COVID-19 and Machine Learning: Investigation and Prediction

Team #5:
Kiran Brar
Olivia Alexander
Kimberly Segura

Abstract

The COVID-19 virus has affected over four million people in the world. In the U.S. alone, the number of positive cases have exceeded one million, making it the most affected country. There is clear urgency to predict and ultimately decrease the spread of this infectious disease. Therefore, this project was motivated to test and determine various machine learning models that can accurately predict the number of confirmed COVID-19 cases in the U.S. using available time-series data. COVID-19 data was coupled with state demographic data to investigate the distribution of cases and potential correlations between demographic features. Concerning the four machine learning models tested, it was hypothesized that LSTM and XGBoost would result in the lowest errors due to the complexity and power of these models, followed by SVR and linear regression. However, linear regression and SVR had the best performance in this study which demonstrates the importance of testing simpler models and only adding complexity if the data requires it. We note that LSTM's low performance was most likely due to the size of the training dataset available at the time of this research as deep learning requires a vast amount of data. Additionally, each model's accuracy improved after implementing time-series preprocessing techniques of power transformations, normalization, and the overall restructuring of the time-series problem to a supervised machine learning problem using lagged values. This research can be furthered by predicting the number of deaths and recoveries as well as extending the models by integrating healthcare capacity and social restrictions in order to increase accuracy or to forecast infection, death, and recovery rates for future dates.

Keywords: COVID-19, time-series forecasting, linear regression, SVR, XGBoost, LSTM, model evaluation

Acknowledgements

We would like to thank Professor Wang for his guidance during this project.

We would also like to thank Kaggle.com for supplying open source data for COVID-19.

Table of Contents

Page

I. INTRODUCTION

Research Objective and Problem

Objective

The objective of this paper is to conduct a comparative study on the performance of different machine learning models in forecasting the number of COVID-19 confirmed cases in the United States.

What is the Problem

The world is at war with a disease that we can't seem to figure out how to stop from spreading. The COVID-19 disease is set apart by its ability to spread easily. Exponentially growing every single day. The biggest challenge we face is a shortage of resources, such as equipment and health care workers, to aid those who are affected with the disease. As stated in the paper "Malaria Epidemics Detection and Control Forecasting and Prevention", "The actual impact of epidemics depends not only on the increase in specific morbidity, but also on the general health of the affected population." Our unreadiness for a disaster of this magnitude has led the government to declare a national emergency with some states order an executive stay at home order, in hopes that we "flatten the curve." The goal is to diminish the amount of people who have contracted the disease so we can properly care for them. The result of social distancing has not only impacted the economy but also the mental health for many americans. As health workers fight tirelessly to save the lives of COVID-19 patients, we believe it will be extremely valuable to accurately forecast the number of confirmed cases that are expected based on the current data.

COVID-19 is an infectious disease that is caused by severe acute respiratory syndrome. First identified in December 2019 in Wuhan, China, the disease has spread to over 210 countries with 1.8 million confirmed cases. COVID-19 is incredibly insidious because of its asymptomatic transmission -from the time of exposure it may take 5 to 14 days to start showing symptoms, which include fever, dry cough, and shortness of breath.

In hopes of diminishing the spread, the United States government has placed a country wide quarantine, where individuals are encouraged to maintain a 6 foot distance and only go out for the essentials. People must also wear a mask to avoid spreading germ droplets. COVID-19 presented unprecedented challenges as many businesses have experienced a huge economic downfall, and tons of citizens are no longer working and are worried about where their next paycheck is going to come from. The impact of COVID-19 is something that we have never seen in our lifetimes, there has never been a situation of this capacity in many years, and we have proved to be unprepared. Relief measures such as a stimulus check have been in the works, as well as urging citizens to wear homemade cotton facemasks.

A few major determinants that are important to note are that the virus is spreading rapidly, there is no cure or medicine to help combat the disease, and everyone is at risk. Due to the way the virus has been spreading so rapidly, thousands of people are now infected and we do not have the resources at hospitals to care for everyone. The possibility of having a cure or vaccine is said to be expected a year from now. People over the age of 65 who have previously had chronic diseases are the most at risk. The disease causes the lungs to have complications, and can cause pneumonia.

Research Comparisons

Why This is a Project Related to Class

In machine learning, it is best to attempt several models, based on the purpose and domain of the data, in order to find the best fit model based on accuracy, performance, cost, etc. Therefore, we will be applying our learnings from this course by utilizing a linear regression model to the data, which was a simple model covered in class. We will also extend our learnings by using machine learning algorithms that are unfamiliar to us, such as SVR, XGBoost, and LSTMs. By testing different models for time-series forecasting of COVID-19 cases, we can compare the strengths and weaknesses of different levels of complexities for models. For example, linear regression is an extremely simple model and will most likely not fit the data well. However, accuracy can be relative to the domain as well as the complexity or relevance of the data used in modeling. Therefore, when comparing the performance of linear regression to more complex models such as SVR, XGBoost, and LSTMs, we can weigh the value of fitting data to complex models versus simple models.

As we explored and researched different machine learning models, we noticed that using these models is common for forecasting and modeling of epidemics. We found a study that evaluated Neural Networks, Support Vector Machines, Random Forests and XGBoost to determine the best technique to show that human mobility has an impact on the spread of dengue. The purpose of this paper was to provide solutions for allocating resources in order to combat the disease.

Why Other Approach is No Good

COVID-19 is an ongoing epidemic at the time of this research. Whereas past viruses can be studied and modeled with the use of mathematical or statistical models once more information about the virus has been concluded such as how rapidly the virus tends to transmit, most vulnerable demographic features, mortality rates, etc., these factors are not yet known for COVID-19 and will not be determined in the foreseeable future until years of research of this epidemic has been done. Therefore, in the current study of COVID-19, it is vastly appropriate to attempt to forecast the rate of this epidemic using machine learning.

As this research is presented, the COVID-19 pandemic is new and growing. There are few resources that are actively working on analyzing the deaths, recovered, and infected patients of the COVID-19. The new disease therefore lacks research literature and there are no approaches to be followed. Other papers have been useful to know how other epidemics have been handled, but COVID-19 has unique factors that differentiate from previous diseases.

Why We Think Our Approach is Better

Machine learning requires less information from the user side, where we currently have knowledge gaps, and will rather attempt to learn from the data in order to assist us with these important insights to the spread of the virus. Forecasting COVID-19 cases with dynamic models through machine learning applications rather than static mathematical and statistical models will assist in learning from the constantly evolving and updated data.

The ongoing battle with COVID-19 has left researchers working tirelessly to evaluate how the disease functions and what we could do to diminish the spread. In recent months, there

have been few contributions to the area of data analysis for COVID-19, we believe our approach will offer valuable insight that no recent paper has discussed. Our goal is to find the best machine learning algorithm that will accurately represent the future of COVID-19 and attempt to make some observations that will aid in the fight against this pandemic.

Scope of Investigation

COVID-19 cases in the United States are now the worlds-leading both in terms of confirmed cases as well as deaths related to the virus. As this current epidemic continues to evolve and scientists and health workers continuously give their best efforts to stop the spread of this deadly disease, data scientists are also trying to contribute to this effort by analyzing data and demographics surrounding COVID-19. In this research, we aim to collect time-series data of COVID-19 cases in the United States in attempts to forecast confirmed cases from the virus using applications of machine learning. As this epidemic is new and evolving, there is no past research or studies involving applications of machine learning for COVID-19 predictions.

Therefore, we will research methods of general epidemic modeling to gain insights into this field as well as researching machine learning models and which may be most beneficial in predicting time series data. Additionally, in an attempt to understand the distribution of COVID-19 cases across the United States, we aim to collect demographic data for each state and perform data exploration on various features. Once time-series and demographic COVID-19 data for the United States has been collected, related literature and methods have been reviewed and relevant machine learning models have been selected, we will test the various machine learning models with the time-series data collected and capture each model's accuracy in forecasting

COVID-19 cases. We will then compare the results of each model by capturing different statistical metrics to evaluate model accuracy. In doing so, we hope to suggest the best fit model from our study for the use of future researchers of the COVID-19 epidemic, which may be studied for decades to come.

II. THEORETICAL BASES AND LITERATURE REVIEW

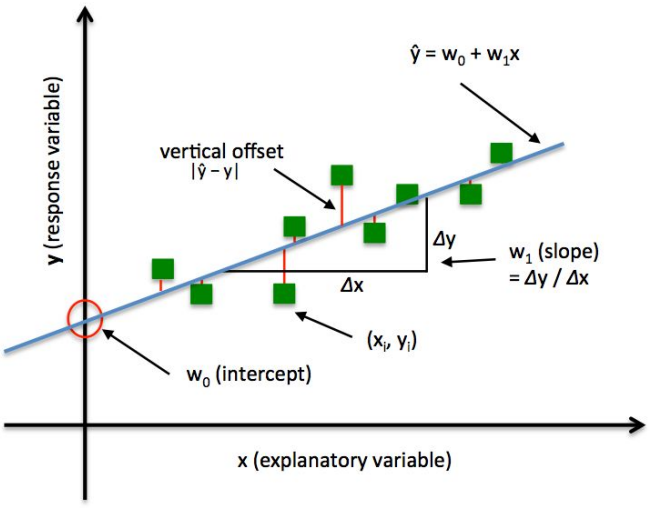
Theoretical Background of the Problem

Linear Regression

Although linear regression can have higher errors than more sophisticated machine learning models, it can nevertheless have strong performance and is one of the most simplest and interpretable models[Ristanoski]. We will consider linear regression in this paper as a method for time series forecasting. Linear regression generates a predictive model by modeling the relationship between the labels (dependent variable) and explanatory variables (independent variables). Put differently, the dependent variable can be modeled as the following :

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{j=0}^{m} \mathbf{w}^\top \mathbf{x}$$

Given a set of training examples, it can find the parameters (best fit) by minimizing the distance(cost function) between the model's prediction and actual labels of training examples.



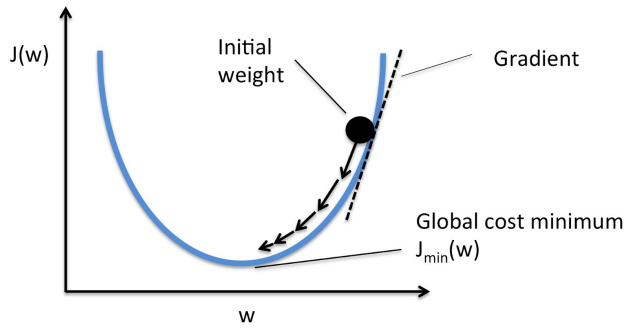
[1]

[1] https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html

For linear regression, there are two main approaches for optimizing the cost function: 1) Solving the system of normal equations(closed form solution) and 2) gradient descent. To find the weights using the closed form solution, weights are found using the following formula:

$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$. However, if the dataset is small (number of features is greater than the number of training examples), this approach fails to compute the weights as the matrix X^TX will not be invertible. Alternatively, optimal weights can be found using gradient descent, which iteratively updates the weights by computing the derivative of the cost function in respect to the parameter using the whole training set. The weight is updated by moving in the opposite direction of the gradient, where J is the cost function and is the learning rate:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \quad \Delta w_j = -\eta \frac{\partial J}{\partial w_j},$$



There are many variants of gradient descent, including Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent (MB-GD). Instead of using the whole training set, SGD and MB-GD use a random training example or small subset of training data to calculate the gradient, respectively.

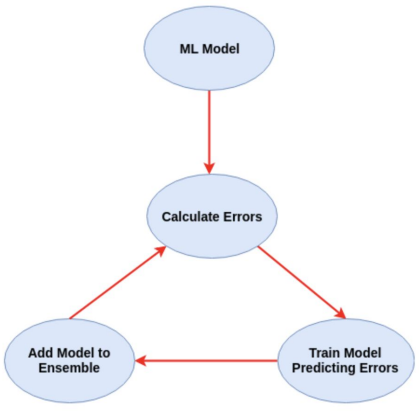
To prevent overfitting and constrain model complexity, *regularization* can be used to penalize high weights. This can be achieved by adding a regularization term to the cost function, where the α controls the extent of regularization.

The cost function using L1 norm (Lasso regularization) : $J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \Sigma_{i=1}^{n} |\theta_i|$

The cost function using L2 norm (Ridge regularization): $J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \Sigma_{i=1}^{n} \theta_i^2$

XGBoost

Extreme Gradient Boosting (XGBoost) implements gradient boosting machines. This method works by training models at the same time, with each model trained to improve the errors of precedent. Models are continued to be added until it has reached its limit on enhancement. What makes XGBoosting fairly convenient is that it is focused on fixing the mistakes of the previous model, rather than in algorithms that work all at once making the same mistakes. Gradient boosting is the name of the approach in which models are trained to predict the errors of the previous models.



The benefits of using XGBoost is that of speed and performance. This method is a very fast implementation of gradient boosting compared to others. The scalability of XGBoost can be attributed to a tree learning algorithm for handling sparse data, parallel and distributed computation, and allowing to process millions of examples in one computer [Chen & Guestrin].

XGBoost solves the following equation: $\frac{\partial L(y, f^{(m-1)}(x) + f_m(x))}{\partial f_m(x)} = 0$ for each x in the data set. Followed by doing second-order Taylor expansion on the loss function, you get g_m(x) the gradient and h_m(x) the Hessian. The loss function can be expressed as:

$$L(f_m) \approx \sum_{i=1}^{n} [g_m(x_i) f_m(x_i) + \frac{1}{2} h_m(x_i) f_m(x_i)^2] + const.$$
$$\propto \sum_{j=1}^{T_m} \sum_{i \in R_{jm}} [g_m(x_i) w_{jm} + \frac{1}{2} h_m(x_i) w_{jm}^2].$$

Letting G_jm represent the sum of gradient in region j and H_jm will equal the sum of hessian, we get a function of $L(f_m) \propto \sum_{j=1}^{T_m} [G_{jm} w_{jm} + \frac{1}{2} H_{jm} w_{jm}^2].$

where the optimal weight is: $w_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, ..., T_m.$

Plugging it back into the loss function yields: $L(f_m) \propto -\frac{1}{2} \sum_{j=1}^{T_m} \frac{G_{jm}^2}{H_{jm}}.$. This is the structure score for the tree and the goal is to make the score smaller. Thus, each split a proxy is gained. XGBoost uses regularization to improve its performance . Thus we can rewrite the function and obtain a the following gain function:

$$Gain = \frac{1}{2}[\frac{T_\alpha(G_{jmL})^2}{H_{jmL}+\lambda} + \frac{T_\alpha(G_{jmR})^2}{H_{jmR}+\lambda} - \frac{T_\alpha(G_{jm})^2}{H_{jm}+\lambda}] - \gamma$$

$$T_\alpha(G) = \begin{cases} G+\alpha & G < -\alpha, \\ G-\alpha & G > \alpha, \\ 0 & else. \end{cases}$$

Support Vector Regression

Support Vector Machines (SVM) have also been explored in time series forecasting [Makridakis, Samsudin]. Ahmed et al. found that SVM to be one of the top performing models in time series forecasting, whereas another comparative study, conducted by Samsudin et al., even suggested that support vector machines outperform neural networks for time series forecasting. The applicability and capability for SVMs to solve the time series forecasting has been attributed to SVM's ability to solve nonlinear regression estimation problems [Samsudin].

Specifically, support-vector machines are supervised learning models that construct a hyperplane such that maximizes the margin between two classes (while minimizing error). When data is not linearly separable, one can map the data into high-dimensional feature space by a nonlinear mapping, and then perform linear regression.

To do this, basis functions can be applied to the data point d and query point q, which can map the point into higher dimensional space. The prediction for a query would look like this:

$$\mathbb{M}_{\boldsymbol{\alpha},\boldsymbol{\phi},w_0}(\mathbf{q}) = \sum_{i=1}^{s} (t_i \times \boldsymbol{\alpha}[i] \times (\boldsymbol{\phi}(\mathbf{d}_i) \cdot \boldsymbol{\phi}(\mathbf{q})) + w_0)$$

However, computing dot products of 2 high-dimensional vectors can become very computationally expensive. To avoid this, nonlinear SVMs use the *kernel trick*: one can replace

the dot product of $\varphi(d) \bullet \varphi(q)$ with the kernel function, which is much less costly. The kernel function is capable of computing the $\varphi(d) \bullet \varphi(q)$ by just using the original vectors d and q, which have less dimensions than $\varphi(d)$ and $\varphi(q)$.

$$\mathbb{M}_{\boldsymbol{\alpha},kernel,w_0}(\mathbf{q}) = \sum_{i=1}^{s} (t_i \times \boldsymbol{\alpha}[i] \times kernel(\mathbf{d}_i, \mathbf{q}) + w_0)$$

There are many kinds of kernel functions, such as radial basis, polynomial, etc. that can be used with support vector machines. One needs to experiment with different kinds and pick the one with the best score.

Long Short Term Memory networks (LSTM)

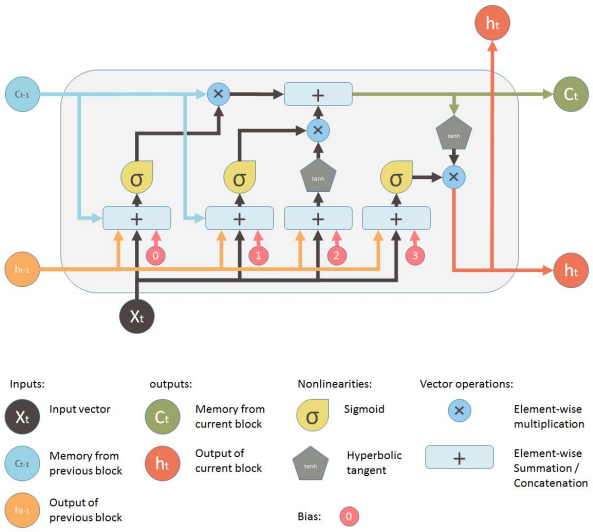
LSTMs have attracted a lot of interest recently in the forecasting field [Chae, Ahmed] and will be examined in this paper. Long Short Term Memory networks are a special kind of artificial neural networks (specifically recurrent neural networks) capable of learning long-term dependencies from long sequences of observations. The key idea behind LSTM is we want some information to persist over time. Because of its complex architecture consisting of several gates, LSTMs have the ability to remember values over time and can effectively regulate the flow of information.

LSTM exploits long term dependencies by the use of *cell states* C_t, sometimes referred to as a conveyor belt, that runs through the different cells, enabling the persistence of information from one memory block to the next. The flow of information is also controlled by *gates,* which determine whether values are added to the cell state or discarded. There are 3 types of gates: input i_t , output o_t, and forget gate f_t. These gates consist of nonlinear sigmoid function, which

given the previous hidden state and input x, it returns a value from 0 (discard value) to 1 (keep value). Given input x_1, .. x_T and y_1, …y_T, the following equations determine unit activations[Chniti]:

$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \\
c_t &= f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o\right) \\
i_t &= o_t \tanh\left(c_t\right)
\end{aligned}
$$

The overall architecture is:



In the article, “E-commerce Time Series Forecasting using LSTM Neural Network and Support Vector Regression”, researchers compared the performance of SVR and LSTM on time

series forecasting using ecommerce data[Chniti], concluding that multivariate LSTM has higher accuracy than multivatiate SVR.

Related Work

Related Research to Solve the Problem

There has been a vast array of research conducted in the domain of epidemic modeling. However, many mathematical and statistical models are unable to capture the complexities of these incidences. A widely used epidemiological model known as the SIR model (Suseptibles, Infected, Recovered) computes the predicted number of infected individuals from an epidemic episode over time within a fixed population, as used in K. Abbas et al., 2005. This study researched the temporal spread of influenza, an infectious disease similar to COVID-19, using a SIR model.

Early detection of epidemics is imperative to the health and safety of the human population. A lot of techniques have been used to identify epidemics, including using social media to help determine when an epidemic is upon us. A paper written by students from the University of Tokyo called, Twitter Catches the Flu: Detecting Influenza Epidemics using Twitter, describes the method of using SVM to extract useful information from tweets of influenza patients. Their hypothesis is that “Twitter texts reflect the real world in real time” and in turn help prevent destructive epidemics from occurring. The paper distinguished between tweets that aren't relevant or provide truthful information, negative tweets, and positive tweets that were in fact written by those suffering from the flu. “First, we build an annotated corpus of pairs of a tweet and positive/negative labels. Then, a support vector machine (SVM) (Cortes and

Vapnik, 1995) based sentence classifier extracts only positive influenza tweets from tweets. This paper helps us determine different outlets we can seek for more information about how many people are actually contracting COVID-19.

Advantage/Disadvantage of those Research

The traditional SIR model used in K. Abbas et al. consists of three differential equations for each dependent variable segment of the population (S,I,R; with independent variable t), that involve two parameters, the rate of transmission and the rate of removal. This may be disadvantageous as these two parameters often need to be estimated. It is difficult to determine, especially in the midst of an actual epidemic when new data is being developed daily, what the value of these two parameters are. Additionally, K. Abbas et al. derived their influenza epidemic data set through synthetic computations using a Bayseian network. However, this raises a problem when producing epidemic curves, which graph the incidence of the disease over time from time series analysis of the data sets on different variants of the demographics to identify risk levels, from the SIR model. This is due to the epidemic curves being based on the probability distribution of features and their estimated conditional probabilities, this can result in very inaccurate predictions.

This model assumes that the total number of people in a certain area is a constant which can help with simplicity and aids information like revealing the overall information transmission law. However, this can also be considered a disadvantage because it limits the application scope of the model as in reality, there are always changes in population and some form of interaction between populations. Also, the model cannot adapt to changes of control policies such as city

lockdowns, which would impact the epidemic curve. Additionally, conventional statistical methods like maximum likelihood estimation require an explicit solution of the time series data whereas this is difficult to obtain from the SIR model due to the nonlinearity of the model. Therefore, several approximations are required to fit the SIR model with the epidemic data of infectious diseases.

Although statistical models have traditionally been used for time series forecasting, machine learning models have become serious contenders [Ahmed]. In a study conducted using Korea Center for Disease Control data, researchers found the machine learning models LSTM and deep neural networks to be more accurate predictors of infectious diseases, such as chickenpox and scarlet fever, than statistical models like ARIMA [Chae].

Additionally, in another study, Ahmed and others conducted large scale comparison study of the following 8 machine learning models for time series forecasting: multilayer perceptron, Bayesian neural networks, radial basis functions, generalized regression neural networks (also called kernel regression), K-nearest neighbor regression, CART regression trees, support vector regression, and Gaussian processes. The study concluded multilayer perceptron and support vector regression to be two of the best performing models, with CART (classification and regression trees) and radial basis functions having the worst overall performance. Keeping this ranking in mind, we will avoid using models, like CART and radial basis functions, which have low accuracy.

Proposed Solution

Our Solution to this Problem, Where it Differs From Others and why it is Better

Viruses are significantly complex to model, even with static mathematical or statistical models, which are generally used for these purposes. Additionally, one of the most widely-used epidemiological models, the SIR model, depends on two critical parameters to produce its results, the rate of transmission of the virus and the rate of removal of the virus. In this research, we are studying an ongoing epidemic where new data, developments, and understandings of the virus are constantly occurring. Therefore, we are not afforded the luxury of having an estimated rate of transmission or removal with a good level of confidence. Even influenza (the flu), which has evidence dating back to 1580, is still being studied and modeled to date. Thus, for the goals of this study to model and forecast the number of confirmed cases related to COVID-19 in the United States, we turn to machine learning. Addressing this epidemic from a machine learning approach rather than a static mathematical or statistical model will be better to allow the model to learn from the data. In static models, usually based on differential equations with manually set parameters, there requires human interaction and knowledge of the epidemic. For COVID-19, it is too early in the epidemics' development to estimate these parameters effectively. Therefore, we will take advantage of the strength of machine learning to help in the understanding and forecasting of COVID-19 cases in the United States.

Applications of machine learning for COVID-19 have not been reported in research papers to date. Therefore, we believe that all models used in this study in attempts to forecast COVID-19 cases in the United States will be original based on our research. However, as a consequence of this work being extremely new, we will be relying on the general performance of different machine learning models to forecast time-series data in order to select the machine learning models best suited for this study and will examine and compare the results of each

respective model. Our hope in modeling this new epidemic and applying new methods to study its behavior, that we can find and recommend an appropriate machine learning model for COVID-19 cases, as it may be studied for decades to follow.

Many researchers have modeled epidemic outbreaks with the use of synthetic data and simulations. We aim to conduct this research by using authentic and current epidemic data in connection with machine learning. A major difference in these two approaches is how the epidemic is modeled: simulations vs. machine learning models. In simulations, one knows and understands the model of the incidence, but does not have the data. In machine learning, one has the data for the incidence, but does not know or understand the exact model of it. Additionally, as stated when using static models such as SIR, there is a requirement to understand several characteristics of the epidemic. Since we are studying a developing virus, it is hypothesized that applications of machine learning will greatly fit the needs of this modeling.

However, these machine learning models have not been deployed to forecast confirmed COVID-19 cases. The purpose of this paper is to 1) solve the problem of forecasting COVID-19 cases and 2) evaluate and compare performance of different machine learning models on forecasting COVID-19. In particular, our team will compare the following models: linear regression, support vector machines, XGBoost and LSTM.

Although there have been significant amounts of studies done on forecasting infectious diseases, there is limited research on how to effectively apply machine learning models for time series forecasting on COVID-19. In our paper, we will try to address this gap in research.

III. HYPOTHESES

Hypotheses for Research

Our team has a two-part hypothesis regarding the 1) prediction of the COVID-19 and 2) performance of machine learning models. In terms of COVID-19 cases, we predict a rise in the number of confirmed cases. There are a number of factors that we have taken into consideration that have led us to this prediction. First and foremost, the rate at which people are getting infected has led in a huge decrease of available resources. Hospitals are struggling to accept all these patients. Additionally, as this virus is completely new, there is not yet a vaccination available and will most likely not be developed in the near future. Secondly, the country wide quarantine has not been taken seriously by many citizens. Thirdly, the uncertainty of this disease and how it has been spreading so quickly, and how it affects the human body. The CDC has said that it could take up to two weeks before showing any symptoms, between that time of being infected and showing symptoms you could have infected a number of people who in turn have affected others; a domino effect.

In terms of our hypothesis for model performance, we predict long short term memory networks (LSTM) to have the best accuracy, followed by XGBoosting, followed by support vector machines (SVM) for performing our time-series forecasting of COVID-19. We hypothesize that the worst accuracy of forecasting prediction will be linear regression as COVID-19 cases seem to be growing exponentially which a linear model may not be able to capture.

IV. METHODOLOGY

How to Generate/Collect Input Data

From the research we have conducted for related work of epidemic modeling, we have found that synthesizing epidemic data can lead to inaccurate results from modeling. Data scientists often use the phrase "Garbage in, garbage out." This expresses that any data with the appropriate format can be inputted to a model. However, if the data itself is not accurate or representative of the subject in study, the results of the model will not be of actual value. Similarly, as complex as viruses and epidemics can be, there is a certain risk with synthesizing data. For example, using a Bayesian network and estimating conditional probabilities between features may not produce accurate results to model the epidemic in study, let alone a new epidemic in consideration. Likewise, synthetic data that may have produced accurate results for another epidemic, like influenza, may not be appropriate to use to model COVID-19, as these two viruses, although they share similar symptoms, do not behave and transmit in an identical manner. Therefore, we will not be synthesizing or producing code to generate data for this study. We feel it is crucial to work with authentic COVID-19 data as we hope our results can have an impact on this critical and evolving epidemic.

We will be utilizing an open-source dataset from Kaggle to collect the time-series data of COVID-19 in the United States. The data set records (rows) indicate cities that have conducted COVID-19 testing within the United States, as well as United States territories, for a total of 3,253 records. The state belonging to each record is also specified, along with its latitude and longitude, total city population, and number of occurrences by day (the time-series data). The time-series data for each city begins on January 22, 2020, as the first case of COVID-19 in the

United States was confirmed on January 20, 2020, less than a month after its discovery in Wuhan, China. As this is an ongoing epidemic, the data set is being updated twice daily. As we are researching an ongoing epidemic with new data being produced daily, we will continue to update our training dataset in the hopes of potentially improving model accuracy as more data becomes available. For the purposes of this research, we will be studying, monitoring, and modeling the daily number of confirmed cases of COVID-19 in the United States through time-series modeling. For reference, this open-source Kaggle dataset is composed from the data repository for the 2019 Novel Coronavirus Visual Dashboard operated by the Johns Hopkins University Center for Systems Science and Engineering (JHU CSSE) which is also supported by ESRI Living Atlas Team and the Johns Hopkins University Applied Physics Lab (JHU APL). This repository is updated once a day around 23:59 (UTC) for files after February 1st, 2020.

For the purposes of understanding and exploring the COVID-19 cases in the United States including the distribution among states, we aim to explore population features such as density, age, gender, health factors, income, etc. to test for possible correlations between the number of individuals tested, confirmed, or deceased from this virus and the state's demographics. To conduct this data exploration, we will be utilizing an open-source data set from Kaggle that includes COVID-19 cases by state as well as general demographics for each state. The data set encompasses all 50 states, 1 state per record, as well as the District of Columbia (Washington D.C., the country's capital), for a total of 51 records. The features incorporated in this dataset are the following: State name, # tested, # confirmed, # deaths, # population, population density, gini coefficient for income inequality, # ICU beds, income per capita, GDP per capita, unemployment % of state, sex ratio males/females, smoking rate %, flu

deaths per 100,000, respitatory deaths per 100,000, # physicians, # hospitals, health spending, pollution, # med and large airports, % population in urban environment, age groups, and school closure date. This data set is being updated daily for the total number of individuals tested for COVID-19, confirmed cases of the virus, and deaths associated with the virus by state.

How to Solve the Problem

Algorithm Design

However, these machine learning models have not been deployed to forecast confirmed COVID-19 cases. The purpose of this paper is to 1) solve the problem of forecasting COVID-19 cases and 2) evaluate and compare performance of different machine learning models on forecasting COVID-19. In particular, we will attempt to solve the problem of forecasting by linear regression, LSTM, support vector machines, and XBboost.

