

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.

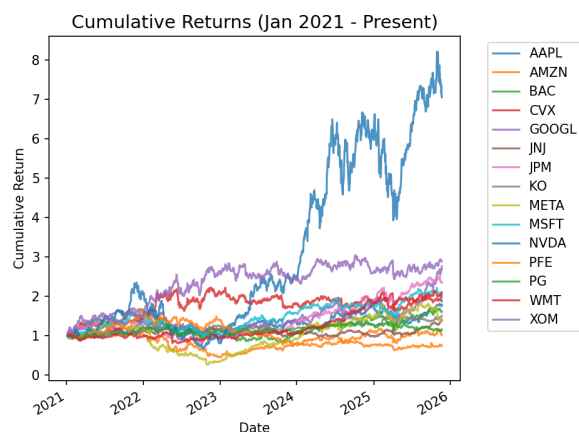


Figure 1: Cumulative Log Returns of Selected Stocks (Jan 2021 - Present)

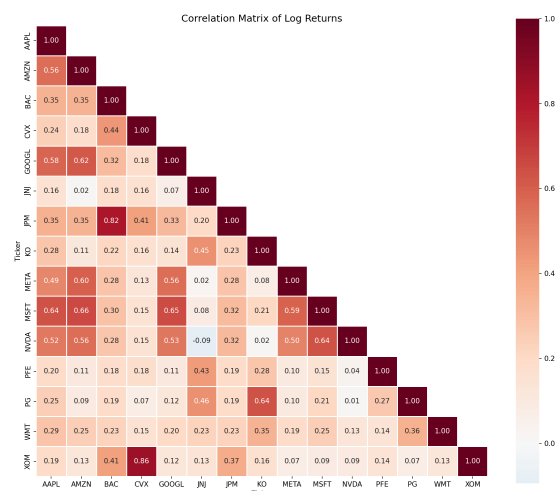


Figure 2: Correlation Heatmap of Daily Log Returns

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

$$\alpha_{\text{Gaussian}} = 0.021287, \quad \alpha_{\text{Nonparamormal}} = 0.013528.$$

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

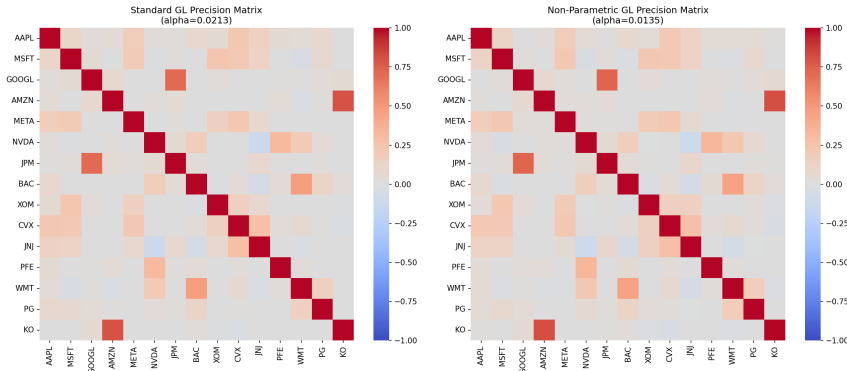


Figure 3: Gaphic Lasso Estimated Precision Matrices: Standard (Left) vs. Non-Parametric (Right)

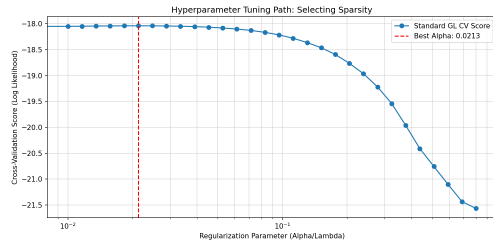


Figure 4: Cross-Validated Log-Likelihood Curve for Standard Graphical Lasso

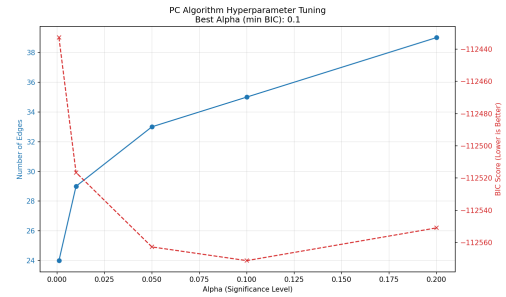


Figure 5: Best Alpha Selection for PC Algorithm via BIC Score

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.

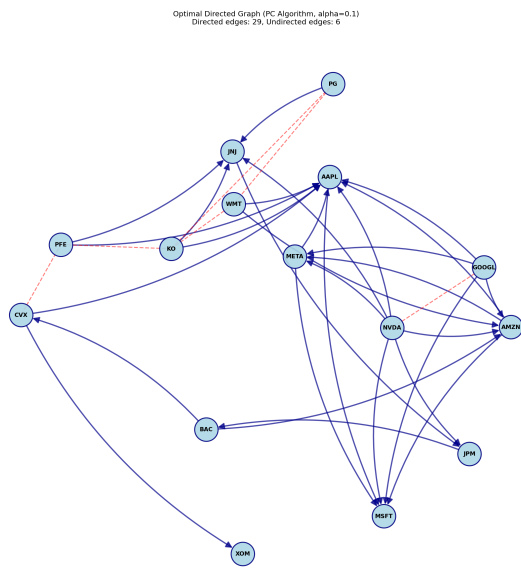


Figure 6: Best PC Algorithm Graph at $\alpha = 0.1$

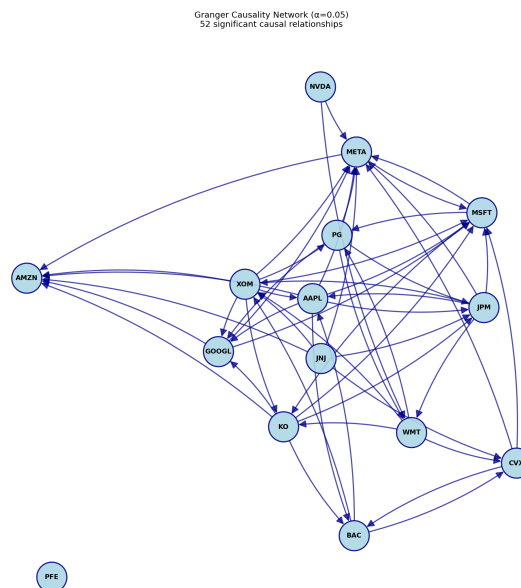


Figure 7: Granger Causal Graphical Model at $\alpha = 0.05$, Max Lag = 5

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.

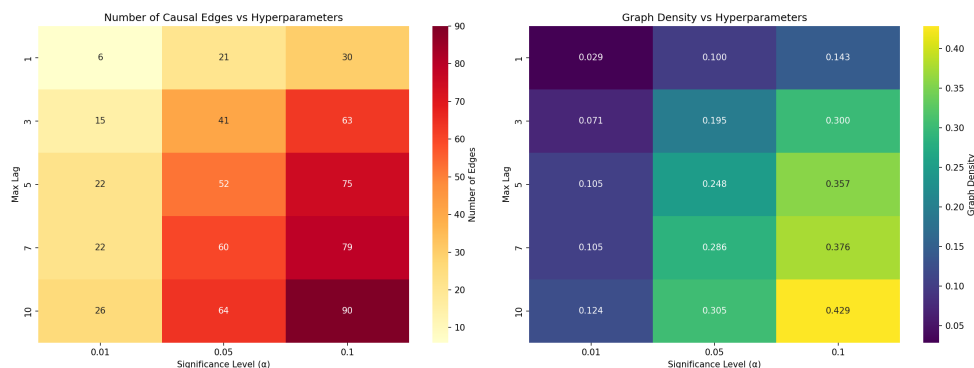


Figure 8: Granger Causality Model Tuning: Edge Counts (Left) and Graph Density (Right) Across Significance Levels and Maximum Lags

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.

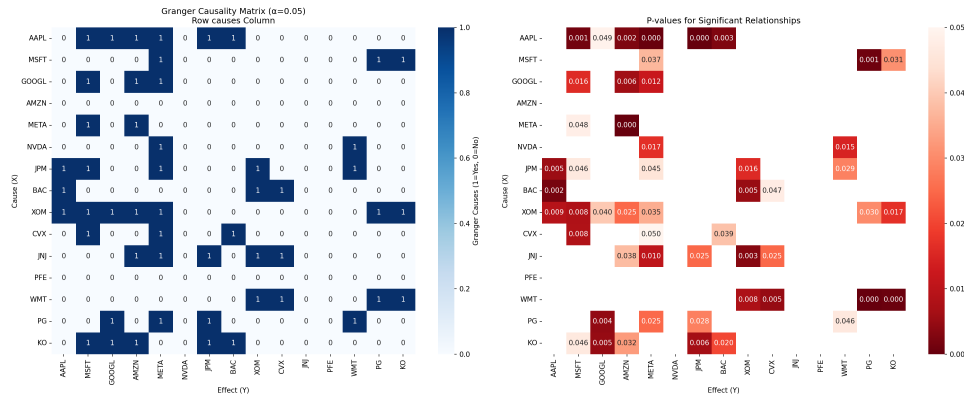


Figure 9: Granger Causality Adjacency Matrix at $\alpha = 0.05$, Max Lag = 5

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 $\text{latent_dim}=32$, $\text{lr_g}=0.0001$, $G=[128, 256]$ 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, $\text{latent}=32$ 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.

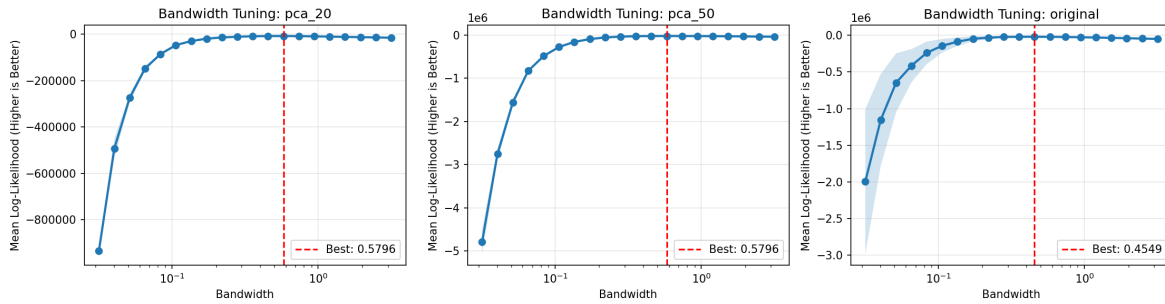


Figure 10: KDE Bandwidth Tuning via 5-Fold Cross-Validation in PCA-20, PCA-50, and Original Spaces



Figure 11: Generated Samples from KDE

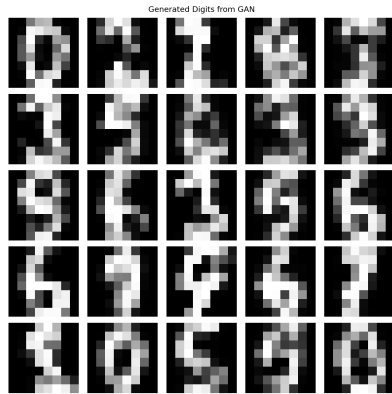


Figure 12: Generated Samples from GAN

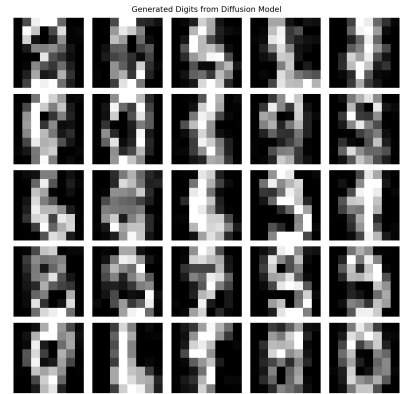


Figure 13: Generated Samples from Diffusion Model

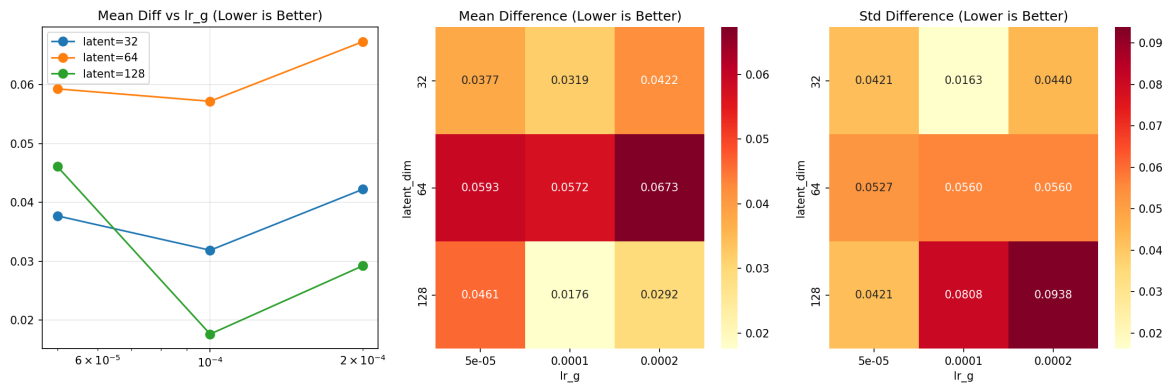


Figure 14: GAN Hyperparameter Tuning Results

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 $lr = 0.0005$ achieves the lowest mean_diff (0.0054), with std_diff also within an acceptable range. Therefore, we select $lr = 0.0005$, timesteps = 500, and architecture = $[256, 512, 256]$ as our optimal model configuration. All subsequent discussions are based on this setting.

