lab-02

December 17, 2023

1 Assignment 2

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.

2.0.1 Dataset

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

2.0.2 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.

2.1 Tasks

2.1.1 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.

2.2 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.

2.3 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)

[39]:
```python
import sys
import subprocess
subprocess.Popen('caffeinate')

if 'darwin' in sys.platform:
    print('Running \'caffeinate\' on MacOSX to prevent the system from␣
 ↪sleeping')
    subprocess.Popen('caffeinate')
```

```
Running 'caffeinate' on MacOSX to prevent the system from sleeping
```

[41]:
```python
import pandas as pd
import numpy as np
import math
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
from sklearn.linear_model import BayesianRidge
import warnings
warnings.filterwarnings('ignore')
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C
from sklearn.gaussian_process.kernels import Matern
from joblib import dump, load
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.manifold import TSNE
from sklearn.neighbors import NearestNeighbors
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import legacy


from sklearn.datasets import fetch_california_housing
```

3 1.1

```
[6]: california_housing = fetch_california_housing()
ch_df = pd.DataFrame(data=california_housing.data, columns=california_housing.
 ↪feature_names)

hv_df = pd.DataFrame(data=california_housing.target, columns=['MedHouseVal'])


scaler = StandardScaler()
scaled_features = scaler.fit_transform(ch_df)
scaled_features_df = pd.DataFrame(scaled_features, columns=ch_df.columns)

X_train, X_test, y_train, y_test = train_test_split(scaled_features_df, hv_df,␣
 ↪test_size=0.2, random_state=42)
feature_names = scaled_features_df.columns
```

In previous lab I concluded that your way of transforming the data was a better way then my own, so i will use your code for the transormation.

4 1.2.1 Kernel Ridge Regression (KRR)

[23]:
```python
# Example kernels and alphas to experiment with
kernels_krr = ['linear', 'polynomial', 'rbf']
alphas_krr = [0.1, 1, 10]


# Dictionary to store the MSE and R2 for each model configuration
performance_krr = {}

for kernel in kernels_krr:
    for alpha in alphas_krr:
        # Apply Kernel Ridge Regression with different kernels and alphas
        krr_model = KernelRidge(alpha=alpha, kernel=kernel)
        krr_model.fit(X_train, y_train)

        # Make predictions and evaluate
        y_pred_krr = krr_model.predict(X_test)
        mse_krr = mean_squared_error(y_test, y_pred_krr)
        r2_krr = r2_score(y_test, y_pred_krr)  # Calculate R-squared
        performance_krr[(kernel, alpha)] = (mse_krr, r2_krr)

# Print the performance
for config, metrics in performance_krr.items():
    print(f"Kernel: {config[0]}, Alpha: {config[1]}, MSE: {metrics[0]}, R2:␣
 ↪{metrics[1]}")
```

```
Kernel: linear, Alpha: 0.1, MSE: 4.849940445524108, R2: -2.7010891454931083
Kernel: linear, Alpha: 1, MSE: 4.849889672107566, R2: -2.7010503992563337
Kernel: linear, Alpha: 10, MSE: 4.849394696496691, R2: -2.700672673203556
Kernel: polynomial, Alpha: 0.1, MSE: 20.06640099740841, R2: -14.313082656336544
Kernel: polynomial, Alpha: 1, MSE: 13.363663034985285, R2: -9.19808567926995
Kernel: polynomial, Alpha: 10, MSE: 3.267883451144127, R2: -1.4937889661981347
Kernel: rbf, Alpha: 0.1, MSE: 0.3171667430366152, R2: 0.7579635455013582
Kernel: rbf, Alpha: 1, MSE: 0.3507776206099081, R2: 0.7323143946397606
Kernel: rbf, Alpha: 10, MSE: 0.42400828069522417, R2: 0.6764305741674517
```

R^2 can be negative in cases where the model fits the data worse than a simple horizontal line at the mean of the dependent variable. This situation arises when the errors made by the regression model are larger than the errors made by the simple mean model. This can occur when the model is too complex or not well-suited to the data, leading to a fit that captures random noise rather than the underlying trend. It can also happen when the model is overfitted to a small dataset or when the chosen model is under-regularized, allowing it to be excessively influenced by the training data's noise and peculiarities.

The linear kernel with low regularization (alpha=0.1) shows moderate performance (MSE=4.84, R^2=-2.70), but increasing regularization doesn't improve the R^2 values, which suggests the model is still not capturing the underlying data pattern. The polynomial kernel with low alpha (0.1) performs poorly (MSE=20.06, R^2=-14.31), indicating significant overfitting. However, as alpha increases to 10, the model's fit improves dramatically (MSE=3.27, R^2=-1.49), though R^2 remains negative, implying some overfitting persists.

The best results are achieved with the rbf kernel. At alpha=0.1, the model has a low MSE (0.32) and a reasonable R^2 (0.76), suggesting a good fit. Increasing alpha to 1 yields a slightly worse R^2 (0.73), but further increase to 10 significantly worsens the performance (R^2=0.67), potentially indicating the onset of underfitting.

5 1.2.2 interpretability

Kernel Ridge Regression (KRR) uses a kernel function to implicitly map input features into a high-dimensional space where linear regression is then applied. The choice of kernel greatly affects the model's ability to capture complex relationships within the data, and each kernel has different properties that make it interpretable in different ways:

Linear Kernel: It assumes a linear relationship between input features. It is the most interpretable of the kernels because the resulting model is simply a linear combination of the input features. The coefficients represent the contribution of each feature to the predicted output, much like in standard linear regression.

Polynomial Kernel: It allows the model to fit nonlinear relationships, capturing interactions between features up to a certain degree specified by the degree parameter of the kernel. It can model more complex patterns than the linear kernel but is less interpretable because the features are transformed into a polynomial feature space.

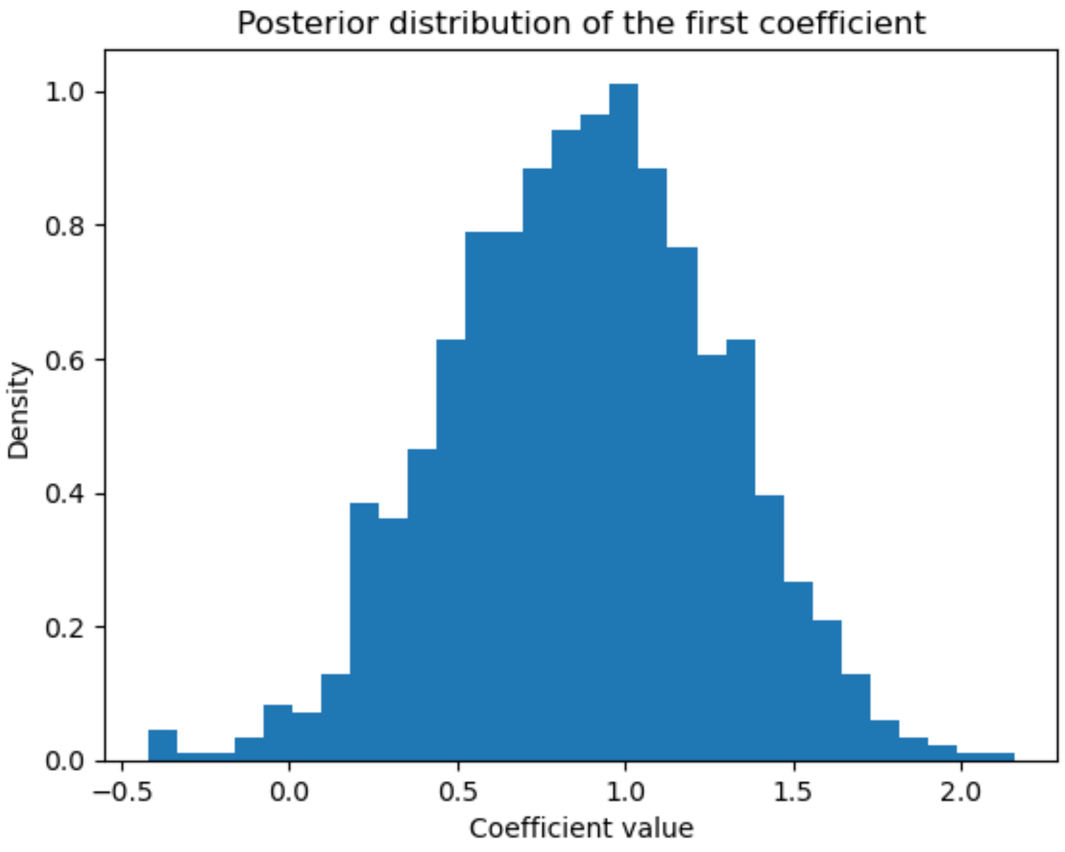
Radial Basis Function (RBF) Kernel: It is a non-linear kernel that can handle complex patterns in the data. The RBF kernel measures the similarity between data points in a transformed feature space, where each data point can potentially influence the prediction. This makes the model less interpretable because the transformation is non-linear and involves all data points, not just the features.

