# 1. INTRODUCTION

In this paper we are going to discuss about handwritten number identification to gain insight about handwritings. R is used to build the model for MBA due to its efficiency in handling large datasets.

## 1.1 Motivation

We have a variety of handwriting. Each and every person has his/her own style of writing. But at the time when others try to identify/recognize, it becomes difficult. So we are making computer learn to recognize the handwritings for analysis purpose.

## 1.2 System Analysis

### 1.2.1 Existing System

There are a vast number of applications which recognize the numbers on the basis of area of contact on the touch screen of the device. For example, we have Google Handwriting Input and Notepads which recognize the text when we write on the screen. Instead, we can consider the images of numbers which are written in papers and then study them

### 1.2.2 Proposed System

In proposed system, we are using neural networks as platform for studying of the handwritten numbers. The images of the numbers are converted into machine understandable format (.csv file) for comparison and analysis

# 2. MACHINE LEARNING

Machine learning (ML) is a category of algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available.

The processes involved in machine learning are similar to that of data mining and predictive modeling. Both require searching through data to look for patterns and adjusting program actions accordingly. Many people are familiar with machine learning from shopping on the internet and being served ads related to their purchase. This happens because recommendation engines use machine learning to personalize online ad delivery in almost real time. Beyond personalized marketing, other common machine learning use cases include fraud detection, spam filtering, network security threat detection, predictive maintenance and building news feeds.

*How machine learning works*?

Machine learning algorithms are often categorized as supervised or unsupervised. Supervised algorithms require a data scientist or data analyst with machine learning skills to provide both input and desired output, in addition to furnishing feedback about the accuracy of predictions during algorithm training. Data scientists determine which variables, or features, the model should analyze and use to develop predictions. Once training is complete, the algorithm will apply what was learned to new data.

Unsupervised algorithms do not need to be trained with desired outcome data. Instead, they use an iterative approach called deep learning to review data and arrive at conclusions. Unsupervised learning algorithms -- also called neural networks -- are used for more complex processing tasks than supervised learning systems, including image recognition, speech-to-text and natural language generation. These neural networks work by combing through millions of examples of training data and automatically identifying often subtle correlations between many variables. Once trained, the algorithm can use its bank of associations to interpret new data. These algorithms have only become feasible in the age of big data, as they require massive amounts of training data.

## 2.1 Machine Learning in R

R is one of the major languages for data science. It provides excellent visualization features, which is essential to explore the data before submitting it to any automated learning, as well as assessing the results of the learning algorithm. Many R package for machine learning are available of the shelf and many modern methods in statistical learning are implemented in R as part of their development.

Real datasets often come with missing values. In R, these should be encoded using `NA`. There are basically two approaches to deal with such cases.

- Drop the observations with missing values, or, if one feature contains a very high proportion of NAs, drop the feature altogether. These approaches are only applicable when the proportion of missing values is relatively small. Otherwise, it could lead to loosing too much data.

- Impute missing values.

Data imputation can however have critical consequences depending on the proportion of missing values and their nature. From a statistical point of view, missing values are classified as *missing completely at random* (MCAR), *missing at random* (MAR) or *missing not at random* (MNAR), and the type of the missing values will influence the efficiency of the imputation method.

## 2.2 Common Machine Learning Algorithms

Just as there are nearly limitless uses of machine learning, there is no shortage of machine learning algorithms. They range from the fairly simple to the highly complex. Here are a few of the most commonly used models:

- This class of machine learning algorithm involves identifying a correlation -- generally between two variables -- and using that correlation to make predictions about future data points.

- **Decision trees.** These models use observations about certain actions and identify an optimal path for arriving at a desired outcome.

- **K-means clustering.** This model groups a specified number of data points into a specific number of groupings based on like characteristics.

- **Neural networks.** These deep learning models utilize large amounts of training data to identify correlations between many variables to learn to process incoming data in the future.

- **Reinforcement learning.** This area of deep learning involves models iterating over many attempts to complete a process. Steps that produce favorable outcomes are rewarded and steps that produce undesired outcomes are penalized until the algorithm learns the optimal process.

## 2.3 Neural Networks

A neural network is a system of hardware and/or software patterned after the operation of neurons in the human brain. Neural networks -- also called artificial neural networks

are a variety of deep learning technology, which also falls under the umbrella of artificial intelligence, or AI.

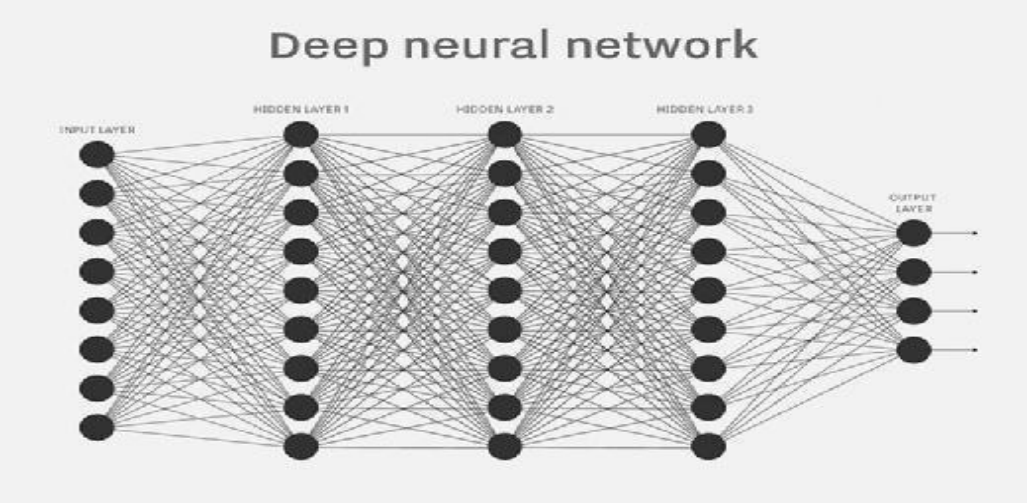
Commercial applications of these technologies generally focus on solving complex signal processing or pattern recognition problems. Examples of significant commercial applications since 2000 include handwriting recognition for check processing, speech-to-text transcription, oil-exploration data analysis, weather prediction and facial recognition.

***How artificial neural networks work***

A neural network usually involves a large number of processors operating in parallel and arranged in tiers. The first tier receives the raw input information -- analogous to optic nerves in human visual processing. Each successive tier receives the output from the tier preceding it, rather than from the raw input -- in the same way neurons further from the optic nerve receive signals from those closer to it. The last tier produces the output of the system.

Each processing node has its own small sphere of knowledge, including what it has seen and any rules it was originally programmed with or developed for itself. The tiers are highly interconnected, which means each node in tier *n* will be connected to many nodes in tier *n-1* -- its inputs -- and in tier *n+1*, which provides input for those nodes. There may be one or multiple nodes in the output layer, from which the answer it produces can be read.

Neural networks are notable for being adaptive, which means they modify themselves as they learn from initial training and subsequent runs provide more information about the world. The most basic learning model is centered on weighting the input streams, which is how each node weights the importance of input from each of its predecessors. Inputs that contribute to getting right answers are weighted higher.




***How neural networks learn***

Typically, a neural network is initially trained or fed large amounts of data. Training consists of providing input and telling the network what the output should be. For example, to build a network to identify the faces of actors, initial training might be a series of pictures of actors, non-actors, masks, statuary, animal faces and so on. Each input is accompanied by the matching identification, such as actors' names, "not actor" or "not human" information. Providing the answers allows the model to adjust its internal weightings to learn how to do its job better

## 2.4 Decision Tree

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions. Though a commonly used tool in data mining for deriving a strategy to reach a particular goal, its also widely used in machine learning, which will be the main focus of this article.

***How can an algorithm be represented as a tree?***

For this let’s consider a very basic example that uses titanic data set for predicting whether a passenger will survive or not. Below model uses 3 features/attributes/columns from the data set, namely sex, age and sibsp (number of spouses or children along).

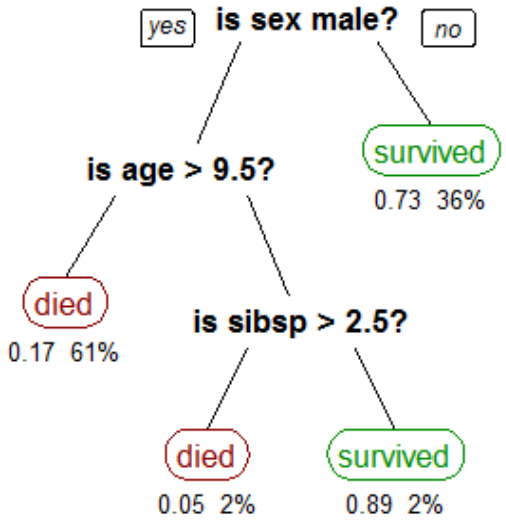



Image taken from wikipedia

*A decision tree is drawn upside down with its root at the top.* In the image on the left, the bold text in black represents a condition/internal node, based on which the tree splits into branches/ edges. The end of the branch that doesn't split anymore is the decision/leaf, in this case, whether the passenger died or survived, represented as red and green text respectively.

Although, a real dataset will have a lot more features and this will just be a branch in a much bigger tree, but you can't ignore the simplicity of this algorithm. The feature importance is clear and relations can be viewed easily. This methodology is more commonly known as learning decision tree from data and above tree is called Classification tree as the target is to classify passenger as survived or died. Regression trees are represented in the same manner, just they predict continuous values like price of a house. In general, Decision Tree algorithms are referred to as CART or Classification and Regression Trees.

So, what is actually going on in the background? Growing a tree involves deciding on which features to choose and what conditions to use for splitting, along with knowing when to stop. As a tree generally grows arbitrarily, you will need to trim it down for it to look beautiful. Lets start with a common technique used for splitting.

***Recursive Binary Splitting***

*In this procedure all the features are considered and different split points are tried and tested using a cost function. The split with the best cost (or lowest cost) is selected.*

Consider the earlier example of tree learned from titanic dataset. In the first split or the root, all attributes/features are considered and the training data is divided into groups based on this split. We have 3 features, so will have 3 candidate splits. Now we will *calculate how much* accuracy *each split will cost us, using a function. The split that costs least is chosen*, which in our example is sex of the passenger. This *algorithm is recursive in nature* as the groups formed can be sub-divided using same strategy. Due to this procedure, this algorithm is also known as the greedy algorithm, as we have an excessive desire of lowering the cost. This makes the root node as best predictor/classifier.

***Cost of a split***

Lets take a closer look at cost functions used for classification and regression. In both cases the cost functions try to find most homogeneous branches, or branches having groups with similar responses. This makes sense we can be more sure that a test data input will follow a certain path.

*Regression : sum(y—prediction)²*

Lets say, we are predicting the price of houses. Now the decision tree will start splitting by considering each feature in training data. The mean of responses of the training data inputs of particular group is considered as prediction for that group. The above function is applied to all data points and cost is calculated for all candidate splits. *Again the split with lowest cost is chosen.* Another cost function involves reduction of standard deviation.

