Problem 1

Please load `data.mat` into your Python code, where you will find $x, y \in \mathbb{R}^{1001}$

Now, do the following procedures.

Solution

In [53]:
```
import torch
from scipy.io import loadmat

data = loadmat("data.mat")
x = torch.as_tensor(data["x"], dtype=torch.float32)
y = torch.as_tensor(data["y"], dtype=torch.float32)

display(dict(x_dim=tuple(x.shape), y_dim=tuple(y.shape)))
```

```
{'x_dim': (1001, 1), 'y_dim': (1001, 1)}
```

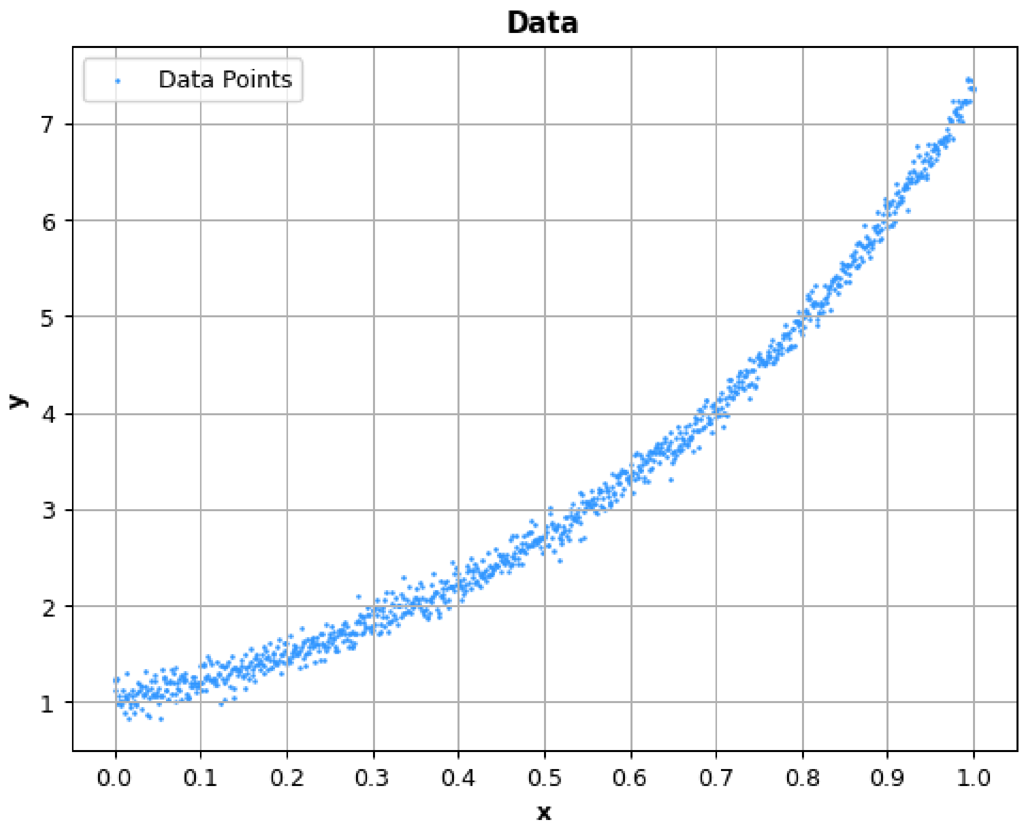
Problem 1.1

Use the plot function to visualize the data.

Solution

In [54]:

```
import matplotlib.pyplot as plt


def plot_data(
    x: torch.Tensor,
    y: torch.Tensor,
    *,
    scale: float | None = None,
    label: str | None = None,
) -> None:
    scale = float(scale or 1.0)
    label = str(label or "Data Points")

    # Create the canvas
    plt.figure(dpi=100 * scale, figsize=(8 * scale, 6 * scale))

    # Draw the grid
    x_min, x_max = x.min(), x.max()
    plt.xticks(torch.arange(x_min, x_max * 1.1, (x_max - x_min) / 10))
    plt.xlabel("x", fontweight="bold")
    plt.ylabel("y", fontweight="bold")
    plt.grid(True)

    # Plot the data points
    plt.scatter(x, y, c="#39F", s=1, label=label)
    plt.legend()


plot_data(x, y, scale=0.9)
plt.title("Data", fontweight="bold")
plt.show()
```


Data
Data Points
y
x
1
2
3
4
5
6
7
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Problem 1.2

Compute the least square line $y = \theta_0 + x\theta_1$ using the given data and overlay the line on the given data.

Solution

Let $\hat{\mathbf{y}} = \mathbf{x}^0\theta_0 + \mathbf{x}^1\theta_1 + \ldots + \mathbf{x}^{p-1}\theta_{p-1},\ \mathbf{x},\ \mathbf{y},\ \hat{\mathbf{y}} \in \mathbb{R}^n,\ \theta_i \in \mathbb{R},\ n,\ p \in \mathbb{N}$

The goal is to find the least $\mathcal{L} = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$

Let $\boldsymbol{\theta} = [\theta_0\ \theta_1 \ldots\ \theta_{p-1}]^T \in \mathbb{R}^{p\times 1},\ \mathbf{X} = [\mathbf{x}^0\ \mathbf{x}^1 \ldots\ \mathbf{x}^{p-1}] \in \mathbb{R}^{n\times p}$

We can rewrite $\mathcal{L}$ as $(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$ and $\hat{\mathbf{y}}$ as $\mathbf{X}\boldsymbol{\theta}$

Then, $\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}} = \frac{\partial\mathcal{L}}{\partial(\mathbf{y}-\mathbf{X}\boldsymbol{\theta})}\frac{\partial(\mathbf{y}-\mathbf{X}\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} = -2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})\mathbf{X}^T$

We know that $\mathcal{L}$ is minimal iff $\nabla\mathcal{L} = 0$. Then, $\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$.

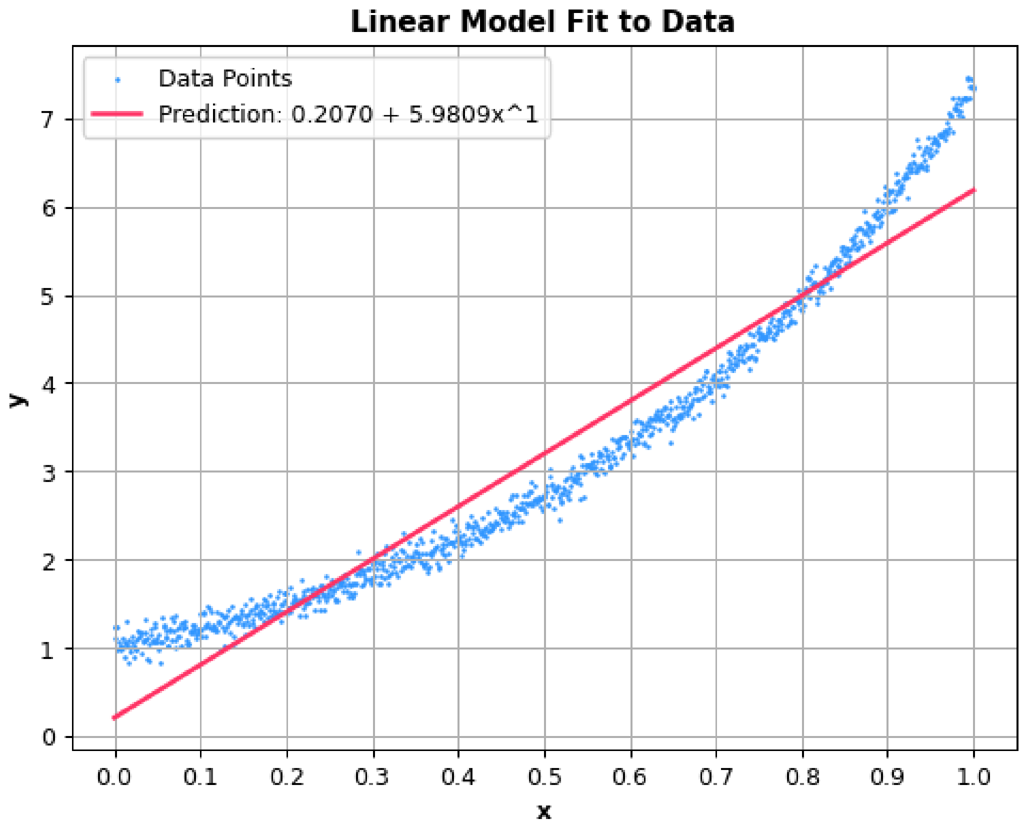
In [55]:

```
from dataclasses import dataclass
import torch
from typing import Callable


@dataclass
class LeastSquareOutput:
    coefficients: torch.Tensor
    predictor: Callable[[torch.Tensor], torch.Tensor]


def least_square(x: torch.Tensor, y: torch.Tensor, degree: int) ->
LeastSquareOutput:
    def get_X(x: torch.Tensor) -> torch.Tensor:
        x = torch.as_tensor(x).flatten()
        return torch.stack([x**p for p in range(degree + 1)], dim=1)

    X = get_X(x)
    y = torch.as_tensor(y).flatten()
    θ = (X.T @ X).inverse() @ X.T @ y
    y_hat_f = lambda x: get_X(x).matmul(θ).reshape_as(torch.as_tensor(x))

    return LeastSquareOutput(coefficients=θ, predictor=y_hat_f)
```

