ep04

July 11, 2021

[1]:
```
name = "Renata Sarmet Smiderle Mendes"  # write YOUR NAME

honorPledge = "I affirm that I have not given or received any unauthorized " \
              "help on this assignment, and that this work is my own.\n"


print("\nName: ", name)
print("\nHonor pledge: ", honorPledge)
```

```
Name:  Renata Sarmet Smiderle Mendes

Honor pledge:  I affirm that I have not given or received any unauthorized help
on this assignment, and that this work is my own.
```

1 Libraries

Versions: - Python: 3.8.6 - scikit-learn: 0.24.2 - tensorflow: 2.5.0 - numpy: 1.19.5 - matplotlib: 3.4.1

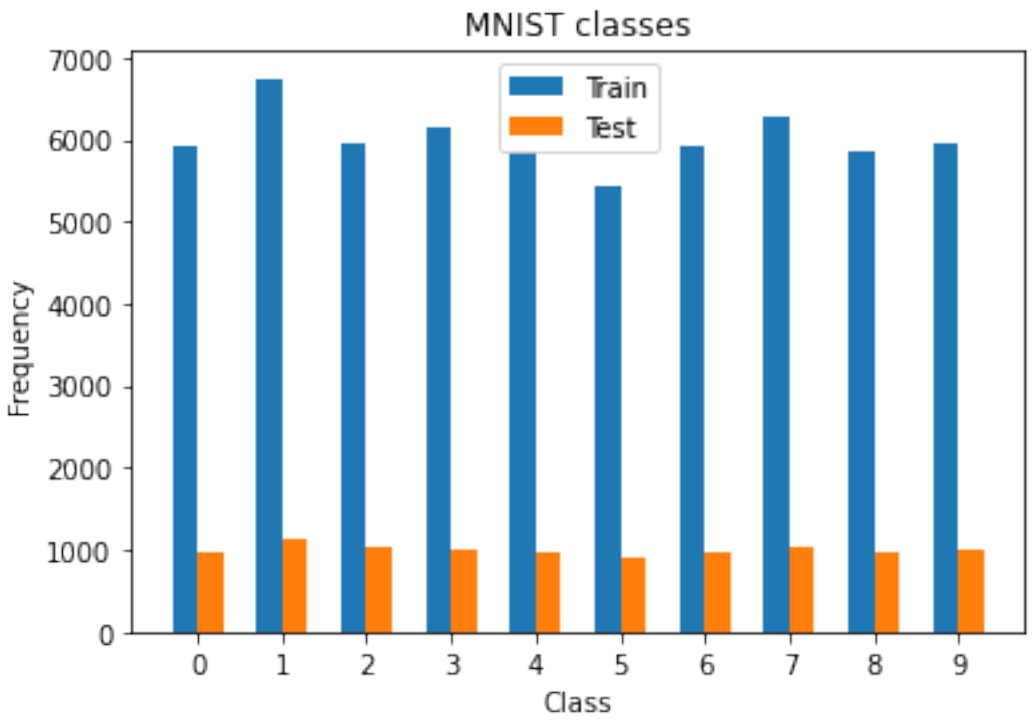
[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

2 Preprocessing

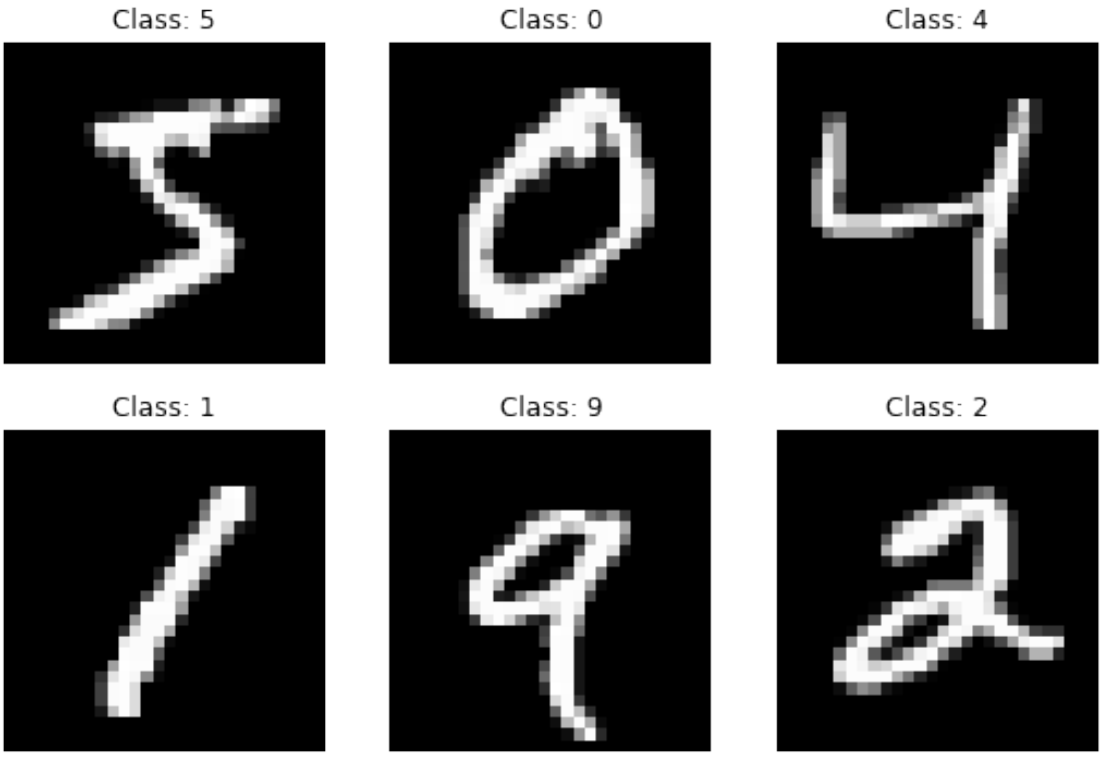
[3]:
```
# Reading the MNIST dataset, already separed in train and test
(X_train_ori, y_train_ori), (X_test_ori, y_test_ori) = mnist.load_data()

print(X_train_ori.shape, y_train_ori.shape)
print(X_test_ori.shape, y_test_ori.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

[4]:
```
# Plotting the number of elements of each class in both train and test set.
# We can see that it is balanced in both datasets.
labels = ["%s"%i for i in range(10)]

unique, counts = np.unique(y_train_ori, return_counts=True)
uniquet, countst = np.unique(y_test_ori, return_counts=True)

fig, ax = plt.subplots()
rects1 = ax.bar(unique - 0.15, counts, 0.3, label='Train')
rects2 = ax.bar(unique + 0.15, countst, 0.3, label='Test')
ax.legend()
ax.set_xticks(unique)
ax.set_xticklabels(labels)

plt.title('MNIST classes')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```

