Generative Neural Networks for the Sciences - Sample Solution Exercise 1

For this sample solution, we chose the submission of Paul Saegert, Nikita Tatsch and Christian Kleiber.

1 Two-dimensional data

In [29]:
```python
import numpy as np
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.cluster import kmeans_plusplus  # For GMM initialization
```

In [30]:
```python
# Implement a custom 2D histogram compatible with the sklearn API
class Histogram(BaseEstimator, TransformerMixin):
    def __init__(self, bins: tuple[int] = None, range_: tuple[int] = 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_ = None

    def fit(self, X, y = None):
        # 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_)

        return self

    def predict(self, X):
        # 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
        # Where the sample is inside the range, return the bin's value
        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 [31]:
```python
class Gaussian(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.mean_ = None
        self.covariance_ = None
        self.determinant_ = None
        self.inverse_ = None

    def fit(self, X, y = None):
        # Calculate the mean
        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):
        # 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.s
qrt(self.determinant_ * (2 * np.pi) ** 2)

        return values

    def sample(self, n=1):
        # First, sample from a standard normal distribution
        x = np.random.normal(size=(n, self.mean_.shape[0]))

        # Then, transform the samples to match the covariance matrix
        # Compute the eigendeomposition 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 [32]:
```python
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_ = np.empty((L, 2, 2))  # Assuming 2 dimensions for simplicity
        self.inverse_ = np.empty((L, 2, 2))
        self.determinant_ = np.empty(L)

    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, X, y=None, max_iter: int = 100):
        # Regularization term to prevent singular covariance matrix
        epsilon = 1e-6

        # Initialize mu_l using kmeans++
        mu, _ = kmeans_plusplus(X, n_clusters=self.L, random_state=0)
        self.means_ = mu  # 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.array([np.diag(np.var(X, axis=0))] * self.L)
        self.update_determinant_inverse()

        # Begin expectation maximization
        for _ in range(max_iter):
            # E-step: Calculate the responsibilities gamma_il
            responsibilities = np.zeros((X.shape[0], self.L))
            for l in range(self.L):
                diff = X - mu[l]
                exponent = np.einsum('ij,ij->i', np.dot(diff, self.inverse_[l]), diff)
                responsibilities[:, l] = self.weights_[l] * np.exp(-0.5 * exponent) / np.sqrt(self.dete
rminant_[l] * (2 * np.pi) ** X.shape[1])

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

            # M-step: Update the means and variances
            for l in range(self.L):
                self.weights_[l] = np.sum(responsibilities[:, l]) / X.shape[0]
                # Update the means
                mu[l] = np.sum(responsibilities[:, l].reshape(-1, 1) * X, axis=0) / np.sum(responsibili
ties[:, l])

                # Update the covariances
                diff = X - mu[l]
                self.covariances_[l] = np.dot(responsibilities[:, l] * diff.T, diff) / np.sum(responsib
ilities[:, l])
                self.covariances_[l] += np.eye(X.shape[1]) * epsilon  # Regularization term


            self.weights_ /= np.sum(self.weights_)
            self.update_determinant_inverse()

        self.means_ = mu

        return self

    def predict(self, X):
        # Calculate the total probability density of each sample under the model
        probabilities = np.zeros(X.shape[0])
        for l in range(self.L):
            diff = X - self.means_[l]
            exponent = np.einsum('ij,ij->i', np.dot(diff, self.inverse_[l]), diff)
            probabilities += self.weights_[l] * np.exp(-0.5 * exponent) / np.sqrt(self.determinant_[l]
 * (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_[l], self.covariances_[l]) for l i
n component])
        return samples
```

In [33]:
```python
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, x, y, x0, y0):
        coeff = 1 / (2 * np.pi * self.bandwidth ** 2)
        exp_val = -((x - x0) ** 2 + (y - y0) ** 2) / (2 * self.bandwidth ** 2)
        return coeff * np.exp(exp_val)

    def predict(self, X):
        # Initialize the predicte value to 0
        value = np.zeros(X.shape[0])

        # For every point in X, calculate the value of the KDE
        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
            value[i] /= len(self.points)

        return value

    def sample(self, n=1):
        # Sample from the KDE
        # Sample a point 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 [34]:
```python
from sklearn.datasets import make_moons, make_blobs
import matplotlib.pyplot as plt
```

In [35]:
```python
n_samples_list = [20, 50, 100, 500, 1000]
X_list = [make_moons(n_samples=n_samples_list, noise=0.1)[0] for n_samples_list in n_samples_list]
# X_list = [make_blobs(n_samples=n_samples_list)[0] for n_samples_list in n_samples_list]
```

In [36]:
```python
# Fit the models
hist_list = [Histogram(bins=(20, 20)).fit(X) for X in X_list]
print("Histogram done")
gaussian_list = [Gaussian().fit(X) for X in X_list]
print("Gaussian done")
gmm_list = [GMM(L=20).fit(X, max_iter = 100) for X in X_list]
print("GMM done")
kde_list = [KDE(bandwidth=0.1).fit(X) for X in X_list]
print("KDE done")
```

