DS2 - ML Tools Assignment 1.

Son N. Nguyen

19 March 2022

```
# Loading packages with pacman
if (!require("pacman")) {
    install.packages("pacman")
}

pacman::p_load(tidyverse, glmnet, pls, rpart,
    ranger, gbm, kableExtra)

theme_set(theme_minimal())
```

1. Predict caravan insurance purchase

```
# load data
caravan_data <- as_tibble(ISLR::Caravan)

# recode yes no to binary
caravan_data$Purchase <- ifelse(caravan_data$Purchase ==
    "Yes", 1, 0)

set.seed(1234)
# create sample
caravan_sample <- slice_sample(caravan_data,
    prop = 0.2)
n_obs <- nrow(caravan_sample)
test_share <- 0.2

# split to train and set
test_indices <- sample(seq(n_obs), floor(test_share *
    n_obs))
caravan_test <- slice(caravan_sample, test_indices)
caravan_train <- slice(caravan_sample, -test_indices)
```

a.) What would be a good evaluation metric for this problem? Think about it from the perspective of the business.

Since this is a classification problem, **a great evaluation metric would be AUC or the expected loss given the definition of our loss function or a simple accuracy ratio.** We should set our assessment on how sensitive we are on False Positives (FP) and False Negatives (FN). I would assume that the loss function is tilted towards FP since surplus of insurance policies can stack up inventories, while shortage can be temporarily treated with increased premiums to select out consumers with high willingness to pay.

b.) Let's use the basic metric for classification problems: the accuracy (% of correctly classified examples). Train a simple logistic regression (using all of the available variables) and evaluate its performance on both the train and the test set. Does this model perform well? (Hint: You might want to evaluate a basic benchmark model first for comparison – e.g. predicting that no one will make a purchase.)

First, I have defined the function for calculating the classification accuracy. If the predicted outcome matches with the actual outcome, add 1, sum all the 1s up and divide with the length of the vector.

```
# define accuracy function
calculateAccuracy <- function(prediction,
    y_obs) {
    sum(ifelse(prediction == y_obs, 1, 0),
        na.rm = T)/length(y_obs)
}
```

I have used the glm function to fit a logistic regression. Then, I have extracted the fitted values from the model and readjusted the values according to the sigmoid shape of the logistic distribution. That is to assign zero to lower probabilities and one to higher probabilities. **The function yields us an accuracy of 95% when we evaluate on the train set. Using the test set to benchmark out model, we receive a similar high accuracy of 93.5%.**

```
# fit model
logit <- glm(Purchase ~ ., family = binomial(link = "logit"),
    data = caravan_train)

# transform fitted values and add to
# table
accuracy_results <- tibble(model = "Logit",
    train = calculateAccuracy(ifelse(logit$fitted.values <
        0.5, 0, 1), caravan_train$Purchase),
    test = calculateAccuracy(ifelse(predict(logit,
        caravan_test) < 0.5, 0, 1), caravan_test$Purchase))

# create reusable table display
show_table <- function() {
    kable(x = accuracy_results, digits = 3,
        caption = "Evaluation Metrics (Accuracy)") %>%
        kable_styling(latex_options = c("hold_position",
            "striped"), font_size = 8) %>%
        add_footnote(c("Source data: Caravan Insurance Purchase"))
}

show_table()
```

Table 1: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935

[a] Source data: Caravan Insurance Purchase

It is worth to check how each outcome is distributed in our data by showing a tabular form of values. It seems that **only 6% of the targeted customers have actually purchased the insurance policy, thus it may happen that our model is accurate because nearly all of our observations are 0**.

This model is actually worse on the test set than if we have predicted that not a single customer is purchasing the policy (see Naive evaluation metrics below).

```
accuracy_results <- add_row(accuracy_results,
    model = "Naive", train = calculateAccuracy(0,
        caravan_train$Purchase), test = calculateAccuracy(0,
        caravan_test$Purchase), )

show_table()
```

Table 2: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940

[a] Source data: Caravan Insurance Purchase

c.) Let's say your accuracy is 95%. Do we know anything about the remaining 5%? Why did we mistake them? Are they people who bought and we thought they won't? Or quite the opposite? Report a table about these mistakes (Hint: I would like to see the Confusion Matrix.)

