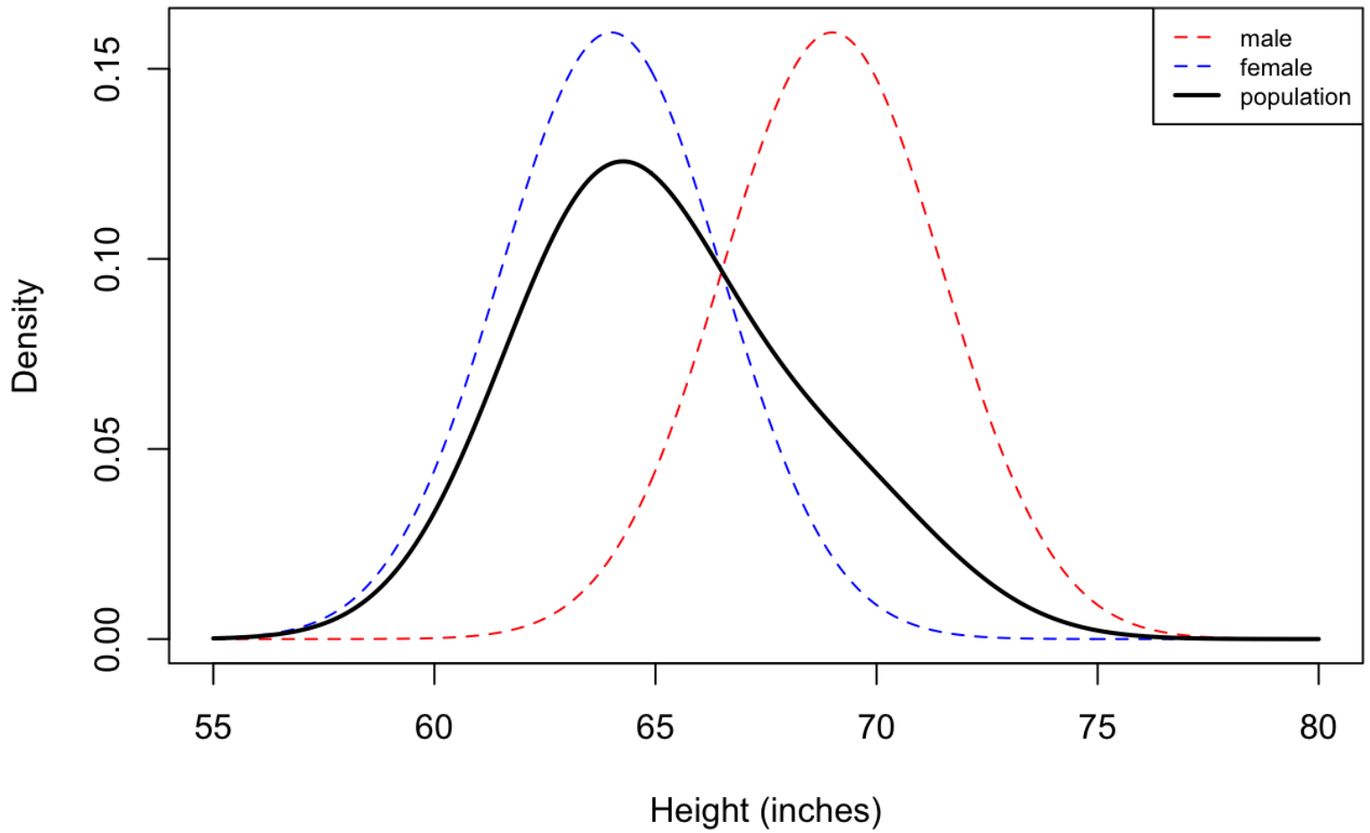
- The size of a KDE models grows directly with the training set size
- In statistical terms, KDE has very little bias and a very large variance

It's time to introduce a new density estimation technique

Gaussian Mixture Models

In particular, we'll now switch to using Gaussian Mixture Models (GMMs)

A GMM describes a distribution via a weighted sum of Gaussian components



Gaussian Mixture Models

In particular, we'll now switch to using Gaussian Mixture Models (GMMs)

A GMM describes a distribution via a weighted sum of Gaussian components

- The model size depends on the dimensionality and on #components
- The #components can be chosen, to control the bias/variance trade-off