```
Histogram done
Gaussian done
GMM done
KDE done
```

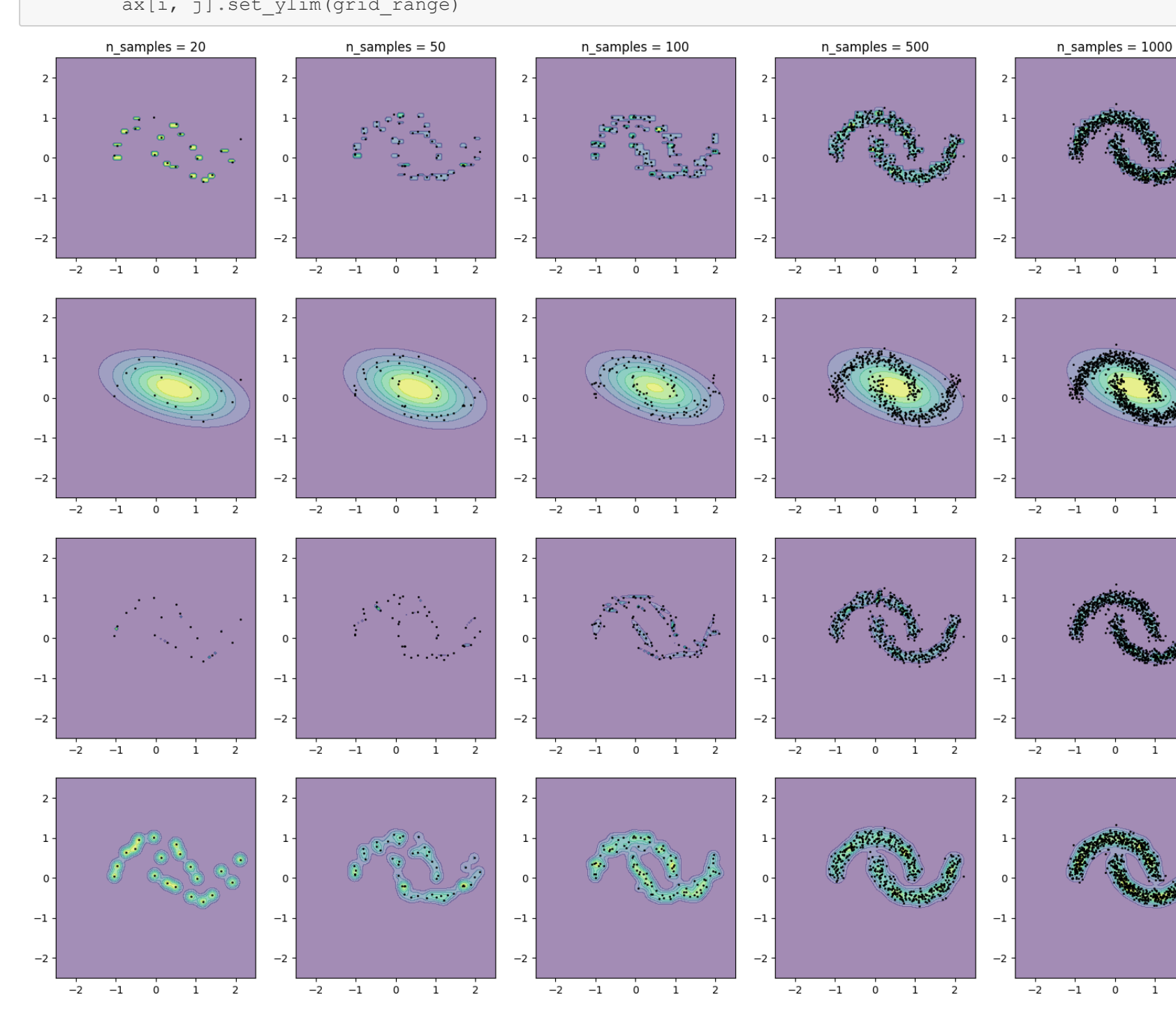
In [37]:
```python
grid_range = (-2.5, 2.5)
```

In [38]:
```python
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 [39]:
```python
y_pred_hist_list = [hist.predict(X_grid).reshape(xx.shape) for hist in hist_list]
print("Histogram done")
y_pred_gaussian_list = [gaussian.predict(X_grid).reshape(xx.shape) for gaussian in gaussian_list]
print("Gaussian done")
y_pred_gmm_list = [gmm.predict(X_grid).reshape(xx.shape) for gmm in gmm_list]
print("GMM done")
y_pred_kde_list = [kde.predict(X_grid).reshape(xx.shape) for kde in kde_list]
print("KDE done")
```

```
Histogram done
Gaussian done
GMM done
KDE done
```

