CS429 - Coursework 1

- Leon Chipchase
- U2039323

For the following assignment, the questions will first have the code, then the explanation or the answer to the question. Additionally, imports will typically be done where they are needed or at the beginning fof the question they are used in.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
from sklearn.metrics import average_precision_score
```

Setup

In [ ]:
```
Xtrain = pd.read_csv("./Xtrain.csv", delimiter=' ', header=None)
Ytrain = pd.read_csv("./Ytrain.csv", delimiter=' ', header=None)
XTest = pd.read_csv("./Xtest.csv", delimiter=' ', header=None)

np.random.seed(42)
```

Question 1 - Data Exploration

Question 1 - i:

In [ ]:
```
# a  - Calculate the total number of examples in the train and test set:
print("Number of train examples: ", len(Xtrain))
print("Number of test exmaples: ", len(XTest))
```

```
Number of train examples:  3000
Number of test exmaples:  3000
```

In [ ]:
```
# ib)
print("Number of positive examples: ", len(Ytrain[Ytrain.iloc[:, 0] == 1]))
print("number of negative examples: ", len(Ytrain[Ytrain.iloc[:, 0] == -1]))
```

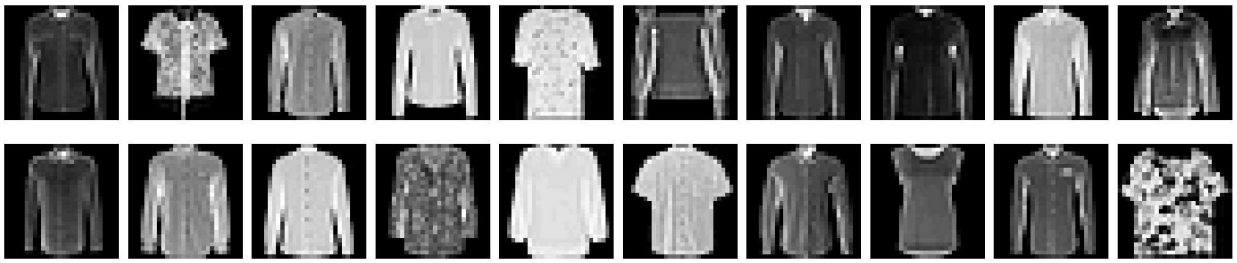
```
Number of positive examples:  311
number of negative examples:  2689
```

There are significantly more examples of the negative class, which may result in the model capturing the relationship between the positive class and the data less effectively.# Mention moare about this

Question 1 - ii:

In [ ]:
```
# Get the negative and positive indices
negative_samples = np.where(np.array(Ytrain) == -1)[0]
```

```
positive_samples = np.where(np.array(Ytrain) == 1)[0]

# Get 10 random indices from each class
random_indices_negative = np.random.choice(negative_samples, size=50, replace=False)
random_indices_positive = np.random.choice(positive_samples, size=50, replace=False)

# Get the 20 random images
selected_elements = np.array(Xtrain)[np.append(random_indices_positive, random_indices

# Plot the 20 random images
fig, axes = plt.subplots(nrows=2, ncols=10, figsize=(20, 5))

for i, ax in enumerate(axes.flatten()):
    # Reshape and plot each image
    ax.matshow(selected_elements[i].reshape(28, 28), cmap='gray')
    ax.axis('off')  # Turn off axis

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



Some patterns that we notice that the positive class (top) appear to be predominantly shirts (both long and short sleeve), whereas the negative class (bottom) contains various items of clothings, some hoodies, jeans, shoes and handbags. It is difficult to determine if one class corresponds to male or female clothing since there appear to be examples of each in both the positive and negattive class.

In [ ]:

```
XTest = np.array(XTest)
print(XTest.shape)
# Get 10 random indices from each class
random_indices = np.random.choice(len(XTest), size=10, replace=False)

# Get the 20 random images
selected_elements = np.array(XTest)[random_indices]

# Plot the 20 random images
fig, axes = plt.subplots(nrows=1, ncols=10, figsize=(20, 5))

for i, ax in enumerate(axes.flatten()):
    # Reshape and plot each image
    ax.matshow(selected_elements[i].reshape(28, 28), cmap='gray')
    ax.axis('off')  # Turn off axis

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

```
(3000, 784)
```

Since each of the examples in the test set seems to be rotated items with additional noise in the pixels compared to the train set, this may cause difficulty in classifying the data since it has been trained on a dataset that has visibly different data than the test set.

Question 1aiii:

- **AUC-PR** focuses on the precision-recall trade-off and is particularly useful when dealing with imbalanced datasets. It takes into account both the precision and the recall, which is critical when the Negative class strongly outbalances the positive class.
- **AUC_ROC** This evaluates TPR vs FPR and may not give an effective indication of the models performance due to the imbalance of the dataset.
- **Matthews Correlation Coefficient** This indicates a high score if there are effective results in all four of the confusion matrix categories, and is therefore relatively imformative even with high class imbalance.
- **F1** The F1 score is the hamonic mean of precision and recall, providing a score that balances both of those concerns.
- **MSE** This is typically used for regression problems rather than classification. This could be very unimformative for a binary classifier.

Overall, **AUC-PR** and **F1** would be the most suitable metrics for this classification task due to the high impbalance of the dataset.

Question iv:

A random classifier would have an expected accuracy of 0.5. This is since the classifier has an equal probability of correctly predicting each class (of which there are two).

In [ ]:
```
# Initialize a list to store the accuracies
accuracies = []

# Run the test 100 times
for _ in range(100):
    # Generate random predictions for Xtrain
    random_predictions = [random.choice([-1, 1]) for _ in range(len(Xtrain))]

    # Calculate the accuracy of the random classifier
    correct_predictions = sum([1 for i in range(len(Xtrain)) if random_predictions[i]
    accuracy = correct_predictions / len(Xtrain)

    # Append the accuracy to the list
    accuracies.append(accuracy)

# Calculate the average accuracy
average_accuracy = sum(accuracies) / len(accuracies)

print("Average accuracy of the random classifier:", average_accuracy)
```

```
Average accuracy of the random classifier: 0.49958000000000014
```

Question 1.V:

- The AUC-ROC for a random classifier would be 0.5, this is because this classifier is expected to be correct 50% of the time. This means the TPR and FPR will increase at the same rate, resulting in a diagonal line from 0,0 to 1,1 which results in an area under the curve of 0.5
- The AUC-PR has an expected value of around 0.1 (as shown by the simulation in the cell below). Since the class split is roughly 10% negative and 90% positive, this seems to make sense. Since that precision looks at the prediction for the positive class only, therefore, this aligns with our intuition.

In [ ]:
```
# Setting parameters
num_simulations = 100
num_positives = 311
num_negatives = 2689
total_samples = num_positives + num_negatives

# List to store AUC-PR values from each simulation
auc_pr_values = []

for _ in range(num_simulations):
    # Generating the true labels (0s for negatives, 1s for positives)
    y_true = np.array([1] * num_positives + [0] * num_negatives)

    # Generating random scores between 0 and 1 for each sample
    y_scores = np.random.rand(total_samples)

    # Calculating the AUC-PR and storing it
    auc_pr = average_precision_score(y_true, y_scores)
    auc_pr_values.append(auc_pr)

# Calculating the average AUC-PR
average_auc_pr = np.mean(auc_pr_values)

print(f"Average AUC-PR for the random classifier over {num_simulations} simulations: {
```

```
Average AUC-PR for the random classifier over 100 simulations: 0.1051225305472888
```

Question 2

In [ ]:
```
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, balanced_accuracy_score, roc_auc_score, av
```

In [ ]:
```
Xtrain = pd.read_csv("./Xtrain.csv", delimiter=' ', header=None)
Ytrain = pd.read_csv("./Ytrain.csv", delimiter=' ', header=None)
XTest = pd.read_csv("./Xtest.csv", delimiter=' ', header=None)
Xtrain = np.array(Xtrain)
XTest = np.array(XTest)
Ytrain = np.array(Ytrain).ravel()

# Define the classifier
```

```
knn = KNeighborsClassifier(n_neighbors=5)

# Define the number of folds for cross-validation
n_folds = 5
kf = StratifiedKFold(n_splits=n_folds)
```

In [ ]:
```
# Metrics storage
metrics = {
    'Accuracy': [],
    'Balanced Accuracy': [],
    'AUC-ROC': [],
    'AUC-PR': [],
    'F1': [],
    'Matthews Correlation Coefficient': []
}
```

i) Defining and calculating the desired metrics for each fold

In [ ]:
```
for train_index, test_index in kf.split(Xtrain, Ytrain):
    X_train, X_test = Xtrain[train_index], Xtrain[test_index]
    Y_train, Y_test = Ytrain[train_index], Ytrain[test_index].reshape(-1, 1)

    # Fit the model
    knn.fit(X_train, Y_train)

    # Predictions
    Y_pred = knn.predict(X_test)
    Y_proba = knn.predict_proba(X_test)

    # Metrics calculation
    metrics['Accuracy'].append(accuracy_score(Y_test, Y_pred))
    metrics['Balanced Accuracy'].append(balanced_accuracy_score(Y_test, Y_pred))
    metrics['AUC-ROC'].append(roc_auc_score(Y_test, Y_proba[:, 1], multi_class='ovr'))
    metrics['AUC-PR'].append(average_precision_score(Y_test, Y_proba[:, 1]))
    metrics['F1'].append(f1_score(Y_test, Y_pred, average='weighted'))
    metrics['Matthews Correlation Coefficient'].append(matthews_corrcoef(Y_test, Y_pre

    # Print the results
    print("============================================================================
    print('Accuracy:', metrics['Accuracy'][-1])
    print('Balanced Accuracy:', metrics['Balanced Accuracy'][-1])
    print('AUC-ROC:', metrics['AUC-ROC'][-1])
    print('AUC-PR:', metrics['AUC-PR'][-1])
    print('F1:', metrics['F1'][-1])
    print('Matthews Correlation Coefficient:', metrics['Matthews Correlation Coefficie
    print("============================================================================


# Calculate mean and standard deviation
metrics_summary = {metric: {'Mean': np.mean(values), 'Std': np.std(values)} for metric

# Display the summary in a table
metrics_df = pd.DataFrame(metrics_summary)
print(metrics_df.to_markdown())
```

```
==========================================================================================
==
Accuracy: 0.9083333333333333
Balanced Accuracy: 0.7134248710876604
AUC-ROC: 0.850221849142583
AUC-PR: 0.43952624505493704
F1: 0.9043354961966713
Matthews Correlation Coefficient: 0.46589789232107987
==========================================================================================
==
==========================================================================================
==
Accuracy: 0.9133333333333333
Balanced Accuracy: 0.6876723827797098
AUC-ROC: 0.891878522604629
AUC-PR: 0.5544496506696233
F1: 0.90485517339905
Matthews Correlation Coefficient: 0.45802938072499066
==========================================================================================
==
==========================================================================================
==
Accuracy: 0.905
Balanced Accuracy: 0.7187012831274733
AUC-ROC: 0.8411380261422232
AUC-PR: 0.49049222197604503
F1: 0.9024651372019794
Matthews Correlation Coefficient: 0.4614118176241173
==========================================================================================
==
==========================================================================================
==
Accuracy: 0.9166666666666666
Balanced Accuracy: 0.7038014150377743
AUC-ROC: 0.8516009113802614
AUC-PR: 0.5040352128894131
F1: 0.9094141867864496
Matthews Correlation Coefficient: 0.4862755443217619
==========================================================================================
==
==========================================================================================
==
Accuracy: 0.905
Balanced Accuracy: 0.7017380509000621
AUC-ROC: 0.8342496526854069
AUC-PR: 0.4662358276643991
F1: 0.9001034755069974
Matthews Correlation Coefficient: 0.447516329395179
==========================================================================================
==
|      |   Accuracy |   Balanced Accuracy |   AUC-ROC |    AUC-PR |         F1 |   Ma
tthews Correlation Coefficient |
|:-----|-----------:|--------------------:|----------:|----------:|-----------:|-----
-----------------------------:|
| Mean |  0.909667  |           0.705068  | 0.853818  | 0.490948  | 0.904235   |
0.463826 |
```

```
| Std   |  0.0046428 |          0.0106865 | 0.0200507 | 0.0386234 | 0.00307963 |
0.012757 |
```

The output for the cell abve is shown here, the table is shown at the end of the results

==========

- Accuracy: 0.9083333333333333
- Balanced Accuracy: 0.7134248710876604
- AUC-ROC: 0.850221849142583
- AUC-PR: 0.43952624505493704
- F1: 0.9043354961966713
- Matthews Correlation Coefficient: 0.46589789232107987

==========

- Accuracy: 0.9133333333333333
- Balanced Accuracy: 0.6876723827797098
- AUC-ROC: 0.891878522604629
- AUC-PR: 0.5544496506696233
- F1: 0.90485517339905
- Matthews Correlation Coefficient: 0.45802938072499066

==========

- Accuracy: 0.905
- Balanced Accuracy: 0.7187012831274733
- AUC-ROC: 0.8411380261422232
- AUC-PR: 0.49049222197604503
- F1: 0.9024651372019794
- Matthews Correlation Coefficient: 0.4614118176241173

==========

- Accuracy: 0.9166666666666666
- Balanced Accuracy: 0.7038014150377743
- AUC-ROC: 0.8516009113802614
- AUC-PR: 0.5040352128894131
- F1: 0.9094141867864496
- Matthews Correlation Coefficient: 0.4862755443217619

==========

- Accuracy: 0.905
- Balanced Accuracy: 0.7017380509000621
- AUC-ROC: 0.8342496526854069
- AUC-PR: 0.4662358276643991
- F1: 0.9001034755069974
- Matthews Correlation Coefficient: 0.447516329395179

	Accuracy	Balanced Accuracy	AUC-ROC	AUC-PR	F1	Matthews Correlation Coefficient
Mean	0.909667	0.705068	0.853818	0.490948	0.904235	0.463826
Std	0.0046428	0.0106865	0.0200507	0.0386234	0.00307963	0.012757

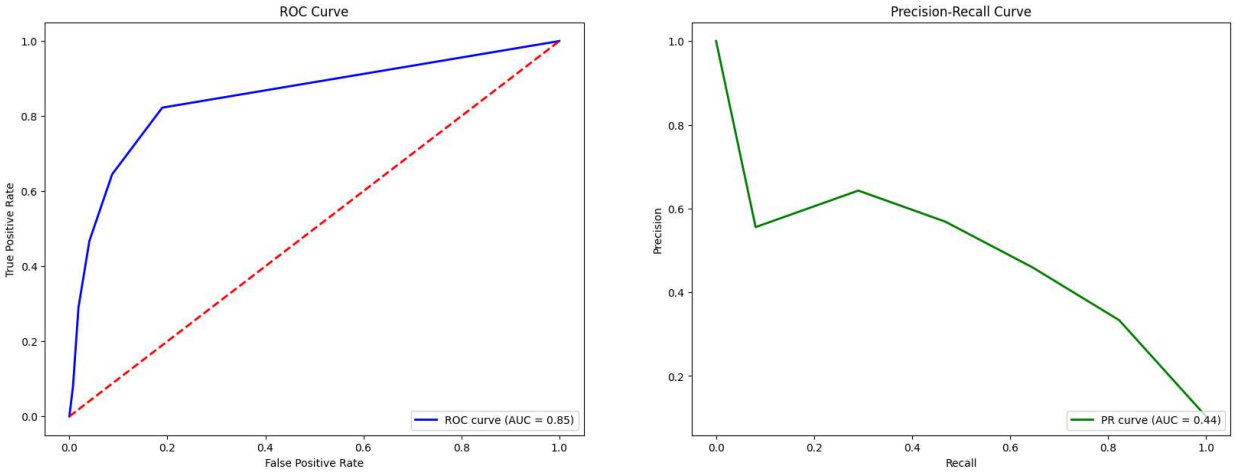
Question 2.II - Plotting ROC Curves

In [ ]:
```
def plot_roc_pr(y_true, y_proba, title):
    # Calculate the ROC curve
    fpr, tpr, _ = roc_curve(y_true, y_proba[:, 1])

