SUPERVISED LEARNING APPROACH FOR FORECASTING TAXI TRAVEL DEMAND

A Project Report

submitted by

A SHIVA

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **SUPERVISED LEARNING APPROACH FOR FORECASTING TAXI TRAVEL DEMAND**, submitted by **A Shiva**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona-fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Gaurav Raina
Research Guide
Associate Professor
Department of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 20th June 2018

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards several people who enabled me to reach this far with their timely guidance, support and motivation.
First and foremost, I offer my earnest gratitude to my guide, Prof. Gaurav Raina whose knowledge and dedication has inspired me to work efficiently on the project and I thank him for invaluable comments and suggestions throughout the course of this project.
I feel privileged for all the motivational and technical discussions with Neema which helped me reach where I am today.
I consider myself fortunate to work with a fun-loving yet sincere group of team-mates. I would like to thank Anurekha, Ramya, SelvaSandhiya and Gangadher for making this journey enjoyable.
My deepest gratitude to my mother and father for their tremendous amount of support, encouragement, patience, and prayers.

ABSTRACT

KEYWORDS: Travel demand forecasting, Time series, Supervised learning (SL), Data analysis.

Transportation planning is a collaborative process involving public and private agencies. Public transport remains the primary mode of transport for most of the population, and Indias public transport system is among the most heavily used in the world. However, this sector has not been able to keep up with the rising demand. Hence, to match transportation demand, an alternate mode of transport such as taxi services are essential. In India, there has been a dramatic increase in the popularity for mobile application based taxi hailing services. In this work, we use Supervised learning algorithms to forecast taxi travel demand, in the context of a mobile application based taxi service. In particular, we model the passenger demand density at various locations in the city of Bengaluru, India. Using the data, we first shortlist time series, machine learning and deep learning algorithms that suit our application. Analyses the performance of these models by using Root Mean Squared Error (RMSE), R Squared (R^2) and Mean Absolute Percentage Error (MAPE) as the performance metrics. We then compare the time series models with the machine learning and deep learning models.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GPS	Global Positioning System
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
$\mathbf{R^2}$	R Squared
TS	Time Series
AR	Autoregression
MA	Moving Average
ARIMA	AutoRegressive Integrated Moving Average
SL	Supervised Learning
ML	Machine Learning
DL	Deep Learning
LR	Linear Regression
LASSO	Least Absolute Shrinkage and Selection Operator Regression
EN	ElasticNet
KNN	k-Nearest Neighbors
CART	Classification and Regression Trees
SVM	Support Vector Machines
SVR	Support Vector Regression
MLP	Multilayer Perceptron
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
CEC	Constant error carrousel
Std	Standard deviation

CHAPTER 1

INTRODUCTION

Traffic is the pulse of a city that impacts the daily life of millions of people. One of the most fundamental questions for future smart cities is how to build an efficient transportation system. To address this question, a critical component is an accurate demand prediction model. The better we can predict demand on travel, the better we can pre-allocate resources to meet the demand. From the Global Positioning System (GPS) data generated by taxi services apps, the demand (number of taxi booking requests) generated by passengers can be obtained [3].

In literature, there has been a long line of studies in traffic data prediction, including traffic volume, taxi pick-ups, and traffic in-out flow volume. To predict traffic, time series prediction methods have frequently been used. Representatively, autoregressive integrated moving average (ARIMA) and its variants have been widely applied for traffic prediction [8, 10, 11]. Recent advances in deep learning have enabled researchers to model the complex nonlinear relationships and have shown promising results in computer vision and natural language processing fields [7]. This success has inspired to attempt deep learning techniques on traffic prediction problems.

In this work, we tackle the problem of analysing and modeling this taxi travel demand, using time series and supervised learning models. We then compare time series models with supervised learning models. We build traditional time series prediction models [6] and supervised learning models on the assumption that the present and the future demand would have some correlation with the past demand, and can be represented as the function of the past data. Indeed, accurate forecasting of taxi demand and supply is crucial for mitigating demand-supply imbalance, by routing idle drivers to passenger hotspots. We model the demand by employing supervised learning techniques on a real-world dataset containing the taxi demand generated for the city of Bengaluru, India. This dataset was acquired from a leading app-based taxi service provider in India. Using these supervised learning models, we predict future demand to reduce the demand-supply imbalance.

Traffic analysis has been an active research area over the recent years. A fundamental approach for traffic analysis is to understand human mobility, i.e., draw statistical inferences from empirical data [2]. We are particularly concerned with taxi demand analysis [9]. Empirically, we noticed that Indian traffic undergoes changes very rapidly from area to area. Each location in the city is identified by a unique set of alphanumeric characters, known as a geohash. A 6-level geohash has six alphanumeric characters and encloses an area of approximately 1.2 km * 0.6 km. In this work, the demand density for the most heavily used geohashes in the city of Bengaluru, India is modeled and fitted using time series and supervised learning models.

1.1 Organization of the report

The rest of the document is organized as follows. In chapter 2 we discuss on re-framing time series forecasting as a supervised learning problem, followed by discuss the models that worked for our dataset in chapter 3 and performance metrics. In chapter 4 a discussion on the model fitting and results. In chapter 5 applications of our work are discussed. Finally, we summarize in chapter 6.

CHAPTER 2

TIME SERIES AS SUPERVISED LEARNING

Time series forecasting can be framed as a supervised learning problem. This re-framing of our time series data allows us access to the suite of standard linear and non-linear algorithms on our problem.

2.1 Supervised Machine Learning

The majority of practical machine learning uses supervised learning. Supervised learning is where we have input variables (X) and an output variable (y) and we use an algorithm to learn the mapping function from the input to the output.

$$y = f(X) \tag{2.1}$$

The goal is to approximate the real underlying mapping so well that when we have new input data (X), we can predict the output variables (y) for that data. It is called supervised learning because an algorithm iteratively makes predictions on the training data and is corrected by making updates. Learning stops when the algorithm achieves an acceptable level of performance.

2.2 Sliding Window

It is method for framing a time series dataset. Given a sequence of numbers for a time series dataset, we can restructure the data to look like a supervised learning problem. We can do this by using previous time steps as input variables and use the next time step as the output variable. Let's make this concrete with our dataset. We have a time series data for Geohash-1 (tdr1w4) as follows:

Table 2.1: Time series data for Geohash-1 (tdr1w4)

Index (Time in hours)	**Demand**
0	74
1	89
2	127
3	162
4	183
5	221

We can restructure this time series dataset as a supervised learning problem by using the value at the previous time step to predict the value at the next time-step.

Table 2.2: Time series dataset as a supervised learning problem

Index (Time in hours)	**X**	**y**
0	Nan	74
1	74	89
2	89	127
3	127	162
4	162	183
5	183	221

Take a look at the above transformed dataset and compare it to the original time series. Here are some observations:

- We can see that the previous time step is the input (X) and the next time step is the output (y) in our supervised learning problem.
- We can see that the order between the observations is preserved, and must continue to preserved when using this dataset to train a supervised model.
- We can see that we have no previous value that we can use to predict the first value in the sequence. We will delete this row as we cannot use it.
- We can also see that we do not have a known next value to predict for the last value in the sequence. We may want to delete this value while training our supervised model also.

The number of previous time steps is called the window width or size of the lag.

CHAPTER 3

MODEL SELECTION

There are many models to choose from. We must know whether the predictions for a given algorithm are good or not. But how do you know? The answer is to use a baseline prediction algorithm. A baseline prediction algorithm provides a set of predictions that you can evaluate as we would any predictions for our problem, such as RMSE.

3.0.1 Baseline model

It is important to establish baseline performance on a predictive modeling problem. A baseline provides a point of comparison for the more advanced methods that you evaluate later. The Zero Rule Algorithm is a better baseline. A good default prediction for real values is to predict the central tendency. This could be the mean or the median. A good default is to use the mean (also called the average) of the output value observed in the training data. Below is a function to do that named Zero rule algorithm and it works by calculating the mean value for the observed output values.

$$Mean = \frac{\sum_{i=0}^{n} value_i}{count(values)} \tag{3.1}$$

Once calculated, the mean is then predicted for each row in the training data.

3.1 Time Series Algorithms

3.1.1 Autoregression (AR) Model

Autoregression is a time series model that uses observations from previous time steps as input to a regression equation to predict the value at the next time step. It is a very simple idea that can result in accurate forecasts on a range of time series problems.

The notation $AR(p)$ indicates an autoregressive model of order p. The $AR(p)$ model is defined as

$$X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t \tag{3.2}$$

where $\varphi_1, \ldots, \varphi_p$ are the parameters of the model, c is a constant, and ε_t is white noise process with zero mean and constant variance σ_ε^2.

$AR(1)$ indicates an autoregressive model of order 1. The $AR(1)$ model is defined as

$$X_t = c + \varphi X_{t-1} + \varepsilon_t \tag{3.3}$$

If AR model parameter $\varphi = 1$, then this process is Random walk and if $\varphi = 0$, then this process is White noise. For model stable and stationary, φ should be between -1 and 1.

Because the regression model uses data from the same input variable at previous time steps, it is referred to as an autoregression (regression of self).

3.1.2 Moving Average (MA) Model

The residual errors from forecasts on a time series provide another source of information that we can model. Residual errors themselves form a time series that can have temporal structure. A simple autoregression model of this structure can be used to predict the forecast error, which in turn can be used to correct forecasts. This type of model is called a moving average model.

The difference between what was expected and what was predicted is called the residual error. It is calculated as:

$$\text{residual error} = \textit{expected} - \textit{predicted}$$

Just like the input observations themselves, the residual errors from a time series can have temporal structure like trends, bias, and seasonality. An autoregression of the residual error time series is called a Moving Average (MA) model. Think of it as the sibling to the autoregressive (AR) process, except on lagged residual error rather than lagged raw observations.

Correct Predictions with a Model of Residuals

we can add the expected forecast error to a prediction to correct it and in turn improve the skill of the model.

$$\text{improved forecast} = \text{forecast} + \text{estimated error}$$

Lets make this concrete with an example. Lets say that the expected value for a time step is 10. The model predicts 8 and estimates the error to be 3. The improved forecast would be:

$$\begin{aligned}\text{improved forecast} &= \text{forecast} + \text{estimated error}\\ &= 8 + 3\\ &= 11\end{aligned}$$

This takes the actual forecast error from 2 units to 1 unit.

3.1.3 ARIMA Model

An ARIMA model [6] is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- AR: Autoregression. A model that uses the dependent relationship between an observation and some number of lagged observations.
- I: Integrated. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- MA: Moving Average. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of ARIMA(p,d,q) where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used. The parameters of the ARIMA model are defined as follows:

- p: The number of lag observations included in the model, also called the lag order.
- d: The number of times that the raw observations are differenced, also called the degree of differencing.
- q: The size of the moving average window, also called the order of moving average.