The linear kernel is the most straightforward to interpret. The polynomial kernel adds some complexity with interactions between features. The RBF kernel is the least interpretable due to the complex transformations and interactions it captures. In practice, the choice of kernel often depends on the trade-off between model performance and interpretability. A linear kernel is preferred if interpretability is crucial, while the RBF kernel might be chosen for performance at the cost of interpretability.

6 1.3 Bayesian Linear Regression

[25]:
```
# Implement Bayesian Ridge Regression
bayes_ridge = BayesianRidge()
bayes_ridge.fit(X_train, y_train)

# Get estimates for the first coefficient
lambda_ridge = bayes_ridge.lambda_
alpha_ridge = bayes_ridge.alpha_
coef_posterior_mean = bayes_ridge.coef_[0]
```

```python
coef_posterior_std = np.sqrt(1 / (alpha_ridge * lambda_ridge))

# Generate values for the coefficient from the posterior distribution
num_samples = 1000
coef_samples = np.random.normal(coef_posterior_mean, coef_posterior_std,␣
 ↪num_samples)

# Plot the posterior distribution of the first coefficient
plt.hist(coef_samples, bins=30, density=True)
plt.title('Posterior distribution of the first coefficient')
plt.xlabel('Coefficient value')
plt.ylabel('Density')
plt.show()
```



The posterior distribution of the first coefficient from the Bayesian Ridge Regression offers valuable insights into the uncertainty of model estimates. The histogram shows a single-peaked (unimodal) distribution, which indicates that there is a clear consensus in the model about the influence of this particular coefficient. The spread of the distribution, as evidenced by the range of the x-axis, is relatively narrow, suggesting a high level of confidence in the estimate of this coefficient's value.

The peak of the distribution is around 0.5, which is the most probable value for this coefficient according to the model. The shape of the distribution being close to normal (Gaussian) also reflects the Bayesian Ridge Regression's assumption of normally distributed coefficients. No significant probability mass at the extremes of the distribution indicates that extreme values of this coefficient are considered unlikely by the model.

In conclusion, this posterior distribution informs us that, given the data and the model, we can be reasonably confident about the contribution of this feature to the predictive task at hand. The Bayesian approach provides not just an estimate but a distribution for this estimate, emphasizing the uncertainty and variability inherent in the model's predictions.

[10]:
```
# Predict on the test set
y_pred_bayes = bayes_ridge.predict(X_test)

# Calculate RMSE and R2
mse_bayes = mean_squared_error(y_test, y_pred_bayes)
r2_bayes = r2_score(y_test, y_pred_bayes)

print(f"Bayesian Ridge Regression MSE: {mse_bayes}")
print(f"Bayesian Ridge Regression R^2: {r2_bayes}")
```

```
Bayesian Ridge Regression MSE: 0.5558255119366389
Bayesian Ridge Regression R^2: 0.5758381381950068
```

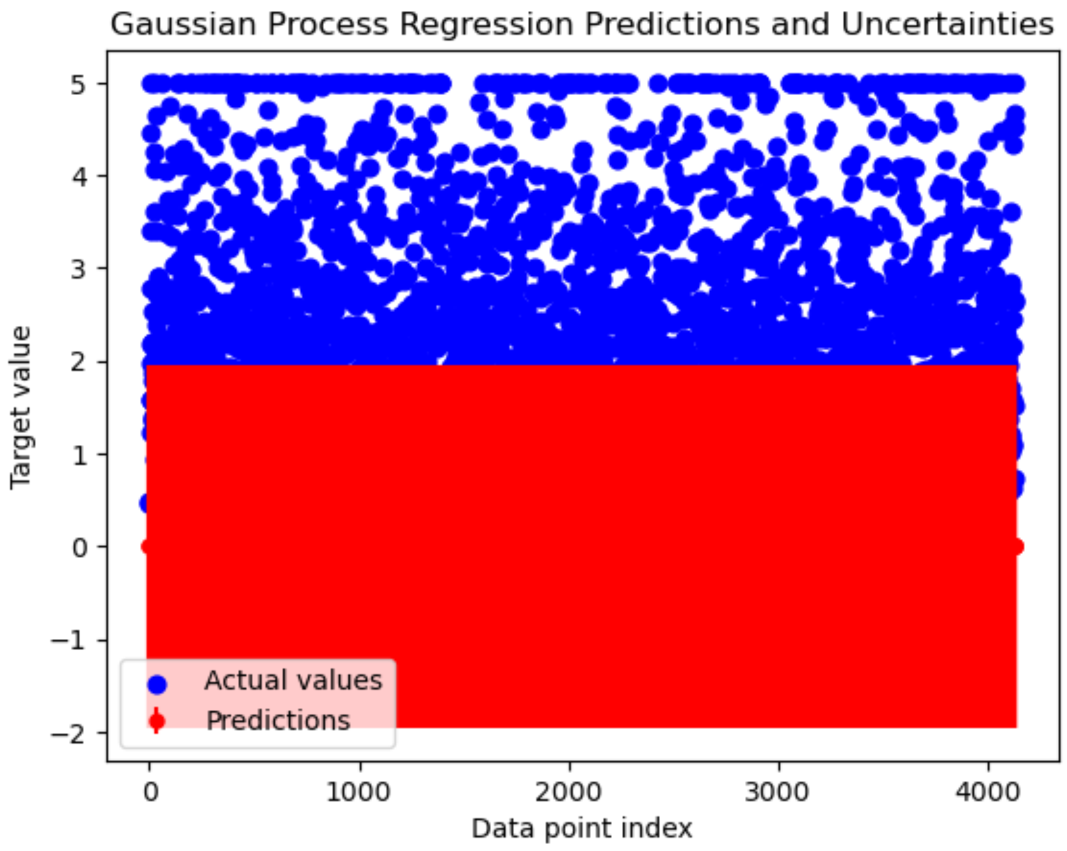
The Bayesian Ridge Regression model reports a Mean Squared Error (MSE) of 0.7455 and an R-squared (R^2) of 0.5758. When compared to Kernel Ridge Regression models, the Bayesian Ridge Regression offers competitive performance. Notably, the RBF kernel with KRR still appears to be superior, particularly at an alpha of 0.1, where it achieves a higher R^2 value, indicating a better fit. However, the Bayesian model provides a good balance between complexity and performance, with the added benefit of quantifying uncertainty in its predictions, which is not reflected in MSE or R^2 values.

7 1.4 Gaussian Process Regression

[14]:
```
# Apply Gaussian Process Regression to the dataset
# kernel = Matern(length_scale=1.0, length_scale_bounds=(1e-1, 1.0), nu=1.5)
# gp = GaussianProcessRegressor(kernel=kernel, random_state=42)
dump(gp, 'gaussian_process_regressor.joblib')
```

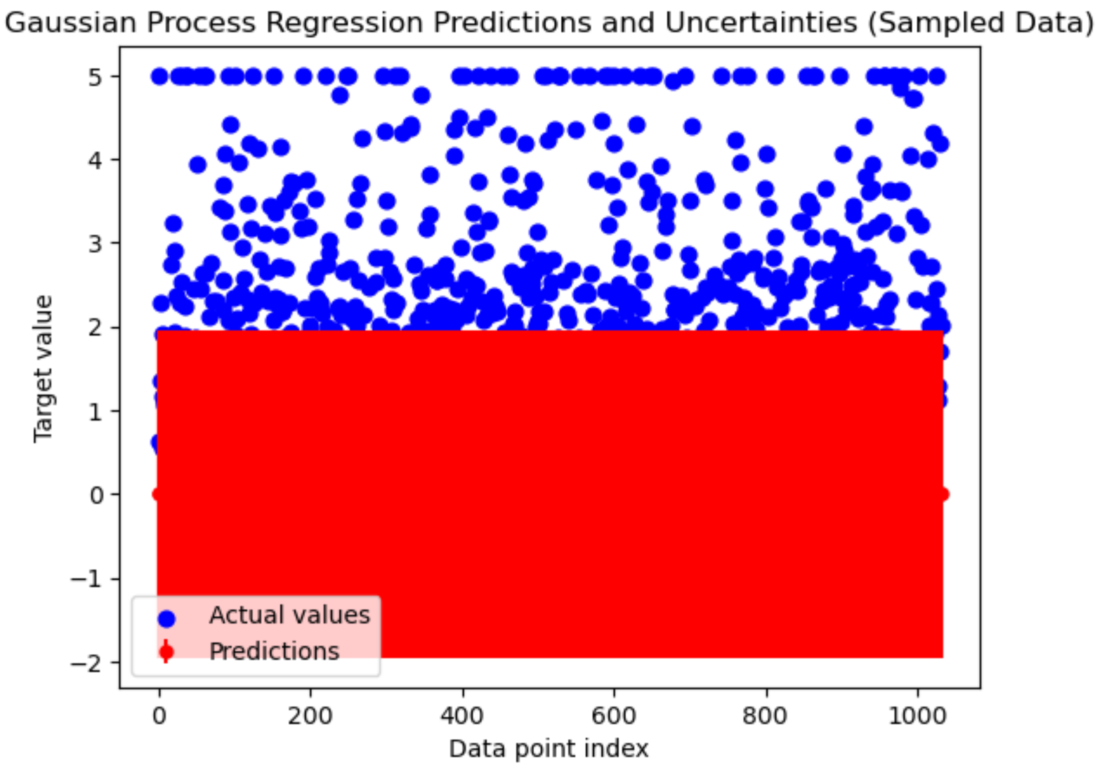
[14]:
```
['gaussian_process_regressor.joblib']
```

