# A Comparative Study on Handwritten Digits Recognition using Classifiers like K-NN, Multiclass Perceptron and SVM

---

## Problem Definition

### Problem Statement:

The task at hand is to classify handwritten digits using supervised machine learning methods. The digits belong to classes of 0 – 9.

*"Given a query instance (a digit) in the form of an image, our machine learning model must correctly classify its appropriate class."*

### Problem Formulation:

Tom Mitchell's formalism for machine learning can be used to formulate our machine learning problem. It says:

A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at task *T*, as measured by *P*, improves with experience *E*.

So, our problem of handwritten digits recognition can be formulated as:

- **T** – classifying handwritten digits within images.
- **P** – percent of digits correctly classified.
- **E** – dataset of handwritten digits with given labels.

## Applications:

Handwritten digits are a common part of everyday life. These days machine learning methods are classifying handwritten digits with accuracy exceeding human accuracy.

These methods are used to make **Optical Character Recognizers** (OCRs) which involve reading text from paper and translating the images into a form that the computer can manipulate (for example, into ASCII codes). This technology is solving a plethora of problems like automated recognition of:

### 1- Zip Codes:

A zip code consists of some digits and is one of the most important parts of a letter for it to be delivered to the correct location. Many years ago, the postman would read the zip code manually for delivery. However, this type of work is now automated by using **optical character recognition** (OCR).

### 2- Bank Cheques:

The most frequent use of OCR is to handle cheques: a handwritten cheque is scanned, its contents converted into digital text, the signature verified and the cheque cleared in real time, all without human involvement.

### 3- Legal Records:

Reams and reams of affidavits, judgements, filings, statements, wills and other legal documents, especially the printed ones, can be digitized, stored, databased and made searchable using the simplest of OCR readers.

### 4- Healthcare Records:

Having one’s entire medical history on a searchable, digital store means that things like past illnesses and treatments, diagnostic tests, hospital records, insurance payments etc. can be made available in one unified place, rather than having to maintain unwieldy files of reports and X-rays. OCR can be used to scan these documents and create a digital database.

## Dataset

MNIST Handwritten Digits dataset is used for this task. It contains images of digits taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required.

Each image is a 28 by 28 pixel square (784 pixels total). The dataset contains 60,000 images for model training and 10,000 images for the evaluation of the model.

*Figure 1 - Sample images from the MNIST dataset.*

## Methodology

We have used supervised machine learning models to predict the digits. Since this is a comparative study hence we will first describe the K-Nearest Neighbors Classifier as the baseline method which will then be compared to Multiclass Perceptron Classifier and SVM Classifier.

### <u>K-Nearest Neighbors Classifier – Our Baseline Method:</u>

k-Nearest Neighbors (k-NN) is an algorithm, which:

- finds a group of k objects in the training set that are closest to the test object, and
- bases the assignment of a label on the predominance of a class in this neighborhood.

There are three key elements of this approach:

- a set of labeled objects, e.g., a set of stored records (training data)
- a distance or similarity metric to compute distance between objects
- and the value of k (the number of nearest neighbors)

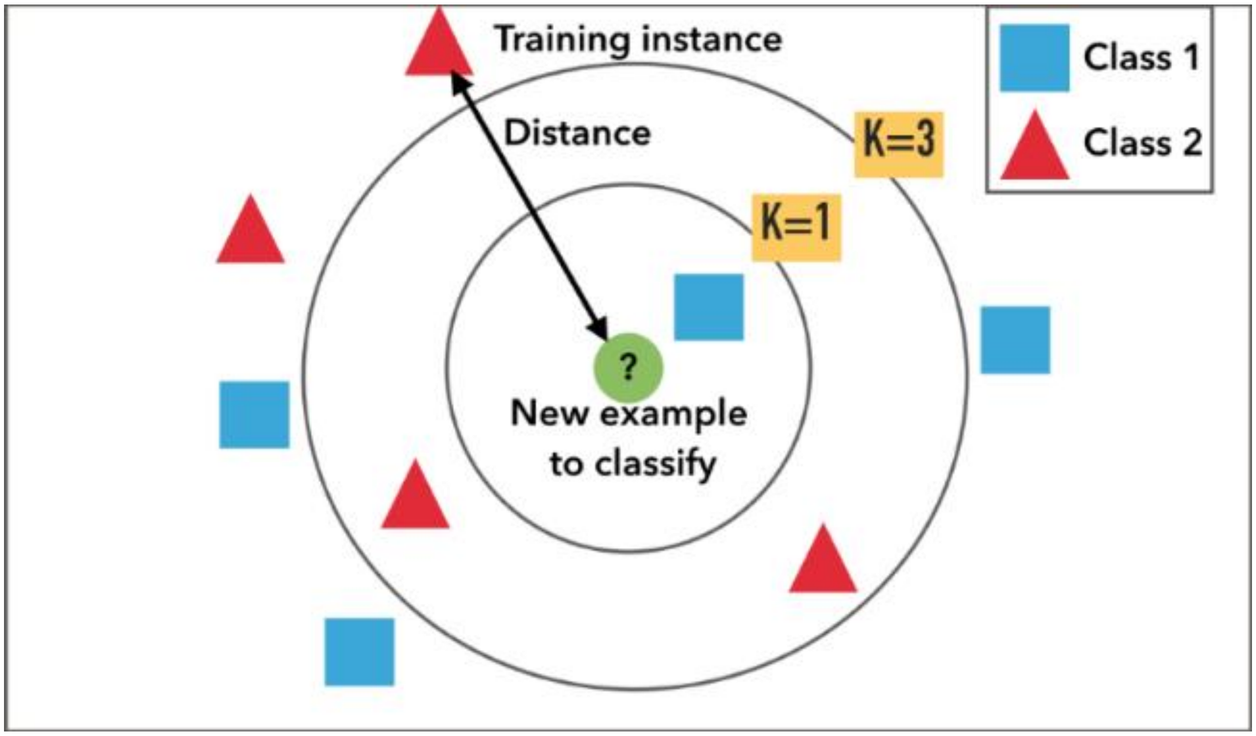


*Figure 2 - An intuitive representation of how a test instance is classified using K-NN*

To classify an unlabeled image:

- the distance of this image to the labeled objects (training data) is computed
- its k-nearest neighbors are identified and sorted in descending order according to similarities computed
- the most occurred nearest neighbor in top k neighbors is then assigned as the class label of the object

