MTU

Ollscoil Teicneolaíochta na Mumhan

Munster Technological University

Supervised Learning

Jesús Fernández López

4º Software Development

MACHINE LEARNING

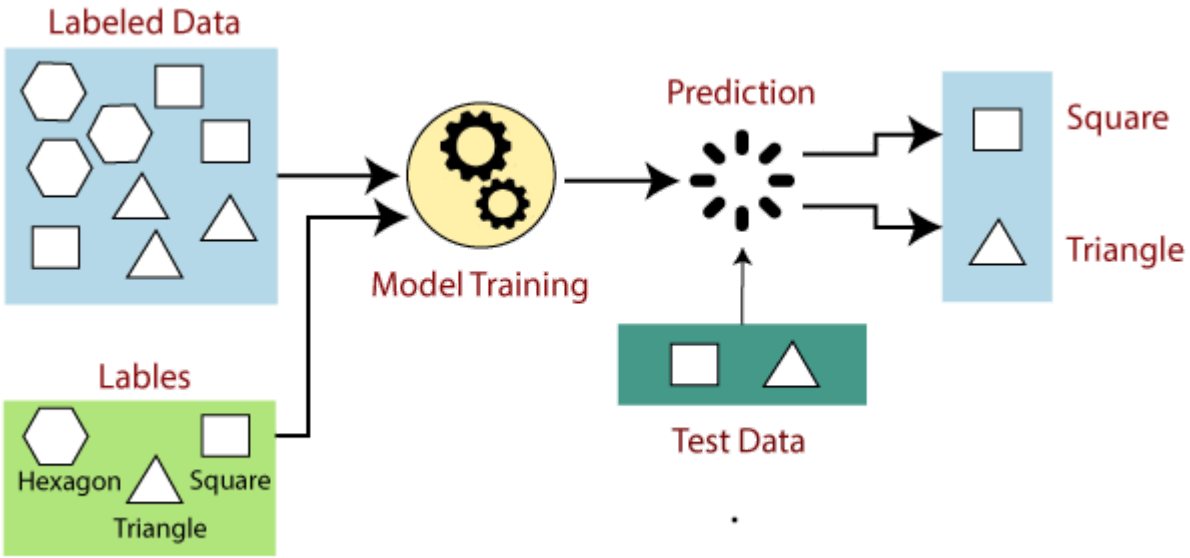
·Index:

·Introduction:

Supervised Learning is a foundational concept in machine learning, characterized by its use of labeled data to train predictive models. In this approach, the model is presented with a dataset where each input (or feature) has an associated known output (or label). Through multiple iterations, the model learns to map inputs to their corresponding outputs, which allows it to generalize and make accurate predictions on new data. The primary goal is to minimize the difference between the predicted and actual outputs, often by adjusting internal parameters using an optimization algorithm.

Supervised learning tasks are typically divided into two main categories: classification and regression. In classification, the aim is to assign data points to one of several predefined categories or classes. For example, a model trained to classify emails as "spam" or "not spam" is a binary classifier, while one trained on images of animals could categorize them as "dog," "cat," or "bird," making it a multi-class classifier. Regression, on the other hand, is used for tasks where the output is continuous, such as predicting house prices based on features like location, size, and age.

In our assignment, we focus on a classification task using the Fashion-MNIST dataset. This dataset contains grayscale images of clothing and footwear, represented as 28x28 pixel grids, along with a label for each item type (e.g., T-shirt, dress, sneaker). Our objective is to classify specific categories by evaluating the performance of various classifiers, such as decision trees, support vector machines, and k-nearest neighbors. By comparing these classifiers, we aim to identify the model that can best distinguish between these categories based on prediction accuracy and processing efficiency.



·Task 1: Pre-processing and visualization

Load all sandals, sneakers, and ankle boots from the dataset and separate the labels from the feature vectors. Display one image for each class.

For Task 1, I began by loading the Fashion-MNIST dataset. This dataset contains grayscale images, each represented as a series of 784 pixel values (28x28) along with a label indicating the type of item. Since my focus is on classifying only sandals, sneakers, and ankle boots, I filtered the dataset to include just these three categories, corresponding to labels 5, 7, and 9 (lines 6 - 10).

Next, I separated the labels from the image data itself. The labels provide the categories (sandal, sneaker, and ankle boot), while the rest of the data contains the pixel values for each image. By isolating these two parts, I made it easier to work with the dataset when training classifiers and visualizing individual images (lines 13 - 14).

