Data Scientist Professional Practical Exam Submission

Use this template to write up your summary for submission. Code in Python or R needs to be included.

📝 Task List

Your written report should include both code, output and written text summaries of the following:

- Data Validation:
  - Describe validation and cleaning steps for every column in the data
- Exploratory Analysis:
  - Include two different graphics showing single variables only to demonstrate the characteristics of data
  - Include at least one graphic showing two or more variables to represent the relationship between features
  - Describe your findings
- Model Development
  - Include your reasons for selecting the models you use as well as a statement of the problem type
  - Code to fit the baseline and comparison models
- Model Evaluation
  - Describe the performance of the two models based on an appropriate metric
- Business Metrics
  - Define a way to compare your model performance to the business
  - Describe how your models perform using this approach
- Final summary including recommendations that the business should undertake

Start writing report here..

We will start by loading the dataset from the workspace.

```r
# Start coding here...
# Checking if the data I need is in the home directory
list.files()

# Loading the data with the function corresponding to files in CSV format
data_recipe_row <- read.csv("recipe_site_traffic_2212.csv")
data_recipe <- data_recipe_row
```

Hidden output

```r
# Reviewing the structure of the data set
str(data_recipe)
```

```
'data.frame':   947 obs. of  8 variables:
 $ recipe      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ calories    : num  NA 35.5 914.3 97 27.1 ...
 $ carbohydrate: num  NA 38.56 42.68 30.56 1.85 ...
 $ sugar       : num  NA 0.66 3.09 38.63 0.8 ...
 $ protein     : num  NA 0.92 2.88 0.02 0.53 ...
 $ category    : chr  "Pork" "Potato" "Breakfast" "Beverages" ...
 $ servings    : chr  "6" "4" "1" "4" ...
 $ high_traffic: chr  "High" "High" NA "High" ...
```

```r
# Reviewing the data structure
head(data_recipe)

cat("Number of rows in the table", length(data_recipe$recipe), "\n")
```

in...	re...	calo...	carbohydrate	su...	pr...	category	servi...
1	1	Invalid Number	Invalid Number	Invalid Num...	Invalid Num...	Pork	6
2	2	35.48	38.56	0.66	0.92	Potato	4
3	3	914.28	42.68	3.09	2.88	Breakfast	1
4	4	97.03	30.56	38.63	0.02	Beverages	4
5	5	27.05	1.85	0.8	0.53	Beverages	4
6	6	691.15	3.46	1.65	53.93	One Dish Meal	2

Rows: 6 Expand

```
Number of rows in the table 947
```

In the table above, you can see the data structure, with 8 columns and 947 rows. You can also see many missing values in several columns. Next, we'll perform data validation on the table and clean up all missing values, taking into account different criteria, which I'll explain as we go through each element.

Data cleaning and validation

```
# Reviewing the 1st variable "recipe"
cat("The variable recipe is of class:", class(data_recipe$recipe), "\n")
```

```
The variable recipe is of class: integer
```

```
# Checking if it has any NA value
cat("Does the recipe variable have any NA value?", any(is.na(data_recipe$recipe)), "\n")
```

```
Does the recipe variable have any NA value? FALSE
```

As you can see, this variable has no NA value since it represents the recipe IDs, so it shouldn't have NA values here. Now we'll move on to the next variable or column in the dataset.

```
# Checking the 2nd variable "calories""
cat("The variable calories is of class:", class(data_recipe$calories), "\n")
cat("Does the variable calories have any NA value?:", any(is.na(data_recipe$calories)), "\n")
cat("The variable calories has", sum(is.na(data_recipe$calories)), "values NA.", "\n")
```

```
The variable calories is of class: numeric
Does the variable calories have any NA value?: TRUE
The variable calories has 52 values NA.
```

We proceed to address these missing values (MVs) by finding the average of the other values in that column and replacing them wherever missing values exist. I do this because there are only 52 missing values, which represents only 5.4% of the data in that column, and that doesn't pose a significant risk. If this value were larger, given that the mean is sensitive to outliers, we would use the median or other more sophisticated methods. Because the column is a numeric column, there are no other forms of missing values other than MVs.

```
# Finding the average of the variable "calories" without taking into account the NA values
avg_calories <- round(mean(data_recipe$calories, na.rm = TRUE), 2)

# Replacing the NA values with the mean of the variable
data_recipe$calories[is.na(data_recipe$calories)] <- avg_calories

# Checking again if after the data transformation there is any NA value left
cat("Does the variable calories have any NA value after replacing it with the average?:", any(is.na(data_recipe$calories)),
"\n")
```

```
Does the variable calories have any NA value after replacing it with the average?: FALSE
```

