project

October 9, 2022

0.1 Part 0 - Setup

0.1.1 Section 0.1 - Imports and functions

[1]:
```
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sb
from sklearn.decomposition import PCA
import warnings
warnings.filterwarnings('ignore')
%store -r

def generate_confusion_matrix(y_true, y_pred):
    # visualize the confusion matrix
    ax = plt.subplot()
    c_mat = metrics.confusion_matrix(y_true, y_pred)
    sb.heatmap(c_mat, annot=False, fmt='g', ax=ax)

    ax.set_xlabel('Predicted labels', fontsize=15)
    ax.set_ylabel('True labels', fontsize=15)
    ax.set_title('Confusion Matrix', fontsize=15)
```

0.1.2 Section 0.2 - Fetching datasets

[2]:
```
# Fetch dataset
data_train = pd.read_csv("./EMNIST/emnist-balanced-train.csv", header=None)
data_testing = pd.read_csv("./EMNIST/emnist-balanced-test.csv", header=None)

cols = ['CHAR']
for i in range(1, 785):
    cols.append(str(i))
data_train.columns = cols
data_testing.columns = cols
print(data_train.shape)
```

```
print(data_testing.shape)
```

```
(112800, 785)
(18800, 785)
```

[3]:
```
# Use stratified random sampling to split the testing dataset into test and␣
↪validation datasets with a 1:1 ratio
test_set = pd.DataFrame(columns=cols)
val_set = pd.DataFrame(columns=cols)
# 1. Group by label
groups = data_testing.groupby("CHAR")
# 2. Distribute evenly
for i in groups.groups:
    group_df = groups.get_group(i)
    # Shuffle data
    shuffled = group_df.sample(frac=1, random_state=0)
    result = np.array_split(shuffled, 2)
    df1 = pd.DataFrame(result[0])
    test_set = pd.concat([test_set, df1], ignore_index=True)
    df2 = pd.DataFrame(result[1])
    val_set = pd.concat([val_set, df2], ignore_index=True)
# 3. Randomize sets once again
val_set = val_set.sample(frac=1, random_state=0)
test_set = test_set.sample(frac=1, random_state=0)
```

[4]:
```
X_train = np.array(data_train.iloc[:,1:].values)
y_train = np.array(data_train["CHAR"].values)
X_val = np.array(val_set.iloc[:,1:].values).astype(np.int64)
y_val = np.array(val_set["CHAR"].values).astype(np.int64)
X_test = np.array(test_set.iloc[:,1:].values).astype(np.int64)
y_test = np.array(test_set["CHAR"].values).astype(np.int64)
print(X_train.shape)
print(y_train.shape)
print(X_val.shape)
print(y_val.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(112800, 784)
(112800,)
(9400, 784)
(9400,)
(9400, 784)
(9400,)
```

0.2 Part 1 - PCA

0.2.1 Section 1.1 - Optimizing the number of features for PCA

[9]:
```
N = 784
pca = PCA(n_components=N)
X_train_reduced = pca.fit_transform(X_train)
```

[12]:
```
# Checking the slope from 784 features to 1 feature
points = list(enumerate(np.cumsum(pca.explained_variance_ratio_).tolist(),␣
 ↪start = 1))[::-1]
# Calculate instantaneous slope (pseudo-derivative)
def getSlope(curr: tuple, next: tuple):
    x1 = curr[0]
    x2 = next[0]
    y1 = curr[1]
    y2 = next[1]
    return (y2 - y1)/(x2 - x1)
scores = []
i = 0
while i < len(points) - 1:
    slope = getSlope(points[i], points[i + 1])
    stop = points[i]
    size_reduc = (1 - stop[0]/784)
    info_ret = stop[1]
    # Scoring function - For maximum feature set reduction and maximum␣
 ↪information retained
    # Feature size reduction is prioritized
    score = (size_reduc**2) * info_ret
    scores.append([stop[0], score, size_reduc, info_ret])
    i += 1
```

