Assignment 2

1. Regression

In this assignment, you will explore the California Housing Prices dataset. Your task is to apply various regression techniques, specifically Kernel Ridge Regression (KRR), Bayesian Linear Regression, and Gaussian Process Regression, to predict housing prices.

Dataset

We will use the "California Housing Prices" dataset from the `sklearn.datasets` module for ease of access.

Objectives

1. Perform exploratory data analysis (EDA) to understand the dataset.
2. Apply Kernel Ridge Regression (KRR) with different hyperparameters and analyze the results.
3. Implement Bayesian Linear Regression and discuss the posterior distributions.
4. Explore Gaussian Process Regression and visualize the prediction uncertainties.
5. Compare the performance of the three regression techniques and discuss your findings.

Tasks

1. Data Exploration and Preprocessing

- Use the knowledge you gain from previouse assignment on this dataset.
- You do not have to repeat the visualization of the data, only remeber the important fact that you gain by analysing it.

2. Kernel Ridge Regression (KRR)

- Apply KRR to the dataset.
- Experiment with different kernels and regularization parameters.
- Analyze the performance and discuss how different hyperparameters impact the model.

3. Bayesian Linear Regression

- Implement Bayesian Linear Regression. You can use Bayesian Ridge Regression function from Scikit-Learn.
- Visualize the posterior distributions of the coefficients (at least one coefficient).
- Discuss the insights gained from the posterior analysis.

4. Gaussian Process Regression

- Apply Gaussian Process Regression to the dataset.
- Visualize the prediction uncertainties.
- Discuss how the Gaussian Process handles uncertainty in predictions.

5. Comparative Analysis

- Compare the results obtained from KRR, Bayesian Linear Regression, and Gaussian Process Regression.
  - Visualize the regression fit provided by each model and compare.
  - Use at least one of the metrics: MSE, RMSE, R2
- Evaluate and discuss the performance, computational efficiency, and ease of interpretation of each model.
  - For Bayesian Linear Regression, analyze the posterior distributions of the coefficients. For KRR, discuss the interpretability of the kernel.

2. Outlier detection

In this assignment, you can choose one of the below dataset and your task is to identify the outliers in them using various machine learning techniques.

Dataset

You can choose One of the below datasets:

1. The dataset is a small sample from the Fashion MNIST dataset with manually added outliers. The data will be provided in the form of two numpy arrays: `images` and `labels`. (Note:check for missing data)
2. You will generate your data with outliers:
   - Use NumPy to create data points that follow a normal distribution. This forms the "normal" part of your dataset.
   - Manually add data points that are significantly different from the normal data.
   - These points should be distant from the mean of the normal data to be considered outliers.
   - Combine the normal data points and outliers into a single dataset.

Objectives

1. Perform exploratory data analysis (EDA) to understand the dataset.
2. Implement PCA (Principal Component Analysis) for dimensionality reduction and visualize the results.
3. Use K-means clustering to identify potential outliers.
4. Apply t-SNE (t-Distributed Stochastic Neighbor Embedding) for visualization and detect anomalies.
5. (Optional) Design and train an autoencoder and use reconstruction error to find outliers.

6. Compare the effectiveness of the above methods in outlier detection.

Tasks

1. Exploratory Data Analysis (EDA)

- Load the dataset and visualize some images.
- Plot the distribution of the different classes in the dataset.

2. PCA for Dimensionality Reduction

- Implement PCA to reduce the dimensionality of the dataset.
- Visualize the data in the reduced dimension space.

3. Choose one of the below tasks:

- QDA for Outlier Detection
    - Apply QDA to the dataset.
    - Analyze how the QDA decision boundary help in outlier detection.
- K-means Clustering
    - Apply K-means clustering on the dataset.
    - Identify clusters that potentially contain outliers.

5. t-SNE for Visualization

- Apply t-SNE to the dataset and visualize the results.
- Discuss how t-SNE helps in identifying outliers.

6. Autoencoder for Outlier Detection (Optional)

- Design and train an autoencoder on the dataset.
- Use the reconstruction error to identify images that are outliers.

7. Comparative Analysis

- Compare the results of the models you chose to study.
- Discuss the effectiveness and limitations of each method in outlier detection.

How to Submit

- First, a Jupyter Notebook containing all the code, comments, and analysis.
- Second report cells in the same Jupyter Notebook, summarizing your findings, including results and a discussion of the results.
- Finally convert the Jupyter Notebook to PDF.
- **Don't write your name**.

- Upload the PDF into convas.

Evaluation Criteria (peer grading)

- Correctness of the implementation of the models. (2 points)
- Quality of the EDA and preprocessing steps. (1 point)
- Depth of the analysis in comparing the models.(1 point)
- Clarity and organization of the submitted report and Jupyter Notebook. (1 point)

1. Regression

In this assignment, you will explore the California Housing Prices dataset. Your task is to apply various regression techniques, specifically Kernel Ridge Regression (KRR), Bayesian Linear Regression, and Gaussian Process Regression, to predict housing prices.

1. Data Exploration and Preprocessing

In [1]:
```
from sklearn.datasets import fetch_california_housing

california_housing = fetch_california_housing(as_frame=True)
```

In [2]:
```
print(california_housing.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude

    :Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surprisingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. topic:: References

    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
      Statistics and Probability Letters, 33 (1997) 291-297
```

In [3]:
```
california_housing.frame.head()
```

Out[3]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

In [4]:
```
california_housing.frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   MedInc       20640 non-null  float64
 1   HouseAge     20640 non-null  float64
 2   AveRooms     20640 non-null  float64
 3   AveBedrms    20640 non-null  float64
 4   Population   20640 non-null  float64
 5   AveOccup     20640 non-null  float64
 6   Latitude     20640 non-null  float64
 7   Longitude    20640 non-null  float64
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

The dataset has 20'640 samples and 8 features;

All features are numerical features encoded as floating number. There are no missing values.

Let's see statistics of our data:

In [5]:
```
california_housing.frame.describe()
```

Out[5]:

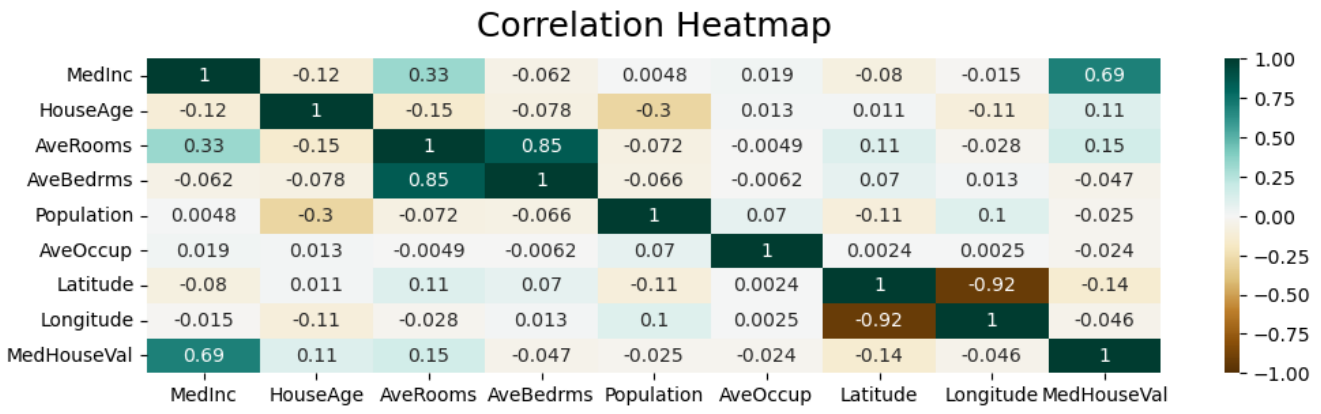
	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20(
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.070655	35.631861	-
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386050	2.135952	
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692308	32.540000	-
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429741	33.930000	-
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818116	34.260000	-
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282261	37.710000	-
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.333333	41.950000	-

Some features have extreme values (outliers): Number of bedrooms, rooms, population and occupation.

In [6]:
```
#To make our analyse better I would get rid of prices that are more than 4.9 because they ar
#california_housing.frame = california_housing.frame.drop(california_housing.frame[californi
#california_housing.frame.head()
```

Now let's create a Correlation Heat map to see any relationships between variables.

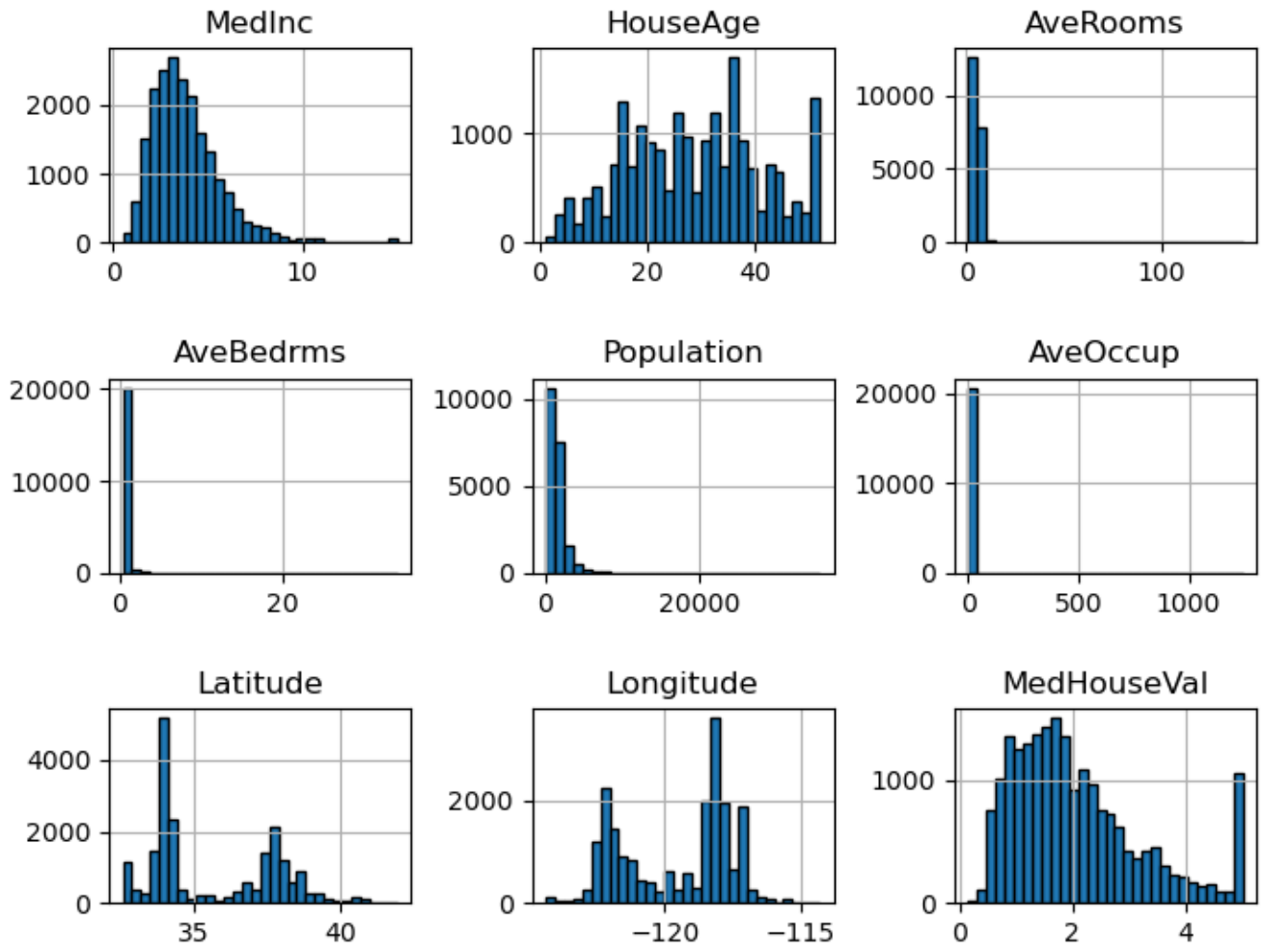
In [7]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 3))
heatmap = sns.heatmap(california_housing.frame.corr(), vmin=-1, vmax=1, annot=True, cmap='Br
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':18}, pad=12);
plt.savefig('heatmap.png', dpi=300, bbox_inches='tight')
```



The features are not very correlated with each other. The Median income has the highest correlation with a target variable meaning that the median income is a useful feature to help at predicting the median house values.

Let's create a histogram

In [8]:
```
import matplotlib.pyplot as plt

california_housing.frame.hist(figsize=(8, 6), bins=30, edgecolor="black")
plt.subplots_adjust(hspace=0.7, wspace=0.4)
```



The median income is a distribution with a long tail. It means that the salary of people is more or less normally distributed but there is some people getting a high salary.

House age has the distribution is more or less uniform.

The target distribution has a long tail as well. Plus there are high-valued houses.

Let's normalize/standardize the features.

In [9]:
```
from sklearn.preprocessing import StandardScaler
# transform data
scaler = StandardScaler()
```

In [10]:
```
df = pd.DataFrame(california_housing.data, columns=california_housing.feature_names)
df_scaled = scaler.fit_transform(df)
df_scaled.shape
```

Out[10]:
```
(20640, 8)
```

In [11]:
```
df['target'] = california_housing.target
target_scaled = scaler.fit_transform(df['target'].values.reshape(-1, 1))
target_scaled.shape
```

Out[11]:
```
(20640, 1)
```

2. Kernel Ridge Regression (KRR)

Apply KRR with different hyperparameters and analyze the results. λ is the regularization parameter, and n is the number of samples.

Background

- Ridge Regression is an extension of linear regression with L2 regularization.
- The model is $\hat{y} = X\theta$.
- The cost function with L2 regularization is:

$$J(\theta) = \frac{1}{2n}\|Y - X\theta\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2$$

- λ is the regularization parameter, and n is the number of samples.

