An Academic Internship report on:

Handwritten Digit Recognition using Machine Learning:

A Comparative Analysis

Submitted by:

Rishika Hazarika *(210710007037)*

Shreyoshi Ghosh *(210710007046)*

Shruti Sarma *(210710007047)*

Diksha Borah *(210710307058)*

Department of Computer Science and Engineering

Jorhat Engineering College

Assam Science and Technology University

ACKNOWLEDGEMENT

We extend our heartfelt gratitude to Assam Science & Technology University (ASTU) for incorporating the social internship within our course curriculum. Our sincere appreciation goes to Prof. Rupam Baruah, Principal of Jorhat Engineering College, for facilitating this invaluable academic internship opportunity.

We take this opportunity to express our profound thanks to Mrs. Nilakhi Saikia Madam, for her unwavering support and guidance throughout the project duration. Her provision of abundant resources and knowledge played a pivotal role in our successful completion of the internship, enhancing our skill sets significantly.

Lastly, we acknowledge and appreciate the unwavering support and encouragement of our parents and family members, whose continuous backing was instrumental in the fulfillment of this internship.

SELF DECLARATION

We hereby declare that the work which is being presented in this report entitled "Handwritten Digit Recognition using Machine Learning: A Comparative Analysis" is an authentic record of our own work carried out during July, 2023 to August, 2023. The matter presented in this report has not been submitted elsewhere for any credit or requirement.

Rishika Hazarika **Shreyoshi Ghosh**

(210710007037) **(210710007046)**

Shruti Sarma **Diksha Borah**

(210710007047) **(210710307058)**

ABSTRACT

This academic internship report investigates the realm of Handwritten Digit Recognition through the lens of Machine Learning. Titled "Handwritten Digit Recognition using Machine Learning: A Comparative Analysis," this study focuses on assessing and contrasting different machine learning models. The report details the implementation and evaluation of distinct models, including linear regression, logistic regression, linear Support Vector Machine (SVM), kernel SVM, and Convolutional Neural Networks (CNN). The main objective is to analyze the performance of these models in terms of accuracy, efficiency, and suitability for recognizing handwritten digits. The report elaborates on the methodologies employed, datasets utilized, model training procedures, and a comprehensive comparative analysis of the models' outcomes. By highlighting the strengths and limitations of each model, this comparative analysis aims to provide valuable insights into their practical applicability and effectiveness in the domain of handwritten digit recognition. The findings and conclusions drawn from this comparative study contribute to a deeper understanding of the diverse machine learning approaches and their respective efficacy in pattern recognition and classification tasks.

CONTENTS

1. INTRODUCTION

The technology, throughout these centuries have gone through many revolutionary changes. The technology at this moment has encompassed a variety of knowledge, methods, tools and systems that leverage scientific understanding and engineering principles to create solutions and devices that enhance human capabilities, automate process, enable communication and progress throughout. Among all, one of the domains that has been the most influential is Computer Science. This field has evolved and expanded in unprecedented ways, including a wide range of disciplines and applications. In all of these aspects, Machine Learning have made significant strides, enabling machines to perform tasks that were once exclusive to humans. In this age of digital transformation, the realm of Machine Learning stands as guiding star, illuminating the path to intelligent decision-making.

In the course of our internship, we had the privilege of collaborating with the esteemed faculty member of Jorhat Engineering College that helped us to unravel all the details of the data, algorithms and predictive analytics using a simple implementation of different models in a common problem of our daily life.

1.1 ABOUT

1.1.1 Assam Science and Technology University:

The Assam Science and Technology University was established by Government of Assam under Assam Science & Technology University Act 2009. It aims to provide education and research in the field of science & technology and other professional courses in Assam. ASTU is the premier and only technical university in the North Eastern Region of India. The university is responsible for academic regulation of all undergraduate and post-graduate programs in engineering, and pharmaceutical sciences and a few professional courses in science and management sectors. ASTU also conducts an in-house post-graduate course in Energy Engineering. Since its inception, ASTU has been undertaking high quality teaching and research in frontier areas of science & technology continuously upgrading the syllabi and creating environment for international standard research and emphasizing in bridging the ancient wisdom of the region with modern technology.

1.1.2 **Jorhat Engineering College:**

Jorhat Engineering College, a premiere technical institute of North East India has completed its 62 years of relentless service to the society. The college started functioning with its first batch of students in Civil Engineering on the 10th of October, 1960. The college, affiliated to Assam Science and Technology University, currently offers AICTE recognized B. Tech courses on Civil, Mechanical, Electrical, Computer Science and Instrumentation and M.Tech courses in Civil and Electrical Engineering. The college has been offering a three year post graduate course leading to Masters of Computer Applications since 1987.

1.2 Machine Learning:

Machine Learning is a field of study in artificial intelligence concerned with the development of statistical algorithms that can effectively generalize and perform the tasks without any instructions i.e., at its core, Machine Learning is the process by which computers are equipped with the capacity to learn from data, recognize the patterns, and make decisions and predictions without being explicitly programmed. This field also involves the concept of probability for understanding uncertainty, and decision-making machine learning algorithms. The models that are used for testing or training are often expressed mathematically to represent relationships between features and outcomes. To understand the working of machine learning, few concepts are required:

- **Data** is the foundation of machine learning which are important for the success of a machine learning model. It can be in either structured (databases) or unstructured (text or images).
- **Learning** involves the process by which algorithms improve their performance over time, based on data. This learning can be supervised (with labeled data), unsupervised (without labels) or semi-supervised (a combination).
- **Machine learning algorithms** are mathematical models that transform data into predictions or decisions. These algorithms can be of different types including supervised learning, unsupervised learning and more.
- **Feature scaling** is an important tool that is useful for modeling and prediction of different data attributes.
- **Model training** is the process of feeding the algorithm with data to estimate model parameters.
- **Model evaluation** ensures the model's quality which is evaluated using confusion matrix, accuracy, precision etc.

The working of the machine learning has three main parts:

1. A **decision process** where the algorithms are used to make prediction or classification based on some input data, either labeled or unlabeled.

2. **An error function** the evaluates the prediction of the model i.e., if there are any known examples, an error function can make a comparison to assess the accuracy of the model.

3. A **model optimization** process that continuously evaluates and optimizes the process of fitting the model better to the data points of training set, after which the weights are adjusted to reduce the discrepancy. This process is repeated until a threshold accuracy is has been met.

After knowing the working of the machine learning, let us see what a model is. A**Machine learning model** is a program that can find patterns or make decisions from a dataset. These models are created from machine learning algorithms, which are trained using different types of datasets. This process of creating machine learning models from algorithms include – representing the problem. The three different methods in which a machine learning model falls into are:

1. **Supervised learning** is defined by its use of labeled datasets to train algorithms or predict outcomes accurately. On providing the input data, the model adjusts its weights until it has been fitted appropriately. Some commonly known models are **CNN, linear regression, logistic regression, random forest and support vector machine (SVM).**
2. **Unsupervised learning** uses machine learning algorithms to analyze and cluster unlabeled datasets. These algorithms can discover any hidden patterns or data groupings and develops an ability to discover the similarities and differences in information to further, make it ideal for exploratory data analysis or image recognition or patter recognition. Commonly known models **include ANN, K-means clustering.**
3. **Semi-supervised learning** offers a medium between both supervised and unsupervised learning. It uses a smaller labeled data set to guide classification and feature scaling from a larger, unlabeled data during the course of training.

On this basis, machine learning has turned itself to a wide range of fields including object recognition, disease diagnosis, robotics, product recommendation and many more. In our course of internship, we tried to implement different models in recognizing varieties of handwritten digits obtained from a pre-defined dataset.

1.3 Handwritten Digit Recognition:

Handwritten digit recognition emerges as a quintessential chapter in the story of artificial intelligence as, the human- written numbers has transcended traditional boundaries and has become the backbone of countless applications of our daily lives. It is a profound exploration of machine learning and computer vision. In other words, it is a way of teaching machines to emulate our ability to understand, interpret and categorize the diverse ways in which we express numerical symbols in a page. During this approach, some of the points we have to take in our minds are:

1. Handwriting styles vary based on the writer so, it is usually very difficult even for human to recognize handwritten digits because of the significant difference of handwriting styles.
2. Similarity between handwritten digits, for example six and four may look the same digit based on the handwriting style of the writer similarly, one and seven may look the same digit which is more commonly known as redundancy of data which needs to be removed to get a better result.
3. There is no perfect machine learning model that is suitable and effective in recognizing handwritten digits. So, it is on the basis of our trial and analysis that we can find a better model of good accuracy.

On the basis of the preview of the topic of our discussion throughout the report, here is a brief overview of the concept of our approach with respect to machine learning models:

- Handwritten digit recognition requires a labeled dataset. A popular dataset is the MNIST dataset which consists of 28x28 pixel grayscale images of digits from 0 to 9.

- Splitting the dataset into trainingand testing sets.

- Preprocessing steps involve resizing images and normalization to ensure consistency and improve model performance.

- Choosing an appropriate machine learning model for the task.

- Using appropriate evaluation metrics like accuracy or confusion matrix, helps us to assess the model’s performance.

2. METHODOLOGY:

2.1 LOGISTIC REGRESSION:

Logistic regression is a supervised machine learning model that Is mainly used for classification tasks. This model is used when the dependent variable is dichotomous or binary (that means they will have only 2 outputs). The model takes the output of linear regression function as input and uses a **sigmoid function** to estimate the probability of the given class. It predicts the output of a categorical dependent variable. The sigmoid function is a mathematical function which maps the predicted values of probabilities within a range of 0 and 1, that gives a 'S' form curve. The common terms used in this model are:

- Independent variables are input characteristics applied to the dependent variable's predictions.
- Dependent variable is the target variable in the model which, we try to predict.
- Logistic function is the formula that is used to represent how the independent and dependent variables relate to one another. This function transforms the input variables into a probability value between 0 and 1.

Logistic regression works on the basis of some assumptions:

1. Each observation is independent of the other, without any correlation between the input variables.
2. The sample size is considered to be sufficiently large.
3. The dependent variable can take only two values.

2.1.1 WORKING OF THE MODEL:

Logistic regression estimates the relationship between the input features and probability of the binary outcome using a logistic (sigmoid) function. The model is trained by finding the optimal weights (w_1, w_2, w_3.....) and the corresponding binary target variable (dependent variable). The logistic function applied to the linear combination of features and weights is:

$$y' = \frac{1}{1 + e^{-z}}$$

where y' is the output of the logistic regression model for a particular example.