    # Calculate the PR curve
    precision, recall, _ = precision_recall_curve(y_true, y_proba[:, 1])

    # Plot the ROC curve
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % ro
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve ({title})')
    plt.legend(loc='lower right')

    # Plot the PR curve
    plt.subplot(1, 2, 2)
    plt.plot(recall, precision, color='darkorange', lw=2, label='PR curve (area = %0.2
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title(f'PR Curve ({title})')
    plt.legend(loc='lower right')

    plt.show()
```

In [ ]:
```
# Select a fold for demonstration
train_index, test_index = next(kf.split(Xtrain, Ytrain))
X_train, X_test = Xtrain[train_index], Xtrain[test_index]
Y_train, Y_test = Ytrain[train_index], Ytrain[test_index]

knn.fit(X_train, Y_train)
```

```
Y_proba = knn.predict_proba(X_test)
plot_roc_pr(Y_test, Y_proba, 'KNN')
```

Out[ ]: <matplotlib.legend.Legend at 0x1f558c8e5b0>



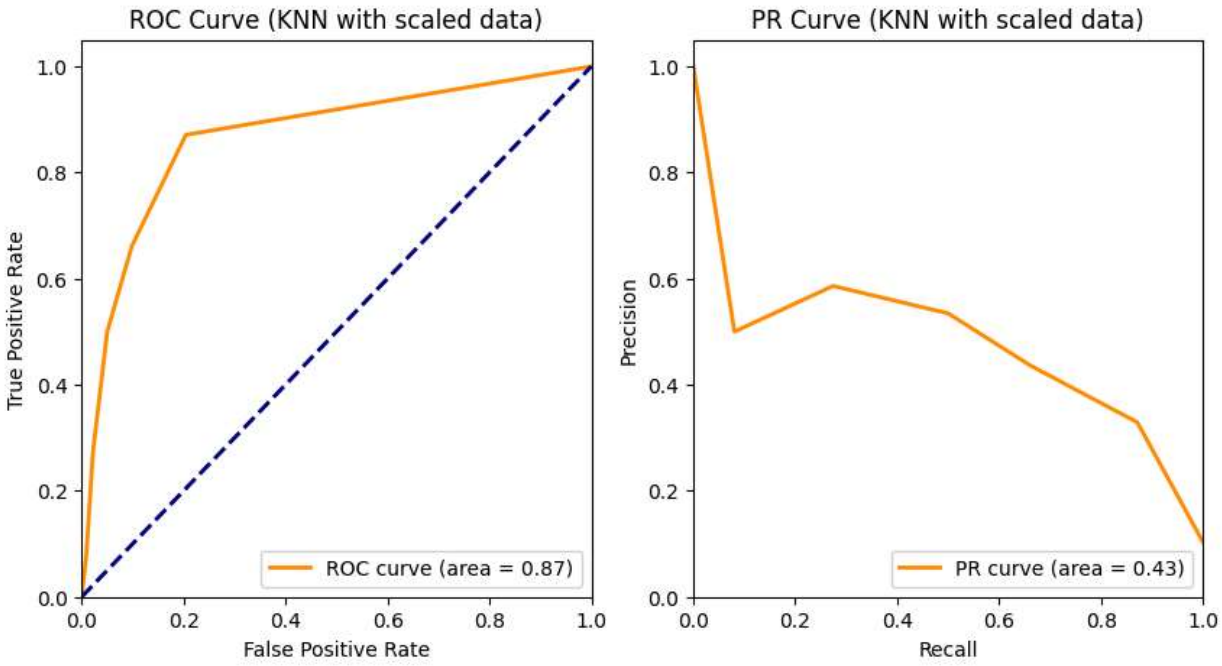
- **ROC Curve**: Reflects the trade-off between the True Positive Rate (TPR) and False Positive Rate (FPR). The area under the curve (AUC) provides an aggregate measure of performance across these.
- **PR Curve**: Shows the trade-off between precision and recall for different thresholds. This may be more informative since the dataset is very unbalanced.

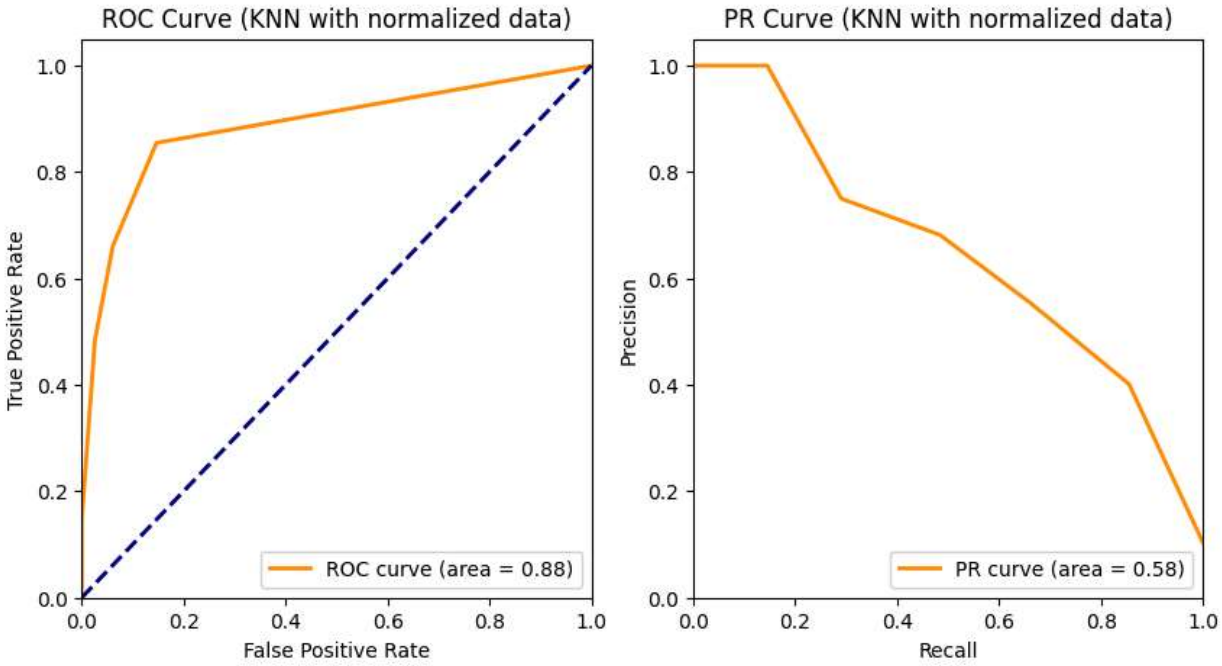
The most important section of this ROC curve is the beginning end (between FPR = 0 and 0.2). It is harder to predict the positive class since our dataset is so heavily weighted towards the negative class. Therefore we want our false positive rate to be as little as possible and the true positive rate to be as high as possible.

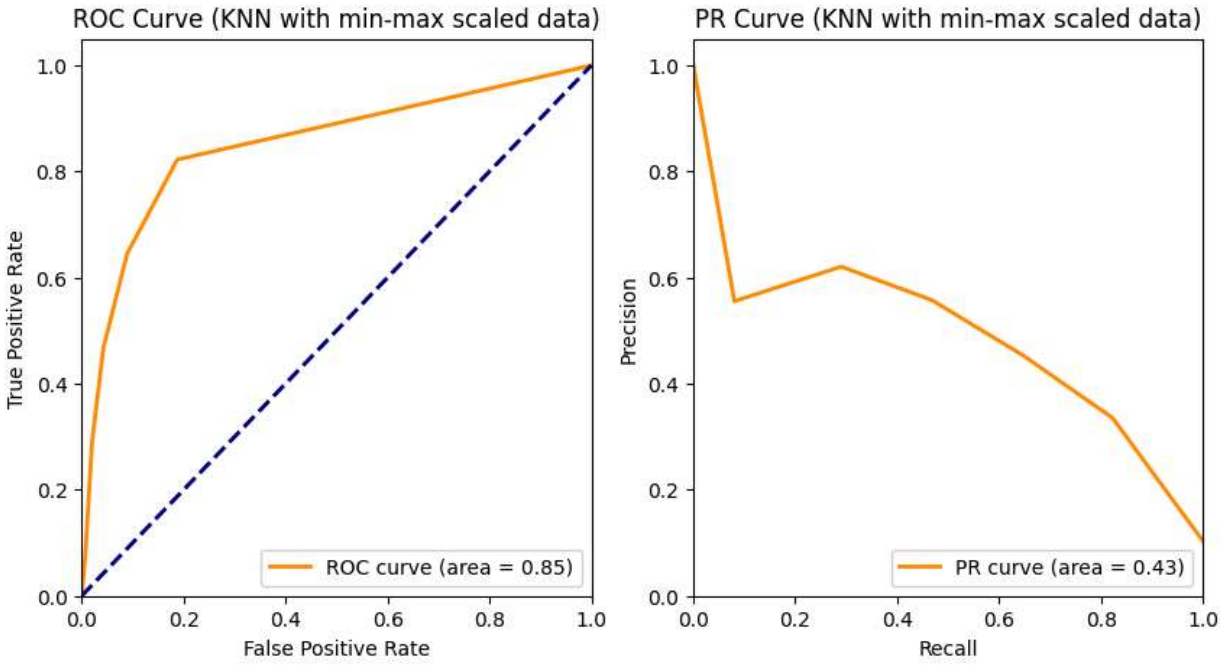
iii Various forms of Pre-Processing

In [ ]:
```
from sklearn.preprocessing import StandardScaler
# Now, we want to process teh data in various ways to see if we can improve the perfor
train_index, test_index = next(kf.split(Xtrain, Ytrain))
X_train, X_test = Xtrain[train_index], Xtrain[test_index]
Y_train, Y_test = Ytrain[train_index], Ytrain[test_index]


# 1. Standard Scaling
scaler = StandardScaler()
Xtrain_scaled = scaler.fit_transform(X_train)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(Xtrain_scaled, Y_train)
X_test_scaled = scaler.transform(X_test)
Y_proba = knn.predict_proba(X_test_scaled)
plot_roc_pr(Y_test, Y_proba, 'KNN with scaled data')

# 2. Normalisation
from sklearn.preprocessing import Normalizer
scaler = Normalizer()
Xtrain_normalized = scaler.fit_transform(X_train)
```

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(Xtrain_normalized, Y_train)
X_test_normalized = scaler.transform(X_test)
Y_proba = knn.predict_proba(X_test_normalized)
plot_roc_pr(Y_test, Y_proba, 'KNN with normalized data')

# 3. Min-Max Scaling
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
Xtrain_minmax = scaler.fit_transform(X_train)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(Xtrain_minmax, Y_train)
X_test_minmax = scaler.transform(X_test)
Y_proba = knn.predict_proba(X_test_minmax)
plot_roc_pr(Y_test, Y_proba, 'KNN with min-max scaled data')
```







As shown by the mean and standard deviation scaling, the PR curve is significantly higher, however, the ROC curve remains mosttly the same. As for the training time, the time taken to train the model seemed very similar. When scaling data, it will depend on how the data is scaled. Data could be scaled up to be more complexx (such as a higher dimensionality), which could increase the training time as well as the performance of the model. However if you scale down and make the model simpler, the training time could also reduce.