Kernel Trick

- KRR employs the kernel trick to handle non-linear relationships.
- The kernel trick maps input data into a high-dimensional (potentially infinite) feature space.
- A kernel function $k(x, x')$ computes the dot product in this high-dimensional space.

Kernel Function

- Common kernels: Polynomial, Gaussian (RBF), and Sigmoid.
- Kernel function: $k(x, x') = \phi(x)^T\phi(x')$.
- $\phi(x)$ is the mapping from input space to feature space.

KRR Formulation

- The model is $\hat{y} = K\alpha$, where K is the kernel matrix with $K_{ij} = k(x_i, x_j)$.
- The cost function in terms of α is:

$$J(\alpha) = \frac{1}{2n}\|Y - K\alpha\|_2^2 + \frac{\lambda}{2}\alpha^T K\alpha$$

Solving for α

- To minimize $J(\alpha)$, set the derivative w.r.t. α to zero:

$$\frac{\partial J}{\partial \alpha} = -\frac{1}{n}K^T(Y - K\alpha) + \lambda K\alpha = 0$$

- Solving gives $\alpha = (K + n\lambda I)^{-1}Y$.

Prediction

- Predictions for new data x^* are made using:

$$\hat{y}(x^*) = \sum_{i=1}^{n} \alpha_i k(x_i, x^*)$$

In [12]:
```
# Import necessary libraries
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np

# Assuming you've already loaded and preprocessed the data as mentioned in your code

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df_scaled, target_scaled, test_size=0.2,
```

In [ ]:
```
# Define a function to train and evaluate KRR with different hyperparameters
def evaluate_krr(kernel, alpha):
    # Create KRR model
    krr = KernelRidge(kernel=kernel, alpha=alpha)

    # Fit the model on the training data
    krr.fit(X_train, y_train)

    # Predict on the test data
    y_pred = krr.predict(X_test)

    # Calculate Mean Squared Error
    mse = mean_squared_error(y_test, y_pred)

    return mse

# Experiment with different kernels and regularization parameters
kernels = ['linear', 'poly', 'rbf']
alphas = [0.1, 1, 10]

for kernel in kernels:
    for alpha in alphas:
        mse = evaluate_krr(kernel, alpha)
        print(f'Kernel: {kernel}, Alpha: {alpha}, MSE: {mse}')
```

```
Kernel: linear, Alpha: 0.1, MSE: 0.41747051534229535
Kernel: linear, Alpha: 1, MSE: 0.41744322644721565
Kernel: linear, Alpha: 10, MSE: 0.41717903997428574
Kernel: poly, Alpha: 0.1, MSE: 15.069941883276087
Kernel: poly, Alpha: 1, MSE: 10.02999168662882
Kernel: poly, Alpha: 10, MSE: 2.449224586177838
Kernel: rbf, Alpha: 0.1, MSE: 0.23537211275383668
Kernel: rbf, Alpha: 1, MSE: 0.25792225018523546
Kernel: rbf, Alpha: 10, MSE: 0.30225074517905
```

In this code, we use three different kernels: linear, polynomial ('poly'), and radial basis function ('rbf'). We also vary the regularization parameter (alpha) to observe how it affects the model's performance. As

we understood that RBF Kernel is the best than **let's try to use GridSearchCV with a cross-validation to find the best hyperparameters in a model with RBF kernel.**

In [13]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score

# Create KernelRidge model
krr = KernelRidge(kernel='rbf')

# Define the parameter grid to search
param_grid = {'alpha': [0.1, 1],
              'gamma': [0.01, 0.1, 1]}

# Create GridSearchCV instance with 5-fold cross-validation
grid_search = GridSearchCV(krr, param_grid, cv=5, scoring='neg_mean_squared_error')

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters from the grid search
best_alpha = grid_search.best_params_['alpha']
best_gamma = grid_search.best_params_['gamma']

# Create the best model with the best hyperparameters
best_krr = KernelRidge(kernel='rbf', alpha=best_alpha, gamma=best_gamma)

# Fit the best model to the training data
best_krr.fit(X_train, y_train)

# Predict on the test data
y_pred = best_krr.predict(X_test)

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

# Calculate and print the R-squared (R2) score
r2 = r2_score(y_test, y_pred)

print(f'Best Alpha: {best_alpha}, Best Gamma: {best_gamma}, Best MSE: {mse}, R-squared (R2)
```

```
Best Alpha: 0.1, Best Gamma: 1, Best MSE: 0.22712906372883976, R-squared (R2) Score on Test
Set: 0.7692065774980987
```

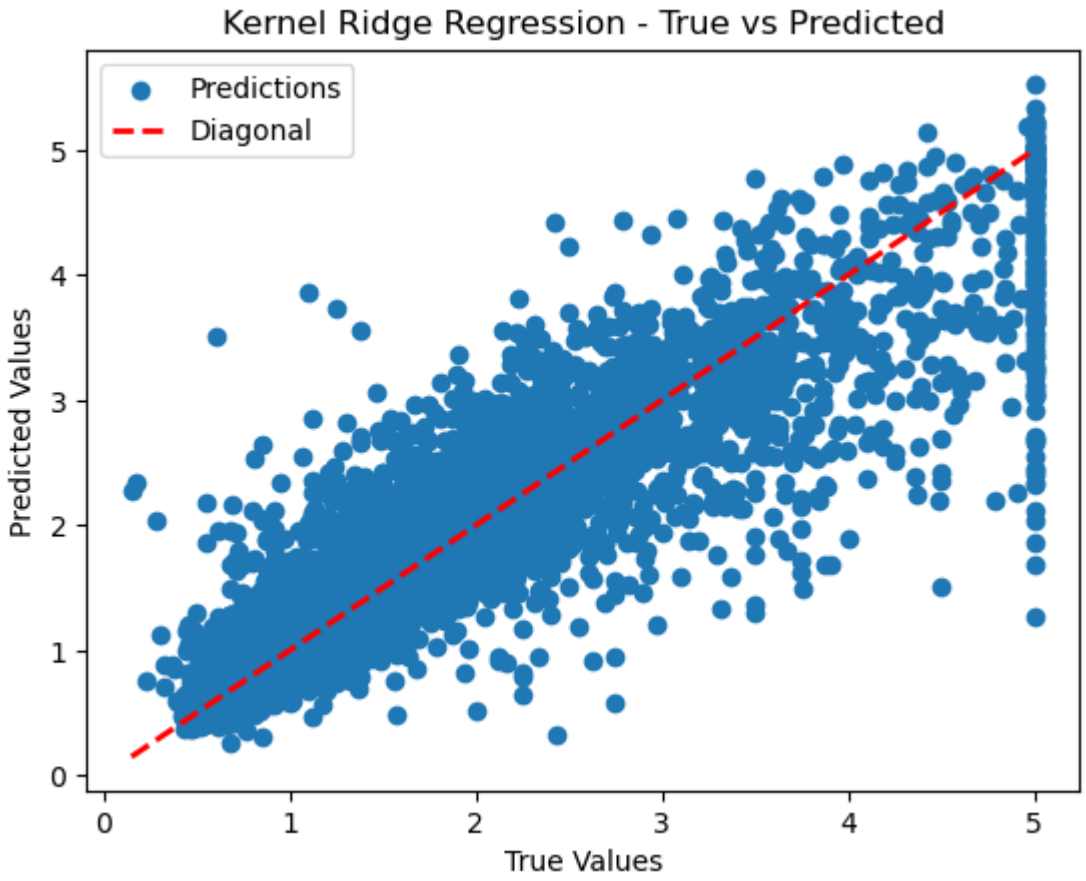
The **'alpha'** values control the regularization strength, and **'gamma'** controls the kernel coefficient for the RBF kernel. Higher values of alpha result in stronger regularization. GridSearchCV performs a search over the specified hyperparameter values and cross-validates the results to find the best combination.

Note that the scoring parameter is set to **'neg_mean_squared_error'** since GridSearchCV maximizes the scoring metric, and we want to minimize the mean squared error.

Let's visualize the performance of my model by plotting the predicted values against the true values.

In [14]:
```
import matplotlib.pyplot as plt

# Predict on the test data using the best model
```

```
y_pred = best_krr.predict(X_test)

# Transform back the scaled values to the original scale
y_test_original = scaler.inverse_transform(y_test)
y_pred_original = scaler.inverse_transform(y_pred)

# Create a scatter plot
plt.scatter(y_test_original, y_pred_original, label='Predictions')
plt.plot([min(y_test_original), max(y_test_original)], [min(y_test_original), max(y_test_ori
plt.title('Kernel Ridge Regression - True vs Predicted')
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.legend()
plt.show()
```

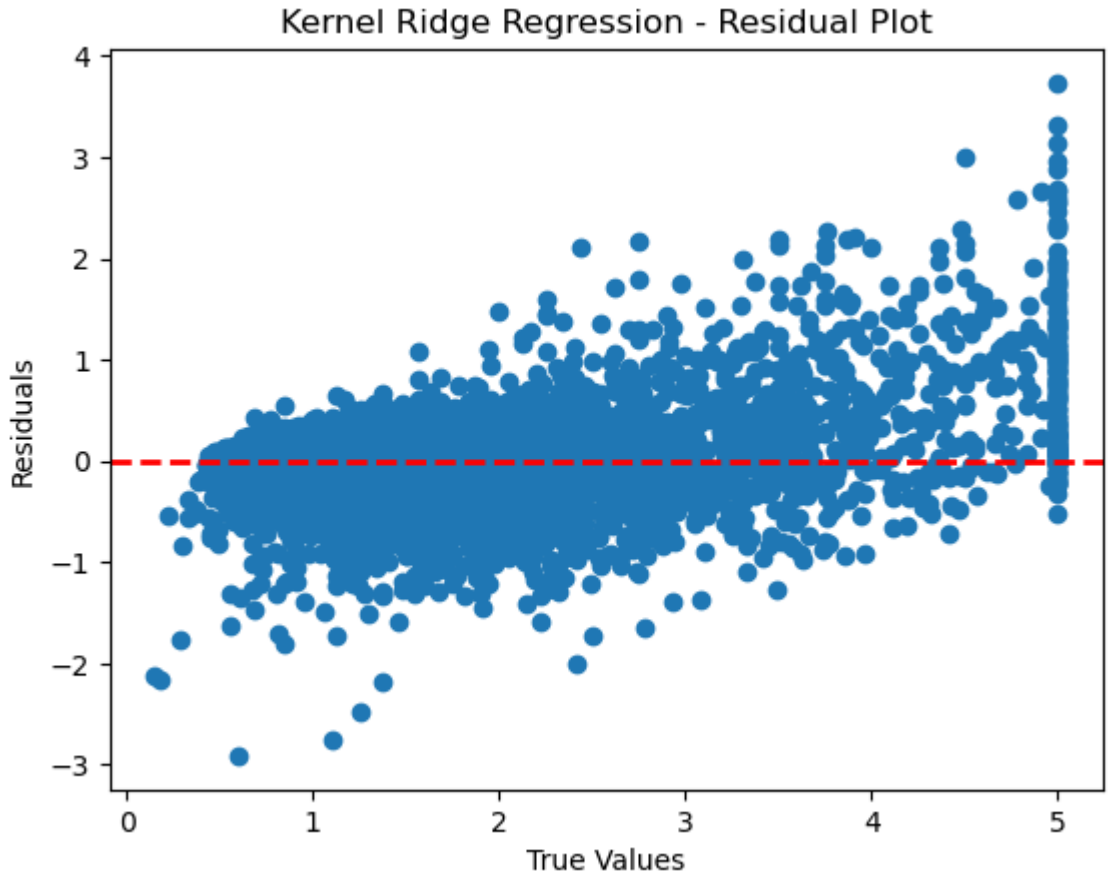


In this code, we first transform the predicted and true values back to their original scale using scaler.inverse_transform. Then, we create a scatter plot where the x-axis represents the true values, and the y-axis represents the predicted values.

Let's creat a residual plot is a useful way to visualize the residuals, which are the differences between the true values and the predicted values.

In [16]:
```
# Calculate residuals
residuals = y_test_original - y_pred_original

# Create a residual plot
plt.scatter(y_test_original, residuals)
plt.axhline(y=0, color='r', linestyle='--', linewidth=2)
```