2.1.2 CODE IMPLEMENTATION:

- **IMPORTING LIBRARIES**

```
[1]   1 import pandas as pd
      2 import matplotlib.pyplot as plt
      3 import numpy as np
```

The following lines are written for data analysis and visualization using Pandas, NumPy and Matplotlib libraries in Python.
Pandas provides data structures and functions for efficiently working with tabular data (like CSV files). It introduces the **DataFrame**, a two-dimensional and heterogenous data structure that allows us to store and manipulate data in a way similar to a database table. It also helps to handle missing data, filter or select data, import and export data from different sources. The **'pd'** is used as an alias for Pandas to make the code more concise.

Matplotlib is a library commonly used for data visualization and create charts, plots and graphs to represent the data. It also helps to customize the plots including colors, styles, labels etc. The **'plt'** is used as an alias for the library. The **'pyplot'** provides functions for creating various types of plots and charts.

NumPy is a library used for numerical computing in Python. It helps us to work with large, multi-dimensional arrays and matrices of data. It also provides linear algebra operations, Fourier transforms and different numerical operations. The library is assigned an alias as **'np'**.

- **LOADING THE DATASET**

```
[2]   1 from sklearn.datasets import fetch_openml
      2 mnist = fetch_openml('mnist_784')
```

The provided lines of code are used to load the MNIST dataset which is specialized mainly for the training of machine learning models in the context of image classification. The main components are-

1. The dataset comprises of 28 x 28 pixel grayscale images of handwritten digits (0-9).
2. There are 70000 such images each labeled with the corresponding digit it represents.
3. The dataset is divided into two parts- training and testing set.

Scikit-learn or **sklearn** is a popular machine learning library that provides datasets for practice and experimentation. The **'fetch-openml'** is a function from **'sklearn. datasets'** module that helps to download and load the MNIST dataset (that is, **mnist-784**). This 'mnist' contains two components:

- ```
  1 X = mnist['data']
  ```
  This represents the pixel values of the images. Each row in this array corresponds to an image. It is stored in X.

- ```
  2 Y = mnist['target']
  ```
  This attribute contains the labels associated with each image.

○ FEATURE SCALING THE DATA

```
1 from sklearn.preprocessing import StandardScaler
2 X = X/255.0
3 sc = StandardScaler()
4 X_scaled = sc.fit_transform(X)
```

Feature scaling is a preprocessing technique used in machine learning to standardize or normalize the range of independent variables of data. It is important when the features in the data have different scales and that machine learning algorithm is sensitive to the scale of the input features. Some of the common techniques used are **'Standardization'** and **'Normalization'**.
The provided lines of code are related to feature scaling using **'StandardScaler'** class from **'sklearn.preprocessing'** module. In this particular model, we are using Normalization method for feature scaling. By dividing the features in the dataset 'X' by 255.0, the pixel values in images are brought to the range of [0,1]. Now, for scaling the features in 'X', an instance **'sc'** is created which helps in applying the **'fit_transform'** method to scale the dataset 'X'. It is stored in the variable **'X_scaled'**.

○ SPLITTING DATASET INTO TRAINING SET AND TESTING SET

```
1 from sklearn.model_selection import train_test_split
```

```
1 X_train, X_test, y_train, y_test = train_test_split( X_scaled,Y, test_size=0.2, shuffle = True)
```

```
1 X_train.shape
```

```
(56000, 784)
```

```
1 X_test.shape
```

```
(14000, 784)
```

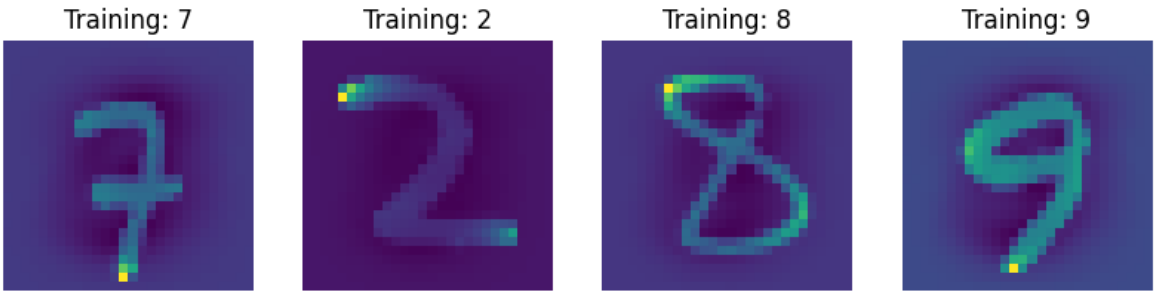
Data splitting is an important part while working on a machine learning model. The 'train_test_split' function is used to split the dataset into training and testing sets. Then the entire data is split into four sets-

- **X_train** holds the training features (80% of data)
- **X_test** holds the testing features (20% of data)
- **y_train** holds the labels for training set.
- **y_test** holds the labels for testing set.

This splitting is done in a feature matrix that we created after feature scaling, that is **'X_scaled'**. The **'Y'** attribute will store the label array for the corresponding features of the 'X_scaled'. The **test_size** indicates the proportion of the dataset that should be reserved for testing. In this case **'0.2'** means 20% is used for testing and remaining 80% is for training. At last, the 'shuffle' parameter is set to 'True' in order to shuffle the data before splitting to avoid any potential bias during the data splitting process.
For printing the number of rows (samples) and columns (features) in the training and testing set, **'X_train.shape'** and **'X_test.shape'** is used.

- **VISUALIZING THE IMAGES OF THE DATASET**

```
1 _, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
2 for ax, image, label in zip(axes, X_train, y_train):
3     ax.set_axis_off()
4     image = image.reshape(28,28)
5     ax.imshow(image)
6     ax.set_title("Training: %i" % int(label))
```



For visualizing a subset of training images along with their labels, we have to create a row of subplots and iterate the same, for all the images. The arguments in the **'plt.subplots'** function indicates the number of rows **('nrows = 1')**, columns **('ncols = 4')** of subplots and height, width of all the figures of the entire set

('figsize = (10,3)'). These, therefore determines the overall dimensions of the subplot grid.
For iteration, we set up a 'for' loop with the following parts-

- **'ax'** represents the current subplot of the loop.
- **'image'** represents an image from the training set.
- **'label'** represents the label associated with the current image.
- **'zip'** combines all the elements of three lists (**'axes, X_train, y_train'**) into tuples. It also helps in pairing each subplot with an image and its corresponding label.

For better visualization, we hid the axis using **'ax.set_axis_off ()'**. Often the images are flattened (1D array) for a machine learning model. So, we have to reshape it to its original dimensions (change to 2D array) by using **'image.reshape'**. The 'image' contains the reshaped 2D image data. Then the images of the current subplot **'ax'**, are displayed using the **'imshow'** function. The title for the current subplot **'ax'** includes the label which is converted to an integer and displayed as **'Training: [label]'**.

- **TRAINING THE LOGISTIC REGRESSION MODEL ON THE TRAINING SET**

```
1 from sklearn.linear_model import LogisticRegression
2 classifier = LogisticRegression(random_state = 1)
3 classifier.fit(X_train, y_train)
```

For training the following dataset, we can use different models. This is by the approach of **Logistic Regression Model.** The model is imported from the **'sklearn.linear_model'** module and an instance of the model is created and assigned to a variable **'classifier'**.

The **'random_state'** parameter is used to initialize the random number generator for certain operations. On setting it to a specific value, (here **random_state = 1**) it will give same results if we run the code again with the same data and settings.

The **'fit'** method fits the Logistic Regression Model to the training data. During this process, the model learns the relationship between the **features (in X_train)** and **labels (in y_train)** that, helps it to make predictions. The **'X_train'** contains the input features used to train the classifier and the **'y_train'** contains the corresponding class labels or outcomes that the **'classifier'** is trying to predict. After the training step, the **'classifier'** holds the trained Logistic Regression Model and we, can evaluate the performance of the model (accuracy, or any other metrics).

- **PREDICTING TEST SET RESULT**

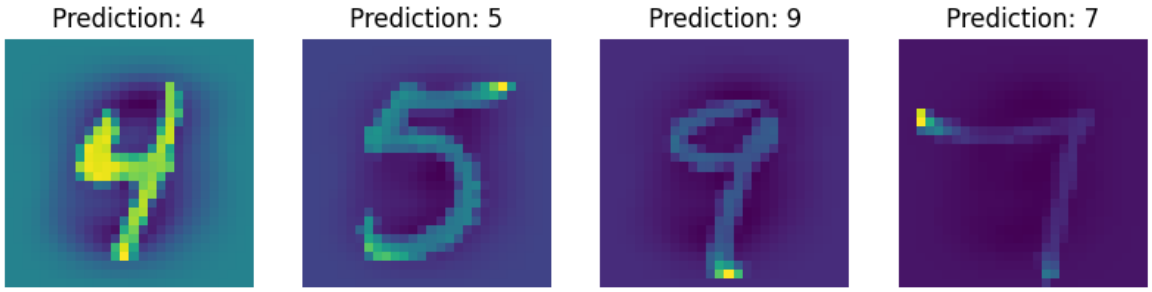
```
1 y_pred = classifier.predict(X_test)
```

```
1 y_pred.shape
```

The **'predict'** methodis used to make predictions on a set of data. Here, the **'predict'** method is of the **trained Logistic Regression 'classifier'**. It takes the **test data ('X_test')** as input and return the predicted class labels for the test, which are stored in the variable **'y_pred'**. Further, the **'shape'** attribute is used to determine the dimensions of an array and provide information about the number of samples for which predictions were generated (which is typically equal to the number of samples in the test dataset).

- **VISUALIZING FIRST FOUR TEST SAMPLES AND SHOWING THEIR PREDICTED VALUES**

```
1 _, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
2 for ax, image, prediction in zip(axes, X_test, y_pred):
3     ax.set_axis_off()
4     image = image.reshape(28,28)
5     ax.imshow(image)
6     ax.set_title(f"Prediction: {prediction}")
```



Similar to the visualization of the training dataset, we wanted to display of any four test samples along with their predicted values. The provided lines of codes create a grid of subplots with the help of **Matplotlib**, iterate over the four, random test samples using a **'for'** loop, reshapes the flattened images to its original dimensions and displays the predicted values as **'Prediction: {prediction}'**.

