Summary of Internship Work at genpf

Bazil Ahmad

December 29, 2025

Abstract

This report summarizes my internship work at genpf, focusing on a regime similarity project. The primary objective was to identify and visualize historical market regimes using advanced dimensionality reduction and clustering techniques, including PCA, t-SNE, DBSCAN, and K-Means. The project aimed to enhance interpretability, reduce feature collinearity, and eliminate thin regimes through robust clustering and visualization approaches. Users can input eight economic parameters, and the program identifies historical periods that align with the new data point.

Key contributions include the development of a Python-based workflow to process and analyze time-series financial data, the implementation of t-SNE for non-linear dimensionality reduction, and the integration of DBSCAN and K-Means for regime clustering. Parameters such as perplexity, epsilon, and minimum samples were tuned to optimize cluster quality and minimize noise. Analytical findings highlight the impact of these parameters on clustering behavior and regime selection, with recommendations for balancing interpretability and computational efficiency.

Challenges included handling small regime clusters, ensuring proper scaling of user-defined inputs, and managing the computational demands of t-SNE on large datasets. The report concludes by summarizing key findings, outlining challenges, and presenting actionable next steps, such as exploring distributed computing and alternative clustering methods to enhance efficiency and accuracy. Metrics used to evaluate performance include the silhouette score, which assesses cluster cohesion and separation, the number of clusters formed, the size of the smallest "thin" regime and checks for collinearity in the data.

Contents

1 Preliminary Considerations and Design

During my internship at genpf, I worked on a *regime similarity project*. The main goal was to identify and visualize market regimes through various clustering and dimensionality-reduction techniques. Below are some key aspects of my initial design and thought process:

- **Motivation:** We wanted to detect periods in the market that share similar characteristics (a.k.a. "regimes"), typically defined by macroeconomic and financial indicators.
- **Data Inputs:** The input data consisted of multiple time-series features such as interest rate spreads, unemployment rate, CPI, GDP growth, etc.
- **Problem of Thin Regimes:** Market regimes that are smaller than 2-3 weeks should not be selected. This problem occured with PCA and was first hard coded out. Hard coded function can be used for t-SNE as well.
- **Approach:** We initially experimented with simpler dimensionality reduction (like PCA) and hard-coded logic to handle small clusters. Later, we incorporated t-SNE for its ability to represent high-dimensional data in lower-dimensional space with local structure preserved as well as DBSCAN and K-Means for further clustering techniques.

2 t-SNE: A Process Flow

2.1 Introduction

t-SNE (t-Distributed Stochastic Neighbor Embedding) is a non-linear dimensionality reduction method introduced by Laurens van der Maaten and Geoffrey Hinton in 2008. It is primarily used to visualize high-dimensional data by mapping it into a lower-dimensional (usually 2D or 3D) space. One of the main goals of t-SNE is to preserve the *local* structure of the data, making points that are close in the high-dimensional space also lie close in the reduced space.

In our code, the main entry point for running t-SNE is the `apply_tsne()` function, which eventually invokes `TSNE(...).fit_transform(...)` from `sklearn.manifold`. By passing in feature data, we let scikit-learn automatically handle several of the steps described below.

2.2 The Process Flow of t-SNE

2.2.1 Step 1: Understanding the Data and Its Local Neighborhoods

Suppose we have a dataset $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ in a high-dimensional space. The core idea of t-SNE is to preserve local relationships among points. Therefore, t-SNE first quantifies how **similar** or **close** any two points are.

What is a Gaussian Kernel? In many machine learning and statistics tasks, a *Gaussian kernel* (or *RBF kernel*) is employed to measure similarity between two points in a continuous and smoothly decaying way. Given two points $\mathbf{x}$ and $\mathbf{y}$ in an n-dimensional space, the Gaussian kernel defines a similarity as

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right),$$

where $\|\mathbf{x}-\mathbf{y}\|$ is the Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$. The parameter σ (sometimes called the bandwidth) controls how quickly similarity drops off with distance. A small σ leads to a rapid decay in similarity as distance grows; a large σ makes the decay more gradual.

We can see this **Gaussian kernel** as one step in a *process*: (1) measure distances, (2) apply the exponential decay, (3) get a smooth similarity measure. This approach becomes the cornerstone of how t-SNE defines local neighborhoods.

In the code: The Gaussian kernel-based local similarity is handled *internally* by scikit-learn's `TSNE()` class. We simply call `tsne.fit_transform(feat_data_df_scaled)` in the `apply_tsne()` function, and scikit-learn computes pairwise distances under the hood.

2.2.2 Step 2: Computing High-Dimensional Similarities

Once we have the Gaussian kernel, t-SNE transforms it into *conditional probabilities* $p_{j|i}$. For each point $\mathbf{x}_i$, the distribution over all other points $\mathbf{x}_j$ is defined so that nearer points $\mathbf{x}_j$ have higher probability under $p_{j|i}$. Specifically,

$$p_{j|i} = \frac{\exp\left(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2\right)}.$$

This probability can be viewed as a normalized Gaussian kernel centered at $\mathbf{x}_i$. The σ_i is chosen (often via *perplexity* search) so that the perplexity matches a user-specified hyperparameter. Perplexity intuitively sets the "effective number of neighbors" each point has.

In the code: The call to `TSNE(perplexity=30.0)` determines the perplexity used to optimize these Gaussian (RBF) bandwidths. You can change `perplexity` in `apply_tsne()` to control how these probabilities are computed.

