Data science tools are getting better and better, which is improving the predictive performance of machine learning models in business. With new, high-performance tools like, ***H2O*** for automated machine learning and ***Keras*** for deep learning, the performance of models are increasing tremendously. **There’s one catch: Complex models are unexplainable… that is until LIME came along!** ***LIME***, which stands for Local Interpretable Model-agnostic Explanations, has opened the doors to black-box (complex, high-performance, but unexplainable) models in business applications! **Explanations are MORE CRITICAL to the business than PERFORMANCE.** Think about it. What good is a high performance model that predicts employee attrition if we can’t tell what features are causing people to quit? We need explanations to improve business decision making. Not just performance.

*Explanations are* ***MORE CRITICAL*** *to the business than* ***PERFORMANCE****. Think about it. What good is a high performance model that predicts employee attrition if we can’t tell what features are causing people to quit? We need explanations to improve business decision making. Not just performance.*

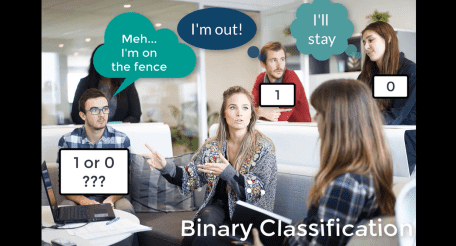
In this Machine Learning Tutorial, We show us how to use LIME for machine learning interpretability on a **Human Resources Employee Turnover Problem**, specifically showing the value of developing interpretablity visualizations. He shows us options for ***Global Importance*** and compares it to LIME for ***Local Importance***. We use machine learning R packages h2o, caret, and ranger in the tutorial, showcasing how to use lime for local explanations. Let’s get started!

**LIME: A Secret Weapon For ROI-Driven Data Science**

Introduction by Matt Dancho, Founder of Business Science

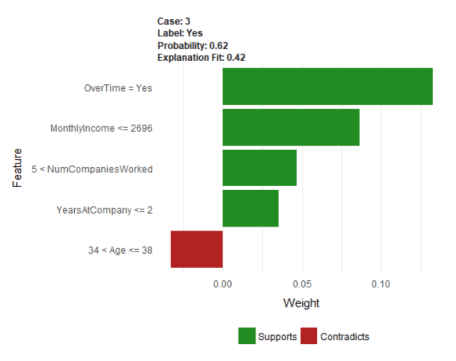
***Business success*** is dependent on the ability for managers, process stakeholders, and key decision makers to make the right decisions often using data to understand what’s going on. This is where machine learning can help. Machine learning can analyze vast amounts of data, creating highly predictive models that tell managers key information such as how likely someone is likely to leave an organization. However, machine learning alone is not enough. Business leaders require explanations so they can determine adjustments that will improve results. These explanations require a different tool: ***LIME***. Let’s find out why **LIME is truly a secret weapon for ROI-driven data science**!

**In the HR Employee Attrition example discussed in this article, the machine learning model predicts the probability of someone leaving the company.** This probability is then converted to a prediction of either leave or stay through a process called ***Binary Classification***. However, this doesn’t solve the main objective, which is to make better decisions. It only tells us if someone is a high flight risk (i.e. has high attrition probability).



Employee Attrition: Machine Learning Predicts Which Employees Are Likely To Leave

**How do we change decision making and therefore improve? It comes down to levers and probability**. Machine learning tells us which employees are highest risk and therefore ***high probability***. We can hone in on these individuals, but we need a different tool to understand why an individual is leaving. This is where LIME comes into play. LIME uncovers the ***levers*** or features we can control to make business improvements.



LIME: Uncovers Levers or Features We Can Control

In our HR Employee Attrition Example, LIME detects “Over Time” (lever) as a key feature that *supports* employee turnover. We can *control* the “Over Time” feature by implementing a “limited-overtime” or “no-overtime” policy.



Analyzing A Policy Change: Targeting Employees With Over Time

Toggling the “OverTime” feature to “No” enables calculating an expected value or benefit of reducing overtime by implementing a new OT policy. For the individual employee, a expected savings results. When applied to the entire organization, this process of *adjusting levers* can result in impactful policy changes that **save the organization millions per year and generate ROI**.



Adjusting The Over Time Results In Expected Savings

**Interested in Learning LIME While Solving A Real-World Churn Problem?**

If you want to solve this real-world employee churn problem developing models with **H2O Automated Machine Learning**, using **LIME For Black-Box ML Model Explanation**, and **analyzing the impact of a policy change through optimization and sensitivity analysis**. You’ll learn ***ROI-Driven Data Science***, implementing the tools (H2O + LIME).

**Learning Trajectory**

Now that we have a flavor for what LIME does, let’s get on with learning how to use it! In this machine learning tutorial, you will learn:

* Global Interpretation: How to perform global intepretation with vip and pdp and how global differs from local with LIME
* Local Interpretation: How To Perform Local Interpretation with lime using models developed with h2o, caret, and ranger
* Visualizing LIME Results: Using plot\_features() and plot\_explanations()
* Tuning LIME: Improving explanation accuracy
* Integrating Models: Working with unsupported models to get LIME integration

Machine learning (ML) models are often considered “black boxes” due to their complex inner-workings. More advanced ML models such as random forests, gradient boosting machines (GBM), artificial neural networks (ANN), among others are typically more accurate for predicting nonlinear, faint, or rare phenomena. Unfortunately, more accuracy often comes at the expense of interpretability, and interpretability is crucial for business adoption, model documentation, regulatory oversight, and human acceptance and trust. Luckily, several advancements have been made to aid in interpreting ML models.

Moreover, it’s often important to understand the ML model that you’ve trained on a global scale, and also to zoom into local regions of your data or your predictions and derive local explanations. **Global interpretations** help us understand the inputs and their entire modeled relationship with the prediction target, but global interpretations can be highly approximate in some cases. **Local interpretations** help us understand model predictions for a single row of data or a group of similar rows.

This post demonstrates how to use the lime package to perform local interpretations of ML models. This will not focus on the theoretical and mathematical underpinnings but, rather, on the practical application of using lime. 1

## Libraries

This tutorial leverages the following packages.

# required packages

# install vip from github repo: devtools::install\_github("koalaverse/vip")

**library**(lime) # ML local interpretation

**library**(vip) # ML global interpretation

**library**(pdp) # ML global interpretation

**library**(ggplot2) # visualization pkg leveraged by above packages

**library**(caret) # ML model building

**library**(h2o) # ML model building

# other useful packages

**library**(tidyverse) # Use tibble, dplyr

**library**(rsample) # Get HR Data via rsample::attrition

**library**(gridExtra) # Plot multiple lime plots on one graph

# initialize h2o

h2o.init()

## Connection successful!

##

## R is connected to the H2O cluster:

## H2O cluster uptime: 3 minutes 17 seconds

## H2O cluster timezone: America/New\_York

## H2O data parsing timezone: UTC

## H2O cluster version: 3.19.0.4208

## H2O cluster version age: 4 months and 6 days !!!

## H2O cluster name: H2O\_started\_from\_R\_mdancho\_zbl980

## H2O cluster total nodes: 1

## H2O cluster total memory: 3.50 GB

## H2O cluster total cores: 4

## H2O cluster allowed cores: 4

## H2O cluster healthy: TRUE

## H2O Connection ip: localhost

## H2O Connection port: 54321

## H2O Connection proxy: NA

## H2O Internal Security: FALSE

## H2O API Extensions: Algos, AutoML, Core V3, Core V4

## R Version: R version 3.4.4 (2018-03-15)

h2o.no\_progress()

To demonstrate model visualization techniques we’ll use the employee attrition data that has been included in the rsample package. This demonstrates a binary classification problem (“Yes” vs. “No”) but the same process that you’ll observe can be used for a regression problem. Note: I force ordered factors to be unordered as h2o does not support ordered categorical variables.

For this exemplar I retain most of the observations in the training data sets and retain 5 observations in the local\_obs set. These 5 observations are going to be treated as new observations that we wish to understand **why** the particular predicted response was made.

# create data sets

df <- rsample::attrition %>%

mutate\_if(is.ordered, factor, ordered = FALSE) %>%

mutate(Attrition = factor(Attrition, levels = c("Yes", "No")))

index <- 1:5

train\_obs <- df[-index, ]

local\_obs <- df[index, ]

# create h2o objects for modeling

y <- "Attrition"

x <- setdiff(names(train\_obs), y)

train\_obs.h2o <- as.h2o(train\_obs)

local\_obs.h2o <- as.h2o(local\_obs)

We will explore how to visualize a few of the more popular machine learning algorithms and packages in R. For brevity I train default models and do not emphasize hyperparameter tuning. The following produces:

* Random forest model using ranger via the caret package
* Random forest model using h2o
* Elastic net model using h2o
* GBM model using h2o
* Random forest model using ranger directly

# Create Random Forest model with ranger via caret

fit.caret <- train(

Attrition ~ .,

data = train\_obs,

method = 'ranger',

trControl = trainControl(method = "cv", number = 5, classProbs = TRUE),

tuneLength = 1,

importance = 'impurity'

)

# create h2o models

h2o\_rf <- h2o.randomForest(x, y, training\_frame = train\_obs.h2o)

h2o\_glm <- h2o.glm(x, y, training\_frame = train\_obs.h2o, family = "binomial")

h2o\_gbm <- h2o.gbm(x, y, training\_frame = train\_obs.h2o)

# ranger model --> model type not built in to LIME

fit.ranger <- ranger::ranger(

Attrition ~ .,

data = train\_obs,

importance = 'impurity',

probability = TRUE

)

