Occupancy Detection

Luke Fiorio

6/23/2020

Introduction

Project Summary and Goal:

This project uses data from the UCI Machine Learning Repository on a room's environmental measures to build an occupancy detection algorithm.

Our goal is to build a model that predicts, as accurately as possible, whether a room is occupied based on various data readings (e.g., temperature, humidity, etc.). Specifically, we aim to build a model that detects occupancy with maximal **overall accuracy**.

Data Description:

The data that we will use to build and test our model contains approximately 20,500 observations taken over the course of about 2.5 weeks in February, 2015. Our data has 1 target variable (`occupancy`) and 6 predictors (`date`, `temperature`, `humidity`, `light`, `co2`, `humidity_ratio`).

Occupancy is a binary field that indicates whether the room is occupied at the time of observation. Each row in our data represents an observation made at a given time (`date`); observations are typically made in 1-minute increments. Each observation includes data readings from electronic sensors reporting information on the temperature, humidity, light, and CO_2 levels.[1] Ground truth occupancy was determined by time-stamped photos taken alongside the sensor readings.

Key Steps:

- Split the data into two datasets for:
  - **training** (80%, ~16,500 observations)
  - **validation** (20%, ~4,100 observations)
- Explore the data to understand variable distributions and trends.
- Prepare the data and use k-fold crossvalidation to train and tune classification models using the following algorithms:
  - k-nearest neighbors
  - GAM Loess
  - Random Forest
- Make predictions on the validation set using the optimally tuned models built on the training data.
- Make predictions on the validation set using an ensemble (of the tuned models).
- And, finally, evaluate model performance.

[1] Note that 'humidity-ratio' is a derived term (based on temperature and humidity). It represents the ratio between the weight of water-vapor::air in the room.

Methods and Analysis

In this section, we explain the methods and analysis done to process, explore, and build our occupancy detection algorithm.

Data Cleaning:

We start by loading the necessary packages.

- tidyverse
- caret
- data.table
- gam
- lubridate
- scale
- gridExtra
- Rborist
- knitr

Then retrieving the datasets from the UCI Machine Learning Repository and combining them into a dataframe, occupancy.

```
# retreive the data from the UCI ML repository
dl <- tempfile()
download.file(
  "https://archive.ics.uci.edu/ml/machine-learning-databases/00357/occupancy_data.zip",
  dl)

# specify column names
col_names <- c("index", "date", "temperature", "humidity",
               "light", "co2", "humidity_ratio", "occupancy")

# read in each of the downloaded data files
data_1 <- fread(text = readLines(unzip(dl, "datatraining.txt")),
                sep = ",", header = FALSE, skip = 1,
                col.names = col_names)
data_2 <- fread(text = readLines(unzip(dl, "datatest.txt")),
                sep = ",", header = FALSE, skip = 1,
                col.names = col_names)
data_3 <- fread(text = readLines(unzip(dl, "datatest2.txt")),
                sep = ",", header = FALSE, skip = 1,
                col.names = col_names)

# combine the datasets and drop the index column
occupancy <- within(rbind(data_1, data_2, data_3), rm(index))
```

Before we go any further, we need to split our occupancy data into training and validation datasets. We'll set a seed for replicability, and then set aside a portion of our data to use **only** for validation of our final models.

We choose to set aside 20% of data for validation since we have only a modest number of observations to begin with (20,560 rows) and want to ensure sufficient sample to obtain an accurate evaluation of model performance on the validation data. We will use crossvalidation while training our models to maintain sufficient sample for model building and parameter tuning.

```
set.seed(1, sample.kind="Rounding") # set seed

# split the data into training & validation
test_index <- createDataPartition(
```

```
  y = occupancy$occupancy,
  times = 1, p = 0.2, list = FALSE)
train_set <- occupancy[-test_index,] # train data
test_set <- occupancy[test_index,] # validation data
```

Before we do additional data preparation for model training, let's explore the data.

Data Exploration and Visualization:

After splitting our data, a few simple commands show us that there are exactly 16,448 observations in our training data. These observations were taken from Monday, February 02, 2015 through Wednesday, February 18, 2015. The prevalence of our occupancy indicator is 22.9%; or, in other words, the room is occupied in 22.9% of the observations.