2.2.3 Step 3: Defining the Low-Dimensional Counterpart

The final visualization occurs in a low-dimensional space (often 2D), where each original point $\mathbf{x}_i$ is mapped to a new point $\mathbf{y}_i$. t-SNE also measures pairwise similarities among these $\{\mathbf{y}_i\}$, but rather than use a Gaussian kernel in the low-dimensional space, it uses a heavy-tailed *Student's t-distribution* (essentially a Cauchy distribution). Mathematically,

$$q_{ij} = \frac{\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2\right)^{-1}}.$$

This heavy-tailed nature reduces the "crowding problem," letting points form clearer clusters in 2D or 3D.

In the code: This step is also encapsulated inside `TSNE.fit_transform(...)`. In scikit-learn's implementation, you will see references to the "embedded_kl_distance," which indicates the Student's t-distribution is being used in the low-dimensional space.

2.2.4 Step 4: Matching High- and Low-Dimensional Distributions

To ensure that the local neighborhoods in the high-dimensional space match those in the low-dimensional space, t-SNE minimizes the Kullback–Leibler (KL) divergence between the two distributions P (composed of $p_{j|i}$) and Q (composed of q_{ij}):

$$\mathcal{C} = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}.$$

Minimizing KL divergence aligns q_{ij} with $p_{j|i}$, preserving local neighborhoods.

In the code: This KL divergence minimization happens behind the scenes when we call `tsne.fit_transform(feat_data_df_scaled)`. The parameter `early_exaggeration` (not shown in the snippet) often helps speed up the alignment in early iterations.

2.2.5 Step 5: Gradient Descent and Iteration

Finally, t-SNE updates the low-dimensional points $\mathbf{y}_i$ iteratively using gradient descent on the KL divergence $\mathcal{C}$. This process repeats for a fixed number of iterations (or until convergence). Over time, points that are neighbors in the high-dimensional space remain or become neighbors in the 2D/3D map, and points that are far apart separate accordingly.

In the code: Scikit-learn's `TSNE` class manages these iterations internally. If you look at the source code, you'll see a loop that updates each $\mathbf{y}_i$ via gradient descent until a specified `n_iter` is reached.

2.3 Putting It All Together

Viewed as a process flow, t-SNE goes like this:

1. **Local Similarity (Gaussian Kernel)**: Use a Gaussian kernel to measure how close points are in the original space. Convert distances to similarity scores.
   In the code: Done internally by scikit-learn's TSNE.

2. **Probabilities and Perplexity**: Convert those similarities into probabilities $p_{j|i}$, guided by a *perplexity* hyperparameter that sets the effective neighborhood size.
   In the code: Controlled by `perplexity` in `apply_tsne()`.

3. **Heavy-Tailed Distribution in Low-D Space**: Place points in 2D (or 3D) and define a heavy-tailed distribution q_{ij} (the t-distribution) so that small distances in 2D reflect high similarity.
   In the code: Automatically used by `TSNE`, as shown in `tsne.fit_transform(...)`.

4. **Match Distributions**: Minimize the KL divergence between P and Q via gradient descent, adjusting the 2D coordinates $\mathbf{y}_i$.
   In the code: Optimized under the hood by scikit-learn.

5. **Visual Output**: Once convergence is reached, the 2D scatter plot typically shows clusters or groupings that mirror local structure in the original data.
   In the code: The data in `tsne_results` is then used for clustering or plotting (see `cluster_tsne_data()` and `regime_similarity_chart_tsne()`).

2.4 Practical Tips

2.4.1 Hyperparameters

- *Perplexity*: This parameter primarily controls the balance between capturing local versus global structure in t-SNE. Conceptually, perplexity can be viewed as a guess about the number of effective nearest neighbors for each point. In practice, perplexity values typically range from 5 to 50. Lower perplexity values focus more on local neighborhoods, potentially highlighting finer-grained clusters; higher perplexity values emphasize broader global structure, but may smooth out smaller clusters.

- *Learning Rate*: Determines the step size used by the gradient descent optimizer in t-SNE. A small learning rate may converge more slowly or get stuck in poor local minima, while a large learning rate can cause divergence or spurious patterns in the embedding. Proper tuning of the learning rate is crucial for achieving stable, meaningful embeddings.

- *Early Exaggeration*: Temporarily amplifies the conditional probabilities $p_{j|i}$ in the t-SNE cost function at the start of training. This step encourages points that are close in the high-dimensional space to move together more aggressively, helping the algorithm form compact clusters early on. After a certain number of iterations, the early exaggeration factor is reduced to 1, allowing the clusters to stabilize at a scale representative of the original distances.

- *Epsilon* (ε): In the context of the DBSCAN clustering algorithm, ε defines the maximum distance between two samples for them to be considered as part of the same neighborhood. If the distance between two points is less than or equal to ε, they are considered density-connected (neighbors in the same cluster). Proper selection of ε is critical: if it is too small, many points end up as outliers; if it is too large, distinct clusters may merge.

- *Min_Samples*: Also a DBSCAN parameter, *Min_Samples* is the minimum number of points required to form a dense region (i.e., a cluster). A higher value increases the strictness of cluster definition, potentially leaving more points labeled as outliers. Conversely, a smaller value may lead to over-fragmentation of the data into many small clusters or a single, overly large cluster.