```
plt.title('Kernel Ridge Regression - Residual Plot')
plt.xlabel('True Values')
plt.ylabel('Residuals')
plt.show()
```



So we have the best model with the current hyperparameters and evaluation metrics: Best Alpha: 0.1, Best Gamma: 1, Best MSE: 0.22712906372883976, R-squared (R2) Score on Test Set: 0.7692065774980987

The result is not so bad compared to models created in assignment 1

3. Bayesian Linear Regression

Bayesian Linear Regression allows us to incorporate uncertainty in our model by placing a prior distribution over the coefficients and updating it with the data to obtain a posterior distribution.

In this example, we fit a Bayesian Ridge Regression model to the training data. We then extract the posterior mean and covariance of the coefficients. The code uses a multivariate normal distribution to generate samples from the posterior distribution of each coefficient. The seaborn library is used to create a histogram for each coefficient, visualizing its posterior distribution.

The insights gained from the posterior analysis involve understanding the uncertainty in the estimated coefficients. A wider distribution implies higher uncertainty, and examining the posterior mean helps assess the significance of each feature in predicting housing prices.

Posterior Mean: The histogram provides an estimate of the most likely value for each coefficient. The higher density regions in the histograms indicate the most probable values.

Uncertainty: The width of the histograms reflects the uncertainty associated with each coefficient. Wider histograms suggest higher uncertainty, while narrower histograms indicate more confidence in the estimated values.

Correlation: If the posterior distributions of two coefficients overlap or have similar shapes, it suggests that those coefficients might be correlated. Understanding these correlations can provide insights into the relationships between different features in your dataset.

Bayesian Ridge Regression has hyperparameters that it's possible to tune to optimize the model. The main hyperparameters for Bayesian Ridge Regression include:

alpha_1 and alpha_2: These are the shape parameters of the Gamma prior for the precision (inverse of variance) of the weights. They control the strength of the regularization on the coefficients.

lambda_1 and lambda_2: These are the shape parameters of the Gamma prior for the noise precision (inverse of variance). They control the strength of the regularization on the noise.

compute_score: This hyperparameter specifies whether to compute the log marginal likelihood during the training. It can be set to True or False.

fit_intercept: This hyperparameter specifies whether to fit an intercept term. It can be set to True or False.

normalize: This hyperparameter specifies whether to normalize the features before fitting the model. It can be set to True or False.

When using Bayesian Ridge Regression with scikit-learn's BayesianRidge class, you can include these hyperparameters in a dictionary and search for the best combination using techniques like GridSearchCV or RandomizedSearchCV.

Assume we have a Gaussian prior distribution than we use BayesianRidge

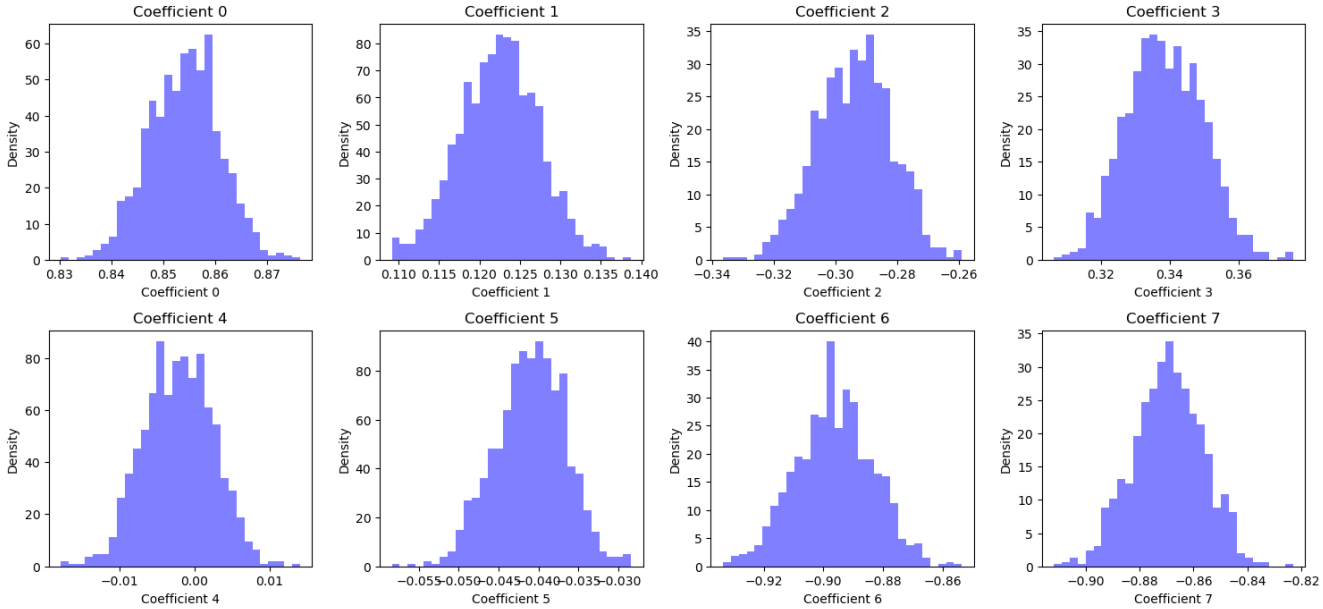
If the noise variance (σ^2) is not known and we want to estimate it along with other parameters through the marginal likelihood, we can follow an empirical Bayes approach. In this case, we can maximize the marginal likelihood to find the optimal hyperparameters, which include both the coefficients and the noise variance.

An example using the BayesianRidge model with an empirical Bayes approach

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import BayesianRidge
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Load the California Housing Prices dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)

# Fit Bayesian Ridge Regression model (Empirical Bayes with normal prior)

#compute_score=True is used when initializing BayesianRidge
#to enable the computation of the score (log marginal likelihood).
bayesian_ridge = BayesianRidge(compute_score=True)
bayesian_ridge.fit(X_train_std, y_train)

# Get the optimal hyperparameters (alpha, lambda_)
optimal_alpha_ = bayesian_ridge.alpha_
optimal_lambda_ = bayesian_ridge.lambda_

# Use the optimal hyperparameters to get the posterior mean and covariance
posterior_mean = bayesian_ridge.coef_
posterior_covariance = \
    np.linalg.inv(optimal_alpha_ * np.eye(X_train_std.shape[1]) + \
                  optimal_lambda_ * np.dot(X_train_std.T, X_train_std))

# Generate posterior samples for all coefficients
posterior_samples = np.random.multivariate_normal(mean=posterior_mean,\
                                                  cov=posterior_covariance, size=1000)
```

In [17]:
```
# Plot histograms for all coefficients in one plot
num_features = X_train_std.shape[1]

plt.figure(figsize=(15, 10))
for i in range(num_features):
    plt.subplot(3, 4, i + 1)
```

```
    plt.hist(posterior_samples[:, i], bins=30, density=True, alpha=0.5, color='blue',\
             label='Posterior Distribution')
    plt.title(f'Coefficient {i}')
    plt.xlabel(f'Coefficient {i}')
    plt.ylabel('Density')

plt.tight_layout()
plt.show()
```



In Bayesian regression, we often model the uncertainty in the parameters using a multivariate normal distribution, and the covariance matrix captures the joint uncertainty between different parameters. When generating samples from this distribution, we use the covariance matrix.

posterior samples line generates 1000 random samples from a multivariate normal distribution, where each sample corresponds to a set of coefficients sampled from the posterior distribution. In Bayesian statistics, this process is often used for Monte Carlo sampling methods, such as Markov Chain Monte Carlo (MCMC), to approximate the posterior distribution and make inferences about the model parameters.

Let's find coefficients with the highest uncertainty

In [6]:
```
# Compute standard deviations of the posterior distribution
posterior_std = np.sqrt(np.diag(posterior_covariance))

# Identify the coefficient with the highest uncertainty
highest_uncertainty_index = np.argmax(posterior_std)
highest_uncertainty_value = posterior_std[highest_uncertainty_index]

print(f"Coefficient {highest_uncertainty_index} has the highest uncertainty with standard de
{highest_uncertainty_value}")
```

```
Coefficient 6 has the highest uncertainty with standard deviation: 0.013210177291444388
```

Uncertainty is expressed through the posterior distribution, which represents a probability distribution over the possible values of the coefficients given the observed data.

In [7]:
```
posterior_mean
```

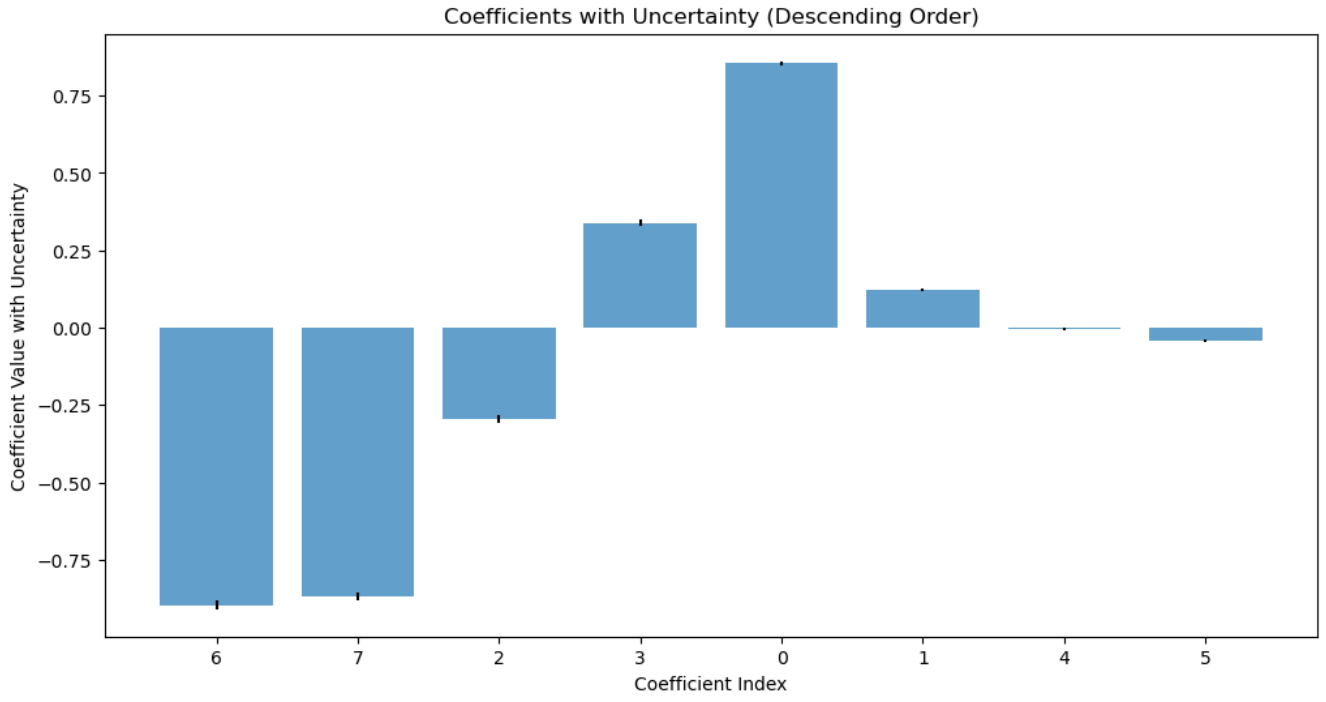
Out[7]:
```
array([ 0.8542899 ,  0.12267475, -0.29407961,  0.33884338, -0.00226554,
       -0.04083557, -0.8956702 , -0.86856698])
```

In [8]:
```
posterior_std
```

Out[8]:
```
array([0.00693931, 0.0048444 , 0.01225109, 0.01119366, 0.00463949,
       0.00437635, 0.01321018, 0.01297125])
```

Sixth, Seventh, Second,and Third features have a high uncertainty. High uncertainty for a coefficient indicates a lack of confidence in the model's estimation of the relationship between a specific feature and the target variable. This could be due to insufficient data, model complexity, or inherent variability in the relationship.

In [9]:
```
# Get indices sorted by decreasing uncertainty
sorted_indices = np.argsort(posterior_std)[::-1]

# Plot bar chart with coefficients and uncertainties
plt.figure(figsize=(12, 6))
plt.bar(range(len(sorted_indices)), posterior_mean[sorted_indices], \
        yerr=posterior_std[sorted_indices], \
        align='center', alpha=0.7)
plt.xticks(range(len(sorted_indices)), sorted_indices)
plt.xlabel('Coefficient Index')
plt.ylabel('Coefficient Value with Uncertainty')
plt.title('Coefficients with Uncertainty (Descending Order)')
plt.show()
```



In [11]:
```
# Get the names of features for the sixth and seventh coefficients
feature_names = data.feature_names
sixth_coefficient_name = feature_names[6]
seventh_coefficient_name = feature_names[7]
second_coefficient_name = feature_names[2]
third_coefficient_name = feature_names[3]
```

```
print(f"The name of the feature for the sixth coefficient is: {sixth_coefficient_name}")
print(f"The name of the feature for the seventh coefficient is: {seventh_coefficient_name}")
print(f"The name of the feature for the second coefficient is: {second_coefficient_name}")
print(f"The name of the feature for the third coefficient is: {third_coefficient_name}")
```

```
The name of the feature for the sixth coefficient is: Latitude
The name of the feature for the seventh coefficient is: Longitude
The name of the feature for the second coefficient is: AveRooms
The name of the feature for the third coefficient is: AveBedrms
```

The most important that Sixth and Seventh coefficients have a high mean AND a high deviation.

Latitude and Longitude.

Potential Instability: The combination of a high mean and high uncertainty can imply that the model is sensitive to variations in the data, and the coefficient's estimate might change significantly with different subsets of the data. This sensitivity could be a sign of potential instability in the model.

