Practica

January 13, 2024

Automatic Learning

Nahuel Vazquez: X6223579D
Yelyzaveta Denysova: Y9986255Z

Introduction

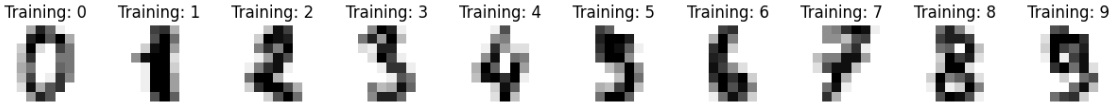
In this notebook, we'll explore machine learning algorithms with a hands-on approach. We will use two datasets: the 'digits' dataset from scikit-learn and another from the UCI Machine Learning Repository. Our goal is to understand how different models like Logistic Regression, Decision Trees, and Random Forests perform on these datasets. We'll start with basic model comparisons and then advance to more complex techniques, including k-fold cross-validation and developing custom features for computer vision to see how they impact on the learning of the models.

First we will start by setting up the dataset and importing all the libraries that will be used..

[81]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from matplotlib.colors import LinearSegmentedColormap
import matplotlib.pyplot as plt
import numpy as np

digits = load_digits()
```

Next we can visualize some examples of the dataset.

[2]:
```
_, axes = plt.subplots(nrows=1, ncols=10, figsize=(15, 4))
plt.subplots_adjust(wspace=0.5, hspace=0.5)
for ax, image, label in zip(axes, digits.images, digits.target):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation="nearest")
    ax.set_title("Training: %i" % label)
```



The images need to be flatten from 2D arrays to a single dimension array, and the ammount of elements will be counted.

[3]:
```
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
```

The set needs to be divided into training data and test data. This will be important later for comparing the different models. All of them are trained and tested on the same subsets of data. Training and testing the models on different subsets might introduce a bias since some subsets might be better for training and testing than others.

[4]:
```
X_train, X_test, Y_train, Y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False
)
```

Model Comparsion (C)

In this section, for each model the same procedure is done. We first start by obtaining the model from sklearn and then training it on the X and Y subset obtained previously. Right after we use the trained models to predict the remaining subset. Finally, with the predicted values and the real ones, we can calculate the metrics to compare fairly each model.

Logistic Regression

Training the Logistic Regression model on the half of the number samples. The other half will be used for testing.

[5]:
```
from sklearn.linear_model import LogisticRegression
import warnings
from sklearn.exceptions import ConvergenceWarning

#Supress the warning for not converging into a solution with 100 iterations
warnings.filterwarnings("ignore", category=ConvergenceWarning)

logisticRegression = LogisticRegression(random_state=0, max_iter=100).
 ↪fit(X_train, Y_train)
```

The values of the test subset are going to be predicted.

[6]:
```
logisticPredicted = logisticRegression.predict(X_test)
```

The predicted values can be compared to the real ones in order to evaluate the performance of this model.

Perceptron

Training the Perceptron model on the half of the number samples. The other half will be used for testing.

[7]:
```
from sklearn.linear_model import Perceptron
perceptron = Perceptron(random_state=0).fit(X_train, Y_train)
```

The values of the test subset are going to be predicted.

[8]:
```
perceptronPredicted = perceptron.predict(X_test)
```

The predicted values can be compared to the real ones in order to evaluate the performance of this model.

Decision Tree

Training the Decision Tree model on the half of the number samples. The other half will be used for testing.

[9]:
```
from sklearn.tree import DecisionTreeClassifier
decisionTree = DecisionTreeClassifier(random_state=0).fit(X_train, Y_train)
```

The values of the test subset are going to be predicted.

[10]:
```
treePredicted = decisionTree.predict(X_test)
```

The predicted values can be compared to the real ones in order to evaluate the performance of this model.

Random Forest

Training the Random Forest model on the half of the number samples. The other half will be used for testing.

[11]:
```
from sklearn.ensemble import RandomForestClassifier
randomForest = RandomForestClassifier(random_state=0).fit(X_train, Y_train)
```

The values of the test subset are going to be predicted.

[12]:
```
forestPredicted = randomForest.predict(X_test)
```

The predicted values can be compared to the real ones in order to evaluate the performance of this model.

Classification Reports

For comparing the performance of the different models, we have to evaluate the next metrics: accuracy, precision, recall and f1 Score. To do this, we will calculate all of them for each model. It is important to note that we are using the macro average for the following reasons: · The mertic is calculated for each class and then the average is done, thus not giving more weight to some class. · Missclassifying any particular digit is as severe as missclassifying any other, and the "macro" average provides a fair way to compare model performance across all classes. · A high macro average means that the model performs well across all the classes. The main difference with micro average is that it takes into account class weights so, if the model was good at classifying 1s and 2s but bad at any other numbers, it would still yield good results.

[13]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
import pandas as pd
# Metrics for Logistic Regression
accuracy_logistic = accuracy_score(Y_test, logisticPredicted)
precision_logistic = precision_score(Y_test, logisticPredicted, average='macro')
```

```
recall_logistic = recall_score(Y_test, logisticPredicted, average='macro')
f1_logistic = f1_score(Y_test, logisticPredicted, average='macro')
# Metrics for Perceptron
accuracy_perceptron = accuracy_score(Y_test, perceptronPredicted)
precision_perceptron = precision_score(Y_test, perceptronPredicted,␣
 ↪average='macro')
recall_perceptron = recall_score(Y_test, perceptronPredicted, average='macro')
f1_perceptron = f1_score(Y_test, perceptronPredicted, average='macro')
# Metrics for Random Forest
accuracy_forest = accuracy_score(Y_test, forestPredicted)
precision_forest = precision_score(Y_test, forestPredicted, average='macro')
recall_forest = recall_score(Y_test, forestPredicted, average='macro')
f1_forest = f1_score(Y_test, forestPredicted, average='macro')
# Metrics for Decision Tree
accuracy_tree = accuracy_score(Y_test, treePredicted)
precision_tree = precision_score(Y_test, treePredicted, average='macro')
recall_tree = recall_score(Y_test, treePredicted, average='macro')
f1_tree = f1_score(Y_test, treePredicted, average='macro')

# Creating the table
data = {
    'Model': ['Logistic Regression', 'Random Forest', 'Perceptron', 'Decision␣
 ↪Tree'],
    'Accuracy': [accuracy_logistic, accuracy_forest, accuracy_perceptron,␣
 ↪accuracy_tree],
    'Precision': [precision_logistic, precision_forest, precision_perceptron,␣
 ↪precision_tree],
    'Recall': [recall_logistic, recall_forest, recall_perceptron, recall_tree],
    'F1 Score': [f1_logistic, f1_forest, f1_perceptron, f1_tree]
}
metrics_df = pd.DataFrame(data)

# Styling
styled_df = metrics_df.style.format({
    'Accuracy': '{:.2%}',
    'Precision': '{:.2%}',
    'Recall': '{:.2%}',
    'F1 Score': '{:.2%}'
}).background_gradient(cmap='coolwarm', subset=['Accuracy', 'Precision',␣
 ↪'Recall', 'F1 Score'])

# Display the styled dataframe
styled_df
```

