DSE3101 Technical Documentation

The Trio: Nguyen Duc Thinh, Tan Win Yi, Li Yu

2025-04-19

Table of Contents

# I. Introduction

## 1. Project overview

Macroeconomic forecasting plays a critical role in shaping national policy, but it faces a major challenge where real-time GDP data is often revised over time, making forecasting decisions difficult. Research (Croushore and Stark, 2001) has shown that forecasts using real-time data tend to be less accurate than those using revised data whereby traditional models may not reflect the actual economic situation.

Our project addresses this issue by developing a web-based interactive application that allows users to benchmark various time-series forecasting models; using real-time and revised GDP data. Our application enables users to visualize and compare the accuracy and robustness of different models across data vintages and forecast horizons.

Our main goal is to help policymakers and economists evaluate how forecasts would have performed in real-time, quantify the impact of data revisions, and identify which models remain reliable under changing economic conditions.

## 2. Overall design

Our application consist of two main tabs: Dataset and Model. The Dataset Tab is where users choose to either work with our provided sample dataset or upload their own. Here user will also set up their interest forecasting period. We also provide a preview of the cleaned data from the starting period, alongside with visualizations to help users to examine the differences in current and vintage data.

The Model tab allows users to select forecasting models for evaluation and customize them by specifying key parameters and features. This interactive setup ensures that users can experiment with different models and directly observe their impact on forecasting performance. Error metrics and visualization will be provided to assess the results.

## 3. Data and methodology

Our primary data source is available at <https://www.philadelphiafed.org/surveys-and-data/real-time-data-research/routput> We also use the FRED API to source additional economic data for model enhancement.

In this project, we evaluate model performance using two different data settings: vintage data (data available at the time of forecasting) and latest vintage data (revised data). This comparison helps quantify the impact of data revisions on forecast accuracy.

We focus on three models:

* Autoregressive (AR): A benchmark model that uses only past values of GDP growth to generate forecasts.
* Autoregressive Distributed Lag (ADL): Extends the AR model by incorporating additional economic indicators.
* K-Nearest Neighbors (KNN): A non-parametric machine learning model that relies on historical patterns

Model performance is assessed using two standard forecasting error metrics: Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). These metrics allow us to rank models consistently based on both accuracy and robustness across different data vintages.

# II. Application Backend

## 1. Application Framework

Before moving into building the application, it is necessary to go through our logic. The framework of how our forecast works is as follow: - Current Vintage Prediction: For each point we want to forecast, we will use the vintage data at that point to train the model and generate forecast. For instance, a forecast for 2000Q1 GDP growth will use the 2000Q1 vintage data to generate forecast. This is the real-time data that we have in 2000Q1 about all other dates. - Latest Vintage Prediction: We will use the latest vintage data for training our model (2025Q1), however, we use only use data up until the time of prediction. Thus, a 2000Q1 forecast with the latest data will only use GDP Growth up until 1999Q4. - Evaluation: We will then generate two sequences of forecasts, and calculate the error metrics of the sequences with the real value in the latest vintage data. The reason is that this most recent data is what closest to the truth by constantly going through revision.

## 2. Data Preparation

For data selection, users can either work with our provided sample dataset or upload their own. For sample dataset, we are using the real-time dataset available here for download: <https://www.philadelphiafed.org/surveys-and-data/real-time-data-research/routput>

For upload option, we currently support uploads in csv, xlsx, or json formats. Additionally, the uploaded dataset must follow a structure similar to the FRED data format to ensure compatibility with our processing.

Our initial plan was to incorporate both quarter vintages and monthly vintages. In order to do this, we need to detect the frequency of the dataset that users chose. This is where we define a function that can detect the frequency of dataset as below.

detect\_frequency <- function(data) {  
 # Get first vintage column name  
 first\_col <- names(data)[2]  
   
 if (str\_detect(first\_col, "M\\d+$")) {  
 return("monthly")  
 } else if (str\_detect(first\_col, "Q\\d$")) {  
 return("quarterly")  
 } else {  
 stop("Unsupported Uploaded File!")  
 }  
 }

After we have select our data, we start our data cleaning process. We begin by cleaning the column names, which are in the format “ROUTPUT65Q1”, to extract the correct year and quarter. For each subsequent column, we compare the last two digits of the year with the previous one: If the new year is greater than or equal to the previous, we keep the current prefix.