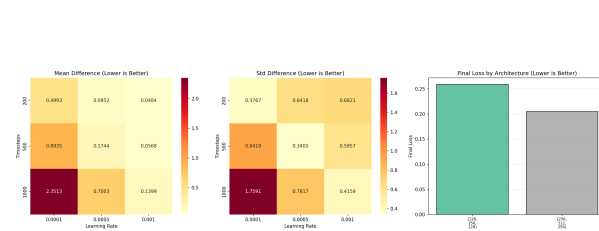


Figure 15: Diffusion Model Hyperparameter Tuning Results

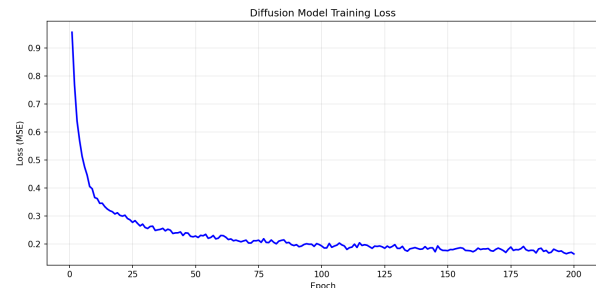


Figure 16: Training Trajectory of the Diffusion Model

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.

```
1 '''
2 Author: Chuyang Su cs4570@columbia.edu
3 Date: 2025-11-23 17:41:45
4 LastEditTime: 2025-11-24 12:27:59
5 FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/main.py
6 Description:
7     Main script to run data processing and graphical model fitting for
8                                     Homework 3.
9 '''
10
11 from Prob1_utils import Prob1Analysis
12
13 def main():
14     # Problem 1a
15     p1=Prob1Analysis()
16     log_returns = p1.process_stock_data(verbose=False)
17
18     # Problem 1b
19     glasso_results = p1.fit_glasso_models(verbose=False)
20
21     # Problem 1c
22     pc_results = p1.fit_pc_model(verbose=False)
23
24     # Problem 1e
25     granger_results = p1.fit_granger_model(verbose=True)
26
27 if __name__ == "__main__":
28     main()
```

A.1.2 Utils Script.

```
1 '''
2 Author: Chuyang Su cs4570@columbia.edu
3 Date: 2025-11-23 17:27:40
4 LastEditTime: 2025-11-26 11:22:20
5 FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/prob1_utils.py
6 Description:
7     Utility functions and classes for Problem 1 analysis in Homework 3.
8 '''
9
10 import os
11 import yfinance as yf
12 import pandas as pd
13 import numpy as np
```

```

13 import matplotlib.pyplot as plt
14 import seaborn as sns
15 from scipy.stats import norm
16 import networkx as nx
17 from sklearn.covariance import GraphicalLassoCV
18 from sklearn.preprocessing import StandardScaler
19 from sklearn.linear_model import LinearRegression, LassoCV
20 from statsmodels.tsa.stattools import grangercausalitytests
21 from causallearn.search.ConstraintBased.PC import pc
22
23 class Prob1Analysis:
24     def __init__(self,
25                 output_dir='Homework 3/Code/Data',
26                 figure_dir='Homework 3/Latex/Figures',
27                 start_date="2021-01-01"):
28         self.output_dir = output_dir
29         self.figure_dir = figure_dir
30         self.start_date = start_date
31
32         os.makedirs(self.output_dir, exist_ok=True)
33         os.makedirs(self.figure_dir, exist_ok=True)
34
35         self.raw_data_file = os.path.join(self.output_dir, 'raw_stock_data.csv')
36         self.log_returns_file = os.path.join(self.output_dir, 'log_returns.csv')
37
38         self.tickers = ["AAPL", "MSFT", "GOOGL", "AMZN", "META", "NVDA",
39                        "JPM", "BAC", "XOM", "CVX", "JNJ", "PFE", "WMT", "PG",
40                        "KO"]
41
42         self.sectors = {
43             'Tech': ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'META', 'NVDA'],
44             'Finance': ['JPM', 'BAC'],
45             'Energy': ['XOM', 'CVX'],
46             'Healthcare': ['JNJ', 'PFE'],
47             'Consumer': ['WMT', 'PG', 'KO']
48         }
49
50         self.log_returns = None
51         self.glasso_results = None
52
53     def process_stock_data(self, verbose=False):
54         # Download stock data and compute log returns
55         if os.path.exists(self.log_returns_file):
56             log_returns = pd.read_csv(self.log_returns_file, index_col=0,
57                                     parse_dates=True)
58
59         else:
60             print(f"Downloading data from yfinance...")
61             raw_data = yf.download(self.tickers, start=self.start_date, end=
62                                 None)
63             raw_data.to_csv(self.raw_data_file)

```

```

62     # Extract closing prices and compute log returns
63     close_prices = raw_data['Close']
64     print(f"\nData shape: {close_prices.shape}")
65     print(f"Date range: {close_prices.index[0]} to {close_prices.index[-1]}")
66
67     log_returns = np.log(close_prices / close_prices.shift(1)).dropna
68                     ()
69     log_returns.to_csv(self.log_returns_file)
70     # print(f"\nMissing values per stock:\n{log_returns.isnull().sum()}")
71
72     self.log_returns = log_returns
73
74     # Visual exploration
75     if verbose:
76         self._plot_visual_exploration()
77
78     return log_returns
79
80 def _plot_visual_exploration(self):
81     # Summary statistics
82     summary_stats = self.log_returns.describe()
83     summary_stats.to_csv(os.path.join(self.output_dir, 'summary_statistics.csv'))
84
85     # Plot correlation heatmap
86     plt.figure(figsize=(12, 10))
87     corr_matrix = self.log_returns.corr()
88     mask = np.triu(np.ones_like(corr_matrix, dtype=bool), k=1)
89     sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='RdBu_r', center=
90                 0,
91                 mask=mask, square=True, linewidths=0.5)
92     plt.title('Correlation Matrix of Log Returns', fontsize=14)
93     plt.tight_layout()
94     plt.savefig(os.path.join(self.figure_dir, '1a_correlation_heatmap.png'), dpi=150)
95
96     plt.show()
97
98     # Plot cumulative returns
99     plt.figure(figsize=(14, 8))
100    cumulative_returns = (1 + self.log_returns).cumprod()
101    cumulative_returns.plot(alpha=0.8)
102    plt.title('Cumulative Returns (Jan 2021 - Present)', fontsize=14)
103    plt.xlabel('Date')
104    plt.ylabel('Cumulative Return')
105    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
106    plt.tight_layout()
107    plt.savefig(os.path.join(self.figure_dir, '1a_cumulative_returns.png'), dpi=150)
108
109    plt.show()
110
111    # Relationships within different industries
112    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

```

```

109     axes = axes.flatten()
110
111     for idx, (sector, stocks) in enumerate(self.sectors.items()):
112         sector_corr = self.log_returns[stocks].corr()
113         sns.heatmap(sector_corr, annot=True, fmt='.2f', cmap='RdBu_r',
114                     center=0,
115                     ax=axes[idx], square=True, vmin=-1, vmax=1)
116         axes[idx].set_title(f'{sector} Sector Correlation')
117
118     axes[-1].axis('off')
119     plt.suptitle('Within-Sector Correlation Analysis', fontsize=14)
120     plt.tight_layout()
121     plt.savefig(os.path.join(self.figure_dir, '1a_sector_correlations.png'),
122                 dpi=150)
123
124     plt.show()
125
126 def fit_glasso_models(self, verbose=False):
127     if self.log_returns is None:
128         self.process_stock_data()
129
130     scaler = StandardScaler()
131     X_standardized = scaler.fit_transform(self.log_returns.values)
132
133     # Grid search best alpha values
134     alphas_grid = np.logspace(np.log10(0.01), np.log10(0.8), 30)
135
136     # Standard Graphical Lasso
137     if verbose:
138         print("Fitting Standard Graphical LassoCV...")
139     gl_cv = GraphicalLassoCV(alphas=alphas_grid, cv=5, n_jobs=-1, max_iter=
140                             3000, tol=1e-4)
141
142     gl_cv.fit(X_standardized)
143     # print(gl_cv.cv_results_)
144
145     # Non-parametric (Rank-based) Graphical Lasso
146     if verbose:
147         print("Fitting Non-parametric (Rank) Graphical LassoCV...")
148
149     # Transform data to ranks
150     n = self.log_returns.shape[0]
151     X_ranks = self.log_returns.rank() / (n + 1)
152     X_normal_scores = norm.ppf(X_ranks)
153     X_np_standardized = StandardScaler().fit_transform(X_normal_scores)
154
155     gl_np = GraphicalLassoCV(alphas=alphas_grid, cv=5, n_jobs=-1, max_iter=
156                             3000, tol=1e-4)
157
158     gl_np.fit(X_np_standardized)
159
160     # Compute partial correlations
161     prec = gl_cv.precision_
162     d = np.sqrt(np.diag(prec))
163     partial_corr = -prec / np.outer(d, d)
164     np.fill_diagonal(partial_corr, 1.0)

```

```

159     prec_np = gl_np.precision_
160     d_np = np.sqrt(np.diag(prec_np))
161     partial_corr_np = -prec_np / np.outer(d_np, d_np)
162     np.fill_diagonal(partial_corr_np, 1.0)
163
164     self.glasso_results = {
165         'gl_cv': gl_cv,
166         'gl_np': gl_np,
167         'precision_gl': prec,
168         'precision_np': prec_np,
169         'partial_corr_gl': partial_corr,
170         'partial_corr_np': partial_corr_np
171     }
172
173     # Verbose output and visualization
174     if verbose:
175         self._plot_glasso_results(gl_cv, gl_np, partial_corr, partial_corr_np)
176
177
178     return self.glasso_results
179
180 def _plot_glasso_results(self, gl_cv, gl_np, partial_corr, partial_corr_np):
181     print("\n--- Graphical Lasso Results ---")
182     print(f"Standard GL Best Alpha: {gl_cv.alpha_:.6f}")
183     print(f"Non-Parametric GL Best Alpha: {gl_np.alpha_:.6f}")
184
185     # Plot CV curves
186     plt.figure(figsize=(10, 5))
187     cv_results = gl_cv.cv_results_
188     cv_means = cv_results['mean_test_score']
189     cv_alphas = cv_results['alphas']
190
191     plt.semilogx(cv_alphas, cv_means, 'o-', label='Standard GL CV Score')
192     plt.axvline(gl_cv.alpha_, linestyle='--', color='r', label=f'Best
193                                     Alpha: {gl_cv.alpha_:.4f}')
194
195     plt.xlabel('Regularization Parameter (Alpha/Lambda)')
196     plt.ylabel('Cross-Validation Score (Log Likelihood)')
197     plt.title('Hyperparameter Tuning Path: Selecting Sparsity')
198     plt.legend()
199     plt.grid(True, which="both", ls="--", alpha=0.5)
200     plt.tight_layout()
201     plt.savefig(os.path.join(self.figure_dir, '1b_glasso_cv_curve.png'),
202                 dpi=150)
203
204     plt.show()
205
206     # Plot precision matrices
207     fig, axes = plt.subplots(1, 2, figsize=(16, 7))
208
209     # Standard
210     prec = gl_cv.precision_
211     d = np.sqrt(np.diag(prec))

```

```

209     partial_corr = -prec / np.outer(d, d)
210     np.fill_diagonal(partial_corr, 1.0)
211
212     sns.heatmap(partial_corr, ax=axes[0], cmap='coolwarm',
213                 xticklabels=self.tickers, yticklabels=self.tickers, vmin=-
214                                     1, vmax=1)
215     axes[0].set_title(f'Standard GL Precision Matrix\n(alpha={gl_cv.alpha_
216                                     :.4f})')
217
218     # Non-parametric
219     prec_np = gl_np.precision_
220     d_np = np.sqrt(np.diag(prec_np))
221     partial_corr_np = -prec_np / np.outer(d_np, d_np)
222     np.fill_diagonal(partial_corr_np, 1.0)
223
224     sns.heatmap(partial_corr_np, ax=axes[1], cmap='coolwarm',
225                 xticklabels=self.tickers, yticklabels=self.tickers, vmin=-
226                                     1, vmax=1)
227     axes[1].set_title(f'Non-Parametric GL Precision Matrix\n(alpha={gl_np.
228                                     alpha_:.4f})')
229
230     plt.tight_layout()
231     plt.savefig(os.path.join(self.figure_dir, '
232                                     1b_glasso_precision_matrices.
233                                     png'), dpi=150)
234
235     plt.show()
236
237 def calculate_bic_for_dag(self, data, G):
238     n_samples, n_features = data.shape
239     bic_total = 0
240
241     nodes = G.get_nodes()
242
243     for i, node in enumerate(nodes):
244         neighbors = G.get_adjacent_nodes(node)
245         neighbor_indices = [nodes.index(n) for n in neighbors]
246
247         y = data.iloc[:, i].values
248
249         if len(neighbor_indices) == 0:
250             residuals = y - np.mean(y)
251         else:
252             X = data.iloc[:, neighbor_indices].values
253             model = LinearRegression()
254             model.fit(X, y)
255             residuals = y - model.predict(X)
256
257         variance = max(1e-9, np.var(residuals))
258
259         # Log-likelihood term for this node
260         log_likelihood = -0.5 * n_samples * (np.log(2 * np.pi * variance)
261                                     + 1)

```

```

255         # Number of parameters = number of parents + 1 (intercept/variance
256             )
257         k = len(neighbor_indices) + 1
258         # BIC = k * ln(n) - 2 * ln(L)
259         bic_node = k * np.log(n_samples) - 2 * log_likelihood
260         bic_total += bic_node
261
262     return bic_total
263
264 def fit_pc_model(self, alphas=[0.001, 0.01, 0.05, 0.1, 0.2], verbose=False
265                     ):
266     if self.log_returns is None:
267         self.process_stock_data()
268
269     data_np = self.log_returns.values
270     labels = self.log_returns.columns.tolist()
271
272     results = []
273     best_bic = float('inf')
274     best_graph = None
275     best_alpha = 0.05
276
277     if verbose:
278         print(f"Tuning PC Algorithm over alphas: {alphas}...")
279
280     for alpha in alphas:
281         cg = pc(data_np, alpha, "fisherz")
282         n_edges = cg.G.get_num_edges()
283         bic = self.calculate_bic_for_dag(self.log_returns, cg.G)
284
285         results.append({
286             'alpha': alpha,
287             'n_edges': n_edges,
288             'bic': bic,
289             'graph': cg.G
290         })
291
292     if verbose:
293         print(f"Alpha: {alpha:<6} | Edges: {n_edges:<4} | BIC: {bic
294             :.2f}")
295
296     if bic < best_bic:
297         best_bic = bic
298         best_graph = cg.G
299         best_alpha = alpha
300
301     if verbose:
302         self._plot_pc_results(results, best_alpha, best_graph, labels)
303
304     return {
305         'best_graph': best_graph,
306         'best_alpha': best_alpha,
307         'results': results
308     }