2.4.2 Computational Complexity

A naive implementation of t-SNE can be $O(N^2)$ in both time and space. However, more efficient methods (e.g., Barnes–Hut t-SNE) approximate distance calculations to handle larger datasets.

2.4.3 Interpretation Cautions

While t-SNE is powerful for visualization, interpret the resulting clusters carefully. Different hyperparameter choices (perplexity, learning rate, etc.) can give very different-looking maps.

2.5 Conclusion

We can now see how t-SNE forms the basis for local similarity measurements in the high-dimensional space. Then, through a series of well-defined steps — culminating in a gradient-based optimization of the KL divergence — t-SNE organizes points in a low-dimensional layout that visually reflects local neighborhoods. Indeed, it is a *flow* or *process*, each stage informing and supporting the next. In our code, most of these stages happen inside scikit-learn's `TSNE` class, configured by parameters passed through `apply_tsne()`.

3 Additions to the Code

As part of the project, I introduced several new functionalities to the existing codebase:

1. **t-SNE Application:** Implemented a function `apply_tsne()` to handle feature scaling, perplexity tuning, and random state control.
2. **DBSCAN Clustering:** Created the function `cluster_tsne_data()` to cluster t-SNE outputs using DBSCAN, allowing for the specification of `eps` and `min_samples` to handle outliers and small clusters.
3. **K-Means Clustering:** Added a second clustering option, `cluster_tsne_data_kmeans()`, for comparative purposes and easier control over the number of clusters.
4. **Regime Similarity Charting:** Extended the code to generate line plots showing how "similar" a given period is to a reference regime (using both PCA-based and t-SNE-based methods).
5. **Nearest-Neighbor Based "Close" Identification:** Instead of simply taking the *mode* of a cluster to define what is "close," we introduced functions to:
   - Find which cluster a *user-defined regime dictionary* belongs to by computing the nearest row in the scaled feature space (`find_regime_cluster`).
   - Label all observations in that cluster as "close = 1" using `label_close_rows`.

   This ensures that "close" is determined by the similarity of the new or hypothetical regime parameters, rather than by defaulting to the most common cluster in the dataset.