In [56]:

```
from matplotlib import pyplot as plt


def plot_least_square_model_and_data(
    x: torch.Tensor,
    y: torch.Tensor,
    degree: int,
    *,
    name: str,
) -> None:
    result = least_square(x, y, int(degree))
    c = result.coefficients
    f = f"{c[0]:.4f} + " + " + ".join([f"{c[i]:.4f}x^{i}" for i in range(1,
len(c))])
```

```
    plot_data(x, y, scale=0.9)
    plt.title(f"{name} Model Fit to Data", fontweight="bold")
    plt.plot(x, result.predictor(x), c="#F36", lw=2, label=f"Prediction: {f}")
    plt.legend()
    plt.show()


plot_least_square_model_and_data(x, y, 1, name="Linear")
```



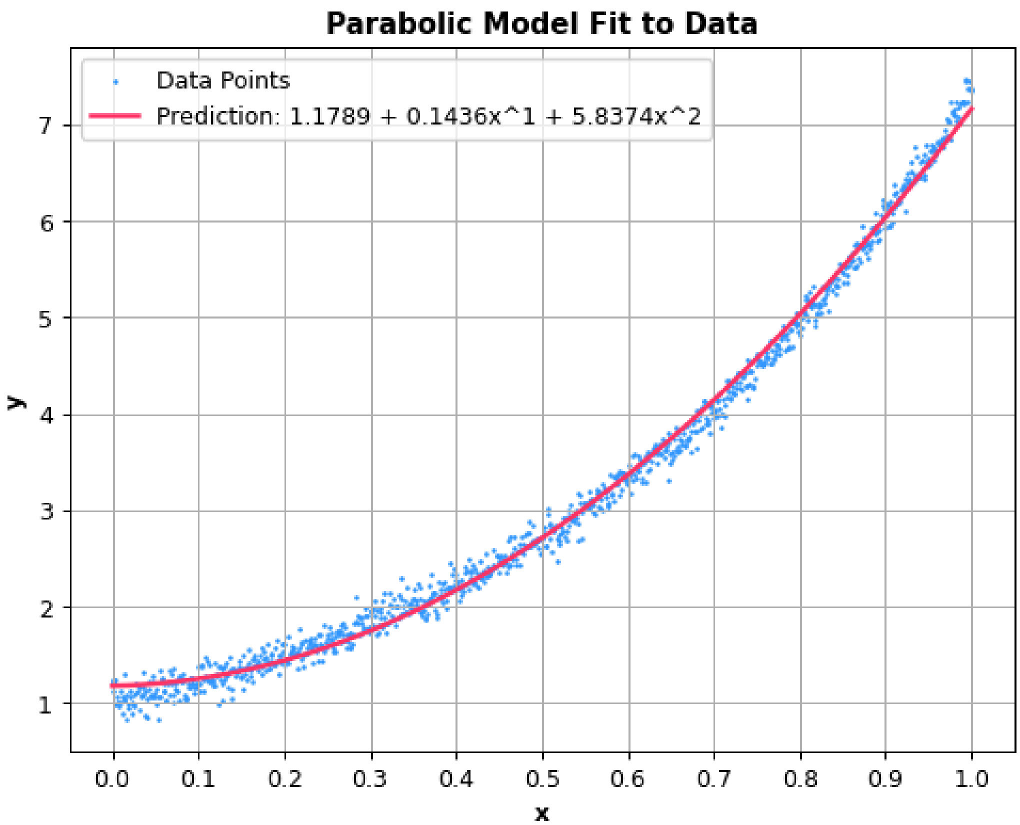
Problem 1.3

Fit a least squares second-order polynomial ($y = \theta_0 + x\theta_1 + x^2\theta_2$) to the data.

Solution

Use the same method as in Problem 1.2, $p = 3$.

In [57]:
```
plot_least_square_model_and_data(x, y, 2, name="Parabolic")
```



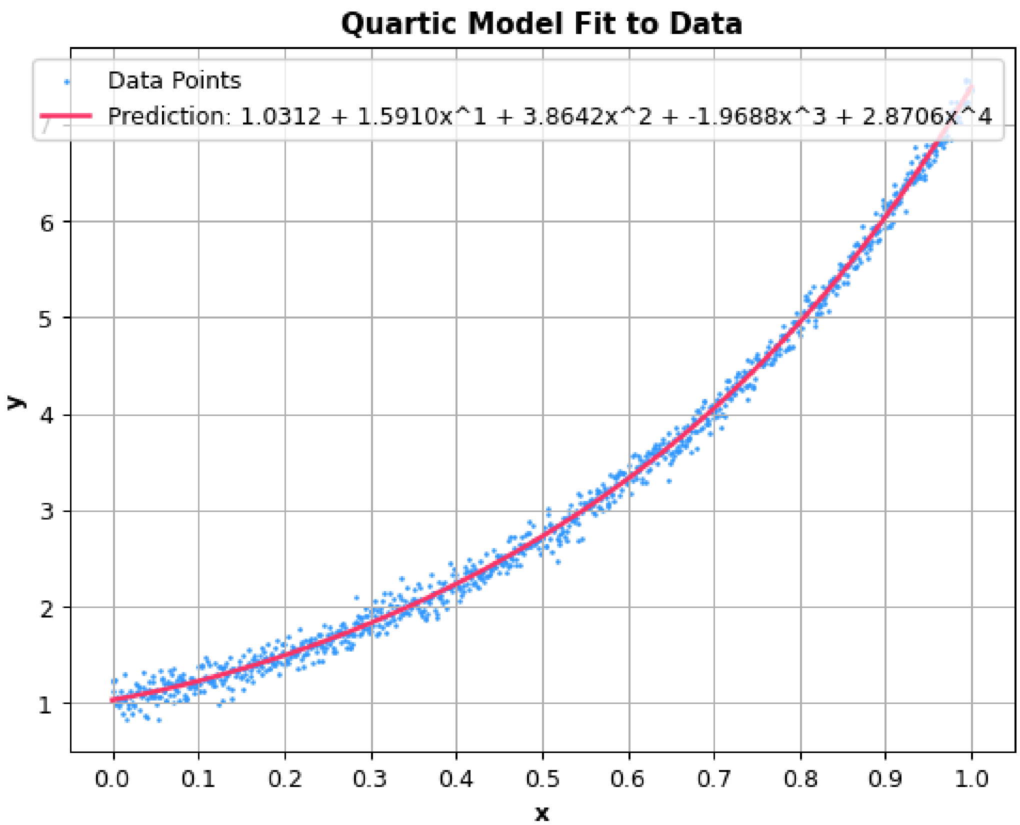
Problem 1.4

Fit a least squares quartic curve ($y = \theta_0 + x\theta_1 + x^2\theta_2 + x^3\theta_3 + x^4\theta_4$) to the data.

Solution

Use the same method as in Problem 1.2, $p = 5$.

In [58]:
```
plot_least_square_model_and_data(x, y, 4, name="Quartic")
```



Problem 1.5

Analyze which model (line, parabola, or quartic curve) is the most appropriate for this dataset.

Justify your answer by calculating and comparing the mean squared error (MSE) for each fitting model.

Solution

We can measure the MSEs of prediction and ground truth of all model and compare them.

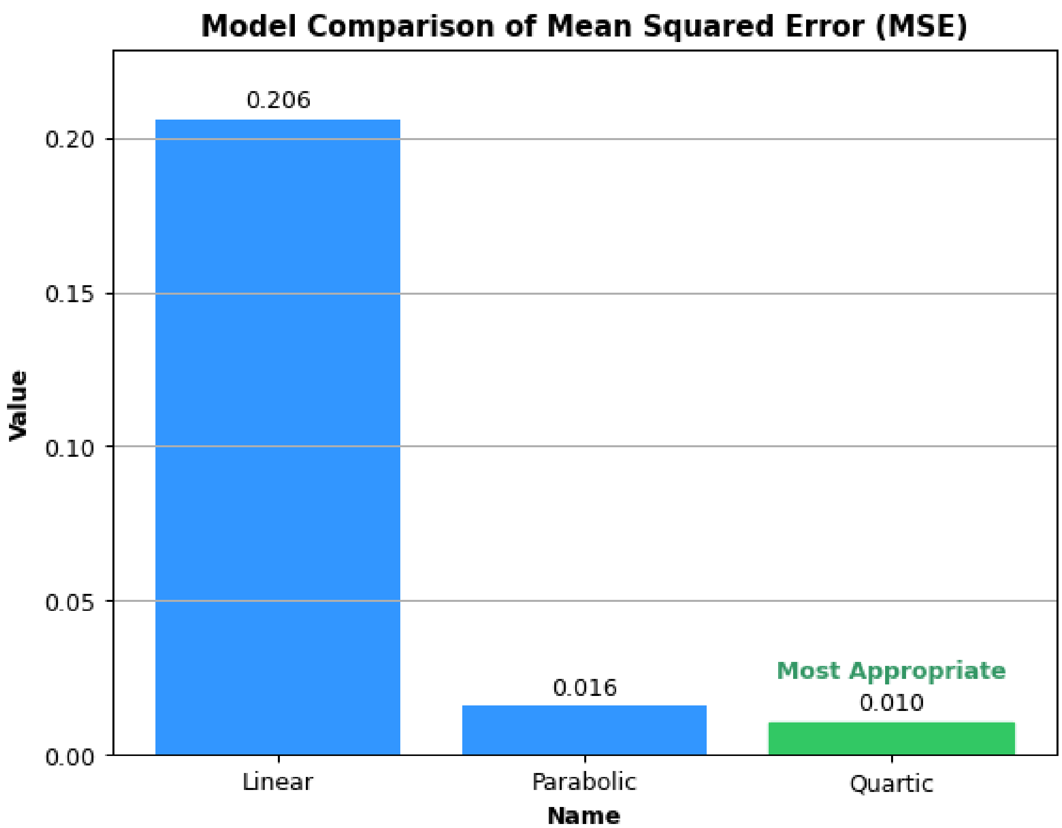
The most appropriate model for this dataset has the lowest MSE.