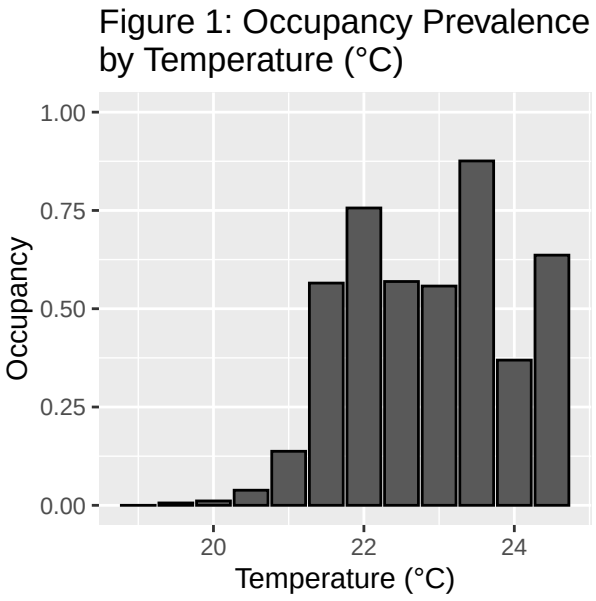
Scanning our dataset for missing values reveals none, so we can proceed.

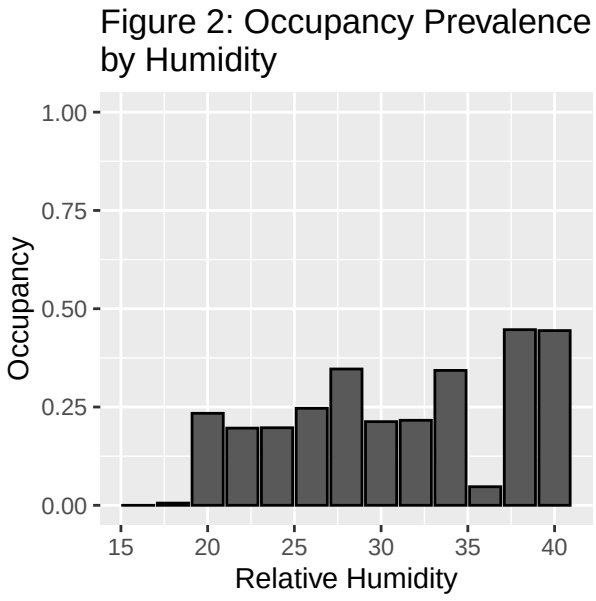
```
sapply(train_set, function(col) sum(is.na(col)))
```

```
##           date    temperature       humidity          light            co2
##              0              0              0              0              0
## humidity_ratio      occupancy
##              0              0
```

Let's get a sense for how each of our predictors relate to the occupancy indicator. Figure 1 shows that the room is much more likely to be occupied when the temperature is higher; the room is almost never occupied for temperatures below 21 Celsius. This makes sense: human bodies generate heat and, especially during February (northern-hemisphere's winter), the room may have even had a heater that occupants would turn on.

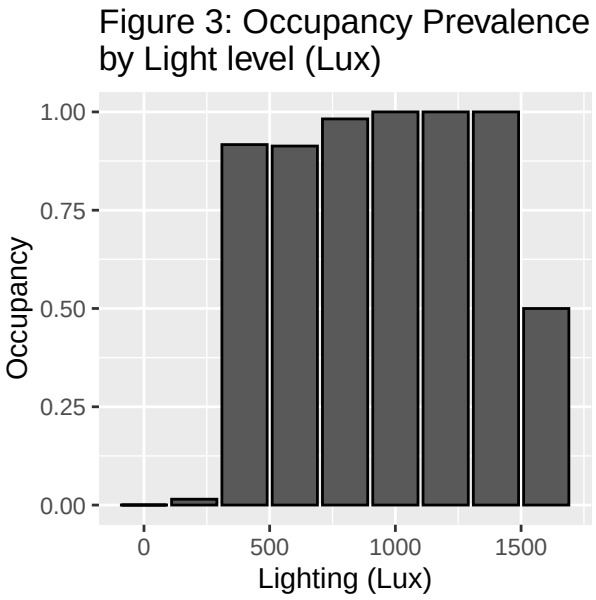
Similarly, Figure 2 shows higher occupancy prevalence at higher levels of humidity, although to a lesser degree than temperature. Occupancy prevalence also seems to be a bit inconsistent at higher humidity levels. This relationship between occupancy and humidity also makes sense: any activity that generates heat will likely lead to additional moisture in the air (and therefore, higher humidity).





Inspecting occupancy by lighting levels (Figure 3) reveals an *extremely* strong correlation. Light levels above 400 Lux almost entirely match occupancy levels. This makes sense intuitively, since most people turn on a light when they enter a room and turn it off when they exit.