Question 3: Cross Validation of SVMs and RFs

The following code and markdown comments show the approach of the question, and below the blocks of code, the solutions and explanations are provided

In [ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import PrecisionRecallDisplay, RocCurveDisplay
import seaborn as sns
```

In [ ]:
```
Xtrain = pd.read_csv("./Xtrain.csv", delimiter=' ', header=None)
Ytrain = pd.read_csv("./Ytrain.csv", delimiter=' ', header=None)
XTest = pd.read_csv("./Xtest.csv", delimiter=' ', header=None)
Xtrain = np.array(Xtrain)
XTest = np.array(XTest)
Ytrain = np.array(Ytrain).ravel()
```

First, our SVM Classifier

First, using a linear kernel.

In [ ]:
```
def perform_linear_grid_search(C_values, pipeline, scoring):
    """
    Perform grid search for a linear model.
    """
    results_vector = np.zeros(len(C_values))

    for i, C in enumerate(C_values):
        param_grid_linear = {
            'svm__C': [C],
        }
        grid_search_linear = GridSearchCV(pipeline, param_grid_linear, cv=5, scoring=s
        grid_search_linear.fit(Xtrain, Ytrain)
        results_vector[i] = grid_search_linear.best_score_

    return results_vector
```

In [ ]:
```
def plot_performnace_matrix(results_matrix, C_values, degrees, title):
    """
    Plot a performance matrix using a heatmap.
    """
    plt.figure(figsize=(10, 6))
    sns.heatmap(results_matrix, annot=True, fmt=".3f", cmap='viridis', xticklabels=deg
    plt.xlabel('Degree')
    plt.ylabel('C')
    plt.title(title)
    plt.show()
```

In [ ]:
```
# Please note that this code is based off the code provided on the sklearn website
def plot_mean_std(ax, mean_x, curves, aucs, xlabel, ylabel, curve_name):
    """
    Plots the mean curve and standard deviation curve with shaded region on the given
    """

    mean_curve = np.mean(curves, axis=0)
    std_auc = np.std(aucs)
    ax.plot(
        mean_x,
        mean_curve,
        color="b",
        label=f"Mean {curve_name} (AUC = %0.2f $\\pm$ %0.2f)" % (np.mean(aucs), std_au
        lw=2,
        alpha=0.8,
    )
    std_curve = np.std(curves, axis=0)
    curves_upper = np.minimum(mean_curve + std_curve, 1)
    curves_lower = np.maximum(mean_curve - std_curve, 0)
    ax.fill_between(
        mean_x,
        curves_lower,
        curves_upper,
        color="grey",
        alpha=0.2,
        label=r"$\pm$ 1 std. dev.",
    )
    ax.set(xlabel=xlabel, ylabel=ylabel)
```

```
    ax.legend(loc="lower right")
    return mean_curve
```

In [ ]:

```
# Please note that this code is based off the code provided on the sklearn website
def plot_cross_validation_curves(pipeline, X_train, Y_train, n_folds=5):
    """
    Plots the cross-validation curves for a given pipeline.
    """

    CV = StratifiedKFold(n_splits=n_folds)
    tprs = []
    aucs = []
    precisions = []
    pr_aucs = []
    mean_recall = np.linspace(0, 1, 100)

    fig, axes = plt.subplots(1, 2, figsize=(16, 8))
    mean_fpr = np.linspace(0, 1, 100)

    for i, (train_index, test_index) in enumerate(CV.split(X_train, Y_train)):
        estimator = pipeline
        estimator.fit(X_train[train_index], Y_train[train_index])

        # ROC
        viz_roc = RocCurveDisplay.from_estimator(
            estimator,
            X_train[test_index],
            Y_train[test_index],
            name=f"ROC fold {i}",
            alpha=0.3,
            lw=1,
            ax=axes[0],
        )
        interp_tpr = np.interp(mean_fpr, viz_roc.fpr, viz_roc.tpr)
        interp_tpr[0] = 0.0
        tprs.append(interp_tpr)
        aucs.append(viz_roc.roc_auc)

        # Precision-Recall
        viz_pr = PrecisionRecallDisplay.from_estimator(
            estimator,
            X_train[test_index],
            Y_train[test_index],
            name=f"PR fold {i}",
            alpha=0.3,
            lw=1,
            ax=axes[1],
        )
        interp_precision = np.interp(mean_recall, viz_pr.recall[::-1], viz_pr.precisio
        precisions.append(interp_precision)
        pr_auc = average_precision_score(Y_train[test_index], estimator.predict_proba(
        pr_aucs.append(pr_auc)

    # Plot ROC Mean and Std
    axes[0].plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Random', alph
    mean_roc = plot_mean_std(axes[0], mean_fpr, tprs, aucs, "False Positive Rate", "Tr
```

```
    # Plot Precision-Recall Mean and Std
    mean_pr = plot_mean_std(axes[1], mean_recall, precisions, pr_aucs, "Recall", "Prec

    fig.suptitle("Cross-validation Curves")
    plt.show()
    return estimator, mean_roc, mean_pr
```

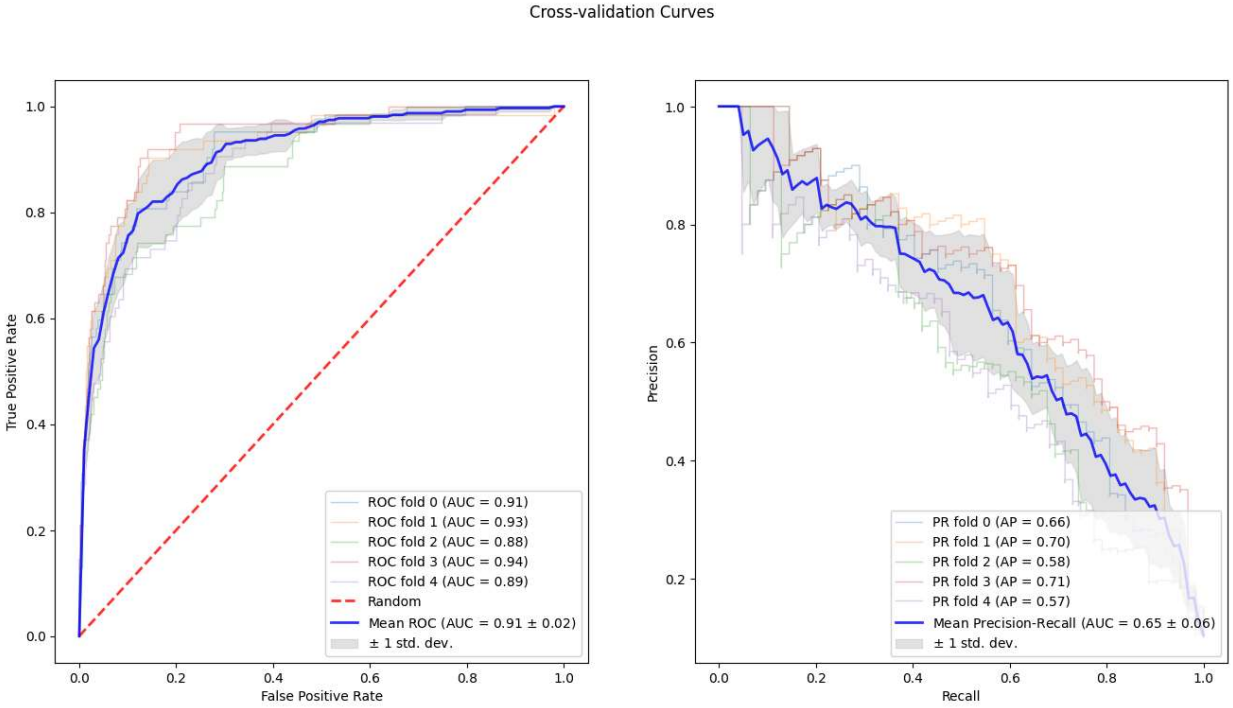
In [ ]:
```
def cross_validation_metrics(pipeline, X_train, Y_train, n_folds=5):
    """
    Get the validation metrics for a given model
    """
    CV = StratifiedKFold(n_splits=n_folds)
    accuracies = []
    balanced_accuracies = []
    f1_scores = []
    pr_aucs = []
    roc_aucs = []
    mccs = []

    # Calculate the metrics for each fold
    for train_index, test_index in CV.split(X_train, Y_train):
        estimator = pipeline
        estimator.fit(X_train[train_index], Y_train[train_index])
        y_pred = estimator.predict(X_train[test_index])
        y_proba = estimator.predict_proba(X_train[test_index])[:, 1]

        accuracies.append(accuracy_score(Y_train[test_index], y_pred))
        balanced_accuracies.append(balanced_accuracy_score(Y_train[test_index], y_pred
        f1_scores.append(f1_score(Y_train[test_index], y_pred))
        pr_aucs.append(average_precision_score(Y_train[test_index], y_proba))
        roc_aucs.append(roc_auc_score(Y_train[test_index], y_proba))
        mccs.append(matthews_corrcoef(Y_train[test_index], y_pred))

    metrics = {
        'mean_accuracy': np.mean(accuracies),
        'mean_balanced_accuracy': np.mean(balanced_accuracies),
        'mean_f1_score': np.mean(f1_scores),
        'mean_auc_pr': np.mean(pr_aucs),
        'mean_auc_roc': np.mean(roc_aucs),
        'mean_mcc': np.mean(mccs),
    }

    return metrics
```

In [ ]:
```
def get_best_params(results_matrix, rows, columns):
    """
    Finds the best parameters based on the results matrix.
    """

    row_index, column_index = np.unravel_index(np.argmax(results_matrix), results_matr
    best_row = rows[row_index]
    best_column = columns[column_index]

    return best_row, best_column
```

Our linear grid search

In [ ]:
```
# Define the parameter grid for SVM
C_values = [0.01, 0.05, 0.1, 0.5, 1, 5, 10]

# Define the pipeline for linear SVM
pipeline_svm_linear = Pipeline([('scaler', Normalizer()), ('svm', SVC(probability=True


results = perform_linear_grid_search(C_values, pipeline_svm_linear, 'roc_auc')
```

In [ ]:
```
best_c_linear = C_values[np.argmax(results)]

# Define the pipeline for linear SVM
best_estimator_linear = Pipeline([('scaler', Normalizer()), ('svm', SVC(probability=Tr

linear_performance_metrics = cross_validation_metrics(best_estimator_linear, Xtrain, Y

# Perform cross-validation and plot the curves
linear_estimator, linear_mean_roc, linear_mean_pr = plot_cross_validation_curves(best_
```



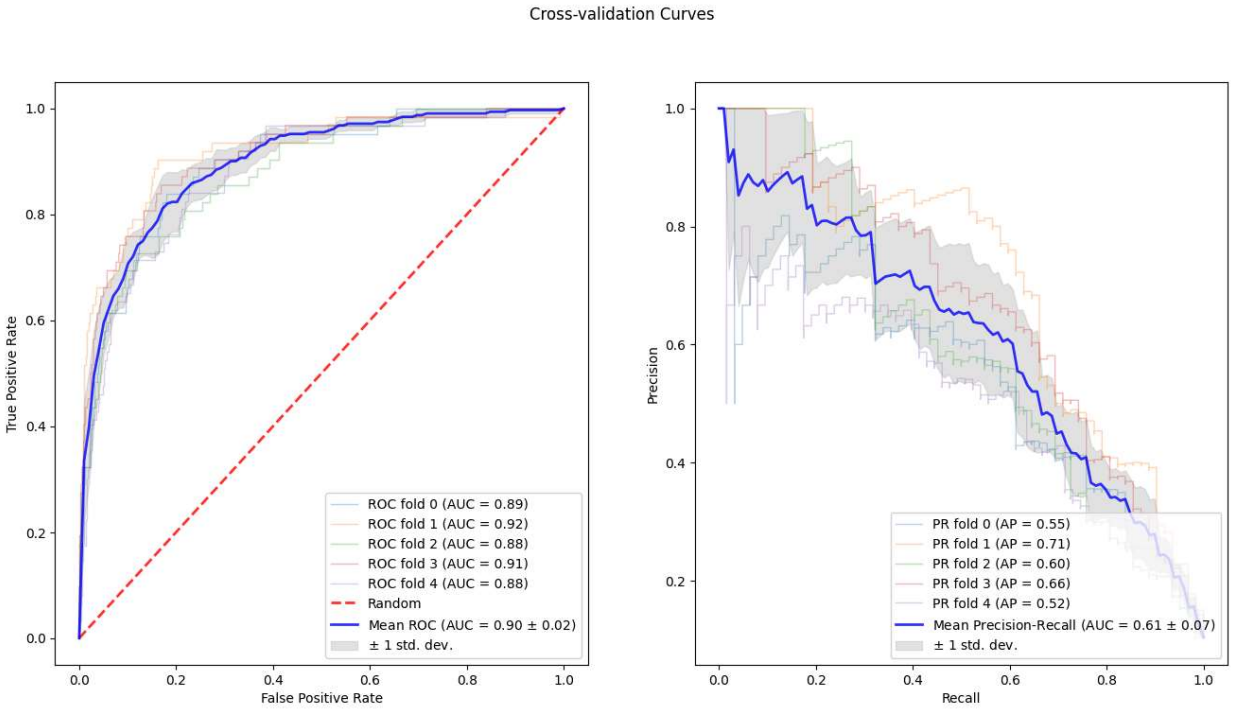
Now our polynomail grid search

In [ ]:
```
def perform_poly_grid_search(C_values, pipeline, gamma, scoring, X_train, Y_train):
    """
    Perform grid search with polynomial SVM and build a performance matrix.
    """
    results_matrix = np.zeros((len(C_values), 4))
    for index, degree in enumerate(range(1, 5)):
        for i, C in enumerate(C_values):
            param_grid_poly = {
                'svm__C': [C],
                'svm__degree': [degree],
                'svm__gamma': [gamma],
            }
            grid_search_poly = GridSearchCV(pipeline, param_grid_poly, cv=5, scoring=s
```

```
            grid_search_poly.fit(X_train, Y_train)
            results_matrix[i, index] = grid_search_poly.best_score_

    return results_matrix
```

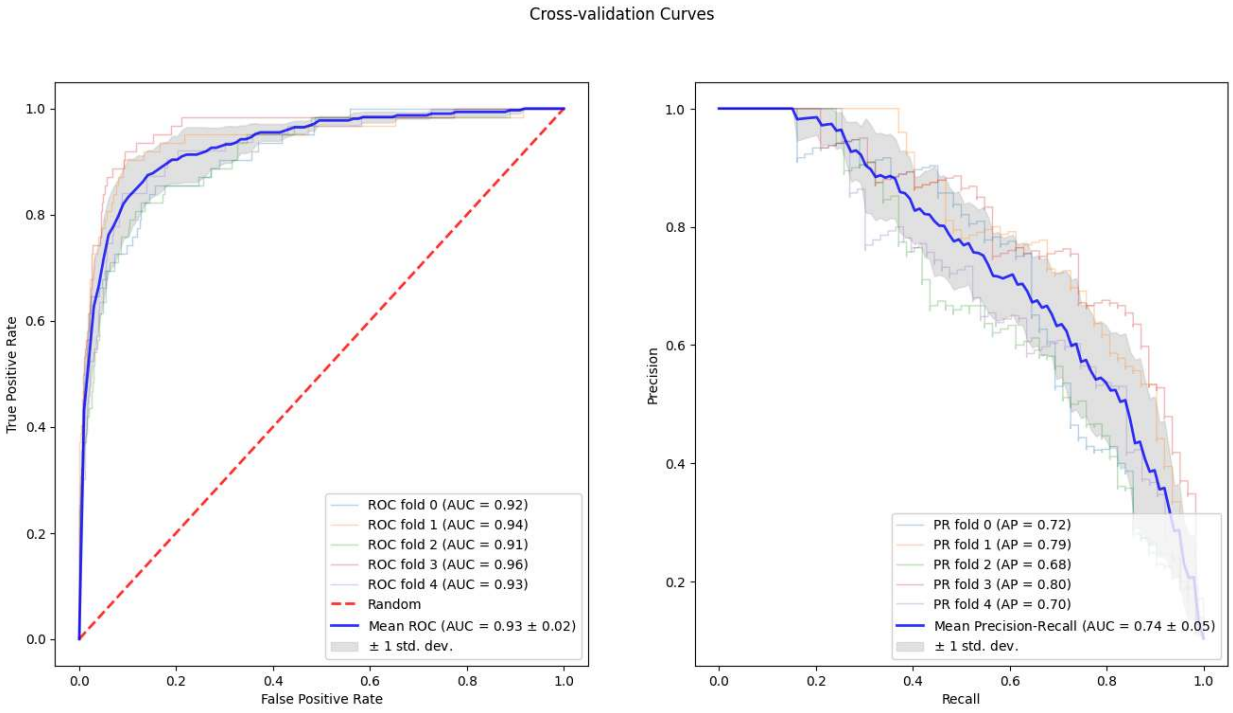
In [ ]:
```
# Do this once for auto and once for scale
pipeline_svm_auto = Pipeline([('scaler', Normalizer()), ('svm', SVC(probability=True,
pipeline_svm_scale = Pipeline([('scaler', Normalizer()), ('svm', SVC(probability=True,

