THEORY OF ESTIMATION USING ARTIFICIAL INTELLIGENCE

By

Surajit Das
Auropriya Sinha
Sana Khan Bano
Kaustav Biswas

UNDER THE GUIDANCE OF

Mr. Jaydip Mukhopadhyay

PROJECT REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY

Session 2018-2019

श्रमम् बिना न किमपि साध्यम्

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY
[Affiliated to MAKAUT, West Bengal]
CANAL SOUTH ROAD, BELIAGHATA, KOLKATA-700015

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
RCC INSTITUTE OF INFORMATION TECHNOLOGY

श्रमम् बिना न किमपि साध्यम्

<u>TO WHOM IT MAY CONCERN</u>

I hereby recommend that the Project entitled **Theory of Estimation using Artificial Intelligence** prepared under my supervision by **Surajit Das, Auropriya Sinha, Sana Khan Bano & Kaustav Biswas** bearing (Reg no.s 141170110084, 141170110021, 141170110057, 141170110035 & R No.s 11700114084, 11700114021, 11700114057, 11700114035 & Class Roll No.s CSE2014/004, CSE2014/020, CSE2014/024, CSE2014/032 respectively of B.Tech (7th /8th Semester), may be accepted in partial fulfillment for the degree of **Bachelor of Technology in Computer Science & Engineering** under MAKAUT, West Bengal.

.

……………………………………………
Project Supervisor
Department of Computer Science and Engineering
RCC Institute of Information Technology

Countersigned:

……………………………………….
Head
Department of Computer Sc. & Engg,
RCC Institute of Information Technology
Kolkata – 700015.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY

श्रमम् बिना न किमपि साध्यम्

CERTIFICATE OF APPROVAL

The foregoing Project is hereby accepted as a credible study of an engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve the project only for the purpose for which it is submitted.

FINAL EXAMINATION FOR EVALUATION OF PROJECT

1. ____________________________

2. ____________________________

(Signature of Examiners)

ACKNOWLEDGEMENT

We, take this opportunity to express our profound gratitude and deep regards to our guide Mr. Jaydip Mukhopadhyay for his exemplary guidance , monitoring and constant encouragement throughout the course of this project . The blessing , help and guidance given by him time to time will carry us a long way in the journey of life on which we are about to embark.

We are obliged to each other as team members for the valuable information provided by each of us in our respective fields. We are grateful for each other's co-operation during the period of our assignment.

Surajit Das (CSE2014/004)

Auropriya Sinha (CSE2014/020)

Sana Khan Bano (CSE2014/024)

Kaustav Biswas (CSE2014/032)

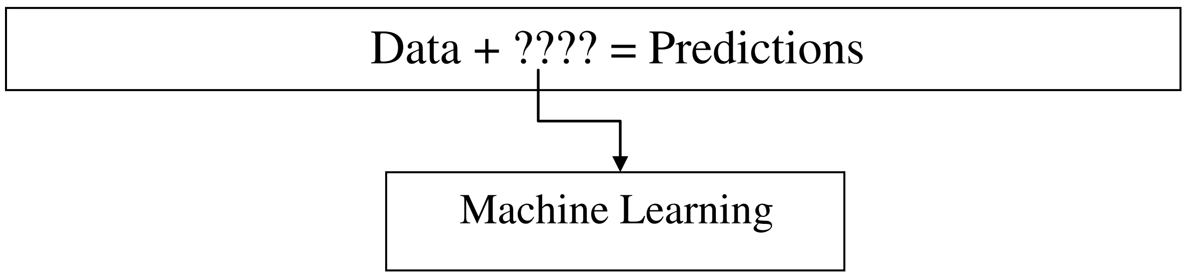
Table of Contents

	Page No.
1. Introduction ……………………………………………………….	**6 - 9**
2. Review of Literature ……………………………………………	**10 - 54**
3. Objective of the Project…………………………………………	**54 - 55**
4. System Design……………………………………………………	**55 - 56**
5. Methodology for implementation (Formulation/Algorithm)……	**56 - 91**
6. Implementation Details………………………………………….	**91 - 110**
7. Results/Sample output…………………………………………..	**110 - 118**
8. Conclusion…………………………………………………….	**118**
Appendix-: *Program Source code with adequate comments…………*	**119**
References	**119 - 120**

1.Introduction:

The domain of application of our project is STOCK MARKET PRICE PREDICTION. The STOCK MARKET allows us to buy and sell units of ownerships in a company, which we call STOCKS. If the company profits go up, we own some of those profits. If they go down, we lose profits with them. STOCK market price prediction is a very complex and challenging research area where different methods have been developed to predict stock price movement in the market.

So, if we were to buy stocks in the right company and at the right time, we could become rich overnight. Is there something we could do to predict future stock prices, given a data set of past prices?



This sounds like a Data Science problem. But according to the **"Efficient Market Hypothesis"**, the Stock Market is random and unpredictable. But major financial firms like J.P. Morgan, Goldman Sachs, Citi Group have been hiring Quantitative Traders for years to build Predictive Models on Past Market Data.
Let's think about the process of creating investment strategy: once we've decided that we do want to make an investment, we need to figure out which companies out there are most likely going to give us big returns. Generally, we start by doing some research on the company's history, use the past news articles and how the company has fared over the years. We would look how stock prices have changed over time, may observe what others are saying about the future of the company on Twitter, and finally, we would collect all of these data that we've gathered in our head, to make a prediction about our future price.

That's the prefect use case for Machine Learning, learning from past data points, to predict future ones. In a very recent paper from **"Auburn"**, on this topic, a small group concluded that using data from several sources like **Google Trends, Wikipedia and Google Correlate**, resulted in a model capable of assisting in investment decisions and have a relatively high accuracy(more than 85% accuracy) from movement predictions. So, we know, academics are researching this, and, we also know that big banks are definitely doing this.

When it comes to the type of models, we could use, we've a huge assortment to try from. Pending results from papers is a good start to see what has been tried before and it's not just about using Pattern Recognition algorithms.

Records for prices for traded commodities goes back to thousands of years. In Finance, the field of **Quantitative Analysis** is about 25 years old. But even now it's still not fully accepted, understood or widely used. It's a study of how certain variables correlate with stock price behaviour. One of the first attempt at this was made in the Seventies by two British statisticians **Box and Jenkins** using **Mainframe computers.** The only historical data they had access to were prices and volumes. They called their model **"Arima"**. At that time, it was and expensive to run. But, by the Eighties, things started to get interesting. Spread-sheets were invented so that firms could model companies' financial performance, and thus automated data collection became a reality and with improvement in computing power, models could analyze data much faster. It was like a renaissance on the Financial market.

In the past few years, we've seen a lot of academic papers published using neural networks, to predict stock prices, with varying degrees of success. But until recently, the ability to build these models has been restricted to academics who spend their days writing very complex codes.

But now with libraries like TensorFlow and Keras, anyone can build powerful predictive models, trained on massive datasets.

The Kalman filter was a significant breakthrough in the area of linear filtering and prediction. It has been used in the processing of signals imbedded in noise for over twenty five years. A major application of Kalman filtering is the solution of navigational problems where information is received from multiple noisy sources. The Kalman filter has also been used for applications outside the area of navigation. C. R. Szelag published an article in the Bell System Technical Journal using a Kalman filter to forecast telephone loading. The Kalman filter has even made its way into the economic literature.

The Kalman filter has been used to forecast economic quantities such as sales and inventories. This project examines the use of the Kalman filter to forecast intraday stock and commodity prices. The price forecasts are based on a market's price history with no external information included. For the Kalman filter to produce beneficial forecasts, the market must not be a random walk process, but must exhibit a statistically significant autocorrelation pattern which can be modeled.

Once an appropriate Kalman filter model is determined, strategies for increasing profits can be studied. This dissertation presents the analysis techniques used to detect autocorrelation in a market and the models used to describe the correlation. Several stock indexes and commodity markets are tested for autocorrelation. The Kalman filter algorithm and an adaptive Kalman filter algorithm are also presented and then are used to forecast prices for the Dow Jones Transportation index. Several buy and sell strategies are used to investigate the use of the Kalman filter forecasts to benefit market traders.

Amazon and Microsoft control the cloud market [through which AI is going to be delivered], but they don’t have frameworks like Tensorflow [Google] or Caffe/Torch [Facebook] to give them a strong leg up.

Amazon and Facebook have the key channels through which AI is mostly accessed by public [Alexa or Facebook messenger].

Amazon and Google have the best speech APIs and NLP.

Microsoft and Amazon provide the best computer vision APIs.

Microsoft and IBM have the best sales teams in this space and work with the widest range of partners to build the AI ecosystem.

Microsoft and Google provide ways to train models through services without worrying about the underlying ML frameworks.

Facebook provides support to the widest range of opensource AI projects, but don’t play the services game. Thus, they might not dominate the AI market.

IBM Watson is the oldest and perhaps the most complete of AI tools/services, but don’t engage well with small developers and thus their applications are limited. Their focus is mostly on enterprise.

Google’s enterprise sales is weak, but it has perhaps the best of AI technology available both inside and outside. The question is just how well can they interact with the ecosystem and help build mission critical applications.

Apple has a good AI team inside but unlike other companies they don’t publish a lot or talk outside their company. No one knows what they do and from what is available public they are perhaps the weakest of the majors in this segment. Not surprisingly Siri has lost out to its competitors in terms of usefulness.

In short, it is a game where no one company really dominates. But, Google perhaps has a slight leg up over the others if everything is taken into consideration.

The approach of our project is unique in the sense that it is aimed at getting better estimation through two successive stages of filtering, namely,

(i) A ***Machine Learning*** (an AI sub-domain) Classifier, which is a new-age efficient estimation technology.

(ii) ***Kalman Filtering***, which is a vintage efficient estimation technology.

2. Review of Literature:

- Artificial Intelligence:

Artificial intelligence (AI, also machine intelligence, MI) is Intelligence displayed by machines, in contrast with the natural intelligence (NI) displayed by humans and other animals. In computer science AI research is defined as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of success at some goal. Colloquially, the term "artificial intelligence" is applied when a machine mimics "cognitive" functions that humans associate with other human minds, such as "learning" and "problem solving".

While thought-capable artificial beings appeared as storytelling devices in antiquity, the idea of actually trying to build a machine to perform useful reasoning may have begun with Ramon Llull (c. 1300 CE). With his Calculus ratiocinator, Gottfried Leibniz extended the concept of the calculating machine (Wilhelm Schickard engineered the first one around 1623), intending to perform operations on concepts rather than numbers. Since the 19th century, artificial beings are common in fiction, as in Frankenstein or Karel Čapek's R.U.R. (Rossum's Universal Robots).

- Machine Learning:

Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel, an American pioneer in the field of computer gaming and artificial intelligence, coined the term "Machine Learning" in 1959 while at IBM. Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs. Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or infeasible; example applications include email filtering, detection of network intruders or malicious insiders working towards a data breach, optical character recognition (OCR), learning to rank, and computer vision.

As a scientific endeavour, machine learning grew out of the quest for artificial intelligence. Already in the early days of AI as an academic discipline, some researchers were interested in having machines learn from data. They attempted to approach the problem with various symbolic methods, as well as what were then termed "neural networks"; these were mostly perceptrons and other models that were later found to be reinventions of the generalized linear models of statistics. Probabilistic reasoning was also employed, especially in automated medical diagnosis.

However, an increasing emphasis on the logical, knowledge-based approach caused a rift between AI and machine learning. Probabilistic systems were plagued by theoretical and practical problems of data acquisition and representation. By 1980, expert systems had come to dominate AI, and statistics was out of favor. Work on symbolic/knowledge-based learning did continue within AI, leading to inductive logic programming, but the more statistical line of research was now outside the field of AI proper, in pattern recognition and information retrieval. Neural networks research had been abandoned by AI and computer science around the same time. This line, too, was continued outside the AI/CS field, as "connectionism", by researchers from other disciplines including Hopfield, Rumel hart and Hinton. Their main success came in the mid-1980s with the reinvention of back propagation.

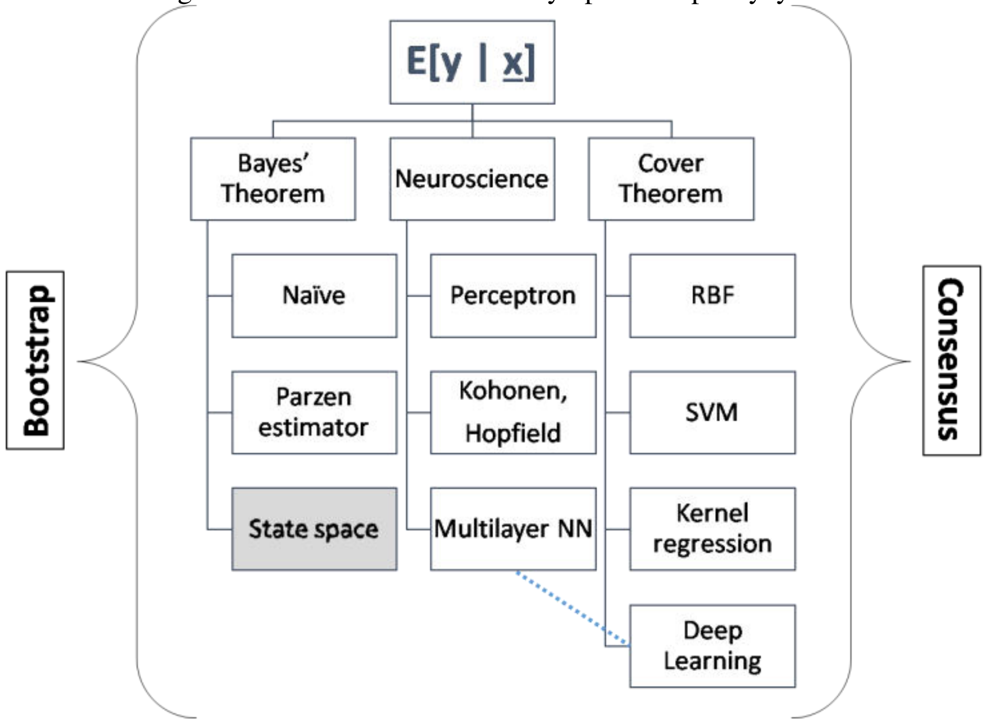


Fig. Machine Learning Ontology

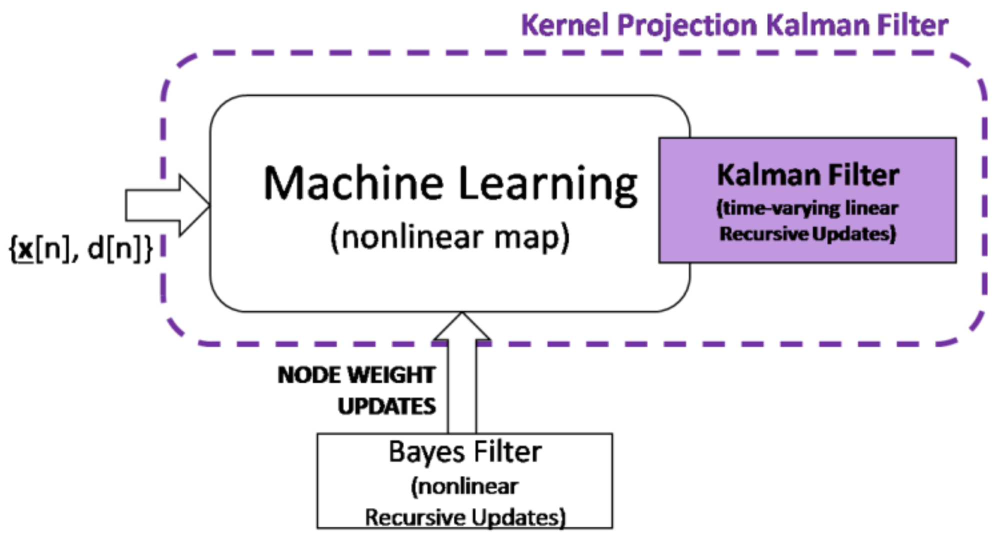

Kernel Projection Kalman Filter
Machine Learning
(nonlinear map)
Kalman Filter
(time-varying linear Recursive Updates)
{x[n], d[n]}
NODE WEIGHT UPDATES
Bayes Filter
(nonlinear Recursive Updates)

Fig. Diagram combining two classes

- Machine Learning Classifiers:
  - Decision Tree Algorithm:

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.

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).

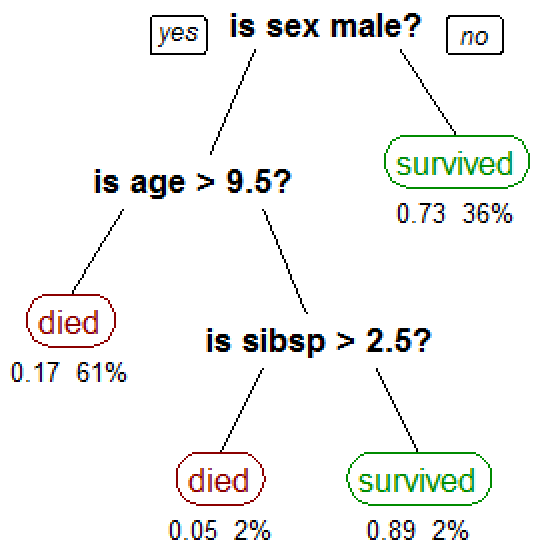


Image taken from Wikipedia

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, more about it can be found here.
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 trees 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.

- Random Forest Model:

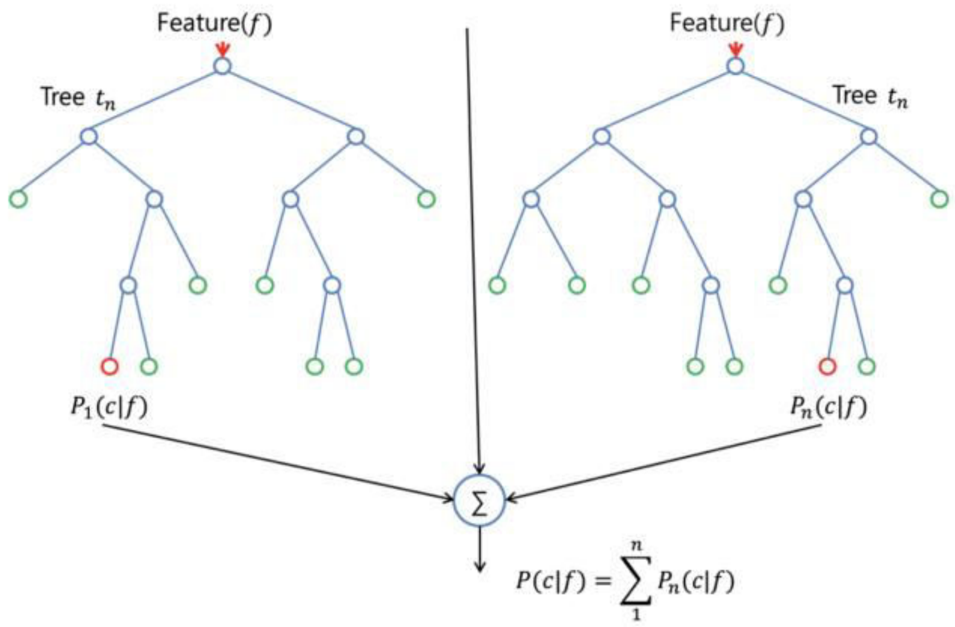
Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because it's simplicity and the fact that it can be used for both classification and regression tasks. In this post, you are going to learn, how the random forest algorithm works and several other important things about it.

How it works:

Random Forest is a supervised learning algorithm. Like you can already see from its 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:



With a few exceptions a random-forest classifier has all the hyper-parameters of a decision-tree classifier and also all the hyper-parameters of a bagging classifier, to control the ensemble itself. Instead of building a bagging-classifier and passing it into a decision-tree-classifier, you can just use the random-forest classifier class, which is more convenient and optimized for decision trees. Note that there is also a random-forest regressor for regression tasks.

The random-forest algorithm brings extra randomness into the model, when it is growing the trees. Instead of searching for the best feature while splitting a node, it searches for the best feature among a random subset of features. This process creates a wide diversity, which generally results in a better model.

Therefore when you are growing a tree in random forest, only a random subset of the features is considered for splitting a node. You can even make trees more random, by using random thresholds on top of it, for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

- Deep Neural Network:

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Neural networks help us cluster and classify. You can think of them as a clustering and classification layer on top of the data you store and manage. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on. (Neural networks can also extract features that are fed to other algorithms for clustering and classification; so you can think of deep neural networks as components of larger machine-learning applications involving algorithms for reinforcement learning, classification and regression.)

Classification

All classification tasks depend upon labeled datasets; that is, humans must transfer their knowledge to the dataset in order for a neural to learn the correlation between labels and data. This is known as *supervised learning*.

Detect faces, identify people in images, recognize facial expressions (angry, joyful)

Identify objects in images (stop signs, pedestrians, lane markers…)

Recognize gestures in video

Detect voices, identify speakers, transcribe speech to text, recognize sentiment in voices

Classify text as spam (in emails), or fraudulent (in insurance claims); recognize sentiment in text (customer feedback)

Any labels that humans can generate, any outcomes you care about and which correlate to data, can be used to train a neural network.

Clustering

Clustering or grouping is the detection of similarities. Deep learning does not require labels to detect similarities. Learning without labels is called *unsupervised learning*. Unlabeled data is the majority of data in the world. One law of machine learning is: the more data an algorithm can train on, the more accurate it will be. Therefore, unsupervised learning has the potential to produce highly accurate models.

Search: Comparing documents, images or sounds to surface similar items.

Anomaly detection: The flipside of detecting similarities is detecting anomalies, or unusual behavior. In many cases, unusual behavior correlates highly with things you want to detect and prevent, such as fraud.

Predictive Analytics: Regressions

With classification, deep learning is able to establish correlations between, say, pixels in an image and the name of a person. You might call this a static prediction. By the same token, exposed to enough of the right data, deep learning is able to establish correlations between present events and future events. It can run regression between the past and the future. The future event is like the label in a sense. Deep learning doesn't necessarily care about time, or the fact that something hasn't happened yet. Given a time series, deep learning may read a string of number and predict the number most likely to occur next.

Hardware breakdowns (data centers, manufacturing, transport)

Health breakdowns (strokes, heart attacks based on vital stats and data from wearables)

Customer churn (predicting the likelihood that a customer will leave, based on web activity and metadata)

Employee turnover (ditto, but for employees)

The better we can predict, the better we can prevent and pre-empt. As you can see, with neural networks, we're moving towards a world of fewer surprises. Not zero surprises, just marginally fewer. We're also moving toward a world of smarter agents that combine neural networks with other algorithms like reinforcement learning to attain goals.

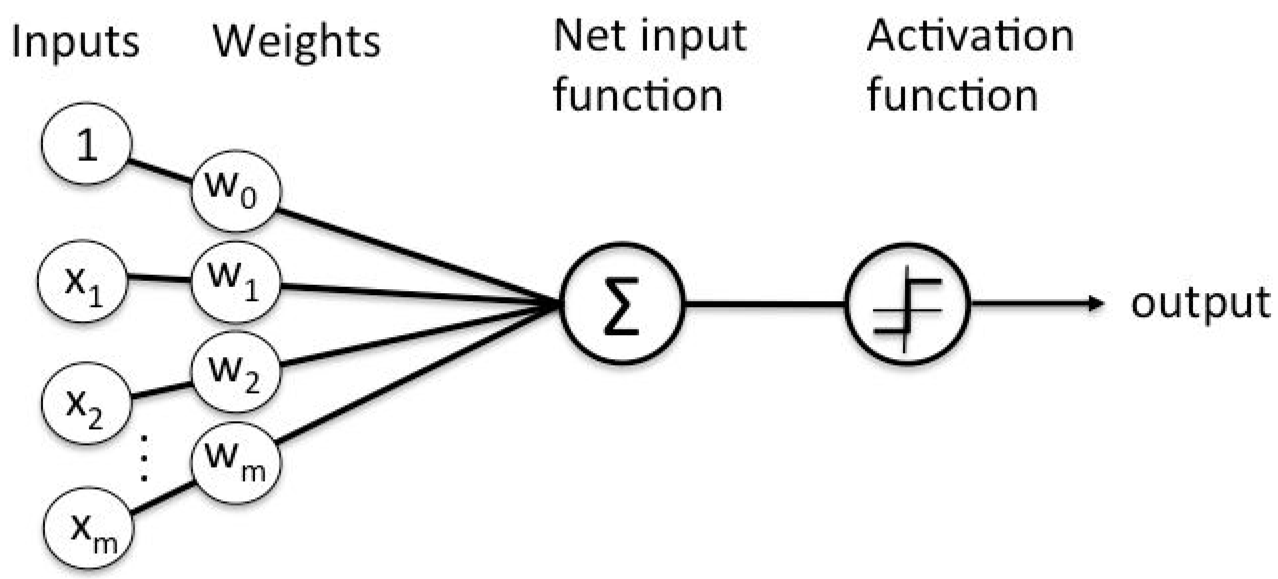
With that brief overview of deep learning use cases, let's look at what neural nets are made of.

Neural Network Elements

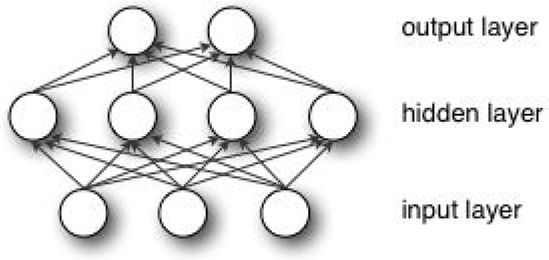
Deep learning is the name we use for "stacked neural networks"; that is, networks composed of several layers.

The layers are made of *nodes*. A node is just a place where computation happens, loosely patterned on a neuron in the human brain, which fires when it encounters sufficient stimuli. A node combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning significance to inputs for the task the algorithm is trying to learn. (For example, which input is most helpful is classifying data without error?) These input-weight products are summed and the sum is passed through a node's so-called activation function, to determine whether and to what extent that signal progresses further through the network to affect the ultimate outcome, say, an act of classification.

Here's a diagram of what one node might look like.



A node layer is a row of those neuronlike switches that turn on or off as the input is fed through the net. Each layer's output is simultaneously the subsequent layer's input, starting from an initial input layer receiving your data.



Pairing adjustable weights with input features is how we assign significance to those features with regard to how the network classifies and clusters input.

Example: Feedforward Networks

Our goal in using a neural net is to arrive at the point of least error as fast as possible. We are running a race, and the race is around a track, so we pass the same points repeatedly in a loop. The starting line for the race is the state in which our weights are initialized, and the finish line is the state of those parameters when they are capable of producing accurate classifications and predictions.

The race itself involves many steps, and each of those steps resembles the steps before and after. Just like a runner, we will engage in a repetitive act over and over to arrive at the finish. Each step for a neural network involves a guess, an error measurement and a slight update in its weights, an incremental adjustment to the coefficients.

A collection of weights, whether they are in their start or end state, is also called a model, because it is an attempt to model data's relationship to ground-truth labels, to grasp the data's structure. Models normally start out bad and end up less bad, changing over time as the neural network updates its parameters.

This is because a neural network is born in ignorance. It does not know which weights and biases will translate the input best to make the correct guesses. It has to start out with a guess, and then try to make better guesses sequentially as it learns from its mistakes. (You can think of a neural network as a miniature enactment of the scientific method, testing hypotheses and trying again – only it is the scientific method with a blindfold on.)

Here is a simple explanation of what happens during learning with a feedforward neural network, the simplest architecture to explain.

Input enters the network. The coefficients, or weights, map that input to a set of guesses the network makes at the end.

```
input * weight = guess
```

Weighted input results in a guess about what that input is. The neural then takes its guess and compares it to a ground-truth about the data, effectively asking an expert "Did I get this right?"

```
ground truth - guess = error
```

The difference between the network's guess and the ground truth is its error. The network measures that error, and walks the error back over its model, adjusting weights to the extent that they contributed to the error.

```
error * weight's contribution to error = adjustment
```

The three pseudo-mathematical formulas above account for the three key functions of neural networks: scoring input, calculating loss and applying an update to the model – to begin the three-step process over again. A neural network is a corrective feedback loop, rewarding weights that support its correct guesses, and punishing weights that lead it to err.

Let's linger on the first step above.

Multiple Linear Regression

Despite their biologically inspired name, artificial neural networks are nothing more than math and code, like any other machine-learning algorithm. In fact, anyone who understands linear regression, one of first methods you learn in statistics, can understand how a neural net works. In its simplest form, linear regression is expressed as

```
Y_hat = bX + a
```

where Y_hat is the estimated output, X is the input, b is the slope and a is the intercept of a line on the vertical axis of a two-dimensional graph. (To make this more concrete: X could be radiation exposure and Y could be the cancer risk; X could be daily pushups and Y could be the total weight you can benchpress; X the amount of fertilizer and Y the size of the crop.) You can imagine that every time you add a unit to X, the dependent variable Y increases proportionally, no matter how far along you are on the X axis. That simple relation between two variables moving up or down together is a starting point.