Let's do prediction on Test data

In [12]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Load the California Housing Prices dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
# Make predictions on the test set
y_pred = bayesian_ridge.predict(X_test_std)

# Optionally, inverse transform the predictions if you scaled the target variable
# Assuming you used StandardScaler on the target variable

# Calculate and print the Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error on Test Set: {mse}')

# Calculate and print the R-squared (R2) score
r2 = r2_score(y_test, y_pred)
print(f'R-squared (R2) Score on Test Set: {r2}')
```

```
Mean Squared Error on Test Set: 0.5558309454544386
R-squared (R2) Score on Test Set: 0.5758339917659987
```

It's obvious that this model has showed not the best performance on our test data.

2. Gaussian Process Regression

In Gaussian Process Regression (GPR), the choice of the prior distribution is a fundamental aspect. The term "Gaussian" in Gaussian Process refers to the fact that the prior distribution over functions is a multivariate Gaussian distribution. This prior distribution captures the assumptions about the smoothness and behavior of the underlying function.

If you deviate from a Gaussian prior, it essentially means using a different kernel function. The kernel function determines the shape and characteristics of the assumed functions in the prior distribution.

Choice of Kernel: The kernel function is a crucial component of GPR. The most common kernel is the Radial Basis Function (RBF), which assumes that nearby points in the input space have similar function values. However, there are various other kernels like Matern, Exponential, Polynomial, etc.

Effects on Predictions and Uncertainty: The choice of kernel affects how the model extrapolates and interpolates between data points. Some kernels may assume more rigid functions, while others may allow for more complex and wiggly functions. The uncertainty in predictions is influenced by the kernel's behavior, with some kernels leading to more conservative uncertainty estimates than others.

I am using kernel RBF(1.0) which specifies the Radial Basis Function (RBF). The RBF kernel is commonly used for non-linear classification. The parameters for the RBF kernel are the length scale (1.0 in this case). The length scale influences the smoothness of the decision boundary.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Load the California Housing Prices dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, \
                                                    random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)

# Define a Gaussian Process with RBF kernel
kernel = RBF(length_scale=1.0)
gpr = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10,\
                               alpha=1e-08, random_state=42)
```

```
# Fit Gaussian Process Regression model
gpr.fit(X_train_std, y_train)
```

Out[3]:

```
GaussianProcessRegressor(alpha=1e-08, kernel=RBF(length_scale=1),
                         n_restarts_optimizer=10, random_state=42)
```

In [ ]:
```
#Make predictions on the test set
y_pred_mean, y_pred_std = gpr.predict(X_test_std, return_std=True)
```

Let's evaluate our model

In [26]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Evaluate the model on the test set
mse = mean_squared_error(y_test, y_pred_mean)
r2 = r2_score(y_test, y_pred_mean)

print(f'Mean Squared Error (MSE): {mse}')
print(f'R-squared (R2): {r2}')
```

```
Mean Squared Error (MSE): 1.3153237277613075
R-squared (R2): -0.0037505462813642865
```

Well... the accuracy is not good at all. Plus, MSE 1.3 and R2 -0.003 look vey strange. And the level of uncertainty is very high expecially for values that are < 2.

Flexibility in Uncertainty Estimation: Gaussian Processes provide a measure of uncertainty for each prediction. The predicted values are not just point estimates, but they come with a distribution, allowing for a more nuanced understanding of the model's confidence in its predictions.

Adaptability to Data: The model adapts its prediction uncertainty based on the data. In regions where there is a lack of training data

Non-Gaussian Kernels: While the term "Gaussian Process" suggests a Gaussian prior, the model is flexible enough to accommodate non-Gaussian kernels. The choice of a non-Gaussian kernel allows you to capture different assumptions about the functions you are modeling. For example, the Matern kernel allows for more flexibility in modeling functions with different degrees of smoothness.

I have decided to try something else. I will use Matern kernel instead of RBF and do prediction on a smaller data set because I dont want to wait long to see a result. I will use 50% of the data set to train my model instead of 80%.

In [30]:
```
from sklearn.gaussian_process.kernels import Matern

# Load the California Housing Prices dataset
data = fetch_california_housing()
X, y = data.data, data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)
```

```
# Standardize the features
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)

# Use Matern kernel
kernel = Matern(length_scale=1.0, nu=1.5)
gpr = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10, random_state=42)

# Fit Gaussian Process Regression model
gpr.fit(X_train_std, y_train)

# Make predictions on the test set
y_pred_mean, y_pred_std = gpr.predict(X_test_std, return_std=True)
```

In [32]:
```
# Evaluate the model on the test set
mse = mean_squared_error(y_test, y_pred_mean)
r2 = r2_score(y_test, y_pred_mean)

print(f'Mean Squared Error (MSE): {mse}')
print(f'R-squared (R2): {r2}')
```

```
Mean Squared Error (MSE): 0.47355781340141057
R-squared (R2): 0.6427244793764255
```

Lets visualize our prediction vs true values and uncertainties

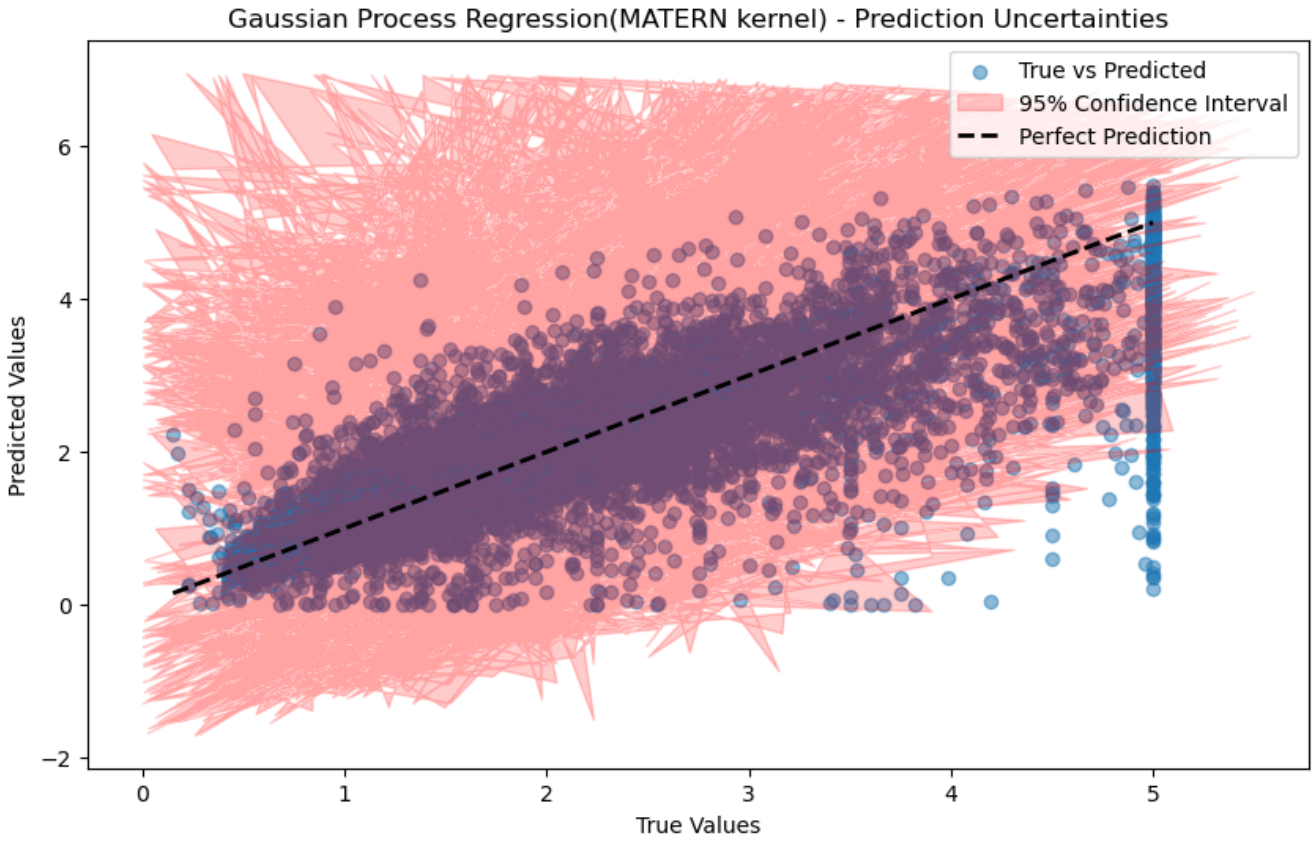
Let's create a shaded region around the predicted mean to visually represent the prediction uncertainties.

This can be done using the fill_between function in Matplotlib.

The Gaussian Process Regression model provides not only the predicted mean values (y_pred_mean) but also the standard deviations of the predictions (y_pred_std). In a Gaussian distribution, approximately 95% of the data falls within 1.96 standard deviations of the mean.

In this example, the fill_between function is used to create a shaded region between y_test - 1.96 *y_pred_std and y_test + 1.96* y_pred_std. This shaded region represents the 95% confidence interval around the predicted mean.

In [33]:
```
# Visualize the prediction uncertainties with shaded region
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_mean, alpha=0.5, label='True vs Predicted')
plt.fill_between(y_pred_mean, y_test - 1.96 * y_pred_std, y_test + 1.96 * y_pred_std,\
                 color='red', alpha=0.2, label='95% Confidence Interval')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], linestyle='--',\
         color='black', linewidth=2, label='Perfect Prediction')
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.title('Gaussian Process Regression(MATERN kernel) - Prediction Uncertainties')
plt.legend()
plt.show()
```



(MSE): 0.47 and R-squared (R2): 0.64 are not a very bad result but there is a very high uncertainty for our predicted values.

Now we can compare our three models.

5. Comparative Analysis

Compare the results obtained from KRR, Bayesian Linear Regression, and Gaussian Process Regression.

Kernel Ridge Regression (KRR)

Mean Squared Error on Test Set: **0.23**

R-squared (R2) Score on Test Set:**0.77**

Bayesian Linear Regression

Mean Squared Error on Test Set: **0.56**

R-squared (R2) Score on Test Set: **0.58**

Gaussian Process Regression

Mean Squared Error on Test Set: **0.47**

R-squared (R2) Score on Test Set: **0.64**

Kernel Ridge Regression (KRR):

Performance: KRR shows the lowest MSE (0.23) and the highest R2 score (0.77), indicating better performance compared to the other models.

Interpretability: KRR's interpretability is related to the choice of the kernel, e.g., the RBF kernel. The RBF kernel controls the smoothness of the function.

Bayesian Linear Regression:

Performance: Bayesian Linear Regression has a higher MSE (0.56) and a lower R2 score (0.58) compared to KRR.

Interpretability: Bayesian Linear Regression provides posterior distributions for coefficients, allowing for uncertainty analysis in predictions.

Gaussian Process Regression (GPR):

Performance: GPR has intermediate performance with MSE of 0.47 and R2 of 0.64.

Interpretability: GPR provides prediction uncertainties, visualized in the scatter plot with error bars.

Overall Observations:

KRR seems to perform the best in terms of both MSE and R2 on the given dataset. Bayesian Linear Regression offers uncertainty estimates for the coefficients. GPR provides prediction uncertainties but has a performance level between KRR and Bayesian Linear Regression.

Computational Efficiency:

The computational efficiency aspect depends on the size of the dataset and the complexity of the models. Generally, KRR is computationally efficient, while GPR is much more computationally demanding.

Ease of Interpretation:

KRR's interpretability is influenced by the choice of the kernel function. Bayesian Linear Regression offers interpretable coefficients with associated uncertainties. GPR provides prediction uncertainties but may be less straightforward to interpret.

2. Outlier detection

In this assignment, you can choose one of the below dataset and your task is to identify the outliers in them using various machine learning techniques.

Dataset

You can choose One of the below datasets:

1. The dataset is a small sample from the Fashion MNIST dataset with manually added outliers. The data will be provided in the form of two numpy arrays: `images` and `labels`. (Note:check for missing data)
2. You will generate your data with outliers:
    - Use NumPy to create data points that follow a normal distribution. This forms the "normal" part of your dataset.
    - Manually add data points that are significantly different from the normal data.
    - These points should be distant from the mean of the normal data to be considered outliers.
    - Combine the normal data points and outliers into a single dataset.

Objectives

1. Perform exploratory data analysis (EDA) to understand the dataset.
2. Implement PCA (Principal Component Analysis) for dimensionality reduction and visualize the results.
3. Use K-means clustering to identify potential outliers.
4. Apply t-SNE (t-Distributed Stochastic Neighbor Embedding) for visualization and detect anomalies.
5. (Optional) Design and train an autoencoder and use reconstruction error to find outliers.
6. Compare the effectiveness of the above methods in outlier detection.

Tasks

1. Exploratory Data Analysis (EDA)

- Load the dataset and visualize some images.
- Plot the distribution of the different classes in the dataset.

2. PCA for Dimensionality Reduction

- Implement PCA to reduce the dimensionality of the dataset.
- Visualize the data in the reduced dimension space.

3. Choose one of the below tasks:

- QDA for Outlier Detection
    - Apply QDA to the dataset.
    - Analyze how the QDA decision boundary help in outlier detection.
- K-means Clustering
    - Apply K-means clustering on the dataset.
    - Identify clusters that potentially contain outliers.

5. t-SNE for Visualization

- Apply t-SNE to the dataset and visualize the results.
- Discuss how t-SNE helps in identifying outliers.

6. Autoencoder for Outlier Detection (Optional)

- Design and train an autoencoder on the dataset.
- Use the reconstruction error to identify images that are outliers.

7. Comparative Analysis

- Compare the results of the models you chose to study.
- Discuss the effectiveness and limitations of each method in outlier detection.

1. Data Exploration and Preprocessing

The Fashion MNIST dataset is a variation of the original MNIST dataset, specifically designed for benchmarking computer vision and machine learning models on fashion-related tasks. It was introduced as an alternative to MNIST to provide a more challenging and realistic dataset for image classification. (The original MNIST dataset is a widely used benchmark dataset in the field of machine learning and computer vision. It stands for "Modified National Institute of Standards and Technology" and is a collection of grayscale images of handwritten digits (0 through 9).)

Images: Similar to the original MNIST, the dataset consists of 28x28-pixel grayscale images. However, instead of handwritten digits, the images represent various types of fashion items.

Categories: The dataset contains 10 categories or classes, each corresponding to a different type of fashion item. The classes are as follows:

0: T-shirt/top **1:** Trouser **2:** Pullover **3:** Dress **4:** Coat **5:** Sandal **6:** Shirt **7:** Sneaker **8:** Bag **9:** Ankle boot

Labels: Each image is associated with a label indicating the category of the fashion item it represents.

Training and Testing Sets: Similar to MNIST, the Fashion MNIST dataset is typically split into a training set for model training and a testing set for evaluating the model's performance.

Usage: Fashion MNIST serves as a more challenging replacement for MNIST and is often used to test and compare the performance of image classification models. It helps researchers and practitioners explore the capabilities of their models in a broader range of applications beyond digit recognition.

In summary, Fashion MNIST is a dataset of fashion-related grayscale images, and it is commonly used for tasks like image classification and pattern recognition in the context of clothing and accessories.

In [4]:
```
#!pip install tensorflow
```

In [3]:
```
import tensorflow as tf
```

```
WARNING:tensorflow:From C:\Users\marai\anaconda3\Lib\site-packages\keras\src\losses.py:29
76: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v
1.losses.sparse_softmax_cross_entropy instead.
```

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Check for missing data
missing_data_train = np.isnan(train_images).any()
missing_data_test = np.isnan(test_images).any()

if missing_data_train or missing_data_test:
    print("There is missing data in the dataset.")
else:
    print("No missing data in the dataset.")

# Further processing or visualization can be done here...
```

```
No missing data in the dataset.
```