# Get the results
results_auto = perform_poly_grid_search(C_values, pipeline_svm_auto, 'auto', 'average_
results_scale = perform_poly_grid_search(C_values, pipeline_svm_scale, 'scale', 'avera
```

In [ ]:
```
best_c_poly_auto, best_degree_poly_auto = get_best_params(results_auto, C_values, [i f

# Best estimator the for kernel being auto
best_estimator_auto = Pipeline([
    ('scaler', Normalizer()),
    ('svm', SVC(probability=True, C=best_c_poly_auto, degree=best_degree_poly_auto, ga
])

# Get the performance metrics
poly_auto_performance_metrics = cross_validation_metrics(best_estimator_auto, Xtrain,

poly_auto_estimator, poly_auto_mean_roc, poly_auto_mean_pr = plot_cross_validation_cur
```

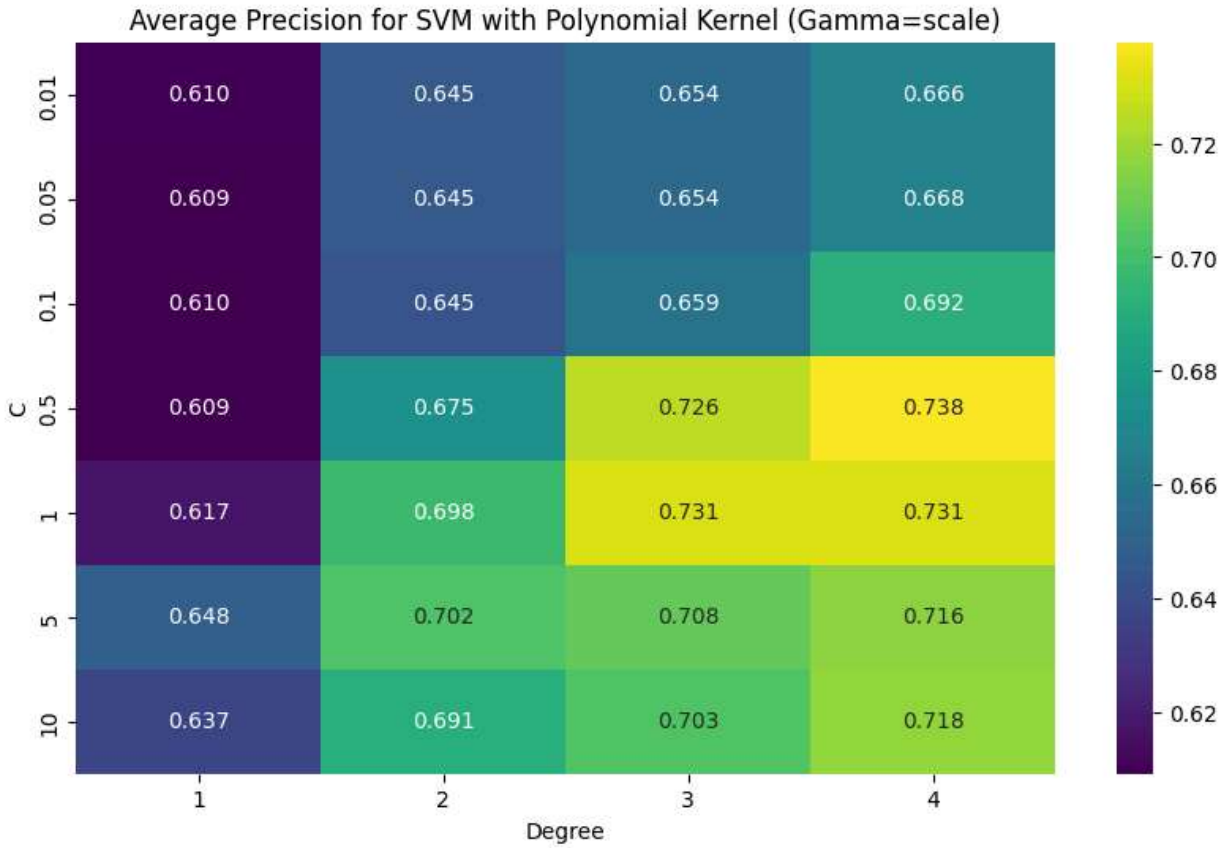


In [ ]:
```
best_c_poly_scale, best_degree_poly_scale = get_best_params(results_scale, C_values, [

# Best estimator the for kernel being auto
best_estimator_scale = Pipeline([
    ('scaler', Normalizer()),
    ('svm', SVC(probability=True, C=best_c_poly_scale, degree=best_degree_poly_scale,
])

poly_scale_performance_metrics = cross_validation_metrics(best_estimator_scale, Xtrain
```

```
poly_scale_estimator, poly_scale_mean_roc, poly_scale_mean_pr = plot_cross_validation_
```



In [ ]:
```
plot_performnace_matrix(results_scale, C_values, range(1, 5), 'Average Precision for S
```



Now we do the same with the rbf kernel

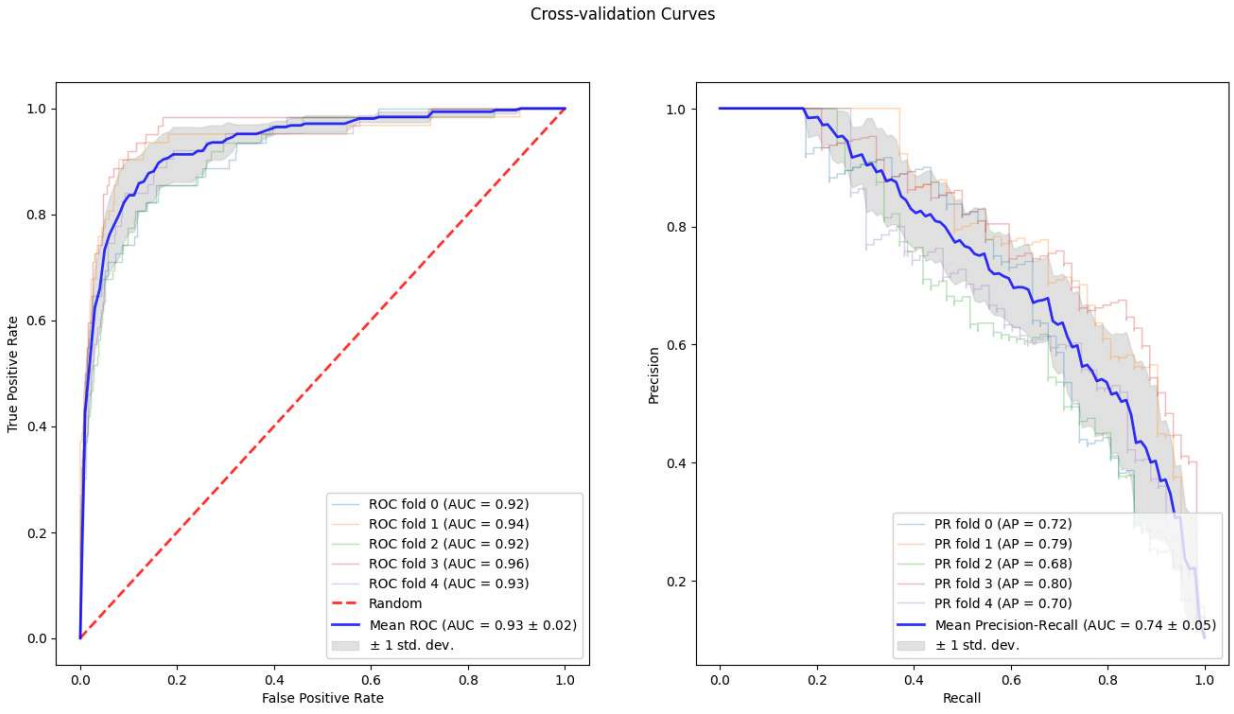
In [ ]:
```
# Do this once for auto and once for scale
pipeline_svm_rbf = Pipeline([('scaler', Normalizer()), ('svm', SVC(probability=True, k
```

```
gammas = ['scale', 'auto']
```

In [ ]:
```
def perform_rbf_grid_search(C_values, pipeline, gammas, scoring):

    results_matrix = np.zeros((len(C_values), len(gammas)))

    for index, gamma in enumerate(gammas):
        for i, C in enumerate(C_values):
            param_grid_rbf = {
                'svm__C': [C],
                'svm__gamma': [gamma],
            }
            grid_search_poly = GridSearchCV(pipeline, param_grid_rbf, cv=5, scoring=sc
            grid_search_poly.fit(Xtrain, Ytrain)
            results_matrix[i, index] = grid_search_poly.best_score_

    return results_matrix
```

In [ ]:
```
results_auto_rbf = perform_rbf_grid_search(C_values, pipeline_svm_rbf, gammas, 'averag
```

In [ ]:
```
best_c_rbf, best_gamma_rbf = get_best_params(results_auto_rbf, C_values, gammas)

# Best estimator for RBF kernel
best_estimator_rbf = Pipeline([
    ('scaler', Normalizer()),
    ('svm', SVC(probability=True, C=best_c_rbf, gamma=best_gamma_rbf, kernel='rbf', ra
])

rbf_performance_metrics = cross_validation_metrics(best_estimator_rbf, Xtrain, Ytrain)

# Plot the results
rbf_estimator, rbf_mean_roc, rbf_mean_pr = plot_cross_validation_curves(best_estimator
```



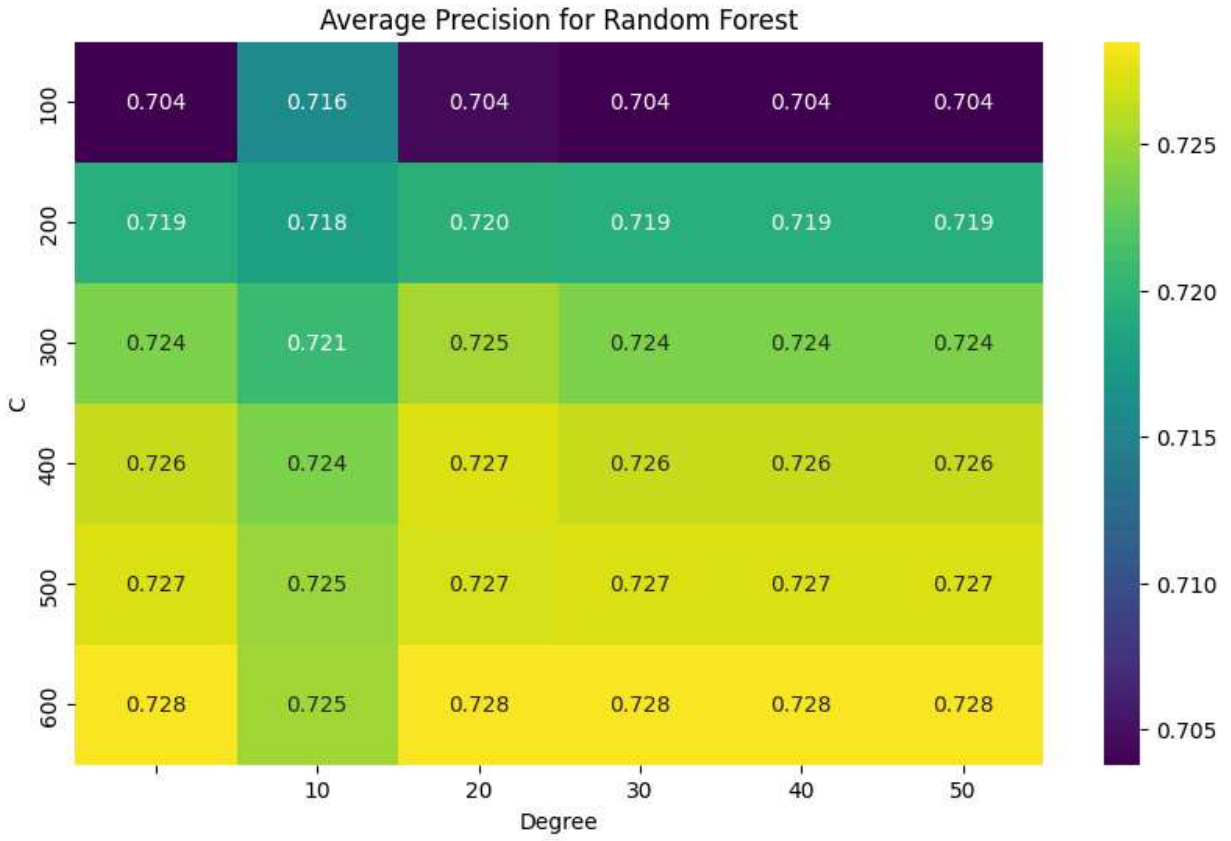
Second, our Random Forest Classifier

In [ ]:
```
rf_n_estimators = [100, 200, 300, 400, 500, 600]
rf_max_depth = [None, 10, 20, 30, 40, 50]
rf_min_samples_split = [2, 5, 10]