```

```

306     }
307
308     def _plot_pc_results(self, results, best_alpha, best_graph, labels):
309         alphas_plot = [r['alpha'] for r in results]
310         edges_plot = [r['n_edges'] for r in results]
311         bics_plot = [r['bic'] for r in results]
312
313         fig, ax1 = plt.subplots(figsize=(10, 6))
314
315         color = 'tab:blue'
316         ax1.set_xlabel('Alpha (Significance Level)')
317         ax1.set_ylabel('Number of Edges', color=color)
318         ax1.plot(alphas_plot, edges_plot, marker='o', color=color, label='
319                                     Sparsity (Edges)')
320         ax1.tick_params(axis='y', labelcolor=color)
321         ax1.grid(True, alpha=0.3)
322
323         ax2 = ax1.twinx()
324         color = 'tab:red'
325         ax2.set_ylabel('BIC Score (Lower is Better)', color=color)
326         ax2.plot(alphas_plot, bics_plot, marker='x', linestyle='--', color=
327                                     color, label='BIC Score')
328         ax2.tick_params(axis='y', labelcolor=color)
329
330         plt.title(f'PC Algorithm Hyperparameter Tuning\nBest Alpha (min BIC):
331                                     {best_alpha}')
332
333         fig.tight_layout()
334         plt.savefig(os.path.join(self.figure_dir, '1c_pc_tuning.png'), dpi=150
335                                     )
336
337         plt.show()
338
339         print(f"\nBest Alpha selected by BIC: {best_alpha}")
340         print(f"Number of edges in best graph: {best_graph.get_num_edges()}")
341
342         G_nx = nx.DiGraph()
343         nodes = best_graph.get_nodes()
344
345         for i in range(len(nodes)):
346             G_nx.add_node(i, label=labels[i])
347
348         graph_edges = best_graph.get_graph_edges()
349         directed_edges = []
350         undirected_edges = []
351
352         for edge in graph_edges:
353             node1 = edge.get_node1()
354             node2 = edge.get_node2()
355             idx1 = nodes.index(node1)
356             idx2 = nodes.index(node2)
357
358             endpoint1 = edge.get_endpoint1()
359             endpoint2 = edge.get_endpoint2()

```



```

355         ep1_name = endpoint1.name if hasattr(endpoint1, 'name') else str(
356             endpoint1)
357         ep2_name = endpoint2.name if hasattr(endpoint2, 'name') else str(
358             endpoint2)
359         if ep1_name == 'TAIL' and ep2_name == 'ARROW':
360             directed_edges.append((idx1, idx2))
361             G_nx.add_edge(idx1, idx2)
362         elif ep1_name == 'ARROW' and ep2_name == 'TAIL':
363             directed_edges.append((idx2, idx1))
364             G_nx.add_edge(idx2, idx1)
365         else:
366             undirected_edges.append((idx1, idx2))
367             G_nx.add_edge(idx1, idx2)
368             G_nx.add_edge(idx2, idx1)
369
370     plt.figure(figsize=(16, 16))
371     pos = nx.spring_layout(G_nx, k=2.5, iterations=100, seed=42)
372
373     nx.draw_networkx_nodes(G_nx, pos, node_size=2000, node_color='
374         lightblue',
375         alpha=0.9, edgecolors='navy', linewidths=2)
376     nx.draw_networkx_labels(G_nx, pos, {i: labels[i] for i in range(len(
377         labels))},
378         font_size=11, font_weight='bold')
379
380     if directed_edges:
381         nx.draw_networkx_edges(G_nx, pos, edgelist=directed_edges,
382             edge_color='darkblue', arrows=True,
383             arrowstyle='->', connectionstyle='arc3,rad=
384                 0.15',
385             width=2.5, alpha=0.7, node_size=2000)
386
387     if undirected_edges:
388         nx.draw_networkx_edges(G_nx, pos, edgelist=undirected_edges,
389             edge_color='red', arrows=False,
390             width=2.0, alpha=0.5, style='dashed')
391
392     plt.title(f"Optimal Directed Graph (PC Algorithm, alpha={best_alpha})\n
393         n"
394         f"Directed edges: {len(directed_edges)}, Undirected edges: {
395             len(undirected_edges)}
396         ",
397             fontsize=14, pad=20)
398
399     plt.axis('off')
400     plt.tight_layout()
401     plt.savefig(os.path.join(self.figure_dir, '1c_pc_graph.png'), dpi=150,
402         bbox_inches='tight')
403
404     plt.show()
405
406 def fit_granger_model(self, max_lag=5, significance_level=0.05,
407     tune_params=True, verbose=False):

```

```

396         if self.log_returns is None:
397             self.process_stock_data()
398
399         # Hyperparameter tuning
400         if tune_params and verbose:
401             self._tune_granger_parameters(max_lag_range=[1, 3, 5, 7, 10],
402                                           alpha_range=[0.01, 0.05, 0.1])
403
404         n_features = len(self.tickers)
405         granger_matrix = np.zeros((n_features, n_features))
406         p_value_matrix = np.zeros((n_features, n_features))
407         optimal_lags = np.zeros((n_features, n_features), dtype=int)
408
409         if verbose:
410             print(f"\nRunning Granger Causality Tests (max_lag={max_lag},
411                                                           alpha={significance_level})
412                                                           ...")
413
414         for i, cause in enumerate(self.tickers):
415             for j, effect in enumerate(self.tickers):
416                 if i == j:
417                     continue
418
419                 data = self.log_returns[[effect, cause]].dropna()
420
421                 try:
422                     test_result = grangercausalitytests(data, max_lag, verbose
423                                                         =False)
424
425                     min_p_value = 1.0
426                     best_lag = 0
427
428                     for lag in range(1, max_lag + 1):
429                         p_values = [test_result[lag][0][test][1]
430                                    for test in ['ssr_ftest', 'ssr_chi2test', '
431                                                  lrtest
432                                                  ', '
433                                                  params_ftest
434                                                  ']]
435
436                     avg_p = np.mean(p_values)
437
438                     if avg_p < min_p_value:
439                         min_p_value = avg_p
440                         best_lag = lag
441
442                     if min_p_value < significance_level:
443                         granger_matrix[i, j] = 1
444                         p_value_matrix[i, j] = min_p_value
445                         optimal_lags[i, j] = best_lag
446
447         except Exception as e:
448             if verbose:
449                 print(f"Warning: Test failed for {cause} -> {effect}
450                       : {e}")

```

```

442         continue
443
444     granger_df = pd.DataFrame(granger_matrix, index=self.tickers, columns=
                                self.tickers)
445     p_value_df = pd.DataFrame(p_value_matrix, index=self.tickers, columns=
                                self.tickers)
446     lags_df = pd.DataFrame(optimal_lags, index=self.tickers, columns=self.
                                tickers)
447
448     if verbose:
449         self._plot_granger_results(granger_df, p_value_df, lags_df,
                                    significance_level)
450
451     return {
452         'granger_matrix': granger_df,
453         'p_values': p_value_df,
454         'optimal_lags': lags_df
455     }
456
457     def _tune_granger_parameters(self, max_lag_range, alpha_range):
458         print("=" * 70)
459         print("Hyperparameter Tuning for Granger Causality Test")
460         print("=" * 70)
461
462         results = []
463
464         for max_lag in max_lag_range:
465             for alpha in alpha_range:
466                 n_features = len(self.tickers)
467                 granger_matrix = np.zeros((n_features, n_features))
468
469                 for i, cause in enumerate(self.tickers):
470                     for j, effect in enumerate(self.tickers):
471                         if i == j:
472                             continue
473
474                         data = self.log_returns[[effect, cause]].dropna()
475
476                         try:
477                             test_result = grangercausalitytests(data, max_lag,
                                                                    verbose=
                                                                    False)
478
479                             min_p_value = 1.0
480                             for lag in range(1, max_lag + 1):
481                                 p_values = [test_result[lag][0][test][1]
482                                             for test in ['ssr_ftest', '

```

```

ssr_chi2
',
',
',
lrtest
'

```

```

483         avg_p = np.mean(p_values)
484         if avg_p < min_p_value:
485             min_p_value = avg_p
486
487         if min_p_value < alpha:
488             granger_matrix[i, j] = 1
489
490     except:
491         continue
492
493     n_edges = int(granger_matrix.sum())
494     density = n_edges / (n_features * (n_features - 1))
495
496     results.append({
497         'max_lag': max_lag,
498         'alpha': alpha,
499         'n_edges': n_edges,
500         'density': density
501     })
502
503     results_df = pd.DataFrame(results)
504     print("\nParameter Tuning Results:")
505     print(results_df.to_string(index=False))
506
507     # Visualization
508     pivot_edges = results_df.pivot(index='max_lag', columns='alpha',
509                                     values='n_edges')
509     pivot_density = results_df.pivot(index='max_lag', columns='alpha',
510                                       values='density')
511
512     _, axes = plt.subplots(1, 2, figsize=(16, 6))
513
514     sns.heatmap(pivot_edges, annot=True, fmt='.0f', cmap='YlOrRd',
515                 ax=axes[0], cbar_kws={'label': 'Number of Edges'})
516     axes[0].set_title('Number of Causal Edges vs Hyperparameters')
517     axes[0].set_xlabel('Significance Level ()')
518     axes[0].set_ylabel('Max Lag')
519
520     sns.heatmap(pivot_density, annot=True, fmt='.3f', cmap='viridis',
521                 ax=axes[1], cbar_kws={'label': 'Graph Density'})
522     axes[1].set_title('Graph Density vs Hyperparameters')
523     axes[1].set_xlabel('Significance Level ()')
524     axes[1].set_ylabel('Max Lag')
525
526     plt.tight_layout()

```

```

,
'
params_f
'
]
]

```

```

526         plt.savefig(os.path.join(self.figure_dir, '1e_granger_tuning.png'),
527                        dpi=150)
528
529     print("\n" + "=" * 70)
530
531     def _plot_granger_results(self, granger_df, p_value_df, lags_df,
532                               significance_level):
533
534         n_edges = int(granger_df.sum().sum())
535
536         print("\n--- Granger Causality Test Results ---")
537         print(f"Significance level: {significance_level}")
538         print(f"Number of significant Granger causal relationships: {n_edges}")
539
540         print(f"Graph density: {n_edges / (len(self.tickers) * (len(self.
541                                     tickers) - 1)):.3f}")
542
543         # Plot 1: Granger causality adjacency matrix
544         _, axes = plt.subplots(1, 2, figsize=(20, 8))
545
546         sns.heatmap(granger_df, annot=True, fmt='.0f', cmap='Blues',
547                    xticklabels=self.tickers, yticklabels=self.tickers,
548                    ax=axes[0], cbar_kws={'label': 'Granger Causes (1=Yes, 0=No
549                                     )'})
550
551         axes[0].set_title(f'Granger Causality Matrix ({significance_level})\
552                           nRow causes Column')
553
554         axes[0].set_xlabel('Effect (Y)')
555         axes[0].set_ylabel('Cause (X)')
556
557         p_value_masked = p_value_df.copy()
558         p_value_masked[granger_df == 0] = np.nan
559
560         sns.heatmap(p_value_masked, annot=True, fmt='.3f', cmap='Reds_r',
561                    xticklabels=self.tickers, yticklabels=self.tickers,
562                    ax=axes[1], cbar_kws={'label': 'P-value'}, vmin=0, vmax=
563                                     significance_level)
564
565         axes[1].set_title(f'P-values for Significant Relationships')
566         axes[1].set_xlabel('Effect (Y)')
567         axes[1].set_ylabel('Cause (X)')
568
569         plt.tight_layout()
570         plt.savefig(os.path.join(self.figure_dir, '1e_granger_matrix.png'),
571                        dpi=150)
572
573         plt.show()
574
575         # Plot 2: Network graph
576         G = nx.DiGraph()
577         for ticker in self.tickers:
578             G.add_node(ticker)
579
580         edge_list = []
581         for i, cause in enumerate(self.tickers):
582             for j, effect in enumerate(self.tickers):
583                 if granger_df.iloc[i, j] == 1:

```

```

572         edge_list.append((cause, effect))
573         G.add_edge(cause, effect,
574                   weight=1 - p_value_df.iloc[i, j],
575                   lag=int(lags_df.iloc[i, j]))
576
577     plt.figure(figsize=(14, 14))
578     pos = nx.spring_layout(G, k=2, iterations=100, seed=42)
579
580     nx.draw_networkx_nodes(G, pos, node_size=2500, node_color='lightblue',
581                           alpha=0.9, edgecolors='navy', linewidths=2)
582     nx.draw_networkx_labels(G, pos, font_size=11, font_weight='bold')
583
584     if edge_list:
585         nx.draw_networkx_edges(G, pos, edgelist=edge_list,
586                               edge_color='darkblue', arrows=True,
587                               arrowstyle='->', connectionstyle='arc3,rad=0.1',
588                               width=2, alpha=0.7, node_size=2500)
589
590     plt.title(f"Granger Causality Network ({significance_level})\n"
591              f"{n_edges} significant causal relationships",
592              fontsize=14, pad=20)
593     plt.axis('off')
594     plt.tight_layout()
595     plt.savefig(os.path.join(self.figure_dir, 'le_granger_network.png'),
596               dpi=150, bbox_inches='tight')
597     plt.show()
598
599     # Summary statistics
600     print("\nTop 10 strongest Granger causal relationships:")
601     relationships = []
602     for i, cause in enumerate(self.tickers):
603         for j, effect in enumerate(self.tickers):
604             if granger_df.iloc[i, j] == 1:
605                 relationships.append({
606                     'Cause': cause,
607                     'Effect': effect,
608                     'P-value': p_value_df.iloc[i, j],
609                     'Lag': int(lags_df.iloc[i, j])
610                 })
611
612     if relationships:
613         relationships_df = pd.DataFrame(relationships).sort_values('P-
614                               value')
        print(relationships_df.head(10).to_string(index=False))