$MSE(\mathbf{y},\, \hat{\mathbf{y}}) = \frac{1}{n}\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2.$

In [59]:
```
from torch import Tensor


def get_mean_squared_error(output: Tensor, target: Tensor) -> Tensor:
    return ((output - target) ** 2).mean()
```

In [60]:
```
from matplotlib import pyplot as plt


def plot_bar(x: list[str], y: list[float], *, scale: float | None = None):
    scale = float(scale or 1.0)

    # Create the canvas
    plt.figure(dpi=100 * scale, figsize=(8 * scale, 6 * scale))

    # Draw the grid
    plt.xlabel("Name", fontweight="bold")
    plt.ylabel("Value", fontweight="bold")
    plt.ylim(0, max(y) * 1.11)
    plt.grid(axis="y")

    # Plot the bars
    bars = plt.bar(x, y, color="#39F")

    # Draw values on top of each bar
    for bar in bars:
        height = bar.get_height()
        plt.text(
            bar.get_x() + bar.get_width() / 2,
            height + max(y) * 0.01,
            f"{height:.3f}",
            ha="center",
            va="bottom",
        )

    return bars


benchmarks_model = ["Linear", "Parabolic", "Quartic"]
benchmarks_degree = [1, 2, 4]
benchmarks_mse = [0.0, 0.0, 0.0]

for index, degree in enumerate(benchmarks_degree):
```

```python
    y_hat = least_square(x, y, degree).predictor(x)
    benchmarks_mse[index] = get_mean_squared_error(y, y_hat).item()

bars = plot_bar(benchmarks_model, benchmarks_mse, scale=0.9)
min_bar = min(bars, key=lambda bar: bar.get_height())
min_bar.set_color("#3C6")
plt.text(
    min_bar.get_x() + min_bar.get_width() / 2,
    min_bar.get_height() + max(benchmarks_mse) * 0.06,
    "Most Appropriate",
    ha="center",
    va="bottom",
    fontweight="bold",
    color="#396",
)
plt.title("Model Comparison of Mean Squared Error (MSE)", fontweight="bold")
plt.show()
```



Problem 2

Following the previous questions, please randomly select 30 data samples and repeat this process 200 times.

Plot the resulting 200 linear regression lines ($y = \theta_0 + x\theta_1$) and 200 quartic curves ($y = \theta_0 + x\theta_1 + x^2\theta_2 + x^3\theta_3 + x^4\theta_4$) in two separate figures: one for lines and one for quartic curves.

In your report, explain the visualizations in the context of bias and variance, discussing how the spread and behavior of the curves or lines relate to the concepts of underfitting, overfitting, and model complexity.

Solution

In problem 1.5, we measured the mean square error (MSE) to determine the most accurate prediction model.

In this problem, we will decompose the MSE into bias and variance.

Let $D = \{(x_i,\ y_i)\}_{i=1}^n$ be the training dataset, $y(x)$ be the ground truth, $\hat{y}(x)$ be the prediction.

$$\begin{aligned} MSE &= \mathbb{E}_x[(y(x) - \hat{y}(x))^2] \\ &= \mathbb{E}_x[((y(x) - \mathbb{E}_D[\hat{y}(x;D)]) + (\mathbb{E}_D[\hat{y}(x;D)] - \hat{y}(x)))^2] \\ &= \mathbb{E}_x[(y(x) - \mathbb{E}_D[\hat{y}(x)])^2] + 0 + \mathbb{E}_x[(\mathbb{E}_D[\hat{y}(x)] - \hat{y}(x))^2] \\ &= \mathbb{E}_x[Bias^2 + Variance] \end{aligned}$$

$$Bias = \mathbb{E}_D[\hat{y}(x;D)] - y(x),\ Variance = (\mathbb{E}_D[\hat{y}(x;D)] - \hat{y}(x))^2.$$

In [51]:

```
from dataclasses import dataclass
import torch
from torch import Tensor
from typing import Callable


@dataclass
class BiasVarianceDecomposeOutput:
    bias: Tensor
    vars: Tensor
    samples: Tensor


def bias_vars_decompose(
    x: Tensor,
    y: Tensor,
    *,
    y_hat: Tensor,
    y_hat_d: Callable[[Tensor, Tensor], Tensor],
    sample_count: int,
    sample_size: int,
) -> BiasVarianceDecomposeOutput:
    x = torch.as_tensor(x)
    y = torch.as_tensor(y)
    y_hat = torch.as_tensor(y_hat)
    samples = torch.empty((x.shape[0], int(sample_count)))

    for d in range(int(sample_count)):
        sample_indices = torch.randperm(x.shape[0])[: int(sample_size)]
```

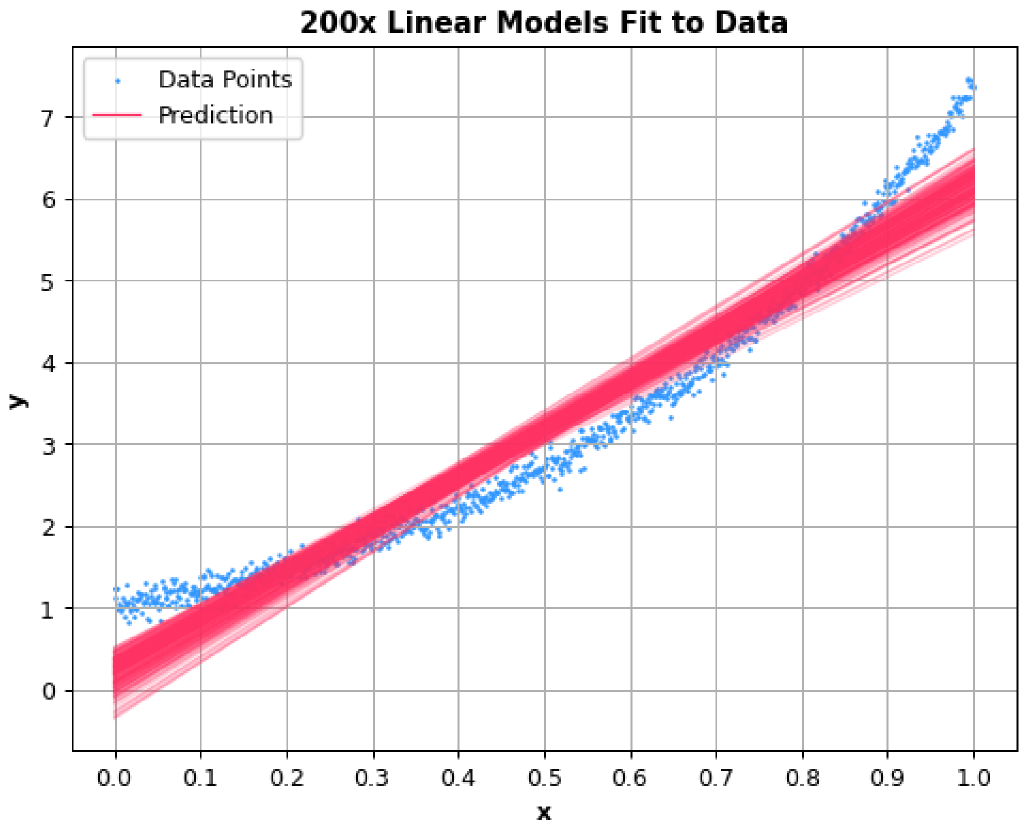
```
        samples[:, d : d + 1] = y_hat_d(x, sample_indices)

    y_hat_mean = samples.mean(1, keepdim=True)
    bias = y_hat_mean - y
    vars = (y_hat_mean - y_hat) ** 2

    return BiasVarianceDecomposeOutput(
        bias=bias,
        vars=vars,
        samples=samples,
    )
```

