**Arrogance is not a good quality for your models.**

It’s a rarely acknowledged fact that models data scientists produce are often not sufficiently robust or fault-tolerant to actually be put into production. Sure, you can trust your predictions when the input data is similar to your training and holdout data. Any data scientist can build a model object, pass in data in the same format as the training data, and get seemingly reliable predictions.

In the real world, things are always messier than you expect. When data scientists throw models over the fence, ML engineers or the IT department often have to rebuild the models with guardrails to ensure they’re useful in production. As data scientists, we should do better—and do away with the arrogance of assuming our models will function as we intend in every production scenario. We should never say, “Here’s my Jupyter notebook; my work is done!” At the very least, we should start by documenting the expected behavior of incoming variables to help ML engineers write runtime tests more easily. Even better, I recommend adding a layer of unit testing to adjust model predictions.

In this article, I will walk you through a simple error-handling example using R with logical conditions wrapping a model’s predict function. Python users are likely familiar with assert, try except and the usual logical operators to accomplish many of the same things covered in this article. This article uses R as an example because many data scientists using R don’t expose their models as endpoints for production, which means these model “humility” aspects may be new to them.

In the end, the goal is to add a layer of protection to your model to enforce expected behaviors so that it can withstand outliers, be fault-tolerant, and in some cases, override a prediction to a safe value.  
These runtime tests and coded guardrails help make models as safe for production as a POST request within an opencpu server. Including these additional prediction behaviors within your model function will build trust among stakeholders that your model isn’t behaving arrogantly, and will deliver value despite outlier or unexpected inputs.

**Example set-up**

In this example, you will [use this small sample dataset](https://s3.amazonaws.com/assets.datacamp.com/email/other/final_Small_Customer_Data.csv?tap_a=5644-dce66f&tap_s=10907-287229) to build a customer propensity model. This fake data has the results of a marketing campaign for a car loan offer. The input variables include current car make and recent savings account balance. Our classification model will learn which car makes and account balances contribute to accepting the marketing offer. Of course, in the real world you would have more data and follow stricter data science practices like partitioning—but in this example, we’ll take some shortcuts since we’re focusing on the prediction layer.

The [rpart] library is used for recursive partitioning to construct our decision tree. Similarly the [rpart.plot](https://www.rdocumentation.org/packages/rpart.plot?tap_a=5644-dce66f&tap_s=10907-287229) library will help us quickly construct a decent-looking visual of our tree. Next we use [yardstick](https://www.rdocumentation.org/packages/yardstick?tap_a=5644-dce66f&tap_s=10907-287229) to easily get model metrics and [ggplot2](https://www.rdocumentation.org/packages/ggplot2?tap_a=5644-dce66f&tap_s=10907-287229) to construct a mosaic plot. The code below simply loads the data with [read.csv()](https://www.rdocumentation.org/packages/utils/versions/3.6.2/topics/read.table?tap_a=5644-dce66f&tap_s=10907-287229) and examines the first six rows with [head()](https://www.rdocumentation.org/packages/utils/versions/3.6.2/topics/head?tap_a=5644-dce66f&tap_s=10907-287229) so you get a sense of the inputs.

# Libs

library(rpart)

library(rpart.plot)

library(yardstick)

library(ggplot2)

# Read in the data

fakeCustomers <- read.csv('final\_Small\_Customer\_Data.csv')

# EDA

head(fakeCustomers)

**Let’s build a simple decision tree**

Now we apply the rpart() function to construct our decision tree. Since you’re accepting all default model parameters, you only need to pass in the model formula Y\_AcceptedOffer ~ . and the data to make the tree. However, using the period (Y\_AcceptedOffer~.,) in the model formula adds risk to your model’s behavior. Suppose later the underlying training data changes to include additional columns. By using the period the model will simply inherit all columns not defined as the Y-variable. So if you rebuild the model by sourcing this code with data that has changed without explicitly declaring the x-variables, you invite target leakage or overfitting without even knowing it. Thus, its often a good idea to declare x-variables explicitly in the formula. In the end, the resulting fit object is a model which we do not want to simply pass to the IT department. Let’s also define a safe model response when fit gets an unknown value!

# Fit the model

fit <- rpart(Y\_AcceptedOffer~., fakeCustomers)

**Make some predictions**

Let’s make sure our model functions as expected with perfect inputs. At this point in your workflow, you should be assessing model performance against a training and validation set. Here we use predict() on the original data, examine a portion of it with tail(), then construct a simple confusion matrix. Finally, you create a confusion matrix with table() and then nest yardstick’s conf\_mat() inside summary() to get 13 model metrics including accuracy. Keep in mind marketers don’t have unlimited budgets, so you should care more about accuracy within the top 1% or 5% of prospects rather than just accuracy alone.