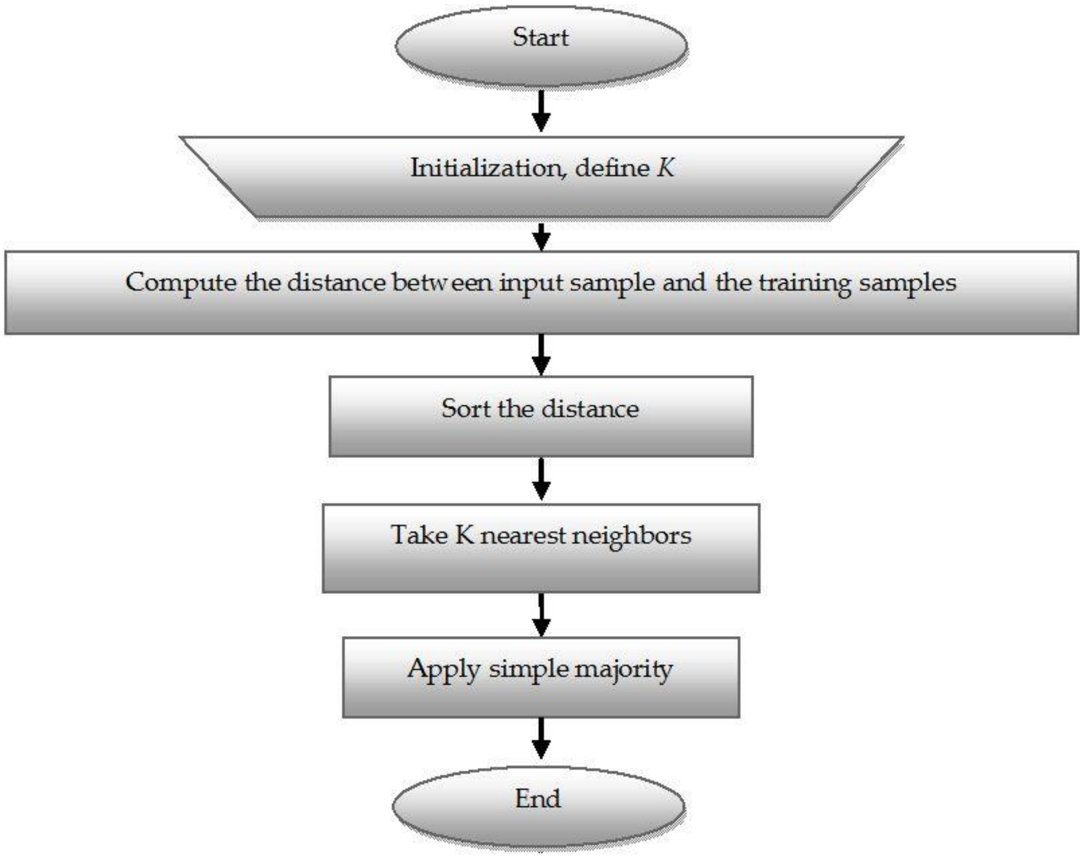
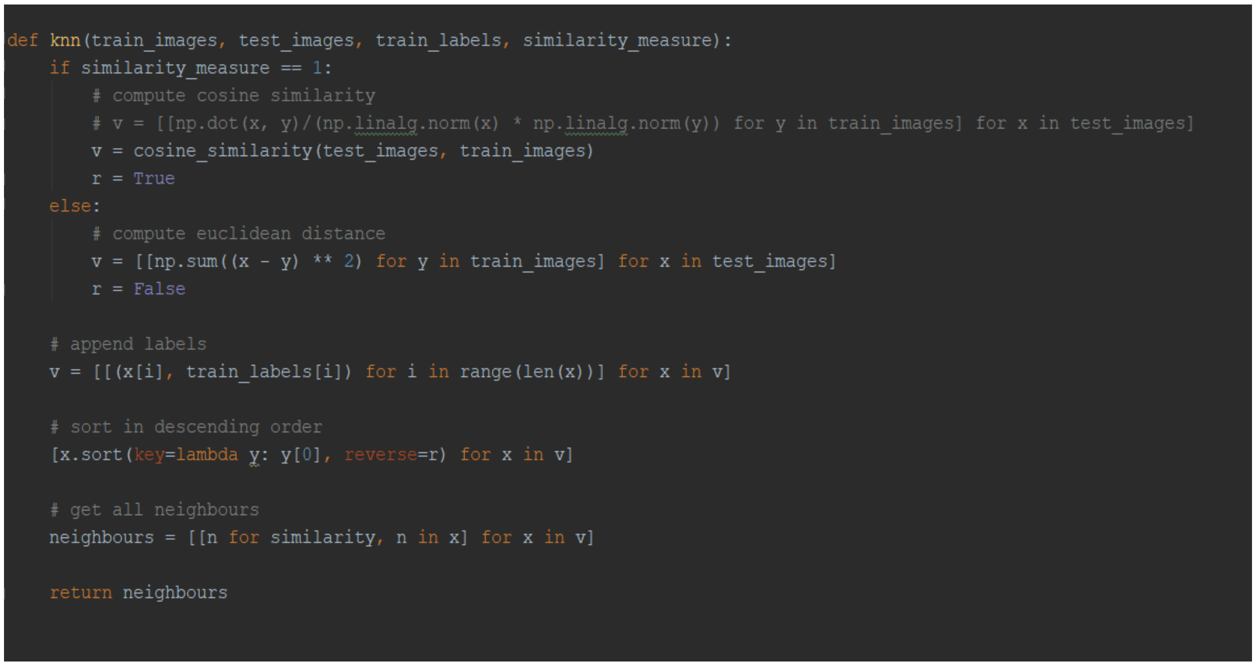
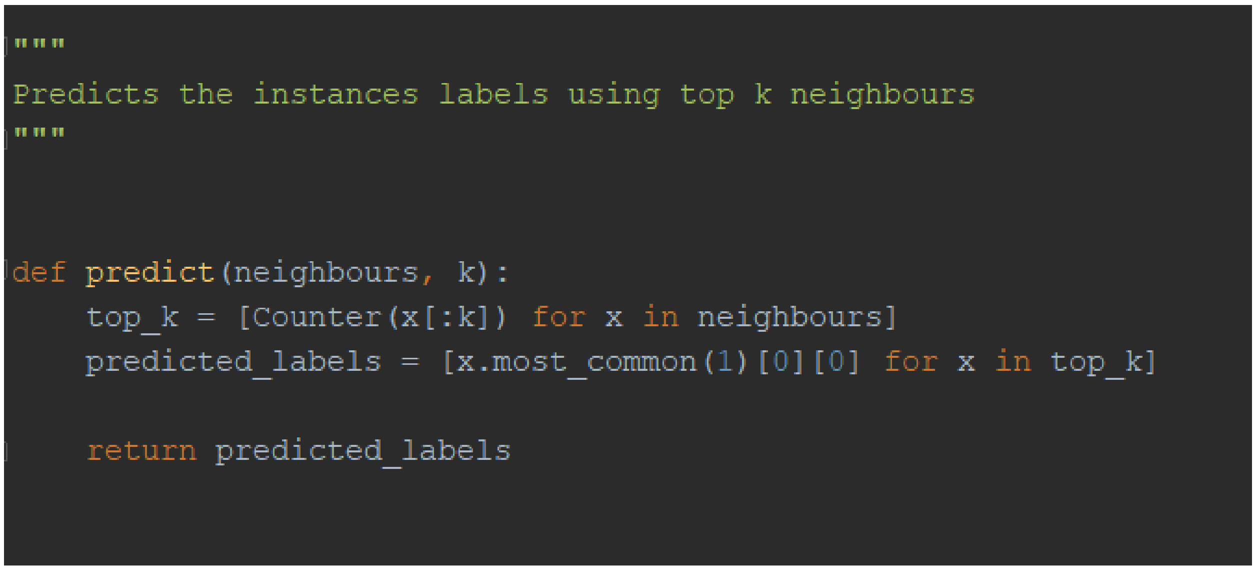


*Figure 3 - Steps involved in K-NN Algorithm*



```
def knn(train_images, test_images, train_labels, similarity_measure):
    if similarity_measure == 1:
        # compute cosine similarity
        # v = [[np.dot(x, y)/(np.linalg.norm(x) * np.linalg.norm(y)) for y in train_images] for x in test_images]
        v = cosine_similarity(test_images, train_images)
        r = True
    else:
        # compute euclidean distance
        v = [[np.sum((x - y) ** 2) for y in train_images] for x in test_images]
        r = False

    # append labels
    v = [[(x[i], train_labels[i]) for i in range(len(x))] for x in v]

    # sort in descending order
    [x.sort(key=lambda y: y[0], reverse=r) for x in v]

    # get all neighbours
    neighbours = [[n for similarity, n in x] for x in v]

    return neighbours
```

*Figure 4 - K-NN implementation of my code.*



```
"""
Predicts the instances labels using top k neighbours
"""


def predict(neighbours, k):
    top_k = [Counter(x[:k]) for x in neighbours]
    predicted_labels = [x.most_common(1)[0][0] for x in top_k]

    return predicted_labels
```

*Figure 5 - After the similarity is computed with all the training instances, the top k neighbors are chosen and the predominant one is assigned to the test instance.*

When we used the K-NN method the following pros and cons were observed:

## Pros:

- K-NN executes quickly for small training data sets.
- No assumptions about data — useful, for example, for nonlinear data
- Simple algorithm — to explain and understand/interpret
- Versatile — useful for classification or regression
- Training phase is extremely quick because it doesn’t learn any data

## Cons:

- Computationally expensive — because the algorithm compares the test data with all examples in training data and then finalizes the label
- The value of K is unknown and can be predicted using cross validation techniques
- High memory requirement – because all the training data is stored
- Prediction stage might be slow if training data is large

