STAT 5244 – Unsupervised Learning

Homework 3

Name: Chuyang Su UNI: cs4570

1 Graphical Models.

1.1 Data Processing.

The log return transformation was applied to the daily closing prices. The daily log return r_t for a stock price P_t was calculated as:

$$r_t = \ln(P_t) - \ln(P_{t-1})$$

This dataset of log returns, spanning 1,228 trading days, was used for all subsequent graphical model fitting.

1.1.1 Descriptive Statistics.

The table below summarizes the descriptive statistics for the daily log returns.

Table 1: Descriptive Statistics of Daily Log Returns (Jan 2021 – Present)

	AAPL	AMZN	BAC	CVX	GOOGL	JNJ	JPM	KO	META	MSFT	NVDA	PFE	PG	WMT	XOM
Count	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00	1228.00
Mean ($\times 10^{-3}$)	0.63	0.27	0.53	0.64	1.02	0.33	0.81	0.38	0.65	0.66	**2.13**	-0.12	0.18	0.68	1.01
Std ($\times 10^{-2}$)	1.76	2.23	1.72	1.60	1.96	1.05	1.53	1.00	2.78	1.63	**3.29**	1.59	1.09	1.32	1.71
Min	-0.097	-0.151	-0.117	-0.086	-0.100	-0.079	-0.078	-0.072	**-0.306**	-0.080	-0.186	-0.070	-0.064	-0.121	-0.082
Max	0.143	0.127	0.081	0.085	0.097	0.060	0.109	0.046	0.209	0.097	**0.218**	0.103	0.042	0.091	0.062

The data clearly demonstrates the risk-return trade-off. The semiconductor stock `NVDA` shows the highest average daily return ($\sim 0.213\%$) but also the highest volatility (Standard Deviation: 3.29%) and largest maximum single-day return ($\sim 21.8\%$). Conversely, consumer staples stocks like `KO` (Coca-Cola) and `PG` (P&G) exhibit the lowest standard deviations ($\sim 1.0\%$), indicating high stability but lower returns. The largest single-day drop belongs to `META` (former FB) at -30.6%.

1.1.2 Time-Series Exploration.

The cumulative returns plot (Figure 1) illustrates the differential performance across sectors over the analysis period.

1.1.3 Correlation Analysis.

The correlation heatmap (Figure 2) reveals strong clustering of dependence among stocks within the same sector, which confirms the pervasive influence of systematic market risk.

Key Observations from the Heatmap:

- **Strong Correlation (0.6+):** High-tech stocks (`AAPL`, `MSFT`, `AMZN`, `GOOGL`, `NVDA`) are tightly coupled (e.g., `MSFT-AMZN` at 0.66, `MSFT-GOOGL` at 0.65). Financials (`JPM-BAC` at 0.82) and Energy stocks (`CVX-XOM` at 0.86) exhibit the highest correlations, reflecting their singular dependence on industry-specific factors (e.g., oil price, interest rates).
- **Weak/Low Correlation (0.0-0.3):** Healthcare stocks (`JNJ`, `PFE`) show low correlation with most other stocks (e.g., `JNJ` vs. Tech stocks often below 0.2), confirming their defensive, counter-cyclical nature.
- **Negative Correlation:** A notable weak negative correlation exists between the pharmaceutical stock `JNJ` and the high-growth technology stock `NVDA` (~ -0.09), suggesting an interesting divergence in their underlying risk drivers.

This preliminary analysis confirms the existence of strong, sector-specific dependencies, which the Graphical Lasso will aim to distill into a network of conditional dependencies.

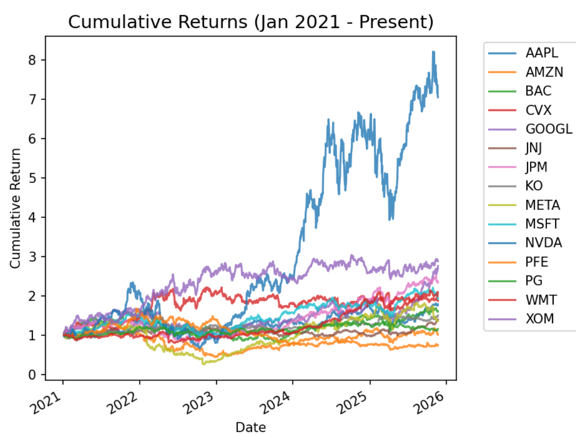

Figure 1: Cumulative Log Returns of Selected Stocks (Jan 2021 - Present)

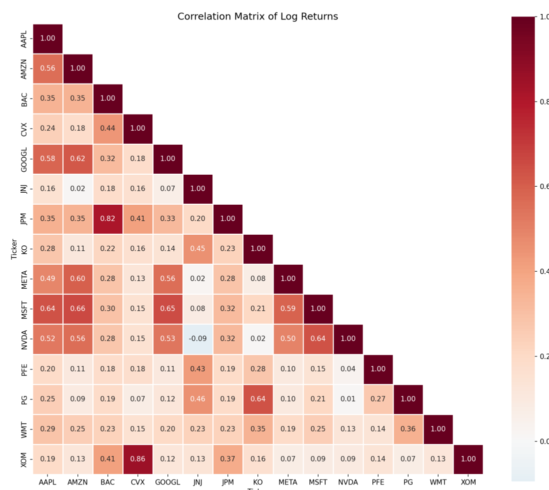

Figure 2: Correlation Heatmap of Daily Log Returns

1.2 Graphical Lasso.

Figure 3 shows the estimated precision matrices from both the Gaussian Graphical Lasso and the nonparanormal (rank-based) Graphical Lasso. The two heatmaps are almost identical, indicating that although individual stock returns are heavy-tailed, the dependence structure is well approximated by a Gaussian copula. Consequently, both methods recover essentially the same sparse conditional dependence network, suggesting that the underlying structure is stable, low-dimensional, and largely driven by sector-level factors.

The estimated graph highlights several strong conditional dependencies (e.g., AMZN–KO, JPM–GOOGL, WMT–BAC, NVDA–PFE) and a clear technology cluster consisting of AAPL, MSFT, GOOGL, AMZN, and META. NVIDIA does not join this cluster, likely due to its unusually strong and volatile performance during the sample period, which weakens its partial correlations with the other technology stocks after conditioning on the full set of variables.

The regularization parameter α was selected via cross-validated Gaussian log-likelihood over a grid of 30 values spanning $\log_{10}(0.01)$ to $\log_{10}(0.8)$. This criterion is appropriate for unsupervised graphical models, as the validation likelihood measures generalization of the estimated precision matrix. The optimal values were

$$\alpha_{\text{Gaussian}} = 0.021287, \qquad \alpha_{\text{Nonparanormal}} = 0.013528.$$

For brevity, only the tuning curve for the Gaussian estimator is shown in Figure 4.

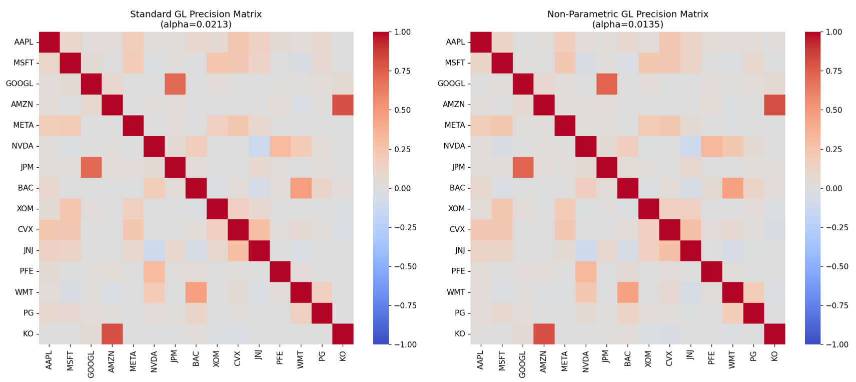

Figure 3: Gaphic Lasso Estimated Precision Matrices: Standard (Left) vs. Non-Parametric (Right)

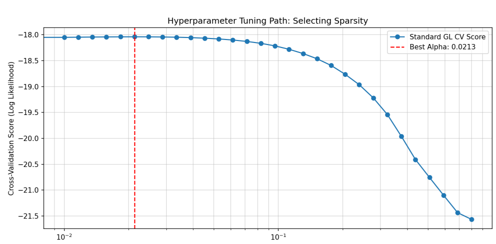

Figure 4: Cross-Validated Log-Likelihood Curve for Standard Graphical Lasso

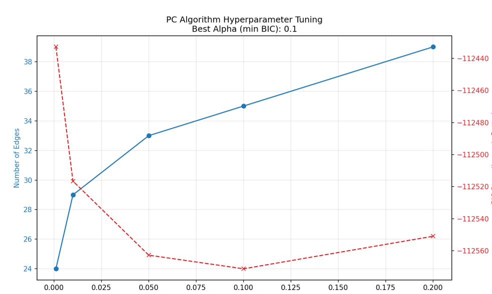

Figure 5: Best Alpha Selection for PC Algorithm via BIC Score

1.3 PC Algorithm.

To determine the optimal regularization level for the PC algorithm, we evaluated the Bayesian Information Criterion (BIC) across a range of significance thresholds

$$\alpha \in \{0.001, 0.01, 0.05, 0.1, 0.2\}.$$

The resulting BIC scores are shown in Figure 5. Although $\alpha = 0.05$ is often used as a conventional threshold, the BIC curve indicates that the model achieves its minimum score at $\alpha = 0.1$, implying that this level of sparsity provides the best balance between model fit and complexity.

Based on this criterion, we select $\alpha = 0.1$ and construct the final directed graph using the PC algorithm. The resulting structure is displayed in Figure 6, which represents the learned conditional independence relations and the corresponding Markov equivalence class under this optimal parameter choice.

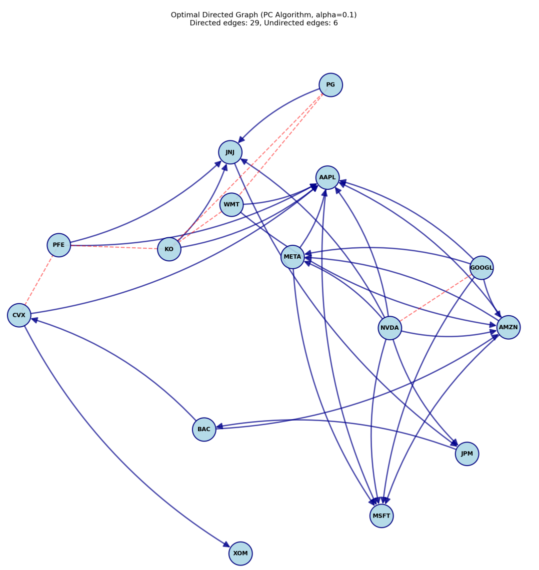

Figure 6: Best PC Algorithm Graph at $\alpha = 0.1$

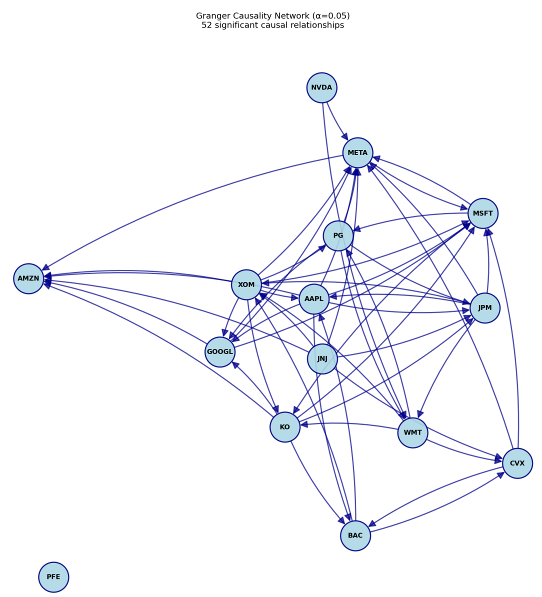

Figure 7: Granger Causal Graphical Model at $\alpha = 0.05$, Max Lag = 5

1.4 Comparison and Interpretation.

The learned structure from Figure 6 displays a clear pattern: technology stocks exhibit substantially richer and more directional dependency relationships compared to the other sectors. For example, MSFT appears as a child of all technology names, including AAPL, GOOGL, NVDA, AMZN and META, indicating that its conditional distribution depends structurally on multiple peers within the same sector. In contrast, NVDA and GOOGL share an undirected edge but act as parents to all other surrounding nodes, placing them in more central positions within the technology cluster. These findings are consistent with the results from part (b), in which the precision matrices also implied a tightly connected technology block.

Other sectors exhibit markedly more isolated behavior. For instance, in the energy sector, XOM is connected only to CVX, and once CVX is conditioned on, XOM becomes conditionally independent of all other stocks in the universe. This highlights both the internal coherence of the energy sector and its relative independence from the remaining market, which are consistent with common sense.

An interesting contrast emerges when comparing the PC graph with the precision matrices from part (b). Pairs such as KO–AMZN and JPM–GOOGL exhibit strong partial correlations under the Gaussian and nonparanormal graphical lasso, yet no direct edge appears between them in the PC graph. This difference reflects the distinct logics of the two models: graphical lasso identifies pairwise partial correlations, while the PC algorithm searches for a directed acyclic graph that satisfies a complete set of conditional independence relations. As a result, some associations are represented not by direct edges but by indirect paths—for example, KO connects to AMZN through AAPL or WMT. This is also consistent with economic intuition, as consumer staples and mega-cap technology firms often share indirect market linkages through broader macro or demand channels.

1.5 Granger Causal Graphical Model.

To evaluate the stability of the Granger causality model, we performed a hyperparameter sweep over significance levels $\alpha \in \{0.01, 0.05, 0.1\}$ and maximum lags $\{1, 3, 5, 7, 10\}$. The resulting edge counts and graph densities are summarized in Figure 8. Because interpretable causal networks should remain reasonably sparse, we selected $\alpha = 0.05$ and a maximum lag of 5.