[13]: <pandas.io.formats.style.Styler at 0x26bb2efcd10>

In order to compare the models, it is important to first understand the meaning of each metric.

Accuracy: It is the ratio of True Positives and True Negatives over all the predictions (True Positives, False Positives, True Negatives, False Negatives). Simply put, it tells what proportion of predictions were correct
Precision: It is the ammout of True Positives over True Positives + False Positives. When it predicts a class, how often it is correct.
Recall: It tells from all the "Yes", how many of them the model got right.
F1 Score: indicates that the classifier has both a high precision (not many false positives) and a high recall (not many false negatives).

The main reason the averages are used instead of the value of the metrics for each class is because for perceptron and lineal regression, the One vs. Rest technique is used, but for Decision Tree and Random Forest, their implementation is inherently multiclass unless it is forced to be just a binary classification and then OvR is used.

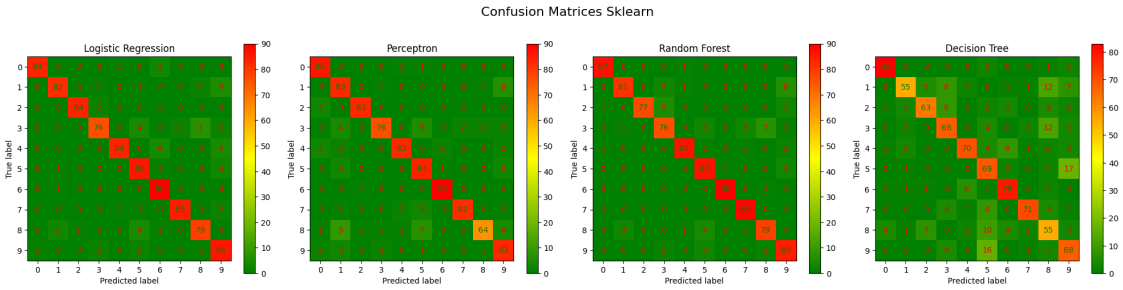
As we can see in the table, Random Forest scores the highest among all the metrics, thus being the most robust algorithm for classifying numbers with the given dataset. Where Decision Tree has the lowest accuracy, so it had the biggest number of wrong identifications; lowest precision, so almost every 10 predictions, 8 were correct; and lowest recall, for each number, every time it was given to classify, it would predict the value of that number 75.77% of the times.

Confusion Matrices

A more visual way of seeing how the models struggle with each number is using confusion matrices

[14]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))  # 1 row, 4 columns

# Confusion Matrix for Logistic Regression
ConfusionMatrixDisplay.from_predictions(Y_test, logisticPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[0])
ax[0].set_title('Logistic Regression')

# Confusion Matrix for Perceptron
ConfusionMatrixDisplay.from_predictions(Y_test, perceptronPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[1])
ax[1].set_title('Perceptron')

# Confusion Matrix for Random Forest
ConfusionMatrixDisplay.from_predictions(Y_test, forestPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[2])
ax[2].set_title('Random Forest')

# Confusion Matrix for Decision Tree
ConfusionMatrixDisplay.from_predictions(Y_test, treePredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[3])
```

```
ax[3].set_title('Decision Tree')

# Adjust layout
plt.tight_layout()
fig.suptitle("Confusion Matrices Sklearn", fontsize=16)
plt.subplots_adjust(top=0.85)  # Adjust the top of the subplots to fit the␣
 ↪suptitle
plt.show()
```



The important thing to see in the Confusin Matrices are very distinct diagonals and green backgrounds, this means that what the model predicted corresponds to the actual value of the number. In the case of the Random Forest and Logistic Regression we see the diagonals with the highest values and the backrounds with the least high values. This is due to their high accuracy.
Even with Random Forest's high accuracy, it still struggles to diffrenciate 3s and 8s, this is due to the 3 being a slice of an 8, so theyre very similar. This struggle is present among all the models except the perceptron. When it comes to the Decision Tree, it also has a hard time with 9s and 5s because of their similarities, and with 8s and 1s, yet this last ones are not similar at all. This behaviour can arise from the greedy nature.

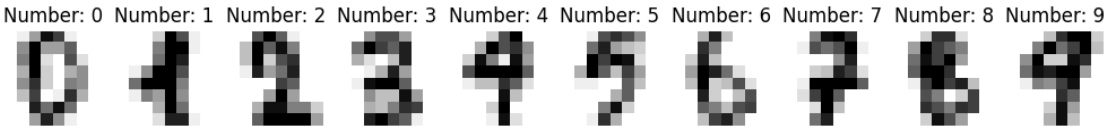
Comparsion with UCI Dataset (B.1)

In this section, the models trained previously on the sklearn dataset will be used to predict numbers from a completely diffrent dataset. As per the handwritten number style, they are very similar to the original set as we can see in the next example. This should theoretically make the models perform in a similar manner, although this will be proven or denied once the metrics are evaluated.

First the dataset needs to be obtained.

[15]:
```
from ucimlrepo import fetch_ucirepo

optical_recognition_of_handwritten_digits = fetch_ucirepo(id=80)
X = optical_recognition_of_handwritten_digits.data.features
Y = optical_recognition_of_handwritten_digits.data.targets
X = np.array(X)
Y = Y['class'].values