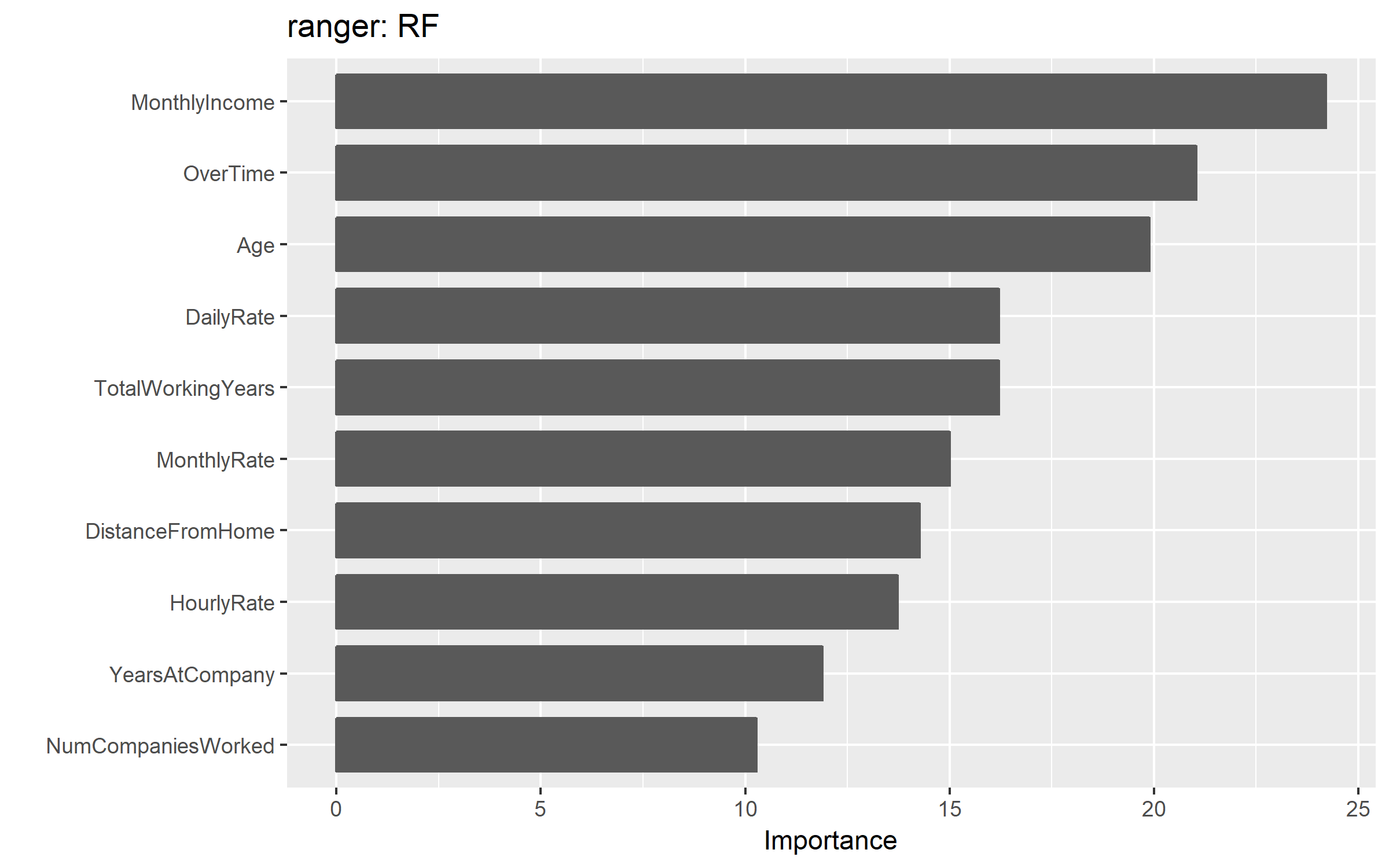
## Global Interpretation

The most common ways of obtaining global interpretation is through:

* variable importance measures
* partial dependence plots

Variable importance quantifies the global contribution of each input variable to the predictions of a machine learning model. Variable importance measures rarely give insight into the average direction that a variable affects a response function. They simply state the magnitude of a variable’s relationship with the response as compared to other variables used in the model. For example, the ranger random forest model identified monthly income, overtime, and age as the top 3 variables impacting the objective function.

vip(fit.ranger) + ggtitle("ranger: RF")

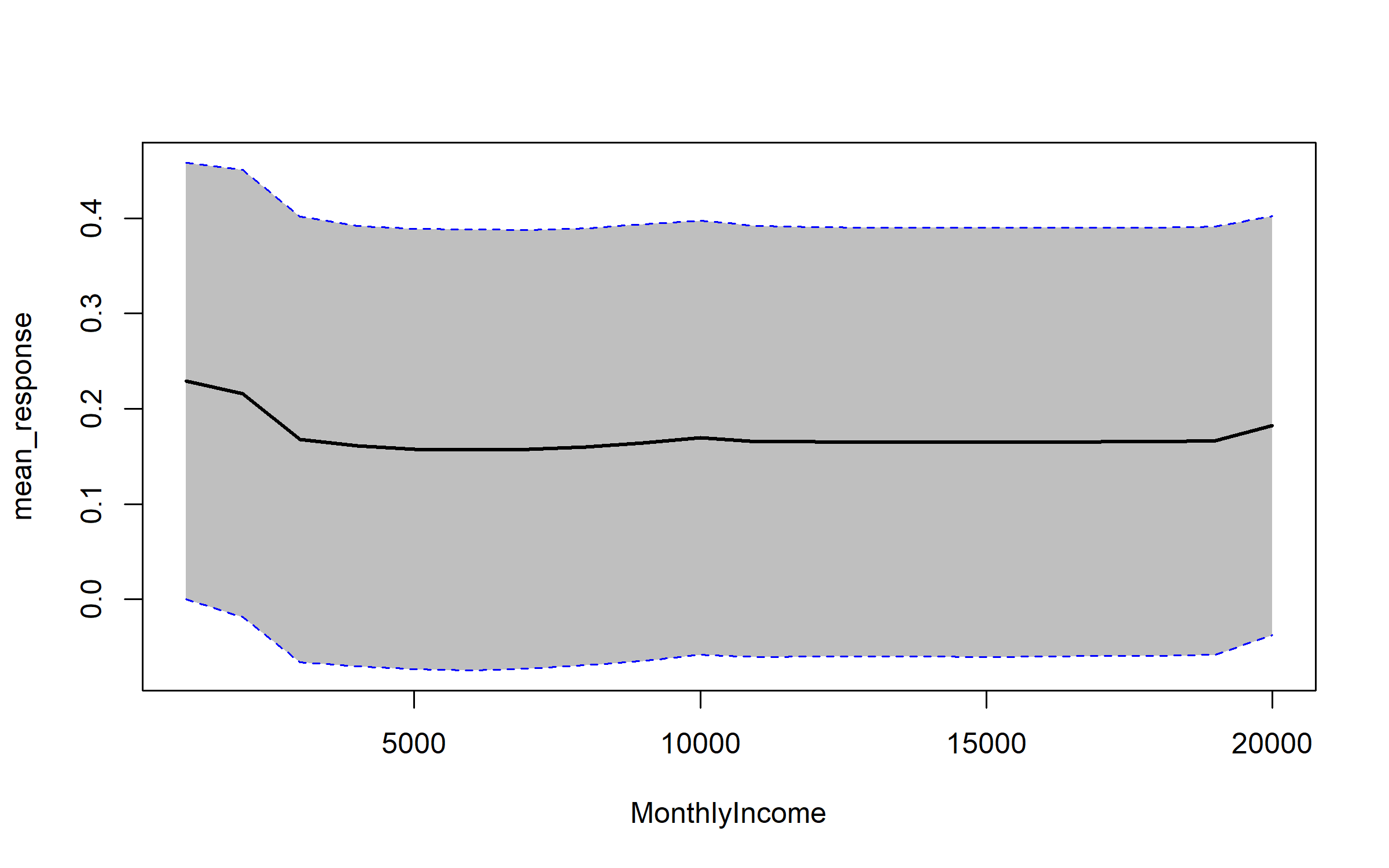


After the most globally relevant variables have been identified, the next step is to attempt to understand how the response variable changes based on these variables. For this we can use partial dependence plots (PDPs) and individual conditional expectation (ICE) curves. These techniques plot the change in the predicted value as specified feature(s) vary over their marginal distribution. Consequently, we can gain some local understanding how the reponse variable changes across the distribution of a particular variable but this still only provides a global understanding of this relationships across all observed data.

For example, if we plot the PDP of the monthly income variable we see that the probability of an employee attriting decreases, on average, as their monthly income approaches $5,000 and then remains relatively flat.

# built-in PDP support in H2O

h2o.partialPlot(h2o\_rf, data = train\_obs.h2o, cols = "MonthlyIncome")