- **DISPLAYING THE CONFUSION MATRIX**

```
1 import sklearn.metrics as metrics
2 disp = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
3 disp.figure_.suptitle("Confusion Matrix")
4 print(f"Confusion matrix:\n{disp.confusion_matrix}")
5 plt.show()
```

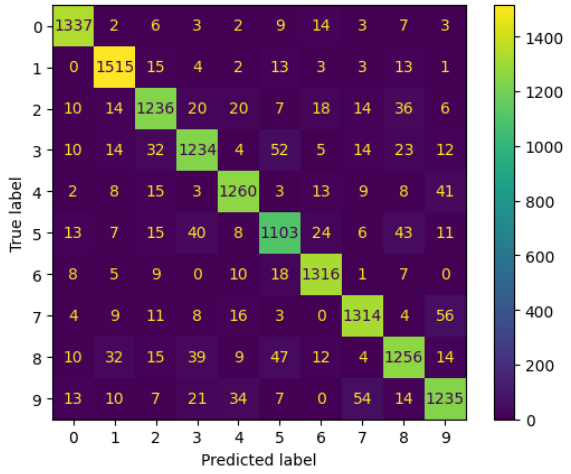
The confusion matrix helps us to assess how well our classification model is performing by providing information about the correct and incorrect predictions for different classes. The **'metrics'** module from scikit-learn library contains various functions to create the matrix.

In the **'disp'** object, a confusion matrix is created and stored using the **true labels ('y_test')** and **predicted labels ('y_pred')** provided as arguments, with the help of **'from_predictions' method'**.

The summary of the model's performance including the number of true positives, true negatives, false positives and false negatives are provided by the **'disp.confusion_matrix'** attribute.

Finally, to display the matrix, we used the **'show'** function.

```
Confusion matrix:
[[1337    2    6    3    2    9   14    3    7    3]
 [   0 1515   15    4    2   13    3    3   13    1]
 [  10   14 1236   20   20    7   18   14   36    6]
 [  10   14   32 1234    4   52    5   14   23   12]
 [   2    8   15    3 1260    3   13    9    8   41]
 [  13    7   15   40    8 1103   24    6   43   11]
 [   8    5    9    0   10   18 1316    1    7    0]
 [   4    9   11    8   16    3    0 1314    4   56]
 [  10   32   15   39    9   47   12    4 1256   14]
 [  13   10    7   21   34    7    0   54   14 1235]]
```



In context to the confusion matrix, let us know the following points:

1. **True Positives** are the cases where the model correctly predicted the class of interest or correctly identified the instances as belonging to the positive class.
2. **True Negatives** are cases where the model correctly predicted the class that is not of interest.
3. **False Positives** are cases where the model incorrectly predicted the positive class when it should have predicted the negative class.
4. **False negatives** are cases where the model incorrectly predicted the negative class when it should have predicted the positive class.

- **CALCULATING THE ACCURACY OF THE MODEL**

```
1 from sklearn.metrics import accuracy_score
2 accuracy_score(y_test, y_pred)
```

After we also testing the model with a certain dataset, it is important to measure the **accuracy** of the model by comparing its predictions to the true labels. This score helps to measure the correctness of the predictions made by the **Logistic Regression 'classifier'** on the test data.

The **'sklearn.metrics'** module has a function termed as **'accuracy_score'** which is used to calculate the accuracy of the classifier's predictions. The **true labels** for the test data are present in **'y_test'** which represents the correct labels that we are trying to predict. The 'y_pred' is the array of **predicted class label** generated by the classifier of the same test data. We have to compare both of them and compute the **accuracy as the ratioof correct predictions to the total number of predictions.**

Here using this model, we attained an accuracy of 91.47 % (approximately).

2.2 SUPPORT VECTOR MACHINE (KERNEL -RBF) MODEL:

Support Vector Machine (SVM) is a machine learning model used for both classification and regression tasks. It's known for its ability to find an optimal hyperplane that separates data points into different classes or fits a hyperplane for regression. SVM is particularly valuable in scenarios with complex data and high-dimensional feature spaces.

SVM models can be mainly classified into two types:

1. **Linear SVM**:
   Linear SVM is used when the data is linearly separable, meaning that a straight line (in 2D) or a hyperplane (in higher dimensions) can cleanly separate the classes. The goal of a linear SVM is to maximize the margin between classes while minimizing classification errors. It works well for data with a clear linear boundary.
2. **Non -Linear SVM:**
   Non-Linear SVM is used when the data is not linearly separable, meaning a straight line or hyperplane cannot effectively separate the classes. Non-linear SVM employs kernel functions (e.g., **Radial Basis Function, polynomial, sigmoid**) to transform the data into a higher-dimensional space where it may become linearly separable. This approach can handle complex, non-linear relationships between features and classes.

SVM is a versatile machine learning model that includes both Linear SVM for linearly separable data and Non-Linear SVM, which uses kernel functions to tackle non-linear data. Linear SVM works well when the relationship between features and classes is linear, while Non-Linear SVM is suitable for data with complex or non-linear patterns.

In this project, we exemplify the utilization of the **Radial Basis Function (RBF) kernel**, a powerful non-linear variant of the **Support Vector Machine (SVM)** model, for the recognition of the benchmark MNIST dataset

2.2.1 Working of RBF SVM model:

Radial Basis Function Support Vector Machine (RBF SVM) is a powerful machine learning algorithm that can be used for classification and regression tasks. It is a non-parametric model that works well with non-linear and high-dimensional data.RBF SVM works by mapping the input data into a higher-dimensional feature space, where the classes can be separated by a hyperplane.

The algorithm uses a **kernel function**, such as the **Radial Basis Function**, to measure the similarity between pairs of data points in the feature space.

The Radial Basis Function is a popular kernel function used with RBF SVM. It is defined as:

$$K(x, x') = \exp(-\gamma ||x - x'||^2)$$

where,

- **x** and **x'** are input data points
- **γ(gamma)**is a hyperparameter that controls the width of the kernel and
- $||x-x'||^2$is the squared Euclidean distance between the points.

The kernel function measures the similarity between pairs of data points based on their distance in the feature space.

In comparison to the linear counterpart the RBF kernel SVM model is a better suited model for prediction of the MNIST dataset pertaining to the nature of the dataset. MNIST as a collection of handwritten digits, present intricate and non - linear relationships between pixel values and corresponding digits. The rbf kernel due to its flexibility in defining non-linear decision boundaries captures these complex patterns, resulting in **higher accuracy** as it accommodates diverse writing styles and variations in the dataset. Additionally, RBF SVM provides better **generalization** abilities making them less prone to overfitting and **dimensionality** as the kernel method implicitly map data into a higher-dimension space, which is especially effective in capturing complex patterns in high-dimensional data.

With that in mind, it's essential to acknowledge that the RBF SVM model has certain drawbacks that don't apply to the linear model. These **limitations** are discussed below:

1. **Computational Intensity:** RBF SVMs are more computationally demanding, especially with larger datasets, making them slower to train and evaluate.
2. **Complexity:** RBF models can be more challenging to tune due to the sensitivity to hyperparameters, such as the kernel width, which can lead to overfitting if not properly adjusted.
3. **Interpretability:** The non-linear decision boundaries of RBF SVMs make them less interpretable than linear SVMs, which have straightforward linear boundaries that can be easily visualized and understood.

2.2.2 IMPLEMENTATION:

Through the implementation of the RBF kernel SVM model, we methodically conducted the MNIST dataset recognition project as delineated below:

- IMPORTING LIBRARIES

```
[ ]  import pandas as pd
     import matplotlib.pyplot as plt
     import numpy as np
```

The aforementioned libraries are imported to facilitate data handling, visualization, and mathematical operations essential for the machine learning study.

import pandas as pd: This line imports the Pandas library and assigns it an alias 'pd.' Pandas is a powerful data manipulation and analysis library in Python. Pandas provides functions for working with tabular data. It is used for handling and organizing data, especially when you want to manipulate and analyze the results or dataset.

Matplotlib library, an essential tool for creating visualizations and plots in Python. It's assigned the alias 'plt.'Matplotlib can be used to generate various plots and graphs to visualize the results, such as accuracy vs. epochs or confusion matrices for different models.

NumPy is fundamental for numerical operations and array manipulations. It is used when working with the MNIST dataset and various mathematical operations related to machine learning model. The library is assigned as an alias as 'np'.

- LOADING THE DATASET

```
[ ]  from sklearn.datasets import fetch_openml
     mnist = fetch_openml('mnist_784')

     /usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968: FutureWarning: The default value of `parser` will change from `'liac-arff'` to `'auto'` in 1.4. You can set
       warn(

[ ]  X = mnist['data']
     Y = mnist['target']
```

Here, we are using the **ski-learn** library and its functions to load the MNIST dataset. **Scikit-Learn** (sklearn) is a widely-used open-source Python library for machine learning and data analysis. It provides a comprehensive set of tools for tasks such as classification, regression, clustering, dimensionality reduction, and model evaluation.

- **from sklearn.datasets import fetch_openml**: This line imports a function named **fetch_openml** from the
- **sklearn.datasets** module. It's used to download datasets from the OpenML repository.
- **mnist = fetch_openml('mnist_784')**: This line uses the **fetch_openml** function to download the MNIST dataset with 784 features (28x28 pixel images). It stores the dataset in the **mnist** variable.
- **X = mnist['data']:** This line extracts the data (image pixel values) from the MNIST dataset and stores it in the X variable.
- **Y = mnist['target']**: This line extracts the target labels (the digits each image represents) from the MNIST dataset and stores them in the Y variable.

- FEATURE SCALING THE DATA:

```
from sklearn.preprocessing import StandardScaler
X = X/255.0
sc = StandardScaler()
X_scaled = sc.fit_transform(X)
```

Feature scaling is a data preprocessing technique used in machine learning to standardize or normalize the range of independent variables or features of a dataset. It involves transforming the features to a common scale, typically between 0 and 1 or with a mean of 0 and a standard deviation of 1. Feature scaling equalizes feature importance, enhances convergence, improves model performance and facilitates interpretation

In the following code snippet, we import the '**StandardScaler'** class from scikit-learn, which is used to standardize the dataset. Then we are dividing each pixel value stored in X by **255**(as the pixel values typically range from 0 to 255) in an attempt to normalize it. Now,**sc= StandardScaler ()**: An instance of the **StandardScaler** is created. This object will be used to perform the scaling operation on the dataset. The sc object is then used to call the '**fit_transform()'** method on X to standardize the pixel values in the dataset. It calculates the mean and standard deviation of the data and scales it so that it has a mean of 0 and a standard deviation of 1. The resulting standardized data is stored in the variable **X_scaled**.