pipeline_rf =  Pipeline([('scaler', Normalizer()), ('rf', RandomForestClassifier(n_job
```

In [ ]:
```
def perform_rf_grid_search(pipeline, rf_n_estimators, rf_max_depth, scoring, Xtrain, Y
    results_matrix = np.zeros((len(rf_n_estimators), len(rf_max_depth)))

    for index, depth in enumerate(rf_max_depth):
        for i, n_estimator in enumerate(rf_n_estimators):
            param_grid_rf = {
                'rf__n_estimators': [n_estimator],
                'rf__max_depth': [depth],
            }
            grid_search_rf = GridSearchCV(estimator=pipeline, param_grid=param_grid_rf
            grid_search_rf.fit(Xtrain, Ytrain)
            results_matrix[i, index] = grid_search_rf.best_score_

    return results_matrix
```

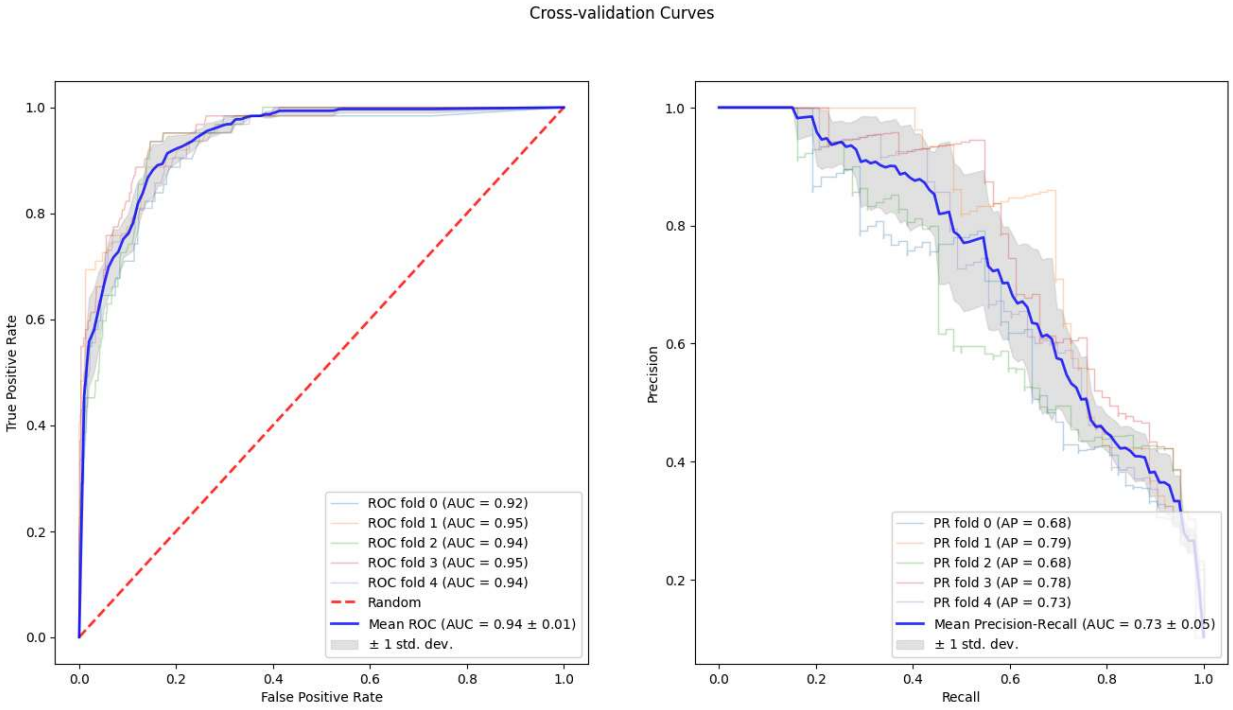
In [ ]:
```
results_matrix_rf = perform_rf_grid_search(pipeline_rf, rf_n_estimators, rf_max_depth,
```

In [ ]:
```
plot_performnace_matrix(results_matrix_rf, rf_n_estimators, rf_max_depth, 'Average Pre
```



In [ ]:
```
best_n_estimator, best_max_depth = get_best_params(results_matrix_rf, rf_n_estimators,
```

```
# Best estimator for Random Forest
best_estimator_rf = Pipeline([('scaler', Normalizer()), ('rf', RandomForestClassifier(

rf_performance_metrics = cross_validation_metrics(best_estimator_rf, Xtrain, Ytrain)

rf_estimator, rf_mean_roc, rf_mean_pr = plot_cross_validation_curves(best_estimator_rf
```



Our Results

i) My strategy for selecting the optimal classifier was to perform a grid search over the different classifiers based on their AUC-PR curve (due to the highligy imbalanced dataset that we have). As you may seee by the code above, there were grid searches performed for each of the types of classifiers. Of course this could all be done in one block of code with a grid search of all parameters across all classifiers, however this was opted against to outline the steps of the work more accurately. Moreover, various pre-processors were trialled (the ones shown in question 2), however the normaliser gave a much higher AUC-PR value for the classifiers (which was our measurement for the grid search for the reasons stated previously). Therefore, in our final classifier this was opted for. For detailed results of each of the clasifiers please refer to the cells above.

ii) The comparison of the classifiers is shown in the table below: | | mean_accuracy | mean_balanced_accuracy | mean_f1_score | mean_auc_pr | mean_auc_roc | mean_mcc | |:---------|----------:|----------:|----------:|----------:|----------:|----------:| | Linear SVM | 0.924667 | 0.703627 | 0.53774 | 0.646327 | 0.909701 | 0.523777 | | Poly SVM (Gamma=auto) | 0.896333 | 0.5 | 0 | 0.607483 | 0.896547 | 0 | | Poly SVM (Gamma=scale) | 0.934 | 0.742978 | 0.60912 | 0.737862 | 0.931794 | 0.593235 | | RBF SVM | 0.933333 | 0.744032 | 0.60788 | 0.737805 | 0.933464 | 0.590119 | | Random Forest | 0.924 | 0.643527 | 0.439412 | 0.728496 | 0.939721 | 0.492535 |

As shwon by the table above, the RBF SVM and the poly SVM show very similar performanced across all metrics. Additionally they outperform the other classifiers in most metrics., Particularly when considering the F1, PR and MCC scores which are vital when considering imbalanced datasets. Howeever, the Random Forest has the highest mestric for ROC-AUC, however doesn't perform as well in the other metrics. Finally, accuracy is not the most relevant factor here due to the imbalanced nature of the dataset.

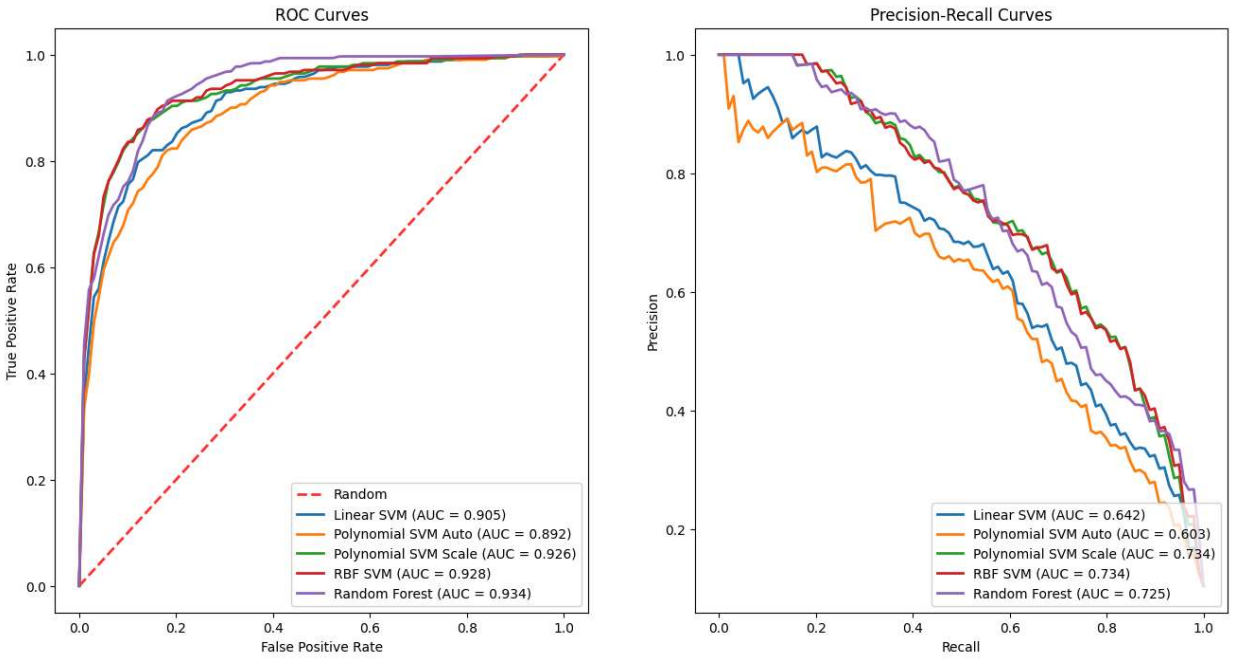
In [ ]:
```
all_performance_metrics = {
    'Linear SVM': linear_performance_metrics,
    'Poly SVM (Gamma=auto)': poly_auto_performance_metrics,
    'Poly SVM (Gamma=scale)': poly_scale_performance_metrics,
    'RBF SVM': rbf_performance_metrics,
    'Random Forest': rf_performance_metrics
}

# Convert this to a dataframe for better visualization
performance_df = pd.DataFrame(all_performance_metrics).T
# print(performance_df.to_markdown())
```

iii) and **iv)** The Plots showing the ROC and PR curves of all the classifiers are shown in the cells below

In [ ]:
```
mean_rocs = {
    'Linear SVM': linear_mean_roc,
    'Polynomial SVM Auto': poly_auto_mean_roc,
    'Polynomial SVM Scale': poly_scale_mean_roc,
    'RBF SVM': rbf_mean_roc,
    'Random Forest': rf_mean_roc
}

mean_prs ={
    'Linear SVM': linear_mean_pr,
    'Polynomial SVM Auto': poly_auto_mean_pr,
    'Polynomial SVM Scale': poly_scale_mean_pr,
    'RBF SVM': rbf_mean_pr,
    'Random Forest': rf_mean_pr

}

fig, axes = plt.subplots(1, 2, figsize=(16, 8))
mean_recall = np.linspace(0, 1, 100)
axes[0].plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Random', alpha=.8
for name, mean_roc in mean_rocs.items():
    axes[0].plot(mean_recall, mean_roc, lw=2, label=f"{name} (AUC = %0.3f)" % np.mean(

axes[0].set(xlabel='False Positive Rate', ylabel='True Positive Rate')
axes[0].legend(loc="lower right")
axes[0].set_title('ROC Curves')

for name, mean_pr in mean_prs.items():
    axes[1].plot(mean_recall, mean_pr, lw=2, label=f"{name} (AUC = %0.3f)" % np.mean(m

axes[1].set(xlabel='Recall', ylabel='Precision')
```

```
axes[1].legend(loc="lower right")
axes[1].set_title('Precision-Recall Curves')
```

Out[ ]: Text(0.5, 1.0, 'Precision-Recall Curves')



v) Firstly, all classifiers perform significantly better than a random classifier which is indicated by the red dashed line. RandomForest shows the highest AUC amonst all classifiers regarding the ROC value. However, for polynomial SVM with hyperparameter gamma = scale has the highest AUC for the PR curve. This implies that this classifier performs best when distinguishing between the positive and negative class. Additionally, since the ROC curve for Random Forest seems to have an issue with classification at the lower end of the scale, this reinforces this fact. Thererfore, it could suggest that the polynomial SVM could be the best classifier for this task. Finally, the curves for the RBF and the polynomial kernel are very very similar, showing almost identical results for both of these values, implying that the polynomial and RBF kernels could be interchangeable for this task (where the hyperparameters have been optimised).

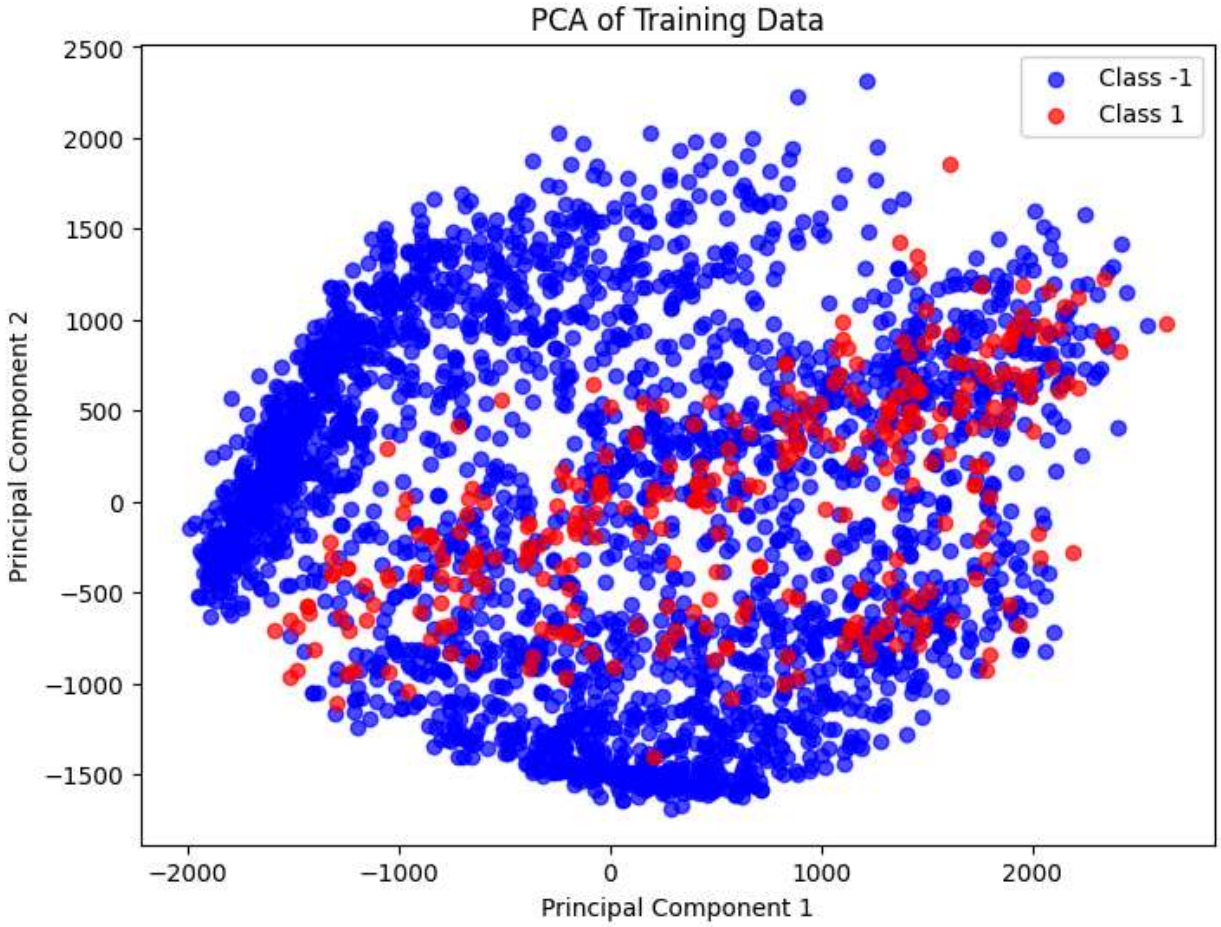
Question 4

In [ ]:
```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np
```

i) The plots for the 2 dimensions are shown below.

In [ ]:
```
# Applying PCA to reduce dimensions to 2
pca = PCA(n_components=2)
Xtrain_reduced = pca.fit_transform(Xtrain)