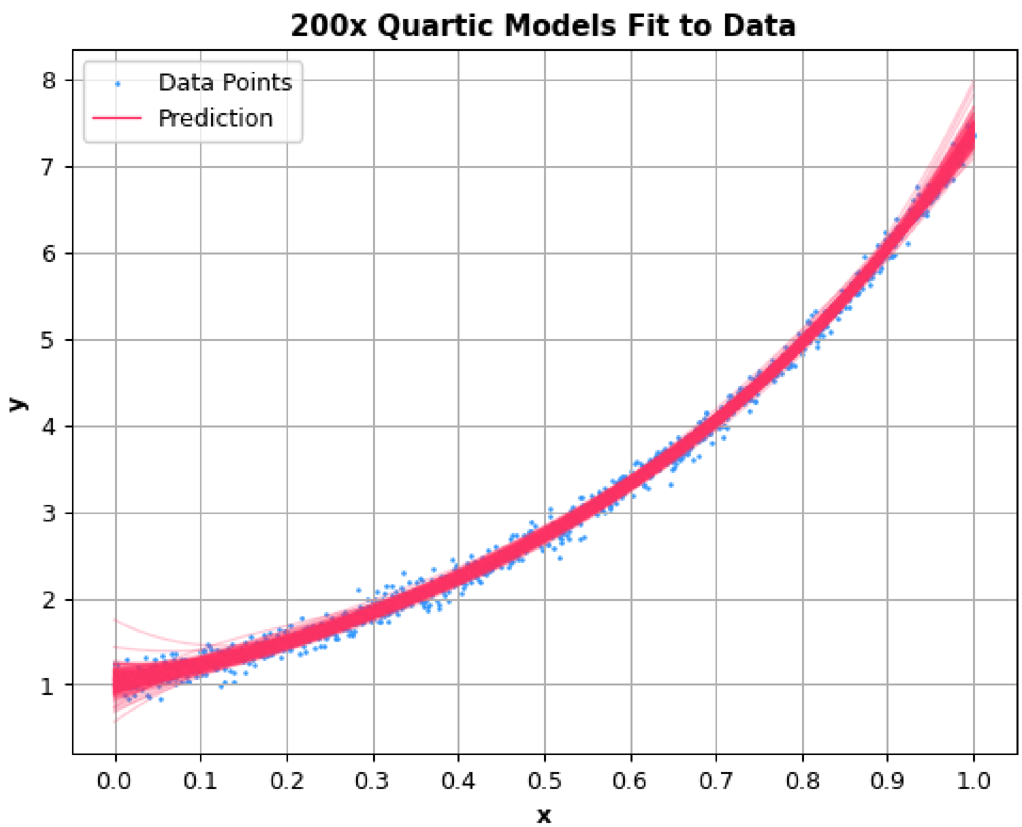
In [61]:
```
SAMPLE_COUNT = 200
SAMPLE_SEED = 2
SAMPLE_SIZE = 30

results_bvd: dict[str, BiasVarianceDecomposeOutput] = {
    "Linear": 1,
    "Quartic": 4,
}

for name, degree in results_bvd.items():
    torch.manual_seed(SAMPLE_SEED)
    results_bvd[name] = bias_vars_decompose(
        x,
        y,
        y_hat=least_square(x, y, degree).predictor(x),
        y_hat_d=lambda x, i: least_square(x[i], y[i], degree).predictor(x),
        sample_count=SAMPLE_COUNT,
        sample_size=SAMPLE_SIZE,
    )
```

```
torch.Size([1001, 1])
torch.Size([1001, 1])
```

Visualize the sampled models

In [38]:
```
from matplotlib import pyplot as plt

for name, result in results_bvd.items():
    plot_data(x, y, scale=0.9)
    plt.title(f"{SAMPLE_COUNT}x {name} Models Fit to Data", fontweight="bold")
    plt.plot(x, result.samples[:, 0], c="#F36", lw=1, label="Prediction")
    plt.plot(x, result.samples[:, 1:], c="#F36", lw=1, alpha=0.25)
    plt.legend()

plt.show()
```

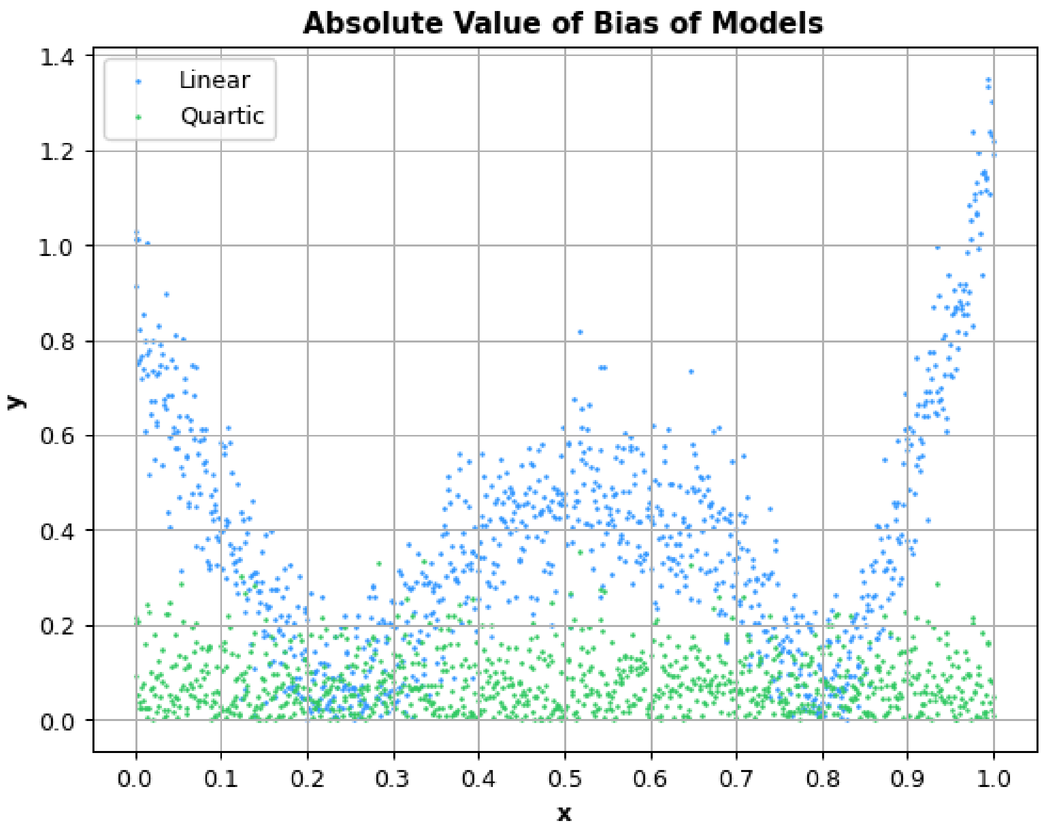


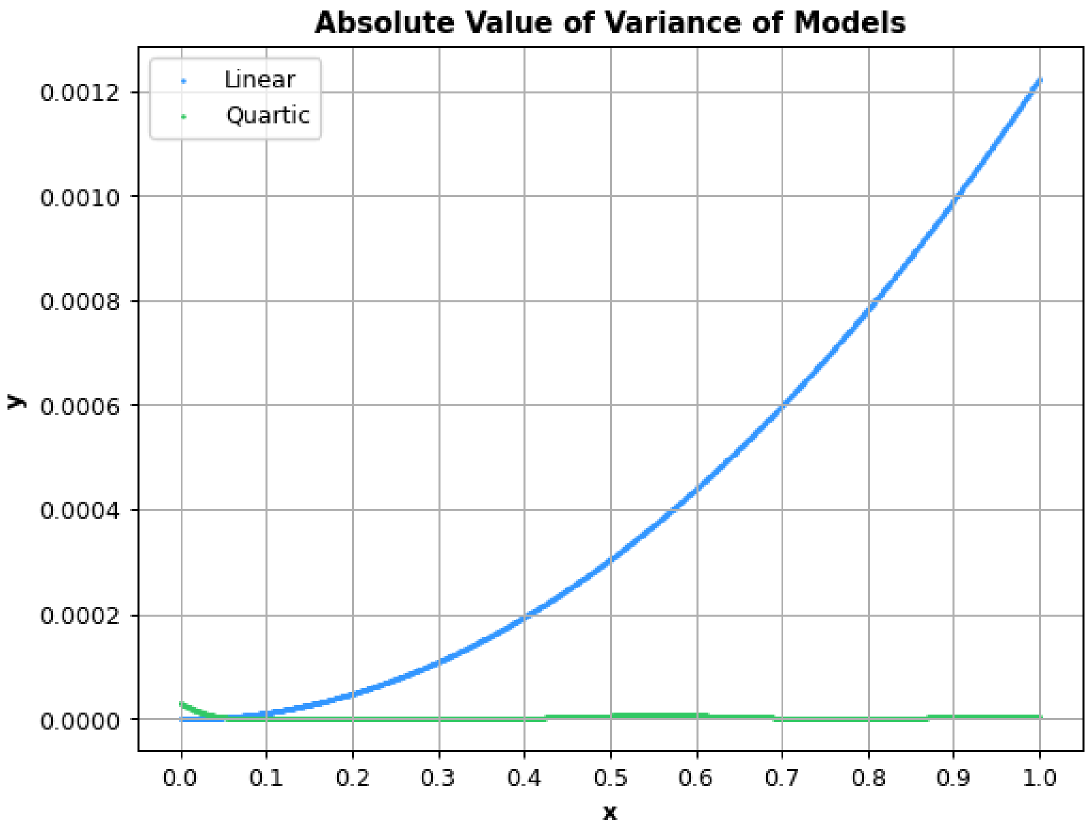


Visualize the bias and variance of all the sampled models

In [39]:
```
import matplotlib.pyplot as plt


PLOT_INFO = [
    ("bias", "Bias"),
```

```
    ("vars", "Variance"),
]

for key, name in PLOT_INFO:
    plot_data(x, results_bvd["Linear"].__dict__[key].abs(), scale=0.9,
label="Linear")
    plt.scatter(
        x, results_bvd["Quartic"].__dict__[key].abs(), c="#3C6", s=1,
label="Quartic"
    )
    plt.legend()
    plt.title(f"Absolute Value of {name} of Models", fontweight="bold")

plt.show()
```





Discussion

The bias and variance values of each x are shown in their absolute form for easier understanding.