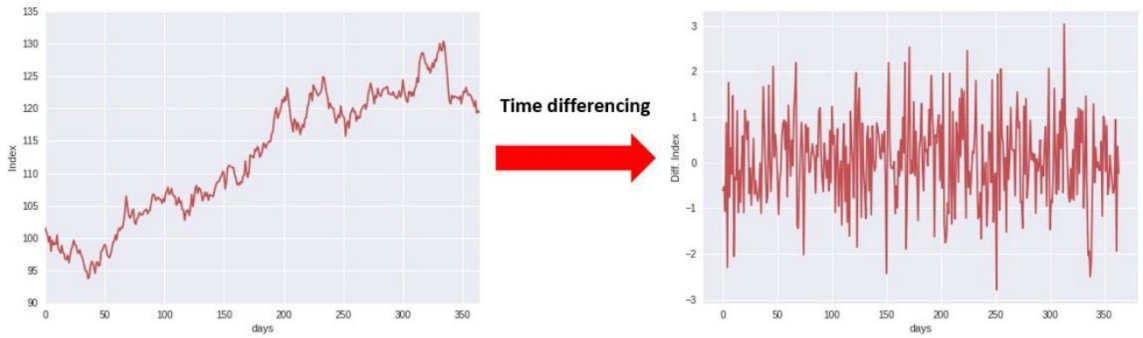
We will first perform exploratory data analysis to find any consistent patterns, significant trends, seasonalities, and any outliers present in the data. To identify strong relationships/ correlations among the variables, we will plot heatmaps and matrix scatterplots. We also use lag plots to check for serial correlation /autocorrelation.

In contrast to statistical time series forecasting, where time series data is required to be stationary (constant mean, variance, autocorrelation over time), there is no such census for making data stationary for machine learning. While some literature emphasizes the importance of preprocessing and making data stationary[Zhang], others suggest that machine learning models are capable of training and predicting raw time series data[Nelson].
Nonetheless, we will consider the common preprocessing techniques for time series data and

choose according to our findings from exploratory data analysis. Some common transformations for time series data include deseasonalization, Box Cox transformation to transform non-normal dependent variables so they have a “normal shape” and log transformation, which can transform data following exponential distribution to be linear by taking the logarithm of the values [Hyndman]. These transformations can also help stabilize the variance of the time series.

We will also consider DIFF, LAGGED- VAL and MOV-AVG preprocessing techniques, as they have been shown to have a huge impact on model performance [Ahmed]. The preprocessing techniques will be chosen based on the shape of time series data and model.

- For LAGGED- VAL (no special preprocessing), pick N lagged time series values which will be the N input variables, where the N is a parameter. The value to be predicted is the next value (for one-step ahead forecasting).
- For MOV-AVG (moving averages) - compute the moving averages with varying sized windows. The purpose of this is to smooth short term fluctuations so long term trends are highlighted.
- For DIFF (differencing) - compute the differencing between consecutive observations. Differencing can also reduce/eliminate trends and seasonality of the data while eliminating varying mean.



The transformations chosen will depend on the machine learning model and shape of the data. If the data indicates long term average to have some significance, MOV-AVG will be applied, etc. In the article “An Empirical Comparison of Machine Learning Models for Time Series Forecasting”, Ahmed demonstrated that “differencing is not always a good strategy for nonstationary time series, and the converse is true for stationary time series”. Moreover, they found that LAGGED-VAL and MOV-AVG yielded the best performance for machine learning models.

Once the data has been transformed appropriately, it will then input to our machine learning models: support vector machines, linear regression, XGBoost, and LSTM.

Language Used

The data cleansing, modeling, and data analysis for this research will be conducted with the Python programming language for its extensive support libraries, open source modules, ease of use, and speed.

Tools Used

Because machine learning can be computationally expensive, we will be using Google Collab, a free cloud service, that executes code on Google’s cloud servers, allowing us to leverage Google’s powerful hardware (specifically GPUs).

We will use matplotlib library for exploratory data analysis, such as scatter plots and heatmaps. We will be using scikit-learn Python library for linear regression

(sklearn.linear_model), support vector regression(sklearn.svm), and hyperparameter optimization (sklearn.GridSearchCV). To implement linear regression, scikit-learn offers two functions- LinearRegression (using closed form solution) and SGDClassifier(stochastic gradient descent).We will consider both of these during our project. To implement deep learning models, such as LSTM, we will be using Keras (Tensorflow's high level API). Specifically, Keras provides a Sequential model API, which we will use to create a Sequential instance and add layers to it.

How to Generate Output

After data cleansing has been completed, the training dataset will be inputted to each program for the different machine learning models in this study. The model's will then attempt to forecast COVID-19 confirmed cases. Additionally, validation of each model's output will be performed by using k-fold cross validation. This validation method is chosen for this study over splitting the entire dataset into training and testing portions as there is already very limited time-series data for COVID-19. Using k-fold cross validation to evaluate the different machine learning models will give us better confidence with our model results by splitting the dataset into training and testing subsets, training the model on the training data portion, and testing the model on the remaining testing data portion. However, with this method, this procedure is done k times, for different splitting subsets of the data set. The advantage of this method for our study is that we do not have to permanently leave out a certain percentage of the data set for testing purposes, as our data set is already limited. Rather, this method replaces the testing set back into the training set after its evaluation is complete, and selects another section of the data set for testing.

After the program has run through each model (SVR, XGBoost, LSTM, and Linear Regression) for each data set (confirmed cases in U.S) including k-fold cross-validation of their respective results, we will capture their forecast results as the models output as well as the forecast accuracy of each run.

How to Test Against Hypotheses

To address our hypothesis of model performance, based on related research in this field as well as general Machine Learning knowledge, several measures of model accuracy will be computed for each of the models to compare successes and shortcomings of each.

The forecasting accuracy of the models will be evaluated by the following statistical metrics: mean squared error(MSE), root mean squared error(RMSE), and SMAPE. From our research, we found all these three to be the most popular for evaluating performance of time series forecasting machine learning models[Hyndman]. RMSE is more sensitive to outliers as it will give a lot more weight to large errors[2].

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2} \qquad \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Both of these, however, scale dependent errors, meaning it can be very hard to compare errors between series of different units. This leads us to include “symmetric” mean absolute percentage error (SMAPE), a scale independent measure, as a performance metric in this study. Although it has been criticized as a symmetric measure[3](over- and under-forecasts are not treated

[2] https://otexts.com/fpp2/accuracy.html
[3] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0174202

the same), it is very robust to outliers and independent of units. The formula for SMAPE is as follows:

$$\frac{1}{M}\sum_{m=1}^{M}\frac{|\hat{y}_m - y_m|}{(|\hat{y}_m| + |y_m|)/2}$$

For the hypothesis of continued rising COVID-19 cases for the remainder of this research, we will continue to update our training dataset to include the new daily confirmed cases. Once the models in study have been assembled, we will run the updated data set daily through the models and capture the different accuracy metrics. If the number of confirmed cases do not rise, and rather remain constant, or drop, the machine learning models would not be capable of predicting and capturing that spontaneous decline. Therefore, we can test this hypothesis by continuously monitoring the accuracy rate of the models which may indicate a sudden change in the data through a plateau or decline in cases rather than an incline. Additionally, we can monitor the visual representations of the data set from the data exploration phase of this study including the time series plots of new daily cases, which will also reveal a potential change in trend.

V. IMPLEMENTATION

(Code) Part 1: Investigation

Feature engineering:

First, we did population adjustments to our data in order to state level comparison. Specifically, we calculated new features as per-capita. We saw this helped with identifying and calculating the correlations. So, instead of the raw number of confirmed cases, we considered data per-hundred thousand:

```
df1['Tested_per_thous'] = (df1['Tested']/df1['Population'])* 100_000
df1['Infected_per_thous'] = (df1['Infected']/df1['Population']) * 100_000
df1['Deaths_per_thous'] = (df1['Deaths']/df1['Population'])* 100_000
df1['ICU Beds_per_thous'] =  (df1['ICU Beds']/df1['Population'])* 100_000
df1['Respiratory Deaths_per_thous'] =  (df1['Respiratory Deaths']/df1['Population'])* 100_000
```

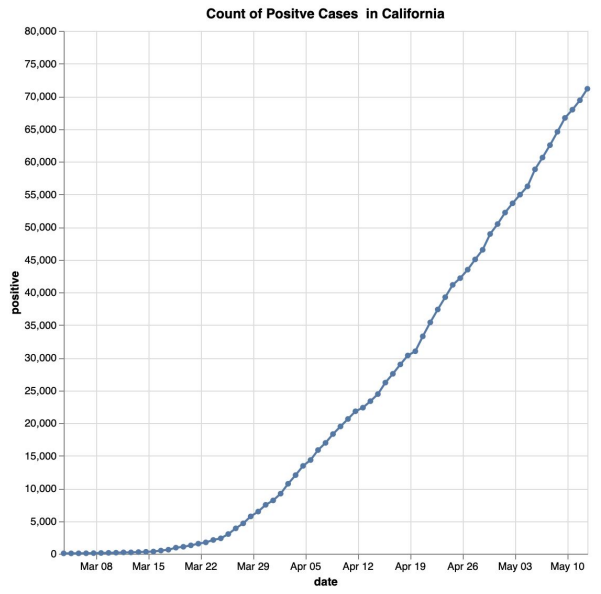
Exploratory Data Analysis:

Line Dot Plots-

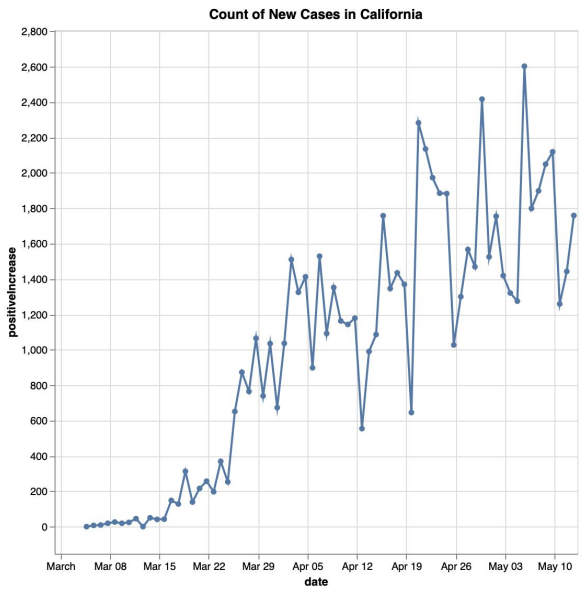
The following line plots gives us an overview of how the covid-19 data is distributed for California. We plotted the count of positive cases, number of deaths, new cases, new deaths, hospitalized, and the number of tests allotted in California. We are seeing an exponential growth in all the cases. Despite incidents starting as early as December 2019, the World Health Organization publicly announced deep concern on COVID-19 in March 2020. The CDC began covid-19 surveillance data in March, which is where our x-axis begins. The data tracker is updated daily and currently the numbers for positive cases in California exceed 70,000. These line plots provide huge significance to our project because it gives us insight to how we should expect our results to be after forecasting.

An interesting observation we found was that in the count of new cases, we saw no new cases recorded from April 11 to April 12. This was very out of the ordinary which led us to consider a few things. April 12 was Easter which is a recognized holiday in the U.S, there may have been no new recordings because many medical offices are closed, or they may have not been any test distributed on this day.

The line plots below show the trend of positive cases and new cases in California. We wanted to see how the trend of positive cases was growing. It is growing exponentially, thus we can see from this plot that our hypothesis of an increase in cases may hold true.




```
#@title
alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='positive:Q'
).properties(
    title = " Count of Positve Cases  in California",
    width = 500,
     height = 500
)
```

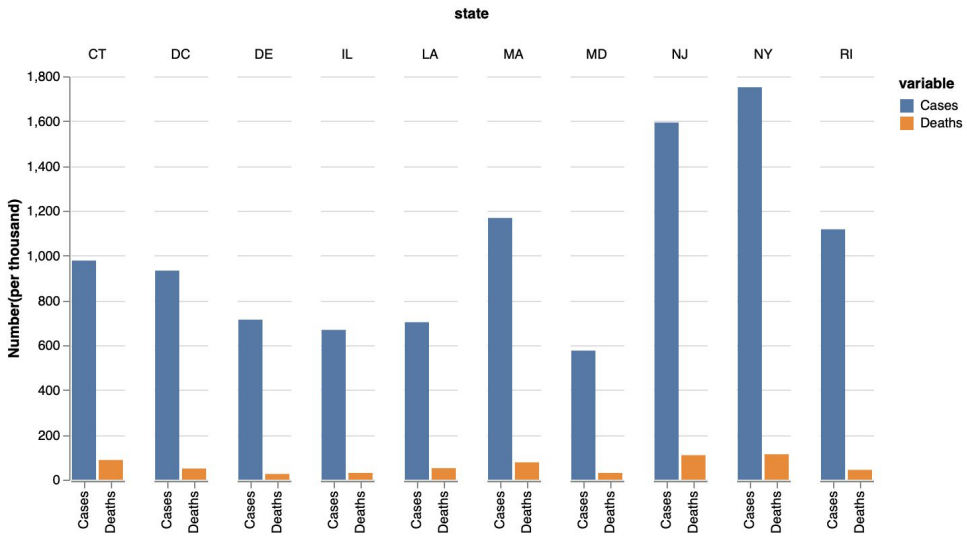



```
#@title
alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='positiveIncrease:Q',
    tooltip = ["date", "positiveIncrease:Q", "postive:Q"]
).properties(
    title = "Count of New Cases in California",
    width = 500,
     height = 500
).interactive()
```

In the process of evaluating the dataset, we thought it would be important to show the deaths compared to the cases.

```
data = pd.concat([top_10_states, top_death_states], axis=1)
data = data[['state', 'cases_per_thousand',"deaths_per_thousand" ]]
data = data.loc[:, ~data.columns.duplicated()]

melted_df = data.melt("state")
melted_df["variable"].replace({"cases_per_thousand": "Cases", "deaths_per_thousand": "Deaths"}, inplace=True)
```

```
alt.Chart(melted_df).mark_bar().encode(
    alt.X('variable:N', axis= alt.Axis(title='',orient = "bottom")),
    alt.Y('value:Q', axis = alt.Axis(title='Number(per thousand)')),
    color= alt.Color('variable:N'),
    column ='state:O'
).configure_view(
    stroke='transparent'
)
```



Top States for COVID-19:

To look at the bigger picture, we found plotting the top ten states with the highest numbers useful information for our project. We listed the top 10 states with most deaths, with New York coming in at numbers in the range of 20,000. Visualizing we wanted to see how those compared to one another through line plots and a bar chart. This allowed us to see the severity of each state, in which we quickly realized that New York has exceeding numbers.

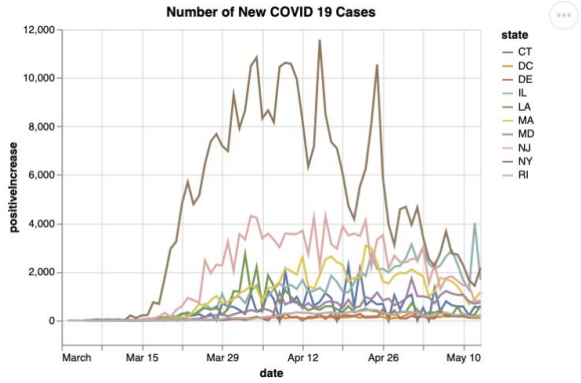
It is very clear from our graphs that the 10 states with the highest number of positive cases are: New York, New Jersey, Illinois, Massachusetts, California, Pennsylvania, Michigan, Texas, Florida, and Georgia. New York has been hit the hardest, with the most cases and deaths than any other state. We found that California has twice the population but less than half the testing. Therefore, it is surprising to see that California does not have more cases given this fact.

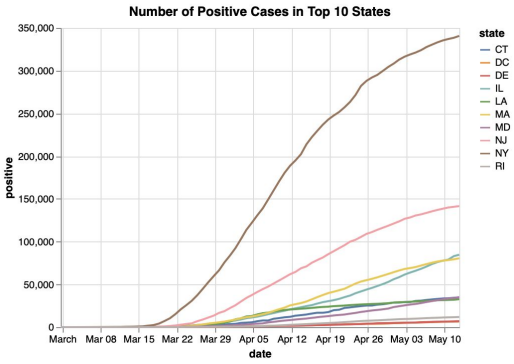
	state_name	death
37	New York	22013.0
34	New Jersey	9702.0
21	Massachusetts	5315.0
24	Michigan	4714.0
41	Pennsylvania	3943.0
16	Illinois	3792.0
7	Connecticut	3125.0
5	California	2934.0
20	Louisiana	2381.0
10	Florida	1898.0

```
most_recent = states_df.head(56)
most_recent.nlargest(10, "death")[['state_name', 'death']]
```

This graph below displays the number of new cases by day where we can see the change each day and observe any significant changes to daily values.

```
#static chart
alt.Chart(filtered_df).mark_line().encode(
    x='date:T',
    y='positiveIncrease',
    color='state',
).properties(title = "Number of New COVID 19 Cases")
```

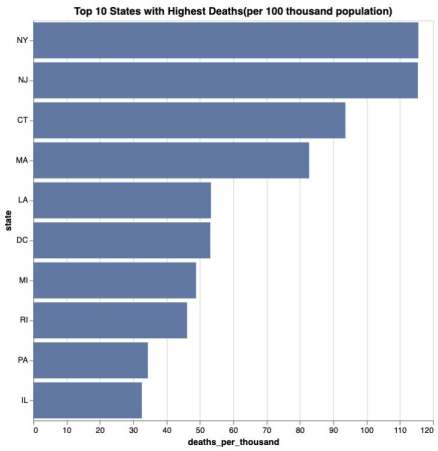




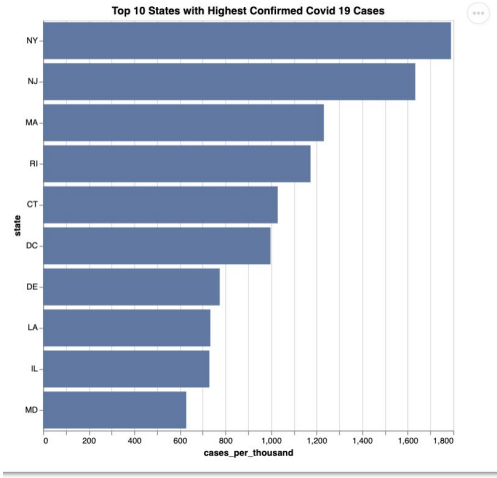

```
states_lst = list(top_10_states["state"])
filtered_df = states_df[states_df.state.isin(states_lst)]
filtered_df[["state","date", "positive"]]


alt.Chart(filtered_df).mark_line().encode(
    x='date:T',
    y='positive',
    color='state',
).properties(
    title = " Number of Positive Cases in Top 10 States"
)
```

In order to see which states have been impacted the most severely, we plotted the highest number of deaths. We also compared the top 10 states with the highest number of deaths with the top 10 states with highest number of cases. In both instances, we found it true that most cases also had most deaths.

```
#@title
top_death_states = top_10_states[['state', 'deaths_per_thousand',"state_name", "death

bars = alt.Chart(top_death_states).mark_bar().encode(
    x = "deaths_per_thousand:Q" ,
    y = alt.Y('state:N', sort = '-x'),
    tooltip = ['top_death_states:Q', "deaths_per_thousand:Q" ,"death:Q", "state_name"
).properties(
      title = "Top 10 States with Highest Deaths(per 100 thousand population)",
      width = 500,
     height = 500
).interactive()
bars
```



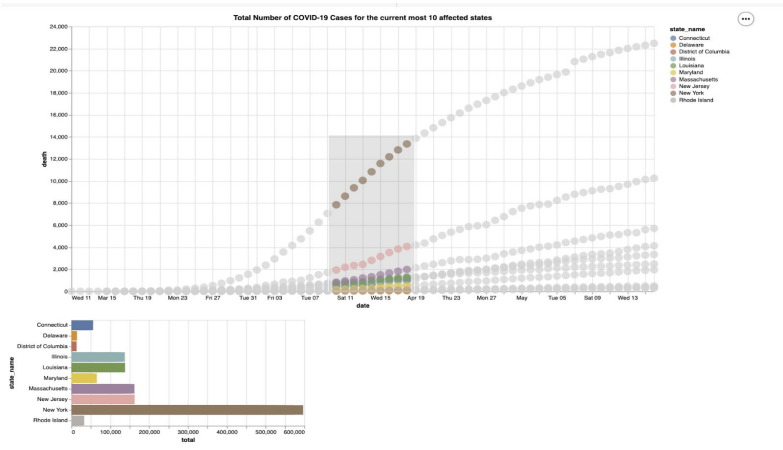

```
#@title
bars = alt.Chart(top_case_states).mark_bar().encode(
    x="cases_per_thousand:Q" ,
    y=alt.Y('state:N', sort='-x'),

    tooltip=['positive',"cases_per_thousand:Q","state"],
).properties(
      title = "Top 10 States with Highest Confirmed Covid 19 Cases",
        width= 500,
        height=500
).interactive()
bars
```

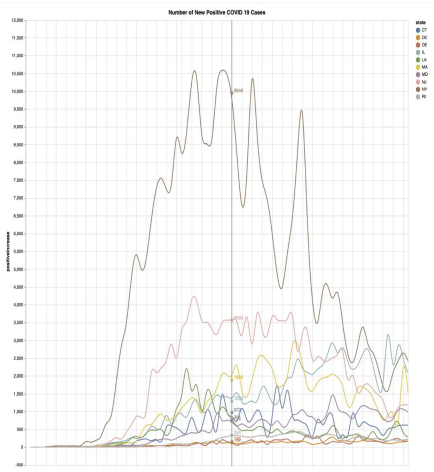


Interactive Plots with Altair for Deeper Analysis

We found static plots to be very useful to give a broader perspective on how each state was different from one another in cases. However, to further understand certain anomalies, we

needed more features, such as the ability to click on certain data points for more information. Below are two of the most helpful plots. The following interactive scatter plot below showcases the total number of covid-19 cases in the US, for the top 10 states. The following left scatter plot and histogram are connected. In particular, we implemented a filter, which allowed us to drag and create a filter. Dragging along the filter (gray box) on the scatterplot changed the corresponding histogram. This was really insightful and revealing to understand how the number of cases change over time among states. The plot on the right which is also interactive, allowed us to see specific numbers and they changed over time for different states.





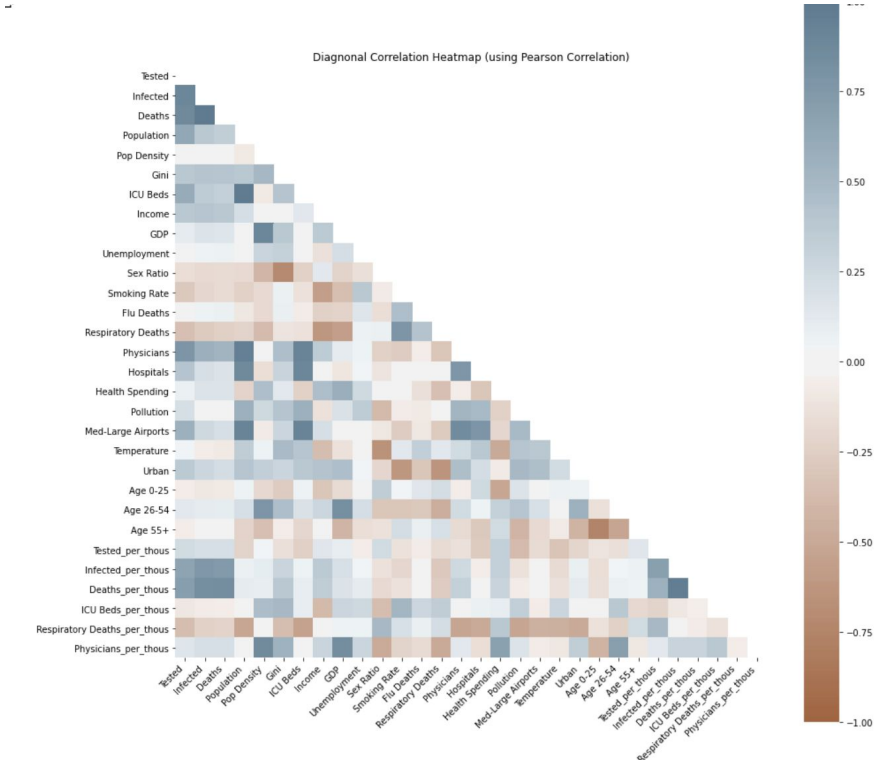

```
states_df["date"] = pd.to_datetime(states_df["date"], format='%Y%m%d')
states_lst = top_10_states["state"]


source =states_df[states_df.state.isin(states_lst)]

brush = alt.selection(type='interval')

points = alt.Chart(source).mark_circle(size=200).encode(
    x='date:T',
    y='death:Q',
    color=alt.condition(brush, 'state_name:N', alt.value('lightgray')),
    tooltip=[ 'date', 'total', 'death', "state_name"]
).add_selection(
    brush
).properties(
      width= 1000,
      height=500,
      title = "Total Number of COVID-19 Cases for the current most 10 affected states",
)

bars = alt.Chart(source).mark_bar().encode(
    y='state_name:N',
    color='state_name:N',
    x='total:Q'
).transform_filter(
    brush
)

points & bars
```

```
#for top 10 states
source = filtered_df[["state","date", "positiveIncrease"]]

# Create a selection that chooses the nearest point & selects based on x-value
nearest = alt.selection(type='single', nearest=True, on='mouseover',
                        fields=['date'], empty='none')


# The basic line
line = alt.Chart(source).mark_line( interpolate='basis').encode(
    x='date:T',
    y='positiveIncrease:Q',
    color='state:N'
)

# Transparent selectors across the chart. This is what tells us
# the x-value of the cursor
selectors = alt.Chart(source).mark_point(point=True).encode(
    x='date:T',
    opacity=alt.value(0),
).add_selection(
    nearest
)

# Draw points on the line, and highlight based on selection
points = line.mark_point().encode(
    opacity=alt.condition(nearest, alt.value(1), alt.value(0))
)

# Draw text labels near the points, and highlight based on selection
text = line.mark_text(align='left', dx=5, dy=-5).encode(
    text=alt.condition(nearest, 'positiveIncrease:Q', alt.value(' '))
)

# Draw a rule at the location of the selection
rules = alt.Chart(source).mark_rule(color='gray').encode(
    x='date:T',
).transform_filter(
    nearest
)

# Put the five layers into a chart and bind the data
alt.layer(
    line, selectors, points, rules, text
).properties(
    width=1000, height=1000,
    title = " Number of New Positive COVID 19 Cases"
)
```

Heatmap

To quickly and most effectively visualize correlations between the features, we plotted a heatmap to identify potential correlations among variables. Here is our code and result:

```
[ ] import seaborn as sns
    import matplotlib.pyplot as plt
    import numpy as np

    fig, ax = plt.subplots(figsize=(15,15))
    corr = df1.corr()

    mask = np.triu(np.ones_like(corr, dtype=np.bool))

    ax = sns.heatmap(
        corr,
        mask = mask,
        vmin=-1, vmax=1, center=0,
        cmap=sns.diverging_palette(20, 220, n=200),
        square=True
    )
    ax.set_xticklabels(
        ax.get_xticklabels(),
        rotation=45,
        horizontalalignment='right'
    );
    plt.title ("Diagnonal Correlation Heatmap (using Pearson Correlation)");
```



We then plotted scatterplots between variables we suspected had a strong correlation. Statistical correlation measures the strength between two variables. To summarize the correlation, we also computed the Pearson coefficient using scipy.stats.pearsonr(x, y), which returns the Pearson correlation coefficient and the p-value for testing non-correlation. The correlation coefficient ranges from -1 to 1, indicating strong negative correlation to strong positive correlation. Value of 0 indicates no correlation.

Furthermore, we computed P-values to determine if the correlation coefficient is statistically significant. The null hypothesis is stated as "No correlation between the two variables". If the $p < 0.05$, we stated that the correlation is statistically significant and can still be due to chance. However, if $p < 0.01$, the correlation coefficient is highly statistically significant and it cannot be attributed to random chance.

Scatter Plot/Correlation

The following are the correlation plots that gave us statistically significant results, using Pearson coefficient as mentioned above. To reiterate, in order to do a state by state comparison, we transformed the raw number of confirmed cases to the number of confirmed cases per (100,000) people. When we looked at those tested, we found correlations between infected, deaths, and respiratory deaths. This makes sense because those who were infected and died, had to have had been tested beforehand in order to come to that conclusion. It is important to note that we assume there have been those who have gone untested and contracted covid-19. Health spending was another feature we looked at and found correlations between infected, death, and testing. We predict that states who have higher health spending are most likely going to have

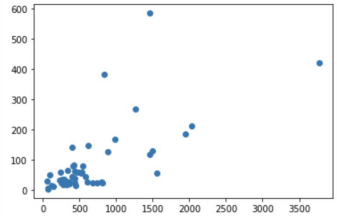
more testing which leads to uncovering more people who are infected and unfortunately eventually die.

In mostly all of these cases, there is one outlier and that is New York. The question arises then, why are we seeing such high numbers in this one state? One reasoning we discussed was the timing of social distancing. Humans can be carrying the disease for 2 weeks at most with little to no symptoms. This means they had the opportunity to unknowingly spread the disease amongst many more. Time is very sensitive to covid-19, if New York had implemented their stay-at-home orders a little too late, this could explain why we are seeing high numbers. We also considered the amount of tests distributed in New York, and who were receiving tests. Only those who showed severe symptoms were being tested. Given this fact, New York has still tested more of the population than California. Also, population density is another factor we considered, New York has a high population density but not by much compared to similar states such as California. There would be no reason to believe that this could play such a huge factor in the special case of NY.

The following code depicts how we plotted our graphs and used Pearson's coefficient in order to test the correlation relationship. Here you will see the correlations between: Tested and Infected, Tested and Deaths, Tested and Respiratory Deaths, Infected and Health Spending, Death and Health Spending, Deaths and Physicians, Tested and Health spending. Lastly, the code we followed in order to do all the plots.