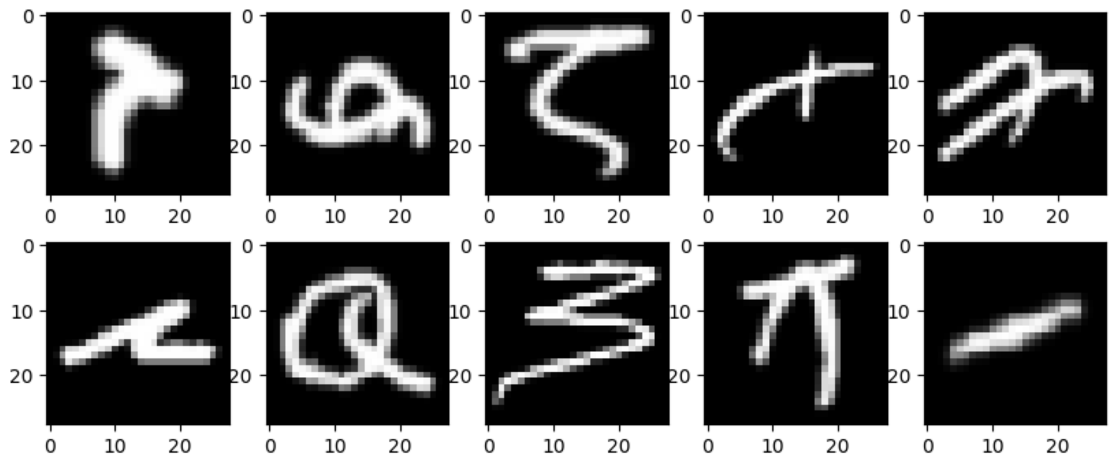
[13]:
```
best = max(scores, key=lambda x: x[1])
print("Best number of features: " + str(best[0]))
print("Score: " + str(round(best[1] * 100, 2)))
print(f"Feature size reduction: {best[2] * 100:.2f}%")
print(f"Cumulative Variance Ratio: {best[3] * 100:.2f}%")
# Best number of features: 82
# Score: 82.56
# Feature size reduction: 89.54%
# Cumulative Variance Ratio: 92.20%
# Best number of features: 61
# Score: 75.29
# Feature size reduction: 92.22%
# Cumulative Variance Ratio: 88.53%
```

```
Best number of features: 61
Score: 75.29
```

```
Feature size reduction: 92.22%
Cumulative Variance Ratio: 88.53%
```

0.2.2 Section 1.2 - PCA Modelling

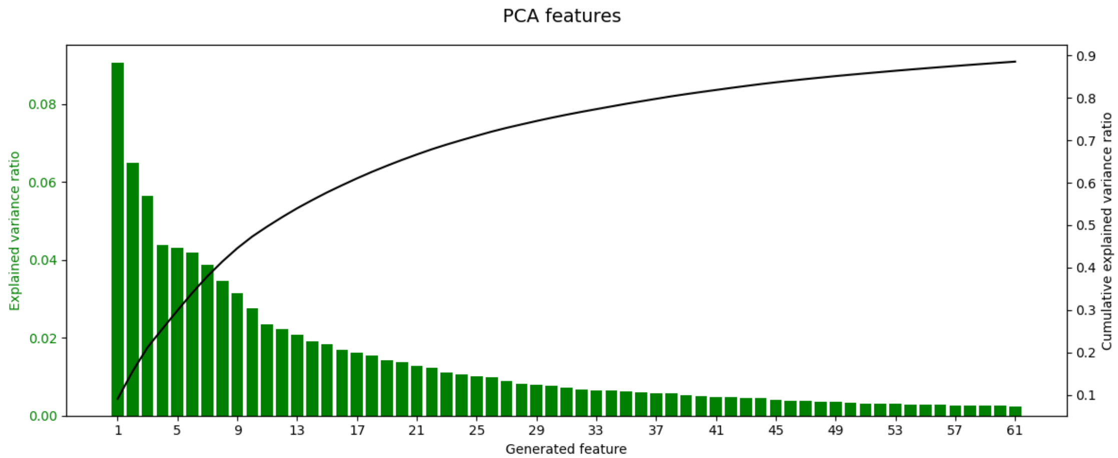
[17]:
```python
# Visualize what the features look like
fig,axes = plt.subplots(2,5,figsize=(10,4))
for i,ax in enumerate(axes.flat):
    ax.imshow(X_train[i].reshape([28,28]), cmap='gray')
```