The next step is to imagine multiple linear regression, where you have many input variables producing an output variable. It's typically expressed like this:

```
Y_hat = b_1*X_1 + b_2*X_2 + b_3*X_3 + a
```

(To extend the crop example above, you might add the amount of sunlight and rainfall in a growing season to the fertilizer variable, with all three affecting Y_hat.)

Now, that form of multiple linear regression is happening at every node of a neural network. For each node of a single layer, input from each node of the previous layer is recombined with input from every other node. That is, the inputs are mixed in different proportions, according to their coefficients, which are different leading into each node of the subsequent layer. In this way, a net tests which combination of input is significant as it tries to reduce error.

Once you sum your node inputs to arrive at Y_hat, it's passed through a non-linear function. Here's why: If every node merely performed multiple linear regression, Y_hat would increase linearly and without limit as the X's increase, but that doesn't suit our purposes.

What we are trying to build at each node is a switch (like a neuron…) that turns on and off, depending on whether or not it should let the signal of the input pass through to affect the ultimate decisions of the network.

When you have a switch, you have a classification problem. Does the input's signal indicate the node should classify it as enough, or not_enough, on or off? A binary decision can be expressed by 1 and 0, and logistic regression is a non-linear function that squashes input to translate it to a space between 0 and 1.

The nonlinear transforms at each node are usually s-shaped functions similar to logistic regression. They go by the names of sigmoid (the Greek word for "S"), tanh, hard tanh, etc., and they shaping the output of each node. The output of all nodes, each squashed into an s-shaped space between 0 and 1, is then passed as input to the next layer in a feed forward neural network, and so on until the signal reaches the final layer of the net, where decisions are made.

Gradient Descent

The name for one commonly used optimization function that adjusts weights according to the error they caused is called "gradient descent."

Gradient is another word for slope, and slope, in its typical form on an x-y graph, represents how two variables relate to each other: rise over run, the change in money over the change in time, etc. In this particular case, the slope we care about describes the relationship between the network's error and a single weight; i.e. that is, how does the error vary as the weight is adjusted.

To put a finer point on it, which weight will produce the least error? Which one correctly represents the signals contained in the input data, and translates them to a correct classification? Which one can hear "nose" in an input image, and know that should be labeled as a face and not a frying pan?

As a neural network learns, it slowly adjusts many weights so that they can map signal to meaning correctly. The relationship between network Error and each of those weights is a derivative, dE/dw, that measures the degree to which a slight change in a weight causes a slight change in the error.

Each weight is just one factor in a deep network that involves many transforms; the signal of the weight passes through activations and sums over several layers, so we use the chain rule of calculus to march back through the networks activations and outputs and finally arrive at the weight in question, and its relationship to overall error.

The chain rule in calculus states that

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

In a feedforward network, the relationship between the net's error and a single weight will look something like this:

$$\frac{dError}{dweight} = \frac{dError}{dactivation} * \frac{dactivation}{dweight}$$

That is, given two variables, Error and weight, that are mediated by a third variable, activation, through which the weight is passed, you can calculate how a change in weight affects a change in Error by first calculating how a change in activation affects a change in Error, and how a change in weight affects a change in activation.

The essence of learning in deep learning is nothing more than that: adjusting a model's weights in response to the error it produces, until you can't reduce the error any more.

Updaters

DL4J support the following Updaters

- ADADELTA
- ADAGRAD
- ADAM
- NESTEROVS
- NONE
- RMSPROP
- SGD
- CONJUGATE GRADIENT
- HESSIAN FREE
- LBFGS
- LINE GRADIENT DESCENT

The JavaDoc for updaters is part of the DeepLearning4J JavaDoc and is available here.

Activation Functions

The activation function determines what output a node will generate base upon its input. Sigmoid activation functions had been very populare, ReLU is currently very popular. In DeepLearnging4J the activation function is set at the layer level and applies to all neurons in that layer.

Supported Activation functions

- CUBE
- ELU
- HARDSIGMOID
- HARDTANH
- IDENTITY
- LEAKYRELU

- RATIONALTANH
- RELU
- RRELU
- SIGMOID
- SOFTMAX
- SOFTPLUS
- SOFTSIGN
- TANH

Configuring an activation function

```
layer(2, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOG
LIKELIHOOD).activation(Activation.SOFTMAX)
```

Custom layers, activation functions and loss functions

Deeplearning4j support custom Layers, activations and Loss Functions.

Logistic Regression

On a deep neural network of many layers, the final layer has a particular role. When dealing with labeled input, the output layer classifies each example, applying the most likely label. Each node on the output layer represents one label, and that node turns on or off according to the strength of the signal it receives from the previous layer's input and parameters.

Each output node produces two possible outcomes, the binary output values 0 or 1, because an input variable either deserves a label or it does not. After all, there is no such thing as a little pregnant.

While neural networks working with labeled data produce binary output, the input they receive is often continuous. That is, the signals that the network receives as input will span a range of values and include any number of metrics, depending on the problem it seeks to solve.

For example, a recommendation engine has to make a binary decision about whether to serve an ad or not. But the input it bases its decision on could include how much a customer has spent on Amazon in the last week, or how often that customer visits the site.

So the output layer has to condense signals such as $67.59 spent on diapers, and 15 visits to a website, into a range between 0 and 1; i.e. a probability that a given input should be labeled or not.

The mechanism we use to convert continuous signals into binary output is called logistic regression. The name is unfortunate, since logistic regression is used for classification rather than regression in the linear sense that most people are familiar with. It calculates the probability that a set of inputs match the label.

$$F(x) = \frac{1}{1 + e^{-x}}$$

Let’s examine this little formula.

For continuous inputs to be expressed as probabilities, they must output positive results, since there is no such thing as a negative probability. That’s why you see input as the exponent of e in the denominator – because exponents force our results to be greater than zero. Now consider the relationship of e’s exponent to the fraction 1/1. One, as we know, is the ceiling of a probability, beyond which our results can’t go without being absurd. (We’re 120% sure of that.)

As the input x that triggers a label grows, the expression e to the x shrinks toward zero, leaving us with the fraction 1/1, or 100%, which means we approach (without ever quite reaching) absolute certainty that the label applies. Input that correlates negatively with your output will have its value flipped by the negative sign on e’s exponent, and as that negative signal grows, the quantity e to the x becomes larger, pushing the entire fraction ever closer to zero.

Now imagine that, rather than having x as the exponent, you have the sum of the products of all the weights and their corresponding inputs – the total signal passing through your net. That’s what you’re feeding into the logistic regression layer at the output layer of a neural network classifier.

With this layer, we can set a decision threshold above which an example is labeled 1, and below which it is not. You can set different thresholds as you prefer – a low threshold will increase the number of false positives, and a higher one will increase the number of false negatives – depending on which side you would like to err.

Loss Functions in DeepLearning4J

DeepLearning4J supports the following Loss Functions.

- MSE: Mean Squared Error: Linear Regression
- EXPLL: Exponential log likelihood: Poisson Regression
- XENT: Cross Entropy: Binary Classification
- MCXENT: Multiclass Cross Entropy
- RMSE_XENT: RMSE Cross Entropy
- SQUARED_LOSS: Squared Loss
- NEGATIVELOGLIKELIHOOD: Negative Log Likelihood

Applying Loss Functions in DeepLearning4J

The Loss Function is applied when building your output Layer.

```
layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
```

The JavaDoc for the Loss Function is part of ND4J javadoc and is available [here.](https://nd4j.org/doc/org/nd4j/linalg/api/ops/LossFunction.html)

Neural Networks & Artificial Intelligence

In some circles, neural networks are thought of as "brute force" AI, because they start with a blank slate and hammer their way through to an accurate model. They are effective, but to some eyes inefficient in their approach to modeling, which can't make assumptions about functional dependencies between output and input.

That said, gradient descent is not recombining every weight with every other to find the best match – its method of pathfinding shrinks the relevant weight space, and therefore the number of updates and required computation, by many orders of magnitude.

Enterprise-Scale Deep Learning

To train complex neural networks on very large datasets, a deep learning cluster using multiple chips, distributed over both GPUs and CPUs, is necessary if one is to train the network in a reasonable amount of time. Software engineers training those nets may avail themselves of GPUs in the cloud, or choose to depend on proprietary racks. Deeplearning4j scales out equally well on both, using Spark as an access layer to orchestrate multiple host threads over many cores. For support, please contact Skymind.

Support Vector Regression:

Support vector machine

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

When data are not labeled, supervised learning is not possible, and an unsupervised learning approach is required, which attempts to find natural clustering of the data to groups, and then map new data to these formed groups. The support vector clustering[2]algorithm created by Hava Siegelmann and Vladimir Vapnik, applies the statistics of support vectors, developed in the support vector machines algorithm, to categorize unlabeled data, and is one of the most widely used clustering algorithms in industrial applications.

Applications

SVMs can be used to solve various real world problems:

- SVMs are helpful in text and hypertext categorization as their application can significantly reduce the need for labeled training instances in both the standard inductive and transductive settings.
- Classification of images can also be performed using SVMs. Experimental results show that SVMs achieve significantly higher search accuracy than traditional query refinement schemes after just three to four rounds of relevance feedback. This is also true of image segmentation systems, including those using a modified version SVM that uses the privileged approach as suggested by Vapnik.
- Hand-written characters can be recognized using SVM.
- The SVM algorithm has been widely applied in the biological and other sciences. They have been used to classify proteins with up to 90% of the compounds classified correctly. Permutation tests based on SVM weights have been suggested as a mechanism for interpretation of SVM models. Support vector machine weights have also been used to interpret SVM models in the past. Posthoc interpretation of support vector machine models in order to identify features used by the model to make predictions is a relatively new area of research with special significance in the biological sciences.

Radial basis function network

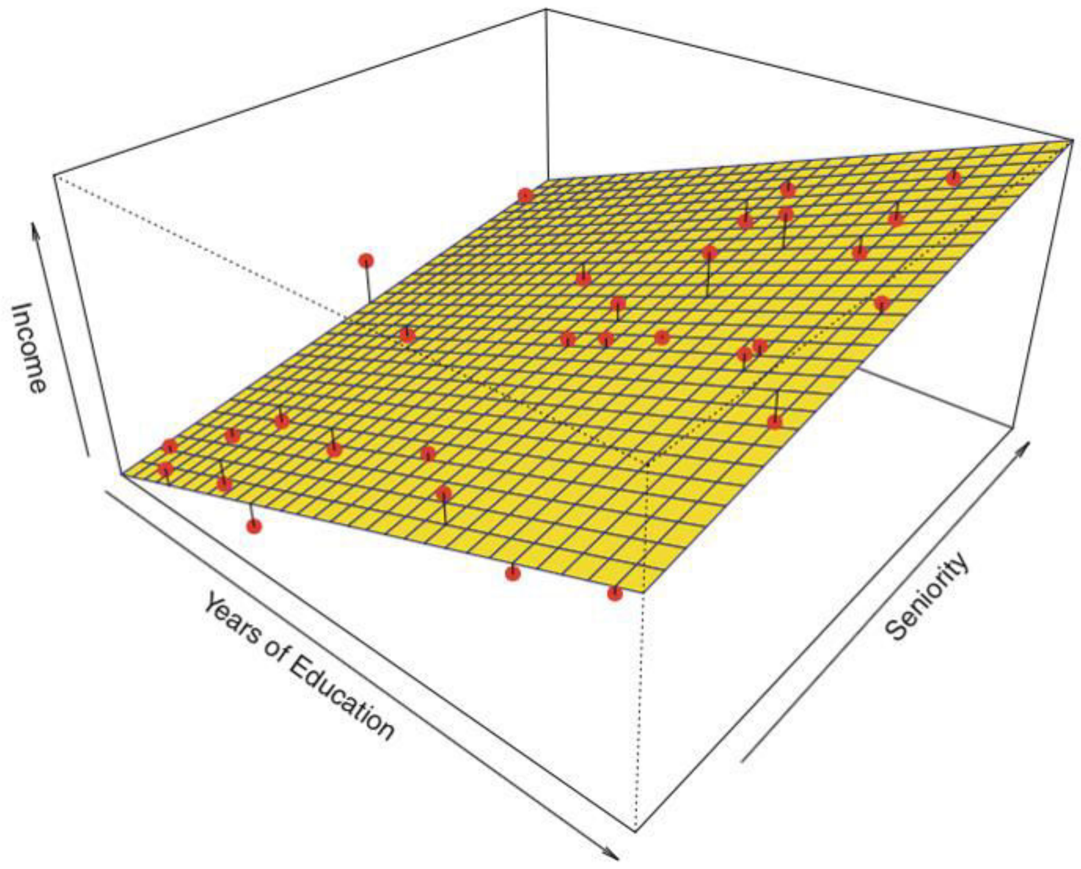
A radial basis function (RBF) is a real-valued function whose value depends only on the distance from the origin. Any function that satisfies the property is called a radial function. The norm is usually Euclidean distance, although other distance functions are also possible.

Sums of radial basis functions are typically used to approximate given functions. This approximation process can also be interpreted as a simple kind of neural network; this was the context in which they originally surfaced, in work by David Broomhead and David Lowe in 1988, which stemmed from Michael J. D. Powell's seminal research from 1977. RBFs are also used as a kernel in support vector classification.

In the field of mathematical modeling, a radial basis function network is an artificial neural network that uses radial basis functions as activation functions. The output of the network is a linear combination of radial basis functions of the inputs and neuron parameters. Radial basis function networks have many uses, including function approximation, time series prediction, classification, and system control. They were first formulated in a 1988 paper by Broomhead and Lowe, both researchers at the Royal Signals and Radar Establishment.

Radial basis function (RBF) networks typically have three layers: an input layer, a hidden layer with a non-linear RBF activation function and a linear output layer. The input can be modeled as a vector of real numbers. Functions that depend only on the distance from a center vector are radially symmetric about that vector, hence the name radial basis function. In the basic form all inputs are connected to each hidden neuron.

- Linear Regression:



Linear Regression Graph

Linear regression is used for finding linear relationship between target and one or more predictors. There are two types of linear regression- Simple and Multiple.

Simple Linear Regression

Simple linear regression is useful for finding relationship between two continuous variables. One is predictor or independent variable and other is response or dependent variable. It looks for statistical relationship but not deterministic relationship. Relationship between two variables is said to be deterministic if one variable can be accurately expressed by the other. For example, using temperature in degree Celsius it is possible to accurately predict Fahrenheit. Statistical relationship is not accurate in determining relationship between two variables. For example, relationship between height and weight.

The core idea is to obtain a line that best fits the data. The best fit line is the one for which total prediction error (all data points) are as small as possible. Error is the distance between the point to the regression line.

Real-time example

We have a dataset which contains information about relationship between 'number of hours studied' and 'marks obtained'. Many students have been observed and their hours of study and grade are recorded. This will be our training data. Goal is to design a model that can predict marks if given the number of hours studied. Using the training data, a regression line is obtained which will give minimum error. This linear equation is then used for any new data. That is, if we give number of hours studied by a student as an input, our model should predict their mark with minimum error.

Y(pred) = b0 + b1*x

The values b0 and b1 must be chosen so that they minimize the error. If sum of squared error is taken as a metric to evaluate the model, then goal to obtain a line that best reduces the error.

$$\text{Error} = \sum_{i=0}^{n}(\text{actual\_output - predicted\_output})^{**}2$$

Figure 2: Error Calculation

If we don't square the error, then positive and negative point will cancel out each other.

For model with one predictor,

$$b_0 = \bar{y} - b_1\bar{x}$$

Figure 3: Intercept Calculation

$$b_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Figure 4: Co-efficient Formula

Exploring ‘b1’

- If b1 > 0, then x(predictor) and y(target) have a positive relationship. That is increase in x will increase y.
- If b1 < 0, then x(predictor) and y(target) have a negative relationship. That is increase in x will decrease y.

Exploring ‘b0’

- If the model does not include x=0, then the prediction will become meaningless with only b0. For example, we have a dataset that relates height(x) and weight(y). Taking x=0(that is height as 0), will make equation have only b0 value which is completely meaningless as in real-time height and weight can never be zero. This resulted due to considering the model values beyond its scope.
- If the model includes value 0, then ‘b0’ will be the average of all predicted values when x=0. But, setting zero for all the predictor variables is often impossible.
- The value of b0 guarantee that residual have mean zero. If there is no ‘b0’ term, then regression will be forced to pass over the origin. Both the regression co-efficient and prediction will be biased.

Co-efficient from Normal equations

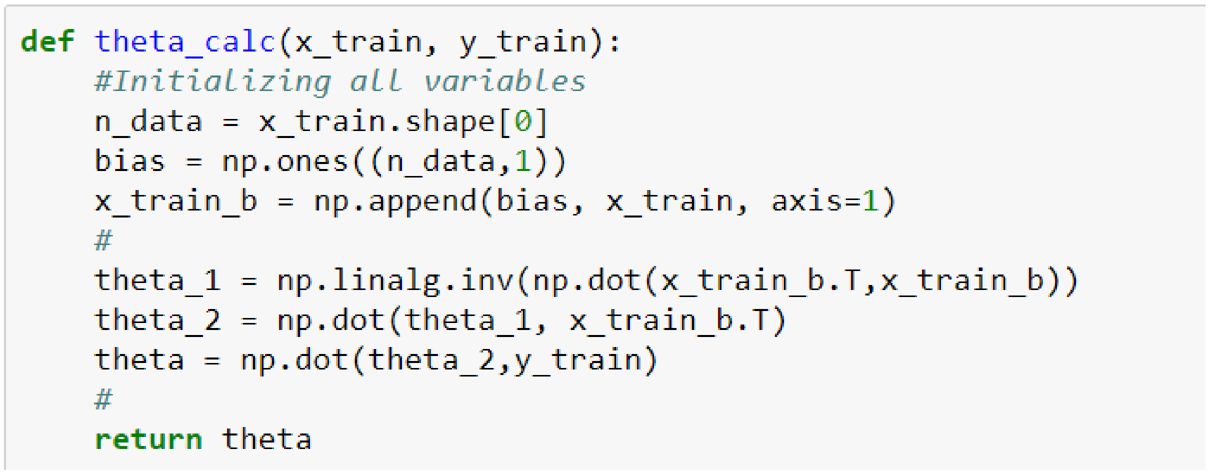
Apart from above equation co-efficient of the model can also be calculated from normal equation.

$$\text{Theta} = (X^TX)^{-1}X^TY$$

Figure 5: Co-efficient calculation using Normal Equation

Theta contains co-efficient of all predictors including constant term ‘b0’. Normal equation performs computation by taking inverse of input matrix. Complexity of the computation will increase as the number of features increase. It gets very slow when number of features grow large.

Below is the python implementation of the equation.



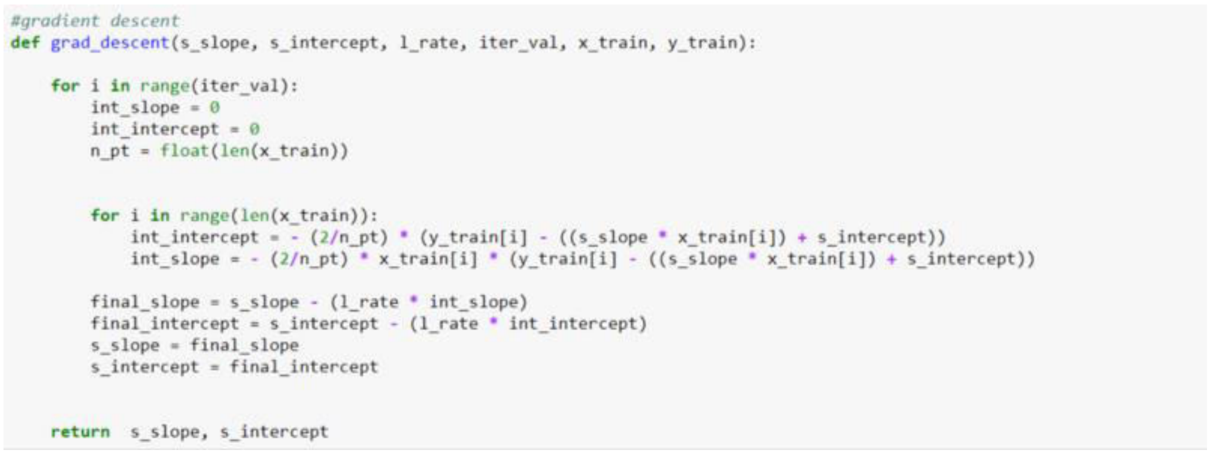
```
def theta_calc(x_train, y_train):
    #Initializing all variables
    n_data = x_train.shape[0]
    bias = np.ones((n_data,1))
    x_train_b = np.append(bias, x_train, axis=1)
    #
    theta_1 = np.linalg.inv(np.dot(x_train_b.T,x_train_b))
    theta_2 = np.dot(theta_1, x_train_b.T)
    theta = np.dot(theta_2,y_train)
    #
    return theta
```

Python implementation of Normal Equation

Optimizing using gradient descent

Complexity of the normal equation makes it difficult to use, this is where gradient descent method comes into picture. Partial derivative of the cost function with respect to the parameter can give optimal co-efficient value.

Python code for gradient descent



```
#gradient descent
def grad_descent(s_slope, s_intercept, l_rate, iter_val, x_train, y_train):

    for i in range(iter_val):
        int_slope = 0
        int_intercept = 0
        n_pt = float(len(x_train))


        for i in range(len(x_train)):
            int_intercept = - (2/n_pt) * (y_train[i] - ((s_slope * x_train[i]) + s_intercept))
            int_slope = - (2/n_pt) * x_train[i] * (y_train[i] - ((s_slope * x_train[i]) + s_intercept))

        final_slope = s_slope - (l_rate * int_slope)
        final_intercept = s_intercept - (l_rate * int_intercept)
        s_slope = final_slope
        s_intercept = final_intercept


    return  s_slope, s_intercept
```

Python Implementation of gradient descent

- Reinforcement Learning:

While neural networks are responsible for recent breakthroughs in problems like computer vision, machine translation and time series prediction – they can also combine with reinforcement learning algorithms to create something astounding like AlphaGo.

Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps; for example, maximize the points won in a game over many moves. They can start from a blank slate, and under the right conditions they achieve superhuman performance. Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.

Reinforcement algorithms that incorporate deep learning can beat world champions at the game of Go as well as human experts playing numerous Atari video games. While that may sound trivial, it’s a vast improvement over their previous accomplishments, and the state of the art is progressing rapidly.

Reinforcement learning solves the difficult problem of correlating immediate actions with the delayed returns they produce. Like humans, reinforcement learning algorithms sometimes have to wait a while to see the fruit of their decisions. They operate in a delayed return environment, where it can be difficult to understand which action leads to which outcome over many time steps.

Reinforcement learning algorithms can be expected to perform better and better in more ambiguous, real-life environments while choosing from an arbitrary number of possible actions, rather than from the limited options of a video game. That is, with time we expect them to be valuable to achieve goals in the real world.

Two reinforcement learning algorithms - Deep-Q learning and A3C - have been implemented in a Deeplearning4j library called RL4J. It can already play Doom.

Reinforcement Learning Definitions

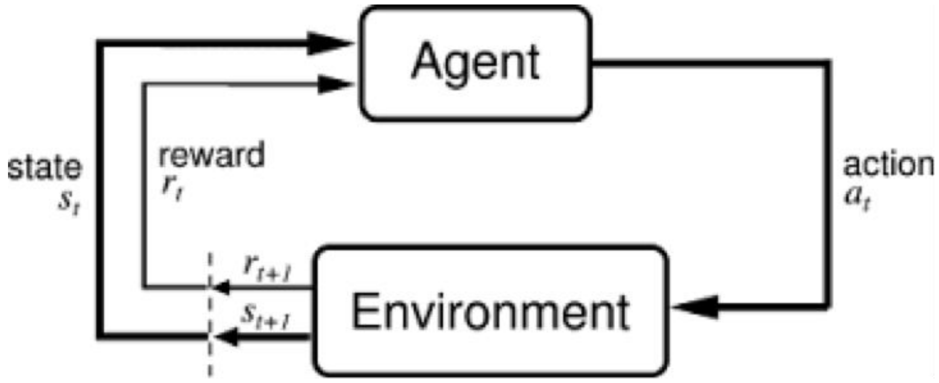
Reinforcement learning can be understand using the concepts of agents, environments, states, actions and rewards, all of which we'll explain below. Capital letters tend to denote sets of things, and lower-case letters denote a specific instance of that thing; e.g. A is all possible actions, while a is a specific action contained in the set.

- Agent: An agent takes actions; for example, a drone making a delivery, or Super Mario navigating a video game. The algorithm is the agent. In life, the agent is you.[1]
- Action (A): A is the set of all possible moves the agent can make.
  An action is almost self-explanatory, but it should be noted that agents choose among a list of possible actions. In video games, the list might include running right or left, jumping high or low, crouching or standing still. In the stock markets, the list might include buying, selling or holding any one of an array of securities and their derivatives. When handling aerial drones, alternatives would include many different velocities and accelerations in 3D space.
- Discount factor: The discount factor is multiplied with future rewards as discovered by the agent in order to dampen their effect on the agent's choice of action. It makes future rewards worth less than immediate rewards; i.e. it

enforces a kind of short-term hedonism on the agent. Often expressed with the lower-case Greek letter gamma: γ. If γ is .8, and there's a reward of 10 points after 3 time steps, the present value of that reward is $0.8^3 \times 10$. A discount factor of 1 would make future rewards worth just as much as immediate rewards.

- Environment: The world through which the agent moves. The environment takes the agent's current state and action as input, and returns as output the agent's reward and next state. If you are the agent, the environment could be the laws of physics and the rules of society that process your actions and determine the consequences of them.
- State (S): A state is a concrete and immediate situation in which the agent finds itself; i.e. a specific place and moment, an instantaneous configuration that puts the agent in relation to other significant things such as tools, obstacles, enemies or prizes. It can the current situation returned by the environment, or any future situation. Were you ever in the wrong place at the wrong time? That's a state.
- Reward (R): A reward is the feedback by which we measure the success or failure of an agent's actions. For example, in a video game, when Mario touches a coin, he wins points. From any given state, an agent sends output in the form of actions to the environment, and the environment returns the agent's new state (which resulted from acting on the previous state) as well as rewards, if there are any. Rewards can be immediate or delayed. They effectively evaluate the agent's action.
- Policy (π): The policy is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, the actions that promise the highest reward.
- Value (V): The expected long-term return with discount, as opposed to the short-term reward R. $V\pi(s)$ is defined as the expected long-term return of the current state under policy π. We discount rewards, or lower their estimated value, the further into the future they occur. See discount factor.
- Q-value or action-value (Q): Q-value is similar to Value, except that it takes an extra parameter, the current action a. $Q\pi(s, a)$ refers to the long-term return of the current state s, taking action a under policy π. Q maps state-action pairs to rewards. Note the difference between Q and policy.
- Trajectory: A sequence of states and actions that influence those states. From the Latin "to throw across."

So environments are functions that transform an action taken in the current state into the next state and a reward; agents are functions that transform the new state and reward into the next action. We can know the agent's function, but we cannot know the function of the environment. It is a black box where we only see the inputs and outputs. Reinforcement learning represents an agent's attempt to approximate the environment's function, such that we can send actions into the black-box environment that maximize the rewards it spits out.



In the feedback loop above, the subscripts denote the time steps t and t+1, each of which refer to different states: the state at moment t, and the state at moment t+1. Unlike other forms of machine learning – such as supervised and unsupervised learning – reinforcement learning can only be thought about sequentially in terms of state-action pairs that occur one after the other.

Reinforcement learning judges actions by the results they produce. It is goal oriented, and its aim is to learn sequences of actions that will lead an agent to achieve its goal, or maximize its objective function. Here are some examples:

- In video games, the goal is to finish the game with the most points, so each additional point obtained throughout the game will affect the agent's subsequent behavior; i.e. the agent may learn that it should shoot battleships, touch coins or dodge meteors to maximize its score.
- In the real world, the goal might be for a robot to travel from point A to point B, and every inch the robot is able to move closer to point B could be counted like points.

Here's an example of an objective function for reinforcement learning; i.e. the way it defines its goal.

$$\sum_{t=0}^{t=\infty} \gamma^t r(x(t), a(t))$$

We are summing reward function r over t, which stands for time steps. So this objective function calculates all the reward we could obtain by running through, say, a game. Here, x is the state at a given time step, and a is the action taken in that state. r is the reward function for x and a. (We'll ignore γ for now.)