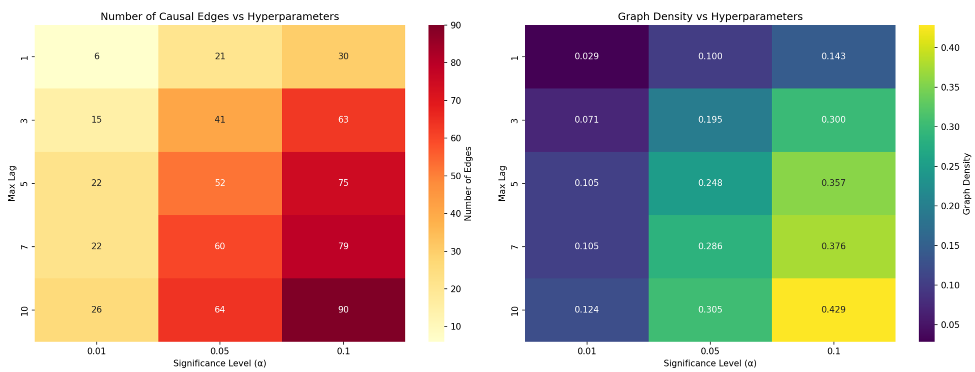

Figure 8: Granger Causality Model Tuning: Edge Counts (Left) and Graph Density (Right) Across Significance Levels and Maximum Lags

However, even under this moderate configuration, the resulting Granger graph still contains 52 significant edges (density = 0.248), which is far more dense than the sparse and interpretable structures obtained in parts (b) and (c). Figure 9 illustrates the adjacency matrix of significant Granger causal relationships, showing an unusually high concentration of edges across sectors. This density pattern suggests that the Granger framework is detecting a large number of spurious lead–lag relations rather than meaningful predictive structure.

The final inferred Granger network is shown in Figure 7. The resulting graph is visibly dense and lacks the sectoral organization and coherent clusters observed in the graphical lasso and PC algorithm results. Instead of revealing identifiable industry-level dependency patterns, the Granger graph displays widespread, cross-sector connections that contradict well-known market structure and are characteristic of noise-driven statistical artifacts in financial return data.

Together, these observations indicate that the Granger causality model does not fit the dataset well. Even after choosing a reasonably conservative hyperparameter setting, the method overfits the volatility and idiosyncratic noise inherent in daily stock returns, producing a dense and unstable network that fails to capture the stable structural dependencies recovered by the graphical lasso and PC models.

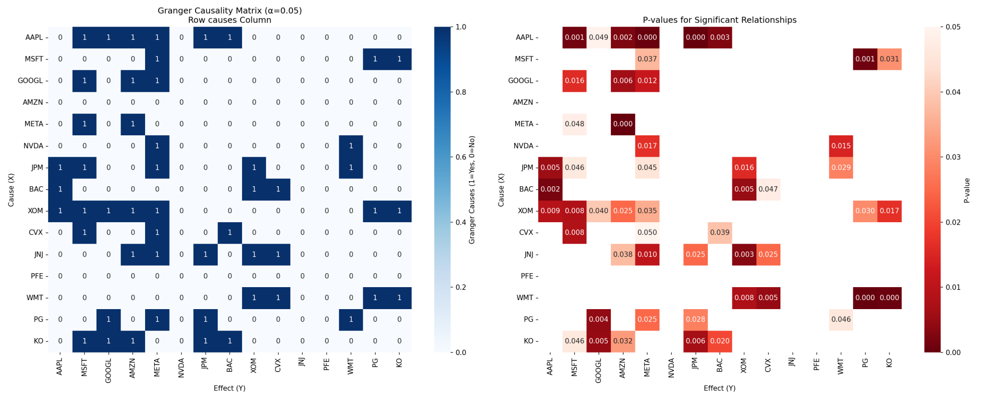

Figure 9: Granger Causality Adjacency Matrix at $\alpha = 0.05$, Max Lag = 5

2 Density estimation & Generative Models.

2.1 Kernel Density Estimation (KDE).

A grid search was performed on the bandwidth parameter h across three spaces (PCA-20/50 and original 64-dimensional space), with 5-fold cross-validation used to select the optimal value based on average log-likelihood, the result is shown in Figure 10. Since the CV score in the original space deteriorated significantly (-19606 vs -8081), confirming the curse of dimensionality, the PCA-20 space (optimal $h = 0.580$) was chosen for subsequent sampling. The pixel intensity distribution of generated samples closely matches the original data (mean 4.88 vs 4.95, standard deviation 6.02 vs 6.64), but spatial structure differs: generated samples exhibit only 23.91% sparsity compared to the original 48.93%, a 50% reduction, indicating KDE's tendency to produce denser pixel distributions. This results in some digits (e.g., 1, 0, 8) being clearly recognizable, though overall contrast remains insufficient(as shown in Figure 11).

2.2 Generative Adversarial Network (GAN).

Overall, the parameter combination `latent_dim=32`, `lr_g=0.0001`, `G=[128, 256]` proves most suitable: it achieves the lowest `std_diff` (indicating generated sample distributions closest in shape to the original data), while maintaining competitive `mean_diff`. Moreover, `latent=32` is more lightweight and less prone to overfitting, with diversity ≈ 1.7 indicating no severe mode collapse.

Other hyperparameters include α_D fixed at 0.0002, consistent with the DCGAN paper. The final choice employs $\alpha_G < \alpha_D$, allowing the Generator to learn more slowly, since my previous experiments with $\alpha_G = \alpha_D$ resulted in training collapse due to Generator dominance over the Discriminator.

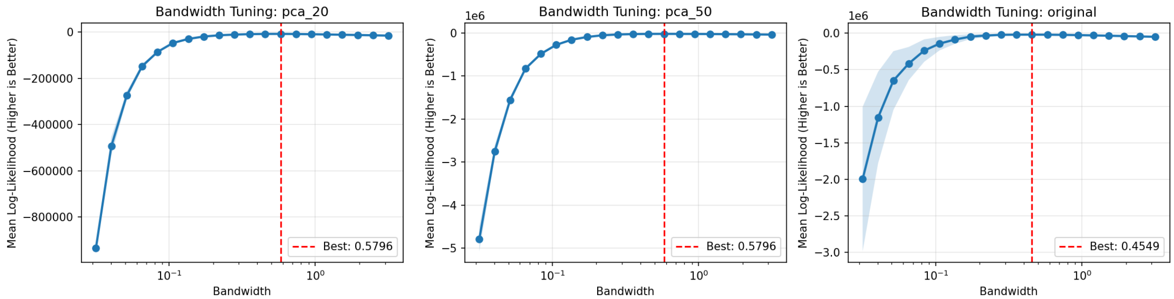

Figure 10: KDE Bandwidth Tuning via 5-Fold Cross-Validation in PCA-20, PCA-50, and Original Spaces

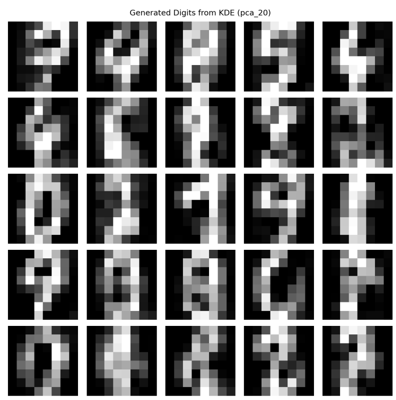

Figure 11: Generated Samples from KDE

Figure 12: Generated Samples from GAN

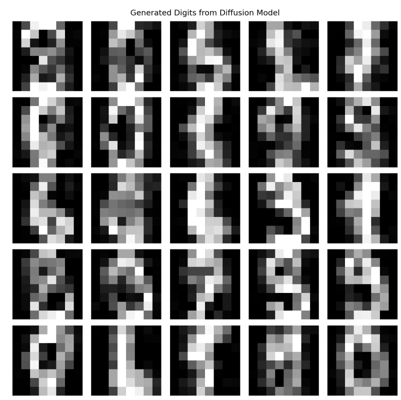

Figure 13: Generated Samples from Diffusion Model

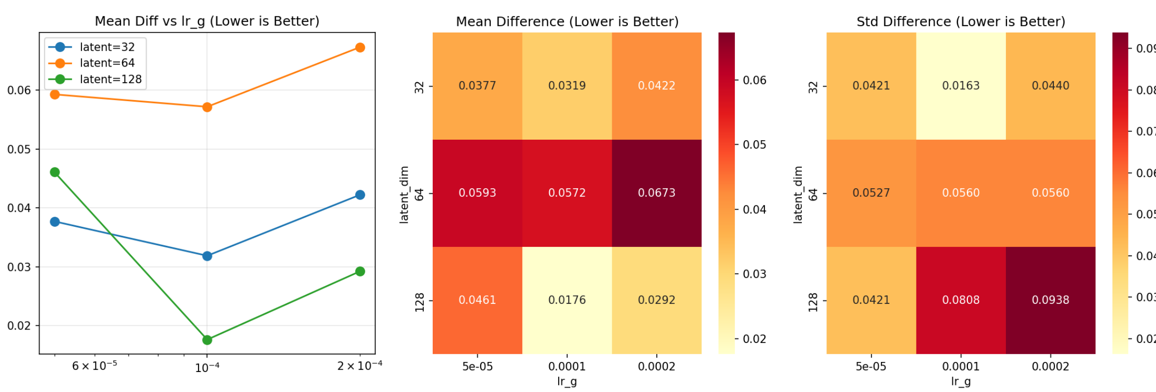

Figure 14: GAN Hyperparameter Tuning Results

Under optimal parameters, the model training process is illustrated in Figure 14. Compared to my previous experiments, both losses converge relatively well, stabilizing around 1.0–1.2, with Discriminator accuracy exceeding 0.5. However, a significant disparity exists in distinguishing real versus generated samples: accuracy remains above 0.8 when identifying fake samples, but falls below 0.6 for real samples. This reflects an inherent limitation of Vanilla GAN—the min-max game struggles to reach Nash equilibrium, explaining the research motivation behind subsequent methods such as WGAN and Diffusion Models.

Generated sample examples are shown in Figure 12. Several digits (e.g., 0, 1, 2, 3, 5, 6, 9) exhibit high recognizability, with overall performance significantly superior to KDE.

2.3 Denoising Diffusion Model.

In this experiment, we optimized three hyperparameters: learning rate (lr), number of timesteps, and network architecture. Among all 18 combinations, we used heatmaps of mean_diff and std_diff to determine lr and timesteps, while final loss guided the selection of architecture. The results are shown in Figure 15.

From the final loss perspective, architecture $[256, 512, 256]$ significantly outperforms $[128, 256, 128]$ on average. Building upon this, the combination of timesteps $= 500$ and $\mathrm{lr} = 0.0005$ achieves the lowest mean_diff (0.0054), with std_diff also within an acceptable range. Therefore, we select $\mathrm{lr} = 0.0005$, timesteps $= 500$, and architecture $= [256, 512, 256]$ as our optimal model configuration. All subsequent discussions are based on this setting.

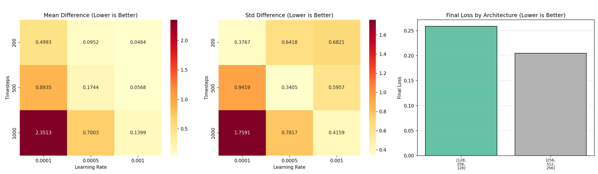

Figure 15: Diffusion Model Hyperparameter Tuning Results

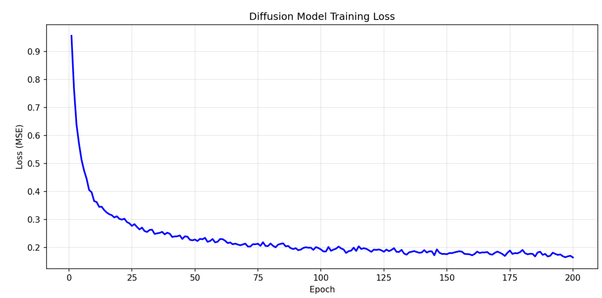

Figure 16: Training Trajectory of the Diffusion Model

The training trajectory of the diffusion model is shown in Figure 16, and representative generated samples are displayed in Figure 13.

Comparing the generated results from KDE, GAN, and diffusion, several clear patterns emerge. From a statistical perspective, the GAN produces samples with higher overall brightness and stronger contrast, and its sparsity level is closer to that of the original dataset. This suggests that the generator is able to capture certain global statistical features of the data distribution. However, in terms of structural clarity and recognizability, the diffusion model performs markedly better: its generated digits exhibit sharp contours and well-defined shapes, whereas the outputs of KDE and GAN appear significantly more blurred.

The weaker structural fidelity of the GAN can be attributed to the imbalance in the adversarial game. In our setting, the discriminator is insufficiently strong and fails to reliably distinguish real from generated digits. As a result, the generator can easily "fool" the discriminator without learning sharper or more realistic digit structures. This issue is further exacerbated by the high sensitivity of the vanilla GAN to hyperparameters, making it difficult to achieve a stable equilibrium between the generator and discriminator. These challenges highlight the fundamental limitations of the GAN approach in this task and explain the performance gap relative to the diffusion model.

A Appendix: Code Implementation.

For further details, please visit my GitHub repository:
https://github.com/Schuyn/Unsupervised-Learning-Homework.git

A.1 Problem 1: Graphical Models.

