

Generative Neural Networks for the Sciences - Sample Solution

This sample solution is based on the submission of Paul Saenger, Nikita Tatsch and Christian Kleiber. We fixed some bugs, added the DensityForest and vectorized the MMD computation (huge speed-up).

Task 1: Two-dimensional data

```
In [1]: import numpy as np

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.cluster import kmeans_plusplus # For GMM initialization
from sklearn.datasets import make_moons, make_lobes
from scipy.spatial.distance import cdist
from tqdm import tqdm # progress bar

import pandas as pd
import matplotlib.pyplot as plt

Code readability is improved when each model type is implemented as a class.
```

```
In [2]: # Implement a custom 2D histogram compatible with the sklearn API
class Histogram(BaseEstimator, TransformerMixin):
    def __init__(self, bins: tuple = None, range_: tuple = None):
        if bins is None:
            bins = (10, 10)
        else:
            assert len(bins) == 2, "bins must be a tuple of length 2"
            self.bins = bins
        self.range_ = range_
        self.histogram_ = np.zeros(self.bins)

    def fit(self, X: np.ndarray):
        # Calculate the range in x and y directions
        if self.range_ is None:
            self.range_ = ((X[:, 0].min(), X[:, 0].max()), (X[:, 1].min(), X[:, 1].max()))

        # Calculate the bin edges
        self.bin_edges = [
            np.linspace(self.range_[0][0], self.range_[0][1], self.bins[0] + 1), # x
            np.linspace(self.range_[1][0], self.range_[1][1], self.bins[1] + 1) # y
        ]

        # Create an empty histogram
        self.histogram_ = np.zeros(self.bins)

        # Calculate the bin indices for each sample
        # Use numpy.digitize to bin the data (faster than for loops)
        bin_indices = np.array([
            np.digitize(X[:, 0], self.bin_edges[0]) - 1,
            np.digitize(X[:, 1], self.bin_edges[1]) - 1,
        ])

        # Count the number of samples in each bin
        for i in range(self.bins[0]):
            for j in range(self.bins[1]):
                self.histogram_[i, j] = np.sum(bin_indices[0] == i & (bin_indices[1] == j))

        self.histogram_ /= np.sum(self.histogram_)

    def predict(self, X: np.ndarray):
        # For a given sample, find the bin it belongs to and return the bin's value
        bin_indices = np.array([
            np.digitize(X[:, 0], self.bin_edges[0]) - 1,
            np.digitize(X[:, 1], self.bin_edges[1]) - 1,
        ])
        values = np.zeros(X.shape[0])

        # Where the sample is outside the range, return 0
        for i in range(self.bins[0]):
            for j in range(self.bins[1]):
                values[bin_indices[0] == i & (bin_indices[1] == j)] = self.histogram_[i, j]

        return values

    def sample(self, n=1, noise=True):
        # Sample from the histogram
        # Calculate the probability of each bin
        bin_probabilities = self.histogram_ / np.sum(self.histogram_)

        # Sample from the bin probabilities
        bin_indices = np.random.choice(self.bins[0] * self.bins[1], size=n, p=bin_probabilities.flatten())

        # Convert the bin indices to x and y coordinates
        x = self.bin_edges[0][bin_indices // self.bins[1]]
        y = self.bin_edges[1][bin_indices % self.bins[1]]

        # If noise is enabled, add uniform noise to the samples (to avoid degenerate solutions)
        if noise:
            x = np.random.uniform(
                -0.5 * (self.bin_edges[0][1] - self.bin_edges[0][0]),
                0.5 * (self.bin_edges[0][1] - self.bin_edges[0][0]), size=n)
            y = np.random.uniform(
                -0.5 * (self.bin_edges[1][1] - self.bin_edges[1][0]),
                0.5 * (self.bin_edges[1][1] - self.bin_edges[1][0]), size=n)

        return np.vstack([x, y]).T
```

```
In [3]: class Gaussian(BaseEstimator, TransformerMixin):
    def __init__(self, L: int = 1):
        self.mean_ = None
        self.determinant_ = None
        self.inverse_ = None

    def fit(self, self, X: np.ndarray, y: np.ndarray):
        # Calculate the covariance matrix
        self.mean_ = np.mean(X, axis=0)
        # Calculate the covariance matrix
        self.covariance_ = np.cov(X, rowvar=False)

        # Calculate the determinant and inverse of the covariance matrix
        self.determinant_ = np.linalg.det(self.covariance_)
        self.inverse_ = np.linalg.inv(self.covariance_)

        return self

    def predict(self, X: np.ndarray):
        # Calculate the value of the Gaussian for each sample
        values = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            values[i] = np.exp(-0.5 * (X[i] - self.mean_) @ self.inverse_ @ (X[i] - self.mean_)) / np.sqrt(self.determinant_ * (2 * np.pi) ** 2)

        return values

    def sample(self, n=1):
        # First, sample from a standard normal distribution
        mu = np.random.normal(size=n, self.mean_, self.covariance_)