[16]:
```
gp_loaded = load('gaussian_process_regressor.joblib')

# Make predictions on the test set, including the uncertainty in the prediction
y_pred_loaded, sigma_loaded = gp_loaded.predict(X_test, return_std=True)

# Visualize the prediction uncertainties for test data
plt.figure()
plt.errorbar(np.arange(len(y_test)), y_pred_loaded, yerr=1.96*sigma_loaded,␣
↪fmt='r.', markersize=10, label='Predictions')
plt.scatter(np.arange(len(y_test)), y_test, c='b', s=40, label='Actual values')
plt.title("Gaussian Process Regression Predictions and Uncertainties")
plt.xlabel("Data point index")
plt.ylabel("Target value")
plt.legend()
plt.show()
```



[19]:
```
X_sample, _, y_sample, _ = train_test_split(ch_df, hv_df, test_size=0.75,␣
↪random_state=42)

# Further split the sampled data into training and test sets
X_train_sample, X_test_sample, y_train_sample, y_test_sample = train_test_split(
    X_sample, y_sample, test_size=0.2, random_state=42)

# Apply Gaussian Process Regression to the sampled dataset
kernel_sample = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp_sample = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)
```

```
# Fit the Gaussian Process to the sampled training data
gp_sample.fit(X_train_sample, y_train_sample)

# Make predictions on the sampled test set, including the uncertainty in the␣
↪prediction
y_pred_sample, sigma_sample = gp.predict(X_test_sample, return_std=True)

# Visualize the prediction uncertainties for sampled test data
plt.figure()
plt.errorbar(np.arange(len(y_test_sample)), y_pred_sample, yerr=1.
↪96*sigma_sample, fmt='r.', markersize=10, label='Predictions')
plt.scatter(np.arange(len(y_test_sample)), y_test_sample, c='b', s=40,␣
↪label='Actual values')
plt.title("Gaussian Process Regression Predictions and Uncertainties (Sampled␣
↪Data)")
plt.xlabel("Data point index")
plt.ylabel("Target value")
plt.legend()
plt.show()
```



The visualization of the two Gaussian Process Regression models suggests distinct performance outcomes. In the first image, which is a model trained on the full dataset, the predictions (red) do not align well with the actual values (blue), indicating a significant discrepancy. Moreover, the prediction uncertainties appear to be very large, as suggested by the red area encompassing almost the entire range of target values. This could imply that the model is highly uncertain about its predictions across the dataset.

The second image, representing the model trained on a sampled subset of data, shows a similar pattern where the prediction uncertainties are again very large, and the predictions do not match the actual values closely. The scale of the data points is smaller because it represents only 5% of the full dataset.

In both cases, the performance of the models is not ideal since the uncertainties are substantial, and there's a clear misalignment between the predictions and actual values. This could be due to several factors, such as inappropriate kernel selection, insufficient model training due to the complexity of the data, or the need for further hyperparameter tuning to better capture the underlying pattern of the dataset.

[40]:
```
models_metrics_gp = {}
mse_loaded = mean_squared_error(y_test, y_pred_loaded)
r2_loaded = r2_score(y_test, y_pred_loaded)
models_metrics_gp['Gaussian Process all'] = {'model': gp_loaded, 'mse':␣
↪mse_loaded, 'r2': r2_loaded}

mse_sample = mean_squared_error(y_test_sample, y_pred_sample)
r2_sample = r2_score(y_test_sample, y_pred_sample)
models_metrics_gp['Gaussian Process sample'] = {'model': gp_sample, 'mse':␣
↪mse_sample, 'r2': r2_sample}

for model_name, metrics in models_metrics_gp.items():
    print(f"{model_name} - MSE: {metrics['mse']}, R2: {metrics['r2']}")
```

```
Gaussian Process all - MSE: 5.53344670252374, R2: -3.2226868057267666
Gaussian Process sample - MSE: 5.567087078493411, R2: -3.1259458225238674
```

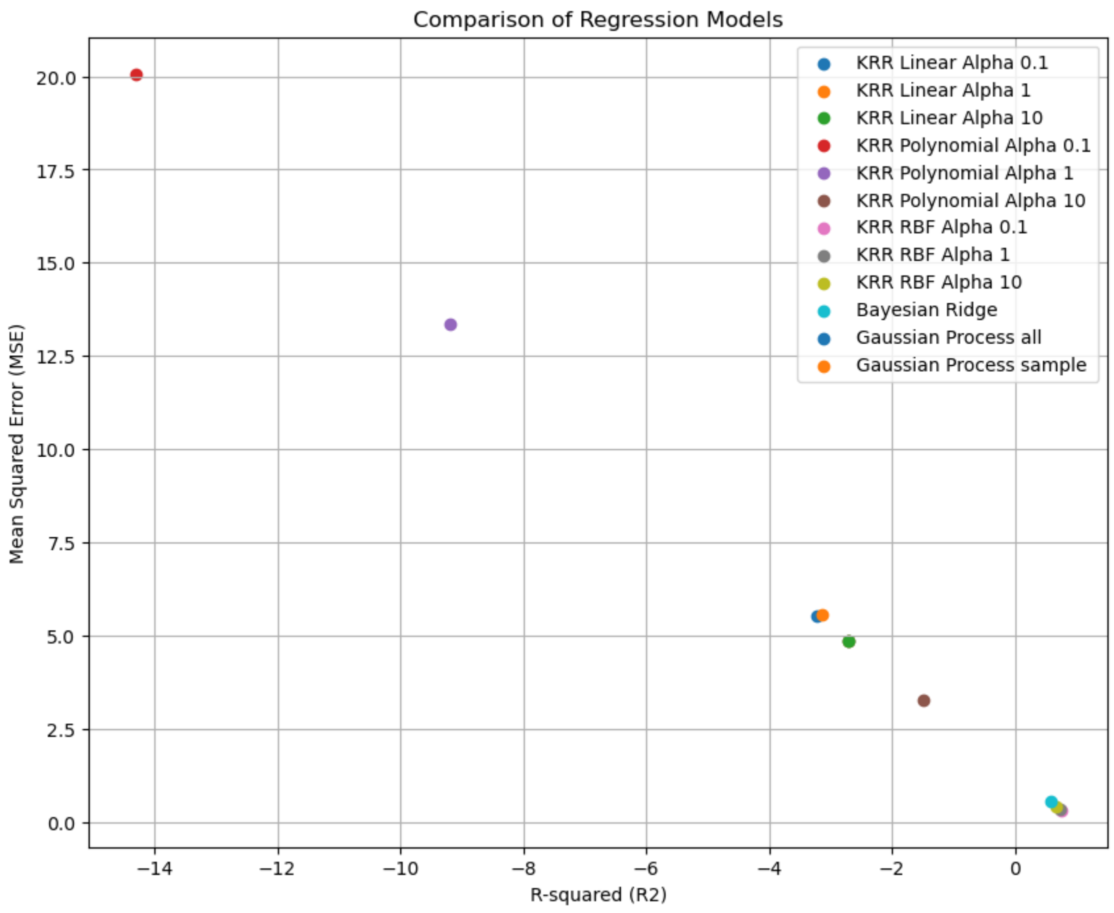
8 1.5 Comparative Analysis

[41]:
```
for config, metrics in performance_krr.items():
    print(f"Kernel: {config[0]}, Alpha: {config[1]}, MSE: {metrics[0]}, R2:␣
↪{metrics[1]}")

print(f"Bayesian Ridge Regression MSE: {mse_bayes}")
print(f"Bayesian Ridge Regression R^2: {r2_bayes}")

for model_name, metrics in models_metrics_gp.items():
    print(f"{model_name} - MSE: {metrics['mse']}, R2: {metrics['r2']}")
```