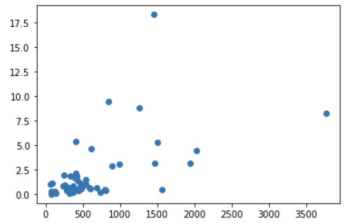
Tested and Infected

```
data1: mean=655.224 stdv=638.914
data2: mean=84.495 stdv=111.784
Pearson Coefficient
(0.6894975646316808, 2.2018792718413426e-08)
```



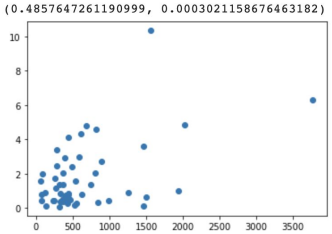
Tested and Deaths

```
Pearson Coefficient
(0.5529806714228136, 2.575571104387515e-05)
```



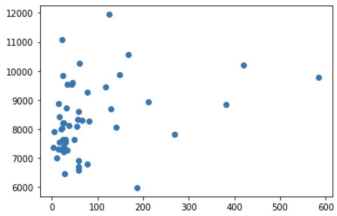
Tested and Respiratory Deaths

```
data1: mean=655.224 stdv=638.914
data2: mean=1.775 stdv=1.919
(0.4857647261190999, 0.0003021158676463182)
```



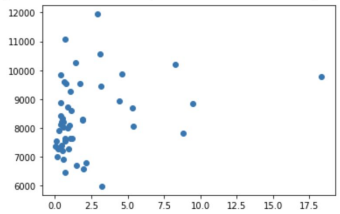
Infected and Health Spending

```
Pearson Coefficient
(0.319189374083178, 0.022428161013691512)
```



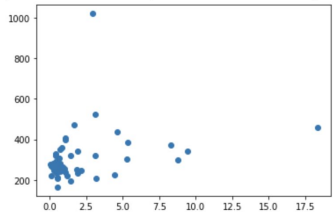
Death and Health Spending

```
Pearson Coefficient
(0.29166690726531974, 0.037831973610154584)
```



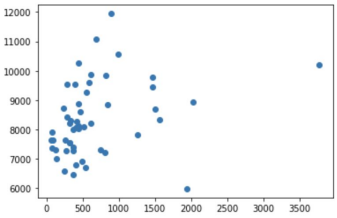
Deaths and Physicians

```
Pearson Coefficient
(0.29079641191199757, 0.038434589171752624)
```



Tested and Health Spending

```
(0.3129457150316881, 0.02535404075948896)
```



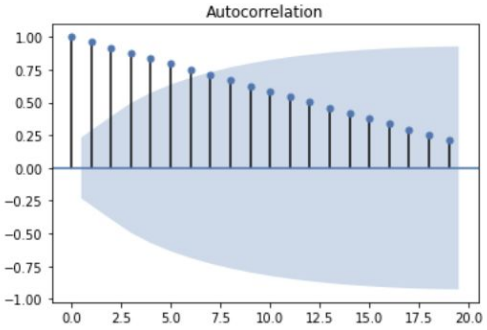
CODE EXAMPLE (Followed same format for plots)

```
x = df1['Tested_per_thous']
y = df1['Infected_per_thous']

print('data1: mean=%.3f stdv=%.3f' % (mean(x), std(x)))
print('data2: mean=%.3f stdv=%.3f' % (mean(y), std(y)))

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()
```

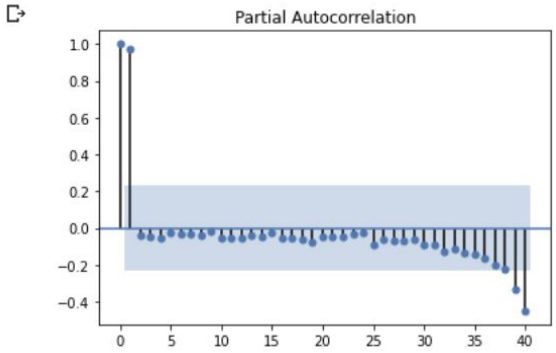
Autocorrelation and Partial Autocorrelation plots

Autocorrelation and partial autocorrelation plots are very crucial in time series analysis and forecasting and are used to summarize the strength of a relationship with an observation in a time series with observations at prior time steps. Autocorrelation is a strong and robust filter to detect bias if noise outweighs the actual signal for the objective time-series. This test is useful as it reveals information both about the variable as well as the model. Autocorrelation can reveal wrong estimations of the error variances which in turn makes confidence interval calculations, significance tests, etc. invalid.

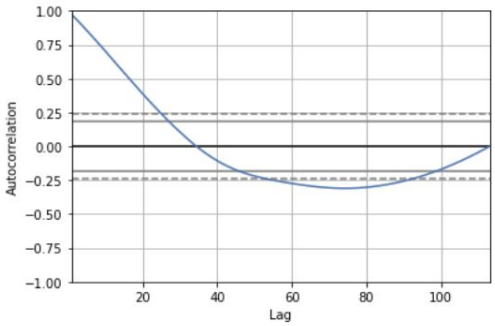
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(ca_data["positive"])

#On the graph, there is a vertical line (a "spike") corresponding to each lag.
#The height of each spike shows the value of the autocorrelation function for the lag.
```




```
[48] plot_pacf(ca_data["positive"], lags=40)
     plt.show()
```



```
[49] from pandas.plotting import autocorrelation_plot

     autocorrelation_plot(us_df["positive"]);
     #There is autocorrelation and this statistically significant
```



(Code) Part 2: Time Series Forecasting (Preprocessing and Model Code)

General approach:

In general, our approach was to first do feature engineering, apply preprocessing techniques (which are discussed in the following three sections). When applying multiple preprocessing techniques, we followed the following general order: Box Cox, differencing, MinMax Scaling/Standardization. In order, to transform our time series data into a supervised learning problem, we used sliding window/lagged value techniques. We then used grid search for models like XGBoost and SVR in order to find optimal value for hyperparameters.

BoxCox transformations

Since Covid cases are exponential, doing near-log transform (boxcox optimizes the lambda transform and resulted in approximately 0 which is a log transform) creates a more-linear representation of the data to use for linear regression.

```
7  us_X = us_df[['month', 'day']]
8  us_Y = us_df["positive"]
9
10 us_box_total, lam = boxcox(us_Y)
11 print('Lambda: %f' % lam)
```

```
Lambda: 0.046134
```

Power transform on confirmed COVID-19 cases where lambda (power transform coefficient) is found automatically with a value of 0.046 (close to 0 which is log transform)

Scale/Normalization

Normalization was tested to scale down the data since the number of confirmed COVID-19 cases are now in the millions in the United States. We normalized y-axis (number of confirmed cases) to around 0-1 (scaled automatically/optimally by MinMaxScaler() sklearn function) to get a more concise reference for error values and accuracy of predictions.

```
minmax_scaler = MinMaxScaler()

scaled_train_labels = minmax_scaler.fit_transform(train_labels.reshape(-1, 1))
scaled_test_labels = minmax_scaler.fit_transform(test_labels.reshape(-1, 1))
```

Standardization

Standardization was tested in attempts to make the distribution of our data Gaussian.

```
scaler =  StandardScaler()
scaler.fit(train_labels.reshape(-1, 1) )
scaled_train_labels = scaler.transform(train_labels.reshape(-1, 1) )
scaled_test_labels = scaler.transform(test_labels.reshape(-1, 1) )
```

Differencing

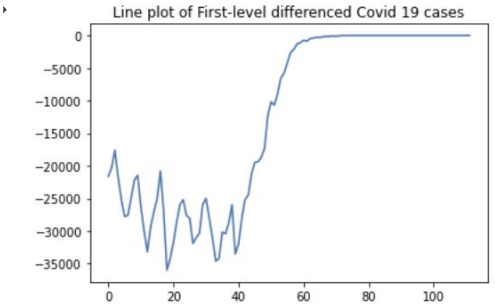
In this project, we wanted to examine if differencing would impact performance on machine learning models. Differencing is a preprocessing technique used to remove trends and seasonality from a time series dataset. Put differently, it removes autocorrelation and makes data

stationary. Since we detected autocorrelation, we tried to perform trend differencing on some of our models.

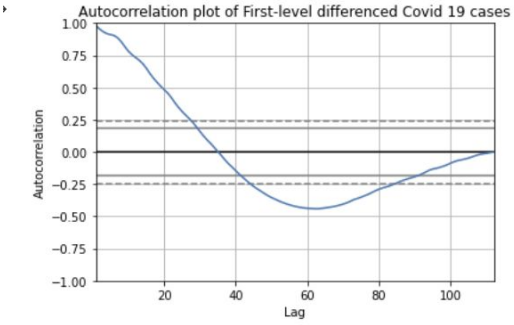
```
[50] def difference(inp):
       "Used to make the data stationary."
       diff = list()
       for i in range(1, len(inp)):
         value = inp[i] - inp[i - 1]
         diff.append(value)
       return diff
```

After applying differencing once, we got the following results:

```
] raw_cases = us_df["positive"]

  first_level =  difference(raw_cases)
  plt.plot(first_level);
  plt.title("Line plot of First-level differenced Covid 19 cases");
```

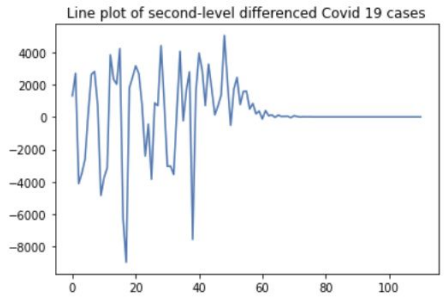



```
!]
  autocorrelation_plot(first_level)
  plt.title("Autocorrelation plot of First-level differenced Covid 19 cases");
```



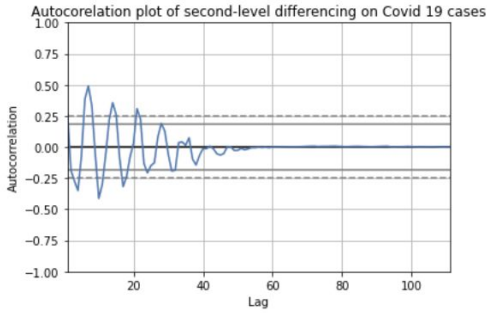
This indicated that there was still autocorrelation so performed differencing again:

```
second_level = difference(first_level)
plt.plot(second_level);
plt.title("Line plot of second-level differenced Covid 19 cases");
```




```
autocorrelation_plot(second_level)
plt.title("Autocorelation plot of second-level differencing on Covid 19 cases");
print("Data is now stationary!")
```

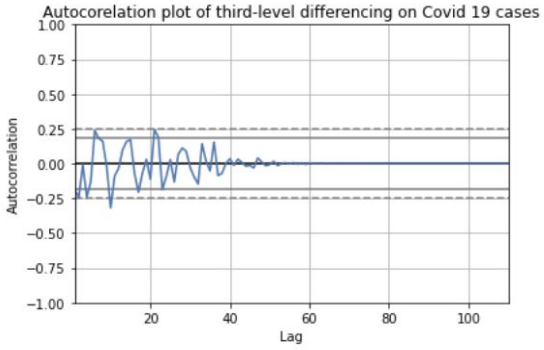
```
Data is now stationary!
```



We were able to achieve stationary data after three level differencing:

```
third_level_diff = difference(second_level)
autocorrelation_plot(third_level_diff)
plt.title("Autocorelation plot of third-level differencing on Covid 19 cases");
print("Data is now stationary!")
```

```
Data is now stationary!
```



Sliding Window / Lagged Values: Converting time series problem into supervised learning

In order to use machine learning algorithms for time series forecasting, you have to reframe/restructure the problem so it is a supervised learning problem. So to do this, we used the sliding window method to to phrase the time series data as supervised learning. Instead of using the month and date as features to predict the number of cases in the time-series, we also investigated the use of sliding windows in order to predict confirmed cases. In this technique, a manually set number of previous days is used to predict the following day's value. For example, the test below uses a time step of 7 which means the first 7 days number of confirmed cases will be the independent value to predict the number of confirmed cases on day 8, then day 2-8 will be used to predict day 9, day 3-9 to predict day 10, etc.

```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)

timesteps =7
train_x, train_y = create_dataset(train_data , train_data , timesteps)
test_x, test_y = create_dataset(test_data , test_data , timesteps)
```

Grid Search:

We performed a 5 fold cross validation grid search. Here are the snippets of the code we used to find the optimal hyperparameter for the estimations.

For the SVR function linear and RBF kernel there are hyperparameters for C, epsilon, and gamma(only for nonlinear SVR). C is a regularization parameter for SVR that optimizes the tradeoff between correct classifications and maximizing the margin whereas gamma defines how close (high value) or far (low value) a training data influence reaches. The grid search computes the accuracy of different values of each hyperparameter using the given data. These optimal hyperparameters are then used in our final models.

```
gsc = GridSearchCV(
        estimator=LinearSVR(),
        param_grid={
            'C': [0.1, 1, 100, 1000],
            'epsilon': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10],
        },
        cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs=-1)


gsc.fit(scaled_train_x, scaled_train_y )
best_params = gsc.best_params_
print(best_params)
lin_svr = LinearSVR( C= best_params["C"], epsilon=best_params["epsilon"])
```

For XGboost there are 5 key hyperparameters to be optimized for its grid search: gamma, subsample, learning rate, number of estimators, and max depth.

- ***Gamma***, as explained for SVR's grid search, defines how close or far a training data influence reaches.
- ***Subsample*** is the ratio of training data sampled before growing trees to prevent overfitting. This ratio of sampled data will be performed once for every boosting iteration.
- The learning rate is used to reduce overfitting of training data as this can be a problem with gradient boosted decision trees by slowing the learning of the model. The learning rate can manage the weight of new trees added to the model whereas smaller rates generally require the model to include more decision trees.

- The number of estimators (***n_estimators***) is used to tune the number of decision trees used in the computation. As each additional tree tries to correct the errors of previous trees, it is beneficial to test larger and larger values for the number of trees.
- However, another component of decision trees that is essential to consider is its depth. Having a shallow tree (***max_depth*** is low integer value) generally results in poor performance as they do not capture the necessary details of the data. Similarly, having a deep tree (max_depth is a high integer value) generally results in overfitting the data by capturing too many details from the training dataset which makes generalization difficult or impossible for incoming data. Therefore, the max_depth parameter is used to tune the optimal maximum depth for each decision tree when implemented with the given data.

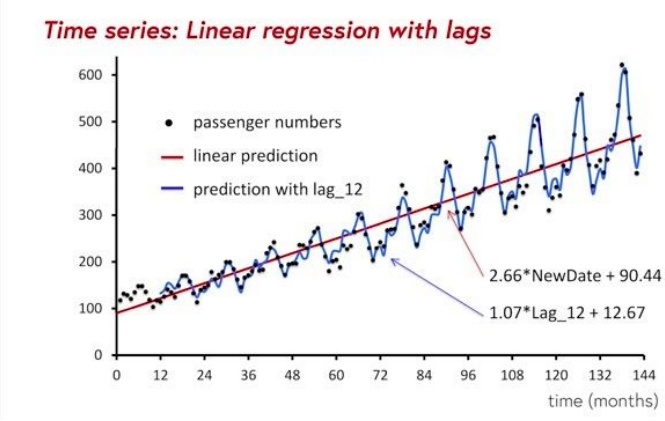
```
]  #'min_child_weight':[4,5,6],
   print("Parameter optimization")
   xgb_model = XGBRegressor(objective ='reg:squarederror')
   clf = GridSearchCV(xgb_model,
                      {'max_depth': [2,4,6,10,20],
                      'gamma' :[0,1,5,10],
                       'learning_rate': [.0001,.001,.001,.10],
                       'subsample':[0.3,.5,0.8,1],
                        'n_estimators': [50,100,200,1000,1500]})
   clf.fit(scaled_train_x, scaled_train_y)
   print(clf.best_score_)
   print(clf.best_params_)
   #print(clf.cv_results_)
```

Models:

Linear Regression Model

Linear Regression was the most simple model used in this study as it can be used for time-series forecasting. The use of linear regression was performed both with linear predictions and non-linear predictions. Linear predictions were tested for date/COVID-19 day number as the

univariate independent variable and number of confirmed/positive cases as the predicted dependent variable. Time-series are rarely simple enough to be linearly increasing or decreasing over time which is a major benefit of utilizing lagged values. Nonlinear predictions were tested for sliding window lagged values where the previous X number of days confirmed case numbers were used to predict the subsequent next day case numbers. Using lagged value windows creates more complex models and breaks out of the linear paradigm and creates nonlinear predictions while still using the simplicity of the linear regression model. The figure below, from futurelearn.com, shows a linear regression model being utilized for time series, both for a linear prediction (red line) as well as a nonlinear prediction from lagged values (blue line) which allows the model to fit cyclical and varying data.



Source: https://www.futurelearn.com/courses/advanced-data-mining-with-weka/0/steps/29456

For both linear and nonlinear predictions, the following lines of code were used to implement a linear regression model to the data.

```
regressor = LinearRegression()
regressor.fit(scaled_train_x, scaled_train_y) #training the algorithm

y_pred = regressor.predict(scaled_test_x)

preds = pd.DataFrame({'Actual': scaled_test_y.flatten(), 'Predicted': y_pred.flatten()})
```

Support Vector Regressions Models - Linear and Nonlinear

We tested linear SVR as well as nonlinear SVR with the RBF and polynomial kernels. Linear SVR and RBF SVR were found to capture the time-series data for confirmed cases most accurately. Therefore, these two models were tested under various preprocessing techniques and lagged value time steps in attempts to find the most optimal combination. It was found that linear SVR with lagged value of 7 days, auto power transformation, no trend differencing, and auto normalization yielded the lowest errors for MSE and SMAPE.

```
lin_svr = LinearSVR( C= best_params["C"], epsilon=best_params["epsilon"])

lin_svr.fit(X= scaled_train_x, y= scaled_train_y)
```

XGBoost Model

For XGBOOST, after using GridSearchCV to find optimal values on the data, we pass those values for the parameters in XGBRegressor method, where the objective function is squared loss. Here is a sample of the code:

```
model2 = XGBRegressor(n_estimators = 1500 , max_depth = 6, learning_rate=0.1, subsample=0.3)
history = model2.fit(scaled_train_x, scaled_train_y)
```

For XGBOOST experiments, we had two main approaches: We used the *month* and *day* as the features(X) and the Covid19 cases.

```
us_df['date'] = pd.to_datetime(us_df['date'], format='%Y%m%d')

us_df['month']= pd.DatetimeIndex(us_df["date"]).month
us_df['year']= pd.DatetimeIndex(us_df["date"]).year
us_df['day']= pd.DatetimeIndex(us_df["date"]).day

us_X = us_df[['month', 'day']]
us_Y = us_df["positive"]
```

We will later discuss in the "Output Analysis" that this resulted in very poor performance. The second approach was to use lagged values/sliding window approach, where the input features for each time step *i* were the previous lag values for *i,* where lagged value code was discussed above in this subchapter.

LSTM Network

There were multiple models we tried for LSTM. Out of all the models, LSTM took the longest to tune. In order to use LSTM for time series forecasting, we first applied Box Cox transformation, then transformed the observations to have a specific scale. Specifically, to rescale the data to values between -1 and 1 to meet the default hyperbolic tangent activation function of the LSTM model. We also tried to scale between 0 and 1. The difference in performance results will be discussed in output analysis. Lastly, we transformed the time series data into supervised learning problems by using the create_dataset method.

Our first approach to generate time series data was to use the TimeseriesGenerator provided by Tensorflow:

```
#LAG preprocessing to frame a sequence as a supervised learning problem
#returns a sequence of overlapping windows
#batch_size = # of samples to return on each iteration
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

n_input =10 # lag
n_features = 1
generator = TimeseriesGenerator(scaled_train_data, scaled_train_data, length = n_input, batch_size = 1)
for i in range(len(generator)):
    x, y = generator[i]
    #print('%s => %s' % (x, y))
```

We understand this is the most common technique used to generate lagged values and sliding windows. When we initially used the Time Series Generator, it worked in terms of

allowing us to train the LSTM model, but it gave us problems splitting into a validation set. So we decided to use our own function create_dataset which gave more flexibility. Specifically, the number of lagged values was set to seven, so each forecast used the previous seven time steps to make a prediction.

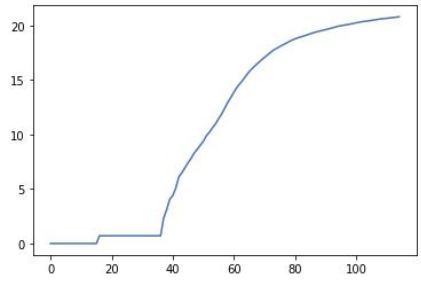
Here is code implementation of the feature engineering steps described above:

LSTM

Preprocessing /Data Preparation

```
[ ] from scipy.stats import boxcox

    box_total, lam = boxcox(total)
    print('Lambda: %f' % lam)
    plt.plot(box_total)
```

```
Lambda: 0.050997
[<matplotlib.lines.Line2D at 0x7fb4a1d5d748>]
```




```
def difference(inp):
  "Used to make the data stationary."
  diff = list()
  for i in range(1, len(inp)):
    value = inp[i] - inp[i - 1]
    diff.append(value)
  return diff

diff = difference(box_total)
```

```
[ ] total = np.array(box_total).reshape(-1, 1)
    num = int(len(total) *.80)

    train_data = total[:num]
    test_data = total[num:]
```

```
[ ] train_data = np.array(total[:num]).reshape(-1,1)
    test_data = np.array(total[num:]).reshape(-1,1)

    scaler = MinMaxScaler()
    scaler.fit(train_data.reshape(-1, 1))
    scaled_train_data = scaler.transform(train_data)
    scaled_test_data = scaler.transform(test_data)
```

Before passing our time series into LSTM, one has to reshape the input into three dimensions, which contain **samples(**aka **Batch size)** number of samples that are trained together for one epoch, **time steps**,and **features**.

```
[11]  time_steps = 7

      # reshape to [samples, time_steps, n_features]

      scaled_train_data = pd.Series(scaled_train_data.reshape(1,-1)[0])
      scaled_test_data = pd.Series(scaled_test_data.reshape(1,-1)[0])

      X_train, y_train = create_dataset(scaled_train_data, scaled_train_data, time_steps)
      X_test, y_test = create_dataset(scaled_test_data, scaled_test_data , time_steps)

      print(X_train.shape, y_train.shape)
```

```
(84, 7) (84,)
```

```
[12] X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
     X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
```

As shown by the code above, we reshaped X_train so the number of time steps were equal to the number of observations in X_train, and the number of features was equal to the lag value.

We picked the best architecture through trial and error, testing various combinations of number of layers and neurons to see which model would give us the best model. While trying different number of layers and neurons, we kept the following in mind:

- Using too few **neurons** in the hidden layers will result in underfitting and the model won't be able can't detect the signals in a complicated data set. On the other hand, using too many neurons not only results in greater training time but can overfit to training data.
- We started with a simple model and strategically added multiple hidden layers and increased the number of neurons, but we found that the number of layers did not always

correlate with better performance on test data. So we proceeded to use the simple model (shown below).

- We also used a **dropout** layer with a 10% drop out rate to reduce overfitting.
- **Batch size:** The number of batch sizes determine how many samples a network used to perform weight update. Smaller batch sizes are noisy and offer a regularizing effect.They generally result in rapid learning. On the other hand, larger batch sizes result in slower, less volatile and more stable learning processes.
- One **epoch** means that each sample in the training dataset has had an opportunity to update the internal model parameters. So, one epoch consists of one or more batches. For example, as above, an epoch that has one batch is called the batch gradient descent learning algorithm.

From multiple trails, the resulting best model is:

```
model = keras.Sequential()
model.add(keras.layers.LSTM(50, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences = True))
model.add(keras.layers.Dropout(0.1))
model.add(keras.layers.LSTM(units = 50, return_sequences = True))
model.add(keras.layers.Dropout(0.1))
model.add(keras.layers.Dense(1))
model.compile(loss='mean_squared_error', optimizer = keras.optimizers.Adam(0.1))
```

We then fit the model using the X_train and y_train, with epoch of 100 and batch_size of 31, and 30% validation split. Since this is a time series and sequence of training instances is important, shuffle was set to False.

```
15] history = model.fit(
        X_train, y_train,
        epochs = 100,
        batch_size = 31,
        validation_split = 0.3,
        verbose = 1,
        shuffle = False
    )
```

```
2/2 [==============================] - 0s 17ms/step - loss: 0.0014 - val_loss: 7.1448e-04
Epoch 74/100
2/2 [==============================] - 0s 18ms/step - loss: 0.0018 - val_loss: 0.0027
Epoch 75/100
2/2 [==============================] - 0s 19ms/step - loss: 0.0018 - val_loss: 0.0028
```

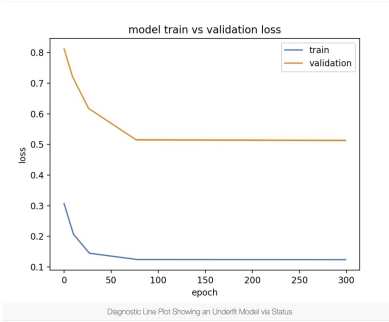
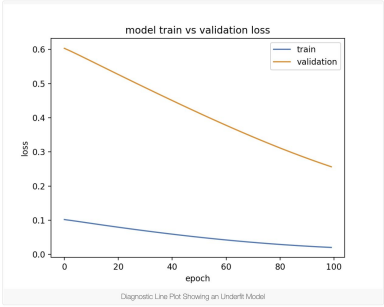
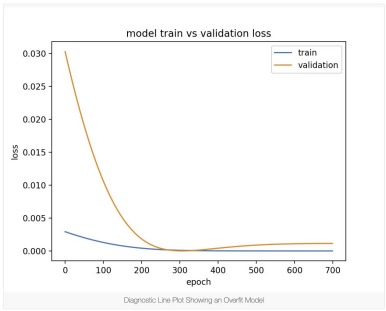
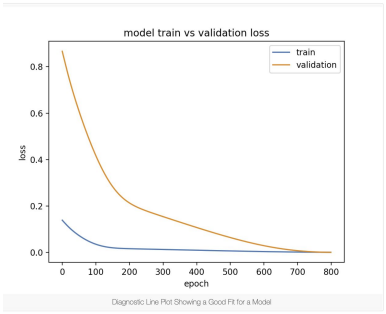
This *fit*() function keeps track of the loss while it trains and will return a history trace. In order to effectively diagnose the behavior of our LSTM models, we decided to plot diagnostic line plots using the training and validation loss of our LSTM model.

```
[16] plt.plot(history.history['loss'], label='train')
     plt.plot(history.history['val_loss'], label='test')
     plt.legend();
```

We used diagnostic line plots to detect possible underfitting and overfitting. In general, here are some examples from the book that demonstrate how to use diagnostic line plots to understand the behavior of the model.

Examples of Diagnostic models for Different Performances

Source: https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/

Underfit	Overfit	Ideal Model Performance
 Diagnostic Line Plot Showing an Underfit Model via Status  Diagnostic Line Plot Showing an Underfit Model	 Diagnostic Line Plot Showing an Overfit Model	 Diagnostic Line Plot Showing a Good Fit for a Model

In general, we used these samples above for interpretation and making adjustments to our LSTM model accordingly.

Evaluation- SMAPE, MSE, RMSE

MSE and SMAPE were the main model evaluation metrics used in this study. MSE is a critical error statistic used in many machine learning algorithms and was computed directly from an sklearn function, seen below. SMAPE was the additional metric to capture error rates for

various model testings in this study since it provides the percentage of error which is more easily interpreted than MSE which can have a range of values from 0 to infinity.

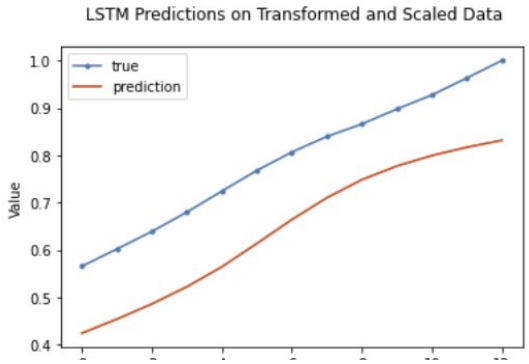
```
] import numpy as np
  def smape(A, F):
      return 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))
```

```
from  sklearn.metrics import mean_squared_error
```

Here is a sample code we used for LSTM evaluations:

```
#Evaluation of LSTM
y_pred = model.predict(X_test)

def smape(A, F):
    return 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))

print('\n Evaluation on test data: MSE')
results = model.evaluate(X_test, y_test, batch_size = 128)
print(results)

print('\n Evaluation on test data: SMAPE')
print(smape(y_test, y_pred.reshape(-1,1)))
print()

plt.plot(y_test, marker='.', label="true")
plt.plot(y_pred.reshape(-1,1), 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" LSTM Predictions on Transformed and Scaled Data" , pad =20);

plt.show();
```

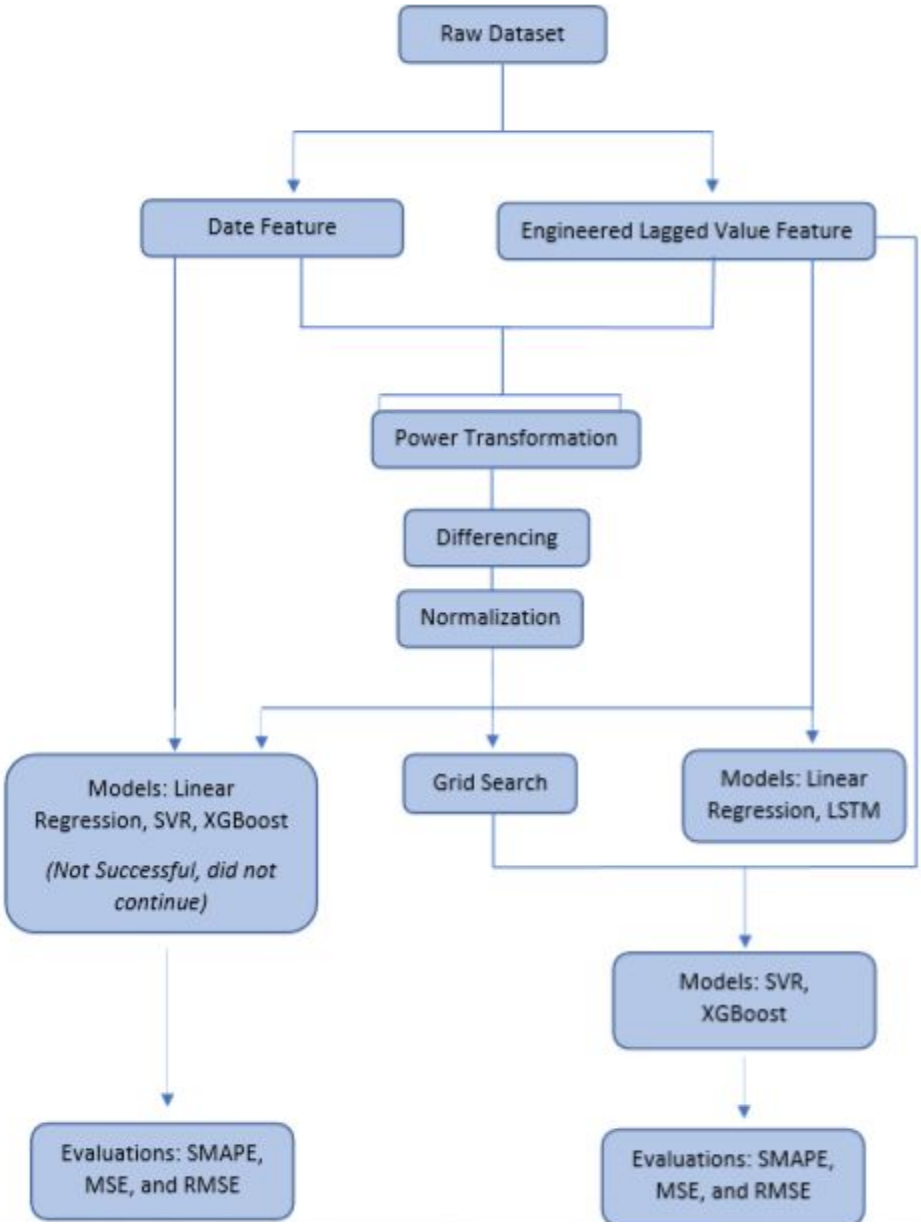
```
 Evaluation on test data: MSE
1/1 [==============================] - 0s 1ms/step - loss: 0.0208
0.02082975022494793