We will perform the above procedure with the following three columns in the data set: "carbohydrate," "sugar," and "protein." This is because they have the same structure.

```
# Reviewing the 3rd variable "carbohydrate"
cat("The variable carbohydrate is of class:", class(data_recipe$carbohydrate), "\n")
any(is.na(data_recipe$carbohydrate))    # revisando si tiene valores faltantes
sum(is.na(data_recipe$carbohydrate))    # revisando la cantidad de valores faltantes

# Finding the average of the variable "carbohydrate" without taking into account the NA values
avg_carbohydrate <- round(mean(data_recipe$carbohydrate, na.rm = TRUE), 2)

# Replacing the NA values with the mean of the variable
data_recipe$carbohydrate[is.na(data_recipe$carbohydrate)] <- avg_carbohydrate
any(is.na(data_recipe$carbohydrate))  # checking again if after the data transformation there is any NA value left


# Reviewing the 4th variable "sugar"
cat("The variable sugar is of class:", class(data_recipe$sugar), "\n")
any(is.na(data_recipe$sugar))    # Checking for missing values
sum(is.na(data_recipe$sugar))    # Checking the number of missing values

# Finding the average of the variable "sugar" without taking into account the NA values
avg_sugar <- round(mean(data_recipe$sugar, na.rm = TRUE), 2)

# Replacing the NA values with the mean of the variable
data_recipe$sugar[is.na(data_recipe$sugar)] <- avg_sugar
any(is.na(data_recipe$sugar))  # checking again if after the data transformation there is any NA value left


# Reviewing the 5th variable "protein"
cat("The variable protein is of class:", class(data_recipe$protein), "\n")
any(is.na(data_recipe$protein))    # Checking for missing values
sum(is.na(data_recipe$protein))    # Checking the number of missing values

# Finding the average of the variable "protein" without taking into account the NA values
avg_protein <- round(mean(data_recipe$protein, na.rm = TRUE), 2)

# Replacing the NA values with the mean of the variable
data_recipe$protein[is.na(data_recipe$protein)] <- avg_protein
any(is.na(data_recipe$protein))  # Checking again if after the data transformation there is any NA value left
```

```
The variable carbohydrate is of class: numeric
```

```
TRUE
```

```
52
```

```
FALSE
```

```
The variable sugar is of class: numeric
```

```
TRUE
```

```
52
```

```
FALSE
```

```
The variable protein is of class: numeric
```

```
TRUE
```

```
52
```

```
FALSE
```

Now we will continue with the category variable, which must be composed of specific categories.

```r
# Reviewing the 6th variable "category"
cat("The category variable is of class:", class(data_recipe$category), "\n")
any(is.na(data_recipe$category))    # Checking if the variable has missing values
table(data_recipe$category)
cat("The variable category has", length(table(data_recipe$category)), "categories", "\n")
```

```
The category variable is of class: character
```

```
FALSE
```

```
     Beverages      Breakfast        Chicken Chicken Breast        Dessert 
            92            106             74             98             83 
  Lunch/Snacks           Meat  One Dish Meal           Pork         Potato 
            89             79             71             84             88 
     Vegetable 
            83 
```

```
The variable category has 11 categories
```

We quickly see a problem, because when we count the number of categories the column has, we see that there is one more category than it should have, which is 10.

```r
# These are the categories that the category variable must have.
categorias <- c("Lunch/Snacks", "Beverages", "Potato", "Vegetable", "Meat", "Chicken", "Pork", "Dessert", "Breakfast", "One Dish Meal")

cat("The category variable should have", length(categorias), "categories", "\n")
```

```
The category variable should have 10 categories
```

```r
# Packages that we will need to be able to perform the transformation of the variable
if(!require(tidyverse)) install.packages("tidyverse")
library(tidyverse)
```

Hidden output

```r
# Checking if some of the category values are not among the values that should be in category
any(!names(table(data_recipe$category)) %in% categorias)

# Checking which values are not among the values that should be
names(table(data_recipe$category))[!names(table(data_recipe$category)) %in% categorias]

# Combining Chicken and Chicken Breast into Chicken (uniting them into 1 category)
if(!require(tidyverse)) install.packages("tidyverse")
library(tidyverse)
data_recipe <- data_recipe %>% mutate(category = ifelse(category == "Chicken Breast", "Chicken", category))

# Checking if after the change there is any other unwanted category
any(!names(table(data_recipe$category)) %in% categorias)
```

```
TRUE
```

```
'Chicken Breast'
```

```
FALSE
```

What I do is identify if there is a value in the category column that doesn't fall into the default values. When I see that the result is true, I proceed to identify the different value, and it says "Chicken Breast." What's happening is that they have one category for chicken and another for chicken breast, which can only be combined into the chicken category. After combining the categories, I double-checked that everything was correct and that the variable contained the categories it should have. Then we move on to the next variable, which is servings.

```
# Reviewing the 7th variable "servings"
cat("The variable servings is of class:", class(data_recipe$servings), "\n")

# Reviewing the composition of the variable
table(data_recipe$servings)
```

```
The variable servings is of class: character
```

```

          1             2             4  4 as a snack             6  6 as a snack
        175           183           389             2           197             1
```

```
# Performing transformations on the variable to make it numeric
data_recipe <- data_recipe %>% mutate(servings = round(as.numeric(case_when(servings == "4 as a snack" ~ 4,
                                                                            servings == "6 as a snack" ~ 6,
                                                                            TRUE ~ as.numeric(servings)))))
```

Hidden output

```
any(is.na(data_recipe$servings))    # Checking for missing values
```

```
FALSE
```

As we can see, this variable is a character class, which is incorrect because it should only contain numeric values. We review its composition and see that the problem lies in the fact that three elements of the variable have text, to specify that within a category, they are a subcategory (4 as a snack and 6 as a snack). Next, we convert these text strings into numbers in a very simple way since there were only three, and we convert the variable to a numeric class.