```

A.2 Problem 2a: KDE

A.2.1 Main Script.

```

1  '''
2  Author: Chuyang Su cs4570@columbia.edu
3  Date: 2025-11-24 19:58:17
4  LastEditTime: 2025-11-25 09:37:20
5  FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2_main.py
6  Description:
7      Main script to run data processing and clustering model fitting for
8          Homework 3.
9
10     Optimized flow:
11     1. process_data() -> X, y
12     2. fit_kde_models() -> trained models
13     3. generate_samples() -> generated data
14     4. evaluate_samples() -> metrics
15     5. visualize_generated_samples() + compare_distributions()
16
17  '''
18  from Prob2a_utils import Prob2Analysis
19
20  def main():
21      # Initialize analysis
22      p2 = Prob2Analysis(
23          output_dir='Homework 3/Code/Data',
24          figure_dir='Homework 3/Latex/Figures'
25      )
26      X, y = p2.process_data(verbose=False)
27
28      kde_results = p2.fit_kde_models(
29          spaces=['pca_20', 'pca_50', 'original'],
30          bandwidth_range=None, # Auto-generate
31          kernel='gaussian',
32          cv_folds=5,
33          verbose=False
34      )
35
36      # Generate from PCA-20 space (fastest and often best quality)
37      generated_pca20 = p2.generate_samples(
38          space='pca_20',
39          n_samples=100,
40          verbose=True
41      )
42
43      metrics = p2.evaluate_samples(verbose=True)
44
45      # Visualization and comparison
46      print("\nVisualizing generated samples...")
47      p2.visualize_generated_samples(n_display=25)
48
49      print("\nComparing original vs generated distributions...")
50      p2.compare_distributions()
51
52  if __name__ == "__main__":
53      main()

```

A.2.2 Utils Script.

```

1  '''
2  Author: Chuyang Su cs4570@columbia.edu
3  Date: 2025-11-24 19:58:10
4  LastEditTime: 2025-11-25 09:43:24
5  FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2a_utils.py
6  Description:
7      Utility functions and classes for Problem 2 of Homework 3.
8      Kernel Density Estimation (KDE) for digit generation.
9      - process_data: Load and preprocess sklearn digits dataset
10     - fit_kde_models: Fit KDE in multiple spaces with hyperparameter tuning
11     - generate_samples: Generate new samples from trained KDE
12     - evaluate_samples: Evaluate quality of generated samples
13  '''
14  import os
15  import numpy as np
16  import pandas as pd
17  import matplotlib.pyplot as plt
18  import seaborn as sns
19  from sklearn.datasets import load_digits
20  from sklearn.preprocessing import StandardScaler
21  from sklearn.decomposition import PCA
22  from sklearn.neighbors import KernelDensity
23  from sklearn.model_selection import KFold
24  from sklearn.metrics import silhouette_score
25  from scipy.spatial.distance import cdist
26
27
28  class Prob2Analysis:
29      def __init__(self,
30                  output_dir='Homework 3/Code/Data',
31                  figure_dir='Homework 3/Latex/Figures'):
32          self.output_dir = output_dir
33          self.figure_dir = figure_dir
34
35          os.makedirs(self.output_dir, exist_ok=True)
36          os.makedirs(self.figure_dir, exist_ok=True)
37
38          self.data = None
39          self.targets = None
40          self.kde_results = None
41          self.generated_samples = None
42
43      def process_data(self, verbose=False):
44          digits = load_digits()
45          X = digits.data
46          y = digits.target
47
48          self.data = X
49          self.targets = y
50
51          if verbose:
52              print(f"\nDataset shape: {X.shape}")

```



```

53     print(f"Number of classes: {len(np.unique(y))}")
54     print(f"Pixel value range: [{X.min():.1f}, {X.max():.1f}]")
55     print(f"Mean pixel value: {X.mean():.2f} ± {X.std():.2f}")
56
57     return X, y
58
59     def fit_kde_models(self,
60                         spaces=['pca_20', 'pca_50', 'original'],
61                         bandwidth_range=None,
62                         kernel='gaussian',
63                         cv_folds=5,
64                         verbose=False):
65         if self.data is None:
66             self.process_data()
67
68         if bandwidth_range is None:
69             bandwidth_range = np.logspace(-1.5, 0.5, 20)
70
71         results = {}
72
73         if verbose:
74             print("Hyperparameter Tuning: Bandwidth Selection")
75
76         for space in spaces:
77             # Prepare data in target space
78             if space == 'original':
79                 X_space = StandardScaler().fit_transform(self.data)
80                 n_features = self.data.shape[1]
81
82             else:
83                 n_comp = int(space.split('_')[1])
84                 pca = PCA(n_components=n_comp)
85                 X_space = pca.fit_transform(self.data)
86                 X_space = StandardScaler().fit_transform(X_space)
87                 n_features = n_comp
88
89             # Hyperparameter tuning
90             kde = KernelDensity(kernel=kernel, algorithm='ball_tree')
91             best_bandwidth = None
92             best_score = -np.inf
93             cv_scores_dict = {'bandwidth': [], 'mean_score': [], 'std_score':
94                               []}
95
96             kf = KFold(n_splits=cv_folds, shuffle=True, random_state=25)
97
98             for bw in bandwidth_range:
99                 fold_scores = []
100                 for train_idx, test_idx in kf.split(X_space):
101                     X_train, X_test = X_space[train_idx], X_space[test_idx]
102                     kde = KernelDensity(bandwidth=bw)
103                     kde.fit(X_train)
104                     score = kde.score(X_test)
105                     fold_scores.append(score)

```

```

106         mean_score = np.mean(fold_scores)
107         std_score = np.std(fold_scores)
108
109         cv_scores_dict['bandwidth'].append(bw)
110         cv_scores_dict['mean_score'].append(mean_score)
111         cv_scores_dict['std_score'].append(std_score)
112
113         if mean_score > best_score:
114             best_score = mean_score
115             best_bandwidth = bw
116
117     if verbose:
118         print(f"    Best bandwidth: {best_bandwidth:.6f}")
119         print(f"    Cross-validation score: {best_score:.4f}")
120
121     # Train final model
122     kde_final = KernelDensity(
123         kernel=kernel,
124         bandwidth=best_bandwidth,
125         algorithm='ball_tree'
126     )
127     kde_final.fit(X_space)
128
129     results[space] = {
130         'model': kde_final,
131         'bandwidth': best_bandwidth,
132         'cv_score': best_score,
133         'X_train': X_space,
134         'cv_results': cv_scores_dict,
135         'n_features': n_features,
136         'pca': PCA(n_components=int(space.split('_')[1]))
137             if space.startswith('pca_') else None
138     }
139
140     self.kde_results = results
141
142     if verbose:
143         self._plot_kde_tuning_results(results)
144
145     return results
146
147     def _plot_kde_tuning_results(self, results):
148         """Visualize bandwidth tuning for each space."""
149         n_spaces = len(results)
150         fig, axes = plt.subplots(1, n_spaces, figsize=(5*n_spaces, 4))
151
152         if n_spaces == 1:
153             axes = [axes]
154
155         for idx, (space, result) in enumerate(results.items()):
156             cv_results = result['cv_results']
157             bandwidths = cv_results['bandwidth']
158             mean_scores = cv_results['mean_score']
159             std_scores = cv_results['std_score']

```

```

160         axes[idx].semilogx(bandwidths, mean_scores, 'o-', linewidth=2,
161                             markersize=6)
162         axes[idx].fill_between(bandwidths,
163                                np.array(mean_scores) - np.array(std_scores),
164                                np.array(mean_scores) + np.array(std_scores),
165                                alpha=0.2)
166         axes[idx].axvline(result['bandwidth'], linestyle='--', color='r',
167                             label=f"Best: {result['bandwidth']:.4f}")
168         axes[idx].set_xlabel('Bandwidth')
169         axes[idx].set_ylabel('Mean Log-Likelihood (Higher is Better)')
170         axes[idx].set_title(f'Bandwidth Tuning: {space}')
171         axes[idx].legend()
172         axes[idx].grid(True, alpha=0.3)
173
174     plt.tight_layout()
175     plt.savefig(os.path.join(self.figure_dir, '2a_kde_tuning.png'), dpi=
176                    150)
177
178     plt.show()
179
180     def generate_samples(self, space='pca_20', n_samples=100, verbose=False):
181         result = self.kde_results[space]
182         model = result['model']
183
184         # Generate samples in latent space
185         X_generated = model.sample(n_samples, random_state=42)
186
187         # Transform back to original space
188         if space == 'original':
189             # Need to inverse standardization
190             # For original space, we need the scaler that was used
191             scaler = StandardScaler()
192             scaler.fit(self.data)
193             X_original = scaler.inverse_transform(X_generated)
194
195         else:
196             # Inverse PCA transformation
197             pca = result['pca']
198             pca.fit(self.data)
199             X_pca_space = X_generated
200
201             # Inverse standardization in PCA space
202             scaler = StandardScaler()
203             scaler.fit(pca.transform(self.data))
204             X_pca_unscaled = scaler.inverse_transform(X_pca_space)
205
206             # Inverse PCA
207             X_original = pca.inverse_transform(X_pca_unscaled)
208
209         # Clip to valid range [0, 16]
210         X_original = np.clip(X_original, 0, 16)

```

```

210     self.generated_samples = {
211         'samples': X_original,
212         'space': space,
213         'n_samples': n_samples
214     }
215
216     if verbose:
217         print(f"\nGenerated {n_samples} samples from space: {space}")
218         print(f"Generated samples shape: {X_original.shape}")
219         print(f"Generated samples range: [{X_original.min():.2f}, {
220                                                     X_original.max():.2f}]")
221
222     return X_original
223
224 def evaluate_samples(self, generated_samples=None, verbose=False):
225     if generated_samples is None:
226         generated_samples = self.generated_samples['samples']
227
228     metrics = {}
229
230     # 1. Sample statistics comparison
231     metrics['original_mean'] = self.data.mean()
232     metrics['original_std'] = self.data.std()
233     metrics['generated_mean'] = generated_samples.mean()
234     metrics['generated_std'] = generated_samples.std()
235
236     # 2. Sparsity check (% of zero pixels)
237     metrics['original_sparsity'] = (self.data == 0).mean()
238     metrics['generated_sparsity'] = (generated_samples == 0).mean()
239
240     # 3. Quality check: Can we classify generated samples?
241     try:
242         from sklearn.neighbors import KNeighborsClassifier
243         from sklearn.model_selection import cross_val_score
244
245         knn = KNeighborsClassifier(n_neighbors=5)
246         scores = cross_val_score(
247             knn, self.data, self.targets, cv=5, scoring='accuracy'
248         )
249         metrics['knn_accuracy_original'] = scores.mean()
250
251     except Exception as e:
252         metrics['knn_accuracy_original'] = None
253
254     if verbose:
255         self._print_evaluation_metrics(metrics)
256
257     return metrics
258
259 def _print_evaluation_metrics(self, metrics):
260     print(f"Mean (Original):      {metrics['original_mean']:.4f}")
261     print(f"Mean (Generated):      {metrics['generated_mean']:.4f}")
262     print(f"Std (Original):         {metrics['original_std']:.4f}")
263     print(f"Std (Generated):        {metrics['generated_std']:.4f}")

```

```

263 print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
264 print(f"Sparsity (Generated):{metrics['generated_sparsity']:.4f}")
265 if metrics['knn_accuracy_original'] is not None:
266     print(f"KNN Accuracy (Original): {metrics['knn_accuracy_original']:.4f}")
267
268 def visualize_generated_samples(self, generated_samples=None, n_display=25
269                                ):
270     if generated_samples is None:
271         generated_samples = self.generated_samples['samples']
272         space = self.generated_samples['space']
273     else:
274         space = 'provided'
275
276     grid_size = int(np.sqrt(n_display))
277     fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
278     axes = axes.flatten()
279
280     for i in range(min(n_display, len(generated_samples))):
281         axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
282         axes[i].axis('off')
283
284     plt.suptitle(f'Generated Digits from KDE ({space})', fontsize=14)
285     plt.tight_layout()
286     plt.savefig(os.path.join(self.figure_dir, f'2a_generated_samples_{space}.png'), dpi=150)
287     plt.show()
288
289 def compare_distributions(self, generated_samples=None):
290     if generated_samples is None:
291         generated_samples = self.generated_samples['samples']
292
293     fig, axes = plt.subplots(1, 2, figsize=(14, 5))
294
295     # Pixel intensity distribution
296     axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original',
297                 color='blue', density=True)
298     axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='Generated',
299                 color='red', density=True)
300     axes[0].set_xlabel('Pixel Intensity')
301     axes[0].set_ylabel('Density')
302     axes[0].set_title('Pixel Intensity Distribution')
303     axes[0].legend()
304     axes[0].grid(alpha=0.3)
305
306     # Sparsity comparison
307     sparsity_orig = (self.data == 0).sum(axis=1).mean()
308     sparsity_gen = (generated_samples == 0).sum(axis=1).mean()
309
310     categories = ['Original', 'Generated']
311     sparsities = [sparsity_orig, sparsity_gen]

```

```

311     axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7,
312                edgecolor='black')
313     axes[1].set_ylabel('Average # of Zero Pixels')
314     axes[1].set_title('Sparsity Comparison')
315     axes[1].grid(axis='y', alpha=0.3)
316
317     plt.tight_layout()
318     plt.savefig(os.path.join(self.figure_dir, '2a_distribution_comparison.
                                png'), dpi=150)
319
320     plt.show()

```

A.3 Problem 2b: GAN

A.3.1 Main Script.

```

1  '''
2  Author: Chuyang Su cs4570@columbia.edu
3  Date: 2025-11-25 10:49:06
4  LastEditTime: 2025-11-25 13:28:54
5  FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2b_main.py
6  Description:
7      Main script to run GAN model fitting for Homework 3 Problem 2b.
8      Optimized flow:
9      1. process_data() -> X, y
10     2. tune_hyperparameters() (optional) -> best config
11     3. fit_gan() -> trained GAN
12     4. generate_samples() -> generated data
13     5. evaluate_samples() -> metrics
14     6. visualize_generated_samples() + compare_distributions()
15  '''
16  from Prob2b_utils import Prob2bAnalysis
17
18
19  def main():
20      # Initialize analysis
21      p2b = Prob2bAnalysis(
22          output_dir='Homework 3/Code/Data',
23          figure_dir='Homework 3/Latex/Figures'
24      )
25
26      # Load and preprocess data
27      X, y = p2b.process_data(verbose=False)
28
29      # Hyperparameter tuning
30      tuning_results = p2b.tune_hyperparameters(
31          latent_dims=[32, 64, 128],
32          hidden_configs=[
33              ([128, 256], [256, 128]),
34              ([256, 512], [512, 256]),
35          ],
36          n_epochs=200,
37          verbose=False