A.1.1 Main Script.

```python
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-23 17:41:45
LastEditTime: 2025-11-24 12:27:59
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/main.py
Description:
    Main script to run data processing and graphical model fitting for Homework 3.
'''
from Prob1_utils import Prob1Analysis

def main():
    # Problem 1a
    p1=Prob1Analysis()
    log_returns = p1.process_stock_data(verbose=False)

    # Problem 1b
    glasso_results = p1.fit_glasso_models(verbose=False)

    # Problem 1c
    pc_results = p1.fit_pc_model(verbose=False)

    # Problem 1e
    granger_results = p1.fit_granger_model(verbose=True)

if __name__ == "__main__":
    main()
```

A.1.2 Utils Script.

```python
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-23 17:27:40
LastEditTime: 2025-11-26 11:22:20
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/prob1_utils.py
Description:
    Utility functions and classes for Problem 1 analysis in Homework 3.
'''
import os
import yfinance as yf
import pandas as pd
import numpy as np
```

```python
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
import networkx as nx
from sklearn.covariance import GraphicalLassoCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, LassoCV
from statsmodels.tsa.stattools import grangercausalitytests
from causallearn.search.ConstraintBased.PC import pc

class Prob1Analysis:
    def __init__(self,
                 output_dir='Homework 3/Code/Data',
                 figure_dir='Homework 3/Latex/Figures',
                 start_date="2021-01-01"):
        self.output_dir = output_dir
        self.figure_dir = figure_dir
        self.start_date = start_date

        os.makedirs(self.output_dir, exist_ok=True)
        os.makedirs(self.figure_dir, exist_ok=True)

        self.raw_data_file = os.path.join(self.output_dir, 'raw_stock_data.csv')
        self.log_returns_file = os.path.join(self.output_dir, 'log_returns.csv')

        self.tickers = ["AAPL", "MSFT", "GOOGL", "AMZN", "META", "NVDA",
                        "JPM", "BAC", "XOM", "CVX", "JNJ", "PFE", "WMT", "PG", "KO"]

        self.sectors = {
            'Tech': ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'META', 'NVDA'],
            'Finance': ['JPM', 'BAC'],
            'Energy': ['XOM', 'CVX'],
            'Healthcare': ['JNJ', 'PFE'],
            'Consumer': ['WMT', 'PG', 'KO']
        }

        self.log_returns = None
        self.glasso_results = None

    def process_stock_data(self, verbose=False):
        # Download stock data and compute log returns
        if os.path.exists(self.log_returns_file):
            log_returns = pd.read_csv(self.log_returns_file, index_col=0,
                                      parse_dates=True)

        else:
            print(f"Downloading data from yfinance...")
            raw_data = yf.download(self.tickers, start=self.start_date, end=None)
            raw_data.to_csv(self.raw_data_file)

```

```python
            # Extract closing prices and compute log returns
            close_prices = raw_data['Close']
            print(f"\nData shape: {close_prices.shape}")
            print(f"Date range: {close_prices.index[0]} to {close_prices.index[-1]}")

            log_returns = np.log(close_prices / close_prices.shift(1)).dropna()
            log_returns.to_csv(self.log_returns_file)
            # print(f"\nMissing values per stock:\n{log_returns.isnull().sum()}")

        self.log_returns = log_returns

        # Visual exploration
        if verbose:
            self._plot_visual_exploration()

        return log_returns

    def _plot_visual_exploration(self):
        # Summary statistics
        summary_stats = self.log_returns.describe()
        summary_stats.to_csv(os.path.join(self.output_dir, 'summary_statistics.csv'))

        # Plot correlation heatmap
        plt.figure(figsize=(12, 10))
        corr_matrix = self.log_returns.corr()
        mask = np.triu(np.ones_like(corr_matrix, dtype=bool), k=1)
        sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='RdBu_r', center=0,
                    mask=mask, square=True, linewidths=0.5)
        plt.title('Correlation Matrix of Log Returns', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1a_correlation_heatmap.png'), dpi=150)
        plt.show()

        # Plot cumulative returns
        plt.figure(figsize=(14, 8))
        cumulative_returns = (1 + self.log_returns).cumprod()
        cumulative_returns.plot(alpha=0.8)
        plt.title('Cumulative Returns (Jan 2021 - Present)', fontsize=14)
        plt.xlabel('Date')
        plt.ylabel('Cumulative Return')
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1a_cumulative_returns.png'), dpi=150)
        plt.show()

        # Relationships within different industries
        fig, axes = plt.subplots(2, 3, figsize=(15, 10))
```

```
        axes = axes.flatten()

        for idx, (sector, stocks) in enumerate(self.sectors.items()):
            sector_corr = self.log_returns[stocks].corr()
            sns.heatmap(sector_corr, annot=True, fmt='.2f', cmap='RdBu_r',
                                                 center=0,
                        ax=axes[idx], square=True, vmin=-1, vmax=1)
            axes[idx].set_title(f'{sector} Sector Correlation')

        axes[-1].axis('off')
        plt.suptitle('Within-Sector Correlation Analysis', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1a_sector_correlations.png'
                                           ), dpi=150)
        plt.show()

    def fit_glasso_models(self, verbose=False):
        if self.log_returns is None:
            self.process_stock_data()

        scaler = StandardScaler()
        X_standardized = scaler.fit_transform(self.log_returns.values)

        # Grid search best alpha values
        alphas_grid = np.logspace(np.log10(0.01), np.log10(0.8), 30)

        # Standard Graphical Lasso
        if verbose:
            print("Fitting Standard Graphical LassoCV...")
        gl_cv = GraphicalLassoCV(alphas=alphas_grid,cv=5, n_jobs=-1, max_iter=
                                           3000,tol=1e-4)
        gl_cv.fit(X_standardized)
        # print(gl_cv.cv_results_)

        # Non-parametric (Rank-based) Graphical Lasso
        if verbose:
            print("Fitting Non-parametric (Rank) Graphical LassoCV...")

        # Transform data to ranks
        n = self.log_returns.shape[0]
        X_ranks = self.log_returns.rank() / (n + 1)
        X_normal_scores = norm.ppf(X_ranks)
        X_np_standardized = StandardScaler().fit_transform(X_normal_scores)

        gl_np = GraphicalLassoCV(alphas=alphas_grid,cv=5, n_jobs=-1, max_iter=
                                           3000,tol=1e-4)
        gl_np.fit(X_np_standardized)

        # Compute partial correlations
        prec = gl_cv.precision_
        d = np.sqrt(np.diag(prec))
        partial_corr = -prec / np.outer(d, d)
        np.fill_diagonal(partial_corr, 1.0)

```

```python
        prec_np = gl_np.precision_
        d_np = np.sqrt(np.diag(prec_np))
        partial_corr_np = -prec_np / np.outer(d_np, d_np)
        np.fill_diagonal(partial_corr_np, 1.0)

        self.glasso_results = {
            'gl_cv': gl_cv,
            'gl_np': gl_np,
            'precision_gl': prec,
            'precision_np': prec_np,
            'partial_corr_gl': partial_corr,
            'partial_corr_np': partial_corr_np
        }

        # Verbose output and visualization
        if verbose:
            self._plot_glasso_results(gl_cv,gl_np,partial_corr,partial_corr_np
                )


        return self.glasso_results

    def _plot_glasso_results(self, gl_cv, gl_np, partial_corr, partial_corr_np
            ):
        print("\n--- Graphical Lasso Results ---")
        print(f"Standard GL Best Alpha: {gl_cv.alpha_:.6f}")
        print(f"Non-Parametric GL Best Alpha: {gl_np.alpha_:.6f}")

        # Plot CV curves
        plt.figure(figsize=(10, 5))
        cv_results = gl_cv.cv_results_
        cv_means = cv_results['mean_test_score']
        cv_alphas = cv_results['alphas']

        plt.semilogx(cv_alphas, cv_means, 'o-', label='Standard GL CV Score')
        plt.axvline(gl_cv.alpha_, linestyle='--', color='r', label=f'Best
            Alpha: {gl_cv.alpha_:.4f}')

        plt.xlabel('Regularization Parameter (Alpha/Lambda)')
        plt.ylabel('Cross-Validation Score (Log Likelihood)')
        plt.title('Hyperparameter Tuning Path: Selecting Sparsity')
        plt.legend()
        plt.grid(True, which="both", ls="-", alpha=0.5)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1b_glasso_cv_curve.png'),
            dpi=150)
        plt.show()

        # Plot precision matrices
        fig, axes = plt.subplots(1, 2, figsize=(16, 7))

        # Standard
        prec = gl_cv.precision_
        d = np.sqrt(np.diag(prec))
```

```python
        partial_corr = -prec / np.outer(d, d)
        np.fill_diagonal(partial_corr, 1.0)

        sns.heatmap(partial_corr, ax=axes[0], cmap='coolwarm',
                    xticklabels=self.tickers, yticklabels=self.tickers, vmin=-1, vmax=1)
        axes[0].set_title(f'Standard GL Precision Matrix\n(alpha={gl_cv.alpha_:.4f})')

        # Non-parametric
        prec_np = gl_np.precision_
        d_np = np.sqrt(np.diag(prec_np))
        partial_corr_np = -prec_np / np.outer(d_np, d_np)
        np.fill_diagonal(partial_corr_np, 1.0)

        sns.heatmap(partial_corr_np, ax=axes[1], cmap='coolwarm',
                    xticklabels=self.tickers, yticklabels=self.tickers, vmin=-1, vmax=1)
        axes[1].set_title(f'Non-Parametric GL Precision Matrix\n(alpha={gl_np.alpha_:.4f})')

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1b_glasso_precision_matrices.png'), dpi=150)
        plt.show()

    def calculate_bic_for_dag(self, data, G):
        n_samples, n_features = data.shape
        bic_total = 0

        nodes = G.get_nodes()

        for i, node in enumerate(nodes):
            neighbors = G.get_adjacent_nodes(node)
            neighbor_indices = [nodes.index(n) for n in neighbors]

            y = data.iloc[:, i].values

            if len(neighbor_indices) == 0:
                residuals = y - np.mean(y)
            else:
                X = data.iloc[:, neighbor_indices].values
                model = LinearRegression()
                model.fit(X, y)
                residuals = y - model.predict(X)

            variance = max(1e-9, np.var(residuals))

            # Log-likelihood term for this node
            log_likelihood = -0.5 * n_samples * (np.log(2 * np.pi * variance) + 1)

```

```
            # Number of parameters = number of parents + 1 (intercept/variance
                )
            k = len(neighbor_indices) + 1

            # BIC = k * ln(n) - 2 * ln(L)
            bic_node = k * np.log(n_samples) - 2 * log_likelihood
            bic_total += bic_node

        return bic_total

    def fit_pc_model(self, alphas=[0.001, 0.01, 0.05, 0.1, 0.2], verbose=False
        ):
        if self.log_returns is None:
            self.process_stock_data()

        data_np = self.log_returns.values
        labels = self.log_returns.columns.tolist()

        results = []
        best_bic = float('inf')
        best_graph = None
        best_alpha = 0.05

        if verbose:
            print(f"Tuning PC Algorithm over alphas: {alphas}...")

        for alpha in alphas:
            cg = pc(data_np, alpha, "fisherz")
            n_edges = cg.G.get_num_edges()
            bic = self.calculate_bic_for_dag(self.log_returns, cg.G)

            results.append({
                'alpha': alpha,
                'n_edges': n_edges,
                'bic': bic,
                'graph': cg.G
            })

            if verbose:
                print(f"  Alpha: {alpha:<6} | Edges: {n_edges:<4} | BIC: {bic
                    :.2f}")

            if bic < best_bic:
                best_bic = bic
                best_graph = cg.G
                best_alpha = alpha

        if verbose:
            self._plot_pc_results(results, best_alpha, best_graph, labels)

        return {
            'best_graph': best_graph,
            'best_alpha': best_alpha,
            'results': results
```

```
        }

    def _plot_pc_results(self, results, best_alpha, best_graph, labels):
        alphas_plot = [r['alpha'] for r in results]
        edges_plot = [r['n_edges'] for r in results]
        bics_plot = [r['bic'] for r in results]

        fig, ax1 = plt.subplots(figsize=(10, 6))

        color = 'tab:blue'
        ax1.set_xlabel('Alpha (Significance Level)')
        ax1.set_ylabel('Number of Edges', color=color)
        ax1.plot(alphas_plot, edges_plot, marker='o', color=color, label='Sparsity (Edges)')
        ax1.tick_params(axis='y', labelcolor=color)
        ax1.grid(True, alpha=0.3)

        ax2 = ax1.twinx()
        color = 'tab:red'
        ax2.set_ylabel('BIC Score (Lower is Better)', color=color)
        ax2.plot(alphas_plot, bics_plot, marker='x', linestyle='--', color=color, label='BIC Score')
        ax2.tick_params(axis='y', labelcolor=color)

        plt.title(f'PC Algorithm Hyperparameter Tuning\nBest Alpha (min BIC): {best_alpha}')
        fig.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1c_pc_tuning.png'), dpi=150)
        plt.show()

        print(f"\nBest Alpha selected by BIC: {best_alpha}")
        print(f"Number of edges in best graph: {best_graph.get_num_edges()}")

        G_nx = nx.DiGraph()
        nodes = best_graph.get_nodes()

        for i in range(len(nodes)):
            G_nx.add_node(i, label=labels[i])

        graph_edges = best_graph.get_graph_edges()
        directed_edges = []
        undirected_edges = []

        for edge in graph_edges:
            node1 = edge.get_node1()
            node2 = edge.get_node2()
            idx1 = nodes.index(node1)
            idx2 = nodes.index(node2)

            endpoint1 = edge.get_endpoint1()
            endpoint2 = edge.get_endpoint2()