# Find the first occurrence of each number in Y
unique_numbers = np.unique(Y)
```

```
fig, axes = plt.subplots(1, len(unique_numbers), figsize=(10, 2))

for num in unique_numbers:
    index = np.where(Y == num)[0][0]
    digit_image = X[index].reshape(8, 8)  # Assuming each image is 8x8 pixels

    axes[num].imshow(digit_image, cmap=plt.cm.gray_r, interpolation='nearest')
    axes[num].set_title(f'Number: {num}')
    axes[num].axis('off')

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



And next we need to predict all the values using the models we trained earlier.

[16]:
```
newLogisticPredicted = logisticRegression.predict(X)
newPerceptronPredicted = perceptron.predict(X)
newTreePredicted = decisionTree.predict(X)
newForestPredicted = randomForest.predict(X)
```

Lastly we calculate the metrics again to compare them with the original ones, so we can see how completely new data is recognised by the models.

Classification reports

[17]:
```
# Metrics for Logistic Regression
newAccuracy_logistic = accuracy_score(Y, newLogisticPredicted)
newPrecision_logistic = precision_score(Y, newLogisticPredicted,␣
 ↪average='macro')
newRecall_logistic = recall_score(Y, newLogisticPredicted, average='macro')
newF1_logistic = f1_score(Y, newLogisticPredicted, average='macro')
# Metrics for Perceptron
newAccuracy_perceptron = accuracy_score(Y, newPerceptronPredicted)
newPrecision_perceptron = precision_score(Y, newPerceptronPredicted,␣
 ↪average='macro')
newRecall_perceptron = recall_score(Y, newPerceptronPredicted, average='macro')
newF1_perceptron = f1_score(Y, newPerceptronPredicted, average='macro')
# Metrics for Random Forest
newAccuracy_forest = accuracy_score(Y, newForestPredicted)
newPrecision_forest = precision_score(Y, newForestPredicted, average='macro')
newRecall_forest = recall_score(Y, newForestPredicted, average='macro')
```

```
newF1_forest = f1_score(Y, newForestPredicted, average='macro')
# Metrics for Decision Tree
newAccuracy_tree = accuracy_score(Y, newTreePredicted)
newPrecision_tree = precision_score(Y, newTreePredicted, average='macro')
newRecall_tree = recall_score(Y, newTreePredicted, average='macro')
newF1_tree = f1_score(Y, newTreePredicted, average='macro')

# Creating the table
newData = {
    'Model': ['Logistic Regression', 'Random Forest', 'Perceptron', 'Decision␣
 ↪Tree'],
    'Accuracy': [newAccuracy_logistic, newAccuracy_forest,␣
 ↪newAccuracy_perceptron, newAccuracy_tree],
    'Precision': [newPrecision_logistic, newPrecision_forest,␣
 ↪newPrecision_perceptron, newPrecision_tree],
    'Recall': [newRecall_logistic, newRecall_forest, newRecall_perceptron,␣
 ↪newRecall_tree],
    'F1 Score': [newF1_logistic, newF1_forest, newF1_perceptron, newF1_tree]
}
newMetrics_df = pd.DataFrame(newData)

# Styling
newStyled_df = newMetrics_df.style.format({
    'Accuracy': '{:.2%}',
    'Precision': '{:.2%}',
    'Recall': '{:.2%}',
    'F1 Score': '{:.2%}'
}).background_gradient(cmap='coolwarm', subset=['Accuracy', 'Precision',␣
 ↪'Recall', 'F1 Score'])

# Display the styled dataframe
newStyled_df
```

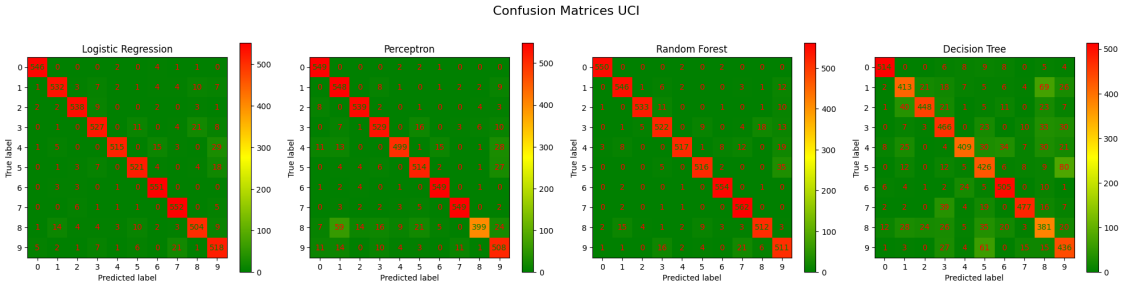
[17]: <pandas.io.formats.style.Styler at 0x26bb7990e10>

We can see on the Metrics that the models yield similar results using the UCI dataset. As hyhypothesised in the beggining of this section, this is due the handwritten style of this set is similar to the sklearn one. It seems as the models have performed better with this dataset than the original, but this might be due to the subset used for training the models. If that subset happens to contain numbers similar to the UCI set, then the models would have learned to identify thst style of handwritten numbers better. This can be proven or denied by training the models with a different training sklearn subset.

Confusion Matrices

As it was done before, the confusion matrices are now showcased to visually identify with which numbers the models struggle the most.