To confirm that the data is processed correctly, I visualized one example from each class. For this, I took the pixel values of a sample from each category, reshaped it from a flat list of 784 values into a 28x28 grid, and displayed it as a grayscale image. This helped me verify that the images matched the expected categories, showing a clear representation of a sandal, a sneaker, and an ankle boot (lines 17 - 28).

```
6.  # Step 1: Load the dataset
7.  data = pd.read_csv('fashion-mnist_train.csv')
8.
9.  # Step 2: Filter to keep only sandals, sneakers, and ankle boots
10. filtered_data = data[data['label'].isin([5, 7, 9])]
11.
12. # Step 3: Separate labels and features
13. labels = filtered_data['label'].values
14. features = filtered_data.drop('label', axis=1).values
15.
16. # Step 4: Display one example for each class (5 - Sandal, 7 - Sneaker, 9 -
    Ankle Boot)
17. sample_classes = {5: "Sandal", 7: "Sneaker", 9: "Ankle Boot"}
18.
19. verbose = False # Controls the print of the 3 images
20. if verbose:
21.     for label, class_name in sample_classes.items():
22.         # Select one sample for the current label
23.         sample_image = features[labels == label][0].reshape(28, 28)
24.
25.         # Plot the image
26.         plt.imshow(sample_image, cmap='gray')
27.         plt.title(f"{class_name} Example")
28.         plt.axis('off')
29.         plt.show()
```

·Task 2: Evaluation procedure

Create a k-fold cross-validation procedure to split the data into training and evaluation subsets. Parameterise the number of samples to use from the dataset to be able to control the runtime of the algorithm evaluation. Start developing using a small number of samples and increase for the final evaluation. Make the function flexible to accommodate different types of classifiers as required in tasks 3-6. Measure for each split of the cross-validation procedure the processing time required for training, the processing time required for prediction and determine the confusion matrix and accuracy score of the classification. Calculate the minimum, the maximum, and the average of
- the training time per training sample
- the prediction time per evaluation sample
- and the prediction accuracy

For Task 2, I started by setting up a cross-validation procedure to evaluate different classifiers on this dataset. Cross-validation is essential because it allows me to assess how well each model generalizes by dividing the dataset into multiple subsets, training on some and validating on others, and repeating this process multiple times. This way, I can obtain an average performance metric that is more resilient to variations in the training data (lines 3 - 42).

First, I set up the process to split the data into training and evaluation subsets. To control runtime, I included a parameter to adjust the number of samples used, starting with a small data subset to speed up initial testing and increasing the sample size for final evaluation. This approach helps reduce computation time early on, allowing for faster iterations, and then maximizes accuracy in the final stage (lines 45 - 46)

To measure each classifier's performance, I recorded both training and prediction times: one for the time it takes to train and the other for prediction. In each fold of cross-validation, I also generated `(if verbose)` a confusion matrix and a prediction accuracy score, which shows how well the model is classifying each category. After each iteration, I calculated the minimum, maximum, and average for training time per sample, prediction time per sample, and prediction accuracy (lines 48 - 82)

Here's the code that implements this evaluation procedure, allowing me to apply this setup to any classifier I'll be using in the subsequent tasks:

```
39. def cross_validation_evaluation(model, features, labels, k=5,
    sample_size=None):
40.     verbose = False # Controls the print of the confusion matrix
41.     kf = KFold(n_splits=k, shuffle=True, random_state=42)
42.     train_times, predict_times, accuracies = [], [], []
43.
44.     # Limit the sample size if needed
45.     if sample_size:
```

```
        features, labels = features[:sample_size], labels[:sample_size]

    for train_index, test_index in kf.split(features):
        model = clone(model)  #Reset del modelo
        X_train, X_test = features[train_index], features[test_index]
        y_train, y_test = labels[train_index], labels[test_index]

        # Measure training time
        start_train = time.time()
        model.fit(X_train, y_train)
        end_train = time.time()

        # Measure prediction time
        start_predict = time.time()
        predictions = model.predict(X_test)
        end_predict = time.time()

        # Calculate metrics
        train_time = end_train - start_train
        predict_time = end_predict - start_predict
        accuracy = accuracy_score(y_test, predictions)

        train_times.append(train_time
        predict_times.append(predict_time
        accuracies.append(accuracy)

    # Print confusion matrix for further inspection
        if verbose:
            print("Confusion Matrix for this fold:")
            print(confusion_matrix(y_test, predictions))
    if verbose:
             print(f"Training Time per Sample - Min: {min(train_times)}, Max: {max(train_times)}, Avg: {np.mean(train_times)}")
          print(f"Prediction Time per Sample - Min: {min(predict_times)}, Max: {max(predict_times)}, Avg: {np.mean(predict_times)}")
                print(f"Prediction Accuracy - Min: {min(accuracies)}, Max: {max(accuracies)}, Avg: {np.mean(accuracies)}")

    # Return average values for training and prediction times
    return np.mean(train_times), np.mean(predict_times), np.mean(accuracies)
```

This setup allows me to evaluate any classifier consistently. For each iteration, I obtain training and prediction times, a confusion matrix, and an accuracy score, which makes comparing different models straightforward and provides a detailed breakdown of each classifier's performance.