If it’s smaller, this signals a rollover into the next century, so we increment the prefix by 1 (e.g., from “19” to “20”).

For example, from “98” to “99”, we stay in the 1900s. When it shifts from “99” to “00”, we move to the 2000s.

## Get the right century for the year  
 clean\_columns <- function(data) {  
   
 data\_cols <- names(data)[-1] %>%  
 str\_remove(pattern = "ROUTPUT")  
   
 yy <- str\_sub(data\_cols, start = 1, end = 2)  
 prev\_yy <- yy[1]  
 century = 19  
 complete\_year <- c()  
 for (i in 1:length(yy)) {  
 cur\_yy <- yy[i]  
 if (as.numeric(cur\_yy) < as.numeric(prev\_yy)) {  
 century = century + 1  
 }  
 complete\_year[i] <- paste0(century, cur\_yy)  
 prev\_yy <- cur\_yy  
 }  
 return(complete\_year)  
 }

This function will get us the right prefix for each column. As a result, we can use this function and concatenate the prefix with the year and quarter/month and get a better column name ,for instance, “ROUTPUT65Q1” to “1965Q1”.

Next, we clean the dataset by reshaping into long format, allowing us to extract and organize the columns into year, quarter, v\_year (vintage year), v\_quarter/v\_month (vintage quarter/month) and the corresponding GDP value. This structure makes it easier to filter for the corresponding vintage in future analysis.

clean.data <- function(data, vintage\_freq = "quarterly") {  
   
 q <- names(data)[-1] %>% str\_remove(pattern = "ROUTPUT") %>% str\_sub(start = 1)  
 clean\_cols <- paste0(clean\_columns(data), q)  
 names(data)[-1] <- clean\_cols  
 total\_col <- length(names(data))  
 ## Clean the data  
 clean\_data <- data %>%  
 mutate(across(2:total\_col, as.numeric)) %>%  
 pivot\_longer(cols = -1, names\_to = "vintage",   
 values\_to = "current\_vintage") %>%  
 mutate(year = str\_sub(DATE, 1,4),  
 quarter = str\_sub(DATE, 7,7),   
 v\_year = str\_sub(vintage, 1,4),  
 log\_current\_vintage = log(current\_vintage)) %>%  
 drop\_na()  
   
 if (vintage\_freq == "quarterly") {  
 final\_data <- clean\_data %>%   
 mutate(v\_quarter = str\_extract(vintage, pattern = "(?<=Q).\*$")) %>%  
 select(year, quarter, v\_year, v\_quarter,   
 current\_vintage, log\_current\_vintage) %>%  
 mutate(across(1:6, as.numeric)) # Turn all value to numeric  
 }  
 else {  
 final\_data <- clean\_data %>%  
 mutate(v\_month = str\_extract(vintage, pattern = "(?<=M).\*$")) %>%  
 select(year, quarter, v\_year, v\_month, current\_vintage, log\_current\_vintage) %>%  
 mutate(across(1:6, as.numeric))  
 }  
   
   
 return(final\_data)  
 }

As the project progressed, we chose to focus exclusively on quarterly vintage data. This decision simplifies the modeling process while still aligning with our primary goal: quantifying the impact of data revisions. With that being said, it is completely possible to extend our model to accompany monthly vintage data as the structure would be very similar to how we handle quarterly data.

Moving on, we define a filtering function to extract data for a specific vintage, based on the selected vintage year and quarter. This function will take in the vintage year and vintage quarter and output the filter table with the same structure, with the additional columns that we need such as current vintage gdp level (current\_vintage) and vintage growth (current\_growth)

filter\_function <- function(v\_year1, v\_quarter1) {  
 cleaned\_data() %>% filter(v\_year == v\_year1,  
 v\_quarter == v\_quarter1) %>%  
   
 mutate(lag\_current\_vintage = lag(current\_vintage,1),  
 log\_lag\_current\_vintage = log(lag\_current\_vintage),  
 current\_growth = 400\*(log\_current\_vintage - log\_lag\_current\_vintage))  
   
 }

Due to missing data from earlier years in some vintages, we begin all analyses from 1965, which is also the first available vintage in our dataset. However, this introduces a challenge: if a user selects a forecast starting point close to 1965, the model will have limited historical data to train on. This lack of training data may negatively impact model performance and forecast reliability. One solution would be to restrict the range of vintages that users can choose to allow a certain level of training size. This would be a potential area for further research to determine the suitable range.