# Plotting the scatter plot
plt.figure(figsize=(8, 6))
colors = ['blue', 'red']  # Colors for the two classes
```

```
labels = ['-1', '1']  # Labels for the two classes

for i, class_value in enumerate(np.unique(Ytrain)):
    # Find indices where the class matches
    indices = np.where(Ytrain == class_value)[0]
    # Plot the data points for these indices
    plt.scatter(Xtrain_reduced[indices, 0], Xtrain_reduced[indices, 1], color=colors[i

plt.title("PCA of Training Data")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend()
plt.show()
```

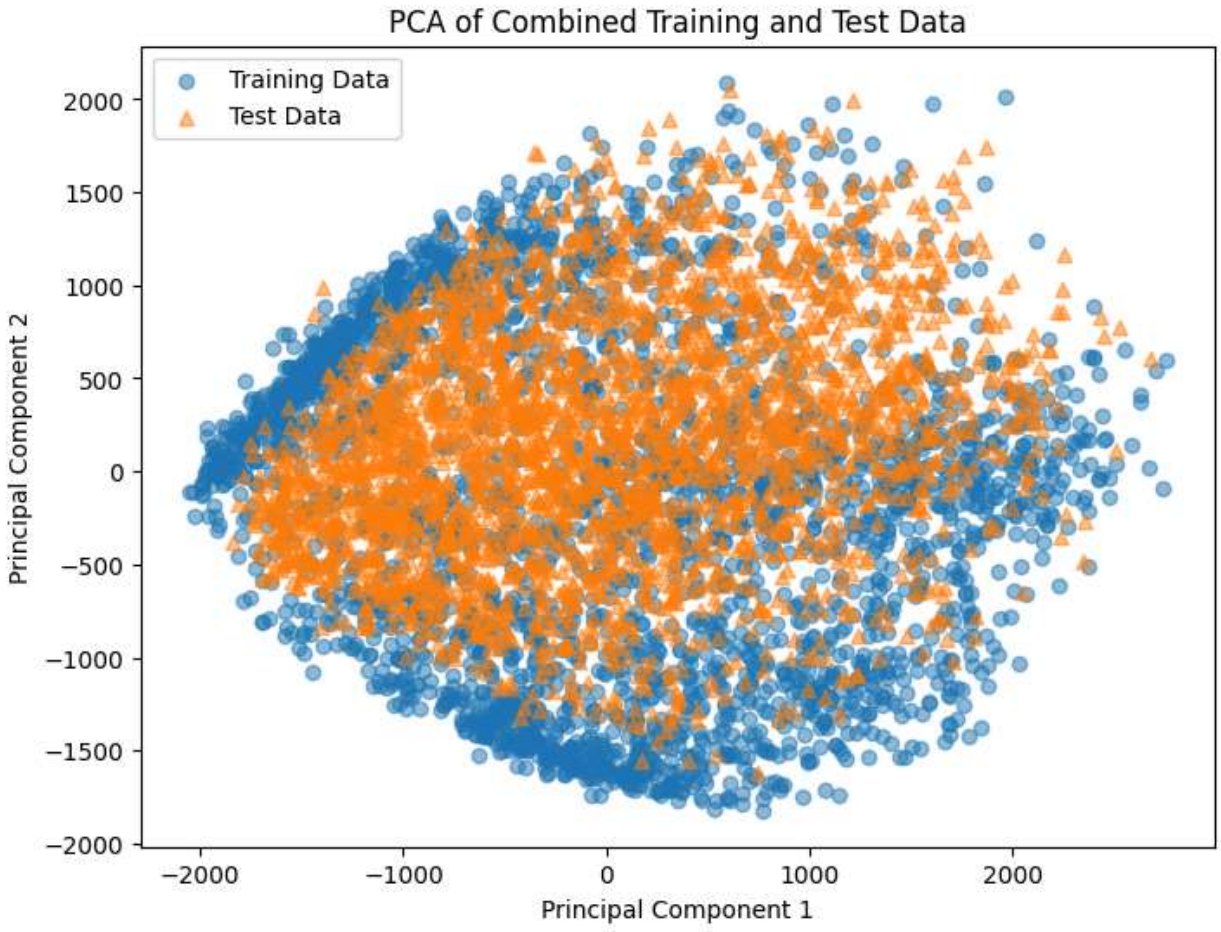


This plot clearly shows that the positive and negative classes are not linearly separable in two dimensions, and that there is alot of overlap with the two calsses themselves. However, you can clearly identify that the top and bottom sections (or groups) could be very well identified as not being part of the positive class. However, in the middle of the data this is very convoluted. Finally, this visualisation does not tell us much about the data itself, but rather provides insight into the distribution of the classes.

ii) Plotting the test vs the training data

In [ ]:
```
# Combining training and test data
X_combined = np.vstack((Xtrain, XTest))
```

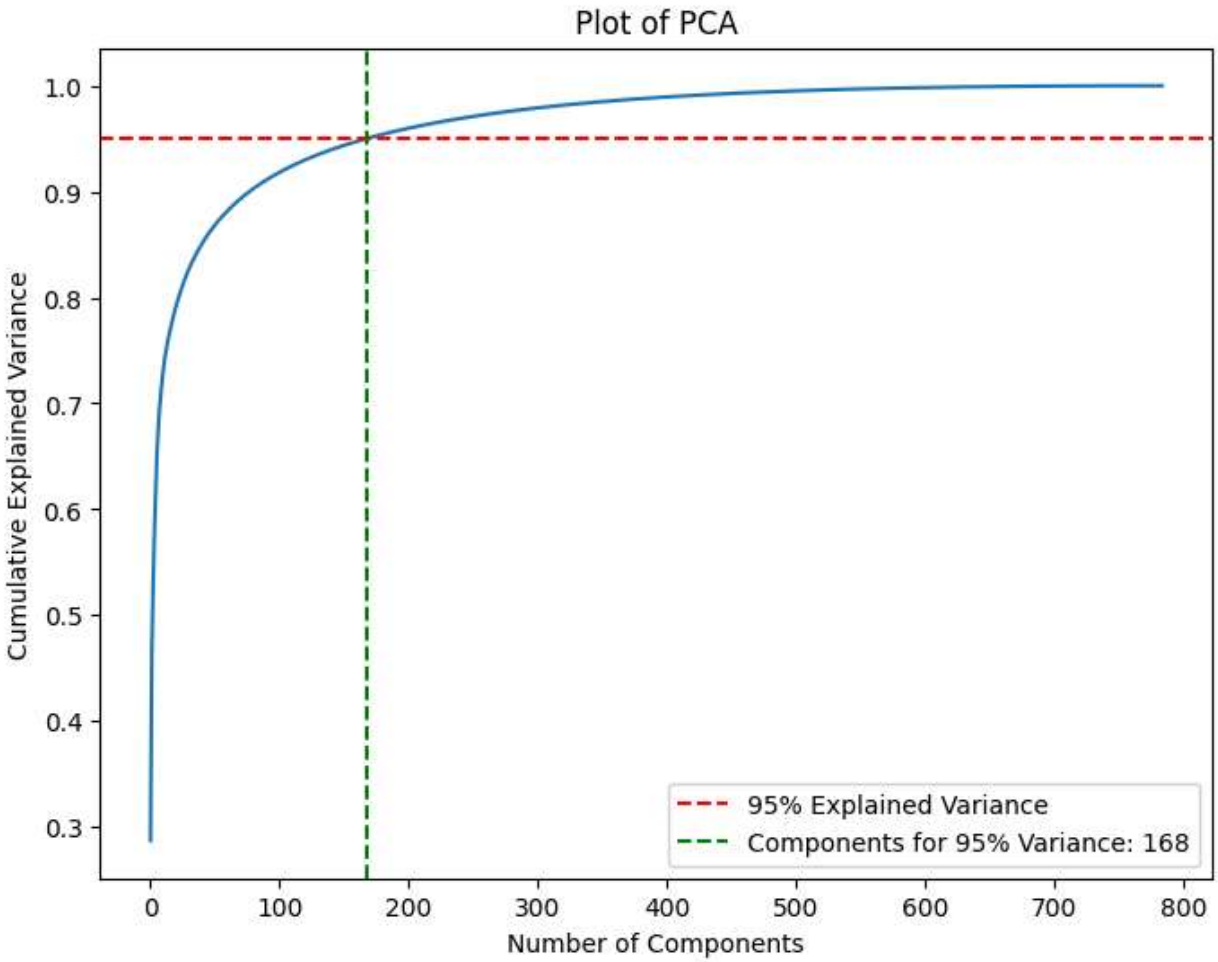
```
# Applying PCA to reduce dimensions to 2
pca_combined = PCA(n_components=2)
X_combined_reduced = pca_combined.fit_transform(X_combined)

# Separating the reduced data back into training and test sets
Xtrain_reduced = X_combined_reduced[:len(Xtrain)]
Xtest_reduced = X_combined_reduced[len(Xtrain):]

# Plotting the scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(Xtrain_reduced[:, 0], Xtrain_reduced[:, 1], label="Training Data", alpha=0

plt.scatter(Xtest_reduced[:, 0], Xtest_reduced[:, 1], marker='^', label="Test Data", a

plt.title("PCA of Combined Training and Test Data")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend()
plt.show()
```



The scatter plot for the train and test data do seem to be relatively similar, and have a large proportion where they overlap. However it appears that they may come from different distributions, since the training set seems to have a higher variance in the Y axis. As with the plot in the previous section, it is very difficult to distinguish between the two calasses for the majority of the data. However, there is once again a section below and above the test data that seems to differentiate very clearly from the test data.

iii) Plotting the scree graph of the PCA, and identifying where 95% variance is explained

In [ ]:
```
# Assuming Xtrain is defined
# Applying PCA without reducing dimensionality
pca_full = PCA()
pca_full.fit(Xtrain)

# Plotting the scree graph
plt.figure(figsize=(8, 6))
cumulative_variance_ratio = np.cumsum(pca_full.explained_variance_ratio_)
plt.plot(cumulative_variance_ratio)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Plot of PCA')

# 95% Explained Variance line
plt.axhline(y=0.95, color='r', linestyle='--', label='95% Explained Variance')

# Finding the number of components for 95% variance
n_components_95 = np.where(cumulative_variance_ratio >= 0.95)[0][0] + 1

# Vertical line at the number of components needed for 95% variance
plt.axvline(x=n_components_95 - 1, color='g', linestyle='--', label=f'Components for 9

plt.legend()
plt.show()

print(f"Number of components to explain 95% variance: {n_components_95}")
```




```
Number of components to explain 95% variance: 168
```

As shown by the plot above, it only requires 168 dimensions to maintain 95% of the original variance.

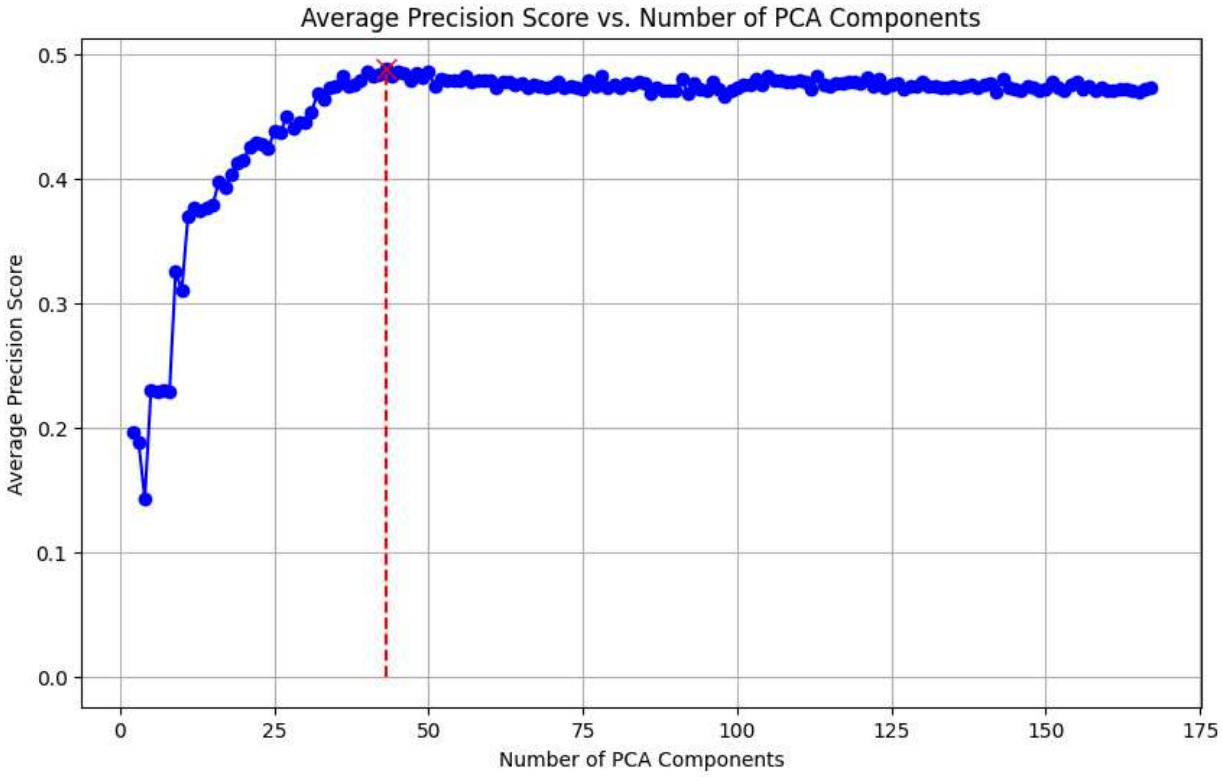
iv) Reducing the number of components in the data.

In [ ]:
```
from sklearn.model_selection import train_test_split, GridSearchCV
```

In [ ]:
```
train_index, test_index = train_test_split(np.arange(len(Xtrain)), test_size=0.2, rand
X_train, X_test = Xtrain[train_index], Xtrain[test_index]
Y_train, Y_test = Ytrain[train_index], Ytrain[test_index]