*Classification : G = sum(pk * (1—pk))*

A Gini score gives an idea of how good a split is by how mixed the response classes are in the groups created by the split. Here, pk is proportion of same class inputs present in a particular group. A perfect class purity occurs when a group contains all inputs from the same class, in which case pk is either 1 or 0 and G = 0, where as a node having a 50–50 split of classes in a group has the worst purity, so for a binary classification it will have pk = 0.5 and G = 0.5.

***When to stop splitting?***

You might ask *when to stop growing a tree?* As a problem usually has a large set of features, it results in large number of split, which in turn gives a huge tree. Such trees are *complex and can lead to overfitting.* So, we need to know when to stop? One way of doing this is to set a minimum number of training inputs to use on each leaf. For example we can use a minimum of 10 passengers to reach a decision(died or survived), and ignore any leaf that takes less than 10 passengers. Another way is to set maximum depth of your model. Maximum depth refers to the the length of the longest path from a root to a leaf.

***Pruning***

The performance of a tree can be further increased by *pruning*. *It involves removing the branches that make use of features having low importance*. This way, we reduce the complexity of tree, and thus increasing its predictive power by reducing overfitting.

Pruning can start at either root or the leaves. The simplest method of pruning starts at leaves and removes each node with most popular class in that leaf, this change is kept if it doesn't deteriorate accuracy. Its also called reduced error pruning. More sophisticated pruning methods can be used such as cost complexity pruning where a learning parameter (alpha) is used to weigh whether nodes can be removed based on the size of the sub-tree. This is also known as weakest link pruning.

Advantages of CART

- Simple to understand, interpret, visualize.
- Decision trees *implicitly perform variable screening or feature selection.*
- Can *handle both numerical and categorical data*. Can also *handle multi-output problems.*
- Decision trees require relatively *little effort from users for data preparation.*

- *Nonlinear relationships between parameters do not affect tree performance.*

Disadvantages of CART

- Decision-tree learners *can create over-complex trees* that do not generalize the data well. This is called *overfitting*.
- Decision trees can be unstable because *small variations in the data might result in a completely different tree being generated.* This is called variance which needs to be *lowered by methods like bagging and boosting.*
- Greedy algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees, where the features and samples are randomly sampled with replacement.
- Decision tree learners create biased if some classes dominate. It is therefore recommended to balance the data set prior to fitting with the decision tree.

This is all the basic, to get you at par with decision tree learning. An improvement over decision tree learning is made using technique of boosting. A popular library for implementing these algorithms is Scikit-learn. It has a wonderful api that can get your model up an running with just a few lines of code in python.

## 2.5 Random Forest Algorithm

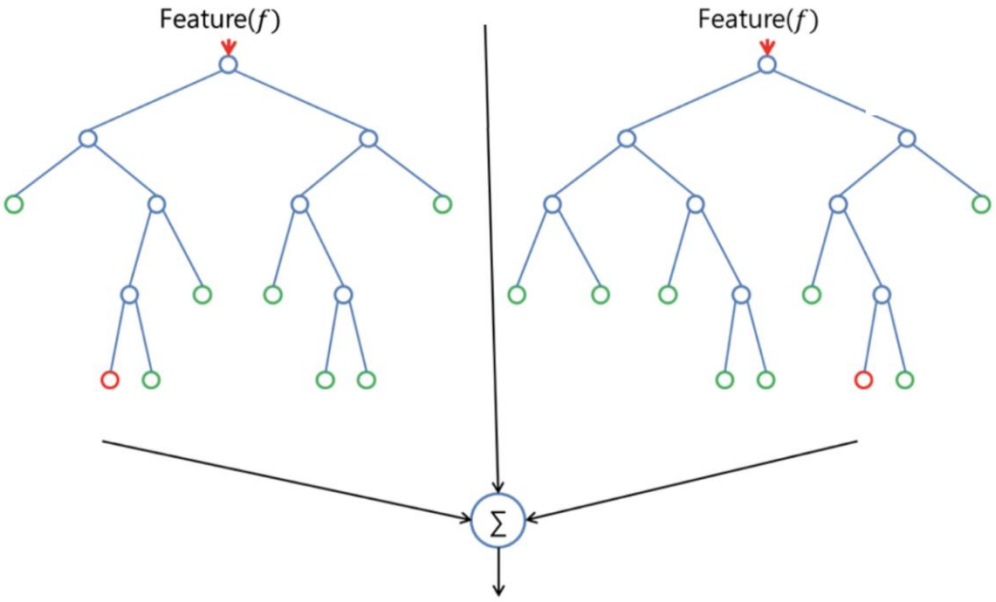
Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because it's simplicity and the fact that it can be used for both classification and regression tasks.

***How it works:***

Random Forest is a supervised learning algorithm. Like you can already see from it's name, it creates a forest and makes it somehow random. The „forest" it builds, is an ensemble of Decision Trees, most of the time trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result.

To say it in simple words: Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.
One big advantage of random forest is, that it can be used for both classification and

regression problems, which form the majority of current machine learning systems. I will talk about random forest in classification, since classification is sometimes considered the building block of machine learning. Below you can see how a random forest would look like with two trees:




Random Forest has nearly the same hyperparameters as a decision tree or a bagging classifier. Fortunately, you don't have to combine a decision tree with a bagging classifier and can just easily use the classifier-class of Random Forest. Like I already said, with Random Forest, you can also deal with Regression tasks by using the Random Forest regressor.

Random Forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

Therefore, in Random Forest, only a random subset of the features is taken into consideration by the algorithm for splitting a node. You can even make trees more random, by additionally using random thresholds for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

***Real Life Analogy:***

Imagine a guy named Andrew, that want's to decide, to which places he should travel during a one-year vacation trip. He asks people who know him for advice. First, he goes to a friend, tha asks Andrew where he traveled to in the past and if he liked it or not. Based on the answers, he will give Andrew some advice.

This is a typical decision tree algorithm approach. Andrews friend created rules to guide his decision about what he should recommend, by using the answers of Andrew.

Afterwards, Andrew starts asking more and more of his friends to advise him and they again ask him different questions, where they can derive some recommendations from. Then he chooses the places that where recommend the most to him, which is the typical Random Forest algorithm approach.

***Feature Importance:***

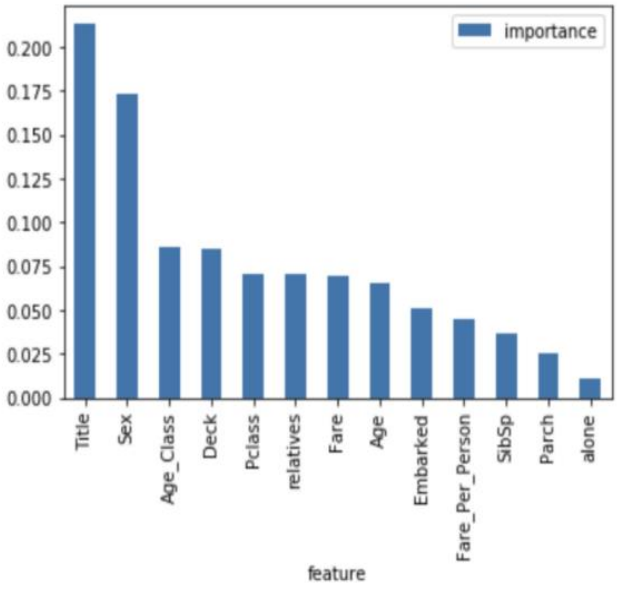
Another great quality of the random forest algorithm is that it is very easy to measure the relative importance of each feature on the prediction. Sklearn provides a great tool for this, that measures a features importance by looking at how much the tree nodes, which use that feature, reduce impurity across all trees in the forest. It computes this score automatically for each feature after training and scales the results, so that the sum of all importance is equal to 1.

If you don't know how a decision tree works and if you don't know what a leaf or node is, here is a good description from Wikipedia: In a decision tree each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). A node that has no children is a leaf.

Through looking at the feature importance, you can decide which features you may want to drop, because they don't contribute enough or nothing to the prediction process. This is important, because a general rule in machine learning is that the more features you have, the more likely your model will suffer from overfitting and vice versa.

Below you can see a table and a visualization that show the importance of 13 features,

| feature | importance |
|---|---|
| Title | 0.213 |
| Sex | 0.173 |
| Age_Class | 0.086 |
| Deck | 0.085 |
| Pclass | 0.071 |
| relatives | 0.070 |
| Fare | 0.069 |
| Age | 0.065 |
| Embarked | 0.051 |
| Fare_Per_Person | 0.045 |
| SibSp | 0.037 |
| Parch | 0.025 |
| alone | 0.011 |




***Difference between Decision Trees and Random Forests:***

Like I already mentioned, Random Forest is a collection of Decision Trees, but there are some differences.

If you input a training dataset with features and labels into a decision tree, it will formulate some set of rules, which will be used to make the predictions.

For example, if you want to predict whether a person will click on an online advertisement, you could collect the ad's the person clicked in the past and some features that describe his decision. If you put the features and labels into a decision tree, it will generate some rules. Then you can predict whether the advertisement will be clicked or not. In comparison, the Random Forest algorithm randomly selects observations and features to build several decision trees and then averages the results.

Another difference is that „deep" decision trees might suffer from overfitting. Random

Forest prevents overfitting most of the time, by creating random subsets of the features and building smaller trees using these subsets. Afterwards, it combines the subtrees. Note that this doesn't work every time and that it also makes the computation slower, depending on how many trees your random forest builds.

***Important Hyperparameters:***

The parameters in random forest are either used to increase the predictive power of the model or to make the model faster. I will here talk about the hyperparameters of sklearns built-in random forest function.

1. Increasing the Predictive Power

Firstly, there is the „n_estimators" hyperparameter, which is just the number of trees the algorithm builds before taking the maximum voting or taking averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation.

Another important hyperparameter is „max_features", which is the maximum number of features Random Forest is allowed to try in an individual tree. Sklearn provides several options, described in their documentation.

The last important hyper-parameter we will talk about in terms of speed, is „min_sample_leaf ". This determines, like its name already says, the minimum number of leafs that are required to split an internal node.

2. Increasing the Models Speed

The „n_jobs" hyperparameter tells the engine how many processors it is allowed to use. If it has a value of 1, it can only use one processor. A value of "-1" means that there is no limit.

„random_state" makes the model's output replicable. The model will always produce the same results when it has a definite value of random_state and if it has been given the same parameters and the same training data.

Lastly, there is the „oob_score" (also called oob sampling), which is a random forest cross validation method. In this sampling, about one-third of the data is not used to train the model and can be used to evaluate its performance. These samples are called the out of bag samples. It is very similar to the leave-one-out cross-validation method, but almost no additional computational burden goes along with it.

***Advantages and Disadvantages:***

An advantage of random forest is that it can be used for both regression and classification tasks and that it's easy to view the relative importance it assigns to the input features.

Random Forest is also considered as a very handy and easy to use algorithm, because it's default hyperparameters often produce a good prediction result. The number of hyperparameters is also not that high and they are straightforward to understand.

One of the big problems in machine learning is overfitting, but most of the time this won't happen that easy to a random forest classifier. That's because if there are enough trees in the forest, the classifier won't overfit the model.

The main limitation of Random Forest is that a large number of trees can make the algorithm to slow and ineffective for real-time predictions. In general, these algorithms are fast to train, but quite slow to create predictions once they are trained. A more accurate prediction requires more trees, which results in a slower model. In most real-world applications the random forest algorithm is fast enough, but there can certainly be situations where run-time performance is important and other approaches would be preferred.