Figure 4 shows that CO_2 levels appear to vary smoothly, but not linearly, with occupancy. Humans exhale CO_2 when breathing and so, conceptually, it makes sense that we tend to see higher occupancy at higher levels of CO_2.



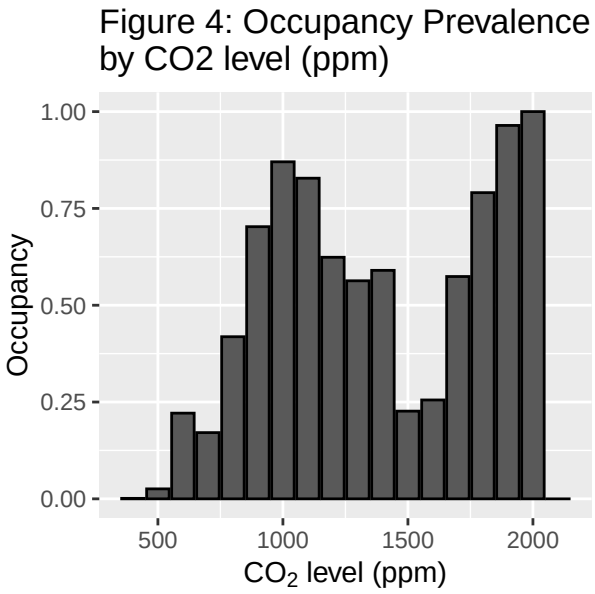
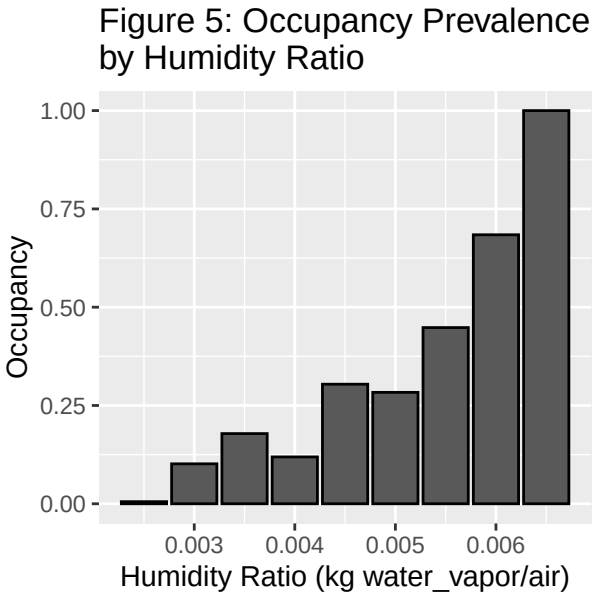
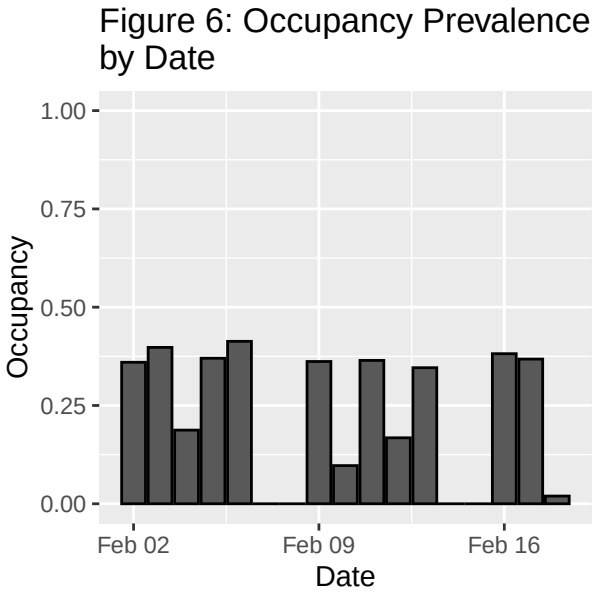


Figure 5 shows a very monotonic relationship between the humidity ratio and occupancy. It is a derived field (derived from Temperature and Relative Humidity) and appears to be a useful addition. Its relationship with occupancy appears stronger than humidity or temperature alone at higher ratio values.

Figure 6 shows that date appears unhelpful in predicting occupancy at first glance, other than a few days there appear to have been no occupancy (which may or may not be generalizable). However, further inspection reveals that there is more that can be gleaned from the time attribute.





Upon closer inspection of the time field, it is revealed that in fact the room is *never* occupied over the weekend (Figure 7). Furthermore, it also does not appear to ever be occupied outside of standard business hours (7AM-7PM) as shown in Figure 8.