[18]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))  # 1 row, 4 columns

# Confusion Matrix for Logistic Regression
ConfusionMatrixDisplay.from_predictions(Y, newLogisticPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[0])
ax[0].set_title('Logistic Regression')

# Confusion Matrix for Perceptron
ConfusionMatrixDisplay.from_predictions(Y, newPerceptronPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[1])
ax[1].set_title('Perceptron')

# Confusion Matrix for Random Forest
ConfusionMatrixDisplay.from_predictions(Y, newForestPredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[2])
ax[2].set_title('Random Forest')

# Confusion Matrix for Decision Tree
ConfusionMatrixDisplay.from_predictions(Y, newTreePredicted,␣
 ↪cmap=LinearSegmentedColormap.from_list('custom_green_red', ['green',␣
 ↪'yellow', 'red'], N=256), ax=ax[3])
ax[3].set_title('Decision Tree')

# Adjust layout
plt.tight_layout()
fig.suptitle("Confusion Matrices UCI", fontsize=16)
plt.subplots_adjust(top=0.85)  # Adjust the top of the subplots to fit the␣
 ↪suptitle
plt.show()
```



As we can see it is very similar to the matrices obtained with the sklearn subset. Logistic regression and Random Forest still outperform the other 2 models, the Perceptron still struggles identifying 8s and Decision Tree is by far the worst model. It seems that by choosing a similar dataset, the

relative number succssesfull and wrong predictions for each model scales with the dataset size.

Reteaching the Models (B.2)

K-fold Crossvalidation

To see if the models can be improved further, every model is going to be trained on a split of the UCI dataset using K-fold Crossvalidation. This technic consists on dividing the dataset into **K** subsets and then for each iteration, choosing a different subset as test. This allows to try different combinations in order to find the subset which trains the models the best.
Common values for k-fold are k=5, k=10, and k=20, being 5 and 10 the best for small datasets. In this case 5 will be used.
To achieve this the *GridSearchCV(**model**, **params**, **k**)* function comes in handy. This function recieve the following parameters:
model: the model that wants to be trained.
params: all the parameters that will be tried on the model.
k: the ammount of subsets that will be created from the original one.
And will allow us to do the k-fold crossvalidation in addition to also try different combinations of the parameters for each model. It will return the best values for each parameter that result in the model that performs the best.

[19]:
```
#Divide the dataset so later we can compare it agains the original models
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.5, shuffle=False
)
```

Logistic Regression

Explanation of every parameter used in *GridSearchCV()* for this model:
C : It prevents overfitting by penalizing the model. Higher values might allow the model to find more complex paterns but might cause overfitting. Lower values do the opposite.**solver: **It is the algorithm to use in the optimization problem.**
max_iter: **The maximum ammount of iterations it will do if it doesnt converge into a solution.**
The parameter penalty** is not used since it is incompatible with some of the solvers..

[20]:
```
paramsLR = {'C': [0.01, 1, 1.5], 'solver': ['newton-cg','lbfgs','liblinear'],␣
 ↪'max_iter':[10, 100, 800]}
newLogisticPredict = GridSearchCV(LogisticRegression(), paramsLR, cv=5)
newLogisticPredict.fit(X_train,Y_train)
print("Best Parameters: "+str(newLogisticPredict.best_params_))
```

```
Best Parameters: {'C': 0.01, 'max_iter': 10, 'solver': 'newton-cg'}
```

Perceptron

Explanation of every parameter used in *GridSearchCV()* for this model:
penalty: It is the technique used for preventing overfitting.
alpha: It is the constant used for multiplying the penalty in case of it being used. Small numbers might cause overfitting.
max_iter: The maximum ammount of iterations it will do if it doesn't converge into a solution.

[21]:
```
paramsP = {'penalty': ['l2','l1','elasticnet', None], 'alpha': [0.001, 0.0001,␣
↪0.00001], 'max_iter':[100, 1000, 1500]}
newPerceptronPredict = GridSearchCV(Perceptron(), paramsP, cv=5)
newPerceptronPredict.fit(X_train,Y_train)
print("Best Parameters: "+str(newPerceptronPredict.best_params_))
```

```
Best Parameters: {'alpha': 0.001, 'max_iter': 100, 'penalty': None}
```

Decision Tree

Explanation of every parameter used in *GridSearchCV()* for this model:
max_depth: It is the maximum depth that the tree can have.
min_samples_split: It is how many samples there has to be in a node for it to split into leafs, it can help with controling overfitting.

[22]:
```
paramsDT = {'max_depth': [60, 70, 80, 90, 100] , 'min_samples_split': [2,3,4]}
newDecisionTreePredict = GridSearchCV(DecisionTreeClassifier(), paramsDT, cv=5)
newDecisionTreePredict.fit(X_train,Y_train)
print("Best Parameters: "+str(newDecisionTreePredict.best_params_))
```