In [9]:
```
train_images.shape
```

Out[9]:
```
(60000, 28, 28)
```

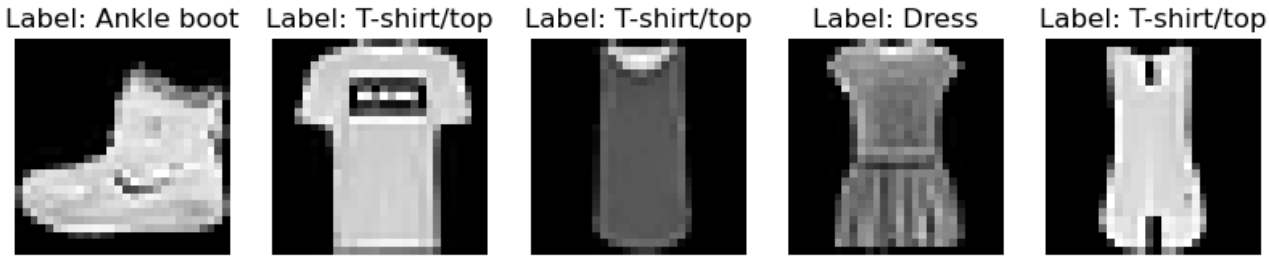
In [10]:
```
test_images.shape
```

Out[10]:
```
(10000, 28, 28)
```

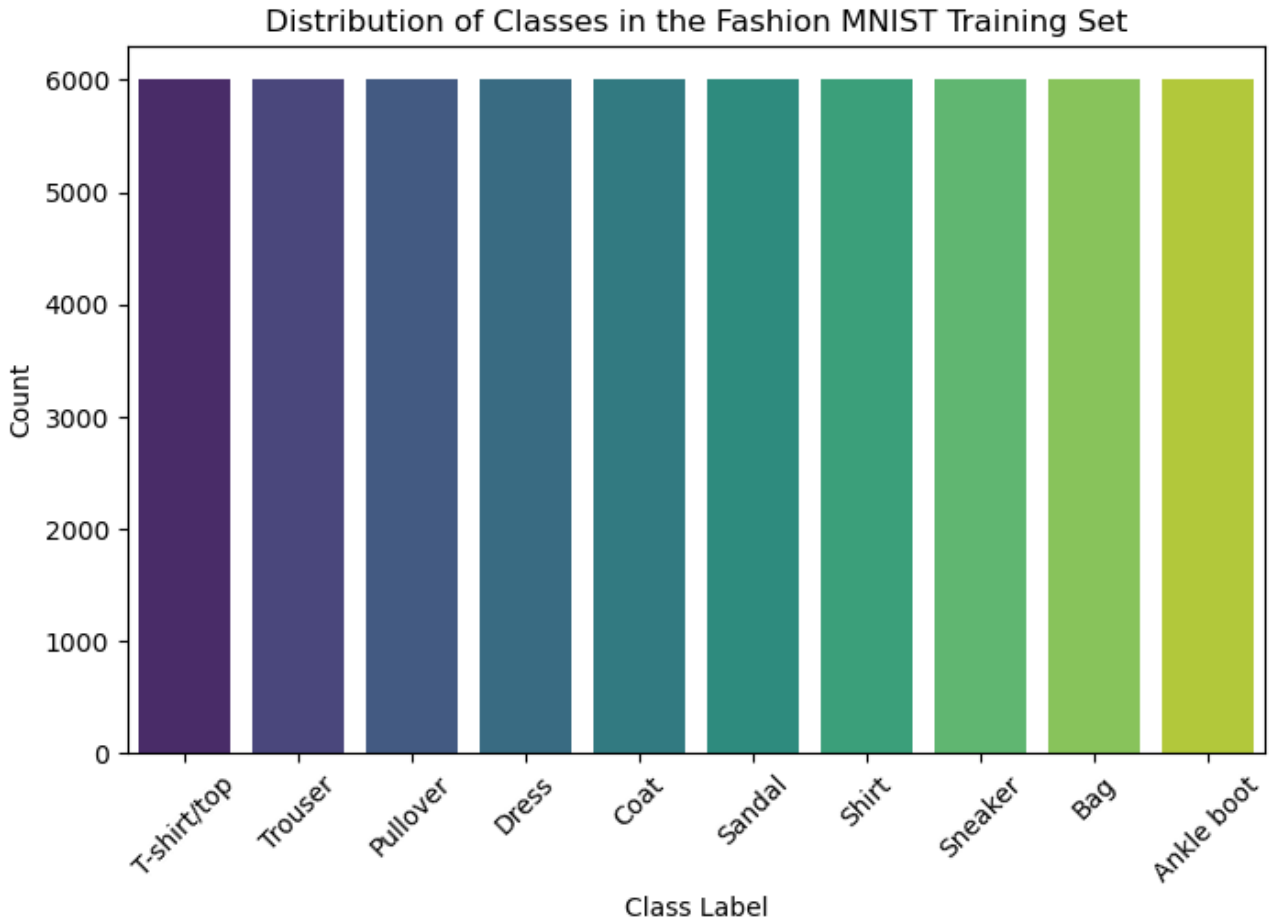
Visualize some images

In [13]:
```
# Map label indices to class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
               'Bag', 'Ankle boot']

# Display the first 5 images with labels
plt.figure(figsize=(10, 4))
for i in range(5):
```

```
    plt.subplot(1, 5, i + 1)
    plt.imshow(train_images[i], cmap='gray')
    plt.title(f'Label: {class_names[train_labels[i]]}')
    plt.axis('off')
plt.show()
```



In [12]:
```
# Task 2: Plot the distribution of different classes

# Plot the distribution of classes in the training set
plt.figure(figsize=(8, 5))
sns.countplot(x=train_labels, palette="viridis")
plt.title('Distribution of Classes in the Fashion MNIST Training Set')
plt.xlabel('Class Label')
plt.ylabel('Count')
plt.xticks(ticks=np.arange(10), labels=class_names, rotation=45)
plt.show()
```



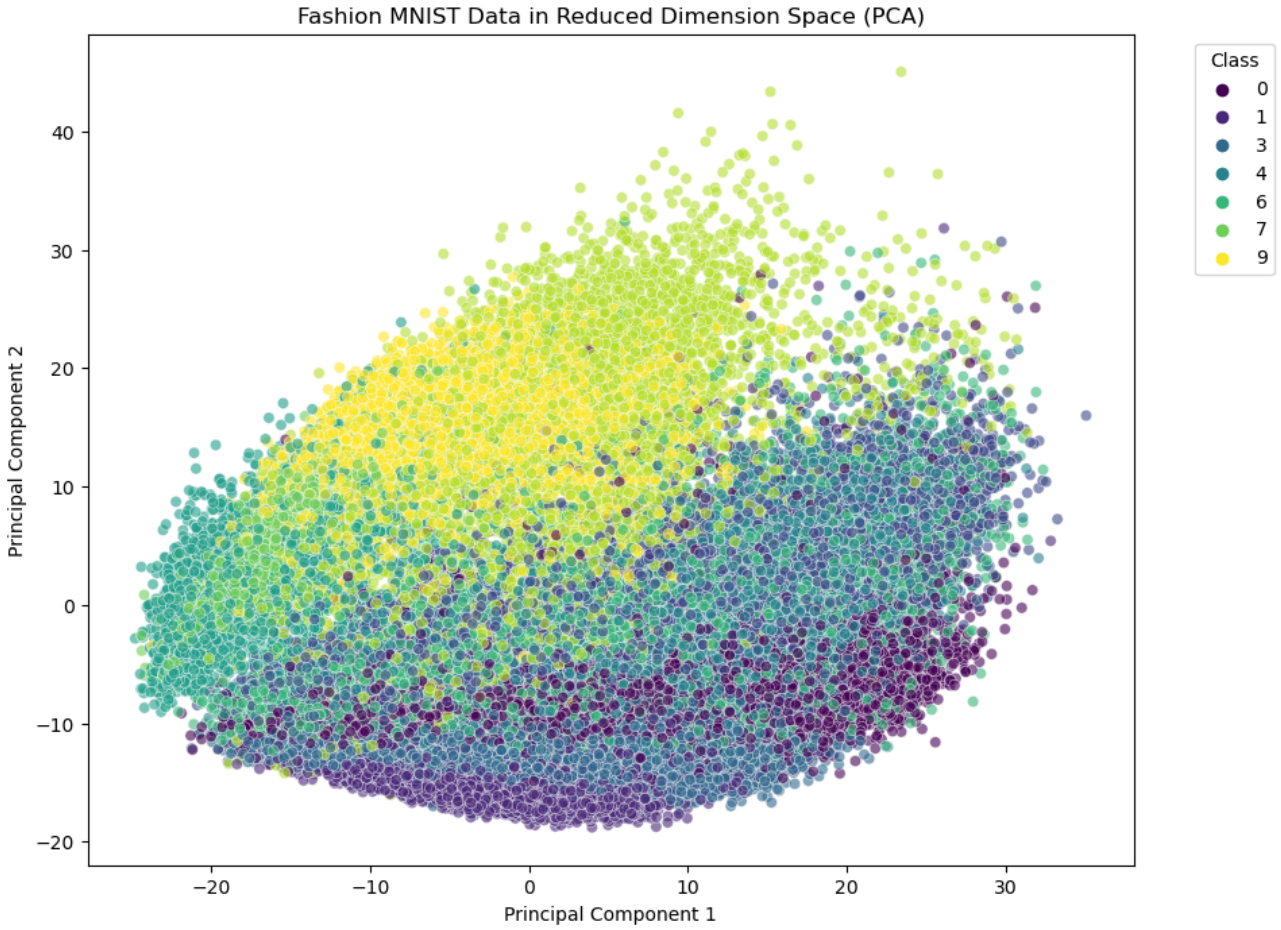
Therefore, every class consists of exactly of 6000 of images.

2. PCA for Dimensionality Reduction

Principal Component Analysis (PCA) is a technique for dimensionality reduction that aims to transform a dataset into a new coordinate system, where the axes are the principal components (PCs) that capture the maximum variance in the data. PCA is widely used in various fields, including statistics, machine learning, and data visualization.

Let's visualize the data in the reduced dimension space 2D for that PCA is using 2 components and hue=train_labels is used in the sns.scatterplot function to color the points based on their true labels. This will provide a visual representation of how different classes are distributed in the PCA plotchart.

In [91]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
import seaborn as sns
import tensorflow as tf

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Flatten the images to use as input for PCA
train_images_flat = train_images.reshape(train_images.shape[0], -1)

# Standardize the data (optional, but often recommended for PCA)
train_images_standardized = (train_images_flat - np.mean(train_images_flat, axis=0)) / np.

# Apply PCA with the optimal number of components
pca = PCA(n_components=2)
train_images_pca = pca.fit_transform(train_images_standardized)

# Visualize the data in the reduced dimension space
plt.figure(figsize=(10, 8))
sns.scatterplot(x=train_images_pca[:, 0], y=train_images_pca[:, 1], hue=train_labels, /
                palette="viridis", alpha=0.6)
plt.title(f'Fashion MNIST Data in Reduced Dimension Space (PCA)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Class', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Just optionally I will just use 2 components of PCA and try to predict on Test data and check the accuracy. For that reason I applied just a simple Logistic Regression.

In [31]:
```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import tensorflow as tf

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(_, _), (test_images, test_labels) = fashion_mnist.load_data()

# Flatten and standardize the test images
test_images_flat = test_images.reshape(test_images.shape[0], -1)
test_images_standardized = (test_images_flat - np.mean(test_images_flat, axis=0)) / np.sto

# Apply PCA with the optimal number of components
pca = PCA(n_components=optimal_components)
train_images_flat = train_images.reshape(train_images.shape[0], -1)
train_images_standardized = (train_images_flat - np.mean(train_images_flat, axis=0)) / np.
train_images_pca = pca.fit_transform(train_images_standardized)
test_images_pca = pca.transform(test_images_standardized)

# Train a logistic regression classifier on the reduced dimension training data
classifier = LogisticRegression()
classifier.fit(train_images_pca, train_labels)

# Make predictions on the test data
```

```
predictions = classifier.predict(test_images_pca)

# Evaluate the accuracy
accuracy = accuracy_score(test_labels, predictions)
print(f'Accuracy on the test data LogisticRegression with PCA: {accuracy:.2%}')
```

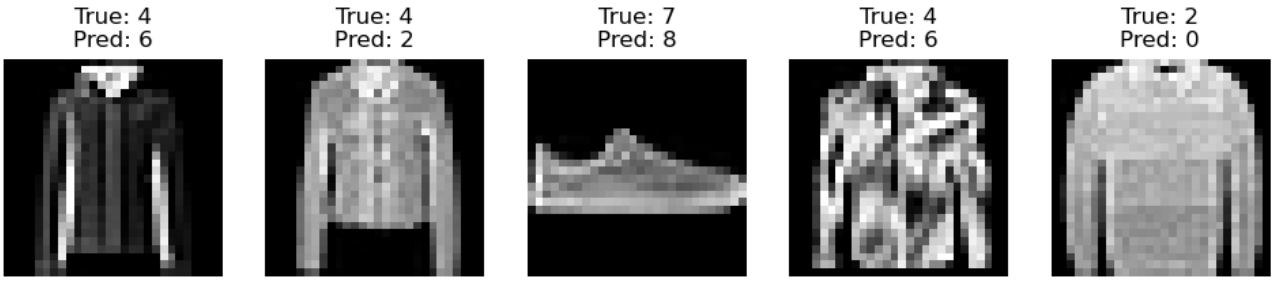
```
Accuracy on the test data LogisticRegression with PCA: 74.54%
```

```
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:460: Converg
enceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

The accuracy is not very bad 74.54%

Below you can see images which were dismatched by our model.

In [19]:
```
# Find indices where predictions do not match true labels
mismatch_indices = np.where(predictions != test_labels)[0]

# Display a few mismatched images
plt.figure(figsize=(12, 6))
for i, idx in enumerate(mismatch_indices[:5]):
    plt.subplot(1, 5, i + 1)
    plt.imshow(test_images[idx], cmap='gray')
    plt.title(f'True: {test_labels[idx]}\nPred: {predictions[idx]}')
    plt.axis('off')

plt.show()
```



In [33]:
```
# Apply Logistic Regresion on data without PCA
qd = LogisticRegression()
qd.fit(train_images_standardized, train_labels)

# Make predictions on the test data
predictions = qd.predict(test_images_standardized)

# Evaluate the accuracy
accuracy = accuracy_score(test_labels, predictions)
print(f'Accuracy on the test data Logistic Regression without PCA: {accuracy:.2%}')
```

```
Accuracy on the test data Logistic Regression without PCA: 84.34%
```

```
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:460: Converg
enceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

3a QDA (Quadratic Discriminant Analysis)

Quadratic Discriminant Analysis (QDA) is a classification and dimensionality reduction technique that falls under the broader category of discriminant analysis. QDA is closely related to Linear Discriminant Analysis (LDA), but it relaxes the assumption that the covariance matrix of each class is the same, allowing for different covariance matrices for each class.