Reinforcement learning differs from both supervised and unsupervised learning by how it interprets inputs. We can illustrate their difference by describing what they learn about a "thing."

- Unsupervised learning: That thing is like this other thing. (The algorithms learn similarities w/o names, and by extension they can spot the inverse and perform anomaly detection by recognizing what is unusual or dissimilar)
- Supervised learning: That thing is a "double bacon cheese burger". (Labels, putting names to faces…) These algorithms learn the correlations between data instances and their labels; that is, they require a labelled dataset. Those labels are used to "supervise" and correct the algorithm as it makes wrong guesses when predicting labels.
- Reinforcement learning: Eat that thing because it tastes good and will keep you alive longer. (Actions based on short- and long-term rewards, such as the amount of calories you ingest, or the length of time you survive.) Reinforcement learning can be thought of as supervised learning in an environment of sparse feedback.

The Relationship Between Machine Learning with Time

You could say that an algorithm is a method to more quickly aggregate the lessons of time. Reinforcement learning algorithms have a different relationship to time than humans do. An algorithm can run through the same states over and over again while experimenting with different actions, until it can infer which actions are best from which states. Effectively, algorithms enjoy their very own Groundhog Day, where they start out as dumb jerks and slowly get wise.

Since humans never experience Groundhog Day outside the movie, reinforcement learning algorithms have the potential to learn more, and better, than humans. Indeed, the true advantage of these algorithms over humans stems not so much from their inherent nature, but from their ability to live in parallel on many chips at once, to train night and day without fatigue, and therefore to learn more. An algorithm trained on the game of Go, such as AlphaGo, will have played many more games of Go than any human could hope to complete in 100 lifetimes.[2]

Neural Networks and Deep Reinforcement Learning

Where do neural networks fit in? Neural networks are the agent that learns to map state-action pairs to rewards. Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error.

In reinforcement learning, convolutional networks can be used to recognize an agent's state; e.g. the screen that Mario is on, or the terrain before a drone. That is, they perform their typical task of image recognition.

But convolutional networks derive different interpretations from images in reinforcement learning than in supervised learning. In supervised learning, the network applies a label to an image; that is, it matches names to pixels.

- Keras:

Keras: The Python Deep Learning library

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*

We use Keras if we need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Guiding principles

- User friendliness. Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- Modularity. A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- Easy extensibility. New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- Work with Python. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

- TensorFlow:

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google often replacing its closed-source predecessor, DistBelief.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open source license on November 9, 2015.

Features

TensorFlow provides official Python API and C API; and without API stability guarantee: C++, Go, and Java. Third party packages are available for C#, Haskell, Julia, R, Scala, Rust, and OCaml.

A "WebGL accelerated, browser based JavaScript library for training and deploying ML models" (where "for inference, TensorFlow.js with WebGL is 1.5-2x slower than TensorFlow Python with AVX. For training, we have seen small models train faster in the browser and large models train up to 10-15x slower in the browser") was released by Tensorflow.org on March 30, 2018. They also have with a note on "Swift for TensorFlow is an early stage research project. It has been released to enable open source development and is not yet ready for general use by machine learning developers."

Applications

Among the applications for which TensorFlow is the foundation, are automated image captioning software, such as DeepDream. RankBrain now handles a substantial number of search queries, replacing and supplementing traditional static algorithm-based search results.

K- NEAREST NEIGHBOR ALGORITHM:

Introduction

Nowadays money investment in stock market gains major attention because of its dynamic nature. So the significant issue in market finance is discovering well organized approaches to outline and envision the stock market information to provide individuals or organizations helpful data about the behavior of the market for making decision about investment.The huge amount of important information produced by the stock market has attracted researchers to investigate this issue utilizing distinctive approaches. Since stock markets produce huge datasets it data mining techniques is found to be more efficient.Data mining is utilized for excavate data from databases and discover the meaningful patterns from the database. The usefulness of this data makes data mining imperative and necessary.The essentials of data mining in finance are originating from the need to adopt specific well organized criteria to predict exactness, facilitate multi-resolution calculation.

k-Nearest Neighbor Classifier (*k*NN)

K-nearest neighbor technique is a machine learning algorithm that is considered as simple to implement (Aha et al. 1991). The stock prediction problem can be mapped into a similarity based classification. The historical stock data and the test data is mapped into a set of vectors. Each vector represents *N* dimension for each stock features. Then, a similarity metric such as Euclidean distance is computed to take a decision. In this section, a description of *k*NN is provided. *k*NN is considered a lazy learning that does not build a model or function previously, but yields the closest *k* records of the training data set that have the highest similarity to the test (i.e. query record). Then, a majority vote is performed among the selected k records to determine the class label and then assigned it to the query record.

The prediction of stock market closing price is computed using *k*NN as follows:

1 Determine the number of nearest neighbors, *k*.
2 Compute the distance between the training samples and the query record.
c) Sort all training records according to the distance values.
d)Use a majority vote for the class labels of k nearest neighbors, and assign it as a prediction value of the query record.

Basics of KNN

The KNN is the principal and most straightforward classification technique when the information about the distribution of the data is insufficient. This convention basically holds the whole training set during learning and allocates to every query a class characterize by the majority label of its k-nearest neighbors in the training set. The Nearest Neighbor (NN) principle is the least complex type of KNN when K = 1.

In this algorithm every training samples ought to be grouped to its samples surrounded by it. Subsequently, if the classification of any of the sample data is obscure, then it could be anticipated by considering the classification of its nearest neighbor tests. Given an obscure sample and a training set consisting of samples, all the distances between the obscure sample and the entire sample in the training set can be calculated by utilizing the accompanying mathematical statement where, x_1, x_2, x_3,x_p are anticipators of the first sample and u_1, u_2,u_3,… u_p are anticipators of the second sample. If distance is of smallest value, then the samples in the training set is close to the obscure sample. Hence, the obscure sample may be categorized based on this nearest neighbor classification.

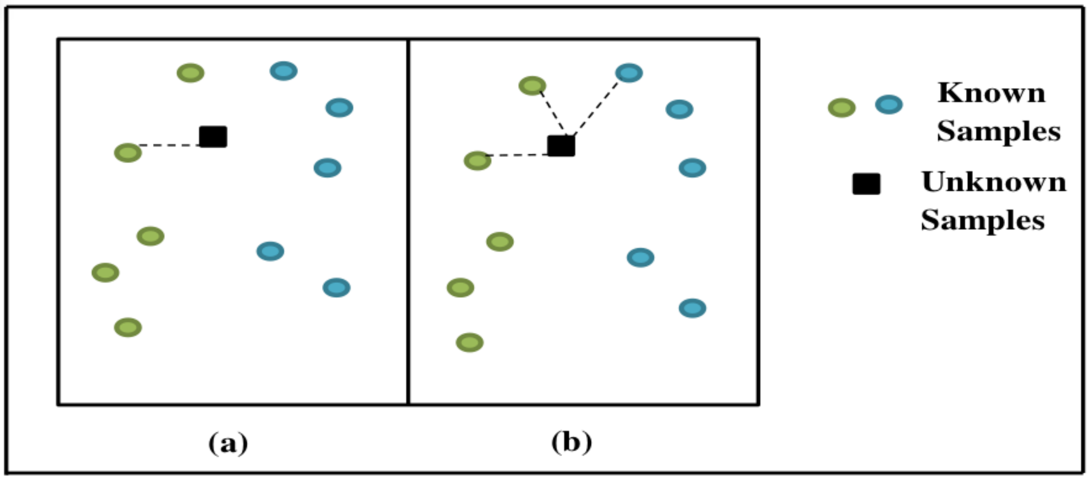


Fig 4.1 KNN decision rule

Fig 4.1 illustrates the KNN decision rule for K= 1 and K= 3 for a set of samples divided into 2 classes.In Fig 4.1(a), an obscure sample (unknown sample) is categorized by using only one known sample; In Fig 4.1(b) more than one known sample is used. In the last case, the parameter K is set to 3, hence the closest three samples is considered for classifying the obscure one. Two of them belong to the same class, whereas only one belongs to the other class. In both cases, the unknown sample is classified as belonging to the class on the left. Fig 4.2 shows the pseudo code for the KNN algorithm

Pseudo code for KNN algorithm

```
Input: Finite set A , Finite Set B, k, function c:B->{1,2,….n}

Output: r:A->{1,2,…..n}

Begin

For each x in A do

Let L<- {}

For each b in B add (a(x,b), c(b)) to L

Sort the elements in L with the first components

Compute the class labels from the first k elements from L

Let r(x) be the class containing highest number of occurrences

End

Return r

End
```

The classifier performance is principally controlled by the decision of K and in addition the distance metric applied [20-25]. This evaluation is influenced by the sensitivity of the choosing the neighborhood size K, since local region radius is calculated by the Kth nearest neighbor distance to the query and diverse value of K yields various conditional class probabilities.

Mathematical Calculations and Visualizations Models

This represents an overview of equations that were applied in this article for predicting next day price. The calculations includes error estimation, total sum of squared error, average error, cumulative closing price when sorted using predicted values, k-values and training Root Mean Square (RMS) errors.

a) Root Mean Square Deviation (RMSD) is accuracy metric that computes the differences between the estimated values, Y, and the actual values, X. The total of RMSD is aggregated into a single value measure. RMSD = SQRT$(Y-X)^2$.
b) Explained Sum of Squares (ESS) is computed as follows:

   ESS = $\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$.

   Where y_i: is the predicted variable, and *y is* the actual value.
c) Average Estimated Error (AEE)

   AEE is the total sum of RMS errors for all variables in stock records divided by the total number of the records.

   AEE = $\frac{\sum_{i=1}^{n} RMSi}{n}$

Visualization Graph

To evaluate the performance of *k*NN learning model, lift graph is applied and drawn for different companies' stock values. The lift chart symbolizes the enhancement that a data mining model offers when distinguished against a random estimation, and the change is expressed in terms of lift score. Through contrasting the lift scores for a variety of parts of the data set and for different models, it can then be decided which model is supreme and which percentage of the cases within the data set would gain from employing the predictions model.

Furthermore, using the lift chart assist in distinguishing how accurate predictions are for various models with identical predictable characteristic. The lift graph also shows the ratio between the results obtained using the predictive model or not. The other graph applied is the plot curves to show the relation between the actual and predicted stock price.

Prediction Performance Evaluations

Table 6 represents a summary of the total squared errors, RMS errors and the average errors for the five companies. The residuals offer the differences between the predicted values and actual the values in the sample data. The table also shows that the values of errors are very small which indicate that the actual value and predicted value are close. This yields a high accuracy of using the *k*NN algorithm in predicting stock values.

.

Non-Linear Regression Results

Non-linear regression is a data analysis technique in which the observed data is incorporated into a model presented in a mathematical non-linear function combining the model parameters that relies on independent variables used. GraphPad Prism v5.02 software was used to apply centered second order polynomial (quadratic) non-linear regression which has the following formula:

$$\mathit{Price} = B_0 + B_1\,(\mathit{day} - \mathit{mean}\,(\mathit{day})) + B_2\,(\mathit{day} - \mathit{mean}\,(\mathit{day}))^2$$

Where:

B0, B1 and ***B2****:*	Constants.
Day*:*	Actual day in which we will predict the price.
Price*:*	Predicted price depending on the day.

- **Kalman Filter:**

In 1960, R. E. Kalman introduced a recursive algorithm to solve the linear filtering and prediction problem using a state-space approach. The Kalman filter is a linear, discrete-time system which provides a recursive solution to a set of difference equations. The recursive nature of the Kalman filter requires only the previous values of the state vector to be retained to produce future estimates. This recursive algorithm makes the Kalman filter useful for real-time applications. The state space format makes it easy to implement the Kalman filter on a digital computer. The Kalman filter provides the optimum estimate in a least squares sense of a random process which is being sampled with noisy measurements.

The Kalman filter can be used to "filter" the best estimate or it can be used to forecast future values of the random process. The Kalman filter models a process as the output of white noise passing through a linear system. The states are selected such that the filter output is formed from the linear combination of the states. A Kalman filter can also be used to model non-stationary processes if a linear differential equation relating the process to white noise can be determined. If the model parameters are time-varying, an adaptive Kalman filter can often be used to estimate the non-stationary process.

Algorithm:

The Kalman filter is based on a discrete state space approach where the random process is modeled by a state equation and a measurement equation .

Xk+1 = «kXk + Hk
zk = HkSk + vk

For a process having a single noisy output and modeled using n internal states and m white noise inputs, x is the n-dimensional state vector, w is the m-dimensional white noise input vector, z is the noisy output measurement, and v is the additive measurement noise. For the single output system, *z* and v are both scalars. The other parameters in the state description are the state transition matrix, and the connection vector, H. The (nxn) state transition matrix describes the change in the states from tjj to tj^+i when there are no driving functions, i.e., w = 0. The n-dimensional connection vector describes the linear combination of states which comprise the output. The process and measurement noise parameters, w and v, respectively, are uncorrelated white Gaussian sequences with zero mean and variances (covariances) defined by:
E [wi*wjjT] r Qj^ i=k
0 *ijlls.*
E [vi*vk] = Rjj. i=k

0 i/k
E [wi*vij] = 0 for all i and k (3-4)
The values of Q and R are calculated prior to execution of the Kalman filter. Each iteration of the Kalman filter is started with an a priori estimate, x"k, which is the expected value of the
state just before assimilating the measurement. The estimation error, e'jj, between the actual state, x^» and the a priori state estimate, x~je is defined by (3-5). S"k = ak - A"k (3-5)
The estimation error is assumed to have zero mean and a covariance matrix, defined as p-k = E[e-k g-k?] = EC(xk - S-k)(2Sk - x-k)T] (3-6) The P~1j matrix describes the confidence level of the a priori state estimate accuracy. After the current measurement, z^, the a priori state
estimate is updated to incorporate the measurement data. The a posteriori estimate, xjj, is defined by the following update equation (3-7), Ak - A"k + Ek(zk - HkÉ"k) (3-7)
where is the Kalman gain vector at time, t^. The n-dimensional Kalman gain vector contains the weighting factors used to combine the new measurement with the a priori estimate to achieve an optimal a posteriori estimate. An optimum estimate minimizes the mean-square error of the updated estimate. The Kalman gain vector which produces an optimal estimate takes into account the confidence in the a priori estimate, P"k, and the reliability of the
measurement, Rk- The Kalman gain is given by (3-8).
Sk = P-kakT(HkP-kakT + Bk)-' (3-8)
With a scalar measurement, the inversion in the Kalman gain is just a scalar inversion.
The error covariance matrix for the a posteriori state estimate is calculated from
Pk = (I - KkHk)P"k (3-9)
where I is an (n x n) identity matrix.
At this point, an updated state estimate and its error covariance matrix have been calculated for the measurement at step k. To prepare for the next iteration of the Kalman filter, an a priori state estimate, x"k+i, and an a priori error covariance matrix, P~k+1> must be projected ahead from their a posteriori estimates. *~k+l for the next measurement can be estimated by taking the expected value of the state equation (3-la). Since the expected value
of wjj is zero, the a priori estimate becomes *"k+l = ®k*k • (3-10) The a priori error covariance matrix is projected ahead by
P"k+1 = ^k^k^k"^ + Ok- (3-11)
The recursive Kalman filter algorithm consists of the
Kalman gain equation (3-8), state estimate (3-7) and error Govariance (3-9) update equations, and state estimate (3-10)

and error covariance (3-11) projection equations. Initially, the Kalman filter must be provided with an estimate of the state vector, xq", and its error covariance matrix, Pq".

a block diagram of the Kalman filter" algorithm is shown in Figure 3-1.

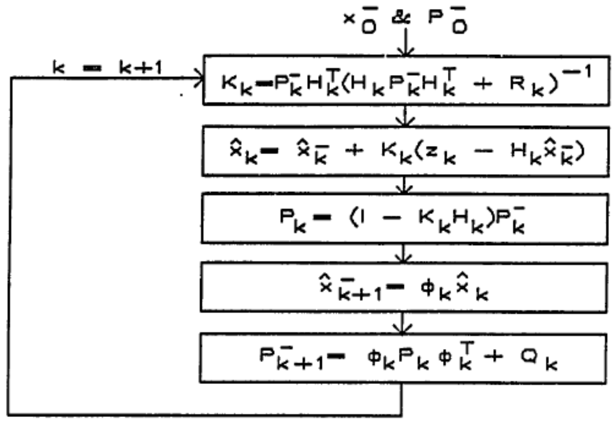

x_0^- & P_0^-
$k = k+1$
$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$
$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-)$
$P_k = (I - K_k H_k) P_k^-$
$\hat{x}_{k+1}^- = \phi_k \hat{x}_k$
$P_{k+1}^- = \phi_k P_k \phi_k^T + Q_k$

Figure 3-1. Block diagram of the Kalman filter algorithm

The Kalman filter can also provide multiple step ahead forecasts. The N-step ahead forecast equation is x"k+N = @k+N,kAk (3-12) where @k+N,k is the N-step ahead transition matrix. This forecast equation is kept separate from the recursive Kalman filter algorithm.

3. **Objective of the Project:**

i. Our project presents a computational approach for predicting the S&P CNX Nifty 50 Index.

ii. An AI based model has been used in predicting the direction of the movement of the closing value of the index.

iii. The model presented in the project also confirms that it can be used to predict price index value of the stock market.

iv. After studying the various features of the AI model, an optimal model is proposed for the purpose of forecasting.

v. The model has used the pre-processed data set of closing value of S&P CNX Nifty 50 Index.

vi. The data set encompassed the trading days of a single company for the initial development of the model. Work with a larger data set is planned in further progress of the model.

vii. Accuracy of the performance of the AI model is compared using various out of sample performance measures.

viii. This project examines the use of the Kalman filter to forecast intraday stock and commodity prices.

ix. The price forecasts are based on a market's price history with no external information included.

x. For the Kalman filter to produce beneficial forecasts, the market must not be a random walk process, but must exhibit a statistically significant auto correlation pattern which can be modeled.

xi. Once an appropriate Kalman filter model is determined, strategies for increasing profits can be studied.

4. System Design:

- Hardware requirements:
  - GPU: 4 x NVIDIA PASCAL GTX 1080Tis **11 GB**
  - CPU: Intel **3.8 GHz** Core i7-6850K
  - RAM: **64 GB** DDR4-2666 System Memory
  - Hard Disk Space: **1 TB** SATA SSD for OS + **3 TB** 7200 rpm HDD for Long-term Data Storage.
  - Cooling: Air cooling / Water cooling.
  - Power Supply: 1400 to 1600 watts.

- Software requirements:

- Anaconda Navigator
- Jupyter Notebook
- Python 3 and different Python libraries, Deep Learning Libraries and Sentiment analysis libraries such as:

  TensorFlow

  Keras

  scikit-learn

  numpy

  scipy

  nsepy

  pykalman

  pandas

  etc.

- Ubuntu 16.04 LTS OS
- Text editor

5. Methodology for implementation (Formulation/Algorithm):

Support Vector Regression:

SVR Linear :

We are reading the dataset and putting them into an array for further operations to be performed. We are interested in the closing price and the dates. So we read those values accordingly.After we have our desired array we can start our operations. We are putting our values through SVR Linear Classifier as explained in the Review of Literature. Then we are getting our desired values. We are then plotting a graph of it with the help of Matplotlib.

SVR Polynomial:

We are reading the dataset and putting them into an array for further operations to be performed. We are interested in the closing price and the dates. So we read those values accordingly.After we have our desired array we can start our operations. We are putting our values through SVR Polynomial Classifier as explained in the Review of Literature. Then we are getting our desired values. We are then plotting a graph of it with the help of Matplotlib.

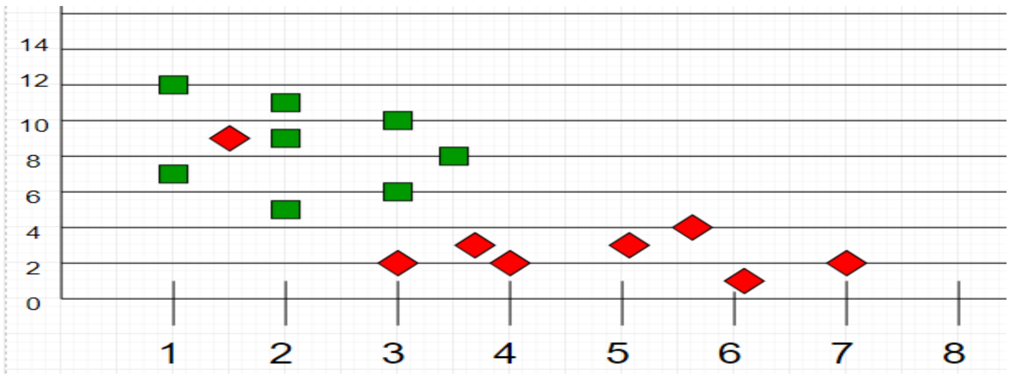
SVR RBF:

We are reading the dataset and putting them into an array for further operations to be performed. We are interested in the closing price and the dates. So we read those values accordingly.After we have our desired array we can start our operations. We are putting our values through SVR RBF Classifier as explained in the Review of Literature. Then we are getting our desired values. We are then plotting a graph of it with the help of Matplotlib.

KNN METHODOLOGY:

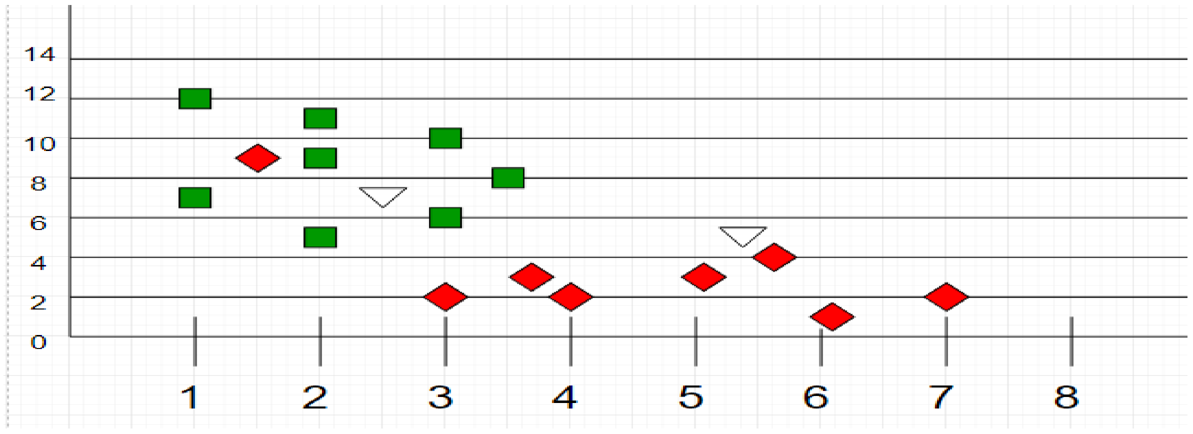
K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition.
A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function. If K = 1, then the case is simply assigned to the class of its nearest neighbor.
Choosing the optimal value for K is best done by first inspecting the data. In general, a large K value is more precise as it reduces the overall noise but there is no guarantee. Cross-validation is another way to retrospectively determine a good K value by using an independent dataset to validate the K value. Historically, the optimal K for most datasets has been between 3-10. That produces much better results than 1NN..
As an example, consider the following table of data points containing two features:Now, given another set of data points (also called testing data), allocate

these points a group by analyzing the training set. Note that the unclassified points are marked as 'yellow'.



Intuition

If we plot these points on a graph, we may be able to locate some clusters, or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbours belong to. This means, a point close to a cluster of points classified as 'Red' has a higher probability of getting classified as 'Red'.



Intuitively, we can see that the first point (2.5, 7) should be classified as 'Blue' and the second point (5.5, 4.5) should be classified as 'Red'.

ALGORITHM:

1. Let m be the number of training data samples. Let p be an unknown point.

2. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).

3. for i=0 to m:

4. Calculate Euclidean distance d(arr[i], p).

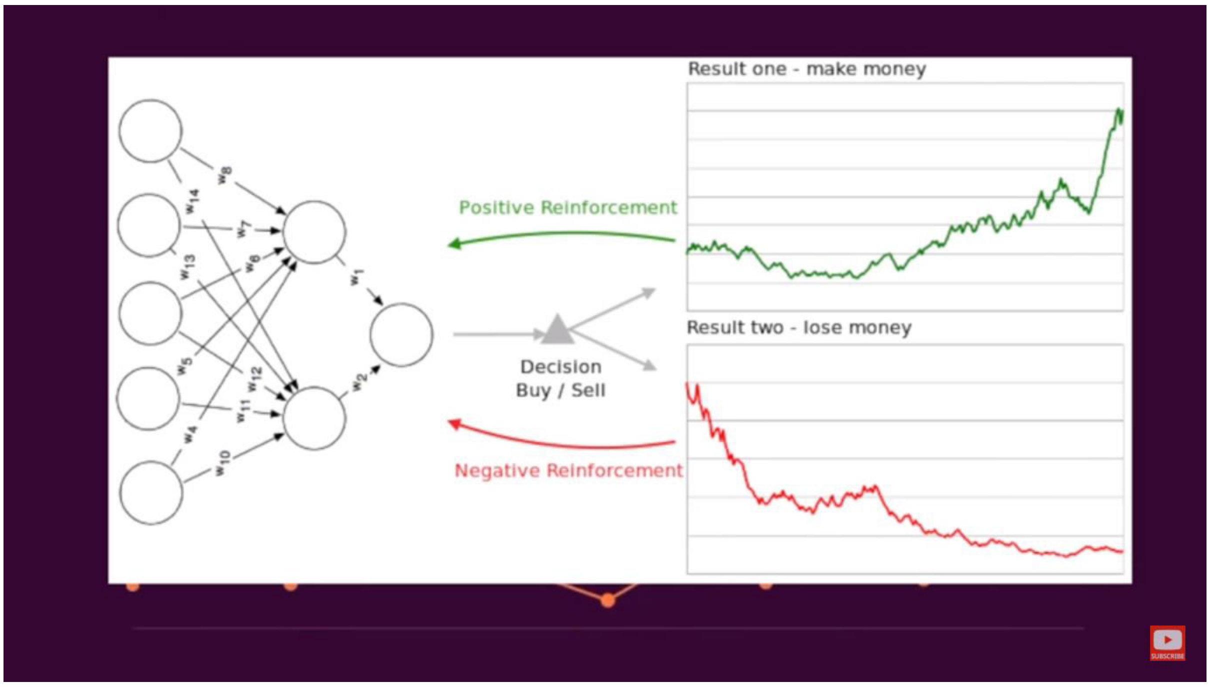
5. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.

6. Return the majority label among S.

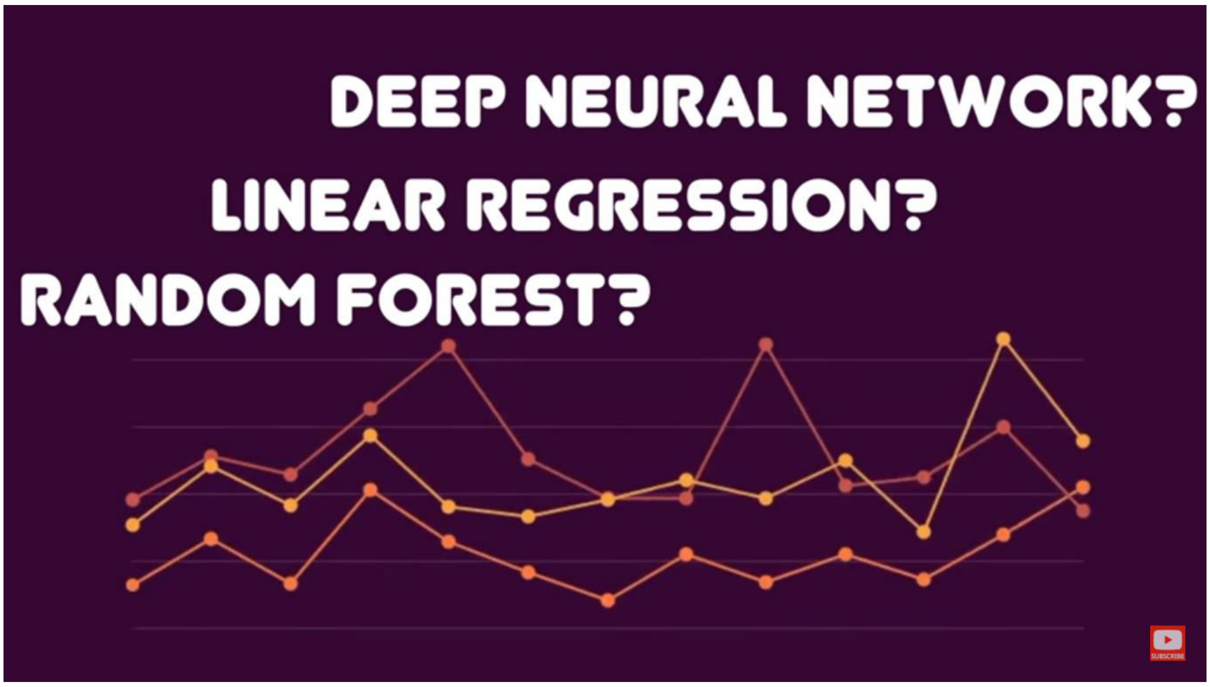
K can be kept as an odd number so that we can calculate a clear majority in the case where only two groups are possible (e.g. Red/Blue). With increasing K, we get smoother, more defined boundaries across different classifications. Also, the accuracy of the above classifier increases as we increase the number of data points in the training set.