After that, we check if there are any NA values, and since none exist, we proceed to analyze the last variable, "high_traffic."

```
# Checking the 8th variable "high_traffic"
cat("The variable high_traffic is of class:", class(data_recipe$high_traffic), "\n")

# Reviewing the composition of the "high traffic" variable
table(data_recipe$high_traffic)

# Checking for missing values
cat("The High Traffic variable has some NA value?", any(is.na(data_recipe$high_traffic)), "\n")

# Checking the number of missing values
cat("The High Traffic variable has", sum(is.na(data_recipe$high_traffic)), "NA value","\n")

# Transforming the data to binary because it is easier to work with, where 1 is high traffic and 0 is low traffic
data_recipe$high_traffic[!is.na(data_recipe$high_traffic)] <- 1
data_recipe$high_traffic[is.na(data_recipe$high_traffic)] <- 0
data_recipe$high_traffic <- as.numeric(data_recipe$high_traffic)
```

```
The variable high_traffic is of class: character
```

```

High
 574
```

```
The High Traffic variable has some NA value? TRUE
The High Traffic variable has 373 NA value
```

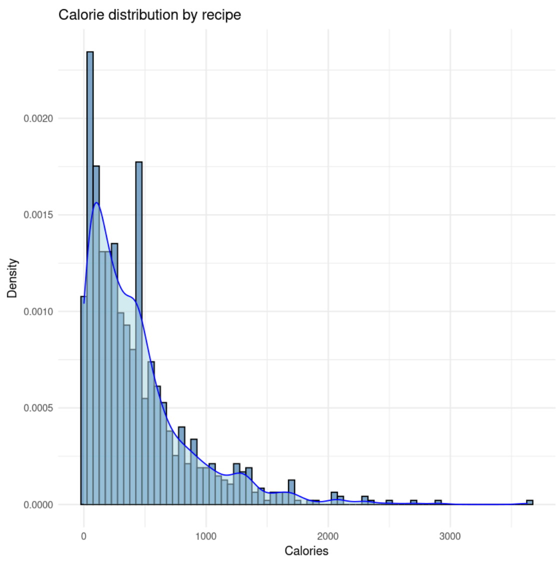
"The high_traffic column is crucial to our goal of predicting popularity. The data provided marks some recipes with the 'High' label, while others appear as missing values (NA). The project warns us that 'Don't forget to double-check the data really does match what they say - it might not', and that 'they haven't given us all of the information'. In this context, and given that the goal is to identify 'high traffic' recipes, it is inferred that recipes not marked as 'High' did not reach that popularity threshold and are therefore considered 'low traffic' for the purposes of this analysis. This is a necessary assumption to convert the variable into a usable binary format. Consequently, 'High' labels were transformed to 1 and NAs to 0. The result is a target variable with 60.6% high traffic instances and 39.4% low traffic."

With the data ready, I can continue with an exploratory analysis to learn a little more about the data set.

Data analysis and exploration

Now, we will show a distribution graph of the calories variable represented by the dencity diagram of the variable and a histogram of the same.

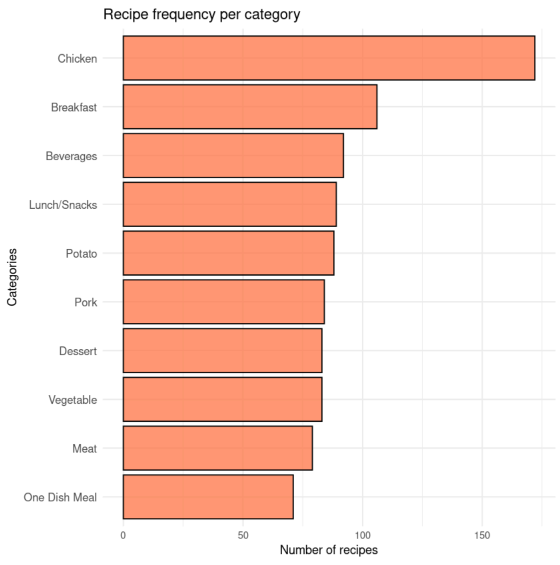
```r
if(!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)

# Histogram and density diagram for 'calories'
ggplot(data_recipe, aes(x = calories)) +
  geom_histogram(aes(y = ..density..), binwidth = 50, fill = "steelblue", color = "black", alpha = 0.7) +
  geom_density(alpha = 0.5, fill = "lightblue", color = "blue") +
  labs(title = "Calorie distribution by recipe",
       x = "Calories",
       y = "Density") +
  theme_minimal()
```



The calorie distribution by recipe shows a clear positive bias (to the right). This indicates that most recipes tend to have a relatively low to moderate calorie number, while there is a long recipe tail with much higher caloric content. A main recipe concentration is observed in the range of (approximately) 0 to 750 calories, with the highest density (peak) probably around 100 calories. The right tail of the distribution suggests the presence of recipes with significantly high caloric content, which could be considered in particularly caloric outliers or recipes.