- SPLITTING THE DATASET INTO TRAINING SET AND TEST SET:

```
[ ] from sklearn.model_selection import train_test_split

[ ] X_train, X_test, y_train, y_test = train_test_split( X_scaled,Y, test_size=0.2, shuffle = True)

[ ] X_train.shape
    (56000, 784)

[ ] X_test.shape
    (14000, 784)
```

Data Splitting: To facilitate the training and evaluation of machine learning models on the MNIST dataset, we used the **train_test_split** function from the scikit-learn library. This function is instrumental in dividing the dataset into two key subsets: a training set and a testing set. Here,

- **X_scaled** contains the standardized image data, and Y holds the corresponding digit labels.
- We chose to allocate 20% of the dataset to the testing set, which is specified using **test_size**=0.2. This means that 20% of the data will be reserved for evaluating the model's performance (Additionally, we set **shuffle=True** to ensure that the data is randomly shuffled before splitting. This helps prevent any inherent order or bias in the dataset from affecting the model's performance.)

The outcome of this code is the creation of four distinct datasets:

- **X_train:** This dataset contains the features (image data) for training the machine learning model.
- **X_test**: This dataset is used for evaluating the model's performance.
- **y_train:** It consists of the corresponding labels for training.
- **y_test:** This dataset contains labels for testing.

- VISUALIZING THE IMAGES OF THE DATASET:

```
_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, label in zip(axes, X_train, y_train):
    ax.set_axis_off()
    image = image.reshape(28,28)
    ax.imshow(image)
    ax.set_title("Training: %i" % int(label))
```

The provided code snippet is used to visualize a subset of the training data.

_, axes = plt.subplots(nrows=1, ncols=4, figsize= (10, 3)): We created a row of four subplots to display a selection of images.
A loop is employed to iterate over the **axes** and corresponding training data. In each iteration, **ax** represents one of the four subplots, **image** holds the image data, and **label** contains the digit label associated with the image.
ax.set_axis_off(): We turn off the axis labels and ticks for a cleaner and more focused image display.
The image data is reshaped (**image = image.reshape(28, 28)**) to its original 28x28 pixel format. This step is crucial because the data was initially flattened for machine learning, and reshaping it restores its original appearance.
ax.imshow(image): The image is displayed on the current subplot using **imshow**. This step visualizes the digit as it would appear in the original MNIST dataset. Then, **ax.set_title("Training: %i" % int(label))**, A title is added to each subplot to indicate that it's a training example and to display the associated digit label.

- TRAINING THE SVM (kernel = rbf) MODEL ON THE TRAINING SET:

```
[ ] from sklearn.svm import SVC
    classifier = SVC(kernel = 'rbf')
    classifier.fit(X_train, y_train)
```

```
▾ SVC
SVC()
```

Here, we imported the Support Vector Classification (SVC) class from scikit-learn library. An SVM classifier is instantiated with the 'rbf' kernel. The Radial Basis Function kernel is a versatile choice for handling non-linear data, which is often the case with handwritten digit recognition.
classifier.fit(X_train, y_train): The training data (**X_train**) and corresponding labels (**y_train**) are used to train the SVM classifier. This step involves the model learning the patterns and relationships within the training data, enabling it to make predictions on unseen data.

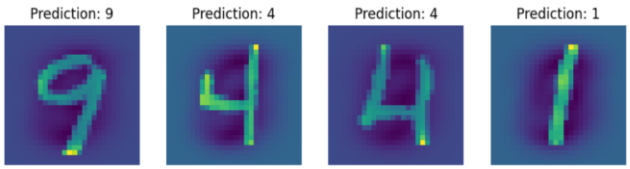
- PREDICTING THE TEST RESULT:

```
[ ] y_pred = classifier.predict(X_test)
```

y_pred = classifier.predict(X_test): The trained SVM classifier is utilized to make predictions on a set of previously unseen or 'test' data (**X_test**). These predictions are stored in the **y_pred**variable. This process enables us to evaluate the model's ability to generalize its learning from the training data to new, unseen examples. The **y_pred** variable now contains the predicted digit labels for the test data.

- VISUALIZING THE TEST SAMPLES AND SHOWING THEIR PREDICTED VALUE:

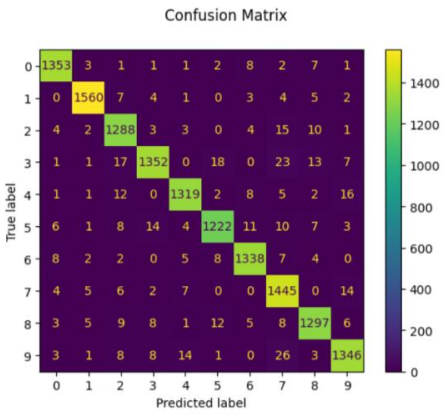
```
[ ] _, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
    for ax, image, prediction in zip(axes, X_test, y_pred):
        ax.set_axis_off()
        image = image.reshape(28,28)
        ax.imshow(image)
        ax.set_title(f"Prediction: {prediction}")
```



This code snippet,visualizes a set of test images alongside their corresponding model predictions. Four images are displayed in a row, each with its predicted digit label.

- DISPLAYING THE CONFUSION MATRIX AND PREDICYING THE ACCURACY:

```
[ ] import sklearn.metrics as metrics
    disp = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
    disp.figure_.suptitle("Confusion Matrix")
    print(f"Confusion matrix:\n{disp.confusion_matrix}")

    plt.show()

    Confusion matrix:
    [[1353    3    1    1    1    2    8    2    7    1]
     [   0 1560    7    4    1    0    3    4    5    2]
     [   4    2 1288    3    3    0    4   15   10    1]
     [   1    1   17 1352    0   18    0   23   13    7]
     [   1    1   12    0 1319    2    8    5    2   16]
     [   6    1    8   14    4 1222   11   10    7    3]
     [   8    2    2    0    5    8 1338    7    4    0]
     [   4    5    6    2    7    0    0 1445    0   14]
     [   3    5    9    8    1   12    5    8 1297    6]
     [   3    1    8    8   14    1    0   26    3 1346]]
```



The **Confusion Matrix** is a critical tool for understanding the model's strengths and weaknesses, particularly in identifying any patterns of misclassification.

Here, we imported the scikit-learn metrics module to access functions and tools for model evaluation. We use the

"**metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)**" function to generate a confusion matrix. It takes the true labels (**y_test**) and the model's predictions (**y_pred**) as input to construct the matrix.

print(f"Confusion matrix:\n{disp.confusion_matrix}"): This statement prints the confusion matrix to the console, displaying the counts of true positive, true negative, false positive, and false negative predictions for each class. Finally, using the '**plt.show()',** the Confusion Matrix display is shown, providing a visual representation of the model's performance in classifying digits.

```
[ ]  from sklearn.metrics import accuracy_score
     accuracy_score(y_test, y_pred)

     0.9657142857142857
```

We imported the **accuracy_score** function from scikit-learn to measure the proportion of correctly predicted digit labels.

accuracy_score(y_test, y_pred): This function takes the true digit labels from the test set **(y_test)** and the model's predicted labels **(y_pred)** as input. It calculates and reports the accuracy of the model's predictions, expressing it as a single accuracy score.

The accuracy score is found to be **0.9657142857142857**

2.3 SUPPORT VECTOR MACHINE (LINEAR KERNEL) MODEL:

Support Vector Machine or SVM is one of the most popular supervised learning algorithms which is used for classification as well as regression problems. But generally, it is used for classification problems.

2.3.1 Working of the Model:

The linear SVM model operates under the assumption of **linear separability**. This means that it works best when it can separate data into two classes using a straight line in two dimensions or a hyperplane in higher dimensions. The goal is to find the hyperplane that best separates the data while maximizing the margin between the two classes. The hyperplane is defined by the equation:

$w^T x + b = 0$

w represents the weight vector perpendicular to the hyperplane.
x is the input data vector.
b is the bias term, which represents an offset or translation of the decision boundary (hyperplane) away from the origin in the feature space.

In case of linear SVM, this hyperplane is a straight line.

SVM uses a **decision function** to classify new data points into one of the two classes based on their features. It is derived from the equation of the hyperplane that the SVM has learned during the training phase. Here's the general form of the decision function in SVM:

$f(x) = \text{sign}\ (w^T x + b)$

$f(x)$ is the output of the decision function for the given input x.

The decision function computes the dot product $w^T x$ and classifies data points into different classes. The result of $w^T x$ is used in the decision function to determine the sign of $w^T x + b$, which classifies the input data point x into one of the two classes.

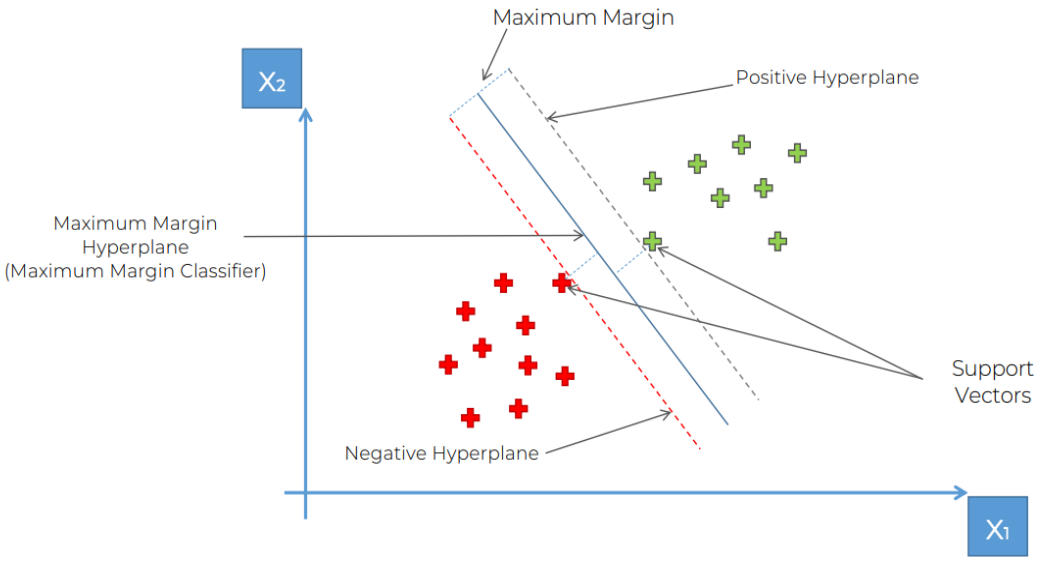
- If $w^T x + b$ is positive, the data point x is classified as the positive class or the class labeled as +1
- If it's negative, x is classified as the negative class or the class labeled as -1.
- If $w^T x + b$ equals exactly 0, the data point x lies on the decision boundary (hyperplane). In many SVM implementations, these points are referred to as "support vectors".