## PartialDependence: Partial Dependence Plot of model DRF\_model\_R\_1529928665020\_106 on column 'MonthlyIncome'

## MonthlyIncome mean\_response stddev\_response

## 1 1009.000000 0.229433 0.229043

## 2 2008.473684 0.216096 0.234960

## 3 3007.947368 0.167686 0.234022

## 4 4007.421053 0.161024 0.231287

## 5 5006.894737 0.157775 0.231151

## 6 6006.368421 0.156628 0.231455

## 7 7005.842105 0.157734 0.230267

## 8 8005.315789 0.160137 0.229286

## 9 9004.789474 0.164437 0.229305

## 10 10004.263158 0.169652 0.227902

## 11 11003.736842 0.165502 0.226240

## 12 12003.210526 0.165297 0.225529

## 13 13002.684211 0.165051 0.225335

## 14 14002.157895 0.165051 0.225335

## 15 15001.631579 0.164983 0.225316

## 16 16001.105263 0.165051 0.225019

## 17 17000.578947 0.165556 0.224890

## 18 18000.052632 0.165556 0.224890

## 19 18999.526316 0.166498 0.224726

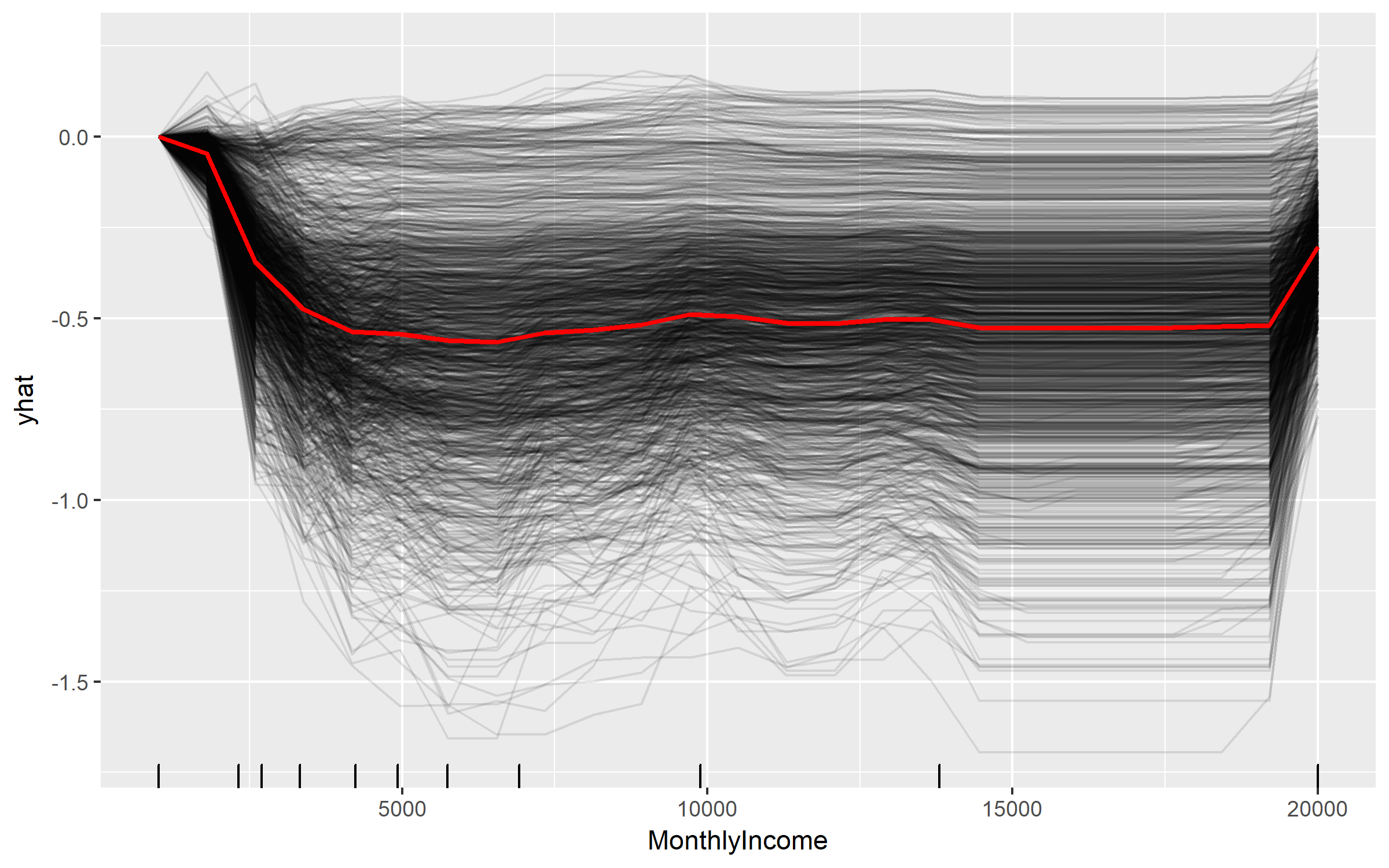
## 20 19999.000000 0.182348 0.219882

We can gain further insight by using centered ICE curves which can help draw out further details. For example, the following ICE curves show a similar trend line as the PDP above but by centering we identify the decrease as monthly income approaches $5,000 followed by an increase in probability of attriting once an employee’s monthly income approaches $20,000. Futhermore, we see some turbulence in the flatlined region between $5-$20K) which means there appears to be certain salary regions where the probability of attriting changes.

fit.ranger %>%

pdp::partial(pred.var = "MonthlyIncome", grid.resolution = 25, ice = TRUE) %>%

autoplot(rug = TRUE, train = train\_obs, alpha = 0.1, center = TRUE)



These visualizations help us to understand our model from a global perspective: identifying the variables with the largest overall impact and the typical influence of a feature on the response variable across all observations. However, what these do not help us understand is given a new observation, what were the most **influential variables that determined the predicted outcome**. Say we obtain information on an employee that makes about $10,000 per month and we need to assess their probabilty of leaving the firm. Although monthly income is the most important variable in our model, it may not be the most influential variable driving this employee to leave. To retain the employee, leadership needs to understand what variables are most influential for that specific employee. This is where lime can help.

## Local Interpretation

Local Interpretable Model-agnostic Explanations (**LIME**) is a visualization technique that helps explain individual predictions. As the name implies, it is model agnostic so it can be applied to any supervised regression or classification model. Behind the workings of LIME lies the assumption that **every complex model is linear on a local scale** and asserting that it is possible to fit a simple model around a single observation that will mimic how the global model behaves at that locality. The simple model can then be used to explain the predictions of the more complex model locally.

The generalized algorithm LIME applies is:

1. Given an observation, **permute** it to create replicated feature data with slight value modifications.
2. Compute **similarity distance measure** between original observation and permuted observations.
3. Apply selected machine learning model to **predict outcomes** of permuted data.
4. **Select m number of features** to best describe predicted outcomes.
5. **Fit a simple model** to the permuted data, explaining the complex model outcome with m features from the permuted data weighted by its similarity to the original observation .
6. Use the resulting **feature weights to explain local behavior**.

Each of these steps will be discussed in further detail as we proceed.

### lime::lime

The application of the LIME algorithm via the lime package is split into two operations: lime::lime() and lime::explain(). The lime::lime() function creates an “explainer” object, which is just a list that contains the machine learning model and the feature distributions for the training data. The feature distributions that it contains includes distribution statistics for each categorical variable level and each continuous variable split into n bins (default is 4 bins). These feature attributes will be used to permute data.

The following creates our lime::lime() object and I change the number to bin our continuous variables into to 5.

explainer\_caret <- lime::lime(train\_obs, fit.caret, n\_bins = 5)

class(explainer\_caret)

## [1] "data\_frame\_explainer" "explainer"

## [3] "list"

summary(explainer\_caret)

## Length Class Mode

## model 24 train list

## bin\_continuous 1 -none- logical

## n\_bins 1 -none- numeric

## quantile\_bins 1 -none- logical

## use\_density 1 -none- logical

## feature\_type 31 -none- character

## bin\_cuts 31 -none- list

## feature\_distribution 31 -none- list

### lime::explain

Once we created our lime objects, we can now perform the generalized LIME algorithm using the lime::explain() function. This function has several options, each providing flexibility in how we perform the generalized algorithm mentioned above.

* x: Contains the one or more single observations you want to create local explanations for. In our case, this includes the 5 observations that I included in the local\_obs data frame. Relates to algorithm step 1.
* explainer: takes the explainer object created by lime::lime(), which will be used to create permuted data. Permutations are sampled from the variable distributions created by the lime::lime() explainer object. Relates to algorithm step 1.
* n\_permutations: The number of permutations to create for each observation in x (default is 5,000 for tabular data). Relates to algorithm step 1.
* dist\_fun: The distance function to use. The default is Gower’s distance but can also use euclidean, manhattan, or any other distance function allowed by ?dist(). To compute similarity distance of permuted observations, categorical features will be recoded based on whether or not they are equal to the actual observation. If continuous features are binned (the default) these features will be recoded based on whether they are in the same bin as the observation. Using the recoded data the distance to the original observation is then calculated based on a user-chosen distance measure. Relates to algorithm step 2.
* kernel\_width: To convert the distance measure to a similarity value, an exponential kernel of a user defined width (defaults to 0.75 times the square root of the number of features) is used. Smaller values restrict the size of the local region. Relates to algorithm step 2.
* n\_features: The number of features to best describe predicted outcomes. Relates to algorithm step 4.
* feature\_select: To select the best n features, lime can use forward selection, ridge regression, lasso, or a tree to select the features. In this example I apply a ridge regression model and select the m features with highest absolute weights. Relates to algorithm step 4.

For classification models we also have two additional features we care about and one of these two arguments must be given:

* labels: Which label do we want to explain? In this example, I want to explain the probability of an observation to attrit (“Yes”).
* n\_labels: The number of labels to explain. With this data I could select n\_labels = 2 to explain the probability of “Yes” and “No” responses.

explanation\_caret <- lime::explain(

x = local\_obs,

explainer = explainer\_caret,

n\_permutations = 5000,

dist\_fun = "gower",

kernel\_width = .75,

n\_features = 5,

feature\_select = "highest\_weights",

labels = "Yes"

)

The explain() function above first creates permutations, then calculates similarities, followed by selecting the m features. Lastly, explain() will then fit a model (algorithm steps 5 & 6). lime applies a ridge regression model with the weighted permuted observations as the simple model. [2] If the model is a regressor, the simple model will predict the output of the complex model directly. If the complex model is a classifier, the simple model will predict the probability of the chosen class(es).