        # Then, transform the samples to match the covariance matrix
        # Compute the eigendecomposition of the covariance matrix U and Lambda
        eigenvalues, eigenvectors = np.linalg.eig(self.covariance_)

        # Compute the square root of Lambda
        Lambda_sqrt = np.diag(np.sqrt(eigenvalues))

        # Transform the samples via x = U Lambda_sqrt z + mu
        transformed_x = []
        for i in range(n):
            transformed_x.append(eigenvectors @ Lambda_sqrt @ x[i] + self.mean_)

        return np.array(transformed_x)
```

```
In [4]: class GMM(BaseEstimator, TransformerMixin):
    def __init__(self, L: int = 1):
        # L: The number of Gaussian components
        self.L = L
        self.weights = np.empty(L)
        self.means = None # Will be initialized after X's shape is known
        self.covariances = None
        self.inverse_ = None
        self.determinant_ = None

    def update_determinant_inverse(self):
        for i, covariance in enumerate(self.covariances):
            self.determinant_[i] = np.linalg.det(covariance)
            self.inverse_[i] = np.linalg.inv(covariance)

    def fit(self, self, X: np.ndarray, max_iter: int = 100):
        # Regularization term to prevent singular covariance matrix
        epsilon = 1e-6

        # Initialize mu_1 using kmeans++
        mu_1 = kmeans_plusplus(X, n_clusters=self.L, random_state=0)
        self.means = mu_1 # Now we initialize it with the right shape

        # Initialize the weights to be uniform
        self.weights = np.ones(self.L) / self.L

        # Initialize the covariances to diagonals
        self.covariances = np.zeros((self.L, self.L))
        self.update_determinant_inverse()

        # Begin expectation maximization
        for i in range(max_iter):
            # E-step: Calculate the responsibilities gamma_i
            responsibilities = np.zeros(X.shape[0])
            diff = X - self.means[i]
            exponent = np.einsum('ij,ij->i', np.dot(diff, self.inverse_[i]), diff)
            probabilities = self.weights[i] * np.exp(-0.5 * exponent) / np.sqrt(self.determinant_[i] * (2 * np.pi) ** X.shape[1])

            responsibilities += epsilon
            responsibilities /= np.sum(responsibilities, axis=1, keepdims=True)

            # M-step: Update the means and variances
            for i in range(self.L):
                self.weights[i] = np.sum(responsibilities[:, i]) / X.shape[0]
                # Update the means
                mu[i] = np.sum(responsibilities[:, i].reshape(-1, 1) * X, axis=0) / np.sum(responsibilities[:, i])
                self.covariances[i] = np.dot(responsibilities[:, i] * diff.T, diff) / np.sum(responsibilities[:, i])
                self.covariances[i] += np.eye(X.shape[1]) * epsilon # Regularization term

                self.weights[i] = np.sum(self.weights)
                self.update_determinant_inverse()

            self.means = mu

        return self

    def predict(self, X: np.ndarray):
        # Calculate the total probability density of each sample under the model
        probabilities = np.zeros(X.shape[0])
        for i in range(self.L):
            diff = X - self.means[i]
            exponent = np.einsum('ij,ij->i', np.dot(diff, self.inverse_[i]), diff)
            probabilities = self.weights[i] * np.exp(-0.5 * exponent) / np.sqrt(self.determinant_[i] * (2 * np.pi) ** X.shape[1])

        return probabilities

    def sample(self, n=1):
        # Sample from the GMM
        component = np.random.choice(self.L, size=n, p=self.weights)
        samples = np.array([np.random.multivariate_normal(self.means[i], self.covariances[i]) for i in range(n)])

        return samples
```

```
In [5]: class KDE(BaseEstimator, TransformerMixin):
    def __init__(self, bandwidth):
        self.bandwidth = bandwidth
        self.points = []

    def fit(self, data):
        self.points = data

        return self

    def gaussian_kernel(self, self, x0, y0):
        coeff = 1 / (2 * np.pi * self.bandwidth ** 2)
        exp_val = np.exp(-(x0 - y0) ** 2 / (2 * self.bandwidth ** 2))
        return coeff * np.exp(exp_val)

    def predict(self, X: np.ndarray):
        # Predict the value of the KDE for each point
        values = np.zeros(X.shape[0])
        for i, (xi, yi) in enumerate(X):
            # Add the value of the KDE for each point
            for x0, y0 in self.points:
                value[i] += self.gaussian_kernel(xi, yi, x0, y0)

        # Normalize the value
        values[i] /= len(self.points)

        return values

    def sample(self, n=1):
        # Sample from the KDE
        # Sample n points from the points
        points_ids = np.random.choice(len(self.points), size=n)
        points = self.points[points_ids]

        # Sample from the KDE
        samples = []
        for i in range(n):
            samples.append(np.random.normal(loc=points[i], scale=self.bandwidth))

        return np.array(samples)
```