```
Best Parameters: {'max_depth': 70, 'min_samples_split': 4}
```

Note that if many executions are done to get the best parameters, each time the value of max_depth is different. The only thing that seems to change from execution to execution is the subsets chosen in the k-fold.

Random Forest

Explanation of every parameter used in *GridSearchCV()* for this model:
n_estimators: It is the number of trees in the forest.
min_samples_leaf: It is the minimum ammount of samples that a leaf must have to become a leaf.
min_samples_split: It is how many samples there has to be in a node for it to split into leafs, it can help with controling overfitting.
max_depth is excluded in this case since it doesn't stay the same from execution to execution.

[23]:
```
paramsRF = {'n_estimators':[50, 100, 150, 200, 250], 'min_samples_leaf':␣
↪[1,2,3] , 'min_samples_split': [2,3,4]}
newRandomForestPredict = GridSearchCV(RandomForestClassifier(), paramsRF, cv=5)
newRandomForestPredict.fit(X_train,Y_train)
print("Best Parameters: "+str(newRandomForestPredict.best_params_))
```

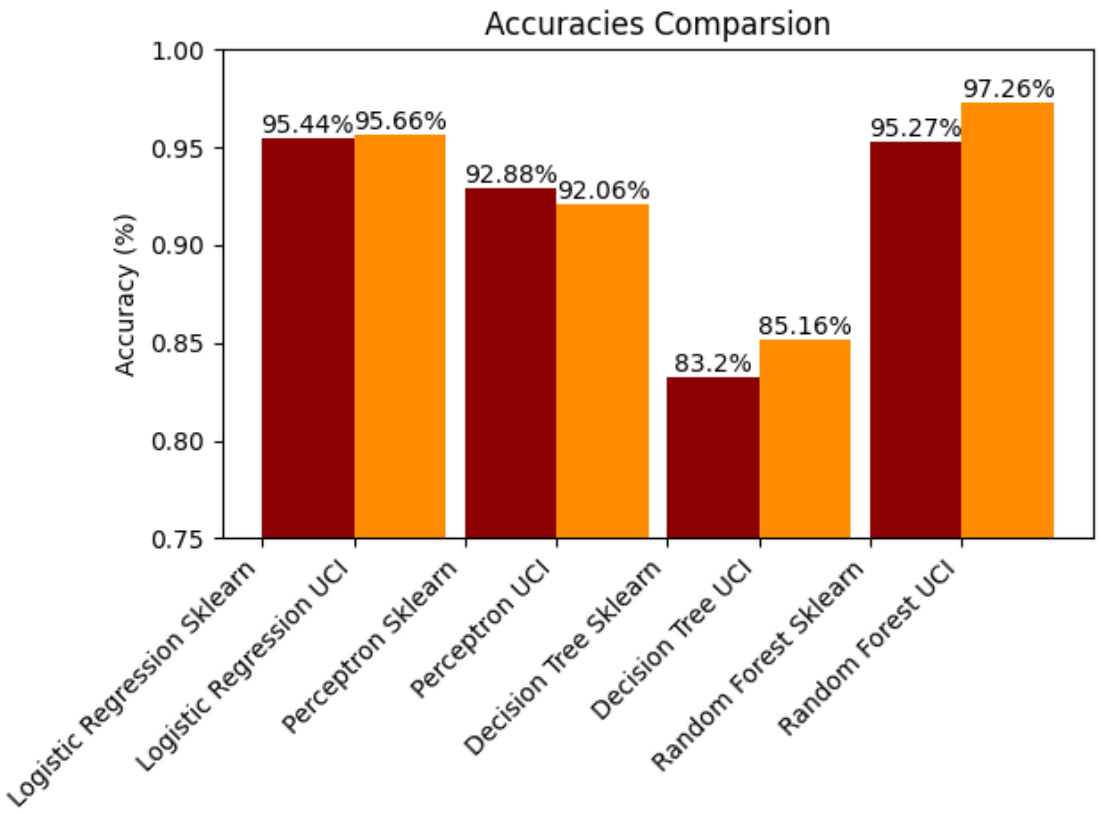
```
Best Parameters: {'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators':
250}
```

Comparsion

In this case the best way to compare the results it is by comparing the Accuracy since it is the most relevant metric in this case, and represents best the performance of the models.

[24]:
```
models = {
    'Logistic Regression Sklearn': logisticRegression,
```

```
    'Logistic Regression UCI': newLogisticPredict,
    'Perceptron Sklearn': perceptron,
    'Perceptron UCI': newPerceptronPredict,
    'Decision Tree Sklearn': decisionTree,
    'Decision Tree UCI': newDecisionTreePredict,
    'Random Forest Sklearn': randomForest,
    'Random Forest UCI': newRandomForestPredict
}

# Calculate accuracies
accuracies = {name: accuracy_score(Y_test, model.predict(X_test)) for name,␣
 ↪model in models.items()}

# Prepare data for plotting
model_names = list(accuracies.keys())
accuracy_values = list(accuracies.values())

# Set up the positions for each group of bars
bar_width = 1
gap = 0.2  # Gap between each pair of bars
positions = np.arange(len(models)) + np.repeat(np.arange(len(models)//2) * gap,␣
 ↪2)

# Plotting
fig, ax = plt.subplots()
colors = ['darkred', 'darkorange'] * 4  # Alternate colors for each bar
bars = ax.bar(positions, accuracy_values, bar_width, color=colors)

# Add the accuracy values on top of the bars
for bar in bars:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{round(yval*100, 2)}%',␣
 ↪ha='center', va='bottom', color='black')

# Set labels and title
ax.set_ylabel('Accuracy (%)')
ax.set_title('Accuracies Comparsion')
ax.set_xticks(positions - bar_width/2)
ax.set_xticklabels(model_names)
ax.set_ylim(0.75, 1.00)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