The explain() output is a data frame containing different information on the simple model predictions. Most importantly, for each observation in local\_obs it contains the simple model fit (model\_r2) and the weighted importance (feature\_weight) for each important feature (feature\_desc) that best describes the local relationship.

tibble::glimpse(explanation\_caret)

## Observations: 25

## Variables: 13

## $ model\_type <chr> "classification", "classification", "cla...

## $ case <chr> "1", "1", "1", "1", "1", "2", "2", "2", ...

## $ label <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes"...

## $ label\_prob <dbl> 0.216, 0.216, 0.216, 0.216, 0.216, 0.070...

## $ model\_r2 <dbl> 0.5517840, 0.5517840, 0.5517840, 0.55178...

## $ model\_intercept <dbl> 0.1498396, 0.1498396, 0.1498396, 0.14983...

## $ model\_prediction <dbl> 0.3233514, 0.3233514, 0.3233514, 0.32335...

## $ feature <chr> "OverTime", "MaritalStatus", "BusinessTr...

## $ feature\_value <fct> Yes, Single, Travel\_Rarely, Sales, Very\_...

## $ feature\_weight <dbl> 0.14996805, 0.05656074, -0.03929651, 0.0...

## $ feature\_desc <chr> "OverTime = Yes", "MaritalStatus = Singl...

## $ data <list> [[41, Yes, Travel\_Rarely, 1102, Sales, ...

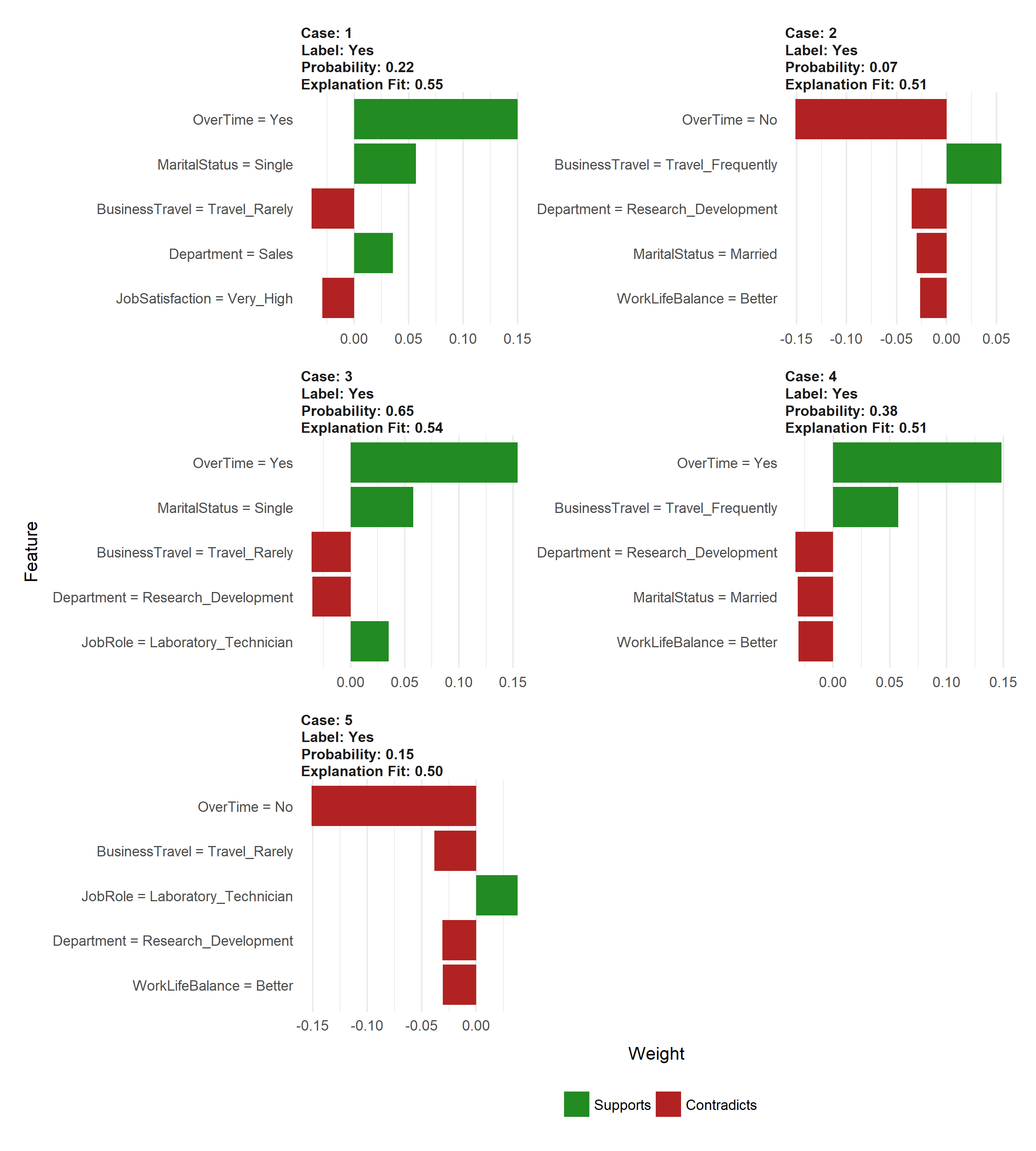
## $ prediction <list> [[0.216, 0.784], [0.216, 0.784], [0.216...

### Visualizing results

However the simplest approach to interpret the results is to visualize them. There are several plotting functions provided by lime but for tabular data we are only concerned with two. The most important of which is plot\_features(). This will create a visualization containing an individual plot for each observation (case 1, 2, …, n) in our local\_obs data frame. Since we specified labels = "Yes" in the explain() function, it will provide the probability of each observation attriting. And since we specified n\_features = 10 it will plot the 10 most influential variables that best explain the linear model in that observations local region and whether the variable is causes an increase in the probability (supports) or a decrease in the probability (contradicts). It also provides us with the model fit for each model (“Explanation Fit: XX”), which allows us to see how well that model explains the local region.

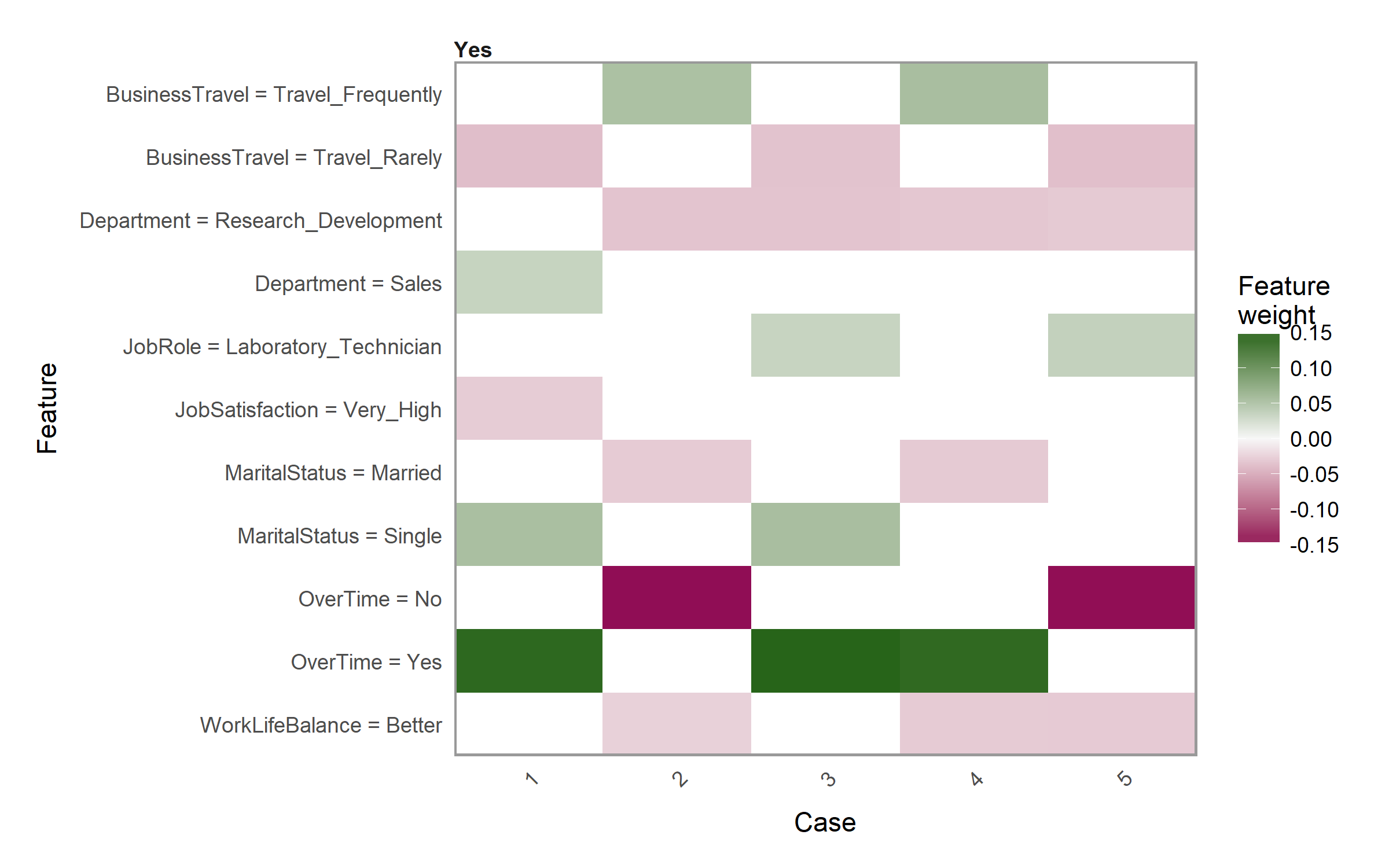
Consequently, we can infer that case 3 has the highest liklihood of attriting out of the 5 observations and the 3 variables that appear to be influencing this high probability include working overtime, being single, and working as a lab tech.

plot\_features(explanation\_caret)



The other plot we can create is a **heatmap** showing how the different variables selected across all the observations influence each case. We use the plot\_explanations() function. This plot becomes useful if you are trying to find common features that influence all observations or if you are performing this analysis across many observations which makes plot\_features() difficult to discern.

plot\_explanations(explanation\_caret)



### Tuning LIME

As you saw in the above plot\_features() plot, the output provides the model fit. In this case the best simple model fit for the given local regions was R^2 = 0.59 for case 3. Considering there are several knobs we can turn when performing the LIME algorithm, we can treat these as tuning parameters to try find the best fit model for the local region. This helps to maximize the amount of trust we can have in the local region explanation.

As an example, the following changes the distance function to use the manhattan distance algorithm, we increase the kernel width substantially to create a larger local region, and we change our feature selection approach to a LARS lasso model. The result is a fairly substantial increase in our explanation fits.