A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

3.2 Machine Learning Algorithms

3.2.1 Simple Linear Regression (Ordinary Least Squares)

Linear regression is a prediction method that is more than 200 years old. Simple linear regression is a great first machine learning algorithm to implement as it requires you to estimate properties from your training dataset.

Linear regression [13] assumes a linear or straight line relationship between the input variables (X) and the single output variable (y). More specifically, that output (y) can be calculated from a linear combination of the input variables (X). When there is a single input variable, the method is referred to as a simple linear regression. In simple linear regression we can use statistics on the training data to estimate the coefficients required by the model to make predictions on new data. The line for a simple linear regression model can be written as:

$$y = b_0 + b_1 * x \tag{3.4}$$

Where, b_0 and b_1 are the coefficients we must estimate from the training data.

$$b_1 = \frac{\sum_{i=0}^{n}((x_i - mean(x)) * (y_i - mean(y)))}{\sum_{i=0}^{n}(x_i - mean(x))^2} = \frac{covariance(x,y)}{variance(x)} \tag{3.5}$$

$$b_0 = mean(y) - b_1 * mean(x) \tag{3.6}$$

Where, the i refers to the value of the ith value of the input x or output y.

Linear Regression fits a linear model with coefficients $b = (b_1, ..., b_p)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_{b} ||X_b - y||_2{}^2 \tag{3.7}$$

However, coefficient estimates for Ordinary Least Squares rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the design matrix becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed response, producing a large variance. This situation of multicollinearity can arise.

Complexity : This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

3.2.2 Ridge Regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares (also called the $\ell_2 - norm$).

$$\min_{b} ||X_b - y||_2{}^2 + \alpha||b||_2{}^2 \tag{3.8}$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

Complexity: This method has the same order of complexity as linear regression.

3.2.3 LASSO Regression

The Least Absolute Shrinkage and Selection Operator (or LASSO for short) is a modification of linear regression, like ridge regression, where the loss function is modified to minimize the complexity of the model measured as the sum absolute value of the coefficient values (also called the $\ell_1 - norm$).

It is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights.

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\underset{b}{min} \frac{1}{2n_{samples}} ||X_b - y||_2^2 + \alpha||b||_1 \tag{3.9}$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha||b||_1$ added, where α is a constant and $||b||_1$ is the $\ell_1 - norm$ of the parameter vector.

3.2.4 ElasticNet Regression

ElasticNet is a linear regression model trained with L1 and L2 prior as regularizer i.e., combines the properties of both Ridge Regression and LASSO regression. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. We control the convex combination of L1 and L2 using the l1-ratio parameter (ρ).

The objective function to minimize is in this case

$$\underset{b}{min} \frac{1}{2n_{samples}}||X_b - y||_2^2 + \alpha\rho||b||_1 + \frac{\alpha(1-\rho)}{2}||b||_2^2 \tag{3.10}$$

3.2.5 K-Nearest Neighbors (KNN)

A simple but powerful approach for making predictions is to use the most similar historical examples to the new data. This is the principle behind the k-Nearest Neighbors algorithm.

When a prediction is required, the k-most similar records to a new record from the training dataset are then located. From these neighbors, a summarized prediction is made. Similarity between records can be measured many different ways. A problem or data-specific method can be used. Generally, with tabular data, a good starting point is the Euclidean distance.

Once the neighbors are discovered, the summary prediction can be made by returning the most common outcome or taking the average. As such, KNN can be used for classification or regression problems. There is no model to speak of other than holding the entire training dataset. Because no work is done until a prediction is required, KNN is often referred to as a *lazy learning* method.

Following steps will give you the foundation that you need to implement the k-Nearest Neighbors algorithm:

- **Euclidean Distance :** The first step needed is to calculate the distance between two rows in a dataset. Rows of data are mostly made up of numbers and an easy way to calculate the distance between two rows or vectors of numbers is to draw a straight line. This makes sense in 2D or 3D and scales nicely to higher dimensions. We can calculate the straight line distance between two vectors using the Euclidean distance measure. It is calculated as the square root of the sum of the squared differences between the two vectors.

$$distance = \sqrt{\sum_{i=0}^{n}(x_{1i} - x_{2i})^2} \tag{3.11}$$

Where x1 is the first row of data, x2 is the second row of data and i is the index to a specific column as we sum across all columns. With Euclidean distance, the smaller the value, the more similar two records will be. A value of 0 means that there is no difference between two records.

Now it is time to use the distance calculation to locate neighbors within a dataset.

- **Get Neighbors :** Neighbors for a new piece of data in the dataset are the k closest instances, as defined by our distance measure. To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using Euclidean distance. Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbors.

  Now that we know how to get neighbors from the dataset, we can use them to make predictions.

3.2.6 Classification and Regression Trees (CART or decision trees)

CART construct a binary tree from the training data. This is the same binary tree from algorithms and data structures, nothing too fancy (each node can have zero, one or two child nodes). A node represents a single input variable (X) and a split point on that variable, assuming the variable is numeric. The leaf nodes (also called terminal nodes) of the tree contain an output variable (y) which is used to make a prediction. CART use the training data to select the best points to split the data in order to minimize a cost metric. The default cost metric for regression decision trees is the mean squared error.

3.2.7 Support Vector Regression (SVR)

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.

- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression (SVR). The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

3.3 Deep Learning Algorithms

3.3.1 Multilayer Perceptron (MLP)

A Perceptron is a single neuron model that was a precursor to larger neural networks. It is a field of study that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. Mathematically, MLPs are capable of learning any mapping function and have been proven to be a universal approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multilayered structure of the networks.

Neurons

The building block for neural networks are artificial neurons. These are simple computational units that have weighted input signals and produce an output signal using an activation function.

Neuron Weights

weights on the inputs are very much like the coefficients used in a regression equation. Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0 and it too must be weighted. For example, a neuron may have two inputs in which case it requires three weights. One for each input and one for the bias.

Activation

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function. An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal. Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0. Traditionally nonlinear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Nonlinear functions like the logistic function also called the sigmoid function were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called Tanh that outputs the same distribution over the range -1 to +1.

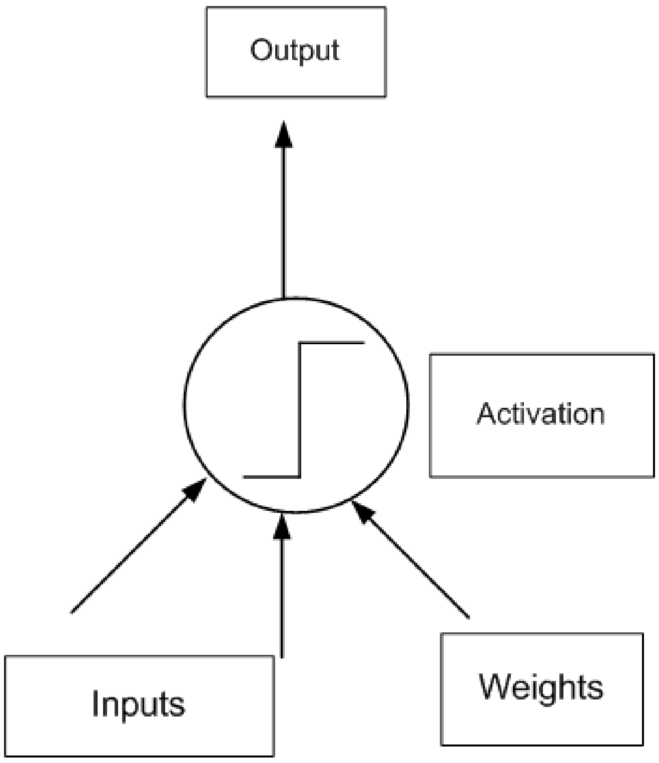


Figure 3.1: Model of simple neuron

Networks of Neurons

Neurons are arranged into networks of neurons. A row of neurons is called a layer and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.

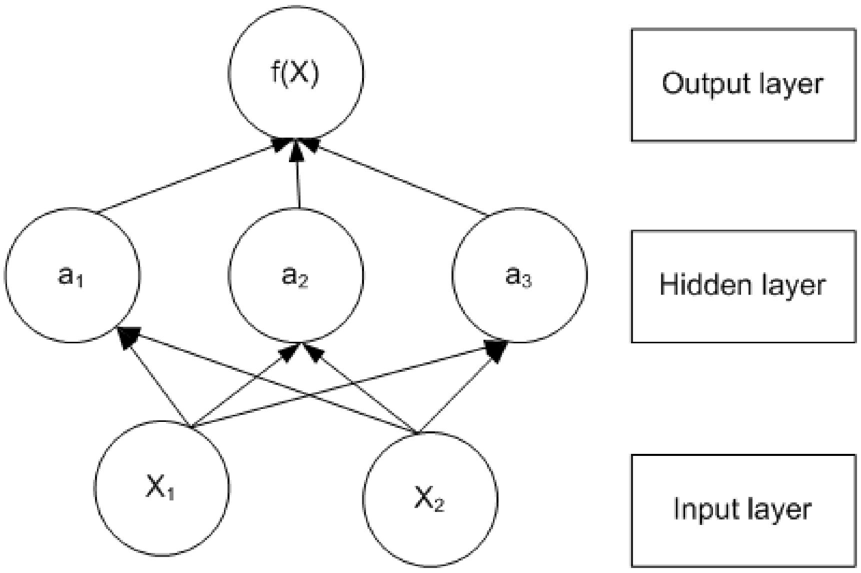


Figure 3.2: Model of a Simple Network

Input or Visible Layers

The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These are not neurons as described above, but simply pass the input value through to the next layer.

Hidden Layers

Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

Output Layer

The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem. The choice of activation function in the output layer is strongly constrained by the type of problem that you are modelling.

MLPs approximate a mapping function from input variables to output variables. This general capability is valuable for sequence prediction problems (notably time series forecasting) for a number of reasons [1].

- Robust to Noise: Neural networks are robust to noise in input data and in the mapping function and can even support learning and prediction in the presence of missing values.
- Non-linear: Neural networks do not make strong assumptions about the mapping function and readily learn linear and non-linear relationships.

More specifically, MLPs can be configured to support an arbitrary defined but fixed number of inputs and outputs in the mapping function. This means that:

- Multivariate Inputs. An arbitrary number of input features can be specified, providing direct support for multivariate prediction.
- Multi-Step Outputs. An arbitrary number of output values can be specified, providing direct support for multi-step and even multivariate prediction.

This capability overcomes the limitations of using classical linear methods (think tools like **ARIMA** for time series forecasting). For these capabilities alone, feedforward neural networks are used for time series forecasting [12].

Complexity: Suppose there are n training samples, m features, k hidden layers, each containing h neurons - for simplicity, and o output neurons. The time complexity of backpropagation is $O(n \cdot m \cdot h^k \cdot o \cdot i)$, where i is the number of iterations. Since backpropagation has a high time complexity, it is advisable to start with smaller number of hidden neurons and few hidden layers for training.