# Show the plot
plt.show()
```



This graphic compares the performance (accuracy) of the different models. The models are paired by type and the colors represent which dataset was used dor training:
RED: Sklearn
Orange: UCI
It seems like all the models perform better on the UCI dataset except the Perceptron. The logical thing to think is that all models trained on UCI dataset, which is bigger, using k-fold crossvalidation and use the best parameter combination should perform better. This is until we realize that one of the varaible parameters used in the perceptron is the alpha. The fact that a smaller number than the default was chosen by the GridSearchCV() means that this model is prone to overfitting. From the point of view of GridSearchCV() a lower number would yield the best results because it would memorize the patterns of the training subset, but this specific patterns don't apply to a different dataset, that is why the perceptron trained on the sklearn with default parameters performed better.

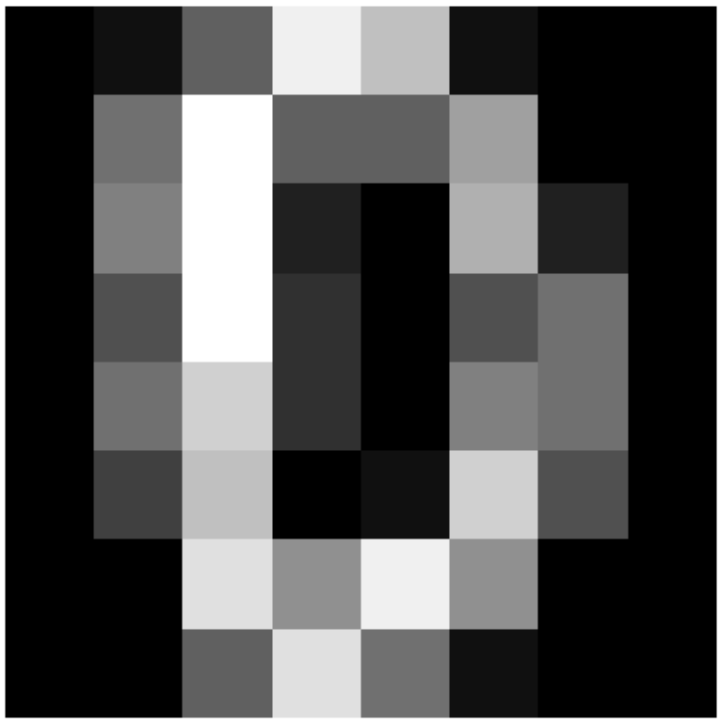
Computer Vision (A)

In this section three computer vision characteristics are developed and tested to see how they affect the training of the models. The 3 characteristics are:
Number of black pixels: It is the ammount of non-white pixels in the image.
The ammount of pixels per quadrants: It is the ammount of non-white pixels in each quadrant.
Distance from mass center: It is the average of how far each non-white pixel is from the concentration of non-white pixels.

Implementations

In order to show the functionality of the images, an example will be shown with the first image of the dataset. Note that the colors are inversed but this is meaningless for the models since the tag of *white* and *non-white* are equivalent to *black* and *non-black*.

[41]:
```
 # Reshape the array into a 2D image
image = X[0].reshape((8, 8))

# Display the image
plt.imshow(image, cmap='gray')
plt.axis('off')  # Turn off axis numbers
plt.show()
```



For the **Number of black pixels** function, all the pixels that have a value bigger than 0, means that they're non-white, so this ones must be counted. Fully black pixels have a value of 16. This characteristic can be useful to identify numbers with simple shapes since they use less pixels to be represented.

[47]:
```
def count_non_white_pixels(image):
    non_white_pixels = np.sum(image > 0)
    return non_white_pixels

# Example usage with the first image in the dataset
```

```
print(X[0])
print("Non-White pixels: ",count_non_white_pixels(X[0]))
```

```
[ 0  1  6 15 12  1  0  0  0  7 16  6  6 10  0  0  0  8 16  2  0 11  2  0
  0  5 16  3  0  5  7  0  0  7 13  3  0  8  7  0  0  4 12  0  1 13  5  0
  0  0 14  9 15  9  0  0  0  0  6 14  7  1  0  0]
Non-White pixels:  38
```

For **The ammount of pixels per quadrants** the image is reshaped into the original 8x8 form and then the non white pixels for each quadrant is counted. This characteristic can be used to identify numbers with vertical or horizontal simetries like 1, 3, 8 or 0.

[58]:
```
def count_cuadrants(image_array):
    # Reshape the array into an 8x8 matrix
    image_matrix = np.reshape(image_array, (8, 8))

    # Initialize counts for each quadrant
    quadrant_counts = [0, 0, 0, 0]

    # Iterate over each quadrant
    for i in range(2):
        for j in range(2):
            quadrant = image_matrix[i*4:(i+1)*4, j*4:(j+1)*4]
            quadrant_counts[i*2 + j] = np.sum(quadrant > 0)

    return quadrant_counts

# Example usage with the first image in the dataset
print("Non-white pixel counts in each quadrant:", count_cuadrants(X[0]))
```

```
Non-white pixel counts in each quadrant: [12, 8, 9, 9]
```

For **Distance from mass center** first all the non white pixels are counted to get the average position of non white pixels, and with this, the distance of each pixel to this imaginary center is calculated. This characteristic can be used to differentiate between numbers because each number might have a different distance to the center of mass due to different shapes.

[50]:
```
def avg_distance_to_center_of_mass(image_array):
    # Reshape the array into an 8x8 matrix
    image = np.reshape(image_array, (8, 8))

    # Coordinates of each pixel
    rows, cols = image.shape
    Y, X = np.ogrid[:rows, :cols]

    # Black pixels are those with value 0
    black_pixels = (image > 0)

    # Total number of black pixels
```

```
    total_black_pixels = np.sum(black_pixels)

    # If there are no black pixels, return zero values
    if total_black_pixels == 0:
        return 0

    # Calculate the center of mass
    center_of_mass_x = np.sum(X * black_pixels) / total_black_pixels
    center_of_mass_y = np.sum(Y * black_pixels) / total_black_pixels
    center_of_mass = (center_of_mass_x, center_of_mass_y)

    # Calculate the average distance to the center of mass
    distances = np.sqrt((X - center_of_mass_x)**2 + (Y - center_of_mass_y)**2)
    avg_distance = np.sum(distances * black_pixels) / total_black_pixels

    return avg_distance

# Example usage with the first image in the dataset
print("Average distance of pixels to the center of mass:",␣
 ↪avg_distance_to_center_of_mass(X[0]))
```

```
Average distance of pixels to the center of mass: 2.627536724031988
```

We will append the information of the characteristics developed to the arrays of the images and train new models with this improved datasets. For training we will use the best parameters found by *GridSearchCV()* and the k-fold crossvalidation, so the comparsion to the original models trained on the UCI dataset is fair.

Training

[71]:
```
#Append infromation to the testing subset
improvedX_test = np.empty((0,70))
for img in X_test:
    copia = np.copy(img)
    copia = np.append(copia, count_non_white_pixels(img)) #Append ammount of␣
 ↪non white pixels
    cuadrants = count_cuadrants(img)
    for cuadrant in cuadrants:
        copia = np.append(copia, cuadrant) #Append the ammount of non white␣
 ↪pixels on each cuadrant
    copia = np.append(copia, avg_distance_to_center_of_mass(img)) #Append the␣
 ↪distance to the center of mass

    improvedX_test = np.vstack((improvedX_test, copia))