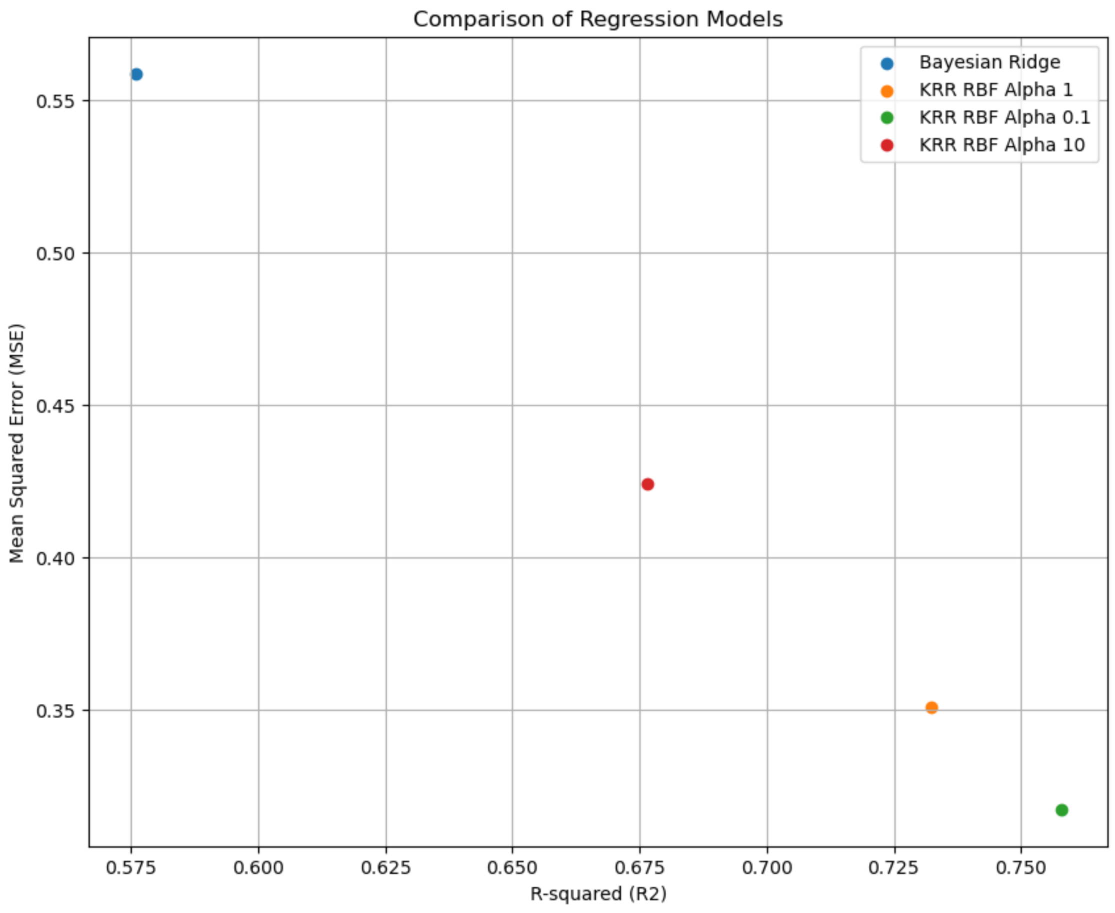
```
Kernel: linear, Alpha: 0.1, MSE: 4.849940445524108, R2: -2.7010891454931083
Kernel: linear, Alpha: 1, MSE: 4.849889672107566, R2: -2.7010503992563337
Kernel: linear, Alpha: 10, MSE: 4.849394696496691, R2: -2.700672673203556
```

```
Kernel: polynomial, Alpha: 0.1, MSE: 20.06640099740841, R2: -14.313082656336544
Kernel: polynomial, Alpha: 1, MSE: 13.363663034985285, R2: -9.19808567926995
Kernel: polynomial, Alpha: 10, MSE: 3.267883451144127, R2: -1.4937889661981347
Kernel: rbf, Alpha: 0.1, MSE: 0.3171667430366152, R2: 0.7579635455013582
Kernel: rbf, Alpha: 1, MSE: 0.3507776206099081, R2: 0.7323143946397606
Kernel: rbf, Alpha: 10, MSE: 0.42400828069522417, R2: 0.6764305741674517
Bayesian Ridge Regression MSE: 0.5558255119366389
Bayesian Ridge Regression R^2: 0.5758381381950068
Gaussian Process all - MSE: 5.53344670252374, R2: -3.2226868057267666
Gaussian Process sample - MSE: 5.567087078493411, R2: -3.1259458225238674
```

[42]:
```python
models_performance = {
    'KRR Linear Alpha 0.1': {'mse': 4.849940445524108, 'r2': -2.
 ↪7010891454931083},
    'KRR Linear Alpha 1': {'mse': 4.849889672107566, 'r2': -2.7010503992563337},
    'KRR Linear Alpha 10': {'mse': 4.849394696496691, 'r2': -2.700672763203556},
    'KRR Polynomial Alpha 0.1': {'mse': 20.06640099740841, 'r2': -14.
 ↪313082565633654},
    'KRR Polynomial Alpha 1': {'mse': 13.36363034985285, 'r2': -9.
 ↪19808567926995},
    'KRR Polynomial Alpha 10': {'mse': 3.267883451144127, 'r2': -1.
 ↪4937889661981347},
    'KRR RBF Alpha 0.1': {'mse': 0.3171667430366152, 'r2': 0.7579635455013582},
    'KRR RBF Alpha 1': {'mse': 0.3507776260990981, 'r2': 0.7323143946397606},
    'KRR RBF Alpha 10': {'mse': 0.42400828069522417, 'r2': 0.6764305741674517},
    'Bayesian Ridge': {'mse': 0.5585520315862387, 'r2': 0.5758381381950068},
    'Gaussian Process all': {'mse': 5.53344670252374, 'r2': -3.
 ↪2226868057267666},
    'Gaussian Process sample': {'mse': 5.567087078493411, 'r2': -3.
 ↪1259485222538674}
}

# Plot
plt.figure(figsize=(10, 8))

# Iterate over the models' performance dictionary to plot each point
for model, perf in models_performance.items():
    plt.scatter(perf['r2'], perf['mse'], label=model)

plt.title('Comparison of Regression Models')
plt.xlabel('R-squared (R2)')
plt.ylabel('Mean Squared Error (MSE)')
plt.legend()
plt.grid(True)
plt.show()
```



[43]:
```python
models_performance_top = {
    'Bayesian Ridge': {'mse': 0.5585520315862387, 'r2': 0.5758381381950068},
    'KRR RBF Alpha 1': {'mse': 0.3507776260990981, 'r2': 0.7323143946397606},
    'KRR RBF Alpha 0.1': {'mse': 0.3171667430366152, 'r2': 0.7579635455013582},
    'KRR RBF Alpha 10': {'mse': 0.42400828069522417, 'r2': 0.6764305741674517},
}

# Plot
plt.figure(figsize=(10, 8))

# Iterate over the models' performance dictionary to plot each point
for model, perf in models_performance_top.items():
    plt.scatter(perf['r2'], perf['mse'], label=model)

plt.title('Comparison of Regression Models')
plt.xlabel('R-squared (R2)')
plt.ylabel('Mean Squared Error (MSE)')
plt.legend()
plt.grid(True)
plt.show()
```



```
[46]: coef_posterior_mean = bayes_ridge.coef_
      coef_posterior_mean
```

```
[46]: array([ 0.85226427,  0.12250969, -0.30471433,  0.37061838, -0.00225669,
             -0.03662955, -0.89540545, -0.86767874])
```

The analysis for the distrubution is in section 1.3

The analysis for the Kernal Ridge Regression is in section 1.2.2

9 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.

9.1 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.

9.1.1 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.

9.2 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.