Results

```
In [6]: n_samples_list = [20, 50, 100, 500, 1000]
X_list = [make_moons(n_samples=n_samples, noise=0.1)[0] for n_samples in n_samples_list]
X_list = [make_lobes(n_samples=n_samples_list)[0] for n_samples_list in n_samples_list]

In [7]: # Fit the models
time_hist_list = [Histogram(bins=(20, 20)).fit(X) for X in X_list]
time_gaussian_list = [Gaussian().fit(X) for X in X_list]
time_gmm_list = [GMM(L=20).fit(X, max_iter=100) for X in X_list]
time_kde_list = [KDE(bandwidth=0.1).fit(X) for X in X_list]

CPU times: total: 15.6 ms
Wall time: 21 ms
Histogram done
CPU times: total: 15.6 ms
Wall time: 6.57 ms
Gaussian done
CPU times: total: 2.08 s
Wall time: 1.06 s
GMM done
CPU times: total: 0 ns
Wall time: 0 ns
KDE done
```

We see that GMM training is slowest, taking 100x as long as the other methods.

Density estimation quality

```
In [8]: grid_range = (-2.5, 2.5)

In [9]: x = np.linspace(grid_range, 100)
y = np.linspace(grid_range, 100)
xx, yy = np.meshgrid(X, y)
X_grid = np.vstack([xx.ravel(), yy.ravel()]).T

In [10]: time_hist_list = [hist.predict(X_grid).reshape(xx.shape) for hist in hist_list]
time_gaussian_list = [gaussian.predict(X_grid).reshape(xx.shape) for gaussian in gaussian_list]
time_gmm_list = [gmm.predict(X_grid).reshape(xx.shape) for gmm in gmm_list]
time_kde_list = [kde.predict(X_grid).reshape(xx.shape) for kde in kde_list]

CPU times: total: 15.6 ms
Wall time: 28 ms
Histogram done
CPU times: total: 375 ms
Wall time: 645 ms
Gaussian done
CPU times: total: 31.2 ms
Wall time: 28.6 ms
GMM done
CPU times: total: 53.9 s
Wall time: 54 s
KDE done
```

Calculating probabilities with a KDE is very slow, because each evaluation requires a loop over the entire training set.



Observations:
Histogram: Fits well to the data but tends to overfit (extreme case: small dataset, where most bins contain either none of just one data point)
Gaussian: Does not fit the multimodal distribution of the data well, but approximates a continuous blob.
GMM: Each Gaussian component captures a portion of the data. With $n_{\text{samples}} \geq 100$, the model begins to fit to the two moons fairly well. It is not constrained to a rectangular grid like the histogram and thus seems more natural.
KDE: Same problem as the histogram for small sample sizes. However, it tries its best (depending on the bandwidth, of course) to interpolate the distribution between individual samples. For large sample sizes, the learned distribution becomes very close to the ground truth. As a rule of thumb, the bandwidth should be in the order of the average distance to the nearest neighbor. Thus, it will automatically decrease as the training set size increases.