# tune LIME algorithm

explanation\_caret <- lime::explain(

x = local\_obs,

explainer = explainer\_caret,

n\_permutations = 5000,

dist\_fun = "manhattan",

kernel\_width = 3,

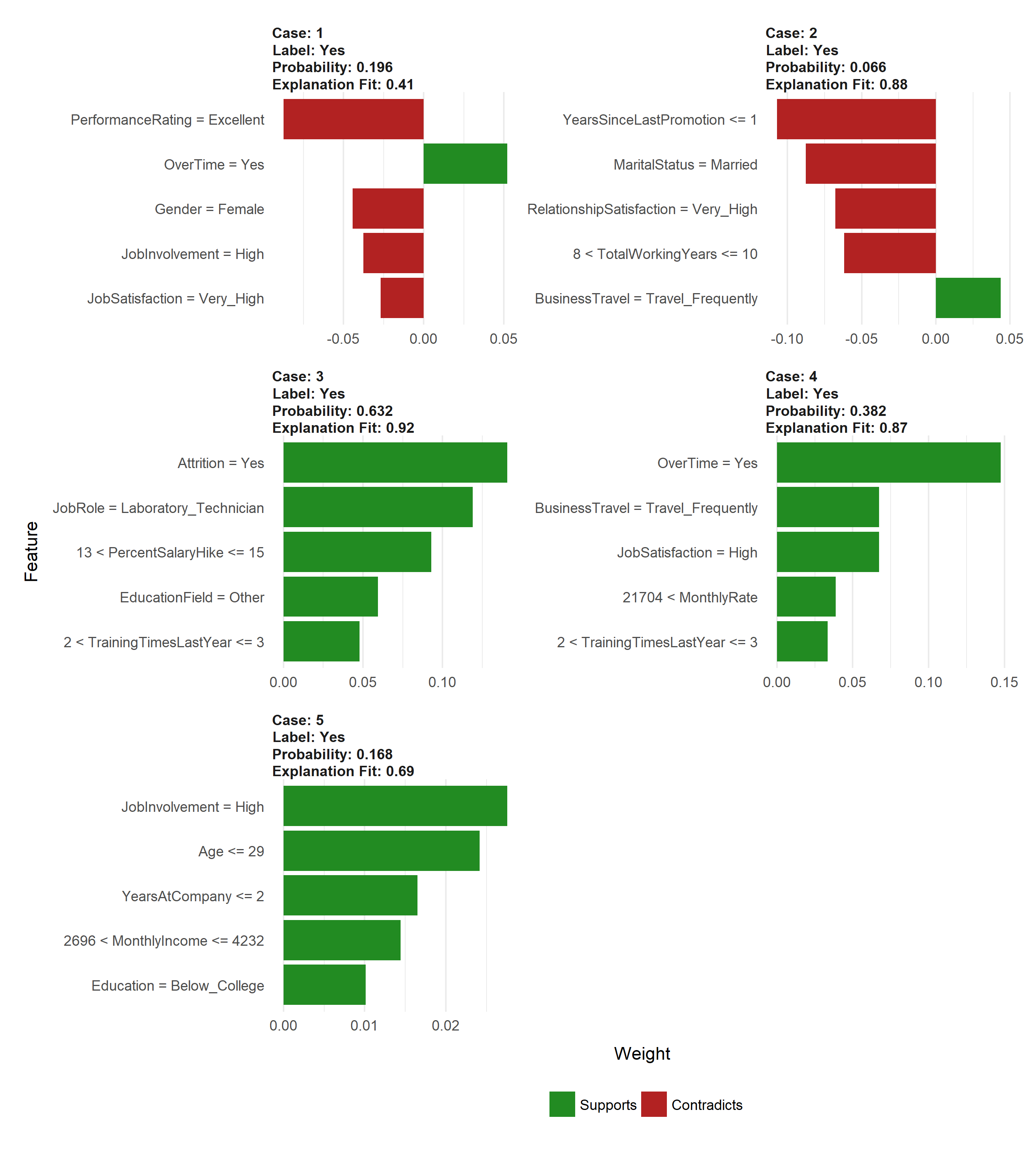
n\_features = 5,

feature\_select = "lasso\_path",

labels = "Yes"

)

plot\_features(explanation\_caret)



### Supported vs Non-support models

Currently, lime supports supervised models produced in caret, mlr, xgboost, h2o, keras, and MASS::lda. Consequently, any supervised models created with these packages will function just fine with lime.

explainer\_h2o\_rf <- lime(train\_obs, h2o\_rf, n\_bins = 5)

explainer\_h2o\_glm <- lime(train\_obs, h2o\_glm, n\_bins = 5)

explainer\_h2o\_gbm <- lime(train\_obs, h2o\_gbm, n\_bins = 5)

explanation\_rf <- lime::explain(local\_obs,

explainer\_h2o\_rf,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

feature\_select = "highest\_weights")

explanation\_glm <- lime::explain(local\_obs,

explainer\_h2o\_glm,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

feature\_select = "highest\_weights")

explanation\_gbm <- lime::explain(local\_obs,

explainer\_h2o\_gbm,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

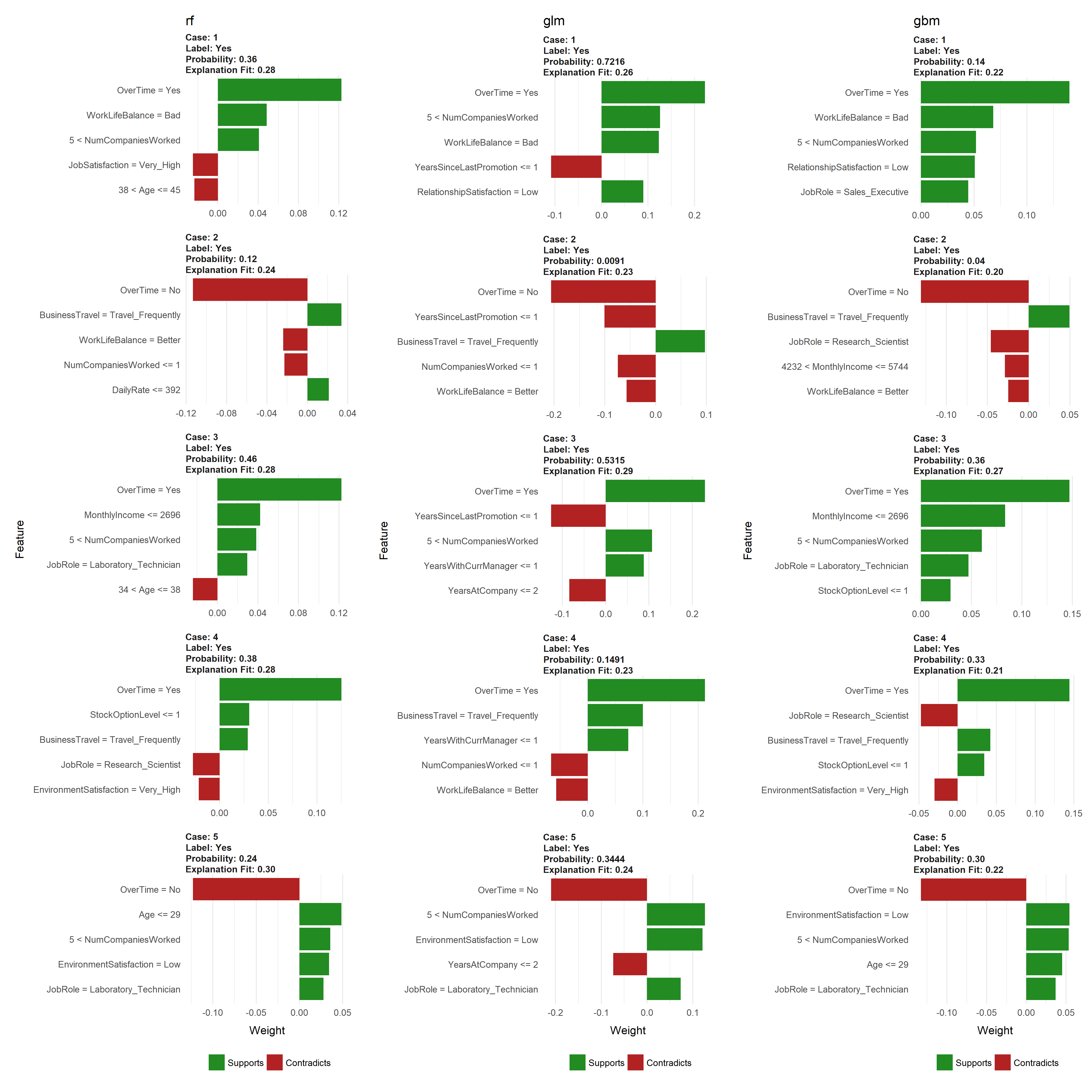
feature\_select = "highest\_weights")

p1 <- plot\_features(explanation\_rf, ncol = 1) + ggtitle("rf")

p2 <- plot\_features(explanation\_glm, ncol = 1) + ggtitle("glm")

p3 <- plot\_features(explanation\_gbm, ncol = 1) + ggtitle("gbm")

gridExtra::grid.arrange(p1, p2, p3, nrow = 1)



However, any models that do not have built in support will produce an error. For example, the model we created directly with ranger is not supported and produces an error.

explainer\_ranger <- lime(train, fit.ranger, n\_bins = 5)

## Error in UseMethod("lime", x): no applicable method for 'lime' applied to an object of class "function"

We can work with this pretty easily by building two functions that make lime compatible with an unsupported package. First, we need to create a model\_type() function that specifies what type of model this unsupported package is using. model\_type() is a lime specific function, we just need to create a ranger specific method. We do this by taking the class name for our ranger object and creating the model\_type.ranger method and simply return the type of model (“classification” for this example).

# get the model class

class(fit.ranger)

## [1] "ranger"

# need to create custom model\_type function

model\_type.ranger <- **function**(x, **...**) {

# Function tells lime() what model type we are dealing with

# 'classification', 'regression', 'survival', 'clustering', 'multilabel', etc

**return**("classification")

}

model\_type(fit.ranger)

## [1] "classification"

We then need to create a predict\_model() method for ranger as well. The output for this function should be a data frame. For a regression problem it should produce a single column data frame with the predicted response and for a classification problem it should create a column containing the probabilities for each categorical class (binary “Yes” “No” in this example).