```

```
            ep1_name = endpoint1.name if hasattr(endpoint1, 'name') else str(endpoint1)
            ep2_name = endpoint2.name if hasattr(endpoint2, 'name') else str(endpoint2)
            if ep1_name == 'TAIL' and ep2_name == 'ARROW':
                directed_edges.append((idx1, idx2))
                G_nx.add_edge(idx1, idx2)
            elif ep1_name == 'ARROW' and ep2_name == 'TAIL':
                directed_edges.append((idx2, idx1))
                G_nx.add_edge(idx2, idx1)
            else:
                undirected_edges.append((idx1, idx2))
                G_nx.add_edge(idx1, idx2)
                G_nx.add_edge(idx2, idx1)

        plt.figure(figsize=(16, 16))
        pos = nx.spring_layout(G_nx, k=2.5, iterations=100, seed=42)

        nx.draw_networkx_nodes(G_nx, pos, node_size=2000, node_color='lightblue',
                              alpha=0.9, edgecolors='navy', linewidths=2)
        nx.draw_networkx_labels(G_nx, pos, {i: labels[i] for i in range(len(labels))},
                               font_size=11, font_weight='bold')

        if directed_edges:
            nx.draw_networkx_edges(G_nx, pos, edgelist=directed_edges,
                                  edge_color='darkblue', arrows=True, arrowsize=25,
                                  arrowstyle='-|>', connectionstyle='arc3,rad=0.15',
                                  width=2.5, alpha=0.7, node_size=2000)

        if undirected_edges:
            nx.draw_networkx_edges(G_nx, pos, edgelist=undirected_edges,
                                  edge_color='red', arrows=False,
                                  width=2.0, alpha=0.5, style='dashed')

        plt.title(f"Optimal Directed Graph (PC Algorithm, alpha={best_alpha})\n"
                 f"Directed edges: {len(directed_edges)}, Undirected edges: {len(undirected_edges)}",
                 fontsize=14, pad=20)
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1c_pc_graph.png'), dpi=150, bbox_inches='tight')
        plt.show()

    def fit_granger_model(self, max_lag=5, significance_level=0.05, tune_params=True, verbose=False):
```

```python
        if self.log_returns is None:
            self.process_stock_data()

        # Hyperparameter tuning
        if tune_params and verbose:
            self._tune_granger_parameters(max_lag_range=[1, 3, 5, 7, 10],
                                          alpha_range=[0.01, 0.05, 0.1])

        n_features = len(self.tickers)
        granger_matrix = np.zeros((n_features, n_features))
        p_value_matrix = np.zeros((n_features, n_features))
        optimal_lags = np.zeros((n_features, n_features), dtype=int)

        if verbose:
            print(f"\nRunning Granger Causality Tests (max_lag={max_lag},
                                                      alpha={significance_level})
                                                      ...")

        for i, cause in enumerate(self.tickers):
            for j, effect in enumerate(self.tickers):
                if i == j:
                    continue

                data = self.log_returns[[effect, cause]].dropna()

                try:
                    test_result = grangercausalitytests(data, max_lag, verbose
                                                             =False)

                    min_p_value = 1.0
                    best_lag = 0

                    for lag in range(1, max_lag + 1):
                        p_values = [test_result[lag][0][test][1]
                                    for test in ['ssr_ftest', 'ssr_chi2test', '
                                                                         lrtest
                                                                         ', '
                                                                         params_ftest
                                                                         ']]
                        avg_p = np.mean(p_values)

                        if avg_p < min_p_value:
                            min_p_value = avg_p
                            best_lag = lag

                    if min_p_value < significance_level:
                        granger_matrix[i, j] = 1
                        p_value_matrix[i, j] = min_p_value
                        optimal_lags[i, j] = best_lag

                except Exception as e:
                    if verbose:
                        print(f"  Warning: Test failed for {cause} -> {effect}
                                                            : {e}")
```

```
                    continue

        granger_df = pd.DataFrame(granger_matrix, index=self.tickers, columns=self.tickers)
        p_value_df = pd.DataFrame(p_value_matrix, index=self.tickers, columns=self.tickers)
        lags_df = pd.DataFrame(optimal_lags, index=self.tickers, columns=self.tickers)

        if verbose:
            self._plot_granger_results(granger_df, p_value_df, lags_df, significance_level)

        return {
            'granger_matrix': granger_df,
            'p_values': p_value_df,
            'optimal_lags': lags_df
        }

    def _tune_granger_parameters(self, max_lag_range, alpha_range):
        print("=" * 70)
        print("Hyperparameter Tuning for Granger Causality Test")
        print("=" * 70)

        results = []

        for max_lag in max_lag_range:
            for alpha in alpha_range:
                n_features = len(self.tickers)
                granger_matrix = np.zeros((n_features, n_features))

                for i, cause in enumerate(self.tickers):
                    for j, effect in enumerate(self.tickers):
                        if i == j:
                            continue

                        data = self.log_returns[[effect, cause]].dropna()

                        try:
                            test_result = grangercausalitytests(data, max_lag, verbose=False)

                            min_p_value = 1.0
                            for lag in range(1, max_lag + 1):
                                p_values = [test_result[lag][0][test][1]
                                            for test in ['ssr_ftest', 'ssr_chi2
'
,
'
lrtest
'
```

```
                                avg_p = np.mean(p_values)
                                if avg_p < min_p_value:
                                    min_p_value = avg_p

                            if min_p_value < alpha:
                                granger_matrix[i, j] = 1

                        except:
                            continue

                n_edges = int(granger_matrix.sum())
                density = n_edges / (n_features * (n_features - 1))

                results.append({
                    'max_lag': max_lag,
                    'alpha': alpha,
                    'n_edges': n_edges,
                    'density': density
                })

        results_df = pd.DataFrame(results)
        print("\nParameter Tuning Results:")
        print(results_df.to_string(index=False))

        # Visualization
        pivot_edges = results_df.pivot(index='max_lag', columns='alpha',
                                       values='n_edges')
        pivot_density = results_df.pivot(index='max_lag', columns='alpha',
                                       values='density')

        _, axes = plt.subplots(1, 2, figsize=(16, 6))

        sns.heatmap(pivot_edges, annot=True, fmt='.0f', cmap='YlOrRd',
                   ax=axes[0], cbar_kws={'label': 'Number of Edges'})
        axes[0].set_title('Number of Causal Edges vs Hyperparameters')
        axes[0].set_xlabel('Significance Level  ()')
        axes[0].set_ylabel('Max Lag')

        sns.heatmap(pivot_density, annot=True, fmt='.3f', cmap='viridis',
                   ax=axes[1], cbar_kws={'label': 'Graph Density'})
        axes[1].set_title('Graph Density vs Hyperparameters')
        axes[1].set_xlabel('Significance Level  ()')
        axes[1].set_ylabel('Max Lag')

        plt.tight_layout()
```

```
        plt.savefig(os.path.join(self.figure_dir, '1e_granger_tuning.png'), dpi=150)
        plt.show()

        print("\n" + "=" * 70)

    def _plot_granger_results(self, granger_df, p_value_df, lags_df, significance_level):
        n_edges = int(granger_df.sum().sum())

        print("\n--- Granger Causality Test Results ---")
        print(f"Significance level: {significance_level}")
        print(f"Number of significant Granger causal relationships: {n_edges}")
        print(f"Graph density: {n_edges / (len(self.tickers) * (len(self.tickers) - 1)):.3f}")

        # Plot 1: Granger causality adjacency matrix
        _, axes = plt.subplots(1, 2, figsize=(20, 8))

        sns.heatmap(granger_df, annot=True, fmt='.0f', cmap='Blues',
                    xticklabels=self.tickers, yticklabels=self.tickers,
                    ax=axes[0], cbar_kws={'label': 'Granger Causes (1=Yes, 0=No)'})
        axes[0].set_title(f'Granger Causality Matrix (={significance_level})\nRow causes Column')
        axes[0].set_xlabel('Effect (Y)')
        axes[0].set_ylabel('Cause (X)')

        p_value_masked = p_value_df.copy()
        p_value_masked[granger_df == 0] = np.nan

        sns.heatmap(p_value_masked, annot=True, fmt='.3f', cmap='Reds_r',
                    xticklabels=self.tickers, yticklabels=self.tickers,
                    ax=axes[1], cbar_kws={'label': 'P-value'}, vmin=0, vmax=significance_level)
        axes[1].set_title(f'P-values for Significant Relationships')
        axes[1].set_xlabel('Effect (Y)')
        axes[1].set_ylabel('Cause (X)')

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1e_granger_matrix.png'), dpi=150)
        plt.show()

        # Plot 2: Network graph
        G = nx.DiGraph()
        for ticker in self.tickers:
            G.add_node(ticker)

        edge_list = []
        for i, cause in enumerate(self.tickers):
            for j, effect in enumerate(self.tickers):
                if granger_df.iloc[i, j] == 1:
```

```
                    edge_list.append((cause, effect))
                    G.add_edge(cause, effect,
                              weight=1 - p_value_df.iloc[i, j],
                              lag=int(lags_df.iloc[i, j]))

        plt.figure(figsize=(14, 14))
        pos = nx.spring_layout(G, k=2, iterations=100, seed=42)

        nx.draw_networkx_nodes(G, pos, node_size=2500, node_color='lightblue',
                              alpha=0.9, edgecolors='navy', linewidths=2)
        nx.draw_networkx_labels(G, pos, font_size=11, font_weight='bold')

        if edge_list:
            nx.draw_networkx_edges(G, pos, edgelist=edge_list,
                                  edge_color='darkblue', arrows=True, arrowsize=25,
                                  arrowstyle='-|>', connectionstyle='arc3,rad=0.1',
                                  width=2, alpha=0.7, node_size=2500)

        plt.title(f"Granger Causality Network  (={significance_level})\n"
                 f"{n_edges} significant causal relationships",
                 fontsize=14, pad=20)
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '1e_granger_network.png'),
                   dpi=150, bbox_inches='tight')
        plt.show()

        # Summary statistics
        print("\nTop 10 strongest Granger causal relationships:")
        relationships = []
        for i, cause in enumerate(self.tickers):
            for j, effect in enumerate(self.tickers):
                if granger_df.iloc[i, j] == 1:
                    relationships.append({
                        'Cause': cause,
                        'Effect': effect,
                        'P-value': p_value_df.iloc[i, j],
                        'Lag': int(lags_df.iloc[i, j])
                    })

        if relationships:
            relationships_df = pd.DataFrame(relationships).sort_values('P-value')
            print(relationships_df.head(10).to_string(index=False))
```

A.2 Problem 2a: KDE

A.2.1 Main Script.

```python
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-24 19:58:17
LastEditTime: 2025-11-25 09:37:20
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2_main.py
Description:
    Main script to run data processing and clustering model fitting for
        Homework 3.
    Optimized flow:
    1. process_data() -> X, y
    2. fit_kde_models() -> trained models
    3. generate_samples() -> generated data
    4. evaluate_samples() -> metrics
    5. visualize_generated_samples() + compare_distributions()
'''
from Prob2a_utils import Prob2Analysis


def main():
    # Initialize analysis
    p2 = Prob2Analysis(
        output_dir='Homework 3/Code/Data',
        figure_dir='Homework 3/Latex/Figures'
    )
    X, y = p2.process_data(verbose=False)

    kde_results = p2.fit_kde_models(
        spaces=['pca_20', 'pca_50', 'original'],
        bandwidth_range=None,  # Auto-generate
        kernel='gaussian',
        cv_folds=5,
        verbose=False
    )

    # Generate from PCA-20 space (fastest and often best quality)
    generated_pca20 = p2.generate_samples(
        space='pca_20',
        n_samples=100,
        verbose=True
    )

    metrics = p2.evaluate_samples(verbose=True)

    # Visualization and comparison
    print("\nVisualizing generated samples...")
    p2.visualize_generated_samples(n_display=25)

    print("\nComparing original vs generated distributions...")
    p2.compare_distributions()


if __name__ == "__main__":
    main()
```

A.2.2 Utils Script.

```python
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-24 19:58:10
LastEditTime: 2025-11-25 09:43:24
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2a_utils.py
Description:
    Utility functions and classes for Problem 2 of Homework 3.
    Kernel Density Estimation (KDE) for digit generation.
    - process_data: Load and preprocess sklearn digits dataset
    - fit_kde_models: Fit KDE in multiple spaces with hyperparameter tuning
    - generate_samples: Generate new samples from trained KDE
    - evaluate_samples: Evaluate quality of generated samples
'''
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KernelDensity
from sklearn.model_selection import KFold
from sklearn.metrics import silhouette_score
from scipy.spatial.distance import cdist


class Prob2Analysis:
    def __init__(self,
                 output_dir='Homework 3/Code/Data',
                 figure_dir='Homework 3/Latex/Figures'):
        self.output_dir = output_dir
        self.figure_dir = figure_dir

        os.makedirs(self.output_dir, exist_ok=True)
        os.makedirs(self.figure_dir, exist_ok=True)

        self.data = None
        self.targets = None
        self.kde_results = None
        self.generated_samples = None

    def process_data(self, verbose=False):
        digits = load_digits()
        X = digits.data
        y = digits.target

        self.data = X
        self.targets = y

        if verbose:
            print(f"\nDataset shape: {X.shape}")
```

```
            print(f"Number of classes: {len(np.unique(y))}")
            print(f"Pixel value range: [{X.min():.1f}, {X.max():.1f}]")
            print(f"Mean pixel value: {X.mean():.2f} ± {X.std():.2f}")

        return X, y

    def fit_kde_models(self,
                       spaces=['pca_20', 'pca_50', 'original'],
                       bandwidth_range=None,
                       kernel='gaussian',
                       cv_folds=5,
                       verbose=False):
        if self.data is None:
            self.process_data()

        if bandwidth_range is None:
            bandwidth_range = np.logspace(-1.5, 0.5, 20)

        results = {}

        if verbose:
            print("Hyperparameter Tuning: Bandwidth Selection")

        for space in spaces:
            # Prepare data in target space
            if space == 'original':
                X_space = StandardScaler().fit_transform(self.data)
                n_features = self.data.shape[1]

            else:
                n_comp = int(space.split('_')[1])
                pca = PCA(n_components=n_comp)
                X_space = pca.fit_transform(self.data)
                X_space = StandardScaler().fit_transform(X_space)
                n_features = n_comp

            # Hyperparameter tuning
            kde = KernelDensity(kernel=kernel, algorithm='ball_tree')
            best_bandwidth = None
            best_score = -np.inf
            cv_scores_dict = {'bandwidth': [], 'mean_score': [], 'std_score': []}

            kf = KFold(n_splits=cv_folds, shuffle=True, random_state=25)

            for bw in bandwidth_range:
                fold_scores = []
                for train_idx, test_idx in kf.split(X_space):
                    X_train, X_test = X_space[train_idx], X_space[test_idx]
                    kde = KernelDensity(bandwidth=bw)
                    kde.fit(X_train)
                    score = kde.score(X_test)
                    fold_scores.append(score)