Sample

```
In [12]: n_samples = 1_000

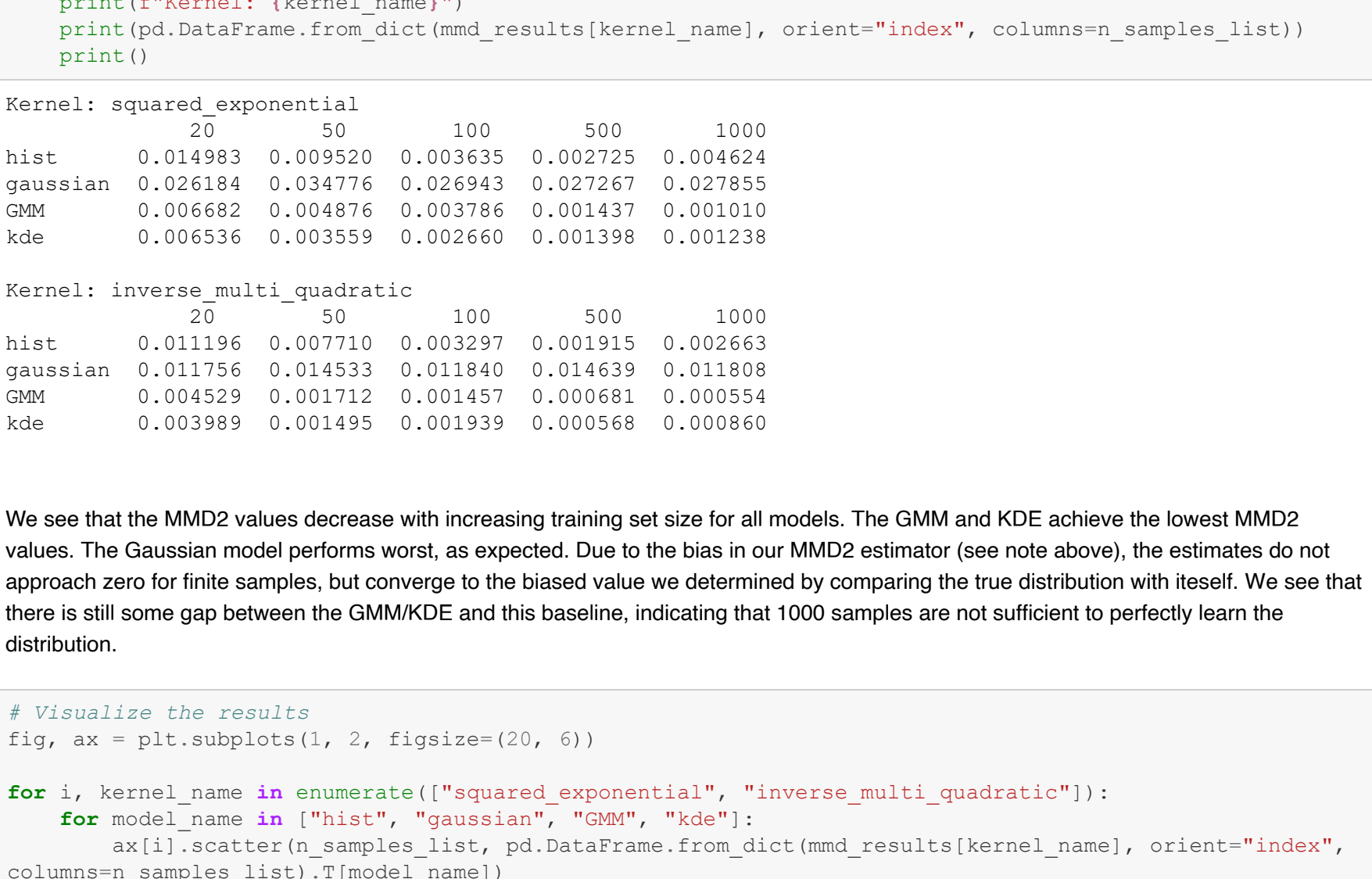
# Sample from the models
X_sample_hist_list = [hist.sample(n_samples, noise=True) for hist in hist_list]
X_sample_gaussian_list = [gaussian.sample(n_samples) for gaussian in gaussian_list]
X_sample_gmm_list = [gmm.sample(n_samples) for gmm in gmm_list]
X_sample_kde_list = [kde.sample(n_samples) for kde in kde_list]

In [13]: fig, ax = plt.subplots(4, len(n_samples_list), figsize=(20, 16))

model_names = ["Histogram", "Gaussian", "GMM", "KDE"]
for i, X_sample_model_list in enumerate([X_sample_hist_list, X_sample_gaussian_list, X_sample_gmm_list, X_sample_kde_list]):
    for j, X_sample_model in enumerate(X_sample_model_list):
        ax[i, j].scatter(X_sample_model[:, 0], X_sample_model[:, 1], s=1, c="black")

        if i == 0:
            ax[i, j].set_title(f"training set size = {n_samples_list[j]}")
        elif j == 0:
            ax[i, j].set_ylabel(model_names[i])
        elif j == 1:
            ax[i, j].set_xlabel(model_names[i])

        ax[i, j].set_xlim(grid_range)
        ax[i, j].set_ylim(grid_range)
```