 Evaluation on test data: SMAPE
366.86302172509227
```



Design Document and Flowchart

The following flowchart was used to test the various models in this study. We had two main approaches to testing the models: using date/time as the predictor, and using lagged values as the predictor which was an engineered value. Both approaches were followed by three main preprocessing techniques of a power transformation to make the data stationary, differencing which is essential to time-series data to remove trends and/or seasonality, and normalization to reduce the range of our predicted COVID-19 cases values for easier interpretation. For the model testings that used the date feature as the predictor, the preprocessing steps were followed by model testing for linear regression, SVR, and XGBoost. However, using the model evaluation metrics of MSE, SMAPE, and RMSE, as well as examining the prediction plots, these attempts were not accurate/successful.

The models that were tested using lagged values from sliding windows showed promising results after preprocessing techniques. Therefore, additional tests to improve performance were done including grid search for the SVR and XGBoost models. Linear regression and LSTM were tested after preprocessing was performed and all four models were again evaluated with various metrics including MSE, SMAPE, and RMSE.


Raw Dataset
Date Feature
Engineered Lagged Value Feature
Power Transformation
Differencing
Normalization
Models: Linear Regression, SVR, XGBoost
(Not Successful, did not continue)
Grid Search
Models: Linear Regression, LSTM
Models: SVR, XGBoost
Evaluations: SMAPE, MSE, and RMSE
Evaluations: SMAPE, MSE, and RMSE

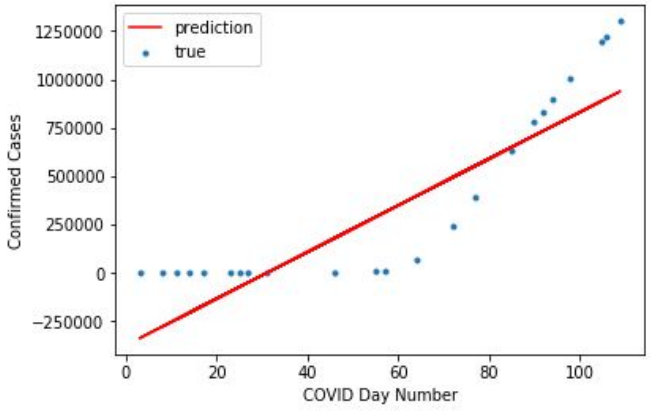
VI. DATA ANALYSIS AND DISCUSSION

Output Generation and Analysis

I. Linear regression

The first tests performed for linear regression were to create linear predictions. For the first attempt, the model was trained on the raw dataset for positive cases against the date variable. The datetime variable was converted to an integer for COVID-19 day countings. As the first date of testing in the United States and data contained in the dataset was January 22nd, 2020, this record's date was changed to 1, and every subsequent date/record was increased by 1. These values were used in the initial test rather than lagged values, as well as no preprocessing techniques included. This was conducted as the first test of this model to form a baseline of error values for eventual implementation of preprocessing techniques. It was hypothesized that if the given preprocessing technique did not vastly improve the results of the model, it should not be implemented in attempts to keep the transformations and resulting model as simple and explainable as possible. The first test conducted on the raw data resulted in the following prediction and error values with Linear Regression, as shown in Model 0.

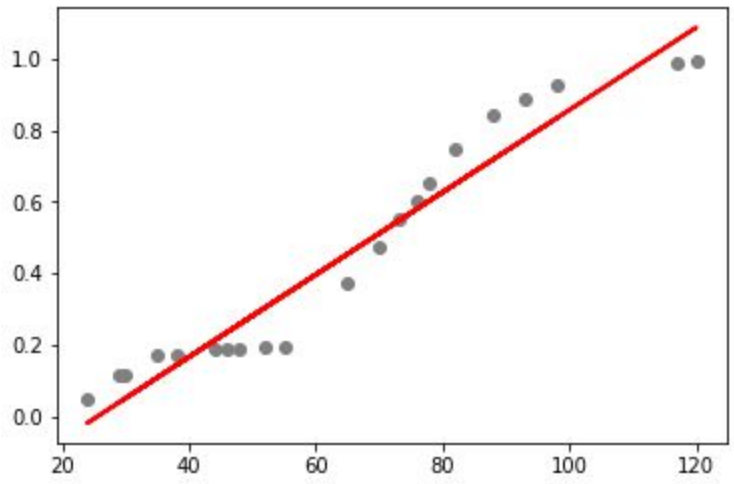
Linear Regression Model 0: Raw Data, No Transformations:




```
Mean Squared Error
49347904775.50535

Symmetric Mean Absolute Percentage Error
120.59411512406648
```

The second linear prediction model tested also used the COVID Day Number as the independent variable to predict the number of positive COVID cases. However, in this test, the number of cases were preprocessed using power transform in an attempt to make the time-series data more stationary and therefore easier to capture its trends. This step to create an effectively stationary transformation proved to be more difficult than expected. As seen in the figure below, using BoxCox() on this date-to-cases prediction yielded a cyclical s-shaped curve to the data which was not expected, as seen in the figure below for Model 0.1. The original distribution of cases as seen in Model 0 is fairly exponential in shape. Therefore, in performing a power transform (of approximately a log transformation), it was expected the resulting data would be fairly linear. This expected result was achieved by using lagged values and is explained in further detail below.

Linear Regression Model 0.1: Power Transformation, Standardization, Normalization:



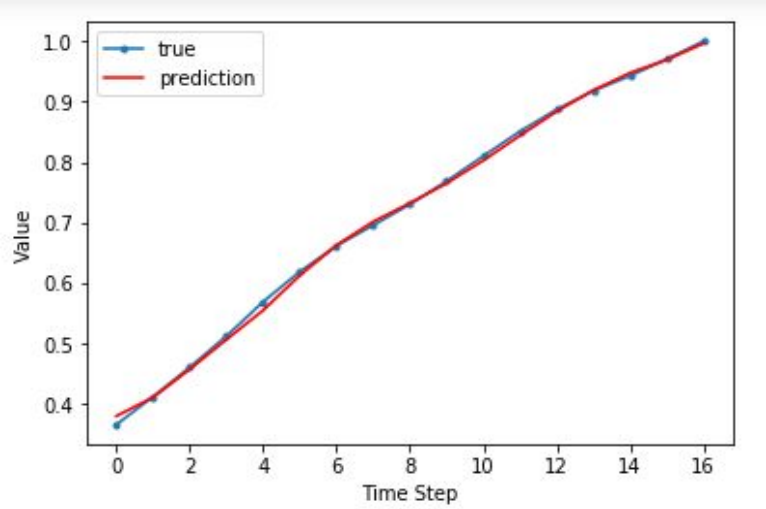
The first method to improve the results from Model 0 and 0.1 was to convert the time-series data for dates into timestep lagged values to predict following positive case values.

Starting at a timestep of 1, a test was conducted for each timestep, increasing by 1 day for each interval, and recording the resulting MSE and SMAPE, as shown in the table below. Power transform and normalization were also performed for the subsequent tests as they were found in all different machine learning models of this study to vastly improve the accuracy results. From a timestep of 1 day, each succeeding test resulted in lower error values through a timestep of 6 days. We found that a timestep of 7 and 8 days slightly raised the MSE and SMAPE error values. Therefore, a lagged value of 6 days was selected for the final linear regression model for this study. Additional tests for this model included manually specifying the values for range normalization to (-1,1) whereas all other tests were computed automatically with the MinMaxScaler() sklearn function. This test resulted in higher error rates than allowing the function to find the optimal range for normalization, therefore was not used in our final linear regression model. A summary of test variations and results can be found in the table below.

Linear Regression with Different Preprocessing Techniques

	Model 0	Model 1.1	Model 1.2	Model 1.3	Model 1.4	Model 1.5	Model 1.6	Model 1.7	Model 1.8	Model 2
Preprocessing										
Lagged Value	None	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	6
Box Cox /Log	None	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Scaling/Normalizing	None	Minmax()	Minmax()	Minmax()	Minmax()	Minmax()	Minmax()	Minmax()	Minmax()	**Minmax(-1,1)**
MSE (Test)	49347904776	0.001348057	0.000220618	0.000107409	8.45E-05	5.36E-05	4.02E-05	6.44E-05	6.61E-05	0.000160831
SMAPE (Test)	120.5941151	16.37057734	3.737763698	2.002087713	1.512598561	1.055856231	0.84284873	0.987350005	0.97504241	7.03589239
Notes:	*LR on raw set*						*lowest error*	*error increasing again*		*auto minmax is better*

Of all 10 models tested for linear regression including different lagged values, a lagged value of 6 days, auto power transformation and auto range normalization performed yielded the lowest error rates. A plot of this test is shown below.

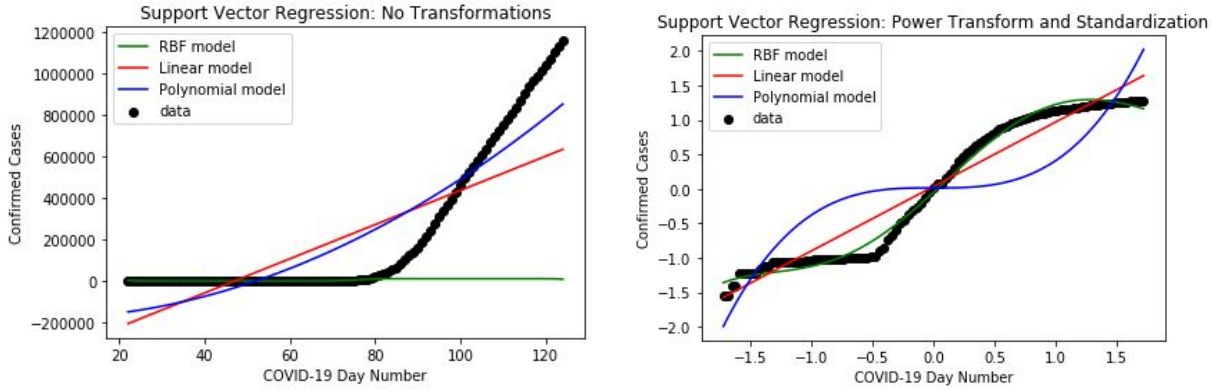



```
Mean Squared Error
4.020778387946393e-05

Symmetric Mean Absolute Percentage Error
0.8428487298294237
```

II. Support Vector Regression (Linear and Nonlinear)

The first tests conducted for SVR were to explore the general fits of our data, both raw and transformed values, with SVR under all 3 main kernels used in this model: linear, RBF (nonlinear), and polynomial (nonlinear). This general exploration was done by feeding the model the entirety of the raw dataset then the preprocessed dataset (power transform and standardization). Both of these tests are shown below. Like the first attempt to power transform the positive cases for linear regression, the right plot below shows the unexpected transform results because it is using the date/COVID day number values rather than lagged values which were found to smooth out the transformation into the expected linear trend. From the plot below on the left, we find that RBF does not fit the raw time-series data in the slightest. The linear

kernel, similarly to the first attempt of linear regression, also does not capture the data well. A degree 2 polynomial is fit to this data in attempts to capture its distribution. From the plot on the right, the SVR function automatically fits a degree 3 polynomial to the data but is reversely synced with the increases and decreases of the data. However, the RBF kernel fits the values of the transformed data very closely, even before we transform the date values to lagged values. The official models tested below (Model 1-8) include the lagged value transformations and become the main models to consider for final selection.



For support vector regression, we used both linear support vector regression (LinearSVR) and nonlinear support vector regression (SVR)from sklearn.svm module. For nonlinear support vector regression, we went with the rbf kernel. In total, there were 12 resulting models. The performance output for these models is included below.

The table below includes eight of the twelve models, in which the lagged values remained consistent. The lag value for all of them was 7, so for each of the predictions, the previous seven time steps were used.

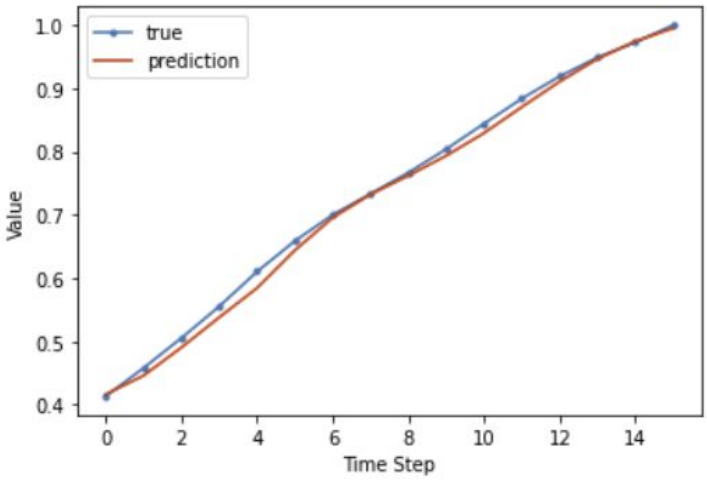
	Support Vector Regression: Performance Results							
	Model 1	**Model 2**	**Model 5**	**Model 5.1**	**Model 5.5**	**Model 6**	**Model 7**	**Model 8**
Features:								
Preprocessing								
Lagged Value	7	7	7	7	7	7	7	7
Box Cox /Log	Yes	Yes	**No**	No	No	Yes	Yes	**No**
Differencing	No	No	No	No	No	No	No	**No**
Scaling/Normaliz	Minmax(0,1)	Minmax(0,1)	Minmax(0,1)	**Minmax(-1,1)**	**StandardScaler**	**None**	*None*	**None**
Model								
Objective	Linear	**Non-linear**	Linear	Linear	Linear	Linear	**Nonlinear**	Linear
Kernel	-	**RBF**	-	-	-	-	rbf	-
Hyperparamters								
C	1000	1000	1000	100	1000	1	1000	1
Epsilon	0.0001	0.0001	0.001	0.0001	0.0001	0.0005	0.001	0.5
Gamma	-	0.005	-	-	-		0.001	
MSE	**0.00009108727**	0.000135	0.000276	0.00049046	0.0033577	0.0246214	0.00213	307124112.3
SMAPE	1.145	1.455	2.419158	9.68705	11.82743	0.810457	**0.236**	1.256246775
RMSE	0.009544	0.011619	0.0166132	0.0221463	0.0579457	0.1569121	0.0461519	17524.95684

(SVR) Best 2 Models

The best two models were Model 1 and Model 7:

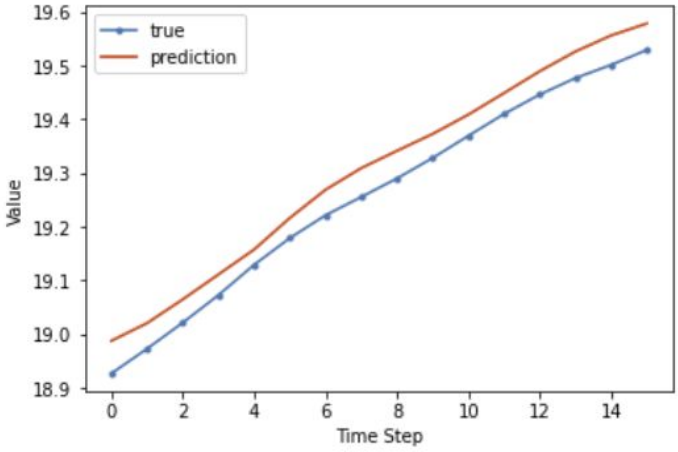
Model 1 used LinearSVR on Box Cox transformed, MinMax scaled (values in range[0,1]) time series data with lagged values with the previous lagged as features. The MSE for Model 1 was 0.00009 and SMAPE was 1.145 on test data. After GridSearchCV, the optimal values for C and epsilon were 1000 and 0.0001, respectively. Model 3.3, the same model but larger lag value (10 instead of 7), gave very similar performance with MSE of 0.0001094 and SMAPE of 1.072. The optimal values for C and epsilon, as given by GridSearchCV, were the same as model -1000 and 0.0001, respectively.

Model 1 SVR prediction plot:



Model 7 resulted in the lowest SMAPE. Model 7 was a **nonlinear SVR model** (rbf kernel) that used BoxCox transformed data and lag of 7 time steps. Interestingly, this model gave the lowest SMAPE results even without any normalization or standardization.

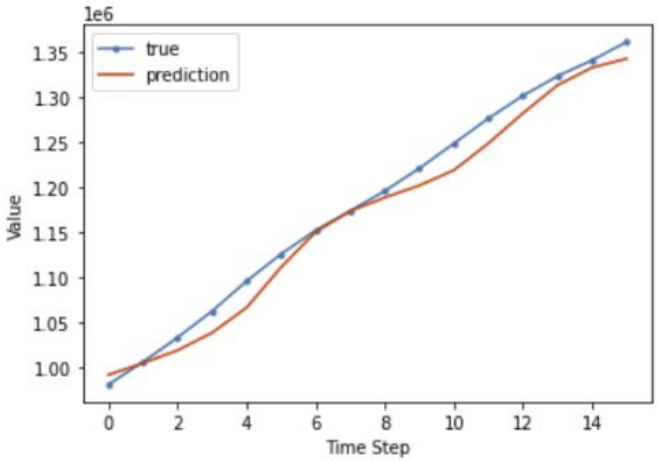
Model 7 prediction plot:




```
Mean Squared Error
0.002130263488241505

Symmetric Mean Absolute Percentage Error
0.2360064682298046
```

(SVR) Performance on Raw Data

We also wanted to examine the performance of LinearSVR on raw data (untransformed and unnormalized) with a lag value of seven. When we tested this(Model 8), our model had a MSE of 307124112.3 and SMAPE of 1.256. But, as illustrated, still predicted quite nicely:



(SVR) Testing different lag values for different features

We wanted to test the effect of different lag values on performance of linear support vector regression for time series forecasting. We found the greater lag values resulted in better performance as shown in the table below:

	Linear SVR: Performance for Different Lag Values			
	Model 3.1	**Model 3.2**	**Model 3.3**	**Model 3.4**
Features:				
Preprocessing				
Lagged Value	**3**	**5**	**10**	**8**
Box Cox /Log	Yes	Yes	Yes	Yes
Differencing	No	No	No	No
Scaling/Normaliz	Minmax(0,1)	Minmax(0,1)	Minmax(0,1)	Minmax(0,1)
Model				
Objective	Linear	Linear	Linear	Linear
Kernel	-	-	-	-
Hyperparamters				
C	100	1000	100	1
Epsilon	0.0005	0.0001	0.0001	0.001
Gamma	-	-	-	-
MSE	0.001026	0.000804	0.0001094	0.000114
SMAPE	5.068	4.284	1.072	1.2952
RMSE	0.0320312	0.0283549	0.0104594	0.0106771

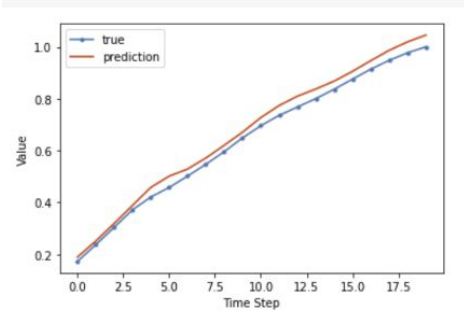
LinearSVR: Experimenting Different Techniques (Normalization/Standardization/None)

	LinearSVR Model Performances		
	Model 5	**Model 5.1**	**Model 5.5**
Features:			
Preprocessing			
Lagged Value	7	7	7
Box Cox /Log	No	No	No
Differencing	No	No	No
Scaling/Normali	**Minmax(0,1)**	**Minmax(-1,1)**	**StandardScaler**
Hyperparamters			
C	1000	100	1000
Epsilon	0.001	0.0001	0.0001
MSE	0.000276	0.00049046	0.0033577
SMAPE	2.419158	9.68705	11.82743
RMSE	0.0166132	0.0221463	0.0579457

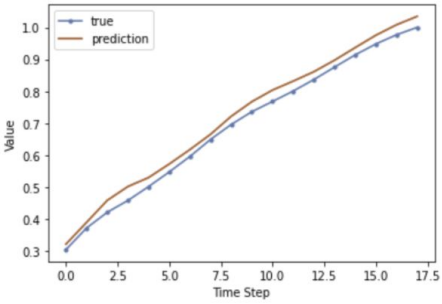
Performance comparison of models 5, 5.1, and 5.5, illustrate the importance of testing different standardization and normalization techniques. All three models were linearSVR models with lag value of and no Box Cox transformation on time series data. They only differed in the normalization/standardization techniques (with other variables constant) they used. Model 5, which used the default feature range for MinMaxScaler ([0,1]), had the best performance. Model 5.1 had the next best performance with a feature range of [-1,1]. Model 5.5, which used the StandardScaler, had the worst performance.

(SVR) Selected Prediction Plots

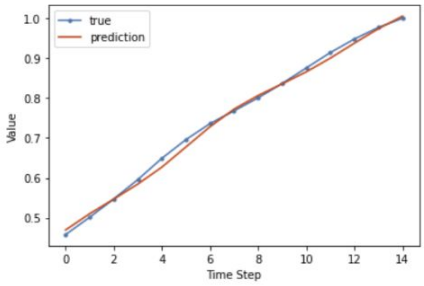
Model 3.1:



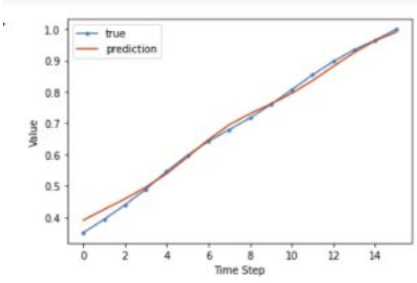
Model 3.2:



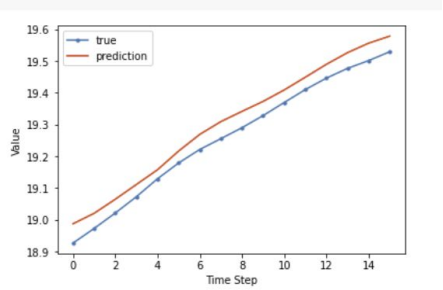
Model 3.4:



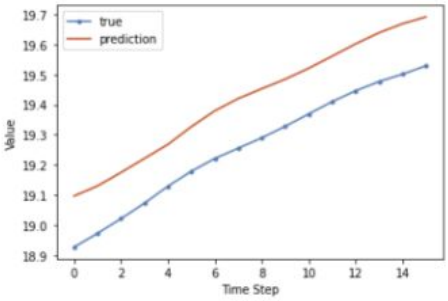
Model 5:



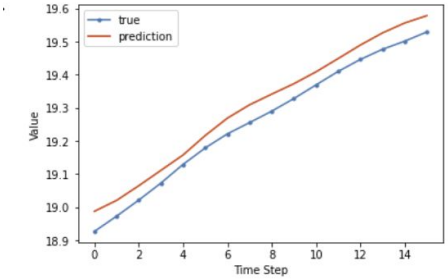
Model 5.5:



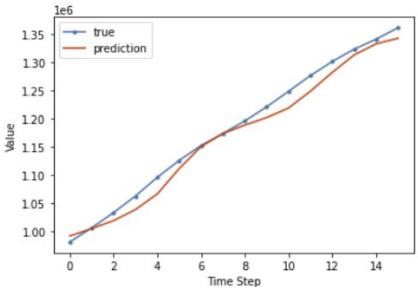
Model 6:



Model 7:



Model 8:

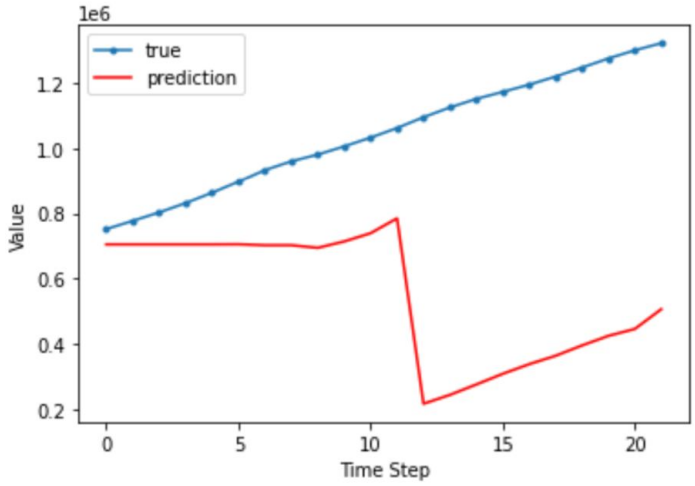


III. XGBoost

There were 3 general approaches (a total of 18 models) our team generated to explore the potential effects of differencing and features.

(XGBoost) Approach 1: Using Date Features (Month, Day)

For the first main approach, we decided to use month and day as 2 input features for the model, with y as the number of COVID19 cases. Even with the Box Cox transformation and Minmax Scaling(or Standardizing), the model resulted in very poor performance. The image below (Model 1) was the best performance we achieved by transforming the data with BoxCox and then standardizing it for preprocessing techniques. However, as highlighted by the image below, even with the best model, the model was not able to make useful forecasts.



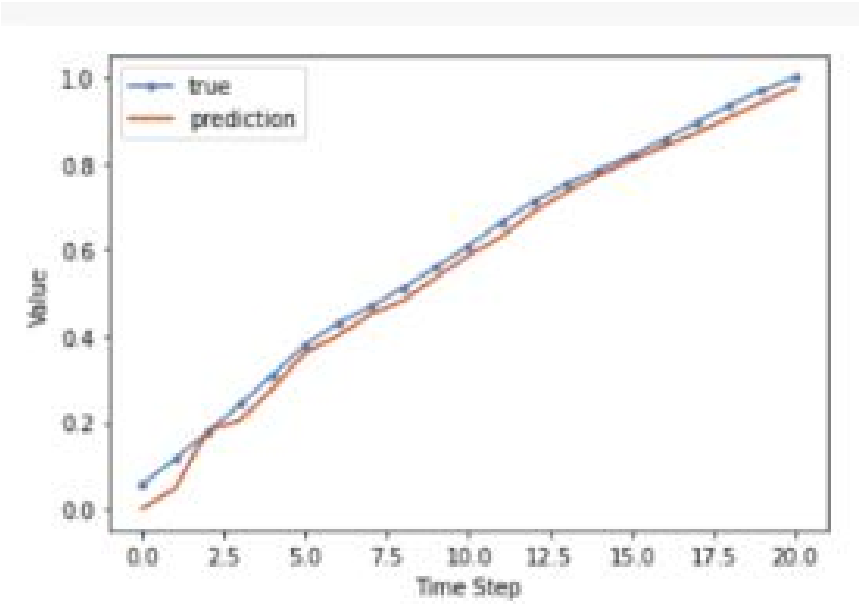
The performances of the models tried using month and day as features are summarized below:

	XGBoost: Using Date Features		
	Model 0	**Model 0.5**	**Model 1**
Features:	Month, day	Month, day	Month, day
Preprocessing			
Box Cox /Log	Yes	Yes	Yes
Scaling/Normali	Minmax	Minmax	StandardScaler
MSE (Test)	883237648.7	168	2.199
SMAPE (Test)	69	21.56	6.789
RMSE(Test)	29719.3144	12.96148	1.4829

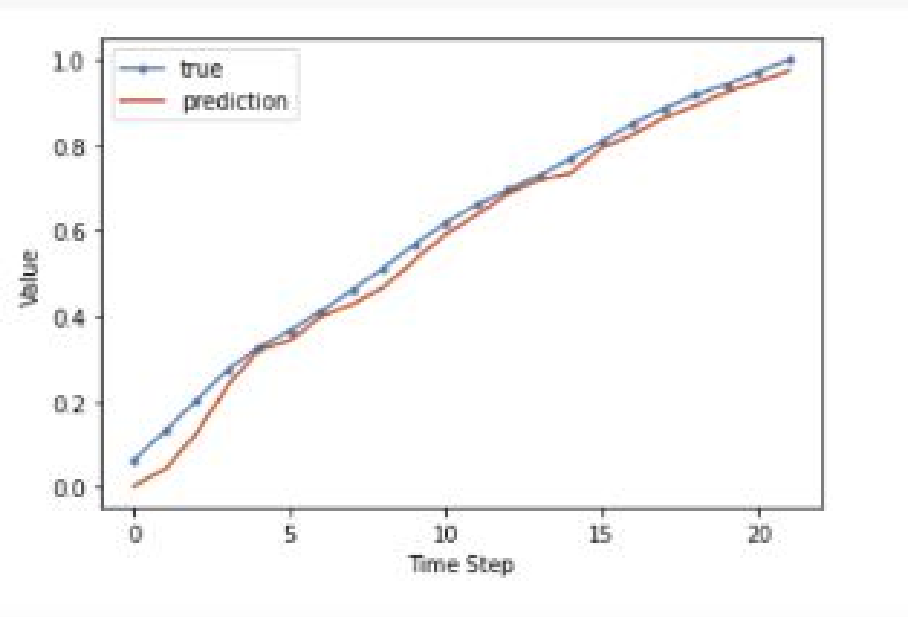
(XGBoost) Approach 2: Lagged Values Features

Next, instead of using month and day as features, we decided to use the *sliding window* /lagged value approach to frame the time series data into a supervised problem. This proved to be an incredibly effective technique to generate accurate forecasts! Overall, model 2.8 and 2.9 gave the best performances for us, both used Box Cox transformed and scaled data (scaled values were in range [0,1]). Model 2.8 used a lag of 2 and it resulted in the lowest SMAPE of all models tested. Model 2.9 used a lag of 1 and it resulted in the lowest MSE of all models tested. Below are images of model 2.8 and 2.9.

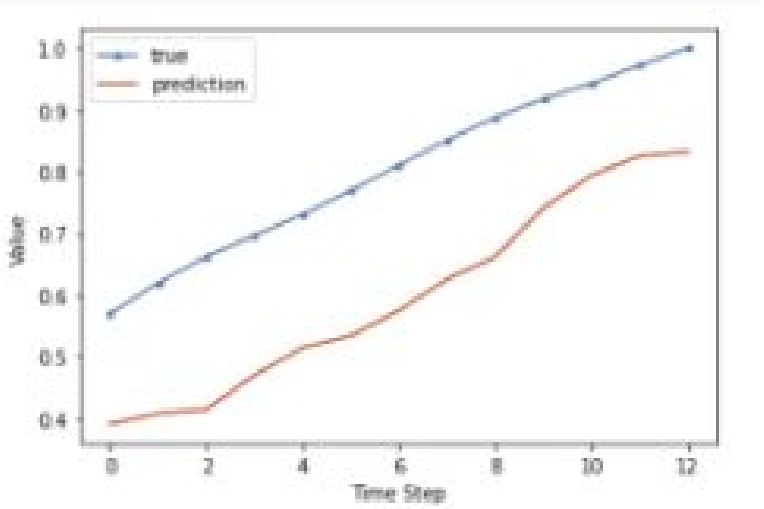
Plot of prediction model 2.8 (lag of 2) against true values:



Plot of prediction model 2.9 (lag of 1) against true values:



The worst performing model was model 2.5, which used a lag of 10 (with Box Cox transformed and normalized data):



In general, our finding was for all of our XGBoost models, a greater lag value results in worse performance than using a smaller lage value(keeping other variables constant).

The performance of models using this approach are summarized below:

	XGBoost Model Performances with Lagged Values Features and No Differencing							
	Model 2	**Model 2.1**	**Model 2.3***	**Model 2.5**	**Model. 2.6**	**Model. 2.7**	**Model 2.8**	**Model 2.9**
Features:	LagVal	LagVal	LagVal	LagVal	LagVal	LagVal	LagVal	LagVal
Preprocessing								
Lagged Value	**7**	7	7	**10**	**3**	**5**	**2**	**1**
Box Cox /Log (C	Yes	Yes	Yes	Yes	Yes	yes	Yes	Yes
Scaling/Normali	Minmax [0,1]	**Minmax[-1,1]**	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]
MSE (Test)	0.02119	0.022	0.01072	0.04252	0.0049	0.011	0.002751	**0.0009066**
SMAPE (Test)	22	23.53	17.649	30.882	12.370	18.46	**13.54425**	17.22109
RMSE	0.1455679	0.148324	0.1035374	0.2062038	0.07	0.1048809	0.05245	0.0301098

(XGBoost) Approach 3: Lagged Feature Values and Differencing

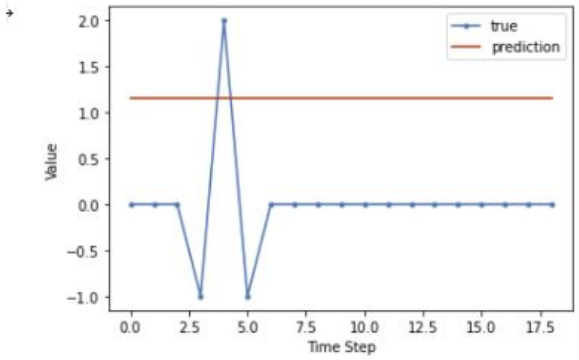
Lastly, we applied the differencing technique. As mentioned before, differencing is applied after Box Cox transformation and before scaling/normalization. We found that all the models that used differencing resulted in the worst performance, with the MSE errors ranging from 46 to 200.

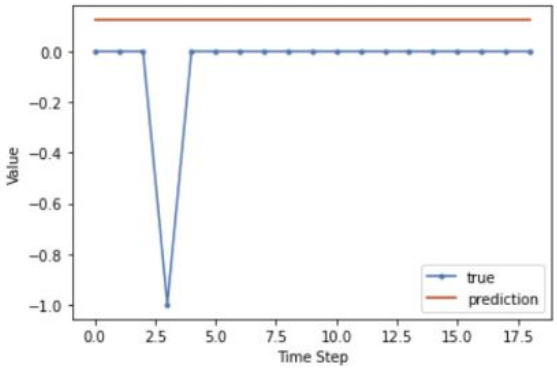
	XGBoost Models using Differencing Preprocessing Techniques						
	Model 3	**Model 3.1**	**Model 3.5**	**Model 3.6**	**Model 3.7**	**Model 3.8**	**Model 3.9**
Features:	LagVal	LagVal	LagVal	LagVal	LagVal	LagVal	LagVal
Preprocessing							
Lagged Value	3	3	3	3	1	1	1
Box Cox /Log	None	None	Yes	Yes	Yes	Yes	Yes
Differencing Le	3	1	1	2	2	3	3
Scaling/Normali	None	None	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]	Minmax [0,1]	StandardScaler
MSE (Test)	192.291	200	68.064	132.445	154.858	**46.12**	174.962
SMAPE (Test)	1.6506	0.08078	0.0883	0.35	0.425	**0.082**	2.047
RMSE	13.8669168	14.1421356	8.2500909	11.5084751	12.4441954	**6.7911707**	13.2273202

As indicated by the table above, increasing the level of differencing (with other variables constants) seemed to actually increase *mean squared error* as shown by Model 3.5 (level one differenced data) and 3.6 (level two differenced data).

The worst performances were given by Model 3 and model 3.1, with MSE of 192.3 and 200, respectively. Both of these models only used differencing as a preprocessing technique (no Box Cox or normalization/scaling). Model 3 used a third level differenced data, whereas Model 3.1 used first-level differenced data. These results suggest that differencing alone is NOT enough to generate accurate time series predictions, even with using lagged value features, highlighting the importance of normalization, standardization and Box Cox transformation.

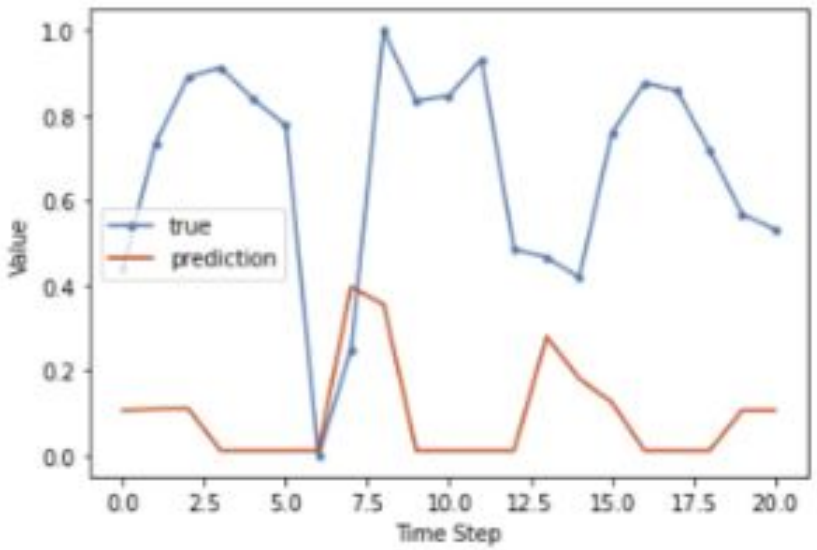
Model 3 and Model 3.1 are given below (in that order):



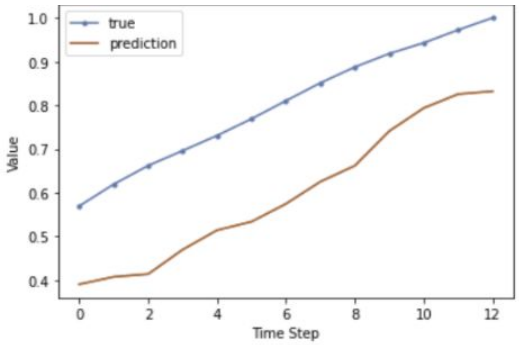
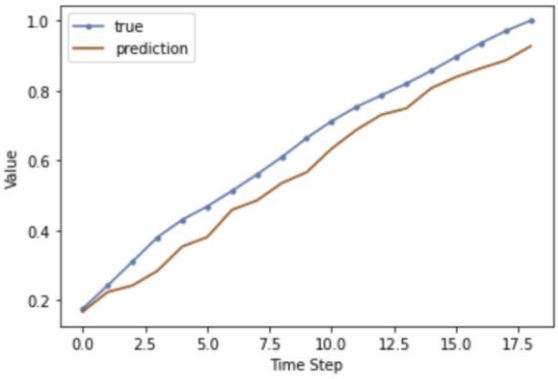
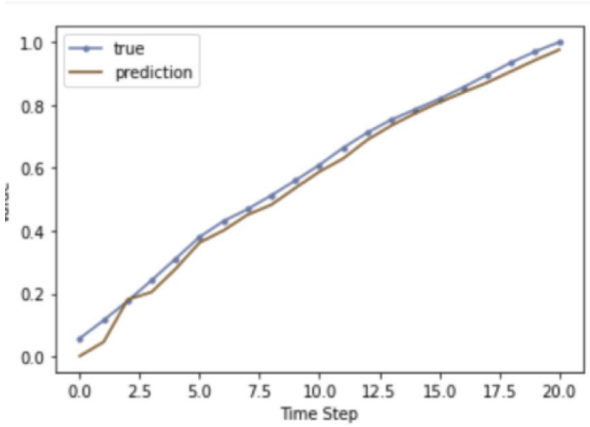
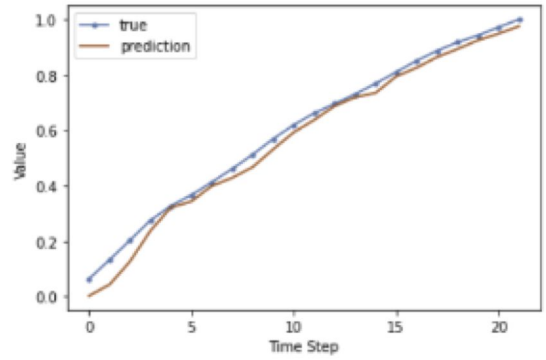


The best performance with the differencing technique was Model 3.8, which used lagged value of 1, Box Cox transformed,normalized, and third- level differenced data.

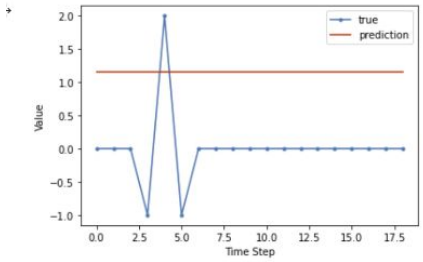
```
[83] plt.plot(scaled_test_y, marker='.', label="true"
     plt.plot(scaled_pred.reshape(-1,1), 'r', label="
     plt.ylabel('Value')
     plt.xlabel('Time Step')
     plt.legend()
     plt.show();
```



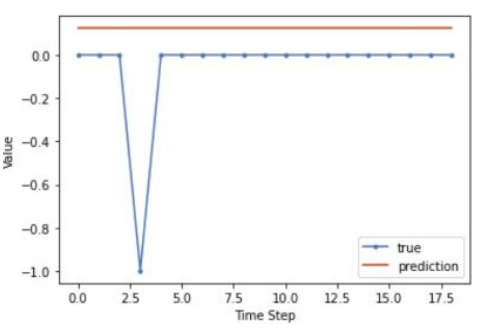
Remaining prediction plots are included here:

<table>
<tr><th colspan="2">Selected XGBoost Prediction Plot</th></tr>
<tr><td>Model 2.5
 
</td><td>Model 2.6
 
</td></tr>
<tr><td>Model 2.8
 
</td><td>Model 2.9
 
</td></tr>
</table>

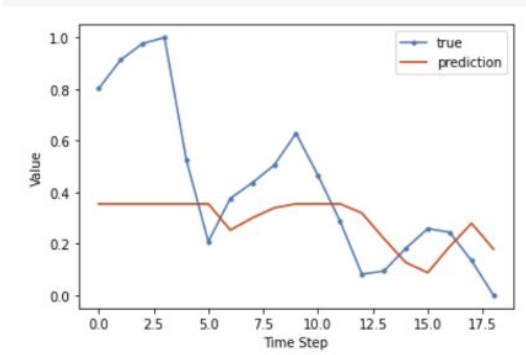
Model 3



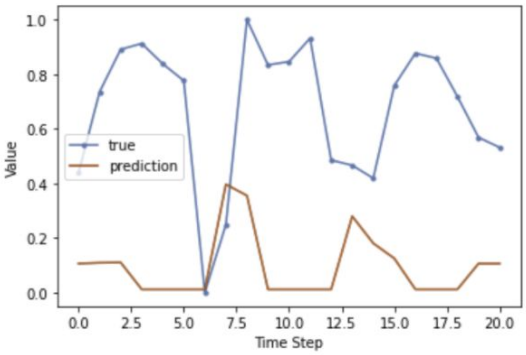
Model 3.1



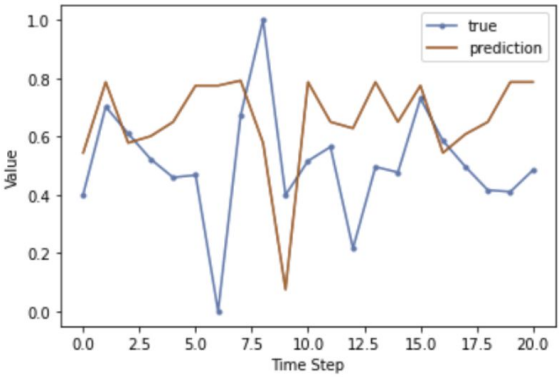
Model 3.5



Model 3.8



Model 3.9



IV. LSTM Network

As mentioned before, we were *not* able to achieve desired performance by using LSTMs despite multiple techniques, such as BatchNormalization and adding more neurons and layers. By using the model included in the "Code/Implementation" section. Once, we found the best model, we ran further experiments to examine the impact of differencing and different lagged values on performance of LSTMs on univariate time series forecasting.

The results of our experiments for LSTMs are included below:

	LSTM Performance of Different Lag Values (No Differencing)				LSTM Performance of Different Lag Values (with Differencing)			
Trial	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Features:								
Preprocessing								
Lagged Value	3	3	5	7	3	5	7	10
Box Cox /Log	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Differencing	No	No	No	No	Yes	Yes	Yes	Yes
Scaling/Normali	MinMax(-1,1)	MinMax(0,1)	MinMax(0,1)	MinMax(0,1)	MinMax(0,1)	MinMax(0,1)	MinMax(0,1)	MinMax(0,1)
MSE (Test)	0.161956	0.043926	0.023220	0.044450	0.030001	0.010664	0.009857	**0.008716**
SMAPE (Test)	917.383506	448.378769	283.922014	359.388216	363.644981	187.775853	159.572204	**121.370556**
RMSE	0.026230	0.001929	0.000539	0.001976	0.000900	0.000114	0.000097	**0.000076**

We found that for LSTMs, differencing resulted in greater performance! This was very interesting to our team because we did find such performance improvements with other models such as XGBoost and SVMs. For example by comparing trials 2, 3, 4 with their corresponding trials 5, 6, 7 (respectively), the impact of differencing is highlighted as trials 5 ,6, 7 use the same model, parameters, and data as trials 2, 3, 4 except with the additional step of differencing performed after Box Cox transformation.

Moreover, another interesting finding was a *general* positive impact of greater lagged values on increased performance! This can be observed by comparing trials 2 to 3: as the number of lagged values increase from 3 to 5, the SMAPE and MSE decrease. Similarly from trials 5 to

7, as lagged values increased from 3 to 7 with iteration of 2, there is a clear downward trend in SMAPE and MSE.

Compare Output Against Hypothesis

Our hypothesis had 2 components: COVID-19 case increases and model performance. We hypothesized that as this research was conducted, the United States would continue to see an increase of COVID-19 cases. This hypothesis was backed by various facts and options in the Hypothesis chapter. This hypothesis has unfortunately shown to be true. Additionally, our initial prediction for model performance was as follows (from best to worse): LSTM, SVM, XGBoost, and Linear Regression. As stated in the hypothesis chapter, this evaluation would be performed using various error metrics such as MSE, SMAPE, and RMSE. With all 3 metrics, we found that our hypothesis was very inaccurate. We predicted that linear regression would provide the worst prediction results, which is indeed the case for the raw dataset. However, with the use of preprocessing techniques that are essential to make time-series data stationary before being tested in models, linear regression ended up producing the lowest errors of all 4 models. We discuss this surprising result further in the Discussion. This reflects the importance of preprocessing techniques as well as the power of simple models and the importance to test such models before jumping to much more complex models. There is no need to overcomplicate a problem if it is not necessary for the data. Additionally, we hypothesized that LSTM would perform the best for predicting COVID-19 cases but actually had the worst results of all 4 tested models. We believe this can be attributed to the size of our dataset. Upon further research, we discovered that for successful LSTM models, a large training dataset is required. We did

hypothesize that SVR would perform 2nd best, which was found to be the case. Additionally, we grew a greater appreciation for SVR for its overall prediction accuracy. Even when predicting on the raw dataset before preprocessing, SVR was able to capture the data well with the RBF kernel then after transformations, the linear kernel predicted the values even more accurately.

Abnormal Case Explanation

- **Differencing did not always improve performance.** Usually, differencing is very important to remove trends in the data by detecting autocorrelation and making the data stationary. However, in this study, differencing only slightly improved the performance for LSTM, but was not the case for any other tested model in this study for the use of trend differencing. For example, when tested with XGBoost, adding differencing **significantly** decreased the performance.
- **SVR had surprisingly accurate predictions on untransformed data, in comparison to other models.** For the raw/untransformed data, SVR's RBF kernel fits the COVID-19 case data very closely. After preprocessing techniques of lagged values, power transformation, and normalization, the linear kernel fit the COVID-19 cases even more accurately, to our surprise as we had seen the power of the RBF kernel in various testings and additional research. This accuracy on the untransformed data can be attributed to the **small dataset** or to poor hyperparameter tuning.
- **Not all models improved with greater lagged values.** With lagged values, it is generally found that the greater the lagged value is, the better the performance is. This is attributed to the fact that more data is being captured to make the prediction. However, in

this study, that was not always found to be the case. For linear regression, SVR, and LSTM, the greater number of lagged values gave better performance. In contrast, **smaller** lag values gave better performances in XGBoost.

Discussion

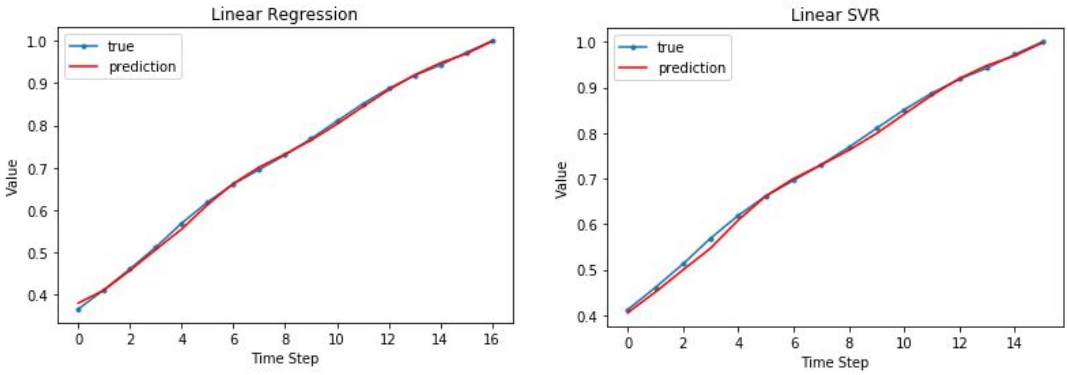
We believe it is generally best to start with models of lower complexity then attempt models of more complexity or add complexity to existing models. As machine learning can already be extremely difficult to explain and interpret results or successes of models, it is beneficial to begin each project with simple models then based on model performance and particular setbacks, test more complex and appropriate models. For example, we began this project with a plan of testing 4 vastly different models as far as complexity--from linear regression to neural networks with LSTM. In our hypothesis, we believed that LSTM would perform the best because of its complexity and power and that linear regression would perform the worst due to its simplicity. However, because of the preprocessing techniques used to stabilize the time-series data in this study with power transform, as well as converting our time-series problem into a supervised learning problem using lagged values, linear regression computed this nonlinear prediction with great performance. Conversely, with a model as complex as LSTM, these tests resulted in the highest errors of all models tested in this study, most likely due to lack of data. However, SVR demonstrated surprisingly accurate results given the limited dataset used in this study, outperforming both XGBoost and LSTM. This shows that no model can solve all problems, no matter how simple or complex. With model selection for machine learning applications to address various studies, it is essential to test various models rather than assuming to know which models will or will not perform best.

In this project, we found that feature engineering as well as testing and selection of preprocessing techniques were some of the most important steps for model performance. In particular, using previous time-step values (lagged values) to predict subsequent time steps had the most significant impact on performance. This approach was a vast improvement over using the raw time-series data by predicting confirmed cases using the date features. Another approach with preprocessing techniques that had incredible contributions to the success of our models was performing power transformation of the data with BoxCox. Normalization was also helpful to more clearly understand the successes and fallbacks of the models by scaling down the range of cases from millions to 0 through 1. In general, using the default MinMaxScaler values of 0 to 1 for range normalization had better performance than setting a manual range of -1 to 1 or performing standardization with StandardScaler. However, our research has shown that StandardScaler generally performs very well which is why we emphasize the importance of testing various preprocessing techniques to verify which have a positive impact on a given dataset, as not all techniques contribute the same performance across all datasets.

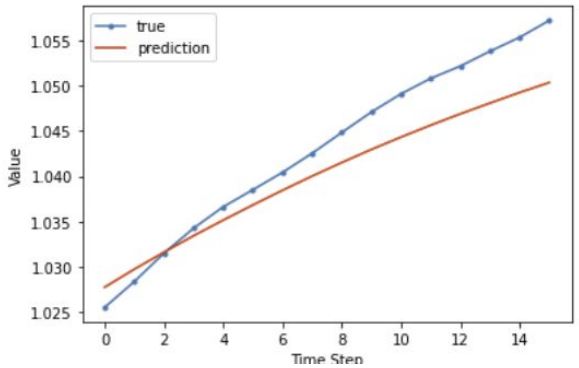
Linear SVR was second best to linear regression prediction results in this study. We hypothesize this was the case because with the power transform of our time-series data in order to make the data stationary, the distribution of COVID-19 cases over time went from approximately exponential to approximately linear after an approximate log transform of the data in order to remove its trends. The resulting nearly linear data worked well with linear regression after transforming the independent variable from date to sliding windows. We believe that if our data was highly nonlinear, SVR (RBF) would have captured our data better and would have been

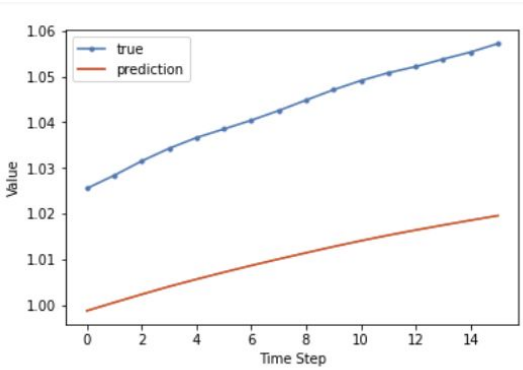
the best performing model in this study, as it tested very well with numerous variations of model studies conducted in this research.

Linear Regression examines the linear relationship between independent variable(s) X and dependent variable Y and fits its prediction line by minimizing the sum of squared prediction error. Support Vector Regression, on the other hand, uses the same classification algorithm of SVM but is applied to the prediction of continuous data rather than categorical data. A strength of SVR is its ability to deal with non-linear data where simple linear regression cannot capture such complexities. The best accuracy using preprocessing data transformations to make the time series data stationary for SVR was found to be a **lagged value of 7 days,** performing a power transform, and normalizing the data. These identical transformations also yielded the second most accurate results for the Linear Regression of all tests performed for this model. However MSE and SMAPE results between these two models reveal the strength of Linear Regression in this study, with an MSE value of 6.44e-05 whereas the same test under SVR yielded an MSE of 9.11e-05. Similarly, Linear Regression also outperformed SVR for this test with an SMAPE of 0.987 and Linear SVR resulting in 1.145.



For XGBoosting, we found generally more trees with less depth gave the best performance. The main takeaways from the testing of this model was the importance and benefit of applying grid search and testing different methods for hyperparameter tuning for the given dataset as there is not an ideal value for each parameter that will work with all datasets. The tuning of these hyperparameters have been found in this study to make a major impact on model performance. Additionally, as LSTM had the worst and very unstable model performance in this study, this finding was examined further. For example, by not setting a random seed value in the model run, which then does not guarantee the same data to be selected for training, the same model parameters produced vastly different results, most likely due to the small training dataset in this study, seen below. LSTM also took an extremely long time to tune in attempts to find the optimal number of layers.





VII. CONCLUSIONS AND RECOMMENDATIONS

Summary and Conclusions

The hypothesis of this study was the performance rankings of various machine learning models in attempts to predict COVID-19 confirmed cases in the United States using time-series data for the virus. In testing models of linear regression, SVR, XGBoost, and LSTM, it was hypothesized that LSTM and XGBoost would result in the lowest errors, followed by SVR and linear regression. It was found through this project that there are various model features and requirements to consider before making such assumptions about general model performance. For example, we assumed that LSTM would provide strong results in this study due to the model's complexity whereas linear regression would show poor performance due to its simplicity. However, after researching essential preprocessing techniques for time-series data, the COVID-19 confirmed cases were power transformed to a semi-linear trend plot and converted to a supervised machine learning problem through lagged values. These two critical steps improved the results of all models in this study but linear regression especially benefited from these steps as the power transform result was easier to fit and the lagged values predictor enabled linear regression to make a nonlinear prediction plot. Linear regression resulted in great performance in this study which is why we emphasize the importance and benefit of testing simpler models first and only adding complexity if the data requires it. Additionally, as the success of linear regression was only found after various preprocessing techniques, restructuring of the data, and feature engineering, we also found to not underestimate the importance of these steps, as they made vast improvements to our models.

Conversely, we believe LSTM had low performance in this study due to the size of the training dataset. After further research of the model and general deep learning, it requires a large amount of data to perform well whereas we cannot provide that at the time of this study since the virus is still fairly new. Additionally, it was found to be extremely time consuming to select and test different numbers of layers and neurons for this model. We believe that years from now when this pandemic has recovered and there will be a vast amount of data on the event, LSTM may provide excellent performance on this subject. We also hypothesized that for the course of this study, COVID-19 cases would continue to rise in the United States which unfortunately has shown to be true.

Recommendations for Future Studies

We believe the main contribution of this study is the testing and results provided of various machine learning models to predict COVID-19 cases. We have addressed the strengths and weaknesses of each model for this task as well as explaining the essential preprocessing methods that helped improve our results. LSTM was a major limitation in this study due to our lack of COVID-19 time-series data as it is a new and ongoing pandemic. Additionally, this complex model requires a lot of time and understanding to tune the number of layers and neurons. With more time devoted to this model, this may be able to be improved with tuning, even without having to wait for more time-series data on the virus. However, we believe the remaining three models of this study were tested extensively and have been captured in this study with clear and helpful results for future COVID-19 studies. The work of this study can be furthered by predicting different COVID-19 related time-series features such as number of

deaths over time or number of recoveries individuals over time. The models used in this study can also be extended or additional models can be tested to account for the healthcare capacity of each state as well as social restrictions in order to increase accuracy or to forecast cases, deaths, and recovery rates for future dates.

VIII. BIBLIOGRAPHY

Abbas, K., Mikler, A. R., Gatti, R., Kaja Abbas University of North Texas, University of North Texas, Armin R. Mikler University of North Texas, … Alma Mater Studiorum. (2005, March 1). Temporal Analysis of Infectious Diseases: Influenza. Retrieved from https://dl.acm.org/doi/10.1145/1066677.1066740

Ahmed, Nesreen & Atiya, Amir & Gayar, Neamat & El-Shishiny, Hisham. (2010). An Empirical Comparison of Machine Learning Models for Time Series Forecasting. Econometric Reviews. 29. 594-621. 10.1080/07474938.2010.481556.

Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). Association for Computing Machinery, New York, NY, USA, 785–794. DOI:https://doi.org/10.1145/2939672.2939785

Eiji Aramaki, Sachiko Maskawa, and Mizuki Morita. 2011. Twitter catches the flu: detecting influenza epidemics using Twitter. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '11). Association for Computational Linguistics, USA, 1568–1576. https://dl.acm.org/doi/10.5555/2145432.2145600

Géron, A. (2017). Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems O'Reilly Media Paperback .

Chniti, G., Bakir H., and Zaher, H.. 2017. E-commerce Time Series Forecasting using LSTM Neural Network and Support Vector Regression. In Proceedings of the International Conference on Big Data and Internet of Thing. Association for Computing Machinery, New York, NY, USA, 80–84. https://doi.org/10.1145/3175684.3175695

Fernando, Lasantha and Lokanathan, Sriganesh and Perera, Amal and Ghouse, Azhar and Tissera, Hasitha, Improving Disease Outbreak Forecasting Models for Efficient Targeting of Public Health Resources (November 16, 2017). Available at SSRN: https://ssrn.com/abstract=3072086 or http://dx.doi.org/10.2139/ssrn.3072086

Han, L., Zhang, Y., Zhang, T., Bosch, Department of Statistics, Yu Zhang Department of Computer Science and Engineering, … Ibm. (2016, August 1). Generalized Hierarchical Sparse Model for Arbitrary-Order Interactive Antigenic Sites Identification in Flu Virus Data. Retrieved from https://dl.acm.org/doi/10.1145/2939672.2939786

Hyndman, R.J., & Athanasopoulos, G. (2018) Forecasting: principles and practice, 2nd edition, OTexts: Melbourne, Australia. OTexts.com/fpp2.

Hongzhan NIE, Guohaui LIU, Xiaoman LIU, Yong WANG, School of Electrical Engineering, Northeast China Grid Company Limited Changchun Extrahigh Voltage Bureau, Changchun, China (2012) Hybrid of ARIMA and SVMs for Short-Term Load Forecasting. Retrieved from https://reader.elsevier.com/reader/sd/pii/S1876610212002391?token=283992B333225AC0C774F4A7384FC457E7833D56846F76E256BBB4C675CEE77D98C246594EF6CCCD0C746ECD99DE8468

Nelson M, Hill T, Remus B, O'Connor M. Can neural networks applied to time series forecasting learn seasonal patterns: an empirical investigation. System Sciences, 1994 Proceedings of the Twenty-Sev- enth Hawaii International Conference on. 1994; 3:649–655. https://doi.org/10.1109/HICSS.1994. 323316

Novel Corona Virus 2019 Dataset. (n.d.). Retrieved from https://www.kaggle.com/sudalairajkumar/novel-corona-virus-2019-dataset

Ranger, N. (2020, April 9). COVID-19 State Data. Retrieved from https://www.kaggle.com/nightranger77/covid19-state-data

Ristanoski, Goce & Liu, Wei & Bailey, James. (2013). Time Series Forecasting using Distribution Enhanced Linear Regression. 10.13140/2.1.3300.9921.

Samsudin, R.,Shabri and P. Saad, 2010. A Comparison of Time Series Forecasting using Support Vector Machine and Artificial Neural Network Model. Journal of Applied Sciences, 10: 950-958.

Sibanda, W. Pretorius, P. Artificial Neural Networks - A review of applications of Neural Networks in the Modeling of HIV Epidemic. International Journal of Computer Applications. 2012; 0975-8887.

Zhang GP, Qi M. Neural network forecasting for seasonal and trend time series. European Journal of Operational Research. 2005; 160(2):501–514. https://doi.org/10.1016/j.ejor.2003.08.037

VII. APPENDIX

Input/Output Generation

Code for the Best Machine Learning Models:

Time Series Forecasting: Covid19 Cases

This is the demo consists of the *best* architectures for each of the following models:

1. Linear Regression
2. Linear Support Vector Regression
3. Nonlinear Support Vector Regression
4. XGBoost
5. LSTM Network

Note that for the purposes of the demo GridSearch step using cross validation is excluded from this notebook.