Formally, we assume data is generated by the following probabilistic model

$$X_Z$$

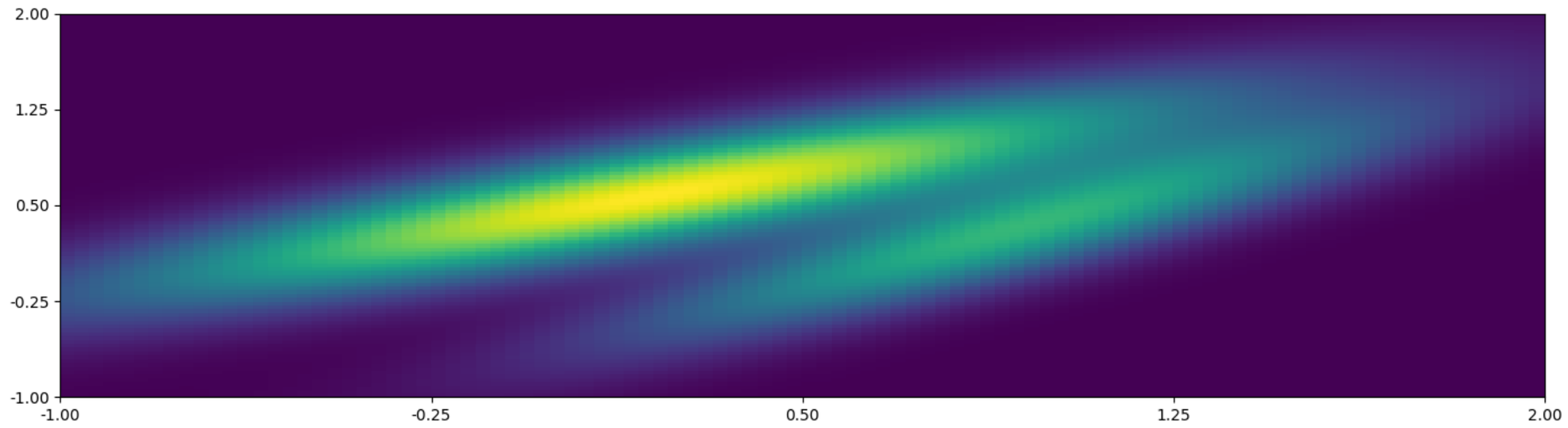
- Z e X_k are both random variables
- Z represents the index of the component that generates the sample
- X_k follows a multivariate Gaussian distribution

In other words, a GMM is a selection-based ensemble

A GMM Example

Let's build a (random) GMM in two dimensions so see an example

In [2]:
```
gt = util.generate_gmm_dist(n_components=2, seed=59)
util.plot_density_estimator_2D(gt, xr=np.linspace(-1, 2, 100), yr=np.linspace(-1, 2, 100), figsi
```



- Our example has two components, each with its own mean and covariance
- One component is slightly less prevalent than the other

GMM Parameters

The PDF of a GMM is given by:

$$g(x, \mu, \Sigma, \tau) = \sum_{k=1}^{n} \tau_k f(x, \mu_k, \Sigma_k)$$

- f is the PDF of a multivariante Normal distribution
- μ_k is the (vector) mean and Σ_k the covariance matrix for the k-th component
- τ_k corresponds to $P(Z = k)$

We can inspect the values for our example GMM

In [3]:
```
print('\\tau:', gt.weights)
print('\\mu', str(gt.mu).replace('\n', ' '))
print('\\sigma', str(gt.sigma).replace('\n', ' '))
```

```
\tau: [0.69756198 0.30243802]
\mu [[0.20612642 0.58696692]  [0.94152811 0.3112852 ]]
\sigma [array([[0.5610369 , 0.3646768 ],        [0.3646768 , 0.31593376]]), array([[0.2986275
4, 0.29550211],        [0.29550211, 0.35832187]])]
```

Sampling from GMMs

When we want to sample from a GMM

- First we need to sample the Z variable
- Then we sample from the corresponding multivariate distribution

In [4]:
```
train_x, train_z = gt.sample(1000, seed=42)
test_x, test_z = gt.sample(1000, seed=42)
```