Reinforcement Learning and Sentiment Analysis(Random Forest Model, Linear Regression Model, Multi Layer Perceptron or Deep Neural Network Model):

In Reinforcement Learning, a model can learn from its past predictive ability, giving itself a reward only for good behaviour, essentially self improving over time:



Reinforcement Learning



We'll use the Sentiment from new headlines and historical price charts together to predict future prices in Python, i.e., **Sentiment Analysis + Reinforcement Learning:**

We'll use two Machine Learning libraries for our problem:

The first is called **NLTK**(Natural Language Tool Kit) and the second is **Scikit-Learn.**

The data set we're using is the adjusted closing price gathered from the past 10 years for Microsoft stock. We'll use 8 of these years for training and 2 of these years for testing as well as a data set of the NYT articles' headlines about Microsoft for Sentiment Analysis:

MS-Data.csv:

```
Date,Open,High,Low,Close,Volume,Adj Close

2016-12-30,19833.169922,19852.550781,19718.669922,19762.599609,271910000,19762.599609

2016-12-29,19835.460938,19878.439453,19788.939453,19819.779297,172040000,19819.779297
```

```
2016-12-
28,19964.310547,19981.109375,19827.310547,19833.679688,188350000,1983
3.679688

2016-12-
27,19943.460938,19980.240234,19939.800781,19945.039062,158540000,1994
5.039062

2016-12-
23,19908.609375,19934.150391,19899.060547,19933.810547,158260000,1993
3.810547

2016-12-
22,19922.679688,19933.830078,19882.189453,19918.880859,258290000,1991
8.880859

2016-12-
21,19968.970703,19986.560547,19941.960938,19941.960938,256640000,1994
1.960938

2016-12-
20,19920.589844,19987.630859,19920.419922,19974.619141,284080000,1997
4.619141

2016-12-
19,19836.660156,19917.779297,19832.949219,19883.060547,302310000,1988
3.060547

2016-12-
16,19909.009766,19923.169922,19821.00,19843.410156,573470000,19843.41
0156

2016-12-
15,19811.50,19951.289062,19811.50,19852.240234,357350000,19852.240234

2016-12-
14,19876.130859,19966.429688,19748.669922,19792.529297,408430000,1979
2.529297

2016-12-
13,19852.210938,19953.75,19846.449219,19911.210938,388420000,19911.21
0938

…………………………………………………......
```

……………………………………………………………………

……………………………………………………………………. . . . .

NYT-Headlines-Data.pkl :

€cpandas.core.frame

DataFrame

q)• q}q————————————(U
_metadataq

]qU

_typqU dataframeqU_dataqcpandas.core.internals

BlockManager

q)• q

(]q
(cpandas.indexes.base

_new_Index

qcpandas.indexes.base

Index

q
}q

(U

---

dataqcnumpy.core.multiarray

_reconstruct

qcnumpy

ndarray

qK …Ub‡Rq(KK————————————…cnumpy

dtype

qUO8K
K‡Rq(K————————————U|NNNJÿÿÿÿJÿÿÿÿK?tb‰]q(UcloseqU adj closeqUarticlesqetbU

---

nameqNu†Rqcpandas.tseries.index

_new_DatetimeIndex

qcpandas.tseries.index

DatetimeIndex

q}q(Utzq-NU

---

freqqcpandas.tseries.offsets

Day

q)• q!}q"(Unormalizeq#‰U_offsetq$cdatetime

timedelta

••

••

•••

We've used **pandas data processing tool** to combine both datasets into one dataframe:

# Reading the saved data pickle file

```
df_stocks = pd.read_pickle('/Users/Dinesh/Documents/Project Stock predictions/data/pickled_ten_year_filtered_data.pkl')

df_stocks['prices'] = df_stocks['adj close'].apply(np.int64)

# selecting the prices and articles

df_stocks = df_stocks[['prices', 'articles']]

df_stocks['articles'] = df_stocks['articles'].map(lambda x: x.lstrip('.-'))
```

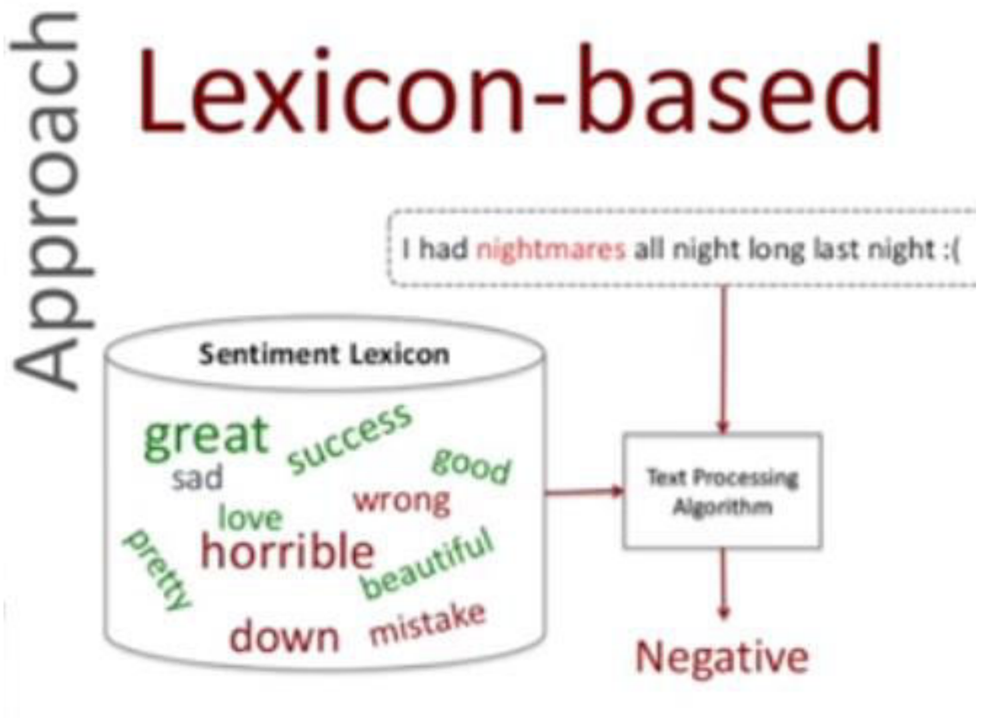
We are next going to perform Sentiment Analysis on these headlines from the NYT. So, we use the sentiment intensity analyzer :

```
# Adding new columns to the data frame

df["compound"] = ''

df["neg"] = ''

df["neu"] = ''

df["pos"] = ''
```

This will output sentiment scores for four classes of sentiments: Negative, Neural, Positive and Compound, which is the aggregated score.:

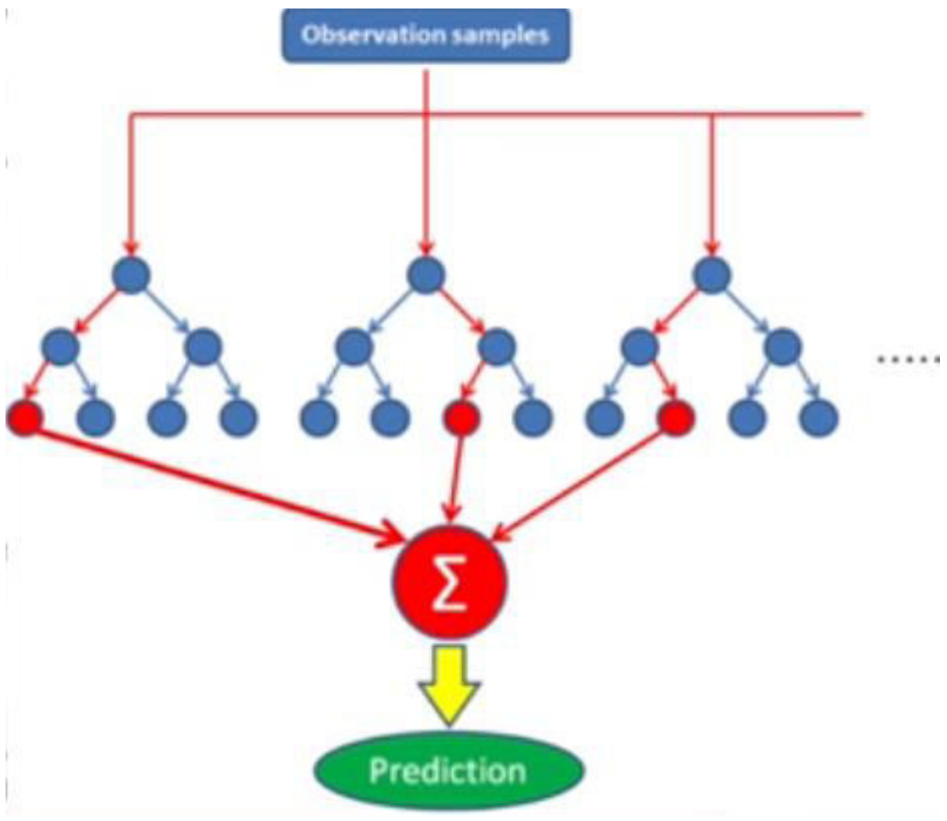
```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

import unicodedata

sid = SentimentIntensityAnalyzer()

for date, row in df_stocks.T.iteritems():

   try:
        sentence = unicodedata.normalize('NFKD', df_stocks.loc[date,
'articles']).encode('ascii','ignore')

        ss = sid.polarity_scores(sentence)

        df.set_value(date, 'compound', ss['compound'])

        df.set_value(date, 'neg', ss['neg'])

        df.set_value(date, 'neu', ss['neu'])

        df.set_value(date, 'pos', ss['pos'])

    except TypeError:

        print df_stocks.loc[date, 'articles']

        print date
```

There are actually several popular sentiment lexicons out there. They are made manually by humans and pattern recognition algorithms use them to summarize the polarities of entire documents:



We'll first try out a Random Forest Model:

```
from treeinterpreter import treeinterpreter as ti

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import classification_report,confusion_matrix


rf = RandomForestRegressor()

rf.fit(numpy_df_train, y_train)
```



Random Forest Model

The second model we use in Linear Regression which will draw the line of best fit between our variables:

```
from treeinterpreter import treeinterpreter as ti

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.linear_model import LogisticRegression

from datetime import datetime, timedelta



# average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']

# # Converting string to date time

# temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()

# # Adding one day from date time

# difference = temp_date + timedelta(days=1)

# # Converting again date time to string

# temp_date = difference.strftime('%Y-%m-%d')



# start_year = datetime.strptime(train_start_date, "%Y-%m-
%d").date().month



years = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016]

prediction_list = []

for year in years:
```

```
# Splitting the training and testing data

train_start_date = str(year) + '-01-01'

train_end_date = str(year) + '-10-31'

test_start_date = str(year) + '-11-01'

test_end_date = str(year) + '-12-31'

train = df.ix[train_start_date : train_end_date]

test = df.ix[test_start_date:test_end_date]



# Calculating the sentiment score

sentiment_score_list = []

for date, row in train.T.iteritems():

    sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date, 
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])

    #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])

    sentiment_score_list.append(sentiment_score)

numpy_df_train = np.asarray(sentiment_score_list)

sentiment_score_list = []

for date, row in test.T.iteritems():

    sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date, 
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])

    #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])

    sentiment_score_list.append(sentiment_score)

numpy_df_test = np.asarray(sentiment_score_list)
```

```
# Generating models

lr = LogisticRegression()

lr.fit(numpy_df_train, train['prices'])




prediction = lr.predict(numpy_df_test)

prediction_list.append(prediction)

#print train_start_date + ' ' + train_end_date + ' ' + test_start_date + ' '
+ test_end_date

idx = pd.date_range(test_start_date, test_end_date)

#print year

predictions_df_list = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])



difference_test_predicted_prices = offset_value(test_start_date, test,
predictions_df_list)

# Adding offset to all the advpredictions_df price values

predictions_df_list['prices'] = predictions_df_list['prices'] +
difference_test_predicted_prices

predictions_df_list



# Smoothing the plot

predictions_df_list['ewma'] = pd.ewma(predictions_df_list["prices"],
span=10, freq="D")

predictions_df_list['actual_value'] = test['prices']
```

```
    predictions_df_list['actual_value_ewma'] = 
pd.ewma(predictions_df_list["actual_value"], span=10, freq="D")

    # Changing column names

    predictions_df_list.columns = ['predicted_price', 
'average_predicted_price', 'actual_price', 'average_actual_price']

    predictions_df_list.plot()

    predictions_df_list_average = 
predictions_df_list[['average_predicted_price', 'average_actual_price']]

    predictions_df_list_average.plot()



#     predictions_df_list.show()
```

The third model we use is a Multi Layer Perceptron(MLP) classifier, also called a Neural Network:

```
from sklearn.neural_network import MLPClassifier

from datetime import datetime, timedelta



# average_upcoming_5_days_predicted += predictions_df.loc[temp_date, 
'prices']

# # Converting string to date time

# temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()

# # Adding one day from date time

# difference = temp_date + timedelta(days=1)

# # Converting again date time to string
# temp_date = difference.strftime('%Y-%m-%d')


# start_year = datetime.strptime(train_start_date, "%Y-%m-
%d").date().month


years = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016]

prediction_list = []

for year in years:

    # Splitting the training and testing data

    train_start_date = str(year) + '-01-01'

    train_end_date = str(year) + '-10-31'

    test_start_date = str(year) + '-11-01'

    test_end_date = str(year) + '-12-31'

    train = df.ix[train_start_date : train_end_date]

    test = df.ix[test_start_date:test_end_date]


    # Calculating the sentiment score

    sentiment_score_list = []

    for date, row in train.T.iteritems():

        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])

        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])

        sentiment_score_list.append(sentiment_score)

    numpy_df_train = np.asarray(sentiment_score_list)
    sentiment_score_list = []

    for date, row in test.T.iteritems():

        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])

        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])

        sentiment_score_list.append(sentiment_score)

    numpy_df_test = np.asarray(sentiment_score_list)



    # Generating models

    mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 100),
activation='relu',

                         solver='lbfgs', alpha=0.005, learning_rate_init = 0.001,
shuffle=False) # span = 20 # best 1

    mlpc.fit(numpy_df_train, train['prices'])

    prediction = mlpc.predict(numpy_df_test)



    prediction_list.append(prediction)

    #print train_start_date + ' ' + train_end_date + ' ' + test_start_date + ' '
+ test_end_date

    idx = pd.date_range(test_start_date, test_end_date)

    #print year

    predictions_df_list = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])



    difference_test_predicted_prices = offset_value(test_start_date, test,
predictions_df_list)
```

```
    # Adding offset to all the advpredictions_df price values

    predictions_df_list['prices'] = predictions_df_list['prices'] +
difference_test_predicted_prices

    predictions_df_list



    # Smoothing the plot

    predictions_df_list['ewma'] = pd.ewma(predictions_df_list["prices"],
span=20, freq="D")

    predictions_df_list['actual_value'] = test['prices']

    predictions_df_list['actual_value_ewma'] =
pd.ewma(predictions_df_list["actual_value"], span=20, freq="D")

    # Changing column names

    predictions_df_list.columns = ['predicted_price',
'average_predicted_price', 'actual_price', 'average_actual_price']

    predictions_df_list.plot()

    predictions_df_list_average =
predictions_df_list[['average_predicted_price', 'average_actual_price']]

    predictions_df_list_average.plot()



#    predictions_df_list.show()
```

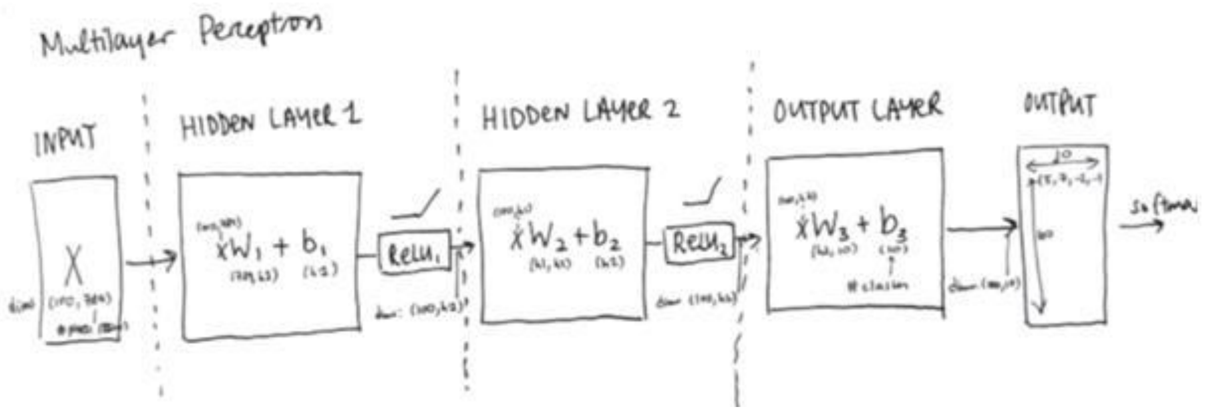
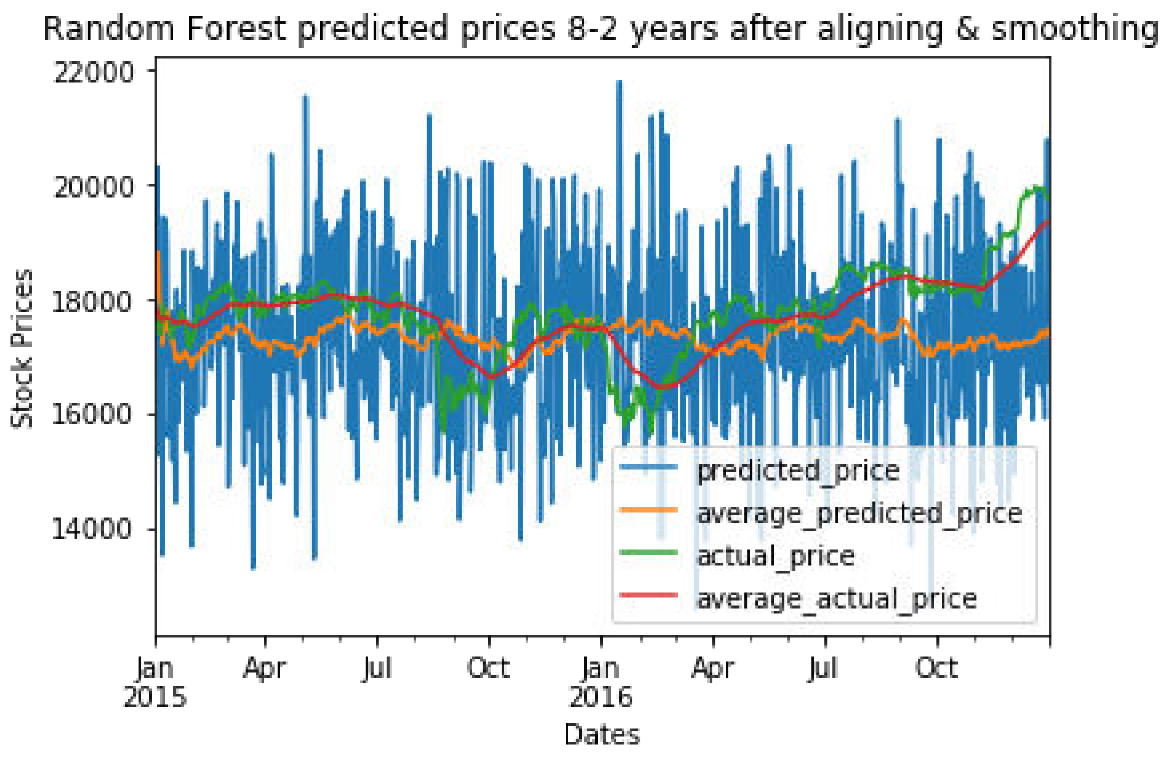
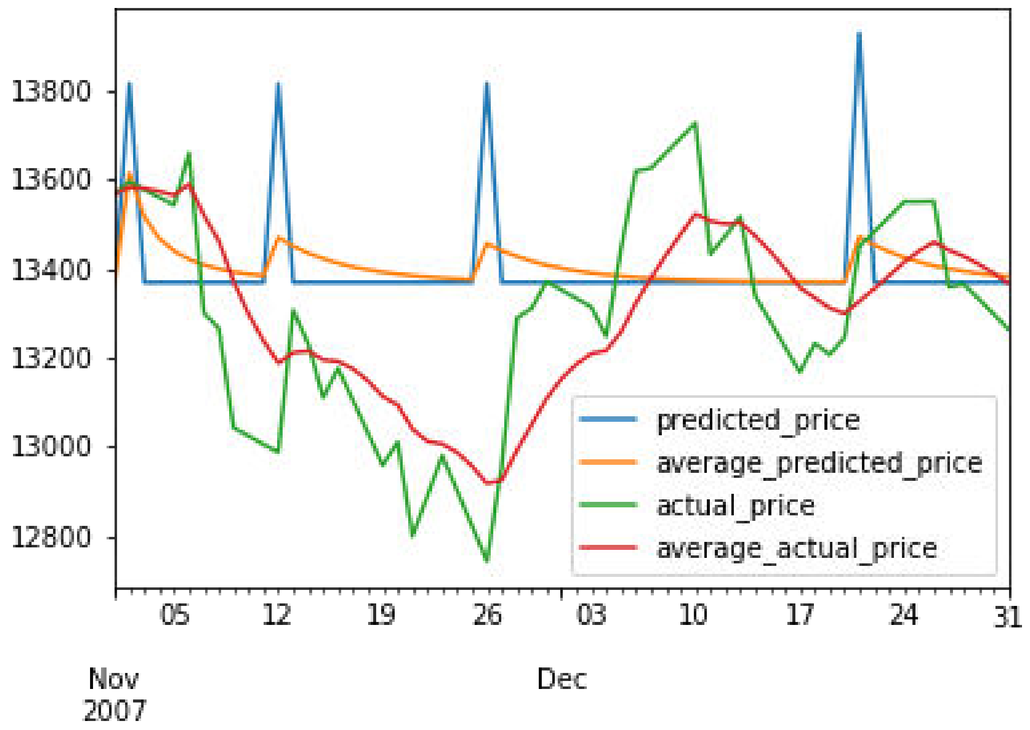


Fig.: Multi Layer Perceptron Or Deep Neural Network

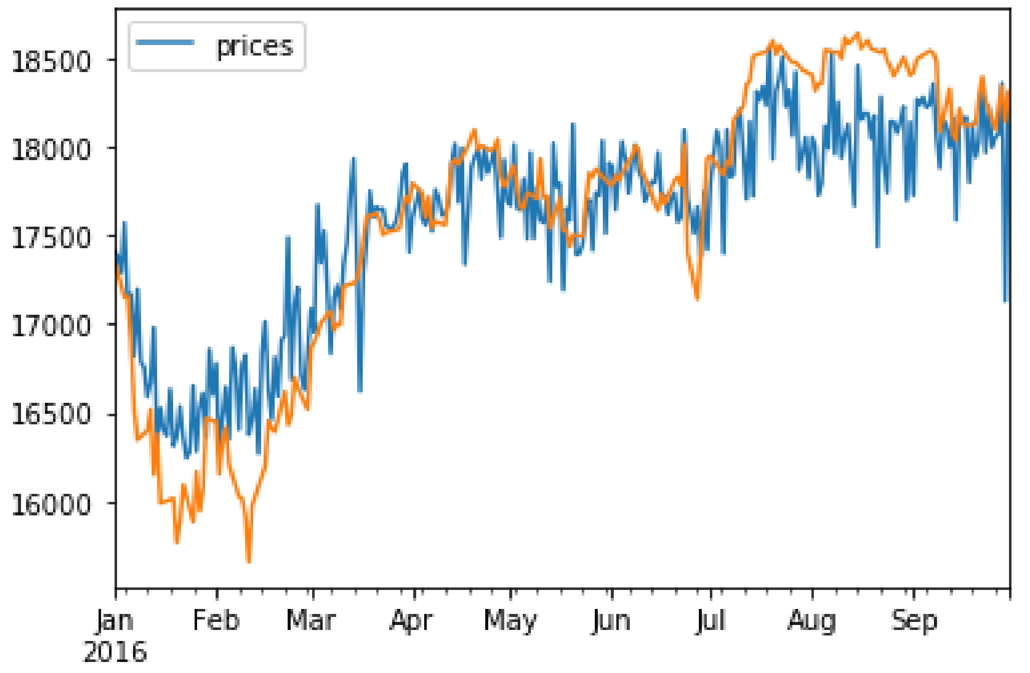
We'll input our data on the three initialized models and observe the results in graph for each of them:



As we can see above, the Random Forest Model doesn't look nice, it's pretty off.



The Linear Regression Model looks a little bit better, but it is still pretty bad.



But the MLP Classifier looks the best of all of them.

Deep Learning Model:

Here we'll build a Deep Learning model to predict stock prices using Keras with TensorFlow backend. For our training data, we'll be using the daily closing price of the **S&P 500** from January 2000 to August 2016.

sp500.csv :

```
1455.219971

1399.420044

1402.109985

1403.449951

1441.469971

1457.599976

1438.560059

1432.25

1449.680054

1465.150024

1455.140015

1455.900024

1445.569946

1441.359985

1401.530029

1410.030029

1404.089966
```

1398.560059

1360.160034

1394.459961

1409.280029

1409.119995

1424.969971

.

.

.

.

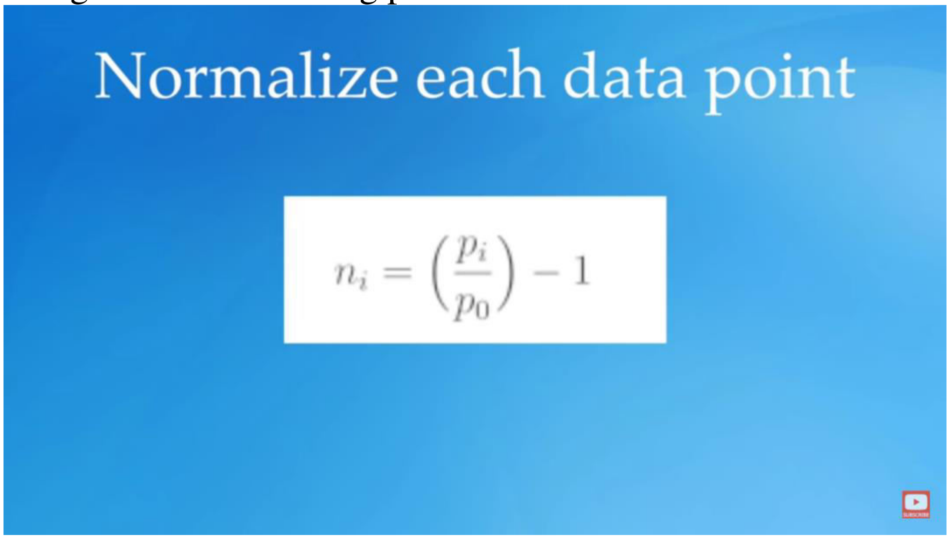
This is a series of data points indexed in time-order or a time series. Our goal will be to predict the closing price for any given day after training.

We can load our data using a custom load_data() function:

X_train, y_train, X_test, y_test = lstm.load_data('sp500.csv', 50, True)

It essestially just reads our **.csv** file into an array of values and normalizes them. Rather than feeding those values directly to our models, normalizing them improves **convergence**.