Objective:

Like LDA, the primary goal of QDA is to find a decision boundary that best separates different classes in a dataset.

Assumption:

QDA assumes that the data for each class follows a multivariate normal distribution.

Decision Boundary:

QDA models the decision boundary as a quadratic function, allowing for more flexibility in capturing the shape of the decision boundary compared to the linear decision boundary in LDA.

Covariance Matrices:

Unlike LDA, QDA does not assume equal covariance matrices for all classes. Each class is allowed to have its own covariance matrix

Parameter Estimation:

QDA involves estimating parameters such as class priors, class means, and covariance matrices based on the training data.

Dimensionality Reduction:

QDA can be used for dimensionality reduction by transforming the original features into a lower-dimensional space that retains most of the discriminatory information.

Classification:

Once trained, QDA can be used for classification by assigning new data points to the class that maximizes the class posterior probability.

Use Cases:

QDA is suitable when the assumption of different covariance matrices for each class is more appropriate than assuming equal covariances (as in LDA). It can be effective when dealing with non-linear decision boundaries.

Let's apply first PCA and then QDA and visualize Decision boundaries and outliers (Red cross)

To find outliers I used the **Mahalanobis distance**. It is used as a measure of the distance between a data point and the center of a distribution. The Mahalanobis distance takes into account the correlations between variables and the variances along each dimension. In the context of QDA, the Mahalanobis distance is used to quantify how far a data point is from the center of its assigned class distribution. Larger Mahalanobis distances suggest that a point is further from the center of the distribution and may be considered as an outlier.

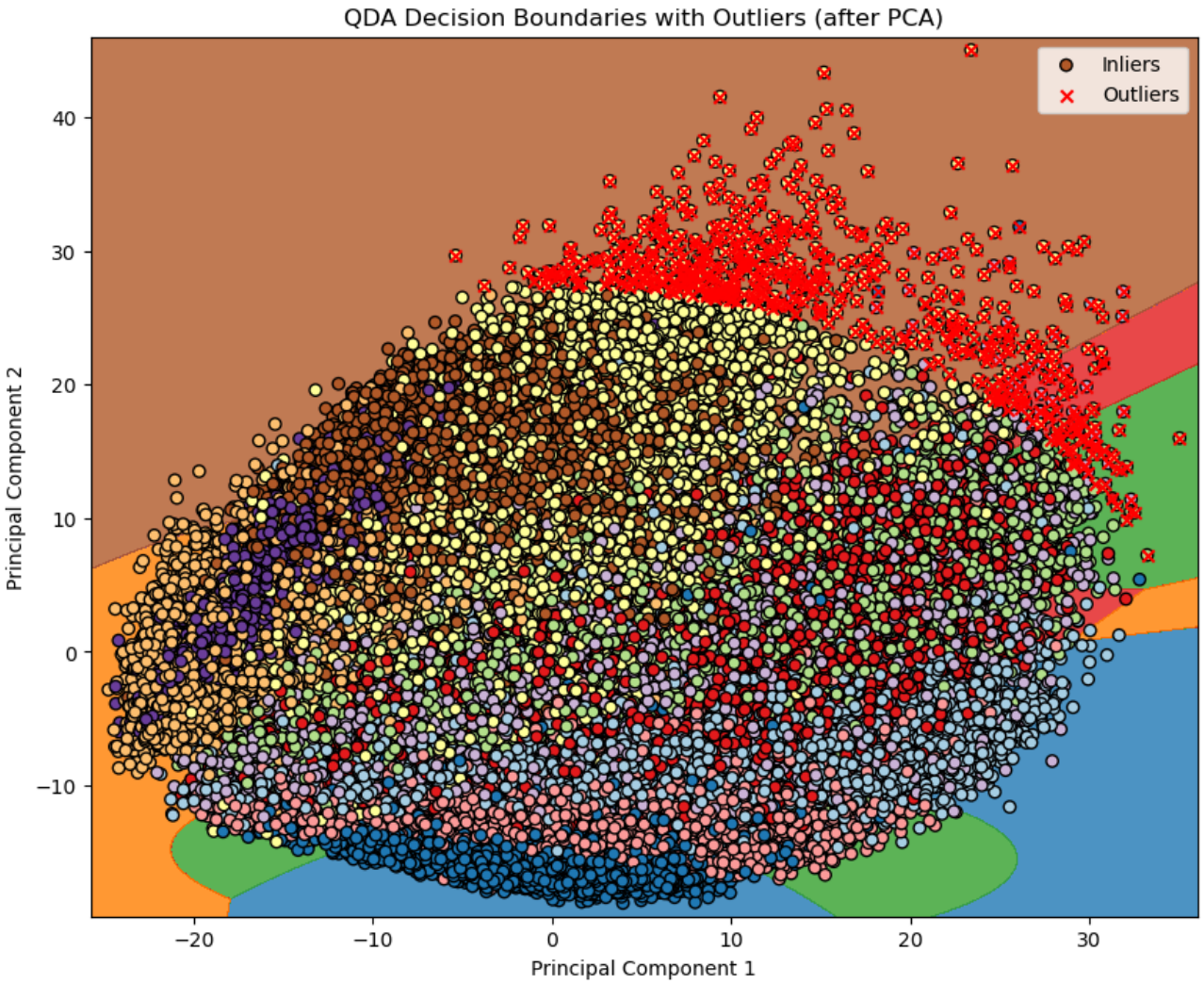
The threshold for identifying outliers is often set based on a percentile of the Mahalanobis distances, such as the 95th or 99th percentile, depending on the desired level of sensitivity to outliers.

In the provided code example, the Mahalanobis distances are calculated for each point in the reduced feature space obtained after PCA. The threshold is set at the **99th** percentile to identify potential outliers. Points with Mahalanobis distances above this threshold are considered outliers.

In [103…

```
import numpy as np
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from scipy.spatial.distance import mahalanobis
import tensorflow as tf
import matplotlib.pyplot as plt

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Flatten the images
train_images_flat = train_images.reshape(train_images.shape[0], -1)
test_images_flat = test_images.reshape(test_images.shape[0], -1)

# Standardize the data
scaler = StandardScaler()
train_images_standardized = scaler.fit_transform(train_images_flat)
test_images_standardized = scaler.transform(test_images_flat)

# Apply PCA for dimensionality reduction
pca = PCA(n_components=2)
train_images_pca = pca.fit_transform(train_images_standardized)

# Apply QDA for classification
qda = QuadraticDiscriminantAnalysis()
qda.fit(train_images_pca, train_labels)

# Calculate Mahalanobis distances for each point
cov_matrix = np.cov(train_images_pca, rowvar=False)
inv_cov_matrix = np.linalg.inv(cov_matrix)
mean = np.mean(train_images_pca, axis=0)
```

```
mahalanobis_distances = np.array([mahalanobis(point, mean, inv_cov_matrix) for point in tr

# Set a threshold for identifying outliers (you can adjust this)
outlier_threshold = np.percentile(mahalanobis_distances, 99)
outliers = train_images_pca[mahalanobis_distances > outlier_threshold]

# Visualize decision boundaries and outliers
plt.figure(figsize=(10, 8))

# Plot decision boundaries
h = .02
x_min, x_max = train_images_pca[:, 0].min() - 1, train_images_pca[:, 0].max() + 1
y_min, y_max = train_images_pca[:, 1].min() - 1, train_images_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = qda.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

# Scatter plot of the data points
plt.scatter(train_images_pca[:, 0], train_images_pca[:, 1], c=train_labels, edgecolors='k
            cmap=plt.cm.Paired, label='Inliers')
plt.scatter(outliers[:, 0], outliers[:, 1], c='red', marker='x', label='Outliers')
plt.title('QDA Decision Boundaries with Outliers (after PCA)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()

plt.show()
```

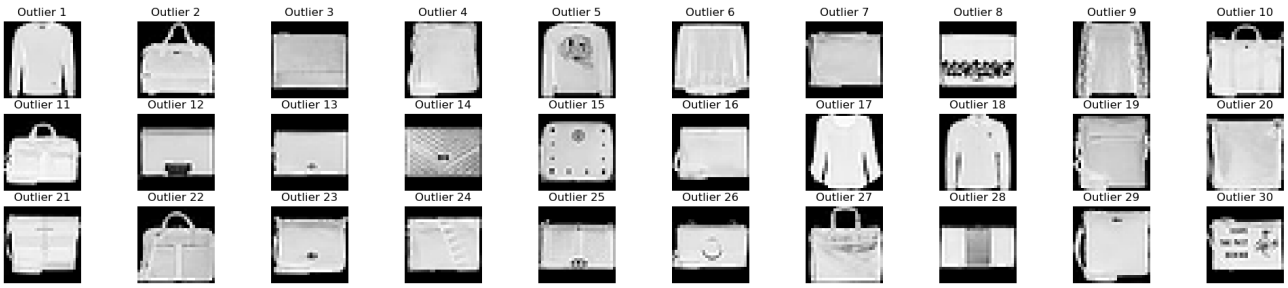


Below there are images of some outliers.

In [110…

```
# Find the indices of outliers in the original dataset
outlier_indices = np.where(mahalanobis_distances > outlier_threshold)[0]

# Visualize decision boundaries and outliers
plt.figure(figsize=(10, 8))

# Plot the outlier images
plt.figure(figsize=(25, 5))
for i, outlier_index in enumerate(outlier_indices[:30]):  # Displaying the first 5 outlier
    plt.subplot(3, 10, i + 1)
    plt.imshow(train_images[outlier_index], cmap='gray')
    plt.title(f'Outlier {i + 1}')
    plt.axis('off')

plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```



3b K-means Clustering

K-means clustering is a popular unsupervised machine learning algorithm used for partitioning a dataset into a set of distinct, non-overlapping groups or clusters. The objective of K-means is to group data points into clusters in such a way that the sum of squared distances from each point to the center of its assigned cluster (centroid) is minimized.

To identify clusters that potentially contain outliers, we can consider clusters with a smaller number of data points as potential candidates. These clusters might represent regions where data points deviate from the majority. After applying K-means clustering to the test data, potential outlier clusters are identified based on their small size relative to the total number of test data points. These potential outlier clusters are highlighted in the visualization using red "x" markers. The threshold for identifying potential outlier clusters less than 99.6

In [4]:
```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import tensorflow as tf

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, _), (test_images, _) = fashion_mnist.load_data()

# Flatten and standardize the data
train_images_flat = train_images.reshape(train_images.shape[0], -1)
test_images_flat = test_images.reshape(test_images.shape[0], -1)
train_images_standardized = (train_images_flat - np.mean(train_images_flat, axis=0)) / np.
test_images_standardized = (test_images_flat - np.mean(test_images_flat, axis=0)) / np.sto

# Combine training and test data
all_images_standardized = np.concatenate((train_images_standardized, test_images_standardi
```

I will try to use Silhouette method

The silhouette method is typically used for determining the optimal number of clusters in algorithms like K-means. This method can be expressed as: 1) For each value of K (number of clusters), perform K-means clustering on your dataset.

2) For each data point, calculate the silhouette score, which is a measure of how similar it is to its own cluster compared to other clusters.

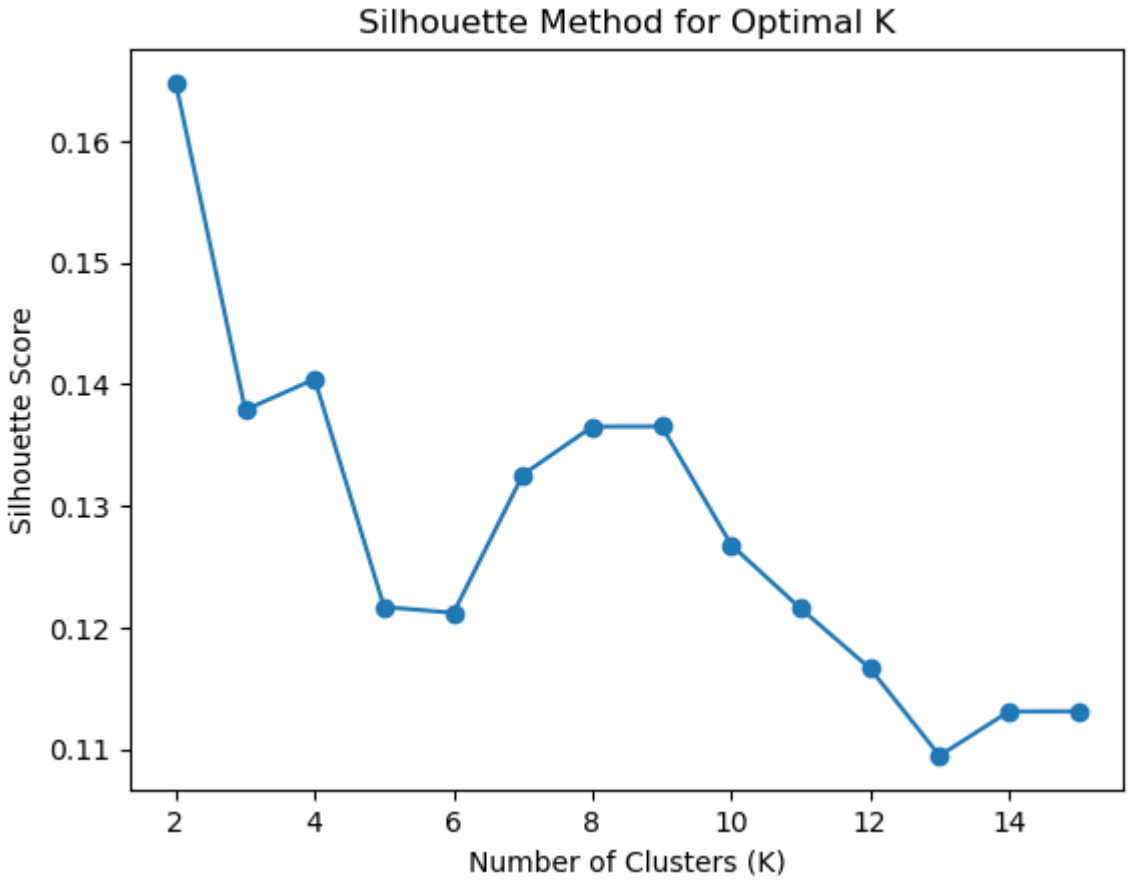
3) Calculate the average silhouette score for each value of K.

4) Choose the K that maximizes the average silhouette score.

In [2]:
```
from sklearn.metrics import silhouette_score