Support vectors are defined as the data points that lie closest to the hyperplane that separates the two classes in a binary classification problem. These points

are the "support" of the decision boundary as they are used to determine the position and orientation of the hyperplane.
Hence,

$$f(x) = y = \begin{cases} +1, if\ w^T x + b \geq 0 \\ -1, if\ w^T x + b \leq 0 \end{cases}$$

The key concept behind SVM is margin maximization. The margin is defined as the distance between the hyperplane and the nearest data point from either class. This distance is measured perpendicular to the hyperplane. SVM aims to find the hyperplane that maximizes this margin. By maximizing the margin, SVM creates a decision boundary that is robust to noise, less susceptible to misclassification and ensures the largest possible and clear separation between the classes.



SVM can be used for face detection, image classification and text categorisation, etc.

2.3.2 Code Implementation:

1. Importing the necessary libraries:

The necessary libraries we are using here are:

Pandas: Pandas is a library which helps to create a new kind of data structure that is called as data frame. We can visualise it just like an excel sheet which has multiple rows and column.

NumPy: it is a library used for working with arrays, domain of linear algebra, Fourier transform, and matrices.

Matplotlib: It is a detailed library used for creating animated, static and interactive visualizations in Python.

```
[ ]  import numpy as np
     import matplotlib.pyplot as plt
     import pandas as pd
```

2. Importing the MNIST dataset:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')
mnist.keys()
```

The 'fetch_openml'function is imported from Scikit-learn's 'datasets'module. This function is used to fetch datasets from the OpenML repository. The MNIST dataset with the identifier 'mnist_784' is fetched from the OpenML repository using mnist =fetch_openml('mnist_784').mnist.keys()retrieves the keys associated with the mnist dataset object. These keys provide information about the dataset's structure and content.

3. Visualizing the dataset:

```
mnist['data']
```

The above line of code retrieves the feature matrix that contains the pixel values of the MNIST dataset's images. Each row of this matrix represents an image, and each column represents a pixel in the image.

```
[ ]  mnist['data'].shape
```

'mnist['data'].shape' retrives a tuple that specifies the number of rows and columns in the feature matrix. Specifically,

- The first value in the tuple represents the number of rows, which corresponds to the number of images in the dataset.
- The second value in the tuple represents the number of columns, which corresponds to the number of pixels in each image.

There are 70,000 images in the datatset and each image has 784 pixels (as MNIST images are 28x28 pixels each).

```
[ ]  mnist['target']
```

Executing the above line of code retrives an array that contains the target labels for each image in the dataset. These target labels are typically integers representing the digit (0 to 9) that each image depicts.

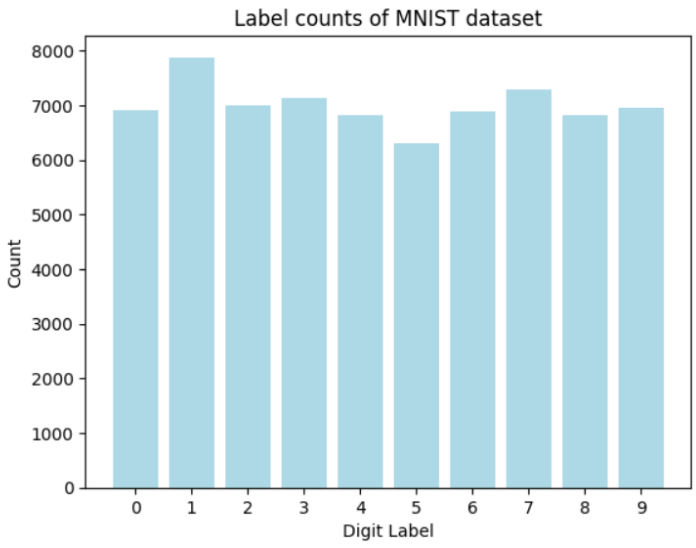
```
[ ]  mnist['target'].shape
```

'mnist['target'].shape'displays a tuple that specifies the number of elements in the target labels array. Specifically, it provides information about the number of target labels in the dataset. There are 70,000 images in the dataset.

```
[ ]  mnist.target.astype('category').value_counts()
```

The above line of code calculates the count of each unique target label of the MNIST dataset. 'mnist.target' accesses the target labels of the MNIST dataset, which represents the digit corresponding to each image. 'astype('category')' converts the target labels into a categorical data type. '.value_counts()' calculates and returns the count of unique values in the categorical data. It counts how many times each digit (0 to 9) appears in the dataset. Thus, we get a count of how many times each digit appears in the MNIST dataset.

```
[ ]  label_counts = mnist.target.value_counts().sort_index()
     plt.bar(label_counts.index.astype(int), label_counts.values, color = 'lightblue')
     plt.xlabel('Digit Label')
     plt.ylabel('Count')
     plt.title('Label counts of MNIST dataset')
     plt.xticks(range(10))
     plt.show()
```



The above line of code generates a bar chat to provide a visual representation of the distribution of digits in the MNIST datatset, showing how many times each digit (0 to 9) appears in the dataset.

4. Defining matrix of features 'X' and target vector 'y':

```
[ ]  X = mnist['data']
     y = mnist['target']
```

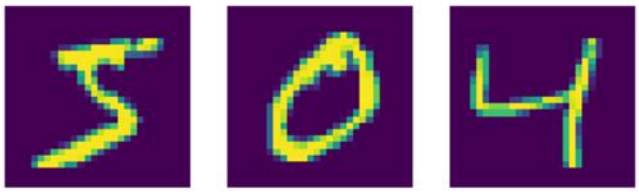
'X = mnist['data']' assigns the feature matrix of the MNIST dataset to the variable X, whereas 'y = mnist['target']' assigns the target labels of the MNIST dataset to the variable y. After executing these two lines of code, we will have the following variables:

- **'X':** This variable contains the feature matrix, which is a NumPy array or DataFrame (depending on the data format of the MNIST dataset).
- **'y':** This variable contains the target labels, which are also NumPy array or Series, representing the digit labels associated with each image in 'X'.

5. Displaying the first three images of the dataset:

```
[ ]  digit = X.iloc[0]
     digit_pixels = np.array(digit).reshape(28,28)
     plt.subplot(131)
     plt.imshow(digit_pixels)
     plt.axis('off')

     digit = X.iloc[1]
     digit_pixels = np.array(digit).reshape(28,28)
     plt.subplot(132)
     plt.imshow(digit_pixels)
     plt.axis('off')

     digit = X.iloc[2]
     digit_pixels = np.array(digit).reshape(28,28)
     plt.subplot(133)
     plt.imshow(digit_pixels)
     plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```



'digit = X.iloc[0]' selects the first row of the feature matrix 'X', which represents the pixel data of the first digit in the dataset.

'digit_pixels = np.array(digit).reshape(28, 28)' converts the pixel data of the first digit into a NumPy array and reshapes it into a 28 x 28 array. This is done because the original pixel data is flattened into a 1D array in the dataset.

'plt.subplot(131)' creates a subplot with 1 row, 3 columns and selects the first subplot (index 1) for displaying the image. 'plt.imshow(digit_pixels)' displays the image represented by the 'digit_pixels' array using Matplotlib's 'imshow' function. 'plt.show('off')' turns off the axis labels and ticks in the plot, providing a cleaner visualisation of the image.

After executing this code, we see that the first digit from the MNIST datatset is displayed as a 28 x 28 pixel image in the first subplot of the figure. The other two images are also displayed using similar code as shown.

6. Feature Scaling:

```
[ ] from sklearn.preprocessing import StandardScaler
    sc = StandardScaler()
    X_scaled = sc.fit_transform(X)
```

The above line of code is performing standardization (also known as z-score normalisation) on the pixel values of the MNIST dataset.Feature scaling ensures that all features (pixel values in this case) have similar scales or units. Each pixel in an image typically has a value between 0 and 255, where 0 represents black and 255 represents white (in grayscale images). After scaling, all pixel values are transformed to a common scale, typically ranging from 0 to 1. Scaling ensures that all featuresin this case have a similar influence on the model, preventing certain features from dominating the learning process.

7. Splitting the dataset into Training set and Test set:

```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size = 0.2, random_state = 42)
```

```
[ ] print(X_train.shape)
    print(y_train.shape)
    print(X_test.shape)
    print(y_test.shape)

    (56000, 784)
    (56000,)
    (14000, 784)
    (14000,)
```

The dataset is split into the Training and Test set based on the parameters specified. 'test_size = 0.2' specifies that 20% of the data should be allocated for testing (the remaining 80% is used for training). After executing this code, we will have the following variables:

- 'X_train': This variable contains the feature matrix of the training setm which is 80% of the data.

- 'X_test': This variable contains the feature matrix of the test set, which is 20% of the data.
- 'y_train': This variable contains the target labels corresponding to the training set.
- 'y_test': This variable contains the target labels corresponding to the test set.

8. Building the Linear SVM Model:

```
[ ] from sklearn.svm import SVC
    model_linear = SVC(kernel = 'linear')
    model_linear.fit(X_train, y_train)
```

```
SVC(kernel='linear')
```

```
[ ] y_pred_1 = model_linear.predict(X_test)
```

Execution of this block of code will create the variable 'model_linear', which will contain the trained SVM classifier with a linear kernel. The choice of a linear kernel implies that the decision boundary created by this SVM will be a straight line in the feature space. We will use this model to make predictions on the test set. 'y_pred_1' will contain the predicted labels for the testing data based on the model's decision boundary learned during training.