```

```

38 )
39 # Best parameters from tuning: latent_dim=32, lr_g=0.0001, G=[128, 256]
40
41 gan_results = p2b.fit_gan(
42     latent_dim=32,
43     g_hidden_dims=[128, 256],
44     d_hidden_dims=[256, 128],
45     lr_g=0.0001,
46     lr_d=0.0002,
47     batch_size=64,
48     n_epochs=300,
49     n_critic=1,
50     beta1=0.5,
51     label_smoothing=0.1,
52     verbose=False
53 )
54
55 # Generate samples
56 generated_samples = p2b.generate_samples(
57     n_samples=100,
58     verbose=False
59 )
60
61 # Evaluate generated samples
62 metrics = p2b.evaluate_samples(verbose=True)
63
64 # Visualization
65 print("\nVisualizing generated samples...")
66 p2b.visualize_generated_samples(n_display=25)
67
68 print("\nComparing original vs generated distributions...")
69 p2b.compare_distributions()
70
71 print("\nVisualizing latent space interpolation...")
72 p2b.visualize_latent_interpolation(n_steps=10)
73
74 print("\nComparing real vs generated samples side-by-side...")
75 p2b.compare_with_real_samples(n_display=10)
76
77
78 if __name__ == "__main__":
79     main()

```

A.3.2 Utils Script.

```

1 '''
2 Author: Chuyang Su cs4570@columbia.edu
3 Date: 2025-11-25 10:49:19
4 LastEditTime: 2025-11-25 12:44:23
5 FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2b_utils.py
6 Description:
7     Utility functions and classes for Problem 2b of Homework 3.

```

```

8     Generative Adversarial Network (GAN) for digit generation.
9     - process_data: Load and preprocess sklearn digits dataset
10    - build_generator: Build generator network
11    - build_discriminator: Build discriminator network
12    - fit_gan: Train GAN with hyperparameter tuning
13    - generate_samples: Generate new samples from trained GAN
14    - evaluate_samples: Evaluate quality of generated samples
15    '''
16    import os
17    import numpy as np
18    import pandas as pd
19    import matplotlib.pyplot as plt
20    import seaborn as sns
21    from sklearn.datasets import load_digits
22    from sklearn.preprocessing import MinMaxScaler, StandardScaler
23    from sklearn.decomposition import PCA
24    from sklearn.neighbors import KNeighborsClassifier
25    from sklearn.model_selection import cross_val_score
26    import torch
27    import torch.nn as nn
28    import torch.optim as optim
29    from torch.utils.data import DataLoader, TensorDataset
30
31    class Generator(nn.Module):
32        def __init__(self, latent_dim=64, hidden_dims=[128, 256], output_dim=64,
33                    activation='relu', use_batchnorm=True):
34            super(Generator, self).__init__()
35
36            self.latent_dim = latent_dim
37
38            # Build layers
39            layers = []
40            in_dim = latent_dim
41
42            for h_dim in hidden_dims:
43                layers.append(nn.Linear(in_dim, h_dim))
44                if use_batchnorm:
45                    layers.append(nn.BatchNorm1d(h_dim))
46                if activation == 'relu':
47                    layers.append(nn.ReLU())
48                elif activation == 'leaky_relu':
49                    layers.append(nn.LeakyReLU(0.2))
50                elif activation == 'tanh':
51                    layers.append(nn.Tanh())
52                in_dim = h_dim
53
54            # Output layer with Sigmoid for [0, 1] range
55            layers.append(nn.Linear(in_dim, output_dim))
56            layers.append(nn.Sigmoid())
57
58            self.model = nn.Sequential(*layers)
59
60    def forward(self, z):
61        return self.model(z)

```



```

62
63
64 class Discriminator(nn.Module):
65     def __init__(self, input_dim=64, hidden_dims=[256, 128],
66                 activation='leaky_relu', dropout_rate=0.3):
67         super(Discriminator, self).__init__()
68
69         layers = []
70         in_dim = input_dim
71
72         for h_dim in hidden_dims:
73             layers.append(nn.Linear(in_dim, h_dim))
74             if activation == 'leaky_relu':
75                 layers.append(nn.LeakyReLU(0.2))
76             elif activation == 'relu':
77                 layers.append(nn.ReLU())
78             if dropout_rate > 0:
79                 layers.append(nn.Dropout(dropout_rate))
80             in_dim = h_dim
81
82         # Output layer with Sigmoid for probability
83         layers.append(nn.Linear(in_dim, 1))
84         layers.append(nn.Sigmoid())
85
86         self.model = nn.Sequential(*layers)
87
88     def forward(self, x):
89         return self.model(x)
90
91
92 class Prob2bAnalysis:
93     def __init__(self,
94                 output_dir='Homework 3/Code/Data',
95                 figure_dir='Homework 3/Latex/Figures',
96                 device=None,
97                 seed=25):
98         self.output_dir = output_dir
99         self.figure_dir = figure_dir
100
101         os.makedirs(self.output_dir, exist_ok=True)
102         os.makedirs(self.figure_dir, exist_ok=True)
103
104         # Set device
105         if device is None:
106             self.device = torch.device('cuda' if torch.cuda.is_available()
107                                     else 'cpu')
108             print(f"Using device: {self.device}")
109         else:
110             self.device = device
111
112         self.data = None
113         self.targets = None
114         self.scaler = None
115         self.gan_results = None

```

```

115         self.generated_samples = None
116         self.training_history = None
117
118         self.seed = seed
119
120     def _set_seed(self):
121         np.random.seed(self.seed)
122         torch.manual_seed(self.seed)
123         if torch.cuda.is_available():
124             torch.cuda.manual_seed_all(self.seed)
125
126     def process_data(self, verbose=False):
127         # Different from 2a process_data, so cannot share code
128         digits = load_digits()
129         X = digits.data
130         y = digits.target
131
132         # Scale to [0, 1] for GAN training
133         self.scaler = MinMaxScaler(feature_range=(0, 1))
134         X_scaled = self.scaler.fit_transform(X)
135
136         self.data = X
137         self.data_scaled = X_scaled
138         self.targets = y
139
140         if verbose:
141             print(f"\nDataset shape: {X.shape}")
142             print(f"Number of classes: {len(np.unique(y))}")
143             print(f"Original pixel range: [{X.min():.1f}, {X.max():.1f}]")
144             print(f"Scaled pixel range: [{X_scaled.min():.3f}, {X_scaled.max():.3f}]")
145
146             print(f"Device: {self.device}")
147
148         return X, y
149
150     def fit_gan(self,
151                 latent_dim=64,
152                 g_hidden_dims=[128, 256],
153                 d_hidden_dims=[256, 128],
154                 lr_g=0.0002,
155                 lr_d=0.0002,
156                 batch_size=64,
157                 n_epochs=300,
158                 n_critic=1,
159                 beta1=0.5,
160                 beta2=0.999,
161                 label_smoothing=0.1,
162                 verbose=False):
163         self._set_seed()
164         if self.data is None:
165             self.process_data()
166
167         output_dim = self.data.shape[1] # 64 for 8x8 images

```

```

168     # Build networks
169     generator = Generator(
170         latent_dim=latent_dim,
171         hidden_dims=g_hidden_dims,
172         output_dim=output_dim,
173         activation='relu',
174         use_batchnorm=True
175     ).to(self.device)
176
177     discriminator = Discriminator(
178         input_dim=output_dim,
179         hidden_dims=d_hidden_dims,
180         activation='leaky_relu',
181         dropout_rate=0.3
182     ).to(self.device)
183
184     # Optimizers
185     optimizer_g = optim.Adam(generator.parameters(), lr=lr_g, betas=(beta1
186                                     , beta2))
187
188     optimizer_d = optim.Adam(discriminator.parameters(), lr=lr_d, betas=(
189                                     beta1, beta2))
190
191     # Loss function
192     criterion = nn.BCELoss()
193
194     # Prepare data
195     X_tensor = torch.FloatTensor(self.data_scaled).to(self.device)
196     dataset = TensorDataset(X_tensor)
197     dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
198                             drop_last=True)
199
200     # Training history
201     history = {
202         'epoch': [],
203         'd_loss': [],
204         'g_loss': [],
205         'd_real_acc': [],
206         'd_fake_acc': []
207     }
208
209     if verbose:
210         print(f"\nTraining GAN...")
211         print(f"  Latent dim: {latent_dim}")
212         print(f"  Generator: {g_hidden_dims}")
213         print(f"  Discriminator: {d_hidden_dims}")
214         print(f"  Epochs: {n_epochs}, Batch size: {batch_size}")
215         print(f"  "-" * 60)
216
217     # Training loop
218     for epoch in range(n_epochs):
219         d_losses = []
220         g_losses = []
221         d_real_accs = []
222         d_fake_accs = []

```

```

219
220     for batch_data in dataloader:
221         real_data = batch_data[0]
222         current_batch_size = real_data.size(0)
223
224         # Labels with smoothing
225         real_labels = torch.ones(current_batch_size, 1).to(self.device
226                                     ) * (1 -
227                                         label_smoothing)
228
229         fake_labels = torch.zeros(current_batch_size, 1).to(self.
230                                     device)
231
232         # -----
233         # Train Discriminator
234         # -----
235         for _ in range(n_critic):
236             optimizer_d.zero_grad()
237
238             # Real data
239             d_real_output = discriminator(real_data)
240             d_real_loss = criterion(d_real_output, real_labels)
241
242             # Fake data
243             z = torch.randn(current_batch_size, latent_dim).to(self.
244                                     device)
245             fake_data = generator(z)
246             d_fake_output = discriminator(fake_data.detach())
247             d_fake_loss = criterion(d_fake_output, fake_labels)
248
249             # Total discriminator loss
250             d_loss = d_real_loss + d_fake_loss
251             d_loss.backward()
252             optimizer_d.step()
253
254             d_losses.append(d_loss.item())
255             d_real_accs.append((d_real_output > 0.5).float().mean().item()
256                               )
257             d_fake_accs.append((d_fake_output < 0.5).float().mean().item()
258                               )
259
260         # -----
261         # Train Generator
262         # -----
263         optimizer_g.zero_grad()
264
265         z = torch.randn(current_batch_size, latent_dim).to(self.device
266                                     )
267         fake_data = generator(z)
268         g_output = discriminator(fake_data)
269
270         # Generator wants discriminator to think fake is real
271         g_loss = criterion(g_output, torch.ones(current_batch_size, 1)
272                                     .to(self.device))
273         g_loss.backward()

```

```

265         optimizer_g.step()
266
267         g_losses.append(g_loss.item())
268
269         # Record epoch metrics
270         history['epoch'].append(epoch + 1)
271         history['d_loss'].append(np.mean(d_losses))
272         history['g_loss'].append(np.mean(g_losses))
273         history['d_real_acc'].append(np.mean(d_real_accs))
274         history['d_fake_acc'].append(np.mean(d_fake_accs))
275
276         # Print progress
277         if verbose and (epoch + 1) % 50 == 0:
278             print(f"Epoch [{epoch+1:4d}/{n_epochs}] | "
279                   f"D Loss: {history['d_loss'][-1]:.4f} | "
280                   f"G Loss: {history['g_loss'][-1]:.4f} | "
281                   f"D(real): {history['d_real_acc'][-1]:.3f} | "
282                   f"D(fake): {history['d_fake_acc'][-1]:.3f}")
283
284         # Store results
285         self.gan_results = {
286             'generator': generator,
287             'discriminator': discriminator,
288             'latent_dim': latent_dim,
289             'history': history,
290             'config': {
291                 'latent_dim': latent_dim,
292                 'g_hidden_dims': g_hidden_dims,
293                 'd_hidden_dims': d_hidden_dims,
294                 'lr_g': lr_g,
295                 'lr_d': lr_d,
296                 'batch_size': batch_size,
297                 'n_epochs': n_epochs
298             }
299         }
300
301         self.training_history = history
302
303         if verbose:
304             print("Training completed!")
305             self._plot_training_history(history)
306
307         return self.gan_results
308
309     def tune_hyperparameters(self,
310                             latent_dims=[32, 64, 128],
311                             lr_g_values=[0.00005, 0.0001, 0.0002],
312                             lr_d=0.0002,
313                             hidden_configs=[
314                                 ([128, 256], [256, 128]),
315                                 ([256, 512], [512, 256]),
316                                 ([64, 128, 256], [256, 128, 64])
317                             ],
318                             n_epochs=200,

```

```

319         n_eval_samples=500,
320         verbose=False):
321     if self.data is None:
322         self.process_data()
323
324     results = []
325
326     for latent_dim in latent_dims:
327         for lr_g in lr_g_values:
328             for g_hidden, d_hidden in hidden_configs:
329                 # Train model
330                 self.fit_gan(
331                     latent_dim=latent_dim,
332                     g_hidden_dims=g_hidden,
333                     d_hidden_dims=d_hidden,
334                     lr_g=lr_g,
335                     lr_d=lr_d,
336                     n_epochs=n_epochs,
337                     verbose=False
338                 )
339
340                 # Generate and evaluate samples
341                 samples = self.generate_samples(n_samples=n_eval_samples,
342                                                 verbose=False)
343                 metrics = self.evaluate_samples(samples, verbose=False)
344
345                 # Store results
346                 result = {
347                     'latent_dim': latent_dim,
348                     'lr_g': lr_g,
349                     'g_hidden': str(g_hidden),
350                     'd_hidden': str(d_hidden),
351                     'mean_diff': abs(metrics['generated_mean'] - metrics['
352                                     original_mean']
353                                     ),
354                     'std_diff': abs(metrics['generated_std'] - metrics['
355                                     original_std'])
356                                     ,
357                     'diversity_ratio': metrics.get('diversity_ratio', 0)
358                 }
359                 results.append(result)
360
361                 if verbose:
362                     print(f" Mean Diff: {result['mean_diff']:.4f} | "
363                           f"Std Diff: {result['std_diff']:.4f} | "
364                           f"Diversity: {result['diversity_ratio']:.3f}")
365
366     results_df = pd.DataFrame(results)
367
368     if verbose:
369         print("Tuning Results Summary:")
370         print(results_df.to_string(index=False))
371         self._plot_tuning_results(results_df)