```
[ ] import requests
    import pandas as pd
    from matplotlib import pyplot
    from matplotlib import pyplot as plt
    from sklearn.preprocessing import MinMaxScaler,StandardScaler
    from pandas.plotting import autocorrelation_plot
    from scipy.stats import boxcox
    import numpy as np
    from  sklearn.metrics import mean_squared_error, accuracy_score
    from sklearn.model_selection import KFold, train_test_split, GridSearchCV,cross_val_score,LeaveOneOut

    import tensorflow as tf
    from tensorflow import keras
    from keras.models import Sequential
    from keras.layers import LSTM,Dense, Dropout ,BatchNormalization

    import seaborn as seabornInstance
    from sklearn.svm import SVR
    from sklearn.linear_model import LinearRegression
    from sklearn.svm import LinearSVR
    from xgboost import XGBRegressor


    from sklearn import metrics
    %matplotlib inline
```

```
def create_dataset(X, y, time_steps=1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)

def smape(A, F):
    return 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))

def invert_boxcox(value, lam):
  # log case
  if lam == 0:
    return np.exp(value)
  # all other cases
  return  np.exp( np.log(lam * value + 1) / lam)
```

```
states_url = "https://covidtracking.com/api/states/daily"
us_url = "https://covidtracking.com/api/us/daily"
req = requests.get(us_url)
us_df = pd.DataFrame(req.json())
us_df= us_df.sort_values(by='date', ascending=True)
```

```
us_df['date'] = pd.to_datetime(us_df['date'], format='%Y%m%d')

us_df['month']= pd.DatetimeIndex(us_df["date"]).month
us_df['year']= pd.DatetimeIndex(us_df["date"]).year
us_df['day']= pd.DatetimeIndex(us_df["date"]).day


us_Y = us_df["positive"]

#Power transform time series data
us_box_total, lam = boxcox(us_Y)
Y = us_box_total

#Split transformed time series data
cut_off = int(len(Y)*.80)
train_labels = Y[0:cut_off]
test_labels  = Y[cut_off:]
```

```
#Scale/Normalize Input
minmax_scaler = MinMaxScaler()

scaled_train_labels = minmax_scaler.fit_transform(train_labels.reshape(-1, 1))
scaled_test_labels = minmax_scaler.fit_transform(test_labels.reshape(-1, 1))
```

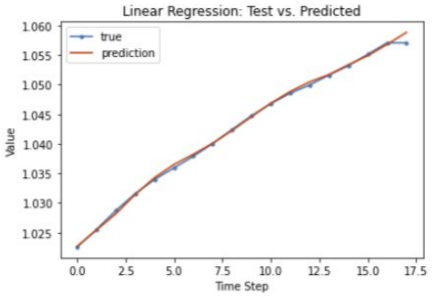
Linear Regression

Model 1.6: Lag=6, Auto BoxCox Power Transform, Auto Normalization

```
[ ] #Generate Lagged Data[Sliding Window Technqiue]
    timesteps = 6

    scaled_train_data = pd.Series(scaled_train_labels.reshape(1,-1)[0])
    scaled_test_data = pd.Series(scaled_test_labels.reshape(1,-1)[0])

    scaled_train_x, scaled_train_y = create_dataset(scaled_train_data , scaled_train_data , timesteps)
    scaled_test_x, scaled_test_y = create_dataset(scaled_test_data , scaled_test_data , timesteps)
```

```
[ ] regressor = LinearRegression()

    #Train the model
    regressor.fit(scaled_train_x, scaled_train_y)

    #Make predictions on scaled test data
    y_pred = regressor.predict(scaled_test_x)

    preds = pd.DataFrame({'Actual': scaled_test_y.flatten(), 'Predicted': y_pred.flatten()})
```

```
[ ] plt.plot(scaled_test_y, marker='.', label="true")
    plt.plot(y_pred, 'r', label="prediction")
    plt.ylabel('Value')
    plt.xlabel('Time Step')
    plt.title("Linear Regression: Test vs. Predicted")
    plt.legend()
    plt.show();
```




```
] print("Mean Squared Error")
  print(mean_squared_error(scaled_test_y, y_pred.reshape(-1,1)))
  print()
  print("Symmetric Mean Absolute Percentage Error")
  print(smape(scaled_test_y, y_pred))
```

```
Mean Squared Error
3.96150697347151e-05

Symmetric Mean Absolute Percentage Error
0.8435423206484249
```

Linear SVR

Model 1: Lag=7, Auto BoxCox Power Transform, Auto Normalization

```
timesteps = 7

scaled_train_data = pd.Series(scaled_train_labels.reshape(1,-1)[0])
scaled_test_data = pd.Series(scaled_test_labels.reshape(1,-1)[0])

scaled_train_x, scaled_train_y = create_dataset(scaled_train_data , scaled_train_data , timesteps)
scaled_test_x, scaled_test_y = create_dataset(scaled_test_data , scaled_test_data , timesteps)
```

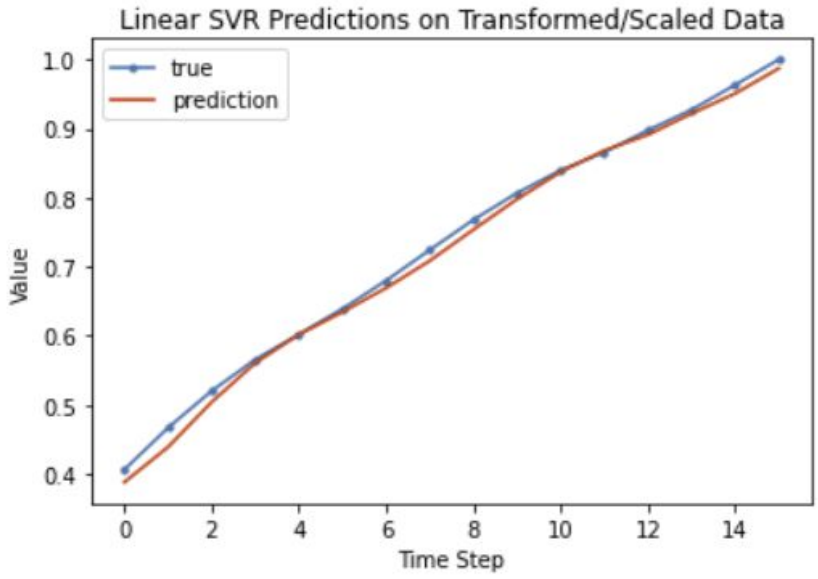
```
[ ]  gsc = GridSearchCV(
        estimator = LinearSVR(),
        param_grid = {
            'C': [0.1, 1, 100, 1000],
            'epsilon': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10],
        },
        cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs=-1)


    #gsc.fit(scaled_train_x, scaled_train_y )
    #best_params = gsc.best_params_

    print("best_params")
    print("{'C': 1000, 'epsilon': 0.001}")
    lin_svr = LinearSVR(C = 1000, epsilon =0.001)

    lin_svr.fit(X= scaled_train_x, y= scaled_train_y);
```

```
best_params
{'C': 1000, 'epsilon': 0.001}
```

```
pred = lin_svr.predict(scaled_test_x)
```

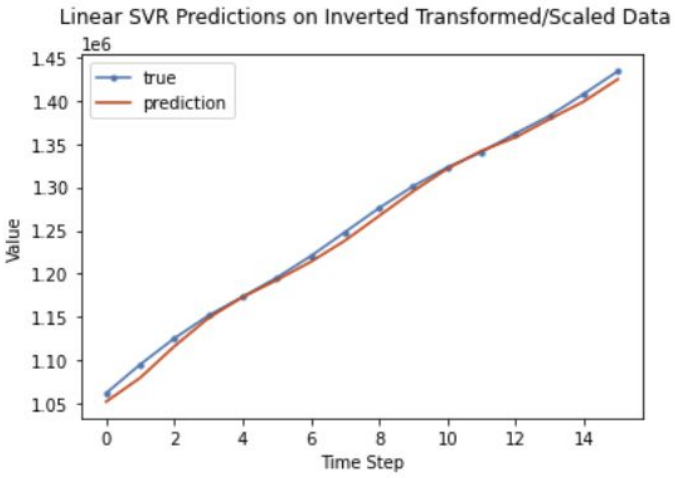
```
print("Mean Squared Error")
print(mean_squared_error(scaled_test_y, pred.reshape(-1,1)))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(scaled_test_y, pred))
print()

plt.plot(scaled_test_y, marker='.', label="true")
plt.plot(pred, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title("Linear SVR Predictions on Transformed/Scaled Data")
plt.show();
```

```
Mean Squared Error
0.0001538944068338607

Symmetric Mean Absolute Percentage Error
1.6737467044961793
```



```
yhat = minmax_scaler.inverse_transform(np.array(pred).reshape(-1, 1) )
ytest = minmax_scaler.inverse_transform(np.array(scaled_test_y).reshape(-1, 1) )


inv_yhat  = [invert_boxcox(x, lam) for x in yhat ]
inv_ytest  = [invert_boxcox(x, lam) for x in ytest ]

print("Mean Squared Error")
print(mean_squared_error(np.array(inv_ytest), np.array(inv_yhat )))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(np.array(inv_ytest), np.array(inv_yhat)))
print()
plt.plot(inv_ytest, marker='.', label="true")
plt.plot(inv_yhat, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" Linear SVR Predictions on Inverted Transformed/Scaled Data" , pad =20);
plt.show();
```

```
Mean Squared Error
55754185.22621552

Symmetric Mean Absolute Percentage Error
0.5169520062375679
```



Nonlinear SVR

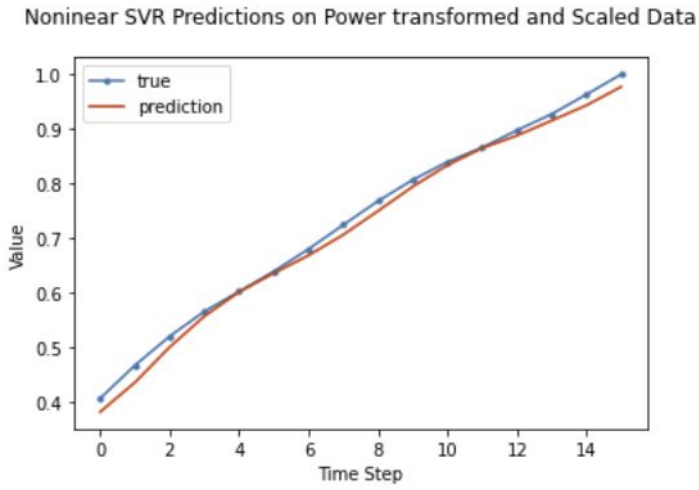
```
[ ]  gsc = GridSearchCV(
             estimator = SVR('rbf'),
             param_grid={
                 'C': [0.1, 1, 100, 1000],
                 'epsilon': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10],
                 'gamma': [0.0001, 0.001, 0.005, 0.1, 1, 3, 5]
             },
             cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs=-1)


     ##grid_result = gsc.fit(X = scaled_train_x, y= scaled_train_y)
     #best_params = grid_result.best_params_
     print("best_params")
     print("{'C': 1000, 'epsilon': 0.0005, 'gamma': 0.005}")
     nonlin_svr = SVR(kernel='rbf', C = 1000, epsilon = 0.0005, gamma= 0.005,
                      coef0=0.1, shrinking = True,
                      tol=0.001, cache_size=200, verbose = False, max_iter=-1)
```

```
best_params
{'C': 1000, 'epsilon': 0.0005, 'gamma': 0.005}
```

```
[ ]  nonlin_svr.fit(X = scaled_train_x, y = scaled_train_y)
     nonlin_pred = nonlin_svr.predict(scaled_test_x)
```

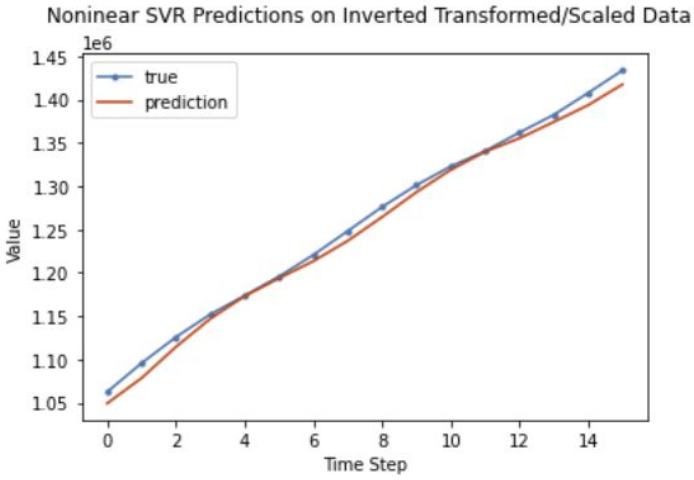
```
print("Mean Squared Error")
print(mean_squared_error(scaled_test_y, nonlin_pred.reshape(-1,1)))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(scaled_test_y, nonlin_pred))
print()

plt.plot(scaled_test_y, marker='.', label="true")
plt.plot(nonlin_pred, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title("Noninear SVR Predictions on Power transformed and Scaled Data", pad= 20)
plt.show();
```

```
Mean Squared Error
0.0002624914331316728

Symmetric Mean Absolute Percentage Error
2.194027584713634
```



```
#Invert power transform and scaled data to evaluate model performance
yhat = minmax_scaler.inverse_transform(np.array(nonlin_pred).reshape(-1, 1) )
ytest = minmax_scaler.inverse_transform(np.array(scaled_test_y).reshape(-1, 1) )

inv_yhat  = [invert_boxcox(x, lam) for x in yhat ]
inv_ytest  = [invert_boxcox(x, lam) for x in ytest ]

print("Mean Squared Error")
print(mean_squared_error(np.array(inv_ytest), np.array(inv_yhat )))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(np.array(inv_ytest), np.array(inv_yhat)))
print()

plt.plot(inv_ytest, marker='.', label="true")
plt.plot(inv_yhat, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" Noninear SVR Predictions on Inverted Transformed/Scaled Data", pad =20);
plt.show();
```

```
Mean Squared Error
99966820.39697167

Symmetric Mean Absolute Percentage Error
0.6953673903669847
```



XGBoost

```
[ ] timesteps = 1
    print("timesteps")
    print(timesteps)
```

```
[ ] scaled_train_data = pd.Series(scaled_train_labels.reshape(1,-1)[0])
    scaled_test_data = pd.Series(scaled_test_labels.reshape(1,-1)[0])

    scaled_train_x, scaled_train_y = create_dataset(scaled_train_data , scaled_train_data , timesteps)
    scaled_test_x, scaled_test_y = create_dataset(scaled_test_data , scaled_test_data , timesteps)
```

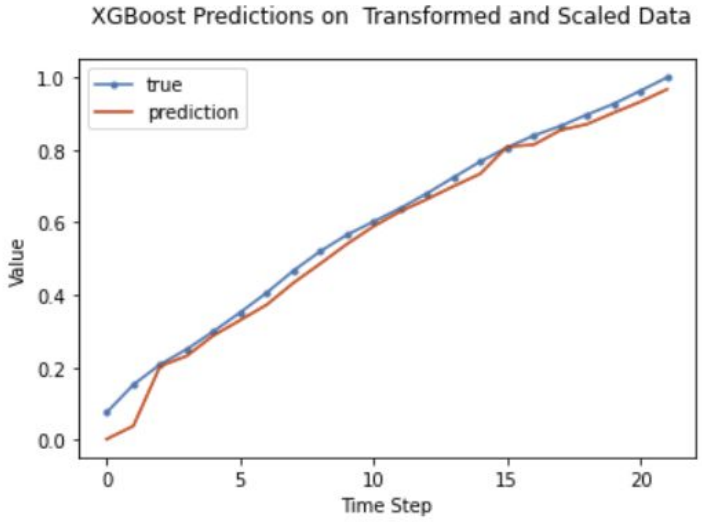
```
timesteps
1
```

```
[ ] print("Parameter optimization")
    xgb_model = XGBRegressor(objective = 'reg:squarederror')
    clf = GridSearchCV(xgb_model,
                       {'max_depth': [2,4,6,10,20],
                        'learning_rate': [.0001,.001,.001,.10],
                        'min_child_weight':[1,4,5,6],
                        'subsample':[0.3, .5,0.8,1],
                         'n_estimators': [50,100,200,1000,1500]})
    #clf.fit(scaled_train_x, scaled_train_y)
    print(clf.best_score_)
    print(clf.best_params_)
```

```
[ ] model2_8 = XGBRegressor( learning_rate=0.1, max_depth = 4,  min_child_weight= 1 , n_estimators = 200 ,subsample=0.3)
    history = model2_8.fit(scaled_train_x, scaled_train_y)
```

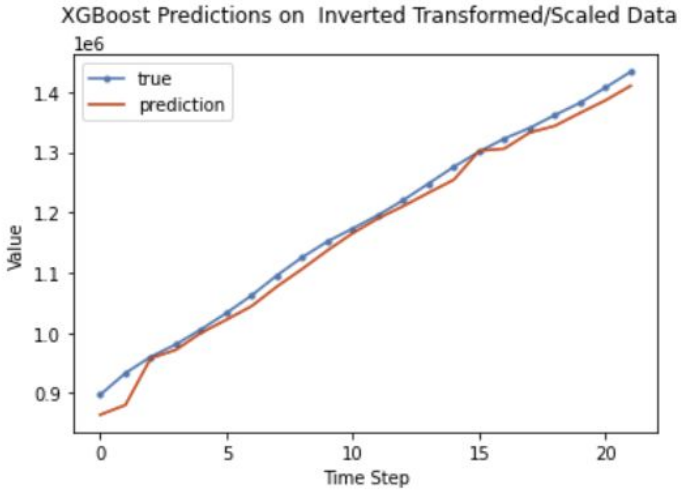
```
scaled_pred = model2_8.predict(scaled_test_x)

print("Mean Squared Error")
print(mean_squared_error(scaled_test_y, scaled_pred ))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(scaled_test_y, scaled_pred))
print()

plt.plot(scaled_test_y, marker='.', label="true")
plt.plot(scaled_pred.reshape(-1,1), 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()

plt.title(" XGBoost Predictions on  Transformed and Scaled Data", pad =20 );
plt.show();
```

```
Mean Squared Error
0.0013465201570721365

Symmetric Mean Absolute Percentage Error
17.562644289675827
```



```
print("XGBoost Performance on Inverted Transformed and Inverted Scaled Data ")
print()
yhat = minmax_scaler.inverse_transform(np.array(scaled_pred).reshape(-1, 1) )
ytest = minmax_scaler.inverse_transform(np.array(scaled_test_y).reshape(-1, 1) )

inv_yhat  = [invert_boxcox(x, lam) for x in yhat ]
inv_ytest  = [invert_boxcox(x, lam) for x in ytest ]

print("Mean Squared Error")
print(mean_squared_error(np.array(inv_ytest), np.array(inv_yhat )))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(np.array(inv_ytest), np.array(inv_yhat)))
print()


plt.plot(inv_ytest, marker='.', label="true")
plt.plot(inv_yhat, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" XGBoost Predictions on  Inverted Transformed/Scaled Data", pad =20 );

plt.show();
```

```
XGBoost Performance on Inverted Transformed and Inverted Scaled Data

Mean Squared Error
373458536.97956306

Symmetric Mean Absolute Percentage Error
1.4254352056399753
```



LSTM

```
[ ] time_steps = 10

    # reshape to [samples, time_steps, n_features]

    X_train, y_train = create_dataset(scaled_train_data, scaled_train_data, time_steps)
    X_test, y_test = create_dataset(scaled_test_data, scaled_test_data , time_steps)

    print(X_train.shape, y_train.shape)
```

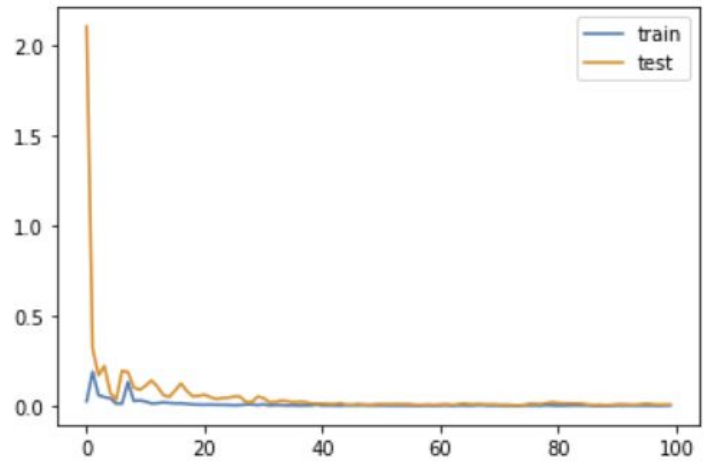
```
(82, 10) (82,)
```

```
[ ] X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
    X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
```

```
[ ] model = keras.Sequential()
    model.add(keras.layers.LSTM(50, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences = True))
    model.add(keras.layers.Dropout(0.1))
    model.add(keras.layers.LSTM(units = 50, return_sequences = True))
    model.add(keras.layers.Dropout(0.1))
    model.add(keras.layers.Dense(1))
    model.compile(loss='mean_squared_error', optimizer = keras.optimizers.Adam(0.1))
```

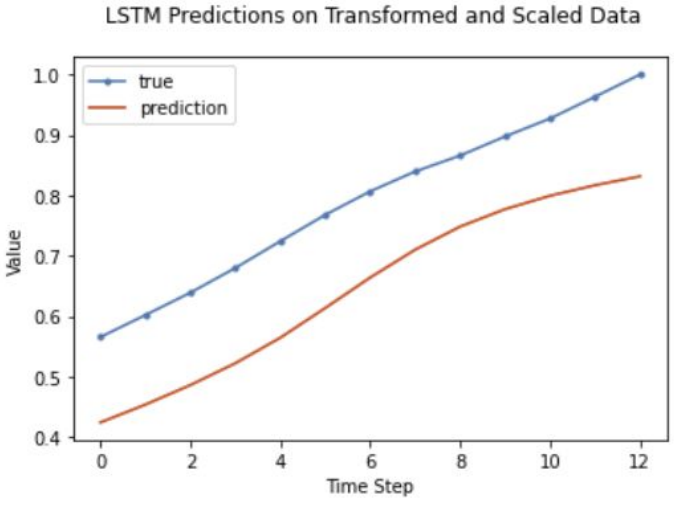
```
[ ] history = model.fit(
        X_train, y_train,
        epochs = 100,
        batch_size = 31,
        validation_split = 0.3,
        verbose = 1,
        shuffle = False
    )
```

```
#Diagnostic Line Plots for Loss and Validation loss of the data
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.legend();
```



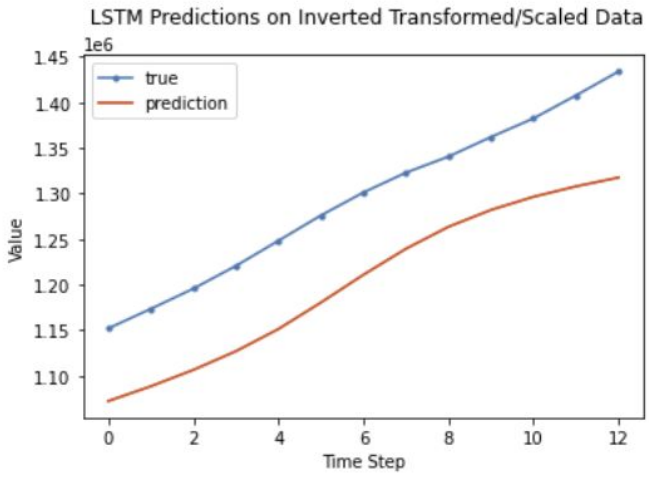
```
#Evaluation of LSTM
y_pred = model.predict(X_test)

def smape(A, F):
    return 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))

print('\n Evaluation on test data: MSE')
results = model.evaluate(X_test, y_test, batch_size = 128)
print(results)

print('\n Evaluation on test data: SMAPE')
print(smape(y_test, y_pred.reshape(-1,1)))
print()

plt.plot(y_test, marker='.', label="true")
plt.plot(y_pred.reshape(-1,1), 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" LSTM Predictions on Transformed and Scaled Data" , pad =20);

plt.show();
```

```
 Evaluation on test data: MSE
1/1 [==============================] - 0s 1ms/step - loss: 0.0208
0.02082975022494793

 Evaluation on test data: SMAPE
366.86302172509227
```



```
print("LSTM Performance on Inverted Transformed and Inverted Scaled Data ")
yhat = minmax_scaler.inverse_transform(np.array(y_pred).reshape(-1, 1) )
ytest = minmax_scaler.inverse_transform(np.array(y_test).reshape(-1, 1) )

inv_yhat  = [invert_boxcox(x, lam) for x in yhat ]
inv_ytest  = [invert_boxcox(x, lam) for x in ytest ]

print("Mean Squared Error")
print(mean_squared_error(np.array(inv_ytest), np.array(inv_yhat )))
print()
print("Symmetric Mean Absolute Percentage Error")
print(smape(np.array(inv_ytest), np.array(inv_yhat)))
print()

plt.plot(inv_ytest, marker='.', label="true")
plt.plot(inv_yhat, 'r', label="prediction")
plt.ylabel('Value')
plt.xlabel('Time Step')
plt.legend()
plt.title(" LSTM Predictions on Inverted Transformed/Scaled Data" , pad =20);

plt.show();
```

```
LSTM Performance on Inverted Transformed and Inverted Scaled Data
Mean Squared Error
8234023373.144158

Symmetric Mean Absolute Percentage Error
7.2395033335666685
```



Code for COVID 19 Investigation and Exploratory Data Analysis

```
#@title

%matplotlib inline
import math
from numpy.random import randn
import matplotlib.pyplot as plt
from numpy.random import seed
from numpy import mean, std
from scipy.stats import  pearsonr
import requests
import pandas as pd
import numpy as np
import altair as alt
from IPython.display import HTML
import matplotlib.pyplot as plt


#https://covidtracking.com/api for details on column names
states_url = "https://covidtracking.com/api/states/daily"
us_url = "https://covidtracking.com/api/us/daily"
```

```
us_req = requests.get(us_url)
us_df = pd.DataFrame(us_req.json())
us_df.sort_values(by=['date'], inplace = True)
us_df.head()
```

	date	states	positive	negative	pending	hospitalizedCurrently	hospitalizedCumulative	inIcuCurrently	inIcuCumulative
115	20200122	1	1	NaN	NaN	NaN	NaN	NaN	NaN
114	20200123	1	1	NaN	NaN	NaN	NaN	NaN	NaN
113	20200124	1	1	NaN	NaN	NaN	NaN	NaN	NaN
112	20200125	1	1	NaN	NaN	NaN	NaN	NaN	NaN
111	20200126	1	1	NaN	NaN	NaN	NaN	NaN	NaN