# Initialize lists to store results
components_list = range(2, 168)
average_precision_scores = []

# Iterate over the range of PCA components
for n_components in components_list:
    # Apply PCA
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)

    # Initialize the SVC with polynomial kernel
    svc = SVC(kernel='poly', degree=4, gamma='scale', probability=True)
```

```
    # Train the SVC model
    svc.fit(X_train_pca, Y_train)

    # Predict on test set
    Y_proba = svc.predict_proba(X_test_pca)

    # Calculate average precision score
    average_precision = average_precision_score(Y_test, Y_proba[:, 1])
    average_precision_scores.append(average_precision)
```

In [ ]:
```
# Plotting the AUC-PR values for each number of PCA components
plt.figure(figsize=(10, 6))
plt.plot(components_list, average_precision_scores, marker='o', linestyle='-', color='
plt.plot(components_list[np.argmax(average_precision_scores)], max(average_precision_s
plt.plot([components_list[np.argmax(average_precision_scores)], components_list[np.arg
plt.title('Average Precision Score vs. Number of PCA Components')
plt.xlabel('Number of PCA Components')
plt.ylabel('Average Precision Score')
plt.grid(True)
plt.show()
```

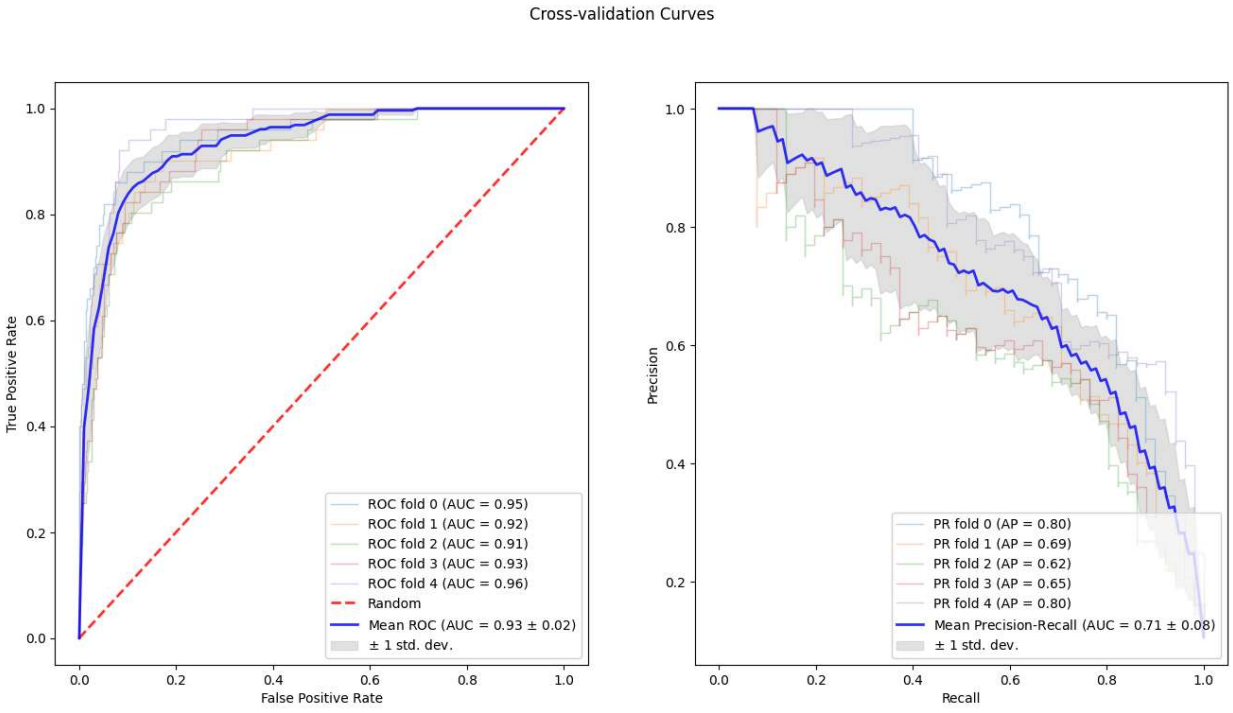


This plot shows that initially, when there are less PC components, the classifier is very very poor at distinguishing between the classes. However, this very rapidly increases as time goes on, with a peak of 43 principal components. Infact, as more components are kept, the performance of the classifier begins to decrease again, implying that the model may perform worse at a higher complexity. To test this, we will once again plot the ROC and PR curve for this classification with the hyperparameters tuned.

In [ ]:
```
print(f"Best Average Precision Score: {max(average_precision_scores)}", f"Number of PC
pca = PCA(n_components=components_list[np.argmax(average_precision_scores)])
```

```
# Define the grid to optimise over
param_grid = {
    'svm__C': [0.01, 0.1, 1, 10, 100],
    'svm__gamma': ['scale'],
    'svm__degree': [2, 3, 4, 5],
}

# Define the pipeline
pipeline = Pipeline([
    ('scaler', Normalizer()),
    ('pca', pca),
    ('svm', SVC(kernel='poly', probability=True, random_state=42))
])

X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)


# Perform the grid search
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='average_precision', n_
grid_search.fit(X_train_pca, Y_train)

pipeline = grid_search.best_estimator_
_ = plot_cross_validation_curves(pipeline, X_train_pca, Y_train)
```

```
Best Average Precision Score: 0.48820518453209655
Number of PCA Components: 43
```



As shown by the plots above, when using only 43 components, it performs almost as well as the model with all the components. This is a significant reduction in the number of features, which can lead to faster training times and less memory usage. This is an exellenet trade-off for a very small reduction in performance.

Question 5

My approach to this question was to match the training set images to the test set images. Therefore, to do this, I rotates each of the images to match those to the examples from the test set. On top of this, small amounts of noise were added to the images.

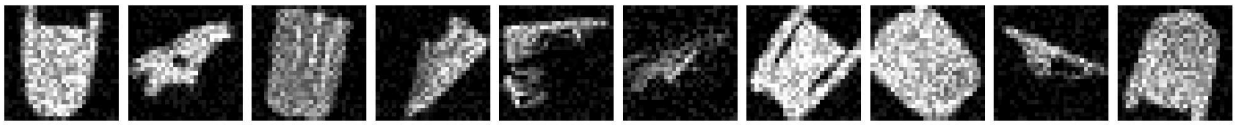
In [ ]:
```
# First import rotate
from scipy.ndimage import rotate
import random
```

In [ ]:
```
def add_padding_rotate_and_crop(flattened_image, pad_size=16, final_size=28):
    # Reshape the flattened image to 28x28
    image_2d = flattened_image.reshape((final_size, final_size))

    # Add padding around the image
    padded_image = np.pad(image_2d, pad_size, mode='constant', constant_values=0)

    # Randomly rotate the image
    angle = random.uniform(1, 360)  # Random angle between 1 and 360
    rotated_image = rotate(padded_image, angle, reshape=False, order=1, mode='reflect'

    # Using a uniform distribution to generate noise - this worked better than Gaussia
    noise_range = (-75, 65)
    noise = np.random.uniform(noise_range[0], noise_range[1], rotated_image.shape)

    # Adaptive noise addition: Modulate noise to be less intense for darker areas and
    # This was meant the training set became more similar to the test set
    adaptive_noise = np.where(rotated_image < 128, noise * 0.5, noise)
    noisy_rotated_image = rotated_image + adaptive_noise

    # Ensure the image values remain in the valid range [0, 255] after noise addition
    noisy_rotated_image = np.clip(noisy_rotated_image, 0, 255)

    # Crop the image back down to the original size by taking the center region
    start = pad_size
    end = start + final_size
    cropped_image = noisy_rotated_image[start:end, start:end]

    # Flatten the image if needed
    noisy_rotated_flattened_image = cropped_image.flatten()

    return noisy_rotated_flattened_image
```

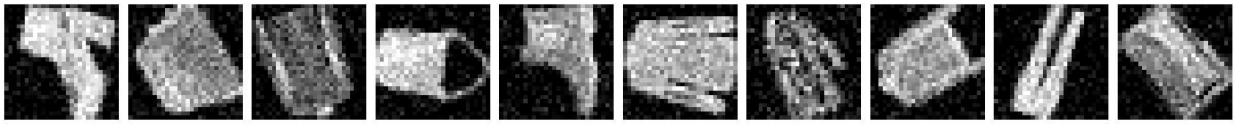
In [ ]:
```
# Now reshape and add noise to the images
XTrain_transformed = np.array([add_padding_rotate_and_crop(image) for image in Xtrain]
```

In [ ]:
```
XTrain_transformed = np.array(XTrain_transformed)
random_indices = np.random.choice(len(XTrain_transformed), size=10, replace=False)
selected_elements = np.array(XTrain_transformed)[random_indices]
fig, axes = plt.subplots(nrows=1, ncols=10, figsize=(20, 5))
for i, ax in enumerate(axes.flatten()):
    # Reshape and plot each image
    ax.matshow(selected_elements[i].reshape(28, 28), cmap='gray')
```

```
    ax.axis('off')  # Turn off axis
plt.tight_layout()
plt.show()
```

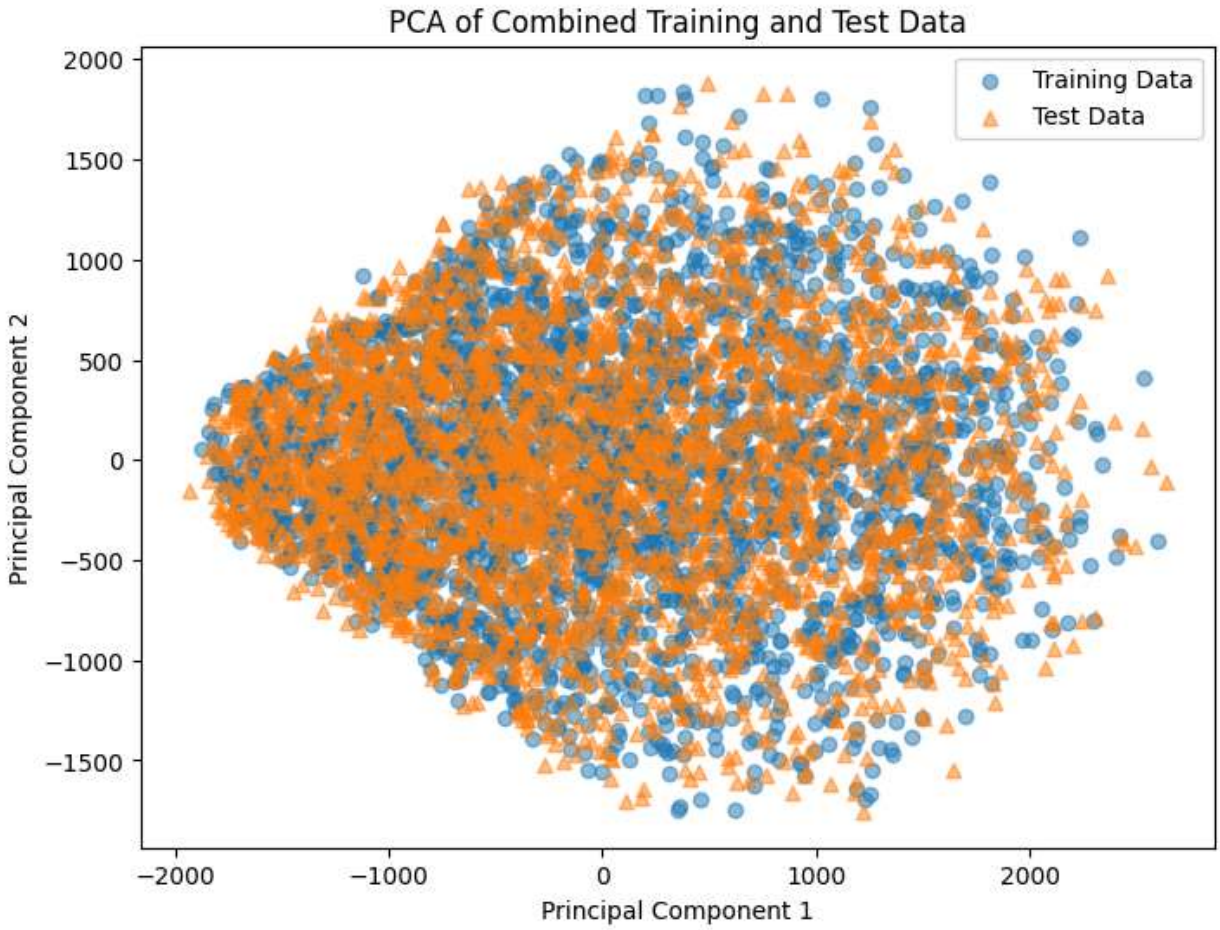


In [ ]:
```
selected_elements = np.array(XTest)[random_indices]