## 3. Model Construction

### 1. Autoregressive (AR)

We choose our baseline model to be the AR model. The AR model serves well as the baseline model because it is a relatively intuitive model. The idea behind the choice is that past growths (lags) of GDP could be used to predict GDP growth in the next period.

First, we define a function to fit an AR model with lag p. Since we are using its own lag for regression, the function will only need to take in the data for the target variable Y, accompany with p for the number of lags and h for forecast horizon.

fitARp=function(Y,p,h){  
   
 #Inputs: Y- predicted variable, p - AR order, h -forecast horizon  
 aux=embed(Y,p+h) #create p lags + forecast horizon shift (=h option)  
 y=aux[,1] # Y variable aligned/adjusted for missing data due to lags  
 X=as.matrix(aux[,-c(1:(ncol(Y)\*h))]) # lags of Y corresponding to forecast horizon   
 if(h==1){   
 X.out=tail(aux,1)[1:ncol(X)] #retrieve last p observations if one-step forecast   
 }else{  
 X.out=aux[,-c(1:(ncol(Y)\*(h-1)))] #delete first (h-1) columns of aux,   
 X.out=tail(X.out,1)[1:ncol(X)] #last p observations to predict T+1   
 }  
   
 model=lm(y~X) #estimate direct h-step AR(p) by OLS   
 coef=coef(model) #extract coefficients  
 #make a forecast using the last few observations: a direct h-step forecast.  
 pred=c(1,X.out)%\*%coef   
   
 return(list("pred"=pred))   
 }

After we have implement our function, we define another function to run our forecast precedure through all the forecasting point

run\_ar <- function(h, p) {  
 # A dataframe to store the result  
 results <- data.frame('year' = numeric(0),  
 'quarter' = numeric(0),  
 'cur\_pred' = numeric(0),  
 'latest\_pred' = numeric(0),  
 'latest\_growth' = numeric(0))  
   
   
 for (i in 1:nrow(forecast\_list())) {  
 item <- forecast\_list()[i,]  
 year\_f <- item$year  
 quarter\_f <- item$quarter   
 # Filter for correct vintage data  
 final\_data <- filter\_function(v\_year1 = year\_f, v\_quarter1 = quarter\_f) %>%  
 select(year, quarter, current\_growth) %>%   
 right\_join(latest\_vintage\_data(), by = c('year', 'quarter')) %>%   
 filter(year >= 1965, year <= year\_f) %>% drop\_na() %>%  
 select(year,quarter, current\_growth, latest\_growth)   
 # Data process to input into model  
 Y\_cur <- as.matrix(final\_data %>% pull(current\_growth))  
 Y\_latest <- as.matrix(final\_data %>% pull(latest\_growth))  
 # Fitting the model   
 model\_cur <- fitARp(Y = Y\_cur, p = p, h = h) # Current Vintage  
 model\_latest <- fitARp(Y = Y\_latest, p = p, h = h) # Latest Vintage  
 pred\_cur <- model\_cur$pred[1]  
 pred\_latest <- model\_latest$pred[1]  
 # Assemble the result  
 result\_df <- data.frame("year" = year\_f, "quarter" = quarter\_f,   
 "cur\_pred" = pred\_cur, "latest\_pred" = pred\_latest)  
 result\_df <- result\_df %>% left\_join(actual\_data(), by = c("year", "quarter")) %>%   
 rename("latest\_growth" ="value")  
 results <- rbind(results, result\_df)  
 }  
 return(results)  
 }

### 2. Autoregressive Distributed Lag (ADL)

### 3. K Nearest Neighbours (KNN)

Our third model is K-Nearest Neighbors (KNN). KNN regression has been widely studied for time series forecasting, and research has consistently shown it to perform well in capturing nonlinear patterns in the data (Lora et al., 2007; Zhang et al., 2017) We choose KNN to explore the self-explanatory power of GDP growth and compare this with parameter tuning models. We utilized the tsfknn package for using KNN regression for time-series forecast. More information on the package can be found here: <https://github.com/franciscomartinezdelrio/tsfknn>