```
us_df["date"] = pd.to_datetime(us_df["date"],format='%Y%m%d')
states_df["date"] = pd.to_datetime(states_df["date"],format='%Y%m%d')
```

Covid 19 State-Level Data

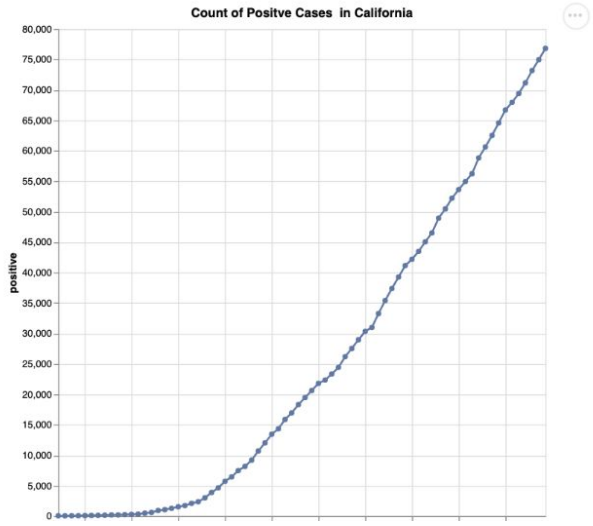
```
#@title
us_state_abbrev = {
    'Alabama': 'AL',
    'Alaska': 'AK',
    'American Samoa': 'AS',
    'Arizona': 'AZ',
    'Arkansas': 'AR',
    'California': 'CA',
    'Colorado': 'CO',
    'Connecticut': 'CT',
    'Delaware': 'DE',
    'District of Columbia': 'DC',
    'Florida': 'FL',
    'Georgia': 'GA',
    'Guam': 'GU',
    'Hawaii': 'HI',
    'Idaho': 'ID',
    'Illinois': 'IL',
    'Indiana': 'IN',
    'Iowa': 'IA',
    'Kansas': 'KS',
    'Kentucky': 'KY',
    'Louisiana': 'LA',
    'Maine': 'ME',
    'Maryland': 'MD',
    'Massachusetts': 'MA',
    'Michigan': 'MI',
    'Minnesota': 'MN',
    'Mississippi': 'MS',
    'Missouri': 'MO',
    'Montana': 'MT',
    'Nebraska': 'NE',
    'Nevada': 'NV',
    'New Hampshire': 'NH',
    'New Jersey': 'NJ',
    'New Mexico': 'NM',
    'New York': 'NY',
    'North Carolina': 'NC',
    'North Dakota': 'ND',
    'Northern Mariana Islands':'MP',
    'Ohio': 'OH',
    'Oklahoma': 'OK',
    'Oregon': 'OR',
    'Pennsylvania': 'PA',
    'Puerto Rico': 'PR',
    'Rhode Island': 'RI',
    'South Carolina': 'SC',
    'South Dakota': 'SD',
    'Tennessee': 'TN',
    'Texas': 'TX',
    'Utah': 'UT',
    'Vermont': 'VT',
    'Virgin Islands': 'VI',
    'Virginia': 'VA',
    'Washington': 'WA',
    'West Virginia': 'WV',
    'Wisconsin': 'WI',
    'Wyoming': 'WY'
}
states_df["state_name"] = states_df["state"].map(dict(map(reversed, us_state_abbrev.items())))
```

California

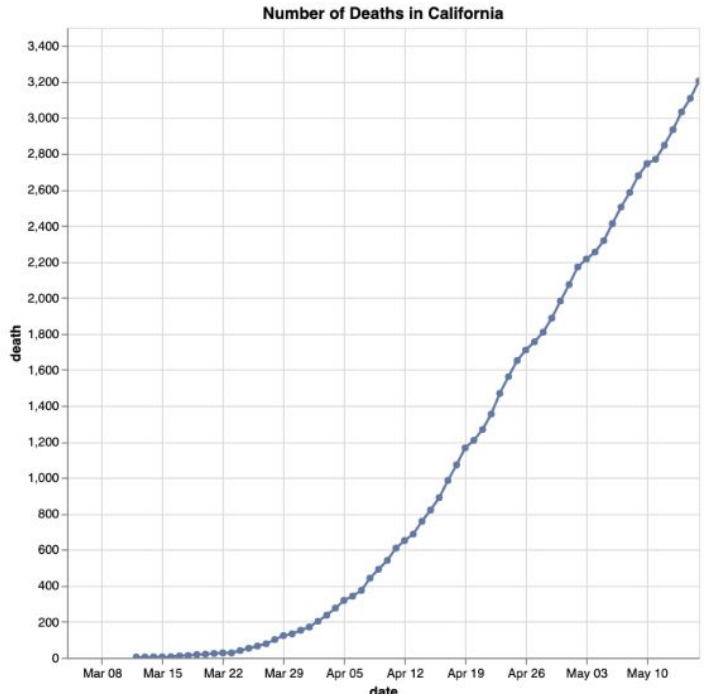
```
[ ] ca_data = states_df[states_df["state"] == "CA"]
    ca_data.head(5)
```

	date	state	positive	negative	pending	hospitalizedCurrently	hospitalizedCumulative	inIcuCurrently	inIcuCumu
5	2020-05-16	CA	76793.0	1102333.0	NaN	4424.0	NaN	1313.0	
61	2020-05-15	CA	74936.0	1058970.0	NaN	4519.0	NaN	1324.0	
117	2020-05-14	CA	73164.0	1031487.0	NaN	4655.0	NaN	1324.0	
173	2020-05-13	CA	71141.0	994451.0	NaN	4545.0	NaN	1314.0	
229	2020-05-12	CA	69382.0	963988.0	NaN	4544.0	NaN	1349.0	

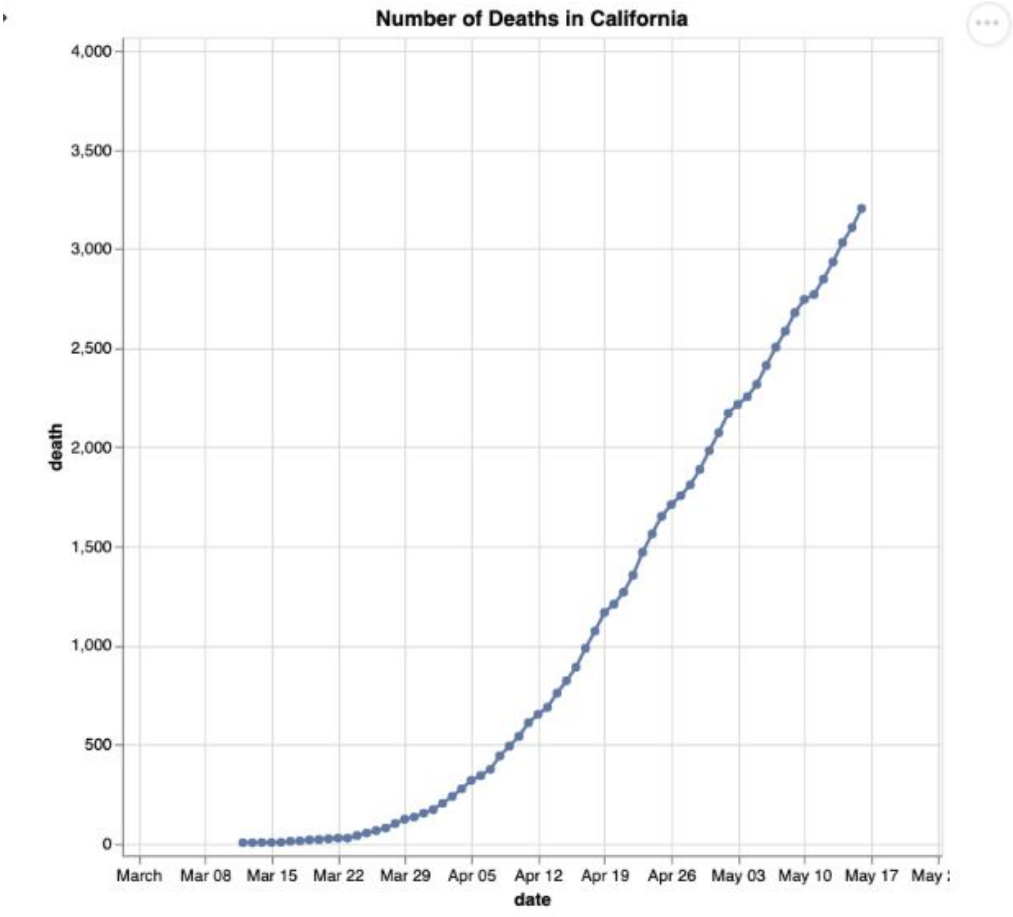
```
[ ] #@title
    alt.Chart(ca_data).mark_line(point=True).encode(
        x='date:T',
        y='positive:Q'
    ).properties(
        title = " Count of Positve Cases  in California",
        width = 500,
         height = 500
    )
```



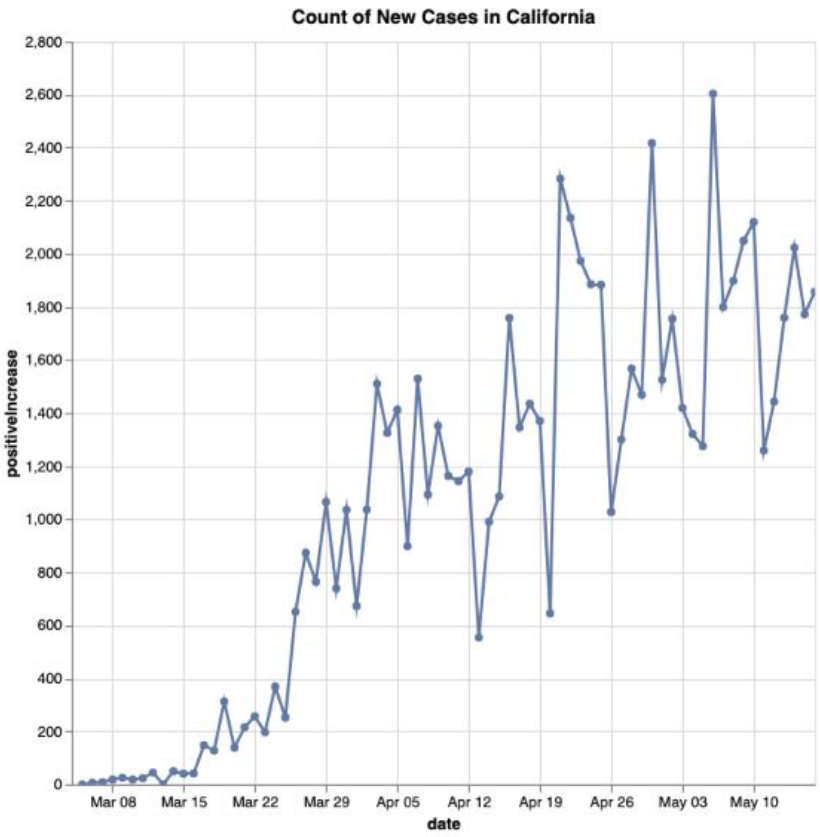
```
#@title
scales = alt.selection_interval(bind='scales')


alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='death:Q',
    tooltip = ["death", "date:T"]
).add_selection(
    scales
).properties(
    title = " Number of Deaths in California",
    width = 500,
     height = 500
).interactive()
```



```
#@title
scales = alt.selection_interval(bind='scales')


alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='death:Q',
    tooltip = ["death", "date:T"]
).add_selection(
    scales
).properties(
    title = " Number of Deaths in California",
    width = 500,
     height = 500
).interactive()
```

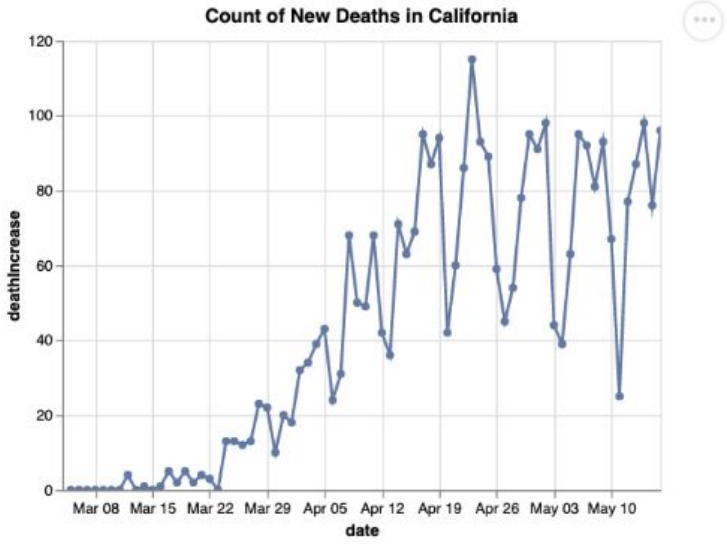


```
[ ] #@title
    alt.Chart(ca_data).mark_line(point=True).encode(
        x='date:T',
        y='positiveIncrease:Q',
        tooltip = ["date", "positiveIncrease:Q", "postive:Q"]
    ).properties(
        title = "Count of New Cases in California",
        width = 500,
         height = 500
    ).interactive()

    #why is it 0 on April 12. No new cases from April 11 to April 12.
```

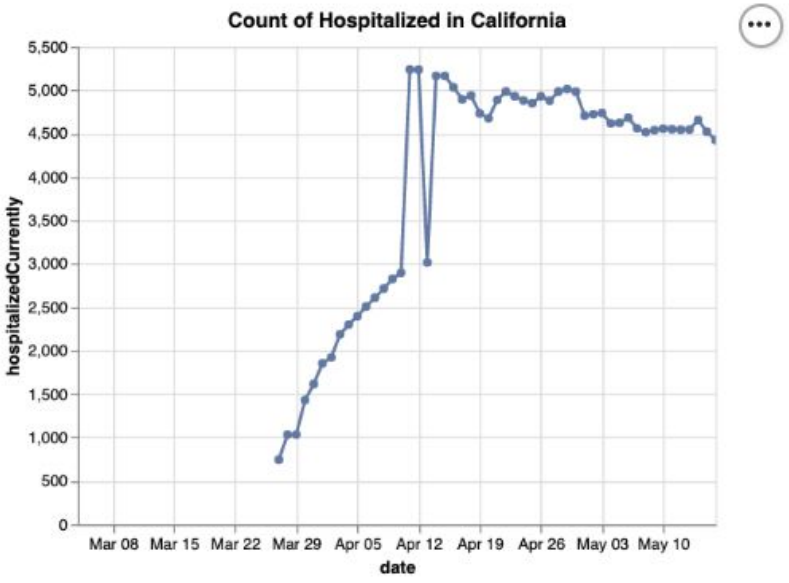



```
#@title
```

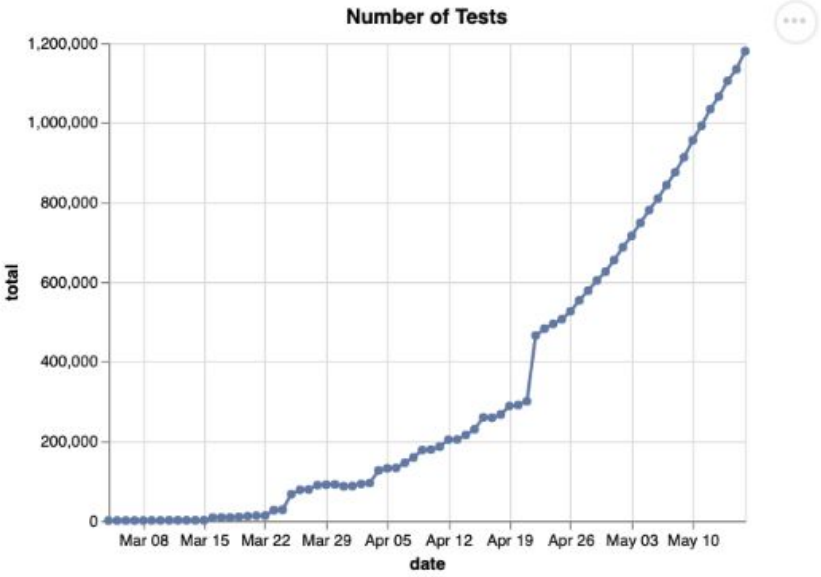
```
#@title

alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='deathIncrease:Q',
    tooltip = ["date",'deathIncrease:Q']
).properties(
    title = "Count of New Deaths in California"
).interactive()

#why is it 0 on April 12. No new cases from April 11 to April 12.
#decrease in deaths after April 23
```



```
[ ] #@title
    #Number of Hospitalized  in California
    alt.Chart(ca_data).mark_line(point=True).encode(
        x='date:T',
        y='hospitalizedCurrently:Q',
         tooltip = ["date", "hospitalizedCurrently:Q"]
    ).properties(
        title = " Count of Hospitalized in California"
    ).interactive()
```




```
#@title
alt.Chart(ca_data).mark_line(point=True).encode(
    x='date:T',
    y='total:Q'
).properties(
    title = "Number of Tests"
)
```



Top 10 States with the Most COVID-19 Deaths per million

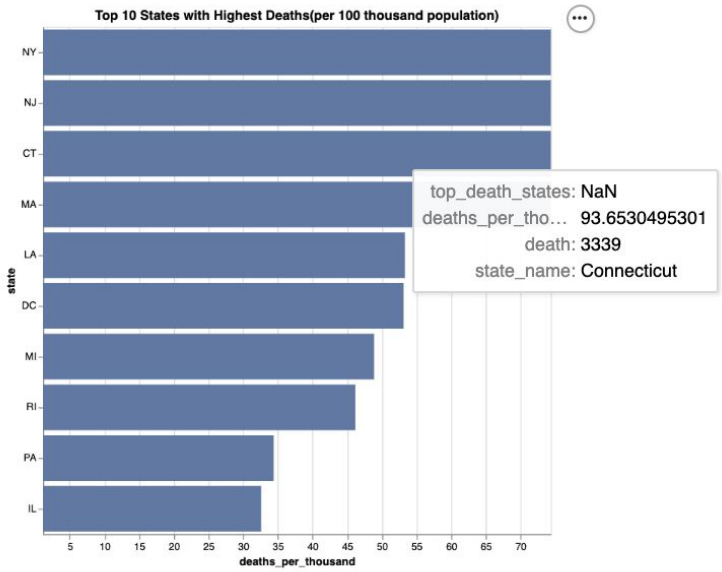
```
[ ] most_recent = states_df.head(56)
    most_recent.nlargest(10, "death")[['state_name', 'death']]
```

	state_name	death
37	New York	22478.0
34	New Jersey	10249.0
21	Massachusetts	5705.0
24	Michigan	4880.0
41	Pennsylvania	4403.0
16	Illinois	4129.0
7	Connecticut	3339.0
5	California	3204.0
20	Louisiana	2479.0
10	Florida	2040.0

```
[ ] most_recent = states_df.head(56)
    top_10_states = most_recent.nlargest(10, "deaths_per_thousand")
    top_10_states.rename(columns = {'deaths_per_thousand': "Mortality Rate"})[["state_name", "Mortality Rate"]]
```

	state_name	Mortality Rate
37	New York	115.546968
34	New Jersey	115.388209
7	Connecticut	93.653050
21	Massachusetts	82.771092
20	Louisiana	53.325658
8	District of Columbia	53.135038
24	Michigan	48.864222
43	Rhode Island	46.159902
41	Pennsylvania	34.393093
16	Illinois	32.584109

```
#@title
top_death_states = top_10_states[['state', 'deaths_per_thousand',"state_name", "death"]]

bars = alt.Chart(top_death_states).mark_bar().encode(
    x = "deaths_per_thousand:Q" ,
    y = alt.Y('state:N', sort = '-x'),
    tooltip = ['top_death_states:Q', "deaths_per_thousand:Q" ,"death:Q", "state_name"]
).properties(
      title = "Top 10 States with Highest Deaths(per 100 thousand population)",
      width = 500,
     height = 500
).interactive()
bars
```



State Comparison: Cases

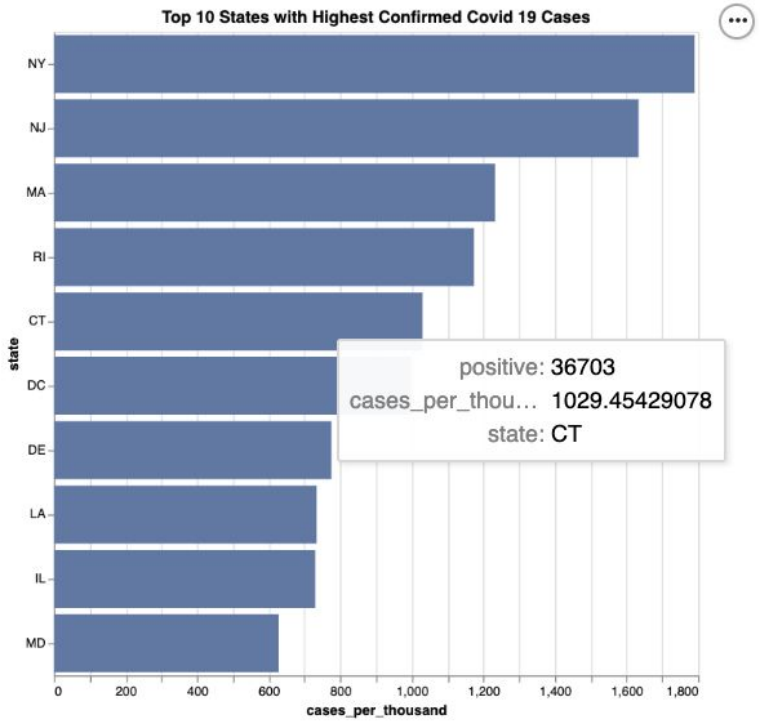
```
[ ] #States with the highest number of COVID19 cases
    states_most_cases = most_recent.nlargest(10, "positive")
    states_most_cases[['state_name', 'positive']]
```

	state_name	positive
37	New York	348232.0
34	New Jersey	145089.0
16	Illinois	92457.0
21	Massachusetts	84933.0
5	California	76793.0
41	Pennsylvania	61611.0
24	Michigan	50504.0
47	Texas	46999.0
10	Florida	44811.0
22	Maryland	37968.0

```
[ ] #States with the highest cases (per thousand)
    top_10_states = most_recent.nlargest(10, "cases_per_thousand")
    top_case_states = top_10_states[['state', 'cases_per_thousand',"state_name", "positive"]]
    top_case_states[["state_name", "cases_per_thousand"]]
```

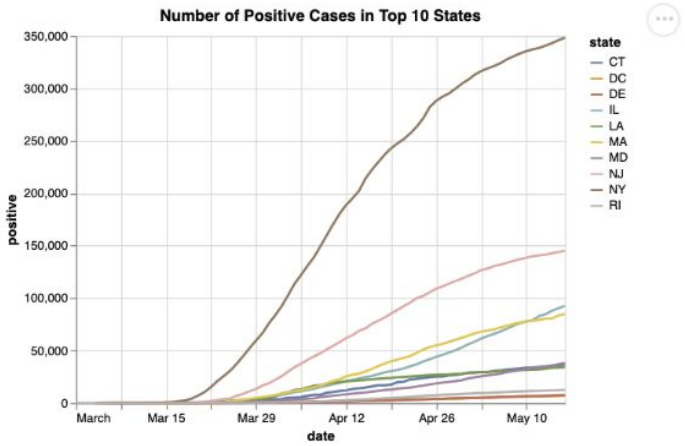
	state_name	cases_per_thousand
37	New York	1790.068153
34	New Jersey	1633.482283
21	Massachusetts	1232.251912
43	Rhode Island	1173.726426
7	Connecticut	1029.454291
8	District of Columbia	997.805169
9	Delaware	775.033786
20	Louisiana	733.889262
16	Illinois	729.626784
22	Maryland	628.018684

```
#@title
bars = alt.Chart(top_case_states).mark_bar().encode(
    x="cases_per_thousand:Q" ,
    y=alt.Y('state:N', sort='-x'),

    tooltip=['positive',"cases_per_thousand:Q","state"],
).properties(
      title = "Top 10 States with Highest Confirmed Covid 19 Cases",
        width= 500,
        height=500
).interactive()
bars
```



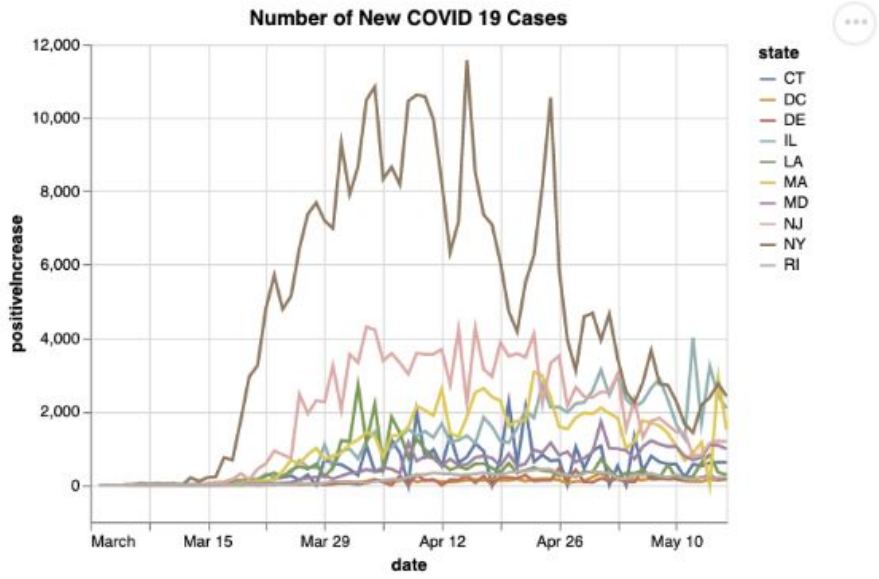
```
top_10_states["date"] = pd.to_datetime(top_10_states["date"],format='%Y%m%d')
```

```
#@title
states_lst = list(top_10_states["state"])
filtered_df = states_df[states_df.state.isin(states_lst)]
filtered_df[["state","date", "positive"]]


alt.Chart(filtered_df).mark_line().encode(
    x='date:T',
    y='positive',
    color='state',
).properties(
    title = " Number of Positive Cases in Top 10 States"
)
```

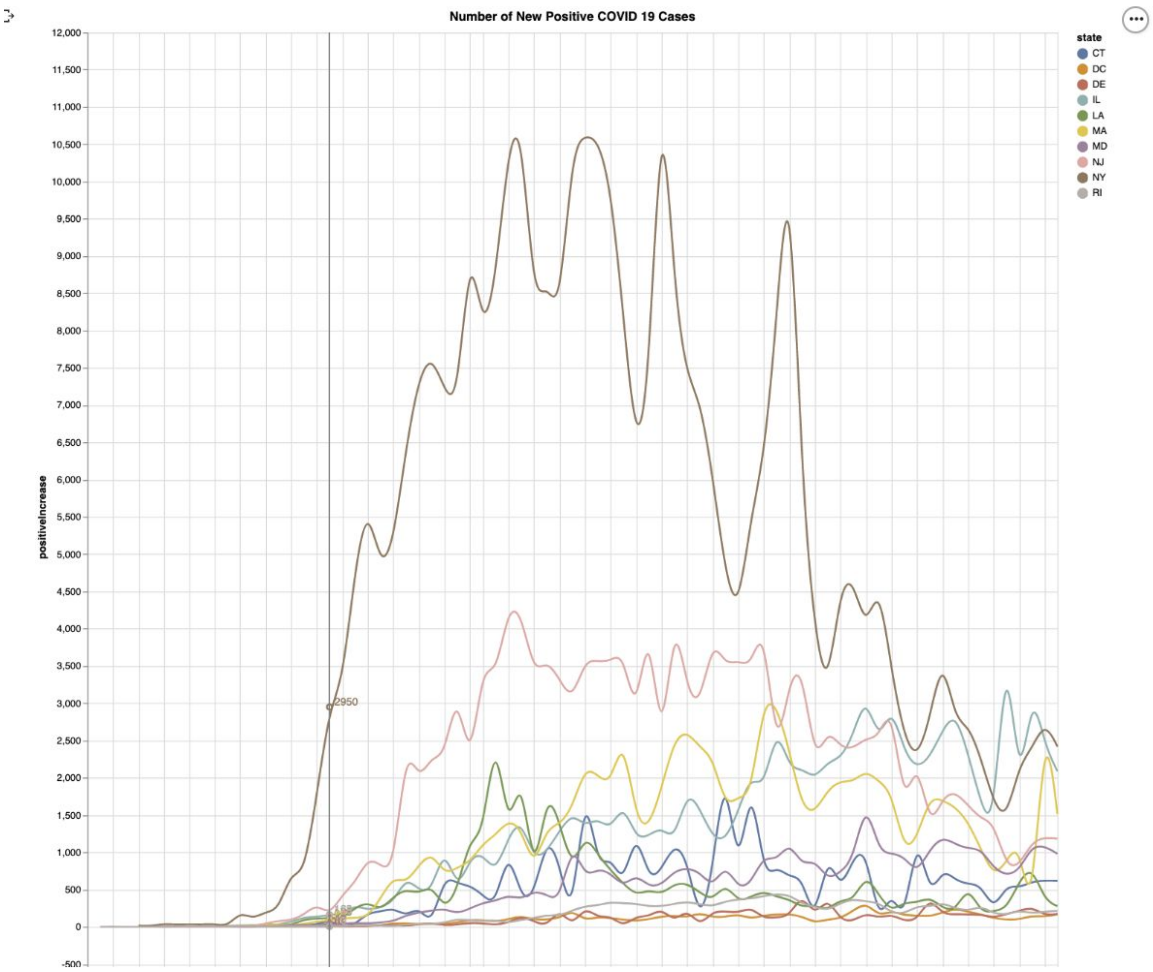


date

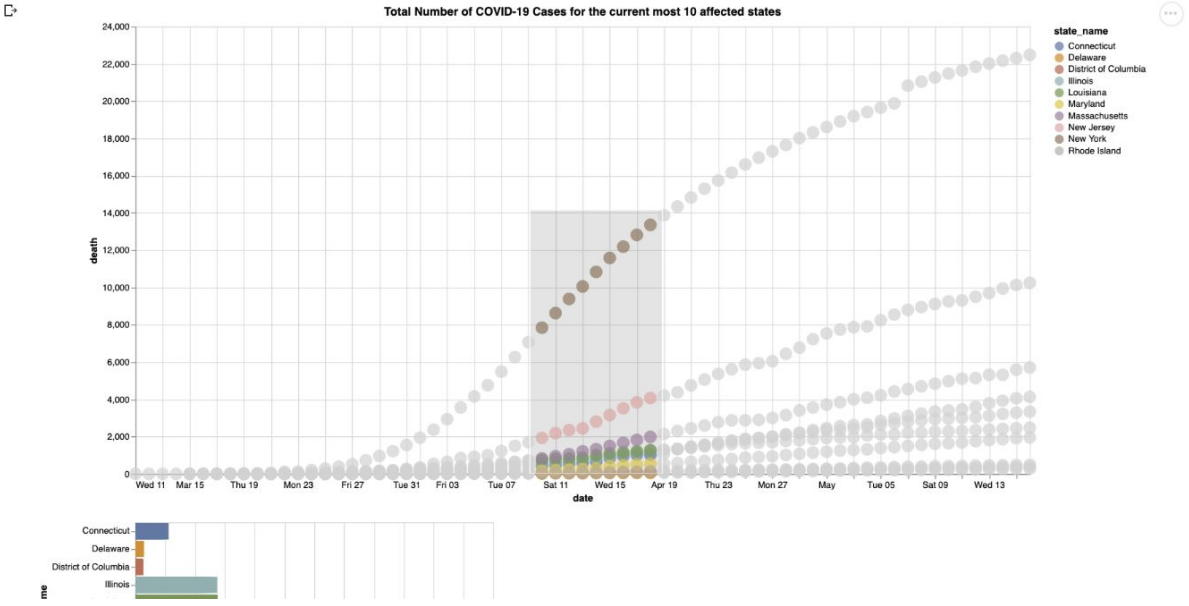
```
#@title
#static chart
alt.Chart(filtered_df).mark_line().encode(
    x='date:T',
    y='positiveIncrease',
    color='state',
).properties(title = "Number of New COVID 19 Cases")
```



```
#for top 10 states
source = filtered_df[["state","date", "positiveIncrease"]]

# Create a selection that chooses the nearest point & selects based on x-value
nearest = alt.selection(type='single', nearest=True, on='mouseover',
                        fields=['date'], empty='none')

# The basic line
line = alt.Chart(source).mark_line( interpolate='basis').encode(
    x='date:T',
    y='positiveIncrease:Q',
    color='state:N'
)

# Transparent selectors across the chart. This is what tells us
# the x-value of the cursor
selectors = alt.Chart(source).mark_point(point=True).encode(
    x='date:T',
    opacity=alt.value(0),
).add_selection(
    nearest
)

# Draw points on the line, and highlight based on selection
points = line.mark_point().encode(
    opacity=alt.condition(nearest, alt.value(1), alt.value(0))
)

# Draw text labels near the points, and highlight based on selection
text = line.mark_text(align='left', dx=5, dy=-5).encode(
    text=alt.condition(nearest, 'positiveIncrease:Q', alt.value(' '))
)

# Draw a rule at the location of the selection
rules = alt.Chart(source).mark_rule(color='gray').encode(
    x='date:T',
).transform_filter(
    nearest
)