Promise of Recurrent Neural Networks

The Long Short-Term Memory, or LSTM, network is a type of Recurrent Neural Network. Recurrent Neural Networks, or RNNs for short, are a special type of neural network designed for sequence problems. Given a standard feedforward MLP network, an RNN can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal latterly (sideways) in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on. The recurrent connections add state or memory to the network and allow it to learn and harness the ordered nature of observations within input sequences.

Recurrent neural networks contain cycles that feed the network activations from a previous time step as inputs to the network to influence predictions at the current time step [4]. These activations are stored in the internal states of the network which can in principle hold long-term temporal contextual information. This mechanism allows RNNs to exploit a dynamically changing contextual window over the input sequence history.

The promise of recurrent neural networks is that the *temporal dependence and contextual information* in the input data can be learned [5].

3.3.2 Long Short-Term Memory (LSTM)

The key technical historical challenge faced with RNNs is how to train them effectively. Experiments show how difficult this was where the weight update procedure resulted in weight changes that quickly became so small as to have no effect (vanishing gradients) or so large as to result in very large changes or even overflow (exploding gradients). LSTMs overcome this challenge by design. An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units - the input, output and forget gates - that provide continuous analogues of write, read and reset operations for the cells.

LSTM Weights

A memory cell has weight parameters for the input, output, as well as an internal state that is built up through exposure to input time steps.

- Input Weights: Used to weight input for the current time step.
- Output Weights: Used to weight the output from the last time step.
- Internal State: Internal state used in the calculation of the output for this time step.

LSTM Gates

The key to the memory cell are the gates. These too are weighted functions that further govern the information flow in the cell. There are three gates:

- Forget Gate: Decides what information to discard from the cell.
- Input Gate: Decides which values from the input to update the memory state.
- Output Gate: Decides what to output based on input and the memory of the cell.

The forget gate and input gate are used in the updating of the internal state. The output gate is a final limiter on what the cell actually outputs. It is these gates and the consistent data flow called the constant error carrousel or CEC that keep each cell stable (neither exploding or vanishing). Unlike a traditional MLP neuron, it is hard to draw an LSTM memory unit cleanly.

3.4 Metrics for Performance Evaluation

3.4.1 Root Mean Squared Error (RMSE)

A popular way to calculate the error in a set of regression predictions is to use the Root Mean Squared Error. the metric is sometimes called Mean Squared Error or MSE, dropping the Root part from the calculation and the name. RMSE is calculated as the square root of the mean of the squared differences between actual outcomes and predictions. Squaring each error forces, the values to be positive, and the square root of the mean squared error returns the error metric back to the original units for comparison. RMSE provides a gross idea of the magnitude of error.

$$RMSE = \sqrt{\frac{\sum_{i=0}^{n}(predicted_i - actual_i)^2}{TotalPredictions}} \tag{3.12}$$

3.4.2 R Squared ($\mathbf{R^2}$)

The R^2 metric provides an indication of the goodness of fit of a set of predictions to the actual values. In statistical literature this measure is called the coefficient of determination. This is a value between 0 and 1 for no-fit and perfect fit respectively.

3.4.3 Mean Absolute Percentage Error (MAPE)

It gives an idea of how wrong the predictions were. The measure gives an idea of the magnitude of the error, but no idea of the direction (e.g. over or under predicting). A value of 0 indicates no error or perfect predictions.

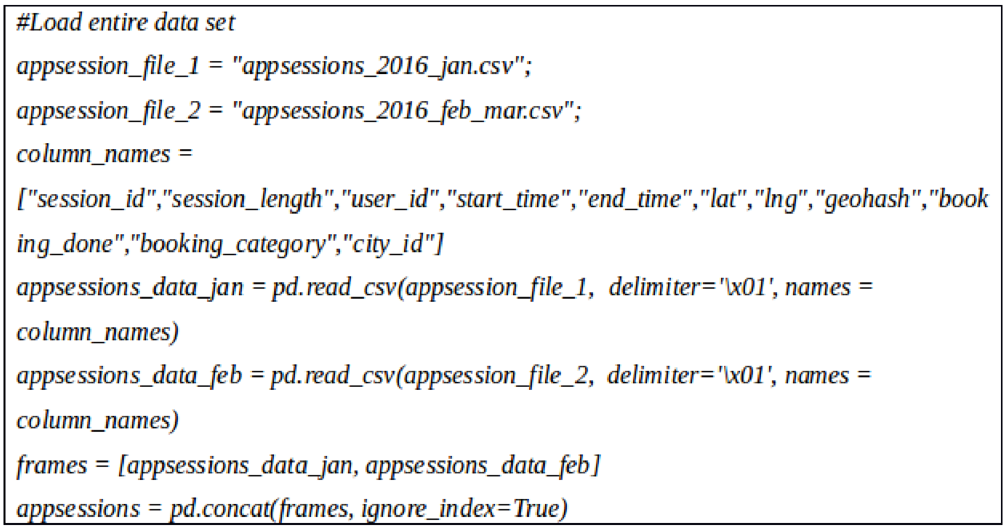
$$M = \frac{100}{N} \sum_{i=0}^{n} \mid \frac{A_i - F_i}{A_i} \mid \tag{3.13}$$

Where N is number of samples, A_i is the actual value and F_i is the forecast or predicted value at time i.

CHAPTER 4

DEMAND MODELING

We worked with a rich dataset spanning over two months, with nearly 7 million taxi trips each month. The data was acquired from a leading Indian transportation company dealing with app-based taxi rental services. It contains the taxi booking demand generated by the public, along with user identification numbers, location in the form of geohash and time.



```
#Load entire data set
appsession_file_1 = "appsessions_2016_jan.csv";
appsession_file_2 = "appsessions_2016_feb_mar.csv";
column_names =
["session_id","session_length","user_id","start_time","end_time","lat","lng","geohash","book
ing_done","booking_category","city_id"]
appsessions_data_jan = pd.read_csv(appsession_file_1,  delimiter='\x01', names =
column_names)
appsessions_data_feb = pd.read_csv(appsession_file_2,  delimiter='\x01', names =
column_names)
frames = [appsessions_data_jan, appsessions_data_feb]
appsessions = pd.concat(frames, ignore_index=True)
```

Let’s look at the few rows of the dataset.

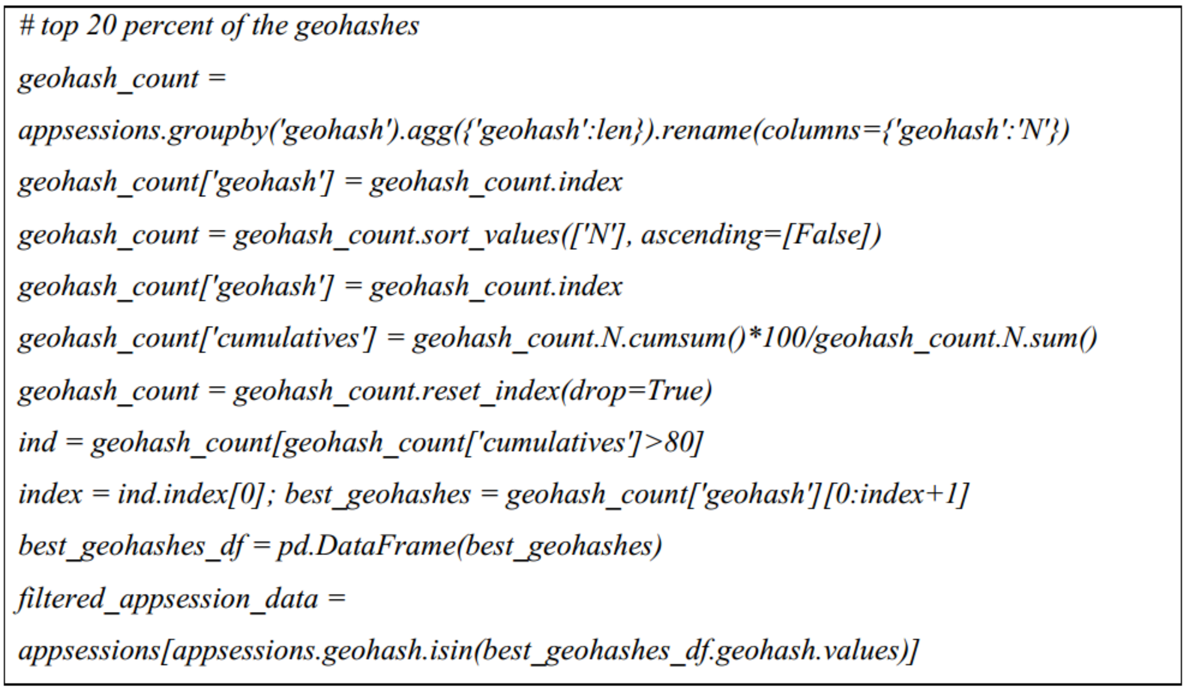
Table 4.1: Few observations of the dataset

Session id	**Session length**	**User id**	**Start time**	**End time**	**Latitude**	**Longitude**	**Geohash**	**Booking cate-gory**	**City id**
21952659-2016-02-27 12:20:44	0.000000	21952659	2016-02-27T12:20:44	2016-02-27T12:20:44	12.956159	77.487059	tdr1eq	None	3
24785741-2016-02-27 12:43:41	1.700000	24785741	2016-02-27T12:43:41	2016-02-27T12:45:23	13.071537	77.586420	tdr4me	None	3
24584002-2016-02-27 12:21:11	1.300000	24584002	2016-02-27T12:21:11	2016-02-27T12:22:29	13.018269	77.531717	tdr4h3	None	3
23525539-2016-02-27 12:56:53	6.733333	23525539	2016-02-27T12:56:53	2016-02-27T13:03:37	12.936587	77.699068	tdr385	None	3
22885955-2016-02-27 12:48:39	171.3667	22885955	2016-02-27T12:48:39	22016-02-27T15:40:01	12.949847	77.700600	tdr38j	None	3

The goal of our work is to model and predict hourly demand generated at a 6-level geohash, so that taxi allocation can be performed every hour. Every city can be divided into geohashes for the ease of allocating taxis and locating demand. A 6-level geohash has six alphanumeric characters and encloses an area of approximately 1.2 km * 0.6 km. For a lower level geohash, namely a 5-level geohash, the area covered is more than a 6-level geohash and therefore, the precision reduces.

4.1 Data Pre-processing and model fitting

To see where the demand is mostly concentrated, all the 1947 unique geohashes in the city are sorted in the decreasing order of passenger demand count and we concentrate on the top 20% of the geohashes in this category.



```
# top 20 percent of the geohashes
geohash_count =
appsessions.groupby('geohash').agg({'geohash':len}).rename(columns={'geohash':'N'})
geohash_count['geohash'] = geohash_count.index
geohash_count = geohash_count.sort_values(['N'], ascending=[False])
geohash_count['geohash'] = geohash_count.index
geohash_count['cumulatives'] = geohash_count.N.cumsum()*100/geohash_count.N.sum()
geohash_count = geohash_count.reset_index(drop=True)
ind = geohash_count[geohash_count['cumulatives']>80]
index = ind.index[0]; best_geohashes = geohash_count['geohash'][0:index+1]
best_geohashes_df = pd.DataFrame(best_geohashes)
filtered_appsession_data =
appsessions[appsessions.geohash.isin(best_geohashes_df.geohash.values)]
```

After running above script.