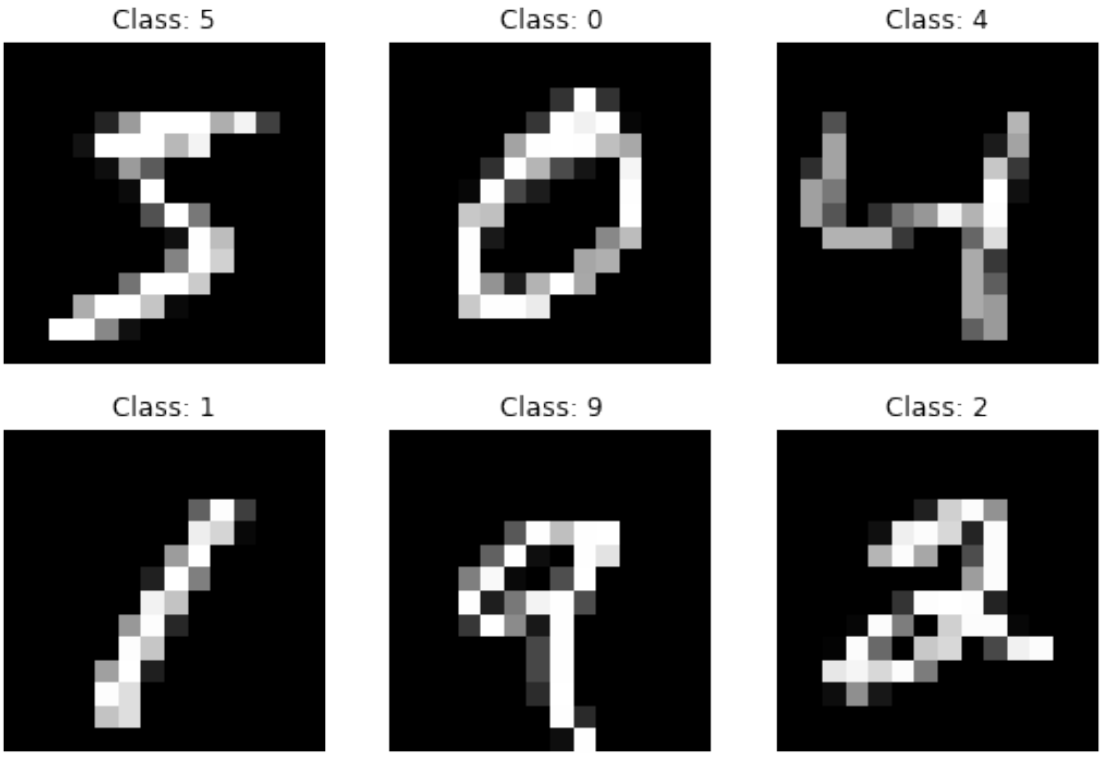


[5]:
```
# Showing examples of some classes
fig, ax = plt.subplots(2, 3, figsize = (9, 6))

for i in range(6):
    ax[i//3, i%3].imshow(X_train_ori[i], cmap='gray')
    ax[i//3, i%3].axis('off')
    ax[i//3, i%3].set_title("Class: %d"%y_train_ori[i])

plt.show()
```



[6]:
```
# Reducing the image size to its half
X_train = np.array([image[::2, 1::2] for image in X_train_ori])
X_test  = np.array([image[::2, 1::2] for image in X_test_ori])

y_train = y_train_ori
y_test = y_test_ori
```

[7]:
```
# Showing examples of some classes with the reduced size
fig, ax = plt.subplots(2, 3, figsize = (9, 6))

for i in range(6):
    ax[i//3, i%3].imshow(X_train[i], cmap='gray')
    ax[i//3, i%3].axis('off')
    ax[i//3, i%3].set_title("Class: %d"%y_train_ori[i])

plt.show()
```



[8]:
```
# Converting the dataset to float and normalizing the intensities to the␣
 ↪interval [0, 1] in both train and test sets
X_train = (X_train/255.0).astype('float32').reshape((60000,14*14))
X_test = (X_test/255.0).astype('float32').reshape((10000,14*14))

print(X_train.dtype)
print(X_test.dtype)

print("\nShape of X_train: ", X_train.shape)
print("Shape of X_test: ", X_test.shape)

print("\nMinimum value in X_train:", np.amin(X_train))
print("Maximum value in X_train:", np.amax(X_train))

print("\nMinimum value in X_test:", np.amin(X_test))
print("Maximum value in X_test:", np.amax(X_test))
```

```
float32
float32

Shape of X_train:  (60000, 196)
Shape of X_test:  (10000, 196)
```

```
Minimum value in X_train: 0.0
Maximum value in X_train: 1.0

Minimum value in X_test: 0.0
Maximum value in X_test: 1.0
```

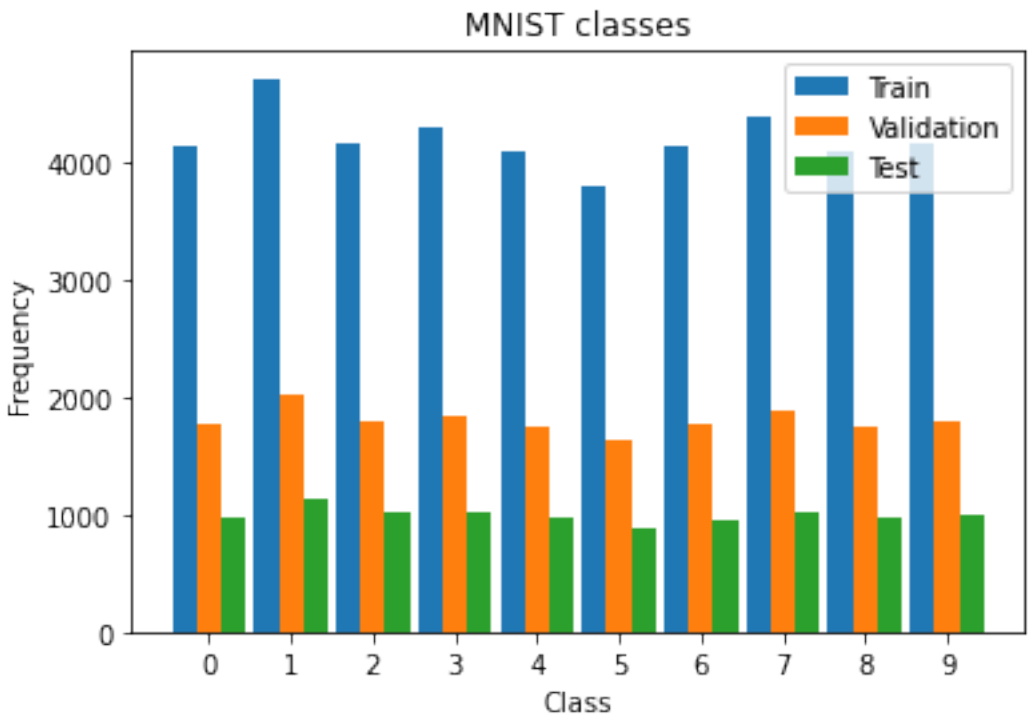
3 Separating Datasets

We will separate the training set into two parts in a stratified way: 70% train and 30% validation.

From this section to the end, we will use random_state=42 in every necessary place to allow reproduction of the results.

[9]:
```
# Splitting the dataset in a stratified way
X_train_set, X_val, y_train_set, y_val = train_test_split(X_train, y_train,
                                                          test_size=0.3,
                                                          stratify=y_train,
                                                          random_state=42)
```

[10]:
```
# Plotting the number of elements of each class in each dataset (new train, new validation and test)
# Checking the classes distributions, we can see that it is indeed stratified and balanced in all datasets.
labels = ["%s"%i for i in range(10)]

unique, counts = np.unique(y_train_set, return_counts=True)
uniquev, countsv = np.unique(y_val, return_counts=True)
uniquet, countst = np.unique(y_test, return_counts=True)

fig, ax = plt.subplots()
rects1 = ax.bar(unique - 0.3, counts, 0.3, label='Train')
rects2 = ax.bar(unique, countsv, 0.3, label='Validation')
rects3 = ax.bar(unique + 0.3, countst, 0.3, label='Test')
ax.legend()
ax.set_xticks(unique)
ax.set_xticklabels(labels)

plt.title('MNIST classes')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```



4 Training, evaluating and selecting models

We will use the X_train_set and y_train_set to train different models, varying the hyperparameters, and choose the best configuration for each one. For that, we will use grid search techniques and cross validation. The grid search considers all parameter combinations and the cross validation splits the data into k folds and use k-1 folds to train and 1 fold to test, so we don't need to split our data one more time. More details about GridSearchCV can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html).

In order to choose the best configuration, we will analyse the precision, the recall and the execution time. The precision is the proportion of predicted Positives that is truly Positive (TP/(TP+FP)) and recall is the proportion of actual Positives that is correctly classified (TP/(TP+FN)).

After this step, we will have three selected models: a logistic regression model, a neural network model, and a SVM model.

[11]:
```
def train_with_grid_search_cv(model, param_grid, X_train, y_train):
    """
    Grid search cv with 5 folds.
    We will analyse both precision and recall,
    but the precision will be used to choose and refit with the best param.