In [40]:
```python
fig, ax = plt.subplots(4, len(n_samples_list), figsize=(20, 16))

for i, y_pred_model_list in enumerate([y_pred_hist_list, y_pred_gaussian_list, y_pred_gmm_list, y_pred_
kde_list]):
    for j in range(len(n_samples_list)):
        ax[i, j].contourf(xx, yy, y_pred_model_list[j], alpha=0.5)
        ax[i, j].scatter(X_list[j][:, 0], X_list[j][:, 1], s=1, c="black")

        if i == 0:
            ax[i, j].set_title(f"n_samples = {n_samples_list[j]}")

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



Observations:

Histogram: Fits well to the data but tends to overfit (extreme case: small dataset, where one bin size only captures one point)

Gaussian: Does not fit the multimodal distribution of the data well, but approximates a continuous blob.

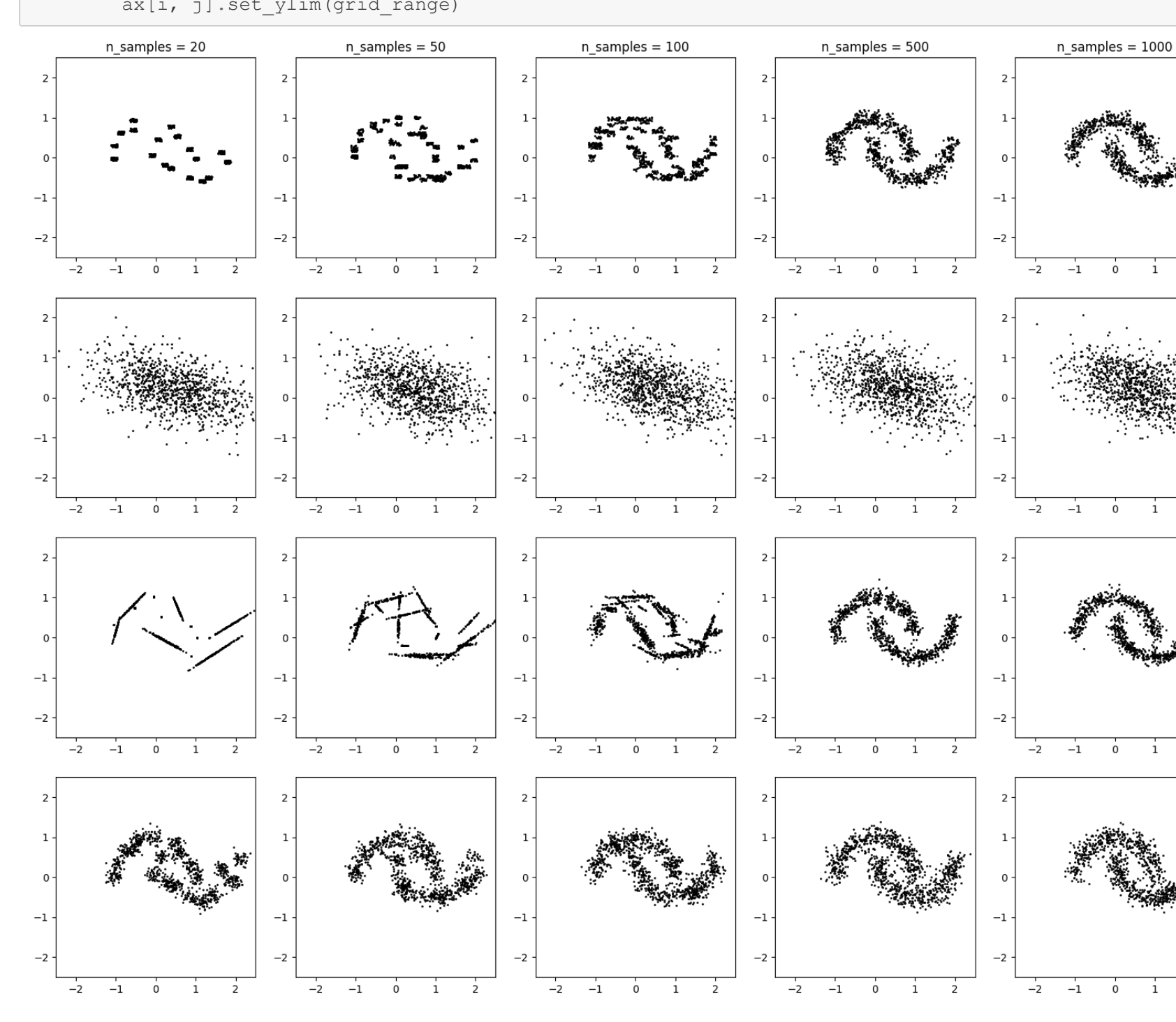
GMM: Each Gaussian component captures a portion of the data. With n_samples $\geq$ 100, the model begins to fit to the two moons fairly well. It is not constrained to a rectangular grid like the histogram and thus seems more natural.

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 comes very close to the ground truth.

Sample

In [41]:
```python
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 [42]:
```python
fig, ax = plt.subplots(4, len(n_samples_list), figsize=(20, 16))

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 in range(len(n_samples_list)):
        ax[i, j].scatter(X_sample_model_list[j][:, 0], X_sample_model_list[j][:, 1], s=1, c="black")

        if i == 0:
            ax[i, j].set_title(f"n_samples = {n_samples_list[j]}")

        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 the 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 fit to two points) and pointy (when a component fit to only one point). The more training samples there are, the better the generated samples align with the ground truth.

KDE: For small training sets, the generated distributions look a bit clumped, which smoothens out for larger training sets.

MMD

In [43]:
```python
# Implement the maximum mean discrepancy (MMD) metric with squared exponential and inverse multi-quadra
tic kernels for evaluation.