We use the following equation to normalize each value to reflect percentage changes from the starting point:



Normalize each data point

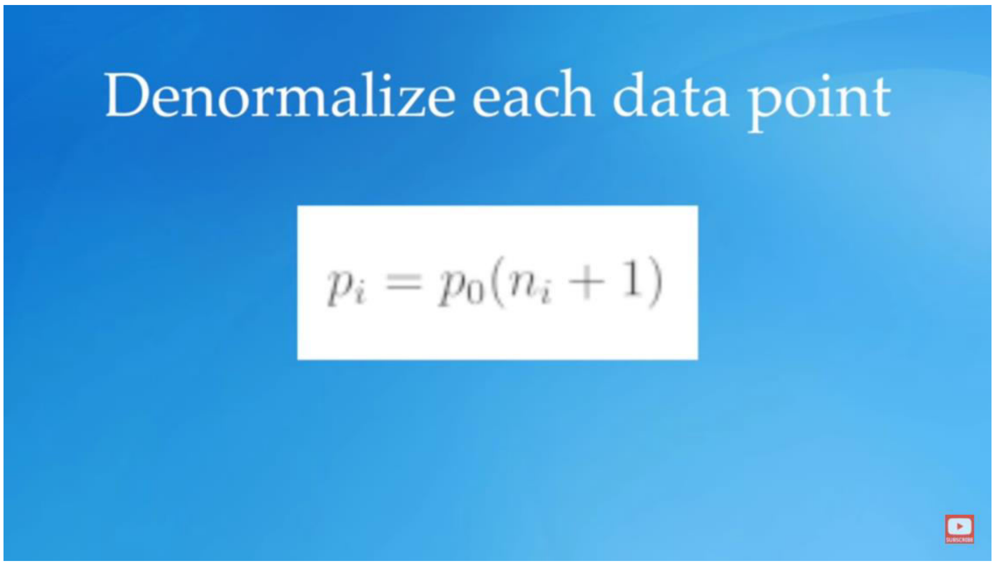
$$n_i = \left(\frac{p_i}{p_0}\right) - 1$$

Where, pi = each price

P0 = initial price

Ni = normalized each price

So, we divide each price by the initial price and subtract 1 to get the normalized price. When our model later makes prediction, we'll denormalize the data using the following formula to get a real world number out of it:



Denormalize each data point

$$p_i = p_0(n_i + 1)$$

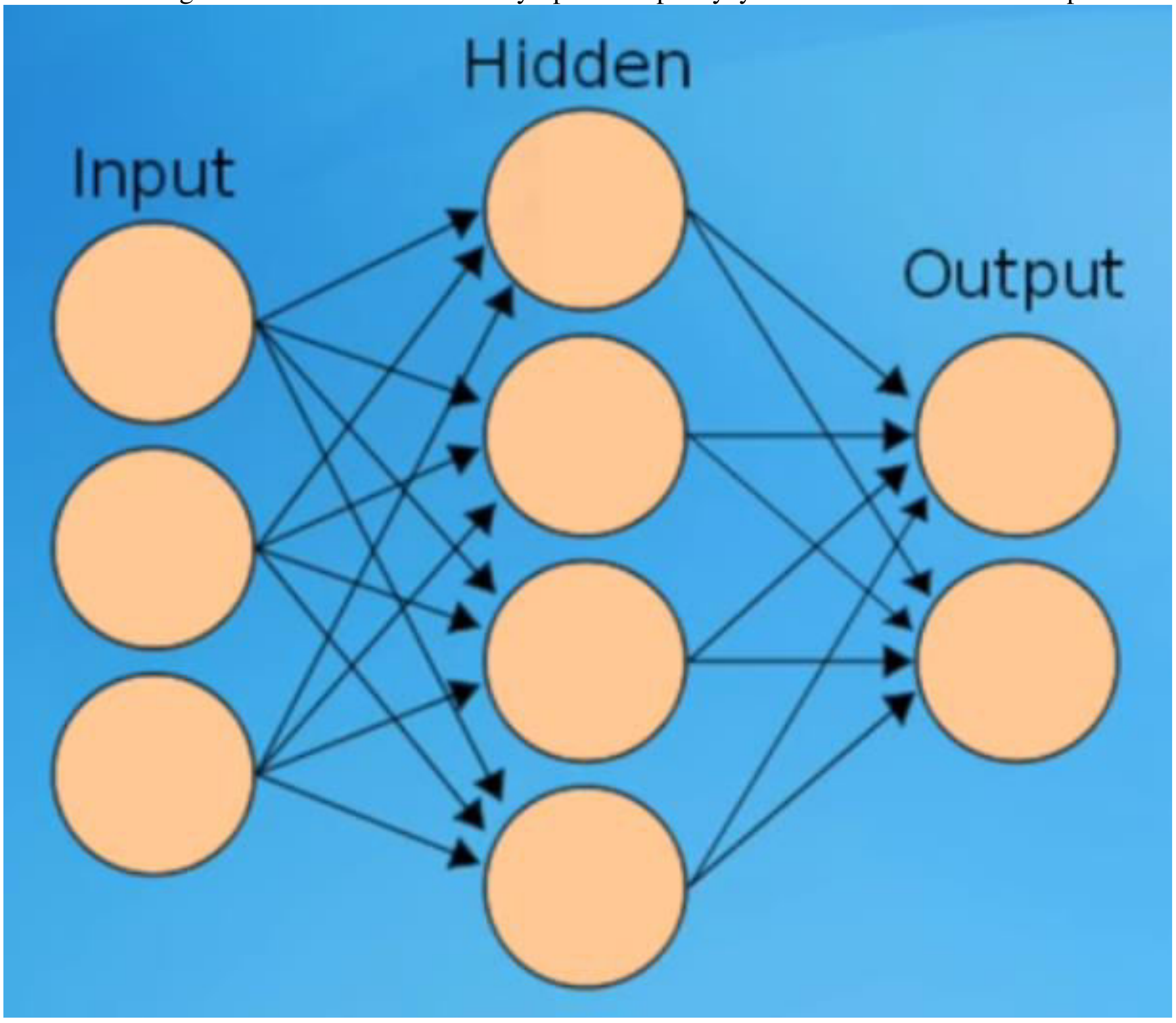
To build our model, we’ll first initialize it as sequential since it’ll be a linear stack of layers. Then we’lll add our first layer which is an LSTM layer:

```
#Step-2 Build model

model = Sequential()

model.add(LSTM(

        input_dim = 1,

        output_dim = 50,

        return_sequences = True))

model.add(Dropout(0.2))
```

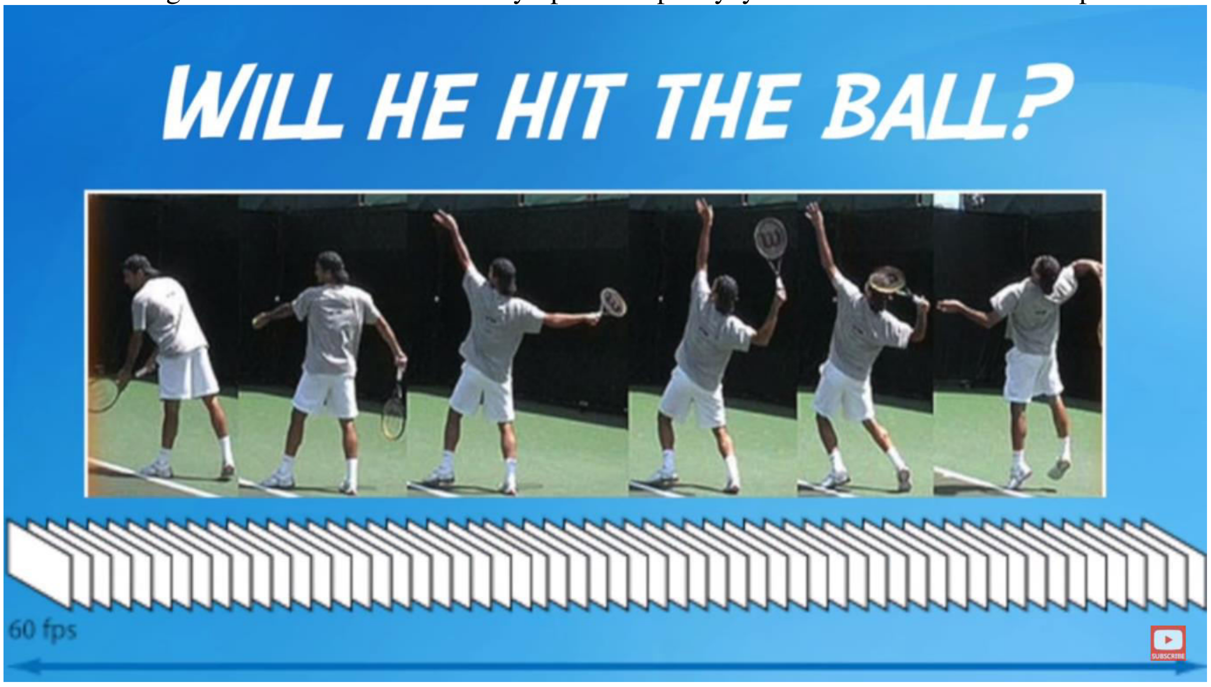
So, what’s LSTM?

It’s easy to record the words(of a song) forward. But could we sing them backwards? No. The reason is, we learn these words in a sequence: it’s **Conditional Meomory**. We can access the words if we could access the words before it:

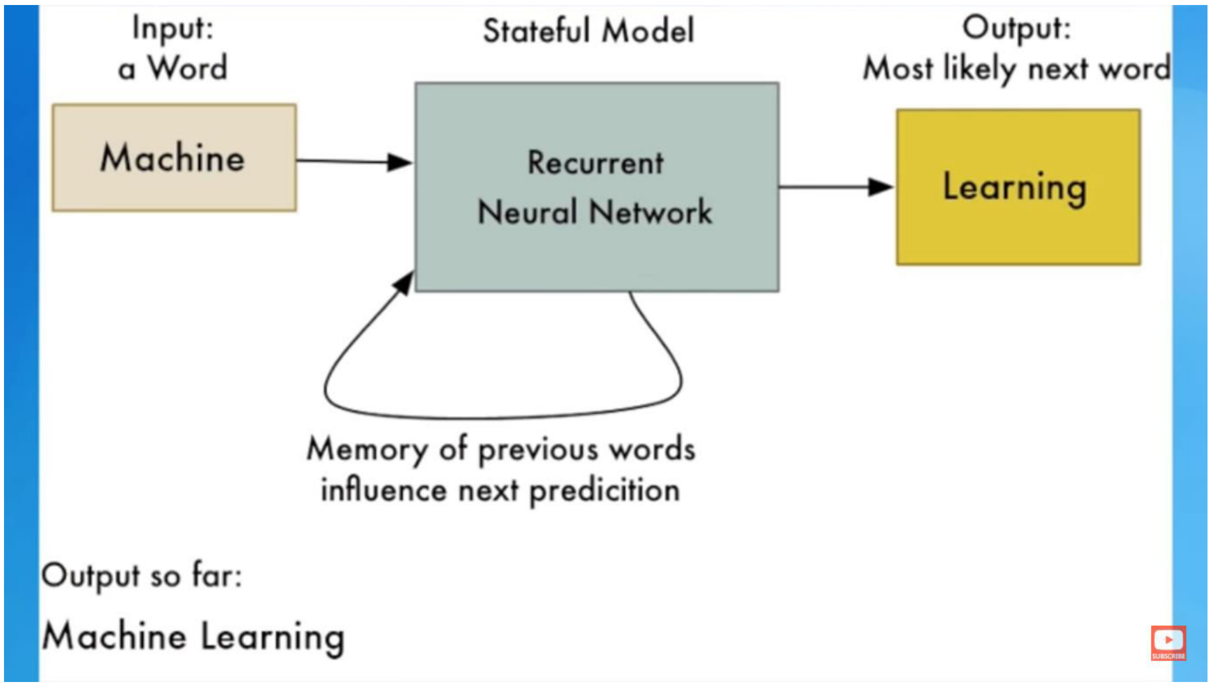
Memory matters when we have sequences. Our thoughts have persistence, but Feed-Forward Neural Networks don’t.:

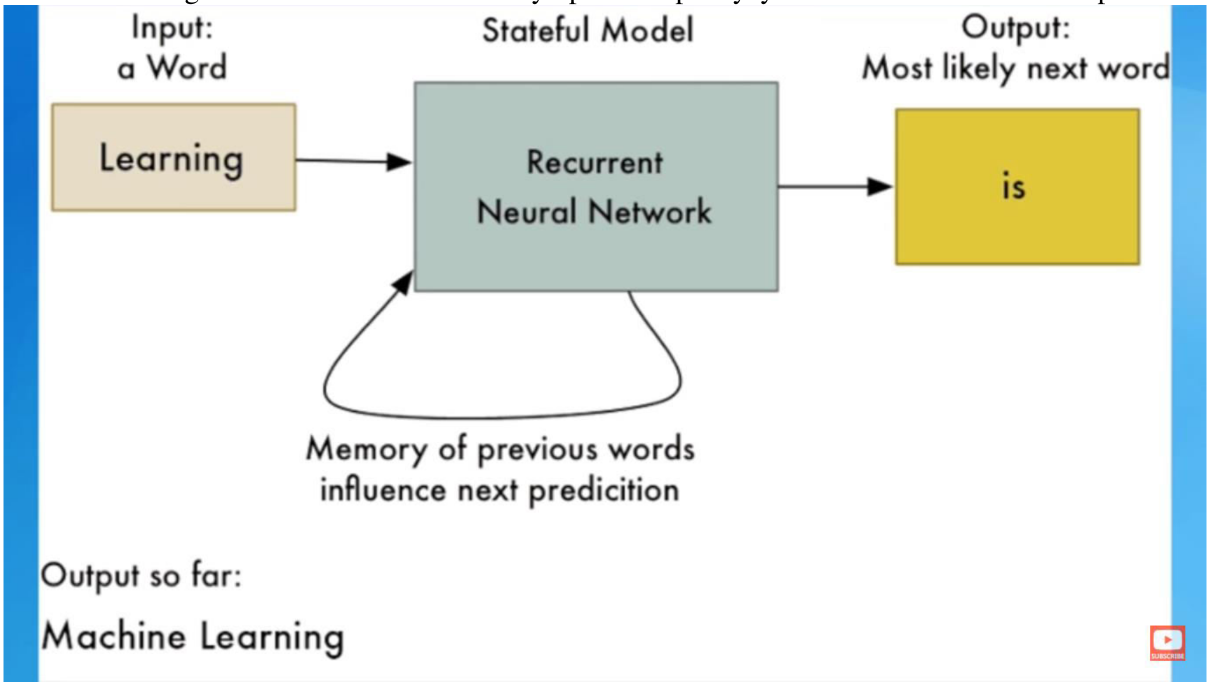


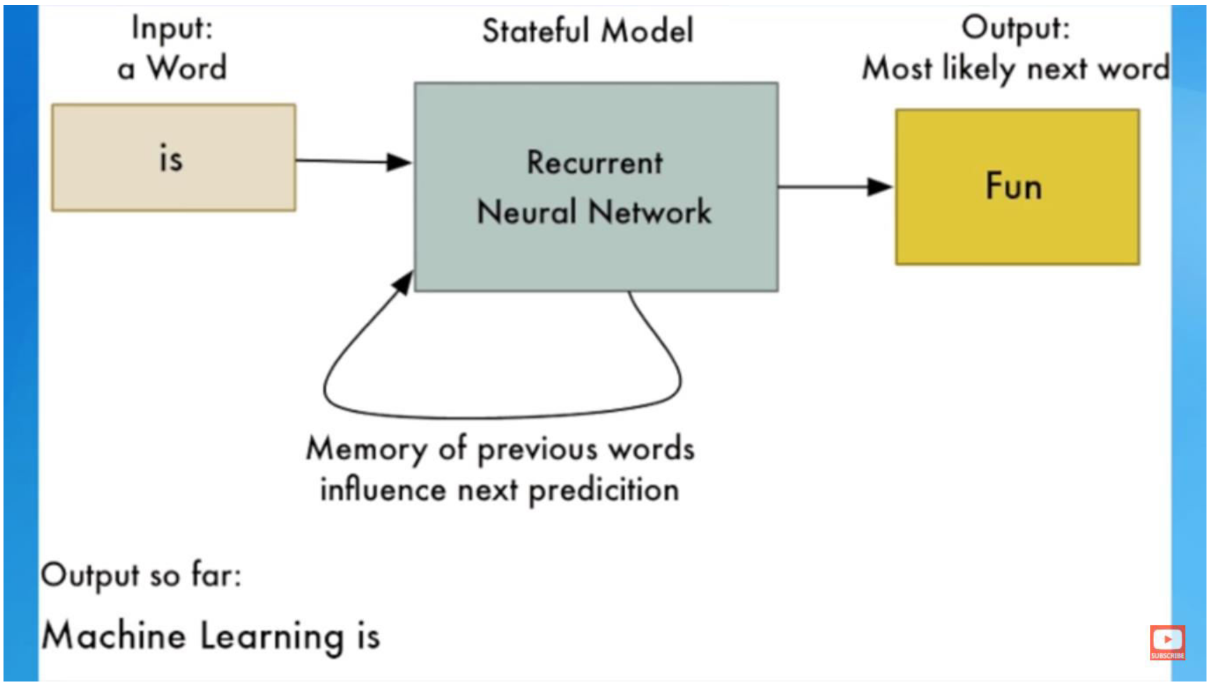
Feed-Forward Neural Networks accept a fixed size vector as input, like an image. So, we could not use it to predict next frame in a movie, because, that would require a series of image vectors as inputs, not just one, since the probability of a certain event happening will depend on every frame that happened before it:



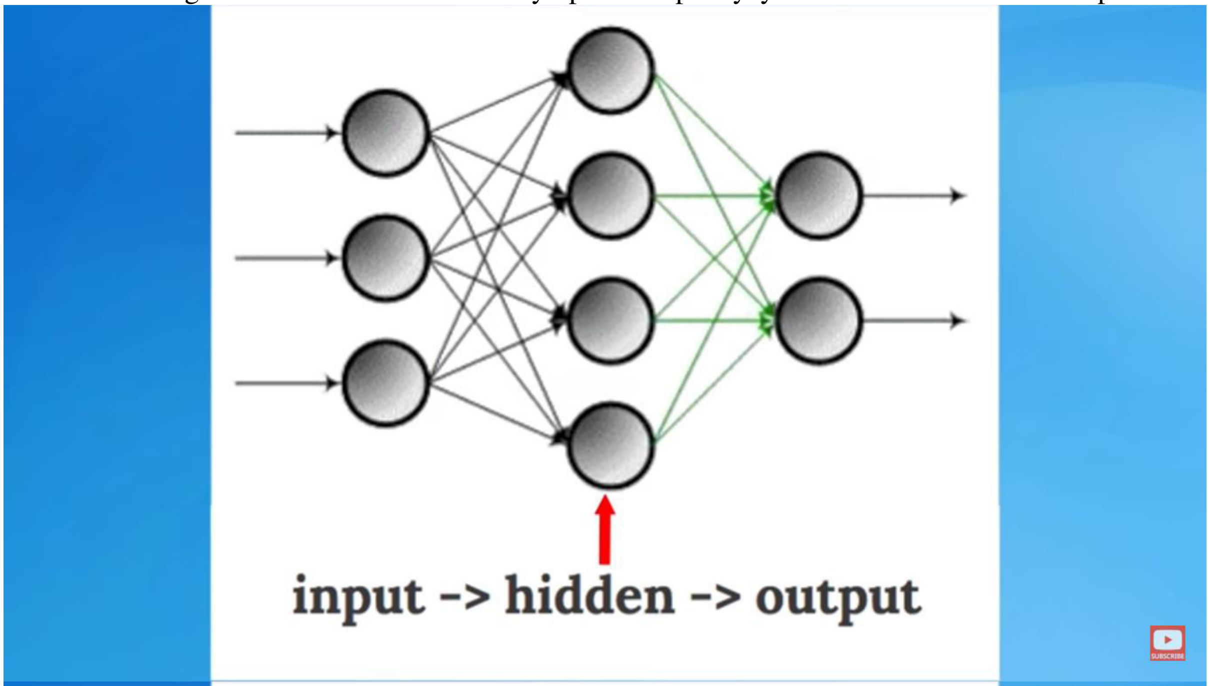
We need a way to allow information to persist. And that's why we'll use a **Recurrent Neural Network(RNN)**. RNNs can accept a series of vectors as inputs.:

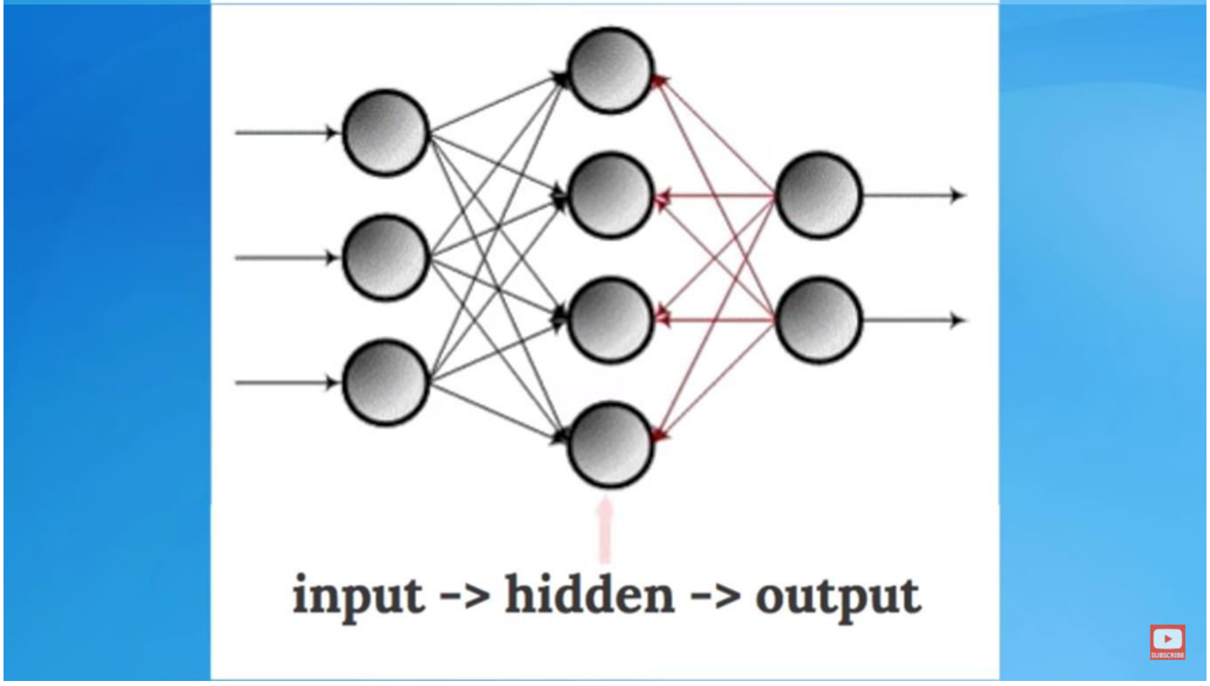




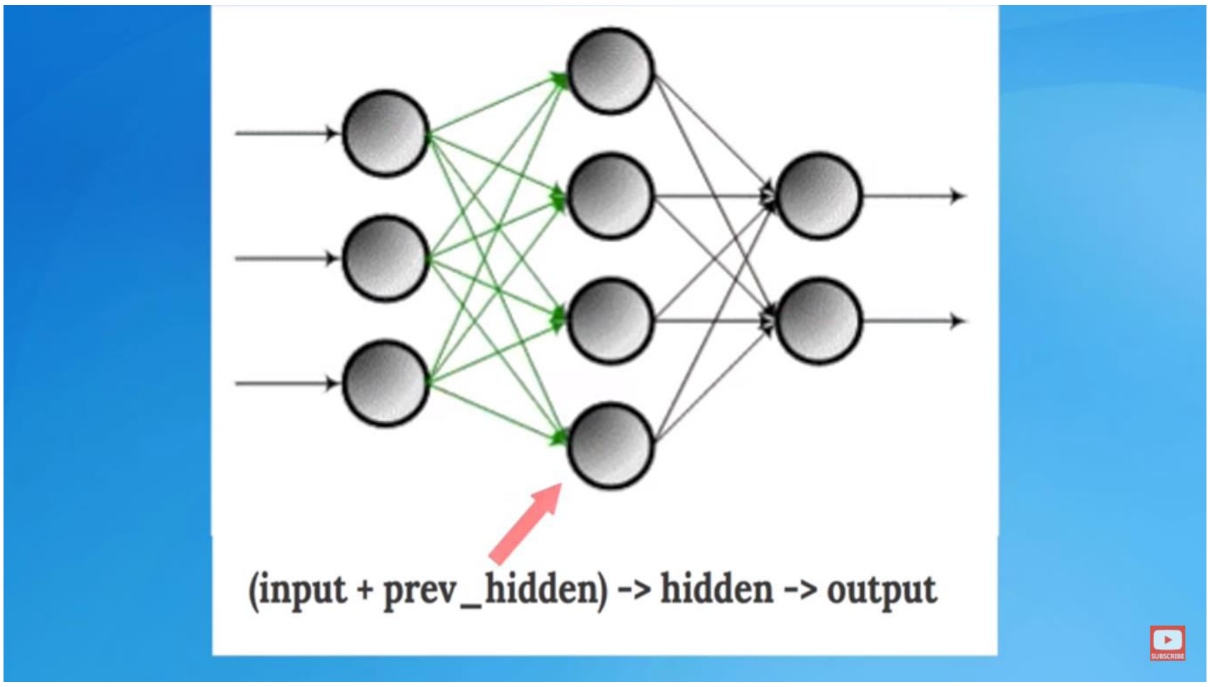


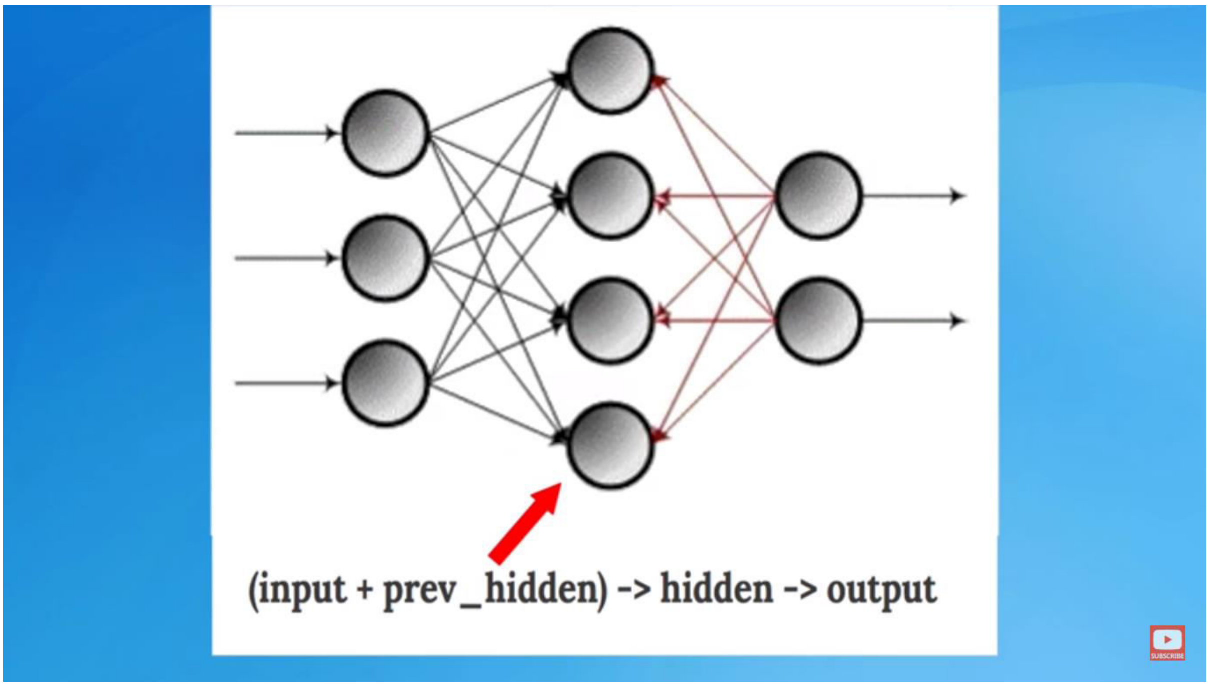
In Feed-Forward Neural Networks, the hidden layers’ weights are based only on the input data:


input -> hidden -> output

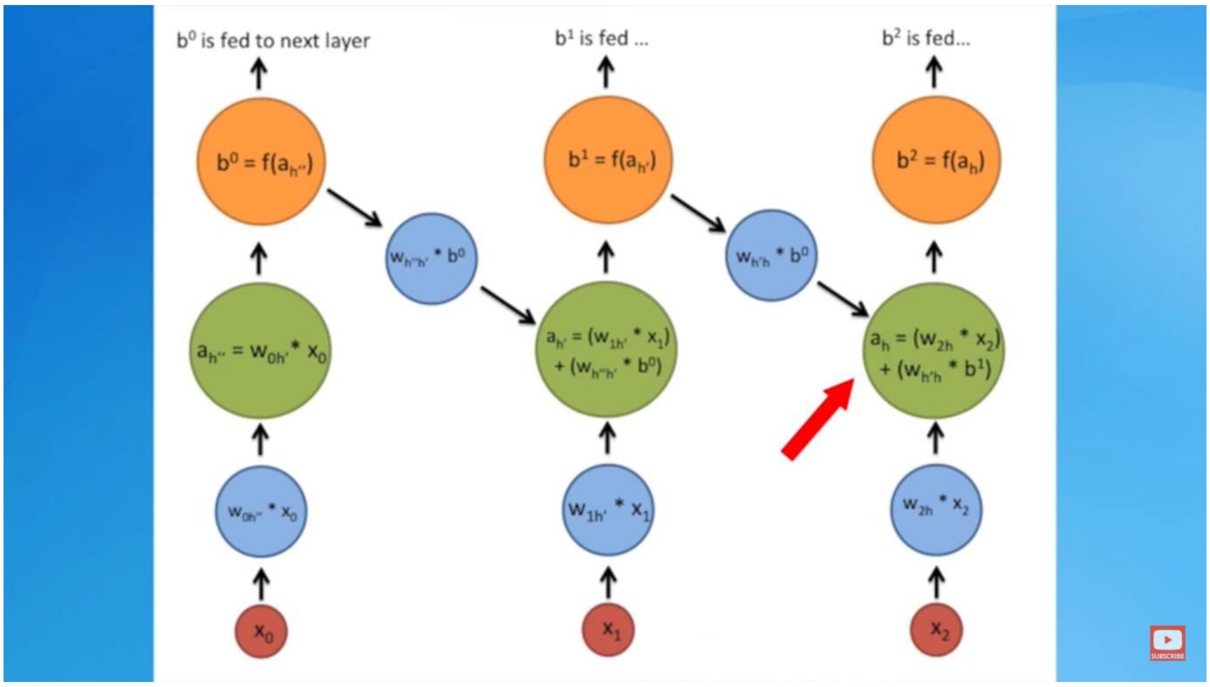

input -> hidden -> output

But, in an RNN, the hidden layer is a combination of the input data at the current time-step and the hidden layer at a previous time-step:

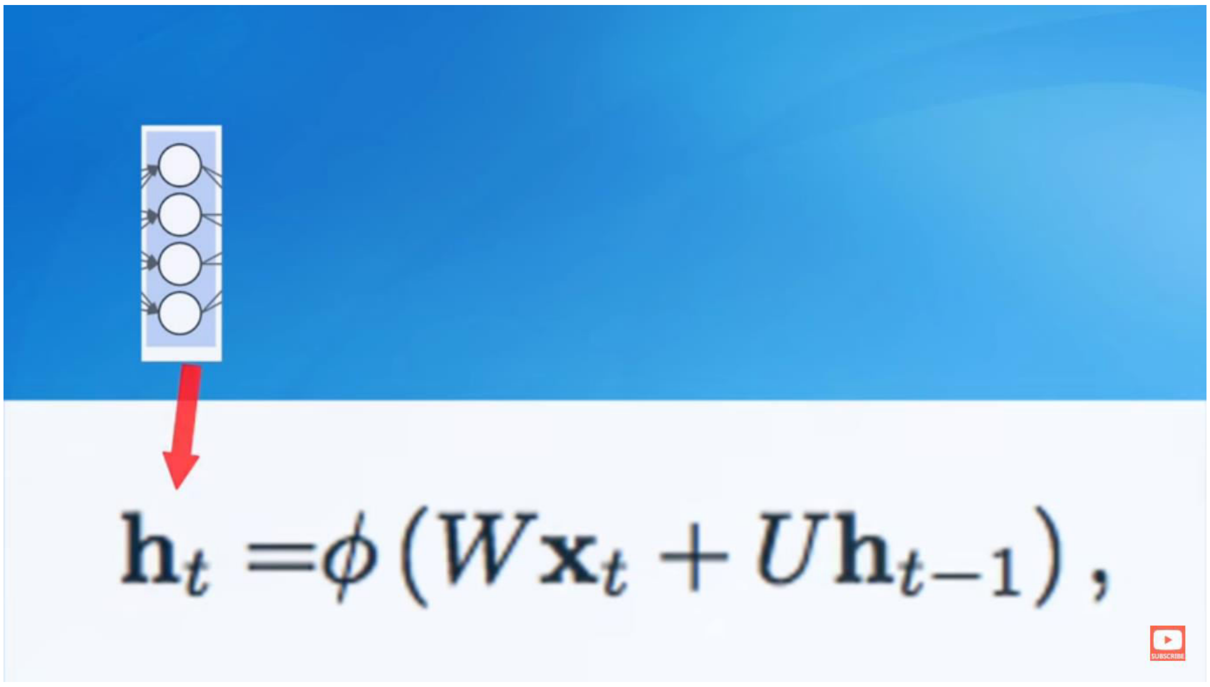




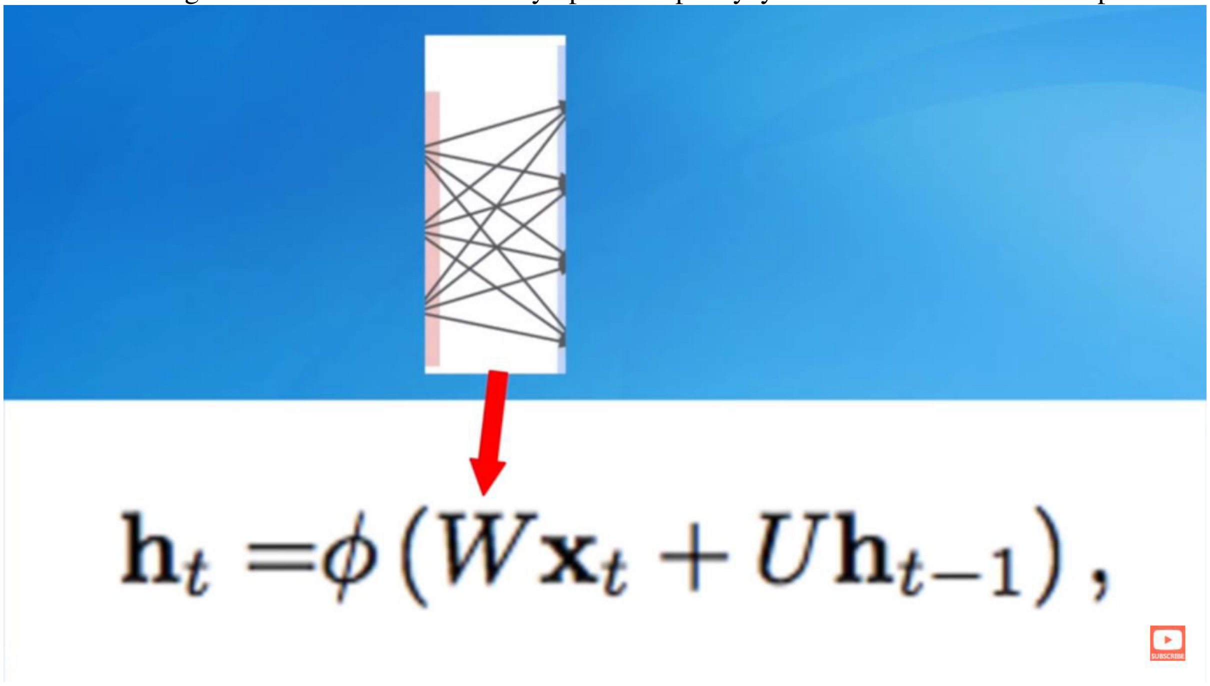
The hidden layer is constantly changing as it gets more inputs and the only way to reach these hidden states is with the correct sequence of inputs. This is how memory is incorporated:



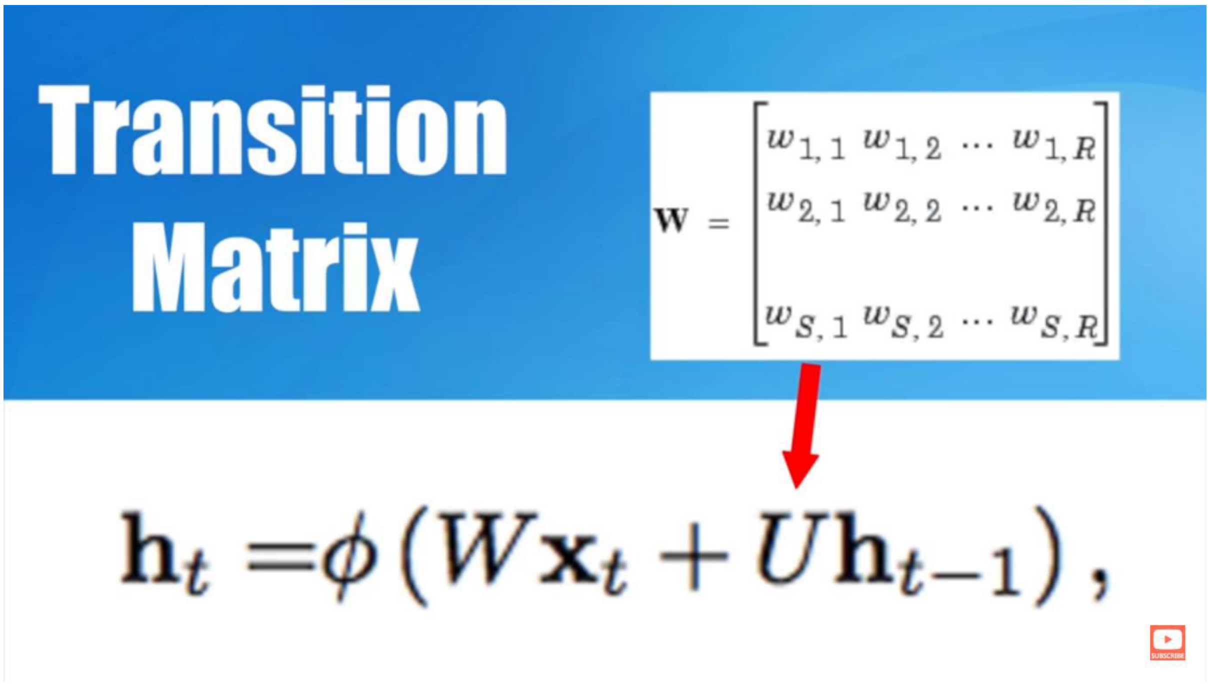
And we can model this process mathematically:



$$\mathbf{h}_t = \phi\left(W\mathbf{x}_t + U\mathbf{h}_{t-1}\right),$$



$$\mathbf{h}_t = \phi(W\mathbf{x}_t + U\mathbf{h}_{t-1}),$$



Transition Matrix

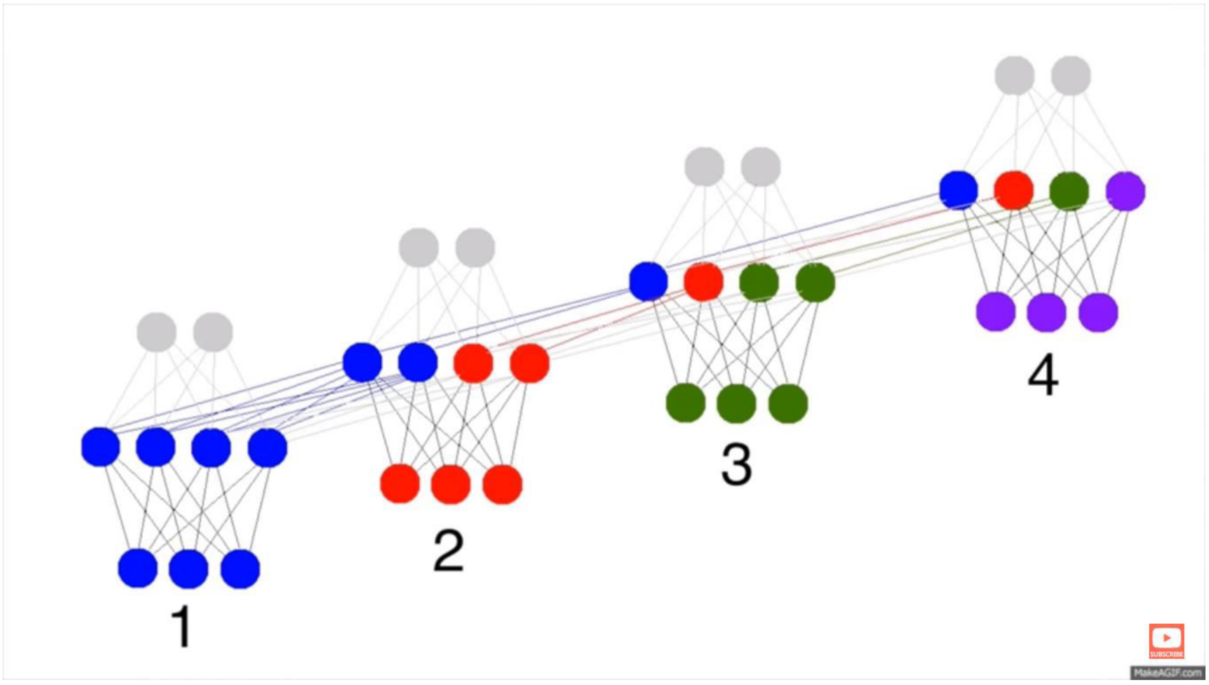
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

$$\mathbf{h}_t = \phi(W\mathbf{x}_t + U\mathbf{h}_{t-1}),$$

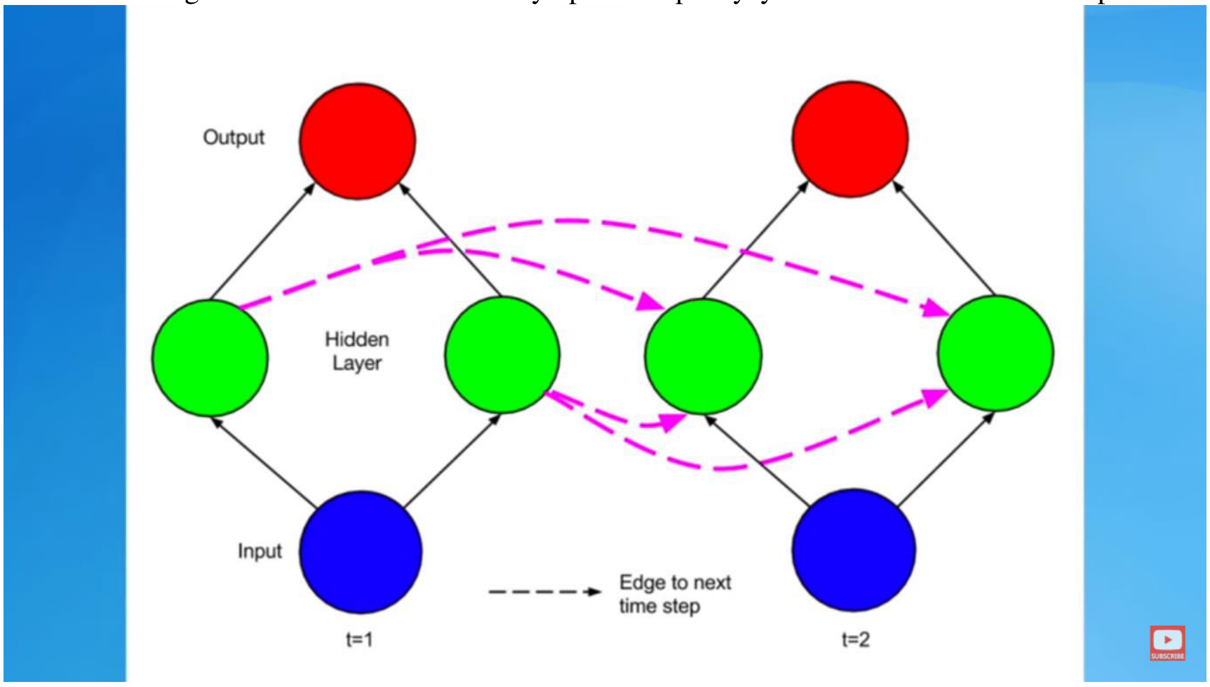
Where, ht = hidden time layer at a given time-step
w = weight matrix
xt = input at that same time-step
ht-1 = hidden state of previous time-step
U = own hidden state to hidden state matrix(or, transition matrix)

So, this hidden-step at a given time-step is a function of the input at the same time-step, modified by a weight-matrix(like the ones used in feed-forward networks) , added to the hidden states of the previous time-step, multiplied by its own hidden-state to hidden-state matrix(otherwise known as transition matrix).

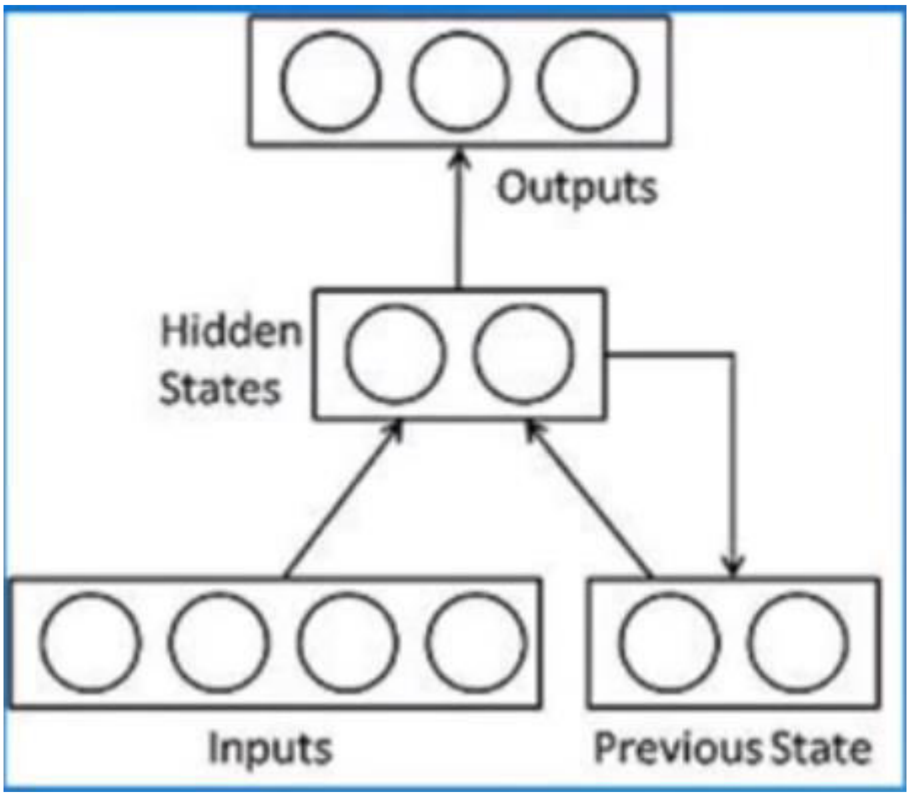
And because this feedback loop is recurrent at every time-step in the series, each hidden-state has traces left. Not only of the previous hidden states, but also of all of those that preceded it. That’s why we call it recurrent:



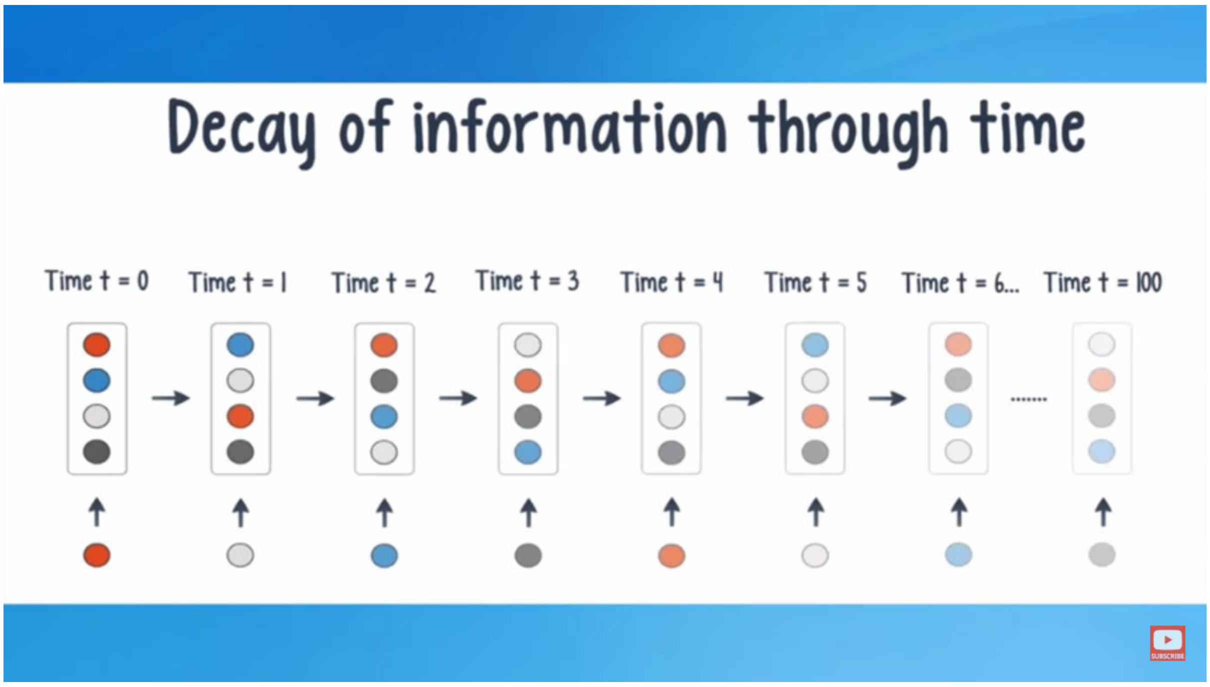
In a way, we think of it as copies of the same network, each passing a message to the next:



So, that's the great thing about RNN: they're able to connect previous data with the present task-

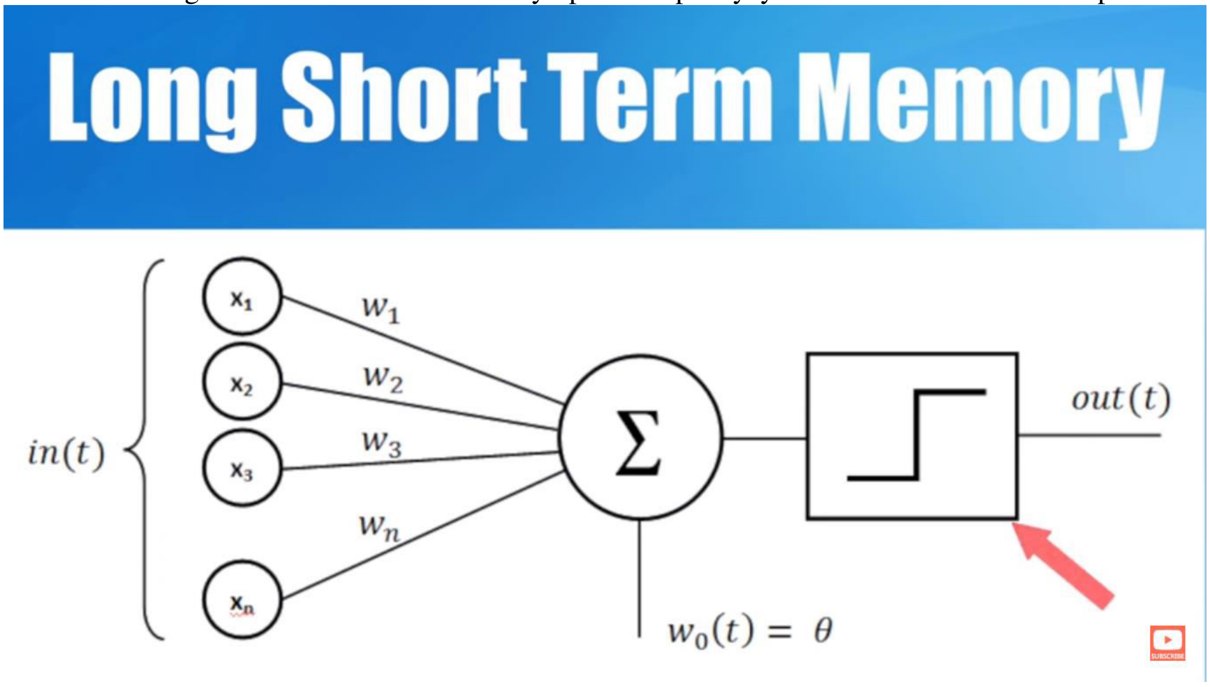


But we still have a problem: in regular RNNs, memories become more weaken as they're fed into the past:

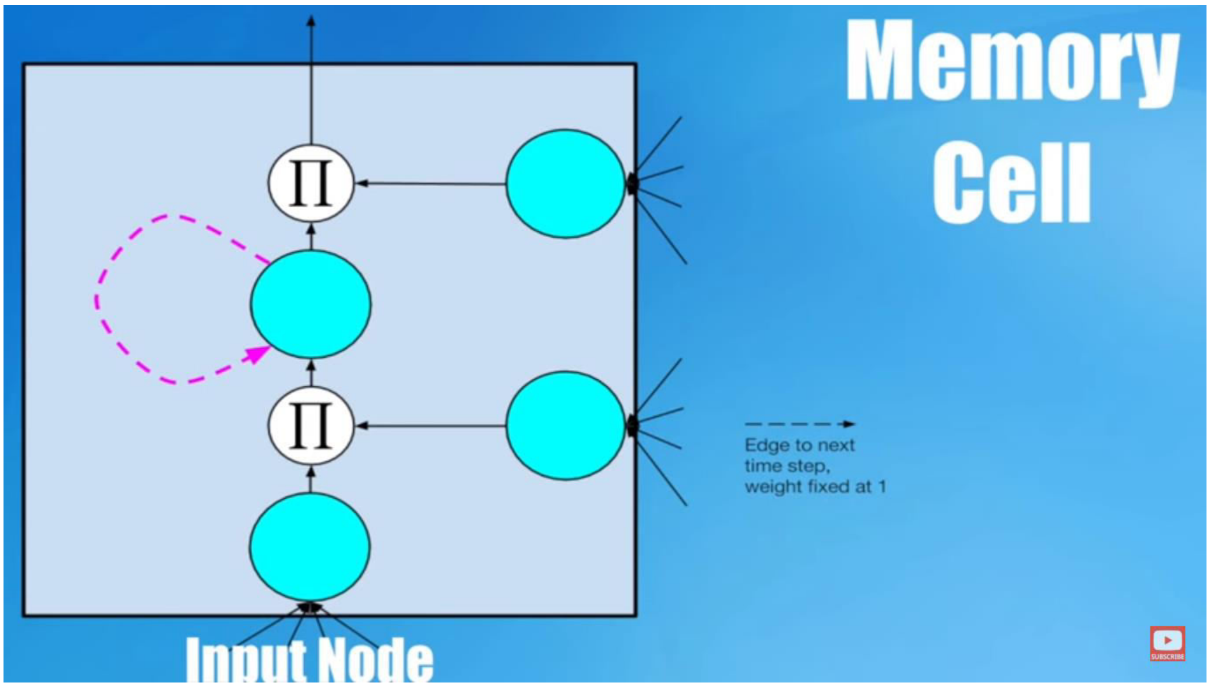


, since the error signal from later time-steps doesn't make far enough back in time to influence the network at earlier time-steps during **Back-Propagation.** **Yoshua Bengio** called it the **"Vanishing Gradient Problem",** in one of his most cited papers called **"Learning Long-Term Dependencies with Gradient Descent is Difficult".**

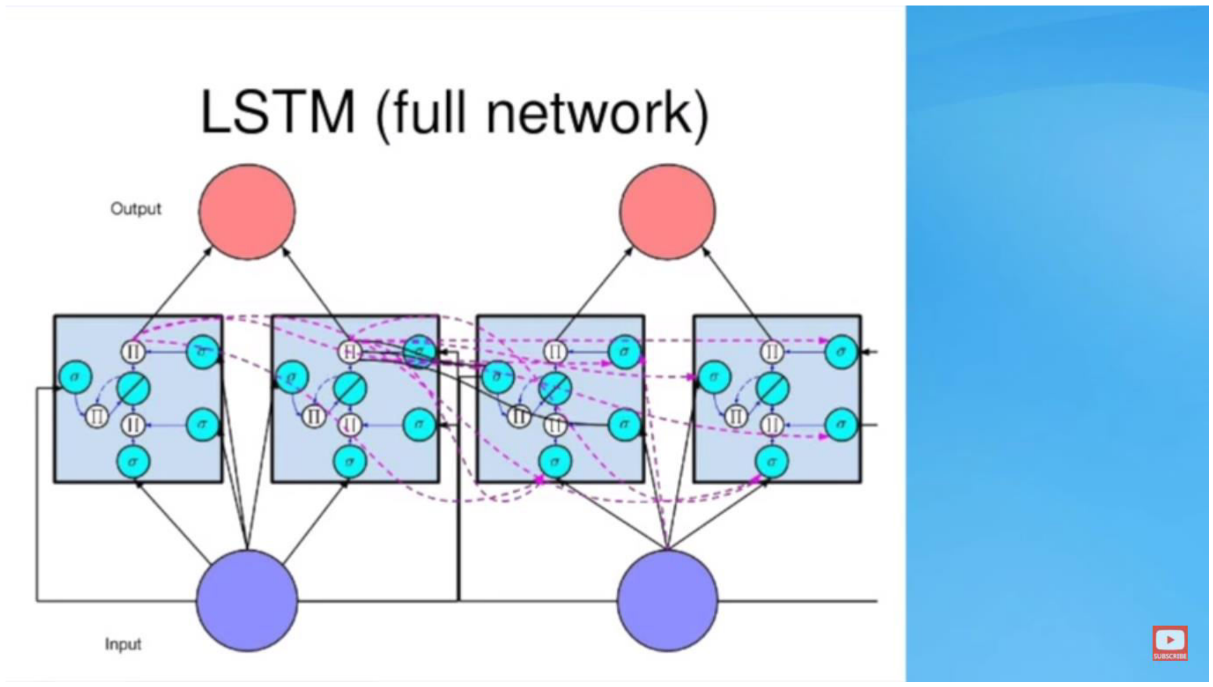
A popular solution to this is **Long Short Term Memory(LSTM):**



Normally, **Neurons** are units that apply **Activation Function,** like a **Sigmoid,** to **a linear combination of the inputs.** We instead replace these Neurons in an LSTM RNN, which are called **Memory Cells**:



So, despite everything else in an RNN staying the same, doing this more powerful **Update Equation** for our Hidden-State results in our network being able to remember Long-Term Dependencies:



So, for our LSTM layer, we'll set our input dimension to 1, and say we want 50 units in this layer. Setting return sequences to True, means this layer's output is always fed to the next layer. We'll add 20% dropout to this layer:

```
model.add(LSTM(
        input_dim = 1,
        output_dim = 50,
        return_sequences = True))
model.add(Dropout(0.2))
```

We'll initialize our Second Layer as another LSTM with 100 units and set return sequence to False:

```
model.add(LSTM(
        100,
        return_sequences = False))
model.add(Dropout(0.2))
```

We use linear dense layer to aggregate the data from this prediction vector into one single value:

```
model.add(Dense(
        output_dim = 1))
model.add(Activation('linear'))
```

Then we can compile our model using a popular loss function called **"Mean Square Function":**

```
start = time.time()
model.compile(loss = 'mse', optimizer = 'rmsprop')
print('Compilation time: ', time.time() - start)
```

We'll train our model the fit() function:

```
#Step-3 train the model
model.fit(
        X_train,
        y_train,
        batch_size = 512,
        nb_epoch = 1,
        validation_split = 0.05)
```

Then we can test it to see what it predicts for the next 50 steps:

```
#Step-4 plot the predictions!
predictions = lstm.predict_sequences_multiple(model, X_test, 50, 50)
run.plot_results_multiple(predictions, y_test, 50)
```

Implementation of Kalman Filter Estimation of Mean in Python using PyKalman, Bokeh and NSEPy:

Kalman Filter is an optimal estimation algorithm to estimate the variable which can be measured indirectly and to find the best estimate of states by combining measurement from various sensors in the presence of noise.
When comes to implementation of Kalman filter python comes very handy as the librry PyKalman makes life easier rather than digging with complex math stuff to calculate kalman estimation.