Table 4.2: Total unique geohashes (1947, 2)

Geohash	**Demand count**
tdr1w4	229853
tdr1v4	113250

tdr1w7	101031
tdr1y1	94900
tdr1w0	92139

Table 4.3: Top 20% of geohashes (433)

Geohash
tdr1w4
tdr1v4
tdr1w7
tdr1y1
tdr1w0

A macroscopic view of the demand patterns in the city can be obtained by analysing the top geohashes. The same techniques can be extended to other geohashes. Hourly demand is defined as the count of passengers opting for a taxi in an hour at each geohash. Hourly demand for geohasg-1 i.e. tdr1w4.

```
geohash_1 = best_geohashes_df['geohash'][0]
geohash_1_data = geohash_time_data[geohash_time_data['geohash'] ==
geohash_1].drop('geohash',axis=1)
geohash_1_data = geohash_1_data.reset_index(drop=True)
```

Table 4.4: Hourly demand for Geohash-1 (tdr1w4)

Index (Time in hours)	**Geohash**	**Demand**
0	tdr1w4	74
1	tdr1w4	89
2	tdr1w4	127
3	tdr1w4	162
4	tdr1w4	183
5	tdr1w4	221

Line plot of Actual data for geohash-1 (tdr1w4)

In this plot, time is shown on the x-axis with observation values along the y-axis.

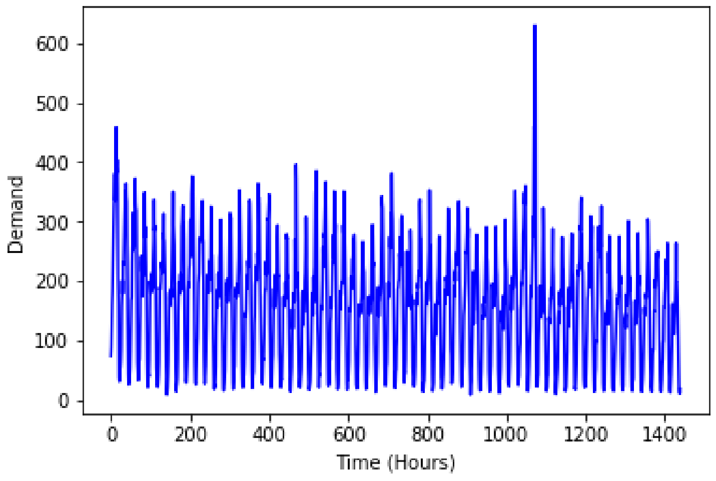


Figure 4.1: Line plot of Actual data for geohash-1 (tdr1w4)

Density (distribution) plot

Provides a useful first check of the distribution of observations both on raw observations and after any type of data transform has been performed.

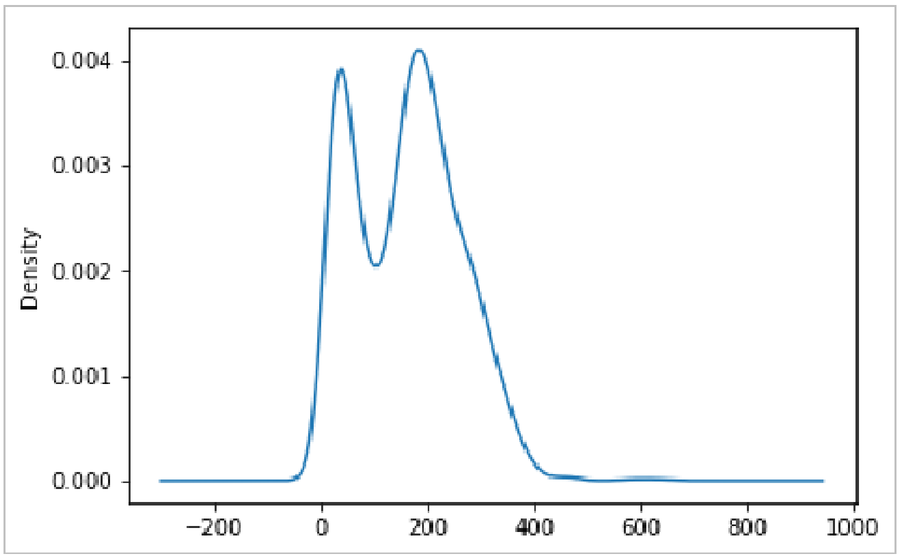


Figure 4.2: Density plot of geohash-1 (tdr1w4)

Histogram (distribution) plot

Provides a useful first check of the distribution of observations both on raw observations and after any type of data transform has been performed.

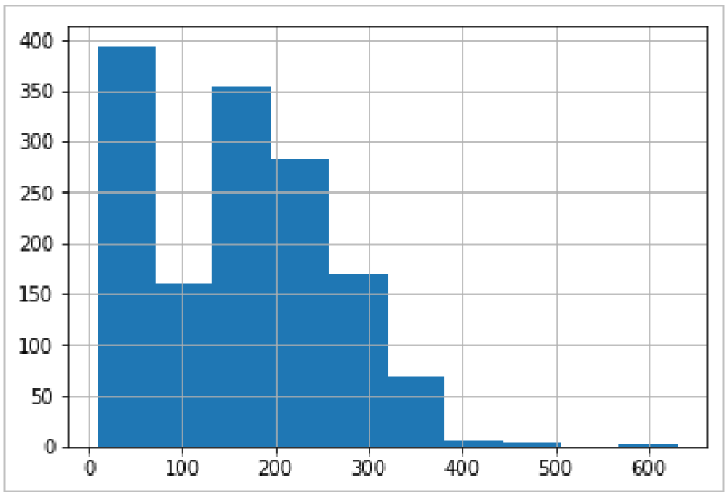


Figure 4.3: Histogram plot of geohash-1 (tdr1w4)

Box and Whisker Plots

This plot draws a box around the 25th and 75th percentiles of the data that captures the middle 50 percent of observations. A line is drawn at the 50th percentile (the median) and whiskers are drawn above and below the box to summarize the general extents of the observations. Dots are drawn for outliers outside the whiskers or extents of the data.

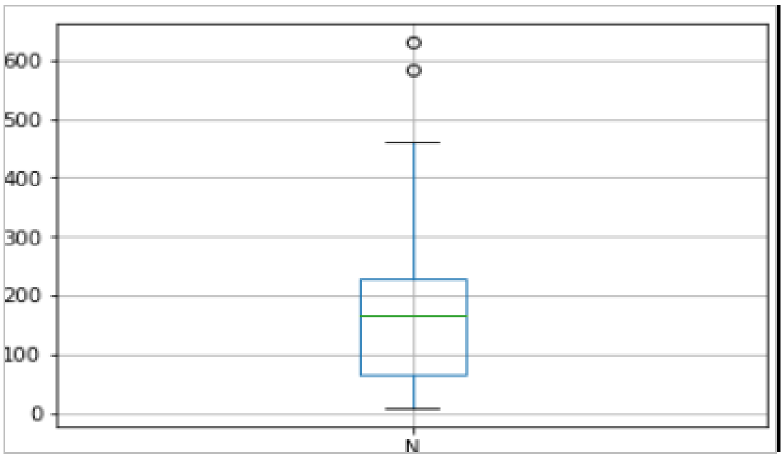


Figure 4.4: Box and Whisker Plots

Lag Scatter Plot

A useful type of plot to explore the relationship between each observation and a lag of that observation is called the scatter plot. If the points cluster along a diagonal line from the bottom-left to the top-right of the plot, it suggests a positive correlation relationship. If the points cluster along a diagonal line from the top-left to the bottom-right, it suggests a negative correlation relationship. Either relationship is good as they can be modeled.

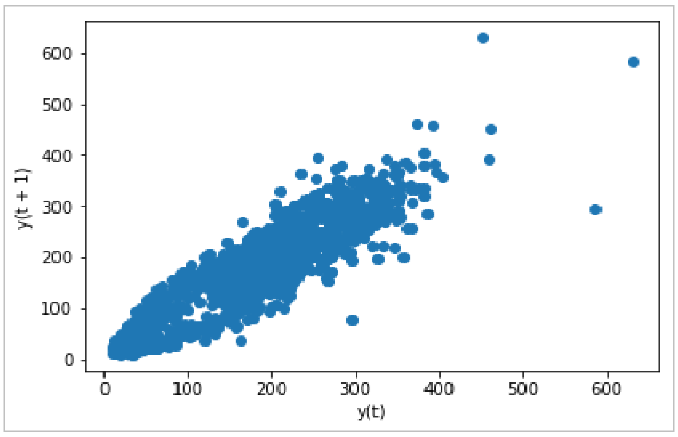


Figure 4.5: Lag Scatter Plot

Auto correlation plot

We can quantify the strength and type of relationship between observations and their lags. In statistics, this is called correlation. A value close to zero suggests a weak correlation, whereas a value closer to -1 or 1 indicates a strong correlation.

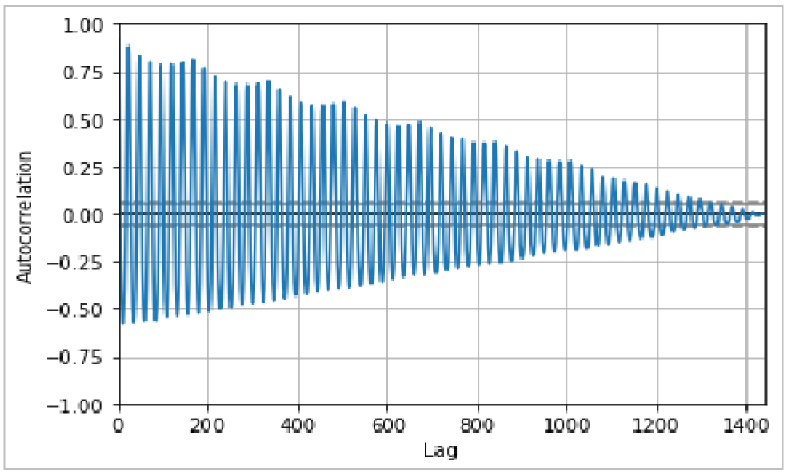


Figure 4.6: Auto correlation plot

Descriptive Statistics

Table 4.5: Descriptive Statistics of Geohash-1 (tdr1w4)

Characteristic	**Value**
Count	1441
Mean	159.509368
Std (Standard deviation)	98.460309
Min	4
25%	65
55%	164
75%	228
Max	717

4.2 Forecasting with Time series models

4.2.1 Forecasting with Autoregression (AR) Model

We can use this AR model by first creating the model AR() and then calling fit() to train it on our dataset. This returns an AR Result object. Once fit, we can use the model to make a prediction by calling the predict() function for a number of observations in the future. We can see that a 23-lag model was chosen and trained.