    Input:
        - model: estimator object that we want to use in the GridSearchCV.
```

```
        - param_grid: dict or list of dictionaries with parameters to test in␣
↪GridSearchCV.
        - X_train: array-like of shape (n_samples, n_features), used to fit.
        - y_train: array-like of shape (n_samples,) with the target, used to␣
↪fit.

    Output:
        - clf: the fitted GridSearchCV with info such as cv_results_,␣
↪best_score_, best_params_ ."
    """
    # Instatiating a GridSearchCV object
    clf = GridSearchCV(estimator=model,
                       param_grid=param_grid,
                       scoring=['precision_macro', 'recall_macro'],
                       refit='precision_macro',
                       cv=5
                       #, verbose=3 # uncomment it if you want to check the␣
↪score for each fold
                      )

    # Fitting the GridSearchCV object
    clf.fit(X_train, y_train)

    # Printing the best configuration according to GridSearchCV considering␣
↪just the precision
    # (not necessary the one we will choose)
    print(f"\nBest parameters set found according to precision ({clf.
↪best_score_:.3f}): {clf.best_params_}.")

    # Printing the mean metrics and execution time
    print("\nGrid scores:")
    means_precision = clf.cv_results_['mean_test_precision_macro']
    means_recall = clf.cv_results_['mean_test_recall_macro']
    means_fit_time = clf.cv_results_['mean_fit_time']
    for mean_p, mean_r, mean_t, params in zip(means_precision, means_recall,␣
↪means_fit_time, clf.cv_results_['params']):
        print(f"mean: precision ({mean_p:.3f}), recall ({mean_r:.3f}) and fit␣
↪time ({mean_t:.3f}s) for {params}")
    print()

    return clf
```

4.1 Logistic Regression

For the Logistic Regression, we will use the LogisticRegression( ) classifier. To select the best configuration, we will vary some hyperparameters. Previous smaller tests were made varying other parameters (such as dual, tol, C, max_iter), but after all, since the GridSearchCV test all the

combinations, and due to long execution time, it was decided to keep the default values for those and focus on deciding two hyperparameters that seemed to stand out in the importance: the penalty and the solver.

The solver can be {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}. It is the algorithm to use in the optimization problem. According to its documentation, for small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones. For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes. So, it was decided to test the {'lbfgs', 'newton-cg', 'sag', 'saga'} options in the solver parameter.

The penalty can be {'l1', 'l2', 'elasticnet', 'none'}. It is used to specify the norm used in the penalization. According to its documentation, 'newton-cg', 'lbfgs', 'sag' and 'saga' handle 'l2' or no penalty, and 'saga' also supports 'elasticnet' and 'l1' penalty. If 'none', no regularization is applied. So, it was decided to test the {'l2', 'none'} options in the penalty parameter.

More details about LogisticRegression can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)

[12]:
```
# Defining the param_grid varying penalty and solver, fixing random_state=42
# and letting the other parameters with its default value
param_grid = [{'penalty': ['l2', 'none'],
               'solver' : ['lbfgs', 'newton-cg', 'sag', 'saga'],
               'random_state': [42]
              }]

# Calling the method to test all the combinations of parameters using cross␣
 ↪validation
logistic = train_with_grid_search_cv(LogisticRegression(),
                                     param_grid,
                                     X_train_set,
                                     y_train_set)
```

```
Best parameters set found according to precision (0.908): {'penalty': 'none',
'random_state': 42, 'solver': 'saga'}.

Grid scores:
mean: precision (0.907), recall (0.906) and fit time (2.547s) for {'penalty':
'l2', 'random_state': 42, 'solver': 'lbfgs'}
mean: precision (0.906), recall (0.906) and fit time (26.182s) for {'penalty':
'l2', 'random_state': 42, 'solver': 'newton-cg'}
mean: precision (0.906), recall (0.906) and fit time (18.217s) for {'penalty':
'l2', 'random_state': 42, 'solver': 'sag'}
mean: precision (0.906), recall (0.906) and fit time (25.146s) for {'penalty':
'l2', 'random_state': 42, 'solver': 'saga'}
mean: precision (0.908), recall (0.907) and fit time (2.665s) for {'penalty':
'none', 'random_state': 42, 'solver': 'lbfgs'}
mean: precision (0.907), recall (0.907) and fit time (502.247s) for {'penalty':
'none', 'random_state': 42, 'solver': 'newton-cg'}
```

```
mean: precision (0.908), recall (0.908) and fit time (18.330s) for {'penalty':
'none', 'random_state': 42, 'solver': 'sag'}
mean: precision (0.908), recall (0.908) and fit time (25.556s) for {'penalty':
'none', 'random_state': 42, 'solver': 'saga'}
```

We can see that there is not much change in the precision and recall, and there are some parameters that take much longer than the others (example solver newton-cg with no penalty, it took around 8 minutes). Considering all of that, the solver lbfgs with no penalty got almost the same result as the best one, but under than 3 seconds.

So, we will keep the params = {'penalty': 'none', 'solver': 'lbfgs'} for the Logistic Regression.

[13]:
```
# Retraning Logistic Regression on the X_train_set with the best params
clf_logistic = LogisticRegression(penalty= 'none',
                                  solver= 'lbfgs',
                                  random_state= 42)
clf_logistic.fit(X_train_set, y_train_set)
```

[13]:
```
LogisticRegression(penalty='none', random_state=42)
```

4.2 Neural Network Model

For the Neural Network, we will use the MLPClassifier( ) classifier. To select the best configuration, we will vary some hyperparameters. Previous smaller tests were made varying other parameters (such as solver, alpha, learning_rate, max_iter), but after all, since the GridSearchCV test all the combinations, and due to long execution time, it was decided to keep the default values for those and focus on deciding two hyperparameters that seemed to stand out in the importance: the hidden_layer_sizes and the activation.

The hidden_layer_sizes is a tuple, with length = n_layers - 2. According to its documentation, the ith element represents the number of neurons in the ith hidden layer. So, it was decided to test the {(10,), (10,10), (100,), (100,100), (500,), (500,500)} options in the hidden_layer_sizes parameter.

The activation can be {'identity', 'logistic', 'tanh', 'relu'}. It is the activation function for the hidden layer. According to its documentation: - 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x. - 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)). - 'tanh', the hyperbolic tan function, returns f(x) = tanh(x). - 'relu', the rectified linear unit function, returns f(x) = max(0, x).

So, it was decided to test all of them, {'identity', 'logistic', 'tanh', 'relu'} options, in the activation parameter.

More details about MLPClassifier can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html)

[14]:
```
# Defining the param_grid varying hidden_layer_sizes and activation, fixing␣
 ↪random_state=42
# and letting the other parameters with its default value
param_grid = [{'hidden_layer_sizes': [(10,), (10,10), (100,), (100,100),␣
 ↪(500,), (500,500)],
```

```
                'activation': ['identity', 'logistic', 'tanh', 'relu'],
                'random_state': [42]
               }]

# Calling the method to test all the combinations of parameters using cross␣
 ↪validation
mlp = train_with_grid_search_cv(MLPClassifier(),
                                param_grid,
                                X_train_set,
                                y_train_set)
```

```
Best parameters set found according to precision (0.975): {'activation': 'tanh',
'hidden_layer_sizes': (500, 500), 'random_state': 42}.