Figure 7: Occupancy Prevalence by Day of Week

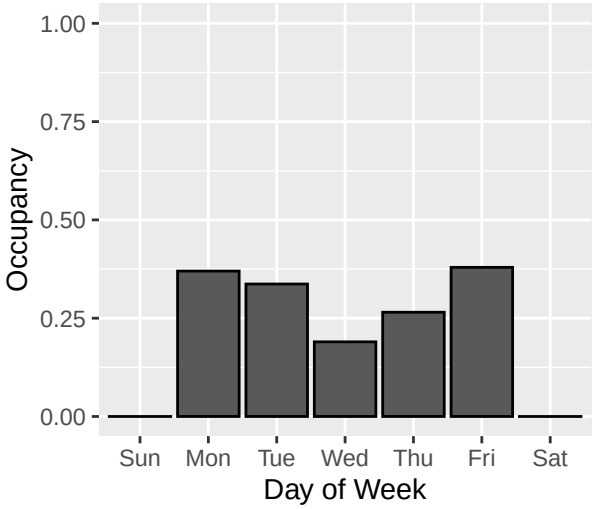
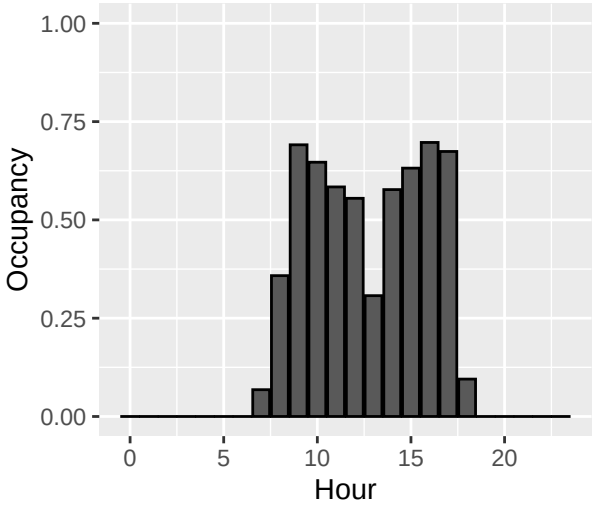


Figure 8: Occupancy Prevalence by Time of Day



Insights Gained:

Through data exploration, we've revealed that we should be able to detect occupancy better than by just guessing. We know that occupancy is *very* strongly related to light levels and business hours. It is also moderately to strongly related to several of our other predictors, namely: temperature, CO_2 levels, and the humidity ratio. Lastly, it also appears to be related to relative humidity, although to a lesser extent.

Further, we have seen that occupancy prevalence does not appear to vary linearly across most of our predictors. In fact, CO_2 levels and time of day both appear bi-modal in that regard. It also appears that several of our predictors have certain thresholds beyond which occupancy prevalence drops to 0.

Modeling Approach:

With these insights, we can devise a modeling approach.

Given that much of our data tend to follow smooth (but not globally linear) relationships with occupancy, it may make sense to select an algorithm (or algorithms) that generate smoothed estimates based on other nearby data.

One candidate for this is **k-nearest neighbors (knn)**, which will make a prediction for a given observation based on the most common occupancy value of the observations most similar to itself.[2] This allows for flexible estimates that will not necessarily be linear across our predictors. However, since the k-nearest neighbors algorithm is sensitive to the magnitude of our predictors, we will need to take additional steps to scale our data.

Another candidate to easily capture smoothed relationships is **Loess (GAM)**, which makes predictions using a smoothing function over a locally defined *span*. By capturing linear relationships within small localized subsets of the data, GAM is also able to make flexible esimates that are **not** globally linear.

Another trend we noticed in our data was that several of our predictors appeared to have 0% occupancy beyond certain thresholds or within certain ranges. To attempt to capture those trends, we will also build a **Random Forest** model, which uses a decision tree algorithm to select predictor cut-points that minimize prediction error. By randomly selecting the predictors used in each decision tree and tuning the model parameters (minimum node size and the number of randomly selected predictors), we can identify the best cut-points while avoiding over-training.

Given the differences in our models, we'll need to prep the data a bit differently for each.

[2]In this context, "similar" observations are defined by Euclidian Distance between predictors.