```

```
                mean_score = np.mean(fold_scores)
                std_score = np.std(fold_scores)

                cv_scores_dict['bandwidth'].append(bw)
                cv_scores_dict['mean_score'].append(mean_score)
                cv_scores_dict['std_score'].append(std_score)

                if mean_score > best_score:
                    best_score = mean_score
                    best_bandwidth = bw

            if verbose:
                print(f"  Best bandwidth: {best_bandwidth:.6f}")
                print(f"  Cross-validation score: {best_score:.4f}")

            # Train final model
            kde_final = KernelDensity(
                kernel=kernel,
                bandwidth=best_bandwidth,
                algorithm='ball_tree'
            )
            kde_final.fit(X_space)

            results[space] = {
                'model': kde_final,
                'bandwidth': best_bandwidth,
                'cv_score': best_score,
                'X_train': X_space,
                'cv_results': cv_scores_dict,
                'n_features': n_features,
                'pca': PCA(n_components=int(space.split('_')[1]))
                        if space.startswith('pca_') else None
            }

        self.kde_results = results

        if verbose:
            self._plot_kde_tuning_results(results)

        return results

    def _plot_kde_tuning_results(self, results):
        """Visualize bandwidth tuning for each space."""
        n_spaces = len(results)
        fig, axes = plt.subplots(1, n_spaces, figsize=(5*n_spaces, 4))

        if n_spaces == 1:
            axes = [axes]

        for idx, (space, result) in enumerate(results.items()):
            cv_results = result['cv_results']
            bandwidths = cv_results['bandwidth']
            mean_scores = cv_results['mean_score']
            std_scores = cv_results['std_score']
```

```python

            axes[idx].semilogx(bandwidths, mean_scores, 'o-', linewidth=2, markersize=6)
            axes[idx].fill_between(bandwidths,
                                   np.array(mean_scores) - np.array(std_scores),
                                   np.array(mean_scores) + np.array(std_scores),
                                   alpha=0.2)
            axes[idx].axvline(result['bandwidth'], linestyle='--', color='r',
                             label=f"Best: {result['bandwidth']:.4f}")
            axes[idx].set_xlabel('Bandwidth')
            axes[idx].set_ylabel('Mean Log-Likelihood (Higher is Better)')
            axes[idx].set_title(f'Bandwidth Tuning: {space}')
            axes[idx].legend()
            axes[idx].grid(True, alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2a_kde_tuning.png'), dpi=150)
        plt.show()

    def generate_samples(self, space='pca_20', n_samples=100, verbose=False):
        result = self.kde_results[space]
        model = result['model']

        # Generate samples in latent space
        X_generated = model.sample(n_samples, random_state=42)

        # Transform back to original space
        if space == 'original':
            # Need to inverse standardization
            # For original space, we need the scaler that was used
            scaler = StandardScaler()
            scaler.fit(self.data)
            X_original = scaler.inverse_transform(X_generated)

        else:
            # Inverse PCA transformation
            pca = result['pca']
            pca.fit(self.data)
            X_pca_space = X_generated

            # Inverse standardization in PCA space
            scaler = StandardScaler()
            scaler.fit(pca.transform(self.data))
            X_pca_unscaled = scaler.inverse_transform(X_pca_space)

            # Inverse PCA
            X_original = pca.inverse_transform(X_pca_unscaled)

        # Clip to valid range [0, 16]
        X_original = np.clip(X_original, 0, 16)

```

```
        self.generated_samples = {
            'samples': X_original,
            'space': space,
            'n_samples': n_samples
        }

        if verbose:
            print(f"\nGenerated {n_samples} samples from space: {space}")
            print(f"Generated samples shape: {X_original.shape}")
            print(f"Generated samples range: [{X_original.min():.2f}, {X_original.max():.2f}]")

        return X_original

    def evaluate_samples(self, generated_samples=None, verbose=False):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        metrics = {}

        # 1. Sample statistics comparison
        metrics['original_mean'] = self.data.mean()
        metrics['original_std'] = self.data.std()
        metrics['generated_mean'] = generated_samples.mean()
        metrics['generated_std'] = generated_samples.std()

        # 2. Sparsity check (% of zero pixels)
        metrics['original_sparsity'] = (self.data == 0).mean()
        metrics['generated_sparsity'] = (generated_samples == 0).mean()

        # 3. Quality check: Can we classify generated samples?
        try:
            from sklearn.neighbors import KNeighborsClassifier
            from sklearn.model_selection import cross_val_score

            knn = KNeighborsClassifier(n_neighbors=5)
            scores = cross_val_score(
                knn, self.data, self.targets, cv=5, scoring='accuracy'
            )
            metrics['knn_accuracy_original'] = scores.mean()

        except Exception as e:
            metrics['knn_accuracy_original'] = None

        if verbose:
            self._print_evaluation_metrics(metrics)

        return metrics

    def _print_evaluation_metrics(self, metrics):
        print(f"Mean (Original):     {metrics['original_mean']:.4f}")
        print(f"Mean (Generated):    {metrics['generated_mean']:.4f}")
        print(f"Std  (Original):     {metrics['original_std']:.4f}")
        print(f"Std  (Generated):    {metrics['generated_std']:.4f}")
```

```
        print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
        print(f"Sparsity (Generated):{metrics['generated_sparsity']:.4f}")
        if metrics['knn_accuracy_original'] is not None:
            print(f"KNN Accuracy (Original): {metrics['knn_accuracy_original']:.4f}")

    def visualize_generated_samples(self, generated_samples=None, n_display=25):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']
            space = self.generated_samples['space']
        else:
            space = 'provided'

        grid_size = int(np.sqrt(n_display))
        fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
        axes = axes.flatten()

        for i in range(min(n_display, len(generated_samples))):
            axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
            axes[i].axis('off')

        plt.suptitle(f'Generated Digits from KDE ({space})', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, f'2a_generated_samples_{space}.png'), dpi=150)
        plt.show()

    def compare_distributions(self, generated_samples=None):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        fig, axes = plt.subplots(1, 2, figsize=(14, 5))

        # Pixel intensity distribution
        axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original',
                     color='blue', density=True)
        axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='Generated',
                     color='red', density=True)
        axes[0].set_xlabel('Pixel Intensity')
        axes[0].set_ylabel('Density')
        axes[0].set_title('Pixel Intensity Distribution')
        axes[0].legend()
        axes[0].grid(alpha=0.3)

        # Sparsity comparison
        sparsity_orig = (self.data == 0).sum(axis=1).mean()
        sparsity_gen = (generated_samples == 0).sum(axis=1).mean()

        categories = ['Original', 'Generated']
        sparsities = [sparsity_orig, sparsity_gen]
```

```
        axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7,
                                                edgecolor='black')
        axes[1].set_ylabel('Average # of Zero Pixels')
        axes[1].set_title('Sparsity Comparison')
        axes[1].grid(axis='y', alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2a_distribution_comparison.
                                                png'), dpi=150)
        plt.show()
```

A.3 Problem 2b: GAN

A.3.1 Main Script.

```
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-25 10:49:06
LastEditTime: 2025-11-25 13:28:54
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2b_main.py
Description:
    Main script to run GAN model fitting for Homework 3 Problem 2b.
    Optimized flow:
    1. process_data() -> X, y
    2. tune_hyperparameters() (optional) -> best config
    3. fit_gan() -> trained GAN
    4. generate_samples() -> generated data
    5. evaluate_samples() -> metrics
    6. visualize_generated_samples() + compare_distributions()
'''
from Prob2b_utils import Prob2bAnalysis


def main():
    # Initialize analysis
    p2b = Prob2bAnalysis(
        output_dir='Homework 3/Code/Data',
        figure_dir='Homework 3/Latex/Figures'
    )

    # Load and preprocess data
    X, y = p2b.process_data(verbose=False)

    # Hyperparameter tuning
    tuning_results = p2b.tune_hyperparameters(
        latent_dims=[32, 64, 128],
        hidden_configs=[
            ([128, 256], [256, 128]),
            ([256, 512], [512, 256]),
        ],
        n_epochs=200,
        verbose=False
```

```
    )
    # Best parameters from tuning: latent_dim=32, lr_g=0.0001, G=[128, 256]

    gan_results = p2b.fit_gan(
        latent_dim=32,
        g_hidden_dims=[128, 256],
        d_hidden_dims=[256, 128],
        lr_g=0.0001,
        lr_d=0.0002,
        batch_size=64,
        n_epochs=300,
        n_critic=1,
        beta1=0.5,
        label_smoothing=0.1,
        verbose=False
    )

    # Generate samples
    generated_samples = p2b.generate_samples(
        n_samples=100,
        verbose=False
    )

    # Evaluate generated samples
    metrics = p2b.evaluate_samples(verbose=True)

    # Visualization
    print("\nVisualizing generated samples...")
    p2b.visualize_generated_samples(n_display=25)

    print("\nComparing original vs generated distributions...")
    p2b.compare_distributions()

    print("\nVisualizing latent space interpolation...")
    p2b.visualize_latent_interpolation(n_steps=10)

    print("\nComparing real vs generated samples side-by-side...")
    p2b.compare_with_real_samples(n_display=10)


if __name__ == "__main__":
    main()
```

A.3.2 Utils Script.

```
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-25 10:49:19
LastEditTime: 2025-11-25 12:44:23
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2b_utils.py
Description:
    Utility functions and classes for Problem 2b of Homework 3.
```

```
    Generative Adversarial Network (GAN) for digit generation.
    - process_data: Load and preprocess sklearn digits dataset
    - build_generator: Build generator network
    - build_discriminator: Build discriminator network
    - fit_gan: Train GAN with hyperparameter tuning
    - generate_samples: Generate new samples from trained GAN
    - evaluate_samples: Evaluate quality of generated samples
'''
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_digits
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

class Generator(nn.Module):
    def __init__(self, latent_dim=64, hidden_dims=[128, 256], output_dim=64,
                 activation='relu', use_batchnorm=True):
        super(Generator, self).__init__()

        self.latent_dim = latent_dim

        # Build layers
        layers = []
        in_dim = latent_dim

        for h_dim in hidden_dims:
            layers.append(nn.Linear(in_dim, h_dim))
            if use_batchnorm:
                layers.append(nn.BatchNorm1d(h_dim))
            if activation == 'relu':
                layers.append(nn.ReLU())
            elif activation == 'leaky_relu':
                layers.append(nn.LeakyReLU(0.2))
            elif activation == 'tanh':
                layers.append(nn.Tanh())
            in_dim = h_dim

        # Output layer with Sigmoid for [0, 1] range
        layers.append(nn.Linear(in_dim, output_dim))
        layers.append(nn.Sigmoid())

        self.model = nn.Sequential(*layers)

    def forward(self, z):
        return self.model(z)
```

```python


class Discriminator(nn.Module):
    def __init__(self, input_dim=64, hidden_dims=[256, 128],
                 activation='leaky_relu', dropout_rate=0.3):
        super(Discriminator, self).__init__()

        layers = []
        in_dim = input_dim

        for h_dim in hidden_dims:
            layers.append(nn.Linear(in_dim, h_dim))
            if activation == 'leaky_relu':
                layers.append(nn.LeakyReLU(0.2))
            elif activation == 'relu':
                layers.append(nn.ReLU())
            if dropout_rate > 0:
                layers.append(nn.Dropout(dropout_rate))
            in_dim = h_dim

        # Output layer with Sigmoid for probability
        layers.append(nn.Linear(in_dim, 1))
        layers.append(nn.Sigmoid())

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)


class Prob2bAnalysis:
    def __init__(self,
                 output_dir='Homework 3/Code/Data',
                 figure_dir='Homework 3/Latex/Figures',
                 device=None,
                 seed=25):
        self.output_dir = output_dir
        self.figure_dir = figure_dir

        os.makedirs(self.output_dir, exist_ok=True)
        os.makedirs(self.figure_dir, exist_ok=True)

        # Set device
        if device is None:
            self.device = torch.device('cuda' if torch.cuda.is_available()
                                       else 'cpu')
            print(f"Using device: {self.device}")
        else:
            self.device = device

        self.data = None
        self.targets = None
        self.scaler = None
        self.gan_results = None
```

```
        self.generated_samples = None
        self.training_history = None

        self.seed = seed

    def _set_seed(self):
        np.random.seed(self.seed)
        torch.manual_seed(self.seed)
        if torch.cuda.is_available():
            torch.cuda.manual_seed_all(self.seed)

    def process_data(self, verbose=False):
        # Different form 2a process_data, so cannot share code
        digits = load_digits()
        X = digits.data
        y = digits.target

        # Scale to [0, 1] for GAN training
        self.scaler = MinMaxScaler(feature_range=(0, 1))
        X_scaled = self.scaler.fit_transform(X)

        self.data = X
        self.data_scaled = X_scaled
        self.targets = y

        if verbose:
            print(f"\nDataset shape: {X.shape}")
            print(f"Number of classes: {len(np.unique(y))}")
            print(f"Original pixel range: [{X.min():.1f}, {X.max():.1f}]")
            print(f"Scaled pixel range: [{X_scaled.min():.3f}, {X_scaled.max()
                                                    :.3f}]")
            print(f"Device: {self.device}")

        return X, y

    def fit_gan(self,
                latent_dim=64,
                g_hidden_dims=[128, 256],
                d_hidden_dims=[256, 128],
                lr_g=0.0002,
                lr_d=0.0002,
                batch_size=64,
                n_epochs=300,
                n_critic=1,
                beta1=0.5,
                beta2=0.999,
                label_smoothing=0.1,
                verbose=False):
        self._set_seed()
        if self.data is None:
            self.process_data()

        output_dim = self.data.shape[1]  # 64 for 8x8 images