[14]:
```python
#Using PCA
N = 61 # As determined in part 1.1

pca = PCA(n_components=N)
X_train_reduced = pca.fit_transform(X_train)
```

[15]:
```python
fig, ax1 = plt.subplots(figsize=(12, 5))
fig.suptitle("PCA features", fontsize=14)
color = 'tab:blue'
ax1.bar(1+np.arange(N), pca.explained_variance_ratio_, color="green")
ax1.set_xticks(1+np.arange(N, step=4))
ax1.tick_params(axis='y', labelcolor="green")
ax1.set_ylabel("Explained variance ratio", color="green")
ax1.set_xlabel("Generated feature")

ax2 = ax1.twinx()
color = 'tab:red'
ax2.tick_params(axis='y', labelcolor="black")
ax2.plot(1+np.arange(N), np.cumsum(pca.explained_variance_ratio_),␣
 ↪color="black")
ax2.set_ylabel("Cumulative explained variance ratio", color="black")
```

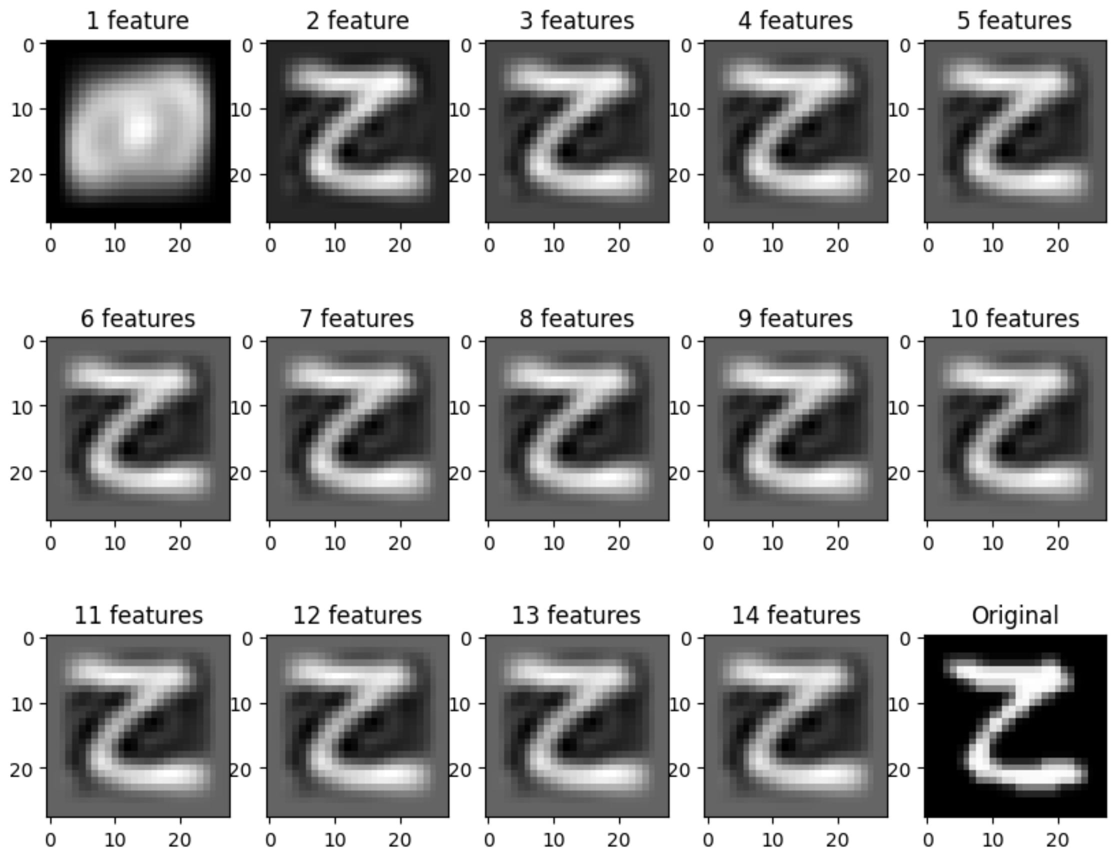
```
fig.tight_layout()
plt.show()
```



[16]:
```
# Visualize the effect of PCA components
fig,axes = plt.subplots(3,5,figsize=(10,8))
fig.suptitle("An image of a 'Z' with varying number of PCA components",␣
 ↪fontsize=14, color="black")
for i,ax in enumerate(axes.flat):
    X_nfeatures = X_train_reduced[20]*[i]
    im = pca.inverse_transform(X_nfeatures).reshape(28, 28)
    ax.set_title(f"{(i+1)} feature{'s' if i>1 else ''}", color="black")
    ax.imshow(im, cmap='gray')
ax.set_title("Original")
ax.imshow(X_train[20].reshape([28,28]), cmap='gray')
```

[16]: <matplotlib.image.AxesImage at 0x7fcdc2d20b80>

An image of a 'Z' with varying number of PCA components



[17]:
```
# PCA used here
clfPCA = LogisticRegression(solver='sag')
X_train_reduced = pca.fit_transform(X_train)

clfPCA.fit(X_train_reduced, y_train)
```

[17]:
```
LogisticRegression(solver='sag')
```

[18]:
```
X_val_reduced = pca.fit_transform(X_val)
```

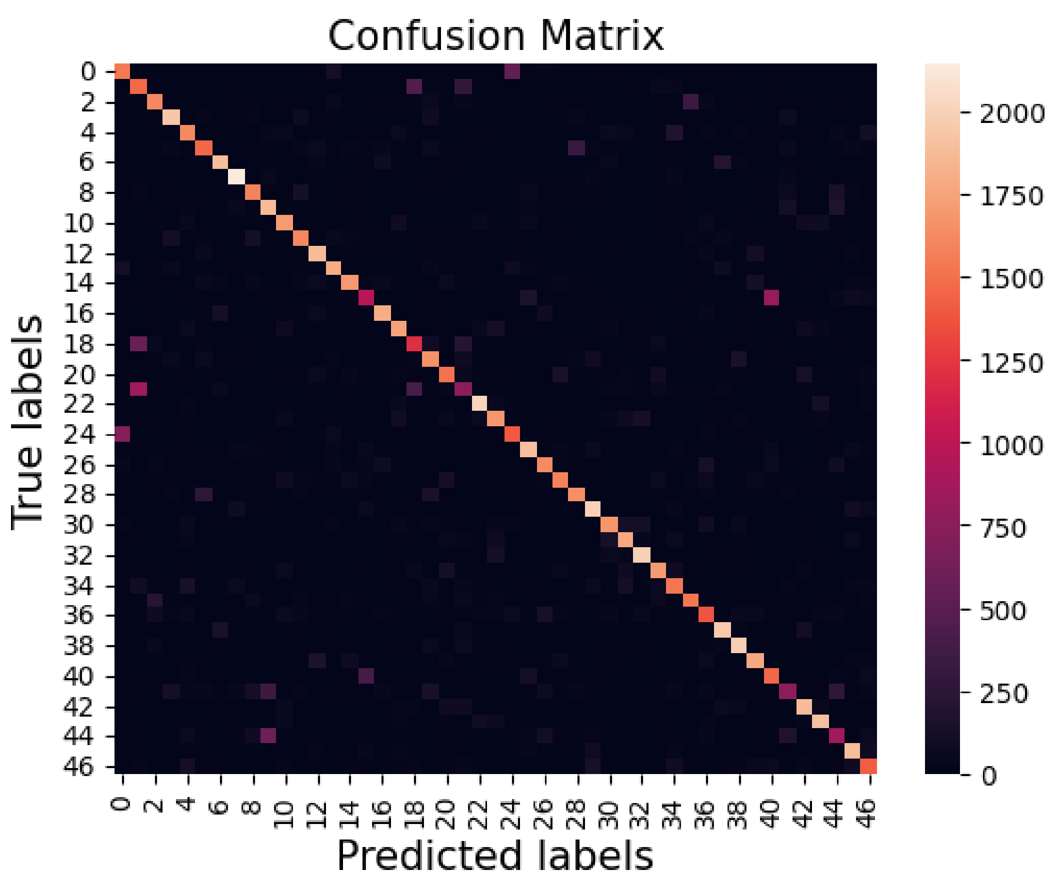
0.2.3 Section 1.3 - Evaluation

[19]:
```
y_pred_train = clfPCA.predict(X_train_reduced)
acc_train = metrics.accuracy_score(y_train, y_pred_train)
loss_train = metrics.log_loss(y_train, clfPCA.
 ↪predict_log_proba(X_train_reduced))
```

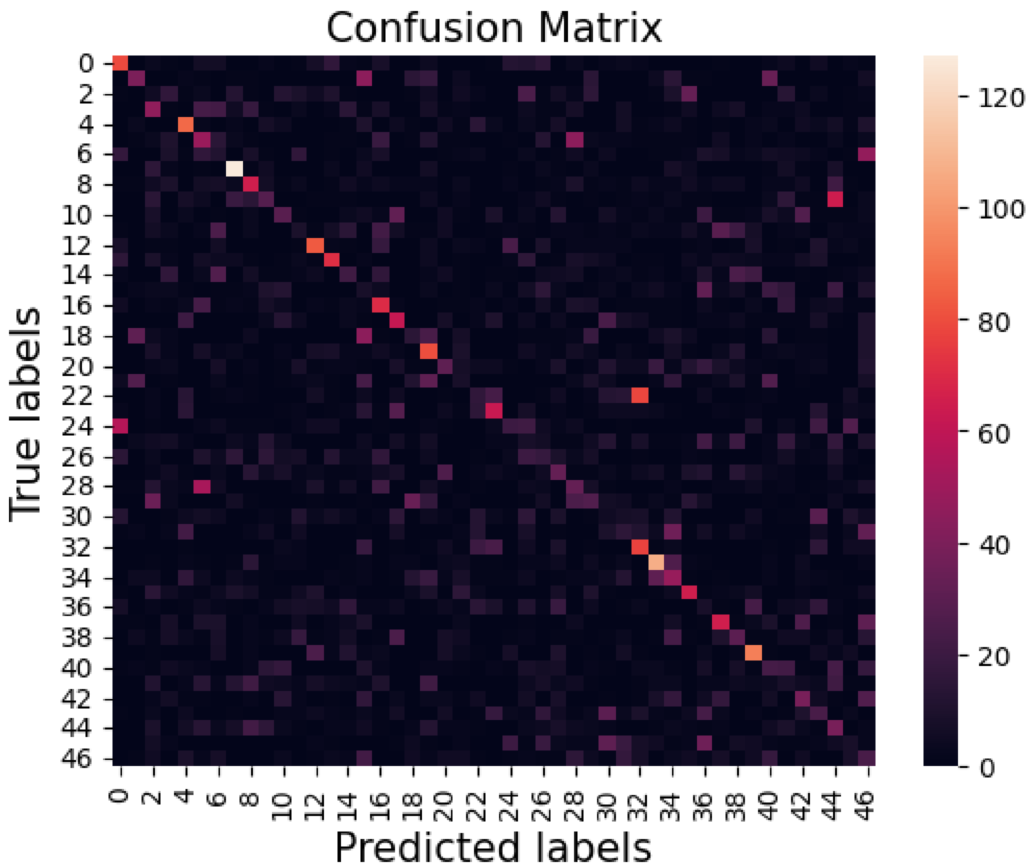
```
y_pred_val = clfPCA.predict(X_val_reduced)
acc_val = metrics.accuracy_score(y_val, y_pred_val)
loss_val = metrics.log_loss(y_val, clfPCA.predict_log_proba(X_val_reduced))

print("Training Set")
print(f" - Accuracy: {100*acc_train:.2f}%")
print(f" - Loss: {loss_train:.2f}")
print("Validation Set")
print(f" - Accuracy: {100*acc_val:.2f}%")
print(f" - Loss: {loss_val:.2f}")
```

```
Training Set
 - Accuracy: 67.99%
 - Loss: 3.85
Validation Set
 - Accuracy: 19.94%
 - Loss: 3.85
```