6. Implementation Details:

Decision Tree:

```
from sklearn import tree

features= [ [177.85, 2987], [175.11, 3088], [177.07, 1081] ]

label= [175.85, 177.85, 176.5]

clf= tree.DecisionTreeClassifier()

clf=clf.fit(features, label)

print (clf.predict([[180, 1587]]))
```

KNN METHODOLOGY:

```
# Python3 program to find groups of unknown
# Points using K nearest neighbour algorithm.

import math

def classifyAPoint(points,p,k=3):
    '''
     This function finds classification of p using
     k nearest neighbour algorithm. It assumes only two
     groups and returns 0 if p belongs to group 0, else
      1 (belongs to group 1).

      Parameters -
         points : Dictionary of training points having two keys - 0 and 1
                Each key have a list of training data points belong to that

         p : A touple ,test data point of form (x,y)

         k : number of nearest neighbour to consider, default is 3
    '''

    distance=[]
    for group in points:
        for feature in points[group]:

            #calculate the euclidean distance of p from training points
            euclidean_distance = math.sqrt((feature[0]-p[0])**2 +(feature[1]-
p[1])**2)

            # Add a touple of form (distance,group) in the distance list
            distance.append((euclidean_distance,group))

    # sort the distance list in ascending order
    # and select first k distances
    distance = sorted(distance)[:k]

    freq1 = 0 #frequency of group 0
    freq2 = 0 #frequency og group 1

    for d in distance:
        if d[1] == 0:
            freq1 += 1
        elif d[1] == 1:
            freq2 += 1

    return 0 if freq1>freq2 else 1

# driver function
```

```
def main():

    # Dictionary of training points having two keys - 0 and 1
    # key 0 have points belong to class 0
    # key 1 have points belong to class 1

    points = {0:[(1,12),(2,5),(3,6),(3,10),(3.5,8),(2,11),(2,9),(1,7)],
             1:[(5,3),(3,2),(1.5,9),(7,2),(6,1),(3.8,1),(5.6,4),(4,2),(2,5)]}

    # testing point p(x,y)
    p = (2.5,7)

    # Number of neighbours
    k = 3

    print("The value classified to unknown point is: {}".\
         format(classifyAPoint(points,p,k)))

if __name__ == '__main__':
    main()
```

Support Vector Regression:

```
import csv

import numpy as np

from sklearn.svm import SVR

import matplotlib.pyplot as plt



dates = []

prices = []

print("Hello")
```

```
def get_data(filename):

    with open(filename, 'r') as csvfile:

        csvFileReader = csv.reader(csvfile)

        next(csvFileReader)

        for row in csvFileReader:

            dates.append(int(row[0].split('-')[2]))

            prices.append(float(row[1]))

    return



def predict_prices(dates, prices, x):

    dates = np.reshape(dates,(len(dates), 1))

    #print(dates)

    svr_lin = SVR(kernel= 'linear', C=1e3)

    svr_poly = SVR(kernel= 'poly', C=1e3, degree = 2)

    svr_rbf = SVR(kernel= 'rbf', C=1e3, gamma=0.1)

    svr_lin.fit(dates, prices)

    svr_poly.fit(dates, prices)

    svr_rbf.fit(dates, prices)

    plt.scatter(dates, prices, color='black', label='Data')

    plt.plot(dates, svr_rbf.predict(dates), color='red', label='RBF model')

    plt.plot(dates, svr_lin.predict(dates), color='green', label='Linear model')
```

```
        plt.plot(dates, svr_poly.predict(dates), color='blue', label='Polynomial
model')

        plt.xlabel('Date')

        plt.ylabel('Price')

        plt.title('Support Vector Regression')

        plt.legend()

        plt.show()

        print("Show end")

        return svr_rbf.predict(x)[0], svr_lin.predict(x)[0], svr_poly.predict(x)[0]



get_data('AAPL.csv')


predicted_price = predict_prices(dates, prices, 29)

print(predicted_price)
```

Deep Learning Model:

```
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.models import Sequential
import lstm, run, time #helper libraries


#Step-1 Load data
```

```
X_train, y_train, X_test, y_test = lstm.load_data('sp500.csv', 50, True)
print(X_train)

#Step-2 Build model
model = Sequential()

model.add(LSTM(
        input_dim = 1,
        output_dim = 50,
        return_sequences = True))
model.add(Dropout(0.2))

model.add(LSTM(
        100,
        return_sequences = False))
model.add(Dropout(0.2))

model.add(Dense(
        output_dim = 1))
model.add(Activation('linear'))

start = time.time()
model.compile(loss = 'mse', optimizer = 'rmsprop')
print('Compilation time: ', time.time() - start)


#Step-3 train the model
model.fit(
        X_train,
        y_train,
        batch_size = 512,
        nb_epoch = 1,
        validation_split = 0.05)


#Step-4 plot the predictions!
predictions = lstm.predict_sequences_multiple(model, X_test, 50, 50)
run.plot_results_multiple(predictions, y_test, 50)
```

Reinforcement Learning and Sentiment Analysis(Random Forest Model, Linear Regression Model, Multi Layer Perceptron or Deep Neural Network Model):

```
import numpy as np
import pandas as pd
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import subjectivity
from nltk.sentiment import SentimentAnalyzer
from nltk.sentiment.util import *

# Reading the saved data pickle file
df_stocks = pd.read_pickle('/Users/Dinesh/Documents/Project Stock
predictions/data/pickled_ten_year_filtered_data.pkl')

df_stocks

df_stocks['prices'] = df_stocks['adj close'].apply(np.int64)

# selecting the prices and articles
df_stocks = df_stocks[['prices', 'articles']]

df_stocks['articles'] = df_stocks['articles'].map(lambda x: x.lstrip('.-'))
df_stocks

df = df_stocks[['prices']].copy()
df

# Adding new columns to the data frame
df["compound"] = ''
df["neg"] = ''
df["neu"] = ''
df["pos"] = ''

df

from nltk.sentiment.vader import SentimentIntensityAnalyzer
import unicodedata
sid = SentimentIntensityAnalyzer()
for date, row in df_stocks.T.iteritems():
    try:
        sentence = unicodedata.normalize('NFKD', df_stocks.loc[date,
'articles']).encode('ascii','ignore')
        ss = sid.polarity_scores(sentence)
        df.set_value(date, 'compound', ss['compound'])
        df.set_value(date, 'neg', ss['neg'])
        df.set_value(date, 'neu', ss['neu'])
        df.set_value(date, 'pos', ss['pos'])
    except TypeError:
        print df_stocks.loc[date, 'articles']
        print date

df

train_start_date = '2007-01-01'
train_end_date = '2014-12-31'
test_start_date = '2015-01-01'
test_end_date = '2016-12-31'
train = df.ix[train_start_date : train_end_date]
test = df.ix[test_start_date:test_end_date]

sentiment_score_list = []
for date, row in train.T.iteritems():
    #sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
    sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
    sentiment_score_list.append(sentiment_score)
numpy_df_train = np.asarray(sentiment_score_list)
sentiment_score_list = []
for date, row in test.T.iteritems():
    #sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
    sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
    sentiment_score_list.append(sentiment_score)
numpy_df_test = np.asarray(sentiment_score_list)

y_train = pd.DataFrame(train['prices'])
y_test = pd.DataFrame(test['prices'])

from treeinterpreter import treeinterpreter as ti
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import classification_report,confusion_matrix
```

```
rf = RandomForestRegressor()
rf.fit(numpy_df_train, y_train)

print rf.feature_importances_

prediction, bias, contributions = ti.predict(rf, numpy_df_test)

prediction

contributions

import matplotlib.pyplot as plt
%matplotlib inline

idx = pd.date_range(test_start_date, test_end_date)
predictions_df = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])

predictions_df

#predictions_df.plot()
#test['prices'].plot()

predictions_plot = predictions_df.plot()

fig = y_test.plot(ax = predictions_plot).get_figure()
fig.savefig("graphs/random forest without smoothing.png")

ax = predictions_df.rename(columns={"prices":
"predicted_price"}).plot(title='Random Forest predicted prices 8-2 years')
ax.set_xlabel("Dates")
ax.set_ylabel("Stock Prices")
fig = y_test.rename(columns={"prices": "actual_price"}).plot(ax =
ax).get_figure()
fig.savefig("graphs/random forest without smoothing.png")

# colors = ['332288', '88CCEE', '44AA99', '117733', '999933', 'DDCC77',
'CC6677', '882255', 'AA4499']

test
```

```
from datetime import datetime, timedelta

temp_date = test_start_date
average_last_5_days_test = 0
total_days = 10
for i in range(total_days):
    average_last_5_days_test += test.loc[temp_date, 'prices']
    # Converting string to date time
    temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
    # Reducing one day from date time
    difference = temp_date + timedelta(days=1)
    # Converting again date time to string
    temp_date = difference.strftime('%Y-%m-%d')
    #print temp_date
average_last_5_days_test = average_last_5_days_test / total_days
print average_last_5_days_test

temp_date = test_start_date
average_upcoming_5_days_predicted = 0
for i in range(total_days):
    average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']
    # Converting string to date time
    temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
    # Adding one day from date time
    difference = temp_date + timedelta(days=1)
    # Converting again date time to string
    temp_date = difference.strftime('%Y-%m-%d')
    print temp_date
average_upcoming_5_days_predicted = average_upcoming_5_days_predicted /
total_days
print average_upcoming_5_days_predicted
#average train.loc['2013-12-31', 'prices'] - advpredictions_df.loc['2014-01-01',
'prices']
difference_test_predicted_prices = average_last_5_days_test -
average_upcoming_5_days_predicted
print difference_test_predicted_prices

from datetime import datetime, timedelta

temp_date = test_start_date
average_last_5_days_test = 0
```

```
total_days = 10
for i in range(total_days):
    average_last_5_days_test += test.loc[temp_date, 'prices']
    # Converting string to date time
    temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
    # Reducing one day from date time
    difference = temp_date + timedelta(days=1)
    # Converting again date time to string
    temp_date = difference.strftime('%Y-%m-%d')
    #print temp_date
average_last_5_days_test = average_last_5_days_test / total_days
print average_last_5_days_test

temp_date = test_start_date
average_upcoming_5_days_predicted = 0
for i in range(total_days):
    average_upcoming_5_days_predicted += predictions_df.loc[temp_date, 
'prices']
    # Converting string to date time
    temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
    # Adding one day from date time
    difference = temp_date + timedelta(days=1)
    # Converting again date time to string
    temp_date = difference.strftime('%Y-%m-%d')
    print temp_date
average_upcoming_5_days_predicted = average_upcoming_5_days_predicted / 
total_days
print average_upcoming_5_days_predicted
#average train.loc['2013-12-31', 'prices'] - advpredictions_df.loc['2014-01-01', 
'prices']
difference_test_predicted_prices = average_last_5_days_test - 
average_upcoming_5_days_predicted
print difference_test_predicted_prices

# Adding 6177 to all the advpredictions_df price values
predictions_df['prices'] = predictions_df['prices'] + 
difference_test_predicted_prices
predictions_df

ax = predictions_df.rename(columns={"prices": 
"predicted_price"}).plot(title='Random Forest predicted prices 8-2 years after 
aligning')
```

```
ax.set_xlabel("Dates")
ax.set_ylabel("Stock Prices")
fig = y_test.rename(columns={"prices": "actual_price"}).plot(ax =
ax).get_figure()
fig.savefig("graphs/random forest with aligning.png")

predictions_df

predictions_df['ewma'] = pd.ewma(predictions_df["prices"], span=60,
freq="D")

predictions_df

 predictions_df['actual_value'] = test['prices']
predictions_df['actual_value_ewma'] =
pd.ewma(predictions_df["actual_value"], span=60, freq="D")

predictions_df

# Changing column names
predictions_df.columns = ['predicted_price', 'average_predicted_price',
'actual_price', 'average_actual_price']

# Now plotting test predictions after smoothing
predictions_plot = predictions_df.plot(title='Random Forest predicted prices 8-2
years after aligning & smoothing')
predictions_plot.set_xlabel("Dates")
predictions_plot.set_ylabel("Stock Prices")
fig = predictions_plot.get_figure()
fig.savefig("graphs/random forest after smoothing.png")

# Plotting just predict and actual average curves
predictions_df_average = predictions_df[['average_predicted_price',
'average_actual_price']]
predictions_plot = predictions_df_average.plot(title='Random Forest 8-2 years
after aligning & smoothing')
predictions_plot.set_xlabel("Dates")
predictions_plot.set_ylabel("Stock Prices")
fig = predictions_plot.get_figure()
fig.savefig("graphs/random forest after smoothing 2.png")

def offset_value(test_start_date, test, predictions_df):
    temp_date = test_start_date
    average_last_5_days_test = 0
    average_upcoming_5_days_predicted = 0
    total_days = 10
    for i in range(total_days):
        average_last_5_days_test += test.loc[temp_date, 'prices']
        temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
        difference = temp_date + timedelta(days=1)
        temp_date = difference.strftime('%Y-%m-%d')
    average_last_5_days_test = average_last_5_days_test / total_days

    temp_date = test_start_date
    for i in range(total_days):
        average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']
        temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
        difference = temp_date + timedelta(days=1)
        temp_date = difference.strftime('%Y-%m-%d')
    average_upcoming_5_days_predicted =
average_upcoming_5_days_predicted / total_days
    difference_test_predicted_prices = average_last_5_days_test -
average_upcoming_5_days_predicted
    return difference_test_predicted_prices

from treeinterpreter import treeinterpreter as ti
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LogisticRegression
from datetime import datetime, timedelta

# average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']
# # Converting string to date time
# temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
# # Adding one day from date time
# difference = temp_date + timedelta(days=1)
# # Converting again date time to string
# temp_date = difference.strftime('%Y-%m-%d')

# start_year = datetime.strptime(train_start_date, "%Y-%m-%d").date().month

years = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016]
```

```
prediction_list = []
for year in years:
    # Splitting the training and testing data
    train_start_date = str(year) + '-01-01'
    train_end_date = str(year) + '-10-31'
    test_start_date = str(year) + '-11-01'
    test_end_date = str(year) + '-12-31'
    train = df.ix[train_start_date : train_end_date]
    test = df.ix[test_start_date:test_end_date]

    # Calculating the sentiment score
    sentiment_score_list = []
    for date, row in train.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_train = np.asarray(sentiment_score_list)
    sentiment_score_list = []
    for date, row in test.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_test = np.asarray(sentiment_score_list)

    # Generating models
    lr = LogisticRegression()
    lr.fit(numpy_df_train, train['prices'])



    prediction = lr.predict(numpy_df_test)
    prediction_list.append(prediction)
    #print train_start_date + ' ' + train_end_date + ' ' + test_start_date + ' ' +
test_end_date
    idx = pd.date_range(test_start_date, test_end_date)
    #print year
    predictions_df_list = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])

    difference_test_predicted_prices = offset_value(test_start_date, test,
predictions_df_list)
```

```
    # Adding offset to all the advpredictions_df price values
    predictions_df_list['prices'] = predictions_df_list['prices'] +
difference_test_predicted_prices
    predictions_df_list

    # Smoothing the plot
    predictions_df_list['ewma'] = pd.ewma(predictions_df_list["prices"],
span=10, freq="D")
    predictions_df_list['actual_value'] = test['prices']
    predictions_df_list['actual_value_ewma'] =
pd.ewma(predictions_df_list["actual_value"], span=10, freq="D")
    # Changing column names
    predictions_df_list.columns = ['predicted_price', 'average_predicted_price',
'actual_price', 'average_actual_price']
    predictions_df_list.plot()
    predictions_df_list_average = predictions_df_list[['average_predicted_price',
'average_actual_price']]
    predictions_df_list_average.plot()

#     predictions_df_list.show()

lr.classes_

lr.coef_[0]

from treeinterpreter import treeinterpreter as ti
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LogisticRegression
from datetime import datetime, timedelta

# average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']
# # Converting string to date time
# temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
# # Adding one day from date time
# difference = temp_date + timedelta(days=1)
# # Converting again date time to string
# temp_date = difference.strftime('%Y-%m-%d')

# start_year = datetime.strptime(train_start_date, "%Y-%m-%d").date().month
```

```
years = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016]
prediction_list = []
for year in years:
    # Splitting the training and testing data
    train_start_date = str(year) + '-01-01'
    train_end_date = str(year) + '-10-31'
    test_start_date = str(year) + '-11-01'
    test_end_date = str(year) + '-12-31'
    train = df.ix[train_start_date : train_end_date]
    test = df.ix[test_start_date:test_end_date]

    # Calculating the sentiment score
    sentiment_score_list = []
    for date, row in train.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_train = np.asarray(sentiment_score_list)
    sentiment_score_list = []
    for date, row in test.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_test = np.asarray(sentiment_score_list)

    # Generating models
    rf = RandomForestRegressor(random_state=)
    rf.fit(numpy_df_train, train['prices'])
    #print rf

    prediction, bias, contributions = ti.predict(rf, numpy_df_test)
    prediction_list.append(prediction)
    #print train_start_date + ' ' + train_end_date + ' ' + test_start_date + ' ' +
test_end_date
    idx = pd.date_range(test_start_date, test_end_date)
    #print year
    predictions_df_list = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])
```

```
    difference_test_predicted_prices = offset_value(test_start_date, test,
predictions_df_list)
    # Adding offset to all the advpredictions_df price values
    predictions_df_list['prices'] = predictions_df_list['prices'] +
difference_test_predicted_prices
    predictions_df_list

    # Smoothing the plot
    predictions_df_list['ewma'] = pd.ewma(predictions_df_list["prices"],
span=10, freq="D")
    predictions_df_list['actual_value'] = test['prices']
    predictions_df_list['actual_value_ewma'] =
pd.ewma(predictions_df_list["actual_value"], span=10, freq="D")
    # Changing column names
    predictions_df_list.columns = ['predicted_price', 'average_predicted_price',
'actual_price', 'average_actual_price']
    predictions_df_list.plot()
    predictions_df_list_average = predictions_df_list[['average_predicted_price',
'average_actual_price']]
    predictions_df_list_average.plot()

#     predictions_df_list.show()

from sklearn.neural_network import MLPClassifier
from datetime import datetime, timedelta

# average_upcoming_5_days_predicted += predictions_df.loc[temp_date,
'prices']
# # Converting string to date time
# temp_date = datetime.strptime(temp_date, "%Y-%m-%d").date()
# # Adding one day from date time
# difference = temp_date + timedelta(days=1)
# # Converting again date time to string
# temp_date = difference.strftime('%Y-%m-%d')

# start_year = datetime.strptime(train_start_date, "%Y-%m-%d").date().month

years = [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016]
prediction_list = []
for year in years:
    # Splitting the training and testing data
    train_start_date = str(year) + '-01-01'
    train_end_date = str(year) + '-10-31'
    test_start_date = str(year) + '-11-01'
    test_end_date = str(year) + '-12-31'
    train = df.ix[train_start_date : train_end_date]
    test = df.ix[test_start_date:test_end_date]

    # Calculating the sentiment score
    sentiment_score_list = []
    for date, row in train.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_train = np.asarray(sentiment_score_list)
    sentiment_score_list = []
    for date, row in test.T.iteritems():
        sentiment_score = np.asarray([df.loc[date, 'compound'],df.loc[date,
'neg'],df.loc[date, 'neu'],df.loc[date, 'pos']])
        #sentiment_score = np.asarray([df.loc[date, 'neg'],df.loc[date, 'pos']])
        sentiment_score_list.append(sentiment_score)
    numpy_df_test = np.asarray(sentiment_score_list)

    # Generating models
    mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 100), activation='relu',
                          solver='lbfgs', alpha=0.005, learning_rate_init = 0.001,
shuffle=False) # span = 20 # best 1
    mlpc.fit(numpy_df_train, train['prices'])
    prediction = mlpc.predict(numpy_df_test)

    prediction_list.append(prediction)
    #print train_start_date + ' ' + train_end_date + ' ' + test_start_date + ' ' +
test_end_date
    idx = pd.date_range(test_start_date, test_end_date)
    #print year
    predictions_df_list = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])

    difference_test_predicted_prices = offset_value(test_start_date, test,
predictions_df_list)
    # Adding offset to all the advpredictions_df price values
    predictions_df_list['prices'] = predictions_df_list['prices'] +
difference_test_predicted_prices
```

```
    predictions_df_list

    # Smoothing the plot
    predictions_df_list['ewma'] = pd.ewma(predictions_df_list["prices"],
span=20, freq="D")
    predictions_df_list['actual_value'] = test['prices']
    predictions_df_list['actual_value_ewma'] =
pd.ewma(predictions_df_list["actual_value"], span=20, freq="D")
    # Changing column names
    predictions_df_list.columns = ['predicted_price', 'average_predicted_price',
'actual_price', 'average_actual_price']
    predictions_df_list.plot()
    predictions_df_list_average = predictions_df_list[['average_predicted_price',
'average_actual_price']]
    predictions_df_list_average.plot()

#     predictions_df_list.show()

mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 100), activation='tanh',
                         solver='lbfgs', alpha=0.010, learning_rate_init = 0.001,
shuffle=False)
mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 100), activation='relu',
                         solver='lbfgs', alpha=0.010, learning_rate_init = 0.001,
shuffle=False) # span = 20
mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 100), activation='relu',
                         solver='lbfgs', alpha=0.005, learning_rate_init = 0.001,
shuffle=False) # span = 20 # best 1
mlpc = MLPClassifier(hidden_layer_sizes=(100, 200, 50), activation='relu',
                         solver='lbfgs', alpha=0.005, learning_rate_init = 0.001,
shuffle=False

# checking the performance of training data itself
prediction, bias, contributions = ti.predict(rf, numpy_df_train)
idx = pd.date_range(train_start_date, train_end_date)
predictions_df1 = pd.DataFrame(data=prediction[0:], index = idx,
columns=['prices'])
predictions_df1.plot()
train['prices'].plot()
```

Implementation of Kalman Filter Estimation of Mean in Python using PyKalman, Bokeh and NSEPy:

```
from math import pi
import pandas as pd
from bokeh.plotting import figure, show, output_notebook
from nsepy.archives import get_price_history
from datetime import date
from datetime import datetime
from pykalman import KalmanFilter

%matplotlib notebook

df = get_price_history(stock = 'TCS',
                       start = date(2015,1,1),
                       end = date(2017,04,19))
kf = KalmanFilter(transition_matrices = [1],
                  observation_matrices = [1],
                  initial_state_mean = df['Close'].values[0],
                  initial_state_covariance = 1,
                  observation_covariance=1,
                  transition_covariance=.01)
state_means,_ = kf.filter(df[['Close']].values)
state_means = state_means.flatten()
df["date"] = pd.to_datetime(df.index)

mids = (df.Open + df.Close)/2
spans = abs(df.Close-df.Open)

inc = df.Close > df.Open
dec = df.Open > df.Close
w = 12*60*60*1000 # half day in ms

output_notebook()

TOOLS = "pan,wheel_zoom,box_zoom,reset,save"

p = figure(x_axis_type="datetime", tools=TOOLS, plot_width=1000, toolbar_lo
cation="left",y_axis_label = "Price",
        x_axis_label = "Date")
```

```
p.segment(df.date, df.High, df.date, df.Low, color="black")
p.rect(df.date[inc], mids[inc], w, spans[inc], fill_color='green', line_color="gree
n")
p.rect(df.date[dec], mids[dec], w, spans[dec], fill_color='red', line_color="red")
p.line(df.date,state_means,line_width=1,line_color = 'blue',legend="Kalman filt
er")

p.title = "Implementation of Kalman Filter Estimation - TCS EOD chart"
p.xaxis.major_label_orientation = pi/4
p.grid.grid_line_alpha=0.3
show(p)
```

7. Results/Sample Output:

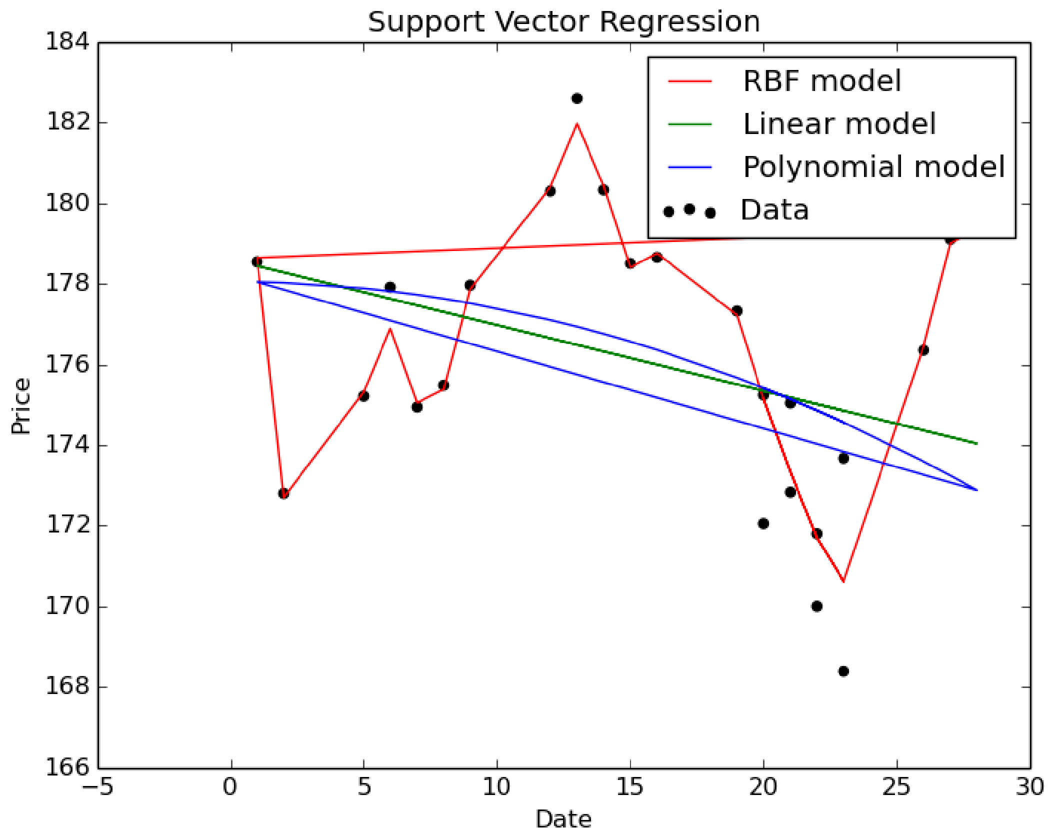
Decision Tree:

We are getting wrong output.