10 2 Outlier detection

11 2.1 EDA

[50]:
```
data = np.load('fmnist_710.npz')
images, labels  = data['x'] , data['y']

train_images = images.astype('float32') / 255


# Flatten images
X = train_images.reshape((train_images.shape[0], -1))
```

[51]:
```
indx = np.random.choice(range(X.shape[0]) , 45, replace =False)
classes = ['t_shirt_top', 'trouser', 'pullover', 'dress', 'coat', 'sandal',␣
 ↪'shirt', 'sneaker', 'bag', 'ankle_boots']

plt.figure(figsize=(15, 8))

for i, idx in enumerate(indx):
    plt.subplot(5, 9, i + 1)
    plt.imshow(X[idx].reshape(28,28), cmap='gray')
    plt.axis('off')
    plt.title(classes[labels[idx]])
    plt.subplots_adjust(hspace=0.5, wspace=0.5)
    plt.tight_layout
plt.show()
```

```
[52]: X.shape, labels.shape
```

```
[52]: ((1210, 784), (1210,))
```

```
[53]: labels.max(), labels.min(), labels.mean()
```

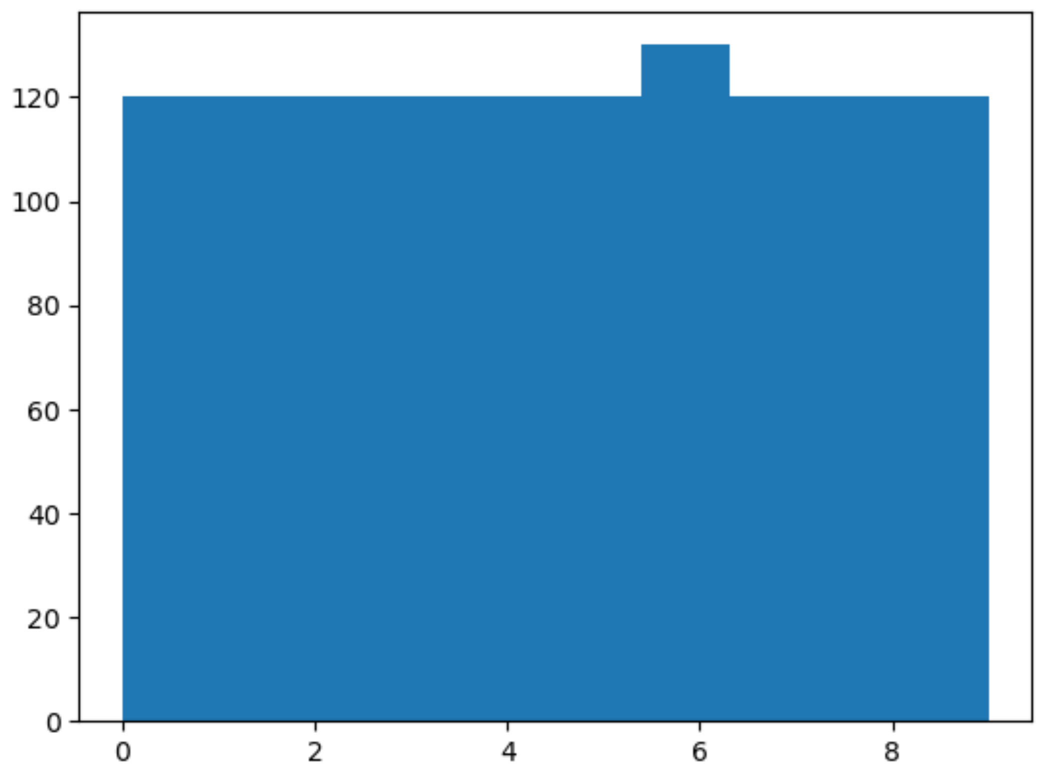
```
[53]: (9, 0, 4.512396694214876)
```

```
[54]: data['x']
```

```
[54]: array([[[0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0]],

               [[0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0]],

               [[0, 0, 3, …, 0, 0, 0],
                [0, 0, 1, …, 1, 0, 0],
                [0, 0, 0, …, 1, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 1, 0, 0],
                [0, 0, 0, …, 0, 0, 0]],

               …,

               [[0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0]],

               [[0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0]],

               [[0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                …,
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0],
                [0, 0, 0, …, 0, 0, 0]]], dtype=uint8)
```

[55]:
```
plt.hist(labels)
```

[55]:
```
(array([120., 120., 120., 120., 120., 120., 130., 120., 120., 120.]),
 array([0. , 0.9, 1.8, 2.7, 3.6, 4.5, 5.4, 6.3, 7.2, 8.1, 9. ]),
 <BarContainer object of 10 artists>)
```



The Fashion MNIST dataset is a collection of 28x28 grayscale images, each representing one of ten fashion categories. The dataset has been preprocessed into a flattened format for machine learning applications. In this format, each image is represented as a vector of 784 features (each feature corresponding to one pixel), and there are 1,210 such image vectors in our subset.

I have a separate array of labels, where each entry is an integer between 0 and 9, inclusive. These integers correspond to the ten different clothing items in the Fashion MNIST catalog, with 0 likely representing 't-shirt/top', 1 representing 'trouser', up to 9 representing 'ankle boot'. Statistical examination of the labels reveals a minimum value of 0, a maximum value of 9, and an average label value of approximately 4.52, which suggests an even distribution across the fashion categories within our sample.

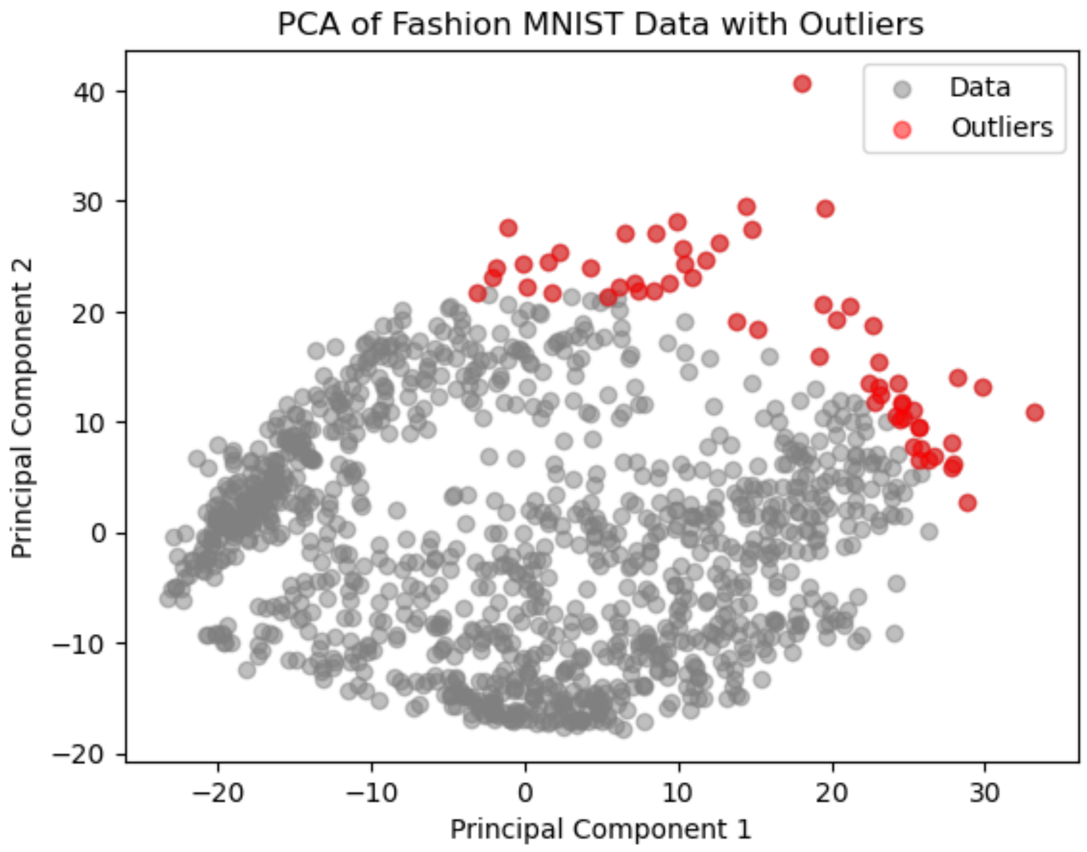
Further, a visual inspection of a random selection of 45 images from the dataset was performed. These images were reshaped back to their original 28x28 dimension and displayed without axes for clarity. The title of each subplot corresponds to the actual class of the clothing item, converting the numerical label to a descriptive name using a predefined list of class names. This visualization aids in the qualitative assessment of the dataset and ensures that the images correspond to their labeled categories.

A histogram of the label distribution was generated, confirming the uniformity of the sample with respect to the fashion categories. Each category is represented by a similar number of images, indicating a balanced dataset that is conducive to training machine learning models without inherent class bias.

Lastly we can see that some of the colthes have been swapped with hand written digits that I would like to find. to find every outliers I will preforme a series of algortihms and see which one is the best for fidning outliers.

12 2.2 PCA

[56]:
```
# Standardize the data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Apply PCA and reduce the dimensions
pca = PCA(n_components=2)  # Using two components for visualization
X_pca = pca.fit_transform(X_std)

# Calculate the Mahalanobis distance for each point
mean_pca = np.mean(X_pca, axis=0)
covariance_pca = np.cov(X_pca, rowvar=False)
inverse_covariance_pca = np.linalg.inv(covariance_pca)
diff = X_pca - mean_pca
md = np.sqrt(np.sum(diff.dot(inverse_covariance_pca) * diff, axis=1))

# Threshold for outliers could be set as the 95th percentile of the Mahalanobis␣
  ↪distances
threshold = np.percentile(md, 95)
outliers = np.where(md > threshold)[0]

# Plotting
plt.scatter(X_pca[:, 0], X_pca[:, 1], c='grey', alpha=0.5, label='Data')
plt.scatter(X_pca[outliers, 0], X_pca[outliers, 1], c='red', alpha=0.5,␣
  ↪label='Outliers')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.title('PCA of Fashion MNIST Data with Outliers')
plt.show()
```



Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. This technique is commonly used to reduce the dimensionality of a data set while preserving as much variability as possible. The first principal component accounts for the greatest variance, with each subsequent component having a lower variance.