Grid scores:
mean: precision (0.908), recall (0.908) and fit time (9.158s) for {'activation':
'identity', 'hidden_layer_sizes': (10,), 'random_state': 42}
mean: precision (0.908), recall (0.907) and fit time (12.479s) for
{'activation': 'identity', 'hidden_layer_sizes': (10, 10), 'random_state': 42}
mean: precision (0.908), recall (0.907) and fit time (8.975s) for {'activation':
'identity', 'hidden_layer_sizes': (100,), 'random_state': 42}
mean: precision (0.906), recall (0.905) and fit time (11.124s) for
{'activation': 'identity', 'hidden_layer_sizes': (100, 100), 'random_state': 42}
mean: precision (0.906), recall (0.906) and fit time (17.334s) for
{'activation': 'identity', 'hidden_layer_sizes': (500,), 'random_state': 42}
mean: precision (0.906), recall (0.905) and fit time (88.602s) for
{'activation': 'identity', 'hidden_layer_sizes': (500, 500), 'random_state': 42}
mean: precision (0.917), recall (0.917) and fit time (17.098s) for
{'activation': 'logistic', 'hidden_layer_sizes': (10,), 'random_state': 42}
mean: precision (0.919), recall (0.919) and fit time (20.385s) for
{'activation': 'logistic', 'hidden_layer_sizes': (10, 10), 'random_state': 42}
mean: precision (0.967), recall (0.967) and fit time (31.484s) for
{'activation': 'logistic', 'hidden_layer_sizes': (100,), 'random_state': 42}
mean: precision (0.967), recall (0.966) and fit time (36.292s) for
{'activation': 'logistic', 'hidden_layer_sizes': (100, 100), 'random_state': 42}
mean: precision (0.973), recall (0.973) and fit time (81.338s) for
{'activation': 'logistic', 'hidden_layer_sizes': (500,), 'random_state': 42}
mean: precision (0.973), recall (0.973) and fit time (123.951s) for
{'activation': 'logistic', 'hidden_layer_sizes': (500, 500), 'random_state': 42}
mean: precision (0.927), recall (0.927) and fit time (15.988s) for
{'activation': 'tanh', 'hidden_layer_sizes': (10,), 'random_state': 42}
mean: precision (0.928), recall (0.928) and fit time (20.205s) for
{'activation': 'tanh', 'hidden_layer_sizes': (10, 10), 'random_state': 42}
mean: precision (0.967), recall (0.967) and fit time (21.456s) for
{'activation': 'tanh', 'hidden_layer_sizes': (100,), 'random_state': 42}
mean: precision (0.969), recall (0.969) and fit time (19.305s) for
```

```
{'activation': 'tanh', 'hidden_layer_sizes': (100, 100), 'random_state': 42}
mean: precision (0.973), recall (0.972) and fit time (50.652s) for
{'activation': 'tanh', 'hidden_layer_sizes': (500,), 'random_state': 42}
mean: precision (0.975), recall (0.975) and fit time (52.290s) for
{'activation': 'tanh', 'hidden_layer_sizes': (500, 500), 'random_state': 42}
mean: precision (0.927), recall (0.926) and fit time (16.536s) for
{'activation': 'relu', 'hidden_layer_sizes': (10,), 'random_state': 42}
mean: precision (0.924), recall (0.924) and fit time (19.831s) for
{'activation': 'relu', 'hidden_layer_sizes': (10, 10), 'random_state': 42}
mean: precision (0.966), recall (0.965) and fit time (22.454s) for
{'activation': 'relu', 'hidden_layer_sizes': (100,), 'random_state': 42}
mean: precision (0.968), recall (0.968) and fit time (16.417s) for
{'activation': 'relu', 'hidden_layer_sizes': (100, 100), 'random_state': 42}
mean: precision (0.975), recall (0.974) and fit time (32.416s) for
{'activation': 'relu', 'hidden_layer_sizes': (500,), 'random_state': 42}
mean: precision (0.975), recall (0.974) and fit time (41.295s) for
{'activation': 'relu', 'hidden_layer_sizes': (500, 500), 'random_state': 42}
```

We can see that the activation 'identity' was the worst, no matter the hidden layers. The other three were very similar, and the 'tanh' and 'relu' were the best two. There were a few parameters that got 0.975 precision and 0.97 recall, but considering the execution time, the 'relu' with hidden_layer_sizes (500,) was the best one.

So, we will keep the params = {'activation': 'relu', 'hidden_layer_sizes': (500,)} for the Neural Network.

[15]:
```
# Retraning MLPClassifier on the X_train_set with the best params
clf_neural_network = MLPClassifier(activation='relu',
                                   hidden_layer_sizes=(500,),
                                   random_state=42)
clf_neural_network.fit(X_train_set, y_train_set)
```

[15]:
```
MLPClassifier(hidden_layer_sizes=(500,), random_state=42)
```

4.3 Support Vector Machines Model

For the SVM, we will use the LinearSVC( ) classifier. To select the best configuration, we will vary some hyperparameters. Previous smaller tests were made varying other parameters (such as ....), but after all, since the GridSearchCV test all the combinations, and due to long execution time, it was decided to keep the default values for those and focus on deciding two hyperparameters that seemed to stand out in the importance: the loss and the multi_class.

The loss can be {'hinge', 'squared_hinge'}. It specifies the loss function. According to its documentation, 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss. The combination of penalty='l1' and loss='hinge' is not supported, that's why we fixed the penalty in its default 'l2'. So, it was decided to test all of them, {'hinge', 'squared_hinge'} options, in the loss parameter.

The multi_class can be {'ovr', 'crammer_singer'}. It determines the multi-class strategy if y

contains more than two classes. According to its documentation, “ovr” trains n_classes one-vs-rest classifiers, while “crammer_singer” optimizes a joint objective over all classes. If “crammer_singer” is chosen, the options loss, penalty and dual will be ignored. So, it was decided to test all of them, {‘ovr’, ‘crammer_singer’} options, in the multi_class parameter.

More details about LinearSVC can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html)

[16]:
```
# Defining the param_grid varying loss and multi_class, fixing random_state=42
# and letting the other parameters with its default value
param_grid = [{'loss': ['hinge', 'squared_hinge'],
               'multi_class': ['ovr', 'crammer_singer'],
               'random_state': [42]
              }]

# Calling the method to test all the combinations of parameters using cross␣
 ↪validation
svc = train_with_grid_search_cv(LinearSVC(),
                                param_grid,
                                X_train_set,
                                y_train_set)
```

```
Best parameters set found according to precision (0.911): {'loss': 'hinge',
'multi_class': 'crammer_singer', 'random_state': 42}.