Our model fall short on predicting those who will purchase the product. The confusion matrix shows that we perform well on the true negatives, **but have a relatively high number of false negatives (13) as well compared to the true positives (1) (predicted them not as purchasers but they have actually bought the policy)**

```
# confusion matrix for the train set
cm <- table(ifelse(predict(logit, caravan_test) <
    0.5, 0, 1), caravan_test$Purchase)

kable(x = cm, digits = 3, caption = "Confusion Matrix Train Set (Purchase)") %>%
    kable_styling(bootstrap_options = c("striped",
        "hover"), full_width = F) %>%
    add_footnote(c("w/ Logit on full set of variables")) %>%
    add_header_above(., header = c(Predicted = 1,
        Reference = 2))
```

Table 3: Confusion Matrix Train Set (Purchase)

Predicted	Reference	
	0	1
0	216	13
1	2	1

[a] w/ Logit on full set of variables

d.) What do you think is a more serious mistake in this situation?

As mentioned before, I believe that generally **predicting true negatives is more important for the insurance company. According to out predictions, we would have a shortage in policies** – driving our premiums up. In this specific case we would set policies to accommodate 1 customer but in fact 13 would be willing to buy it. This ratio can cause the company a large revenue loss and future customer churn. Therefore, in this particular case false positives can hurt more than we can imagine.

e.) You might have noticed (if you checked your data first) that most of your features are categorical variables coded as numbers. Turn your features into factors and rerun your logistic regression. Did the prediction performance improve?

```
# convert all columns to factor
col_names <- head(names(caravan_data), -1)
caravan_dataf <- caravan_sample

set.seed(1234)
n_obs <- nrow(caravan_dataf)
test_share <- 0.2

test_indices <- sample(seq(n_obs), floor(test_share *
    n_obs))
caravan_test <- slice(caravan_dataf, test_indices)
caravan_train <- slice(caravan_dataf, -test_indices)

caravan_test[, col_names] <- lapply(caravan_test[,
    col_names], factor)
caravan_train[, col_names] <- lapply(caravan_train[,
    col_names], factor)
```

The prediction performance has improved for the train set (97%) but substantially regressed for the external validity (83%). Overfitting might occurred in this case, but now our features are unordered and do not relate to scaling.

```
# drop factors with only 1 level from
# training set
# sapply(lapply(caravan_train, unique),
# length)

caravan_trainf <- subset(caravan_train, select = -c(AZEILPL,
    PZEILPL))

flogit1 <- glm(Purchase ~ ., family = binomial(link = "logit"),
    data = caravan_trainf, maxit = 100)
```

```
# there are levels present in the
# training set but not in the test set

# refit model for test set evaluation
# and drop factors levels not present
# in the test set
flogit2 <- glm(Purchase ~ ., family = binomial(link = "logit"),
```

```
    data = caravan_trainf, maxit = 100)

# relevel factors of test on train
for (fac in head(names(caravan_trainf), -1)) {
    caravan_test[fac] <- factor(caravan_test[[fac]],
        levels = levels(caravan_trainf[[fac]]))
}

accuracy_results <- add_row(accuracy_results,
    model = "FLogit", train = calculateAccuracy(ifelse(flogit1$fitted.values <
        0.5, 0, 1), caravan_train$Purchase),
    test = calculateAccuracy(ifelse(predict(flogit2,
        caravan_test) < 0.5, 0, 1), caravan_test$Purchase))

show_table()
```

Table 4: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940
FLogit	0.973	0.832

[a] Source data: Caravan Insurance Purchase

f.) Let's try a nonlinear model: build a simple tree model and evaluate its performance.

I've built the simplest tree model with no configuration whatsoever – feature engineering is to be figured out by the algo with no domain knowledge. After evaluating its accuracy on the test and training set, we can notice that the **train accuracy slightly decreased while test performance increased (96 and 90%, respectively).**

```
tree_model <- rpart(Purchase ~ ., caravan_train)

accuracy_results <- add_row(accuracy_results,
    model = "CART", train = calculateAccuracy(ifelse(predict(tree_model,
        caravan_train) < 0.5, 0, 1), caravan_train$Purchase),
    test = calculateAccuracy(ifelse(predict(tree_model,
        caravan_test) < 0.5, 0, 1), caravan_test$Purchase))

show_table()
```

Table 5: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940
FLogit	0.973	0.832
CART	0.958	0.897

[a] Source data: Caravan Insurance Purchase

If we check the confusion matrix for the regression tree, we can see that **the model not predicted a single customer and indeed there wasn't anyone who purchased.**

Table 6: Confusion Matrix Train Set (Purchase)

Predicted	Reference	
	0	1
0	208	12
1	12	0

[a] w/ CART on full set of variables

```
# confusion matrix for the test set
cm <- table(ifelse(predict(tree_model, caravan_test) <
    0.5, 0, 1), caravan_test$Purchase)

kable(x = cm, digits = 3, caption = "Confusion Matrix Train Set (Purchase)") %>%
    kable_styling(bootstrap_options = c("striped",
        "hover"), full_width = F) %>%
    add_footnote(c("w/ CART on full set of variables")) %>%
    add_header_above(., header = c(Predicted = 1,
        Reference = 2))
```

g.) Run a more flexible model (like random forest or GBM). Did it help?