4 Documentation of Code

4.1 Code Overview

Below is an updated version of the Python code used in this project. It contains the main functions responsible for data transformation, clustering, and visualization. Notice that **Step 1 (Gaussian kernel)** and **Step 2 (Probabilities)** are handled implicitly under the hood by `sklearn.manifold.TSNE`, while steps 3–5 relate to the low-dimensional embedding, matching distributions, and the gradient descent procedure. We also show how we incorporate the new `find_regime_cluster` and `label_close_rows` methods to identify a user-defined regime's nearest cluster and mark it as "close":

```
# Cleaned-up version of the original code
import pandas as pd
import numpy as np
from sklearn.manifold import TSNE
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.dates as mdates
import logging
import math
import dictdiffer
from typing import Callable, List

# Pandas display options for debugging and data exploration
pd.options.display.max_colwidth = 200
pd.options.display.max_columns = None
pd.options.display.width = 200
pd.options.display.max_rows = 100

# Global variables for parameters
BACKTEST_CHART_DATE_RANGE = ('2003-01-01', '2024-09-10')
MAX_DISTANCE_FOR_SIMILAR_OBS = 0.50

def regime_similarity_chart_tsne(working_df: pd.DataFrame, title: str) -> plt.Axes:
    """
    Generate a t-SNE-based regime similarity chart.

    :param working_df: DataFrame with ['Date', 'cluster_label', 'close'] and a
    ↪  DatetimeIndex.
    :param title: Title of the chart.
    :return: Matplotlib Axes object.
    """
    plt.clf()
    sns.set_style('ticks')
    g = sns.lineplot(data=working_df.reset_index(), x='Date', y='cluster_label',
                     color='darkgreen', linewidth=1.0)
    # Fill area where 'close' is True
```

```
    g.fill_between(x=working_df.index, y1=g.get_ylim()[0], y2=g.get_ylim()[1],
                   where=working_df['close'], facecolor='teal', alpha=0.3)
    g.set_title(title, fontsize=14)
    g.figure.set_figwidth(12)
    g.figure.set_figheight(6)
    return g

def apply_tsne(feat_data_df: pd.DataFrame, n_components=3, perplexity=30.0):
    """
    Apply t-SNE to reduce feature dimensionality. Returns:
      - scaled DataFrame (for nearest-neighbor computations),
      - t-SNE-embedded DataFrame,
      - the trained StandardScaler,
      - the TSNE instance itself (for any introspection).
    """
    feat_data_df = feat_data_df.dropna()
    if perplexity >= len(feat_data_df):
        logging.warning(f'Perplexity ({perplexity}) adjusted to {len(feat_data_df) - 1}')
        perplexity = max(1, len(feat_data_df) - 1)

    scaler = StandardScaler()
    feat_data_df_scaled = pd.DataFrame(
        scaler.fit_transform(feat_data_df),
        index=feat_data_df.index,
        columns=feat_data_df.columns
    )

    tsne = TSNE(n_components=n_components, perplexity=perplexity, random_state=42)
    tsne_results = tsne.fit_transform(feat_data_df_scaled)

    tsne_df = pd.DataFrame(tsne_results, index=feat_data_df.index,
                           columns=[f'tsne_{i+1}' for i in range(n_components)])
    return feat_data_df_scaled, tsne_df, scaler, tsne

def cluster_tsne_data(tsne_df: pd.DataFrame, eps=2, min_samples=25):
    """
    Cluster t-SNE data using DBSCAN.
    """
    db = DBSCAN(eps=eps, min_samples=min_samples).fit(tsne_df)
    labels = pd.Series(db.labels_, index=tsne_df.index, name='cluster_label')
    return labels

def filter_consecutive_ones(df: pd.DataFrame, col_name: str, min_length: int) ->
↪   pd.DataFrame:
    """
    Retain sequences of 1s in the column that meet minimum length criteria.
    """
    change_points = df[col_name].diff().ne(0).cumsum()
    for _, group in df.groupby(change_points):
        if group[col_name].iloc[0] == 1 and len(group) < min_length:
            df.loc[group.index, col_name] = 0
    return df

def regime_distance(regime_origin: dict, feat_data_df: pd.DataFrame,
```

```
                    dist_func: Callable, pca_dims=None) -> pd.Series:
    """
    Calculate distances between feature observations and a regime origin.
    Optionally apply PCA.
    """
    feat_cols = feat_data_df.columns.to_list()
    if pca_dims:
        pca = PCA(n_components=min(pca_dims, len(feat_cols)), svd_solver='full')
        feat_data_df = pd.DataFrame(pca.fit_transform(feat_data_df),
        ↪  index=feat_data_df.index)
        regime_origin = pca.transform(pd.DataFrame([regime_origin]))[0]

    return feat_data_df.apply(lambda row: dist_func(row, regime_origin), axis=1)

# K-Means can be used instead of DBSCAN
from sklearn.cluster import KMeans

def cluster_tsne_data_kmeans(tsne_df: pd.DataFrame, n_clusters=30) -> pd.Series:
    """
    Cluster t-SNE data using K-Means.
    """
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans.fit_predict(tsne_df)
    return pd.Series(labels, index=tsne_df.index, name='cluster_label')

def find_regime_cluster(
    regime_dict: dict,
    raw_dict_is_scaled: bool,
    scaled_data: pd.DataFrame,
    cluster_labels: pd.Series,
    scaler,
):
    """
    Find which cluster a user-defined regime belongs to by computing
    Euclidean distance in the scaled feature space.
    """
    feature_cols = scaled_data.columns.intersection(regime_dict.keys())
    regime_df = pd.DataFrame([regime_dict], columns=feature_cols)

    # If the user-provided regime is not already scaled, transform it
    if not raw_dict_is_scaled:
        regime_df = pd.DataFrame(scaler.transform(regime_df), columns=feature_cols)

    # Compute distances to each row in scaled_data
    dist = ((scaled_data[feature_cols] - regime_df.iloc[0])**2).sum(axis=1).pow(0.5)
    closest_idx = dist.idxmin()

    # Return that row's cluster label
    return cluster_labels.loc[closest_idx]

def label_close_rows(feature_data: pd.DataFrame, cluster_id: int):
    """
    Set 'close' = 1 if a row's cluster_label == cluster_id, otherwise 0.
    """
```

```
    feature_data['close'] = (feature_data['cluster_label'] == cluster_id).astype(int)

if __name__ == "__main__":
    # Path to your pickle file
    pkl_file_path = "/Users/bazilahmad/Desktop/genpf/Bazil/feature_data.pkl"

    # Load the feature data
    feature_data = pd.read_pickle(pkl_file_path)

    # Example regime dictionary
    regime_dict = {
        'two_year_treasury_yield': 0.015,
        'ten_two_year_spread': -0.002,
        'corporate_bbb_spread': 0.03,
        'high_yield_spread': 0.07,
        'unemployment_rate': 0.08,
        'median_cpi': 0.004,
        'real_gdp_growth': -0.01,
        'oecd_lei_us': 0.48
    }

    # Define the feature columns
    features = [
        'two_year_treasury_yield', 'ten_two_year_spread', 'corporate_bbb_spread',
        'high_yield_spread', 'unemployment_rate', 'median_cpi',
        'real_gdp_growth', 'oecd_lei_us'
    ]

    # 1. Apply t-SNE
    scaled_data, tsne_df, scaler, _ = apply_tsne(feature_data[features],
                                                 n_components=3,
                                                 perplexity=30)

    # 2. Cluster the t-SNE data
    cluster_labels = cluster_tsne_data(tsne_df, eps=6.5, min_samples=25)
    feature_data['cluster_label'] = cluster_labels

    # 3. Identify which cluster the user-defined regime belongs to
    user_cluster_id = find_regime_cluster(
        regime_dict=regime_dict,
        raw_dict_is_scaled=False,
        scaled_data=scaled_data,
        cluster_labels=cluster_labels,
        scaler=scaler
    )
    print(f"User-defined regime belongs to cluster {user_cluster_id}")

    # 4. Label all rows in that cluster as close = 1
    label_close_rows(feature_data, user_cluster_id)

    # 5. Plot the t-SNE regime similarity chart
    regime_similarity_chart_tsne(feature_data, title="t-SNE Regime Similarity Chart")
    plt.show()
```

Order of Operations in the Code. To summarize the sequence more concretely:

1. In `apply_tsne()`:

   (a) **Drop NaNs** to clean up data.

   (b) **Scale the features** using `StandardScaler`, so that each feature has zero mean and unit variance.