To generate an h-step ahead forecast, we use data only up to time T-h. This approach ensures that our model does not unintentionally peek into the future. We construct lag-based features by using the first four lags of the target variable as autoregressive inputs. Since we are forecasting multiple steps ahead (rather than just the next time point), we also need to specify a multi-step ahead strategy. In our case, we use the Multiple Input Multiple Output (MIMO) method, which allows the model to generate all future predictions in one step. This avoids the compounding error problem commonly encountered in recursive forecasting. Another parameter that is needed to be specified is the cf, the method to aggregrate the target from their nearest neighbors. Some

Then we extract the latest forecast which will be at time T for our purpose. The loop will run through all forecast point in the forecast list, generate forecast and assemble the result for evalulation.

run\_prediction\_knn <- function(h, k, cf) {  
 results = data.frame(year = numeric(0), quarter = numeric(0), cur\_forecast = numeric(0), latest\_forecast = numeric(0))  
 for (i in 1:nrow(forecast\_list())) {  
 item = forecast\_list()[i,]  
 year\_f = item$year  
 quarter\_f = item$quarter  
 data <- filter\_function(v\_year1 = year\_f, v\_quarter1 = quarter\_f)   
 %>% left\_join(latest\_vintage\_data(), by = c("year", "quarter")) %>%  
 select(year, quarter, current\_growth, latest\_growth) %>% filter(year >= 1965)  
 data <- data[1:(nrow(data)-h+1),] # Extract data up until T - h  
 train\_data\_cur <- data %>% select(-latest\_growth)  
 train\_data\_latest <- data %>% select(-current\_growth)  
 ts\_train\_cur <- ts\_transform(train\_data\_cur)   
 ts\_train\_latest <- ts\_transform\_latest(train\_data\_latest)  
 model\_current <- knn\_forecasting(ts\_train\_cur, h = h,   
 lags = 1:4, k = k,   
 msas = "MIMO", cf = cf)  
 cur\_pred <- tail(model\_current$prediction, 1)  
 model\_latest <- knn\_forecasting(ts\_train\_latest, h = h,   
 lags = 1:4, k = k,   
 msas = "MIMO", cf = cf)  
 latest\_pred <- tail(model\_current$prediction, 1)  
 results[i, 'year'] = year\_f  
 results[i, 'quarter'] = quarter\_f  
 results[i, 'cur\_forecast'] = cur\_pred  
 results[i, 'latest\_forecast'] = latest\_pred   
   
 }  
 return(results)  
 }

# III. Application Frontend

The layout of our application will consist of two component: Dataset and Model.

## 1. Dataset Tab

This is where user choose their dataset as well specifying the range of forecast period they are interested in.

### 1.1 Side Panel

As mentioned earlier, our initial plan was to support both quarterly and monthly data. Therefore, we need to render this UI feature on the server that can detect the data frequency and adjust the vintage selection options accordingly.

output$vintage\_period\_ui <- renderUI({  
 req(data\_frequency())  
   
 if (data\_frequency() == "quarterly") {  
 selectInput("vintage\_period", "Select Starting Quarter:", choices = 1:4, selected = 1)  
 } else if (data\_frequency() == "monthly") {  
 selectInput("vintage\_period", "Select Vintage Month:", choices = 1:12, selected = 1)  
 }  
 })

Thus, you may find instead of using calling inputvintage\_period when pulling the starting quarter as inputs. But since we our analysis only conduct on quarterly data, you can assume vintage\_period here is vintage\_quarter

To ensure meaningful forecasting results, we want the forecasting period to include a sufficient number of out-of-sample observations. In this case, we set a minimum of 30 out-of-sample points. Thus, we define a function that can determine the minimum ending period given the chosen starting period.

# Server code  
min\_end\_date\_from\_start <- function(start\_year, start\_quarter, min\_quarters = 30) {  
 start\_year <- as.numeric(start\_year)  
 start\_quarter <- as.numeric(start\_quarter)  
 start\_index <- start\_year \* 4 + (start\_quarter - 1)  
 end\_index <- start\_index + (min\_quarters - 1)  
 max\_index <- 2024 \* 4 + 3  
 end\_index <- min(end\_index, max\_index)  
 end\_year <- end\_index %/% 4  
 end\_quarter <- (end\_index %% 4) + 1  
 list(year = end\_year, quarter = end\_quarter)  
 }

Now we can dynamically update the ending period that is allowed based on user input for the starting period.