# Initialize a list to store silhouette scores
silhouette_scores = []

# Try different values of k
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(all_images_standardized)

    # Calculate silhouette score
    silhouette_avg = silhouette_score(all_images_standardized, labels)
    silhouette_scores.append(silhouette_avg)

# Plot the silhouette scores
plt.plot(range(2, 16), silhouette_scores, marker='o')
plt.title('Silhouette Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Silhouette Score')
plt.show()
```

```
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\joblib\externals\loky\backend\context.py:110:
UserWarning: Could not find the number of physical cores for the following reason:
found 0 physical cores < 1
Returning the number of logical cores instead. You can silence this warning by setting LO
KY_MAX_CPU_COUNT to the number of cores you want to use.
  warnings.warn(
  File "C:\Users\marai\anaconda3\Lib\site-packages\joblib\externals\loky\backend\context.
py", line 217, in _count_physical_cores
    raise ValueError(
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
```

```
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```



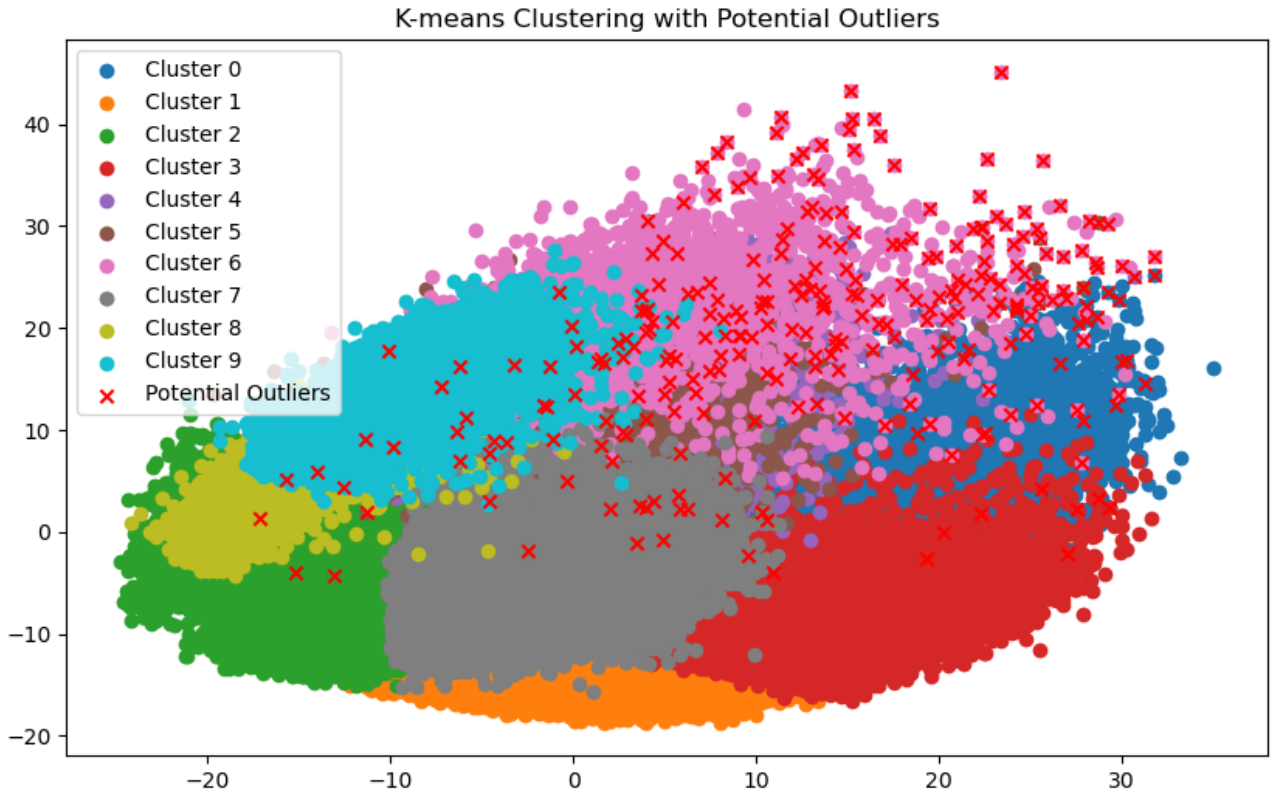
Well, the method showed that the best is one cluster with the highest Silhouette score. But in our case we know that there are 10 classes and we will assign the class to be 10 to see if we can find any outliers out of our 10 classes.

Let's apply KMeans with 10 clusters.

In [3]:
```
# Choose the number of clusters (K)
k = 10

# Apply K-means clustering
kmeans = KMeans(n_clusters=k, random_state=42)
labels = kmeans.fit_predict(all_images_standardized)

# Identify potential outliers based on distance to centroid
distances = np.zeros(all_images_standardized.shape[0])
for i in range(k):
    cluster_points = all_images_standardized[labels == i]
    centroid = kmeans.cluster_centers_[i]
    distances[labels == i] = np.linalg.norm(cluster_points - centroid, axis=1)

# Set a threshold for identifying outliers
outlier_threshold = np.percentile(distances, 99.6)
outliers = all_images_standardized[distances > outlier_threshold]
outliers_indices = np.where(distances > outlier_threshold)[0]
```

```
# Perform PCA for dimensionality reduction
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(all_images_standardized)

# Visualize clusters
plt.figure(figsize=(10, 6))
for i in range(k):
    cluster_points = reduced_data[labels == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster {i}')

# Highlight potential outliers
outliers_reduced = pca.transform(outliers)
plt.scatter(outliers_reduced[:, 0], outliers_reduced[:, 1], color='red', marker='x', label

plt.title('K-means Clustering with Potential Outliers')
plt.legend()
plt.show()

print("Number of Potential Outliers:", len(outliers_indices))
```

```
C:\Users\marai\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `
n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```



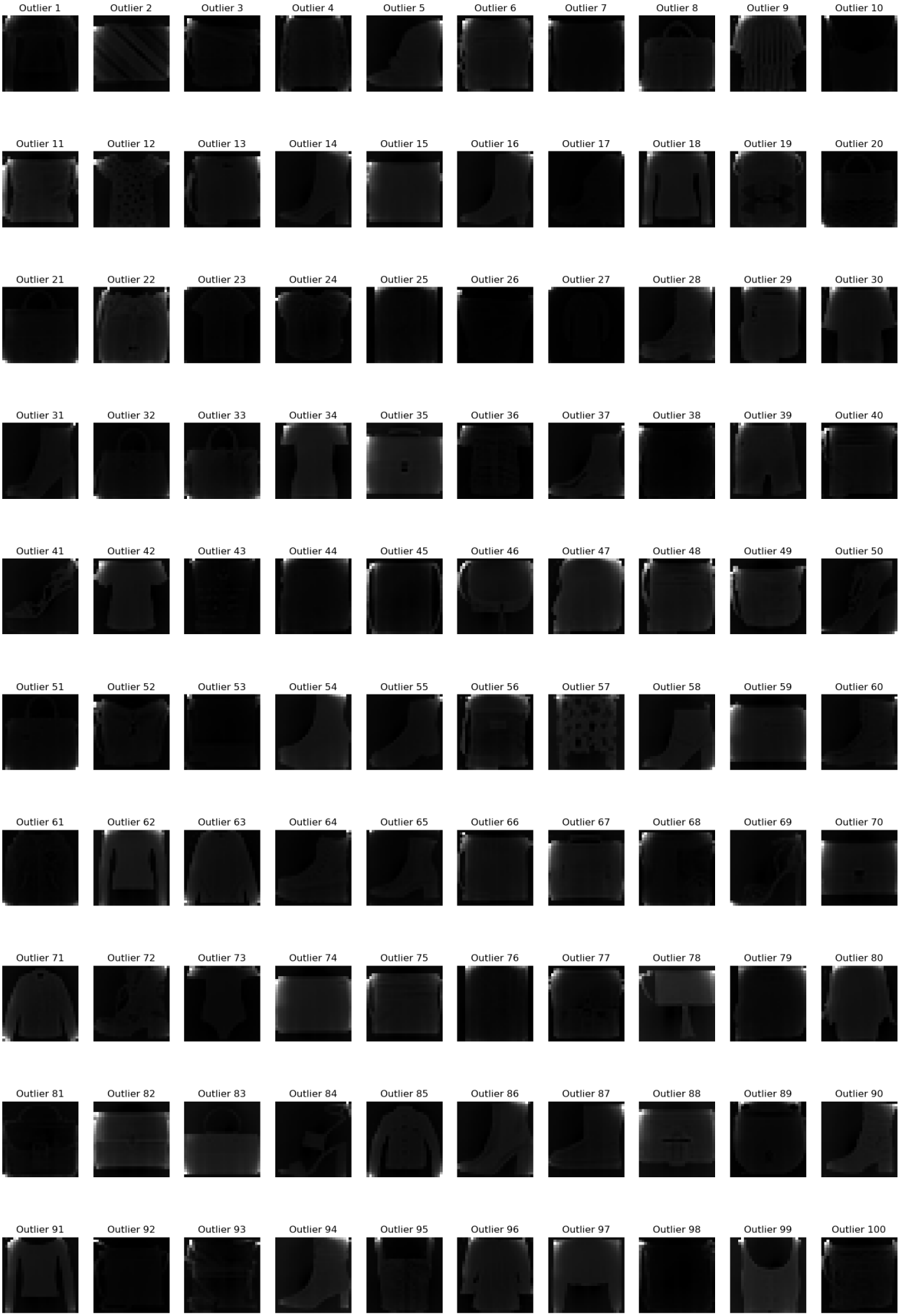

```
Number of Potential Outliers: 280
```

I used PCA reduction method to visualize the result in 2D space. Once again, this code is setting a threshold to identify potential outliers in the data based on the distances of points to their cluster centroids in a K-means clustering context. The threshold is a statistical measure indicating that 99.6% of the distances fall below this value. Points with distances above this threshold are considered potential outliers. **Number of Potential Outliers: 280**

Let's vizualize some images of those outliers

In [118…

```
import matplotlib.pyplot as plt

# Reshape outliers to the original image dimensions
outliers_images = outliers[:100]

# Display the 20 outlier images
plt.figure(figsize=(20, 30))
for i in range(len(outliers_images)):
    plt.subplot(10, 10, i+1)  # Adjust the number of subplots accordingly
    plt.imshow(outliers_images[i].reshape(28, 28), cmap='gray')
    plt.title(f'Outlier {i+1}')
    plt.axis('off')