   (c) **Check if perplexity is larger than the dataset size**, and adjust if necessary. Perplexity cannot exceed (number_of_samples − 1).

   (d) **Instantiate the TSNE** object with user-defined parameters such as:

   - `n_components` (2D or 3D mapping),
   - `perplexity`,
   - `random_state` for reproducibility.

   (e) **Call** `fit_transform()` on the scaled data. Internally, scikit-learn does:

   - Pairwise distance calculations among the high-dimensional samples.
   - Conversion of distances to Gaussian-based similarities $p_{j|i}$.
   - Determination of the bandwidth σ_i for each point via the *perplexity* parameter.
   - Initialization of low-dimensional points (often randomly).
   - **Iterative gradient descent** to minimize the KL divergence between the high-dimensional distribution $p_{j|i}$ and the low-dimensional distribution q_{ij}.

   Once this optimization converges (or the iteration limit is reached), we obtain the final set of **2D or 3D coordinates** (depending on `n_components`).

   (f) **Return the scaled data, the t-SNE DataFrame**, and the `scaler` for future use (e.g., when comparing new regime parameters).

2. In `cluster_tsne_data(...)` or `cluster_tsne_data_kmeans(...)`:

   (a) You take the columns tsne_1, tsne_2, tsne_3 from the t-SNE output and apply **DBSCAN** or **K-Means** to cluster the *already* embedded coordinates.

   (b) In the code, `cluster_tsne_data(...)` uses DBSCAN with parameters `eps` and `min_samples` to identify dense clusters and label outliers as -1.

   (c) This step is a **separate clustering approach** that uses the final, low-dimensional points from t-SNE and classifies them into regimes (clusters).

3. **Determining "Close" via Nearest-Neighbor Logic**:

(a) We pass a user-defined `regime_dict` (the hypothetical regime) into the function `find_regime_cluster(...)`.

(b) That function checks whether the `regime_dict` values are scaled or not; if not, it uses the stored `StandardScaler` to transform them.

(c) It then computes Euclidean distances to each row in the scaled feature set, finds the closest row, and returns its `cluster_label`.

(d) Finally, `label_close_rows()` sets `close = 1` for all rows with that label, and 0 otherwise.

Note on the Alternate Approach Shown in the Code Snippet Below:
The Python script below uses a different technique. It *re-runs* t-SNE on (1) the original historical data and (2) the combined dataset (historical plus the new observation), so the new point is explicitly placed in t-SNE space. After that, it applies a nearest-neighbor search *in the newly generated embedding* to determine which cluster the new observation most closely aligns with.

While this ensures the new data point is embedded and clustered *directly* in t-SNE space, it can be computationally intensive to re-run t-SNE each time new parameters arrive. Since our primary goal is simply to identify "which existing cluster the new regime belongs to," we use the simpler nearest-neighbor approach in *scaled feature space*, skipping the repeated t-SNE re-embedding step.

```
# Combine the new observation with historical data
input_df = pd.DataFrame([input_variables])
combined_data = pd.concat([feature_data[features], input_df[features]], axis=0)

# Apply t-SNE to the combined dataset (historical + new observation)
combined_tsne = apply_tsne(combined_data, n_components=n_components)

# Separate the new observation's t-SNE coordinates from the historical set
new_observation_tsne = combined_tsne.iloc[-1].values.reshape(1, -1)
historical_tsne = combined_tsne.iloc[:-1]

# Find the nearest historical cluster for the new observation
neigh = NearestNeighbors(n_neighbors=1)
neigh.fit(historical_tsne)
nearest_index = neigh.kneighbors(new_observation_tsne, return_distance=False)
nearest_cluster_label = cluster_labels.iloc[nearest_index[0][0]]

# Mark 'close' = True for all rows that share this cluster_label
feature_data_with_clusters['close'] = (
    feature_data_with_clusters['cluster_label'] == nearest_cluster_label
)
```

4. In `regime_similarity_chart_tsne(...)`:

(a) We plot the `cluster_label` (or any relevant indicator) *over time*, leveraging the `Date` index.

(b) This is **purely a visualization** to see how the cluster membership changes or persists across dates, highlighting which time periods are considered "close" to the new regime.