Hence, we don't get to now just the sample value

...But also which of the Gaussian components it was generated by

In [5]:
```
print('z values:', train_z[:4])
print('x values:', train_x[:4])
```

```
z values: [0 1 1 0]
x values: [[ 0.25595526  0.24144331]
 [ 0.65952904  0.16087875]
 [ 0.50240258  0.11145718]
 [-0.47036887  0.2317656 ]]
```

Training a GMM

We can train a GMM to approximate other distributions

The training problem can be formulated in terms of likelihood maximization

$$\arg\max_{\mu,\Sigma,\tau} \; \mathbb{E}_{\hat{x}\sim X}\left[L(\hat{x}, \mu, \Sigma, \tau)\right]$$

$$\text{s.t.} \sum_{k=1}^{n} \tau_k = 1$$

- As usual, the likelihood function L measures how likely it is...
- ...that the training sample $\hat{x}$ is generated by a GMM with parameters μ, Σ, τ

We can approximate the expectation by using the training set

$$\mathbb{E}_{\hat{x}\sim X}\left[L(\hat{x}, \mu, \Sigma, \tau)\right] \simeq \prod_{i=1}^{m} g(\hat{x}_i, \mu, \Sigma, \tau)$$

Training a GMM

Let's put everything together

$$\arg\max_{\mu,\Sigma,\tau} \prod_{i=1}^{m} \sum_{k=1}^{n} \tau_k f(x, \mu_k, \Sigma_k)$$

$$\text{s.t.} \sum_{k=1}^{n} \tau_k = 1$$

From an optimization point of view, this is very annoying problem:

- There's a constraint
- There's both a product and a sum
- The product cannot be decomposed (μ, Σ, τ appear in every term)

So we'll need to get clever!

An Apparent Overcomplication

We get clever by apparently overcomplicating the problem

In particular, we introduce a random variable Z_i for each example

- $Z_i = k$ iff i-th example was drawn from the k-th component
- The Z_i variables are latent, since we do not (really) know their value

With the new variables, the PDF becomes:

$$\tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau) = \tau_{z_i} f(x, \mu_k, \Sigma_k)$$

- The PDF is now specific for each example and does not contain a sum
- The value z_i is now an input to $\tilde{g}_i$
- ...And we can use it as an index to retrieve the correct τ_k

An Apparent Overcomplication

We now need to compute the likelihood expectation over both X and Z

$$\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}\left[L(\hat{x},\hat{z},\mu,\Sigma,\tau)\right]$$

- We can deal with X by using the training set as the single sample
- It's only one, but at least it is large

By doing this we obtain:

$$\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}\left[L(\hat{x},\hat{z},\mu,\Sigma,\tau)\right] \simeq \mathbb{E}_{\hat{z}\sim Z}\left[\prod_{i=1}^{m}\tilde{g}_i(x_i,z_i,\mu,\Sigma,\tau)\right]$$

- We cannot use the same technique for Z
- ...Since the values of the Z_i variables are unknown

An Apparent Overcomplication

To deal with the expectation on Z, we add yet another set of variables

- The variables represent the (unknown) distribution of the latent Z_i variables
- In particular, $\tilde{\tau}_{i,k}$ corresponds to $P(Z_i = k)$

With the new variable, we can compute the expectation in closed form:

$$\mathbb{E}_{\hat{x}\sim X,\hat{z}\sim Z}\left[L(\hat{x},\hat{z},\mu,\Sigma,\tau)\right] \simeq \prod_{i=1}^{m}\prod_{k=1}^{n}\tilde{g}_i(x_i,z_i,\mu,\Sigma,\tau)^{\tilde{\tau}_{i,k}}$$

- Intuitively, we generate $\tilde{\tau}_{i,k}$ samples for each example i and component k
- ...Then we multiply their densities as usual

An Apparent Overcomplication

The reworked training problem therefore is

$$\arg\max_{\mu,\Sigma,\tau,\tilde{\tau}} \prod_{i=1}^{m}\prod_{k=1}^{n} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)^{\tilde{\tau}_{i,k}}$$

$$\text{s.t.} \sum_{k=1}^{n} \tau_k = 1$$

$$\text{s.t.} \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1 \qquad \forall i = 1..m$$

We have even more variables (the $\tilde{\tau}_{i,k}$ ones)

- ...But there's no longer a combination of sums and products
- And even more importantly...