The AR model with lag 23 gives root mean squared error (RMSE) for test data of 24.675 and R^2 of 0.877. The expected values for the next 24 hours are plotted compared to the predictions from the model.

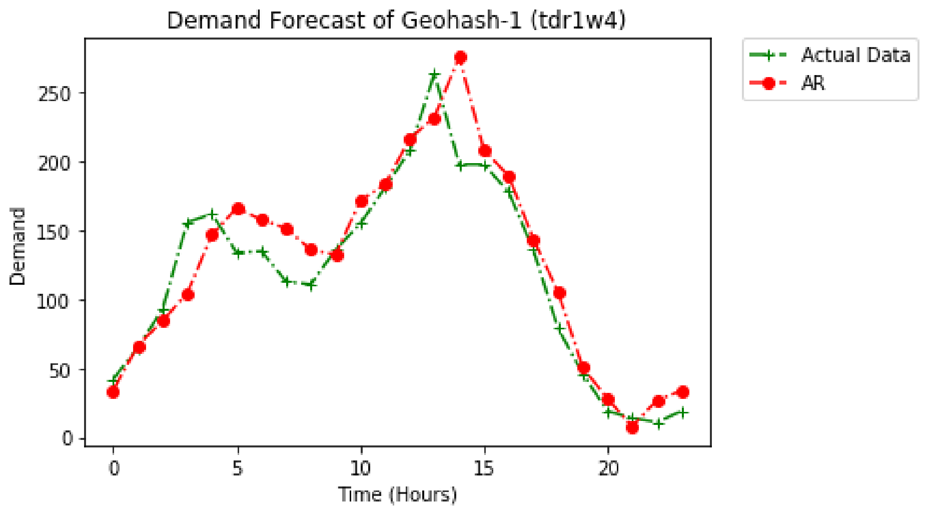


Figure 4.7: Line plot of the forecast with AR on actual test dataset.

4.2.2 Forecasting with Moving Average (MA) Model

We can develop a test harness for the problem by splitting the observations into training and test sets, with only the last 24 observations in the dataset assigned to the test set as unseen data that we wish to predict.

The residual errors are calculated as the difference between the expected outcome (TestY) and the prediction (Predictions). Once we have a residual error time series, we can model the residual error time series using an autoregression (AR) model. Next, we can predict the residual error using the autoregression model. The actual residual error for the time series is plotted compared to the predicted residual error.

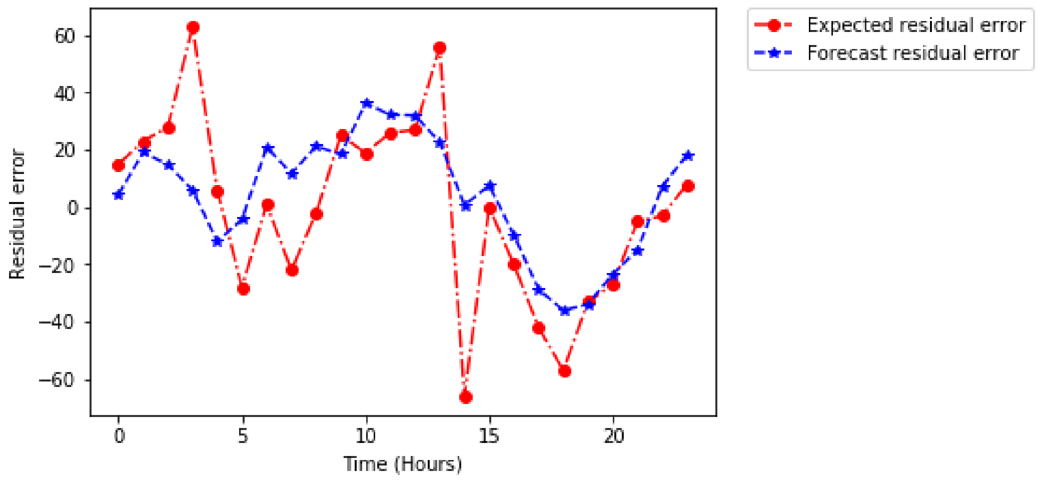


Figure 4.8: Line plot of expected residual error and forecast residual error on actual test dataset

Now correct predictions with a model of residuals, the RMSE of the corrected forecasts of Test data is calculated to be 22.804 and R^2 of 0.895, which are much better than persistence model and AR model. Finally, the expected values for the test dataset are plotted compared to the corrected forecast.

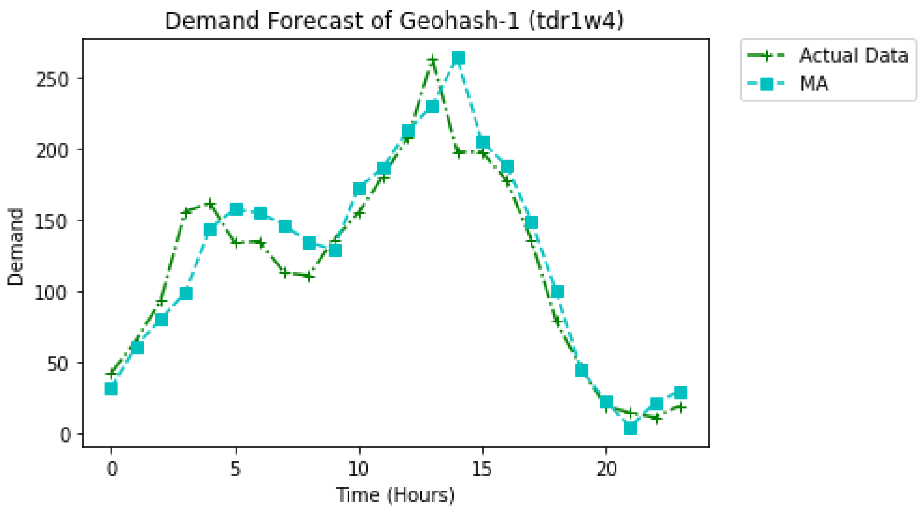


Figure 4.9: Line plot of expected values and corrected forecasts on actual test dataset

4.2.3 Forecasting with ARIMA Model

An ARIMA model can be created using the Statsmodels library as follows:

- Define the model by calling ARIMA() and passing in the p, d, and q parameters.

- The model is prepared on the training data by calling the fit() function.
- Predictions can be made by calling the predict() function and specifying the index of the time or times to be predicted.

First, we fit an ARIMA(2,1,2) model. This sets the lag value to 2 for autoregression, uses a difference order of 1 to make the time series stationary, and uses a moving average model of 2. Which gives root mean squared error (RMSE) for test data of 24.871 and R^2 of 0.875. A line plot is created showing the expected values compared to the predictions.

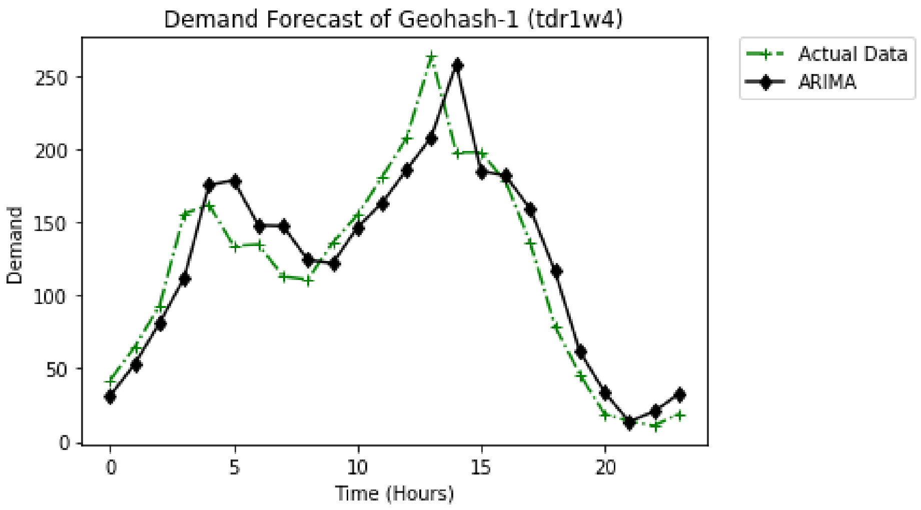


Figure 4.10: Line plot of expected values and forecast with an ARIMA model on actual test data

4.2.4 Comparison of Time series models

Table 4.6: Comparison of Time series models for the top 80 geohashes

Algorithm	**RMSE**	$\mathbf{R^2}$	**MAPE**
Autoregression (AR)	15.443	0.744	34.12
Moving Average (MA)	15.884	0.729	35.57
ARIMA	17.173	0.754	33.61

A line plot is created showing the expected values compared to the predictions with different time series models.

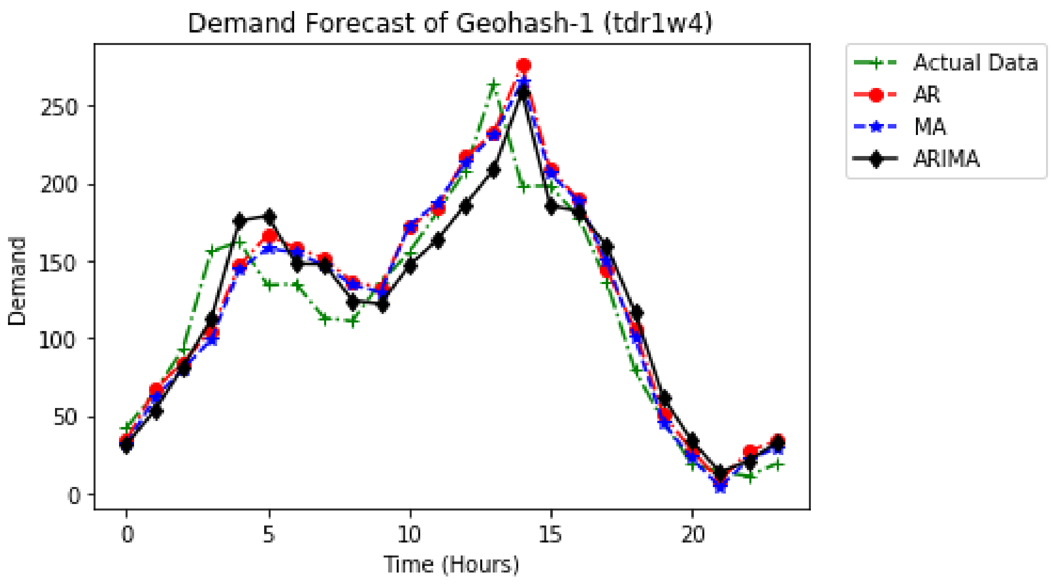


Figure 4.11: Comparison of the actual test data with the forecasts of different models over one day.

4.3 Forecasting with Machine learning

Below is a sample of the first few lines of geohasg-1 (tdr1w4).