And of course Random Forest is a predictive modeling tool and not a descriptive tool. That means, if you are looking for a description of the relationships in your data, other approaches would be preferred.

***Use Cases:***

The random forest algorithm is used in a lot of different fields, like Banking, Stock Market, Medicine and E-Commerce. In Banking it is used for example to detect customers who will use the bank's services more frequently than others and repay their debt in time. In this domain it is also used to detect fraud customers who want to scam the bank. In finance, it is used to determine a stock's behaviour in the future. In the healthcare domain it is used to identify the correct combination of components in medicine and to analyze a patient's medical history to identify diseases. And lastly, in E-commerce random forest is used to determine whether a customer will actually like the product or not.

***Summary:***

Random Forest is a great algorithm to train early in the model development process, to see how it performs and it's hard to build a "bad" Random Forest, because of its simplicity. This algorithm is also a great choice, if you need to develop a model in a short period of time. On top of that, it provides a pretty good indicator of the importance it assigns to your features.

Random Forests are also very hard to beat in terms of performance. Of course you can probably always find a model that can perform better, like a neural network, but these usually take much more time in the development. And on top of that, they can handle a lot of different feature types, like binary, categorical and numerical.

Overall, Random Forest is a (mostly) fast, simple and flexible tool, although it has its limitations.

## 3. MNIST Database

The **MNIST database** (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset. There have been a number of scientific papers on attempts to achieve the lowest error rate; one paper, using a hierarchical system of convolution neural networks, manages to get an error rate on the MNIST database of 0.23 percent. The original creators of the database keep a list of some of the methods tested on it.[5] In their original paper, they use a support vector machine to get an error rate of 0.8 percent. An extended dataset similar to MNIST called EMNIST has been published in 2017, which contains 240,000 training images, and 40,000 testing images of handwritten digits and characters.

Sample Image of MNIST Dataset:

### 3.1 Confusion Matrix

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. The confusion matrix itself is relatively simple to understand, but the related terminology can be confusing.

Let's start with an example confusion matrix for a binary classifier (though it can easily be extended to the case of more than two classes):

| n=165 | **Predicted: NO** | **Predicted: YES** |
|---|---|---|
| **Actual: NO** | 50 | 10 |
| **Actual: YES** | 5 | 100 |

What can we learn from this matrix?

- There are two possible predicted classes: "yes" and "no". If we were predicting the presence of a disease, for example, "yes" would mean they have the disease, and "no" would mean they don't have the disease.
- The classifier made a total of 165 predictions (e.g., 165 patients were being tested for the presence of that disease).
- Out of those 165 cases, the classifier predicted "yes" 110 times, and "no" 55 times.
- In reality, 105 patients in the sample have the disease, and 60 patients do not.

Let's now define the most basic terms, which are whole numbers (not rates):

- true positives (TP): These are cases in which we predicted yes (they have the disease), and they do have the disease.
- true negatives (TN): We predicted no, and they don't have the disease.
- false positives (FP): We predicted yes, but they don't actually have the disease. (Also known as a "Type I error.")
- false negatives (FN): We predicted no, but they actually do have the disease. (Also known as a "Type II error.")

| n=165 | Predicted: NO | Predicted: YES | |
|---|---|---|---|
| **Actual: NO** | TN = 50 | FP = 10 | 60 |
| **Actual: YES** | FN = 5 | TP = 100 | 105 |
| | 55 | 110 | |

This is a list of rates that are often computed from a confusion matrix for a binary classifier:

- Accuracy: Overall, how often is the classifier correct?
    - (TP+TN)/total = (100+50)/165 = 0.91
- Misclassification Rate: Overall, how often is it wrong?
    - (FP+FN)/total = (10+5)/165 = 0.09
    - equivalent to 1 minus Accuracy
    - also known as "Error Rate"
- True Positive Rate: When it's actually yes, how often does it predict yes?
    - TP/actual yes = 100/105 = 0.95
    - also known as "Sensitivity" or "Recall"
- False Positive Rate: When it's actually no, how often does it predict yes?
    - FP/actual no = 10/60 = 0.17
- Specificity: When it's actually no, how often does it predict no?
    - TN/actual no = 50/60 = 0.83
    - equivalent to 1 minus False Positive Rate
- Precision: When it predicts yes, how often is it correct?
    - TP/predicted yes = 100/110 = 0.91
- Prevalence: How often does the yes condition actually occur in our sample?
    - actual yes/total = 105/165 = 0.64

A couple other terms are also worth mentioning:

- Positive Predictive Value: This is very similar to precision, except that it takes prevalence into account. In the case where the classes are perfectly balanced

(meaning the prevalence is 50%), the positive predictive value (PPV) is equivalent to precision. (More details about PPV.)

- Null Error Rate: This is how often you would be wrong if you always predicted the majority class. (In our example, the null error rate would be 60/165=0.36 because if you always predicted yes, you would only be wrong for the 60 "no" cases.) This can be a useful baseline metric to compare your classifier against. However, the best classifier for a particular application will sometimes have a higher error rate than the null error rate, as demonstrated by the Accuracy Paradox.
- Cohen's Kappa: This is essentially a measure of how well the classifier performed as compared to how well it would have performed simply by chance. In other words, a model will have a high Kappa score if there is a big difference between the accuracy and the null error rate. (More details about Cohen's Kappa.)
- F Score: This is a weighted average of the true positive rate (recall) and precision. (More details about the F Score.)
- ROC Curve: This is a commonly used graph that summarizes the performance of a classifier over all possible thresholds. It is generated by plotting the True Positive Rate (y-axis) against the False Positive Rate (x-axis) as you vary the threshold for assigning observations to a given class. (More details about ROC Curves.)

And finally, for those of you from the world of Bayesian statistics, here's a quick summary of these terms from Applied Predictive Modeling:

# 4. PHYSICAL DESIGN

## 4.1 UML Diagrams

The Unified Modelling Language allows the software engineer to express an analysis model using the modelling notation that is governed by a set of syntactic semantic and pragmatic rules.

### 4.1.1 Usecase Diagram

Use case diagrams are usually referred to as behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system

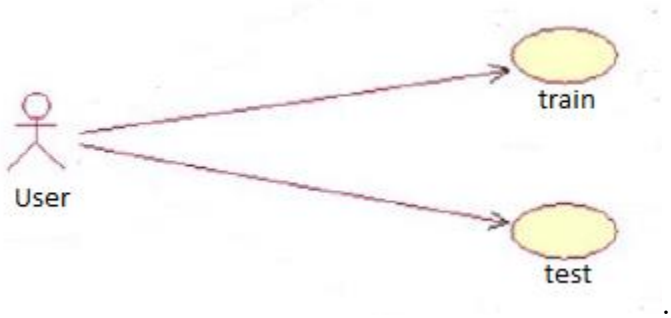



**Fig 4.1 Use Case Diagram**

### 4.1.2 Sequence Diagram

A sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.

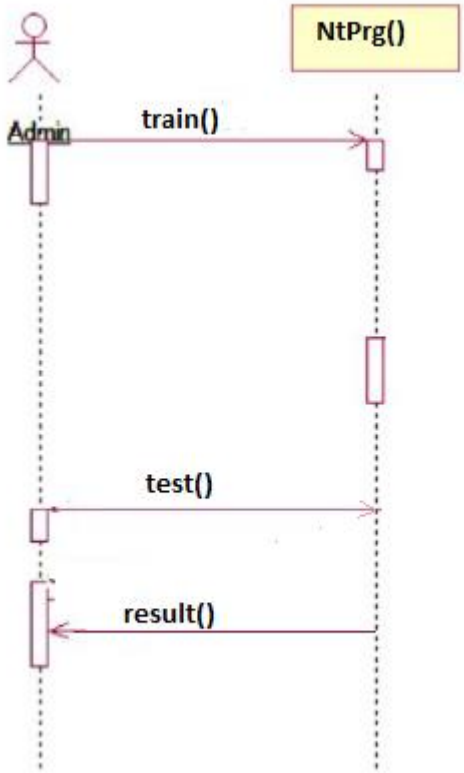



**Fig 4.1.2 Sequence Diagram**

# 5. IMPLEMENTATION

## 5.1 R/RStudio Installation

There are two programs to download and install in order to implement the ANN analysis-
a. R
b. R-studio

**R:**

It is a powerful statistical programming language that also free and open source. These software packages can be downloaded from http://www.r-project.org/ and http://rstudio.org/ respectively and are available on the Windows, Linux and Mac OS X platforms.

**R-Studio:**

It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, and debugging and workspace management.

### 5.1.1 Prerequisites of R Studio

Any version of R (3.3.3 or higher)
In order to run R and R-studio on system, we need to follow the following three steps order wise.

a. Install R
b. Install R-Studio
c. Install R-Packages (If needed)

#### 5.1.1.1 Installation of R on Windows System

1. Download the latest precompiled binary distributions from CRAN website [http://www.rproject.org/]

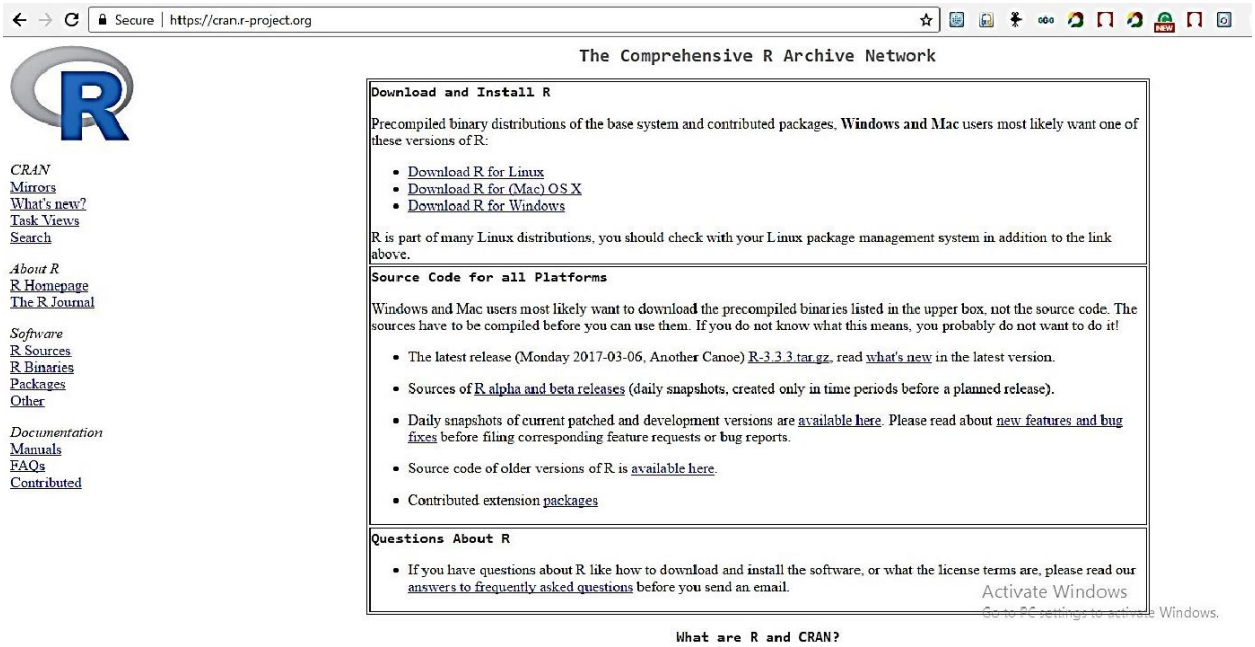
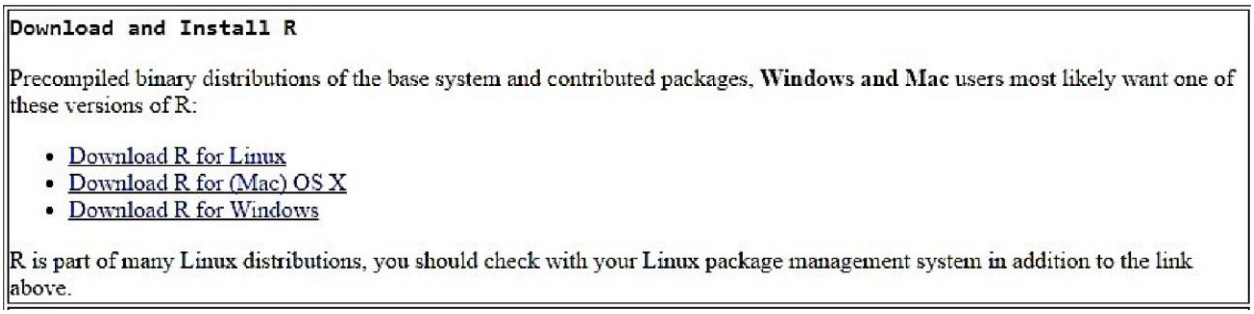