# Plot the 20 random images
fig, axes = plt.subplots(nrows=1, ncols=10, figsize=(20, 5))

for i, ax in enumerate(axes.flatten()):
    # Reshape and plot each image
    ax.matshow(selected_elements[i].reshape(28, 28), cmap='gray')
    ax.axis('off')  # Turn off axis

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



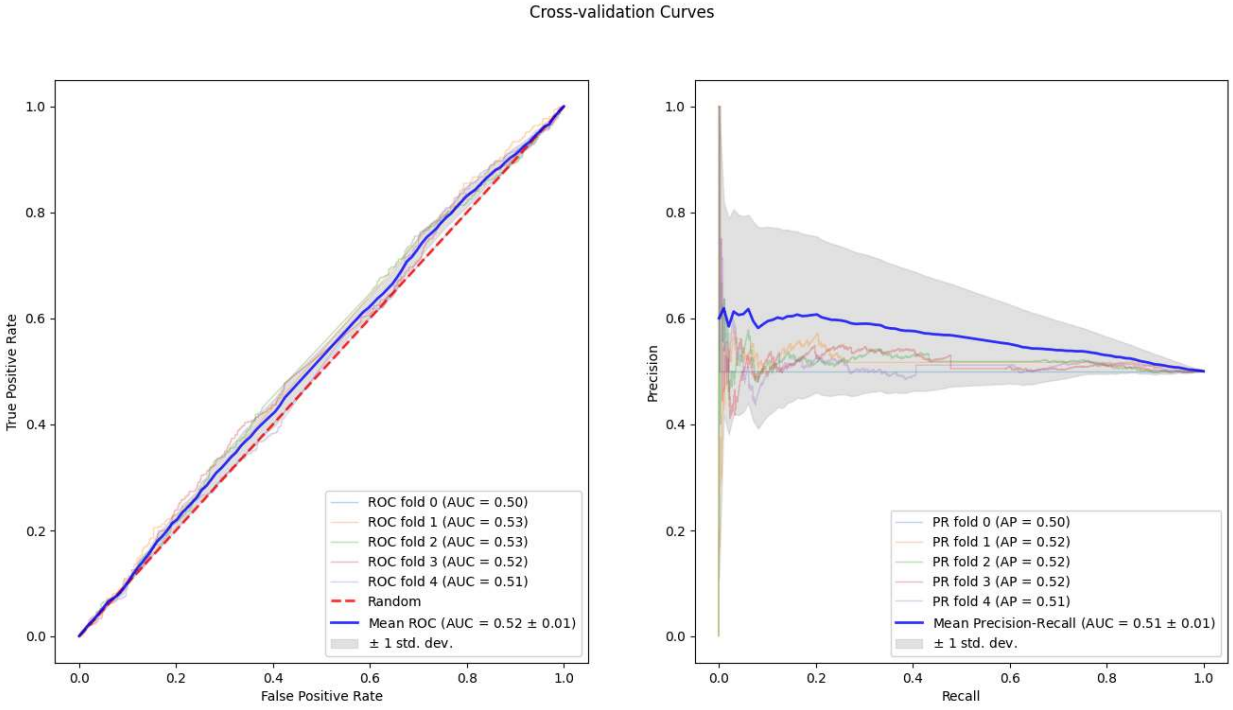
As shown in the image above, the training set seems to resemble the test set much more accurately, therefore, we may train ouyr classifier on that. We are able to take two possible options, re-run our grid searches as done in the earlier questions, or alternitavely use the found hyperparameters to train the classifier again.

In [ ]:
```
# First run PCA to see how the dataset looks in 2d
# Combining training and test data


# Applying PCA to reduce dimensions to 2
pca_combined = PCA(n_components=2)

# Separating the reduced data back into training and test sets
Xtrain_reduced = pca_combined.fit_transform(XTrain_transformed)
Xtest_reduced = pca_combined.fit_transform(XTest)

# Plotting the scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(Xtrain_reduced[:, 0], Xtrain_reduced[:, 1], label="Training Data", alpha=0

plt.scatter(Xtest_reduced[:, 0], Xtest_reduced[:, 1], marker='^', label="Test Data", a

plt.title("PCA of Combined Training and Test Data")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend()
plt.show()
```



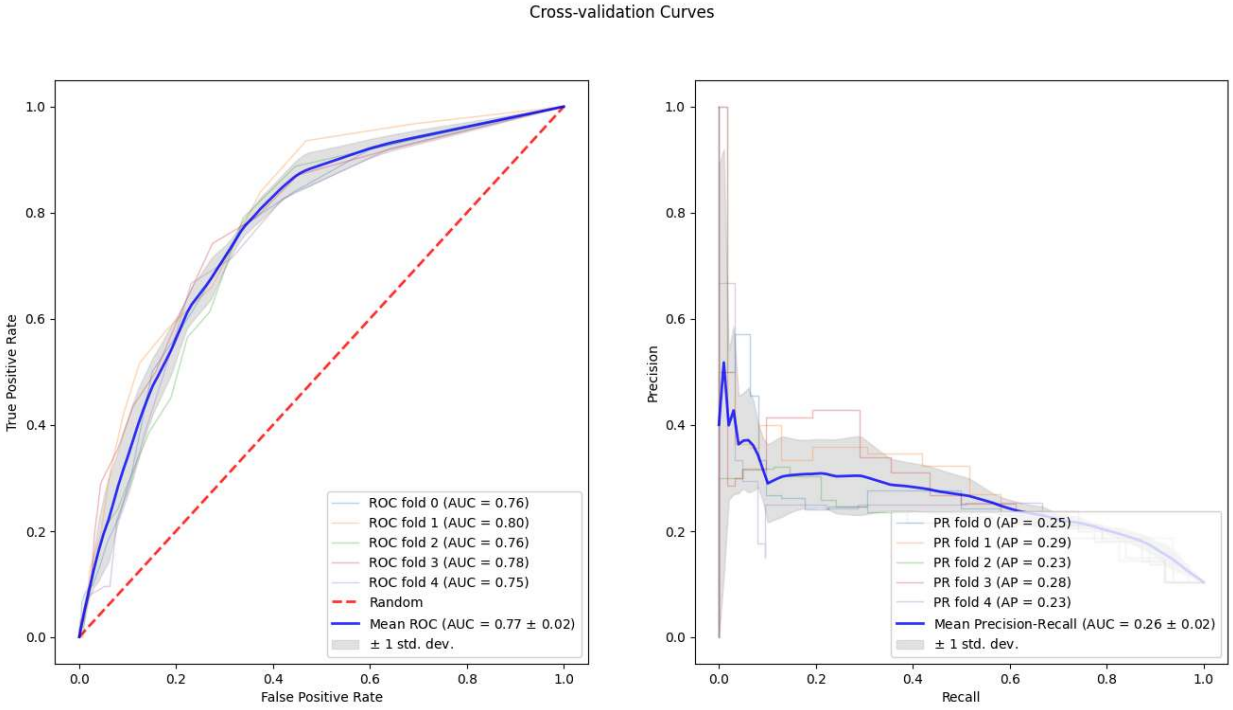
To check that our datasets are really that similar, we will test the performance of a classifier on distinguishing between the test and the train data.

In [ ]:
```
# Combine the training and test datasets
pca_10 = PCA(n_components=10)

XTrain_transformed_reduced = pca_10.fit_transform(XTrain_transformed)
XTest_reduced = pca_10.transform(XTest)

X_combined = np.vstack((XTrain_transformed_reduced, XTest_reduced))
Y_combined = np.array([-1]*len(Xtrain) + [1]*len(XTest))  # -1 for training data, +1 f

# Define our pipeline - random forest
pipeline = Pipeline([
    ('scaler', Normalizer()),
    ('svc', SVC(gamma='scale', kernel='poly', probability=True, random_state=42))
])

# Perform cross-validation and plot the curves
_ = plot_cross_validation_curves(pipeline, X_combined, Y_combined, n_folds=5)
```



As we can see by the plot above, the data sseems to be much more similar now than previously when compared with the top 10 principal components (the classifier struggles very strongly to differentiate).

In [ ]:
```
# Best estimator for Random Forest
best_estimator_rf = Pipeline([('scaler', Normalizer()), ('rf', RandomForestClassifier(
q5_estimator_fit = best_estimator_rf.fit(XTrain_transformed_reduced, Ytrain)

q5_estimator = plot_cross_validation_curves(best_estimator_rf, XTrain_transformed_redu
```



Please note, that the number of principle components used here is only 10, since any higher and it seemed to have a much stronger bias towards predicting the positive class. So much so that in

many cases, all 3000 test examples were output to be negative, which could imply the model had been over fit too significantly.

In [ ]:
```
Xtest_reduced = pca_10.transform(XTest)
pd.DataFrame(q5_estimator_fit.predict_proba(Xtest_reduced)[:,1]).to_csv('2039323.csv',

# Get the number of positive predictions
print(len([i for i in q5_estimator_fit.predict(Xtest_reduced) if i == 1]))
```

```
54
```

Please note the prediction scores are in the 2039323.csv file,

Question 6

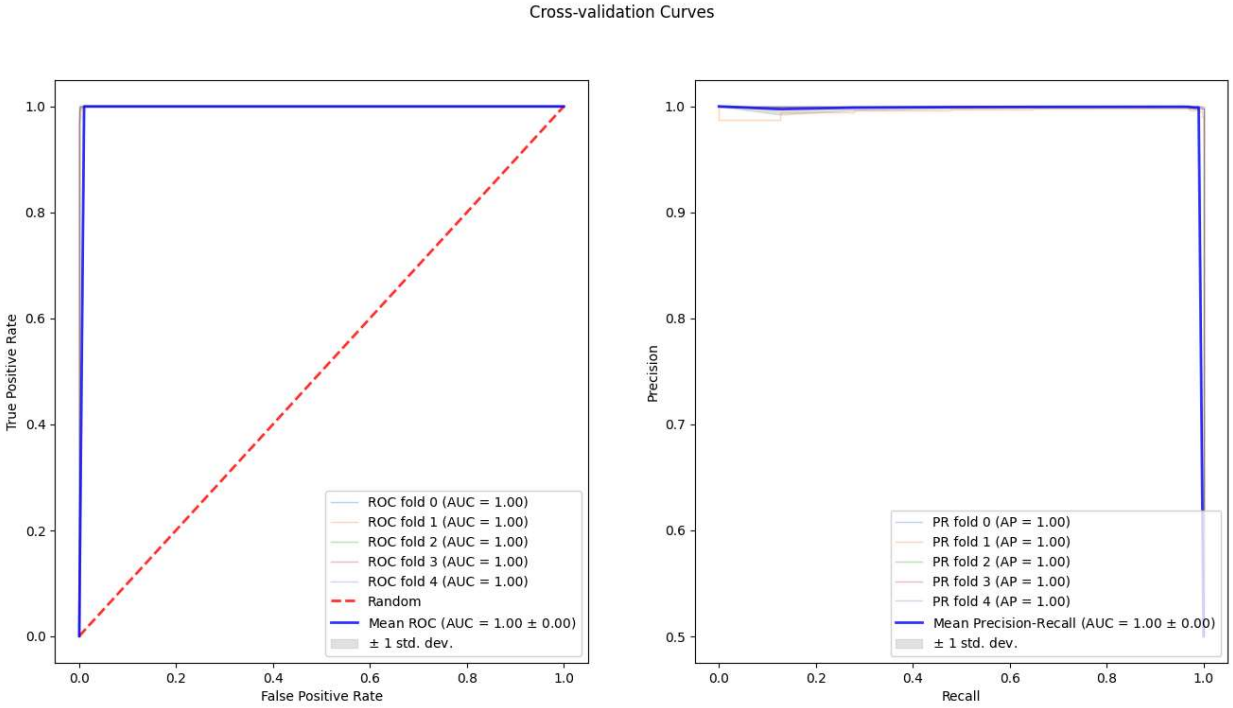
Introducing a new classification problem, where the original training set are in the positive class and the original test set are in the test class.

- Xtrain has the label -1
- Xtest has the label + 1

In [ ]:
```
# Combine the training and test datasets
X_combined = np.vstack((Xtrain, XTest))
Y_combined = np.array([-1]*len(Xtrain) + [1]*len(XTest))  # -1 for training data, +1 f

classifier = RandomForestClassifier(n_jobs=-1, random_state=42)

skf = StratifiedKFold(n_splits=5)

auc_roc_scores = []

for train_index, test_index in skf.split(X_combined, Y_combined):
    X_train_fold, X_test_fold = X_combined[train_index], X_combined[test_index]
    Y_train_fold, Y_test_fold = Y_combined[train_index], Y_combined[test_index]

    classifier.fit(X_train_fold, Y_train_fold)

    probs = classifier.predict_proba(X_test_fold)[:, 1]

    auc_roc = roc_auc_score(Y_test_fold, probs)
    auc_roc_scores.append(auc_roc)

avg_auc_roc = np.mean(auc_roc_scores)
std_auc_roc = np.std(auc_roc_scores)

print(f"Average AUC-ROC: {avg_auc_roc:.3f}")
print(f"Standard Deviation of AUC-ROC: {std_auc_roc:.3f}")
```

```
Average AUC-ROC: 1.000
Standard Deviation of AUC-ROC: 0.001
```

In [ ]:
```
_ = plot_cross_validation_curves(classifier, X_combined, Y_combined)
```



- i) The high value of the ROC value shows that the classifier can easily distinguish between the training and the test data. This indicates strongly the differences in the way the training an test sets were created and is understangable given the rotation of the images in the test set as well as the added noise when comparing the two. Additionally, the perfect scores on both the ROC curves and the PR curves indicate that the model was not overfit to the training data, and was clearly able to reproduce it's performance when tested. Additionally, this could indicate when performing the cross validation, that the test set in each fold follows the distribution of the test data exactly. Moreover, the standard deviation of the AUC-ROC was 0.001 (as shown in the cell before the plots), meaning that across all of the folds the classifier performed excellently.

- ii) This experiment can be used to elinate some of the systematic differences between the training and test sets in multiple ways. This could be done by applying different transformations to the original training set and re-running the experiment. For example, firstly adding some noise to the images. Then once the experiment is re-run, identify how much that changed the performance of the classifier when attempting to determine between the training and the test set. After this, images could be rotated at varying degrees of randomness and varying angles to idetify how well the classifier at identifying the differences within the dataset at each of these steps. From the reductions in the performance of the classifiers in the experiment, this allows you to identify and eliminate the systematic differences between the two sets.