```

```

368         return results_df
369
370     def _plot_tuning_results(self, results_df):
371         fig, axes = plt.subplots(1, 3, figsize=(15, 5))
372
373         # Mean diff by lr_g, grouped by latent_dim
374         for latent in results_df['latent_dim'].unique():
375             subset = results_df[results_df['latent_dim'] == latent]
376             avg_by_lr = subset.groupby('lr_g')['mean_diff'].mean()
377             axes[0].plot(avg_by_lr.index, avg_by_lr.values, 'o-', label=f'
                                     latent={latent}',
                                     markersize=8)
378         axes[0].set_title('Mean Diff vs lr_g (Lower is Better)')
379         axes[0].set_xscale('log')
380         axes[0].legend()
381         axes[0].grid(True, alpha=0.3)
382
383         # Mean difference
384         pivot = results_df.pivot_table(
385             values='mean_diff',
386             index='latent_dim',
387             columns='lr_g',
388             aggfunc='mean'
389         )
390         sns.heatmap(pivot, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[1])
391         axes[1].set_title('Mean Difference (Lower is Better)')
392
393         # Std difference
394         pivot_std = results_df.pivot_table(
395             values='std_diff',
396             index='latent_dim',
397             columns='lr_g',
398             aggfunc='mean'
399         )
400         sns.heatmap(pivot_std, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[2]
401                     ])
402         axes[2].set_title('Std Difference (Lower is Better)')
403
404         plt.tight_layout()
405         plt.savefig(os.path.join(self.figure_dir, '2b_gan_tuning.png'), dpi=
406                     150)
407         plt.show()
408
409     def _plot_training_history(self, history):
410         fig, axes = plt.subplots(1, 2, figsize=(14, 5))
411
412         epochs = history['epoch']
413
414         # Losses
415         axes[0].plot(epochs, history['d_loss'], label='Discriminator Loss',
416                     alpha=0.8)
417         axes[0].plot(epochs, history['g_loss'], label='Generator Loss', alpha=
418                     0.8)
419         axes[0].set_xlabel('Epoch')

```

```

416 axes[0].set_ylabel('Loss')
417 axes[0].set_title('GAN Training Losses')
418 axes[0].legend()
419 axes[0].grid(True, alpha=0.3)
420
421 # Discriminator accuracy
422 axes[1].plot(epochs, history['d_real_acc'], label='D(real) accuracy',
423             alpha=0.8)
424 axes[1].plot(epochs, history['d_fake_acc'], label='D(fake) accuracy',
425             alpha=0.8)
426 axes[1].axhline(y=0.5, color='r', linestyle='--', alpha=0.5, label='
427             Random guess')
428 axes[1].set_xlabel('Epoch')
429 axes[1].set_ylabel('Accuracy')
430 axes[1].set_title('Discriminator Performance')
431 axes[1].legend()
432 axes[1].grid(True, alpha=0.3)
433 axes[1].set_ylim([0, 1])
434
435 plt.tight_layout()
436 plt.savefig(os.path.join(self.figure_dir, '2b_gan_training.png'), dpi=
437             150)
438
439 plt.show()
440
441 def generate_samples(self, n_samples=100, verbose=False):
442     generator = self.gan_results['generator']
443     latent_dim = self.gan_results['latent_dim']
444
445     generator.eval()
446     with torch.no_grad():
447         z = torch.randn(n_samples, latent_dim).to(self.device)
448         generated_scaled = generator(z).cpu().numpy()
449
450     # Inverse transform to original scale
451     generated = self.scaler.inverse_transform(generated_scaled)
452
453     # Clip to valid range [0, 16]
454     generated = np.clip(generated, 0, 16)
455
456     self.generated_samples = {
457         'samples': generated,
458         'samples_scaled': generated_scaled,
459         'n_samples': n_samples
460     }
461
462     if verbose:
463         print(f"\nGenerated {n_samples} samples from GAN")
464         print(f"Generated samples shape: {generated.shape}")
465         print(f"Generated samples range: [{generated.min():.2f}, {
466             generated.max():.2f}]")
467
468     return generated
469
470 def evaluate_samples(self, generated_samples=None, verbose=False):

```



```

465     if generated_samples is None:
466         generated_samples = self.generated_samples['samples']
467
468     metrics = {}
469
470     # 1. Sample statistics comparison
471     metrics['original_mean'] = self.data.mean()
472     metrics['original_std'] = self.data.std()
473     metrics['generated_mean'] = generated_samples.mean()
474     metrics['generated_std'] = generated_samples.std()
475
476     # 2. Sparsity check (% of near-zero pixels)
477     threshold = 0.5
478     metrics['original_sparsity'] = (self.data < threshold).mean()
479     metrics['generated_sparsity'] = (generated_samples < threshold).mean()
480
481     # 3. Coverage: fraction of original data modes covered
482     # Using PCA + clustering approximation
483     try:
484         pca = PCA(n_components=10)
485         orig_pca = pca.fit_transform(self.data)
486         gen_pca = pca.transform(generated_samples)
487
488         # Simple coverage: check if generated samples are near original
489                                     samples
490         from scipy.spatial.distance import cdist
491         distances = cdist(gen_pca, orig_pca, metric='euclidean')
492         min_distances = distances.min(axis=1)
493         coverage_threshold = np.percentile(
494             cdist(orig_pca, orig_pca).flatten(), 50
495         )
496         metrics['coverage'] = (min_distances < coverage_threshold).mean()
497     except Exception as e:
498         metrics['coverage'] = None
499
500     # 4. Mode collapse check: diversity of generated samples
501     try:
502         gen_pca = PCA(n_components=10).fit_transform(generated_samples)
503         pairwise_dist = cdist(gen_pca, gen_pca, metric='euclidean')
504         np.fill_diagonal(pairwise_dist, np.inf)
505         metrics['avg_nn_distance'] = pairwise_dist.min(axis=1).mean()
506
507         orig_pca = PCA(n_components=10).fit_transform(self.data)
508         orig_pairwise = cdist(orig_pca, orig_pca, metric='euclidean')
509         np.fill_diagonal(orig_pairwise, np.inf)
510         metrics['orig_avg_nn_distance'] = orig_pairwise.min(axis=1).mean()
511
512         # Diversity ratio (higher is better, 1.0 means same diversity as
513                                     original)
514         metrics['diversity_ratio'] = metrics['avg_nn_distance'] / metrics[
515             'orig_avg_nn_distance']
516     except Exception as e:
517         metrics['diversity_ratio'] = None
518 
```

```

516         if verbose:
517             self._print_evaluation_metrics(metrics)
518
519         return metrics
520
521     def _print_evaluation_metrics(self, metrics):
522         """Print evaluation metrics."""
523         print("\n--- GAN Evaluation Metrics ---")
524         print(f"Mean (Original):      {metrics['original_mean']:.4f}")
525         print(f"Mean (Generated):      {metrics['generated_mean']:.4f}")
526         print(f"Std (Original):      {metrics['original_std']:.4f}")
527         print(f"Std (Generated):      {metrics['generated_std']:.4f}")
528         print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
529         print(f"Sparsity (Generated): {metrics['generated_sparsity']:.4f}")
530         if metrics.get('coverage') is not None:
531             print(f"Coverage:      {metrics['coverage']:.4f}")
532         if metrics.get('diversity_ratio') is not None:
533             print(f"Diversity Ratio:      {metrics['diversity_ratio']:.4f}")
534
535     def visualize_generated_samples(self, generated_samples=None, n_display=25
536                                   ):
537         """Visualize grid of generated samples."""
538         if generated_samples is None:
539             generated_samples = self.generated_samples['samples']
540
541         grid_size = int(np.sqrt(n_display))
542         fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
543         axes = axes.flatten()
544
545         for i in range(min(n_display, len(generated_samples))):
546             axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
547             axes[i].axis('off')
548
549         plt.suptitle('Generated Digits from GAN', fontsize=14)
550         plt.tight_layout()
551         plt.savefig(os.path.join(self.figure_dir, '2b_generated_samples_gan.
552                               png'), dpi=150)
553
554         plt.show()
555
556     def compare_distributions(self, generated_samples=None):
557         if generated_samples is None:
558             generated_samples = self.generated_samples['samples']
559
560         fig, axes = plt.subplots(1, 3, figsize=(18, 5))
561
562         # Pixel intensity distribution
563         axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original',
564                     color='blue', density=True)
565         axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='Generated',
566                     color='red', density=True)
567         axes[0].set_xlabel('Pixel Intensity')
568         axes[0].set_ylabel('Density')

```

```

566 axes[0].set_title('Pixel Intensity Distribution')
567 axes[0].legend()
568 axes[0].grid(alpha=0.3)
569
570 # Sparsity comparison
571 sparsity_orig = (self.data < 0.5).sum(axis=1).mean()
572 sparsity_gen = (generated_samples < 0.5).sum(axis=1).mean()
573
574 categories = ['Original', 'Generated']
575 sparsities = [sparsity_orig, sparsity_gen]
576 axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7,
577           edgecolor='black')
578 axes[1].set_ylabel('Average # of Near-Zero Pixels')
579 axes[1].set_title('Sparsity Comparison')
580 axes[1].grid(axis='y', alpha=0.3)
581
582 # PCA projection comparison
583 pca = PCA(n_components=2)
584 orig_pca = pca.fit_transform(self.data)
585 gen_pca = pca.transform(generated_samples)
586
587 axes[2].scatter(orig_pca[:, 0], orig_pca[:, 1], alpha=0.3, label='
588           Original', s=10)
589 axes[2].scatter(gen_pca[:, 0], gen_pca[:, 1], alpha=0.5, label='
590           Generated', s=10)
591 axes[2].set_xlabel('PC1')
592 axes[2].set_ylabel('PC2')
593 axes[2].set_title('PCA Projection Comparison')
594 axes[2].legend()
595 axes[2].grid(alpha=0.3)
596
597 plt.tight_layout()
598 plt.savefig(os.path.join(self.figure_dir, '
599           2b_distribution_comparison_gan.
600           png'), dpi=150)
601
602 plt.show()
603
604 def visualize_latent_interpolation(self, n_steps=10):
605     generator = self.gan_results['generator']
606     latent_dim = self.gan_results['latent_dim']
607
608     generator.eval()
609
610     # Generate two random latent vectors
611     z1 = torch.randn(1, latent_dim).to(self.device)
612     z2 = torch.randn(1, latent_dim).to(self.device)
613
614     # Interpolate
615     interpolations = []
616     for alpha in np.linspace(0, 1, n_steps):
617         z = (1 - alpha) * z1 + alpha * z2
618         with torch.no_grad():
619             img = generator(z).cpu().numpy()
620             img = self.scaler.inverse_transform(img)

```

```

615         img = np.clip(img, 0, 16)
616         interpolations.append(img.reshape(8, 8))
617
618     # Plot
619     fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2))
620     for i, img in enumerate(interpolations):
621         axes[i].imshow(img, cmap='gray')
622         axes[i].axis('off')
623         if i == 0:
624             axes[i].set_title('Start')
625         elif i == n_steps - 1:
626             axes[i].set_title('End')
627
628     plt.suptitle('Latent Space Interpolation', fontsize=14)
629     plt.tight_layout()
630     plt.savefig(os.path.join(self.figure_dir, '2b_latent_interpolation.png'),
631                 dpi=150)
632
633     plt.show()
634
635     def compare_with_real_samples(self, n_display=10):
636         generated = self.generated_samples['samples'][:n_display]
637         real_indices = np.random.choice(len(self.data), n_display, replace=False)
638         real = self.data[real_indices]
639
640         fig, axes = plt.subplots(2, n_display, figsize=(2 * n_display, 4))
641
642         for i in range(n_display):
643             axes[0, i].imshow(real[i].reshape(8, 8), cmap='gray')
644             axes[0, i].axis('off')
645             if i == 0:
646                 axes[0, i].set_ylabel('Real', fontsize=12)
647
648             axes[1, i].imshow(generated[i].reshape(8, 8), cmap='gray')
649             axes[1, i].axis('off')
650             if i == 0:
651                 axes[1, i].set_ylabel('Generated', fontsize=12)
652
653         plt.suptitle('Real vs Generated Samples', fontsize=14)
654         plt.tight_layout()
655         plt.savefig(os.path.join(self.figure_dir, '2b_real_vs_generated.png'),
656                     dpi=150)
657
658         plt.show()

```

A.4 Problem 2c: Diffusion Model.