·Task 3: Perceptron

Use the procedure developed in task 2 to train and evaluate the Perceptron classifier. What is the mean prediction accuracy of this classifier? Vary the number of samples and plot the relationship between input data size and runtimes for the classifier.

For Task 3, I focused on using the cross-validation procedure to evaluate the performance of a Perceptron classifier on the Fashion-MNIST dataset. The Perceptron is a linear model that tries to find a hyperplane separating the data into classes. It's generally effective for binary or linearly separable data, but here I'm testing how well it handles a more complex, multi-class problem.

I used cross-validation to get a reliable measure of the Perceptron's performance across different data splits. For each fold, I recorded the training and prediction times to evaluate the efficiency of the model, and I also captured the accuracy scores, which reflect how accurately the model predicts the correct class for each test sample (lines 101- 111).

And the calculated mean Prediction Accuracy for Perceptron is 0.8331.

To better understand how the Perceptron performs with different amounts of data, I varied the sample sizes used in training. Starting with small samples allowed me to test the setup quickly, while larger samples provided a more comprehensive evaluation. By averaging the training and prediction times for each sample size, I could see how the model's runtime scales as the amount of data increases.

I also visualized this relationship by plotting sample size against runtime. This plot shows how training and prediction times increase with more data, offering insights into the Perceptron's computational efficiency as the data grows. This analysis gives me a good foundation for comparing the Perceptron to other classifiers in the next tasks, as I can now assess its trade-off between accuracy and runtime (lines 114 - 120).

```
97. def evaluate_perceptron(features, labels, sample_sizes=[100, 500, 1000, 5000]):
98.     perceptron_model = Perceptron()
99.     training_times, prediction_times, accuracies = [], [], []
100.
101.    for size in sample_sizes:
102.        print(f"Evaluating for sample size: {size}")
103.                        avg_train_time,    avg_predict_time,    avg_accuracy    =
    cross_validation_evaluation(
104.                model=perceptron_model, features=features, labels=labels, k=5,
    sample_size=size)
105.
106.        training_times.append(avg_train_time)
107.        prediction_times.append(avg_predict_time)
108.        accuracies.append(avg_accuracy)
109.
110.    mean_accuracy = np.mean(accuracies)
111.    print(f"Mean Prediction Accuracy for Perceptron: {mean_accuracy:.4f}")
```

```
112.
113.     # Plot the effect of sample size on runtime
114.     plt.plot(sample_sizes, training_times, label="Training Time")
115.     plt.plot(sample_sizes, prediction_times, label="Prediction Time")
116.     plt.xlabel("Sample Size")
117.     plt.ylabel("Time (seconds)")
118.     plt.title("Perceptron Runtime vs Sample Size")
119.     plt.legend()
120.     plt.show()
```

Task 4: Decision trees

Use the procedure developed in task 2 to train and evaluate the Decision tree classifier. What is the mean prediction accuracy of this classifier? Vary the number of samples and plot the relationship between input data size and runtimes for the classifier

For Task 4, I used the same cross-validation procedure to train and evaluate a Decision Tree classifier on the Fashion-MNIST dataset. Decision Trees are a non-linear model, making them potentially more flexible than the Perceptron, as they can capture complex decision boundaries within the data. This makes Decision Trees particularly useful for datasets with more intricate patterns, like Fashion-MNIST.

I began by applying the cross-validation function to the Decision Tree model. For each fold, I measured the accuracy of the predictions, as well as the training and prediction times. Tracking these metrics across different folds allowed me to calculate averages, providing a reliable view of the Decision Tree's performance on this specific task (lines 133 - 136).