#Append infromation to the training subset
improvedX_train = np.empty((0,70))
for img in X_train:
```

```
    copia = np.copy(img)
    copia = np.append(copia, count_non_white_pixels(img)) #Append ammount of␣
 ↪non white pixels
    cuadrants = count_cuadrants(img)
    for cuadrant in cuadrants:
        copia = np.append(copia, cuadrant) #Append the ammount of non white␣
 ↪pixels on each cuadrant
    copia = np.append(copia, avg_distance_to_center_of_mass(img))#Append the␣
 ↪distance to the center of mass

    improvedX_train = np.vstack((improvedX_train, copia))

#Create new models with the same properties as the models trained with UCI
improvedRandomForestPredict = GridSearchCV(RandomForestClassifier(), {k: [v]␣
 ↪for k, v in newRandomForestPredict.best_params_.items()}, cv=5)
improvedRandomForestPredict.fit(improvedX_train,Y_train)
improvedDecisionTreePredict = GridSearchCV(DecisionTreeClassifier(), {k: [v]␣
 ↪for k, v in newDecisionTreePredict.best_params_.items()}, cv=5)
improvedDecisionTreePredict.fit(improvedX_train,Y_train)
improvedPerceptronPredict = GridSearchCV(Perceptron(), {k: [v] for k, v in␣
 ↪newPerceptronPredict.best_params_.items()}, cv=5)
improvedPerceptronPredict.fit(improvedX_train,Y_train)
improvedLogisticPredict = GridSearchCV(LogisticRegression(), {k: [v] for k, v␣
 ↪in newLogisticPredict.best_params_.items()}, cv=5)
improvedLogisticPredict.fit(improvedX_train,Y_train)
```

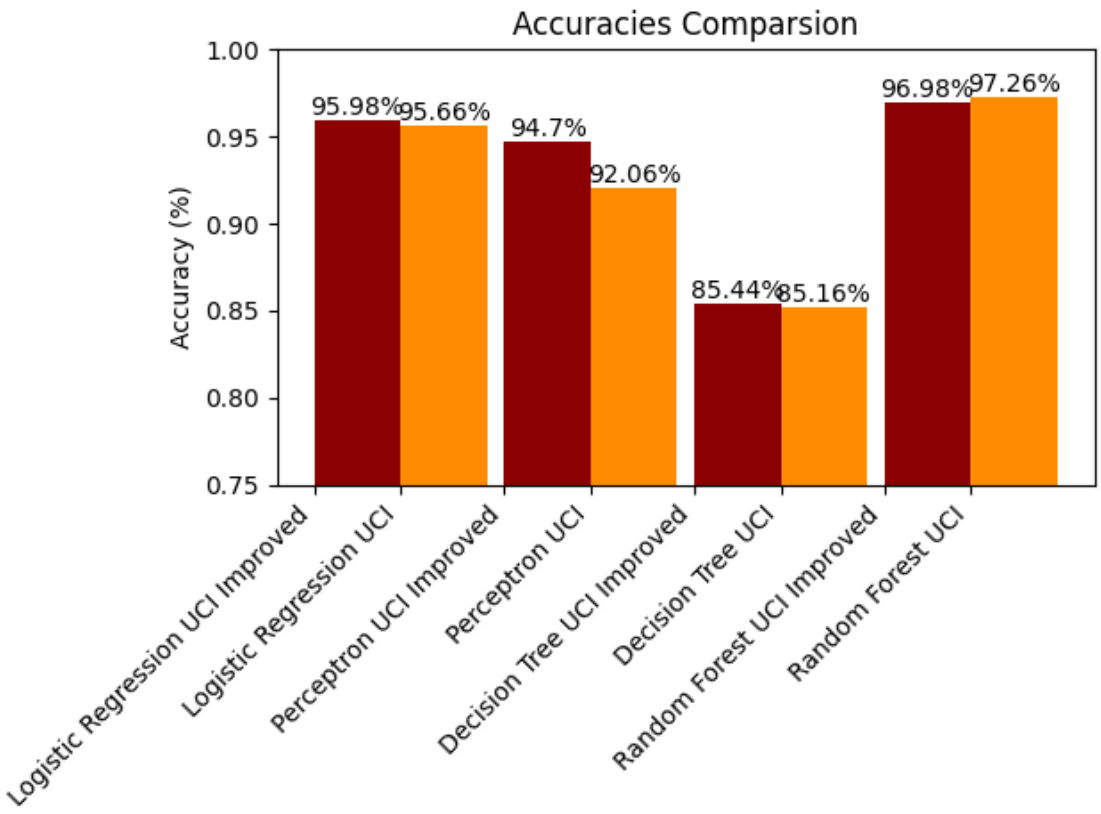
```
dict_items([('min_samples_leaf', 1), ('min_samples_split', 2), ('n_estimators',
250)])
dict_items([('min_samples_leaf', 1), ('min_samples_split', 2), ('n_estimators',
250)])
```

Comparsion

Once obtained the new models trained on the improved UCI dataset, we will compare the accuracy of them with the previous models trained on the UCI dataset with k-fold crossvalidation.

[72]:
```
models = {
    'Logistic Regression UCI Improved': improvedLogisticPredict,
    'Logistic Regression UCI': newLogisticPredict,
    'Perceptron UCI Improved': improvedPerceptronPredict,
    'Perceptron UCI': newPerceptronPredict,
    'Decision Tree UCI Improved': improvedDecisionTreePredict,
    'Decision Tree UCI': newDecisionTreePredict,
    'Random Forest UCI Improved': improvedRandomForestPredict,
    'Random Forest UCI': newRandomForestPredict
}