A.4.1 Main Script.

```

1  '''
2  Author: Chuyang Su cs4570@columbia.edu
3  Date: 2025-11-25 15:08:40
4  LastEditTime: 2025-11-25 20:54:04

```

```

5 FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2c_main.py
6 Description:
7     Main script to run Diffusion Model fitting for Homework 3 Problem 2c.
8     Optimized flow:
9     1. process_data() -> X, y
10    2. tune_hyperparameters() (optional) -> best config
11    3. fit_diffusion() -> trained model
12    4. generate_samples() -> generated data
13    5. evaluate_samples() -> metrics
14    6. visualize_generated_samples() + compare_distributions()
15    '''
16 from Prob2c_utils import Prob2cAnalysis
17
18
19 def main():
20     # Initialize analysis (with seed for reproducibility)
21     p2c = Prob2cAnalysis(
22         output_dir='Homework 3/Code/Data',
23         figure_dir='Homework 3/Latex/Figures',
24         seed=25
25     )
26
27     # Load and preprocess data
28     X, y = p2c.process_data(verbose=False)
29
30     # Hyperparameter tuning
31     # tuning_results = p2c.tune_hyperparameters(
32     #     num_timesteps_values=[200, 500, 1000],
33     #     lr_values=[1e-4, 5e-4, 1e-3],
34     #     hidden_configs=[
35     #         [128, 256, 128],
36     #         [256, 512, 256],
37     #     ],
38     #     n_epochs=150,
39     #     verbose=False
40     # ) # Best results: lr=0.0005, timesteps=500, architecture=[256, 512, 256]
41
42     # Train Diffusion Model with selected hyperparameters
43     diffusion_results = p2c.fit_diffusion(
44         num_timesteps=500,
45         hidden_dims=[256, 512, 256],
46         time_emb_dim=64,
47         lr=5e-4,
48         batch_size=128,
49         n_epochs=200,
50         beta_start=1e-4,
51         beta_end=0.02,
52         verbose=False
53     )
54
55     # Generate samples
56     generated_samples = p2c.generate_samples(
57         n_samples=100,
58         verbose=True

```

```

59     )
60
61     # Evaluate generated samples
62     metrics = p2c.evaluate_samples(verbose=True)
63
64     # Visualization
65     print("\nVisualizing generated samples...")
66     p2c.visualize_generated_samples(n_display=25)
67
68     print("\nComparing original vs generated distributions...")
69     p2c.compare_distributions()
70
71     print("\nVisualizing reverse diffusion process...")
72     p2c.visualize_diffusion_process(n_steps=10)
73
74     print("\nComparing real vs generated samples side-by-side...")
75     p2c.compare_with_real_samples(n_display=10)
76
77
78 if __name__ == "__main__":
79     main()

```

A.4.2 Utils Script.

```

1  '''
2  Author: Chuyang Su cs4570@columbia.edu
3  Date: 2025-11-25 15:08:49
4  LastEditTime: 2025-11-25 20:56:24
5  FilePath: /Unsupervised-Learning-Homework/Homework 3/Code/Prob2c_utils.py
6  Description:
7      Utility functions and classes for Problem 2c of Homework 3.
8      Denoising Diffusion Probabilistic Model (DDPM) for digit generation.
9      - process_data: Load and preprocess sklearn digits dataset
10     - build_denoiser: Build denoising network
11     - fit_diffusion: Train diffusion model with hyperparameter tuning
12     - generate_samples: Generate new samples via reverse diffusion
13     - evaluate_samples: Evaluate quality of generated samples
14  '''
15  import os
16  import numpy as np
17  import pandas as pd
18  import matplotlib.pyplot as plt
19  import seaborn as sns
20  from sklearn.datasets import load_digits
21  from sklearn.preprocessing import MinMaxScaler
22  from sklearn.decomposition import PCA
23  from scipy.spatial.distance import cdist
24  import torch
25  import torch.nn as nn
26  import torch.optim as optim
27  from torch.utils.data import DataLoader, TensorDataset
28  from tqdm import tqdm

```

```

29
30
31 class SinusoidalPositionEmbeddings(nn.Module):
32     def __init__(self, dim):
33         super().__init__()
34         self.dim = dim
35
36     def forward(self, time):
37         device = time.device
38         half_dim = self.dim // 2
39         embeddings = np.log(10000) / (half_dim - 1)
40         embeddings = torch.exp(torch.arange(half_dim, device=device) * -
                                     embeddings)
41         embeddings = time[:, None] * embeddings[None, :]
42         embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1) #
                                     Use both sin and cos to
                                     capture more information
43
44         return embeddings
45
46 class DenoisingMLP(nn.Module):
47     def __init__(self, input_dim=64, hidden_dims=[256, 512, 256], time_emb_dim
                                     =64):
48         super().__init__()
49
50         self.time_mlp = nn.Sequential(
51             SinusoidalPositionEmbeddings(time_emb_dim),
52             nn.Linear(time_emb_dim, time_emb_dim * 2),
53             nn.GELU(), # Instead of ReLU, GELU(Gaussian Error Linear Unit) as
                                     a smooth version often
                                     performs better
54             nn.Linear(time_emb_dim * 2, time_emb_dim)
55         )
56
57         # Input layer
58         self.input_layer = nn.Linear(input_dim, hidden_dims[0])
59
60         # Time embedding projection for each layer
61         self.time_projections = nn.ModuleList([
62             nn.Linear(time_emb_dim, h_dim) for h_dim in hidden_dims
63         ])
64
65         # Hidden layers
66         self.hidden_layers = nn.ModuleList()
67         for i in range(len(hidden_dims) - 1):
68             self.hidden_layers.append(
69                 nn.Sequential(
70                     nn.Linear(hidden_dims[i], hidden_dims[i + 1]),
71                     nn.GroupNorm(8, hidden_dims[i + 1]),
72                     nn.GELU()
73                 )
74             )
75
76         # Output layer

```

```

77         self.output_layer = nn.Linear(hidden_dims[-1], input_dim)
78
79         self.act = nn.GELU()
80         self.norm_layers = nn.ModuleList([
81             nn.GroupNorm(8, h_dim) for h_dim in hidden_dims
82         ])
83
84     def forward(self, x, t):
85         # Time embedding
86         t_emb = self.time_mlp(t)
87
88         # Input
89         h = self.input_layer(x)
90         h = self.norm_layers[0](h)
91         h = self.act(h)
92         h = h + self.time_projections[0](t_emb)
93
94         # Hidden layers
95         for i, layer in enumerate(self.hidden_layers):
96             h = layer(h)
97             h = h + self.time_projections[i + 1](t_emb)
98
99         # Output
100        return self.output_layer(h)
101
102
103    class DiffusionScheduler:
104        def __init__(self, num_timesteps=1000, beta_start=1e-4, beta_end=0.02,
105                      device='cpu'):
106            self.num_timesteps = num_timesteps
107            self.device = device
108
109            # Linear schedule
110            self.betas = torch.linspace(beta_start, beta_end, num_timesteps,
111                                         device=device)
112
113            self.alphas = 1.0 - self.betas
114            self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
115            self.alphas_cumprod_prev = torch.cat([
116                torch.tensor([1.0], device=device),
117                self.alphas_cumprod[:-1]
118            ])
119
120            # Pre-compute values for q(x_t | x_0)
121            self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
122            self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - self.
123                                                            alphas_cumprod)
124
125            # Pre-compute values for posterior q(x_{t-1} | x_t, x_0)
126            self.posterior_variance = (
127                self.betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.
128                                                                alphas_cumprod)
129            )
130            self.sqrt_recip_alphas = torch.sqrt(1.0 / self.alphas)

```



```

127 def q_sample(self, x_0, t, noise=None):
128     """Forward diffusion:  $q(x_t | x_0)$ """
129     if noise is None:
130         noise = torch.randn_like(x_0)
131
132     sqrt_alphas_cumprod_t = self.sqrt_alphas_cumprod[t][:, None]
133     sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t]
134                                    [:, None]
135     return sqrt_alphas_cumprod_t * x_0 + sqrt_one_minus_alphas_cumprod_t *
136                                     noise
137
138 def p_sample(self, model, x_t, t):
139     """Reverse diffusion:  $p(x_{t-1} | x_t)$ """
140     betas_t = self.betas[t][:, None]
141     sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t]
142                                    [:, None]
143     sqrt_recip_alphas_t = self.sqrt_recip_alphas[t][:, None]
144
145     # Predict noise
146     predicted_noise = model(x_t, t.float())
147
148     # Compute mean
149     model_mean = sqrt_recip_alphas_t * (
150         x_t - betas_t * predicted_noise / sqrt_one_minus_alphas_cumprod_t
151     )
152
153     # Add noise (except for t=0)
154     if (t > 0).any():
155         posterior_variance_t = self.posterior_variance[t][:, None]
156         noise = torch.randn_like(x_t)
157         # Only add noise where t > 0
158         mask = (t > 0).float()[:, None]
159         return model_mean + mask * torch.sqrt(posterior_variance_t) *
160                                     noise
161     else:
162         return model_mean
163
164 class Prob2cAnalysis:
165     def __init__(self,
166                 output_dir='Homework 3/Code/Data',
167                 figure_dir='Homework 3/Latex/Figures',
168                 device=None,
169                 seed=25):
170         self.output_dir = output_dir
171         self.figure_dir = figure_dir
172         self.seed = seed
173
174         os.makedirs(self.output_dir, exist_ok=True)
175         os.makedirs(self.figure_dir, exist_ok=True)
176
177         # Set device
178         if device is None:

```

```

177         self.device = torch.device('cuda' if torch.cuda.is_available()
178                                     else 'cpu')
179     else:
180         self.device = device
181
182     self.data = None
183     self.targets = None
184     self.scaler = None
185     self.diffusion_results = None
186     self.generated_samples = None
187     self.training_history = None
188
189     def _set_seed(self):
190         np.random.seed(self.seed)
191         torch.manual_seed(self.seed)
192         if torch.cuda.is_available():
193             torch.cuda.manual_seed_all(self.seed)
194
195     def process_data(self, verbose=False):
196         digits = load_digits()
197         X = digits.data
198         y = digits.target
199
200         # Scale to [0, 1] then to [-1, 1] for diffusion
201         self.scaler = MinMaxScaler(feature_range=(-1, 1))
202         X_scaled = self.scaler.fit_transform(X)
203
204         self.data = X
205         self.data_scaled = X_scaled
206         self.targets = y
207
208         if verbose:
209             print(f"\nDataset shape: {X.shape}")
210             print(f"Number of classes: {len(np.unique(y))}")
211             print(f"Original pixel range: [{X.min():.1f}, {X.max():.1f}]")
212             print(f"Scaled pixel range: [{X_scaled.min():.3f}, {X_scaled.max():.3f}]")
213             print(f"Device: {self.device}")
214
215         return X, y
216
217     def fit_diffusion(self,
218                       num_timesteps=500,
219                       hidden_dims=[256, 512, 256],
220                       time_emb_dim=64,
221                       lr=1e-3,
222                       batch_size=128,
223                       n_epochs=200,
224                       beta_start=1e-4,
225                       beta_end=0.02,
226                       verbose=False):
227         self._set_seed()
228
229         if self.data is None:

```

```

229         self.process_data()
230
231     input_dim = self.data.shape[1]  # 64 for 8x8 images
232
233     # Build model
234     model = DenoisingMLP(
235         input_dim=input_dim,
236         hidden_dims=hidden_dims,
237         time_emb_dim=time_emb_dim
238     ).to(self.device)
239
240     # Build scheduler
241     scheduler = DiffusionScheduler(
242         num_timesteps=num_timesteps,
243         beta_start=beta_start,
244         beta_end=beta_end,
245         device=self.device
246     )
247
248     # Optimizer
249     optimizer = optim.Adam(model.parameters(), lr=lr)
250
251     # Loss function
252     criterion = nn.MSELoss()
253
254     # Prepare data
255     X_tensor = torch.FloatTensor(self.data_scaled).to(self.device)
256     dataset = TensorDataset(X_tensor)
257     dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
258                             drop_last=True)
259
260     # Training history
261     history = {
262         'epoch': [],
263         'loss': []
264     }
265
266     if verbose:
267         print(f"\nTraining Diffusion Model...")
268         print(f"  Timesteps: {num_timesteps}")
269         print(f"  Hidden dims: {hidden_dims}")
270         print(f"  Learning rate: {lr}")
271         print(f"  Epochs: {n_epochs}, Batch size: {batch_size}")
272
273     # Training loop
274     model.train()
275     for epoch in range(n_epochs):
276         epoch_losses = []
277
278         for batch_data in dataloader:
279             x_0 = batch_data[0]
280             current_batch_size = x_0.size(0)
281
282             # Sample random timesteps

```

```

282         t = torch.randint(0, num_timesteps, (current_batch_size,),
283                               device=self.device)
284
285         # Sample noise
286         noise = torch.randn_like(x_0)
287
288         # Forward diffusion
289         x_t = scheduler.q_sample(x_0, t, noise)
290
291         # Predict noise
292         optimizer.zero_grad()
293         predicted_noise = model(x_t, t.float())
294
295         # Loss
296         loss = criterion(predicted_noise, noise)
297         loss.backward()
298         optimizer.step()
299
300         epoch_losses.append(loss.item())
301
302         avg_loss = np.mean(epoch_losses)
303         history['epoch'].append(epoch + 1)
304         history['loss'].append(avg_loss)
305
306         if verbose and (epoch + 1) % 20 == 0:
307             print(f"Epoch [{epoch+1:4d}/{n_epochs}] | Loss: {avg_loss:.6f}"
308                   ")
309
310         # Store results
311         self.diffusion_results = {
312             'model': model,
313             'scheduler': scheduler,
314             'history': history,
315             'config': {
316                 'num_timesteps': num_timesteps,
317                 'hidden_dims': hidden_dims,
318                 'time_emb_dim': time_emb_dim,
319                 'lr': lr,
320                 'batch_size': batch_size,
321                 'n_epochs': n_epochs
322             }
323         }
324
325         self.training_history = history
326
327         if verbose:
328             self._plot_training_history(history)
329
330         return self.diffusion_results
331
332     def tune_hyperparameters(self,
333                             num_timesteps_values=[200, 500, 1000],
334                             lr_values=[1e-4, 5e-4, 1e-3],
335                             hidden_configs=[

```

```

334         [128, 256, 128],
335         [256, 512, 256],
336         [256, 512, 512, 256]
337     ],
338     n_epochs=150,
339     n_eval_samples=500,
340     verbose=False):
341 if self.data is None:
342     self.process_data()
343
344 results = []
345 total_configs = len(num_timesteps_values) * len(lr_values) * len(
346                                     hidden_configs)
347
348 current_config = 0
349
350 if verbose:
351     print("Hyperparameter Tuning for Diffusion Model")
352     print(f"Total configurations: {total_configs}")
353
354 for num_timesteps in num_timesteps_values:
355     for lr in lr_values:
356         for hidden_dims in hidden_configs:
357             current_config += 1
358             if verbose:
359                 print(f"\n[{current_config}/{total_configs}] "
360                       f"T={num_timesteps}, lr={lr}, hidden={
361                                     hidden_dims
362                                     }")
363
364             # Train model
365             self.fit_diffusion(
366                 num_timesteps=num_timesteps,
367                 hidden_dims=hidden_dims,
368                 lr=lr,
369                 n_epochs=n_epochs,
370                 verbose=False
371             )
372
373             # Generate and evaluate samples
374             samples = self.generate_samples(n_samples=n_eval_samples,
375                                           verbose=False)
376             metrics = self.evaluate_samples(samples, verbose=False)
377
378             # Store results
379             result = {
380                 'num_timesteps': num_timesteps,
381                 'lr': lr,
382                 'hidden_dims': str(hidden_dims),
383                 'final_loss': self.training_history['loss'][-1],
384                 'mean_diff': abs(metrics['generated_mean'] - metrics['
385                                     original_mean']
386                                 ),
387                 'std_diff': abs(metrics['generated_std'] - metrics['
388                                     original_std'])