As in Task 3, I varied the sample sizes used to train the model. Starting with smaller sample sizes helped speed up the process, while larger samples allowed for a more thorough evaluation. By plotting the sample sizes against the training and prediction times, I could observe how the Decision Tree's runtime scales with data size, which is helpful for comparing computational efficiency to that of other models (lines 142 - 148).

```
128. def evaluate_decision_tree(features, labels, sample_sizes=[100, 500, 1000,
     5000]):
129.     # Initialize the Decision Tree model
130.     decision_tree_model = DecisionTreeClassifier()
131.     training_times, prediction_times, accuracies = [], [], []
132.
133.     for size in sample_sizes:
134.         print(f"Evaluating for sample size: {size}")
135.                             avg_train_time,    avg_predict_time,    avg_accuracy    =
     cross_validation_evaluation(
136.                 model=decision_tree_model, features=features, labels=labels, k=5,
     sample_size=size)
137.
138.         training_times.append(avg_train_time)
139.         prediction_times.append(avg_predict_time)
140.         accuracies.append(avg_accuracy)
141.
142.     mean_accuracy = np.mean(accuracies)
143.     print(f"Mean Prediction Accuracy for Perceptron: {mean_accuracy:.4f}")
144.
145.     # Plotting the effect of sample size on runtime
146.     plt.plot(sample_sizes, training_times, label="Training Time")
147.     plt.plot(sample_sizes, prediction_times, label="Prediction Time")
148.     plt.xlabel("Sample Size")
149.     plt.ylabel("Time (seconds)")
150.     plt.title("Decision Tree Runtime vs Sample Size")
151.     plt.legend()
152.     plt.show()
```

·Task 5: k-nearest Neighbors

Use the procedure developed in task 2 to train and evaluate the k-nearest neighbor classifier. Try different choices for the parameter k and determine a good value based on mean prediction accuracy. What is the best achievable mean prediction accuracy of this classifier? Vary the number of samples and plot the relationship between input data size and runtimes for the optimal classifier.

For Task 5, I applied the cross-validation procedure to evaluate a k-Nearest Neighbors (k-NN) classifier on the Fashion-MNIST dataset. The k-NN classifier is an instance-based method, meaning it doesn't build a specific model during training; instead, it makes predictions by comparing a sample to its nearest neighbors in the training set. This makes k-NN straightforward but computationally intensive as the dataset grows.

To find the optimal number of neighbors (the parameter *k*), I experimented with several values, measuring the accuracy for each. A low *k* value could lead to more sensitivity to noise, while a higher *k* value provides smoother decision boundaries but might underfit. By comparing the accuracy across different *k* values, I could select the one that achieved the highest mean accuracy (lines 161 - 163).

For each *k*, I also tracked training and prediction times across various sample sizes, similar to the previous tasks. While k-NN doesn't involve a typical training process (since it only stores the data), prediction times can increase significantly with larger sample sizes because it calculates distances between the test samples and all training samples (lines 165 - 204).

Once this function completes, the output will provide the best value for *k*, which is the one that maximizes mean prediction accuracy. The best achievable mean prediction accuracy for this *k*-NN classifier is the accuracy corresponding to this optimal *k* value, printed as best_accuracy. This is the highest average accuracy obtained across all tested sample sizes (line 206).

```
161. def evaluate_knn(features, labels, k_values=[1, 3, 5, 7], sample_sizes=[100,
   500, 1000, 5000]):
162.     best_k = None
163.     best_accuracy = 0  # Initialize the best accuracy as zero
164.
165.     for k in k_values:
166.         print(f"Evaluating k-NN with k={k}")
167.         knn_model = KNeighborsClassifier(n_neighbors=k)
168.         accuracies = []
169.         training_times, prediction_times = [], []
170.
171.         for size in sample_sizes:
172.             print(f"Sample size: {size}")
173.                                 avg_train_time,  avg_predict_time,  avg_accuracy  =
   cross_validation_evaluation(
```

```
                        model=knn_model, features=features, labels=labels, k=5,
    sample_size=size)

            training_times.append(avg_train_time)
            prediction_times.append(avg_predict_time)
            accuracies.append(avg_accuracy)

        # Calculate mean accuracy across all sample sizes for current k
        mean_accuracy_k = np.mean(accuracies)
        print(f"Mean accuracy for k={k}: {mean_accuracy_k}")

        # Update best k if this k's accuracy is higher
        if mean_accuracy_k > best_accuracy:
            best_accuracy = mean_accuracy_k
            best_k = k

        # Plot accuracy and runtime for each k
        plt.figure()
        plt.plot(sample_sizes, accuracies, label="Accuracy")
        plt.xlabel("Sample Size")
        plt.ylabel("Accuracy")
        plt.title(f"k-NN Accuracy vs Sample Size (k={k})")
        plt.legend()
        plt.show()

        plt.figure()
        plt.plot(sample_sizes, prediction_times, label="Prediction Time")
        plt.xlabel("Sample Size")
        plt.ylabel("Time (seconds)")
        plt.title(f"k-NN Prediction Time vs Sample Size (k={k})")
        plt.legend()
        plt.show()

    print(f"The best value for k is {best_k} with a mean prediction accuracy of
    {best_accuracy:.4f}")
```

·Task 6: Support Vector Machine

Use the procedure developed in task 2 to train and evaluate the Support Vector Machine classifier. Use a radial basis function kernel and try different choices for the parameter γ. Determine a good value for γ based on mean prediction accuracy. What is the best achievable mean prediction accuracy of this classifier? Vary the number of samples and plot the relationship between input data size and runtimes for the optimal classifier.