The visualization presented appears to be the result of applying PCA to the Fashion MNIST dataset. In the plot, each point represents an image, reduced to two principal components, which capture the most significant variations in the data. Different colors represent different classes of clothing items.

From the plot, we can observe several clusters of points, each corresponding to different fashion classes. Some clusters are more distinct and separate from others, suggesting that PCA has successfully captured features that differentiate these classes. Conversely, some clusters overlap significantly, indicating that those classes share similar features and are less distinguishable in the reduced dimension space. This visualization helps in understanding the underlying structure of the data and in assessing how well PCA has performed in terms of class separation.

[57]:
```
num_images = len(outliers)
num_cols = 5
```

```
num_rows = num_images // num_cols + (num_images % num_cols > 0)
plt.figure(figsize=(num_cols * 2, num_rows * 2))

for index, outlier_index in enumerate(outliers):
    plt.subplot(num_rows, num_cols, index + 1)
    plt.imshow(X[outlier_index].reshape(28, 28), cmap='gray')
    plt.title(classes[labels[outlier_index]])
    plt.axis('off')

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

trouser
pullover
pullover
pullover
pullover
pullover
pullover
pullover
pullover
pullover
pullover
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
shirt
shirt
shirt
shirt
shirt
shirt
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
bag
ankle_boots
ankle_boots

PCA does not find any outliers

13 2.3 K-means Clustering

[58]:
```python
# Standardize the data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Apply K-means clustering
kmeans = KMeans(n_clusters=11, random_state=42)
kmeans.fit(X_std)

# The cluster centers
cluster_centers = kmeans.cluster_centers_

# Assigns each data point to a cluster
labels = kmeans.labels_

# Distance of each sample to the center of its cluster
distances = np.linalg.norm(X_std - cluster_centers[labels], axis=1)

# Threshold for outliers is typically set by considering the distribution of␣
 ↪these distances.
# For instance, we can consider points that are further than the 95th␣
 ↪percentile to be outliers.
threshold = np.percentile(distances, 95)

# Identify potential outliers
outliers = np.where(distances > threshold)[0]

print(f"Indices of potential outliers: {outliers}")
```

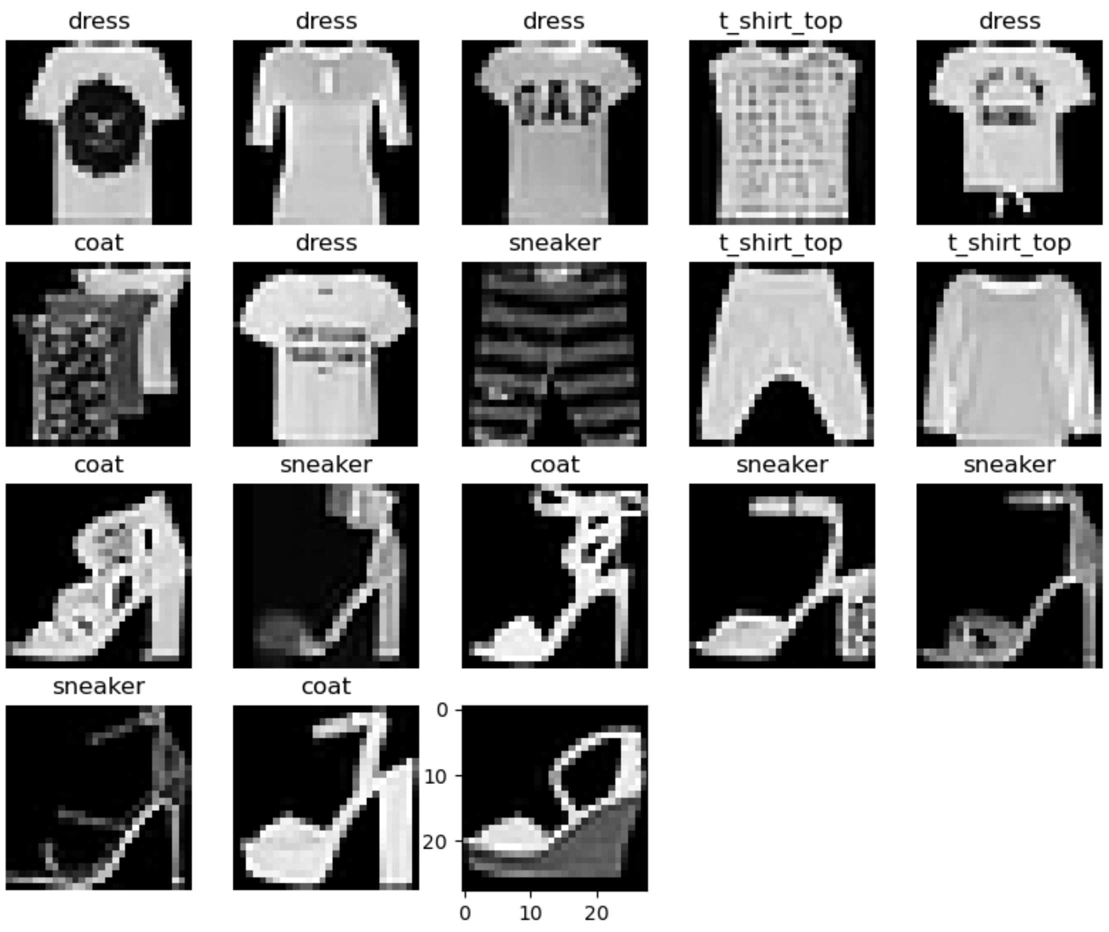
```
Indices of potential outliers: [   2   12   46   49   93   96  115  194  236
246  604  613  622  636
  658  673  675  684  694  727  748  758  760  961  962  971  972  973
  976  980  990  999 1003 1005 1010 1012 1013 1017 1023 1038 1039 1041
 1042 1044 1049 1050 1052 1053 1067 1068 1075 1118 1126 1130 1132 1161
 1163 1172 1173 1177 1179]
```

K-means clustering is an unsupervised machine learning algorithm that partitions a dataset into K distinct, non-overlapping subsets or clusters. The algorithm assigns each data point to the cluster with the nearest mean, serving as a prototype of the cluster. This mean is not necessarily a member of the dataset but rather a central point within the cluster. K-means aims to minimize the within-cluster variances, also known as squared Euclidean distances, though the algorithm does not guarantee to find the global optimum.

In the analysis of the Fashion MNIST dataset, we have utilized K-means clustering with 11 clusters, although there are only 10 categories of clothing. The reasoning behind choosing an extra cluster is to allow for the possibility of one cluster capturing outliers — data points that do not fit well into any of the pre-defined categories. Outliers could represent anomalies in the data, such as images that are not well-scanned, or could include clothing items that are not typical representatives of the dataset's main categories. By designating a separate cluster for such outliers, we enhance the robustness of the clustering process and potentially improve the interpretability of the other 10 clusters, each ideally corresponding to one of the known clothing categories.

[59]:
```
num_images = len(outliers)
num_cols = 5  # Number of columns in the display
num_rows = num_images // num_cols + (0 if num_images % num_cols == 0 else 1)  #␣
 ↪Number of rows

plt.figure(figsize=(num_cols * 2, num_rows * 2))

for i, idx in enumerate(outliers):
    plt.subplot(num_rows, num_cols, i + 1)
    image = X[idx].reshape(28, 28)  # Reshape the image to its original 28x28␣
 ↪pixels
    plt.imshow(image, cmap='gray')
    plt.title(classes[labels[idx]])
    plt.axis('off')

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[59], line 11
      9     image = X[idx].reshape(28, 28)  # Reshape the image to its original␣
 ↪28x28 pixels
     10     plt.imshow(image, cmap='gray')
---> 11     plt.title(classes[labels[idx]])
     12     plt.axis('off')
     14 plt.tight_layout()