9. Accuracy, confusion matrix and classification report:

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
print("Accuracy: ", accuracy_score(y_test, y_pred_1))
print(confusion_matrix(y_test, y_pred_1))
```

```
Accuracy:  0.9207857142857143
[[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   36 1302    3   47    1    9   19   10]
 [   5    3   15    2 1214    4    5    6    3   38]
 [  13   10   12   52    9 1124   21    1   20   11]
 [  11    3   29    2   17   31 1300    1    2    0]
 [   1    6   24   13   23    7    0 1404    2   23]
 [  15   26   26   54    6   44   10   10 1152   14]
 [   7   11   10   18   51    8    0   40   11 1264]]
```

```
[ ]  report = classification_report(y_test, y_pred_1)
     print("Classification report:")
     print(report)
```

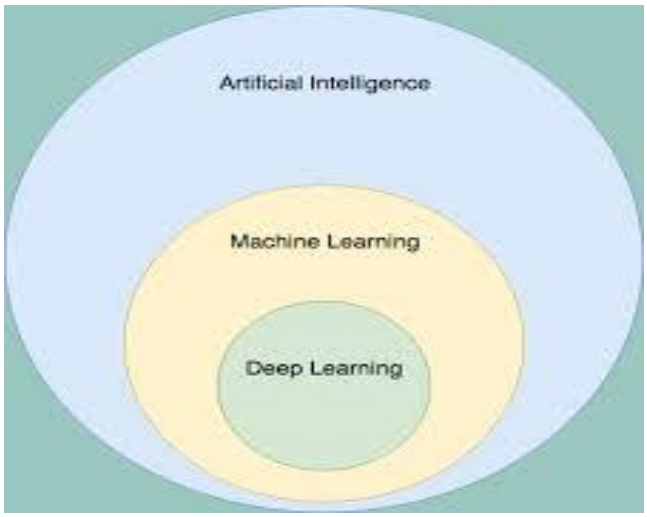
```
     Classification report:
                   precision    recall  f1-score   support

                0       0.95      0.97      0.96      1343
                1       0.95      0.98      0.97      1600
                2       0.89      0.92      0.90      1380
                3       0.88      0.91      0.90      1433
                4       0.90      0.94      0.92      1295
                5       0.87      0.88      0.88      1273
                6       0.96      0.93      0.94      1396
                7       0.95      0.93      0.94      1503
                8       0.92      0.85      0.88      1357
                9       0.92      0.89      0.91      1420

         accuracy                           0.92     14000
        macro avg       0.92      0.92      0.92     14000
     weighted avg       0.92      0.92      0.92     14000
```

We get an accuracy of 92% (approx) in this model.

2.4 CONVOLUTIONAL NEURAL NETWORK MODEL :

2.4.1 PROPOSED RECOGNITION SYSTEMS :

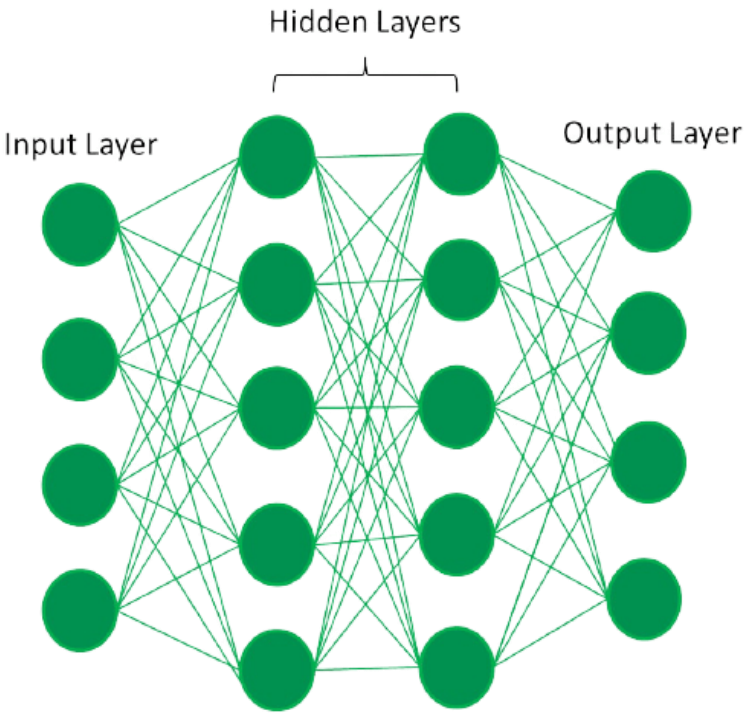
- DEEP LEARNING: Deep Learning is the subset of machine learning or can be said as a special kind of machine learning. It works technically in the same way as machine learning does, but with different capabilities and approaches. It is inspired by the functionality of human brain cells, which are called neurons, and leads to the concept of artificial neural networks. It is also called a deep neural network or deep neural learning.



Deep learning is the branch of machine learning which is based on Artificial neural network architecture. An artificial neural network or ANN uses layers of interconnected nodes called neurons that work together to process and learn from the input data.

ANN are built on the principles of the structure and operation of human neurons. It is also known as neural networks or neural nets. An ANN's input layer, which is the first layer, receives input from external sources and passes it on to the hidden layer, which is the second layer. Each neuron in the hidden layer gets information from the neurons in the previous layer, computes the weighted total, and then transfers it to the neurons in the next layer. These connections are weighted, which means that the impacts of the inputs from the

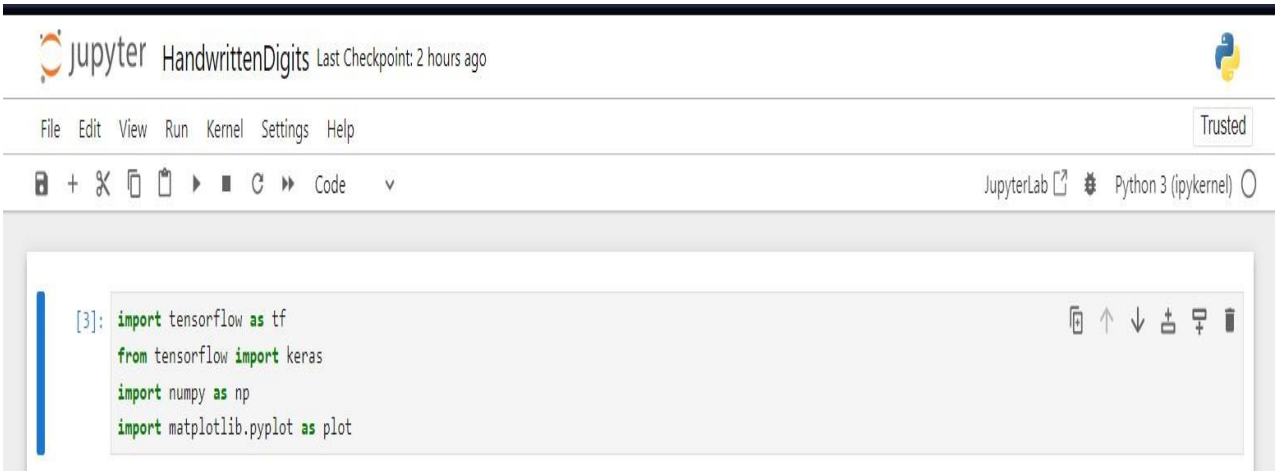
preceding layer are more or less optimized by giving each input a distinct weight.



A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

2.4.2 IMPLEMENTATION :



- TENSORFLOW: TensorFlow is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning. TensorFlow was originally developed for large numerical computations without keeping deep learning in mind. However, it proved to be very useful for deep learning development as well, and therefore Google open-sourced it.

  TensorFlow accepts data in the form of multi-dimensional arrays of higher dimensions called tensorFlow works on the basis of data flow graphs that have nodes and edges. As the execution mechanism is in the form of graphs, it is much easier to execute TensorFlow code in a distributed manner across a cluster of computers while using GPUs.

  KERAS : TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while **Keras** is a high-level neural network library that runs on top of TensorFlow. KERAS is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.

- PYTHON 3.11.4 : Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented and functional programming.

The 3.11.4 version of python is downloaded in the operating system from the website Welcome to Python.org.

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

- **JUPYTER NOTEBOOK**: A Jupyter notebook is an open source web application that allows data scientists to create and share documents that include live code, equations, and other multimedia resources. They are used for all sorts of data science tasks such as exploratory data analysis (EDA), data cleaning and transformation, data visualization, statistical modeling, machine learning, and deep learning.

  A Jupyter notebook has two components: a front-end web page and a back-end kernel. The front-end web page allows data scientists to enter programming code or text in rectangular "cells." The browser then passes the code to the back-end kernel which runs the code and returns the results.

- **MATPLOT LIBRARY:** Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible.

  - Create publication quality plots.
  - Make interactive figures that can zoom, pan, update.
  - Customize visual style and layout.
  - Export to many file formats.
  - Embedded in JupyterLab and GUI.
  - Use a rich array of third-party packages built on Matplotlib.

2.4.3 ANALYZING AND PREPARING THE DATASET:

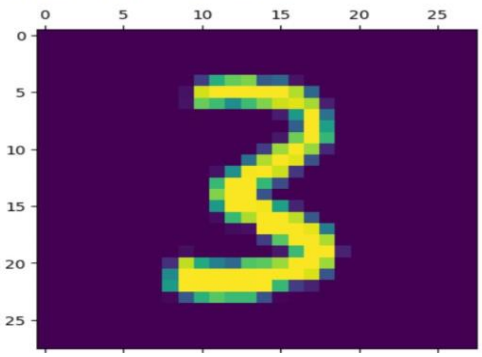
- ANALYZE :

```
[4]: (x_train,y_train), (x_test,y_test) = keras.datasets.mnist.load_data()
[5]: len(x_train)
[5]: 60000
[6]: len(x_test)
[6]: 10000
[7]: x_train[0].shape
[7]: (28, 28)
```

The dataset used in our model is the **MNIST Dataset** in CNN. Loading the dataset using function in keras. It returns two tuples. First tuple holding the train data and train label and another tuple holding the test data and test label.

The MNIST (Modified National Institute of Standards and Technology) database is a large database of handwritten numbers or digits that are used for training various image processing systems. The dataset also widely used for training and testing in the field of machine learning. The MNIST dataset has **60,000** training images and **10,000** testing images. It has a large amount of data and is commonly used to demonstrate the real power of deep neural networks. It is a multilevel dataset consisting of 10 classes in which we can classify numbers from 0 to 9. In the MNIST dataset, a single data point comes in the form of an image. These images included in the **MNIST** dataset are typical of **28*28** pixels such as 28 pixels crossing the horizontal axis and 28 pixels crossing the vertical axis.