Observations:
Histogram: The sampled data looks very blocky for small training sets because the distribution is sampled from only a few bins. We added uniform noise on a scale of a bin to partially account for this.
Gaussian: The generated samples do not represent the ground truth distribution well, but rather a weakly correlated cloud of datapoints.
GMM: For small training sets, the generated distribution looks very sharp (in case a component fits to two points) and points (when a component fits to only one point). However, the inverse multiquadratic sometimes surprisingly increases for large training sets. This may be a consequence of a sub-optimal choice of bandwidth or a statistical variation due to finite test set size (note that the inverse multiquadratic has a heavier tail and is thus more sensitive to outliers in the test set).
KDE: For small training sets, the generated distributions look a bit clumped, which smooths out for larger training sets. The bandwidth should probably be a bit larger for 20 training samples, and a bit smaller for 1000 training samples.

```
In [14]: # Implement the maximum mean discrepancy (MMD) metric
# with squared exponential and inverse multi-quadratic kernels.

# Compute the full kernel matrices between two datasets X1 and X2.
# It is crucial to vectorize this code, as it would otherwise be too slow in
# high dimensions, e.g. for images.
# There are many vectorization possibilities, and we show two for illustration:
# squared_exponential_kernel() uses 'cols' from 'scipy.spatial.distance'
# inverse_multi_quadratic_kernel() uses the expansion of the squared norm
def squared_exponential_kernel(x1, x2, h):
    squared_dist = cdist(x1, x2, 'sqeuclidean') / h ** 2
    return np.exp(-squared_dist)

def inverse_multi_quadratic_kernel(x1, x2, h):
    x1 = np.vstack(x1).T
    x2 = np.vstack(x2).T
    xx2 = np.sum(x2 ** 2, axis=1)
    squared_dist = (x1 - 2 * x1 @ x2.T + xx2) / (h ** 2)
    return 1 / np.sqrt(1 + squared_dist)

# Estimate the squared MMD between datasets X_true, X_pred.
def mmd2(X_true, X_pred, kernel, h):
    self_similarity_true = np.mean(kernel(X_true, X_true, h))
    self_similarity_pred = np.mean(kernel(X_pred, X_pred, h))
    cross_similarity = np.mean(kernel(X_true, X_pred, h))

    return self_similarity_true + self_similarity_pred - 2 * cross_similarity

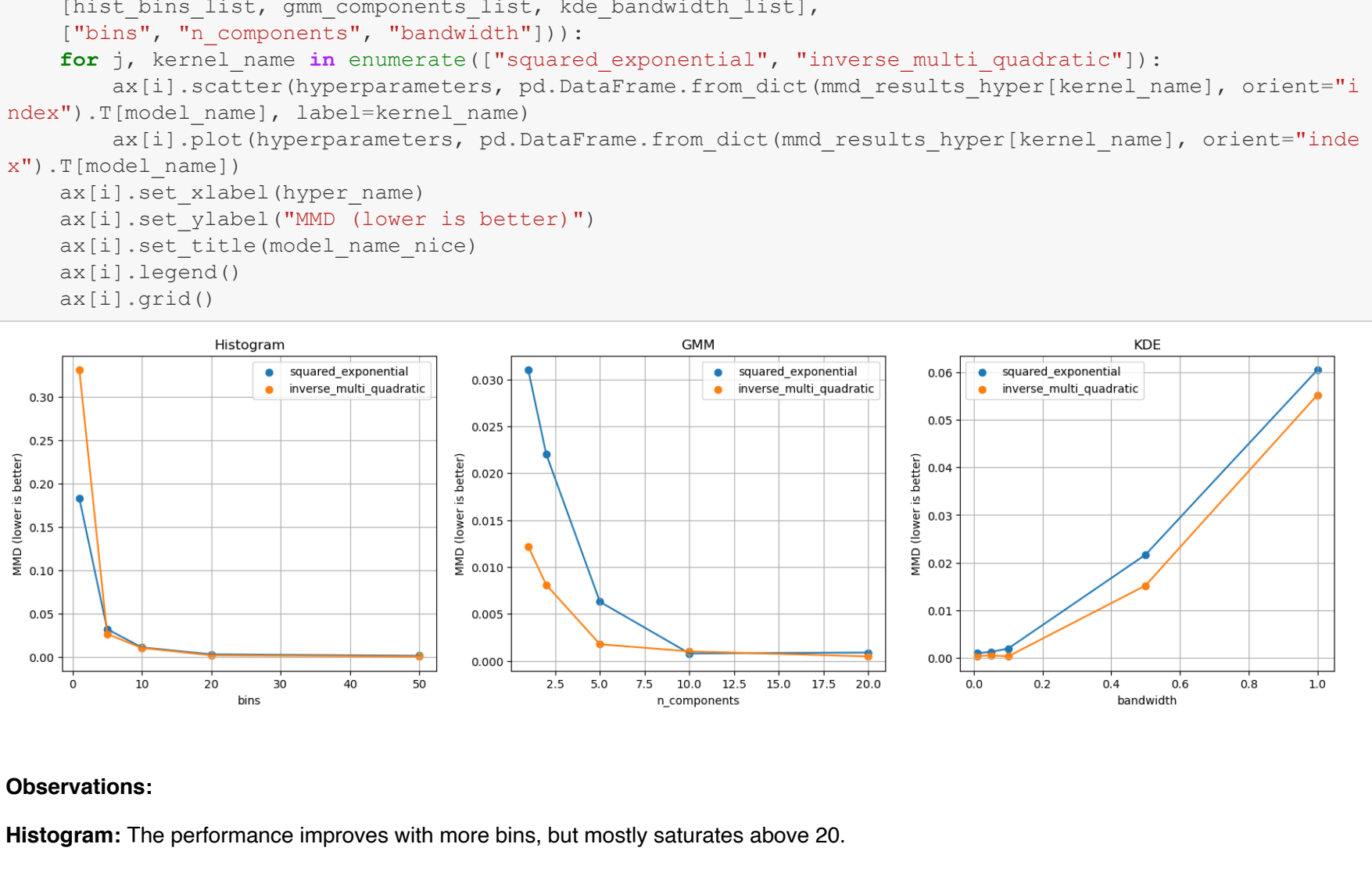
# To reduce the variance of the MMD2 estimator for finite samples,
# we may repeat it with different test sets and compute its mean and standard deviation.
# (default: only one repetition)
def mean_mmd2(model, n_samples, kernel, h, n_repeats=1, return_stddev=False):
    mmdres = 0
    mmdres2 = 0
    for k in range(n_repeats):
        true_samples = make_moons(n_samples, noise=0.1)[0]
        model_samples = model.sample(n_samples)
        x = mmd2(true_samples, model_samples, kernel, h)
        mmdres += x
        tmodel_name, label=kernel.name
    if return_stddev:
        return mmdres / n_repeats, np.sqrt(mmdres2 / n_repeats - (mmdres / n_repeats) ** 2)
    else:
        return mmdres / n_repeats
```