For Task 6, I used a Support Vector Machine (SVM) classifier with an RBF kernel to classify images from the Fashion-MNIST dataset. SVMs with RBF kernels can handle complex, non-linear relationships, making them well-suited for image classification tasks. However, they are sensitive to data scale, so I began by normalizing the pixel values in the dataset, which originally ranged from 0 to 255, to a range of 0 to 1. This normalization ensures that all features contribute equally to the model, significantly improving the classifier's effectiveness (lines 217 - 218).

To optimize the SVM, I tested different values of the parameter γ, which controls the influence of individual training points. Higher values of γ allow the SVM to create a tighter decision boundary, focusing more on each training point but potentially leading to overfitting. Lower values of γ encourage smoother decision boundaries, which generalize better but may underfit. The goal was to find the γ value that yields the best mean prediction accuracy, balancing model complexity and generalization.

For each γ, I ran cross-validation on different sample sizes to evaluate both the accuracy and runtime efficiency of the SVM. For each combination of γ and sample size, I recorded training and prediction times to observe how the SVM scales with larger datasets. The function cross_validation_evaluation calculated the average training time, prediction time, and accuracy across all cross-validation folds, and I stored these results to assess the performance of each configuration (lines 225 - 241)

After testing all specified γ values, I compared their mean accuracies to identify the best γ. This value, which produced the highest mean accuracy, represents the best achievable prediction accuracy for this SVM configuration on the dataset. The function then outputs both the best γ value and the corresponding mean prediction accuracy, allowing me to confidently conclude which configuration performs best in terms of accuracy and runtime. Finally I plot and visualize the results (lines 244 - 265)

```
213. from sklearn.svm import SVC
214. from sklearn.preprocessing import MinMaxScaler
215.
216. # Normalize features
217. scaler = MinMaxScaler()
218. features_normalized = scaler.fit_transform(features)
```

```
def evaluate_svm(features, labels, gamma_values=[0.01, 0.1, 1], sample_sizes=[100, 500, 1000, 5000]):
    best_gamma = None
    best_accuracy = 0

    for gamma in gamma_values:
        print(f"Evaluating SVM with gamma={gamma}")
        svm_model = SVC(kernel='rbf', gamma=gamma)
        accuracies, training_times, prediction_times = [], [], []

        for size in sample_sizes:
            print(f"Sample size: {size}")
                            avg_train_time, avg_predict_time, avg_accuracy = cross_validation_evaluation(
                        model=svm_model, features=features, labels=labels, k=5, sample_size=size)

            training_times.append(avg_train_time)
            prediction_times.append(avg_predict_time)
            accuracies.append(avg_accuracy)

        # Calculate mean accuracy for the current gamma
        mean_accuracy_gamma = np.mean(accuracies)
        print(f"Mean accuracy for gamma={gamma}: {mean_accuracy_gamma}")

        # Update best gamma if this gamma's accuracy is higher
        if mean_accuracy_gamma > best_accuracy:
            best_accuracy = mean_accuracy_gamma
            best_gamma = gamma

        # Plot accuracy and runtime for each gamma
        plt.figure()
        plt.plot(sample_sizes, accuracies, label="Accuracy")
        plt.xlabel("Sample Size")
        plt.ylabel("Accuracy")
        plt.title(f"SVM Accuracy vs Sample Size (gamma={gamma})")
        plt.legend()
        plt.show()

        plt.figure()
        plt.plot(sample_sizes, prediction_times, label="Prediction Time")
        plt.xlabel("Sample Size")
        plt.ylabel("Time (seconds)")
        plt.title(f"SVM Prediction Time vs Sample Size (gamma={gamma})")
        plt.legend()
        plt.show()

    print(f"The best value for gamma is {best_gamma} with a mean prediction accuracy of {best_accuracy:.4f}")


# Example usage with normalized features
evaluate_svm(features_normalized, labels)
```

·Task 7: Comparison

Compare the training and prediction times of the four classifiers. What trend do you observe for each of the classifiers and why? Also taking the accuracy into consideration, how would you rank the four classifiers and why?