```

```
        # Build networks
        generator = Generator(
            latent_dim=latent_dim,
            hidden_dims=g_hidden_dims,
            output_dim=output_dim,
            activation='relu',
            use_batchnorm=True
        ).to(self.device)

        discriminator = Discriminator(
            input_dim=output_dim,
            hidden_dims=d_hidden_dims,
            activation='leaky_relu',
            dropout_rate=0.3
        ).to(self.device)

        # Optimizers
        optimizer_g = optim.Adam(generator.parameters(), lr=lr_g, betas=(beta1, beta2))
        optimizer_d = optim.Adam(discriminator.parameters(), lr=lr_d, betas=(beta1, beta2))

        # Loss function
        criterion = nn.BCELoss()

        # Prepare data
        X_tensor = torch.FloatTensor(self.data_scaled).to(self.device)
        dataset = TensorDataset(X_tensor)
        dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, drop_last=True)

        # Training history
        history = {
            'epoch': [],
            'd_loss': [],
            'g_loss': [],
            'd_real_acc': [],
            'd_fake_acc': []
        }

        if verbose:
            print(f"\nTraining GAN...")
            print(f"  Latent dim: {latent_dim}")
            print(f"  Generator: {g_hidden_dims}")
            print(f"  Discriminator: {d_hidden_dims}")
            print(f"  Epochs: {n_epochs}, Batch size: {batch_size}")
            print("-" * 60)

        # Training loop
        for epoch in range(n_epochs):
            d_losses = []
            g_losses = []
            d_real_accs = []
            d_fake_accs = []
```

```python

            for batch_data in dataloader:
                real_data = batch_data[0]
                current_batch_size = real_data.size(0)

                # Labels with smoothing
                real_labels = torch.ones(current_batch_size, 1).to(self.device) * (1 - label_smoothing)
                fake_labels = torch.zeros(current_batch_size, 1).to(self.device)

                # ---------------------
                # Train Discriminator
                # ---------------------
                for _ in range(n_critic):
                    optimizer_d.zero_grad()

                    # Real data
                    d_real_output = discriminator(real_data)
                    d_real_loss = criterion(d_real_output, real_labels)

                    # Fake data
                    z = torch.randn(current_batch_size, latent_dim).to(self.device)
                    fake_data = generator(z)
                    d_fake_output = discriminator(fake_data.detach())
                    d_fake_loss = criterion(d_fake_output, fake_labels)

                    # Total discriminator loss
                    d_loss = d_real_loss + d_fake_loss
                    d_loss.backward()
                    optimizer_d.step()

                d_losses.append(d_loss.item())
                d_real_accs.append((d_real_output > 0.5).float().mean().item())
                d_fake_accs.append((d_fake_output < 0.5).float().mean().item())

                # ---------------------
                # Train Generator
                # ---------------------
                optimizer_g.zero_grad()

                z = torch.randn(current_batch_size, latent_dim).to(self.device)
                fake_data = generator(z)
                g_output = discriminator(fake_data)

                # Generator wants discriminator to think fake is real
                g_loss = criterion(g_output, torch.ones(current_batch_size, 1).to(self.device))
                g_loss.backward()
```

```
                optimizer_g.step()

                g_losses.append(g_loss.item())

            # Record epoch metrics
            history['epoch'].append(epoch + 1)
            history['d_loss'].append(np.mean(d_losses))
            history['g_loss'].append(np.mean(g_losses))
            history['d_real_acc'].append(np.mean(d_real_accs))
            history['d_fake_acc'].append(np.mean(d_fake_accs))

            # Print progress
            if verbose and (epoch + 1) % 50 == 0:
                print(f"Epoch [{epoch+1:4d}/{n_epochs}] | "
                      f"D Loss: {history['d_loss'][-1]:.4f} | "
                      f"G Loss: {history['g_loss'][-1]:.4f} | "
                      f"D(real): {history['d_real_acc'][-1]:.3f} | "
                      f"D(fake): {history['d_fake_acc'][-1]:.3f}")

        # Store results
        self.gan_results = {
            'generator': generator,
            'discriminator': discriminator,
            'latent_dim': latent_dim,
            'history': history,
            'config': {
                'latent_dim': latent_dim,
                'g_hidden_dims': g_hidden_dims,
                'd_hidden_dims': d_hidden_dims,
                'lr_g': lr_g,
                'lr_d': lr_d,
                'batch_size': batch_size,
                'n_epochs': n_epochs
            }
        }

        self.training_history = history

        if verbose:
            print("Training completed!")
            self._plot_training_history(history)

        return self.gan_results

    def tune_hyperparameters(self,
                             latent_dims=[32, 64, 128],
                             lr_g_values=[0.00005, 0.0001, 0.0002],
                             lr_d=0.0002,
                             hidden_configs=[
                                 ([128, 256], [256, 128]),
                                 ([256, 512], [512, 256]),
                                 ([64, 128, 256], [256, 128, 64])
                             ],
                             n_epochs=200,
```

```
                             n_eval_samples=500,
                             verbose=False):
        if self.data is None:
            self.process_data()

        results = []

        for latent_dim in latent_dims:
            for lr_g in lr_g_values:
                for g_hidden, d_hidden in hidden_configs:
                    # Train model
                    self.fit_gan(
                        latent_dim=latent_dim,
                        g_hidden_dims=g_hidden,
                        d_hidden_dims=d_hidden,
                        lr_g=lr_g,
                        lr_d=lr_d,
                        n_epochs=n_epochs,
                        verbose=False
                    )

                    # Generate and evaluate samples
                    samples = self.generate_samples(n_samples=n_eval_samples,
                                                    verbose=False)
                    metrics = self.evaluate_samples(samples, verbose=False)

                    # Store results
                    result = {
                        'latent_dim': latent_dim,
                        'lr_g': lr_g,
                        'g_hidden': str(g_hidden),
                        'd_hidden': str(d_hidden),
                        'mean_diff': abs(metrics['generated_mean'] - metrics['original_mean']),
                        'std_diff': abs(metrics['generated_std'] - metrics['original_std']),
                        'diversity_ratio': metrics.get('diversity_ratio', 0)
                    }
                    results.append(result)

                    if verbose:
                        print(f"  Mean Diff: {result['mean_diff']:.4f} | "
                              f"Std Diff: {result['std_diff']:.4f} | "
                              f"Diversity: {result['diversity_ratio']:.3f}")

        results_df = pd.DataFrame(results)

        if verbose:
            print("Tuning Results Summary:")
            print(results_df.to_string(index=False))
            self._plot_tuning_results(results_df)

```

```python
        return results_df

    def _plot_tuning_results(self, results_df):
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # Mean diff by lr_g, grouped by latent_dim
        for latent in results_df['latent_dim'].unique():
            subset = results_df[results_df['latent_dim'] == latent]
            avg_by_lr = subset.groupby('lr_g')['mean_diff'].mean()
            axes[0].plot(avg_by_lr.index, avg_by_lr.values, 'o-', label=f'
                                                    latent={latent}',
                                                    markersize=8)
        axes[0].set_title('Mean Diff vs lr_g (Lower is Better)')
        axes[0].set_xscale('log')
        axes[0].legend()
        axes[0].grid(True, alpha=0.3)

        # Mean difference
        pivot = results_df.pivot_table(
            values='mean_diff',
            index='latent_dim',
            columns='lr_g',
            aggfunc='mean'
        )
        sns.heatmap(pivot, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[1])
        axes[1].set_title('Mean Difference (Lower is Better)')

        # Std difference
        pivot_std = results_df.pivot_table(
            values='std_diff',
            index='latent_dim',
            columns='lr_g',
            aggfunc='mean'
        )
        sns.heatmap(pivot_std, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[2
                                                ])
        axes[2].set_title('Std Difference (Lower is Better)')

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_gan_tuning.png'), dpi=
                                                150)
        plt.show()

    def _plot_training_history(self, history):
        fig, axes = plt.subplots(1, 2, figsize=(14, 5))

        epochs = history['epoch']

        # Losses
        axes[0].plot(epochs, history['d_loss'], label='Discriminator Loss',
                                                alpha=0.8)
        axes[0].plot(epochs, history['g_loss'], label='Generator Loss', alpha=
                                                0.8)
        axes[0].set_xlabel('Epoch')
```

```
        axes[0].set_ylabel('Loss')
        axes[0].set_title('GAN Training Losses')
        axes[0].legend()
        axes[0].grid(True, alpha=0.3)

        # Discriminator accuracy
        axes[1].plot(epochs, history['d_real_acc'], label='D(real) accuracy', alpha=0.8)
        axes[1].plot(epochs, history['d_fake_acc'], label='D(fake) accuracy', alpha=0.8)
        axes[1].axhline(y=0.5, color='r', linestyle='--', alpha=0.5, label='Random guess')
        axes[1].set_xlabel('Epoch')
        axes[1].set_ylabel('Accuracy')
        axes[1].set_title('Discriminator Performance')
        axes[1].legend()
        axes[1].grid(True, alpha=0.3)
        axes[1].set_ylim([0, 1])

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_gan_training.png'), dpi=150)
        plt.show()

    def generate_samples(self, n_samples=100, verbose=False):
        generator = self.gan_results['generator']
        latent_dim = self.gan_results['latent_dim']

        generator.eval()
        with torch.no_grad():
            z = torch.randn(n_samples, latent_dim).to(self.device)
            generated_scaled = generator(z).cpu().numpy()

        # Inverse transform to original scale
        generated = self.scaler.inverse_transform(generated_scaled)

        # Clip to valid range [0, 16]
        generated = np.clip(generated, 0, 16)

        self.generated_samples = {
            'samples': generated,
            'samples_scaled': generated_scaled,
            'n_samples': n_samples
        }

        if verbose:
            print(f"\nGenerated {n_samples} samples from GAN")
            print(f"Generated samples shape: {generated.shape}")
            print(f"Generated samples range: [{generated.min():.2f}, {generated.max():.2f}]")

        return generated

    def evaluate_samples(self, generated_samples=None, verbose=False):
```

```
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        metrics = {}

        # 1. Sample statistics comparison
        metrics['original_mean'] = self.data.mean()
        metrics['original_std'] = self.data.std()
        metrics['generated_mean'] = generated_samples.mean()
        metrics['generated_std'] = generated_samples.std()

        # 2. Sparsity check (% of near-zero pixels)
        threshold = 0.5
        metrics['original_sparsity'] = (self.data < threshold).mean()
        metrics['generated_sparsity'] = (generated_samples < threshold).mean()

        # 3. Coverage: fraction of original data modes covered
        # Using PCA + clustering approximation
        try:
            pca = PCA(n_components=10)
            orig_pca = pca.fit_transform(self.data)
            gen_pca = pca.transform(generated_samples)

            # Simple coverage: check if generated samples are near original samples
            from scipy.spatial.distance import cdist
            distances = cdist(gen_pca, orig_pca, metric='euclidean')
            min_distances = distances.min(axis=1)
            coverage_threshold = np.percentile(
                cdist(orig_pca, orig_pca).flatten(), 50
            )
            metrics['coverage'] = (min_distances < coverage_threshold).mean()
        except Exception as e:
            metrics['coverage'] = None

        # 4. Mode collapse check: diversity of generated samples
        try:
            gen_pca = PCA(n_components=10).fit_transform(generated_samples)
            pairwise_dist = cdist(gen_pca, gen_pca, metric='euclidean')
            np.fill_diagonal(pairwise_dist, np.inf)
            metrics['avg_nn_distance'] = pairwise_dist.min(axis=1).mean()

            orig_pca = PCA(n_components=10).fit_transform(self.data)
            orig_pairwise = cdist(orig_pca, orig_pca, metric='euclidean')
            np.fill_diagonal(orig_pairwise, np.inf)
            metrics['orig_avg_nn_distance'] = orig_pairwise.min(axis=1).mean()

            # Diversity ratio (higher is better, 1.0 means same diversity as original)
            metrics['diversity_ratio'] = metrics['avg_nn_distance'] / metrics['orig_avg_nn_distance']
        except Exception as e:
            metrics['diversity_ratio'] = None