```

```

381         'diversity_ratio': metrics.get('diversity_ratio', 0)
382     }
383     results.append(result)
384
385     if verbose:
386         print(f"    Loss: {result['final_loss']:.6f} | "
387               f"Mean Diff: {result['mean_diff']:.4f} | "
388               f"Std Diff: {result['std_diff']:.4f}")
389
390 results_df = pd.DataFrame(results)
391
392 if verbose:
393     print("Tuning Results Summary:")
394     print(results_df.to_string(index=False))
395
396     # Find best configuration
397     best_idx = results_df['mean_diff'].idxmin()
398     best_config = results_df.iloc[best_idx]
399     print("Best Configuration (by Mean Diff):")
400     print(f"    Timesteps: {best_config['num_timesteps']}")
401     print(f"    Learning Rate: {best_config['lr']}")
402     print(f"    Hidden Dims: {best_config['hidden_dims']}")
403     print(f"    Mean Diff: {best_config['mean_diff']:.4f}")
404     print(f"    Std Diff: {best_config['std_diff']:.4f}")
405
406     self._plot_tuning_results(results_df)
407
408     return results_df
409
410 def _plot_tuning_results(self, results_df):
411     fig, axes = plt.subplots(1, 3, figsize=(18, 5))
412
413     # Mean difference heatmap (timesteps x lr)
414     pivot_mean = results_df.pivot_table(
415         values='mean_diff',
416         index='num_timesteps',
417         columns='lr',
418         aggfunc='mean'
419     )
420     sns.heatmap(pivot_mean, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[
421         0])
422     axes[0].set_title('Mean Difference (Lower is Better)')
423     axes[0].set_xlabel('Learning Rate')
424     axes[0].set_ylabel('Timesteps')
425
426     # Std difference heatmap
427     pivot_std = results_df.pivot_table(
428         values='std_diff',
429         index='num_timesteps',
430         columns='lr',
431         aggfunc='mean'

```

```

432     sns.heatmap(pivot_std, annot=True, fmt='.4f', cmap='YlOrRd', ax=axes[1
433                  ])
434     axes[1].set_title('Std Difference (Lower is Better)')
435     axes[1].set_xlabel('Learning Rate')
436     axes[1].set_ylabel('Timesteps')
437
438     # Final loss by architecture
439     arch_loss = results_df.groupby('hidden_dims')['final_loss'].mean()
440     colors = plt.cm.Set2(np.linspace(0, 1, len(arch_loss)))
441     axes[2].bar(range(len(arch_loss)), arch_loss.values,
442                color=colors, edgecolor='black')
443     axes[2].set_xticks(range(len(arch_loss)))
444     axes[2].set_xticklabels([s.replace(',', '\n') for s in arch_loss.
445                             index],
446                             fontsize=8, rotation=0)
447     axes[2].set_ylabel('Final Loss')
448     axes[2].set_title('Final Loss by Architecture (Lower is Better)')
449     axes[2].grid(axis='y', alpha=0.3)
450
451     plt.tight_layout()
452     plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_tuning.png'),
453                dpi=150)
454
455     plt.show()
456
457     def _plot_training_history(self, history):
458         fig, ax = plt.subplots(figsize=(10, 5))
459
460         epochs = history['epoch']
461         losses = history['loss']
462
463         ax.plot(epochs, losses, 'b-', linewidth=2)
464         ax.set_xlabel('Epoch')
465         ax.set_ylabel('Loss (MSE)')
466         ax.set_title('Diffusion Model Training Loss')
467         ax.grid(True, alpha=0.3)
468
469         plt.tight_layout()
470         plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_training.png'),
471                    , dpi=150)
472
473         plt.show()
474
475     def generate_samples(self, n_samples=100, verbose=False):
476         model = self.diffusion_results['model']
477         scheduler = self.diffusion_results['scheduler']
478         num_timesteps = self.diffusion_results['config']['num_timesteps']
479         input_dim = self.data.shape[1]
480
481         model.eval()
482
483         with torch.no_grad():
484             # Start from pure noise
485             x = torch.randn(n_samples, input_dim, device=self.device)
486
487             # Reverse diffusion

```

```

482         for t in reversed(range(num_timesteps)):
483             t_batch = torch.full((n_samples,), t, device=self.device,
                                   dtype=torch.long)
484             x = scheduler.p_sample(model, x, t_batch)
485
486             generated_scaled = x.cpu().numpy()
487
488             # Inverse transform to original scale
489             generated = self.scaler.inverse_transform(generated_scaled)
490
491             # Clip to valid range [0, 16]
492             generated = np.clip(generated, 0, 16)
493
494             self.generated_samples = {
495                 'samples': generated,
496                 'samples_scaled': generated_scaled,
497                 'n_samples': n_samples
498             }
499
500             if verbose:
501                 print(f"\nGenerated {n_samples} samples from Diffusion Model")
502                 print(f"Generated samples shape: {generated.shape}")
503                 print(f"Generated samples range: [{generated.min():.2f}, {
504                                                             generated.max():.2f}]")
505
506             return generated
507
508     def evaluate_samples(self, generated_samples=None, verbose=False):
509         """Evaluate quality of generated samples."""
510         if generated_samples is None:
511             generated_samples = self.generated_samples['samples']
512
513         metrics = {}
514
515         # Sample statistics comparison
516         metrics['original_mean'] = self.data.mean()
517         metrics['original_std'] = self.data.std()
518         metrics['generated_mean'] = generated_samples.mean()
519         metrics['generated_std'] = generated_samples.std()
520
521         # Sparsity check
522         threshold = 0.5
523         metrics['original_sparsity'] = (self.data < threshold).mean()
524         metrics['generated_sparsity'] = (generated_samples < threshold).mean()
525
526         # Diversity check
527         try:
528             gen_pca = PCA(n_components=10).fit_transform(generated_samples)
529             pairwise_dist = cdist(gen_pca, gen_pca, metric='euclidean')
530             np.fill_diagonal(pairwise_dist, np.inf)
531             metrics['avg_nn_distance'] = pairwise_dist.min(axis=1).mean()
532
533             orig_pca = PCA(n_components=10).fit_transform(self.data)
534             orig_pairwise = cdist(orig_pca, orig_pca, metric='euclidean')

```



```

534         np.fill_diagonal(orig_pairwise, np.inf)
535         metrics['orig_avg_nn_distance'] = orig_pairwise.min(axis=1).mean()
536
537         metrics['diversity_ratio'] = metrics['avg_nn_distance'] / metrics[
538             'orig_avg_nn_distance']
539
540     except Exception as e:
541         metrics['diversity_ratio'] = None
542
543     if verbose:
544         self._print_evaluation_metrics(metrics)
545
546     return metrics
547
548 def _print_evaluation_metrics(self, metrics):
549     """Print evaluation metrics."""
550     print("\n--- Diffusion Model Evaluation Metrics ---")
551     print(f"Mean (Original):      {metrics['original_mean']:.4f}")
552     print(f"Mean (Generated):      {metrics['generated_mean']:.4f}")
553     print(f"Std (Original):          {metrics['original_std']:.4f}")
554     print(f"Std (Generated):         {metrics['generated_std']:.4f}")
555     print(f"Sparsity (Original): {metrics['original_sparsity']:.4f}")
556     print(f"Sparsity (Generated): {metrics['generated_sparsity']:.4f}")
557     if metrics.get('diversity_ratio') is not None:
558         print(f"Diversity Ratio:      {metrics['diversity_ratio']:.4f}")
559
560 def visualize_generated_samples(self, generated_samples=None, n_display=25
561                                ):
562     if generated_samples is None:
563         generated_samples = self.generated_samples['samples']
564
565     grid_size = int(np.sqrt(n_display))
566     fig, axes = plt.subplots(grid_size, grid_size, figsize=(10, 10))
567     axes = axes.flatten()
568
569     for i in range(min(n_display, len(generated_samples))):
570         axes[i].imshow(generated_samples[i].reshape(8, 8), cmap='gray')
571         axes[i].axis('off')
572
573     plt.suptitle('Generated Digits from Diffusion Model', fontsize=14)
574     plt.tight_layout()
575     plt.savefig(os.path.join(self.figure_dir, '
576                                     2c_generated_samples_diffusion.
577                                     png'), dpi=150)
578
579     plt.show()
580
581 def compare_distributions(self, generated_samples=None):
582     if generated_samples is None:
583         generated_samples = self.generated_samples['samples']
584
585     fig, axes = plt.subplots(1, 3, figsize=(18, 5))
586
587     # Pixel intensity distribution
588     axes[0].hist(self.data.flatten(), bins=50, alpha=0.6, label='Original'

```

```

583         color='blue', density=True)
584 axes[0].hist(generated_samples.flatten(), bins=50, alpha=0.6, label='
Generated',
585             color='red', density=True)
586 axes[0].set_xlabel('Pixel Intensity')
587 axes[0].set_ylabel('Density')
588 axes[0].set_title('Pixel Intensity Distribution')
589 axes[0].legend()
590 axes[0].grid(alpha=0.3)
591
592 # Sparsity comparison
593 sparsity_orig = (self.data < 0.5).sum(axis=1).mean()
594 sparsity_gen = (generated_samples < 0.5).sum(axis=1).mean()
595
596 categories = ['Original', 'Generated']
597 sparsities = [sparsity_orig, sparsity_gen]
598 axes[1].bar(categories, sparsities, color=['blue', 'red'], alpha=0.7,
599           edgecolor='black')
600 axes[1].set_ylabel('Average # of Near-Zero Pixels')
601 axes[1].set_title('Sparsity Comparison')
602 axes[1].grid(axis='y', alpha=0.3)
603
604 # PCA projection comparison
605 pca = PCA(n_components=2)
606 orig_pca = pca.fit_transform(self.data)
607 gen_pca = pca.transform(generated_samples)
608
609 axes[2].scatter(orig_pca[:, 0], orig_pca[:, 1], alpha=0.3, label='
Original', s=10)
610 axes[2].scatter(gen_pca[:, 0], gen_pca[:, 1], alpha=0.5, label='
Generated', s=10)
611 axes[2].set_xlabel('PC1')
612 axes[2].set_ylabel('PC2')
613 axes[2].set_title('PCA Projection Comparison')
614 axes[2].legend()
615 axes[2].grid(alpha=0.3)
616
617 plt.tight_layout()
618 plt.savefig(os.path.join(self.figure_dir, '
2c_distribution_comparison_diffusion
.png'), dpi=150)
619
620 plt.show()
621
622 def visualize_diffusion_process(self, n_steps=10):
623     if self.diffusion_results is None:
624         raise ValueError("Diffusion model not trained. Call fit_diffusion
() first.")
625
626     model = self.diffusion_results['model']
627     scheduler = self.diffusion_results['scheduler']
628     num_timesteps = self.diffusion_results['config']['num_timesteps']
629     input_dim = self.data.shape[1]
630
631     model.eval()

```

```

630
631     # Choose timesteps to visualize
632     vis_timesteps = np.linspace(num_timesteps - 1, 0, n_steps, dtype=int)
633
634     with torch.no_grad():
635         # Start from pure noise
636         x = torch.randn(1, input_dim, device=self.device)
637
638         samples_at_timesteps = []
639
640         # Reverse diffusion
641         for t in reversed(range(num_timesteps)):
642             t_batch = torch.full((1,), t, device=self.device, dtype=torch.
643                                   long)
644             x = scheduler.p_sample(model, x, t_batch)
645
646             if t in vis_timesteps:
647                 sample = x.cpu().numpy()
648                 sample = self.scaler.inverse_transform(sample)
649                 sample = np.clip(sample, 0, 16)
650                 samples_at_timesteps.append((t, sample.reshape(8, 8)))
651
652         # Sort by timestep (descending)
653         samples_at_timesteps.sort(key=lambda x: x[0], reverse=True)
654
655         # Plot
656         fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2.5))
657         for i, (t, img) in enumerate(samples_at_timesteps):
658             axes[i].imshow(img, cmap='gray')
659             axes[i].axis('off')
660             axes[i].set_title(f't={t}', fontsize=10)
661
662         plt.suptitle('Reverse Diffusion Process (Noise → Image)', fontsize=14)
663         plt.tight_layout()
664         plt.savefig(os.path.join(self.figure_dir, '2c_diffusion_process.png'),
665                     dpi=150)
666
667         plt.show()
668
669     def compare_with_real_samples(self, n_display=10):
670         if self.generated_samples is None:
671             self.generate_samples(n_samples=n_display)
672
673         generated = self.generated_samples['samples'][:n_display]
674         real_indices = np.random.choice(len(self.data), n_display, replace=
675                                         False)
676         real = self.data[real_indices]
677
678         fig, axes = plt.subplots(2, n_display, figsize=(2 * n_display, 4))
679
680         for i in range(n_display):
681             axes[0, i].imshow(real[i].reshape(8, 8), cmap='gray')
682             axes[0, i].axis('off')
683             if i == 0:
684                 axes[0, i].set_ylabel('Real', fontsize=12)

```

```
681         axes[1, i].imshow(generated[i].reshape(8, 8), cmap='gray')
682         axes[1, i].axis('off')
683         if i == 0:
684             axes[1, i].set_ylabel('Generated', fontsize=12)
685
686     plt.suptitle('Real vs Generated Samples (Diffusion)', fontsize=14)
687     plt.tight_layout()
688     plt.savefig(os.path.join(self.figure_dir, '
689                                     2c_real_vs_generated_diffusion.
                                     png'), dpi=150)
690     plt.show()
```