def squared_exponential_kernel(x1, x2, h=1):
    return np.exp(-np.sum((x1 - x2) ** 2) / (2 * h))

def inverse_multi_quadratic_kernel(x1, x2, h):
    return 1 / np.sqrt(np.sum((x1 - x2) ** 2) / h + 1)

def mmd2(X_true, X_pred, kernel):
    N = X_true.shape[0]
    M = X_pred.shape[0]
    repulsive = 1 / (M + (M-1)) * np.sum([kernel(X_pred[i], X_pred[j], 1) for i in range(M) for j in [k
 for k in range(M) if k != i]])
    attractive = 2 / (M * N) * np.sum([kernel(X_pred[i], X_true[j], 1) for i in range(M) for j in range
(N)])

    return repulsive - attractive
```

In [44]:
```python
from tqdm import tqdm
import pandas as pd
```

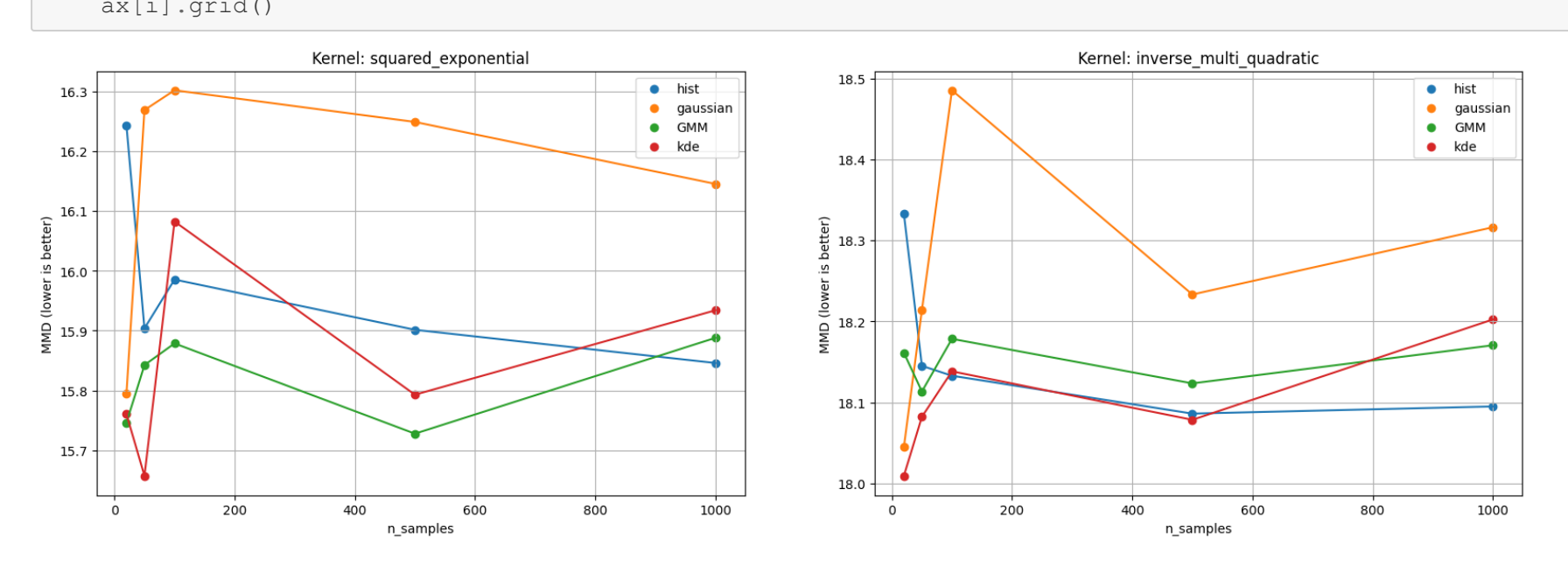
In [99]:
```python
mmd_results = {
    kernel_name: {
        model_name: [
            mmd2(X, model.sample(n_samples), kernel) for X, model in tqdm(zip(X_list, model_list))
        ] for model_name, model_list in zip(["hist", "gaussian", "GMM", "kde"], [hist_list, gaussian_li
st, gmm_list, kde_list])
    } for kernel_name, kernel in zip(
        ["squared_exponential", "inverse_multi_quadratic"],
        [squared_exponential_kernel, inverse_multi_quadratic_kernel])
}
```

```
5it [00:15,  3.15s/it]
5it [00:15,  3.14s/it]