```

```python
        if verbose:
            self._print_evaluation_metrics(metrics)

        return metrics

    def _print_evaluation_metrics(self, metrics):
        """Print evaluation metrics."""
        print("\n--- GAN Evaluation Metrics ---")
        print(f"Mean (Original):     {metrics['original_mean']:.4f}")
        print(f"Mean (Generated):    {metrics['generated_mean']:.4f}")
        print(f"Std  (Original):     {metrics['original_std']:.4f}")
        print(f"Std  (Generated):    {metrics['generated_std']:.4f}")
        print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
        print(f"Sparsity (Generated):{metrics['generated_sparsity']:.4f}")
        if metrics.get('coverage') is not None:
            print(f"Coverage:            {metrics['coverage']:.4f}")
        if metrics.get('diversity_ratio') is not None:
            print(f"Diversity Ratio:     {metrics['diversity_ratio']:.4f}")

    def visualize_generated_samples(self, generated_samples=None, n_display=25):
        """Visualize grid of generated samples."""
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        grid_size = int(np.sqrt(n_display))
        fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
        axes = axes.flatten()

        for i in range(min(n_display, len(generated_samples))):
            axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
            axes[i].axis('off')

        plt.suptitle('Generated Digits from GAN', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_generated_samples_gan.png'), dpi=150)
        plt.show()

    def compare_distributions(self, generated_samples=None):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        fig, axes = plt.subplots(1, 3, figsize=(18, 5))

        # Pixel intensity distribution
        axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original',
                     color='blue', density=True)
        axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='Generated',
                     color='red', density=True)
        axes[0].set_xlabel('Pixel Intensity')
        axes[0].set_ylabel('Density')
```

```
        axes[0].set_title('Pixel Intensity Distribution')
        axes[0].legend()
        axes[0].grid(alpha=0.3)

        # Sparsity comparison
        sparsity_orig = (self.data < 0.5).sum(axis=1).mean()
        sparsity_gen = (generated_samples < 0.5).sum(axis=1).mean()

        categories = ['Original', 'Generated']
        sparsities = [sparsity_orig, sparsity_gen]
        axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7, edgecolor='black')
        axes[1].set_ylabel('Average # of Near-Zero Pixels')
        axes[1].set_title('Sparsity Comparison')
        axes[1].grid(axis='y', alpha=0.3)

        # PCA projection comparison
        pca = PCA(n_components=2)
        orig_pca = pca.fit_transform(self.data)
        gen_pca = pca.transform(generated_samples)

        axes[2].scatter(orig_pca[:, 0], orig_pca[:, 1], alpha=0.3, label='Original', s=10)
        axes[2].scatter(gen_pca[:, 0], gen_pca[:, 1], alpha=0.5, label='Generated', s=10)
        axes[2].set_xlabel('PC1')
        axes[2].set_ylabel('PC2')
        axes[2].set_title('PCA Projection Comparison')
        axes[2].legend()
        axes[2].grid(alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_distribution_comparison_gan.png'), dpi=150)
        plt.show()

    def visualize_latent_interpolation(self, n_steps=10):
        generator = self.gan_results['generator']
        latent_dim = self.gan_results['latent_dim']

        generator.eval()

        # Generate two random latent vectors
        z1 = torch.randn(1, latent_dim).to(self.device)
        z2 = torch.randn(1, latent_dim).to(self.device)

        # Interpolate
        interpolations = []
        for alpha in np.linspace(0, 1, n_steps):
            z = (1 - alpha) * z1 + alpha * z2
            with torch.no_grad():
                img = generator(z).cpu().numpy()
            img = self.scaler.inverse_transform(img)
```

```
            img = np.clip(img, 0, 16)
            interpolations.append(img.reshape(8, 8))

        # Plot
        fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2))
        for i, img in enumerate(interpolations):
            axes[i].imshow(img, cmap='gray')
            axes[i].axis('off')
            if i == 0:
                axes[i].set_title('Start')
            elif i == n_steps - 1:
                axes[i].set_title('End')

        plt.suptitle('Latent Space Interpolation', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_latent_interpolation.png
                                                  '), dpi=150)
        plt.show()

    def compare_with_real_samples(self, n_display=10):
        generated = self.generated_samples['samples'][:n_display]
        real_indices = np.random.choice(len(self.data), n_display, replace=
                                                  False)
        real = self.data[real_indices]

        fig, axes = plt.subplots(2, n_display, figsize=(2 * n_display, 4))

        for i in range(n_display):
            axes[0, i].imshow(real[i].reshape(8, 8), cmap='gray')
            axes[0, i].axis('off')
            if i == 0:
                axes[0, i].set_ylabel('Real', fontsize=12)

            axes[1, i].imshow(generated[i].reshape(8, 8), cmap='gray')
            axes[1, i].axis('off')
            if i == 0:
                axes[1, i].set_ylabel('Generated', fontsize=12)

        plt.suptitle('Real vs Generated Samples', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2b_real_vs_generated.png'),
                                                  dpi=150)
        plt.show()
```

A.4 Problem 2c: Diffusion Model.

A.4.1 Main Script.

```
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-25 15:08:40
LastEditTime: 2025-11-25 20:54:04
```

```python
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2c_main.py
Description:
    Main script to run Diffusion Model fitting for Homework 3 Problem 2c.
    Optimized flow:
    1. process_data() -> X, y
    2. tune_hyperparameters() (optional) -> best config
    3. fit_diffusion() -> trained model
    4. generate_samples() -> generated data
    5. evaluate_samples() -> metrics
    6. visualize_generated_samples() + compare_distributions()
'''
from Prob2c_utils import Prob2cAnalysis


def main():
    # Initialize analysis (with seed for reproducibility)
    p2c = Prob2cAnalysis(
        output_dir='Homework 3/Code/Data',
        figure_dir='Homework 3/Latex/Figures',
        seed=25
    )

    # Load and preprocess data
    X, y = p2c.process_data(verbose=False)

    # Hyperparameter tuning
    # tuning_results = p2c.tune_hyperparameters(
    #     num_timesteps_values=[200, 500, 1000],
    #     lr_values=[1e-4, 5e-4, 1e-3],
    #     hidden_configs=[
    #         [128, 256, 128],
    #         [256, 512, 256],
    #     ],
    #     n_epochs=150,
    #     verbose=False
    # ) # Best results: lr=0.0005, timesteps=500, architecture=[256, 512, 256]

    # Train Diffusion Model with selected hyperparameters
    diffusion_results = p2c.fit_diffusion(
        num_timesteps=500,
        hidden_dims=[256, 512, 256],
        time_emb_dim=64,
        lr=5e-4,
        batch_size=128,
        n_epochs=200,
        beta_start=1e-4,
        beta_end=0.02,
        verbose=False
    )

    # Generate samples
    generated_samples = p2c.generate_samples(
        n_samples=100,
        verbose=True
```

```
    )

    # Evaluate generated samples
    metrics = p2c.evaluate_samples(verbose=True)

    # Visualization
    print("\nVisualizing generated samples...")
    p2c.visualize_generated_samples(n_display=25)

    print("\nComparing original vs generated distributions...")
    p2c.compare_distributions()

    print("\nVisualizing reverse diffusion process...")
    p2c.visualize_diffusion_process(n_steps=10)

    print("\nComparing real vs generated samples side-by-side...")
    p2c.compare_with_real_samples(n_display=10)


if __name__ == "__main__":
    main()
```

A.4.2 Utils Script.

```
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-11-25 15:08:49
LastEditTime: 2025-11-25 20:56:24
FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2c_utils.py
Description:
    Utility functions and classes for Problem 2c of Homework 3.
    Denoising Diffusion Probabilistic Model (DDPM) for digit generation.
    - process_data: Load and preprocess sklearn digits dataset
    - build_denoiser: Build denoising network
    - fit_diffusion: Train diffusion model with hyperparameter tuning
    - generate_samples: Generate new samples via reverse diffusion
    - evaluate_samples: Evaluate quality of generated samples
'''
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_digits
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from scipy.spatial.distance import cdist
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from tqdm import tqdm
```

```python


class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = np.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -
                                                embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1) #
                                                 Use both sin and cos to
                                                capture more information
        return embeddings


class DenoisingMLP(nn.Module):
    def __init__(self, input_dim=64, hidden_dims=[256, 512, 256], time_emb_dim
                                            =64):
        super().__init__()

        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim * 2),
            nn.GELU(), # Instead of ReLU, GELU(Gaussian Error Linear Unit) as
                                                    a smooth version often
                                                    performs better
            nn.Linear(time_emb_dim * 2, time_emb_dim)
        )

        # Input layer
        self.input_layer = nn.Linear(input_dim, hidden_dims[0])

        # Time embedding projection for each layer
        self.time_projections = nn.ModuleList([
            nn.Linear(time_emb_dim, h_dim) for h_dim in hidden_dims
        ])

        # Hidden layers
        self.hidden_layers = nn.ModuleList()
        for i in range(len(hidden_dims) - 1):
            self.hidden_layers.append(
                nn.Sequential(
                    nn.Linear(hidden_dims[i], hidden_dims[i + 1]),
                    nn.GroupNorm(8, hidden_dims[i + 1]),
                    nn.GELU()
                )
            )

        # Output layer
```

```python
        self.output_layer = nn.Linear(hidden_dims[-1], input_dim)

        self.act = nn.GELU()
        self.norm_layers = nn.ModuleList([
            nn.GroupNorm(8, h_dim) for h_dim in hidden_dims
        ])

    def forward(self, x, t):
        # Time embedding
        t_emb = self.time_mlp(t)

        # Input
        h = self.input_layer(x)
        h = self.norm_layers[0](h)
        h = self.act(h)
        h = h + self.time_projections[0](t_emb)

        # Hidden layers
        for i, layer in enumerate(self.hidden_layers):
            h = layer(h)
            h = h + self.time_projections[i + 1](t_emb)

        # Output
        return self.output_layer(h)


class DiffusionScheduler:
    def __init__(self, num_timesteps=1000, beta_start=1e-4, beta_end=0.02, device='cpu'):
        self.num_timesteps = num_timesteps
        self.device = device

        # Linear schedule
        self.betas = torch.linspace(beta_start, beta_end, num_timesteps, device=device)
        self.alphas = 1.0 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.alphas_cumprod_prev = torch.cat([
            torch.tensor([1.0], device=device),
            self.alphas_cumprod[:-1]
        ])

        # Pre-compute values for q(x_t | x_0)
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - self.alphas_cumprod)

        # Pre-compute values for posterior q(x_{t-1} | x_t, x_0)
        self.posterior_variance = (
            self.betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.alphas_cumprod)
        )
        self.sqrt_recip_alphas = torch.sqrt(1.0 / self.alphas)