Figure 5.1: Site for R installation

2. Click on download R for windows as we are working on windows operation system



**Download and Install R**

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Figure 5.2: Download R for different operating systems

3. Then click on downloads R 3.3.3 for windows

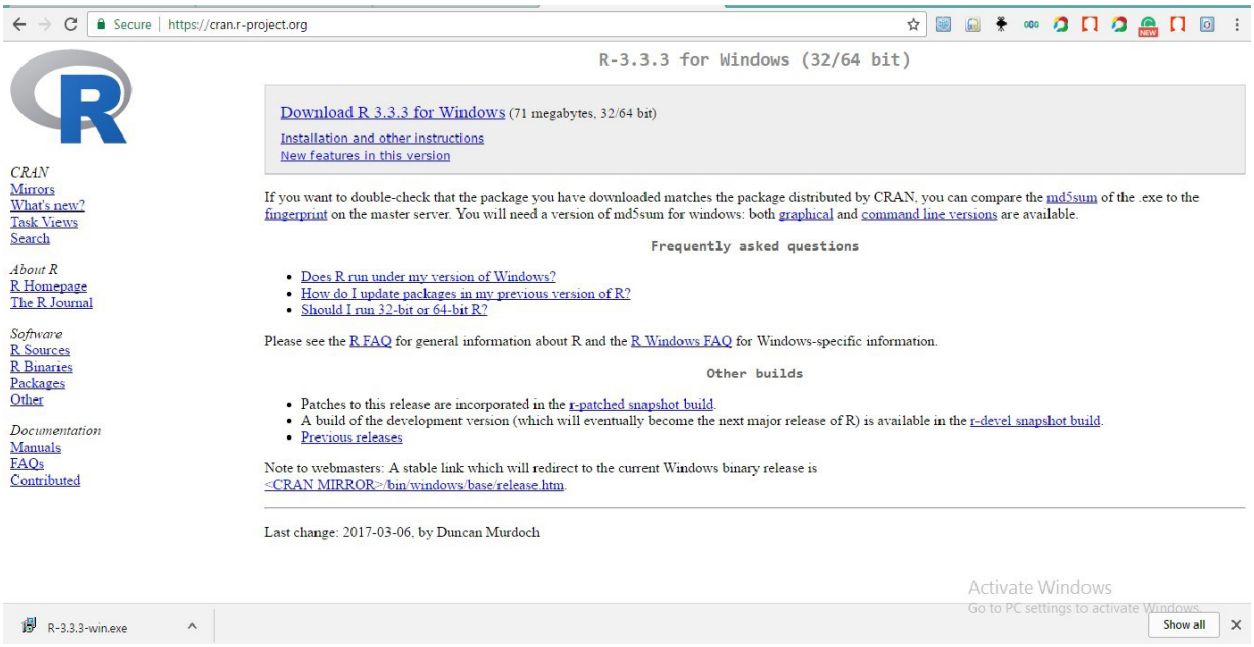



Figure 5.3: Download R

4. Follow the following instructions to complete the installation of R

5. Select the language


Figure 5.4: Language Setup

6. Click on next to continue

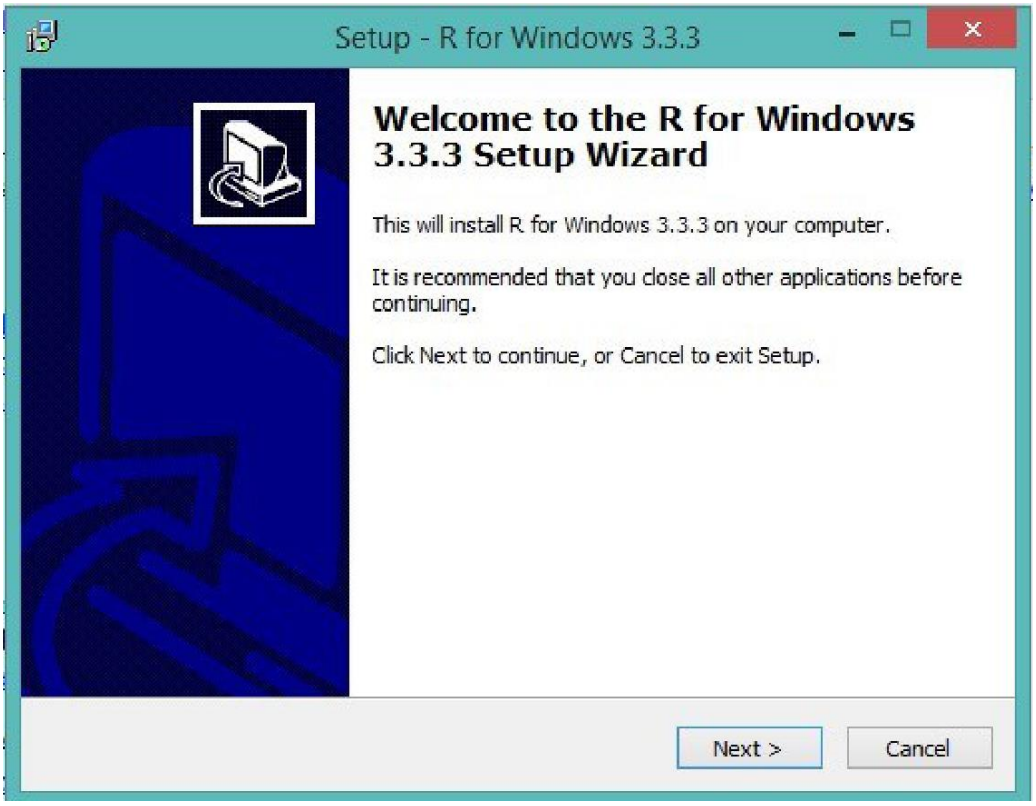



Figure 5.5: Setup Wizard

7. Click on next to continue the process

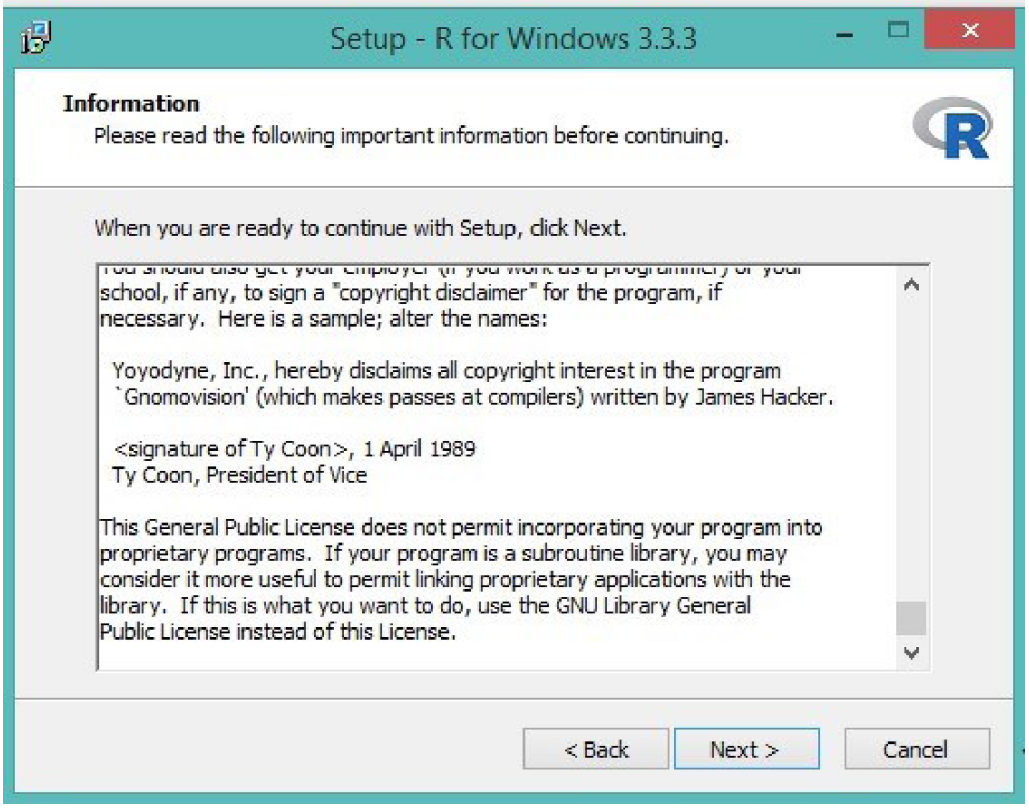



Figure 5.6: Setup

8. Choose the directory for R to be installed

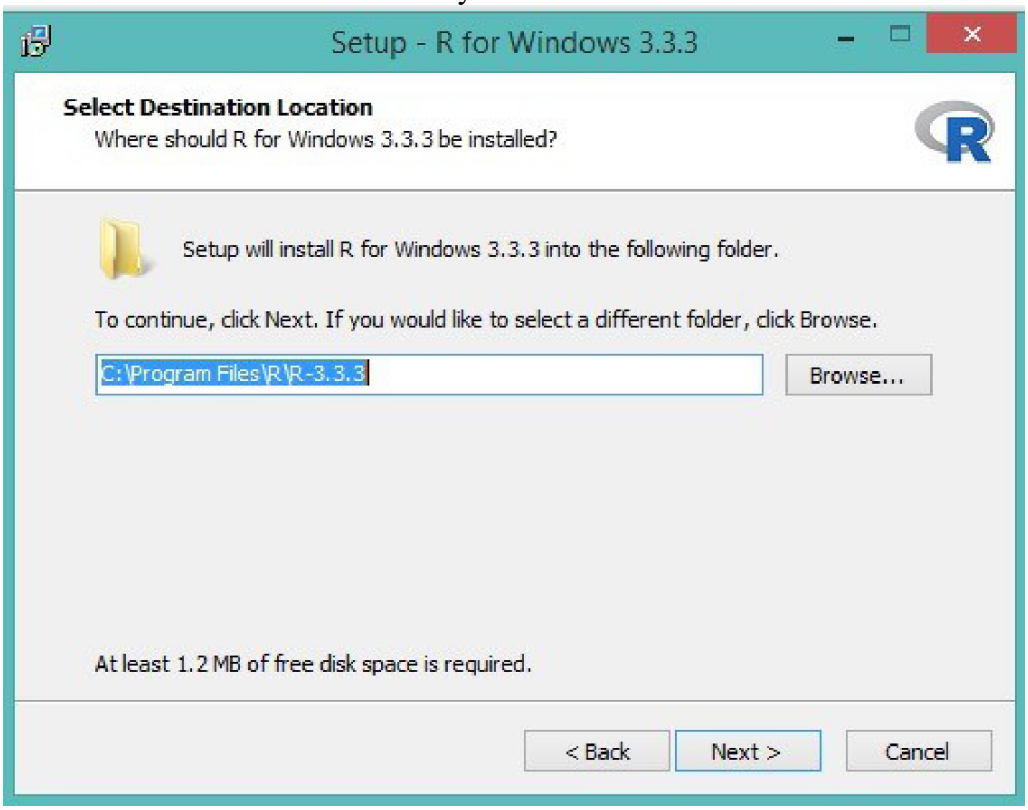



Figure 5.7: Choosing directory

9. Select the components to be installed