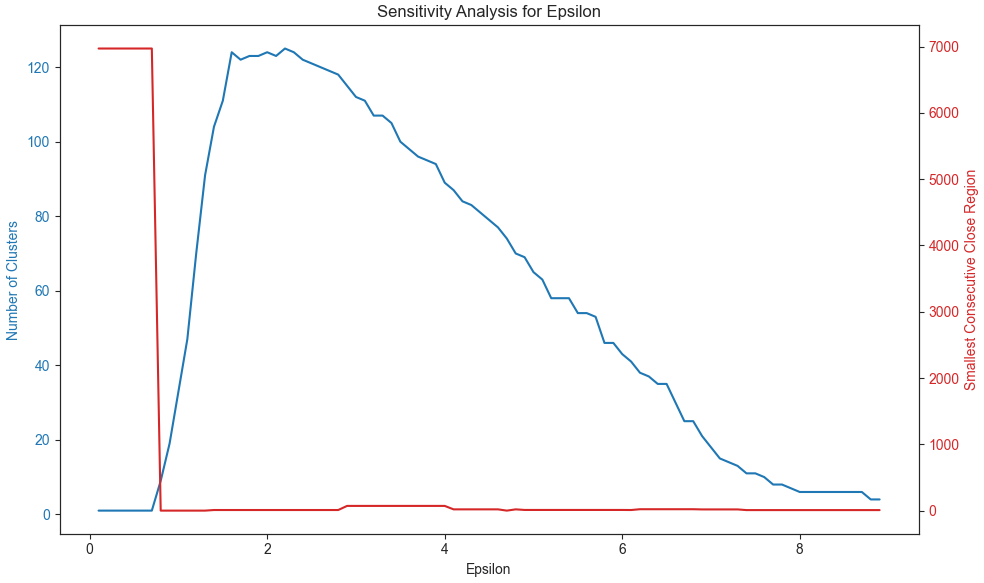
5 Analysis

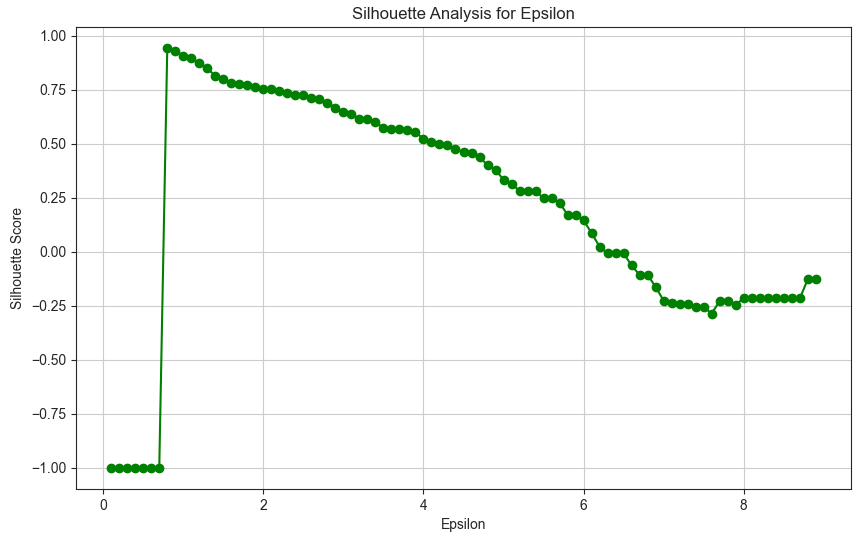
In this section, I provide a deeper look into how changing various parameters (such as **epsilon**, **min_samples**, **perplexity**) affects the resulting number of clusters, the smallest regime selection and the silhouette score (which evaluate how well-separated the clusters are). This involves:

- **Perplexity (t-SNE)**: Influences the balance between local and global aspects of the data. Higher perplexity values emphasize the global structure of the data, while lower values focus on local relationships.
- **epsilon (DBSCAN)**: Controls the radius of the neighborhood around a data point. Lower epsilon values result in more clusters with smaller neighborhoods, while higher values merge clusters into fewer groups.
- **min_samples (DBSCAN)**: Specifies the minimum number of points required to form a dense region. Higher values increase the strictness of forming clusters, potentially leading to fewer but larger clusters.

For the sake of simplicity, when changing one parameter, the other two remain constant. The constant values are 6, 25, and 30 for epsilon, min_samples, and perplexity, respectively.