There are several possibilities to approximate the exact MMD2 from finite samples. Above, we use an estimate that includes the diagonal terms of the self-similarity matrices. Since $\text{kernel}(x, x) = 1$ holds for both kernels used here, these diagonal terms are all one. In contrast, the cross-similarity matrix does not usually have any entries $= 1$, because it is unlikely that a true and a generated point exactly coincide. So, we end up with an estimate that is biased upwards. In other words, our MMD2 value remains > 0 even when the learned model is perfect. The amount of bias depends on the dataset size, and it only disappears in the limit of infinite samples.

In the lecture, we learned about an estimate that excludes the diagonal self-similarity terms, the proposed method from the original paper. Since the cross-similarity matrix is not adjusted in the same way, e.g. by excluding the closest pairs, this leads to a downward bias for finite datasets. Consequently, however, the inverse multiquadratic sometimes surprisingly increases for large training sets. This may be a consequence of a sub-optimal choice of bandwidth or a statistical variation due to finite test set size (note that the inverse multiquadratic has a heavier tail and is thus more sensitive to outliers in the test set).

The third possibility is an unbiased estimate, where the self-similarities are calculated with two independent datasets from the same distribution. Then, the self-similarity matrices have only entries $= 1$, and the expected values of the MMD2 estimate is actually 0 if the two distributions are identical. However, this estimate still has non-zero variance for finite datasets, and we will get $\text{MMD2} < 0$ about half of the time. This is a general property of a random variable with zero mean and non-zero variance.

In practice, it is not so important which estimate one uses, because the ranking of competing models according to MMD2 remains unaffected by the bias. Similarly, when MMD2 is used as a loss for learning, its gradient is independent of the bias. So, we settle on the upwards biased estimator that includes the diagonal self-similarity terms for simplicity and speed.



Observations:
Histogram: The performance improves with more bins, but mostly saturates above 20.
GMM: The performance profits from more components, but the gains are smaller beyond 10.
KDE: For a training set of 1000 samples, KDE with a bandwidth between 0.05 and 0.1 works best. Larger kernels smooth out the details of the distribution. Note that a larger bandwidth would be needed for smaller training sets. The experiments in the previous section used bandwidth=0.1.

Task 2: Higher-dimensional data

```
In [26]: from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X = digits.data
Y = digits.target

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=42)
```

```
In [27]: from sklearn.neighbors import KernelDensity
from sklearn.mixture import GaussianMixture
from sklearn.ensemble import RandomForestClassifier
from density_forest import DensityForest
```

```
In [28]: # Fit the models with good hyperparameters
kde = KernelDensity(bandwidth=0.1).fit(X_train)
gmm = GaussianMixture(n_components=64).fit(X_train)
dforest = DensityForest(n_min=5)
dforest.fit(X_train)

c:\Users\ukoethe\conda\envs\pytorch\lib\site-packages\sklearn\cluster\_kmeans.py:1436: UserWarning: K
Means is known to have a memory leak on Windows with MKL, when there are less chunks than available t
hreadsa. You can avoid it by setting the environment variable OMP_NUM_THREADS=5.
  warnings.warn(
```

```
In [29]: n_samples = X_test.shape[0]
print(n_samples)

594
```

```
In [30]: kde_samples = kde.sample(n_samples)
gmm_samples = gmm.sample(n_samples)[0]
# Shuffle the gmm samples
np.random.shuffle(gmm_samples)
dforest_samples = dforest.sample(n_samples)
```

```
In [31]: # Compute the MMD

mmd_results_high_d = {
    kernel_name: {
        model_name: mmd2(X_test, model_samples, kernel, 1)
        for model_name, model_samples in zip(["kde", "gmm", "dforest"], [kde_samples, gmm_samples, dfor
est_samples])
    } for kernel_name, kernel in zip(
        ("squared_exponential", "inverse_multi_quadratic"),
        [squared_exponential_kernel, inverse_multi_quadratic_kernel])
}
```

```
In [32]: pd.DataFrame.from_dict(mmd_results_high_d, orient="index")
```