[20]:
```
generate_confusion_matrix(y_train, y_pred_train)
```



[21]:
```
generate_confusion_matrix(y_val, y_pred_val)
```



0.3 Part 2 - MLP

0.3.1 Section 2.1 - Optimizing parameters

1. Find the ideal number of neurons for an ANN with one hidden layer
2. Find the ideal number of layers with the "ideal number of neurons" determined in 1

[ ]:
```
# DO NOT RUN THIS CELL AGAIN (unless you have a lot of spare time and␣
↪computational power)
# Uncomment the block below when testing number of neurons
#num_neurons = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200]
#num_layers = 1

# Comment the block below when testing the number of neurons
num_neurons = 900
num_layers = [1,2,3,4,5]

accuracies = []
```

```
# for num in num_neurons: # Uncomment this when testing number of neurons
for num in num_layers: # Comment this whe ntesting number of neurons
    print("Now starting: {} layers".format(num))
    # mlp = MLPClassifier(max_iter=200, hidden_layer_sizes=(num,),␣
 ↪random_state=0) # Uncomment this when testing number of neurons
    mlp = MLPClassifier(max_iter=200,␣
 ↪hidden_layer_sizes=([num_neurons]*num_layers), random_state=0) # Comment␣
 ↪this when testing number of neurons
    X_train_small = X_train[:10000]
    y_train_small = y_train[:10000]
    mlp.fit(X_train_small, y_train_small)
    y_pred_tr = mlp.predict(X_train_small)
    y_pred_test = mlp.predict(X_test)
    accuracies.append(metrics.accuracy_score(y_test, y_pred_test))

# Uncomment the block below when testing number of neurons
# evaluation = pd.DataFrame({
#     "num_neurons": num_neurons,
#     "accuracy": accuracies
# })
# evaluation

# Comment the block below when testing number of neurons
evaluation = pd.DataFrame({
    "num_layers": num_layers,
    "accuracy": accuracies
})
evaluation

# === RESULTS ===

# Ideal layer size analysis - Number of layers is kept constant as 1.
#       num_neurons accuracy
# 0     100         0.543564
# 1     200         0.613085
# 2     300         0.595851
# 3     400         0.600904
# 4     500         0.604681
# 5     600         0.624894
# 6     700         0.659894
# 7     800         0.646330
# 8     900         0.689734 <== Local maximum
# 9     1000        0.656011
# 10    1100        0.672606
# 11    1200        0.658936
```

```
# Ideal number of layers analysis - Number of neurons per layer is kept␣
↪constant at 900 (as determined by the previous table)
#               num_layers  accuracy
# 0     1           0.689734
# 1     2           0.652181
# 2     3           0.700106 <== Local maximum
# 3     4           0.677021
# 4     5           0.710691 <== would take far too long to train

# Therefore, a MLP with 3 hidden layers and 900 neurons per layer is ideal.
# Of course, it's not actually ideal as varying layer sizes may lead to better␣
↪results,
# however, I do not have a significant amount of computational power or time to␣
↪test all possibilities.
```

0.3.2 Section 2.2 - Training the MLP Model

[ ]:
```
# PLEASE DO NOT OVERWRITE THE MODEL (run if you wish but I shall not do it␣
↪again)
mlp_trained = MLPClassifier(random_state=0, max_iter=2,␣
↪hidden_layer_sizes=(900,900,900,), verbose=True)
print("Model created")
mlp_trained.fit(X_train, y_train)
# On a Ryzen 7 5700U, it took 78 minutes to train

# Uncomment to overwrite the locally saved model
%store mlp_trained
```