## Multiclass Perceptron Classifier:

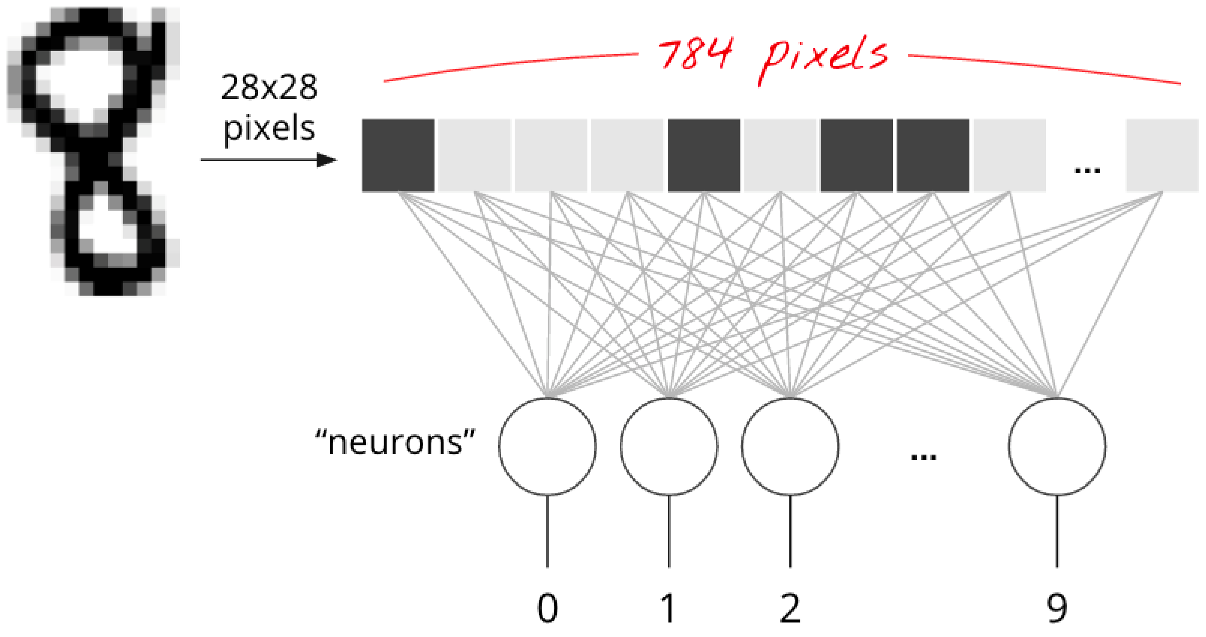


*Figure 6 - A query instance is being fed to all classifiers in the Multiclass Perceptron Classifier*

A multiclass perceptron classifier can be made using multiple binary class classifiers trained with **1 vs all** strategy. In this strategy, while training a perceptron the training labels are such that e.g. for the classifier 2 vs all, the labels with 2 will be labeled as 1 and rest will be labeled as 0 for Sigmoid Unit while for Rosenblatt's perceptron the labels would be 1 and -1 respectively for positive and negative examples.

Now all we have to do is to train (learn the weights for) 10 classifiers separately and then feed the query instance to all these classifiers (as shown in figure above). The label of classifier with highest confidence will then be assigned to the query instance.

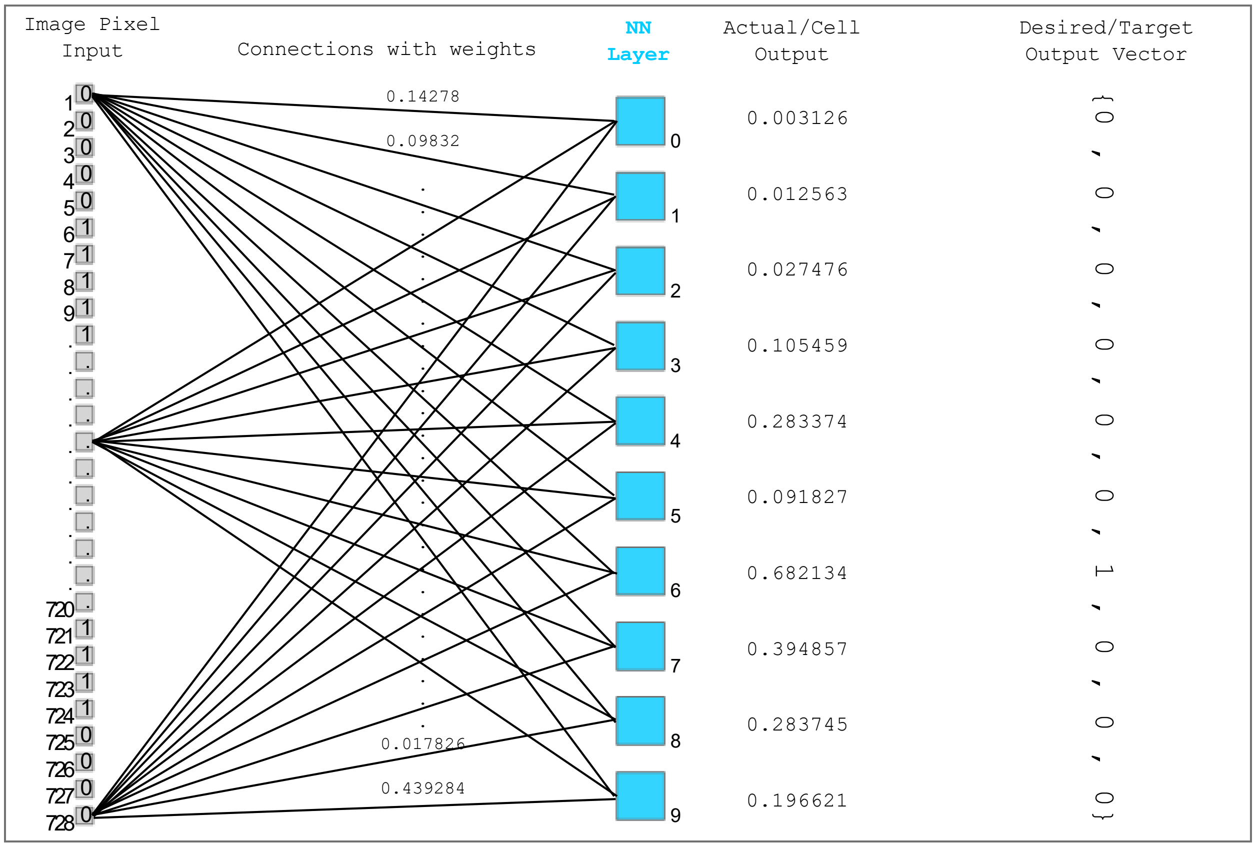


*Figure 7 - A query instance is fed to the 1 Layer Multiperceptron Classifier and the label of classifier with highest confidence is assigned to the test instance, in this case classifier '6' has highest confidence so its assigned to the test query.*

## Training Algorithm:

The training algorithm for each perceptron is as follows:

1. Initialize the weights and threshold to small random numbers (in our case – zero).
2. Present a vector **x** to the neuron inputs and calculate the output.
3. Update the weights according to:

$$\boxed{w_j(t+1) = w_j(t) + \eta(d-y)x}$$

where

**d** is the desired output which uses signum function in case of perceptron rule and sigmoid function in case of delta rule

**t** is the iteration number, and

**eta** is the learning rate or step size, where 0.0 < eta < 1.0

4. Repeat steps 2 and 3 until:

   the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed

## Activation Functions:

An activation function is applied to the output of a perceptron in order to decide whether to fire that neuron or not. On the basis of the activation functions used, we have two types of perceptrons:

- Perceptron Rule with Signum Activation Function
- Delta Rule with Sigmoid Activation Function

## 1) Perceptron Rule with Signum Activation Function: (Rosenblatt’s Perceptron)

In this rule, our hypothesis function is as follows:

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Given below is a visual representation of Rosenblatt's Perceptron:

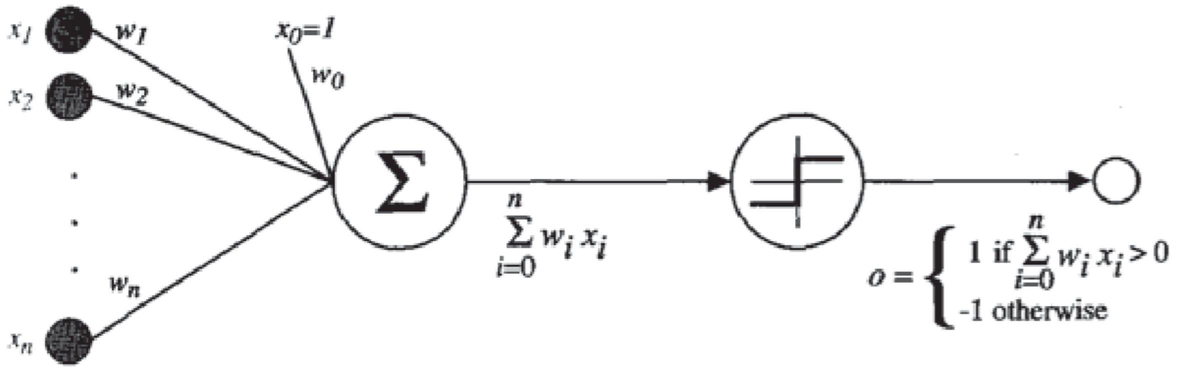
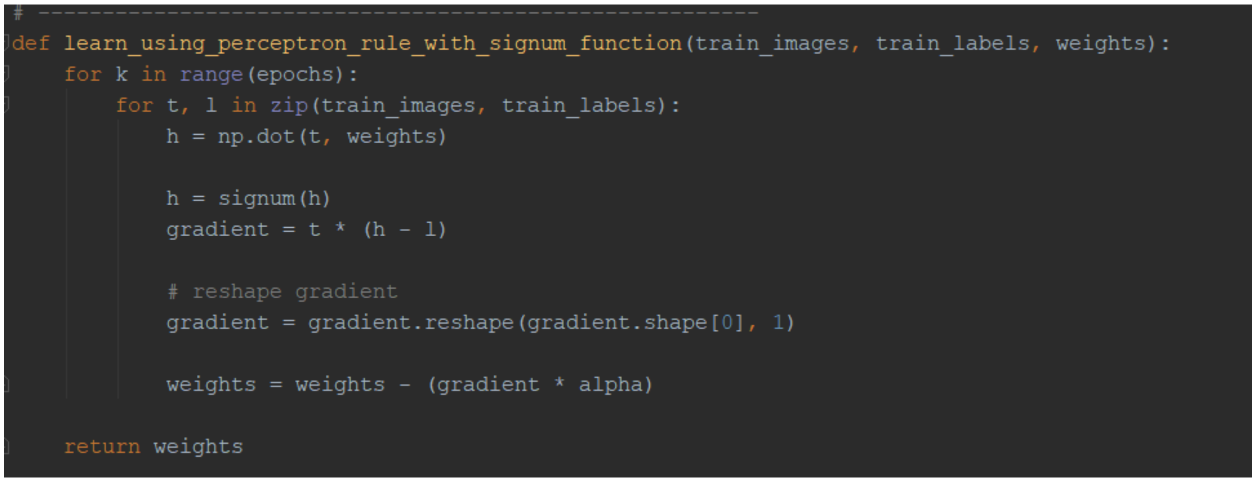

$x_1$
$w_1$
$x_2$
$w_2$
$x_0=1$
$w_0$
$w_n$
$x_n$
$\Sigma$
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

*Figure 8 - Rosenblatt's Perceptron with Signum Activation Function*

Here's my training code for a single perceptron:



```
# -------------------------------------------------------
def learn_using_perceptron_rule_with_signum_function(train_images, train_labels, weights):
    for k in range(epochs):
        for t, l in zip(train_images, train_labels):
            h = np.dot(t, weights)

            h = signum(h)
            gradient = t * (h - l)

            # reshape gradient
            gradient = gradient.reshape(gradient.shape[0], 1)

            weights = weights - (gradient * alpha)

    return weights
```

*Figure 9 - My code snippet for training of Rosenblatt's Perceptron for some predefined epochs using stochastic gradient descent for the minimization of its cost function and upgradation of weights*

## 2) Delta Rule with Sigmoid Activation Function:

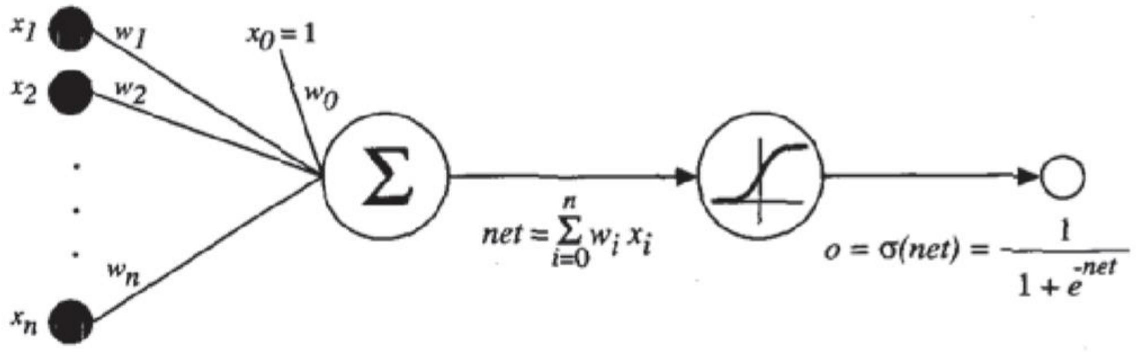
In this rule, our hypothesis function is as follows:

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

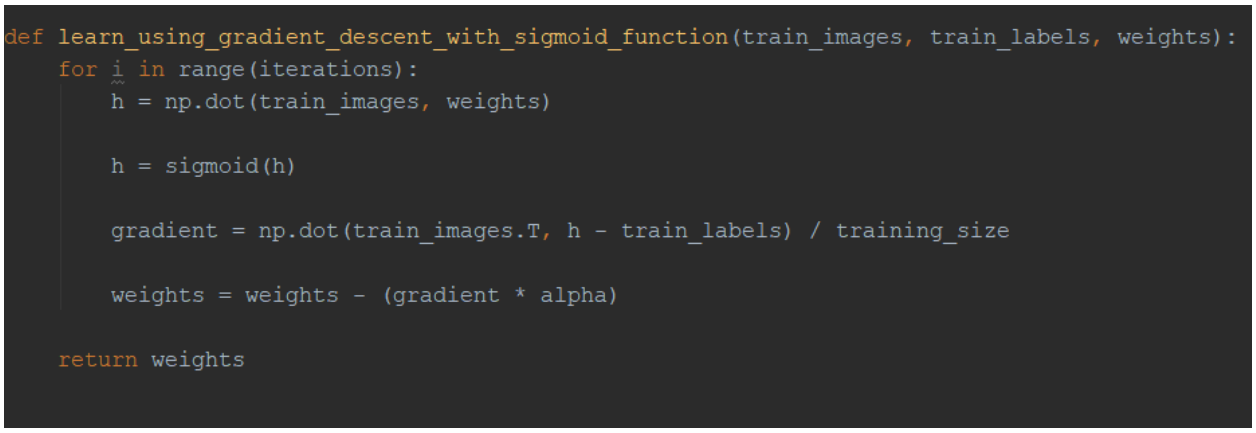
Given below is a visual representation of Perceptron with Sigmoid Activation Function:



$x_1$
$x_2$
$x_n$
$w_1$
$w_2$
$w_n$
$x_0 = 1$
$w_0$
$\Sigma$
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

*Figure 10 – Perceptron with Sigmoid Activation Function*

Here’s my training code for a single perceptron:



```
def learn_using_gradient_descent_with_sigmoid_function(train_images, train_labels, weights):
    for i in range(iterations):
        h = np.dot(train_images, weights)

        h = sigmoid(h)

        gradient = np.dot(train_images.T, h - train_labels) / training_size

        weights = weights - (gradient * alpha)

    return weights
```

*Figure 11 - My code snippet for training of Perceptron with Sigmoid Activation Function using Batch Mode of Gradient Descent*

## Difference between Perceptron Rule and Delta Rule:

There are two differences between the perceptron and the delta rule:

- The perceptron rule is based on an output from a step function (signum function), whereas the delta rule uses the sigmoid activation function.

- The perceptron is guaranteed to converge to a consistent hypothesis assuming the data is linearly separable (in case it isn’t we then define epochs to train). On the other hand, the delta rule’s convergence does not depend on the condition of linearly separable data.

## How Multiclass Perceptron mitigates the limitations of K-NN:

As we already discussed, K-NN stores all the training data and when a new query instance comes it compares its similarity with all the training data which makes it expensive both computationally and memory-wise. There is no learning involved as such. On the other hand, Multiclass perceptron takes some time in learning phase but after its training is done, it learns the new weights which can be saved and then used. Now, when a query instance comes, it only has to take to dot product of that instance with the weights learned and there comes the output (after applying activation function).