# need to create custom predict\_model function

predict\_model.ranger <- **function**(x, newdata, **...**) {

# Function performs prediction and returns data frame with Response

pred <- predict(x, newdata)

**return**(as.data.frame(pred$predictions))

}

predict\_model(fit.ranger, newdata = local\_obs)

## Yes No

## 1 0.2915452 0.7084548

## 2 0.0701619 0.9298381

## 3 0.4406563 0.5593437

## 4 0.3465294 0.6534706

## 5 0.2525397 0.7474603

Now that we have those methods developed and in our global environment we can run our lime functions and produce our outputs.3

explainer\_ranger <- lime(train\_obs, fit.ranger, n\_bins = 5)

explanation\_ranger <- lime::explain(local\_obs,

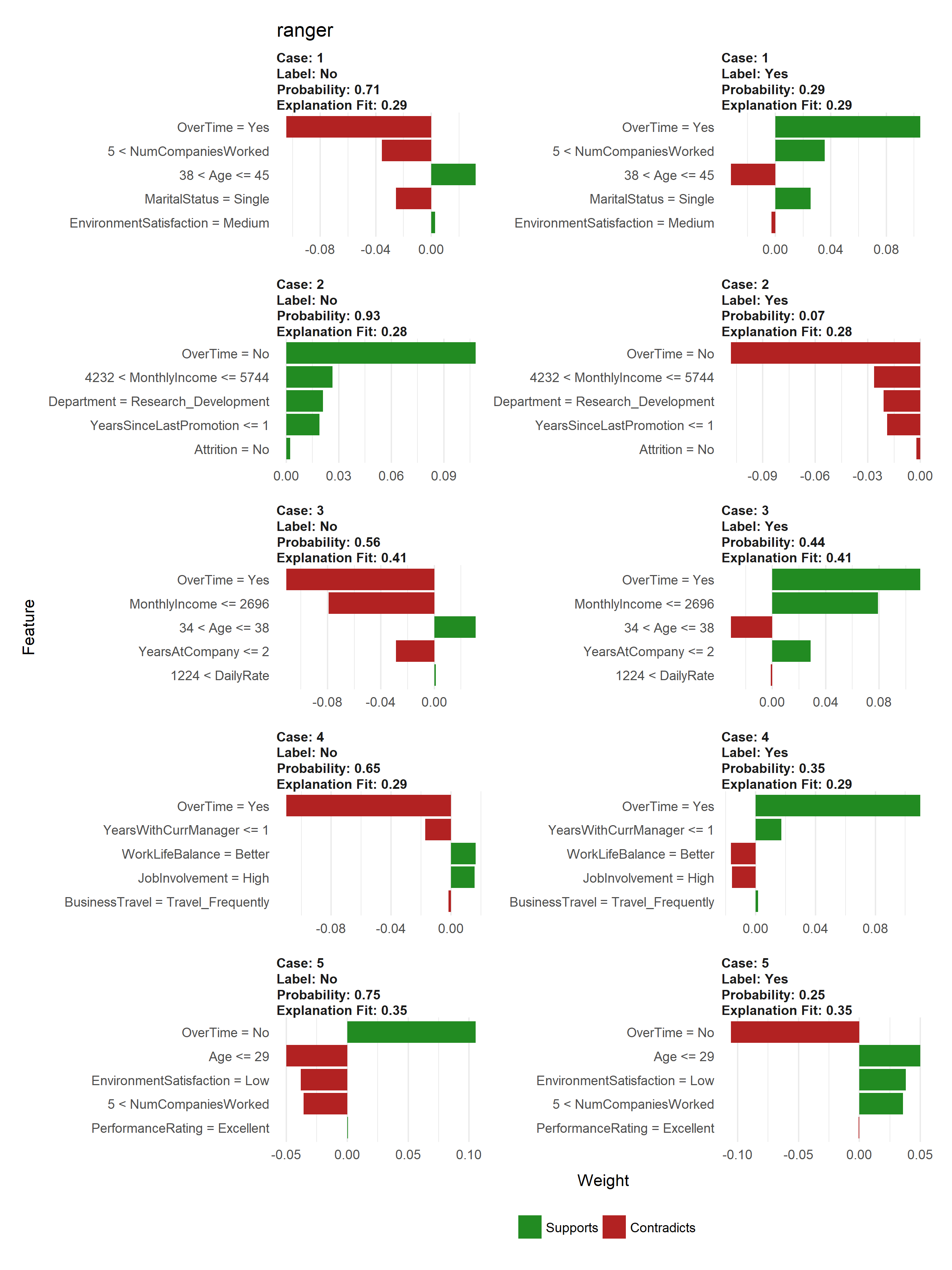
explainer\_ranger,

n\_features = 5,

n\_labels = 2,

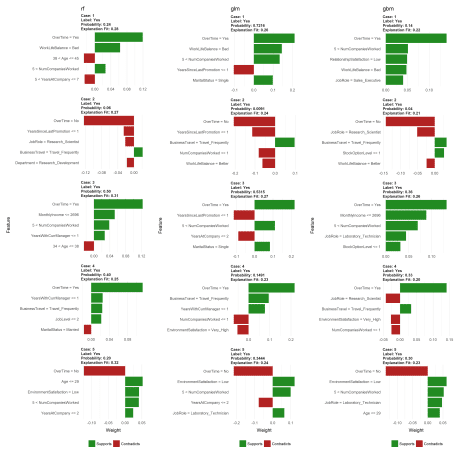
kernel\_width = .1)

plot\_features(explanation\_ranger, ncol = 2) + ggtitle("ranger")



In fact, one of the coolest things you’ll learn is how to create a visualization that compares multiple H2O modeling algorithms that examine employee attrition. This is akin to getting different perspectives for how each of the models view the problem:

* Random Forest (RF)
* Generalized Linear Regression (GLM)
* Gradient Boosted Machine (GBM).



Comparing LIME results of different H2O modeling algorithms

**Machine Learning Tutorial: Visualizing Machine Learning Models with LIME**

By Brad Boehmke, Director of Data Science at 84.51°

Machine learning (ML) models are often considered “black boxes” due to their complex inner-workings. More advanced ML models such as random forests, gradient boosting machines (GBM), artificial neural networks (ANN), among others are typically more accurate for predicting nonlinear, faint, or rare phenomena. Unfortunately, more accuracy often comes at the expense of interpretability, and interpretability is crucial for business adoption, model documentation, regulatory oversight, and human acceptance and trust. Luckily, several advancements have been made to aid in interpreting ML models.

Moreover, it’s often important to understand the ML model that you’ve trained on a global scale, and also to zoom into local regions of your data or your predictions and derive local explanations. ***Global interpretations*** help us understand the inputs and their entire modeled relationship with the prediction target, but global interpretations can be highly approximate in some cases. ***Local interpretations*** help us understand model predictions for a single row of data or a group of similar rows.

This post demonstrates how to use the lime package to perform local interpretations of ML models. This will not focus on the theoretical and mathematical underpinnings but, rather, on the practical application of using lime. [1](http://www.business-science.io/business/2018/06/25/lime-local-feature-interpretation.html#fn:lime_paper)

**Libraries**

This tutorial leverages the following packages.

# required packages

# install vip from github repo: devtools::install\_github("koalaverse/vip")

library(lime) # ML local interpretation

library(vip) # ML global interpretation

library(pdp) # ML global interpretation

library(ggplot2) # visualization pkg leveraged by above packages

library(caret) # ML model building

library(h2o) # ML model building

# other useful packages

library(tidyverse) # Use tibble, dplyr

library(rsample) # Get HR Data via rsample::attrition

library(gridExtra) # Plot multiple lime plots on one graph

# initialize h2o

h2o.init()

## Connection successful!

##

## R is connected to the H2O cluster:

## H2O cluster uptime: 3 minutes 17 seconds

## H2O cluster timezone: America/New\_York

## H2O data parsing timezone: UTC

## H2O cluster version: 3.19.0.4208

## H2O cluster version age: 4 months and 6 days !!!

## H2O cluster name: H2O\_started\_from\_R\_mdancho\_zbl980

## H2O cluster total nodes: 1

## H2O cluster total memory: 3.50 GB

## H2O cluster total cores: 4

## H2O cluster allowed cores: 4

## H2O cluster healthy: TRUE

## H2O Connection ip: localhost

## H2O Connection port: 54321

## H2O Connection proxy: NA

## H2O Internal Security: FALSE

## H2O API Extensions: Algos, AutoML, Core V3, Core V4

## R Version: R version 3.4.4 (2018-03-15)

h2o.no\_progress()

To demonstrate model visualization techniques we’ll use the employee attrition data that has been included in the rsample package. This demonstrates a binary classification problem (“Yes” vs. “No”) but the same process that you’ll observe can be used for a regression problem. Note: I force ordered factors to be unordered as h2o does not support ordered categorical variables.

For this exemplar I retain most of the observations in the training data sets and retain 5 observations in the local\_obs set. These 5 observations are going to be treated as new observations that we wish to understand ***why*** the particular predicted response was made.