Expectation-Maximization

...We can now use the Expectation-Maximization algorithm

The EM algorithm is an optimization method based on alternating steps

- In the expectation step:
  - We consider μ, Σ, τ as fixed and we optimize over $\tilde{\tau}$
  - We compute the expectation over Z in a symbolic form
- In maximization step:
  - We consider $\tilde{\tau}$ as fixed and we optimize over μ, Σ, τ

The method stops when likelihood improvement become too small

- It can be proved to converge to a local optimum
- ...Under reasonable assumptions

Expectation-Maximization

Let's see the expectation step in our case

- This is where we handle the latent variables we introduced
- We consider μ, Σ, τ fixed, so that we need to solve:

$$\arg\max_{\tilde{\tau}} \prod_{i=1}^{m} \prod_{k=1}^{n} \tilde{g}_i(x_i, z_i, \mu, \Sigma, \tau)^{\tilde{\tau}_{i,k}}$$

$$\text{s.t.} \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1 \qquad \forall i = 1..m$$

- The optimization problem can be easily decomposed
- ...So we can optimize over each example individually

Expectation-Maximization

Since τ is known, the expectation over Z can be computed exactly

By substituting $\tilde{g}_i$, for a single example i we have:

$$\arg\max_{\tilde{\tau}} \prod_{k=1}^{n} (\tau_k f(x, \mu_k, \Sigma_k))^{\tilde{\tau}_{i,k}}$$

$$\text{s.t.} \sum_{k=1}^{n} \tilde{\tau}_{i,k} = 1$$

Which (since μ, Σ, τ are fixed) is solved by choosing:

$$\tau_{i,k} = \frac{\tau_k f(\hat{x}_i, \mu_k, \Sigma_k)}{\sum_{h=1}^{n} \tau_h f(\hat{x}_i, \mu_k, \Sigma_k)}$$

I.e. the relative density for component k

Expectation-Maximization

For the maximization step the math is bit more difficult

So we provide only a the main ideas

Each τ_k is optimized by relative sum of the corresponding $\tilde{\tau}_{i,k}$ variables:

$$\tau_k = \frac{1}{m} \sum_{i=1}^{m} \tilde{\tau}_{i,k}$$

- In fact, the latent variables represent samples drawn from the Z_k variables

The μ_k and Σ_k parameters can be estimated based on classical methods

- In particular, we give to each example a sample weight equal to $\tilde{\tau}_{i,k}$
- Then we estimate μ and Σ via a Least Square approach

GMM in Action

There are many implementations and variants of the EM method

We will use the code from scikit-learn:

In [6]:
```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=2, random_state=4)
gm.fit(train_x);
```

- The API is the usual one

We need to specify the number of components a priori

- We can tune it using a maximum likelihood approach on a validation set
- ...Or using other criteria (e.g. elbow method)


Inspecting the Results

Let's inspect the learned parameters

In [7]:
```
print('Learned weights', gm.weights_)
print('True weights', gt.weights)
```

```
Learned weights [0.65497081 0.34502919]
True weights [0.69756198 0.30243802]
```

In [8]:
```
print('Learned means', str(gm.means_).replace('\n', ' '))
print('True means', str(gm.means ).replace('\n', ' '))
```

```
Learned means [[0.06381907 0.54345656]  [0.9698073  0.40784152]]
True means [[0.06381907 0.54345656]  [0.9698073  0.40784152]]
```