Let's start by training a **knn model**. First, we'll convert our timestamp field into two new fields, representing day-of-week and hour-of-day. We'll store it all in a new dataframe, since we'll prep data for each model a bit differently (and because our data is small enough to permit this additional use of memory).

```
train_set_knn <-
  train_set %>%
  mutate(dow = wday(date, label = TRUE),
         hour = hour(date))
```

From our ealier exploration, we know that there is never occupancy over the weekend or during late-night/early-morning hours. To reduce the number of dummy variables we'll create later, let's consolidate each of those categories.[3]

```
# combine weekend
levels(train_set_knn$dow) <-
  c("Wknd", "Mon", "Tue", "Wed", "Thu", "Fri", "Wknd")
train_set_knn$dow <- as.character(train_set_knn$dow)

# combine late-night/early-morning hours
train_set_knn$hour <-
  as.factor(ifelse(
    train_set_knn$hour <=6 | train_set_knn$hour >=19,
    0, train_set_knn$hour))
```

And let's drop the original timestamp field from this dataframe.

```
train_set_knn <- within(train_set_knn, rm(date))
```

Now, let's fit a dummy-encoder that we can use to dummy-encode our factor variables. We'll use this again when we process our validation data to make sure we dummy-encode our validation data using the same pipeline as we did for our training data.[4]

```
# dummy encoding pipeline
train_dummies <- dummyVars(~., train_set_knn)

# apply dummy encoding to train data
train_set_knn <- predict(train_dummies,
                         newdata = train_set_knn)
```

Since knn is sensitive to differences in predictor magnitude (it is a distance-based model), we'll need to also scale our data. Since we have dummy variables (in addition to the continuous variables that we need to scale), we'll use a min-max scalar, which will scale each of our predictors from 0 to 1.[5] We can specify this in our `trainControl` object using `preProc = c("range")`, which we'll use when we train our model.

This allows us to scale our crossvalidated (and validation) data using the min and max value **from the training data**, which is important in avoiding leakage from the validation data when training the model.

```
control <-
  trainControl(
    method="cv",
    number=5,
    preProc = c("range") # min-max scaler
```

[3] Fewer dimensions will help keep our knn model from suffering unnecessarily from the "curse of dimensionality" where the nearest neighbors included becomes skewed due to the calculation of Euclidian distance. Also, note that we dummy-encode the `hour` field to account for non-monotonic trends in occupancy within business hours, such as apparent lunch breaks around 1PM (low occupancy prevalence).

[4] Note that dummy-encoding is the process of creating a series of [0, 1] flags for each category in a categorical variable.

[5] Leaving our dummy variables unaffected, while putting our continuous variables all on the same scale. Otherwise, predictors with larger magnitudes will have an outsized effect on the model.

```
    )
```

Let's specify a sequence of `k` values to tune our model with. The `k` parameter informs the model on exactly *how many* neighbors to include when making its prediction for a given observation.

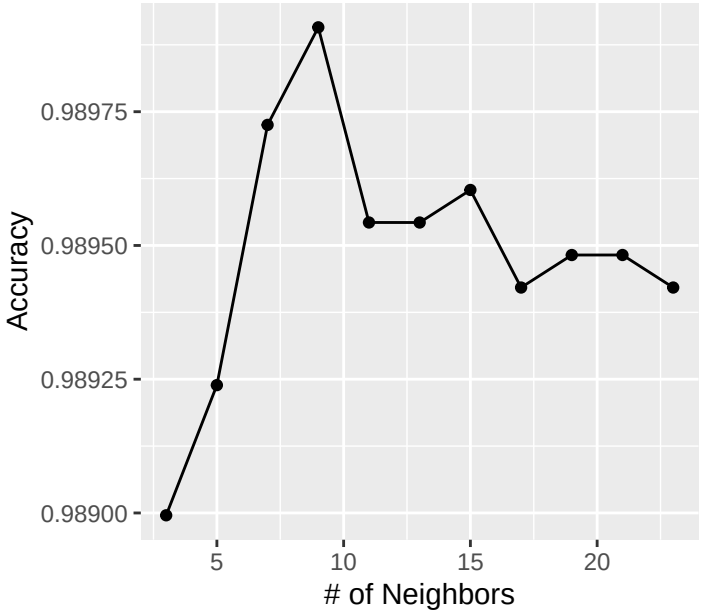
```
grid <- expand.grid(k = seq(from=3, to=23, by=2))
```

And now let's train our model. We convert `occupancy` to a factor so that `train()` knows this is a classification problem (not a regression problem).

```
train_knn <- train(
  as.factor(occupancy) ~ .,
  method = "knn",
  tuneGrid=grid,
  trControl = control,
  data = train_set_knn)
```

The crossvalidation results show that our best tune was with k = 9 and had high accuracy (99.0%) on the crossvalidated data. Later on, when we make predictions on our validation data, we'll see whether this model is generalizable (or just a result of over-training).