Now we continue with a bar graph, where we can see the frequency of recipes for each of the categories.

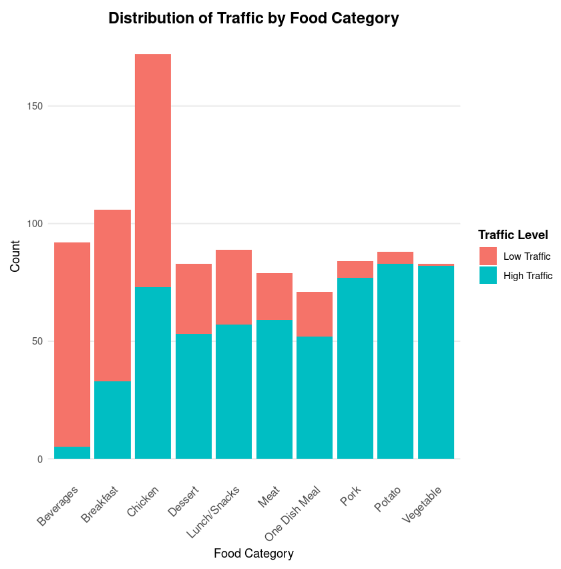
```r
# Bars chart for 'category'
ggplot(data_recipe, aes(x = fct_rev(fct_infreq(category)))) +
  geom_bar(fill = "coral", color = "black", alpha = 0.8) +
  coord_flip() +
  labs(title = "Recipe frequency per category",
       x = "Categories",
       y = "Number of recipes") +
  theme_minimal() +
  theme(axis.text.y = element_text(size = 10))
```



The bar chart reveals that the 'Chicken' category is the most numerous in our dataset. It is followed in frequency by the 'Breakfast' and 'Beverages' categories. On the other hand, categories such as 'One Dish Meal' and 'Meat' are the least represented in the dataset. The distribution of recipes across categories is uneven. There is considerable variation in the number of recipes per category, which could impact the model's ability to learn specific patterns for the underrepresented categories.

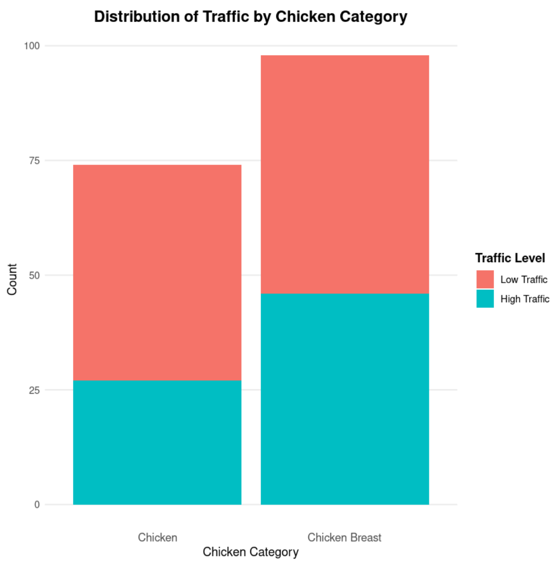
The following graph shows a count of all recipe categories, with the portion of those categories that do not produce an increase in website traffic highlighted in red.

```
# Distribution of categories according to their traffic
data_recipe %>% ggplot(aes(x = category, fill = as.factor(high_traffic))) +
  geom_bar() + scale_fill_manual(name = "Traffic Level",
                               labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                               values = c("0" = "#F8766D", "1" = "#00BFC4")) +
  labs(title = "Distribution of Traffic by Food Category", x = "Food Category", y = "Count") +
  theme_minimal() + theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1, size = 10),
                        plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                        legend.title = element_text(face = "bold"),
                        panel.grid.major.x = element_blank(),
                        panel.grid.minor.y = element_blank())
```



At first glance, we immediately notice that the Beverages category is having almost no impact on increasing website traffic, so we should consider removing it completely from the options available in the future. Another category quite similar to Beverages is Breakfast, which isn't as extreme as Beverages, but still has a significant negative impact on website traffic. We can also see that chicken is the category with the most recipes, but just over half of them don't represent an increase in website traffic. Remember that the chicken category is made up of chicken and chicken breast, so we must analyze this data using the original dataset, which I show in the following graph:

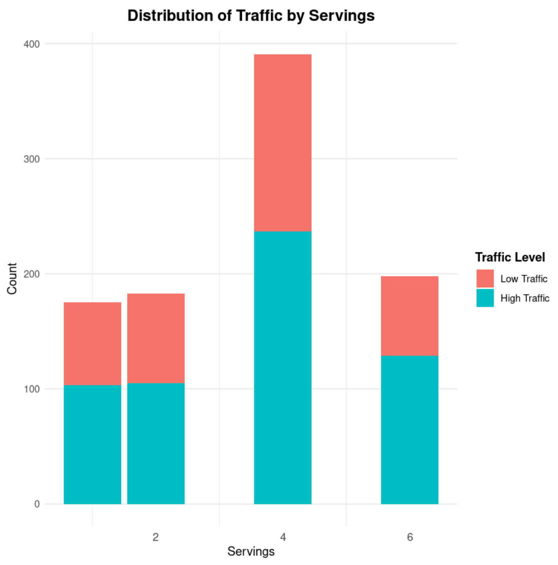
```
data_recipe_row %>% select(category, high_traffic) %>%
mutate(high_traffic = as.numeric(ifelse(is.na(high_traffic), 0, 1))) %>%
filter(category %in% c("Chicken", "Chicken Breast")) %>%
ggplot(aes(category, fill = as.factor(high_traffic))) + geom_bar() + scale_fill_manual(name = "Traffic Level",
                                 labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                                 values = c("0" = "#F8766D", "1" = "#00BFC4")) +
  labs(title = "Distribution of Traffic by Chicken Category", x = "Chicken Category", y = "Count") +
  theme_minimal() + theme(axis.text.x = element_text(size = 10),
                          plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                          legend.title = element_text(face = "bold"),
                          panel.grid.major.x = element_blank(),
                          panel.grid.minor.y = element_blank())
```



But as we can see, the two classes are quite well balanced with respect to the number of successful recipes they have, so there is no bias in combining the two.

We continue with a graph showing the number of prescriptions versus the number of services provided:

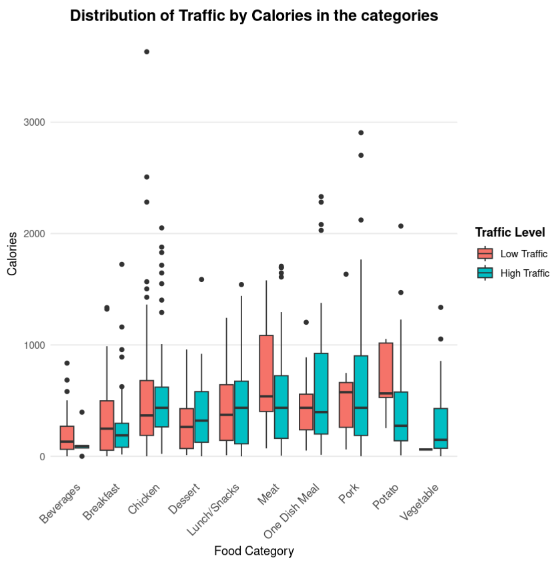
```
# Traffic according to the number of services
data_recipe %>% ggplot(aes(servings, fill = as.factor(high_traffic))) + geom_bar() +
scale_fill_manual(name = "Traffic Level", labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                  values = c("0" = "#F8766D", "1" = "#00BFC4")) +
labs(title = "Distribution of Traffic by Servings", x = "Servings", y = "Count") +
theme_minimal() + theme(axis.text.x = element_text(size = 10),
                        plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                        legend.title = element_text(face = "bold"),
                        panel.grid.major.x = element_blank(),
                        panel.grid.minor.y = element_blank())
```



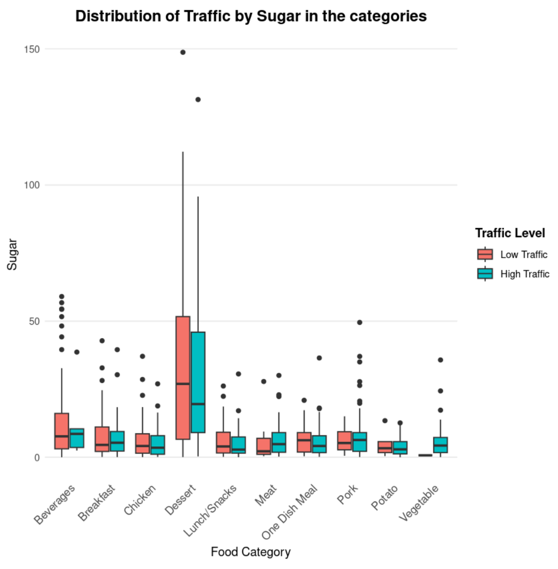
It is clear that 4-serving recipes are the most common, and in turn, these are also the ones most accepted by our public, because a large part of them generate good traffic to our website, which is exactly 41.3%.

Now let's analyze what happens with the different types of macronutrients, such as calories, proteins, and sugars, across all categories and taking into account the traffic they generate for the website.