# Server code  
observeEvent({  
 input$vintage\_year  
 input$vintage\_period  
 }, {  
 req(input$vintage\_year, input$vintage\_period)  
   
 min\_date <- min\_end\_date\_from\_start(as.numeric(input$vintage\_year), as.numeric(input$vintage\_period), 30)  
   
 # Render end year input  
 output$end\_year\_ui <- renderUI({  
 numericInput(  
 "end\_year",  
 "Select End Year (for forecast)",  
 value = min\_date$year,  
 min = min\_date$year,  
 max = 2024  
 )  
 })  
 # Render end quarter input  
 output$end\_quarter\_ui <- renderUI({  
 selectInput(  
 "end\_quarter",  
 "Select End Quarter (for forecast)",  
 choices = 1:4,  
 selected = min\_date$quarter)  
 })  
 })

# UI code  
numericInput(inputId = "vintage\_year", label = "Select Starting Year",   
 min = 1965, max = 2024, value = 2000),  
   
 uiOutput("vintage\_period\_ui"),  
 uiOutput("end\_year\_ui"),  
 uiOutput("end\_quarter\_ui")

### 1.2 Main Panel

The main panel will preview the vintage data from the starting period as demonstration the structure of the cleaned data as well a visualization of the current vintage and latest vintage data. User can switch to the growth rate by clicking on the Growth button in the setting sidebar

## 2. Model Tab

This is where user can set up their models and features and view the evaluation results.

### 2.1 Side Panel

#### Model Setup

In order to get the result for right model, we assign an id to each model to keep track of the chosen model.

#Server code  
  
output$all\_models\_ui <- renderUI({  
 lapply(model\_ids(), function(i) {  
 wellPanel(  
 h4(paste("Model", i)),  
   
 # Model type selection  
 selectInput(paste0("model\_type\_", i), "Choose a model:",  
 choices = c("KNN", "AR", "ADL")),  
   
 # Placeholder for model-specific parameters  
 uiOutput(paste0("model\_params\_", i)),  
   
 # Delete button  
 actionButton(paste0("delete\_model\_", i), "Delete", class = "btn-danger")  
 )  
 }) %>% tagList()  
 })

We also need to the parameters to be provided to the correct model by assigning it with the corresponding model id.

# Server code  
observe({  
 lapply(model\_ids(), function(i) {  
 output[[paste0("model\_params\_", i)]] <- renderUI({  
 model\_type <- input[[paste0("model\_type\_", i)]]  
 req(model\_type)  
   
 if (model\_type == "KNN") {  
 tagList(textInput(paste0("knn\_k\_",i), "Enter a number or a comma-separated vector:", value = "5"),  
 selectInput(paste0("knn\_cf\_", i), "CF", choices = c("mean","median", "weighted"), selected = "mean"))  
   
 } else if (model\_type == "AR") {  
 tagList(numericInput(paste0("ylag\_ar\_",i), "Enter how many lag for Y", value = 1, min = 1))  
 } else if (model\_type == "ADL") {  
 tagList(numericInput(paste0("ylag\_adl\_",i), "Enter how many lag for Y", value = 2, min = 1))  
 }  
 })  
 })  
 })

#### Feature selection

For feature selection we also assign an id. This is because each feature also have different parameters to tune in. Then we store them into a list for later retrieval

# Server code  
output$all\_features\_ui <- renderUI({  
 lapply(feature\_ids(), function(i) {  
 wellPanel(  
 h4(paste("Feature", i)),  
 textInput(paste0("feature\_id\_", i), label = "Feature Series ID"),  
 numericInput(paste0("feature\_lag\_id\_", i), label = "Lags (optional)", value = 1, min = 1, max = 5),  
 textInput(paste0("feature\_transform\_id\_", i), label = "Transformation (optional)", value = "lin"),  
 actionButton(paste0("delete\_feature\_", i), "Delete", class = "btn-danger")  
 )  
 }) %>% tagList()  
 })

### 2.2 Main Panel

The most important thing for our application is to run the model. We will create placeholders to store the model, prediction result and prediction performance when we loop through the model\_id list and run the corresponding model.

modeltype = reactiveVal(list())  
results = reactiveVal(list())  
performance = reactiveVal(list())

For AR model and KNN model, since both model does not required additional features to be tune in, we only need to retrieve the parameters for each model and return the result. An example for retrieving the result for the KNN model can demonstrated as follow