Figure 9: knn
Crossvalidated Accuracy



OK, now let's train a **Loess (GAM) model**. We'll start by rounding our timestamp to the nearest hour, which should help with smoothing. As with our knn modeling, we'll also keep this data in a separate dataframe.

```
train_set_gam <-
  train_set %>%
  mutate(date = round_date(as_datetime(date), "hour"))
```

Since Loess is not sensitive to differences in magnitude, we don't need to scale our data before training this algorithm.[6]

[6]This is because Loess is based on the *relative* relationships between our predictors and the target variable, which would not change with scaling.

So let's specify our `trainControl` object (using crossvalidation again) and a sequence of *span* values to tune our model with. We will keep our smoothing function linear by setting `degree` = 1.

```
control <- trainControl(method="cv", number=5,)
```

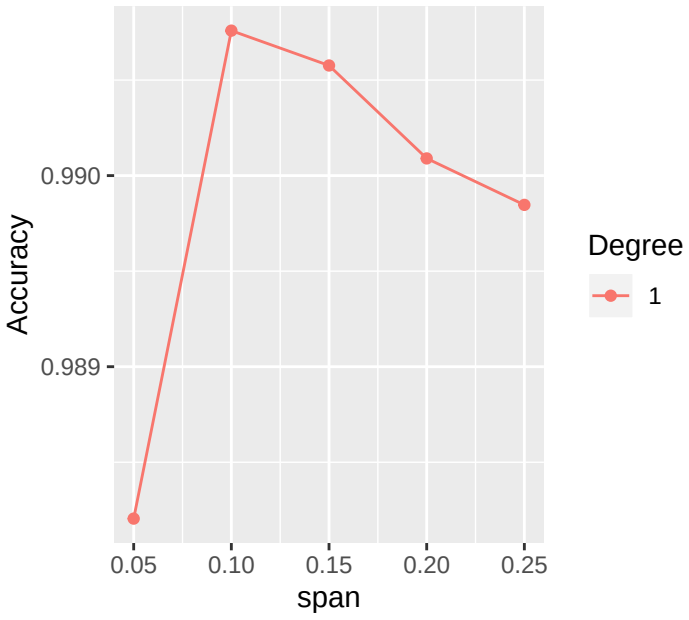
```
grid <- expand.grid(
  span = seq(from=0.05, to=.25, by=0.05),
  degree = 1
  )
```

And now let's train our model. Again, we'll convert `occupancy` to a factor so that `train()` knows this is a classification problem.

```
train_gam <- train(
  as.factor(occupancy) ~ .,
  method = "gamLoess",
  tuneGrid=grid,
  trControl = control,
  data = train_set_gam)
```

The crossvalidation results show that our best tune was with span = 0.1 and also had high accuracy (99.1%) on the crossvalidated data. Later on, we'll also validate whether this model is generalizable on our validation data.

Figure 10: GAM Loess Crossvalidated Accuracy



And, finally, let's train a **Random Forest model**. As with knn, we'll start by converting our timestamp field into two new fields; day-of-week and hour-of-day.

Since Random Forest is not sensitive to magnitudes, however, there's no need to consolidate weekend days or late-night/early-morning hours (as we did for knn). Further, there's no need to dummy encode these variables; Random Forest can handle them as they are.

```
train_set_rf <-
  train_set %>%
  mutate(dow = wday(date),
         hour = hour(date))
```

Let's drop the original timestamp field from this dataframe, also.

```
train_set_rf <- within(train_set_rf, rm(date))
```

Again, we don't need to scale our data for this algorithm. We'll define a `trainControl` object to specify crossvalidation and a grid of parameter values to tune our model with.

```
control <- trainControl(method="cv", number=5)
```

The `predFixed` parameter specifies how many (randomly selected) predictors to include when building each decision tree in our "forest." The `minNode` parameter tells our model what the minimum number of observations to include in each end-node (of a set of decision rules) must be. Unchecked, small values of `minNode` could lead to overtraining.