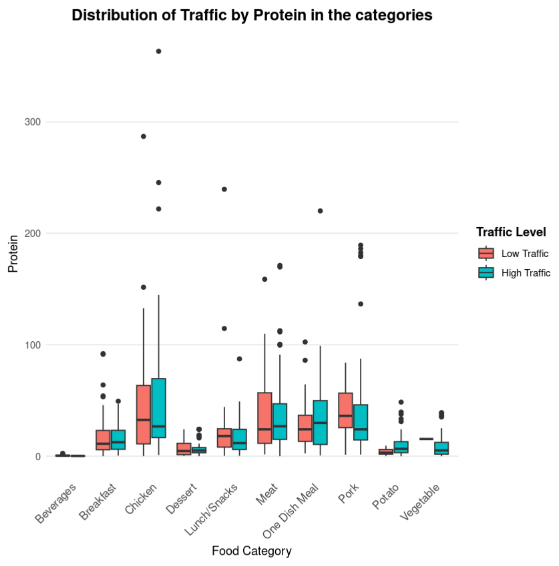
```
# Distribution of sugar according to traffic in the different categories
data_recipe %>% ggplot(aes(category, calories, fill = as.factor(high_traffic))) + geom_boxplot() +
                       scale_fill_manual(name = "Traffic Level",
                                         labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                                         values = c("0" = "#F8766D", "1" = "#00BFC4")) +
                    labs(title = "Distribution of Traffic by Calories in the categories", x = "Food Category", y = "Calories")
+ theme_minimal() + theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1, size = 10),
                                        plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                                        legend.title = element_text(face = "bold"),
                                        panel.grid.major.x = element_blank(),
                                        panel.grid.minor.y = element_blank())
```



```
# Distribution of sugar according to traffic in the different categories
data_recipe %>% ggplot(aes(category, sugar, fill = as.factor(high_traffic))) + geom_boxplot() +
                       scale_fill_manual(name = "Traffic Level",
                                         labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                                         values = c("0" = "#F8766D", "1" = "#00BFC4")) +
                    labs(title = "Distribution of Traffic by Sugar in the categories", x = "Food Category", y = "Sugar") +
theme_minimal() + theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1, size = 10),
                                   plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                                   legend.title = element_text(face = "bold"),
                                   panel.grid.major.x = element_blank(),
                                   panel.grid.minor.y = element_blank())
```



```r
# Distribution of proteins according to traffic in the different categories
data_recipe %>% ggplot(aes(category, protein, fill = as.factor(high_traffic))) + geom_boxplot() +
                       scale_fill_manual(name = "Traffic Level",
                                         labels = c("0" = "Low Traffic", "1" = "High Traffic"),
                                         values = c("0" = "#F8766D", "1" = "#00BFC4")) +
                    labs(title = "Distribution of Traffic by Protein in the categories", x = "Food Category", y = "Protein") +
theme_minimal() + theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1, size = 10),
                                     plot.title = element_text(hjust = 0.5, face = "bold", size = 14),
                                     legend.title = element_text(face = "bold"),
                                     panel.grid.major.x = element_blank(),
                                     panel.grid.minor.y = element_blank())
```



As can be seen in the graphs above, there are many outliers in each of the different categories, considering each of the macronutrients separately. In some of the categories, there is not much variability in traffic, but in others it is clearly evident, both for and against.

When analyzing the influence of calories, sugar, and protein on recipe traffic within each category, distinct patterns are observed:

- Calories: The relationship with high traffic varies by category.
  - For entrees such as Chicken, Meat, and Pork, similar or even slightly higher calorie content does not appear to deter high traffic.
  - In contrast, in Beverages and Vegetables, popular recipes tend to be lower in calories.
  - For Desserts, calories do not appear to be a clear differentiator of popularity.
- Sugar: Its impact is very category-specific.
  - In Desserts, higher sugar content is clearly associated with high traffic.
  - In Beverages, popular options tend to be less sweet.
  - For most savory dishes, sugar has a minimal or unclear impact on popularity.
- Protein: Higher protein content generally correlates positively with high traffic, especially in:
  - Categories such as 'Chicken', 'Meat', 'Pork', and 'One Dish Meal', where popular recipes tend to have more protein.
  - In 'Dessert' and 'Beverages', protein is not a relevant factor.

Overall Conclusion from Macronutrient Analysis: These findings underscore that user preferences regarding macronutrients depend on the recipe type. There is no single nutritional "recipe for success"; instead, the characteristics that drive popularity vary significantly across categories. This heterogeneity is an important consideration for developing the predictive model.

Creating the machine learning model

To address the product team's request to predict popular recipes, we will now develop a machine learning model. The specific objective is to classify recipes based on their expected traffic, distinguishing between those likely to generate 'high_traffic' (coded as 1) and those not (coded as 0). ***This is a binary classification task***.

Baseline model

Our model development strategy will include fitting a baseline model for initial comparison, followed by a more complex model. To ensure robust evaluation, we will divide our dataset into a training set, for model learning, and a test set, to gauge performance on new, unobserved data.