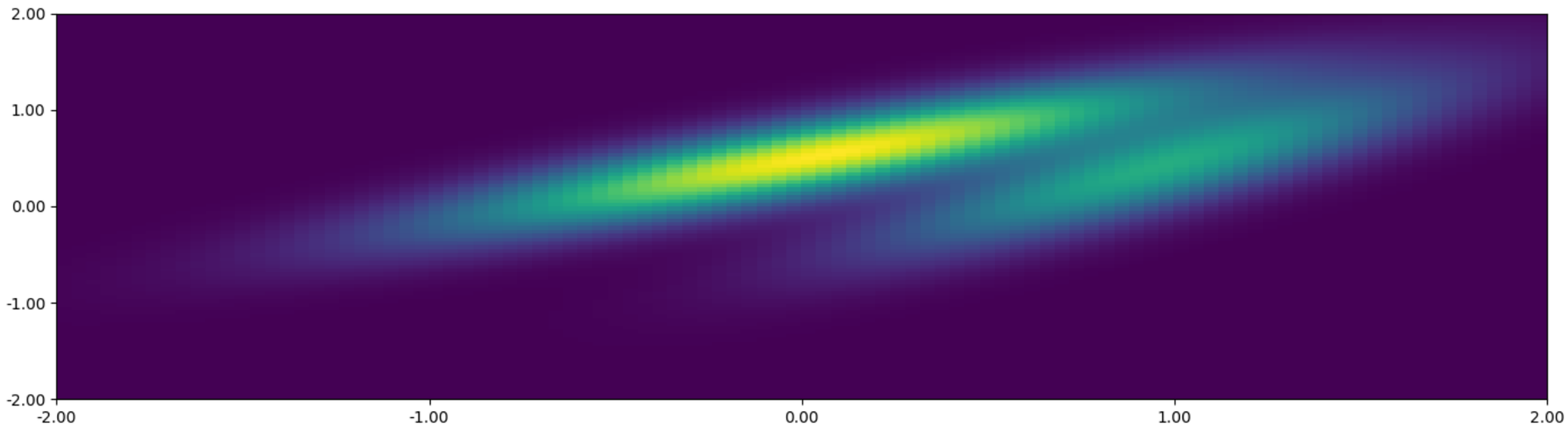
In [9]:
```
print('Learned covariance #1', str(gm.covariances_[0]).replace('\n', ' '))
print('True covariance #1', str(gt.sigma[0]).replace('\n', ' '))
print('Learned covariance #2', str(gm.covariances_[1]).replace('\n', ' '))
print('True covariance #2', str(gt.sigma[1]).replace('\n', ' '))
```

```
Learned covariance #1 [[0.47395009 0.31851071]  [0.31851071 0.28472541]]
True covariance #1 [[0.5610369  0.3646768 ]  [0.3646768  0.31593376]]
Learned covariance #2 [[0.26663143 0.2615946 ]  [0.2615946  0.35002731]]
True covariance #2 [[0.29862754 0.29550211]  [0.29550211 0.35832187]]
```

Inspecting the Results

Here is the approximated PDF

In [10]:
```
util.plot_density_estimator_2D(gm, xr=np.linspace(-2, 2, 100), yr=np.linspace(-2, 2, 100), figsi
```



Which Kind of Prediction

GMMs are very flexible in terms of what they can do

We can use them to evaluate the (log) density of a sample:

In [11]:
```
pred_lf = np.exp(gm.score_samples(train_x))
print('Log densities:', pred_lf[:3])
```

```
Log densities: [0.17614612 0.29057101 0.21176569]
```

We can use them to generate a sample:

In [12]:
```
pred_x, pred_z = gm.sample(3)
print('Sampled values:', str(pred_x).replace('\n', ' '))
print('Sampled components:', pred_z)
```

```
Sampled values: [[ 0.74135464  0.15453881]  [ 0.31047866 -0.01835778]  [ 0.31015942 -0.4740671
2]]
Sampled components: [1 1 1]
```


More then Densities

GMMs are very flexible in terms of what they can do

We can use them for the probability that a sample belongs to a component

In [13]:
```
pred_p = gm.predict_proba(train_x)
print('Probability of belonging to a component:')
print(pred_p[:3])
```

```
Probability of belonging to a component:
[[0.83493525 0.16506475]
 [0.0176366  0.9823634 ]
 [0.05233837 0.94766163]]
```

- The approach is the same we used to optimize $\tilde{\tau}_{i,k}$ in the expectation step

By picking the maximum probability, we can assign samples to a component

In [14]:
```
pred_c = gm.predict(train_x)
print(pred_c[:3])
```

```
[0 1 1]
```

More then Densities

GMMs can certainly act as density estimators

...But can do much more!

- Sampling
- Component assignment
- ...And therefore clustering

This is so true that GMM are often presented as a generalization of k-means

And this (partially) addresses the last limitation of KDE

- By choosing certain density estimator
- ...We can obtain additional information in addition to the densities