# Calculate accuracies
```

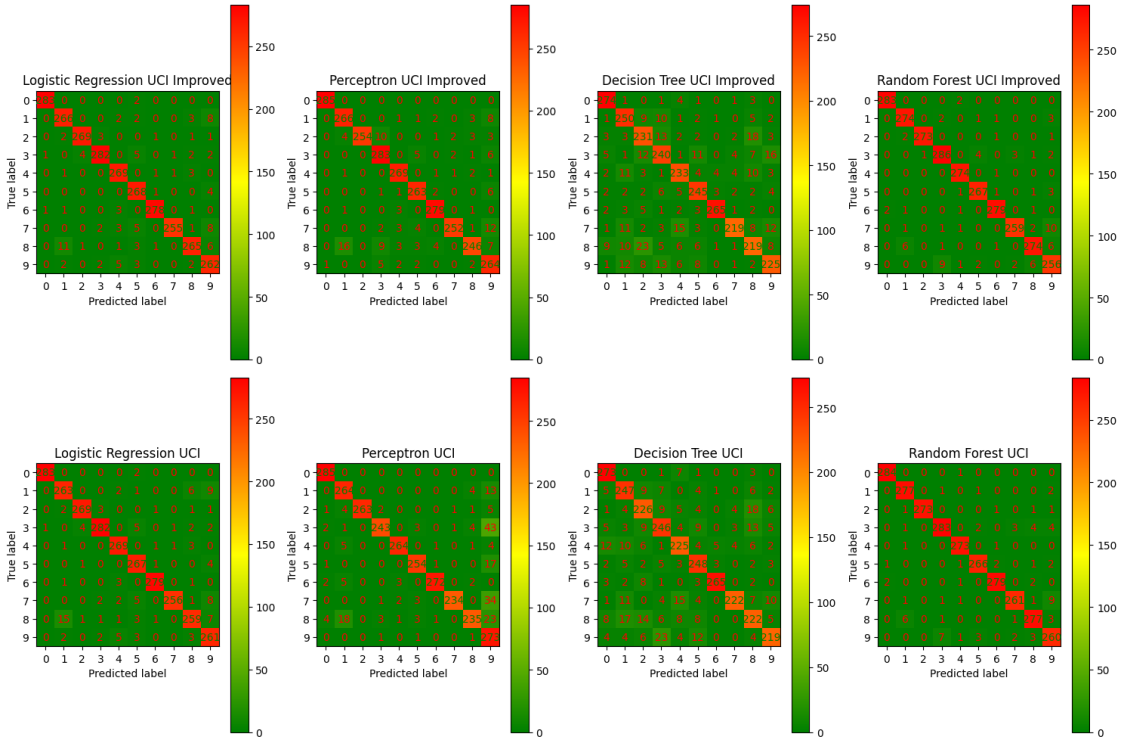
```
accuracies = {}
for index, (name, model) in enumerate(models.items()):
    if index % 2 == 0:
        # Even-indexed models use improvedX_test
        accuracies[name] = accuracy_score(Y_test, model.predict(improvedX_test))
    else:
        # Odd-indexed models use X_test
        accuracies[name] = accuracy_score(Y_test, model.predict(X_test))

# Prepare data for plotting
model_names = list(accuracies.keys())
accuracy_values = list(accuracies.values())

# Set up the positions for each group of bars
bar_width = 1
gap = 0.2  # Gap between each pair of bars
positions = np.arange(len(models)) + np.repeat(np.arange(len(models)//2) * gap,␣
 ↪2)

# Plotting
fig, ax = plt.subplots()
colors = ['darkred', 'darkorange'] * 4  # Alternate colors for each bar
bars = ax.bar(positions, accuracy_values, bar_width, color=colors)

# Add the accuracy values on top of the bars
for bar in bars:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{round(yval*100, 2)}%',␣
 ↪ha='center', va='bottom', color='black')

# Set labels and title
ax.set_ylabel('Accuracy (%)')
ax.set_title('Accuracies Comparsion')
ax.set_xticks(positions - bar_width/2)
ax.set_xticklabels(model_names)
ax.set_ylim(0.75, 1.00)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

# Show the plot
plt.show()
```



The Logistic Regression, Decision Tree and specially the Perceptron are benefited from the extra information since it gives clues to distinguish between numbers that originally might seem similar. The Random Forest seems to have not been benefited from the extra information. The random nature of the random forest might not allow it to take advantage of the additional information.
To compare further the models, the confussion matrices of all of them will be displayed::

[89]:
```
# Number of models
num_models = len(models)

# Create a figure for subplots
fig, axes = plt.subplots(nrows=2, ncols=num_models // 2 + num_models % 2,␣
 ↪figsize=(15, 10))

# Adjust if the number of models is odd
if num_models % 2 != 0:
    axes[-1, -1].axis('off')  # Turn off the last subplot if odd number of␣
 ↪models

# Iterate over models and plot confusion matrices
for index, (name, model) in enumerate(models.items()):
    # Determine row and column based on index
```

```
    row = 0 if index % 2 == 0 else 1  # Even-indexed models in first row,␣
 ↪odd-indexed in second row
    col = index // 2

    # Select the appropriate test set
    X_test_used = improvedX_test if index % 2 == 0 else X_test

    # Compute confusion matrix
    cm = confusion_matrix(Y_test, model.predict(X_test_used))

    # Plotting
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(ax=axes[row, col], cmap=LinearSegmentedColormap.
 ↪from_list('custom_green_red', ['green', 'yellow', 'red'], N=256))
    axes[row, col].set_title(name)

# Adjust layout
plt.tight_layout()
plt.show()
```



In the confussion matrices it easy to appreciate that the models that get the most benefited from the additional information are the Logistic Regression and Perceptron. Even tough the Decision Tree's accurracy is improved, in the diagonal some values are decreased and some other increased,

meaning that the extra information is useful for predicting some numbers better but might cause confussion in some others. The same can be said about the Random Forest.

[ ]: