Research on the effectiveness of artificial neural networks for large scale classification

My research is based on this paper:

X. Wu, Y. Zhang, and W. Fan, "Linear Regression based Efficient SVM Learning for Large Scale Classification," IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 12, pp. 3025-3038, Dec. 2014. DOI: 10.1109/TKDE.2014.2308558.

MNIST

In [45]:
```
import time
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
```

In [2]:
```
# Load the MNIST dataset
mnist = fetch_openml('mnist_784')
X = mnist['data']
y = mnist['target']
y = y.astype(np.int)
```

```
C:\Users\Checkout\AppData\Local\Temp\ipykernel_14700\2806148950.py:5: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y = y.astype(np.int)
```

In [3]:
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Reshaping the images from a flattened format to their original 28x28 size.

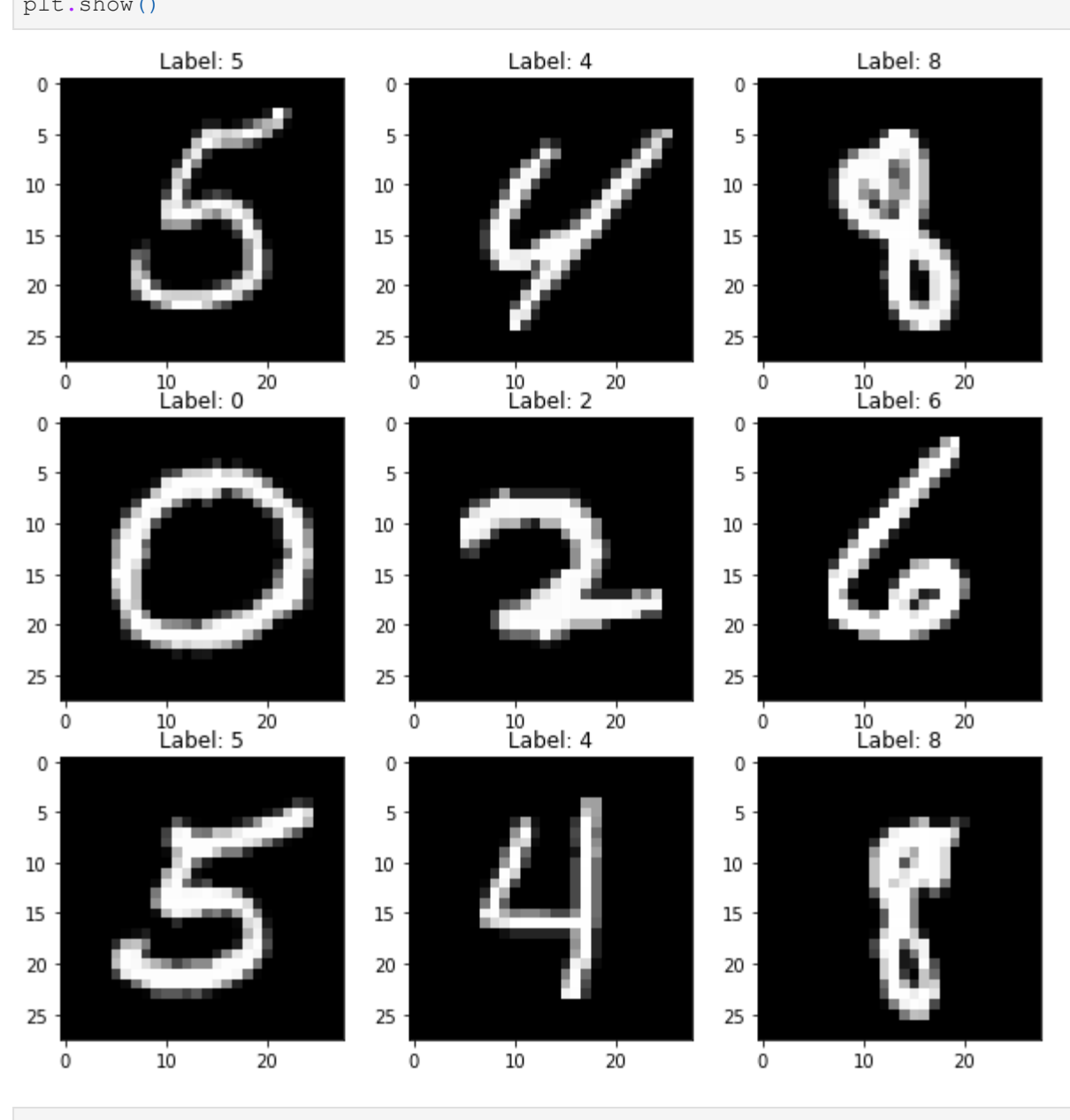
In [4]:
```
# Reshape the images to their original 28x28 size
#converting the images back to 3D array
X_train_images = X_train.values.reshape((X_train.shape[0], 28, 28))
X_test_images = X_test.values.reshape((X_test.shape[0], 28, 28))
```

But, for the training and testing sets, we want the images in 1D array, so as to work with ML models. The below result indicates that the images have been flattened to 1D array of size 784. (28x28 = 784).

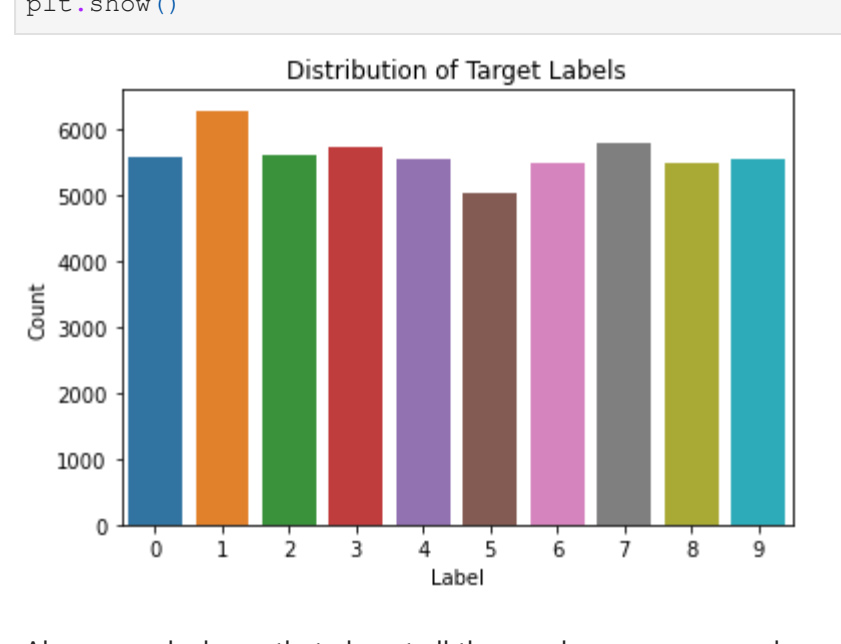
In [5]:
```
# Print the shape of the training and test sets
print("Training Set Shape:", X_train.shape, y_train.shape)
print("Test Set Shape:", X_test.shape, y_test.shape)
```

```
Training Set Shape: (56000, 784) (56000,)
Test Set Shape: (14000, 784) (14000,)
```

In [6]:
```
import matplotlib.pyplot as plt
# Visualize a random sample of the images
fig, ax = plt.subplots(3, 3, figsize=(10, 10))
y_train_reset = y_train.reset_index(drop=True)
for i, axi in enumerate(ax.flat):
    axi.imshow(X_train_images[i], cmap='gray')
    axi.set_title("Label: {}".format(y_train_reset[i]))
plt.show()
```

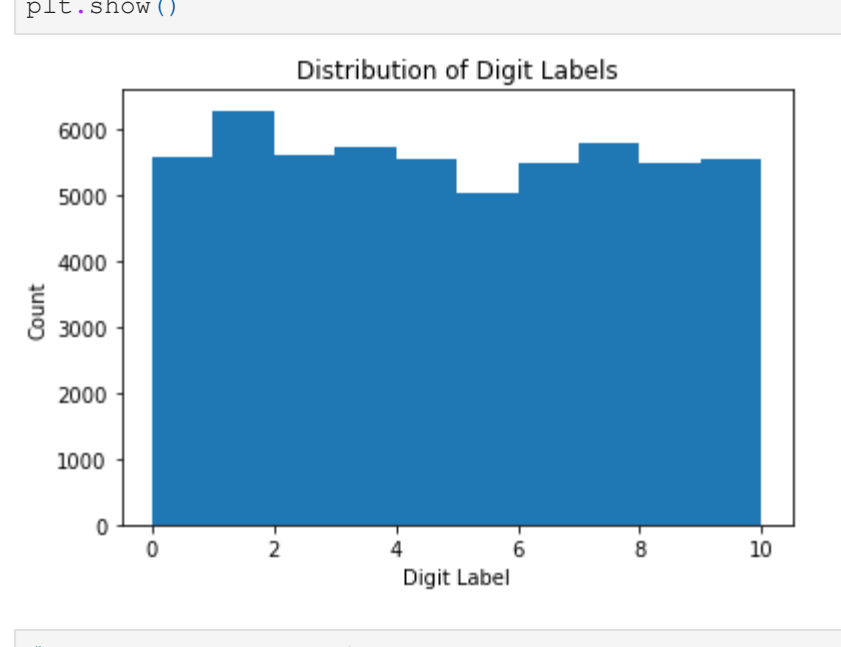


In [7]:
```
import seaborn as sns
# Plot the distribution of the target labels
sns.countplot(x=y_train)
plt.title("Distribution of Target Labels")
plt.xlabel("Label")
plt.ylabel("Count")
plt.show()
```

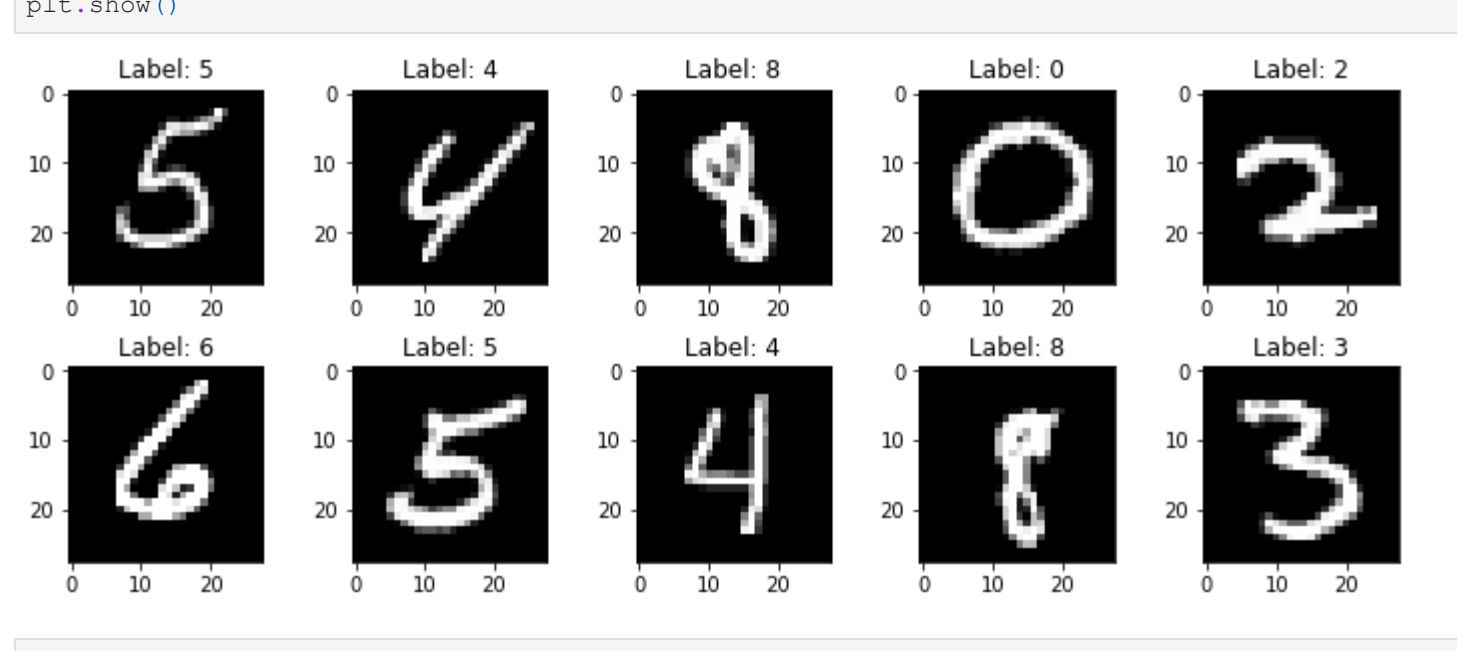


Above graph shows that almost all the numbers are more or less equally distributed in the dataset. The same can be verified from the below histogram.

In [8]:
```
import matplotlib.pyplot as plt

# Plot a histogram of the target variable
plt.hist(y_train, bins=range(11))
plt.title("Distribution of Digit Labels")
plt.xlabel("Digit Label")
plt.ylabel("Count")
plt.show()
```

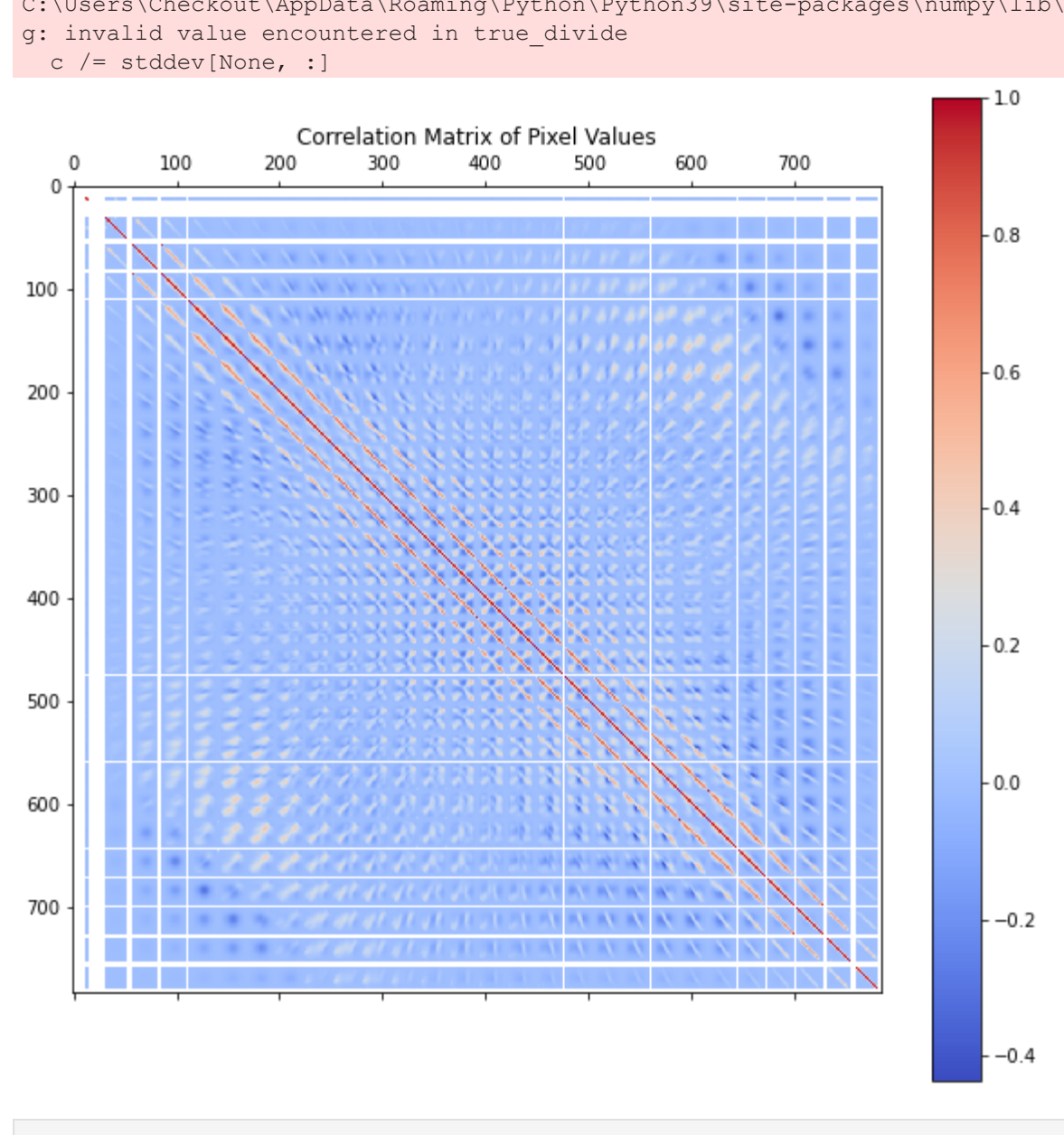


In [15]:
```
# Plot some example images from the dataset
y_train = y_train.reset_index(drop=True)  # reset index
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 4))
for i, ax in enumerate(axes.flat):
    ax.imshow(X_train_images[i], cmap='gray')
    ax.set(title=f"Label: {y_train[i]}")
plt.tight_layout()
plt.show()
```

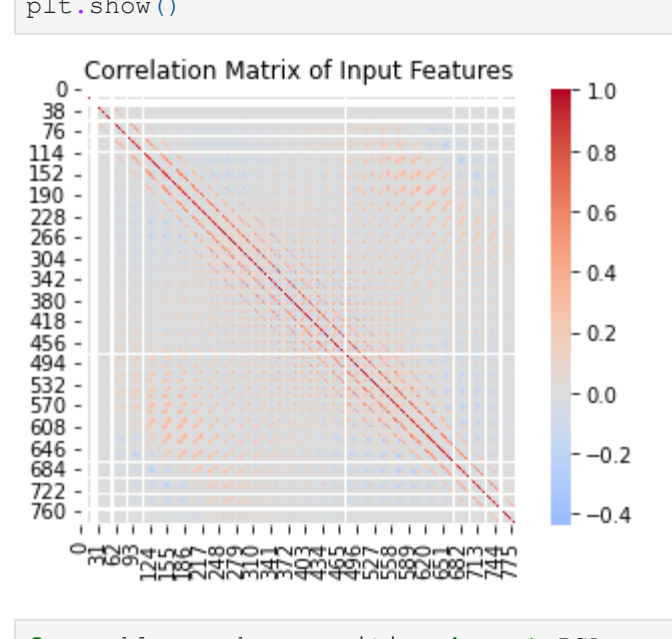


In [9]:
```
# Compute the correlation matrix of pixel values
corr_matrix = np.corrcoef(X.T)

# Visualize the correlation matrix
fig, ax = plt.subplots(figsize=(10, 10))
cax = ax.matshow(corr_matrix, cmap='coolwarm')
plt.title('Correlation Matrix of Pixel Values')
fig.colorbar(cax)
plt.show()
```

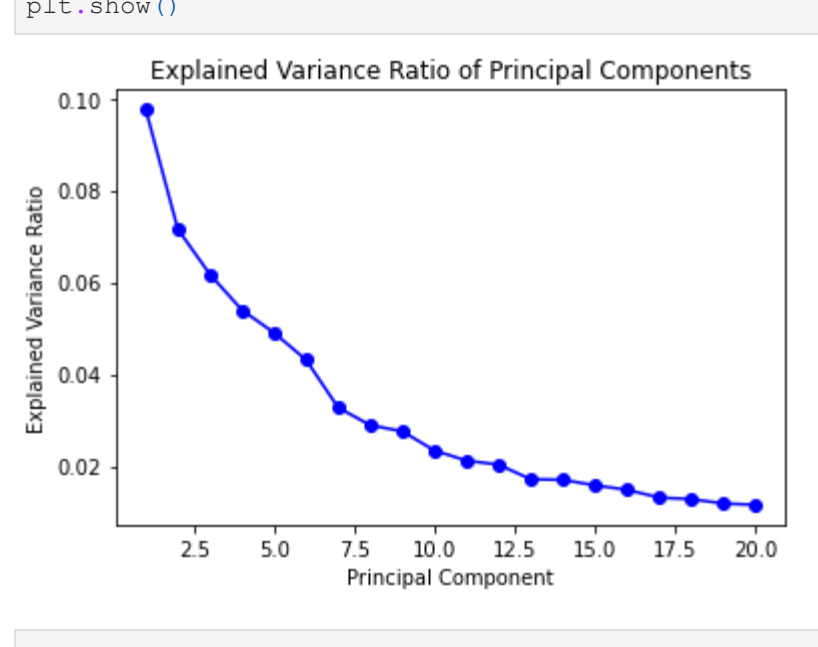
```
C:\Users\Checkout\AppData\Roaming\Python\Python39\site-packages\numpy\lib\function_base.py:2829: RuntimeWarning: invalid value encountered in true_divide
  c /= stddev[:, None]
C:\Users\Checkout\AppData\Roaming\Python\Python39\site-packages\numpy\lib\function_base.py:2830: RuntimeWarning: invalid value encountered in true_divide
  c /= stddev[None, :]
```



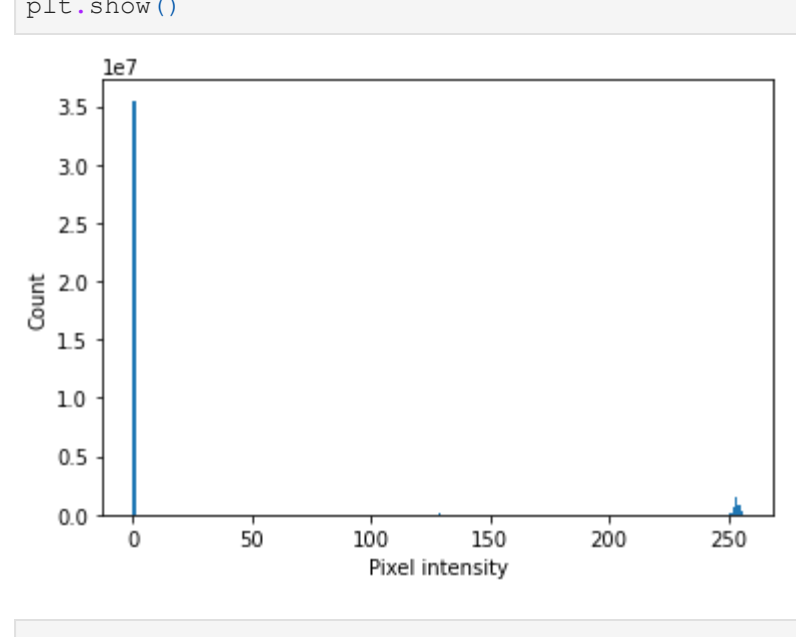
In [10]:
```
import seaborn as sns

# Compute the correlation matrix between the input features
corr_matrix = np.corrcoef(X_train.T)

# Visualize the correlation matrix using a heatmap
sns.heatmap(corr_matrix, cmap='coolwarm', center=0, square=True)
plt.title("Correlation Matrix of Input Features")
plt.show()
```



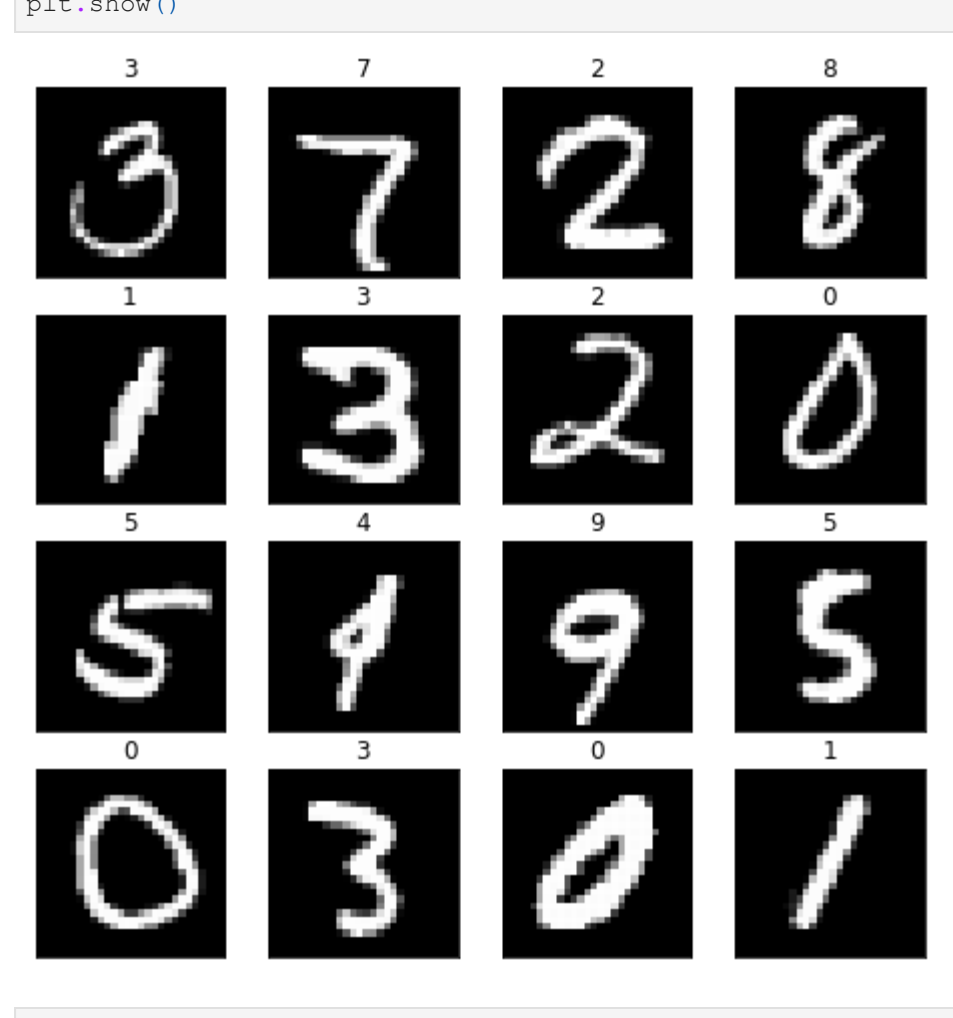
In [11]:
```
from sklearn.decomposition import PCA

# Compute the first 20 principal components of the input features
pca = PCA(n_components=20)
X_train_pca = pca.fit_transform(X_train)

# Visualize the explained variance ratio of the principal components
plt.plot(range(1, 21), pca.explained_variance_ratio_, 'bo-')
plt.title("Explained Variance Ratio of Principal Components")
plt.xlabel("Principal Component")
plt.ylabel("Explained Variance Ratio")
plt.show()
```



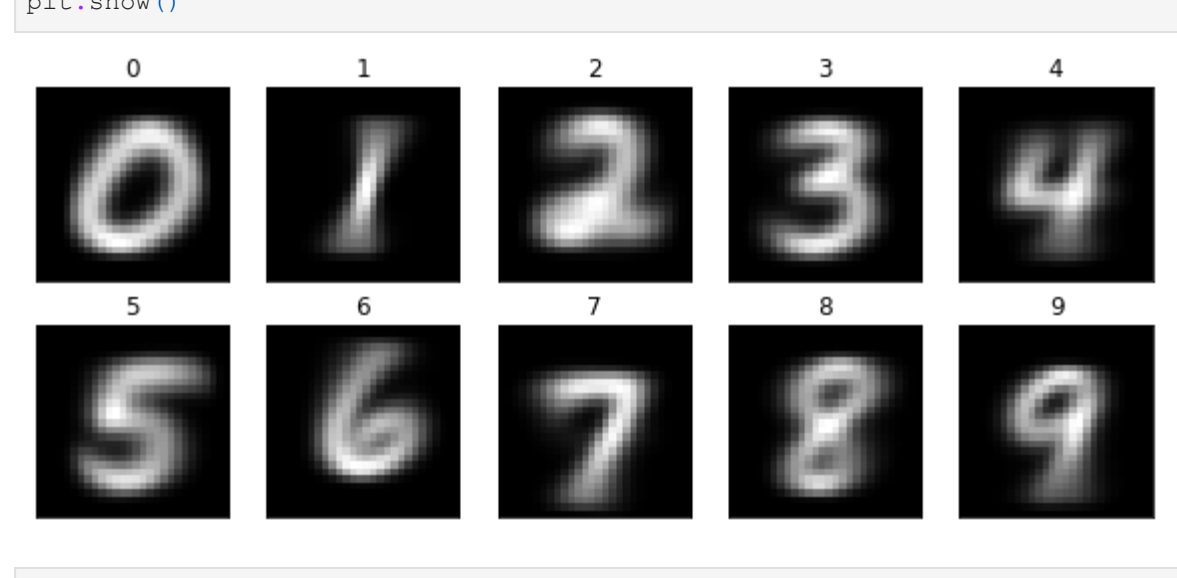
In [16]:
```
# Visualize the distribution of pixel intensities
plt.hist(X_train.values.flatten(), bins=range(257))
plt.xlabel("Pixel intensity")
plt.ylabel("Count")
plt.show()
```



In [18]:
```
import numpy as np
import matplotlib.pyplot as plt

# choose a random sample of images
sample = np.random.choice(X_train.shape[0], 16, replace=False)

# plot the sample
fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(8,8))
for i, ax in enumerate(axes.flat):
    img = X_train.iloc[sample[i]].values.reshape(28, 28)  # reshape the image data to 28x28
    ax.imshow(img, cmap='gray')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(y_train.iloc[sample[i]])
plt.show()
```



In [21]:
```
#plotting average image of each class
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 4))
for i, ax in enumerate(axes.flat):
    # get the mean image of the i-th class
    mean_image = np.mean(X_train[np.array(y_train) == i].values, axis=0)
    mean_image = mean_image.reshape(28, 28)
    ax.imshow(mean_image, cmap='gray')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(i)
plt.show()
```



In [22]:
```
# Check the number of unique labels
num_classes = len(np.unique(y_train))
print("Number of classes:", num_classes)
```

```
Number of classes: 10
```

Label encoding is a common preprocessing step used to convert categorical labels into numeric representations suitable for machine learning algorithms. It assigns a unique integer value to each unique label, essentially mapping each label to a corresponding numeric code. By performing label encoding, the original categorical labels are transformed into a format that can be easily processed and used for training a machine learning model.

In [23]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_test = le.fit_transform(y_test)
```

Standardizing the variables in the dataset by subtracting the mean and dividing by the standard deviation of each feature, which transforms the data to have zero mean and unit variance.

In [24]:
```
# Standardize the data
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
```

Below is the function I wrote to recreate the algorithm that the authors are talking about in their paper. Linear regression algorithm is used to estimate the weights of the SVM algorithm and also helps it take a lead in deciding what the support vectors should be, thereby helping it reach the optimal hyperplane faster for classification.

In [25]:
```
import numpy as np

class LinearRegressionSVM:
    def __init__(self, learning_rate=0.001, C=1.0, subset_size=50000, max_iterations=100):
        self.learning_rate = learning_rate
        self.C = C
        self.subset_size = subset_size
        self.max_iterations = max_iterations

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.coef_ = np.zeros(n_features)
        self.intercept_ = 0.0

        for iteration in range(self.max_iterations):
            subset_indices = np.random.choice(range(n_samples), self.subset_size, replace=False)
            X_subset = X[subset_indices]
            y_subset = np.array(y[subset_indices])  # convert y_subset to numpy array
            margin = y_subset * (np.dot(X_subset, self.coef_) + self.intercept_)
            if np.any(margin <= 0):
                misclassified_indices = np.where(margin <= 0)[0]
                random_misclassified_index = np.random.choice(misclassified_indices)
                self.coef_ += self.learning_rate * (np.dot(X_subset[random_misclassified_index], y_subset[rando
                self.intercept_ += self.learning_rate * y_subset[random_misclassified_index]

    def predict(self, X):
        y_pred = np.sign(np.dot(X, self.coef_) + self.intercept_)
        return y_pred
```

In [26]:
```
# Convert y_train to numpy array
y_train = y_train.astype(np.int)
```

```
C:\Users\Checkout\AppData\Local\Temp\ipykernel_14700\2855365193.py:2: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_train = y_train.astype(np.int)
```

Trying to fit RBF Kernel SVM to the data and compare it with the performance of the LinearRegression SVM that the authors are talking about. Tried to evaluate the performance of all the models on three parameters - accuracy, f1 score and training time.

In [23]:
```
# Train and evaluate SVM with RBF kernel
svm_rbf = SVC(kernel='rbf', C=1)
start_time = time.time()
svm_rbf.fit(X_train_std, y_train)
training_time_rbf = time.time() - start_time

y_pred_rbf = svm_rbf.predict(X_test_std)
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
f1_rbf = f1_score(y_test, y_pred_rbf, average='macro')
confusion_rbf = confusion_matrix(y_test, y_pred_rbf)
```

F1 score is a measure of model's accuracy that combines precision and recall into a single metric. Precision is the ability of a model to identify relevant instances correctly among the total predicted instances. Recall is the ability of a model to identify all relevant instances correctly among the total actual instances.

F1 score = 2 (*precision* recall) / (precision + recall)

Accuracy is the proportion of correctly classified instances out of the total number of instances in the dataset.

Accuracy = (Number of correctly predicted instances) / (Total number of instances)

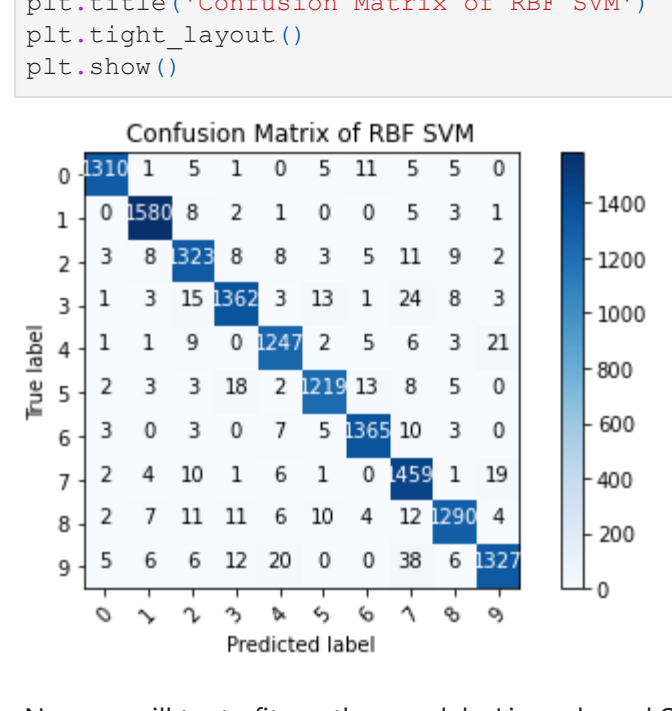
Training time is the amount of time the model took to train on the dataset. This is to check how much computation resources the model would consume to predict correctly.

In [24]:
```
print('Accuracy_RBF: ', accuracy_rbf)
print('F1 score_RBF: ', f1_rbf)
print('Training time_RBF: ', training_time_rbf)
print(confusion_rbf)
```

```
Accuracy_RBF:  0.963
F1 score_RBF:  0.9629116662632391
Training time_RBF:  556.1538064479828
[[1310    1    5    1    0    5   11    5    5    0]
 [   0 1580    8    2    1    0    0    5    3    1]
 [   3    8 1323    8    8    3    5   11    9    2]
 [   1    3   15 1362    3   13    1   24    8    3]
 [   1    1    9    0 1247    2    5    6    3   21]
 [   2    3    3   18    2 1219   13    8    5    0]
 [   3    0    3    0    7    5 1365   10    3    0]
 [   2    4   10    1    6    1    0 1459    1   19]
 [   2    7   11   11    6   10    4   12 1290    4]
 [   5    6    6   12   20    0    0   38    6 1327]]
```

From the above results, we see that RBF kernel has got an accuracy of 0.963, an f1 score of 0.962 and it took 556 seconds to train on the dataset. These are very good scores for a model. Now let's see how other models fare. I tried to plot the above obtained confusion matrix in a graphical format below.

In [32]:
```
import numpy as np
import matplotlib.pyplot as plt

confusion_matrix = np.array([[1310, 1, 5, 1, 0, 5, 11, 5, 5, 0],
                             [0, 1580, 8, 2, 1, 0, 0, 5, 3, 1],
                             [3, 8, 1323, 8, 8, 3, 5, 11, 9, 2],
                             [1, 3, 15, 1362, 3, 13, 1, 24, 8, 3],
                             [1, 1, 9, 0, 1247, 2, 5, 6, 3, 21],
                             [2, 3, 3, 18, 2, 1219, 13, 8, 5, 0],
                             [3, 0, 3, 0, 7, 5, 1365, 10, 3, 0],
                             [2, 4, 10, 1, 6, 1, 0, 1459, 1, 19],
                             [2, 7, 11, 11, 6, 10, 4, 12, 1290, 4],
                             [5, 6, 6, 12, 20, 0, 0, 38, 6, 1327]])

class_names = np.arange(10)  # Assuming the classes are labeled from 0 to 9

# Plot confusion matrix
plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
plt.colorbar()

# Add labels and ticks
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)

# Add annotations
thresh = confusion_matrix.max() / 2
for i in range(confusion_matrix.shape[0]):
    for j in range(confusion_matrix.shape[1]):
        plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if confusion_matrix[i, j] > thresh else "black")

plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.title('Confusion Matrix of RBF SVM')
plt.tight_layout()
plt.show()
```



Now we will try to fit another model - Linear kernel SVM and see how it scores.

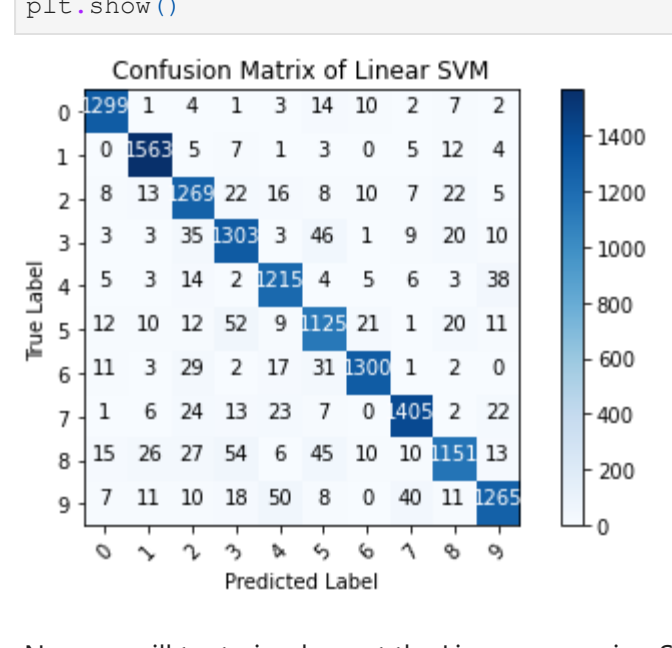
In [25]:
```
# Train and evaluate SVM with linear kernel
svm_linear = SVC(kernel='linear', C=1)
start_time = time.time()
svm_linear.fit(X_train_std, y_train)
training_time_linear = time.time() - start_time

y_pred_linear = svm_linear.predict(X_test_std)
accuracy_linear = accuracy_score(y_test, y_pred_linear)
f1_linear = f1_score(y_test, y_pred_linear, average='macro')
confusion_linear = confusion_matrix(y_test, y_pred_linear)
```

In [26]:
```
print('Accuracy_linear: ', accuracy_linear)
print('F1 score_linear: ', f1_linear)
print('Training time_linear: ', training_time_linear)
print(confusion_linear)
```

```
Accuracy_linear:  0.9210714285714285
F1 score_linear:  0.9198935548264234
Training time_linear:  587.7250008583069
[[1299    1    4    1    3   14   10    2    7    2]
 [   0 1563    5    7    1    3    0    5   12    4]
 [   8   13 1269   22   16    8   10    7   22    5]
 [   3    3   35 1303    3   46    1    9   20   10]
 [   5    3   14    2 1215    4    5    6    3   38]
 [  12   10   12   52    9 1125   21    1   20   11]
 [  11    3   29    2   17   31 1300    1    2    0]
 [   1    6   24   13   23    7    0 1405    2   22]
 [  15   26   27   54    6   45   10   10 1151   13]
 [   7   11   10   18   50    8    0   40   11 1265]]
```

From the above code snippet, we see that the Linear kernel SVM achieved an accuracy of 0.92, an f1 score of 0.92 and took 587 seconds to train. This is comparitively less accuracy compared to the RBF kernel SVM. Below I tried to plot the confusion matrix to visualize it in a graphical format.

In [34]:
```
# Define the confusion matrix
confusion_matrix = np.array([[1299, 1, 4, 1, 3, 14, 10, 2, 7, 2],
                             [0, 1563, 5, 7, 1, 3, 0, 5, 12, 4],
                             [8, 13, 1269, 22, 16, 8, 10, 7, 22, 5],
                             [3, 3, 35, 1303, 3, 46, 1, 9, 20, 10],
                             [5, 3, 14, 2, 1215, 4, 5, 6, 3, 38],
                             [12, 10, 12, 52, 9, 1125, 21, 1, 20, 11],
                             [11, 3, 29, 2, 17, 31, 1300, 1, 2, 0],
                             [1, 6, 24, 13, 23, 7, 0, 1405, 2, 22],
                             [15, 26, 27, 54, 6, 45, 10, 10, 1151, 13],
                             [7, 11, 10, 18, 50, 8, 0, 40, 11, 1265]])

# Define class labels
labels = np.arange(10)

# Plot the confusion matrix
plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix of Linear SVM')
plt.colorbar()
tick_marks = np.arange(len(labels))
plt.xticks(tick_marks, labels, rotation=45)
plt.yticks(tick_marks, labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')

# Display the values in the matrix
thresh = confusion_matrix.max() / 2
for i in range(confusion_matrix.shape[0]):
    for j in range(confusion_matrix.shape[1]):
        plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if confusion_matrix[i, j] > thresh else "black")

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



Now we will try to implement the Linear regression SVM algorithm that the authors are talking about.

In [27]:
```
#Train and evaluate the dataset with Linear regression SVM that the authors are talking about
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

svm_lrsvm = LinearSVC(C=1)
start_time = time.time()
svm_lrsvm.fit(X_train_std, y_train)
training_time_lrsvm = time.time() - start_time
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

In [28]:
```
y_pred_lrsvm = svm_lrsvm.predict(X_test_std)
accuracy_lrsvm = accuracy_score(y_test, y_pred_lrsvm)
f1_score_lrsvm = f1_score(y_test, y_pred_lrsvm, average='macro')
confusion_lrsvm = confusion_matrix(y_test, y_pred_lrsvm)
```

In [30]:
```
# Train and evaluate neural network
import tensorflow as tf
from tensorflow.keras import models, layers
```

In [31]:
```
# Build the neural network
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

Sequential() is used to initialize a sequential model, which is a linear stack of layers where each layer communicates directly with the next one. Dense() adds a fully connected layer to the model, with 512 neurons and uses the ReLU. The images we serve to the neural network are all flattened to 1D array.

Softmax activation is used for representing the probabilities of each class in the classification task. The class with highest probability is chosen as the predicted class for a given input. Mathematically, softmax function is represented as: softmax(x) = (e^x) / (sum(e^x))

In [32]:
```
# Compile the model
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

In the above code snippet, RMSProp Optimizer is used, which is a commonly used optimizer for neural networks. The 'loss' parameter is set tp 'categorical crossentropy' which is commonly used for multi class classification and it measures the dissimilarity between predicted class labels and the true class labels.

In [33]:
```
# Preprocess the data
X_train_nn = X_train.astype('float32') / 255 #normalizing the data as NNs are sensitive to unnormalized data
X_test_nn = X_test.astype('float32') / 255
y_train_nn = tf.keras.utils.to_categorical(y_train, 10)
y_test_nn = tf.keras.utils.to_categorical(y_test, 10)
```

In [34]:
```
# Train the model
start_time = time.time()
history = model.fit(X_train_nn, y_train_nn, epochs=10, batch_size=128, validation_data=(X_test_nn, y_test_nn))
training_time_nn = time.time() - start_time
```

```
Epoch 1/10
438/438 [==============================] - 7s 14ms/step - loss: 0.2644 - accuracy: 0.9233 - val_loss: 0.1586 - val_accuracy: 0.9531
Epoch 2/10
438/438 [==============================] - 6s 13ms/step - loss: 0.1079 - accuracy: 0.9682 - val_loss: 0.1140 - val_accuracy: 0.9669
Epoch 3/10
438/438 [==============================] - 6s 13ms/step - loss: 0.0714 - accuracy: 0.9792 - val_loss: 0.0877 - val_accuracy: 0.9743
Epoch 4/10
438/438 [==============================] - 6s 13ms/step - loss: 0.0516 - accuracy: 0.9845 - val_loss: 0.0794 - val_accuracy: 0.9766
Epoch 5/10
438/438 [==============================] - 7s 17ms/step - loss: 0.0381 - accuracy: 0.9887 - val_loss: 0.0872 - val_accuracy: 0.9738
Epoch 6/10
438/438 [==============================] - 9s 20ms/step - loss: 0.0285 - accuracy: 0.9917 - val_loss: 0.0752 - val_accuracy: 0.9781
Epoch 7/10
438/438 [==============================] - 9s 20ms/step - loss: 0.0220 - accuracy: 0.9938 - val_loss: 0.0772 - val_accuracy: 0.9776
Epoch 8/10
438/438 [==============================] - 8s 17ms/step - loss: 0.0166 - accuracy: 0.9956 - val_loss: 0.0862 - val_accuracy: 0.9766
Epoch 9/10
438/438 [==============================] - 6s 13ms/step - loss: 0.0129 - accuracy: 0.9966 - val_loss: 0.0779 - val_accuracy: 0.9795
Epoch 10/10
438/438 [==============================] - 9s 20ms/step - loss: 0.0090 - accuracy: 0.9979 - val_loss: 0.0789 - val_accuracy: 0.9786
```

In [35]:
```
# Evaluate the model
accuracy_nn = model.evaluate(X_test_nn, y_test_nn)[1]
y_pred_nn = model.predict(X_test_nn)
y_pred_nn = np.argmax(y_pred_nn, axis=1)
f1_nn = f1_score(y_test, y_pred_nn, average='macro')
confusion_nn = confusion_matrix(y_test, y_pred_nn)
```

```
438/438 [==============================] - 2s 4ms/step - loss: 0.0789 - accuracy: 0.9786
438/438 [==============================] - 2s 4ms/step
```

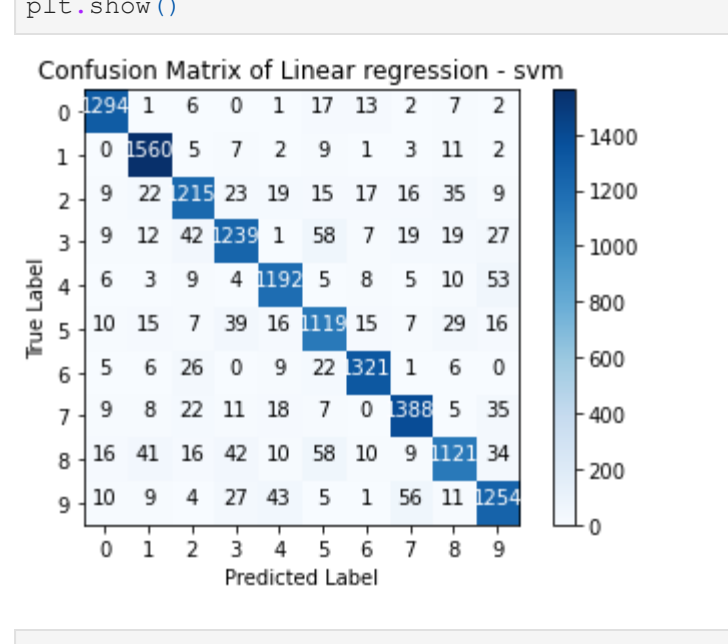
In [37]:
```
print("Linear Regression SVM:")
print(f"Accuracy_LRSVM: {accuracy_lrsvm:.3f}")
print(f"F1 Score_LRSVM: {f1_score_lrsvm:.3f}")
print(f"Training Time_LRSVM: {training_time_lrsvm:.3f} seconds")
print(f"Confusion Matrix: \n{confusion_lrsvm}")
```

```
Linear Regression SVM:
Accuracy_LRSVM: 0.907
F1 Score_LRSVM: 0.906
Training Time_LRSVM: 967.631 seconds
Confusion Matrix:
[[1294    1    6    0    1   17   13    2    7    2]
 [   0 1560    5    7    2    9    1    3   11    2]
 [   9   22 1215   23   19   15   17   16   35    9]
 [   9   12   42 1239    1   58    7   19   19   27]
 [   6    3    9    4 1192    5    8    5   10   53]
 [  10   15    7   39   16 1119   15    7   29   16]
 [   5    6   26    0    9   22 1321    1    6    0]
 [   9    8   22   11   18    7    0 1388    5   35]
 [  16   41   16   42   10   58   10    9 1121   34]
 [  10    9    4   27   43    5    1   56   11 1254]]
```

We can see that it got an accuracy of 0.907, an f1 score of 0.906 and took 967 seconds to train on the dataset. From these results, we can conclude that the Linear regression-svm algorithm doesn't perform as good on the MNIST dataset as the authors claim.

We will proceed to implement neural networks on the dataset and see how they fare. I chose neural networks because they are now being extensively used for large scale classification and I want to see how they fare on the MNIST dataset and compare them to the Linear regression-SVM and other traditional SVM models.

In [36]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the confusion matrix
confusion_matrix = np.array([[1294, 1, 6, 0, 1, 17, 13, 2, 7, 2],
                             [0, 1560, 5, 7, 2, 9, 1, 3, 11, 2],
                             [9, 22, 1215, 23, 19, 15, 17, 16, 35, 9],
                             [9, 12, 42, 1239, 1, 58, 7, 19, 19, 27],
                             [6, 3, 9, 4, 1192, 5, 8, 5, 10, 53],
                             [10, 15, 7, 39, 16, 1119, 15, 7, 29, 16],
                             [5, 6, 26, 0, 9, 22, 1321, 1, 6, 0],
                             [9, 8, 22, 11, 18, 7, 0, 1388, 5, 35],
                             [16, 41, 16, 42, 10, 58, 10, 9, 1121, 34],
                             [10, 9, 4, 27, 43, 5, 1, 56, 11, 1254]])

# Plot the confusion matrix
plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix of Linear regression - svm')
plt.colorbar()
tick_marks = np.arange(10)
plt.xticks(tick_marks, tick_marks)
plt.yticks(tick_marks, tick_marks)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')

# Display the value inside each cell
thresh = confusion_matrix.max() / 2.0
for i in range(10):
    for j in range(10):
        plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if confusion_matrix[i, j] > thresh else "black")

plt.show()
```

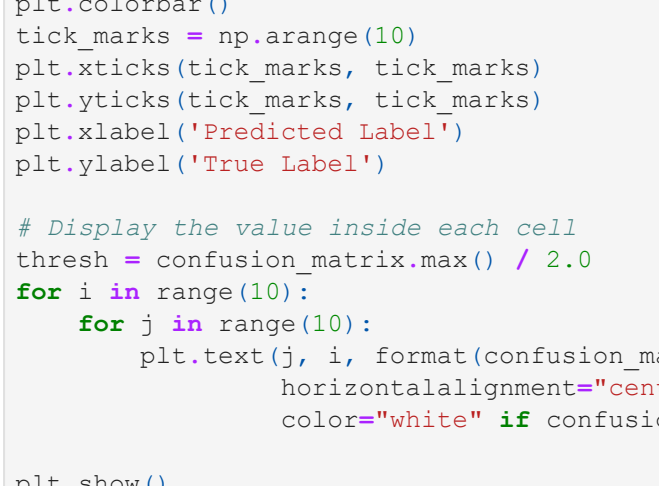


In [38]:
```
print("Neural Network:")
print(f"Accuracy_FFNN: {accuracy_nn:.3f}")
print(f"F1 Score_FFNN: {f1_nn:.3f}")
print(f"Training Time_FFNN: {training_time_nn:.3f} seconds")
print(f"Confusion Matrix: \n{confusion_nn}")
```

```
Neural Network:
Accuracy_FFNN: 0.979
F1 Score_FFNN: 0.978
Training Time_FFNN: 71.616 seconds
Confusion Matrix:
[[1313    1    3    0    1    3    8    3    8    3]
 [   0 1588    1    1    1    0    0    6    3    0]
 [   0    5 1326    3    2    2    4   24   12    2]
 [   0    0    7 1384    0   19    0    7    7    9]
 [   0    0    3    0 1266    0    0    5    1   20]
 [   0    0    0    5    2 1251    7    1    6    1]
 [   2    1    0    0    5    7 1379    0    2    0]
 [   1    1    4    0    2    1    0 1490    0    4]
 [   2    4    3    3    4    5    3    5 1322    6]
 [   2    3    1    2    9    3    0   13    6 1381]]
```

For the feedforward neural networks, we can see that we got an accuracy of 0.979 and an f1 score of 0.978. It took just 71.6 seconds to train on the dataset (compare that with the 967 seconds the LR-SVM model took. From this result, we can safely conclude that the Feedforward NN is much better than the Linear regression-svm algorithm, in all the parameters that we considered.

In [39]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the confusion matrix
confusion_matrix = np.array([[1313, 1, 3, 0, 1, 3, 8, 3, 8, 3],
                             [0, 1588, 1, 1, 1, 0, 0, 6, 3, 0],
                             [0, 5, 1326, 3, 2, 2, 4, 24, 12, 2],
                             [0, 0, 7, 1384, 0, 19, 0, 7, 7, 9],
                             [0, 0, 3, 0, 1266, 0, 0, 5, 1, 20],
                             [0, 0, 0, 5, 2, 1251, 7, 1, 6, 1],
                             [2, 1, 0, 0, 5, 7, 1379, 0, 2, 0],
                             [1, 1, 4, 0, 2, 1, 0, 1490, 0, 4],
                             [2, 4, 3, 3, 4, 5, 3, 5, 1322, 6],
                             [2, 3, 1, 2, 9, 3, 0, 13, 6, 1381]])

# Plot the confusion matrix
plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix of FFNN')
plt.colorbar()
tick_marks = np.arange(10)
plt.xticks(tick_marks, tick_marks)
plt.yticks(tick_marks, tick_marks)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')

# Display the value inside each cell
thresh = confusion_matrix.max() / 2.0
for i in range(10):
    for j in range(10):
        plt.text(j, i, format(confusion_matrix[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if confusion_matrix[i, j] > thresh else "black")

plt.show()
```



MLPRegressor is a type of Feedforward neural network, but it is mostly used for regression tasks. It implements backpropagation, where the weights are adjusted based on the errors between predicted and true target values. This iterative process of forward pass, error calculation and backward pass is repeated until the model converges to optimal weights.

$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b)$ where y is the output, $x_1, x_2, ..., x_n$ are the inputs, $w_1, w_2,..., w_n$ are the corresponding weights, b is the bias term,and f() represents the activation function.

The gradient descent formula is typically used to update the weight in backpropagation: $\Delta w = \eta\ \delta\ x$, where Δw is the change in weights, η is the learning rate, δ is the error gradient, and x is the input.

In [39]:
```python
from sklearn.neural_network import MLPRegressor
# Initialize MLPRegressor with hidden_layer_sizes=(100,), learning_rate_init=0.01, max_iter=500
mlp = MLPRegressor(hidden_layer_sizes=(100,), learning_rate_init=0.01, max_iter=500)
```

In [40]:
```python
# Train MLPRegressor
start_time = time.time()
mlp.fit(X_train_std, y_train)
training_time_mlp = time.time() - start_time
```

In [41]:
```python
# Predict using MLPRegressor
y_pred_mlp = mlp.predict(X_test_std)
```

In [42]:
```python
# Convert predicted labels to integer type
y_pred_mlp = np.around(y_pred_mlp).astype(np.int)
```

```
C:\Users\Checkout\AppData\Local\Temp\ipykernel_10028\3062492246.py:2: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred_mlp = np.around(y_pred_mlp).astype(np.int)
```

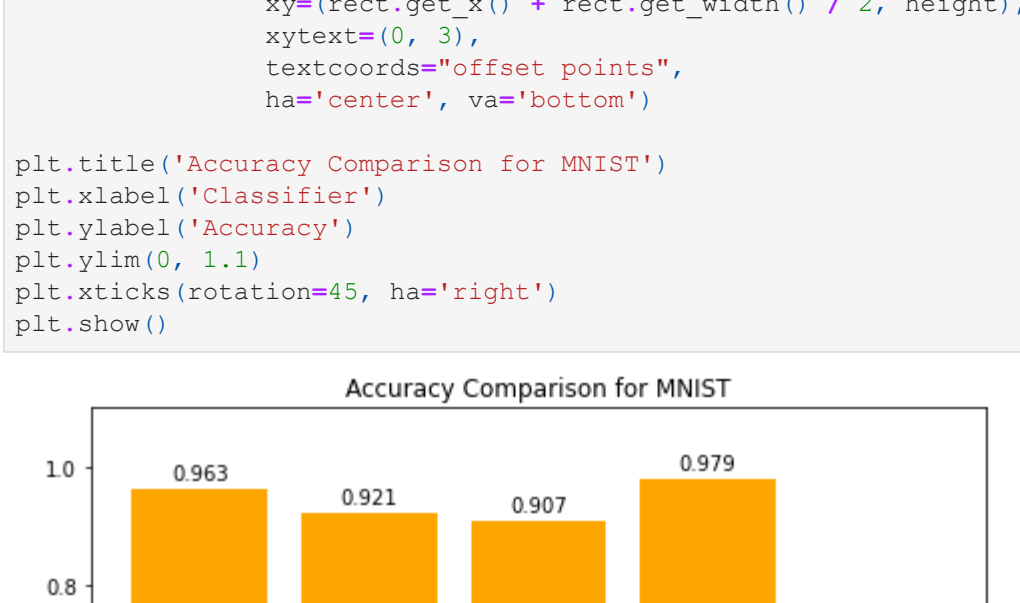
In [43]:
```python
# Calculate accuracy metrics
accuracy_mlp = accuracy_score(y_test, y_pred_mlp)
f1_mlp = f1_score(y_test, y_pred_mlp, average='macro')
confusion_mlp = confusion_matrix(y_test, y_pred_mlp)
```

In [40]:
```python
#print performance metrics
print('Accuracy_MLP: ', accuracy_mlp)
print('F1 Score_MLP: ', f1_mlp)
print('Training Time_MLP: ', training_time_mlp, ' seconds')
```

```
Accuracy_MLP:  0.3395
F1 Score_MLP:  0.09934170762503548
Training Time_MLP:  39.84637975692749  seconds
```

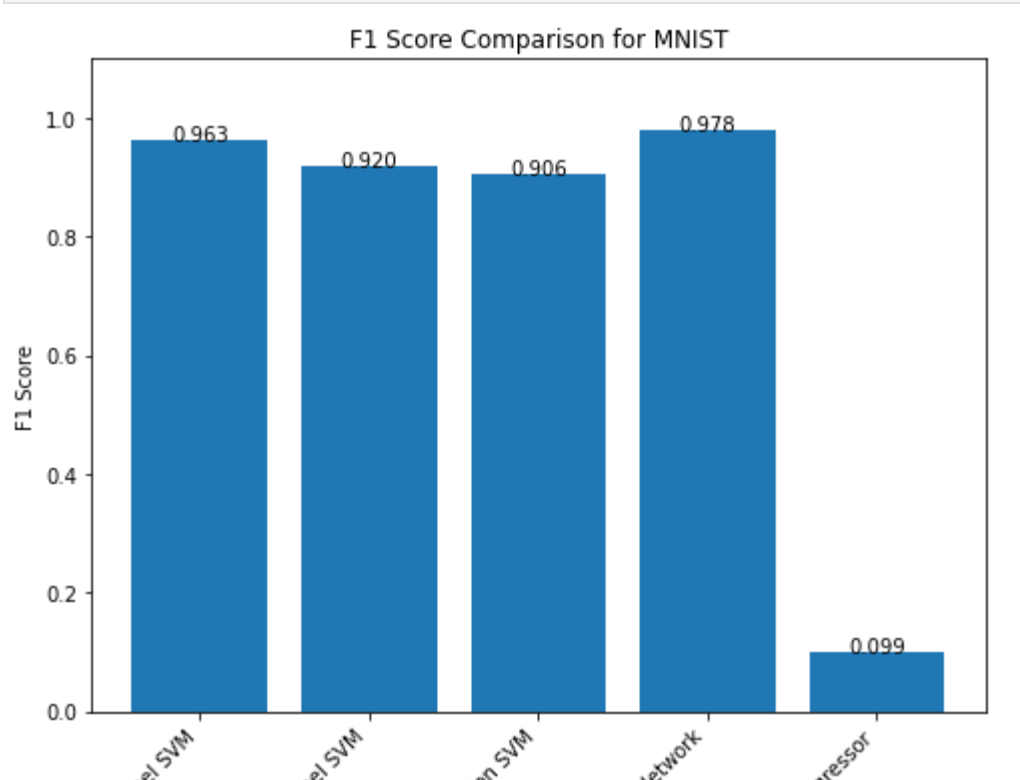
For the MLPRegressor, we got very poor results - accuracy of just 0.34, and an f1 score of 0.09. May be, the algorithm is simply not suited for classification tasks, since it is designed for regression. Since the authors in their paper considered linear regression, I thought, MLPRegressor might work. But it did not. Or may be, more work on the hyperparameter tuning could help us achieve good results. I will work on this further to see how it goes.

In [84]:
```python
import matplotlib.pyplot as plt
# Create the plots
fig, ax = plt.subplots(figsize=(8, 6))
rects = ax.bar(['RBF Kernel SVM', 'Linear Kernel SVM', 'Linear Regression SVM', 'Feedforward Neural Network', '
               [accuracy_rbf, accuracy_linear,  accuracy_lrsvm, accuracy_nn, accuracy_mlp],
              color = 'orange')

# Display the number on top of the bar
for rect in rects:
    height = rect.get_height()
    ax.annotate('{}'.format(round(height, 3)),
                xy=(rect.get_x() + rect.get_width() / 2, height),
                xytext=(0, 3),
                textcoords="offset points",
                ha='center', va='bottom')

plt.title('Accuracy Comparison for MNIST')
plt.xlabel('Classifier')
plt.ylabel('Accuracy')
plt.ylim(0, 1.1)
plt.xticks(rotation=45, ha='right')
plt.show()
```



From the above graph of accuracy, we can clearly see FF Neural network achieved the highest accuracy, followed by RBF kernel SVM and Linear kernel SVM. The Linear regression SVM did not get better accuracy than the other two traditional SVMs, as the authors claimed in their paper.

In [85]:
```python
# Create the plots
plt.figure(figsize=(8, 6))

x = ['RBF Kernel SVM', 'Linear Kernel SVM', 'Linear Regression SVM', 'Feedforward Neural Network', 'MLPRegresso
y = [f1_rbf, f1_linear, f1_score_lrsvm, f1_nn, f1_mlp]

plt.bar(x, y)
plt.title('F1 Score Comparison for MNIST')
plt.xlabel('Classifier')
plt.ylabel('F1 Score')
plt.ylim(0, 1.1)

# Add value labels on top of bars
for i, v in enumerate(y):
    plt.text(i, v, "{:.3f}".format(v), color='black', ha='center')

# Rotate x-axis labels to avoid overlapping
plt.xticks(rotation=45, ha='right')

plt.show()
```



For the F1 scores too, FF Neural network achieved the highest followed by RBF Kernel. Linear regression-svm again did not get better F1 scores than the traditional SVM algorithms. unlike what the authors had claimed. MLPRegressor sadly faired the worst compared to all the models.

In [79]:
```python
# Create the plots
fig, ax = plt.subplots(figsize=(8, 6))
rects = ax.bar(['RBF Kernel SVM', 'Linear Kernel SVM','Linear Regression SVM', 'Feedforward Neural Network', 'M
        [training_time_rbf, training_time_linear, training_time_lrsvm, training_time_nn, training_time_mlp],
              color= 'purple')

# Add labels on top of the bars
for rect in rects:
    height = rect.get_height()
    ax.annotate(f'{height:.2f}', xy=(rect.get_x() + rect.get_width() / 2, height),
                xytext=(0, 3), textcoords="offset points", ha='center', va='bottom', fontsize=10)

plt.title('Training Time Comparison for MNIST')
plt.xlabel('Classifier')
plt.ylabel('Training Time (seconds)')

# Tilt the x-axis labels
plt.xticks(rotation=45, ha='right')

plt.show()
```

Linear regression-svm surprisingly took the longest time to train, followed by Linear kernel SVM and RBF kernel svm. FF Neural network took very less time to train, while having the highest accuracy of all. MLPRegressor took the least time to train, but given its poor accuracy, we cannot consider it.

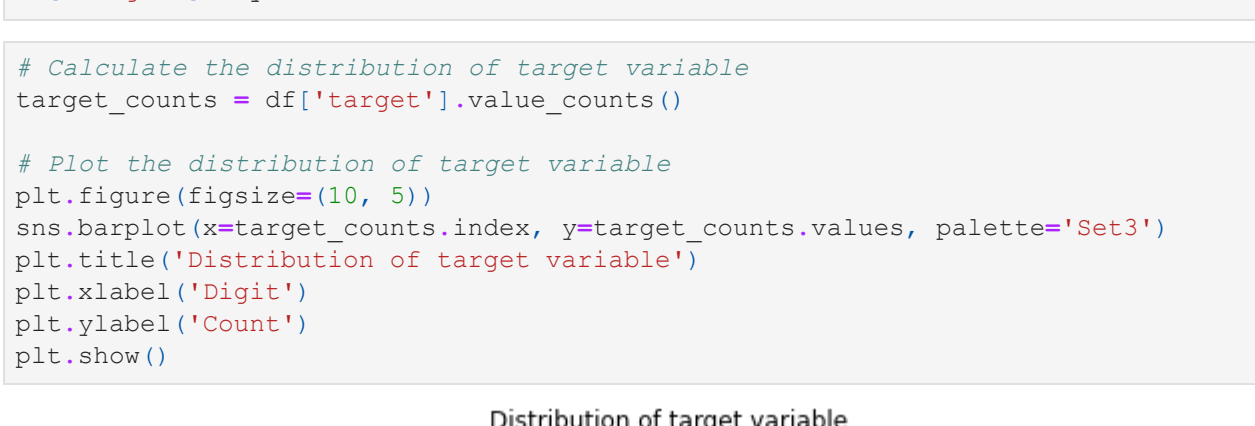
USPS

In [4]:
```python
# Load the USPS dataset
usps = fetch_openml('usps', version=1)
# Preprocess the data
X = usps['data']
y = usps['target']
X = X / 255.0    # scale the pixel values to [0, 1]
y = y.astype(int)
```

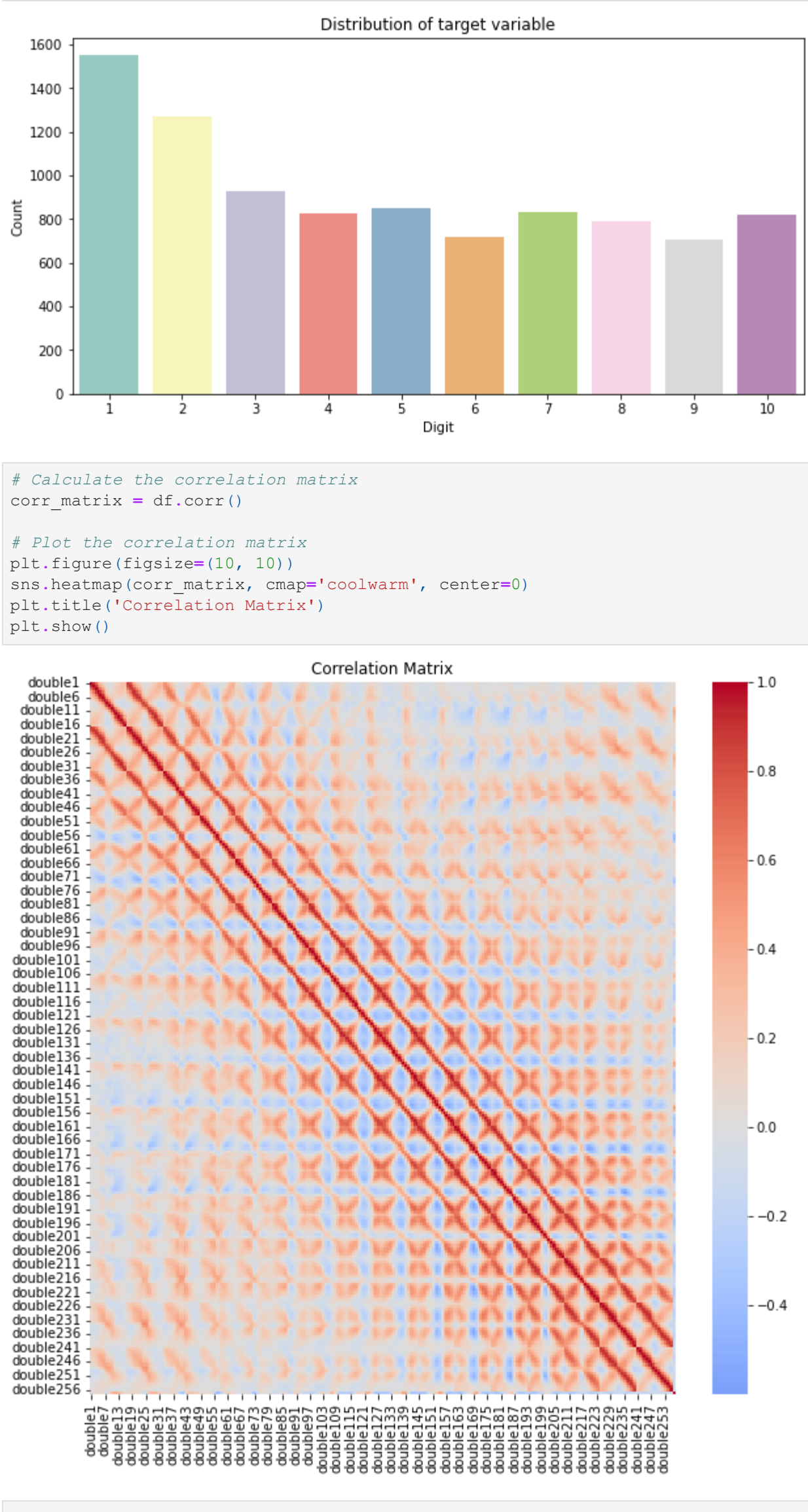
```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\datasets\_openml.py:879: UserWarning: Version 1 of dataset USPS is inactive, meaning that issues have been found in the dataset. Try using a newer version from this URL: https://api.openml.org/data/v1/download/18805612/USPS.arff
  warn(
```

In [5]:
```python
# Convert the feature matrix into a DataFrame
df = pd.DataFrame(X)
# Add the target variable as a new column
df['target'] = y
```

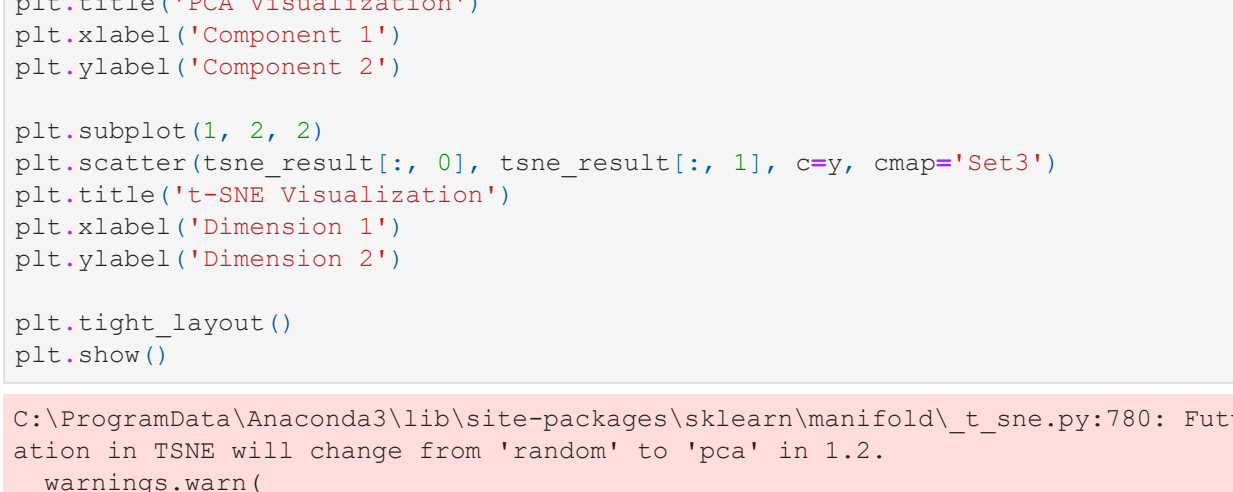
In [6]:
```python
# Calculate the distribution of target variable
target_counts = df['target'].value_counts()

# Plot the distribution of target variable
plt.figure(figsize=(10, 5))
sns.barplot(x=target_counts.index, y=target_counts.values, palette='Set3')
plt.title('Distribution of target variable')
plt.xlabel('Digit')
plt.ylabel('Count')
plt.show()
```



In [7]:
```python
# Calculate the correlation matrix
corr_matrix = df.corr()

# Plot the correlation matrix
plt.figure(figsize=(10, 10))
sns.heatmap(corr_matrix, cmap='coolwarm', center=0)
plt.title('Correlation Matrix')
plt.show()
```

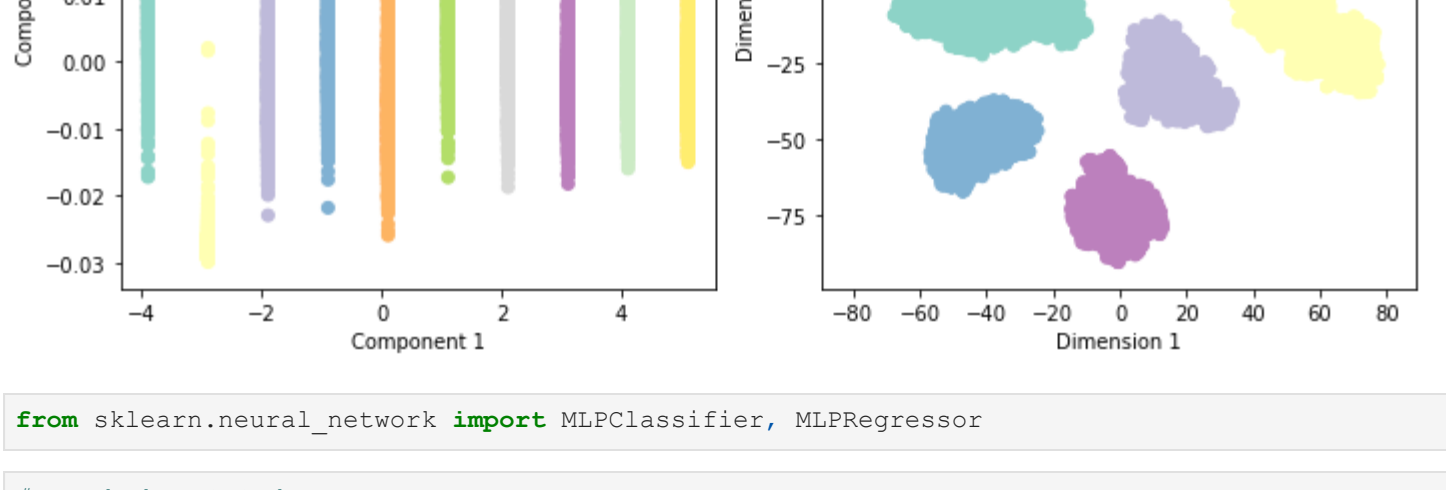


In [16]:
```python
plt.figure(figsize=(10, 5))
sns.boxplot(x='target', y='target', data=df)  # Replace 'y' with the desired column name
plt.title('Pixel Value Distribution by Digit')
plt.xlabel('Digit')
plt.ylabel('Pixel Value')
plt.show()
```



In [10]:
```python
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# Apply PCA for dimensionality reduction
pca = PCA(n_components=2)
pca_result = pca.fit_transform(X)

# Apply t-SNE for dimensionality reduction
tsne = TSNE(n_components=2)
tsne_result = tsne.fit_transform(X)

# Plot PCA and t-SNE results
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=y, cmap='Set3')
plt.title('PCA Visualization')
plt.xlabel('Component 1')
plt.ylabel('Component 2')

plt.subplot(1, 2, 2)
plt.scatter(tsne_result[:, 0], tsne_result[:, 1], c=y, cmap='Set3')
plt.title('t-SNE Visualization')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')

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

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:780: FutureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:790: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



In [20]:
```python
from sklearn.neural_network import MLPClassifier, MLPRegressor
```

In [21]:
```python
# Load the USPS dataset
usps = fetch_openml('usps', version=1)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\datasets\_openml.py:879: UserWarning: Version 1 of dataset USPS is inactive, meaning that issues have been found in the dataset. Try using a newer version from this URL: https://api.openml.org/data/v1/download/18805612/USPS.arff
  warn(
```

In [22]:
```python
# Preprocess the data
X = usps['data']
y = usps['target']
X = X / 255.0    # scale the pixel values to [0, 1]
y = y.astype(int)
```

In [23]:
```python
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [24]:
```python
class LinearRegressionSVM:
    def __init__(self, learning_rate=0.001, C=1.0, subset_size=50000, max_iterations=100):
        self.learning_rate = learning_rate
        self.C = C
        self.subset_size = subset_size
        self.max_iterations = max_iterations

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.coef_ = np.zeros(n_features)
        self.intercept_ = 0.0

        for iteration in range(self.max_iterations):
            subset_indices = np.random.choice(range(n_samples), self.subset_size, replace=False)
            X_subset = X[subset_indices]
            y_subset = np.array(y[subset_indices])  # convert y_subset to numpy array
            margin = y_subset * (np.dot(X_subset, self.coef_) + self.intercept_)
            if np.any(margin <= 0):
                misclassified_indices = np.where(margin <= 0)[0]
                random_misclassified_index = np.random.choice(misclassified_indices)
                self.coef_ += self.learning_rate * (np.dot(X_subset[random_misclassified_index], y_subset[rando
                self.intercept_ += self.learning_rate * y_subset[random_misclassified_index]

    def predict(self, X):
        y_pred = np.sign(np.dot(X, self.coef_) + self.intercept_)
        return y_pred
```

In [25]:
```python
clf_lsvc = LinearSVC(random_state=42)
start_time = time.time()
clf_lsvc.fit(X_train, y_train)
train_time_lsvc = time.time() - start_time
```

In [26]:
```python
y_pred_lsvc = clf_lsvc.predict(X_test)
accuracy_lsvc = accuracy_score(y_test, y_pred_lsvc)
f1_lsvc = f1_score(y_test, y_pred_lsvc, average='weighted')
confusion_lsvc = confusion_matrix(y_test, y_pred_lsvc)
```

In [28]:
```python
# RBF kernel SVM classifier
clf_rbf_svm = SVC(kernel='rbf', random_state=42)
start_time = time.time()
clf_rbf_svm.fit(X_train, y_train)
train_time_rbf_svm = time.time() - start_time
```

In [29]:
```python
y_pred_rbf_svm = clf_rbf_svm.predict(X_test)
accuracy_rbf_svm = accuracy_score(y_test, y_pred_rbf_svm)
f1_rbf_svm = f1_score(y_test, y_pred_rbf_svm, average='weighted')
confusion_rbf_svm = confusion_matrix(y_test, y_pred_rbf_svm)
```

In [27]:
```python
print("Linear Regression SVM: ")
print("Accuracy: ", accuracy_lsvc)
print("F1 score: ", f1_lsvc)
print("Training time: ", train_time_lsvc)
```

```
Linear Regression SVM:
Accuracy:  0.7951612903225806
F1 score:  0.7855664377675992
Training time:  3.8985209465026855
```

For the linear regression SVM algorithm on the USPS dataset, we got an accuracy of 0.795, f1 score of 0.785 and it took 3.89 seconds to train on the USPS dataset. Now let's try to implement other traditional SVM algorithms on the dataset and see how they fair.

In [30]:
```python
print("SVM RBF: ")
print("Accuracy: ", accuracy_rbf_svm)
print("F1 score: ", f1_rbf_svm)
print("Training time: ", train_time_rbf_svm)
```

```
SVM RBF:
Accuracy:  0.9741935483870968
F1 score:  0.9741777385617408
Training time:  3.099534034729004
```

For the RBF SVM, we got an impressive accuracy of 0.974, f1 score of 0.974 and it took 3.09 seconds to train on the dataset. This traditional SVM model performed far better than the linear regression-svm model that the authors proposed. Now let's try to implement the Linear kernel SVM and see how it goes.

In [33]:
```python
# Linear kernel SVM classifier
clf_lin_svm = SVC(kernel='linear', random_state=42)
start_time = time.time()
clf_lin_svm.fit(X_train, y_train)
train_time_lin_svm = time.time() - start_time
```

In [34]:
```python
y_pred_lin_svm = clf_lin_svm.predict(X_test)
accuracy_lin_svm = accuracy_score(y_test, y_pred_lin_svm)
f1_lin_svm = f1_score(y_test, y_pred_lin_svm, average='weighted')
confusion_lin_svm = confusion_matrix(y_test, y_pred_lin_svm)
```

In [35]:
```python
print("Linear SVM: ")
print("Accuracy: ", accuracy_lin_svm)
print("F1 score: ", f1_lin_svm)
print("Training time: ", train_time_lin_svm)
```

```
Linear SVM:
Accuracy:  0.30483870967741933
F1 score:  0.1485371329226645
Training time:  24.030107021331787
```

For the linear SVM, we got an accuracy of just 0.30, f1 score of 0.14 and it took 24 seconds to train on the dataset. This model performed poorer than the linear regression-svm model that the authors are talking about. Now let's try to implement FF neural network and MLPRegressor and see how it goes.

In [36]:
```python
# Feedforward neural network classifier
clf_ffnn = MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', random_state=42)
start_time = time.time()
clf_ffnn.fit(X_train, y_train)
train_time_ffnn = time.time() - start_time
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
```

Used ReLU activation for the FF neural network and adam solver with a random state number of 42.

In [37]:
```python
y_pred_ffnn = clf_ffnn.predict(X_test)
accuracy_ffnn = accuracy_score(y_test, y_pred_ffnn)
f1_ffnn = f1_score(y_test, y_pred_ffnn, average='weighted')
confusion_ffnn = confusion_matrix(y_test, y_pred_ffnn)
```

In [38]:
```python
print("Feedforward Neural network: ")
print("Accuracy: ", accuracy_ffnn)
print("F1 score: ", f1_ffnn)
print("Training time: ", train_time_ffnn)
```

```
Feedforward Neural network:
Accuracy:  0.9370967741935484
F1 score:  0.9372480545259878
Training time:  39.62293195724487
```

For the Feedforward NN, we got an impressive accuracy of 0.93, f1 score of 0.93 and it took 39 seconds to train on the data. This is much better than the linear regression-svm model.

In [ ]:
```python
# MLP regressor
reg_bpnn = MLPRegressor(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', random_state=42)
start_time = time.time()
reg_bpnn.fit(X_train, y_train)
train_time_bpnn = time.time() - start_time
```

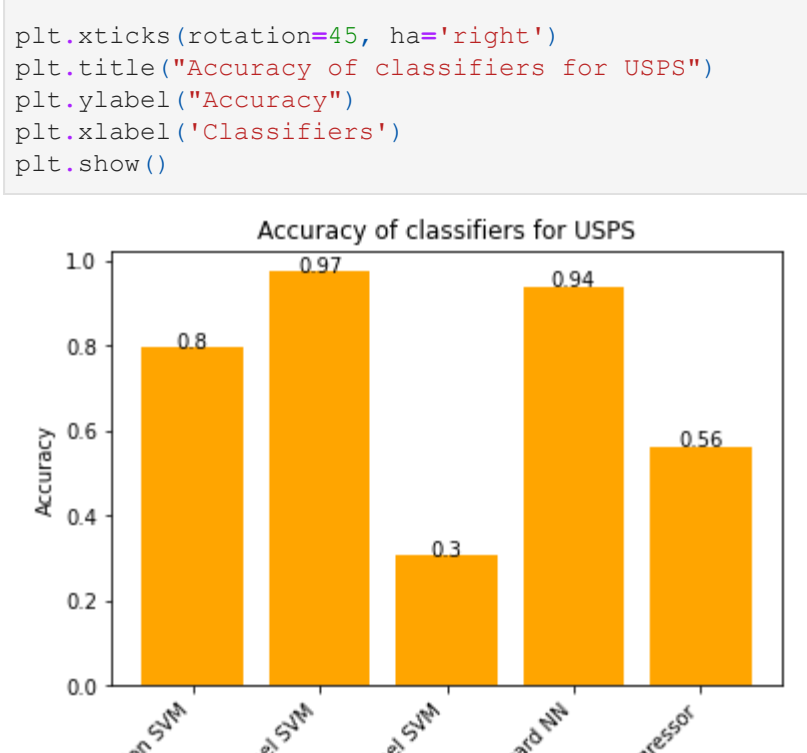
In [40]:
```python
y_pred_bpnn = np.round(reg_bpnn.predict(X_test)).astype(int)
accuracy_bpnn = accuracy_score(y_test, y_pred_bpnn)
f1_bpnn = f1_score(y_test, y_pred_bpnn, average='weighted')
confusion_bpnn = confusion_matrix(y_test, y_pred_bpnn)
```

In [41]:
```python
print("MLPRegressor: ")
print("Accuracy: ", accuracy_bpnn)
print("F1 score: ", f1_bpnn)
print("Training time: ", train_time_bpnn)
```

```
MLPRegressor:
Accuracy:  0.560752688172043
F1 score:  0.5715679057551382
Training time:  35.36492156982422
```

For the MLPRegressor, we got an accuracy of 0.56, f1 score of 0.57 and it took 35 seconds to train on the dataset. This is not as good as the linear regression-svm algorithm, but certainly better than the linear kernel SVM.

In [42]:
```python
import matplotlib.pyplot as plt

# Accuracy plot
plt.bar(["Linear Regression SVM", "RBF kernel SVM", "Linear kernel SVM", "Feedforward NN", "MLPRegressor"],
        [accuracy_lsvc, accuracy_rbf_svm, accuracy_lin_svm, accuracy_ffnn, accuracy_bpnn],
       color = 'orange')
for i, v in enumerate([accuracy_lsvc, accuracy_rbf_svm, accuracy_lin_svm, accuracy_ffnn, accuracy_bpnn]):
    plt.text(i, v, str(round(v, 2)), color='black', ha='center')

plt.xticks(rotation=45, ha='right')
plt.title("Accuracy of classifiers for USPS")
plt.ylabel("Accuracy")
plt.xlabel('Classifiers')
plt.show()
```

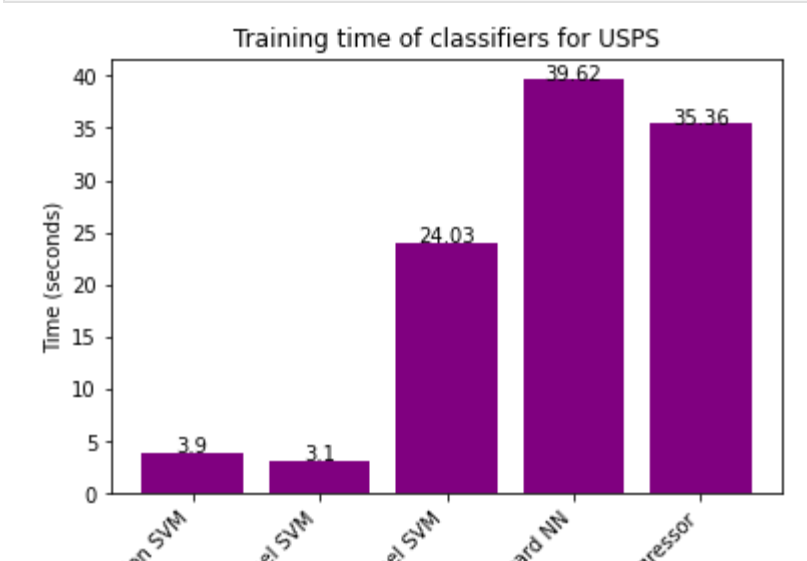


From the above graph, we can clearly see that RBF kernel SVM achieved the highest accuracy on the USPS dataset, followed by FF Neural network. Linear regression-svm model did well by getting an accuracy of 0.8, but still less than the traditional RBF kernel svm model.

In [43]:
```python
# F1 score plot
plt.bar(["Linear Regression SVM", "RBF kernel SVM", "Linear kernel SVM", "Feedforward NN", "MLPRegressor"],
        [f1_lsvc, f1_rbf_svm, f1_lin_svm, f1_ffnn, f1_bpnn])
for i, v in enumerate([f1_lsvc, f1_rbf_svm, f1_lin_svm, f1_ffnn, f1_bpnn]):
    plt.text(i, v, str(round(v, 2)), color='black', ha='center')

plt.xticks(rotation=45, ha='right')
plt.title("F1 score of classifiers for USPS")
plt.ylabel("F1 score")
plt.xlabel('Classifiers')
plt.show()
```

This graph looks similar to the one for the accuracy. RBF kernel got the highest with 0.97 followed by FF Neural network with 0.94 and linear regression-svm with 0.79.

In [44]:
```python
# Training time plot
plt.bar(["Linear Regression SVM", "RBF kernel SVM", "Linear kernel SVM", "Feedforward NN", "MLPRegressor"],
        [train_time_lsvc, train_time_rbf_svm, train_time_lin_svm, train_time_ffnn, train_time_bpnn],
       color = 'purple')
for i, v in enumerate([train_time_lsvc, train_time_rbf_svm, train_time_lin_svm, train_time_ffnn, train_time_bpn
    plt.text(i, v, str(round(v, 2)), color='black', ha='center')

plt.xticks(rotation=45, ha='right')
plt.title("Training time of classifiers for USPS")
plt.ylabel("Time (seconds)")
plt.xlabel('Classifiers')
plt.show()
```



FF Neural network took more time (39 seconds) than the linear regression-svm model (3.9 seconds). But given the accuracies and f1-scores of the two models, I guess this is a reasonable tradeoff. RBF kernel SVM performed the best by getting the highest accuracym while training for the least time.