5it [00:15,  3.15s/it]
5it [00:15,  3.13s/it]
5it [00:15,  3.17s/it]
5it [00:15,  3.15s/it]
5it [00:15,  3.17s/it]
5it [00:15,  3.17s/it]
```

In [100]:
```python
# Print the results in a pandas dataframe for each kernel
for kernel_name in mmd_results:
    print(f"Kernel: {kernel_name}")
    print(np.sqrt(pd.DataFrame.from_dict(mmd_results[kernel_name], orient="index", columns=n_samples_li
st)))
    print()
```

```
Kernel: squared_exponential
                20         50         100        500        1000
hist      16.243284  15.903887  15.985504  15.901490  15.845968
gaussian  15.794292  16.268804  16.301880  16.248907  16.145385
GMM       15.746146  15.842532  15.878706  15.727855  15.888045
kde       15.760532  15.657398  16.082515  15.792988  15.934068

Kernel: inverse_multi_quadratic
                20         50         100        500        1000
hist      18.333044  18.145060  18.132870  18.086046  18.094930
gaussian  18.044688  18.213901  18.485684  18.233271  18.316349
GMM       18.161287  18.113361  18.178472  18.123408  18.170503
kde       18.009179  18.082094  18.138319  18.078386  18.202399
```

In [101]:
```python
# Visualize the results
fig, ax = plt.subplots(1, 2, figsize=(20, 6))