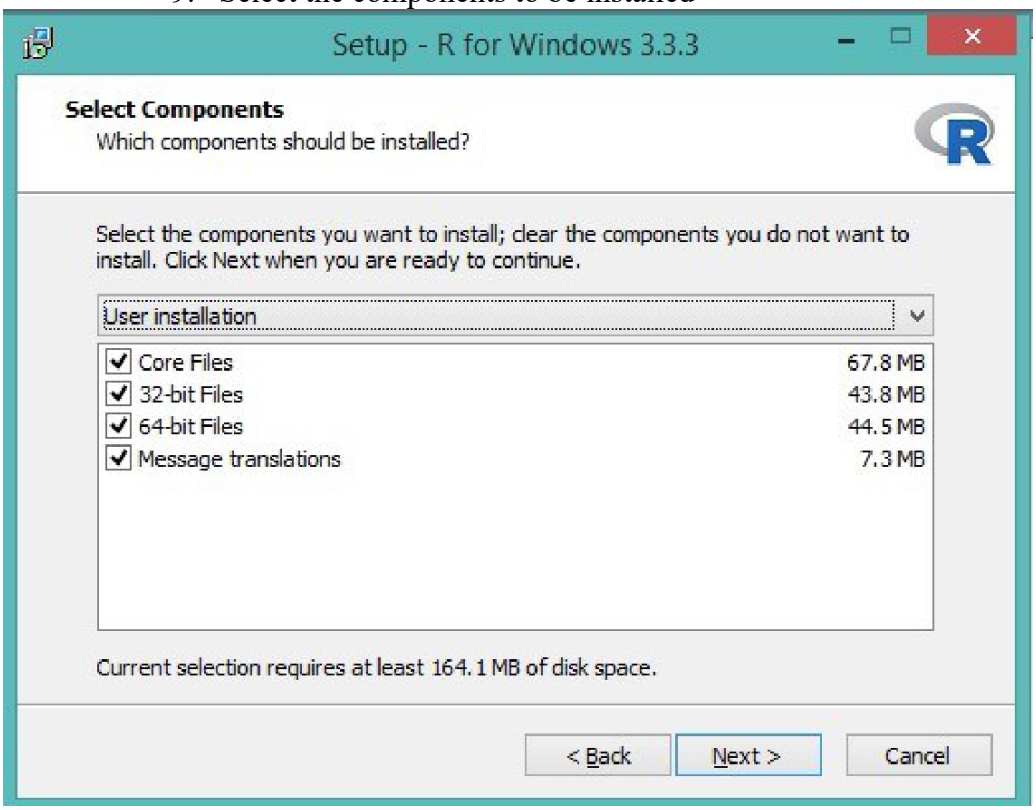


Figure 5.8: Components Selection for installation

10. Select the startup option by default NO and then click on next

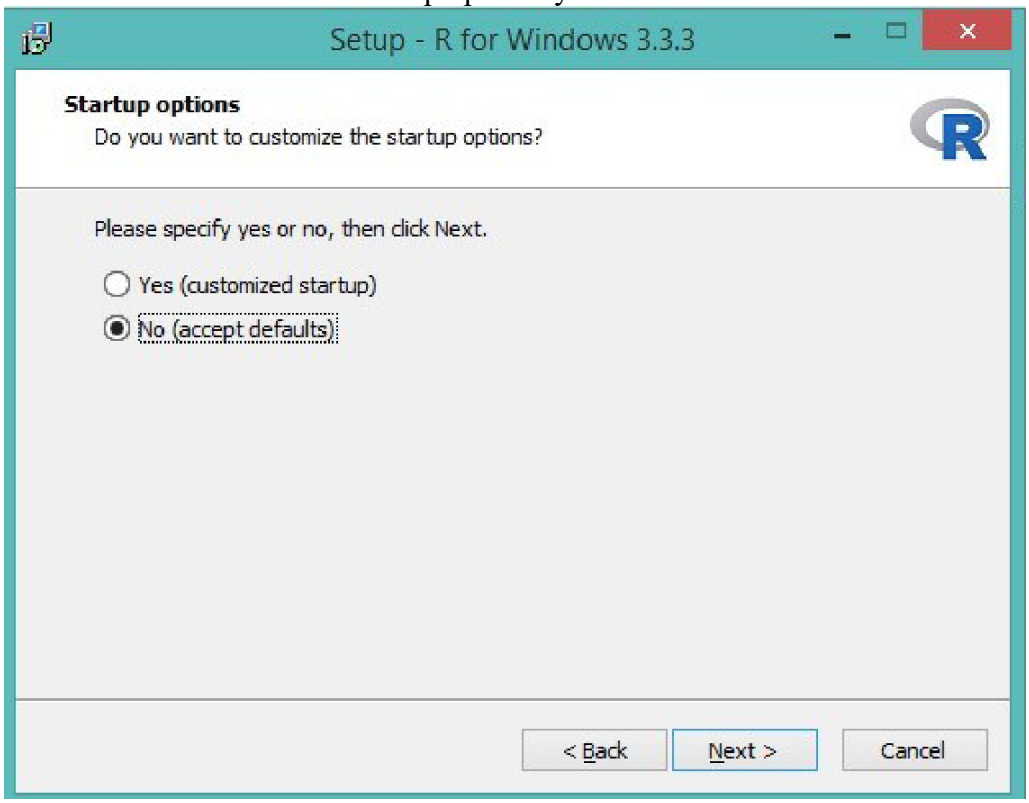



Figure 5.9: Startup Options

11. R is the folder where the program shortcuts created

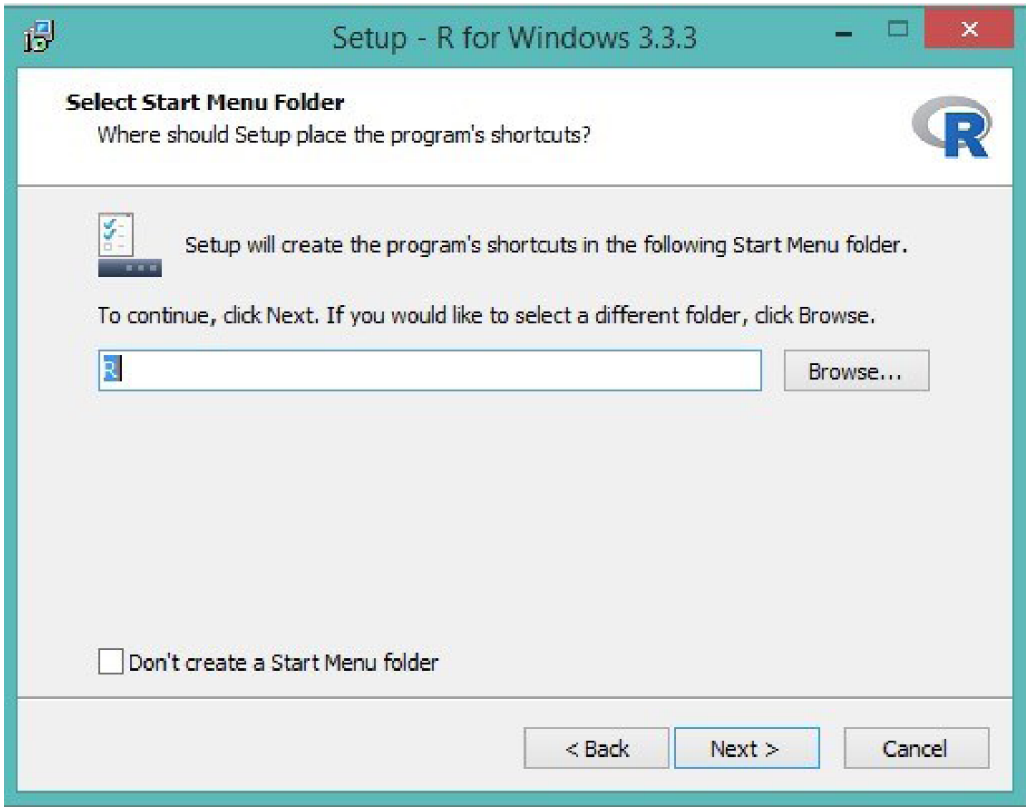



Fig 5.10: Start menu folder selection

12. Select additional icons if necessary and then the installation process gets complete

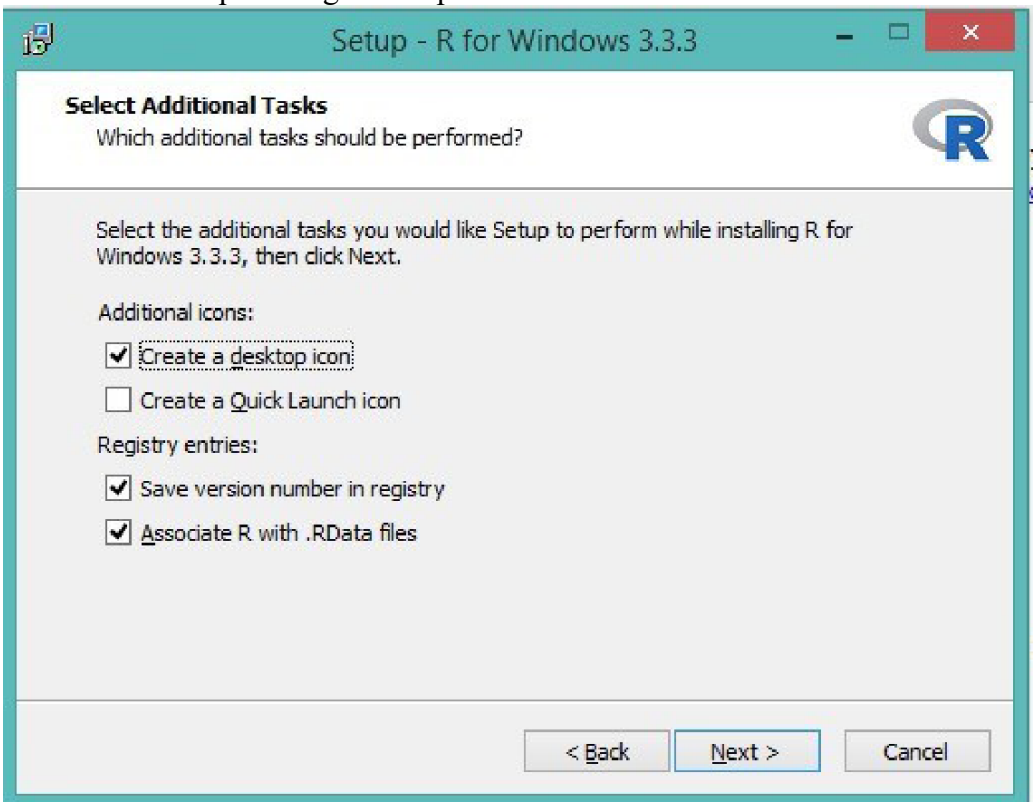


Figure 5.11: Creating desktop icon

Once completed, launch R-GUI from the shortcut. Or you can locate RGui.exe rom your installation path. The default path for Windows is "C:\Program Files\R\R-3.3.3\bin\x64\Rgui.exe"

Type help.Start() at the R-Console prompt and press Enter. If you can see the help server page then you have successfully installed and configured your R package.