# Server code   
  
if (model\_type == "KNN") {  
 k\_input <- input[[paste0("knn\_k\_", i)]]  
 k <- as.numeric(unlist(strsplit(as.character(k\_input), ",")))  
 cf <- input[[paste0('knn\_cf\_', i)]]  
   
 knn\_result\_df <- run\_prediction\_knn(h = h, k = k, cf = cf) %>%   
 left\_join(latest\_vintage\_data(), by = c('year', 'quarter')) %>%  
 select(year,quarter, cur\_forecast, latest\_forecast, latest\_growth) %>%  
 rename("cur\_pred" = "cur\_forecast",  
 "latest\_pred" = "latest\_forecast")  
   
   
 knn\_performance\_df <- knn\_result\_df %>%   
 summarize(cur\_mae = round(mean(abs(cur\_pred - latest\_growth)),2),  
 cur\_rmse = round(sqrt(mean((cur\_pred - latest\_growth)^2)),2),  
 latest\_mae = round(mean(abs(latest\_pred - latest\_growth)),2),  
 latest\_rmse = round(sqrt(mean((latest\_pred - latest\_growth)^2)),2))   
 model\_lst <- append(model\_lst, list(model\_type))  
 res\_lst <- append(res\_lst, list(knn\_result\_df))  
 per\_lst <- append(per\_lst, list(knn\_performance\_df))

For ADL model, we will need to retrieve the feature id as well its lags and transformation method.

# Server code  
else if (model\_type == "ADL") {  
 ylag\_input\_adl <- input[[paste0("ylag\_adl\_", i)]]  
 if (length(feature\_series\_ids()) == 0) {  
 showNotification("Please add a feature for ADL or use AR model instead."  
 , type = "error")  
 return()  
 }  
 invalid\_ids <- feature\_series\_ids()[!sapply(feature\_series\_ids(),   
 is\_valid\_series)]  
   
 # Warning when user input an invalid feature id  
 if (length(invalid\_ids) > 0) {  
 showNotification(paste("The following series IDs are invalid:",   
 paste(invalid\_ids, collapse = ", ")))  
 return()   
 }  
 features <- c()   
 feature\_lags <- c()   
 feature\_transforms <- c()  
 for (feature in feature\_series\_ids()) {  
 features <- c(features,feature)  
 }  
   
 for (feature\_lag in feature\_lag\_series()) {  
 feature\_lags <- c(feature\_lags, feature\_lag)  
 }  
 for (feature\_transform in feature\_transform\_series()) {  
 feature\_transforms <- c(feature\_transforms, feature\_transform)  
 }  
   
 adl\_result\_df <- run\_adl\_model(h = h, features = features,   
 lag\_y = ylag\_input\_adl,   
 lags = feature\_lags,  
 units = feature\_transforms) %>%   
 rename("latest\_growth" = "value")   
 adl\_per\_df <- adl\_result\_df %>%   
 summarize(cur\_mae = round(mean(abs(cur\_pred - latest\_growth)),2),  
 latest\_mae = round(mean(abs(latest\_pred - latest\_growth)),2),  
 cur\_rmse = round(sqrt(mean((cur\_pred - latest\_growth)^2)),2),   
 latest\_rmse =round(sqrt(mean((latest\_pred - latest\_growth)^2)),2))   
   
   
 model\_lst <- append(model\_lst, list(model\_type))  
 res\_lst <- append(res\_lst, list(adl\_result\_df))  
 per\_lst <- append(per\_lst, list(adl\_per\_df))  
   
 }

Now that we have all the results we need, the remaining work is to output to the correct id Here we have the Forecast tab (for prediction result), Comparison tab (for compare the performance between models) and a Visualization tab (for displaying the prediction with the real gdp growth.)

# IV. Future improvement

We want to highligh two improvements that we want to address for our application ## Development Tool

One key limitation of our current application is the limited range of forecasting models available, especially as forecasting techniques continue to evolve rapidly. Rather than attempting to include every possible model, our next step is to introduce a Development Tab, allowing users to integrate their own models by following a simple input-output structure.

## Forecast

Additionally, while model evaluation is important, the ultimate goal of any forecasting model is to generate forecasts. Therefore, we plan to extend the workflow to include actual out-of-sample forecasting after evaluation, providing users with more actionable insights.

# V. Conclusion

We present a simple tool to compare performance of different forecasting model to quantify the impact of data revision on our forecasting model. I We want to highlight the importance of evaluating models using both real-time and latest vintage data to ensure a more realistic assessment of how models perform under conditions faced by actual forecasters at the time of prediction