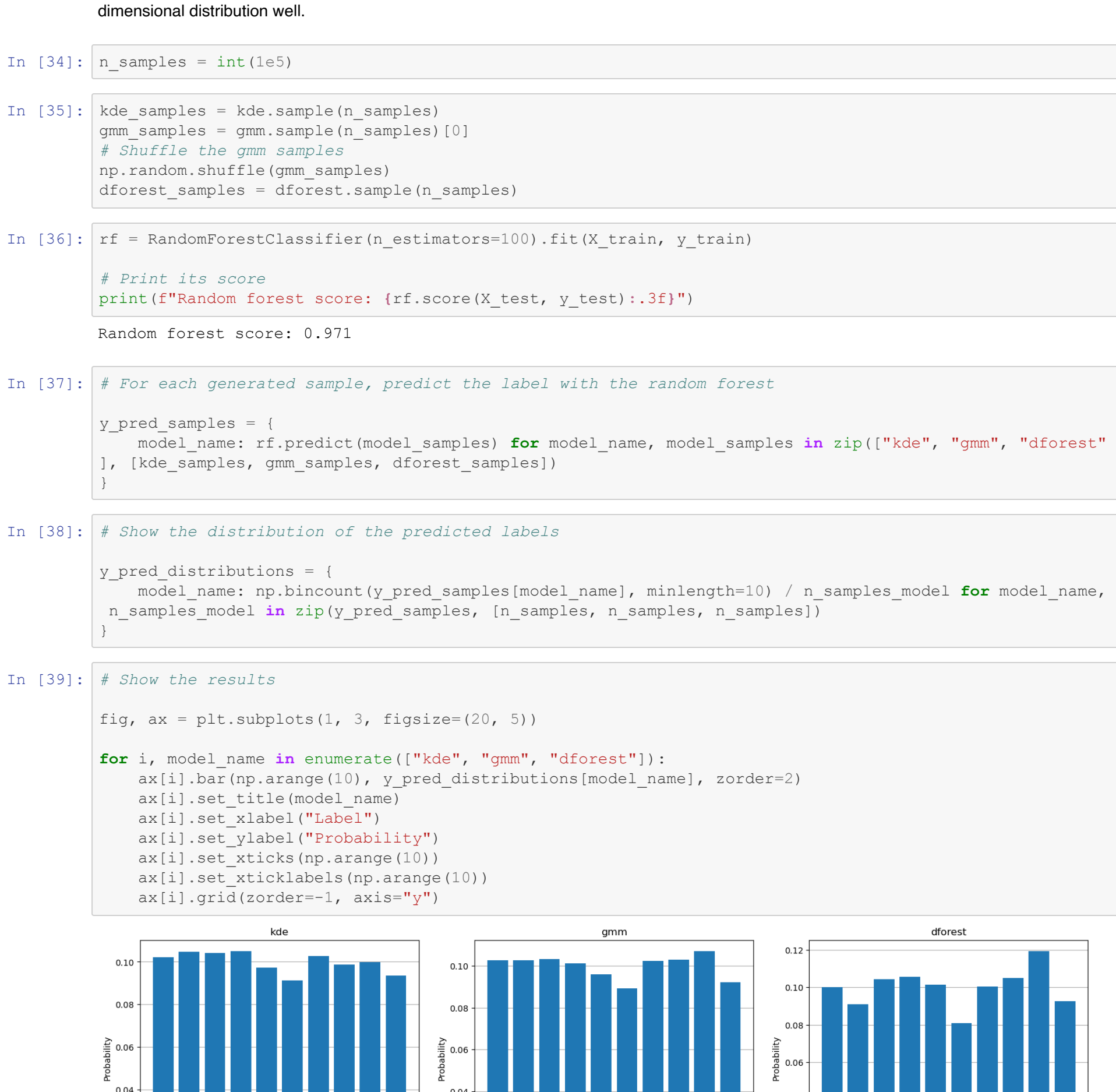
Out [32]:

| | kde | gmm | dforest |
|-------------------------|----------|----------|----------|
| squared_exponential | 0.003590 | 0.003367 | 0.003378 |
| inverse_multi_quadratic | 0.003797 | 0.003331 | 0.003417 |

```
In [33]: # Show some samples

fig, ax = plt.subplots(10, 3, figsize=(3, 10))

for j, (model_name, model_samples) in enumerate(zip(["kde", "gmm", "dforest"], [kde_samples, gmm_sample
s, dforest_samples])):
    for i in range(10):
        ax[i, j].imshow(model_samples[i].reshape(8, 8), cmap="gray")
        if i == 0:
            ax[i, j].set_title(model_name)
        ax[i, j].axis("off")
```