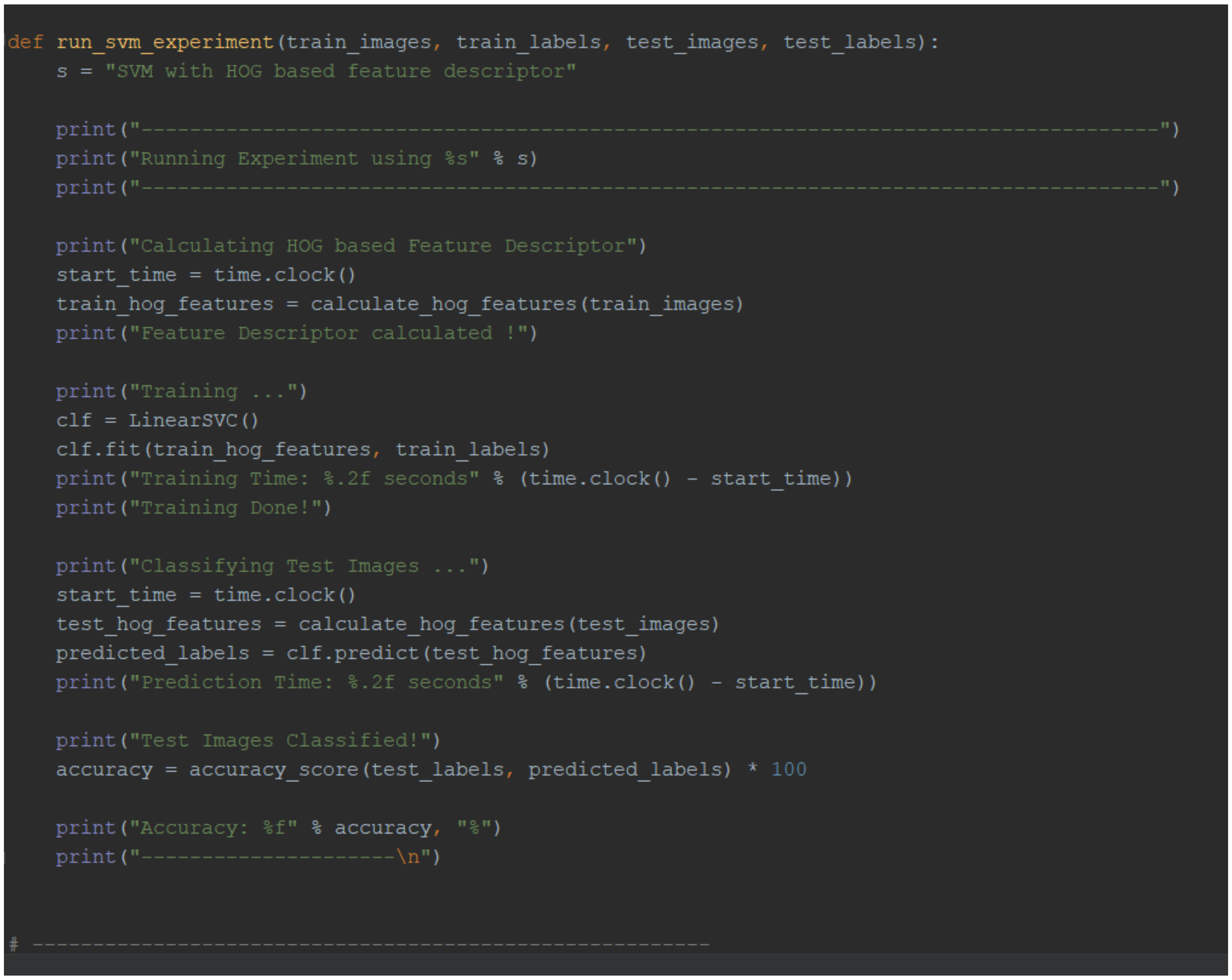
- ***The prediction phase is extremely fast as compared to that of K-NN.***

- ***Also, it's a lot more efficient in terms of computation (during prediction phase) and memory (because now it only has to store the weights instead of all the training data).***

## SVM Classifier:

Just for comparison purposes, we have also used a third supervised machine learning technique named Support Vector Machine Classifier.

The model isn't implemented. Its imported directly from scikit learn module of python and used.



```
def run_svm_experiment(train_images, train_labels, test_images, test_labels):
    s = "SVM with HOG based feature descriptor"

    print("-------------------------------------------------------------------------------------")
    print("Running Experiment using %s" % s)
    print("-------------------------------------------------------------------------------------")

    print("Calculating HOG based Feature Descriptor")
    start_time = time.clock()
    train_hog_features = calculate_hog_features(train_images)
    print("Feature Descriptor calculated !")

    print("Training ...")
    clf = LinearSVC()
    clf.fit(train_hog_features, train_labels)
    print("Training Time: %.2f seconds" % (time.clock() - start_time))
    print("Training Done!")

    print("Classifying Test Images ...")
    start_time = time.clock()
    test_hog_features = calculate_hog_features(test_images)
    predicted_labels = clf.predict(test_hog_features)
    print("Prediction Time: %.2f seconds" % (time.clock() - start_time))

    print("Test Images Classified!")
    accuracy = accuracy_score(test_labels, predicted_labels) * 100

    print("Accuracy: %f" % accuracy, "%")
    print("---------------------\n")


# --------------------------------------------------------
```

*Figure 12 - Running Experiment with SVM Classifier*

# Experimental Settings

## Feature Scaling:

**Feature scaling** is a method used to standardize the range of independent variables or features of data. In data processing, it is also known as data normalization and is generally performed during the data preprocessing step.

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, the majority of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

### Rescaling:

The simplest method is rescaling the range of features to scale the range in [0, 1] or [−1, 1]. Selecting the target range depends on the nature of the data. The general formula is given as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where $x$ is an original value, $x'$ is the normalized value.

Our MNIST dataset is first rescaled. Since the min value of every pixel is 0 and maximum values is 255. Hence each input vector is divided by 255 to rescale it.

## Experimental Settings for K-NN:

The K-NN experiment was run twice. Once with Cosine Similarity Measure and once with Euclidean Distance as similarity measure.

### Data Split:

After running the algorithm for different splits of training and testing data, the following split is chosen since it served the purpose of better results:

| | |
|---|---|
| Training Data Used: | 3000 Data Points |
| Testing Data Used: | 1000 Data Points |
| Validation Data Used: | 1000 Data Points |

### Similarity Measure (Distance Measure):

There are two major things to be considered for K-NN. One is the similarity measure to be used and the most important one is the value of K.

We opted two similarity measures and ran the experiment once for each.

1- **Cosine Similarity:**

The cosine similarity between two vectors is a measure that calculates the cosine of the angle between them. This metric is a measurement of orientation and not magnitude.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}}$$

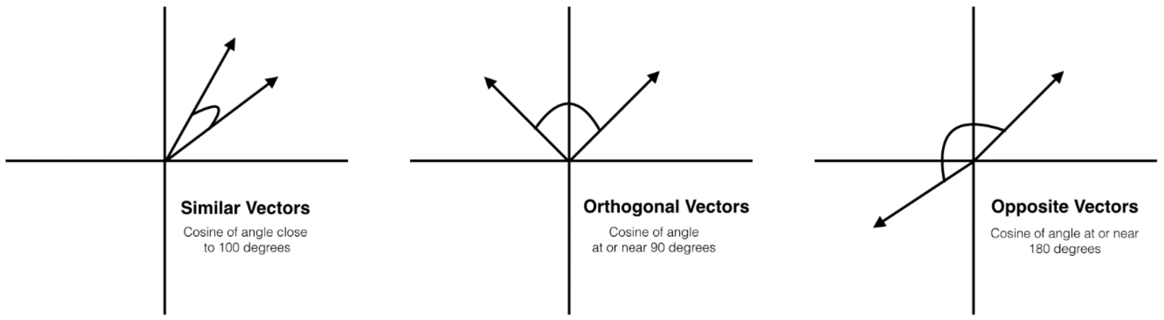

Similar Vectors
Cosine of angle close to 100 degrees
Orthogonal Vectors
Cosine of angle at or near 90 degrees
Opposite Vectors
Cosine of angle at or near 180 degrees

*Figure 13 – Similarity of Two Vectors according to the cosine of angle between them.*

## 2- Euclidean Distance:

The **Euclidean distance** is the straight-line distance between two vectors in Euclidean space. It tells how far two vectors from each other are by taking into account their magnitude.