Table 4.7: First few observations of the geohasg-1 (tdr1w4)

Index (Time in hours)	**Demand**
0	74
1	89
2	127
3	162
4	183
5	221

Now convert Time series forecasting problem to Supervised learning problem using lag features.

```
#Lag Features (Time series forecasting problem to Supervised learning problem)
temps=DataFrame(series.values)
dataframe = concat([temps.shift(1), temps],axis=1)
dataframe.columns = ['input','target']
dataframe.dropna(inplace=True)
```

Table 4.8: Time series forecasting as Supervised learning using lag features

Index (Time in hours)	**X**	**y**
1	74	89
2	89	127
3	127	162
4	162	183
5	183	221

We can develop a test harness for the problem by splitting the observations into training and test sets, with only the last 24 observations in the dataset assigned to the test set as unseen data that we wish to predict.

Evaluate Algorithms

We have no idea which algorithms will do well on this problem. We will evaluate algorithms using the Mean Squared Error (MSE) metric. MSE will give a gross idea of how wrong all predictions are (0 is perfect).

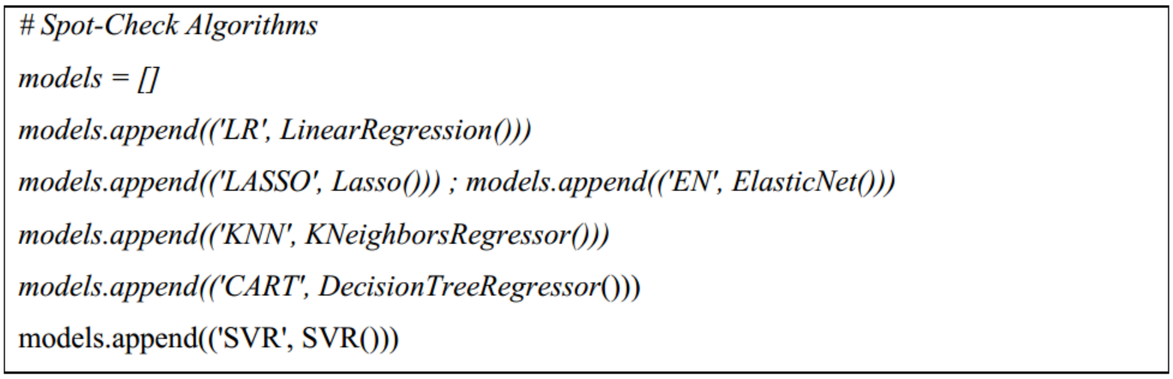
```
# Test options and evaluation metric
num_folds = 10 ;  seed = 7 ;  scoring = 'neg_mean_squared_error'
```

Let's create a baseline of performance on this problem and spot-check a number of different algorithms. We will select a suite of different algorithms capable of working on this regression problem.

The six algorithms selected include:

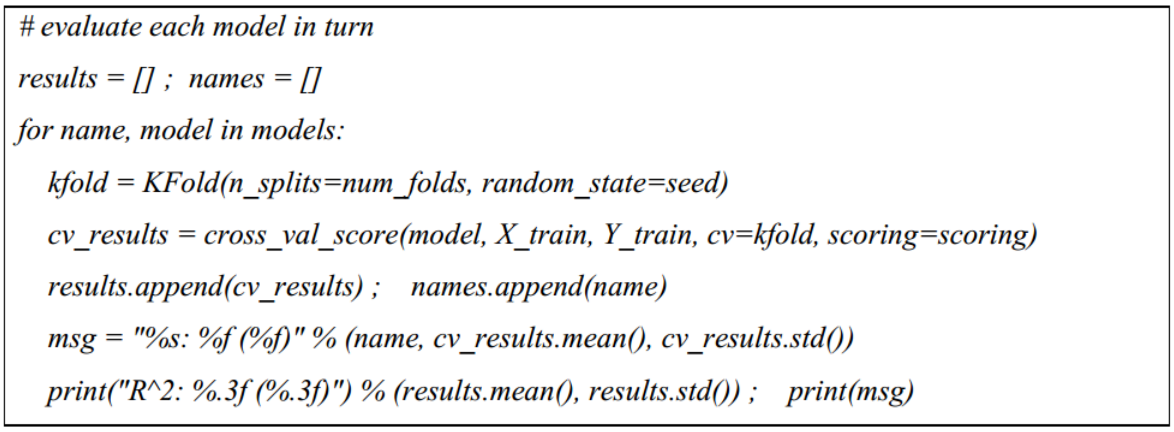
- Linear Algorithms: Linear Regression (LR), Lasso Regression (LASSO) and ElasticNet (EN).

- Non-linear Algorithms: Classification and Regression Trees (CART), Support Vector Regression (SVR) and k-Nearest Neighbors (KNN).



```
# Spot-Check Algorithms
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso())) ; models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

Let's compare the algorithms.



```
# evaluate each model in turn
results = [] ;  names = []
for name, model in models:
   kfold = KFold(n_splits=num_folds, random_state=seed)
   cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
   results.append(cv_results) ;    names.append(name)
   msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
   print("R^2: %.3f (%.3f)") % (results.mean(), results.std()) ;    print(msg)
```

It looks like LR has the lowest RMSE, followed closely by LASSO.

Table 4.9: Comparison of Machine learning models for the top 80 geohashes

Algorithm	**RMSE**	**R^2**	**MAPE**
Linear Regression (LR)	27.508	0.631	43.44
Least Absolute Shrinkage and Selection Operator Regression (LASSO)	31.612	0.623	45.35
ElasticNet (EN)	31.784	0.628	46.71
K-Nearest Neighbors (KNN)	35.416	0.603	48.24

Classification and Regression Trees (CART)	39.513	0.571	49.89
Support Vector Regression (SVR)	40.106	0.657	50.37

Let's take a look at the distribution of scores across all cross-validation folds by algorithm.

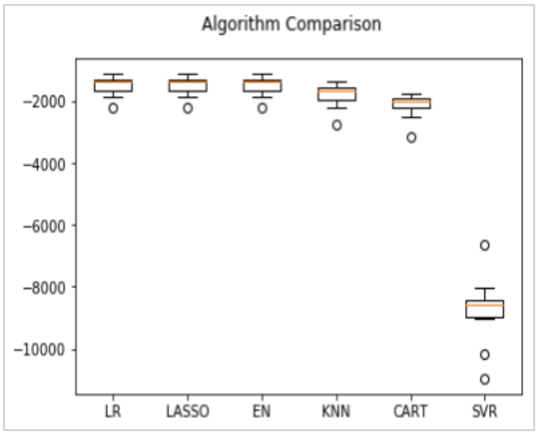


Figure 4.12: Comparison of machine learning algorithms using box plot

We can see similar distributions for the regression algorithms and perhaps a tighter distribution of scores for LR. The expected values for the next 24 hours are plotted compared to the predictions from the model.

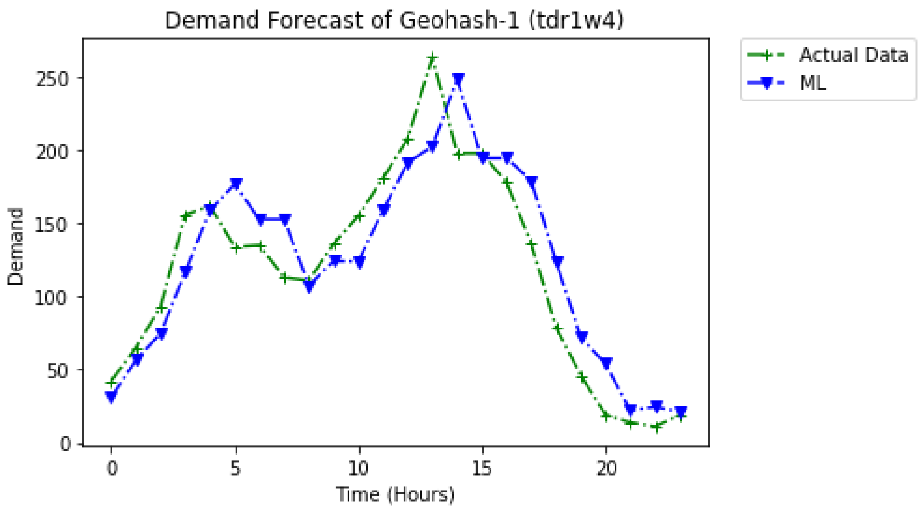


Figure 4.13: Line plot of the ML forecast on actual test dataset.

4.4 Forecasting with Multilayer Perceptron (MLP)

We will phrase the time series prediction problem as a regression problem. That is, given the number of passengers request this hour, what is the number of passengers request next hour. We can write a simple function to convert our single column of data into a two-column dataset. The first column containing this hours (t) passenger count and the second column containing next hours (t+1) passenger count, to be predicted. After we model our data and estimate the skill of our model on the training dataset, we need to get an idea of the skill of the model on new unseen data. For a normal regression problem, we would do this using k-fold cross-validation. With time series data, the sequence of values is important. We can develop a test harness for the problem by splitting the observations into training and test sets, with only the last 24 observations in the dataset assigned to the test set as unseen data that we wish to predict.

Now we can define a function to create a new dataset as described above. The function takes two arguments, the dataset which is a NumPy array that we want to convert into a dataset and the look back which is the number of previous time steps to use as input variables to predict the next time period, in this case, we use 3. For example, given the current time (t) we want to predict the value at the next time in the sequence (t+1), we can use the current time (t) as well as the two prior times (t-1 and t-2). When phrased as a regression problem the input X variables are t-2, t-1, t and the output Y variable is t+1.

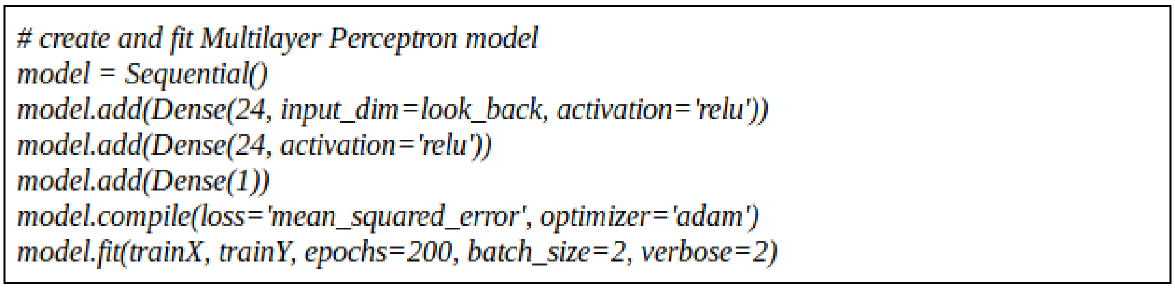
```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back)]
    dataX.append(a)
    dataY.append(dataset[i + look_back])
  return numpy.array(dataX), numpy.array(dataY)
```

Let's take a look at the effect of this function on the first few rows of the dataset.

Table 4.10: Reshaped training dataset

X_1	X_2	X_3	**y**
74	89	127	162
89	127	162	183
127	162	183	221
162	183	221	277
183	221	277	290

We can now fit a Multilayer Perceptron model to the training data. We use a simple network with 1 input layer, 2 hidden layers with 24 neurons each and an output layer. The model is fit using mean squared error, if we take the square root gives us an error score in the units of the dataset.