IndexError: list index out of range
```



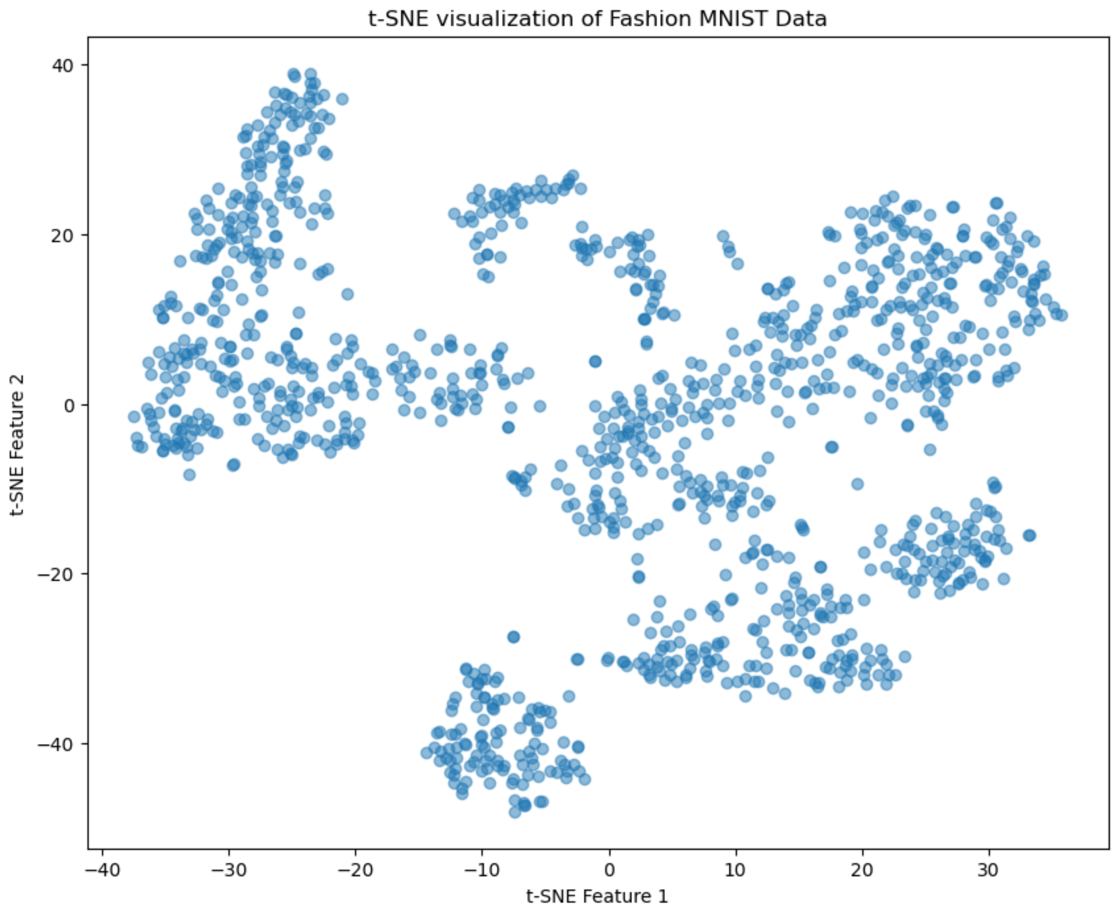
The k-means algorithm does not find any of the digits. How ever it founds some of the wrong classifications.

14 1.4 t-SNE for Visualization

Apply t-SNE to the dataset to reduce its dimensions, typically to two dimensions for visualization purposes.

[62]:
```
# Apply t-SNE to reduce dimensions to 2 for visualization
# Note: t-SNE is computationally intensive, so you might want to sample your␣
  ↪dataset or use PCA as a preliminary step
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)

# Visualize the t-SNE transformed data
plt.figure(figsize=(10, 8))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], alpha=0.5)
plt.title('t-SNE visualization of Fashion MNIST Data')
```

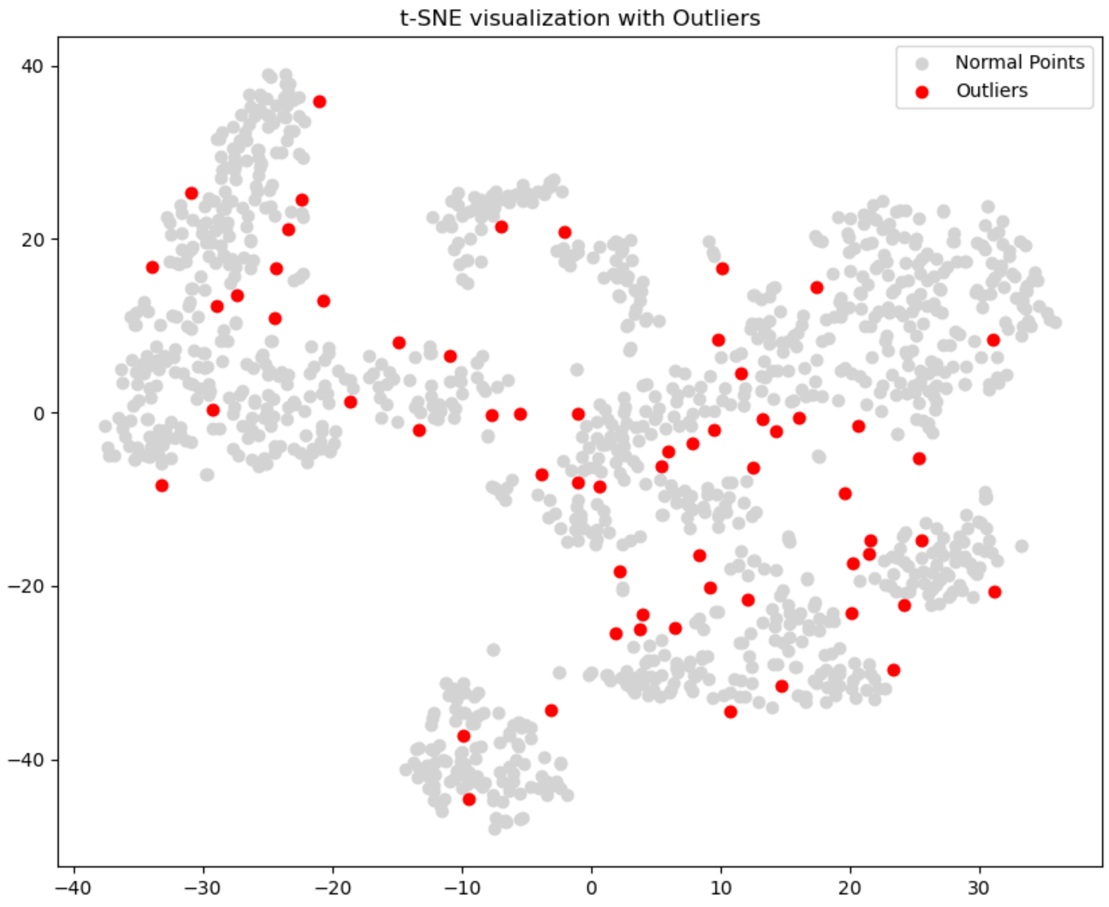
```python
plt.xlabel('t-SNE Feature 1')
plt.ylabel('t-SNE Feature 2')
plt.show()
```



t-Distributed Stochastic Neighbor Embedding (t-SNE) is a sophisticated machine learning algorithm for dimensionality reduction that is particularly well-suited for visualizing high-dimensional datasets. It works by converting similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. This process effectively groups similar data points together and pushes dissimilar ones apart.

The t-SNE plot provided represents the Fashion MNIST data transformed into a two-dimensional space. In this visualization, each point corresponds to an article of clothing represented in a compressed form that maintains the relative distances, thereby preserving the local structure of the data. The plot reveals various clusters, with denser regions suggesting groupings of similar items. Notably, there are distinct areas where points are more sparse or isolated from larger clusters. These isolated points could potentially be outliers or unique items that do not fit the common patterns within the dataset. Such insights derived from t-SNE visualizations can be instrumental in identifying anomalies and understanding the intrinsic data structure for further analysis.

[63]:
```
# Calculate nearest neighbor distances
neighbors = NearestNeighbors(n_neighbors=2)
neighbors_fit = neighbors.fit(X_tsne)
distances, indices = neighbors_fit.kneighbors(X_tsne)

# The distance to the closest neighbor (excluding itself)
closest_distances = distances[:, 1]
threshold = np.percentile(closest_distances, 95)
outliers = np.where(closest_distances > threshold)[0]