```
[9]: plot.matshow(x_train[50000])
[9]: <matplotlib.image.AxesImage at 0x184849b2510>
```



- **NORMALIZE:**

```
[13]: x_train = x_train/255
      x_test = x_test/255
      x_train[0]
```

```
[13]: array([[0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        ],
             [0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        ],
             [0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
              0.        , 0.        , 0.        , 0.        , 0.        ,
```

As grayscale image intensities are scaled from 0 to 255, we need to normalize the values in the range of 0-1. This normalize step is to improve computational effieciency.

- **RESHAPE :**

```
[14]: #flattening 2-D array into 1-D
      x_train_flattened = x_train.reshape(len(x_train),28*28)
      x_train_flattened.shape

[14]: (60000, 784)

[15]: x_test_flattened = x_test.reshape(len(x_test),28*28)
      x_test_flattened.shape

[15]: (10000, 784)
```

This is a needed step. We need to convert the two-dimensional input data into a single-dimensional format for feeding into the model. This is achieved by a process called flattening. In this process, the 28 x 28 grid image is converted into a single-dimensional array of 784(28x28).

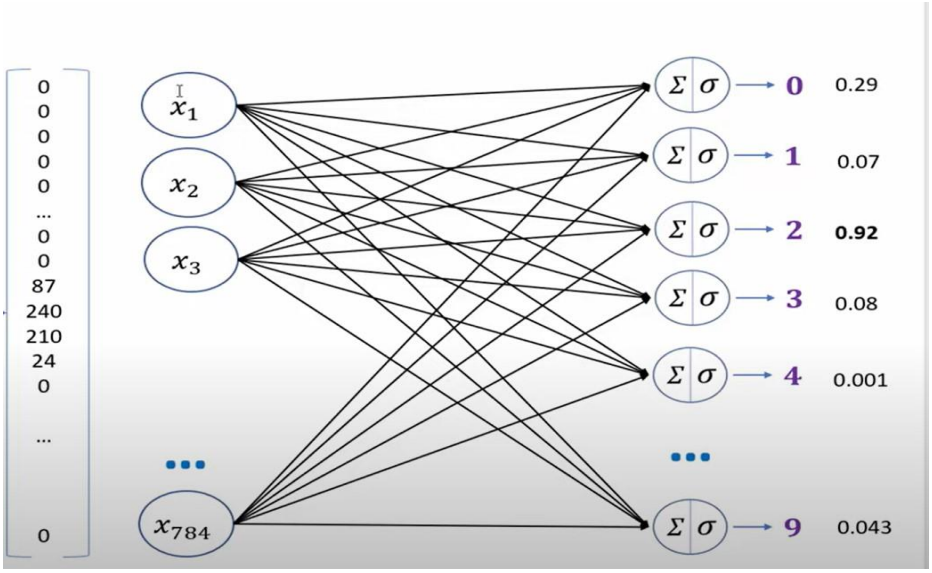
2.4.4 CREATING AND COMPILING THE MODEL :

A. (Without using hidden layer)

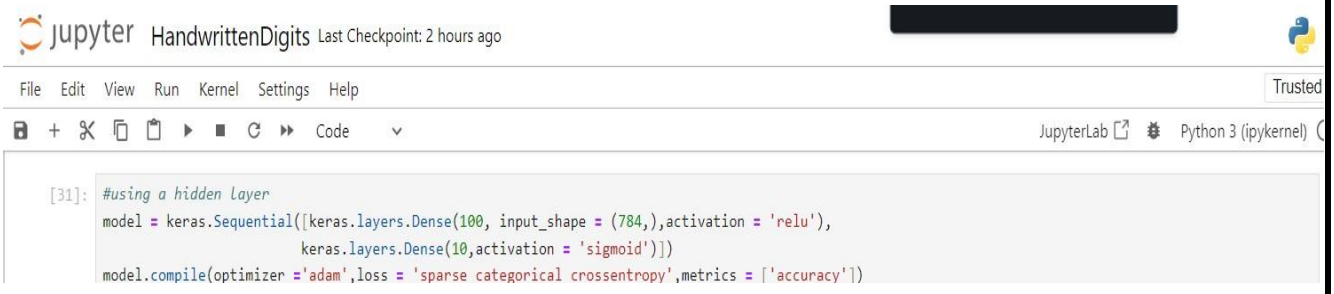
```
[16]: #defining our simple neural network
      model = keras.Sequential([keras.layers.Dense(10, input_shape = (784,),activation = 'sigmoid')])
      model.compile(optimizer ='adam',loss = 'sparse_categorical_crossentropy',metrics = ['accuracy'])
```

CREATING THE MODEL : In this model we have used two layer perceptron – the input and output layer. A perceptron is a single neuron model that is the basic building block to larger neural networks. We have used Sequential Keras model which has two pairs of Convolution2D and MaxPooling2D layers. The input layer has 784 neurons. The output layer has a dense layer with 10 outputs. **Sigmoid** is used as the activation function for the output layer.

COMPILING THE MODEL : After creating the model, we need to compile the model for optimization and learning. The optimizer used in the model is "Adam". In deep learning, optimizers are algorithms that adjust the model's parameters during training to minimize a loss function. They enable neural networks to learn from data by iteratively updating weights and biases. **Adam optimizer**, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training. The loss function used is "sparse categorical crossentropy". It is used as a loss function for multi-class classification model where the output layer is assigned integer value (0, 1, 2,…) .



B. (Using Hidden layer)



CREATING THE MODEL : In this model we have used multi-layer perceptron – the input, hidden and output layer. Sequential keras model is used. The input layer has 784 neurons, hidden layer has 100 neurons and the output layer is a dense layer with 10 outputs. **Relu** is the activation function used for hidden layer and **Sigmoid** is the activation function used for output layer.

COMPILING THE MODEL : Here also compilation is done with "Adam" optimizer and the loss function used is "Sparse categorical crossentropy".

2.4.5 FITTING THE MODEL :

A. WITHOUT HIDDEN LAYER:

```
model.fit(x_train_flattened, y_train, epochs = 5)

Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4662 - accuracy: 0.8782
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3032 - accuracy: 0.9155
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2830 - accuracy: 0.9207
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2733 - accuracy: 0.9236
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2663 - accuracy: 0.9258
[16]: <keras.src.callbacks.History at 0x18488385b90>
```

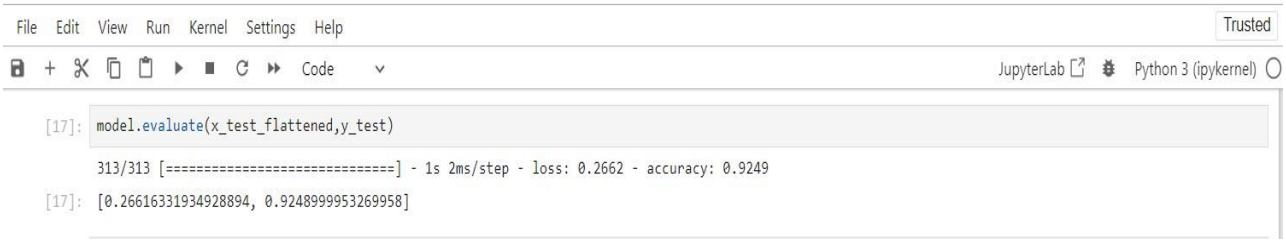
B. USING HIDDEN LAYER

```
model.compile(optimizer = adam ,loss = 'sparse_categorical_crossentropy',metrics = [ accuracy ])
model.fit(x_train_flattened, y_train, epochs = 5)

Epoch 1/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2699 - accuracy: 0.9227
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1213 - accuracy: 0.9642
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0847 - accuracy: 0.9743
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0641 - accuracy: 0.9807
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0510 - accuracy: 0.9842
[31]: <keras.src.callbacks.History at 0x184a9319910>
```

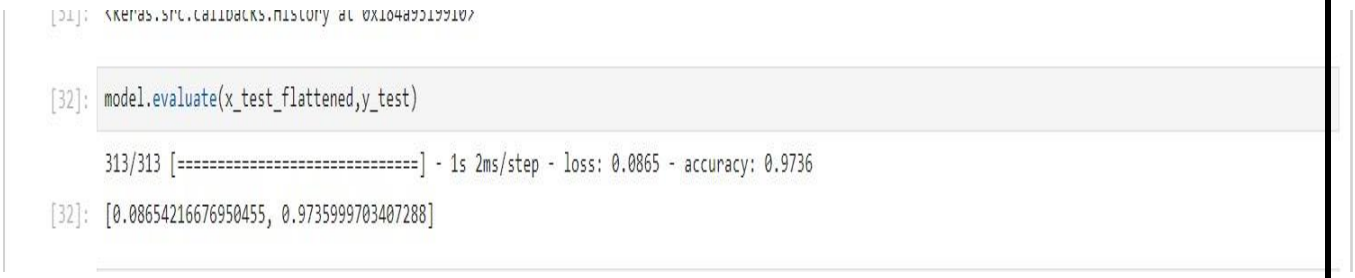
After compilation, we need to fit the model using MNIST dataset. Model fitting is done to measure how well a machine learning model generalizes to similar data to that on which it was trained. A function "fit" is called where we passed the training data as first parameter and target values as second parameter. Number of epochs given is 5, an epoch means one complete pass of the training dataset through the algorithm.

2.4.6 EVALUATING THE MODEL :

A. WITHOUT HIDDEN LAYER:



B. USING HIDDEN LAYER:



After fitting the model, the model can be evaluated on the unseen test data. Matplotlib is used to visualize how our model reacts at different epochs on both training and testing data.

After evaluating, our result shows that without using hidden layer the accuracy obtained is 92.48 and using hidden layer the accuracy obtained is 97.35.

2.4.7 CONFUSION MATRIX:

A **confusion matrix** is a matrix that summarizes the performance of a machine learning model on a set of test data. It is often used to measure the performance of classification models, which aim to predict a categorical label for each input instance. The matrix displays the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) produced by the model on the test data.

For binary classification, the matrix will be of a 2X2 table, For multi-class classification, the matrix shape will be equal to the number of classes i.e for n classes it will be nXn.

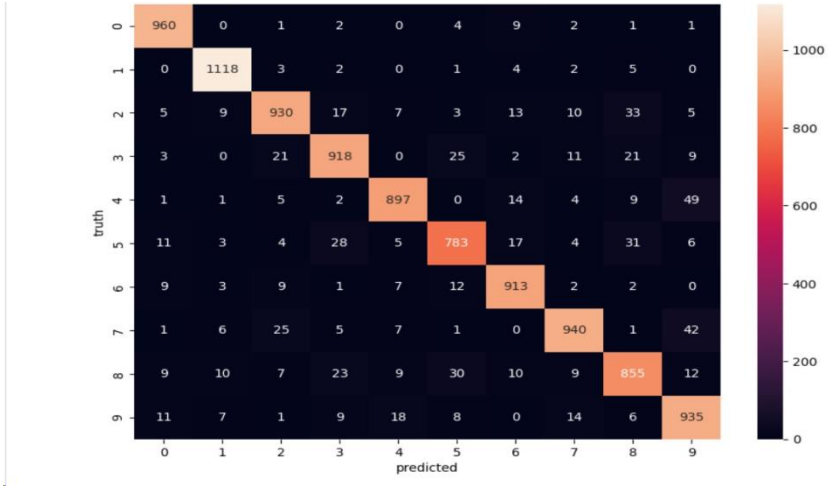
A. Without hidden layer ->