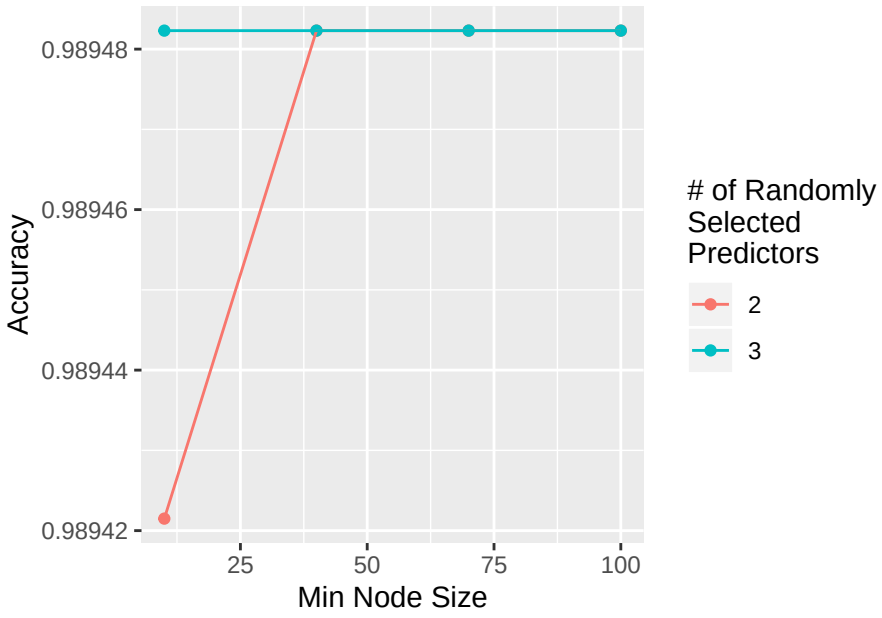
```
grid <- expand.grid(
  predFixed = seq(from=2, to=3, by=1),
  minNode = seq(from=10, to=100, by=30)
  )
```

And now we're ready to train the Random Forest model.

```
train_rf <-
  train(as.factor(occupancy) ~ .,
        method = "Rborist",
        tuneGrid = grid,
        trControl = control,
        data = train_set_rf
        )
```

The crossvalidation results show that our best tune was with *minimum node size* = 40 and *Number of predictors* = 2 with an accuracy of 98.9% on the crossvalidated data. Later on, we'll also validate whether this model is generalizable on our validation data.

Figure 11: Random Forest Crossvalidated Accuracy



We've built three different models (knn, GAM, Random Forest); let's use them to make predictions on the validation data that we set aside earlier. First, we'll need to make the same data transformations on our

validation data that we did on the train data. By and large, we take the same steps, and so don't present them again here. However, recall that as part of our data prep for knn we fit a dummy-encoder to also be used on our validation data, which we *do* show here.

```
# convert factors to dummy vars using fitted encoder
test_set_knn <- data.frame(
  predict(train_dummies, newdata = test_set_knn)
  )
```

Once we've replicated the data transformations on the validation data, we can now make row-by-row predictions for each observation in the validation data.

```
# make predictions using each model
pred_knn <- predict(train_knn, newdata=test_set_knn) # knn
pred_gam <- predict(train_gam, newdata=test_set_gam) # gam
pred_rf <- predict(train_rf, newdata=test_set_rf) # random forest
```

Before we evaluate our final model accuracies, let's build an ensemble using the predictions made by each model. In general, by combining the model predictions made above, this should result in a more stable model than any of them individually. To build our ensemble, we'll predict `occupancy = 1` if at least two of our models did (i.e. the majority of them); otherwise we predict unoccupied (`occupancy = 0`).

```
# combine model predictions into a dataframe
all_predictions <- data.frame(pred_rf, pred_gam, pred_knn)

# take the most common prediction
all_predictions['pred_ensemble'] <-
  as.factor(ifelse(
    rowMeans(all_predictions == "1") > 0.5, "1", "0")
    )
```

Finally, let's compare our predictions to the actual observed values. We'll store the information in a dataframe for further evaluation.

```
# calculate the accuracy of each model
accuracy <-
  c(mean(all_predictions$pred_knn==as.factor(test_set$occupancy)),
    mean(all_predictions$pred_gam==as.factor(test_set$occupancy)),
    mean(all_predictions$pred_rf==as.factor(test_set$occupancy)),
    mean(all_predictions$pred_ensemble==as.factor(test_set$occupancy))
    )

# set column names and place in dataframe.
models <- c("K nearest neighbors", "Loess (GAM)", "Random Forest", "Ensemble")
model_accuarcies <- data.frame(Model = models, Accuracy = accuracy)
```