To compare various classifiers, I implemented an evaluate_all_classifiers function that provides insights into each classifier's accuracy, prediction time, and training time. This function evaluates the following classifiers: Perceptron, Decision Tree, k-Nearest Neighbors (k-NN), and Support Vector Machine (SVM).

The function performs 5-fold cross-validation for each classifier and captures the following metrics:

- **Mean Accuracy**: The average accuracy achieved by the classifier.
- **Mean Prediction Time**: The average time it takes for the classifier to predict labels for a sample.
- **Mean Training Time**: The average time required to train the classifier per sample.

Each metric helps analyze the trade-offs between accuracy and computational efficiency.

The function starts by defining the classifiers in a dictionary, classifiers, where I specify the Perceptron, Decision Tree, k-Nearest Neighbors (k-NN with k=4), and Support Vector Machine (SVM with γ=0.03). The results dictionary stores each classifier's mean accuracy, prediction time, and training time for easy access during visualization (lines 272 - 278).

For each classifier, I use the cross_validation_evaluation function to compute the average training time, prediction time, and accuracy across 5 folds, with a sample size specified as an argument (len(features)). Each result is stored in the results dictionary under the relevant classifier (lines 282 - 291).

The visualization is designed to plot the three metrics side-by-side. First, I set up a bar chart for mean accuracy on the primary y-axis (ax1), with the blue color indicating accuracy. Next, I overlay line charts on a secondary y-axis (ax2) for mean prediction time (in red) and mean training time (in green). This dual-axis approach allows me to compare accuracy and both time metrics clearly on a single plot.

To improve readability, I've adjusted the figure size and margins, using fig.subplots_adjust to prevent any text from being cut off. The legend is placed at the top left for accuracy (primary axis) and top right for prediction and training times (secondary axis), making it easy to differentiate between the metrics. The title summarizes the purpose of the chart, showing all three metrics for each classifier (lines 295 - 319).

This updated function provides a complete visual and numerical comparison of the classifiers, helping to identify which model balances accuracy, prediction time, and training time best for the Fashion-MNIST dataset.

```
def evaluate_all_classifiers(features, labels, sample_size= None):
    classifiers = {
        "Perceptron": Perceptron(),
        "Decision Tree": DecisionTreeClassifier(),
        "k-NN": KNeighborsClassifier(n_neighbors=4),
        "SVM": SVC(kernel='rbf', gamma=0.03)
    }

     results = {"Classifier": [], "Mean Accuracy": [], "Mean Prediction Time": [], "Mean Training Time": []}

    for name, model in classifiers.items():
        print(f"Evaluating {name}...")

                           avg_train_time,    avg_predict_time,    avg_accuracy    = cross_validation_evaluation(
                            model=model,  features=features,  labels=labels,  k=5, sample_size=sample_size)

        results["Classifier"].append(name)
        results["Mean Accuracy"].append(avg_accuracy)
        results["Mean Prediction Time"].append(avg_predict_time)
        results["Mean Training Time"].append(avg_train_time)

           print(f"{name} - Mean Accuracy: {avg_accuracy:.4f}, Mean Prediction Time: {avg_predict_time:.4f} seconds, Mean Training Time: {avg_train_time:.4f} seconds")

     fig, ax1 = plt.subplots(figsize=(10, 6))  # Tamaño ajustado para mejorar la visibilidad

    # Gráfico de barras para Mean Accuracy
         ax1.bar(results["Classifier"],  results["Mean  Accuracy"],  color='b', alpha=0.6, label='Mean Accuracy')
    ax1.set_xlabel('Classifier')
    ax1.set_ylabel('Mean Accuracy', color='b')
    ax1.tick_params(axis='y', labelcolor='b')

    # Gráfico de línea para Mean Prediction Time y Mean Training Time
    ax2 = ax1.twinx()
     ax2.plot(results["Classifier"], results["Mean Prediction Time"], color='r', marker='o', label='Mean Prediction Time')
      ax2.plot(results["Classifier"], results["Mean Training Time"], color='g', marker='x', label='Mean Training Time')
    ax2.set_ylabel('Time (seconds)', color='g')
    ax2.tick_params(axis='y', labelcolor='g')

    # Ajuste de márgenes y leyenda
     fig.subplots_adjust(left=0.15, right=0.85)  # Expande márgenes para evitar recortes
       fig.tight_layout(pad=2.0)    # Ajuste adicional para espaciar bien los elementos

    # Añadir leyendas para cada eje
    ax1.legend(loc="upper left")
```