```
[23]: #checking how many errors our model has
      y_predicted_labels = [np.argmax(i) for i in y_predicted]
      y_predicted_labels[:5]

[23]: [7, 2, 1, 0, 4]

[27]: cm = tf.math.confusion_matrix(labels = y_test, predictions = y_predicted_labels)
      cm

[27]: <tf.Tensor: shape=(10, 10), dtype=int32, numpy=
      array([[ 960,    0,    1,    2,    0,    4,    9,    2,    1,    1],
             [   0, 1118,    3,    2,    0,    1,    4,    2,    5,    0],
             [   5,    9,  930,   17,    7,    3,   13,   10,   33,    5],
             [   3,    0,   21,  918,    0,   25,    2,   11,   21,    9],
             [   1,    1,    5,    2,  897,    0,   14,    4,    9,   49],
             [  11,    3,    4,   28,    5,  783,   17,    4,   31,    6],
             [   9,    3,    9,    1,    7,   12,  913,    2,    2,    0],
             [   1,    6,   25,    5,    7,    1,    0,  940,    1,   42],
             [   9,   10,    7,   23,    9,   30,   10,    9,  855,   12],
             [  11,    7,    1,    9,   18,    8,    0,   14,    6,  935]])>
```

```
[30]: import seaborn as sn
      plot.figure(figsize = (10,7))
      sn.heatmap(cm,annot = True, fmt ='d')
      plot.xlabel('predicted')
      plot.ylabel('truth')

[30]: Text(95.72222222222221, 0.5, 'truth')
```

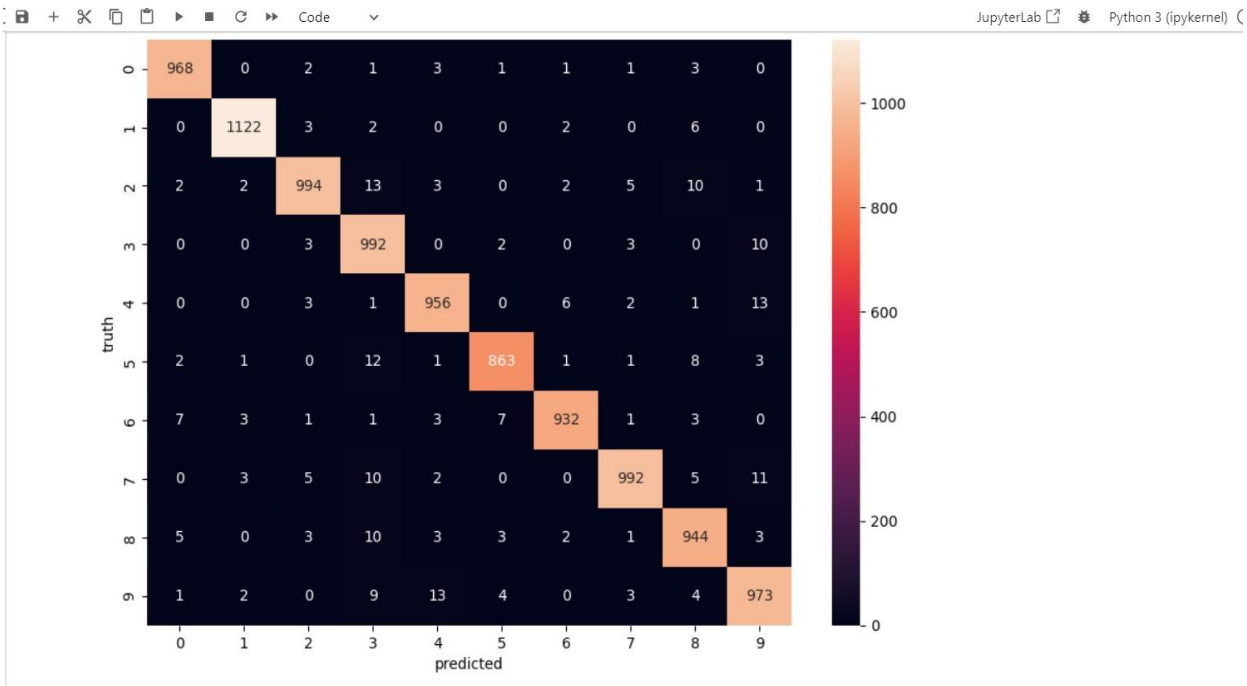
Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas data structures.

Seaborn helps you explore and understand your data. Its plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots. Its dataset-oriented, declarative API lets you focus on what the different elements of your plots mean, rather than on the details of how to draw them.



B. Using hidden layer –

```
[33]: y_predicted = model.predict(x_test_flattened)
      y_predicted_labels = [np.argmax(i) for i in y_predicted]
      cm = tf.math.confusion_matrix(labels = y_test, predictions = y_predicted_labels)

      import seaborn as sn
      plot.figure(figsize = (10,7))
      sn.heatmap(cm,annot = True, fmt ='d')
      plot.xlabel('predicted')
      plot.ylabel('truth')

      313/313 [==============================] - 1s 2ms/step
[33]: Text(95.72222222222221, 0.5, 'truth')
```



3. FINDINGS AND ANALYSIS:

Sl no.	METHOD	ACCURACY	PROS	CONS
1	LOGISTIC REGRESSION	91.47%	Logistic regression is easier to implement, interpret and very efficient to train.	Logistic Regression can only be used to predict discrete functions. Therefore, the dependent variable of Logistic Regression is restricted to the discrete number set. This restriction itself is problematic, as it is prohibitive to the prediction of continuous data.
2	SVM(KERNEL-RBF)	96.57%	RBF Kernel is popular because of its similarity to K-Nearest Neighborhood Algorithm. It has the advantages of K-NN and overcomes the space complexity problem as RBF Kernel Support Vector Machines just needs to store the support vectors during training and not the entire dataset.	if the data is linearly separable in the expanded feature space, the linear SVM maximizes the margin better and can lead to a sparser solution
3	SVM(LINEAR KERNEL)	92%	Linear Kernel is used when the data is Linearly separable, that is, it can be separated using a single Line. It is one of the most common kernels to be used. It is mostly used when there are a Large number of Features in a particular Data Set	If the data is linearly separable in the expanded feature space, the linear SVM maximizes the margin better and can lead to a sparser solution.
4	CNN(Without using hidden layer)	92.48%	The CNN architecture is especially useful for image recognition and image classification, as well as other computer vision tasks because they can process large amounts of data and produce highly accurate predictions.	• High computational requirements. • Needs large amount of labeled data. • Large memory footprint. • Interpretability challenges.
	CNN(USING HIDDEN LAYER)	97.35%		

4. CONCLUSION :

In this study, the achievements and performances of different machine learning algorithms in handwritten digit recognition process were examined. The algorithms selected were Logistic Regression, SVM (linear and non-linear kernel) and CNN. All of them were discussed in detail and trained and tested on the same MNIST dataset to find comparisons between them. During these experiments, Python programming language was used, along with Keras and Scikit-learn libraries accordingly. The result reports and accuracy values were obtained using the Scikit-learn library metrics to compare the success rate of each algorithm. With deeper learning models, higher accuracy levels were found. Highest accuracy was achieved with CNN, with a success rate of 97.35% with hidden layer and lowest with Logistic Regression at 91.47%. This shows that a neural network is more successful than other algorithms. CNN is the most widely used machine learning algorithm. The algorithms unattended in this study are decision tree, random forest, ANN and K-Nearest Neighbor, which also provide high efficiency in recognition and make very little mistakes in clustering. However, linear regression is not used for recognition as it assumes a linear relationship between input and output, which does not capture the intricate variations and features present in handwritten digits. Handwritten digit recognition involves complex patterns and relationships within data that are non-linear and not easily represented by a linear model.

With the increase in data being generated every day, machine learning represents a transformative force reshaping industries and enabling advancements across various domains by teaching machines to learn from data. These models provide insights, predictions and automation that drive innovation and efficiency. Its continued evolution promises to revolutionize how we solve complex real-life problems, empowering businesses and society through data-driven decision-making and personalized experiences.

BIBLIOGRAPHY

1. A Scalable Handwritten Text Recognition System R. Reeve Ingle, Yasuhisa Fujii, Thomas Deselaers, Jonathan Baccash, Ashok C. Popat Google Research Mountain View, CA 94043, USA. Available: https://arxiv.org/abs/1904.09150
2. A REVIEW ON CONVERSION OF HANDWRITTEN NOTES TO DIGITALIZED VERSION USING TENSER FLOW. Ruchita Mute*1, Dhanashri Khadse*2, Ayushi Lanjewar*3, Snehal Selokar*4, Divya Yadav*5, Jayant Adhikari*6
   Available:
   https://www.irjmets.com/uploadedfiles/paper/issue_6_june_2023/41473/final/fin_irjmets1685957474.pdf
3. Machine Learning for Handwriting Recognition Preetha Sa *, Afrid I Mb , Karthik Hebbar Pc , Nishchay S Kd a,b,c,d Department of ISE,B.M.S. College of Engineering/ VTU, India, {preetha.ise,1bm16is006, 1bm16is044, 1bm16is060}
   Available: https://core.ac.uk/download/pdf/327266589.pdf
4. A Comparative Study of Handwriting Recognition Techniques, Prem Chand Vashist Department of Information Technology G. L. Bajaj Institute of Technology and Management, Anmol Pandey Department of Information Technology G. L. Bajaj Institute of Technology and Management Greater Noida, India, Ashish Tripathi Department of Information Technology G. L. Bajaj Institute of Technology and Management Greater Noida, India
5. Handwritten Digit Recognition System using Machine Learning in Python A Project report submitted in partial fulfilment of the requirements for the award of the degree of BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE ENGINEERING
   Available:
   http://www.ir.juit.ac.in:8080/jspui/bitstream/123456789/8269/1/Handwritten%20Digit%20Recognizer.pdf
6. https://scikit-learn.org/
7. https://www.tensorflow.org/datasets/catalog/mnist