```
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(24, input_dim=look_back, activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=200, batch_size=2, verbose=2)
```

Once the model is fit, we can estimate the performance of the model on test dataset. MLP gives root mean squared error (RMSE) for test data of 22.72 and R^2 of 0.86 for geohash-1 (tdr1w4). We can see that the model has an average error of 23 passengers count on the test dataset.
A line plot is created showing the expected values compared to the predictions.

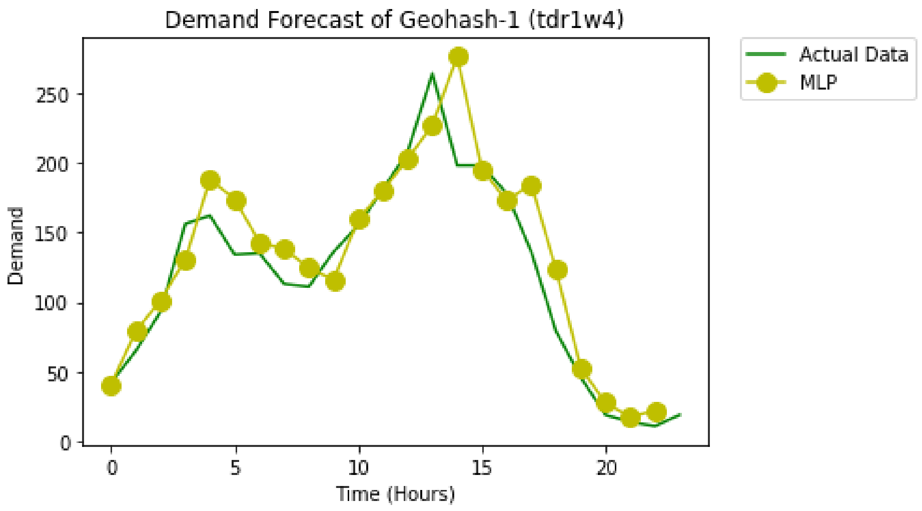


Figure 4.14: Line plot of expected values and forecast with MLP model on actual test data

4.5 Forecasting with LSTM Recurrent Neural Networks

Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables. A powerful type of neural network designed to handle sequence dependence are called recurrent neural networks. The Long Short-Term Memory Networks or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained.

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data, also called standardization. Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1. This can be thought of as subtracting the mean value or centering the data. It assumes that your observations fit a Gaussian distribution (bell curve) with a well behaved mean and standard deviation. We can easily standardize the dataset using the StandardScaler preprocessing class from the scikit-learn library.

```
# Rescale the dataset
scaler = StandardScaler()
scaler = scaler.fit(dataset)
standardized = scaler.transform(dataset)
```

Table 4.11: standardized dataset

Demand
-0.869864
-0.71747
-0.331407
0.0241784
0.237529
0.623593

The LSTM network expects the input data (X) to be provided with a specific array structure in the form of: [samples, time steps, features]. Our prepared data is in the form: [samples, features] and we are framing the problem as one-time step for each sample. We can transform the prepared train and test input data into the expected structure using numpy.reshape() as follows:

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

We are now ready to design and fit our LSTM network for this problem. The network has a visible layer with 1 input, 2 hidden layers with 24 LSTM blocks or neurons each and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM memory blocks. The LSTM network has memory which is capable of remembering across long sequences. Normally, the state within the network is reset after each training batch when fitting the model, as well as each call to model.predict() or model.evaluate(). We can gain finer control over when the internal state of the LSTM network is cleared in Keras by making the LSTM layer stateful. This means that it can build state over the entire training sequence and even maintain that state if needed to make predictions.