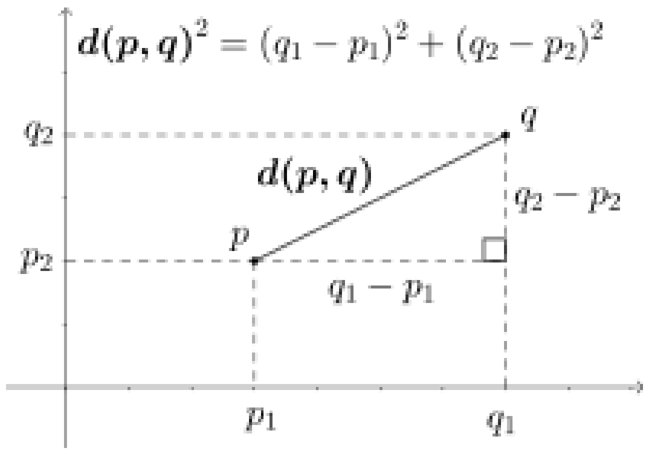


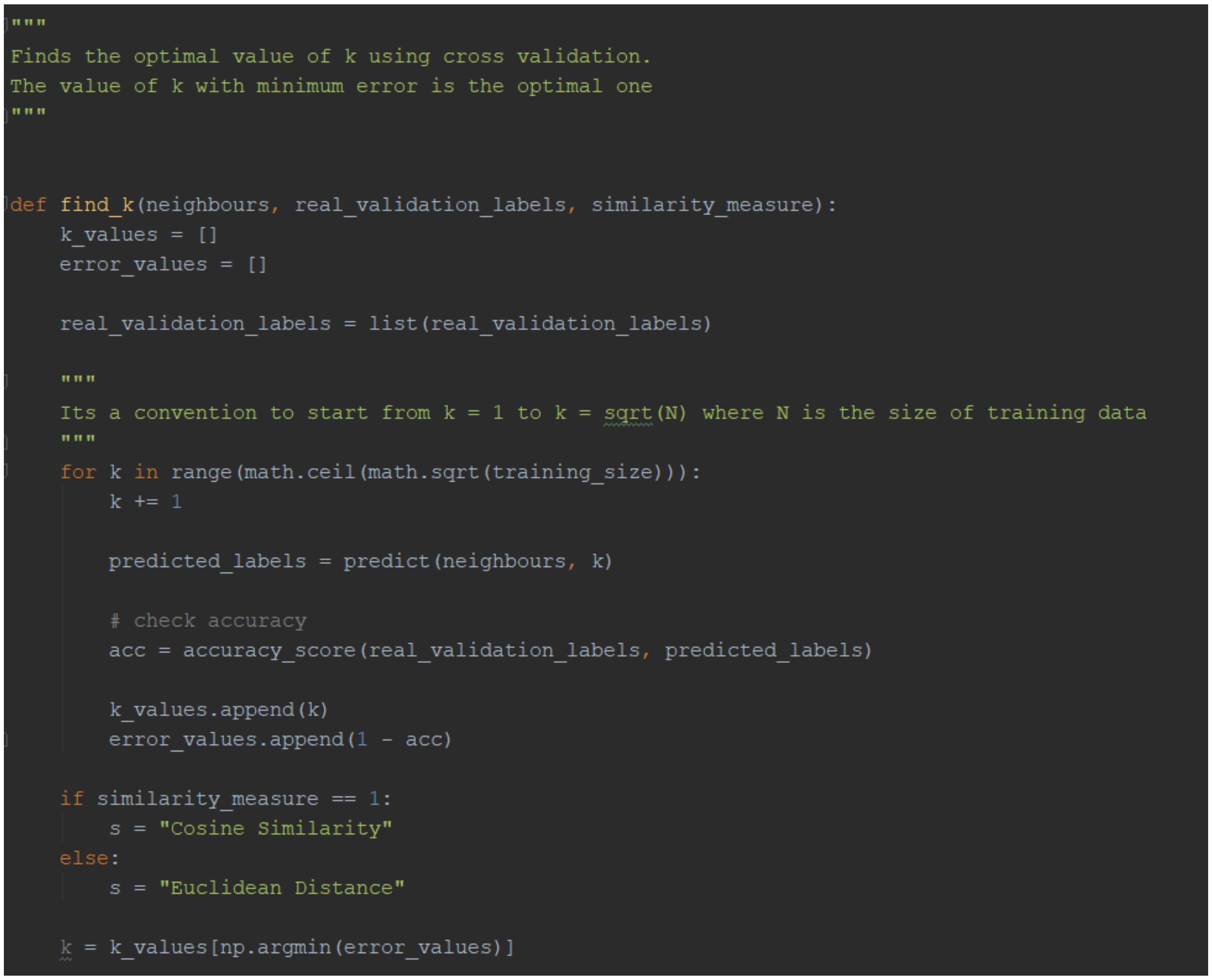
*Figure 14 – Euclidean Distance between two points p and q.*

## Optimal Value of K: (Using Elbow Method)

In order to find the optimal value of K, we splitted the training data into two parts.

- Part one is the new training data for KNN (3000 Images)
- Part two is the validation data (1000 Images) on which validation error curve would be computed.

Now in order to find that value of K where validation error is the least, we ran KNN for different values of K starting from **1 to sqrt(N)** where N is the size of the new training data.



```python
"""
Finds the optimal value of k using cross validation.
The value of k with minimum error is the optimal one
"""


def find_k(neighbours, real_validation_labels, similarity_measure):
    k_values = []
    error_values = []

    real_validation_labels = list(real_validation_labels)

    """
    Its a convention to start from k = 1 to k = sqrt(N) where N is the size of training data
    """
    for k in range(math.ceil(math.sqrt(training_size))):
        k += 1

        predicted_labels = predict(neighbours, k)

        # check accuracy
        acc = accuracy_score(real_validation_labels, predicted_labels)

        k_values.append(k)
        error_values.append(1 - acc)

    if similarity_measure == 1:
        s = "Cosine Similarity"
    else:
        s = "Euclidean Distance"

    k = k_values[np.argmin(error_values)]
```

*Figure 15 – K-NN is run for k = 1 to sqrt(N) and accuracy is computed for each k. We previously computed all the neighbors once so now only k is being changed.*

## Validation Error Curves:

We use the **elbow method** to find the optimal value of K.

Since the experiment is run twice, so we have to find the optimal value of K for KNN with cosine similarity and KNN with Euclidean Distance.

***The optimal value of K is the one where the validation error is the least.***

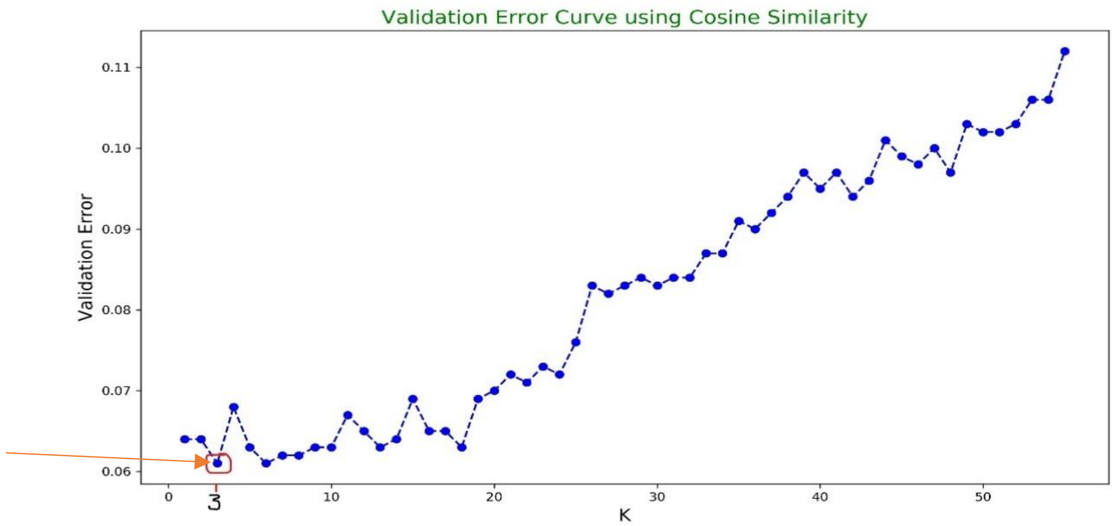


*Figure 16 – For K = 3 the validation error is the minimum, so this k is chosen for K-NN with Cosine Similarity*

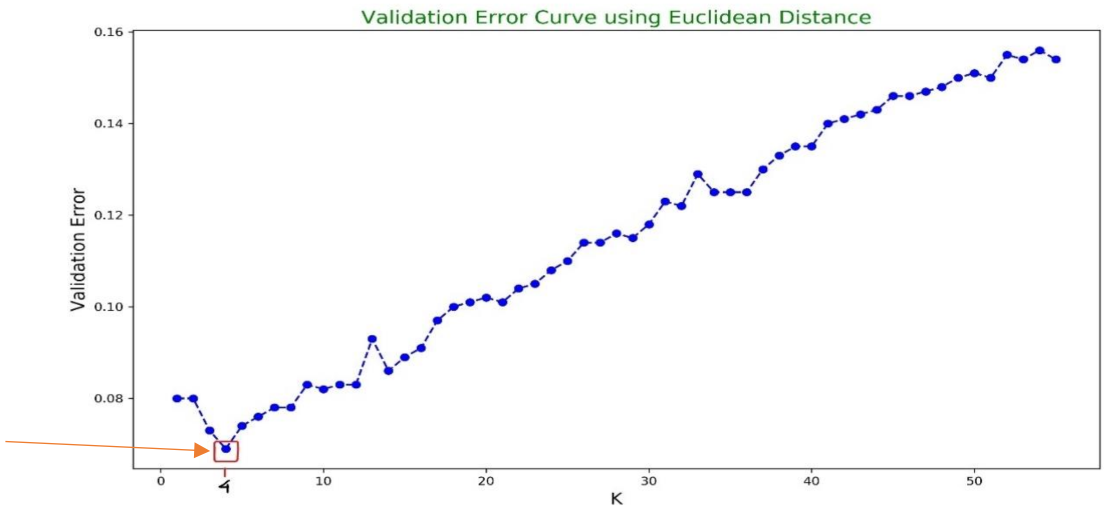


*Figure 17 – For K = 4 the validation error is the minimum, so this k is chosen for K-NN with Euclidean Distance*

## Experimental Settings for Multiclass Perceptron:

The Multiclass Perceptron experiment was run twice. Once with Perceptron with Signum Activation Function and once with Perceptron with Sigmoid Activation Function.