for i, kernel_name in enumerate(["squared_exponential", "inverse_multi_quadratic"]):
    for model_name in ["hist", "gaussian", "GMM", "kde"]:
        ax[i].scatter(n_samples_list, np.sqrt(pd.DataFrame.from_dict(mmd_results[kernel_name], orient=
"index", columns=n_samples_list)).T[model_name])
        ax[i].plot(n_samples_list, np.sqrt(pd.DataFrame.from_dict(mmd_results[kernel_name], orient="index",
 columns=n_samples_list)).T)
    ax[i].set_xlabel("n_samples")
    ax[i].set_ylabel("MMD (lower is better)")
    ax[i].set_title(f"Kernel: {kernel_name}")
    ax[i].legend(["hist", "gaussian", "GMM", "kde"])
    ax[i].grid()
```



Observations:

Histogram: MMD decreases with larger training set.

Gaussian: MMD increases with larger training set. This is expected, since the model does not profit from more data.

GMM: MMD increases and then plateaus for larger training sets.

KDE: MMD does not improve nor worsen reliably.

Most changes happen in the range of 20 to 50 training samples.

Hyperparameters

In [91]:
```python
n_samples = 1000
```

In [92]:
```python
X = make_moons(n_samples=n_samples, noise=0.1)[0]
```

In [93]:
```python
hist_bins_list = [1, 5, 10, 20, 50]
gmm_components_list = [1, 2, 5, 10, 20]
kde_bandwidth_list = [0.01, 0.05, 0.1, 0.5, 1]
```

In [94]:
```python
# Fit the models
hist = {bins: Histogram(bins=(bins, bins)).fit(X) for bins in tqdm(hist_bins_list)}
gmm = {components: GMM(L=components).fit(X, max_iter=100) for components in tqdm(gmm_components_list)}
kde = {bandwidth: KDE(bandwidth=bandwidth).fit(X) for bandwidth in tqdm(kde_bandwidth_list)}
```

```
100%|██████████| 5/5 [00:00<00:00, 450.94it/s]
100%|██████████| 5/5 [00:00<00:00, 22.97it/s]
100%|██████████| 5/5 [00:00<00:00, 149796.57it/s]
```

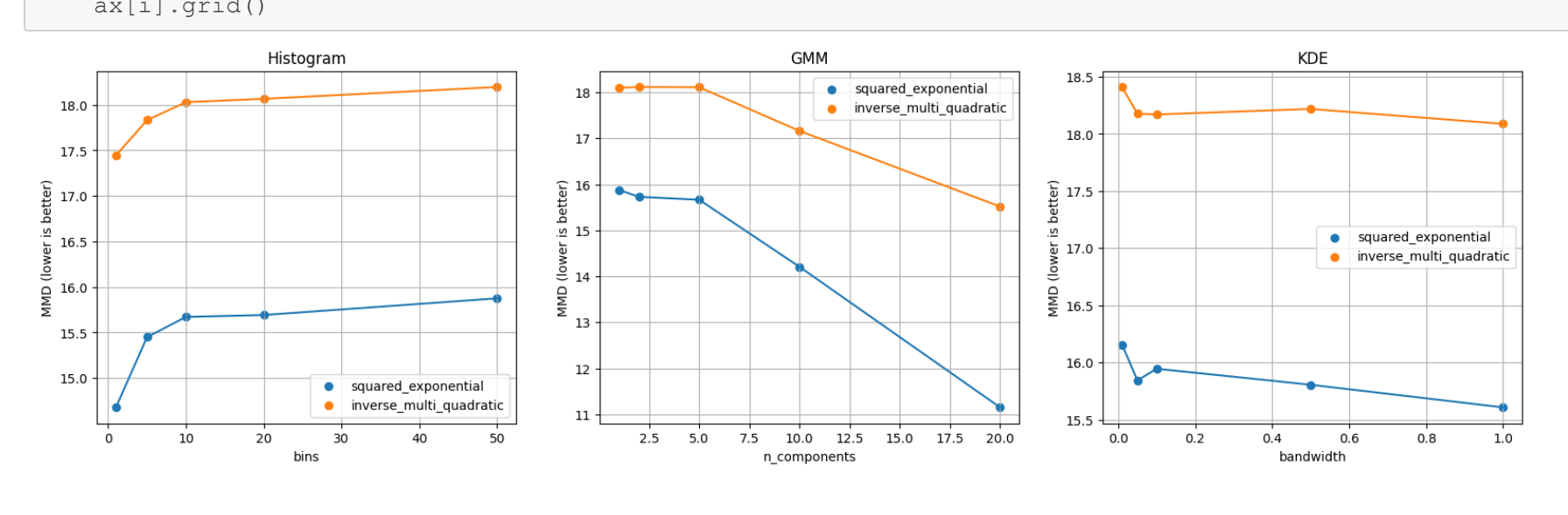
In [95]:
```python
# Compute the MMD
mmd_results_hyper = {
    kernel_name: {
        model_name: [
            mmd2(X, model.sample(n_samples), kernel) for model in tqdm(model_list)
        ] for model_name, model_list in zip(["hist", "gmm", "kde"], [hist.values(), kde.values(), gmm.v
alues()])
    } for kernel_name, kernel in zip(
        ["squared_exponential", "inverse_multi_quadratic"],
        [squared_exponential_kernel, inverse_multi_quadratic_kernel])
}
```