# Get predictions

pred <- predict(fit, fakeCustomers, type ='class')

# Examine

results <- data.frame(preds = pred,

actuals = fakeCustomers$Y\_AcceptedOffer)

tail(results, 10)

# Simple Confusion Matrix

(confMat <-table(results$preds, results$actuals))

# Obtain model metrics

summary(conf\_mat(confMat))

**Visual Inspection**

In addition to numeric KPI you can inspect the confusion matrix visually with a mosaic plot. In this example, a mosaic plot will have rectangles representing each section of the confusion matrix such as true positives and false positives. The area of each rectangle corresponds to the value from the confusion matrix. This view lets you easily understand how balanced your class assignments are compared to actuals. The code below nests the original confusion matrix in conf\_mat and ggplot2’s autoplot function to create a basic mosaic plot.

autoplot(conf\_mat(confMat))

One benefit of using a simple model is that you can interrogate the model’s behavior. For decision trees you can use rpart.plot() function to visualize the result. This plot will let you understand the split values and importance of variables in each node.

rpart.plot(fit, roundint = F)

**All good right? Not so fast.**

Don’t send this model code to IT and expect a warm response! Sure, it works just fine with these fake prospects—because they are exactly like the training data. Even in normal model building, you usually pass in a partition with similar distributions and certainly the same factor levels. But in reality, data integrity and other factors can be issues with the real incoming data—**and they can break your model.**

fakeNew <- fakeCustomers[c(6:8),]

fakeNew

# Make a prediction

predict(fit, fakeNew, type = 'prob')

**Add a layer of protection for your predictions.**

In this section, you explore what happens when the make of a car is changed from Lexus to lexus. Data entry errors and mis-keys happen all the time because people are involved. Mis-keying factors and transposing numerical inputs often break models in production, as you’ll see if you run predict(fit, fakeRecord) below.

Error: factor carMake has new level lexus

# Entry Form Error

fakeRecord <- fakeNew[1,]

fakeRecord[,2] <- as.factor('lexus')

# Uh-Oh!; Error: factor carMake has new level lexus

#predict(fit, fakeRecord)

**Adding a humble layer to your model**

Let’s add a protective prediction layer by checking the inputs all make sense and if so then call predict(). In this code, you write a wrapper function called humblePredict() that accepts the new observations to be scored. Within a for-loop the function checks:

1. That each row is part of a dataframe using is.data.frame
2. That the dataframe’s columns match the model training formula using the match %in% operator.
3. That the value of the observation in the carMake column is an expected level from model training data. This is another match operator call with %in%
4. Finally, that the 'RecentBalance' column is a numeric value using the is.numeric function.

If all four of these logical conditions are met then the if statement simply calls predict() as usual. However the logical conditions occur within an if-else statement. Therefore if any of these conditions returns a FALSE, then the false code block executes. In this example the default response is the “safe” model response of “DidNotAccept”. This level is safe because it means the company wouldn’t spend money marketing to this potential customer. Of course in your own work, you could have a more explicit error, use a different model or simply return the average Y value from your training set. The point is that you have complete control of the false code behavior and should ensure that your model has guardrails corresponding to the business need. This type of function wrapping helps you dictate how your model should behave with bad inputs. Do you want an error, a safe value, NA or other output when a model is confronted with bad inputs?

humblePredict <- function(x){

classifications <- list()

for(i in 1:nrow(x)){

if(is.data.frame(x[i,]) == T &

all(all.vars(formula(fit)[-2]) %in% names(x[i,])) == T &

x[i,grep('carMake',names(x[i,]))] %in%

unlist(attributes(fit)$xlevels) == T &

is.numeric(x[i,grep('RecentBalance', names(x[i,]))])==T){

response <- predict(fit, x, type = 'class')

classifications[[i]] <- response

} else {

response <- 'DidNotAccept'

classifications[[i]] <- response

}

}

return(unlist(classifications))

}

humblePredict(fakeRecord)

This is just the tip of the iceberg for making models more robust in production. You can write code within humblePredict to change outlier numeric inputs or change factor levels to the most frequent if a level is unknown. If you want to learn more, start with both the testthat() and assertive() libraries for unit testing and run-time testing respectively. **No model should be sent to IT without assertions or at least documentation for safe behaviors.**