Results

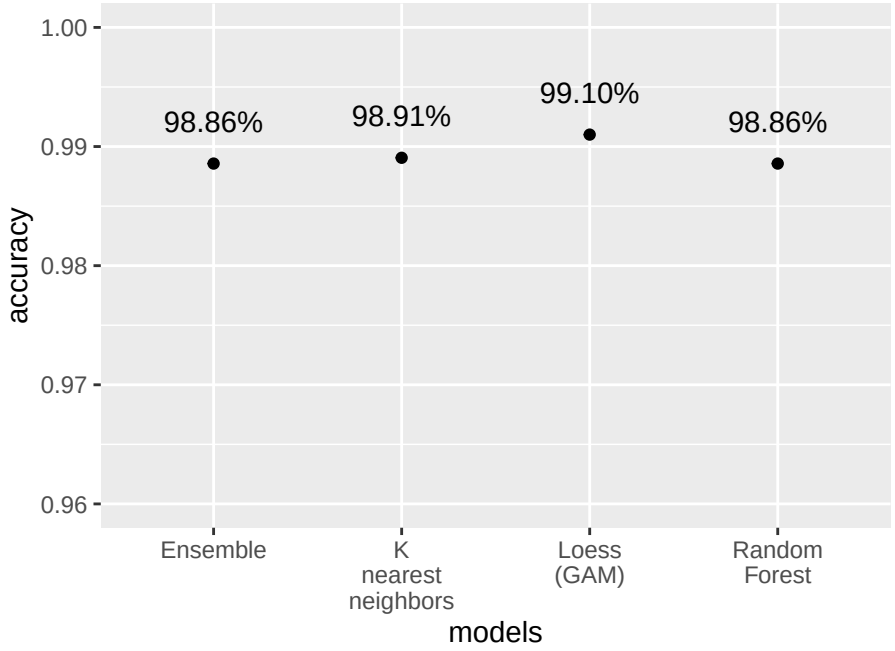
The below creates a table that shows our overall accuracy on the validation dataset for each model.

```
kable(model_accuarcies)
```

Model	Accuracy
K nearest neighbors	0.98906
Loess (GAM)	0.99100
Random Forest	0.98857
Ensemble	0.98857

And we can also visualize this information:

Figure 12: Final Accuracy, by Model



Overall, each of our models (including the ensemble) achieved an accuracy rate of about 99% in detecting whether a room is occupied. Our accuracy is somewhat inflated because of the class imbalance (recall, that the room was occupied in only 22.9% of observations), but even still is quite accurate.

The best performing model was Loess (GAM), which may have been better able to use the timestamp field to obtain smoothed estimates of occupancy, but all models performed relatively similarly to one another. Compared to our accuracy on the crossvalidated data, we see that the models obtained very similar accuracy, which leads us to say that these models are indeed generalizable.

The lack of boost achieved by the ensembling is notable and indicates that these models tend to have the same blindspots (when one is wrong, they are all wrong). Adding new models to the ensemble and/or using a bootstrap aggregation method could lead to improvements in ensemble accuracy.

Conclusion

Summary:

In conclusion, we were successfully able to use a room's environmental measures to build an occupancy detection algorithm. Our model takes advantage of both the environmental data and the timestamps included in the dataset to make extremely accurate predictions.

Our individual models are generalizable in that they predict just as well on new data as on the data they were trained on. Our ensembling model also predicts quite accurately, but was not an improvement over our individual models; it may suffer from over-fitting (relative to our individual models).

Limitations:

The models predict extremely accurately, but are partly limited by their reliance on timestamp data. Depending on the real-life application, this may or may not be a limiting factor (for example, if the room was in a house or store, as opposed to - presumably - an office building).

Our final ensemble is limited by the relatively few number of models that it incorporates, and its predictions appear to suffer as a result. Given the relatively small number of predictors (6), models that incorporate the conditional distribution of our predictors (such as LDA or QDA) may have been good additions. Also, since our ensemble was not created through Bagging (Bootstrap Aggregating), it appears to have limited generalizability relative to our individual models.[7]

Lastly, the relatively small sample size of our dataset may have constrained the predictive power of our model training (or limited our ability to use ensemble methods such as Bagging).

Future work:

Addressing the limitations presented above, our models could likely be further improved. Specifically, the creation of models that do not rely on timestamp data could improve generalizability in real-world applications. The incorporation of additional models into our ensembling, and potentially using Bagging techniques (while navigating or increasing sample size) could also help improve our ensemble predictions. Additional feature engineering beyond what was done in this analysis could also lead to accuracy improvements.

[7]Bagging is when we train each of our ensemble models on a **different** sample of the training data. Then with those separately trained models, make predictions on the validation set, which get ensembled together (take the mode prediction).