In the bias plot, the linear model has **high bias** across most x values, especially at the edges, which shows that its **simplicity** makes it unable to handle the data's complexity. On the other hand, the quartic model, with its **greater complexity**, has **low bias**, meaning it fits the data **more accurately**.

The variance and prediction plots show that the linear model has a **wide spread** of predictions, meaning it struggles to adjust to more complex patterns, leading to **underfitting**. The quartic model, being more flexible due to its **higher complexity**, has **low variance** and offers a **good fit**.

Overall, the linear model's **simplicity** results in both **high bias** and **underfitting**. Meanwhile, the quartic model, with its **greater complexity**, finds a better balance, fitting the data well without **overfitting**.

	Low Bias	High Bias
Low Variance	Good fit (Quartic)	Underfitting
High Variance	Overfitting	Underfitting (Linear)

Problem 3

In `train.mat`, you will find 2-D points $\mathbf{X} = [\mathbf{x}_1\ \mathbf{x}_2]$ and their corresponding labels $\mathbf{Y} = \mathbf{y}$

Please use logistic regression $h(\boldsymbol{\theta};\ \mathbf{x}) = \frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}}}$ to find the decision boundary (optimal $\boldsymbol{\theta}^*$) based on `train.mat`.

Solution

Let $\mathbf{X} = [1\ \mathbf{x}^{(1)}\ \mathbf{x}^{(2)}\ \dots\ \mathbf{x}^{(p)}] \in \mathbb{R}^{n\times(p+1)}$, $\mathbf{Y} = \mathbf{y} \in \mathbb{R}^n$, $\boldsymbol{\theta} = [\theta_0\ \theta_1\ \theta_2\ \dots \theta_p]^T \in \mathbb{R}^{(p+1)\times 1}$, $n,\ p \in \mathbb{N}$

The logistic regression model is $h(\boldsymbol{\theta};\ \mathbf{X}) = \frac{1}{1+e^{-\mathbf{X}\boldsymbol{\theta}}}$

To find the decision boundary, we can maximize the likelihood function $L(\boldsymbol{\theta}) = \prod_{i=1}^{n}(h(\boldsymbol{\theta};\ \mathbf{x}_i))^{\mathbf{y}_i}(1-h(\boldsymbol{\theta};\ \mathbf{x}_i))^{1-\mathbf{y}_i}$

$h(x) = \frac{1}{1+e^{-x}}$ is strictly log-concave, since $\ln(h(x))'' = \frac{-e^{-x}}{(1+e^{-x})^2} < 0$. Therefore, $L(\boldsymbol{\theta})$ is also strictly log-concave.

Minimizing $-\ln L(\boldsymbol{\theta})$ is equivalent to maximizing $L(\boldsymbol{\theta})$

Since $-\ln L(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta})$ is a strictly convex function, we can approach its global minima using gradient descent.

$$\nabla\mathcal{L}(\boldsymbol{\theta}) = \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}$$

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\mathbf{y}_i\ln(h(\boldsymbol{\theta};\ \mathbf{x}_i)) + (1-\mathbf{y}_i)\ln(1-h(\boldsymbol{\theta};\ \mathbf{x}_i))$$

$$\begin{aligned}
\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} &= -\sum_{i=1}^{n}(\mathbf{y}_i h(\boldsymbol{\theta};\ \mathbf{x}_i)^{-1} - (1-\mathbf{y}_i)(1-h(\boldsymbol{\theta};\ \mathbf{x}_i))^{-1})\frac{\partial h(\boldsymbol{\theta};\ \mathbf{x}_i)}{\partial\boldsymbol{\theta}} \\
&= -\sum_{i=1}^{n}(\mathbf{y}_i h(\boldsymbol{\theta};\ \mathbf{x}_i)^{-1} - (1-\mathbf{y}_i)(1-h(\boldsymbol{\theta};\ \mathbf{x}_i))^{-1})h(\boldsymbol{\theta};\ \mathbf{x}_i)(1-h(\boldsymbol{\theta};\ \mathbf{x}_i))\mathbf{x}_i \\
&= -\sum_{i=1}^{n}(\mathbf{y}_i - h(\boldsymbol{\theta};\ \mathbf{x}_i))\mathbf{x}_i \\
&= (h(\boldsymbol{\theta};\ \mathbf{X}) - \mathbf{Y})^T\mathbf{X}
\end{aligned}$$

The optimization process can be written as $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha\nabla\mathcal{L}(\boldsymbol{\theta}_t)$

The optimal $\boldsymbol{\theta}^*$ can be found by iterating gradient descent until convergence.

In [40]:
```
import torch


def sigmoid(z):
    return 1 / (1 + torch.exp(-z))


class LogisticRegression(torch.nn.Module):
    def __init__(self, p: int):
        super(LogisticRegression, self).__init__()
        self.θ = torch.nn.Parameter(torch.full((p, 1), 1e-3),
requires_grad=True)
        assert self.θ.requires_grad

    def classify(self, X: torch.Tensor) -> torch.Tensor:
        return self.forward(X).round()

    def fit(self, X: torch.Tensor, Y: torch.Tensor) -> None:
```

```
        Y = torch.as_tensor(Y)
        H = self.forward(X)
        self.backward(H, X, Y)

    def forward(self, X: torch.Tensor) -> torch.Tensor:
        X = torch.as_tensor(X)
        H = sigmoid(X @ self.θ)
        return H

    def backward(self, H: torch.Tensor, X: torch.Tensor, Y: torch.Tensor) ->
None:
        H = torch.as_tensor(H)
        X = torch.as_tensor(X)
        Y = torch.as_tensor(Y)

        if H.grad_fn:
            H = H.detach()
            dL_dθ = (H - Y).T @ X
            self.θ.grad = dL_dθ.T
```

Problem 3.1

Plot the 2-D data points and the decision boundary on the same graph to visually illustrate the model's separation of classes.

Solution

In [41]:

```
import matplotlib.pyplot as plt


def plot_data_2d(
    x1: torch.Tensor,
    x2: torch.Tensor,
    y: torch.Tensor,
    *,
    scale: float | None = None,
    label1: str | None = None,
    label2: str | None = None,
) -> None:
    scale = float(scale or 1.0)
    label1 = str(label1 or "Data Points 1")
    label2 = str(label2 or "Data Points 2")
    labels = torch.as_tensor(y).unique()

    # Create the canvas
    plt.figure(dpi=100 * scale, figsize=(8 * scale, 6 * scale))

    # Draw the grid
    x1_max, x1_min, x2_max, x2_min = x1.max(), x1.min(), x2.max(), x2.min()
    plt.xticks(torch.arange(x1_min * 0.9, x1_max * 1.1, (x1_max - x1_min) / 10))
    plt.yticks(torch.arange(x2_min * 0.9, x2_max * 1.1, (x2_max - x2_min) / 10))
    plt.xlabel("x1", fontweight="bold")
    plt.ylabel("x2", fontweight="bold")

    # Plot the data points
    plt.scatter(x1[y == labels[0]], x2[y == labels[0]], c="#39F", s=4,
label=label1)
    plt.scatter(x1[y == labels[1]], x2[y == labels[1]], c="#F36", s=4,
label=label2)
    plt.legend()
```

In [42]:

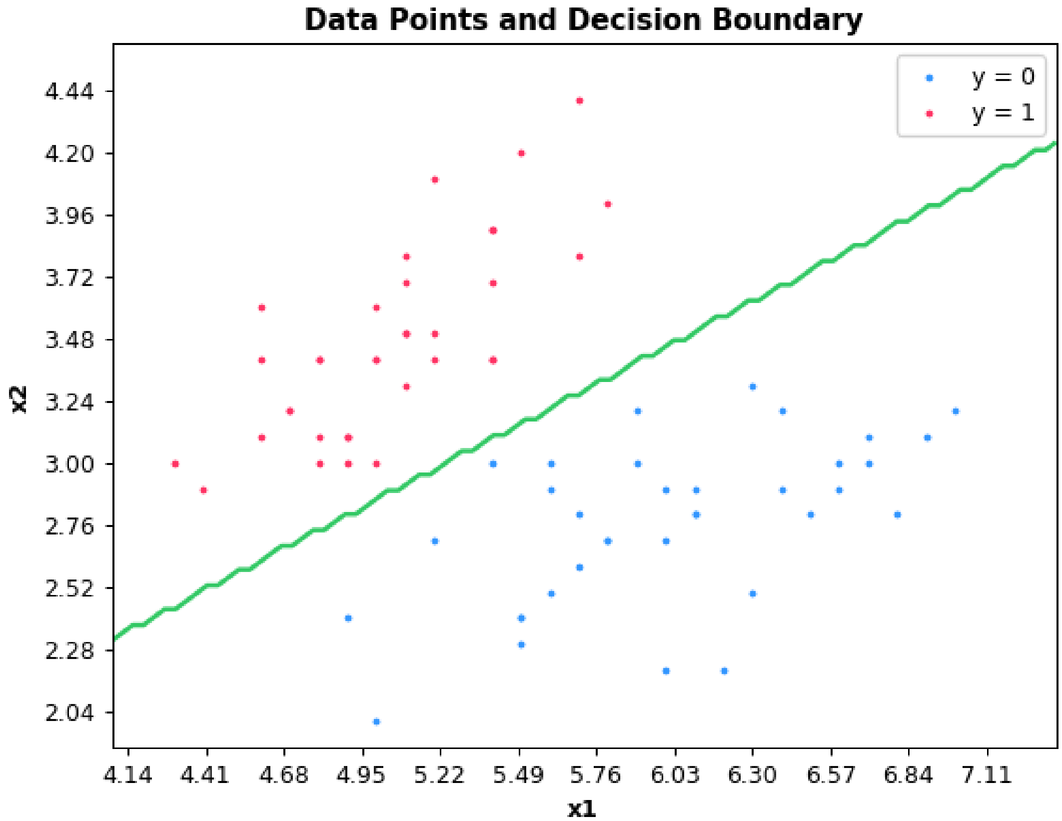
```
import matplotlib.pyplot as plt
import torch


def analyze_decision_boundary_2d(
    model: LogisticRegression,
    X: torch.Tensor,
    Y: torch.Tensor,
    *,
    scale: float | None = None,
) -> None:
    scale = float(scale or 1.0)
    X = torch.as_tensor(X)
    Y = torch.as_tensor(Y)
    x1 = X[..., 1:2]
    x2 = X[..., 2:3]
    y = Y
```

```python
    with torch.no_grad():
        model = model.eval()
        eval_error = model.classify(X).ne(Y).float().mean().item()

    display(
        dict(
            x1_dim=tuple(x1.shape),
            x2_dim=tuple(x2.shape),
            y_dim=tuple(y.shape),
            eval_error=f"{eval_error:.3%}",
        )
    )

    labels = y.int().unique()
    x1_max, x1_min, x2_max, x2_min = x1.max(), x1.min(), x2.max(), x2.min()
    x1_grid, x2_grid = torch.meshgrid(
        torch.linspace(x1_min * 0.95, x1_max * 1.05, int(100 * scale)),
        torch.linspace(x2_min * 0.95, x2_max * 1.05, int(100 * scale)),
        indexing="xy",
    )

    with torch.no_grad():
        X = torch.stack(
            [torch.ones_like(x1_grid.flatten()), x1_grid.flatten(),
x2_grid.flatten()],
            dim=1,
        )
        y_grid = model.classify(X).reshape_as(x1_grid)

    plot_data_2d(
        x1,
        x2,
        y,
        scale=scale,
        label1=f"y = {labels[0]}",
        label2=f"y = {labels[1]}",
    )
    plt.contour(x1_grid, x2_grid, y_grid, levels=[0.5], colors="#3C6",
linewidths=2)
    plt.title("Data Points and Decision Boundary", fontweight="bold")
```

In [43]:
```python
import torch
from scipy.io import loadmat

ITERATION_COUNT = 10000
LEARNING_RATE = 1e-5

data = loadmat("train.mat")
x1 = torch.as_tensor(data["x1"], dtype=torch.float32)
x2 = torch.as_tensor(data["x2"], dtype=torch.float32)
y = torch.as_tensor(data["y"], dtype=torch.float32)

X = torch.cat([torch.ones_like(x1), x1, x2], dim=1)
Y = y

model = LogisticRegression(2 + 1)
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
for i in range(ITERATION_COUNT):
    model.fit(X, Y)
    optimizer.step()
    optimizer.zero_grad()
```

```
analyze_decision_boundary_2d(model, X, Y, scale=0.9)
```

```
{'x1_dim': (70, 1),
 'x2_dim': (70, 1),
 'y_dim': (70, 1),
 'eval_error': '0.000%'}
```



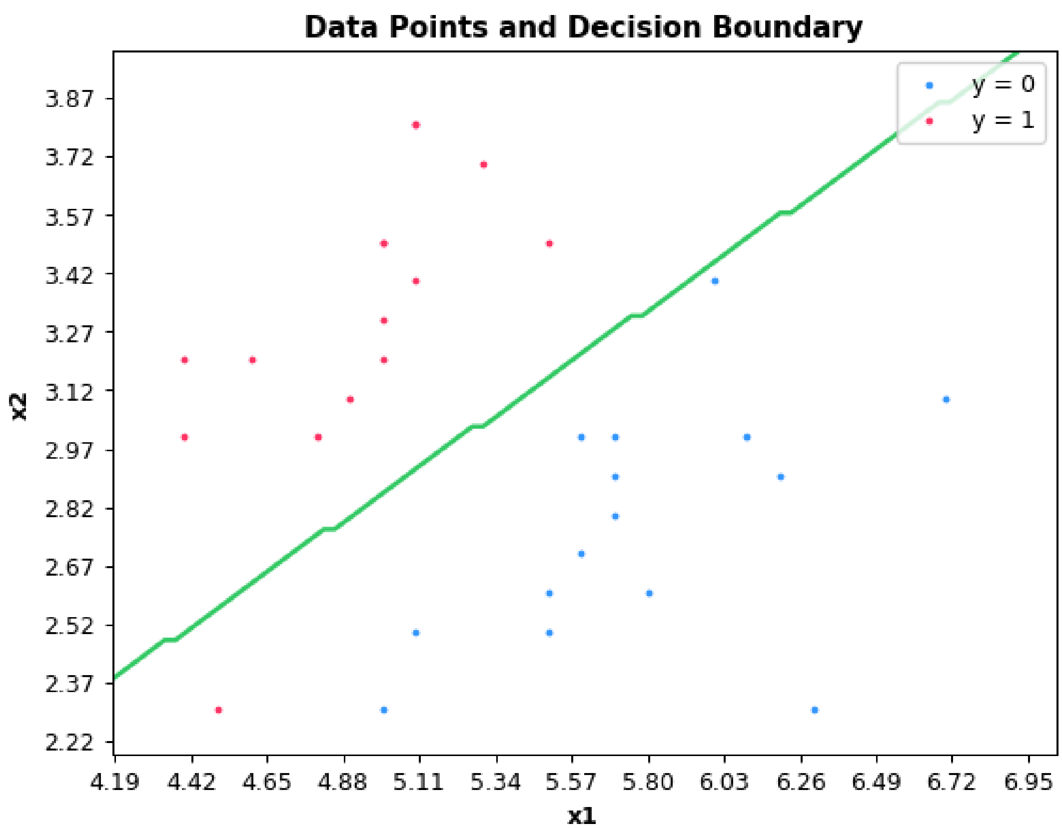
Problem 3.2

Evaluate the model on the test dataset (`test.mat`) and report the test error, defined as the percentage of misclassified test samples.

Solution

In [44]:
```
import torch
from scipy.io import loadmat

data = loadmat("test.mat")
x1 = torch.as_tensor(data["x1"], dtype=torch.float32)
x2 = torch.as_tensor(data["x2"], dtype=torch.float32)
y = torch.as_tensor(data["y"], dtype=torch.float32)

X = torch.cat([torch.ones_like(x1), x1, x2], dim=1)
Y = y

analyze_decision_boundary_2d(model, X, Y, scale=0.9)
```

```
{'x1_dim': (30, 1),
 'x2_dim': (30, 1),
 'y_dim': (30, 1),
 'eval_error': '3.333%'}
```



Problem 4

Download the MNIST dataset using the following example code:

```
# The code is moved to the solution code cell.
```

Please randomly choose 5,000 handwritten images from either the training or the testing dataset to construct your own dataset, with 500 data samples for each digit.