```
100%|██████████| 5/5 [00:23<00:00,  4.73s/it]
100%|██████████| 5/5 [00:23<00:00,  4.73s/it]
100%|██████████| 5/5 [00:23<00:00,  4.73s/it]
100%|██████████| 5/5 [00:23<00:00,  4.75s/it]
100%|██████████| 5/5 [00:23<00:00,  4.74s/it]
100%|██████████| 5/5 [00:23<00:00,  4.78s/it]
```

In [96]:
```python
# Print the results in a pandas dataframe for each kernel
for kernel_name in mmd_results_hyper:
    print(f"Kernel: {kernel_name}")
    print(np.sqrt(pd.DataFrame.from_dict(mmd_results_hyper[kernel_name], orient="index")))
    print()
```

```
Kernel: squared_exponential
              0          1          2          3          4
hist  14.677911  15.449488  15.670949  15.692174  15.875053
gmm   15.877567  15.729625  15.664891  14.206359  11.149662
kde   16.157561  15.844109  15.946050  15.805962  15.608165

Kernel: inverse_multi_quadratic
              0          1          2          3          4
hist  17.444086  17.832204  18.029457  18.066019  18.195827
gmm   18.102013  18.117604  18.113399  17.163230  15.517855
kde   18.410015  18.177891  18.170217  18.218007  18.087590
```

In [97]:
```python
fig, ax = plt.subplots(1, 3, figsize=(20, 5))

for i, (model_name, model_name_nice, hyperparameters, hyper_name) in enumerate(zip(
    ["hist", "gmm", "kde"],
    ['Histogram', 'GMM', 'KDE'],
    [hist_bins_list, gmm_components_list, kde_bandwidth_list],
    ["bins", "n_components", "bandwidth"])):
    for j, kernel_name in enumerate(["squared_exponential", "inverse_multi_quadratic"]):
        ax[i].scatter(hyperparameters, np.sqrt(pd.DataFrame.from_dict(mmd_results_hyper[kernel_name], o
rient="index")).T[model_name], label=kernel_name)
        ax[i].plot(hyperparameters, np.sqrt(pd.DataFrame.from_dict(mmd_results_hyper[kernel_name], orie
nt="index")).T[model_name])
    ax[i].set_xlabel(hyper_name)
    ax[i].set_ylabel("MMD (lower is better)")
    ax[i].set_title(model_name_nice)
    ax[i].legend()
    ax[i].grid()
```



Observations:

Histogram: Performance drops with more bins, which is unexpected.

GMM: Significantly profits from more components.

KDE: Does not profit from more bins.

---

2 Higher-dimensional data

In [106]:
```python
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 [121]:
```python
from sklearn.neighbors import KernelDensity
from sklearn.mixture import GaussianMixture
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KDTree
```

In [235]:
```python
# Fit the models
kde = KernelDensity(bandwidth=0.1).fit(X_train)
gmm = GaussianMixture(n_components=64).fit(X_train)
kdtree = KDTree(X_train)
```

In [236]:
```python
n_samples = X_test.shape[0]
print(n_samples)
```

```
594
```

In [237]:
```python
kde_samples = kde.sample(n_samples)
gmm_samples = gmm.sample(n_samples)[0]
# Shuffle the gmm samples
np.random.shuffle(gmm_samples)
kdtree_samples = X_train[kdtree.query(X_test, k=1)[1].flatten()]
```