All methods generate recognisable digits. However, the quality is not great, since 1203 training instances are not sufficient to learn a 64-dimensional distribution well.

```
In [34]: n_samples = int(1e5)
```

```
In [35]: kde_samples = kde.sample(n_samples)
gmm_samples = gmm.sample(n_samples)[0]
# Shuffle the gmm samples
np.random.shuffle(gmm_samples)
dforest_samples = dforest.sample(n_samples)
```

```
In [36]: rf = RandomForestClassifier(n_estimators=100).fit(X_train, y_train)

# Print its score
print(f"Random forest score: {rf.score(X_test, y_test):.3f}")

Random forest score: 0.971
```

```
In [37]: # For each generated sample, predict the label with the random forest

y_pred_samples = {
    model_name: rf.predict(model_samples) for model_name, model_samples in zip(["kde", "gmm", "dforest"
], [kde_samples, gmm_samples, dforest_samples])
}
```

```
In [38]: # Show the distribution of the predicted labels

y_pred_distributions = {
    model_name: np.bincount(y_pred_samples[model_name], minlength=10) / n_samples_model for model_name,
n_samples_model in zip(y_pred_samples, [n_samples, n_samples, n_samples])
}
```

```
In [39]: # Show the results

fig, ax = plt.subplots(1, 3, figsize=(20, 5))

for i, model_name in enumerate(["kde", "gmm", "dforest"]):
    ax[i].bar(np.arange(10), y_pred_distributions[model_name], zorder=2)
    ax[i].set_title(model_name)
    ax[i].set_ylabel("label")
    ax[i].set_ylabel("probability")
    ax[i].set_xticks(np.arange(10))
    ax[i].set_yticks(np.arange(10))
    ax[i].set_yticklabels(np.arange(10))
    ax[i].grid(zorder=-1, axis="y")
```



Observations:

KDE results in a rather uniform distribution with digits 4, 5 and 9 slightly underrepresented. GMM looks similar, but 8's are too frequent. The density tree also tends to generate too many 8's and too few 1's, 5's, and 9's.

Illustration of bad hyperparameters

```
In [40]: # Fit the models with bad hyperparameters for comparison
kde_bad = KernelDensity(bandwidth=3).fit(X_train)
gmm_bad = GaussianMixture(n_components=1).fit(X_train)
dforest_bad = DensityForest(n_min=200)
dforest_bad.fit(X_train)

c:\Users\ukoethe\conda\envs\pytorch\lib\site-packages\sklearn\cluster\_kmeans.py:1436: UserWarning: K
Means is known to have a memory leak on Windows with MKL, when there are less chunks than available t
hreadsa. You can avoid it by setting the environment variable OMP_NUM_THREADS=5.
  warnings.warn(
```

```
In [41]: kde_bad_samples = kde_bad.sample(n_samples)
gmm_bad_samples = gmm_bad.sample(n_samples)[0]
# Shuffle the gmm samples
np.random.shuffle(gmm_bad_samples)
dforest_bad_samples = dforest_bad.sample(n_samples)
```

```
In [42]: # Show some samples

fig, ax = plt.subplots(10, 3, figsize=(3, 10))

for j, (model_name, model_samples) in enumerate(zip(["kde", "gmm", "dforest"], [kde_bad_samples, gmm_ba
d_samples, dforest_bad_samples])):
    for i in range(10):
        ax[i, j].imshow(model_samples[i].reshape(8, 8), cmap="gray")
        if i == 0:
            ax[i, j].set_title(model_name)
        ax[i, j].axis("off")
```



Observations:

The KDE bandwidth is too high, thus its samples become blurry. The GMM has only one component, so its reduces to a single Gaussian. In contrast to the two-moons dataset, the resulting samples still look somewhat reasonable, if blurry. The Density Forest now requires 200 samples per bin. At a training set size of 1203, it can only create about 5 bins, so it also becomes very blurry.

If we were to change the hyperparameters to the other extreme (e.g. KDE bandwidth too small, GMM with too many components, Density Forest with too few samples per bin), the models would overfit to the training data, i.e. the generated samples would essentially become replicates of the training instances.