Solution

In [45]:

```python
SEED = 2
SIZE_PER_TARGET = 500

import torch
from torchvision.datasets import MNIST
from tempfile import tempdir

torch.manual_seed(SEED)

dataset = MNIST(tempdir, download=True, train=False)
scatter_indices = torch.randperm(len(dataset))
X, Y = dataset.data[scatter_indices], dataset.targets[scatter_indices]
D = {y.item(): X[Y == y][:SIZE_PER_TARGET] for y in Y.unique()}
X, Y = None, None

display({y: tuple(X.shape) for y, X in D.items()})
```

```
{0: (500, 28, 28),
 1: (500, 28, 28),
 2: (500, 28, 28),
 3: (500, 28, 28),
 4: (500, 28, 28),
 5: (500, 28, 28),
 6: (500, 28, 28),
 7: (500, 28, 28),
 8: (500, 28, 28),
 9: (500, 28, 28)}
```

Problem 4.1

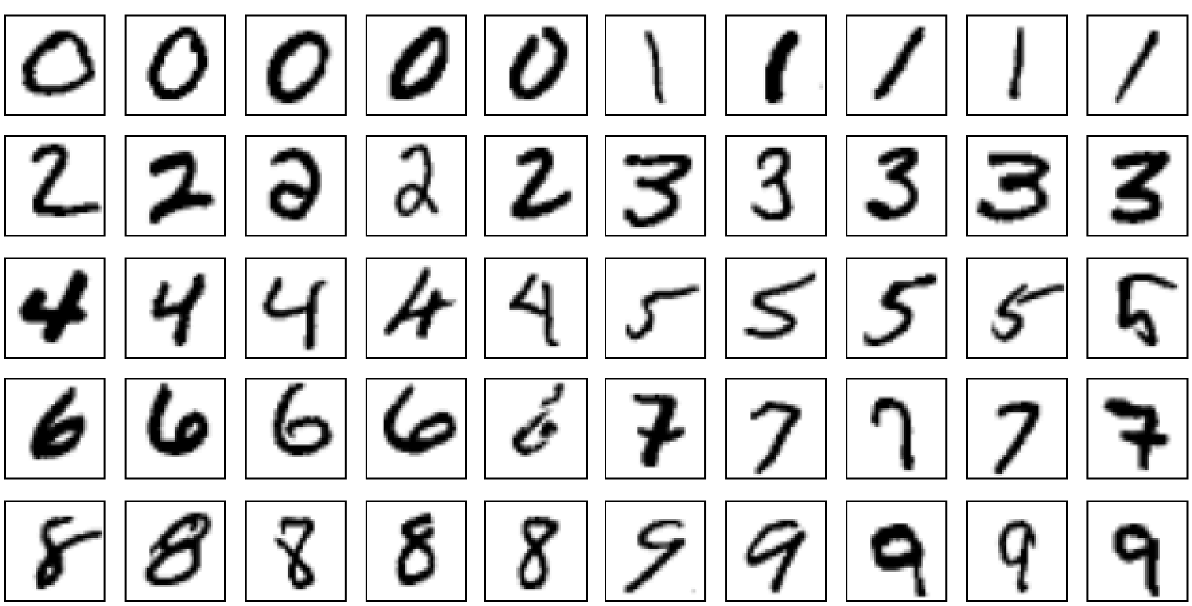
Use the following code to show 50 images in your own dataset.

 # The code is moved to the solution code cell.

Solution

In [46]:

```
import matplotlib.pyplot as plt


def plot_tensor_images(
    images: torch.Tensor,
    width: int,
    height: int,
    *,
    scale: float | None = None,
) -> None:
    width = int(width)
    height = int(height)
    scale = float(scale or 1.0)

    plt.figure(dpi=100 * scale, figsize=(width * scale, height * scale))
    for i in range(width * height):
        plt.subplot(height, width, i + 1)
        plt.imshow(images[i], cmap="binary")
        plt.xticks([])
        plt.yticks([])


plot_tensor_images(
    torch.cat([X[:5] for X in D.values()]),
    width=10,
    height=5,
    scale=0.9,
)
plt.show()
```



Problem 4.2

Apply PCA (Principal Component Analysis) to reduce the 784-dimensional data to 500, 300, 100, and 50 dimensions. For each reduction, show ten decoded results for each digit and analyze how the data reconstruction changes with decreasing dimensions.

In your report, interpret the results by discussing how the dimensionality reduction affects the quality of the decoded images and explain any observed trade-offs between dimensionality and image clarity.

Solution

Given a data matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, where m is the number of samples and n is the number of features ($n \geq k$), the Principal Component Analysis (PCA) process can be formulated as follows:

1. **Center the Data**: $\mathbf{X}_\mu = \mathbf{X} - \mu,\ \mu = \frac{1}{m} \sum_{i=1}^{m} \mathbf{X}_i$
2. **Compute the Covariance Matrix**: $\mathbf{C} = \frac{1}{m-1} \mathbf{X}_\mu^\top \mathbf{X}_\mu$
3. **Eigen Decomposition**: $\mathbf{C} = \mathbf{V} \mathbf{L} \mathbf{V}^\top$

   where:

   - $\mathbf{V} \in \mathbb{R}^{n \times n}$ is the matrix of eigenvectors.
   - $\mathbf{L} \in \mathbb{R}^{n \times n}$ is the diagonal matrix of eigenvalues.
4. **Select Top k Eigenvectors**: $\mathbf{V}_k \text{ where } \mathbf{V}_k \in \mathbb{R}^{n \times k}$
5. **Project the Data onto the Principal Components**: $\mathbf{X}_k = \mathbf{X}_\mu \mathbf{V}_k \text{ where } \mathbf{X}_k \in \mathbb{R}^{m \times k}$

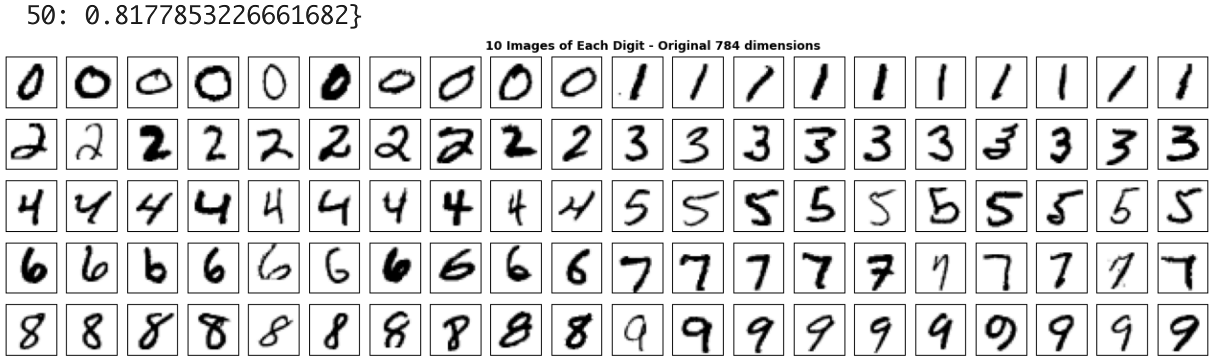
To reconstruct the data, we can use the following formula: $\hat{\mathbf{X}} = \mathbf{X}_k \mathbf{V}_k^\top + \mu$

In [47]:
```python
from dataclasses import dataclass
import torch


@dataclass
class PcaOutput:
    dim_X: torch.Size
    V_k: torch.Tensor
    X_k: torch.Tensor
    μ: torch.Tensor


def pca_encode(x: torch.Tensor, k: int) -> PcaOutput:
    dim_X = x.shape
    X = torch.as_tensor(x, dtype=torch.float32).flatten(1)
    μ = X.mean(dim=0)
    X_μ = X - μ
    C = (X_μ.T @ X_μ) / (X.size(0) - 1)
    L, V = torch.linalg.eigh(C)
    V_k = V[..., torch.argsort(L, descending=True)[: int(k)]]
    X_k = X_μ @ V_k

    return PcaOutput(
        dim_X=dim_X,
        V_k=V_k,
        X_k=X_k,
        μ=μ,
    )
```

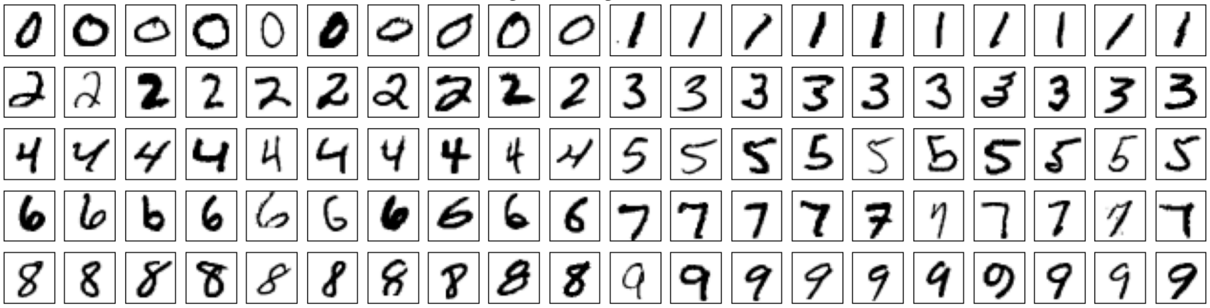
```python
def pca_decode(ctx: PcaOutput) -> torch.Tensor:
    X_hat = ctx.X_k @ ctx.V_k.T + ctx.μ
    return X_hat.reshape(ctx.dim_X)
```