Grid scores:
mean: precision (0.901), recall (0.901) and fit time (1.986s) for {'loss':
'hinge', 'multi_class': 'ovr', 'random_state': 42}
mean: precision (0.911), recall (0.910) and fit time (14.309s) for {'loss':
'hinge', 'multi_class': 'crammer_singer', 'random_state': 42}
mean: precision (0.899), recall (0.899) and fit time (13.640s) for {'loss':
'squared_hinge', 'multi_class': 'ovr', 'random_state': 42}
mean: precision (0.911), recall (0.910) and fit time (15.140s) for {'loss':
'squared_hinge', 'multi_class': 'crammer_singer', 'random_state': 42}
```

We can see that the multi_class ‘ovr’ did a little worse than the ‘crammer_singer’ on both loss functions. Also, if ‘crammer_singer’ is chosen on the multi_class parameter, the option loss is ignored, that’s why it got the same precision (0.911) and recall (0.910) regardless of the loss function and with execution time very close to each other.

So, we will keep the params = {‘multi_class’: ‘crammer_singer’} for the SVM.

[17]:
```
# Retraning LinearSVC on the X_train_set with the best params
clf_svm = LinearSVC(multi_class='crammer_singer',
                    random_state=42)
clf_svm.fit(X_train_set, y_train_set)
```

[17]:
```
LinearSVC(multi_class='crammer_singer', random_state=42)
```

5 Choosing a Final Model

In this step, we will use the X_val and y_val to evaluate the three models selected in the previous step and select the best one.

As we used the precision to decide the best model in the previous step, we will keep doing it here. Although, to enrich the analysis, we will also check some other metrics.

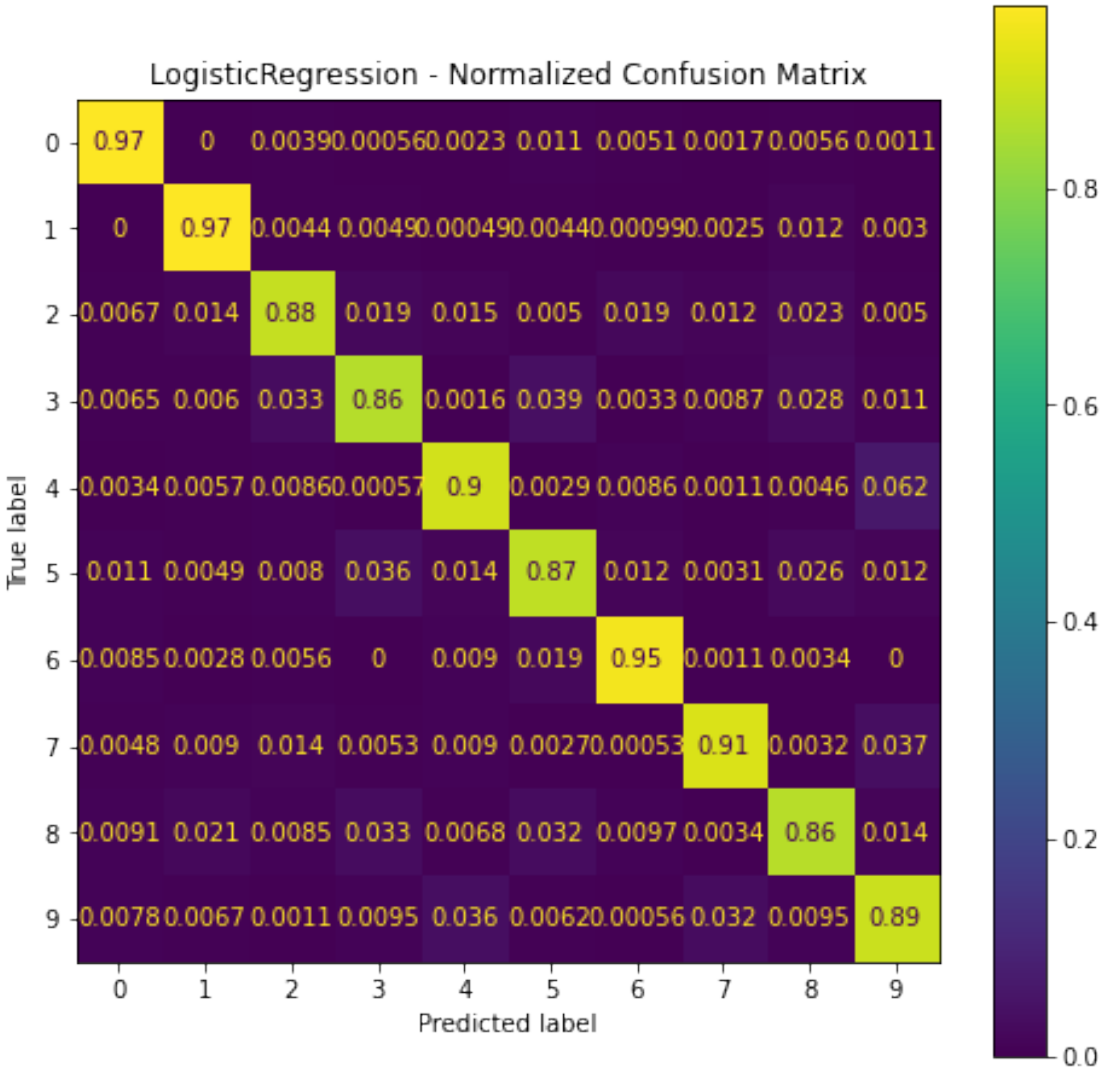
The confusion matrix is used to evaluate the quality of the output of a classifier. According to its documentation, the diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions. More details about plot_confusion_matrix can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html).

In addition to the confusion matrix, we will print a classification report, with information about precision, recall and f1-score, separated by class and overall. More details about classification_report can be found in the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html).

[27]:
```
def analyse_metrics(model_name, model, X_true, y_true, y_pred):
    """
    Method to analyse the metrics, plotting the confusion matrix and printing␣
↪the classification report.

    Input:
        - model_name: a string with the model name to use in the plot title.
        - model: estimator instance, fitted classifier to use in␣
↪plot_confusion_matrix.
        - X_true: array-like of shape (n_samples, n_features), used as input␣
↪values in plot_confusion_matrix.
        - y_true: 1d array-like of shape (n_samples,), ground truth (correct)␣
↪target values, used in plot_confusion_matrix and classification_report.
        - y_pred: 1d array-like of shape (n_samples,), estimated targets as␣
↪returned by a classifier, used in classification_report.

    Output:
        None
    """

    # Plot confusion matrix
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_title(f'{model_name} - Normalized Confusion Matrix')
    disp = plot_confusion_matrix(model, X_true, y_true, ax=ax, normalize='true')
    plt.show()
```

```
    # Print classification report
    print(classification_report(y_true, y_pred))
```

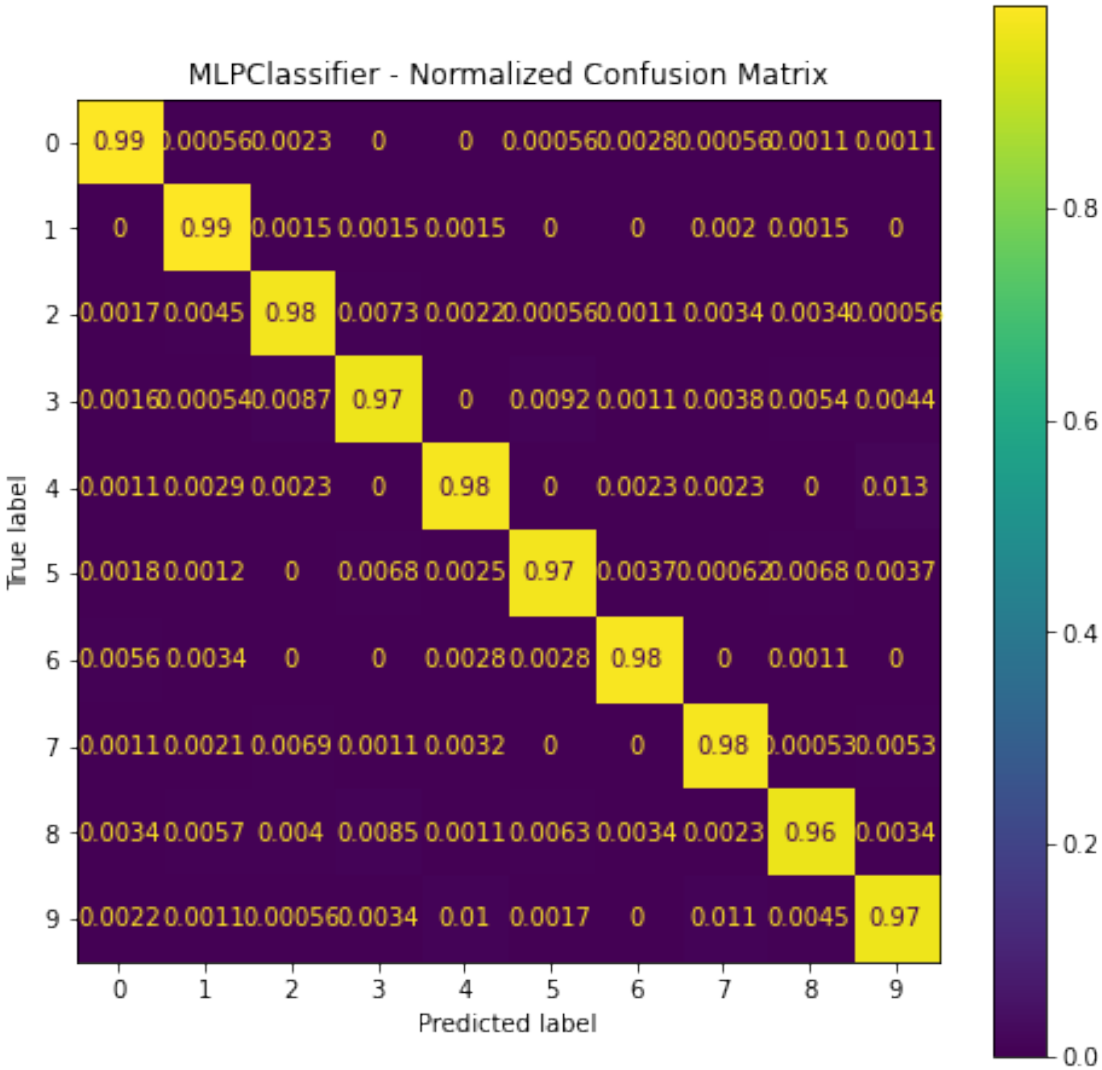
[19]:
```
# Evaluating Logistic Regression
y_pred = clf_logistic.predict(X_val)