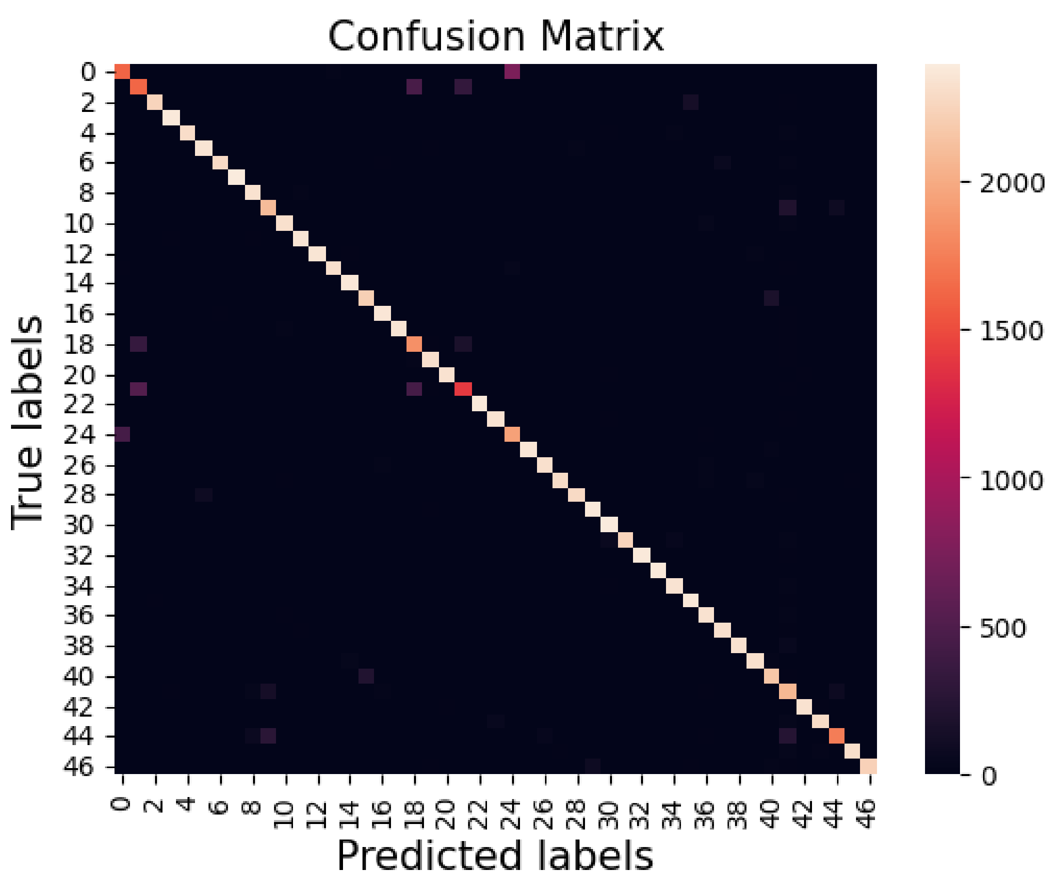
0.3.3 Section 2.3 - Evaluation

[25]:
```
# mlp_trained is a saved variable. Layers: (900,900,900), max_iter: 100
# mlp_test is the model generated above
to_test = mlp_trained
# On training dataset
y_pred_train = to_test.predict(X_train)
y_pred_proba_train = to_test.predict_proba(X_train)
accuracy_train = metrics.accuracy_score(y_train, y_pred_train)
loss_train = metrics.log_loss(y_train, y_pred_proba_train)
print("Training Set:")
print(f" - Loss: {loss_train:.2f}")
print(f" - Accuracy: {accuracy_train*100:.2f}%")
# On testing dataset
y_pred_val = to_test.predict(X_val)
y_pred_proba_val = to_test.predict_proba(X_val)
accuracy_val = metrics.accuracy_score(y_val, y_pred_val)
loss_val = metrics.log_loss(y_val, y_pred_proba_val)
```