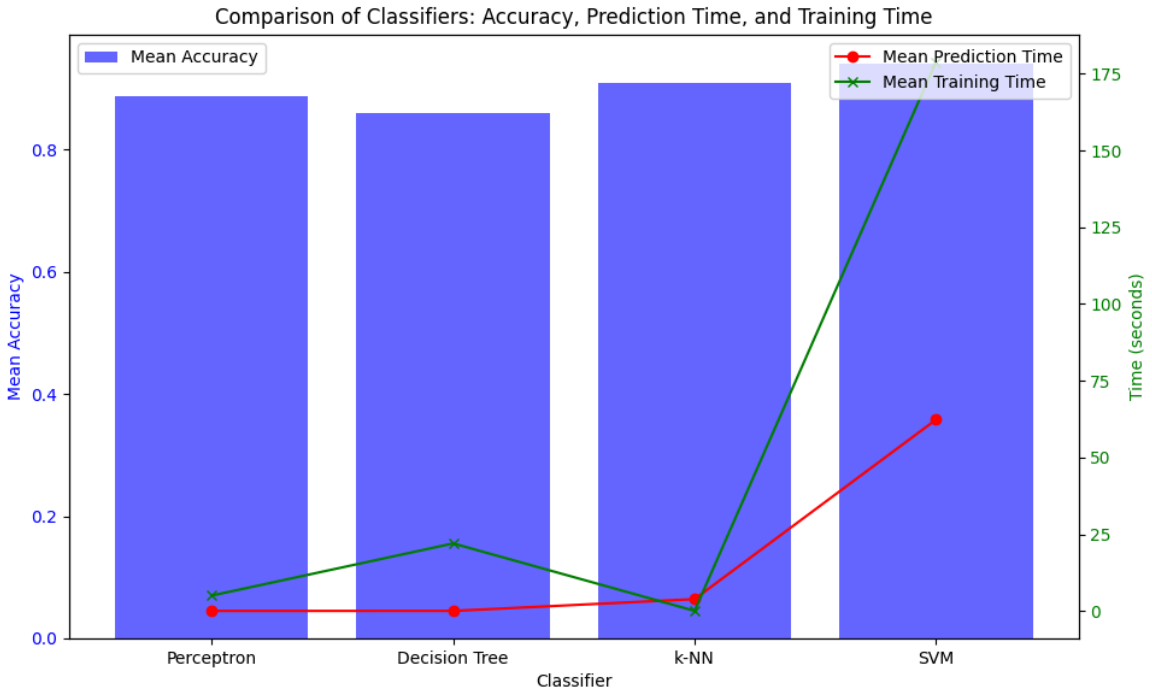
```
316.     ax2.legend(loc="upper right")
317.
318.         plt.title("Comparison of Classifiers: Accuracy, Prediction Time, and
    Training Time")
319.     plt.show()
320.
321.     return results
```

Output of the code: sample_size = len(features)

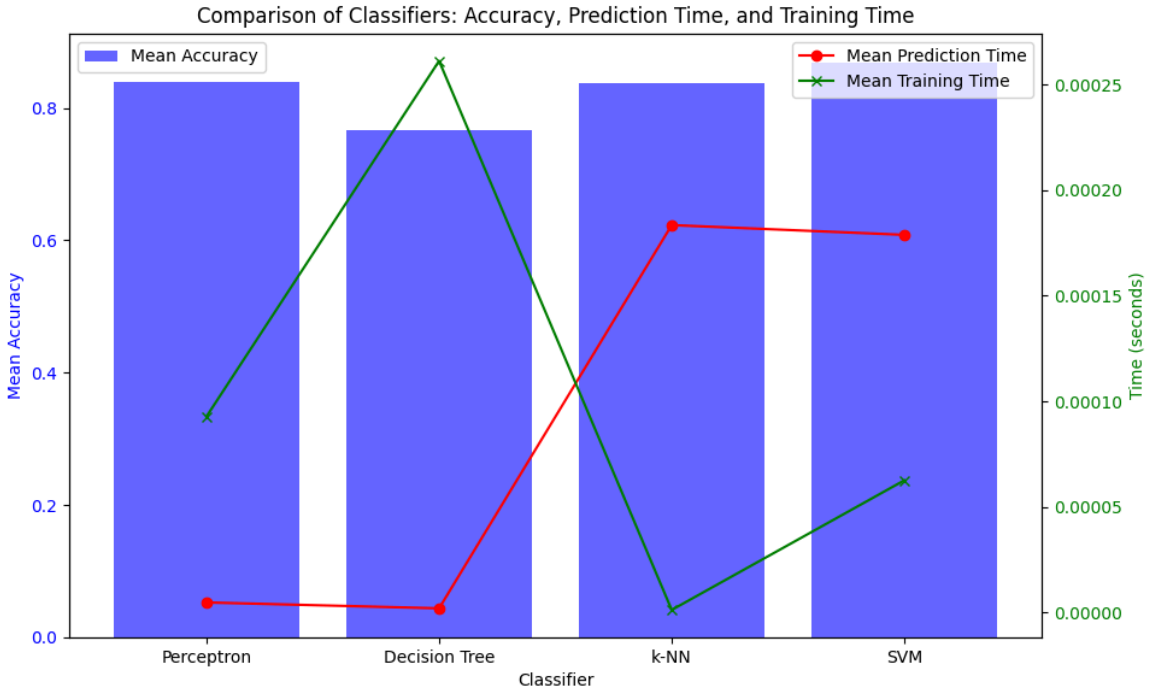
- **Perceptron** - Mean Accuracy: 0.8865, Mean Prediction Time: 0.0138 seconds, Mean Training Time: 5.0121 seconds
- **Decision Tree** - Mean Accuracy: 0.8605, Mean Prediction Time: 0.0184 seconds, Mean Training Time: 21.9998 seconds
- **k-NN** - Mean Accuracy: 0.9081, Mean Prediction Time: 3.8578 seconds, Mean Training Time: 0.0444 seconds
- **SVM** - Mean Accuracy: 0.9413, Mean Prediction Time: 62.3260 seconds, Mean Training Time: 178.8606 seconds