# create data sets

df <- rsample::attrition %>%

mutate\_if(is.ordered, factor, ordered = FALSE) %>%

mutate(Attrition = factor(Attrition, levels = c("Yes", "No")))

index <- 1:5

train\_obs <- df[-index, ]

local\_obs <- df[index, ]

# create h2o objects for modeling

y <- "Attrition"

x <- setdiff(names(train\_obs), y)

train\_obs.h2o <- as.h2o(train\_obs)

local\_obs.h2o <- as.h2o(local\_obs)

We will explore how to visualize a few of the more popular machine learning algorithms and packages in R. For brevity I train default models and do not emphasize hyperparameter tuning. The following produces:

* Random forest model using ranger via the caret package
* Random forest model using h2o
* Elastic net model using h2o
* GBM model using h2o
* Random forest model using ranger directly

# Create Random Forest model with ranger via caret

fit.caret <- train(

Attrition ~ .,

data = train\_obs,

method = 'ranger',

trControl = trainControl(method = "cv", number = 5, classProbs = TRUE),

tuneLength = 1,

importance = 'impurity'

)

# create h2o models

h2o\_rf <- h2o.randomForest(x, y, training\_frame = train\_obs.h2o)

h2o\_glm <- h2o.glm(x, y, training\_frame = train\_obs.h2o, family = "binomial")

h2o\_gbm <- h2o.gbm(x, y, training\_frame = train\_obs.h2o)

# ranger model --> model type not built in to LIME

fit.ranger <- ranger::ranger(

Attrition ~ .,

data = train\_obs,

importance = 'impurity',

probability = TRUE

)

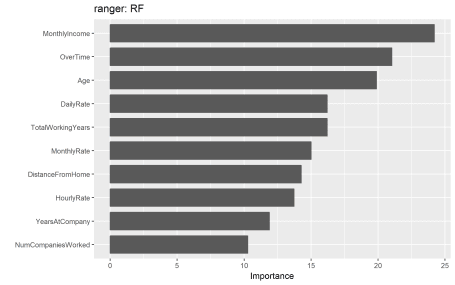
**Global Interpretation**

The most common ways of obtaining global interpretation is through:

* variable importance measures
* partial dependence plots

Variable importance quantifies the global contribution of each input variable to the predictions of a machine learning model. Variable importance measures rarely give insight into the average direction that a variable affects a response function. They simply state the magnitude of a variable’s relationship with the response as compared to other variables used in the model. For example, the ranger random forest model identified monthly income, overtime, and age as the top 3 variables impacting the objective function.

vip(fit.ranger) + ggtitle("ranger: RF")

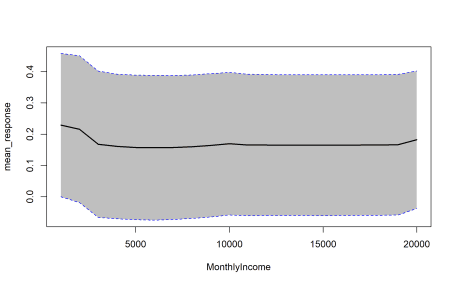


After the most globally relevant variables have been identified, the next step is to attempt to understand how the response variable changes based on these variables. For this we can use partial dependence plots (PDPs) and individual conditional expectation (ICE) curves. These techniques plot the change in the predicted value as specified feature(s) vary over their marginal distribution. Consequently, we can gain some local understanding how the reponse variable changes across the distribution of a particular variable but this still only provides a global understanding of this relationships across all observed data.

For example, if we plot the PDP of the monthly income variable we see that the probability of an employee attriting decreases, on average, as their monthly income approaches $5,000 and then remains relatively flat.

# built-in PDP support in H2O

h2o.partialPlot(h2o\_rf, data = train\_obs.h2o, cols = "MonthlyIncome")



## PartialDependence: Partial Dependence Plot of model DRF\_model\_R\_1529928665020\_106 on column 'MonthlyIncome'

## MonthlyIncome mean\_response stddev\_response

## 1 1009.000000 0.229433 0.229043

## 2 2008.473684 0.216096 0.234960

## 3 3007.947368 0.167686 0.234022

## 4 4007.421053 0.161024 0.231287

## 5 5006.894737 0.157775 0.231151

## 6 6006.368421 0.156628 0.231455

## 7 7005.842105 0.157734 0.230267

## 8 8005.315789 0.160137 0.229286

## 9 9004.789474 0.164437 0.229305

## 10 10004.263158 0.169652 0.227902

## 11 11003.736842 0.165502 0.226240

## 12 12003.210526 0.165297 0.225529

## 13 13002.684211 0.165051 0.225335

## 14 14002.157895 0.165051 0.225335

## 15 15001.631579 0.164983 0.225316

## 16 16001.105263 0.165051 0.225019

## 17 17000.578947 0.165556 0.224890

## 18 18000.052632 0.165556 0.224890

## 19 18999.526316 0.166498 0.224726

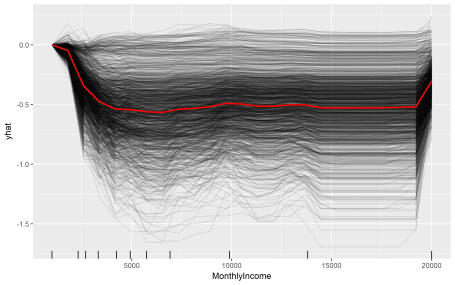
## 20 19999.000000 0.182348 0.219882

We can gain further insight by using centered ICE curves which can help draw out further details. For example, the following ICE curves show a similar trend line as the PDP above but by centering we identify the decrease as monthly income approaches $5,000 followed by an increase in probability of attriting once an employee’s monthly income approaches $20,000. Futhermore, we see some turbulence in the flatlined region between $5-$20K) which means there appears to be certain salary regions where the probability of attriting changes.

fit.ranger %>%

pdp::partial(pred.var = "MonthlyIncome", grid.resolution = 25, ice = TRUE) %>%

autoplot(rug = TRUE, train = train\_obs, alpha = 0.1, center = TRUE)



These visualizations help us to understand our model from a global perspective: identifying the variables with the largest overall impact and the typical influence of a feature on the response variable across all observations. However, what these do not help us understand is given a new observation, what were the most ***influential variables that determined the predicted outcome***. Say we obtain information on an employee that makes about $10,000 per month and we need to assess their probabilty of leaving the firm. Although monthly income is the most important variable in our model, it may not be the most influential variable driving this employee to leave. To retain the employee, leadership needs to understand what variables are most influential for that specific employee. This is where lime can help.

**Local Interpretation**

Local Interpretable Model-agnostic Explanations (**LIME**) is a visualization technique that helps explain individual predictions. As the name implies, it is model agnostic so it can be applied to any supervised regression or classification model. Behind the workings of LIME lies the assumption that ***every complex model is linear on a local scale*** and asserting that it is possible to fit a simple model around a single observation that will mimic how the global model behaves at that locality. The simple model can then be used to explain the predictions of the more complex model locally.

The generalized algorithm LIME applies is:

1. Given an observation, ***permute*** it to create replicated feature data with slight value modifications.
2. Compute ***similarity distance measure*** between original observation and permuted observations.
3. Apply selected machine learning model to ***predict outcomes*** of permuted data.
4. ***Select m number of features*** to best describe predicted outcomes.
5. ***Fit a simple model*** to the permuted data, explaining the complex model outcome with *m* features from the permuted data weighted by its similarity to the original observation .
6. Use the resulting ***feature weights to explain local behavior***.

Each of these steps will be discussed in further detail as we proceed.

**lime::lime**

The application of the LIME algorithm via the lime package is split into two operations: lime::lime() and lime::explain(). The lime::lime() function creates an “explainer” object, which is just a list that contains the machine learning model and the feature distributions for the training data. The feature distributions that it contains includes distribution statistics for each categorical variable level and each continuous variable split into *n* bins (default is 4 bins). These feature attributes will be used to permute data.

The following creates our lime::lime() object and I change the number to bin our continuous variables into to 5.

explainer\_caret <- lime::lime(train\_obs, fit.caret, n\_bins = 5)

class(explainer\_caret)

## [1] "data\_frame\_explainer" "explainer"

## [3] "list"

summary(explainer\_caret)

## Length Class Mode

## model 24 train list

## bin\_continuous 1 -none- logical

## n\_bins 1 -none- numeric

## quantile\_bins 1 -none- logical

## use\_density 1 -none- logical

## feature\_type 31 -none- character

## bin\_cuts 31 -none- list

## feature\_distribution 31 -none- list

**lime::explain**

Once we created our lime objects, we can now perform the generalized LIME algorithm using the lime::explain() function. This function has several options, each providing flexibility in how we perform the generalized algorithm mentioned above.

* x: Contains the one or more single observations you want to create local explanations for. In our case, this includes the 5 observations that I included in the local\_obs data frame. *Relates to algorithm step 1*.
* explainer: takes the explainer object created by lime::lime(), which will be used to create permuted data. Permutations are sampled from the variable distributions created by the lime::lime() explainer object. *Relates to algorithm step 1*.
* n\_permutations: The number of permutations to create for each observation in x (default is 5,000 for tabular data). *Relates to algorithm step 1*.
* dist\_fun: The distance function to use. The default is Gower’s distance but can also use euclidean, manhattan, or any other distance function allowed by ?dist(). To compute similarity distance of permuted observations, categorical features will be recoded based on whether or not they are equal to the actual observation. If continuous features are binned (the default) these features will be recoded based on whether they are in the same bin as the observation. Using the recoded data the distance to the original observation is then calculated based on a user-chosen distance measure. *Relates to algorithm step 2*.
* kernel\_width: To convert the distance measure to a similarity value, an exponential kernel of a user defined width (defaults to 0.75 times the square root of the number of features) is used. Smaller values restrict the size of the local region. *Relates to algorithm step 2*.
* n\_features: The number of features to best describe predicted outcomes. *Relates to algorithm step 4*.
* feature\_select: To select the best *n* features, lime can use forward selection, ridge regression, lasso, or a tree to select the features. In this example I apply a ridge regression model and select the *m* features with highest absolute weights. *Relates to algorithm step 4*.

For classification models we also have two additional features we care about and one of these two arguments must be given:

* labels: Which label do we want to explain? In this example, I want to explain the probability of an observation to attrit (“Yes”).
* n\_labels: The number of labels to explain. With this data I could select n\_labels = 2 to explain the probability of “Yes” and “No” responses.

explanation\_caret <- lime::explain(

x = local\_obs,

explainer = explainer\_caret,

n\_permutations = 5000,

dist\_fun = "gower",

kernel\_width = .75,

n\_features = 5,

feature\_select = "highest\_weights",

labels = "Yes"

)

The explain() function above first creates permutations, then calculates similarities, followed by selecting the *m* features. Lastly, explain() will then fit a model (*algorithm steps 5 & 6*). lime applies a ridge regression model with the weighted permuted observations as the simple model. [[2](http://www.business-science.io/business/2018/06/25/lime-local-feature-interpretation.html#fn:glmnet)] If the model is a regressor, the simple model will predict the output of the complex model directly. If the complex model is a classifier, the simple model will predict the probability of the chosen class(es).