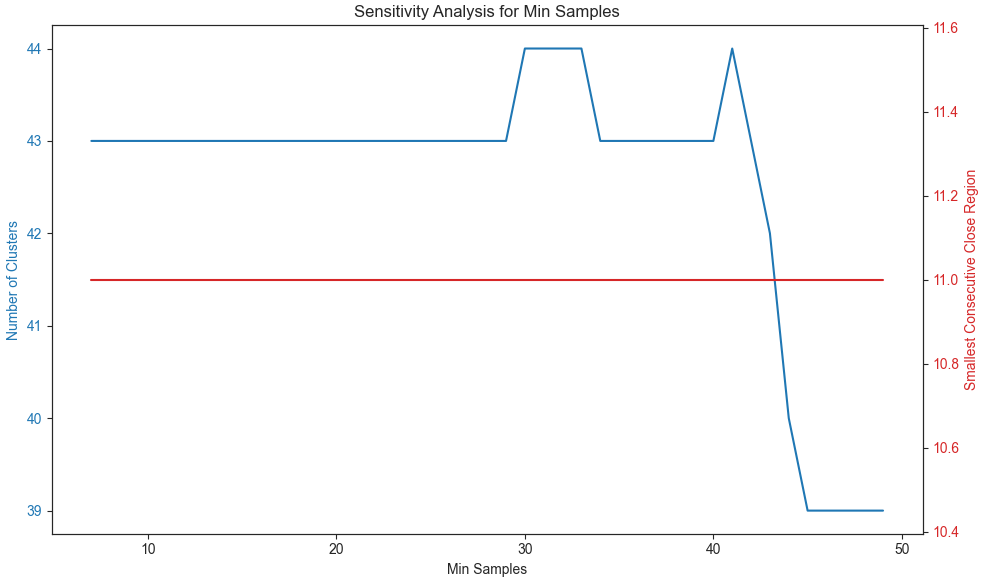
5.1 Sweeping Epsilon

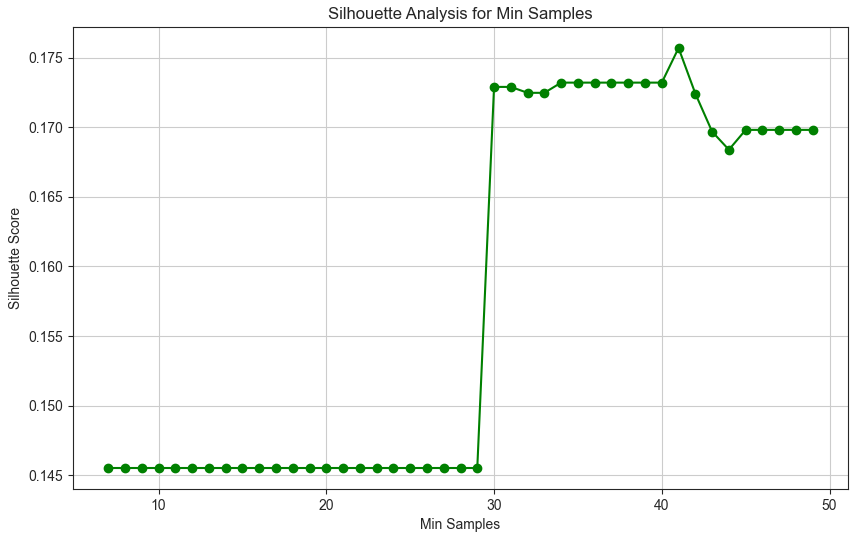




The first plot shows that as **epsilon** increases, the number of clusters generally decreases as smaller neighborhoods merge into larger ones. However, the length of the smallest consecutive "close" region stabilizes after an initial fluctuation, suggesting that at higher epsilon values, regime selection becomes less sensitive to noise.

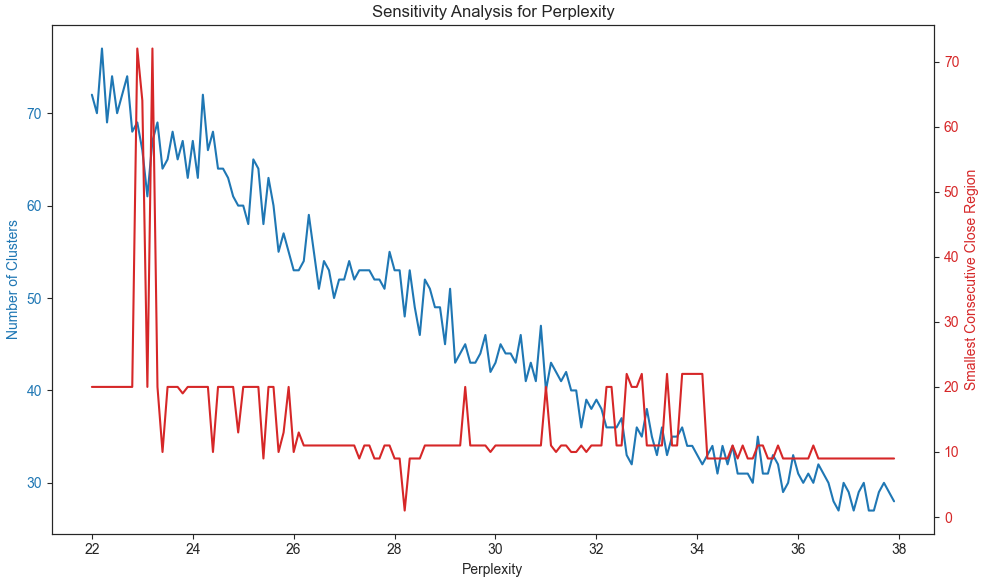
5.2 Sweeping Min Samples

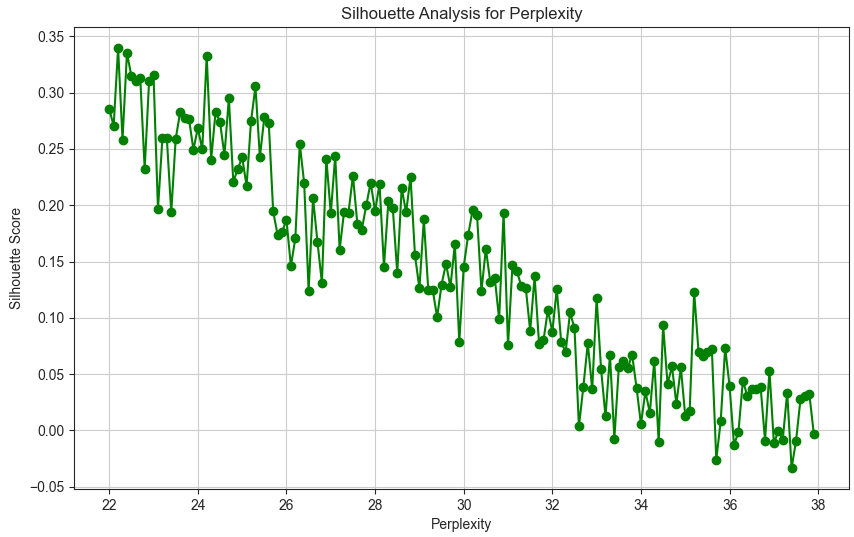




For **min_samples**, the number of clusters remains fairly constant until a threshold is reached, beyond which clusters consolidate. The smallest consecutive "close" region shows minimal change, indicating that this parameter has limited impact on regime selection within the tested range.