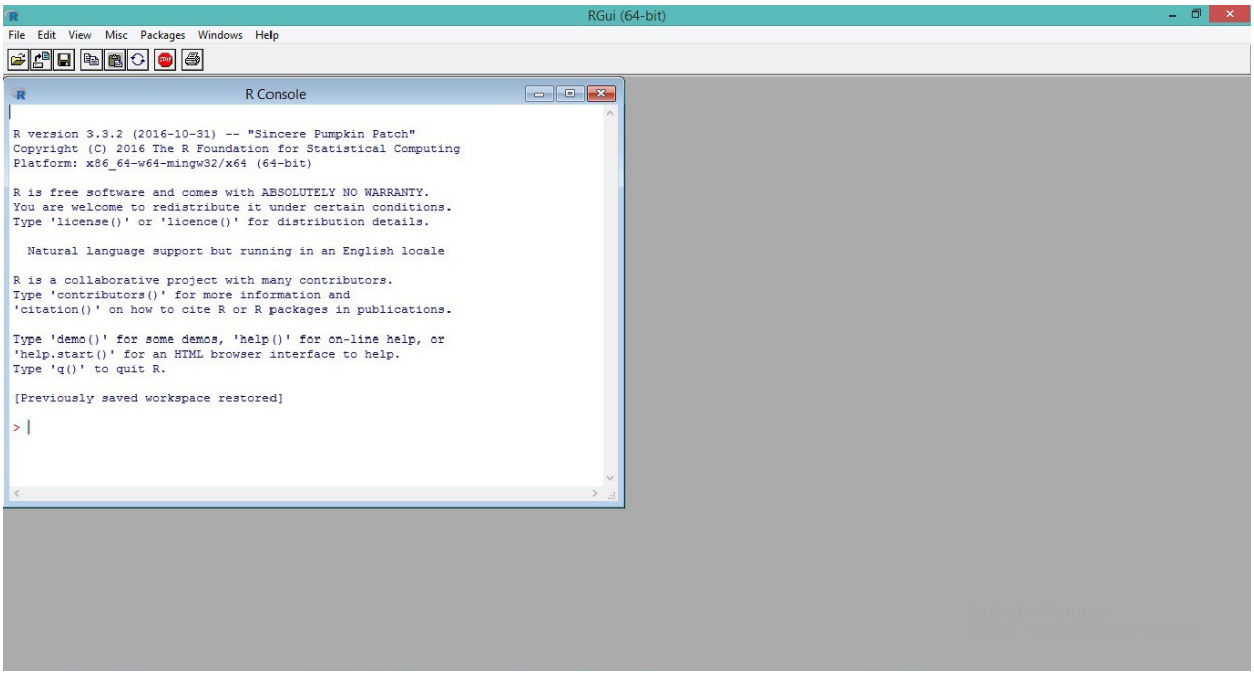



Figure 5.12: R console

### 5.1.1.2 Installation of RStudio on Windows System

Download the latest version of R-Studio for your Windows platform from "http://rstudio.org/download/desktop" (At the time of writing the latest available version of R-Studio is v0.96)

Start the installation and follow the steps required by the Setup Wizard
Once completed, launch R-Studio from Start -> All Programs-> RStudio -> Rstudio.exe or from your custom installation directory. The default installation directory for R-Studio is "C:\Program Files\R-Studio\bin\rstudio.exe"

Type help.start() at the R-Studio prompt and press Enter. If you can see the following screen then you have successfully installed and configured R-Studio to run with R.

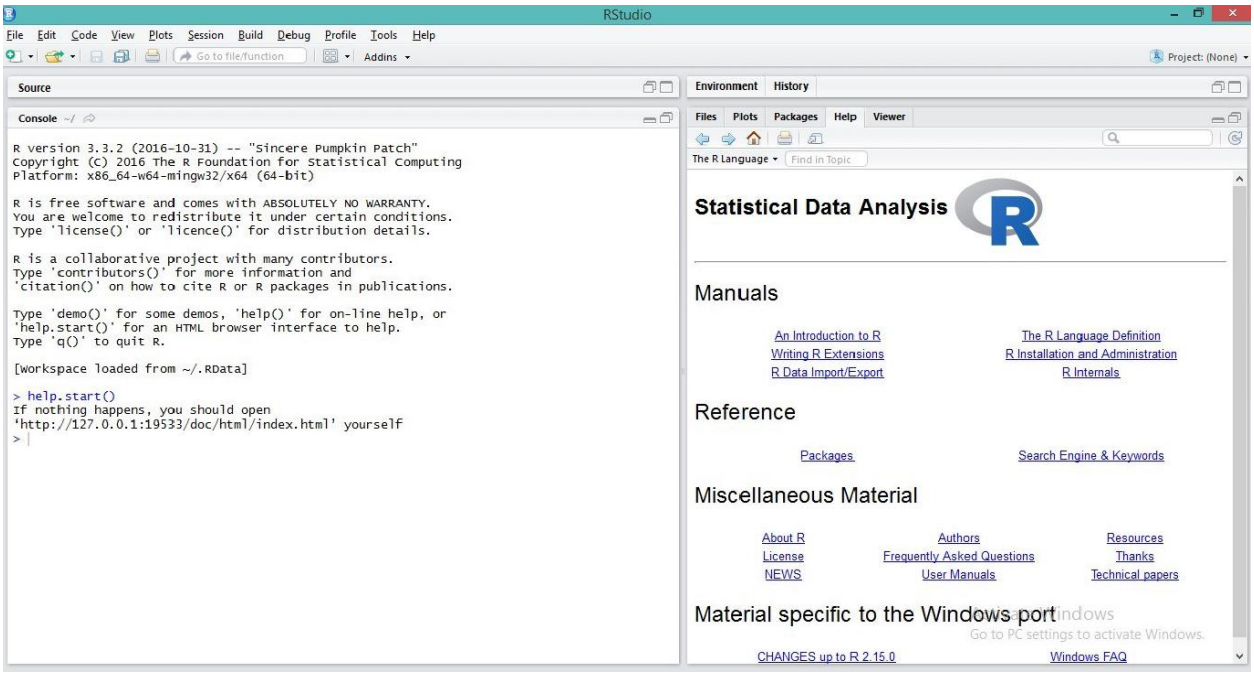



Figure 5.13: R-studio console

Make sure that this file exists in your extracted directory and make sure you have installed the required packages/library.
Set your working directory to your R-scripts using the command, “setwd(dir)”
To run a script, open the script in R-Studio's script editor and choose "Source"

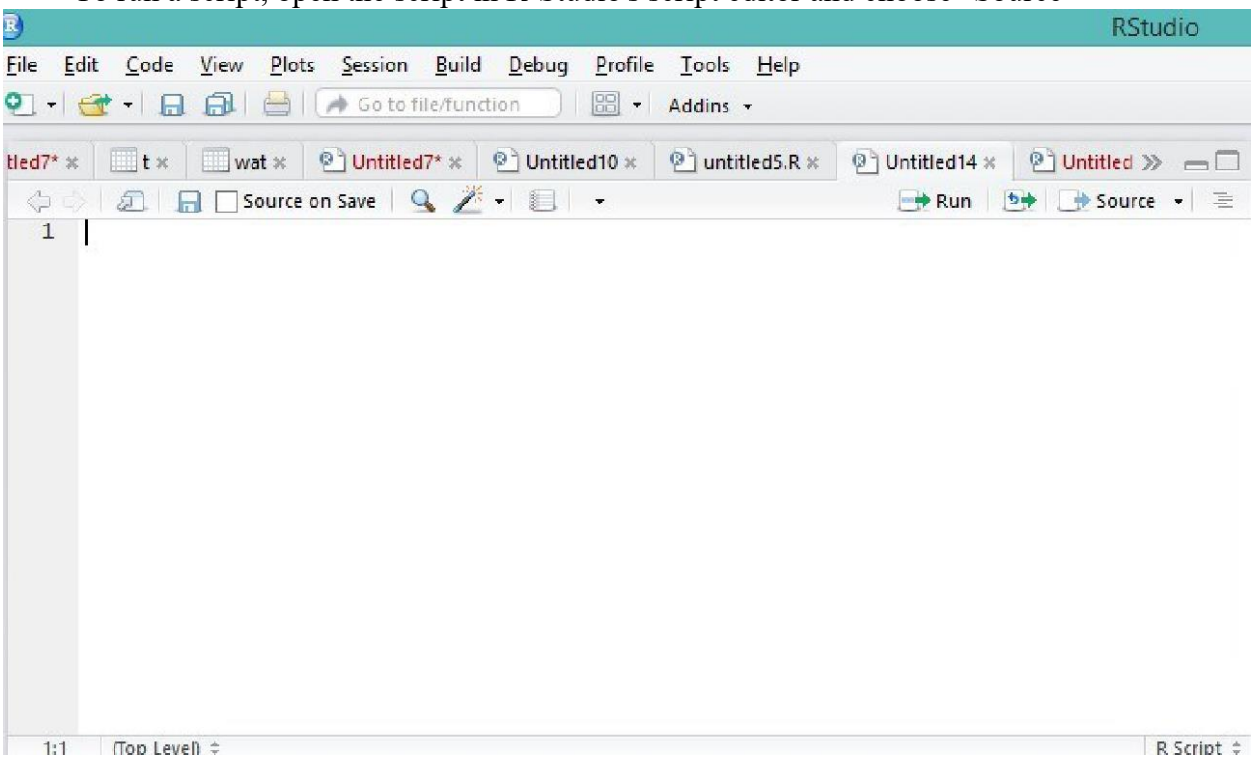



Figure 5.14: R-studio script

### 5.1.1.3 Additional Packages Installation

Packages additionally required for processing of MNIST database and processing of images (image comparison) for the handwritten dataset are

a. caret
b. e1071

Steps for installing Additional packages:

1. Go to “Tools” is Menu bar and Click-“Install Packages” in R Studio

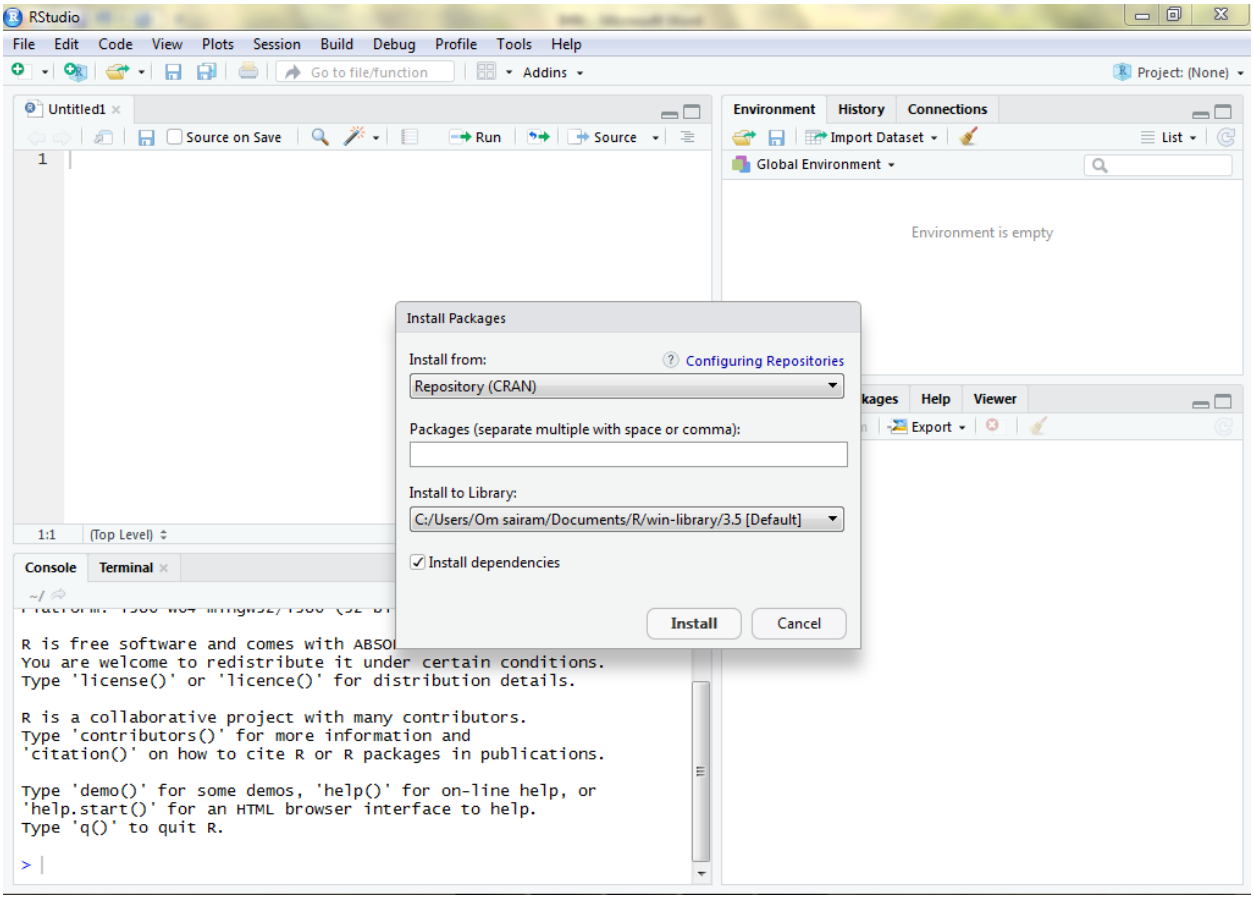



Figure 5.15: Installation of new Package in R Studio

2. Enter Name of the package which you want to update / install and then click-“Install”
3. Required Packages will be installed automatically.

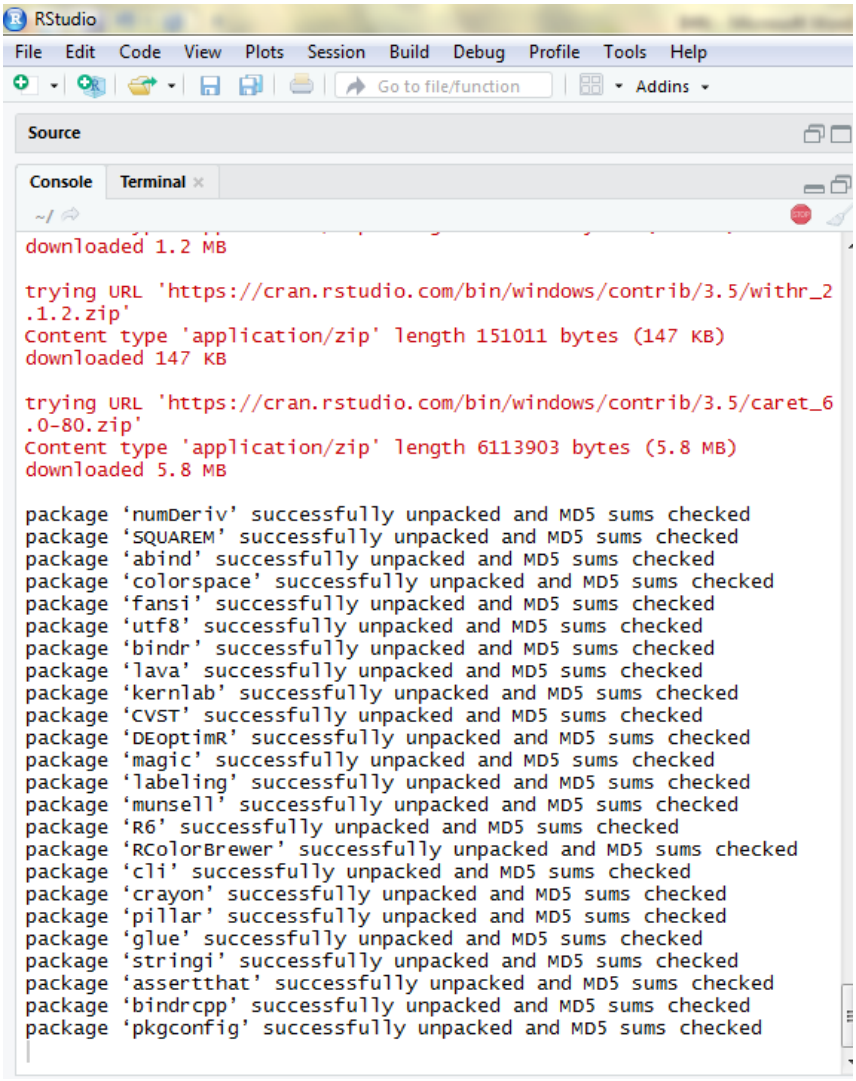



Figure 5.16: 'caret' Package Installation

# 6. ANALYSIS

## 6.1 Sample Code

```
# Read train and test data
train <- read.csv("train.csv")
test <- read.csv("test.csv")

# Check if the data is uniformly distributed over all
labels
label.freq = table(train$label)
#barplot(label.freq)

# Check if NAs exist:
which(is.na(train))
# No NAs present

# convert lable to a categorical variable
train$label <- as.factor(train$label)

# separate pixels from the training and testing sets
train_pixel <- train[,2:ncol(train)]
test_pixel <- test[,2:ncol(test)]


## Excluding highly correlated variables
# Obtaining Correlation matrix
MatCor <- cor(train[sapply(train,is.numeric)])

# The standard deviation is zero, hence correlation matrix
cannot be used

# Use nearZeroVar to preprocess the pixels
library(caret)
```

```
#zeroVar <-
as.data.frame(colnames(train_pixel[nearZeroVar(train_pixel
)]))

train_nzv <- nearZeroVar(train_pixel)
test_nzv <-  nearZeroVar(test_pixel)
# nearZeroVar selects 532 out of 784 factors/pixels

# create a new dataframe with these 532 factors.
# This dataframe will be used for further preprocessing

train_postnzv <- train_pixel[,-train_nzv]
test_postnzv <- train_pixel[,-train_nzv]

# Preprocess using PCA
train_preProcValues <- preProcess(train_postnzv,
method=c("pca"))
test_preProcValues <- preProcess(test_postnzv, method =
c("pca"))

x_trainTransformed <- predict(train_preProcValues,
train_postnzv)
x_testTransformed <- predict(test_preProcValues,
test_postnzv)


dim(x_trainTransformed)
dim(x_testTransformed)
```

```
### Building Models

# 1. Multiclass logistic regression
train_final <- x_trainTransformed
train_final$label <- train$label

dim(train_final)

LR_Model <- glm(label~.,
                family=binomial(link='logit'),
data=train_final)



View(LR_Model$y)
summary(LR_Model$y)
LR_Fitted <- predict(LR_Model, newdata= x_testTransformed,
type = "response")
View(LR_Fitted)

table <- table(train_final$label, LR_Fitted >0.5)



LR_confusionMatrix <- confusionMatrix(LR_Fitted,
train_final$label)
LR_confusionMatrix
str(train_final$label)
str(LR_Fitted)
```

```
View(LR_Fitted)
LR_Fitted <- as.data.frame(LR_Fitted)

#Error in confusionMatrix.default(LR_Fitted,
train_final$label) :
#the data cannot have more levels than the reference

table(factor(LR_Fitted,
levels=min(x_testTransformed):max(x_testTransformed)),
      factor(x_testTransformed,
levels=min(x_testTransformed):max(x_testTransformed)))



LR_accuracy <-
as.numeric(LR_confusionMatrix$overall["Accuracy"])




# Random forests:
library(randomForest)

samplerows <- sample(1:nrow(train_pixel), nrow(train)*0.6,
replace=FALSE)
train_rf <- x_trainTransformed[samplerows,]
test_rf <- x_trainTransformed[-samplerows,]

train_labels <- as.factor(train[samplerows,]$label)
test_labels<- as.factor(train[-samplerows]$labels)
```

```
RF_Model <- randomForest(train_rf, train_labels, ntree =
100)
predict_labels <- predict(RF_Model, test_rf)

accuracy <-
sum(diag(RF_Model$confusion))*100/sum(RF_Model$confusion)
accuracy
```

# 7. RESULTS

## 7.1 Screen Shots

Following are the images containing the information of execution of the R code:

```
> # Check if NAs exist:
> which(is.na(train))
integer(0)
>
```

Fig 7.1.1: No Missing values

```
> dim(x_testTransformed)
[1] 42000    91
>
```

Fig 7.1.2: Dimensions of Test data (rows, columns)

```
> dim(x_trainTransformed)
[1] 42000    91
>
```