```
# create and fit the LSTM network
batch_size = 4
model = Sequential()
model.add(LSTM(24, batch_input_shape=(batch_size, look_back, 1), stateful=True,
return_sequences=True))
model.add(LSTM(24, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=200, batch_size=batch_size, verbose=2, shuffle=False)
model.reset_states()
```

Once the model is fit, we can estimate the performance of the model on test data. LSTM gives root mean squared error (RMSE) for test data of 19.95 and R^2 of 0.91. We can see that the model has an average error of 20 demand count on the test data.
A line plot is created showing the expected values compared to the predictions.

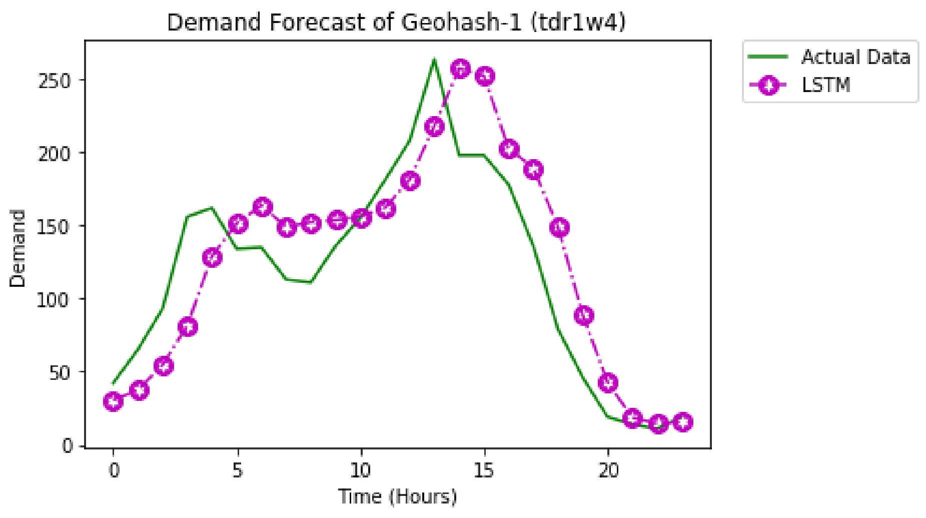


Figure 4.15: Line plot of expected and forecast values with LSTM model on actual test data

4.6 Comparison of all the selected models

Table 4.12 shows the performance of Long Short Term Memory (LSTM) model as compared to all other competing models. LSTM achieves the lowest MAPE (31.47 %) and the lowest RMSE (15.303) among all the methods. More specifically, we can see that Machine learning Models (LR, LASSO, EN, KNN, CART and SVR) perform poorly (i.e., have a MAPE of 43.44, 45.35, 46.71, 48.24, 49.89 and 50.37 respectively). Time series models further consider historical demand values for prediction and therefore achieve better performance than machine learning models.

Table 4.12: Comparison of different models for the top 80 geohashes

Algorithm	**RMSE**	**R^2**	**MAPE**
Autoregression (AR)	15.443	0.744	34.12
Moving Average (MA)	15.884	0.729	35.57
ARIMA	17.173	0.754	33.61
Linear Regression (LR)	27.508	0.631	43.44
Least Absolute Shrinkage and Selection Operator Regression (LASSO)	31.612	0.623	45.35
ElasticNet (EN)	31.784	0.628	46.71
K-Nearest Neighbors (KNN)	35.416	0.603	48.24
Classification and Regression Trees (CART)	39.513	0.571	49.89
Support Vector Regression (SVR)	40.106	0.657	50.37
Multilayer Perceptron (MLP)	17.712	0.748	38.76
Long Short Term Memory (LSTM)	15.303	0.770	31.47

A line plot is created showing the expected values compared to the predictions with different models.

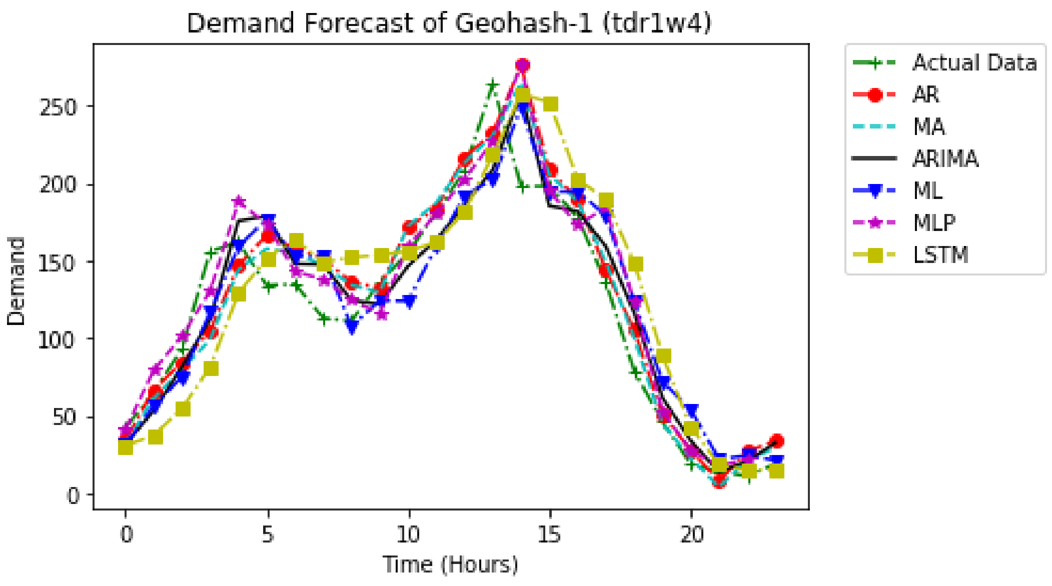


Figure 4.16: Comparison of the actual test data with the forecasts of selected models over one day.

CHAPTER 5

DEMAND PREDICTION APPLICATIONS

The taxi driver sends out information regarding his location every few seconds using Global Positioning System (GPS). This data can be filtered to find the supply, i.e., availability of taxis ready to take in customers, at different locations. If we project supply as a supervised per hour per geohash, supervised learning modeling can be applied there. Once we have both demand and supply data, we can aim to address the demand-supply imbalance problem. Another application of demand analysis is to detect passenger hotspots in the city. Demand prediction is an integral part of idle taxi reallocation problem. If we know that the demand will be high in a region and the supply will be low in that region, then we can readily route more drivers to this high demand region. For example, for a 6 level geohash, we can accurately predict the demand and supply for the next time interval in that area. If the demand for taxi is predicted to increase and the supply is predicted to be low there, we find idle taxis (this information is obtained from the taxi pings given out during its course) within a few kilometer radius of this location, that can reach the location in a fixed time and route those taxis there.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Travel demand modeling is an intrinsic part of transportation planning. Analysing demand can help us in finding passenger hotspots, balancing demand-supply problem and re-allocating taxis to aid drivers find customers. Deep learning models such as the Multilayer Perceptron (MLP) and Long Short Term Memory (LSTM) are evidently effective learners on training data with the LSTM more capable for recognising longer-term dependencies. The LSTM out-performed the Time series and machine learning models marginally, but there was not significant difference in the results of both. However, the LSTM takes considerably longer to train. In this work, the passenger demand for taxis was modeled and we were able to achieve Root Mean Squared Error (RMSE) of 15.303 and MAPE of 31% i.e., an accuracy of 69% at 1 km^2 area level, for the most common demand range using LSTM Recurrent Neural Network (Deep Learning).

A natural extension of our work would be to improve the forecast model by exploring the correlation between adjacent geohashes. Convolutional neural network (CNN) and Clustering should be explored in future studies to model the complex spatial correlation i.e., captures local characteristics of regions in relation to their neighbors.

APPENDIX A

SUPERVISED LEARNING MODELS

A.1 Machine Learning Algorithms

Linear algorithms:

A.1.1 Linear Regression (Ordinary Least Squares)

Linear regression is a prediction method that is more than 200 years old. Simple linear regression is a great first machine learning algorithm to implement as it requires you to estimate properties from your training dataset.

Linear regression assumes a linear or straight line relationship between the input variables (X) and the single output variable (y). More specifically, that output (y) can be calculated from a linear combination of the input variables (X). When there is a single input variable, the method is referred to as a simple linear regression. In simple linear regression we can use statistics on the training data to estimate the coefficients required by the model to make predictions on new data. The line for a simple linear regression model can be written as:

$$y = b_0 + b_1 * x \tag{A.1}$$

Where, b_0 and b_1 are the coefficients we must estimate from the training data.

$$b_1 = \frac{\sum_{i=0}^{n}((x_i - mean(x)) * (y_i - mean(y)))}{\sum_{i=0}^{n}(x_i - mean(x))^2} = \frac{covariance(x, y)}{variance(x)} \tag{A.2}$$

$$b_0 = mean(y) - b_1 * mean(x) \tag{A.3}$$

Where, the i refers to the value of the i th value of the input x or output y.

Linear Regression fits a linear model with coefficients $b = (b_1, ..., b_p)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\underset{b}{min} \, ||Xb - y||_2{}^2 \tag{A.4}$$

However, coefficient estimates for Ordinary Least Squares rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the design matrix becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed response, producing a large variance. This situation of multicollinearity can arise.

Complexity : This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

A.1.2 Ridge Regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares (also called the $\ell_2 - norm$).

$$\underset{b}{min} \, ||Xb - y||_2{}^2 + \alpha ||b||_2{}^2 \tag{A.5}$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

Complexity: This method has the same order of complexity as linear regression

A.1.3 LASSO Regression

The Least Absolute Shrinkage and Selection Operator (or LASSO for short) is a modification of linear regression, like ridge regression, where the loss function is modified to minimize the complexity of the model measured as the sum absolute value of the coefficient values (also called the $\ell_1 - norm$).

It is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights.

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\underset{w}{min} \frac{1}{2n_{samples}} ||Xw - y||_2^2 + \alpha ||w||_1 \tag{A.6}$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha||w||_1$ added, where α is a constant and $||w||_1$ is the $\ell_1 - norm$ of the parameter vector.

A.1.4 ElasticNet Regression

ElasticNet is a linear regression model trained with L1 and L2 prior as regularizer i.e., combines the properties of both Ridge Regression and LASSO regression. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. We control the convex combination of L1 and L2 using the l1-ratio parameter (ρ).

The objective function to minimize is in this case

$$\underset{w}{min} \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2 \tag{A.7}$$

Nonlinear algorithms :

A.1.5 K-Nearest Neighbors (KNN)

A simple but powerful approach for making predictions is to use the most similar historical examples to the new data. This is the principle behind the k-Nearest Neighbors algorithm.

When a prediction is required, the k-most similar records to a new record from the training dataset are then located. From these neighbors, a summarized prediction is made. Similarity between records can be measured many different ways. A problem or data-specific method can be used. Generally, with tabular data, a good starting point is the Euclidean distance.

Once the neighbors are discovered, the summary prediction can be made by returning the most common outcome or taking the average. As such, KNN can be used for classification or regression problems. There is no model to speak of other than holding the entire training dataset. Because no work is done until a prediction is required, KNN is often referred to as a *lazy learning* method.

Following steps will give you the foundation that you need to implement the k-Nearest Neighbors algorithm:

- **Euclidean Distance :** The first step needed is to calculate the distance between two rows in a dataset. Rows of data are mostly made up of numbers and an easy way to calculate the distance between two rows or vectors of numbers is to draw a straight line. This makes sense in 2D or 3D and scales nicely to higher dimensions. We can calculate the straight line distance between two vectors using the Euclidean distance measure. It is calculated as the square root of the sum of the squared differences between the two vectors.

$$distance = \sqrt{\sum_{i=0}^{n}(x_{1i} - x_{2i})^2} \tag{A.8}$$

Where x1 is the first row of data, x2 is the second row of data and i is the index to a specific column as we sum across all columns. With Euclidean distance, the smaller the value, the more similar two records will be. A value of 0 means that there is no difference between two records.

Now it is time to use the distance calculation to locate neighbors within a dataset.

- **Get Neighbors :** Neighbors for a new piece of data in the dataset are the k closest instances, as defined by our distance measure. To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using Euclidean distance. Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbors.

  Now that we know how to get neighbors from the dataset, we can use them to make predictions.

A.1.6 Classification and Regression Trees (CART or decision trees)

Decision trees are a powerful prediction method and extremely popular. They are popular because the final model is so easy to understand by practitioners and domain experts alike. Classification and Regression Trees or CART for short is an acronym introduced by Leo Breiman to refer to Decision Tree algorithms that can be used for classification or regression predictive modeling problems.

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualised.
- Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable.

- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

CART construct a binary tree from the training data. This is the same binary tree from algorithms and data structures, nothing too fancy (each node can have zero, one or two child nodes). A node represents a single input variable (X) and a split point on that variable, assuming the variable is numeric. The leaf nodes (also called terminal nodes) of the tree contain an output variable (y) which is used to make a prediction. Once created, a tree can be navigated with a new row of data following each branch with the splits until a final prediction is made.

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is used called recursive binary splitting. This is a numerical procedure where all the values are lined up and different split points are tried and tested using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner based on the cost function. The cost function that is minimized to choose split points is the sum squared error across all training samples that fall within the rectangle. Splitting continues until nodes contain a minimum number of training examples or a maximum tree depth is reached.

A.1.7 Support Vector Regression (SVR)

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression (SVR). The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

A.2 Deep Learning Algorithms

A.2.1 Multilayer Perceptron (MLP)

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, ..., x_m$ and a target y, it can learn a **non-linear** function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers.

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using partial-fit.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

MLPs approximate a mapping function from input variables to output variables. This general capability is valuable for sequence prediction problems (notably time series forecasting) for a number of reasons.

- Robust to Noise: Neural networks are robust to noise in input data and in the mapping function and can even support learning and prediction in the presence of missing values.
- Non-linear: Neural networks do not make strong assumptions about the mapping function and readily learn linear and non-linear relationships.

More specifically, MLPs can be configured to support an arbitrary defined but fixed number of inputs and outputs in the mapping function. This means that:

- Multivariate Inputs. An arbitrary number of input features can be specified, providing direct support for multivariate prediction.
- Multi-Step Outputs. An arbitrary number of output values can be specified, providing direct support for multi-step and even multivariate prediction.

This capability overcomes the limitations of using classical linear methods (think tools like **ARIMA** for time series forecasting). For these capabilities alone, feedforward neural networks are used for time series forecasting.

Complexity: Suppose there are n training samples, m features, k hidden layers, each containing h neurons - for simplicity, and o output neurons. The time complexity of backpropagation is $O(n \cdot m \cdot h^k \cdot o \cdot i)$, where i is the number of iterations. Since backpropagation has a high time complexity, it is advisable to start with smaller number of hidden neurons and few hidden layers for training.

A.2.2 Long Short-Term Memory (LSTM)

Promise of Recurrent Neural Networks

The Long Short-Term Memory, or LSTM, network is a type of Recurrent Neural Network. Recurrent Neural Networks, or RNNs for short, are a special type of neural network designed for sequence problems. Given a standard feedforward MLP network, an RNN can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal latterly (sideways) in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on. The recurrent connections add state or memory to the network and allow it to learn and harness the ordered nature of observations within input sequences.

Recurrent neural networks contain cycles that feed the network activations from a previous time step as inputs to the network to influence predictions at the current time step. These activations are stored in the internal states of the network which can in principle hold long-term temporal contextual information. This mechanism allows RNNs to exploit a dynamically changing contextual window over the input sequence history.

The promise of recurrent neural networks is that the *temporal dependence and contextual information* in the input data can be learned.

The key technical historical challenge faced with RNNs is how to train them effectively. Experiments show how difficult this was where the weight update procedure resulted in weight changes that quickly became so small as to have no effect (vanishing gradients) or so large as to result in very large changes or even overflow (exploding gradients). LSTMs overcome this challenge by design. An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units - the input, output and forget gates - that provide continuous analogues of write, read and reset operations for the cells.

REFERENCES

[1] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[2] Dirk Brockmann, Lars Hufnagel, and Theo Geisel. “The scaling laws of human travel”. In: *Nature* 439.7075 (2006), p. 462.

[3] Neema Davis, Gaurav Raina, and Krishna Jagannathan. “A multi-level clustering approach for forecasting taxi travel demand”. In: *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on.* IEEE. 2016, pp. 223–228.

[4] Felix A Gers, Douglas Eck, and Jürgen Schmidhuber. “Applying LSTM to time series predictable through time-window approaches”. In: *Neural Nets WIRN Vietri-01.* Springer, 2002, pp. 193–200.

[5] Sepp Hochreiter and Jürgen Schmidhuber. “LSTM can solve hard long time lag problems”. In: *Advances in neural information processing systems.* 1997, pp. 473–479.

[6] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice.* OTexts, 2014.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[8] Xiaolong Li et al. “Prediction of urban human mobility using large-scale taxi traces and its applications”. In: *Frontiers of Computer Science* 6.1 (2012), pp. 111–121.

[9] Xiao Liang et al. “The scaling of human mobility by taxis is exponential”. In: *Physica A: Statistical Mechanics and its Applications* 391.5 (2012), pp. 2135–2144.

[10] Luis Moreira-Matias et al. “Predicting taxi–passenger demand using streaming data”. In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (2013), pp. 1393–1402.

[11] Shashank Shekhar and Billy Williams. “Adaptive seasonal time series models for forecasting short-term traffic flow”. In: *Transportation Research Record: Journal of the Transportation Research Board* 2024 (2008), pp. 116–125.

[12] Zaiyong Tang, Chrys de Almeida, and Paul A Fishwick. “Time series forecasting using neural networks vs. Box-Jenkins methodology”. In: *Simulation* 57.5 (1991), pp. 303–310.

[13] Ian H Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.