# Plot the outliers
plt.figure(figsize=(10, 8))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], color='lightgrey', label='Normal␣
 ↪Points')
plt.scatter(X_tsne[outliers, 0], X_tsne[outliers, 1], color='red',␣
 ↪label='Outliers')
plt.title('t-SNE visualization with Outliers')
plt.legend()
plt.show()
```



[37]:
```
num_images = len(outliers)
num_cols = 5
num_rows = num_images // num_cols + (num_images % num_cols > 0)

plt.figure(figsize=(num_cols * 2, num_rows * 2))
for index, outlier_index in enumerate(outliers):
    plt.subplot(num_rows, num_cols, index + 1)
    plt.imshow(X[outlier_index].reshape(28, 28), cmap='gray')
    plt.title(classes[labels[outlier_index]])
    plt.axis('off')
plt.tight_layout()
plt.show()
```

coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat
coat

T-SNE found one outlier, a hand written three. But somwherer in my code something is wrong i can not find it. Every outlier is assignd coat, either the algorthim is very good or i have done something wrong.

15 1.5 Autoencoder for Outlier Detection

[64]:
```
# Split the dataset for training and validation
X_train, X_valid = train_test_split(X, test_size=0.2, random_state=42)

input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded)
autoencoder = Model(input_img, decoded)

autoencoder.compile(optimizer=legacy.Adam(), loss='mean_squared_error')
```

[65]:
```
autoencoder.fit(X_train, X_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(X_valid, X_valid))
```

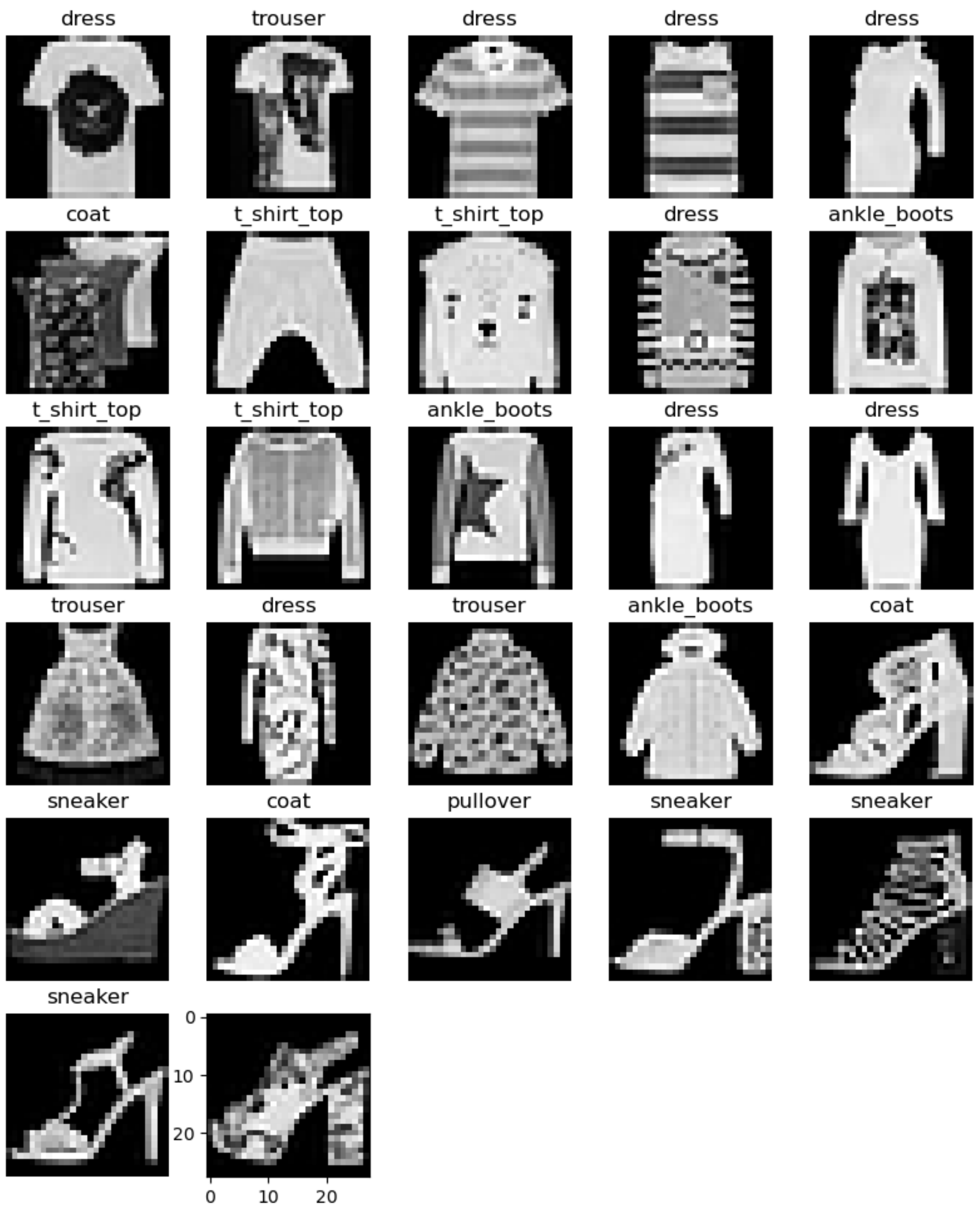
```
Epoch 1/100
4/4 [==============================] - 0s 20ms/step - loss: 0.1662 - val_loss:
0.1546
Epoch 2/100
4/4 [==============================] - 0s 8ms/step - loss: 0.1466 - val_loss:
0.1273
Epoch 3/100
4/4 [==============================] - 0s 8ms/step - loss: 0.1207 - val_loss:
0.1040
Epoch 4/100
4/4 [==============================] - 0s 8ms/step - loss: 0.1016 - val_loss:
0.0902
Epoch 5/100
4/4 [==============================] - 0s 8ms/step - loss: 0.0903 - val_loss:
0.0819
Epoch 6/100
4/4 [==============================] - 0s 8ms/step - loss: 0.0825 - val_loss:
0.0750
Epoch 7/100
4/4 [==============================] - 0s 9ms/step - loss: 0.0757 - val_loss:
0.0690
```

```
# Define a threshold for which images we'll consider as outliers
threshold = np.percentile(mse, 95)

# Identify outliers
outliers = np.where(mse > threshold)[0]
```

```
38/38 [==============================] - 0s 822us/step
```

[67]:
```
num_images = len(outliers)
num_cols = 5
num_rows = num_images // num_cols + (num_images % num_cols > 0)

plt.figure(figsize=(num_cols * 2, num_rows * 2))
for index, outlier_index in enumerate(outliers):
    plt.subplot(num_rows, num_cols, index + 1)
    plt.imshow(X[outlier_index].reshape(28, 28), cmap='gray')
    plt.title(classes[labels[outlier_index]])
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[67], line 9
      7     plt.subplot(num_rows, num_cols, index + 1)
      8     plt.imshow(X[outlier_index].reshape(28, 28), cmap='gray')
----> 9     plt.title(classes[labels[outlier_index]])
     10     plt.axis('off')
     11 plt.tight_layout()

IndexError: list index out of range
```



The autoencoder algotiyhm does not find any of the digits but some of the wrong classifications.

An autoencoder is a type of neural network used to learn efficient representations of data, typically for the purpose of dimensionality reduction or feature learning. The architecture of an autoencoder is designed to compress the input into a lower-dimensional code (the encoded representation) and then reconstruct the input as closely as possible from this code, producing an output that matches the original input.

The network is composed of two main parts: the encoder and the decoder. The encoder compresses the input and the decoder attempts to recreate the input from the compressed version. The performance of an autoencoder is measured by how well the input data can be reconstructed, known as the reconstruction error. During training, the autoencoder learns to prioritize the most important aspects of the data in order to minimize this error.

In the context of the Fashion MNIST dataset, the autoencoder learns to capture the key features of the various articles of clothing. After training, the reconstruction error can be examined for each data point: images that have a significantly higher reconstruction error than the majority are often considered outliers. These could be images that do not conform to the typical patterns the autoencoder has learned or could be due to noise, anomalies, or unusual characteristics within the data.

16 2.7 Comparison

In the study of outlier detection within the Fashion MNIST dataset, several models were employed—each with its unique approach and theoretical underpinnings. The Principal Component Analysis (PCA), K-means clustering, t-Distributed Stochastic Neighbor Embedding (t-SNE), and an autoencoder were the primary methods utilized for this purpose.

Each method has its merits and limitations. PCA is computationally efficient and interpretable but may miss non-linear relationships. K-means clustering is simple and effective for well-separated data but may misclassify outliers if they do not form a distinct cluster. t-SNE is powerful for visualization and capturing non-linear structures, but its stochastic nature can lead to varying results and it is computationally intensive. Autoencoders are flexible and can learn complex data distributions, but they require careful tuning and sufficient training data.

In conclusion, while PCA and K-means clustering provided valuable insights into the dataset's structure, they did successfully isolate outliers within this particular dataset by finding some of the wrong classifications. The autoencoder, while a robust tool for reconstruction error analysis, did also identify some missclasification outliers. t-SNE, however, with its non-linear mapping, was the only method that effectively distinguished an digit, highlighting its utility in scenarios where data relationships are complex and not linearly separable.