Classifying alphanumeric characters using ML classification methods

1 - Introduction

Text recognition has many applications in the modern world. It can be used by any phone with a camera to automatically detect the words captured in a picture. A real-world application for text recognition is for archival efforts. For example, the New York Times has used Optical Character Recognition to digitize old articles [1]. As the particular ML problem that will be explained in this report only deals with alphanumeric characters, the resulting model of the problem can potentially be used in conjunction with image manipulation to detect license plate numbers of motor vehicles.

This report will be structured in the following manner: part 1 generally introduces the ML problem, part 2 explains the data used and formulates the problem, part 3 discusses the two methods that we used to approach the ML problem, part 4 contains the analysis of the results obtained from the methods, and part 5 summarizes the results. The paper ends with the bibliography, and the relevant code is attached at the very end of this paper in appendices.

This report will be referring to the set of characters consisting of the digits from 0 to 9 and the lowercase and uppercase alphabet as “alphanumeric characters”.

2 - Problem formulation

2.1 - About the Dataset

The dataset used in this paper is the EMNIST database, which is freely available on Kaggle [2]. This dataset is licensed under CCO 1.0 license, and was originally created by Cornell University [3].

2.2 - Data Points

The data points of the dataset consist of handwritten alphanumeric characters. Each datapoint has features that represent the pixels of a 28x28 pixel grayscale image (more on this in 2.3), along with a label that denotes what alphanumeric character the image is supposed to represent. Examples of the images can be seen in Figure 1 in Appendix A.

2.3 - Features

As the features of each data point represents a 28x28 pixel grayscale image, there is a feature for each pixel in the image. Thus, each data point has 784 features in the form of a feature vector which represents the handwritten image. Each individual feature has an integer value between 0 and 255 inclusive, which represents a grayscale value. That is, 0 would be a black pixel and 255 would be a white pixel. To convert the feature vector into a grayscale image, the first 28 values from the feature vector are taken and converted into grayscale pixels to create the first row of the image, the next 28 values (the 29th feature to the 56th feature) are taken and converted into grayscale pixels to create the second row of the image, and so on.

2.4 - Labels

Each data point is given a categorical label, which is an integer between 0 and 46. It should be noted that there are 62 total alphanumeric characters, while the data points are differentiated into 47 categories. The reason for this is due to some alphabetical characters having similar-looking lowercase and uppercase characters, thus they are both categorized similarly. The 15 characters fulfilling this condition are as follows: c, i, j, k, l o, p, s, u, v, w, x, y, z. Subtracting this from our 62 total alphanumeric characters yields our 47 categories. The corresponding character of each categorical label is shown in Figure 2 in Appendix A.

3 - Methods

3.1 - Dataset Splitting

Six versions of the dataset were created by its creators. We decided to use the "balanced" version, as it covers all previously-mentioned 47 categories and it provides a consistent number of samples for each category. It contains 131,600 data points in total, which has been pre-split by the creators into a training set containing 112,800 data points and a testing set containing 18,800 data points. The data points in the testing set appear to have been selected through stratified random sampling. This results in approximately a ratio of 86:14 between the training and testing dataset sizes. When taking into consideration the rules-of-thumb for splitting data [6], we feel that the split offered by the creator is reasonable as firstly, it is close to the recommended 80:20 split ratio, and secondly, since the full dataset is very large, a larger training set may be beneficial for accuracy.

However, since we needed to have a validation set as well, we split the testing set into another testing set and validation set using stratified random sampling with a 1:1 ratio between the datasets. Thus, both the validation and testing sets contain 9,400 data points.

As all the values within the feature vector are of the same type (they are all integers between 0 and 255 inclusive), feature preprocessing is not necessary.

3.2 - Method 1 - Logistic Regression with PCA feature preprocessing

As the labels are categorical, logistic regression is used as it is a classification method. However, due to our dataset consisting of a vast 131,600 data points and each data point having 784 features, the training time of the model would be very large if no preprocessing is used. Thus, principal component analysis (PCA) can be used to reduce the number of features and thus the training time. It may also have the added benefit of reducing the likelihood of overfitting.

By applying a scoring system that maximizes the cumulative variance ratio while minimizing the number of features, we decided that we would use PCA to reduce the number of features from 784 to 61. More about the scoring system can be found in Section 1.1 of Appendix B.

Logistic regression is a classification algorithm. This method analyzes relationships between variables. As we are dealing with large amount of numbers, and differences between them determine what kind of character it is, finding differences between variables is of key importance. As the logistic loss function is associated with logistic regression, the loss function used will be the logistic loss function, which is provided by *sklearn* library.

3.3 - Method 2 - Multilayer Perceptron (MLP)

As our problem involves the recognition of handwritten characters, pattern recognition is essential for the accurate recognition of the characters. Since neural networks excel at pattern recognition compared to alternative classifying ML models due to their use of hidden layers [4], we will be using the multilayer perceptron (MLP) as a method to create a ML model for our problem, which is an artificial neural network (ANN). As neural networks have hidden layers and neurons, with enough training data, the model can be much more resilient to outliers compared to logistic regression, thus PCA is not used in this method.

As this ML model is present in the ready-made python library *sklearn*, the loss function used will be the default loss function used by the MLPClassifier model provided by the library, which is the logistic loss function [5].

4 - Results

4.1 - Accuracies on Training Set

The maximum accuracy obtained on the training dataset with the logistic regression model is 69.30% with the result of the loss function being 3.85, which is promising. As for the MLP model, the maximum accuracy obtained when used on the training dataset was 93.0% with the result of the loss function being 1.35. This is quite excellent, as it is above 90%. However, this metric alone cannot define how good the model is as there is a possibility of overfitting.

4.2 - Final Choice & Errors

Using the testing dataset is essential for properly evaluating the two models to eliminate the possibility of overfitting. The logistic regression model that utilized PCA yielded an accuracy score of 19.94% with the result of the loss function being 3.85, which is unsatisfactory. Meanwhile, the MLP model achieves an accuracy score of 81.8% on the testing dataset with the result of the loss function being 1.35, which is very quite satisfactory. It is clear that overfitting did not occur in the MLP model while it did occur in the logistic regression model.

As the MLP model yields a higher accuracy on the testing set than the logistic regression, we can conclude that the former method is better, and thus it is chosen as the final method.

4.3 - Results of the Chosen Model on the Testing Set

On the testing set, the MLP model achieved an accuracy score of 82.1% with the result of the loss function being 1.30. Note that the construction of the test set was detailed in section 3.1.

5 - Conclusion

5.1 - Findings

In terms of actual performance, the second method is vastly superior to the first, as it avoided overfitting and yielded an accuracy score of above 80% on all datasets. However, it is also important to look at the training time used for each method. Although the first method yielded a worse accuracy score than the second, it took 53 seconds to train the model from the first method, while the model from the second method took 78 minutes to train. The training times may vary depending on the hardware used, but there will remain a substantial difference between the two durations regardless of hardware. The MLP model yielded vastly better results than the logistic regression model, but required over 77 more minutes to train. Given that the logistic regression model still shows a distinct diagonal on the confusion matrix when used to predict labels for the validation set, as seen in Figure 6 of Appendix A, it could be stated that the logistic regression model that utilized PCA yields adequate results if there is very little time to train the model.

5.2 - Caveats

For the method that utilized logistic regression and PCA, it is clear that the logistic regression is not ideal for this particular classification problem. As for the method that utilized MLP, a lack of time and computational power led to not being able to figure out the absolute ideal configuration for the hidden layers. Due to each training session for the whole training dataset taking roughly an hour, our model had to resort on small-scale optimizations on a subset of the training set before running it on the full training set. Additionally, to reduce time spent optimizing parameters, an assumption was made about the number of neurons per hidden layer and number of hidden layers being independent variables, when in reality, they are not.

5.3 - Further Improvement

For further improvement, as it can be seen that neural networks are indeed effective at pattern recognition, it may be possible to obtain better results using a more complex neural network model, such as a convolutional neural network (CNN).

6 - Bibliography:

[1] L. Nordahl, A. Chavar, L. Porter, M. Kim, and A. Blufarb, “Using Computer Vision to Create A More Accurate Digital Archive,” rd.nytimes.com, Jul. 21, 2021. https://rd.nytimes.com/projects/using-computer-vision-to-create-a-more-accurate-digital-archive (accessed Sep. 22, 2022).

[2] C. Crawford, “EMNIST (Extended MNIST),” www.kaggle.com, Dec. 20, 2017. https://www.kaggle.com/datasets/crawford/emnist (accessed Sep. 22, 2022).

[3] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, “EMNIST: an extension of MNIST to handwritten letters,” Feb. 2017, Accessed: Oct. 03, 2022. [Online]. Available: https://arxiv.org/abs/1702.05373v1

[4] B. D. Ripley, Pattern recognition and neural networks. Cambridge ; New York: Cambridge University Press, 2007.

[5] “sklearn.neural_network.MLPClassifier — scikit-learn 0.20.3 documentation,” Scikit-learn.org, 2010. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html (accessed Oct. 06, 2022).

[6] B. Giba, “How to Split Your Dataset the Right Way,” Machine Learning Compass, May 01, 2021. https://machinelearningcompass.com/dataset_optimization/split_data_the_right_way/ (accessed Oct. 07, 2022).

Appendix A - Figures

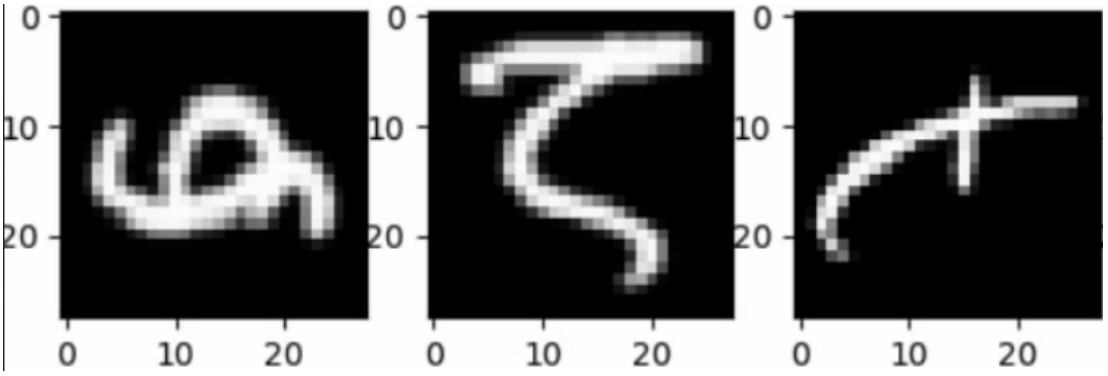


Figure 1: Example image representations of feature vectors. The characters are g, N, and t from left to right.

Label	Character
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B

Label	Character
12	C
13	D
14	E
15	F
16	G
17	H
18	I
19	J
20	K
21	L
22	M
23	N

Label	Character
24	O
25	P
26	Q
27	R
28	S
29	T
30	U
31	V
32	W
33	X
34	Y
35	Z

Label	Character
36	a
37	b
38	d
39	e
40	f
41	g
42	h
43	n
44	q
45	r
46	t

Figure 2: Categorical labels of the dataset and their corresponding characters

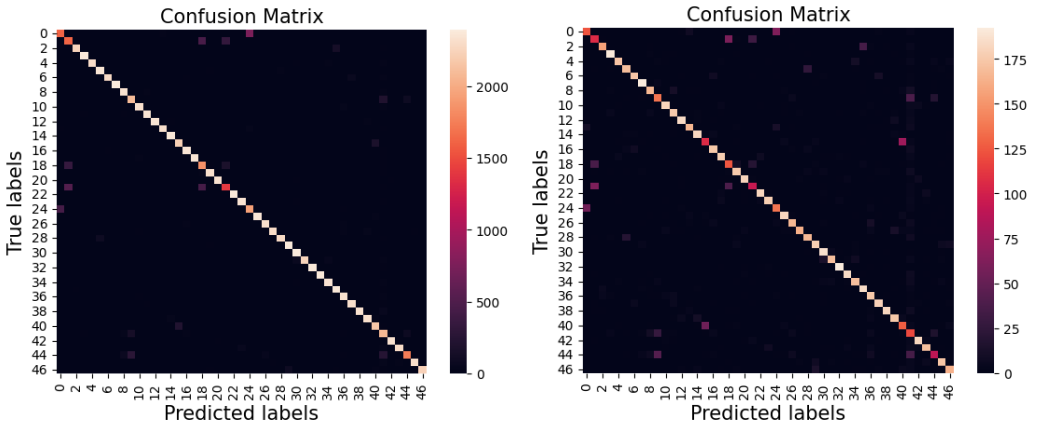


Figure 4. Confusion matrices for MLP model on training set (left) and validation set (right)

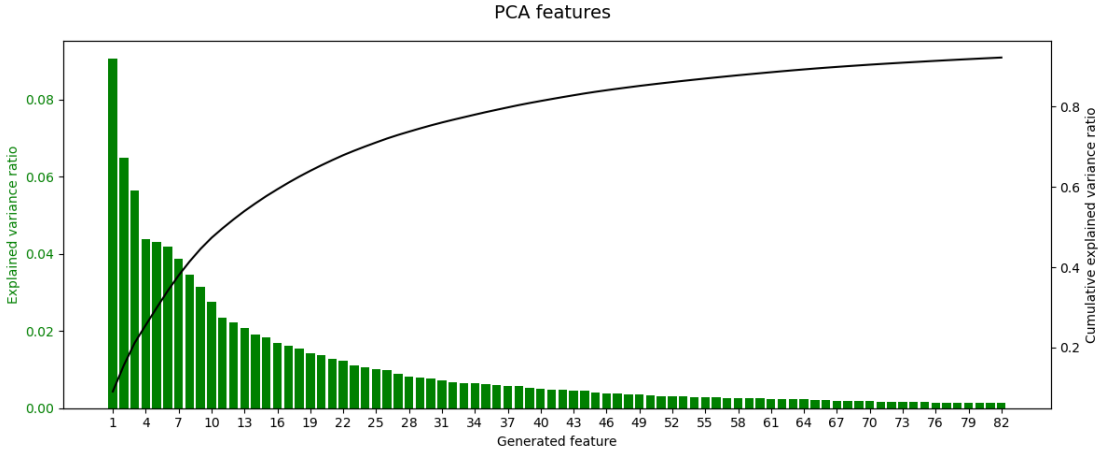


Figure 5. Graph representing the effect the amount of PCA features on variance ratio and cumulative explained variance ratio.

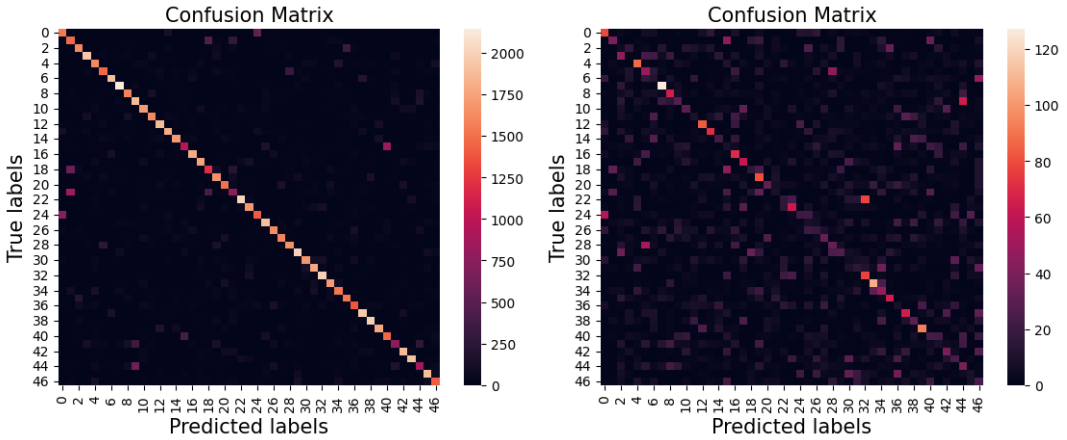


Figure 6. Confusion matrices for Logistic Regression model on training set (left) and validation set (right).

Appendix B - Code

In order to set up the environment for the code to work properly:

1. Download the EMNIST dataset from [2]
2. Take the emnist-balanced-test.csv and emnist-balanced-train.csv files and put them into a folder called “EMNIST”.
3. Put the “EMNIST” folder into the same directory as the jupyter notebook.
4. Make sure the following packages are installed through pip:
    a. sklearn
    b. matplotlib
    c. pandas
    d. numpy
    e. seaborn

Your file structure should look like this:

- EMNIST
    - emnist-balanced-test.csv
    - emnist-balanced-train.csv
- project.ipynb

After the steps above, all code blocks in the code below should run successfully.

project

October 9, 2022

Part 0 - Setup

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)
```

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,)
```

Part 1 - PCA

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
```

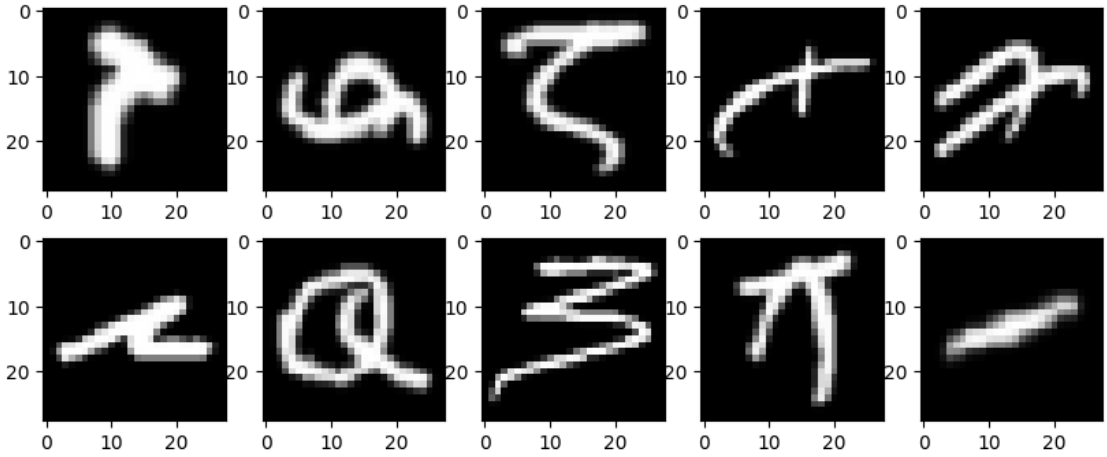
[13]:
```
best = max(scores, key=lambda x: x[1])
print("Best number of features: " + str(best[0]))
print("Score: " + str(round(best[1] * 100, 2)))
print(f"Feature size reduction: {best[2] * 100:.2f}%")
print(f"Cumulative Variance Ratio: {best[3] * 100:.2f}%")
# Best number of features: 82
# Score: 82.56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

Section 1.2 - PCA Modelling

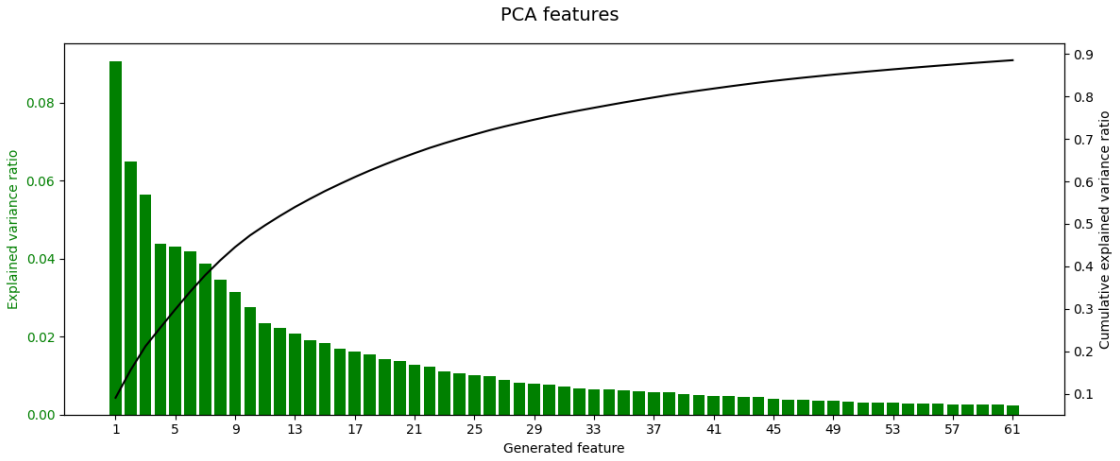
[17]:
```
# Visualize what the features look like
fig,axes = plt.subplots(2,5,figsize=(10,4))
for i,ax in enumerate(axes.flat):
    ax.imshow(X_train[i].reshape([28,28]), cmap='gray')
```



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

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

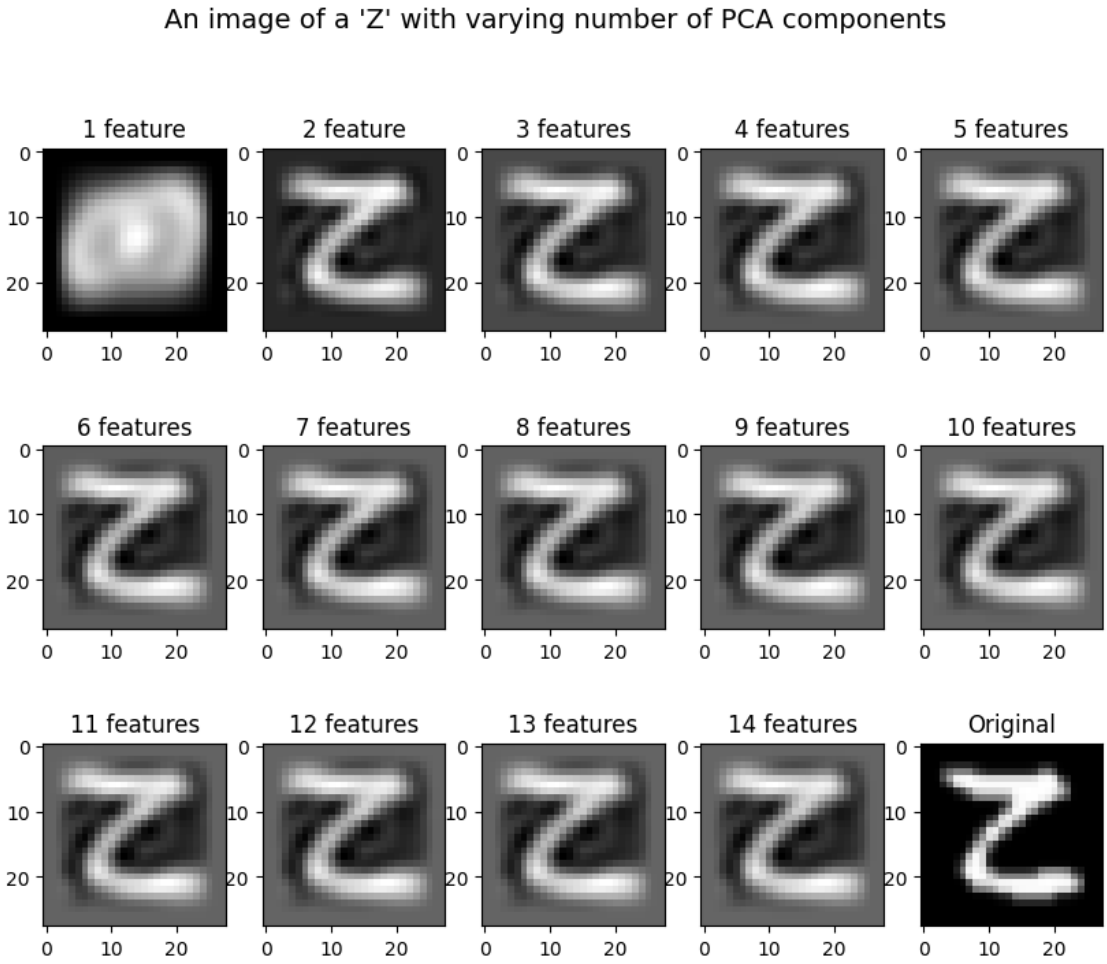
[15]:
```
fig, ax1 = plt.subplots(figsize=(12, 5))
fig.suptitle("PCA features", fontsize=14)
color = 'tab:blue'
ax1.bar(1+np.arange(N), pca.explained_variance_ratio_, color="green")
ax1.set_xticks(1+np.arange(N, step=4))
ax1.tick_params(axis='y', labelcolor="green")
ax1.set_ylabel("Explained variance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



[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)
```

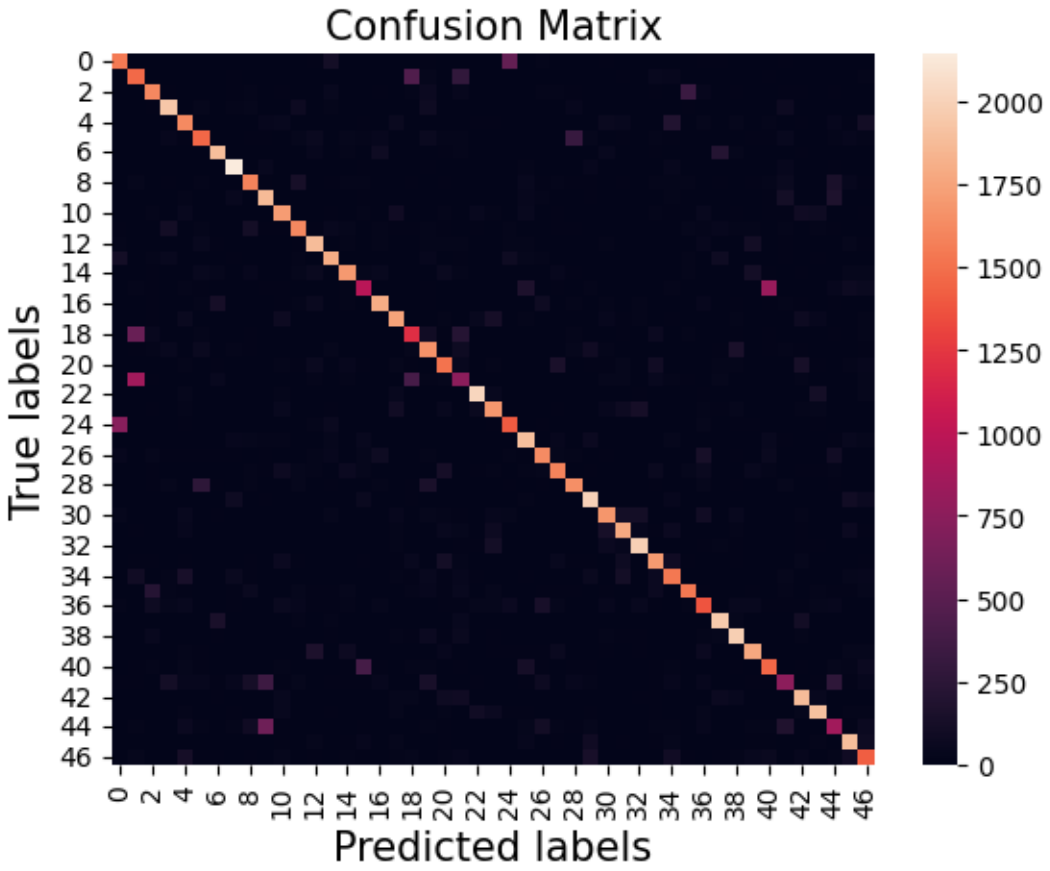
Section 1.3 - Evaluation

[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))
```

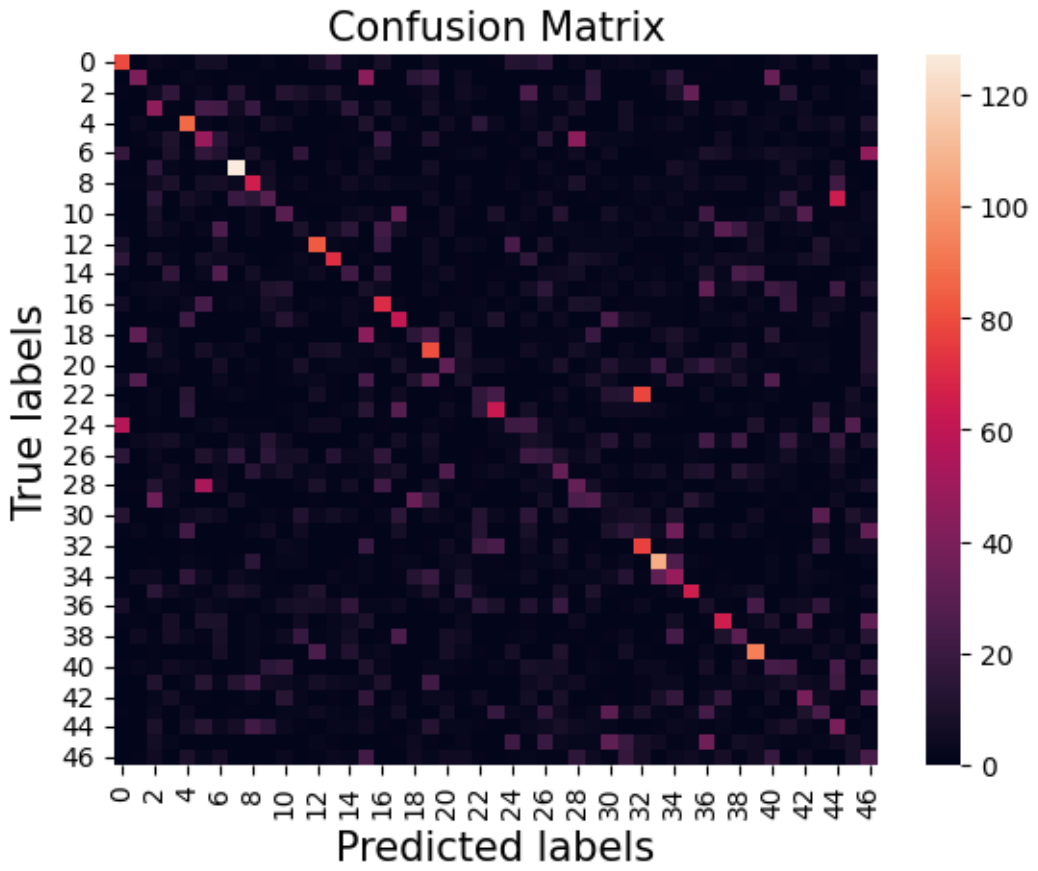
```
y_pred_val = clfPCA.predict(X_val_reduced)
acc_val = metrics.accuracy_score(y_val, y_pred_val)
loss_val = metrics.log_loss(y_val, clfPCA.predict_log_proba(X_val_reduced))

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)
```



Part 2 - MLP

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.
```

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
```

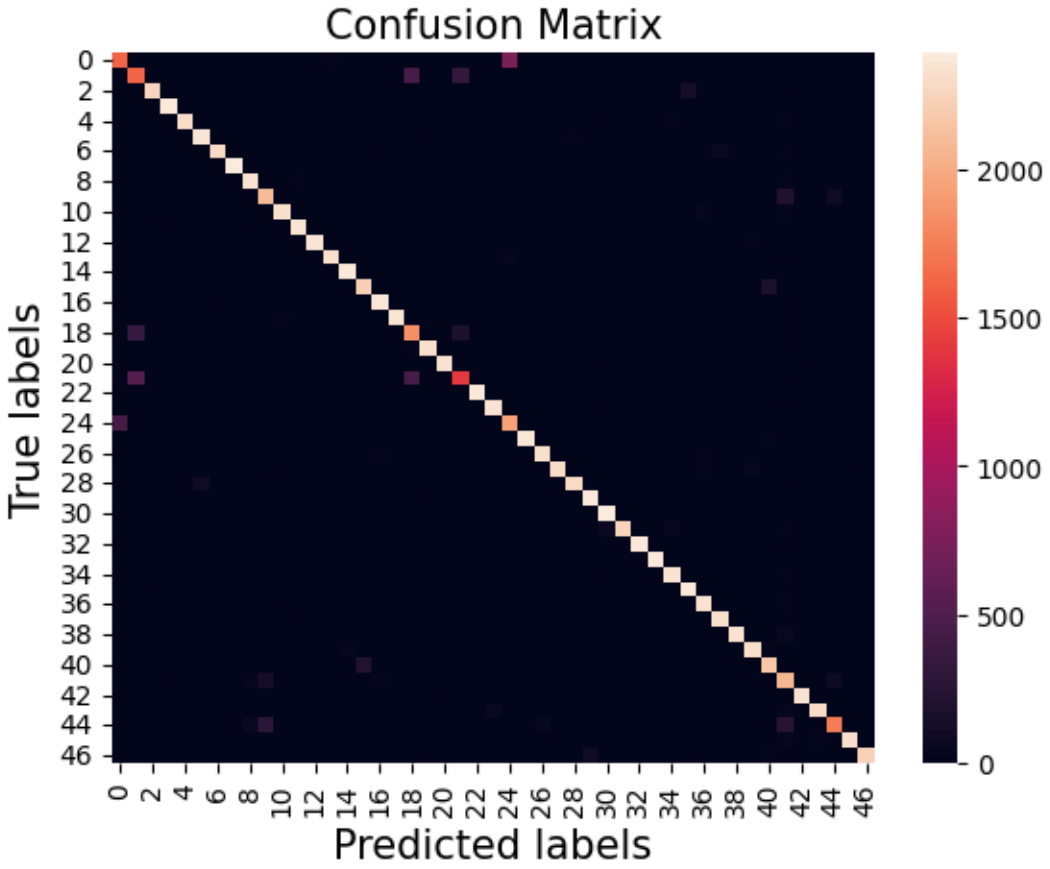
Section 2.3 - Evaluation

[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)
```

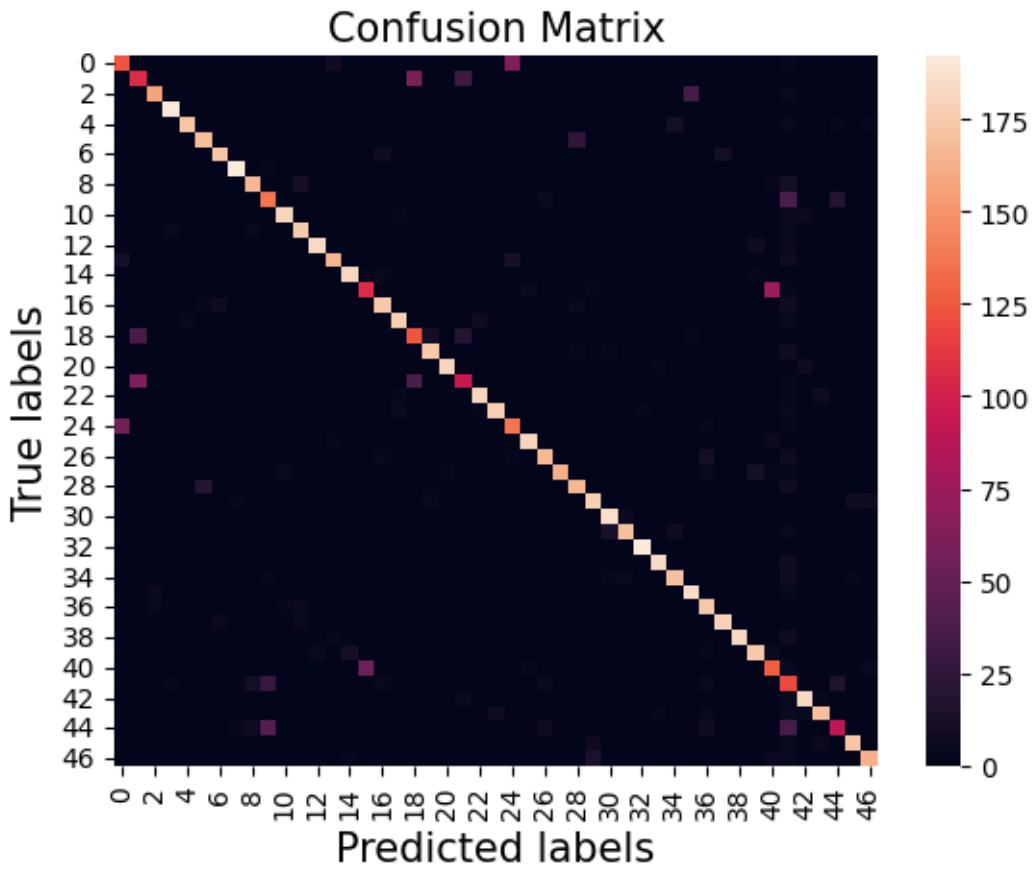
```
print("Validation Set")
print(f" - Loss: {loss_val:.2f}")
print(f" - Accuracy: {accuracy_val*100:.2f}%")

# === 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)
```



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%
```