I've decided to fit a random forest algorithm on the train set. **The train set performance is slightly better (97%) and a the test set has significantly improved from the outcome of a single regression tree (95%). In comparison with all the previous designs, Random Forest has been the best performer.**

```
rf <- ranger(Purchase ~ ., data = caravan_train)

# drop 10 rows from test with missing
# values
accuracy_results <- add_row(accuracy_results,
    model = "Random Forest", train = calculateAccuracy(ifelse(predict(rf,
        caravan_train, na.action = na.omit)$predictions <
        0.5, 0, 1), caravan_train$Purchase),
    test = calculateAccuracy(ifelse(predict(rf,
        na.omit(caravan_test))$predictions <
        0.5, 0, 1), na.omit(caravan_test)$Purchase))

show_table()
```

h.) Rerun two of your previous models (a flexible and a less flexible one) on the full train set. Ensure that your test result remains comparable by keeping that dataset intact. (Hint: use the anti_join() function as we did in class.) Interpret your results.

I've decided to re-run the CART and Random Forest model, since those two were the best performers after the refactoring of the dataset.

With a larger training sample, **we closed the gap between CART and Random Forest**. This is due to the fact that a single tree now can be larger with more terminal nodes and the algo can construct more

Table 7: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940
FLogit	0.973	0.832
CART	0.958	0.897
Random Forest	0.973	0.950

[a] Source data: Caravan Insurance Purchase

complex features itself. **It did not help specifically the ensemble model as test accuracy remained intact.**

```
caravan_data[, col_names] <- lapply(caravan_data[,
    col_names], factor)
caravan_train_full <- anti_join(caravan_data,
    caravan_test)
```

```
# CART
tree_model_full <- rpart(Purchase ~ ., caravan_train_full)

accuracy_results <- add_row(accuracy_results,
    model = "CART Full", train = calculateAccuracy(ifelse(predict(tree_model_full,
        caravan_train_full) < 0.5, 0, 1),
        caravan_train_full$Purchase), test = calculateAccuracy(ifelse(predict(tree_model_full,
        caravan_test) < 0.5, 0, 1), caravan_test$Purchase))

show_table()
```

Table 8: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940
FLogit	0.973	0.832
CART	0.958	0.897
Random Forest	0.973	0.950
CART Full	0.940	0.948

[a] Source data: Caravan Insurance Purchase

```
# Regression Tree
rf_full <- ranger(Purchase ~ ., caravan_train_full)

accuracy_results <- add_row(accuracy_results,
    model = "Random Forest Full", train = calculateAccuracy(ifelse(predict(rf_full,
        caravan_train_full)$predictions <
        0.5, 0, 1), caravan_train_full$Purchase),
    test = calculateAccuracy(ifelse(predict(rf_full,
        na.omit(caravan_test))$predictions <
        0.5, 0, 1), na.omit(caravan_test)$Purchase))

show_table()
```

Finally, let's see a confusion matrix:

Table 9: Evaluation Metrics (Accuracy)

model	train	test
Logit	0.952	0.935
Naive	0.943	0.940
FLogit	0.973	0.832
CART	0.958	0.897
Random Forest	0.973	0.950
CART Full	0.940	0.948
Random Forest Full	0.968	0.950

[a] Source data: Caravan Insurance Purchase

Table 10: Confusion Matrix Train Set (Purchase)

Predicted	Reference	
	0	1
0	211	11

[a] w/ RF on full set of variables

The model did not predict a single customer who are going to purchase and indeed the number of false and true positives are zero. However, we have 11 false negative customers which could generate a large expected loss.

```
# confusion matrix for the train set
cm <- table(ifelse(predict(rf_full, na.omit(caravan_test))$predictions <
    0.5, 0, 1), na.omit(caravan_test)$Purchase)

kable(x = cm, digits = 3, caption = "Confusion Matrix Train Set (Purchase)") %>%
    kable_styling(bootstrap_options = c("striped",
        "hover"), full_width = F) %>%
    add_footnote(c("w/ RF on full set of variables")) %>%
    add_header_above(., header = c(Predicted = 1,
        Reference = 2))
```

2. Predict real estate value

a.) Think about an appropriate loss function you can use to evaluate your predictive models. What is the risk (from the business perspective) you would have to take by a wrong prediction?