# Analysing metrics
analyse_metrics(model_name= 'LogisticRegression',
                model=clf_logistic,
                X_true=X_val,
                y_true=y_val,
                y_pred=y_pred)
```




```
              precision    recall  f1-score   support
```

```
              0       0.94      0.97      0.96      1777
              1       0.94      0.97      0.95      2023
              2       0.91      0.88      0.89      1787
              3       0.89      0.86      0.88      1839
              4       0.91      0.90      0.90      1753
              5       0.87      0.87      0.87      1626
              6       0.94      0.95      0.95      1775
              7       0.94      0.91      0.92      1880
              8       0.88      0.86      0.87      1755
              9       0.86      0.89      0.87      1785

    accuracy                            0.91     18000
   macro avg       0.91      0.91      0.91     18000
weighted avg       0.91      0.91      0.91     18000
```

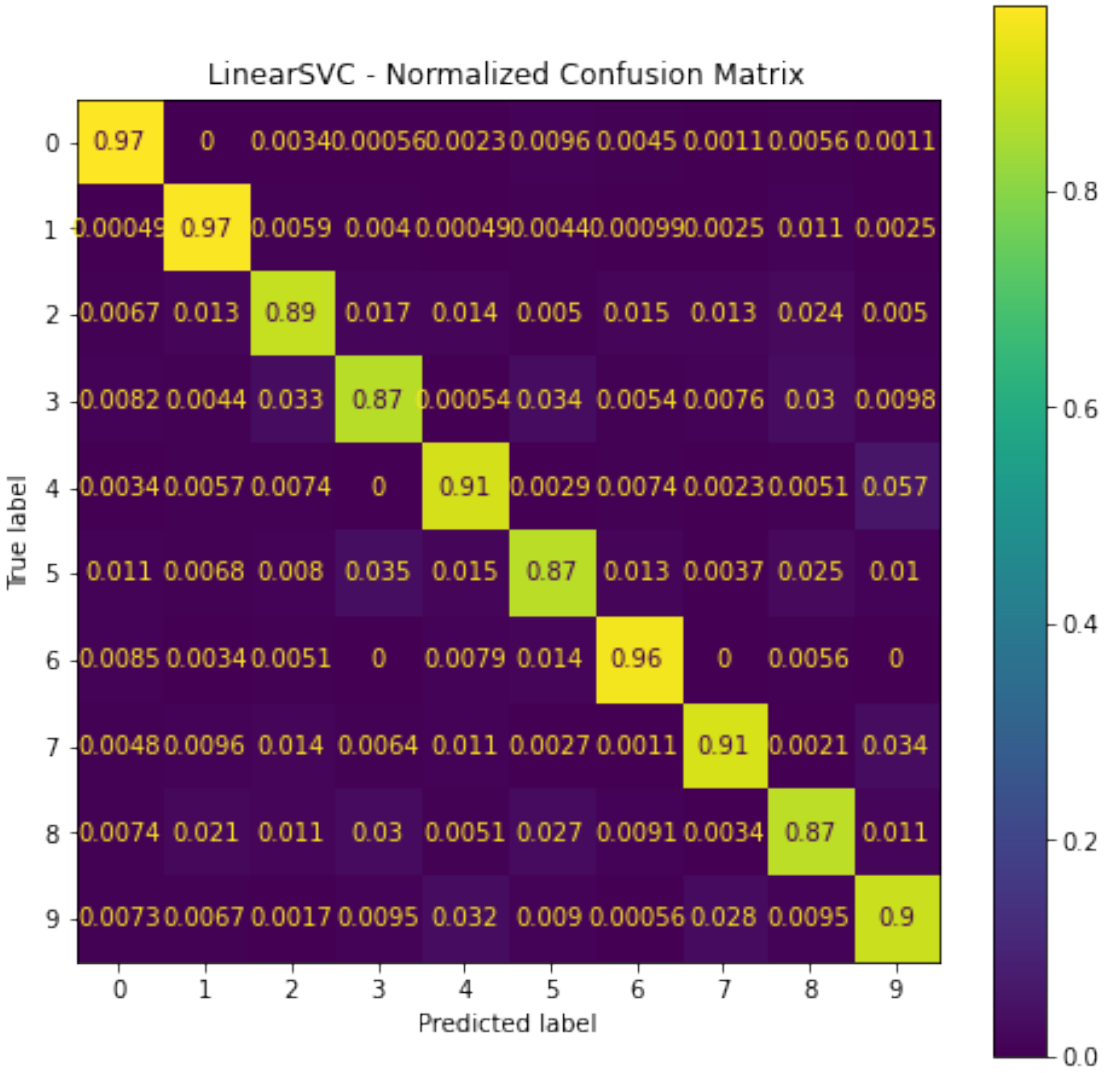
[20]:
```python
# Evaluating Neural Network
y_pred = clf_neural_network.predict(X_val)

# Analysing metrics
analyse_metrics(model_name= 'MLPClassifier',
                model=clf_neural_network,
                X_true=X_val,
                y_true=y_val,
                y_pred=y_pred)
```



	precision	recall	f1-score	support
0	0.98	0.99	0.99	1777
1	0.98	0.99	0.99	2023
2	0.97	0.98	0.97	1787
3	0.97	0.97	0.97	1839
4	0.98	0.98	0.98	1753
5	0.98	0.97	0.97	1626
6	0.99	0.98	0.99	1775
7	0.98	0.98	0.98	1880
8	0.98	0.96	0.97	1755
9	0.97	0.97	0.97	1785
accuracy			0.98	18000

```
   macro avg       0.98      0.98      0.98     18000
weighted avg       0.98      0.98      0.98     18000
```

[21]:
```
# Evaluating SVM
y_pred = clf_svm.predict(X_val)