The explain() output is a data frame containing different information on the simple model predictions. Most importantly, for each observation in local\_obs it contains the simple model fit (model\_r2) and the weighted importance (feature\_weight) for each important feature (feature\_desc) that best describes the local relationship.

tibble::glimpse(explanation\_caret)

## Observations: 25

## Variables: 13

## $ model\_type "classification", "classification", "cla...

## $ case "1", "1", "1", "1", "1", "2", "2", "2", ...

## $ label "Yes", "Yes", "Yes", "Yes", "Yes", "Yes"...

## $ label\_prob 0.216, 0.216, 0.216, 0.216, 0.216, 0.070...

## $ model\_r2 0.5517840, 0.5517840, 0.5517840, 0.55178...

## $ model\_intercept 0.1498396, 0.1498396, 0.1498396, 0.14983...

## $ model\_prediction 0.3233514, 0.3233514, 0.3233514, 0.32335...

## $ feature "OverTime", "MaritalStatus", "BusinessTr...

## $ feature\_value Yes, Single, Travel\_Rarely, Sales, Very\_...

## $ feature\_weight 0.14996805, 0.05656074, -0.03929651, 0.0...

## $ feature\_desc "OverTime = Yes", "MaritalStatus = Singl...

## $ data [[41, Yes, Travel\_Rarely, 1102, Sales, ...

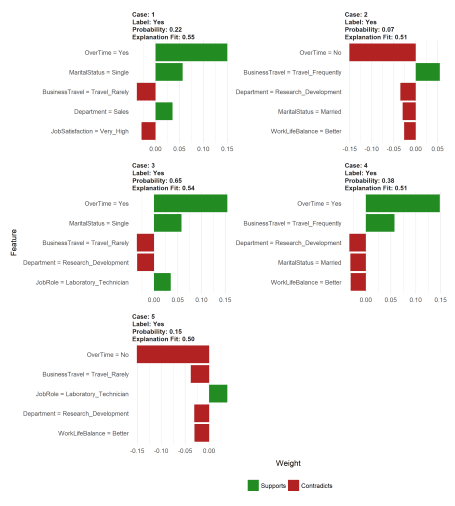
## $ prediction [[0.216, 0.784], [0.216, 0.784], [0.216...

**Visualizing results**

However the simplest approach to interpret the results is to visualize them. There are several plotting functions provided by lime but for tabular data we are only concerned with two. The most important of which is plot\_features(). This will create a visualization containing an individual plot for each observation (case 1, 2, …, n) in our local\_obs data frame. Since we specified labels = "Yes" in the explain() function, it will provide the probability of each observation attriting. And since we specified n\_features = 10 it will plot the 10 most influential variables that best explain the linear model in that observations local region and whether the variable is causes an increase in the probability (supports) or a decrease in the probability (contradicts). It also provides us with the model fit for each model (“Explanation Fit: XX”), which allows us to see how well that model explains the local region.

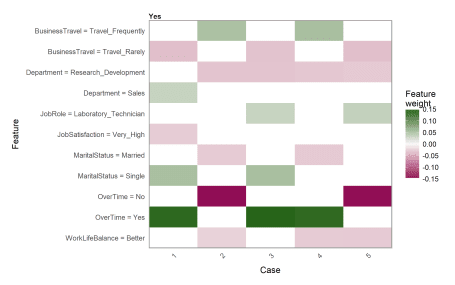
Consequently, we can infer that case 3 has the highest liklihood of attriting out of the 5 observations and the 3 variables that appear to be influencing this high probability include working overtime, being single, and working as a lab tech.

plot\_features(explanation\_caret)



The other plot we can create is a **heatmap** showing how the different variables selected across all the observations influence each case. We use the plot\_explanations() function. This plot becomes useful if you are trying to find common features that influence all observations or if you are performing this analysis across many observations which makes plot\_features() difficult to discern.

plot\_explanations(explanation\_caret)



**Tuning LIME**

As you saw in the above plot\_features() plot, the output provides the model fit. In this case the best simple model fit for the given local regions was *R^2 = 0.59* for case 3. Considering there are several knobs we can turn when performing the LIME algorithm, we can treat these as tuning parameters to try find the best fit model for the local region. This helps to maximize the amount of trust we can have in the local region explanation.

As an example, the following changes the distance function to use the manhattan distance algorithm, we increase the kernel width substantially to create a larger local region, and we change our feature selection approach to a LARS lasso model. The result is a fairly substantial increase in our explanation fits.

# tune LIME algorithm

explanation\_caret <- lime::explain(

x = local\_obs,

explainer = explainer\_caret,

n\_permutations = 5000,

dist\_fun = "manhattan",

kernel\_width = 3,

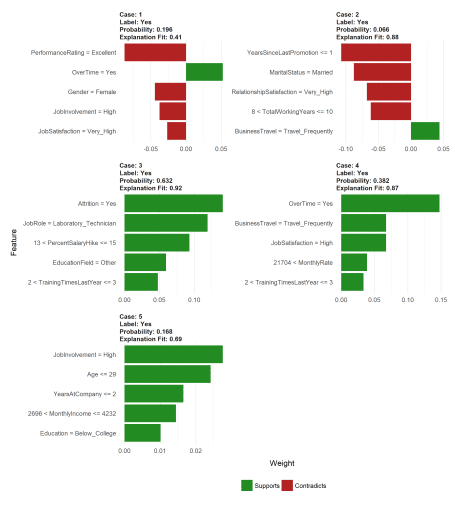
n\_features = 5,

feature\_select = "lasso\_path",

labels = "Yes"

)

plot\_features(explanation\_caret)



**Supported vs Non-support models**

Currently, lime supports supervised models produced in caret, mlr, xgboost, h2o, keras, and MASS::lda. Consequently, any supervised models created with these packages will function just fine with lime.

explainer\_h2o\_rf <- lime(train\_obs, h2o\_rf, n\_bins = 5)

explainer\_h2o\_glm <- lime(train\_obs, h2o\_glm, n\_bins = 5)

explainer\_h2o\_gbm <- lime(train\_obs, h2o\_gbm, n\_bins = 5)

explanation\_rf <- lime::explain(local\_obs,

explainer\_h2o\_rf,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

feature\_select = "highest\_weights")

explanation\_glm <- lime::explain(local\_obs,

explainer\_h2o\_glm,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

feature\_select = "highest\_weights")

explanation\_gbm <- lime::explain(local\_obs,

explainer\_h2o\_gbm,

n\_features = 5,

labels = "Yes",

kernel\_width = .1,

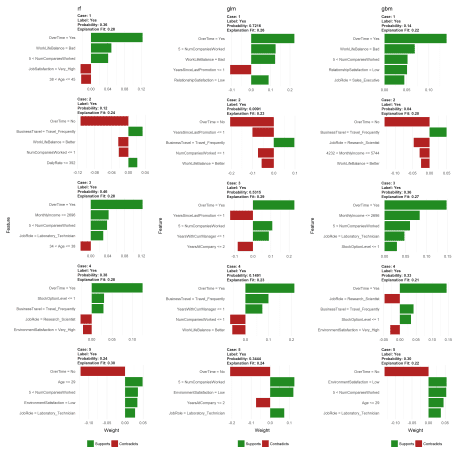
feature\_select = "highest\_weights")

p1 <- plot\_features(explanation\_rf, ncol = 1) + ggtitle("rf")

p2 <- plot\_features(explanation\_glm, ncol = 1) + ggtitle("glm")

p3 <- plot\_features(explanation\_gbm, ncol = 1) + ggtitle("gbm")

gridExtra::grid.arrange(p1, p2, p3, nrow = 1)



However, any models that do not have built in support will produce an error. For example, the model we created directly with ranger is not supported and produces an error.

explainer\_ranger <- lime(train, fit.ranger, n\_bins = 5)

## Error in UseMethod("lime", x): no applicable method for 'lime' applied to an object of class "function"

We can work with this pretty easily by building two functions that make lime compatible with an unsupported package. First, we need to create a model\_type() function that specifies what type of model this unsupported package is using. model\_type() is a lime specific function, we just need to create a ranger specific method. We do this by taking the class name for our ranger object and creating the model\_type.ranger method and simply return the type of model (“classification” for this example).

# get the model class

class(fit.ranger)

## [1] "ranger"

# need to create custom model\_type function

model\_type.ranger <- function(x, ...) {

# Function tells lime() what model type we are dealing with

# 'classification', 'regression', 'survival', 'clustering', 'multilabel', etc

return("classification")

}

model\_type(fit.ranger)

## [1] "classification"

We then need to create a predict\_model() method for ranger as well. The output for this function should be a data frame. For a regression problem it should produce a single column data frame with the predicted response and for a classification problem it should create a column containing the probabilities for each categorical class (binary “Yes” “No” in this example).

# need to create custom predict\_model function

predict\_model.ranger <- function(x, newdata, ...) {

# Function performs prediction and returns data frame with Response

pred <- predict(x, newdata)

return(as.data.frame(pred$predictions))

}

predict\_model(fit.ranger, newdata = local\_obs)

## Yes No

## 1 0.2915452 0.7084548

## 2 0.0701619 0.9298381

## 3 0.4406563 0.5593437

## 4 0.3465294 0.6534706

## 5 0.2525397 0.7474603

Now that we have those methods developed and in our global environment we can run our lime functions and produce our outputs.[3](http://www.business-science.io/business/2018/06/25/lime-local-feature-interpretation.html#fn:dynamic)

explainer\_ranger <- lime(train\_obs, fit.ranger, n\_bins = 5)

explanation\_ranger <- lime::explain(local\_obs,

explainer\_ranger,

n\_features = 5,

n\_labels = 2,

kernel\_width = .1)

plot\_features(explanation\_ranger, ncol = 2) + ggtitle("ranger")