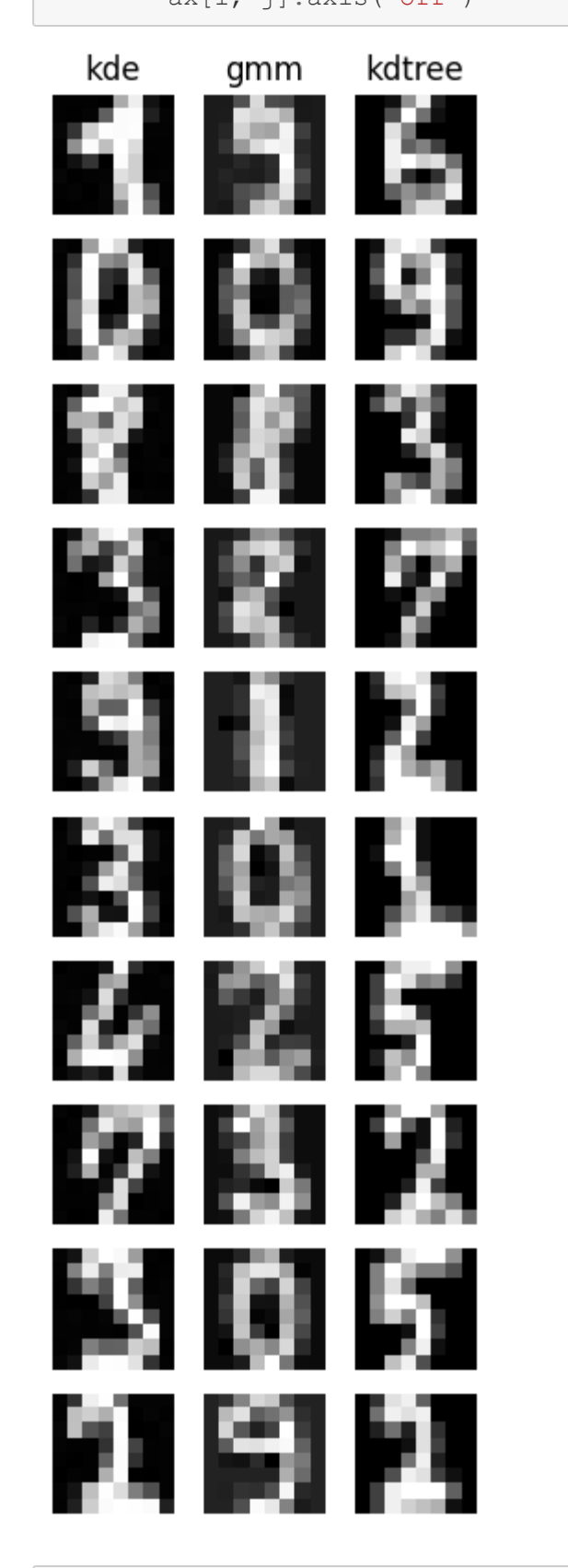
In [238]:
```python
# Compute the MMD

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

In [239]:
```python
pd.DataFrame.from_dict(mmd_results_high_d, orient="index")
```

Out[239]:

	kde	gmm	kdtree
squared_exponential	0.146513	7.602031e-13	0.363943
inverse_multi_quadratic	6.478762	6.303493e+00	6.804585

In [240]:
```python
# Show some samples

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

for j, (model_name, model_samples) in enumerate(zip(["kde", "gmm", "kdtree"], [kde_samples, gmm_samples
, kdtree_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")
```



In [266]:
```python
n_samples = int(1e5)
```

In [267]:
```python
kde_samples = kde.sample(n_samples)
gmm_samples = gmm.sample(n_samples)[0]
# Shuffle the gmm samples
np.random.shuffle(gmm_samples)
kdtree_samples = X_train[kdtree.query(X_test, k=1)[1].flatten()]
```

In [268]:
```python
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: 0.976
```

In [269]:

```
# 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", "kdtree"
], [kde_samples, gmm_samples, kdtree_samples])
}
```

In [270]:

```
# 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, X_test.shape[0]])
}
```

In [274]:

```
# Show the results

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

for i, model_name in enumerate(["kde", "gmm", "kdtree"]):
    ax[i].bar(np.arange(10), y_pred_distributions[model_name], zorder=2)
    ax[i].set_title(model_name)
    ax[i].set_xlabel("Label")
    ax[i].set_ylabel("Probability")
    ax[i].set_xticks(np.arange(10))
    ax[i].set_xticklabels(np.arange(10))
    ax[i].grid(zorder=-1, axis="y")
```


Observations:

KDE and GMM result in similar, rather uniform distributions with less generated samples for digits 4, 5 and 9. Meanwhile, the KDTree tends to generate more digits of classes 4, 5 and 9 and is less uniform and has more digit-to-digit variation.