To scale down extreme values, **I would choose RMSLE as the loss function. It is known that RMSLE has a feature of imposing larger penalties for underestimation of the actual value, which is perfect in our use case as we tolerate more overestimation**. By undervaluing the true value of a home, loss is imposed on the those who sell, while overvaluing impacts buyers negatively. The risk with the former is larger since sellers will not use the product and buyers will not meet the demand of sellers on the market. In the latter case, sellers will stay on the app benefiting them and buyers will less likely to leave because of the information asymmetry they have (more likely to believe the valuation).

```
# Loss function
calculateRMSLE <- function(prediction, y_obs) {
    sqrt(mean((log(ifelse(prediction < 0,
```

```
        0, prediction) + 1) - log(y_obs +
        1))^2))
}
```

b.) Put aside 20% of your data for evaluation purposes (using your chosen loss function). Build a simple benchmark model and evaluate its performance on this hold-out set.

```
real_estate <- read_csv("../data/real_estate/real_estate.csv")
set.seed(1234)

n_obs <- nrow(real_estate)
test_share <- 0.2

test_indices <- sample(seq(n_obs), floor(test_share *
    n_obs))
real_estate_test <- slice(real_estate, test_indices)
real_estate_train <- slice(real_estate, -test_indices)
```

I have chosen the mean of target value as a benchmark model. It's performance is relatively poor on the hold-out set, since it suggests that* **we would make a wrong prediction in 36% of the cases.**

```
# Benchmark model: Median value
prediction <- rep(median(real_estate_train$house_price_of_unit_area),
    length(real_estate_train$house_price_of_unit_area))

rmsle_results <- tibble(model = "Benchmark (Median)",
    train = calculateRMSLE(prediction, real_estate_train$house_price_of_unit_area),
    test = calculateRMSLE(prediction, real_estate_test$house_price_of_unit_area))

# create reusable table display
show_table <- function() {
    kable(x = rmsle_results, digits = 3,
        caption = "Evaluation Metrics (RMSLE)") %>%
        kable_styling(latex_options = c("hold_position",
            "striped"), font_size = 8) %>%
        add_footnote(c("Source data: Real Estate Value"))
}

show_table()
```

Table 11: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365

[a] Source data: Real Estate Value

c.) Build a simple linear regression model and evaluate its performance. Would you launch your evaluator web app using this model?

Not necessarily because we **would make an expected error of 38% when predicting the value of houses**. Such inaccuracy would make customers lose their trust in our product. Also in terms of test

performance, it is **worse than the median.**

```
lm <- lm(house_price_of_unit_area ~ ., data = real_estate_train)

rmsle_results <- add_row(rmsle_results, model = "Linear Regression",
    train = calculateRMSLE(predict(lm, real_estate_train),
        real_estate_train$house_price_of_unit_area),
    test = calculateRMSLE(predict(lm, real_estate_test),
        real_estate_test$house_price_of_unit_area))

show_table()
```

Table 12: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365
Linear Regression	0.244	0.375

[a] Source data: Real Estate Value

d.) Try to improve your model. Take multiple approaches (e.g. feature engineering, more flexible models, stacking, etc.) and document their successes.

1. Feature Engineering: I have decided to calculate the km distance of houses from the city center of Taipei. This uses a Havershine method ("distance as the bird flies"), taking into account the sphere shape of the globe.

```
# calculate distance form New Taipei
# City Hall

# Vector of distances in the same unit
# as r (default is meters)
library(geosphere)
centroid <- c(121.5654, 25.033)
vect <- c()
for (i in 1:nrow(real_estate)) {
    v <- distm(c(as.numeric(real_estate[i,
        "longitude"]), as.numeric(real_estate[i,
        "latitude"])), centroid, fun = distHaversine)/1000
    vect <- c(vect, v)
}

# assign distance to new column
real_estate$km_from_center <- as.integer(vect)

set.seed(1234)
n_obs <- nrow(real_estate)
test_share <- 0.2

test_indices <- sample(seq(n_obs), floor(test_share *
    n_obs))
real_estate_test <- slice(real_estate, test_indices)
real_estate_train <- slice(real_estate, -test_indices)
```

2. Model Selection: I tried out CART, Random Forest and GBM with minor feature engineering. I included the quadratic term of age as I expect it to have an exponential effect on price. In case of ensemble models we rarely need to do a detailed feature engineering since the algorithm can figure the whole thing out itself (e.g. interactions).