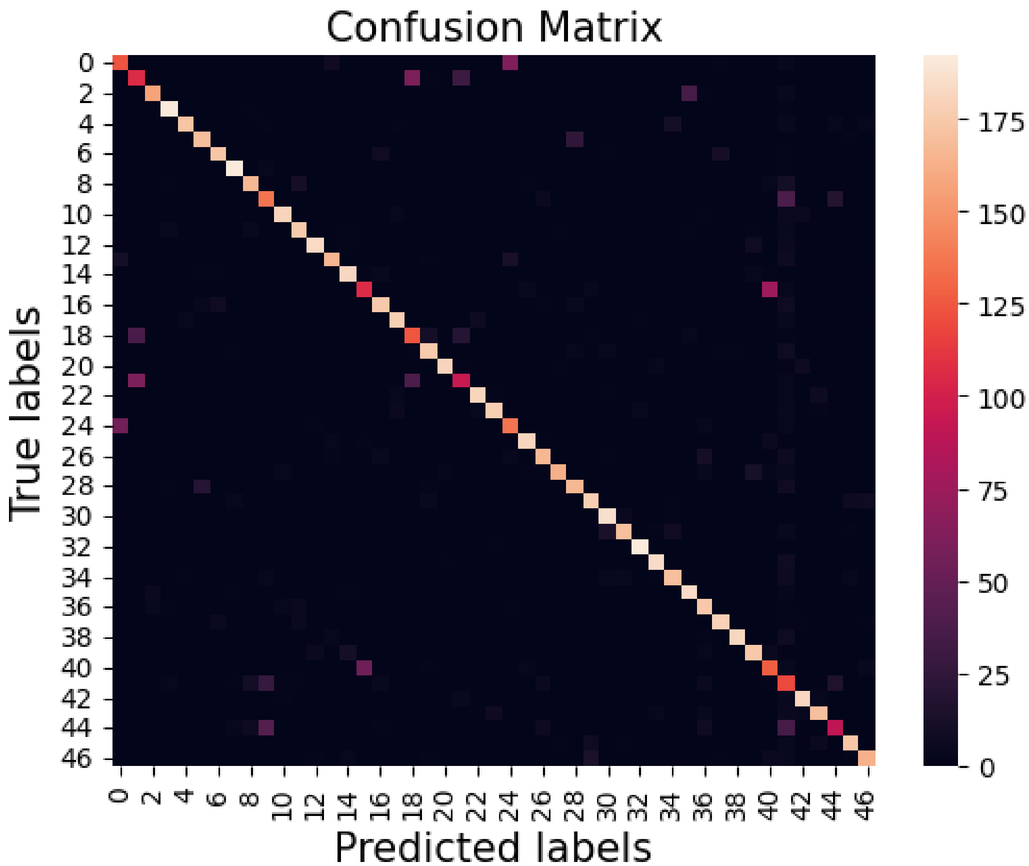
```
print("Validation Set")
print(f" - Loss: {loss_val:.2f}")
print(f" - Accuracy: {accuracy_val*100:.2f}%")

# === RESULTS ===

# Training Set:
#   - Loss: 0.23752188350972647
#   - Accuracy: 0.9301152482269504
# Testing Set
#   - Loss: 1.3251768763344238
#   - Accuracy: 0.8181914893617022
```

```
Training Set:
 - Loss: 0.24
 - Accuracy: 93.01%
Validation Set
 - Loss: 1.35
 - Accuracy: 81.54%
```

[26]:
```
generate_confusion_matrix(y_train, y_pred_train)
```



[28]:
```
generate_confusion_matrix(y_val, y_pred_val)
```



0.4 Part 3 - Chosen Model (MLP) and test dataset results

[30]:
```
y_pred_test = to_test.predict(X_test)
y_pred_proba_test = to_test.predict_proba(X_test)
accuracy_test = metrics.accuracy_score(y_test, y_pred_test)
loss_test = metrics.log_loss(y_test, y_pred_proba_test)
print("Testing Set")
print(f" - Loss: {loss_test:.2f}")
print(f" - Accuracy: {accuracy_test*100:.2f}%")
```

```
Testing Set
 - Loss: 1.30
 - Accuracy: 82.10%
```