### Bias:

In order to make the model fit the training data, a bias of w0 is also added which breaks the restriction of the model’s decision boundary to pass through the origin.

In order to make the dot product of weights and input features consistent, a new feature is added to input vector as x0 whose value is set to 1.

### Data Split:

After running the algorithm for different splits of training and testing data, the following split is chosen since it served the purpose of better results:

| | |
|---|---|
| Training Data Used: | 5000 Data Points |
| Testing Data Used: | 10,000 Data Points |

### Learning Rate:

The learning rate should be neither too small that the convergence becomes too slow nor too large that the gradient never converges and keeps on bouncing back and forth.

After running the experiment with different values of eta (learning rate), **0.01** is chosen.

## Cost Function:

- **Sigmoid Unit:**

With sigmoid unit, the mean squared cost function cannot be used since when sigmoid activation function is plugged into it, it doesn't remain convex. In order to have a convex cost function whose gradient descent will lead to a global minimum, we use the following logarithmic cost function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

Where:

m = training size

h(x) = label predicted

y = original label

- **Threshold Unit (Rosenblatt's Perceptron):**

The Perceptron cost function is

$$\text{cost} = -\sum_{i\in\mathcal{M}} y_i h_i = \sum_{i=1}^{N} |-y_i h_i|_+$$

where $\mathcal{M} \subseteq \{1,\ldots,N\}$ is the index set of the currently misclassified patterns. $|arg|_+ = \max(arg, 0)$.

Obviously, we get $\text{cost} = 0$ only, when all patterns are correctly classified (then $\mathcal{M} = \emptyset$ ); otherwise $\text{cost} > 0$, since $y_i$ and $h_i$ have different signs for misclassified patterns

## Epochs:

One epoch is basically one complete round over all the training data.

- ## Sigmoid Unit:

The Sigmoid Unit is trained using Batch Mode of Gradient Descent. The number of iterations/epochs chosen for this mode is 2000. A plot is also plotted for one perceptron's training for different iterations and their respective costs.

It is clear that the gradient is converged when iterations reach 1500 and there is less decrease in cost after that.

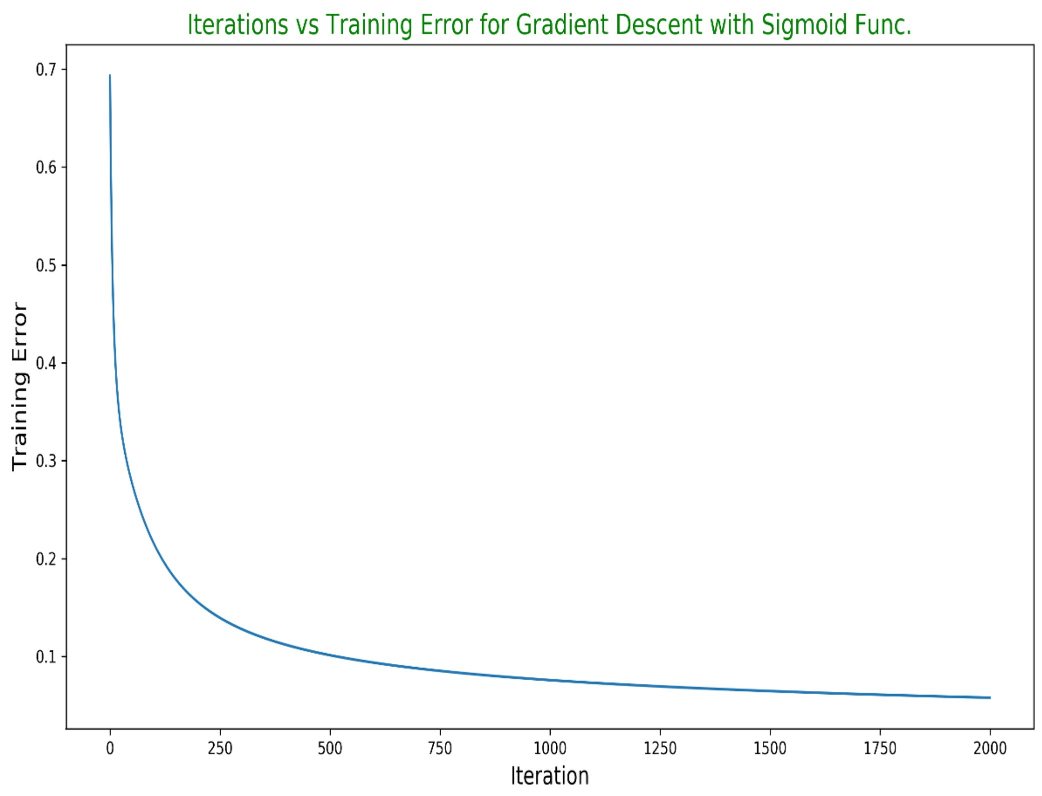


*Figure 18 – Iterations vs Training Error Curve for one Perceptron of Multiclass Classifier using Gradient Descent (Batch Mode) with Sigmoid Activation Function*

- **Threshold Unit (Rosenblatt's Perceptron):**

The Threshold Unit is trained using Stochastic Gradient Descent where instead of updating the weights after one whole epoch, the weights are updated for every training instance which makes this approach converge faster than gradient descent and less expensive cost wise.

Originally, this perceptron is made to keep updating the weights until all the training examples are correctly classified. For this to happen, the training data should be linearly separable. But in our case the data is not linearly separable so in order to avoid infinite looping we make the perceptron train for some pre-defined epochs.

In order to choose the best epochs for our scenario, we ran the experiment for different epochs and a curve is plotted as shown below.

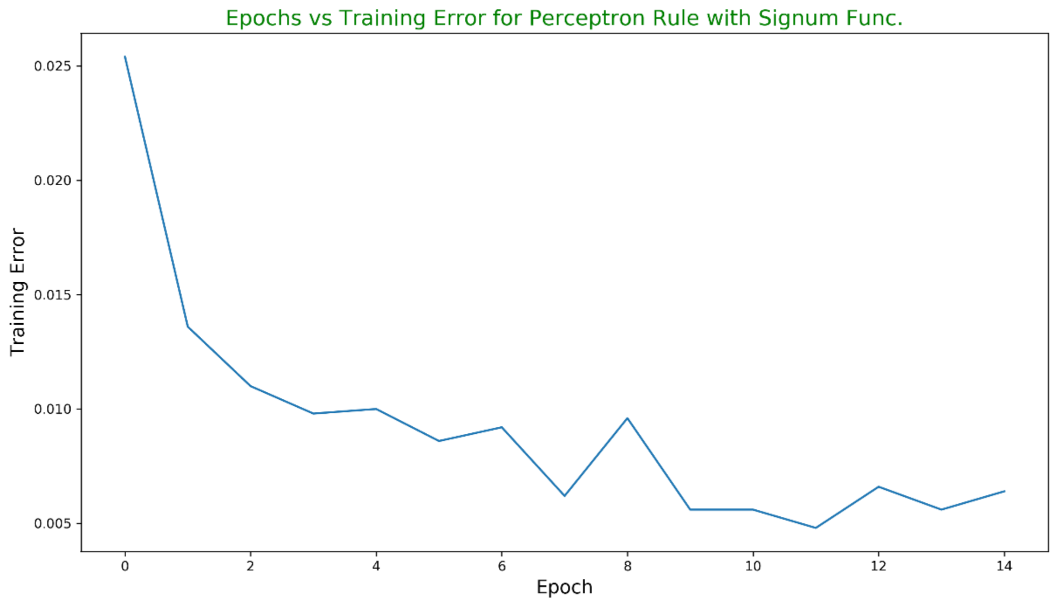


*Figure 19 – The training cost is generally decreasing as epochs are increased and is converging. This is a plot for one perceptron of the Multi Class Perceptron Classifier using Stochastic Gradient Descent with Signum Function*

Just for experimentation purposes, we made the epochs to 2000 and plotted the cost curve again. Now it is clear that between 10 – 30 epochs the cost is converged and after that there is almost no change in cost.