KNN Methodology:

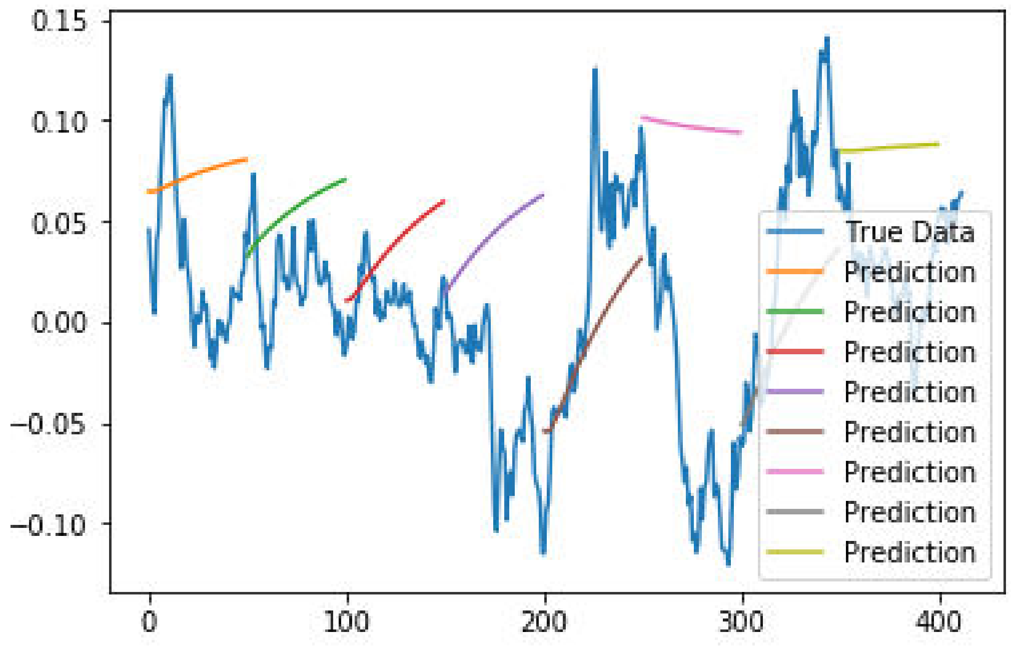
The value classified to unknown point is: 0

Support Vector Regression:



Deep Learning Model:

```
[[[ 0.        ]
  [ 0.00305228]
  [-0.00033845]
  ...
  [ 0.06331986]
  [ 0.06780923]
  [ 0.06386026]]

 [[ 0.        ]
  [ 0.00811193]
  [ 0.02051003]
  ...
  [ 0.01546984]
  [ 0.01262037]
  [ 0.00921522]]

 [[ 0.        ]
  [ 0.00232927]
  [ 0.00744792]
  ...
```
……



Reinforcement Learning and Sentiment Analysis(Random Forest Model, Linear Regression Model, Multi Layer Perceptron or Deep Neural Network Model):

2007-01-01	12469.971875	12469.971875	. What Sticks from '06. Somalia Orders Islamis...
2007-01-02	12472.245703	12472.245703	. Heart Health: Vitamin Does Not Prevent Death...
2007-01-03	12474.519531	12474.519531	. Google Answer to Filling Jobs Is an Algorith...
2007-01-04	12480.690430	12480.690430	. Helping Make the Shift From Combat to Commer...
2007-01-05	12398.009766	12398.009766	. Rise in Ethanol Raises Concerns About Corn a...
2007-01-06	12406.503255	12406.503255	. A Status Quo Secretary General. Best Buy and...
2007-01-07	12414.996745	12414.996745	. THE COMMON APPLICATION; Typo.com. Jumbo Bonu...
2007-01-08	12423.490234	12423.490234	. VW Group's Sales Rose Sharply in 2006. Conso...
2007-01-09	12416.599609	12416.599609	. The Claim: Hot Leftovers Should Cool at Roo...
2007-01-10	12442.160156	12442.160156	. Love Among the Ruins. Dell Says Plant a Tree...
2007-01-11	12514.980469	12514.980469	. The Computer With a TV, and a Family's Virtu...
2007-01-12	12556.080078	12556.080078	. Make Them Fight All of Us. Hire by the Contr...

2007-01-13	12562.707519	12562.707519	. Blair Urges Britain to Pursue an Aggressive ...
2007-01-14	12569.334961	12569.334961	. Smoke Damage. Mr. Spitzer’s Task on Court Re...
2007-01-15	12575.962403	12575.962403	. The Mentally Ill, Behind Bars. BP’s Chief to...
2007-01-16	12582.589844	12582.589844	. King Day in Atlanta, ‘the One Without Mrs. K...
2007-01-17	12577.150391	12577.150391	. Racial Hate Feeds a Gang War’s Senseless Kil...

……………………. •

	prices	**articles**
2007-01-01	12469	What Sticks from '06. Somalia Orders Islamist...
2007-01-02	12472	Heart Health: Vitamin Does Not Prevent Death ...
2007-01-03	12474	Google Answer to Filling Jobs Is an Algorithm...
2007-01-04	12480	Helping Make the Shift From Combat to Commerc...
2007-01-05	12398	Rise in Ethanol Raises Concerns About Corn as...
2007-01-06	12406	A Status Quo Secretary General. Best Buy and ...
2007-01-07	12414	THE COMMON APPLICATION; Typo.com. Jumbo Bonus...
2007-01-08	12423	VW Group’s Sales Rose Sharply in 2006. Consol...
2007-01-09	12416	The Claim: Hot Leftovers Should Cool at Room...
2007-01-10	12442	Love Among the Ruins. Dell Says Plant a Tree,...
2007-01-11	12514	The Computer With a TV, and a Family's Virtua...
2007-01-12	12556	Make Them Fight All of Us. Hire by the Contra...
2007-01-13	12562	Blair Urges Britain to Pursue an Aggressive F...
2007-01-14	12569	Smoke Damage. Mr. Spitzer’s Task on Court Ref...
2007-01-15	12575	The Mentally Ill, Behind Bars. BP’s Chief to ...
2007-01-16	12582	King Day in Atlanta, ‘the One Without Mrs. Ki...
2007-01-17	12577	Racial Hate Feeds a Gang War’s Senseless Kill...
2007-01-18	12567	Taliban Detainee Says Rebel Chief Hides in Pa...
2007-01-19	12565	Data Breach Could Affect Millions of TJX Shop...
2007-01-20	12536	Archives of Spin. H.P. Chief Defends Timing o...
2007-01-21	12506	Connecticut’s Diaspora. Son of Dogs Playing P

……………

2007-01-01	12469
2007-01-02	12472
2007-01-03	12474
2007-01-04	12480
2007-01-05	12398
2007-01-06	12406
2007-01-07	12414
2007-01-08	12423
2007-01-09	12416
2007-01-10	12442

…………

prices	**compound**	**neg**	**neu**	**pos**
2007-01-01	12469			
2007-01-02	12472			
2007-01-03	12474			
2007-01-04	12480			
2007-01-05	12398			
2007-01-06	12406			
2007-01-07	12414			
2007-01-08	12423			
2007-01-09	12416			
2007-01-10	12442			
2007-01-11	12514			
2007-01-12	12556			

………… • •

prices	**compound**	**neg**	**neu**	**pos**	
2007-01-01	12469	-0.9735	0.153	0.748	0.099
2007-01-02	12472	-0.9664	0.122	0.784	0.095

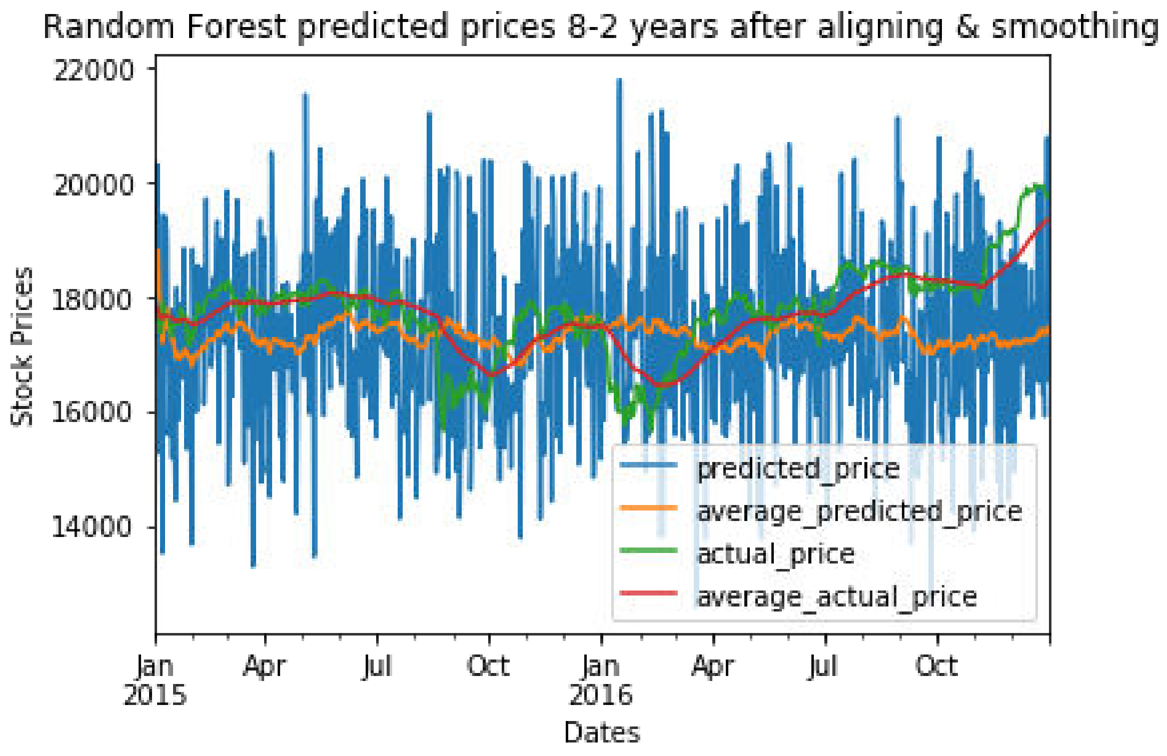
prices	compound	neg	neu	pos	
2007-01-03	12474	-0.9994	0.207	0.733	0.06
2007-01-04	12480	-0.9982	0.131	0.806	0.062
2007-01-05	12398	-0.9901	0.124	0.794	0.082
2007-01-06	12406	-0.965	0.134	0.771	0.094
2007-01-07	12414	-0.9975	0.193	0.739	0.069
2007-01-08	12423	-0.973	0.114	0.788	0.098
2007-01-09	12416	-0.9945	0.104	0.844	0.052
2007-01-10	12442	-0.9863	0.141	0.742	0.117
2007-01-11	12514	-0.9981	0.131	0.81	0.059

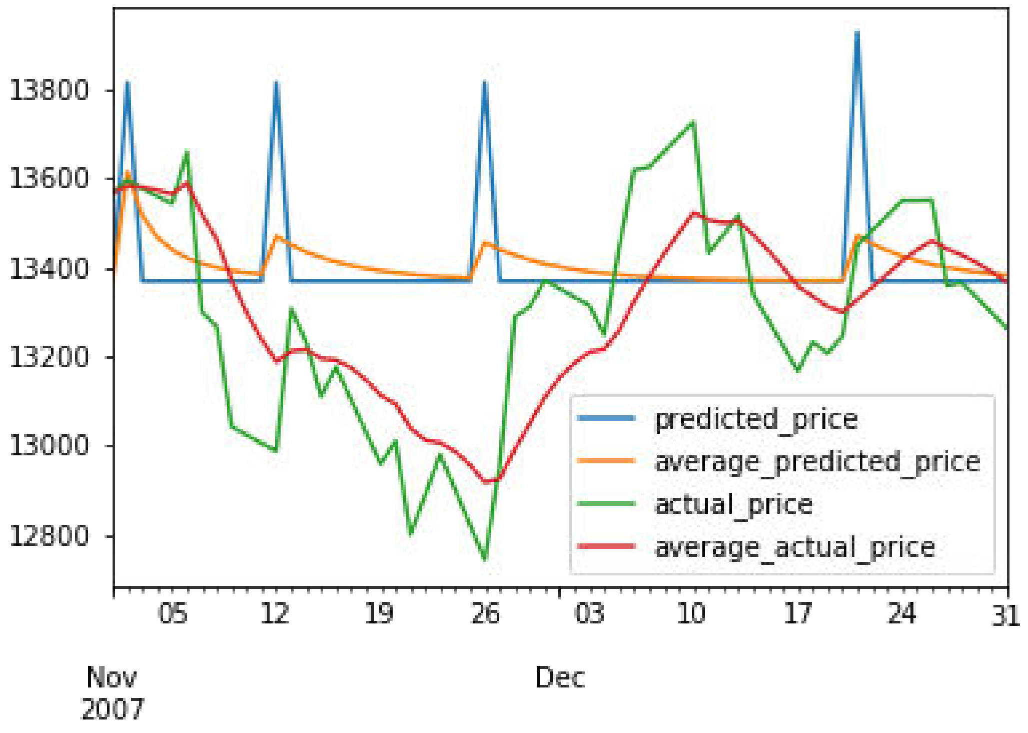
……… • •

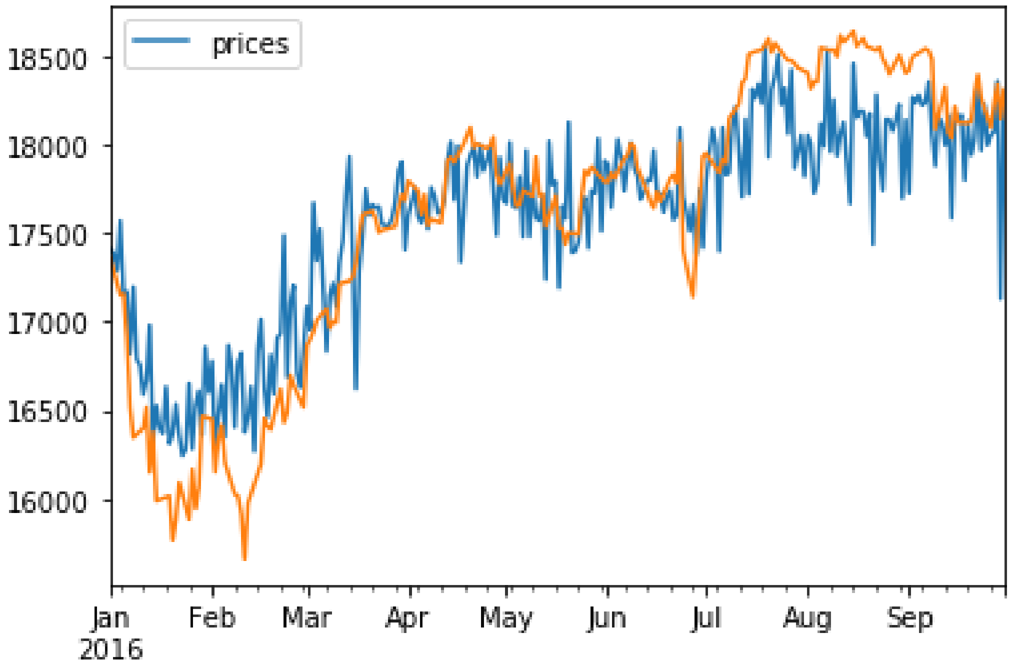
```
array([ 13641.5       ,  13461.6       ,  15840.38333333,  13780.
  ,
        10800.1       ,  13148.4       ,   9041.4       ,  14952.4
  ,
        12361.9       ,  14916.2       ,  14543.1       ,  11104.113333
33,
        11381.55      ,  12849.13333333,  10697.54      ,  13305.55
  ,
        10913.8       ,   9965.16666667,  13685.6       ,  12008.65
  ,
        11371.34      ,  13397.6       ,  12677.125     ,  12108.3
  ,
        14366.7       ,  12970.8       ,  10861.9       ,  12791.6
  ,
        11023.92      ,  13064.2       ,   9194.7       ,  14356.6
  ,
        12995.8       ,  13851.2       ,  11510.25      ,  14062.3
  ,
        12786.23333333,  12650.       ,  13515.8       ,  14025.
  ,
        11637.85      ,  12327.86666667,  15235.7       ,  13036.4
  ,
        13642.        ,  12938.1       ,  12299.05      ,  12517.4
  ,
        13859.17857143,  12800.6       ,  14177.87      ,  14851.2
  ,
        10956.8       ,  12583.35      ,  14543.80833333,  13524.
  ,
        14326.        ,  12712.7       ,  12912.63333333,  15375.
  ,
        10239.1       ,  11562.6       ,  13225.81666667,  11772.8
  ,
        13399.9       ,  14459.4       ,  13572.3       ,  15218.4
  ,
        13196.5        ,  12623.83333333,  13181.8        ,  14188.4
  ,
        12084.2        ,  10620.3        ,  12294.43333333,  14259.6
  ,
```
…. .

	prices
2015-01-01	13641.500000
2015-01-02	13461.600000
2015-01-03	15840.383333
2015-01-04	13780.000000
2015-01-05	10800.100000
2015-01-06	13148.400000
2015-01-07	9041.400000
2015-01-08	14952.400000
2015-01-09	12361.900000
2015-01-10	14916.200000
2015-01-11	14543.100000

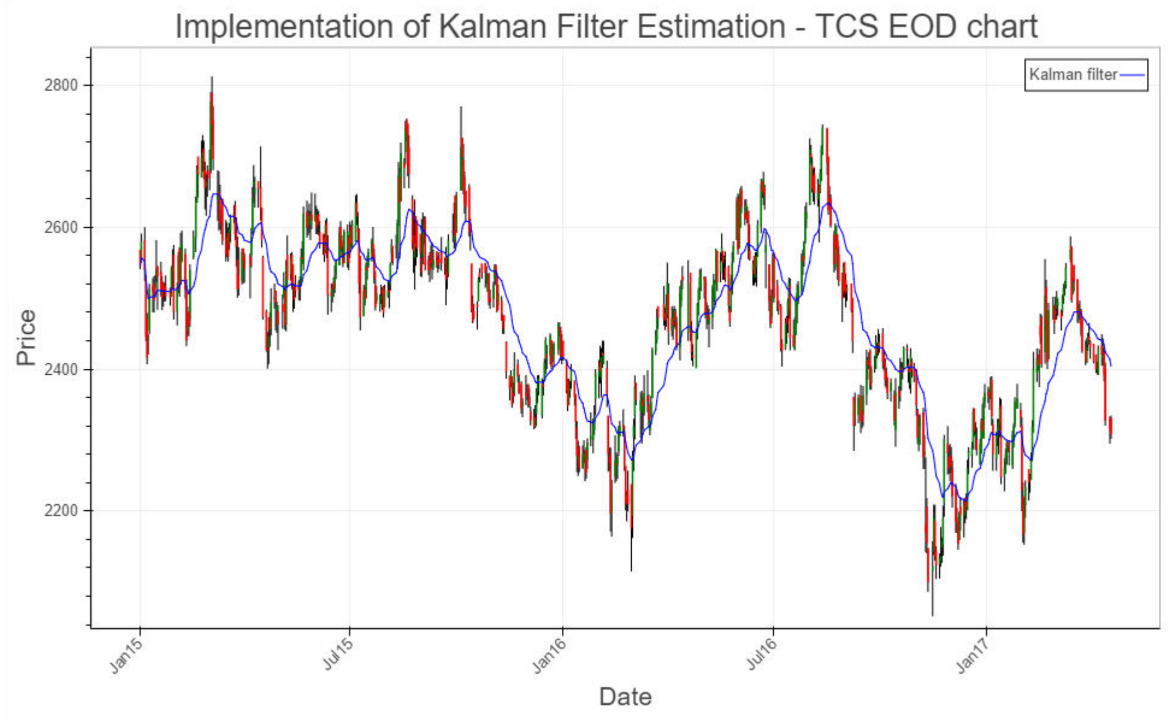
……………….


Random Forest predicted prices 8-2 years after aligning & smoothing
Stock Prices
22000
20000
18000
16000
14000
Jan 2015
Apr
Jul
Oct
Jan 2016
Apr
Jul
Oct
Dates
predicted_price
average_predicted_price
actual_price
average_actual_price


13800
13600
13400
13200
13000
12800
05
12
19
26
03
10
17
24
31
Nov 2007
Dec
predicted_price
average_predicted_price
actual_price
average_actual_price



Implementation of Kalman Filter Estimation of Mean in Python using PyKalman, Bokeh and NSEPy:



8.Conclusion:

We have implemented many a algorithms in our search for the optimal one. Wehave found the following two among them to give us better results:

1. SVR RBF Model
2. MLP Model

Our research is still going on and now we are trying to implement a suitable algorithm for Kalman Filtering so that we can fine-tune our results to some more extent.

Till now, we have faced both success and failure. But that is a part of research. We have found ways in which it would work as well as ways in which won't work.

Appendix:

Kindly refer to the "6. Implementation Details" section.

References:

1. Abhyankar, A., Copeland, L. S., & Wong, W. (1997). Uncovering nonlinear structure in real-time stockmarket indexes: The S&P 500, the DAX, the Nikkei 225, and the FTSE-100. Journal of Business & Economic Statistics, 15, 1–14.
2. Austin, M. Looney, C., & Zhuo, J. (1997). Security market timing using neural network models. New Review of Applied Expert Systems, 3, 3–14.
3. Box, G. E. P., & Jenkins, G. M. (1970). Time series analysis: Forecasting and control, Holden Day.
4. Brownstone, D., (1996). Using percentage accuracy to measure neural network predictions in stock market movements. Neurocomputing, 10, 237–250.
5. Brown, S. J., Goetzmann, W. N., & Kumar, A. (1998). The Dow Theory: William Peter Hamilton's track record reconsidered. Journal of Finance, 53, 1311–1333.
6. Desai, V. S., & Bharati, R., (1998). A comparison of linear regression and neural network methods for predicting excess returns on large stocks. Annals of Operations Research, 78, 127–163.
7. Engle, RF, (1982), Autoregression Conditional Heteroscedasticity Estimates of the variance of UK Inflation, Econometrica, 50, 987-1008.
8. Fama, Eugene (1970), "Efficient Capital Markets: A Review of Theory and Empirical Work," Journal of Finance.
9. Fernandez-Rodriguez, F., Gonzalez-Martel, C., & Sosvilla-Rivero, S. (2000). On the profitability of technical trading rules based on artificial neural networks: Evidence from the Madrid stock market. Economic Letters, 69, 89–94.
10. Goutam Dutta, Pankaj Jha, Arnab Kumar Laha and Neeraj Mohan Artificial Neural Network Models for Forecasting Stock Price Index in the Bombay Stock Exchange Journal of Emerging Market Finance,
Vol. 5, No. 3, 283-295.
11. Gleick James, Chaos: The Amazing Science of the Unpredictable, Penguin.
12. Hakins, Simon., Neural Networks, A Comprehensive Foundation, 2nd Edition, Prentice Hall International.
13. Huang Wei., Wang Shouyang., Yu Lean., Bao1 Yukun., and Wang Lin., (2006) A New Computational Method of Input Selection for Stock Market Forecasting with Neural Networks, Lecture Notes In Computer Science.
14. Jung-Hua Wang; Jia-Yann Leu Stock market trend prediction using ARIMA-based neural networks Neural Networks, 1996., IEEE International Conference on Volume 4, Issue , 3-6 Jun 1996 Page(s):2160 – 2165 vol.4.
15. Kim, Kyoung-Jae. Artificial neural networks with feature transformation based on domain knowledge for the prediction of stock index futures Intelligent Systems in Accounting, Finance & Management,
Vol. 12, Issue 3 , Pages 167 – 176.
16. Kim, S, K., and S.H Chun (1998) "Graded Forecasting Using An Array Of Bipolar Predictions: Application Of Probabilistic Neural Network To A Stock Market Index." International Journal of Forecasting, 14, 323-337.
17. Kim, K, J and I Han (2000) Genetic algorithm approach to feature discretization in artificial neural network for the prediction of stock price index." Published by Elsevier science, Ltd., Experts systems with application, 19, 125-132.
18. Kuvayev Leonid, (1996) Predicting Financial Markets with Neural Networks.

19. Leigh, W.Paz, M., & Purvis, R. (2002). An analysis of a hybrid neural network and pattern recognition technique for predicting short-term increases in the NYSE composite index. Omega-International Journal of Management Science, 30, 69–76.
20. Malkiel, Burton G. (2003). "The Efficient Market Hypothesis and Its Critics". CEPS Working paper No
91.
21. Mendelsohn Louis B. (2000) Trend Forecasting with Technical Analysis: Unleashing the Hidden Power of Intermarket Analysis to Beat the Market, Marketplace Books.
22. Marius Januskevicius, Testing Stock Market Efficiency Using Neural Network
23. Pan H. P. (2003), A joint review of Technical and Quantitative Analysis of Financial Markets Towards a Unified Science of Intelligent Finance, Paper for the 2003 Hawaii International Conference on Statistics and Related Fields.
24. Pan H.P. (2004): A swingtum Theory of Intelligent Finance for swing trading and momentum trading, 1st International workshop on Intelligent Finance.
25. Pan Heping,Tilakaratne C and Yearwood John(2005): "Predicting Australian Stock Market Index using Neural Networks Exploiting Dynamic Swings and Inter-market Influences" Journal of Research and Practice in Information Technology, Vol 37, No 1
26. Panda, C. and Narasimhan, V. (2006) Predicting Stock Returns : An Experiment of the Artificial Neural Network in Indian Stock Market South Asia Economic Journal, Vol. 7, No. 2, 205-218.
27. Pantazopoulos, K. N., Tsoukalas, L. H., Bourbakis, N. G., Brun, M. J., & Houstis, E. N. (1998). Financial prediction and trading strategies using neurofuzzy approaches. IEEE Transactions on Systems, Man, and Cybernetics-PartB: Cybernetics, 28, 520–530.
28. Roman, Jovina and Jameel, Akhtar Backpropagation and Recurrent Neural Networks in Financial Analysis of Multiple Stock Market Returns.
29. Refenes, Zapranis, and Francis, (1994) Journal of Neural Networks, Stock Performance Modeling Using Neural Networks: A Comparative Study with Regression Models, Vol. 7, No. 2,. 375-388.
30. Saad, E.W.; Prokhorov, D.V.; Wunsch, D.C., II (1998) Comparative study of stock trend prediction using time delay, recurrent and probabilistic neural networks IEEE Transactions on Neural Networks, Volume 9, Issue 6, Page(s): 1456 - 1470
31. Schoeneburg, E., (1990) Stock Price Prediction Using Neural Networks: A Project Report, Neurocomputing, vol. 2, 17-27.
32. Siekmann, S., Kruse, R., & Gebhardt, J. (2001). Information fusion in the context of stock index prediction. International Journal of Intelligent Systems, 16, 1285–1298.
33. S.-I. Wu and H. Zheng (USA) (2003) Can Profits Still be made using Neural Networks in Stock Market? (410) Applied Simulation and Modeling
34. Tsaih, R., Hsu, Y., & Lai, C. C., (1998). Forecasting S&P 500 stock index futures with a hybrid AI system. Decision Support Systems, 23, 161–174.
35. https://www.marketcalls.in/python/implementation-kalman-filter-estimation-mean-python-using-pykalman-bokeh-nsepy.html
36. http://www.haikulabs.com/pmdwkf26.htm
37. https://pykalman.github.io/
38. https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd
39. https://deeplearning4j.org/neuralnet-overview
40. https://towardsdatascience.com/linear-regression-detailed-view-ea73175f6e86
41. https://deeplearning4j.org/deepreinforcementlearning
42. https://keras.io/
43. https://www.tensorflow.org/
44.https://www.csdojo.io/
45. http://www.sirajraval.com/
46. https://pandas.pydata.org/
47. https://www.scipy.org/
48. http://www.numpy.org/
49. https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average
50. https://github.com/

51.https://quora.com