# Put the five layers into a chart and bind the data
alt.layer(
    line, selectors, points, rules, text
).properties(
    width=1000, height=1000,
    title = " Number of New Positive COVID 19 Cases"
)
```


Number of New Positive COVID 19 Cases
positiveIncrease
state
CT
DC
DE
IL
LA
MA
MD
NJ
NY
RI
12,000
11,500
11,000
10,500
10,000
9,500
9,000
8,500
8,000
7,500
7,000
6,500
6,000
5,500
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
-500
2950

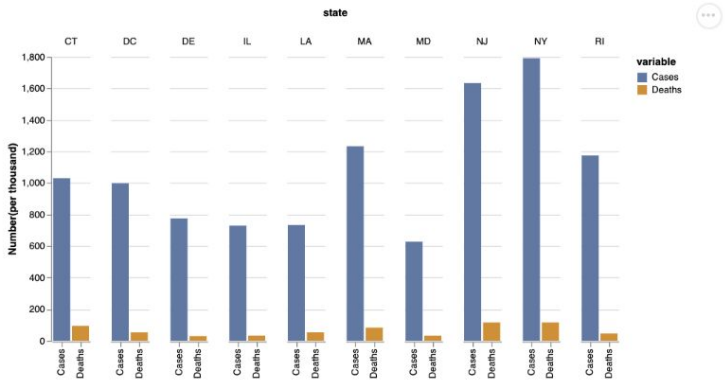
```
states_df["date"] = pd.to_datetime(states_df["date"], format='%Y%m%d')
states_lst = top_10_states["state"]


source =states_df[states_df.state.isin(states_lst)]

brush = alt.selection(type='interval')

points = alt.Chart(source).mark_circle(size=200).encode(
    x='date:T',
    y='death:Q',
    color=alt.condition(brush, 'state_name:N', alt.value('lightgray')),
    tooltip=[ 'date', 'total', 'death', "state_name"]
).add_selection(
    brush
).properties(
      width= 1000,
      height=500,
      title = "Total Number of COVID-19 Cases for the current most 10 affected states",
)

bars = alt.Chart(source).mark_bar().encode(
    y='state_name:N',
    color='state_name:N',
    x='total:Q'
).transform_filter(
    brush
)

points & bars
```



Comparison between deaths and covid 19 cases

```
data = pd.concat([top_10_states, top_death_states], axis=1)
data = data[['state', 'cases_per_thousand',"deaths_per_thousand" ]]
data = data.loc[:, ~data.columns.duplicated()]

melted_df = data.melt("state")
melted_df["variable"].replace({"cases_per_thousand": "Cases", "deaths_per_thousand": "Deaths"}, inplace=True)
```

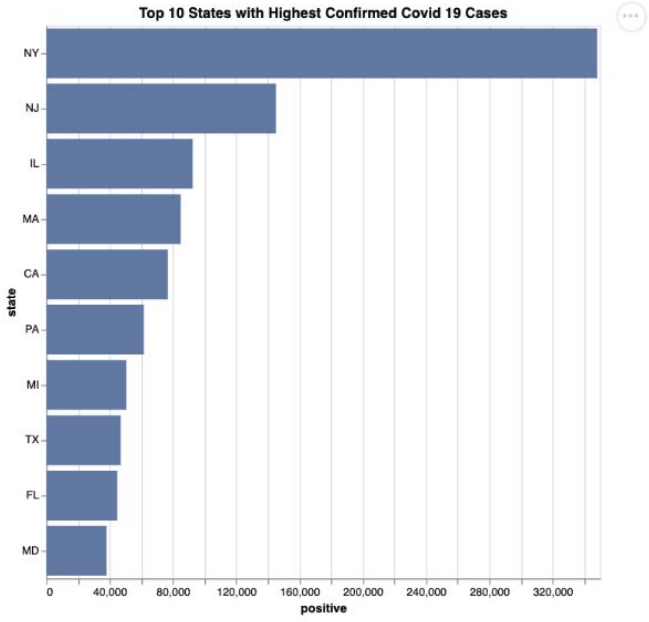
```
alt.Chart(melted_df).mark_bar().encode(
    alt.X('variable:N', axis= alt.Axis(title='',orient = "bottom")),
    alt.Y('value:Q', axis = alt.Axis(title='Number(per thousand)')),
    color= alt.Color('variable:N'),
    column ='state:O'
).configure_view(
    stroke='transparent'
)
```



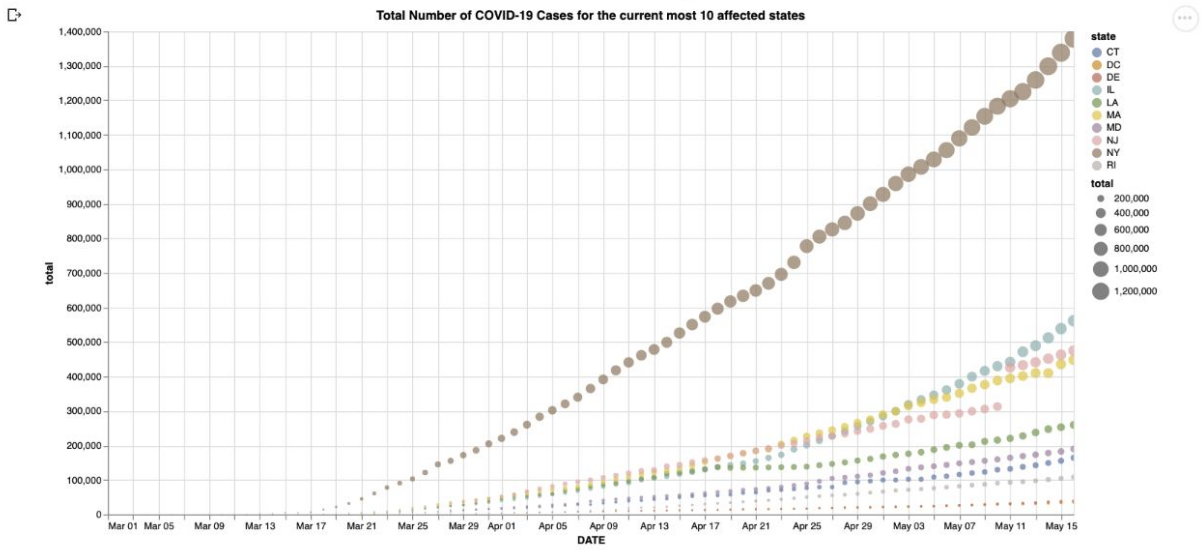
State Comparison : Baseline States

```
[ ]  Baseline = ["WA", "NY", "CA"]
```

```
#Raw count numbers

#States with the highest number of COVID-19 cases
most_num_cases = most_recent.nlargest(10, "positive")[['state', 'positive']]
most_num_cases

bars = alt.Chart(most_num_cases).mark_bar().encode(
    x="positive",
    y=alt.Y('state', sort='-x'),
    tooltip=["positive", "state"],
).properties(
      title = "Top 10 States with Highest Confirmed Covid 19 Cases",
        width= 500,
        height=500
).interactive()
bars
```



```
[ ]
    categorical_chart = alt.Chart(states_df[states_df.state.isin(states_lst)]).mark_circle(size=200).encode(

                                alt.X('date:T', axis = alt.Axis(title = 'Date'.upper(), format = ("%b %d"))),
                                y='total:Q',
                                color='state:N',
                                size ='total:Q' ,
                                tooltip=['state', 'date', 'total', 'death']).properties(
                                    title = "Total Number of COVID-19 Cases for the current most 10 affected states",
                                      width= 1000,
                                      height=500
    ).interactive()

    categorical_chart
```



How effectively are states testing for COVID-19?

Statisitical correlation measures the strength between two variables. To summarize the correlation,we also computed the Pearson coefficient. The correlation coefficient range from -1 to 1, indicating strong negative correlation to strong positve correlation. Value of 0 indicates no correlation.

Furthermore, we computed P-values to determine if the correlation coefficient is statistically significant. The null hypothesis is stated as "No correlation between the two variables". If the $p < 0.05$, we stated that the correlation is statistically significant and can still be due to chance. However, if $p < 0.01$, the correlation coefficient is highly statistcally signifcant and it cannot be attributed to random chance.

Lastly, correlation does not imply causation!

In order to do a state by state comparison, we transformed the raw number of confirmed cases to number of confirmed cases per (100,000) people.

```
[ ] #@title
    #Calculate testing rate per capita
    print("States with the most number of cases:")
    list(states_most_cases["state_name"])
```

```
States with the most number of cases:
['New York',
 'New Jersey',
 'Illinois',
 'Massachusetts',
 'California',
 'Pennsylvania',
 'Michigan',
 'Texas',
 'Florida',
 'Maryland']
```

```
#@title
print("States with the most number of cases per capita:")
lst = list(top_10_states["state_name"])
lst
```

```
States with the most number of cases per capita:
['New York',
 'New Jersey',
 'Massachusetts',
 'Rhode Island',
 'Connecticut',
 'District of Columbia',
 'Delaware',
 'Louisiana',
 'Illinois',
 'Maryland']
```

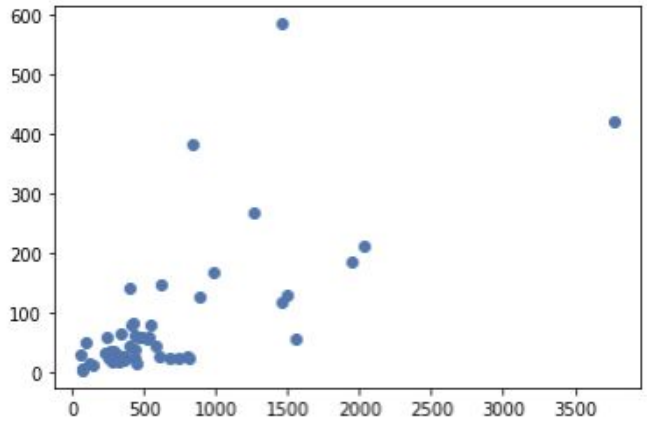
```
    Maryland ]
```

```
[ ] df1 = pd.read_csv('COVID19_state.csv' )

    df1['Tested_per_thous'] = (df1['Tested']/df1['Population'])* 100_000
    df1['Infected_per_thous'] = (df1['Infected']/df1['Population']) * 100_000
    df1['Deaths_per_thous'] = (df1['Deaths']/df1['Population'])* 100_000
    df1['ICU Beds_per_thous'] =  (df1['ICU Beds']/df1['Population'])* 100_000
    df1['Respiratory Deaths_per_thous'] =  (df1['Respiratory Deaths']/df1['Population'])* 100_000
    df1['Physicians_per_thous'] =  (df1['Physicians']/df1['Population'])* 100_000


    df1[df1.State.isin(lst)]
```

	State	Tested	Infected	Deaths	Population	Pop Density	Gini	ICU Beds	Income	GDP	Unemployment	Ra
6	Connecticut	22029	5276	165.0	3563077	735.8689	0.4945	674	74561	76342	3.8	0.947
7	District of Columbia	6438	902	21.0	720687	11814.5410	0.5420	314	47285	200277	5.2	0.888
8	Delaware	4289	593	14.0	982895	504.3073	0.4522	186	51449	77253	3.9	0.926
14	Illinois	53581	10357	243.0	12659682	228.0243	0.4810	3144	56933	67268	3.4	0.955
18	Louisiana	58498	12496	409.0	4645184	107.5175	0.4990	1289	45542	53589	5.2	0.931
19	Massachusetts	68800	11736	216.0	6976597	894.4355	0.4786	1326	70073	82480	2.8	0.943
20	Maryland	28337	3609	67.0	6083116	626.6731	0.4499	1134	62914	68573	3.3	0.929
31	New Jersey	75356	34124	846.0	8936574	1215.1991	0.4813	1822	67609	69378	3.8	0.946
34	New York	283621	113704	3565.0	19440469	412.5211	0.5229	3952	68667	85746	3.7	0.937
39	Rhode Island	4190	452	18.0	1056161	1021.4323	0.4781	279	54523	57852	3.4	0.942

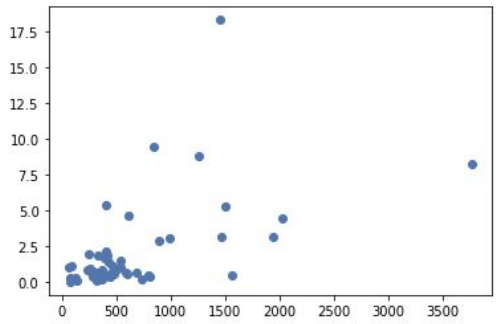
```
#@title
x = df1['Tested_per_thous']
y = df1['Infected_per_thous']

print('data1: mean=%.3f stdv=%.3f' % (mean(x), std(x)))
print('data2: mean=%.3f stdv=%.3f' % (mean(y), std(y)))

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#We can conclude that this statistically significant
```

```
data1: mean=655.224 stdv=638.914
data2: mean=84.495 stdv=111.784
Pearson Coefficient
(0.6894975646316808, 2.2018792718413426e-08)
```



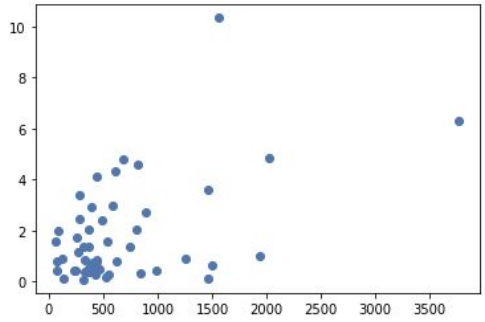
```
x = df1['Tested_per_thous']
y = df1['Deaths_per_thous']

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#Postive correlation between tested and  death and its satistically significant
```

```
Pearson Coefficient
(0.5529806714228136, 2.575571104387515e-05)
```




```
#Tested and Respiratory deaths
x = df1['Tested_per_thous']
y = df1[ 'Respiratory Deaths_per_thous']

print('data1: mean=%.3f stdv=%.3f' % (mean(x), std(x)))
print('data2: mean=%.3f stdv=%.3f' % (mean(y), std(y)))

print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#We can conclude that this statistically significant
#positive
```

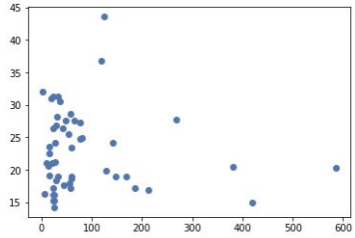
```
data1: mean=655.224 stdv=638.914
data2: mean=1.775 stdv=1.919
(0.4857647261190999, 0.0003021158676463182)
```



Are states responding affectively?

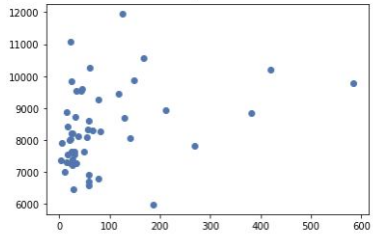
```
[ ] x = df1['Infected_per_thous']
    y = df1['ICU Beds_per_thous']

    print('Pearson Coefficient')
    print(pearsonr(x, y ))

    # plot
    plt.scatter(x, y)
    plt.show()

    #No correlation between Infected and ICU Beds
```

```
Pearson Coefficient
(-0.10082838525577763, 0.48142977781457547)
```



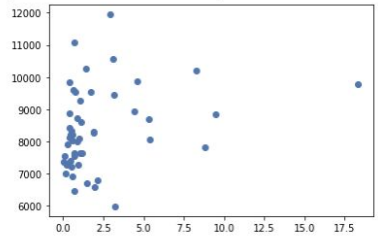

```
[ ] x = df1['Infected_per_thous']
    y = df1['Health Spending']

    print('Pearson Coefficient')
    print(pearsonr(x, y ))

    # plot
    plt.scatter(x, y)
    plt.show()

    #Positive Correlation between infected and spending. This is statistically signifance(could be due to chance)
```

```
Pearson Coefficient
(0.319189374083178, 0.022428161013691512)
```



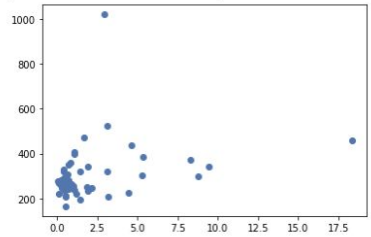
```
x = df1['Deaths_per_thous']
y = df1['Health Spending']

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#Positive Correlation between deaths and health spending. This is statistically signifance(could be due to chance)
```

```
Pearson Coefficient
(0.29166690726531974, 0.037831973610154584)
```



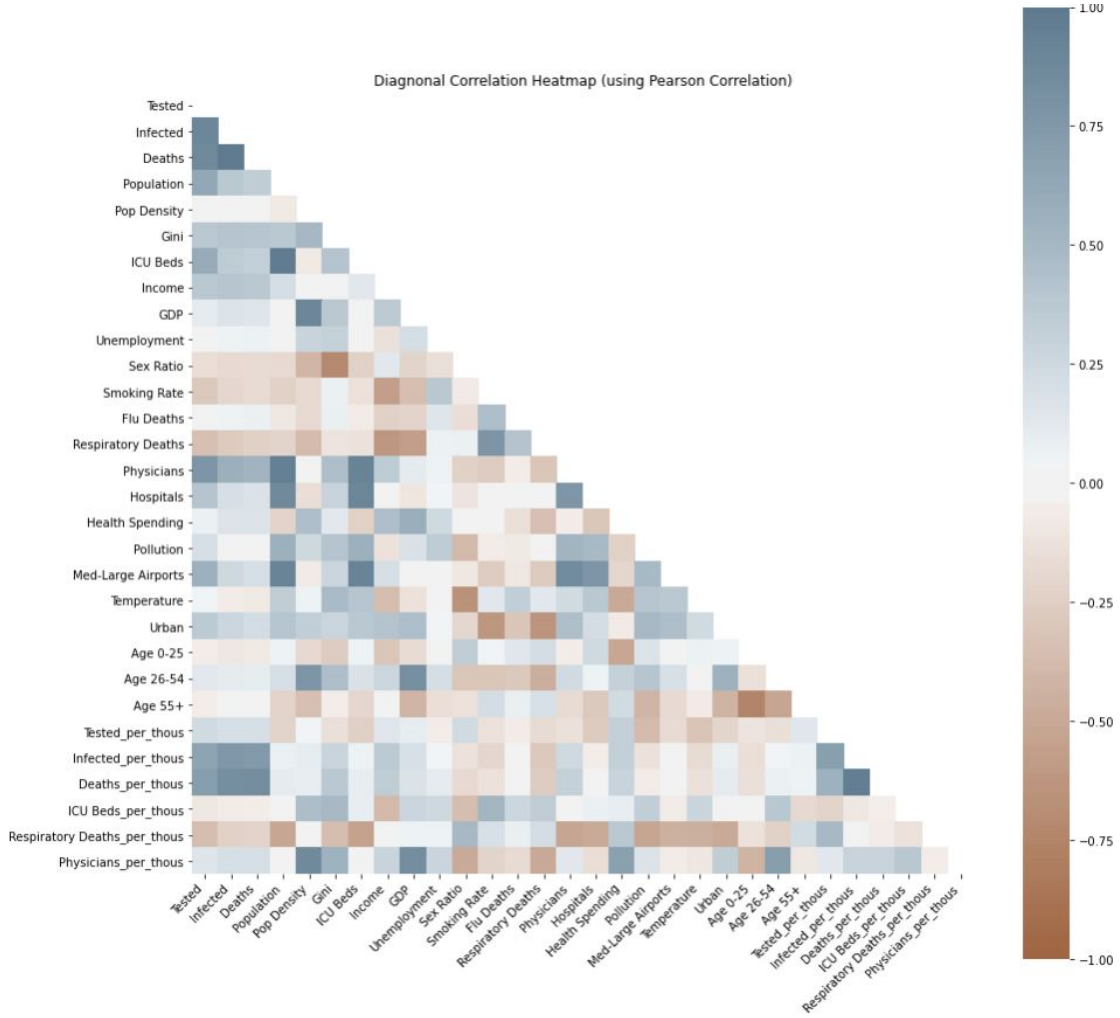

```
x = df1['Deaths_per_thous']
y = df1['Physicians_per_thous']

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#Positive Correlation between number of deaths and number of physicians
```

```
Pearson Coefficient
(0.29079641191199757, 0.038434589171752624)
```



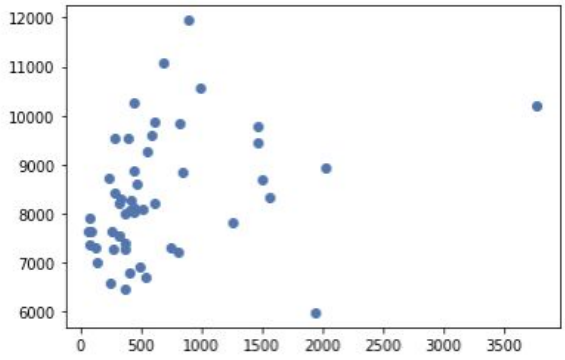
Diagonal Correlation Heatmap

```
[ ] import seaborn as sns
    import matplotlib.pyplot as plt
    import numpy as np

    fig, ax = plt.subplots(figsize=(15,15))
    corr = df1.corr()

    mask = np.triu(np.ones_like(corr, dtype=np.bool))

    ax = sns.heatmap(
        corr,
        mask = mask,
        vmin=-1, vmax=1, center=0,
        cmap=sns.diverging_palette(20, 220, n=200),
        square=True
    )
    ax.set_xticklabels(
        ax.get_xticklabels(),
        rotation=45,
        horizontalalignment='right'
    );
    plt.title ("Diagnonal Correlation Heatmap (using Pearson Correlation)");
```


Diagnonal Correlation Heatmap (using Pearson Correlation)
Tested
Infected
Deaths
Population
Pop Density
Gini
ICU Beds
Income
GDP
Unemployment
Sex Ratio
Smoking Rate
Flu Deaths
Respiratory Deaths
Physicians
Hospitals
Health Spending
Pollution
Med-Large Airports
Temperature
Urban
Age 0-25
Age 26-54
Age 55+
Tested_per_thous
Infected_per_thous
Deaths_per_thous
ICU Beds_per_thous
Respiratory Deaths_per_thous
Physicians_per_thous
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

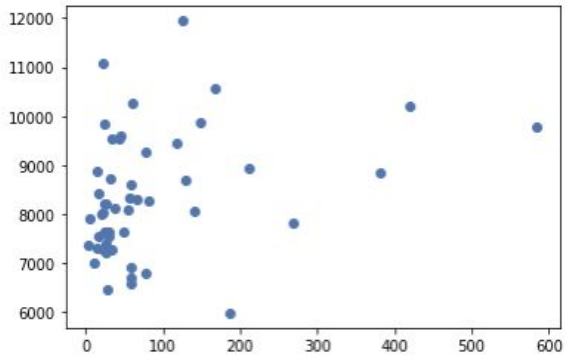
```
[ ] x = df1['Tested_per_thous']
    y = df1['Health Spending']

    print(pearsonr(x, y )) # assumes linear relationship

    # plot
    plt.scatter(x, y)
    plt.show()

    #Positive correlation between health spending and  testing
    #becuase pvalue is <.05 this correlation is statistically significant
```

```
(0.3129457150316881, 0.025354040759488896)
```



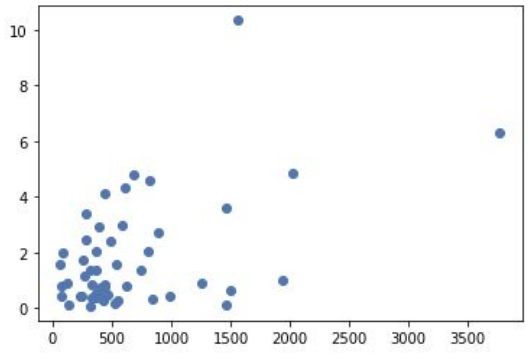

```
[ ] x = df1['Infected_per_thous']
    y = df1['Health Spending']

    print('data1: mean=%.3f stdv=%.3f' % (mean(x), std(x)))
    print('data2: mean=%.3f stdv=%.3f' % (mean(y), std(y)))

    print(pearsonr(x, y )) # assumes linear relationship

    # plot
    plt.scatter(x, y)
    plt.show()

    #We can conclude that this statistically significant
    #positive relationship
```

```
data1: mean=84.495 stdv=111.784
data2: mean=8332.157 stdv=1244.369
(0.319189374083178, 0.022428161013691512)
```



```
#Tested and Respiratory deaths
x = df1['Tested_per_thous']
y = df1[ 'Respiratory Deaths_per_thous']

print('data1: mean=%.3f stdv=%.3f' % (mean(x), std(x)))
print('data2: mean=%.3f stdv=%.3f' % (mean(y), std(y)))

print('Pearson Coefficient')
print(pearsonr(x, y )) # assumes linear relationship

# plot
plt.scatter(x, y)
plt.show()

#We can conclude that this statistically significant
#positive
```

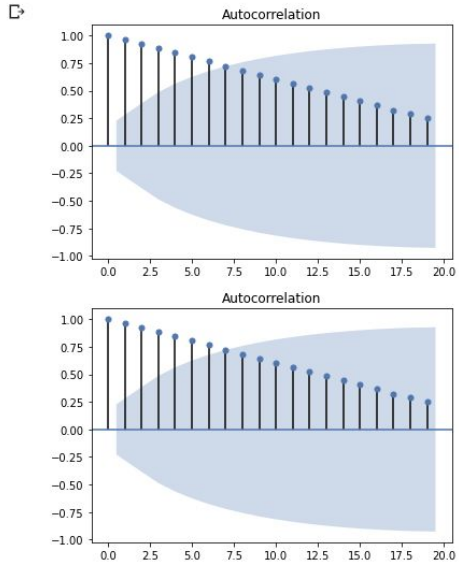
```
data1: mean=655.224 stdv=638.914
data2: mean=1.775 stdv=1.919
Pearson Coefficient
(0.4857647261190999, 0.0003021158676463182)
```



Autocorrelation and Partial Autocorrelation Plots

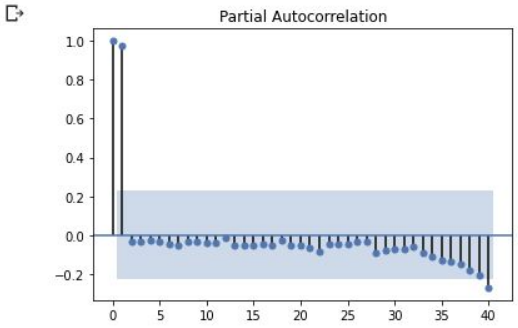
Autocorrelation and partial autocorrelation plots are very crucial in time series analysis and forecasting and are used to summarize the strength of a relationship with an observation in a time series with observations at prior time steps.

```
[ ] #how correlated points are with each other, based on how many time steps they are separated by
    #how correlated past data points are to future data points, for different values of the time separatio

    from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
    plot_acf(ca_data["positive"])

    #On the graph, there is a vertical line (a "spike") corresponding to each lag.
    #The height of each spike shows the value of the autocorrelation function for the lag.
```

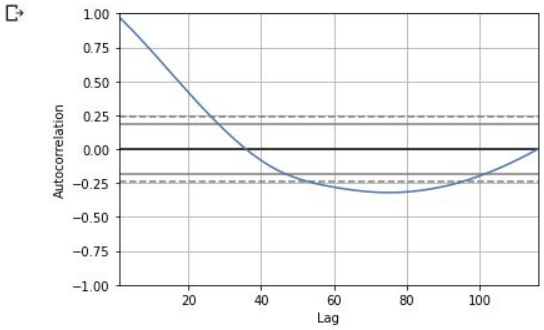


Partial autocorrelation summarizes the relationship between an observed value at a particular time step with the observe values at prior time steps with the relationships of intervening observations removed.

```
[ ] plot_pacf(ca_data["positive"], lags=40)
    plt.show()
```



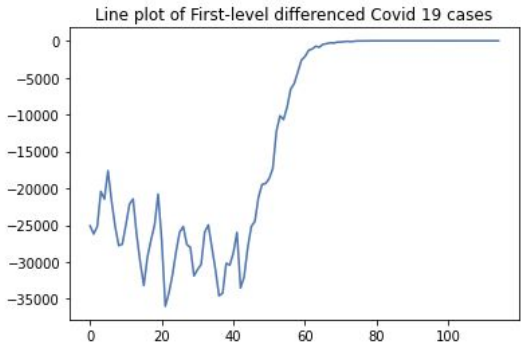

```
[ ] from pandas.plotting import autocorrelation_plot

    autocorrelation_plot(us_df["positive"]);
    #There is autocorrelation and this statistically significant
```



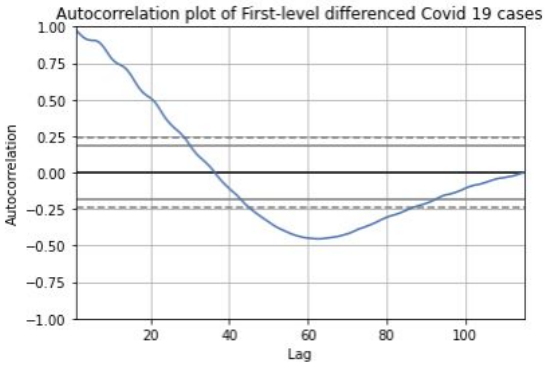
Differencing

```
[ ] def difference(inp):
      "Used to make the data stationary."
      diff = list()
      for i in range(1, len(inp)):
        value = inp[i] - inp[i - 1]
        diff.append(value)
      return diff
```

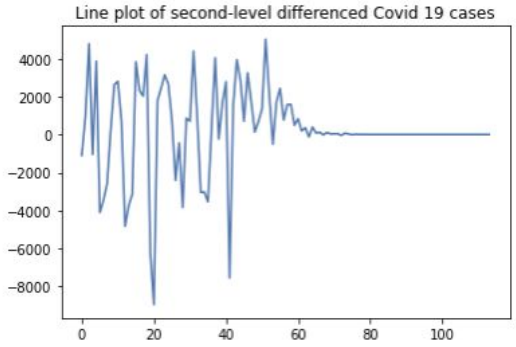
```
[ ] raw_cases = us_df["positive"]

    first_level =  difference(raw_cases)
    plt.plot(first_level);
    plt.title("Line plot of First-level differenced Covid 19 cases");
```




```
[ ]
    autocorrelation_plot(first_level)
    plt.title("Autocorrelation plot of First-level differenced Covid 19 cases");
```

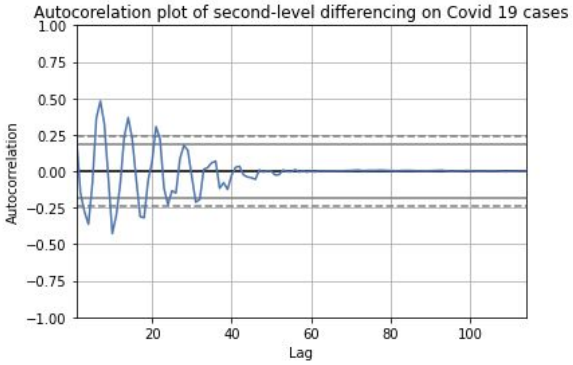


```
second_level = difference(first_level)
plt.plot(second_level);
plt.title("Line plot of second-level differenced Covid 19 cases");
```




```
autocorrelation_plot(second_level)
plt.title("Autocorelation plot of second-level differencing on Covid 19 cases");
print("Data is now stationary!")
```

```
Data is now stationary!
```




```
third_level_diff = difference(second_level)
autocorrelation_plot(third_level_diff)
plt.title("Autocorelation plot of third-level differencing on Covid 19 cases");
print("Data is now stationary!")
```

```
Data is now stationary!
```