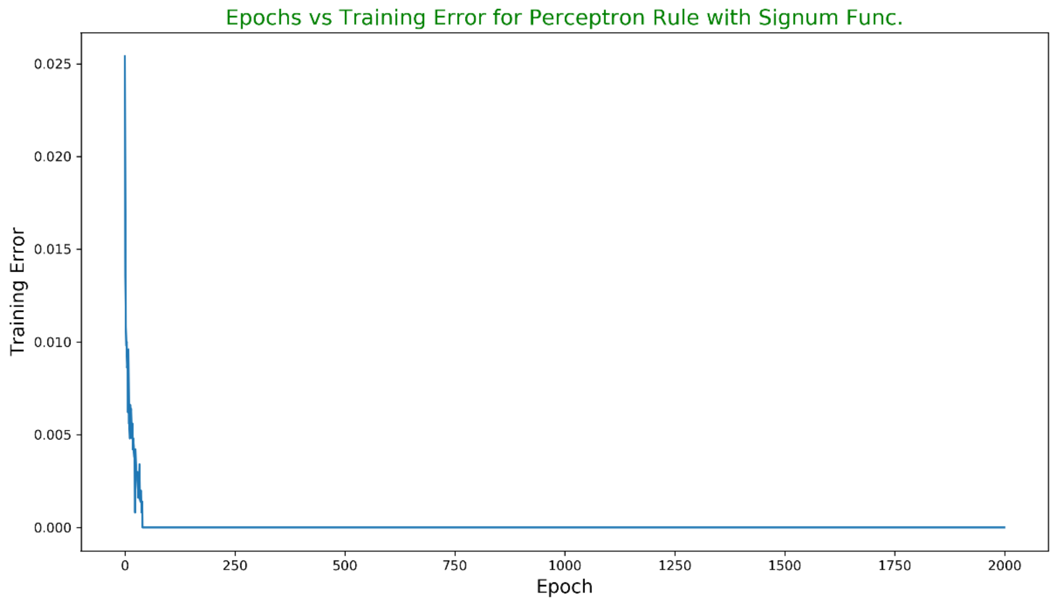


*Figure 20 - The training cost has converged between 10 – 30 epochs. After that there is no change in the training cost.*

So in the light of above experimentation, we decided to choose 15 epochs for training of Rosenblatt’s Perceptrons.

## Experimental Settings for SVM Classifier:

In K-NN and Multiclass Perceptron Classifier we trained our models on raw images directly instead of computing some features from the input image and training the model on those computed measurements/features.

A feature descriptor is a representation of an image that simplifies the image by extracting useful information and throwing away extraneous information.

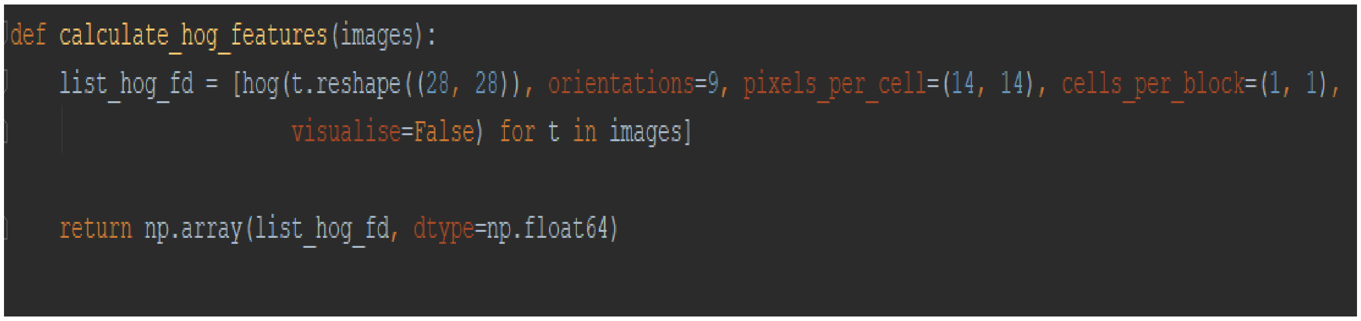
## HOG Descriptor:

Now we are going to compute the **Histogram of Oriented Gradients** as features from the digit images and we will train the SVM Classifier on that.

The HOG descriptor technique counts occurrences of gradient orientation in localized portions of an image - detection window.

## Calculation:

The HOG Features vector is calculated for all training examples and then the model is trained on these features. Also, while testing the test images, the HOG features of test images are calculated, and the model is evaluated on those features instead of raw test images.



```python
def calculate_hog_features(images):
    list_hog_fd = [hog(t.reshape((28, 28)), orientations=9, pixels_per_cell=(14, 14), cells_per_block=(1, 1),
                       visualise=False) for t in images]

    return np.array(list_hog_fd, dtype=np.float64)
```

*Figure 21 – Code Snippet for Calculating HOG Features of given images.*

To calculate the HOG features, we set:

- the number of cells in each block equal to one each individual cell is of size 14×14. Since our image is of size 28×28, we will have four blocks/cells of size 14×14 each.

- Also, we set the size of orientation vector equal to 9. So, our HOG feature vector for each sample will be of size 4×9 = 36.

## Accuracy Measure:

The accuracy measure used for each algorithm is:

Accuracy = Correctly Classified Images/Total Images * 100

# Analysis

Now the final phase. After running the experiment with different algorithms, the results are summarized. First comparing the techniques on basis of Accuracy:

## Accuracy (Performance):

| Algorithm | Training Size | Testing Size | Accuracy |
|---|---|---|---|
| K-NN with Cosine Similarity **(K = 3)** | 3000 | 1000 | 91.40% |
| K-NN with Euclidean Distance **(K=4)** | 3000 | 1000 | 89.90% |
| Multiclass Perceptron with Signum Activation Function **(15 Epochs)** | 5000 | 10000 | 81.79% |
| Multiclass Perceptron with Sigmoid Activation Function **(2000 Iterations)** | 5000 | 10000 | 86.94% |
| SVM with HOG Based Features | 5000 | 10000 | 87.25% |

When we compare the K-NN method with Multiclass Perceptron and SVM on basis of accuracy then its accuracy is similar to that of other two classifiers which means despite its simplicity K-NN is really a good classifier.

## Prediction Time (Efficiency):

### Our Observations:

One of the main limitations of K-NN was that it was **computationally expensive.** Its prediction time was large because whenever a new query instance came it had to compare its similarity with all the training data and then sort the neighbors according to their confidence and then separating the top k neighbors and choosing the label of the most occurred neighbor in top k. In all this process, it takes a comparable amount of time.

While for Multiclass Perceptron Classifier we observed it will mitigate this limitation in efficiency such that its prediction time will be short because now it will only compute the dot product in the prediction phase. The majority of time is spent only once in its learning phase. Then it's ready to predict the test instances.

### Results:

| **Algorithm** | **Learning Time** | **Prediction Time (Efficiency)** |
|---|---|---|
| K-NN with Cosine Similarity **(K = 3)** | - | 63.94 seconds |
| K-NN with Euclidean Distance **(K=4)** | - | 42.78 seconds |
| Multiclass Perceptron with Signum Activation Function **(15 Epochs)** | 20.01 seconds | 0.09 seconds |
| Multiclass Perceptron with Sigmoid Activation Function **(2000 Iterations)** | 149.71 seconds | 0.09 seconds |
| SVM with HOG Based Features | 2.79 seconds | 5.04 seconds |

## Conclusion:

When the times were calculated for the prediction phases of K-NN, Multiclass Perceptron and SVM, the Multiclass Perceptron clearly stands out with the shortest prediction time while on the other side, K-NN took a large time in predicting the test instances.

Hence Multiclass Perceptron clearly leaves K-NN behind in terms of efficiency in Prediction Time and also in terms of computation and memory load. Thus, it mitigates the limitations of our baseline method K-NN.