# Analysing metrics
analyse_metrics(model_name= 'LinearSVC',
                model=clf_svm,
                X_true=X_val,
                y_true=y_val,
                y_pred=y_pred)
```



	precision	recall	f1-score	support
0	0.94	0.97	0.96	1777
1	0.94	0.97	0.95	2023
2	0.91	0.89	0.90	1787
3	0.90	0.87	0.88	1839
4	0.91	0.91	0.91	1753
5	0.88	0.87	0.88	1626
6	0.94	0.96	0.95	1775
7	0.94	0.91	0.93	1880
8	0.88	0.87	0.88	1755
9	0.87	0.90	0.88	1785
accuracy			0.91	18000
macro avg	0.91	0.91	0.91	18000
weighted avg	0.91	0.91	0.91	18000

We can see in the confusion matrices that all of the models performed very well in all classes. On Logistic and SVM, the classes 3, 5 and 8 were a little confused with each other, as well as the classes 4 and 9 and the classes 7 and 9. On Neural Network, we can see lots of zeros in the off-diagonal elements, and others very close to zero, which means it has not confused almost any class.

Evaluating all three models, we can see that the Logistic Regression and the SVM are very similar, considering all the metrics (precision, recall and f1-score) with 0.91 in all of them. On the other hand, we can see that the Neural Network performed much better, with all the three metrics as 0.98.

Moreover, we can see that all of the analysis were not very different in the training and validation set, obtening similar metrics.

Therefore, the Neural Network (MLPClassifier) is the best model and the chosen one.

6 Error Estimation

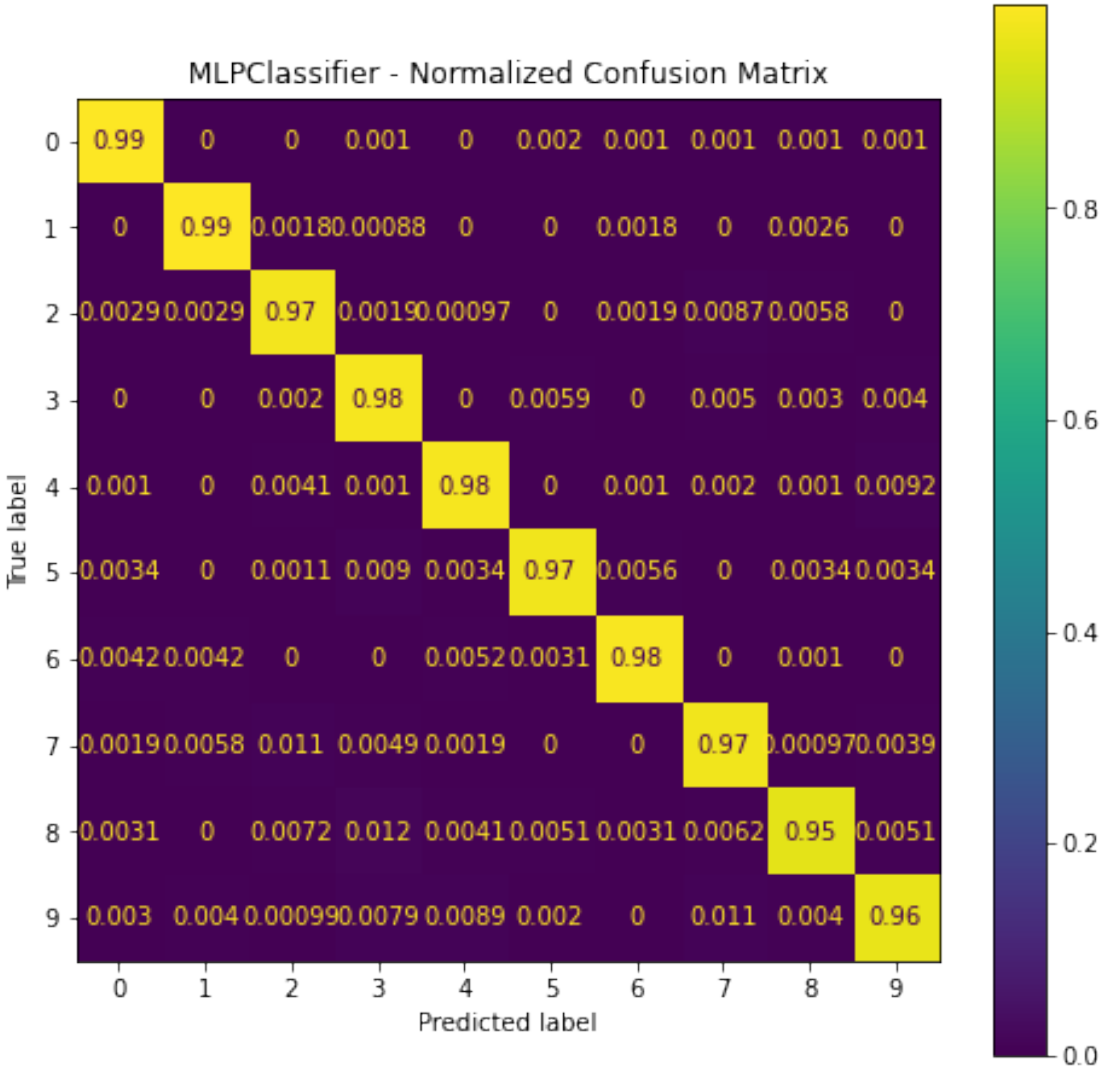
Now that we have chosen the final model, we can compute an estimate of its expected performance using X_test and y_test.

The model we will evaluate here on the test set is the one that presented the best validation error with respect to X_val and y_val, so the Neural Network (MLPClassifier).

Similarly to the previous section, we will analyse the confusion matrix and the classification report, with precision, recall and f1-score.

[22]:
```
# Evaluating Neural Network
y_pred = clf_neural_network.predict(X_test)

# Analyse metrics
analyse_metrics(model_name= 'MLPClassifier',
                model=clf_neural_network,
```

```
                X_true=X_test,
                y_true=y_test,
                y_pred=y_pred)
```



	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	0.99	0.99	0.99	1135
2	0.97	0.97	0.97	1032
3	0.96	0.98	0.97	1010
4	0.98	0.98	0.98	982
5	0.98	0.97	0.98	892
6	0.99	0.98	0.98	958
7	0.97	0.97	0.97	1028

```
           8       0.98      0.95      0.96       974
           9       0.97      0.96      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

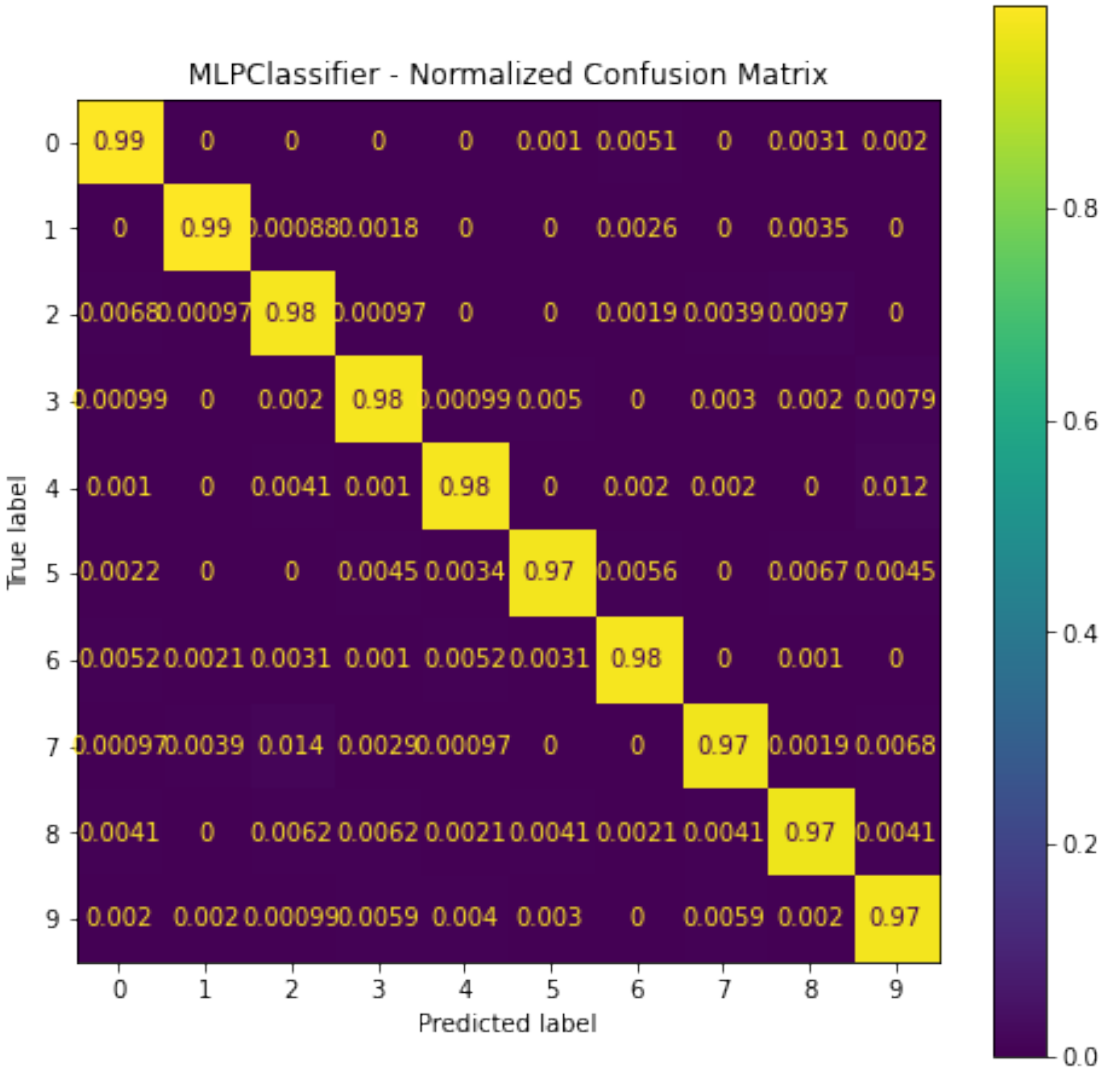
We can see that the Neural Network also performed very well on the test set, with all of the metrics as 0.98. Moreover, we can see that all of the analysis were not different in the validation and testing set, obtening similar metrics.