We'll start by installing and loading the "caret" package, which we'll use to carry out these tasks. Then, we'll split the data by creating a random index, which we'll use to create the training and test sets.

```
###################################
# Creating the baseline model
###################################

if(!require(caret)) install.packages("caret")
library(caret)
```

```
# Creating an index to be able to separate the data into training and testing
set.seed(1)
train_index <- createDataPartition(data_recipe$high_traffic, times = 1, p = 0.8, list = FALSE)
data_modif <- data_recipe %>% select(-recipe)

# Using the index to create the training and test sets
train <- data_modif[train_index,]
test <- data_modif[-train_index,]
```

We will continue with the creation of the baseline model, for which we will use the training data and predict the values it returns for a new data set, which would be the test set.

```
# Creating the baseline model
model <- lm(high_traffic ~ ., data = train)
summary(model)

# Making predictions with the model
predicciones <- predict(model, newdata = test)
```

```

Call:
lm(formula = high_traffic ~ ., data = train)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.99020 -0.31912  0.06635  0.32731  0.95885 

Coefficients:
                        Estimate Std. Error t value Pr(>|t|)    
(Intercept)            6.234e-02  6.098e-02   1.022 0.306986    
calories              -5.039e-06  3.486e-05  -0.145 0.885120    
carbohydrate           1.339e-04  3.880e-04   0.345 0.730203    
sugar                 -7.768e-04  1.367e-03  -0.568 0.570041    
protein                3.946e-04  4.490e-04   0.879 0.379760    
categoryBreakfast      2.430e-01  6.814e-02   3.566 0.000385 ***
categoryChicken        3.546e-01  6.625e-02   5.353 1.15e-07 ***
categoryDessert        5.850e-01  7.916e-02   7.390 3.96e-13 ***
categoryLunch/Snacks   5.747e-01  7.154e-02   8.033 3.72e-15 ***
categoryMeat           6.934e-01  7.589e-02   9.137  < 2e-16 ***
categoryOne Dish Meal  6.050e-01  7.700e-02   7.857 1.38e-14 ***
categoryPork           8.336e-01  7.473e-02  11.155  < 2e-16 ***
categoryPotato         8.528e-01  7.331e-02  11.633  < 2e-16 ***
categoryVegetable      9.166e-01  7.116e-02  12.880  < 2e-16 ***
servings               1.303e-03  8.710e-03   0.150 0.881087    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4085 on 743 degrees of freedom
Multiple R-squared:  0.3127,    Adjusted R-squared:  0.2997 
F-statistic: 24.14 on 14 and 743 DF,  p-value: < 2.2e-16
```

The model's predictions are given in probability format, so we must convert them to binary as follows:

```
# Converting predictions into binary
predict_high_trafic <- as.factor(ifelse(predicciones >= 0.5, 1, 0))
```

Next, we calculate the confusion matrix, which will give us the metrics we need to evaluate the model's performance, which in this particular case, we are only interested in accuracy.

```
# Creating a confusion matrix to find the accuracy of the model
CM <- confusionMatrix(predict_high_trafic, as.factor(test$high_traffic))
CM
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 58 18
         1 19 94

               Accuracy : 0.8042
                 95% CI : (0.7404, 0.8583)
    No Information Rate : 0.5926
    P-Value [Acc > NIR] : 4.623e-10

                  Kappa : 0.5937

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.7532
            Specificity : 0.8393
         Pos Pred Value : 0.7632
         Neg Pred Value : 0.8319
             Prevalence : 0.4074
         Detection Rate : 0.3069
   Detection Prevalence : 0.4021
      Balanced Accuracy : 0.7963

       'Positive' Class : 0
```

As we can see, the baseline model offers us a very good starting point, as we already obtain an 80.4% accuracy with it on the test set. This primarily indicates that the data have a strong linear correlation.

Comparison model

Next, we will create another model to compare it to the baseline model and its performance. Due to all the characteristics of the data set, such as the outliers in the macronutrient variables, the binary form of the target variable, and the multiple predictor variables, which increase further, I decided to use the ***Generalized Linear Models with Elastic Net Regularization*** (glmnet) method, which can handle all these characteristics very well, for example, the outliers and high dimensionality, attacking them with L2 and L1 regularization, respectively.

Let's start by loading the packages we will need for this model.

```
if(!require(pROC)) install.packages("pROC")
if(!require(MLmetrics)) install.packages("MLmetrics")
if(!require(glmnet)) install.packages("glmnet")

library(glmnet)
library(pROC)
library(MLmetrics)
```

Next, we move on to the control methods and hyperparameters that will be used to create the model.