In [48]:
```python
from torchmetrics.image import StructuralSimilarityIndexMeasure

COLUMN_COUNT = 20
IMAGE_COUNT = 100
LINE_COUNT = (IMAGE_COUNT + COLUMN_COUNT - 1) // COLUMN_COUNT

X = torch.cat([X[-10:] for X in D.values()]).float()
rank_to_ssim_score: dict[int, float] = {k: float() for k in [500, 300, 100, 50]}
get_ssim = StructuralSimilarityIndexMeasure(data_range=255.0)

plot_tensor_images(X, COLUMN_COUNT, LINE_COUNT, scale=0.9)
plt.text(
    -280,
    -140,
    f"10 Images of Each Digit - Original {X[0].shape.numel()} dimensions",
    fontweight="bold",
    ha="center",
    va="bottom",
)

for rank in rank_to_ssim_score.keys():
    X_hat = pca_decode(pca_encode(X, rank))
    plot_tensor_images(X_hat, COLUMN_COUNT, LINE_COUNT, scale=0.9)
    plt.text(
        -280,
        -140,
        f"10 Images of Each Digit - Recoded from {rank} dimensions",
        fontweight="bold",
        ha="center",
        va="bottom",
    )

    rank_to_ssim_score[rank] = get_ssim(X.unsqueeze(1),
X_hat.unsqueeze(1)).item()

display({"Rank": "SSIM", **rank_to_ssim_score})
plt.show()
```

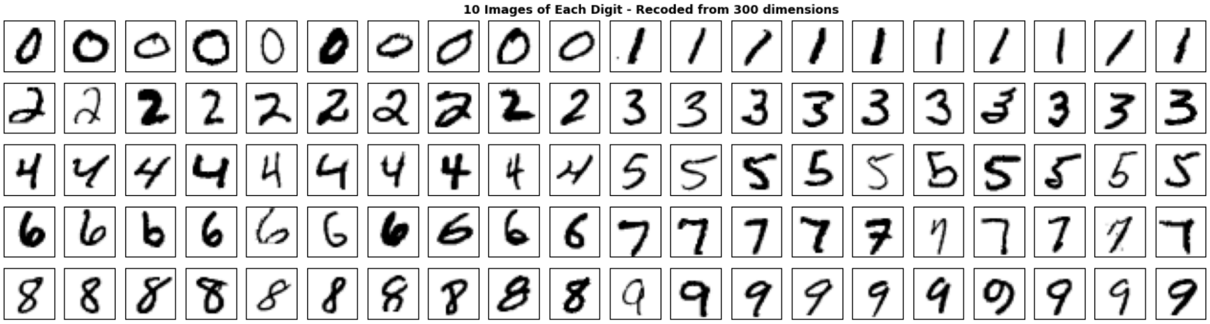
```
{'Rank': 'SSIM',
 500: 0.9999997615814209,
 300: 0.9999998211860657,
 100: 0.9999998211860657,
 50: 0.8177853226661682}
```



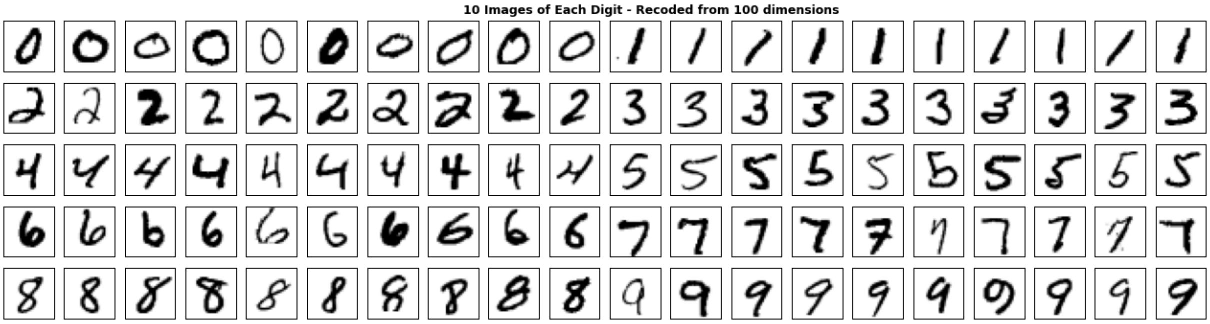
10 Images of Each Digit - Recoded from 500 dimensions



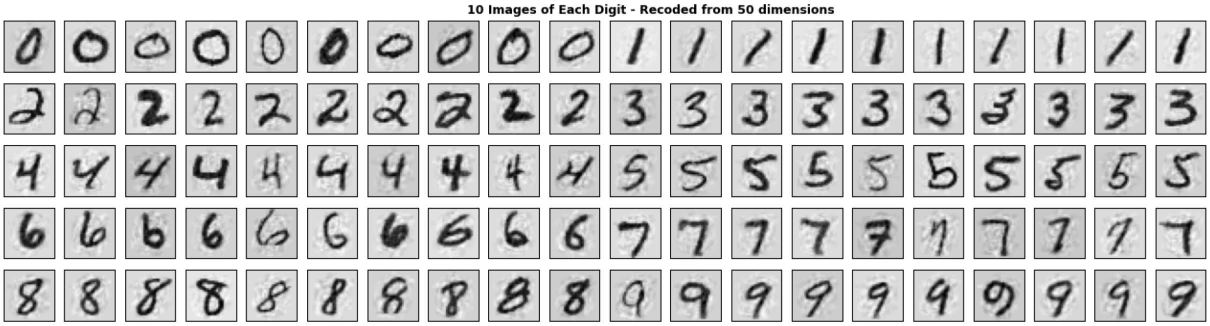
10 Images of Each Digit - Recoded from 300 dimensions



10 Images of Each Digit - Recoded from 100 dimensions



10 Images of Each Digit - Recoded from 50 dimensions



Discussion

When we reduce the PCA dimensions to 500, 300, and 100, the SSIM scores are nearly perfect (around 1), indicating high image quality. However, at 50 dimensions, the SSIM drops to about 0.818, resulting in blurrier, less detailed images. This highlights the trade-off between dimensionality reduction and image clarity in PCA.

Problem 4.3

Use PCA to project the MNIST dataset down to 2D and sample at least 4 decoded images from different regions across the four quadrants of this 2D projection.

Solution

In [49]:

```
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import torch
from dataclasses import dataclass


@dataclass
class PcaRecodeQuadOutput:
    Q1: list[torch.Tensor]
    Q2: list[torch.Tensor]
    Q3: list[torch.Tensor]
    Q4: list[torch.Tensor]


def pca_recode_quad(
    ctx: PcaOutput, quad_size: int | None = None
) -> PcaRecodeQuadOutput:
    quad_size = int(quad_size or 1)

    X_hat = pca_decode(ctx)
    output = PcaRecodeQuadOutput(Q1=[], Q2=[], Q3=[], Q4=[])

    for i in range(ctx.X_k.size(0)):
        pc1 = ctx.X_k[i, 0].item()
        pc2 = ctx.X_k[i, 1].item()

        if pc1 >= 0:
            if pc2 >= 0:
                output_quad = output.Q1
            else:
                output_quad = output.Q4
        else:
            if pc2 >= 0:
                output_quad = output.Q2
            else:
                output_quad = output.Q3

        if all(len(q) >= quad_size for q in output.__dict__.values()):
            break
        if len(output_quad) < quad_size:
            output_quad.append(X_hat[i])

    return output


X = torch.cat([X for X in D.values()])
pca_2d = pca_encode(X, 2)
decoded_images = pca_recode_quad(pca_2d, 4)

plt.figure(dpi=100, figsize=(10, 10))
plt.scatter(pca_2d.X_k[..., 0], pca_2d.X_k[..., 1], alpha=0.3, s=10, c="#39F")
plt.axhline(0, color="#F36", linestyle="-.")
plt.axvline(0, color="#F36", linestyle="-.")
```

```
plt.xlabel("PC 1")
plt.ylabel("PC 2")
plt.title("2D Projection of MNIST Dataset using PCA")
plt.grid(True)

x_min, x_max = plt.xlim()
y_min, y_max = plt.ylim()
offset_x = (x_max - x_min) / 15.0
offset_y = (y_max - y_min) / 15.0
offsets = {
    "Q1": [
        (x_max - offset_x, y_max - offset_y),
        (x_max - 2 * offset_x, y_max - offset_y),
        (x_max - offset_x, y_max - 2 * offset_y),
        (x_max - 2 * offset_x, y_max - 2 * offset_y),
    ],
    "Q2": [
        (x_min + offset_x, y_max - offset_y),
        (x_min + 2 * offset_x, y_max - offset_y),
        (x_min + offset_x, y_max - 2 * offset_y),
        (x_min + 2 * offset_x, y_max - 2 * offset_y),
    ],
    "Q3": [
        (x_min + offset_x, y_min + offset_y),
        (x_min + 2 * offset_x, y_min + offset_y),
        (x_min + offset_x, y_min + 2 * offset_y),
        (x_min + 2 * offset_x, y_min + 2 * offset_y),
    ],
    "Q4": [
        (x_max - offset_x, y_min + offset_y),
        (x_max - 2 * offset_x, y_min + offset_y),
        (x_max - offset_x, y_min + 2 * offset_y),
        (x_max - 2 * offset_x, y_min + 2 * offset_y),
    ],
}

for quad, images in decoded_images.__dict__.items():
    for idx, img in enumerate(images):
        imagebox = OffsetImage(img, cmap="binary", zoom=1.0)
        plt.gca().add_artist(AnnotationBbox(imagebox, offsets[quad][idx]))

plt.show()
```

2D Projection of MNIST Dataset using PCA
PC 1
PC 2
−2500
−2000
−1500
−1000
−500
0
500
1000
1000
500
0
−500
−1000
−1500