As a final analysis, we will retrain the model with the whole training set (X_train), the union of X_train_set and X_val, and compare the results.

[23]:
```
# Retraning MLPClassifier on the X_train
clf_neural_network = MLPClassifier(activation='relu',
                                   hidden_layer_sizes=(500,),
                                   random_state=42)
clf_neural_network.fit(X_train, y_train)
```

[23]:
```
MLPClassifier(hidden_layer_sizes=(500,), random_state=42)
```

[24]:
```
# Evaluating Neural Network
y_pred = clf_neural_network.predict(X_test)

# Analyse metrics
analyse_metrics(model_name= 'MLPClassifier',
                model=clf_neural_network,
                X_true=X_test,
                y_true=y_test,
                y_pred=y_pred)
```



	precision	recall	f1-score	support
0	0.98	0.99	0.98	980
1	0.99	0.99	0.99	1135
2	0.97	0.98	0.97	1032
3	0.98	0.98	0.98	1010
4	0.98	0.98	0.98	982
5	0.98	0.97	0.98	892
6	0.98	0.98	0.98	958
7	0.98	0.97	0.98	1028
8	0.97	0.97	0.97	974
9	0.96	0.97	0.97	1009
accuracy			0.98	10000

```
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

We can see that it is very similar, with all of the metrics as 0.98, so it didn't make much difference to retrain with the whole set.

7 Final comments

In this section we will make an overview of the implemented experiments, results and relevant comments.

All of the analisys was made using the MNIST dataset. At the beggining, we loaded it from tensorflow.keras.datasets, already separed in train and test, and we could see that it was balanced in both. Then, we started with a preprocessing step, reducing the image size to its half, converting the dataset to float and normalizing the intensities to the interval [0,1] in both train and test sets. Finishing the initial step, we separated the training set into two parts in a stratified way, 70% train and 30% validation, and checked if it was indeed stratified and all of the datasets (X_train_set, X_val and X_test) were balanced. Also, we established the use of random_state=42 in every necessary place to allow reproduction of the results.

Then, we started the training, evaluating and selecting models section to have three selected models: a logistic regression model, a neural network model, and a SVM model. We used the X_train_set and y_train_set to train different models, varying the hyperparameters, and chose the best configuration for each one. For that, we used grid search techniques and cross validation with 5 folds. Previous smaller tests were made varying other parameters, but after all, since the GridSearchCV tests all parameter combinations, and due to long execution time, it was decided to keep the default values for those and focus on deciding two hyperparameters that seemed to stand out in the importance for each model. Since the cross validation splits the data into k folds and use k-1 folds to train and 1 fold to test, we didn't need to split our data one more time. In order to choose the best configuration, we analysed the precision, the recall and the execution time. The precision is the proportion of predicted Positives that is truly Positive (TP/(TP+FP)) and recall is the proportion of actual Positives that is correctly classified (TP/(TP+FN)).

For the Logistic Regression, we used the LogisticRegression( ) classifier. Here, we chose to focus on deciding two parameters: the penalty, used to specify the norm used in the penalization (with {'l2', 'none'} options), and the solver, algorithm to use in the optimization problem (with {'lbfgs', 'newton-cg', 'sag', 'saga'} options). As result, we could see that there is not much change in the precision and recall, and there are some parameters that took much longer than the others, so, considering all of that, the solver lbfgs with no penalty got almost the same result as the best one, but under than 3 seconds. So, we kept the params = {'penalty': 'none', 'solver': 'lbfgs'}.

For the Neural Network, we used the MLPClassifier( ) classifier. Here, we chose to focus on deciding two parameters: the hidden_layer_sizes, a tuple with length = n_layers - 2, in which the ith element represents the number of neurons in the ith hidden layer (with {(10,), (10,10), (100,), (100,100), (500,), (500,500)} options), and the activation, the activation function for the hidden layer (with {'identity', 'logistic', 'tanh', 'relu'} options). As result, we could see that the activation 'identity' was the worst, no matter the hidden layers. The other three were very similar, and the 'tanh' and 'relu' were the best two. There were a few parameters that got 0.975 precision and 0.97

recall, but considering the execution time, the 'relu' with hidden_layer_sizes (500,) was the best one. So, we kept the params = {'activation': 'relu', 'hidden_layer_sizes': (500,)}.

For the SVM, we used the LinearSVC( ) classifier. Here, we chose to focus on deciding two parameters: the loss, that specifies the loss function (with {'hinge', 'squared_hinge'} options), and the multi_class, that determines the multi-class strategy if y contains more than two classes (with {'ovr', 'crammer_singer'} options). As result, we could see that the multi_class 'ovr' did a little worse than the 'crammer_singer' on both loss functions. Also, if 'crammer_singer' is chosen on the multi_class parameter, the option loss is ignored, that's why it got the same precision (0.911) and recall (0.910) regardless of the loss function and with execution time very close to each other. So, we kept the params = {'multi_class': 'crammer_singer'}.

After selecting the best configuration for each algorithm, we went to the next section to choose the final model. In this step, we used the X_val and y_val to evaluate the three models selected in the previous step and select the best one. Here, we kept analysing the precision, as in the previous steps, but to enrich the analysis, we also checked some other metrics: the confusion matrix, used to evaluate the quality of the output of a classifier, in which the higher the diagonal values the better, indicating many correct predictions; and the classification report, with information about precision, recall and f1-score, separated by class and overall. As result, we could see in the confusion matrices that all of the models performed very well in all classes. On Logistic and SVM, the classes 3, 5 and 8 were a little confused with each other, as well as the classes 4 and 9 and the classes 7 and 9. On Neural Network, we could see lots of zeros in the off-diagonal elements, and others very close to zero, which means it has not confused almost any class. Evaluating all three models, we could see that the Logistic Regression and the SVM were very similar, considering all the metrics (precision, recall and f1-score) with 0.91 in all of them. On the other hand, we could see that the Neural Network performed much better, with all the three metrics as 0.98. Moreover, we could see that all of the analysis were not very different in the training and validation set, obtening similar metrics. Therefore, the Neural Network (MLPClassifier) was the best model and the chosen one as the final model.

Finally, we went to the error estimation section. Here, using the chosen final model, we could compute an estimate of its expected performance using X_test and y_test. Similarly to the previous section, we analysed the confusion matrix and the classification report. We could see that the Neural Network also performed very well on the test set, with all of the metrics as 0.98. As a final analysis, we retrained the model with the whole training set (X_train), the union of X_train_set and X_val, and when we compared the results, we could see that the Neural Network also performed very well on the test set, with all of the metrics as 0.98. Moreover, we could see that all of the analysis were not different in the validation and testing set, retraining with the whole training set or not, obtening similar metrics.

Therefore, we were able to go through all the steps, from data preprocessing to training and selection of the best model, always analysing its performance. It was very interesting to see how each parameter can completely change the functioning of the model, and have the opportunity to put into practice the use of all available tools to make a good analysis. Besides, we could practice dividing the dataset correctly, to be able to complete all the steps without danger of false metrics, and also we could see how these algorithms can take time to run, depending on their parameters.