```

```
    def q_sample(self, x_0, t, noise=None):
        """Forward diffusion: q(x_t | x_0)"""
        if noise is None:
            noise = torch.randn_like(x_0)

        sqrt_alphas_cumprod_t = self.sqrt_alphas_cumprod[t][:, None]
        sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t][:, None]

        return sqrt_alphas_cumprod_t * x_0 + sqrt_one_minus_alphas_cumprod_t * noise

    def p_sample(self, model, x_t, t):
        """Reverse diffusion: p(x_{t-1} | x_t)"""
        betas_t = self.betas[t][:, None]
        sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t][:, None]
        sqrt_recip_alphas_t = self.sqrt_recip_alphas[t][:, None]

        # Predict noise
        predicted_noise = model(x_t, t.float())

        # Compute mean
        model_mean = sqrt_recip_alphas_t * (
            x_t - betas_t * predicted_noise / sqrt_one_minus_alphas_cumprod_t
        )

        # Add noise (except for t=0)
        if (t > 0).any():
            posterior_variance_t = self.posterior_variance[t][:, None]
            noise = torch.randn_like(x_t)
            # Only add noise where t > 0
            mask = (t > 0).float()[:, None]
            return model_mean + mask * torch.sqrt(posterior_variance_t) * noise
        else:
            return model_mean


class Prob2cAnalysis:
    def __init__(self,
                 output_dir='Homework 3/Code/Data',
                 figure_dir='Homework 3/Latex/Figures',
                 device=None,
                 seed=25):
        self.output_dir = output_dir
        self.figure_dir = figure_dir
        self.seed = seed

        os.makedirs(self.output_dir, exist_ok=True)
        os.makedirs(self.figure_dir, exist_ok=True)

        # Set device
        if device is None:
```

```
            self.device = torch.device('cuda' if torch.cuda.is_available()
                                        else 'cpu')
        else:
            self.device = device

        self.data = None
        self.targets = None
        self.scaler = None
        self.diffusion_results = None
        self.generated_samples = None
        self.training_history = None

    def _set_seed(self):
        np.random.seed(self.seed)
        torch.manual_seed(self.seed)
        if torch.cuda.is_available():
            torch.cuda.manual_seed_all(self.seed)

    def process_data(self, verbose=False):
        digits = load_digits()
        X = digits.data
        y = digits.target

        # Scale to [0, 1] then to [-1, 1] for diffusion
        self.scaler = MinMaxScaler(feature_range=(-1, 1))
        X_scaled = self.scaler.fit_transform(X)

        self.data = X
        self.data_scaled = X_scaled
        self.targets = y

        if verbose:
            print(f"\nDataset shape: {X.shape}")
            print(f"Number of classes: {len(np.unique(y))}")
            print(f"Original pixel range: [{X.min():.1f}, {X.max():.1f}]")
            print(f"Scaled pixel range: [{X_scaled.min():.3f}, {X_scaled.max()
                                        :.3f}]")
            print(f"Device: {self.device}")

        return X, y

    def fit_diffusion(self,
                      num_timesteps=500,
                      hidden_dims=[256, 512, 256],
                      time_emb_dim=64,
                      lr=1e-3,
                      batch_size=128,
                      n_epochs=200,
                      beta_start=1e-4,
                      beta_end=0.02,
                      verbose=False):
        self._set_seed()

        if self.data is None:
```

```
            self.process_data()

        input_dim = self.data.shape[1]  # 64 for 8x8 images

        # Build model
        model = DenoisingMLP(
            input_dim=input_dim,
            hidden_dims=hidden_dims,
            time_emb_dim=time_emb_dim
        ).to(self.device)

        # Build scheduler
        scheduler = DiffusionScheduler(
            num_timesteps=num_timesteps,
            beta_start=beta_start,
            beta_end=beta_end,
            device=self.device
        )

        # Optimizer
        optimizer = optim.Adam(model.parameters(), lr=lr)

        # Loss function
        criterion = nn.MSELoss()

        # Prepare data
        X_tensor = torch.FloatTensor(self.data_scaled).to(self.device)
        dataset = TensorDataset(X_tensor)
        dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
                                                        drop_last=True)

        # Training history
        history = {
            'epoch': [],
            'loss': []
        }

        if verbose:
            print(f"\nTraining Diffusion Model...")
            print(f"  Timesteps: {num_timesteps}")
            print(f"  Hidden dims: {hidden_dims}")
            print(f"  Learning rate: {lr}")
            print(f"  Epochs: {n_epochs}, Batch size: {batch_size}")

        # Training loop
        model.train()
        for epoch in range(n_epochs):
            epoch_losses = []

            for batch_data in dataloader:
                x_0 = batch_data[0]
                current_batch_size = x_0.size(0)

                # Sample random timesteps
```

```
                t = torch.randint(0, num_timesteps, (current_batch_size,),
                                                                    device=self.device)

                # Sample noise
                noise = torch.randn_like(x_0)

                # Forward diffusion
                x_t = scheduler.q_sample(x_0, t, noise)

                # Predict noise
                optimizer.zero_grad()
                predicted_noise = model(x_t, t.float())

                # Loss
                loss = criterion(predicted_noise, noise)
                loss.backward()
                optimizer.step()

                epoch_losses.append(loss.item())

            avg_loss = np.mean(epoch_losses)
            history['epoch'].append(epoch + 1)
            history['loss'].append(avg_loss)

            if verbose and (epoch + 1) % 20 == 0:
                print(f"Epoch [{epoch+1:4d}/{n_epochs}] | Loss: {avg_loss:.6f}
                                                                    ")

        # Store results
        self.diffusion_results = {
            'model': model,
            'scheduler': scheduler,
            'history': history,
            'config': {
                'num_timesteps': num_timesteps,
                'hidden_dims': hidden_dims,
                'time_emb_dim': time_emb_dim,
                'lr': lr,
                'batch_size': batch_size,
                'n_epochs': n_epochs
            }
        }

        self.training_history = history

        if verbose:
            self._plot_training_history(history)

        return self.diffusion_results

    def tune_hyperparameters(self,
                             num_timesteps_values=[200, 500, 1000],
                             lr_values=[1e-4, 5e-4, 1e-3],
                             hidden_configs=[
```

```
                                [128, 256, 128],
                                [256, 512, 256],
                                [256, 512, 512, 256]
                            ],
                            n_epochs=150,
                            n_eval_samples=500,
                            verbose=False):
        if self.data is None:
            self.process_data()

        results = []
        total_configs = len(num_timesteps_values) * len(lr_values) * len(
                                                        hidden_configs)
        current_config = 0

        if verbose:
            print("Hyperparameter Tuning for Diffusion Model")
            print(f"Total configurations: {total_configs}")

        for num_timesteps in num_timesteps_values:
            for lr in lr_values:
                for hidden_dims in hidden_configs:
                    current_config += 1
                    if verbose:
                        print(f"\n[{current_config}/{total_configs}] "
                              f"T={num_timesteps}, lr={lr}, hidden={
                                                                hidden_dims
                                                                }")

                    # Train model
                    self.fit_diffusion(
                        num_timesteps=num_timesteps,
                        hidden_dims=hidden_dims,
                        lr=lr,
                        n_epochs=n_epochs,
                        verbose=False
                    )

                    # Generate and evaluate samples
                    samples = self.generate_samples(n_samples=n_eval_samples,
                                                        verbose=False)
                    metrics = self.evaluate_samples(samples, verbose=False)

                    # Store results
                    result = {
                        'num_timesteps': num_timesteps,
                        'lr': lr,
                        'hidden_dims': str(hidden_dims),
                        'final_loss': self.training_history['loss'][-1],
                        'mean_diff': abs(metrics['generated_mean'] - metrics['
                                                            original_mean']
                                                            ),
                        'std_diff': abs(metrics['generated_std'] - metrics['
                                                            original_std'])
```

```
                                                 ,
                        'diversity_ratio': metrics.get('diversity_ratio', 0)
                    }
                    results.append(result)

                    if verbose:
                        print(f"  Loss: {result['final_loss']:.6f} | "
                              f"Mean Diff: {result['mean_diff']:.4f} | "
                              f"Std Diff: {result['std_diff']:.4f}")

        results_df = pd.DataFrame(results)

        if verbose:
            print("Tuning Results Summary:")
            print(results_df.to_string(index=False))

            # Find best configuration
            best_idx = results_df['mean_diff'].idxmin()
            best_config = results_df.iloc[best_idx]
            print("Best Configuration (by Mean Diff):")
            print(f"  Timesteps: {best_config['num_timesteps']}")
            print(f"  Learning Rate: {best_config['lr']}")
            print(f"  Hidden Dims: {best_config['hidden_dims']}")
            print(f"  Mean Diff: {best_config['mean_diff']:.4f}")
            print(f"  Std Diff: {best_config['std_diff']:.4f}")

            self._plot_tuning_results(results_df)

        return results_df

    def _plot_tuning_results(self, results_df):
        fig, axes = plt.subplots(1, 3, figsize=(18, 5))

        # Mean difference heatmap (timesteps x lr)
        pivot_mean = results_df.pivot_table(
            values='mean_diff',
            index='num_timesteps',
            columns='lr',
            aggfunc='mean'
        )
        sns.heatmap(pivot_mean, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[0])
        axes[0].set_title('Mean Difference (Lower is Better)')
        axes[0].set_xlabel('Learning Rate')
        axes[0].set_ylabel('Timesteps')

        # Std difference heatmap
        pivot_std = results_df.pivot_table(
            values='std_diff',
            index='num_timesteps',
            columns='lr',
            aggfunc='mean'
        )
```

```
        sns.heatmap(pivot_std, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[1])
        axes[1].set_title('Std Difference (Lower is Better)')
        axes[1].set_xlabel('Learning Rate')
        axes[1].set_ylabel('Timesteps')

        # Final loss by architecture
        arch_loss = results_df.groupby('hidden_dims')['final_loss'].mean()
        colors = plt.cm.Set2(np.linspace(0, 1, len(arch_loss)))
        axes[2].bar(range(len(arch_loss)), arch_loss.values,
                    color=colors, edgecolor='black')
        axes[2].set_xticks(range(len(arch_loss)))
        axes[2].set_xticklabels([s.replace(', ', ',\n') for s in arch_loss.index],
                                fontsize=8, rotation=0)
        axes[2].set_ylabel('Final Loss')
        axes[2].set_title('Final Loss by Architecture (Lower is Better)')
        axes[2].grid(axis='y', alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_tuning.png'), dpi=150)
        plt.show()

    def _plot_training_history(self, history):
        fig, ax = plt.subplots(figsize=(10, 5))

        epochs = history['epoch']
        losses = history['loss']

        ax.plot(epochs, losses, 'b-', linewidth=2)
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Loss (MSE)')
        ax.set_title('Diffusion Model Training Loss')
        ax.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_training.png'), dpi=150)
        plt.show()

    def generate_samples(self, n_samples=100, verbose=False):
        model = self.diffusion_results['model']
        scheduler = self.diffusion_results['scheduler']
        num_timesteps = self.diffusion_results['config']['num_timesteps']
        input_dim = self.data.shape[1]

        model.eval()

        with torch.no_grad():
            # Start from pure noise
            x = torch.randn(n_samples, input_dim, device=self.device)

            # Reverse diffusion
```

```
            for t in reversed(range(num_timesteps)):
                t_batch = torch.full((n_samples,), t, device=self.device,
                                                        dtype=torch.long)
                x = scheduler.p_sample(model, x, t_batch)

            generated_scaled = x.cpu().numpy()

        # Inverse transform to original scale
        generated = self.scaler.inverse_transform(generated_scaled)

        # Clip to valid range [0, 16]
        generated = np.clip(generated, 0, 16)

        self.generated_samples = {
            'samples': generated,
            'samples_scaled': generated_scaled,
            'n_samples': n_samples
        }

        if verbose:
            print(f"\nGenerated {n_samples} samples from Diffusion Model")
            print(f"Generated samples shape: {generated.shape}")
            print(f"Generated samples range: [{generated.min():.2f}, {
                                                generated.max():.2f}]")

        return generated

    def evaluate_samples(self, generated_samples=None, verbose=False):
        """Evaluate quality of generated samples."""
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        metrics = {}

        # Sample statistics comparison
        metrics['original_mean'] = self.data.mean()
        metrics['original_std'] = self.data.std()
        metrics['generated_mean'] = generated_samples.mean()
        metrics['generated_std'] = generated_samples.std()

        # Sparsity check
        threshold = 0.5
        metrics['original_sparsity'] = (self.data < threshold).mean()
        metrics['generated_sparsity'] = (generated_samples < threshold).mean()

        # Diversity check
        try:
            gen_pca = PCA(n_components=10).fit_transform(generated_samples)
            pairwise_dist = cdist(gen_pca, gen_pca, metric='euclidean')
            np.fill_diagonal(pairwise_dist, np.inf)
            metrics['avg_nn_distance'] = pairwise_dist.min(axis=1).mean()

            orig_pca = PCA(n_components=10).fit_transform(self.data)
            orig_pairwise = cdist(orig_pca, orig_pca, metric='euclidean')
```

```python
            np.fill_diagonal(orig_pairwise, np.inf)
            metrics['orig_avg_nn_distance'] = orig_pairwise.min(axis=1).mean()

            metrics['diversity_ratio'] = metrics['avg_nn_distance'] / metrics['orig_avg_nn_distance']
        except Exception as e:
            metrics['diversity_ratio'] = None

        if verbose:
            self._print_evaluation_metrics(metrics)

        return metrics

    def _print_evaluation_metrics(self, metrics):
        """Print evaluation metrics."""
        print("\n--- Diffusion Model Evaluation Metrics ---")
        print(f"Mean (Original):     {metrics['original_mean']:.4f}")
        print(f"Mean (Generated):    {metrics['generated_mean']:.4f}")
        print(f"Std  (Original):     {metrics['original_std']:.4f}")
        print(f"Std  (Generated):    {metrics['generated_std']:.4f}")
        print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
        print(f"Sparsity (Generated):{metrics['generated_sparsity']:.4f}")
        if metrics.get('diversity_ratio') is not None:
            print(f"Diversity Ratio:     {metrics['diversity_ratio']:.4f}")

    def visualize_generated_samples(self, generated_samples=None, n_display=25):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        grid_size = int(np.sqrt(n_display))
        fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
        axes = axes.flatten()

        for i in range(min(n_display, len(generated_samples))):
            axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
            axes[i].axis('off')

        plt.suptitle('Generated Digits from Diffusion Model', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2c_generated_samples_diffusion.png'), dpi=150)
        plt.show()

    def compare_distributions(self, generated_samples=None):
        if generated_samples is None:
            generated_samples = self.generated_samples['samples']

        fig, axes = plt.subplots(1, 3, figsize=(18, 5))

        # Pixel intensity distribution
        axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original',
```

```
                    color='blue', density=True)
        axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='Generated',
                    color='red', density=True)
        axes[0].set_xlabel('Pixel Intensity')
        axes[0].set_ylabel('Density')
        axes[0].set_title('Pixel Intensity Distribution')
        axes[0].legend()
        axes[0].grid(alpha=0.3)

        # Sparsity comparison
        sparsity_orig = (self.data < 0.5).sum(axis=1).mean()
        sparsity_gen = (generated_samples < 0.5).sum(axis=1).mean()

        categories = ['Original', 'Generated']
        sparsities = [sparsity_orig, sparsity_gen]
        axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7, edgecolor='black')
        axes[1].set_ylabel('Average # of Near-Zero Pixels')
        axes[1].set_title('Sparsity Comparison')
        axes[1].grid(axis='y', alpha=0.3)

        # PCA projection comparison
        pca = PCA(n_components=2)
        orig_pca = pca.fit_transform(self.data)
        gen_pca = pca.transform(generated_samples)

        axes[2].scatter(orig_pca[:, 0], orig_pca[:, 1], alpha=0.3, label='Original', s=10)
        axes[2].scatter(gen_pca[:, 0], gen_pca[:, 1], alpha=0.5, label='Generated', s=10)
        axes[2].set_xlabel('PC1')
        axes[2].set_ylabel('PC2')
        axes[2].set_title('PCA Projection Comparison')
        axes[2].legend()
        axes[2].grid(alpha=0.3)

        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2c_distribution_comparison_diffusion.png'), dpi=150)
        plt.show()

    def visualize_diffusion_process(self, n_steps=10):
        if self.diffusion_results is None:
            raise ValueError("Diffusion model not trained. Call fit_diffusion() first.")

        model = self.diffusion_results['model']
        scheduler = self.diffusion_results['scheduler']
        num_timesteps = self.diffusion_results['config']['num_timesteps']
        input_dim = self.data.shape[1]

        model.eval()
```

```python

        # Choose timesteps to visualize
        vis_timesteps = np.linspace(num_timesteps - 1, 0, n_steps, dtype=int)

        with torch.no_grad():
            # Start from pure noise
            x = torch.randn(1, input_dim, device=self.device)

            samples_at_timesteps = []

            # Reverse diffusion
            for t in reversed(range(num_timesteps)):
                t_batch = torch.full((1,), t, device=self.device, dtype=torch.long)
                x = scheduler.p_sample(model, x, t_batch)

                if t in vis_timesteps:
                    sample = x.cpu().numpy()
                    sample = self.scaler.inverse_transform(sample)
                    sample = np.clip(sample, 0, 16)
                    samples_at_timesteps.append((t, sample.reshape(8, 8)))

        # Sort by timestep (descending)
        samples_at_timesteps.sort(key=lambda x: x[0], reverse=True)

        # Plot
        fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2.5))
        for i, (t, img) in enumerate(samples_at_timesteps):
            axes[i].imshow(img, cmap='gray')
            axes[i].axis('off')
            axes[i].set_title(f't={t}', fontsize=10)

        plt.suptitle('Reverse Diffusion Process (Noise → Image)', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_process.png'), dpi=150)
        plt.show()

    def compare_with_real_samples(self, n_display=10):
        if self.generated_samples is None:
            self.generate_samples(n_samples=n_display)

        generated = self.generated_samples['samples'][:n_display]
        real_indices = np.random.choice(len(self.data), n_display, replace=False)
        real = self.data[real_indices]

        fig, axes = plt.subplots(2, n_display, figsize=(2 * n_display, 4))

        for i in range(n_display):
            axes[0, i].imshow(real[i].reshape(8, 8), cmap='gray')
            axes[0, i].axis('off')
            if i == 0:
                axes[0, i].set_ylabel('Real', fontsize=12)
```

```

            axes[1, i].imshow(generated[i].reshape(8, 8), cmap='gray')
            axes[1, i].axis('off')
            if i == 0:
                axes[1, i].set_ylabel('Generated', fontsize=12)

        plt.suptitle('Real vs Generated Samples (Diffusion)', fontsize=14)
        plt.tight_layout()
        plt.savefig(os.path.join(self.figure_dir, '
                                                   2c_real_vs_generated_diffusion.
                                                   png'), dpi=150)
        plt.show()
```