- Graphic Results:

sample_size = len(features)



sample_size = 1000



Comparison of Training and Prediction Times

1. **Perceptron**: The Perceptron classifier demonstrates a moderate training time (5.01 seconds) and an extremely low prediction time (0.0138 seconds). Its efficiency in prediction makes it suitable for rapid, real-time applications, though its accuracy might be slightly lower than more complex models.
2. **Decision Tree**: With a training time of 21.99 seconds, the Decision Tree is more computationally demanding to train than Perceptron. However, it maintains a relatively low prediction time (0.0184 seconds), making it effective for scenarios where quick predictions are necessary after training.
3. **k-NN (k-Nearest Neighbors)**: This classifier has minimal training time (0.0444 seconds) because it doesn't build a model during training. Its prediction time, however, is higher (3.8578 seconds) as it calculates distances to each training sample. Notably, k-NN achieves a high accuracy (90.81%) and, when accounting for total time (training + prediction), it is faster than Decision Tree, which indicates that k-NN can be highly competitive with larger datasets when accuracy is prioritized.
4. **SVM (Support Vector Machine)**: SVM has the highest training time (178.86 seconds) and prediction time (62.3260 seconds), primarily due to the complexity of finding optimal decision boundaries with the RBF kernel. While SVM achieves the highest accuracy (94.13%), its high computational demands make it suitable only for cases where accuracy is critical and the system can accommodate these times.

Observed Trends and the Impact of Sample Size

These results indicate a clear trend: simpler models (like Perceptron) exhibit shorter training and prediction times, while complex models (such as SVM and k-NN) require more resources but often achieve higher accuracy. Importantly, **the choice of the best classifier is heavily influenced by the sample size**. For instance, with larger datasets, the performance of k-NN stands out. It achieves high accuracy while maintaining a total time (training + prediction) that is lower than Decision Tree, making it particularly effective for datasets where instance-based accuracy is advantageous.

Ranking the Classifiers by Accuracy and Efficiency

1. **SVM**: Achieves the highest accuracy (94.13%) and is the best choice when accuracy is the primary requirement, though its time costs are substantial.
2. **k-NN**: Given the full dataset, k-NN performs with a high accuracy (90.81%) and, despite its prediction time, has a faster total time (training + prediction) than Decision Tree, making it ideal for applications where accuracy is valued and some latency is acceptable.
3. **Perceptron**: With good accuracy (88.65%) and minimal prediction time, Perceptron offers a balanced choice for scenarios needing efficient, real-time predictions.
4. **Decision Tree**: While Decision Tree has a reasonable prediction time, its lower accuracy (86.05%) and relatively high training time make it less competitive, especially when k-NN performs better in both accuracy and total time with a large dataset.

In conclusion, SVM ranks highest in terms of accuracy, while k-NN also performs well in accuracy but with lower prediction time. Perceptron balances accuracy and efficiency, making it a practical choice for applications requiring quicker response times.And the decision tree has a good prediction time, but I wouldn't recommend it for this dataset.