5.3 Sweeping Perplexity





The **perplexity** parameter affects the t-SNE embedding structure. Higher perplexity values reduce the number of clusters as global structure is emphasized, but the smallest consecutive "close" region varies non-monotonically, reflecting sensitivity to how t-SNE balances local and global relationships.

5.4 Silhouette Score Analysis

The silhouette score provides a measure of how well-separated and cohesive the clusters are. Below is an analysis of the silhouette plots for each parameter sweep:

Sweeping Epsilon

- For very small ε, the silhouette score is strongly negative (around -1), indicating over-fragmented clusters or points labeled as outliers.
- At moderate ε values (around 1–5), the score climbs sharply toward 1.0, reflecting well-separated and internally coherent clusters. This suggests an optimal range for forming distinct regimes.
- As ε continues to increase, the score gradually decreases and approaches 0.0, indicating that clusters are merging and becoming less distinct.
- **Takeaway:** Choose ε within the narrow mid-range to maximize cluster quality based on silhouette scores.

Sweeping Min Samples

- At lower `min_samples` values (< 30), the silhouette score is steady but low (~ 0.14), suggesting that clusters are loosely defined and potentially noisy.
- Around `min_samples` ≈ 30, the score jumps to ~ 0.17–0.18, indicating improved cluster separation and density.
- Beyond 35–40, the score stabilizes, showing that increasing `min_samples` past a certain threshold has minimal additional impact.
- **Takeaway:** Use `min_samples` values near the transition point (~ 30) for the best balance between cluster density and separation.

Sweeping Perplexity

- At lower perplexity values (~ 22), silhouette scores are highest (around 0.30), as t-SNE emphasizes local relationships and creates distinct clusters.
- Between 25 and 30, the score decreases to the 0.15–0.20 range, indicating a trade-off as t-SNE balances local and global structures.
- For higher perplexity values (> 30), the score drops further, even approaching 0.0, as clusters blur together and boundaries become less distinct.
- **Takeaway:** Lower perplexities (22–25) yield embeddings with better-defined clusters, while higher perplexities emphasize global structure at the expense of separation.

Overall Takeaways

- Silhouette scores range from -0.1 to 0.3, which is typical for noisy or complex datasets like financial/macro data.
- Optimal parameters vary across sweeps:
  - For ε, the optimal range is 1–5.
  - For `min_samples`, a transition around 30 yields improved clustering.

  - For perplexity, lower values (22–25) provide better cluster separation.

- The silhouette score complements other metrics like the number of clusters, outlier ratios, domain relevance, and the smallest consecutive close region.

- It is important to acknowledge that this analysis was conducted by varying each parameter independently while keeping the others constant. As a result, interactions between parameters, which can significantly influence clustering behavior, were not explored in depth. For future analyses, it would be valuable to examine how these parameters interact with each other by plotting higher-dimensional relationships (e.g., Silhouette Score as a function of multiple parameters). This could help identify the "sweet spot" in the parameter space where clustering quality is optimized. Techniques such as grid search or gradient descent could be employed to navigate this multi-parameter space more effectively, enabling a more comprehensive understanding of the parameter interactions and their impact on clustering performance.

5.5 Conclusion

The analysis reveals that **epsilon** has the most significant impact on the number of clusters and regime selection sensitivity, with clear stabilization at higher values. **Min_samples** primarily affects cluster formation thresholds but has limited influence on regime sensitivity. **Perplexity** introduces variability in embedding structure, making it crucial for visualizing meaningful relationships in the data. Further tuning of these parameters can provide insights into regime dynamics and clustering behavior. Note that next steps once some reasonable parameter ranges are chosen is to try different inputs (unemployment rate, high yield spread, etc.) to ensure that no thin regimes are chosen. It is important to expand the analysis of perplexity and epsilon in the future (3-D graph).

6 Summary and Next Steps

- **Findings:**

  - t-SNE can help remove "thin" regimes by separating clusters and using methods like K-means.

  - DBSCAN could identify clusters without forcing every point into a group, but it was highly sensitive to `eps` and `min_samples`.

  - K-Means offered a straightforward way to get a fixed number of clusters, which was useful for certain analyses.

  - Nearest-neighbor logic (rather than the mode) provides a more direct way to find which cluster a *user-defined* regime belongs to.

- **Challenges:**

  - Handling very small regime clusters (thin regimes) requires more parameter tuning and additional logic.
  - t-SNE can be computationally expensive for very large datasets.
  - Balancing interpretability (PCA) vs. representational power (t-SNE).
  - Ensuring we properly scale (or unscale) new regime parameters before matching them to historical data.

- **Next Steps:**
  - Perform a more systematic parameter sweep and document how the number of clusters (and cluster composition) changes with each setting.
  - Create mappings of points in 3D and 2D space, shaded by cluster. Optimize the t-SNE and clustering parameters not by the number of clusters but by minimizing "noise," aiming for clearly separated and distinct clusters.
  - Investigate more robust outlier detection methods or alternative clustering algorithms (e.g., HDBSCAN).
  - Consider complementing the existing approach with domain knowledge or additional features (e.g., volatility indices).
  - Evaluate adding approximate `transform` methods for t-SNE (e.g., openTSNE) if we need to embed new data points directly in t-SNE space rather than using a nearest-neighbor approach in scaled space.
  - Increase speeds via distributed computing and faster algorithms. Check out Parallelized Approach