plt.show()
```


Outlier 1
Outlier 2
Outlier 3
Outlier 4
Outlier 5
Outlier 6
Outlier 7
Outlier 8
Outlier 9
Outlier 10
Outlier 11
Outlier 12
Outlier 13
Outlier 14
Outlier 15
Outlier 16
Outlier 17
Outlier 18
Outlier 19
Outlier 20
Outlier 21
Outlier 22
Outlier 23
Outlier 24
Outlier 25
Outlier 26
Outlier 27
Outlier 28
Outlier 29
Outlier 30
Outlier 31
Outlier 32
Outlier 33
Outlier 34
Outlier 35
Outlier 36
Outlier 37
Outlier 38
Outlier 39
Outlier 40
Outlier 41
Outlier 42
Outlier 43
Outlier 44
Outlier 45
Outlier 46
Outlier 47
Outlier 48
Outlier 49
Outlier 50
Outlier 51
Outlier 52
Outlier 53
Outlier 54
Outlier 55
Outlier 56
Outlier 57
Outlier 58
Outlier 59
Outlier 60
Outlier 61
Outlier 62
Outlier 63
Outlier 64
Outlier 65
Outlier 66
Outlier 67
Outlier 68
Outlier 69
Outlier 70
Outlier 71
Outlier 72
Outlier 73
Outlier 74
Outlier 75
Outlier 76
Outlier 77
Outlier 78
Outlier 79
Outlier 80
Outlier 81
Outlier 82
Outlier 83
Outlier 84
Outlier 85
Outlier 86
Outlier 87
Outlier 88
Outlier 89
Outlier 90
Outlier 91
Outlier 92
Outlier 93
Outlier 94
Outlier 95
Outlier 96
Outlier 97
Outlier 98
Outlier 99
Outlier 100

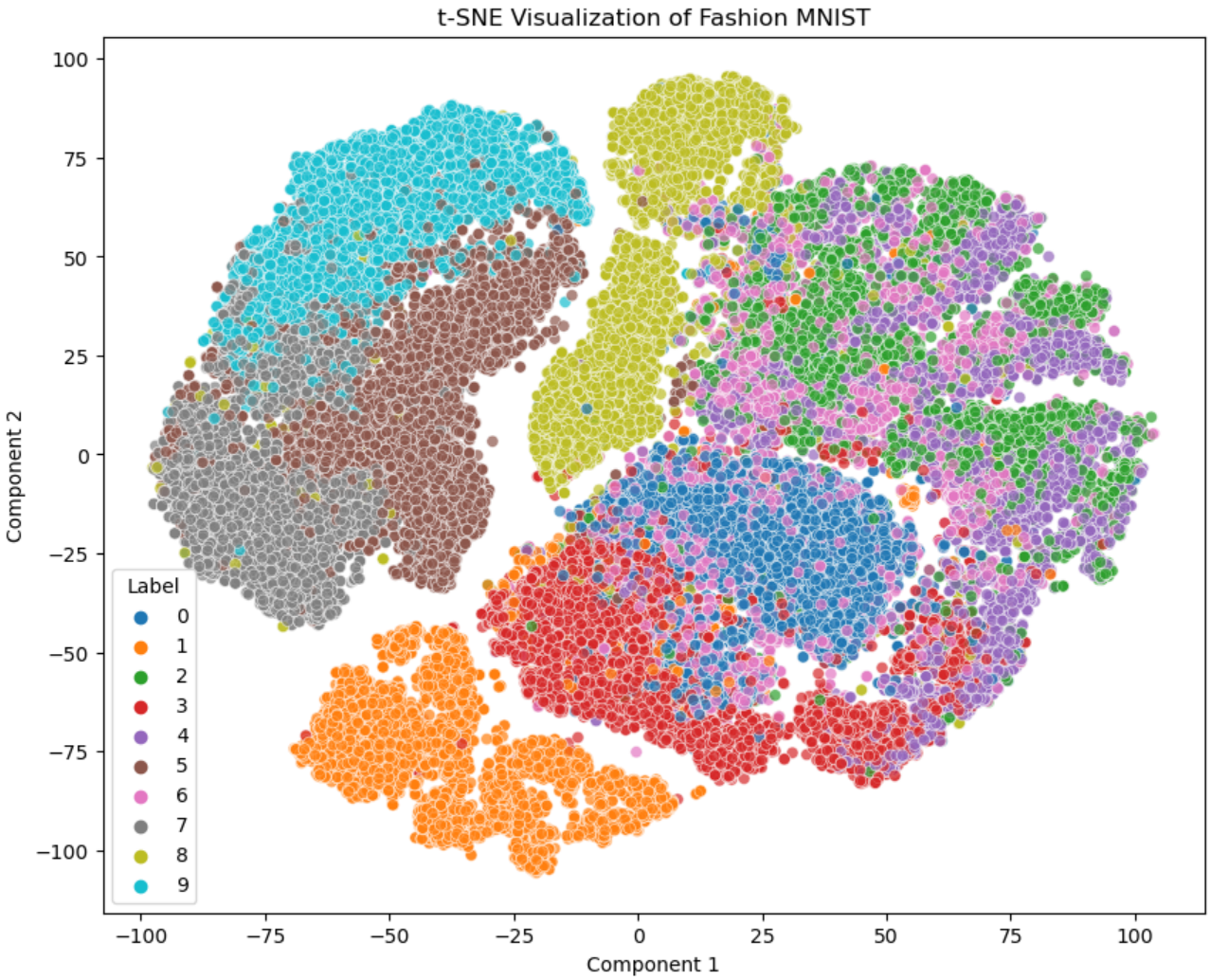
Most of those images are black and with a very bad visual picture. I think that our KMmen algorithm did a very good job by identifying outliers

5. t-SNE for Visualization

t-SNE, or t-Distributed Stochastic Neighbor Embedding, is a dimensionality reduction technique commonly used for visualizing high-dimensional data in lower-dimensional spaces. It is particularly effective for capturing the local and non-linear structure of data, making it useful for visualizing complex relationships and patterns.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import seaborn as sns
import tensorflow as tf

# Load Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Flatten and standardize the data
train_images_flat = train_images.reshape(train_images.shape[0], -1)
test_images_flat = test_images.reshape(test_images.shape[0], -1)

train_images_standardized = (train_images_flat - np.mean(train_images_flat, axis=0)) / np.
test_images_standardized = (test_images_flat - np.mean(train_images_flat, axis=0)) / np.st

# Combine training and test data
all_images_standardized = np.concatenate((train_images_standardized, test_images_standardi
all_labels = np.concatenate((train_labels, test_labels), axis=0)

# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42)
all_images_tsne = tsne.fit_transform(all_images_standardized)
```

In [7]:
```
# Create a DataFrame for visualization with Seaborn
import pandas as pd

df_tsne = pd.DataFrame(data=all_images_tsne, columns=['Component 1', 'Component 2'])
df_tsne['Label'] = all_labels

# Visualize t-SNE result
plt.figure(figsize=(10, 8))
sns.scatterplot(x='Component 1', y='Component 2', hue='Label', palette='tab10', data=df_ts
plt.title('t-SNE Visualization of Fashion MNIST')
plt.legend(title='Label')
plt.show()
```



We see our data in 2D space with true labeling.

DBSCAN method

DBSCAN, which stands for Density-Based Spatial Clustering of Applications with Noise, is a clustering algorithm commonly used in machine learning and data analysis. Unlike K-means, DBSCAN does not require specifying the number of clusters beforehand and is particularly effective at finding clusters of arbitrary shapes.

(DBSCAN defines clusters as dense regions of data points separated by sparser regions. It identifies clusters based on the density of data points in the feature space.)

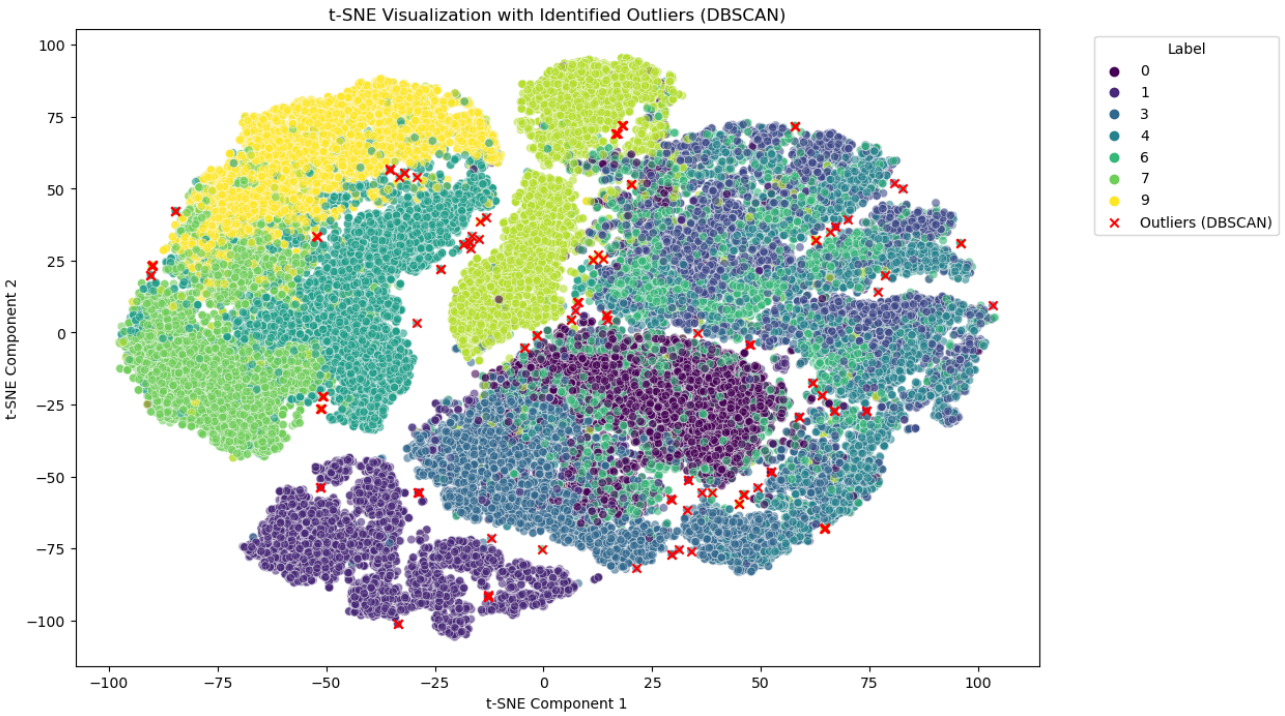
Finding outliers in t-SNE data involves identifying data points that deviate significantly from the overall pattern or clusters. One common approach is to use clustering algorithms, such as DBSCAN, to detect outliers based on density.

Here's an example of how you can use DBSCAN to find outliers in your t-SNE-transformed data:

In this code, hue=all_labels is used in the sns.scatterplot function to color the points based on their labels. This will provide a visual representation of how different classes are distributed in the t-SNE

plot.

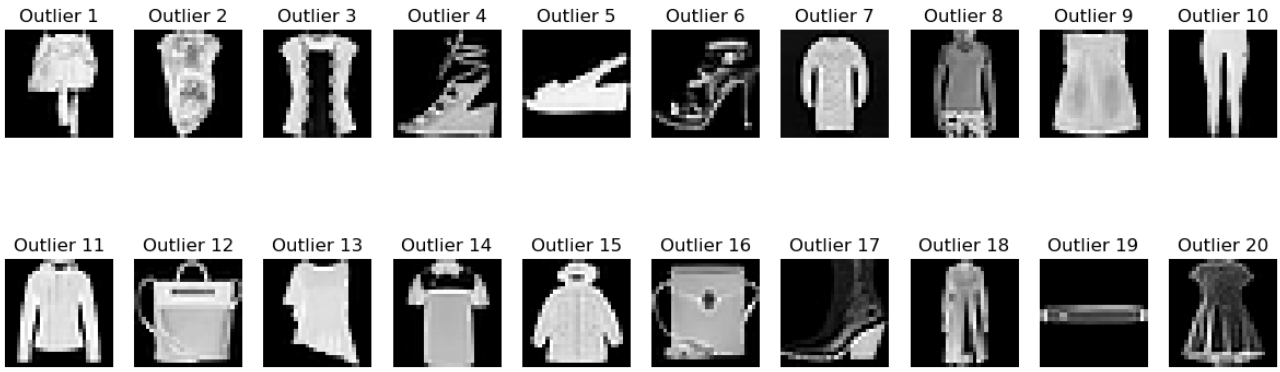
In [88]:
```
from sklearn.cluster import DBSCAN

# Apply DBSCAN to t-SNE-transformed data
dbscan = DBSCAN(eps=2, min_samples=5)
labels = dbscan.fit_predict(all_images_tsne)

# Extract outliers based on DBSCAN labels
outlier_mask = labels == -1
outliers = all_images_tsne[outlier_mask]

# Visualize the t-SNE plot with outliers highlighted
plt.figure(figsize=(12, 8))
sns.scatterplot(x=all_images_tsne[:, 0], y=all_images_tsne[:, 1], hue=all_labels, palette=
plt.scatter(outliers[:, 0], outliers[:, 1], color='red', marker='x', label='Outliers (DBSC
plt.title('t-SNE Visualization with Identified Outliers (DBSCAN)')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.legend(title='Label', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



The **eps** parameter determines the maximum distance between two samples for one to be considered in the neighborhood of the other, and **min_samples** is the minimum number of samples in a neighborhood for a point to be considered a core point.

In [89]:
```
# Extract indices of outliers
outlier_indices = np.where(outlier_mask)[0]

# Extract corresponding images
outlier_images = np.concatenate((train_images, test_images), axis=0)[outlier_indices]

# Display the images of outliers
plt.figure(figsize=(15, 5))
for i, image in enumerate(outlier_images[:20]):  # Displaying the first 20 outliers
```

```
    plt.subplot(2, 10, i+1)
    plt.imshow(image, cmap='gray')
    plt.title(f'Outlier {i+1}')
    plt.axis('off')

plt.show()
```



6. Comparative analysis

K-means Clustering:

Effectiveness:

K-means clustering is primarily designed for partitioning data into clusters, and it may not be the most suitable method for outlier detection. The method relies on Euclidean distance and assumes spherical clusters, which may not capture complex cluster shapes.

Limitations:

Sensitivity to the number of clusters (K) chosen. Assumes clusters are of similar size and density. Outliers may be assigned to the nearest cluster, but they won't be explicitly labeled as outliers.

t-SNE Visualization:

Effectiveness:

t-SNE is effective for visualizing high-dimensional data in 2D or 3D space, allowing the identification of clusters and potential outliers.

Limitations:

t-SNE does not explicitly provide outlier labels; it is a visualization tool. The effectiveness may vary based on the choice of hyperparameters.

DBSCAN (Density-Based Clustering):

Effectiveness:

DBSCAN is suitable for identifying outliers based on data density, and it explicitly labels points as outliers. It can handle clusters of different shapes and sizes.

Limitations:

Sensitive to the choice of hyperparameters, such as eps and min_samples. Performance may degrade in high-dimensional spaces.

Quadratic Discriminant Analysis (QDA):

Effectiveness:

QDA is a classification algorithm, and outliers may be identified based on classification confidence or distance from class centroids.

Limitations:

Assumes that data is normally distributed within each class. May not perform well with high-dimensional data.

Conclusion:

t-SNE provides powerful visualization but does not explicitly label outliers. DBSCAN, on the other hand, explicitly identifies outliers based on density.

DBSCAN and t-SNE are sensitive to the choice of hyperparameters. Tuning these parameters is crucial for obtaining meaningful results.

DBSCAN's performance may degrade in high-dimensional spaces. t-SNE, while effective for visualization, may not be suitable for high-dimensional outlier detection.

Each method makes certain assumptions about the data distribution. K-means assumes spherical clusters, QDA assumes normal distribution within classes, etc.

I think that for explicit outlier labeling, it's better to consider methods like DBSCAN with t-SNE for visualization and cluster exploration. Those methods are very convenient to use, however in my case Kmeans performed better in finding true outliers, so it's all depends on data and a goal you are pursuing.