Fig 7.1.3: Dimensions of Train Data (rows, columns)

| Data | |
|---|---|
| LR_Fitted | 42000 obs. of 1 variable |
| LR_Model | Large glm (30 elements, 109.1 Mb) |
| MatCor | Large matrix (614656 elements, 4.8 Mb) |
| RF_Model | Large randomForest (18 elements, 25.9 Mb) |
| test | 28000 obs. of 784 variables |
| test_pixel | 28000 obs. of 783 variables |
| test_postnzv | 42000 obs. of 252 variables |
| test_preProcValues | List of 21 |
| test_rf | 16800 obs. of 91 variables |
| train | 42000 obs. of 785 variables |
| train_final | 42000 obs. of 92 variables |
| train_pixel | 42000 obs. of 784 variables |
| train_postnzv | 42000 obs. of 252 variables |
| train_preProcValues | List of 21 |
| train_rf | 25200 obs. of 91 variables |
| x_testTransformed | 42000 obs. of 91 variables |
| x_trainTransformed | 42000 obs. of 91 variables |
| **Values** | |
| accuracy | 92.7195126087261 |
| label.freq | 'table' int [1:10(1d)] 4132 4684 4177 4351 4072 3795 4137 4401 40… |
| predict_labels | Large factor (16800 elements, 1.1 Mb) |
| samplerows | int [1:25200] 41832 33093 41721 41748 35347 5675 21004 15126 1093… |
| table | 'table' int [1:10, 1:2] 3893 0 28 17 4 39 43 18 12 14 ... |
| test_labels | Factor w/ 0 levels: |
| test_nzv | int [1:535] 1 2 3 4 5 6 7 8 9 10 ... |
| train_labels | Factor w/ 10 levels "0","1","2","3",..: 2 6 3 1 8 9 10 8 8 2 ... |
| train_nzv | int [1:532] 1 2 3 4 5 6 7 8 9 10 ... |

Fig 7.1.4: Variables and Data Types used in Code

```
> accuracy
[1] 92.71951
```

Fig 7.1.5: Accuracy Obtained (using RandomForest algorithm)

```
> RF_Model$confusion
     0    1    2    3    4    5    6    7    8    9 class.error
0 2405    0    9   10    1   18   23   10   13    3  0.03491172
1    1 2716   22   12    3    8   11    4    8    2  0.02547542
2   19    8 2254   29   27    6   17   35   43   12  0.08000000
3    7    4   42 2393    7   61   18   20   57   18  0.08907499
4    2   12   22    2 2277    8   29   11    9   88  0.07439024
5   25    5   11   82   25 2073   23    8   28   18  0.09791123
6   31    4   16    1   11   23 2411    0   13    1  0.03982477
7    4   19   43    9   25    4    4 2468   12   58  0.06727135
8    7   13   27  106   14   53   21   18 2119   37  0.12256729
9   12    7   12   52   79   21    5   55   21 2250  0.10501193
>
```

Fig 7.1.6: Confusion Matrix of RandomForest Algorithm

**7.2 Graphs**

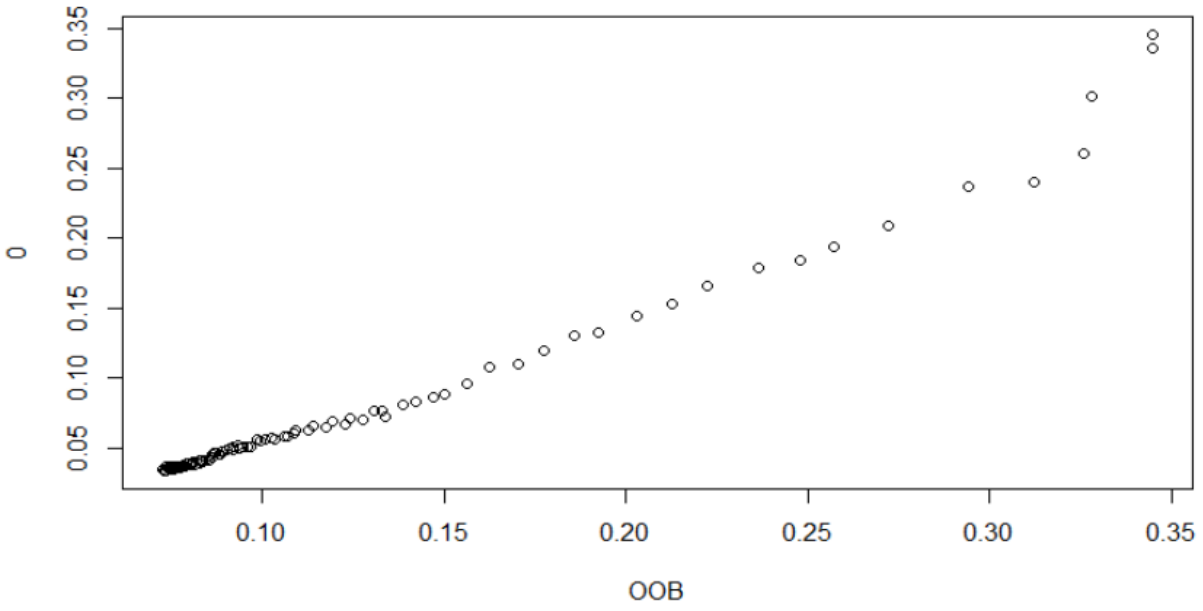



Fig 7.2.1: Error Rate (Random Forest Algorithm)

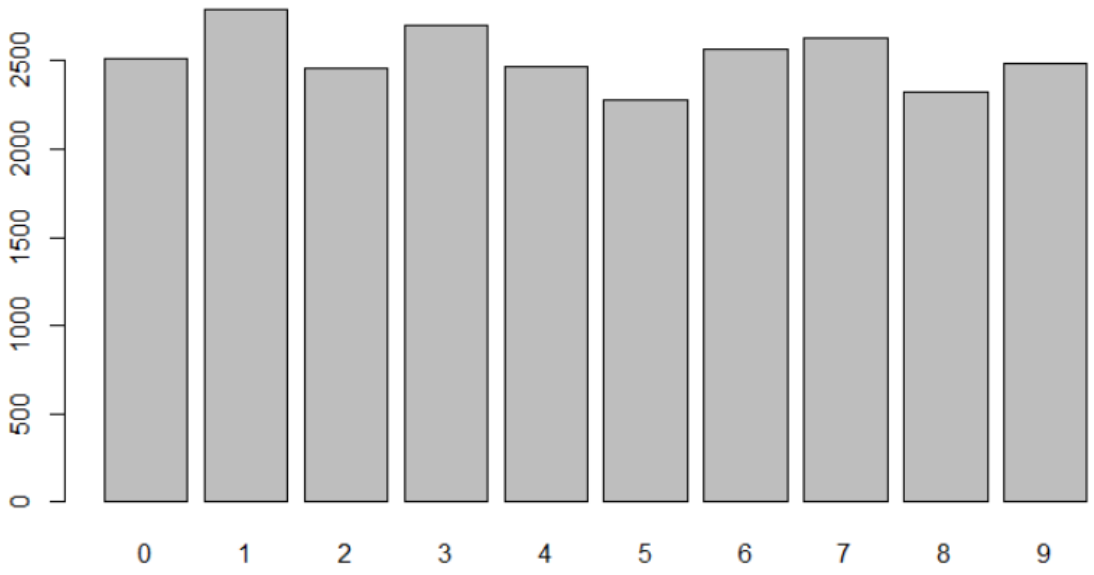



Fig 7.2.2: Predicted Data

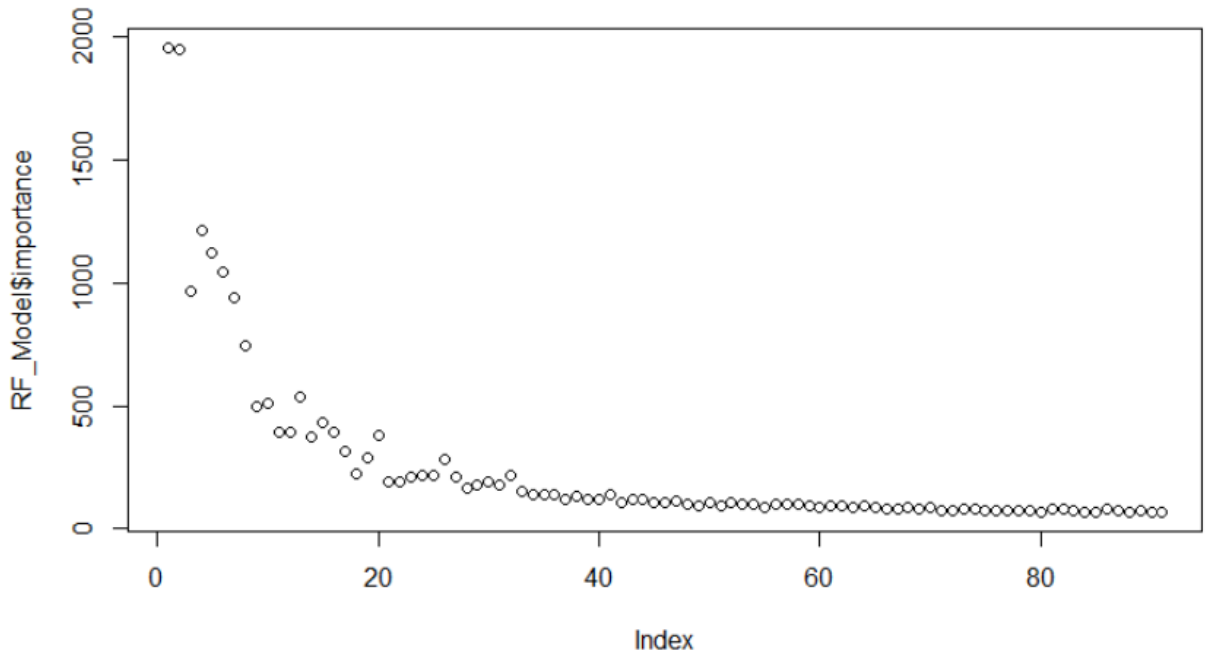



Fig 7.2.3: Importance (Random Forest Algorithm)

# 8. FURTHER IMPROVEMENTS

Now we are using MNIST dataset for training and testing purpose which already had data converted into machine understandable format. After improvising the code, we can directly consider the images and obtain the results (the image is converted by the code itself). We can also consider the use of alphabets for further study of the neural network. By obtaining more and more accurate results, we can develop applications which help in conversion of the images into data. These applications can in used in different sectors for easy conversion of recorded data into digital data.

# 9. CONCLUSION

Explored R programming with RStudio as a platform. Build a model which included data pre-processing and data modeling (Hand written number identification) gain insights about MNIST database, in R. Experimental results are considered to explore the computational techniques for the study of Neural networks and Deep Learning.

# 10. REFERENCES

https://www.rstudio.com/online-learning/

https://en.wikipedia.org/wiki/Principal_component_analysis

https://en.wikipedia.org/wiki/Random_forest

https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

http://yann.lecun.com/exdb/mnist/

http://neuralnetworksanddeeplearning.com/

Books:

- Machine Learning with R, By Brett Lantz
- A Beginner's Guide to R (Use R) By Alain F. Zuur, Elena N. Ieno, Erik H.W.G. Meesters, Springer 2009
- Introduction to Statistics and Data Analysis - With Exercises, Solutions and Applications in R By Christian Heumann, Michael Schomaker and Shalabh, Springer, 2016