As expected, **CART and Random Forest are performing much better than linear regression as the test RMSLE have decreased from 0.375 to 0.214 and 0.192, respectively.**

```
# CART
tree_model <- rpart(house_price_of_unit_area ~
    house_age + house_age^2 + km_from_center +
        number_of_convenience_stores + distance_to_the_nearest_MRT_station,
    real_estate_train)

rmsle_results <- add_row(rmsle_results, model = "CART",
    train = calculateRMSLE(predict(tree_model,
        real_estate_train), real_estate_train$house_price_of_unit_area),
    test = calculateRMSLE(predict(tree_model,
        real_estate_test), real_estate_test$house_price_of_unit_area))

show_table()
```

Table 13: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365
Linear Regression	0.244	0.375
CART	0.193	0.214

[a] Source data: Real Estate Value

```
# Regression Tree
rf_re <- ranger(house_price_of_unit_area ~
    house_age + house_age^2 + km_from_center +
        number_of_convenience_stores + distance_to_the_nearest_MRT_station,
    real_estate_train)

rmsle_results <- add_row(rmsle_results, model = "Random Forest",
    train = calculateRMSLE(predict(rf_re,
        real_estate_train)$predictions, real_estate_train$house_price_of_unit_area),
    test = calculateRMSLE(predict(rf_re,
        real_estate_test)$predictions, real_estate_test$house_price_of_unit_area))

show_table()
```

Table 14: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365
Linear Regression	0.244	0.375
CART	0.193	0.214
Random Forest	0.102	0.193

[a] Source data: Real Estate Value

Fitting a GBM with 1000 trees and up to 4-way interactions, it has not improved from the Random Forest setup.

```
# GBM
gbm <- gbm(house_price_of_unit_area ~ house_age +
    house_age^2 + km_from_center + number_of_convenience_stores +
    distance_to_the_nearest_MRT_station,
    data = real_estate_train, n.trees = 1000,
    shrinkage = 0.01, interaction.depth = 4)

rmsle_results <- add_row(rmsle_results, model = "GBM",
    train = calculateRMSLE(predict(gbm, real_estate_train),
        real_estate_train$house_price_of_unit_area),
    test = calculateRMSLE(predict(gbm, real_estate_test),
        real_estate_test$house_price_of_unit_area))
```

Table 15: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365
Linear Regression	0.244	0.375
CART	0.193	0.214
Random Forest	0.102	0.193
GBM	0.146	0.209

[a] Source data: Real Estate Value

At last, we can do a simple stacking by taking the average of the predictions across the previous four models. **It seems that the method came as second best in terms of test evaluation, but still falling behind the Random Forest model.**

```
# gather predictions in a tibble
real_estate_predictions <- select(real_estate_test,
    house_price_of_unit_area) |>
    mutate(prediction_lm = predict(lm, real_estate_test),
        prediction_tree = predict(tree_model,
            real_estate_test), prediction_rf = predict(rf_re,
            real_estate_test)$predictions,
        prediction_gbm = predict(gbm, real_estate_test))
```

```
# Stacking average out predictions
stacked_prediction <- (real_estate_predictions$prediction_lm +
    real_estate_predictions$prediction_tree +
    real_estate_predictions$prediction_rf +
    real_estate_predictions$prediction_gbm)/4

rmsle_results <- add_row(rmsle_results, model = "Stacking",
    test = calculateRMSLE(stacked_prediction,
        real_estate_predictions$house_price_of_unit_area))

show_table()
```

Table 16: Evaluation Metrics (RMSLE)

model	train	test
Benchmark (Median)	0.394	0.365
Linear Regression	0.244	0.375
CART	0.193	0.214
Random Forest	0.102	0.193
GBM	0.146	0.209
Stacking	NA	0.202

[a] Source data: Real Estate Value

e.) Would you launch your web app now? What options you might have to further improve the prediction performance?

To conclude, we can already run a beta version of the app but would refrain from putting it into full production. According to out model ((we still have a 20% uncertainty in predicting values which is crucial in this industry. We could improve the model with **collecting more data –> able to fit more complex structures, as well as improve the dataset horizontally with additional features like amenities, orientation, etc.** Also note that we have applied "black-box" models with the least hyperparameter-tuning. Thus, we are right to assume that even **with a minimal amount of tuning, and domain knowledge we can marginally improve our models**.