For the control methods, we will use 10-fold cross-validation, repeated three times. This will allow us to find the best hyperparameters in the most robust way possible thanks to repetition. For the metrics to be evaluated, we use twoClassSummary, which will evaluate the area under the ROC curve, Sensitivity, and Specificity, which are more robust in a binary classification problem like this. Because we will be working with the ROC curve, it is necessary for the model to predict class probabilities with classProbs = TRUE. Finally, we instruct it to show us the training progress and, if possible, run the training in parallel to increase its speed.

```
fit_control <- trainControl(
  method = "repeatedcv",              # Repeated cross-validation
  number = 10,                        # 10 folds
  repeats = 3,                        # Repeat the CV 3 times
  summaryFunction = twoClassSummary,  # For ROC, Sensitivity, Specificity
  classProbs = TRUE,                  # Important! Required for ROC and probabilities
  verboseIter = TRUE,                 # Show progress
  allowParallel = TRUE                # Allows parallelization if you have multiple cores
)
```

We continue with the hyperparameter configuration we'll be using, where alpha values range from pure L2 regularization to pure L1 regularization, with Elastic Net values in between, which gives us the opportunity to find the optimal penalty. We also have lambda, which represents the strength of the regulation.

```
# Hyperparameters to be tested to determine the best combination for the model
glmnet_grid <- expand.grid(alpha = seq(0, 1, length = 5),
                           lambda = 10^seq(-3, 0, length = 20))
```

Next, we'll separate the training and test datasets into predictor and target variables. We'll also need to use one-hot encoding for the categorical variables, which in this case is just the category variable, and we'll convert the target variable into a factor. All of this is necessary to train the model; otherwise, we wouldn't get results that meet the client's criteria.

```
# # Separating the predictive characteristics of the targets for the training set and for the test set
x_train <- train %>% select(-high_traffic)
y_train <- train %>% select(high_traffic) %>% pull() %>% as.factor()
levels(y_train) <- make.names(levels(y_train))

x_test <- test %>% select(-high_traffic)
y_test <- test %>% select(high_traffic) %>% pull() %>% as.factor()
levels(y_test) <- make.names(levels(y_test))

# Create a dummy variable for the predictors to convert the category column into one-hot encoding.
dummy_obj <- dummyVars(~ ., data = x_train, fullRank = TRUE)

# Apply the transformation to x_train and x_test
x_train_processed <- predict(dummy_obj, newdata = x_train)
x_test_processed <- predict(dummy_obj, newdata = x_test)

# Convert to data.frame if predict returns an array (optional but good practice)
x_train <- as.data.frame(x_train_processed)
x_test <- as.data.frame(x_test_processed)
```

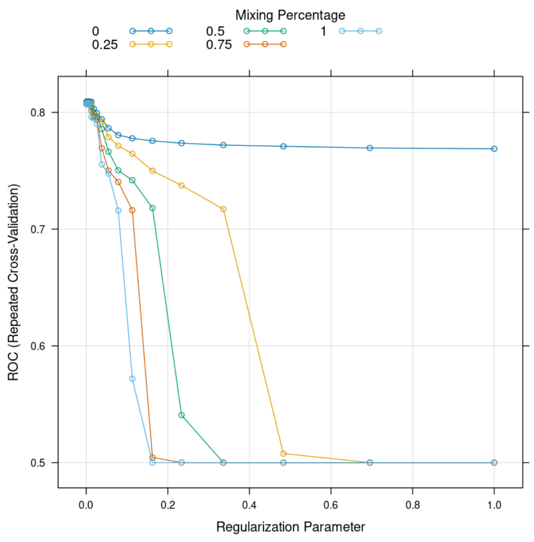
We will proceed to train the model, using all the parameters we defined previously, as well as the controls. We also use the ROC metric due to the atypical values presented by the data and because the glmnet model returns probabilities for a classification, as in our case, and ROC works very well with those probabilities. I also use data preprocessing, which centers and scales all the variables so that the calculations are more precise and easier for the model to interpret.

1 hidden cell

```
# This is to have an overview of the model metrics
print(glmnet_model)
```

```
glmnet

758 samples
 14 predictor
  2 classes: 'X0', 'X1'

Pre-processing: centered (14), scaled (14)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 683, 682, 683, 682, 681, 683, ...
Resampling results across tuning parameters:

  alpha  lambda       ROC        Sens         Spec
  0.00   0.001000000  0.8091897  0.652260536  0.8080944
  0.00   0.001438450  0.8091897  0.652260536  0.8080944
  0.00   0.002069138  0.8091897  0.652260536  0.8080944
  0.00   0.002976351  0.8091897  0.652260536  0.8080944
  0.00   0.004281332  0.8091897  0.652260536  0.8080944
  0.00   0.006158482  0.8091897  0.652260536  0.8080944
  0.00   0.008858668  0.8091897  0.652260536  0.8080944
  0.00   0.012742750  0.8089728  0.650038314  0.8080944
  0.00   0.018329807  0.8031139  0.647777778  0.8102529
  0.00   0.026366509  0.7993561  0.644406130  0.8109775
  0.00   0.037926902  0.7940966  0.642145594  0.8152791
  0.00   0.054555948  0.7865753  0.637662835  0.8174530
  0.00   0.078475997  0.7805986  0.619616858  0.8203515
  0.00   0.112883789  0.7777454  0.521800766  0.8275825
  0.00   0.162377674  0.7755834  0.405938697  0.8376812
  0.00   0.233572147  0.7736089  0.351839080  0.8636756
  0.00   0.335981829  0.7719958  0.257088123  0.9062442
```

```r
# Shows how ROC varies with alpha and lambda
plot(glmnet_model)
```



We print the model to get an overview of it and all its metrics throughout the training process. We can see that it begins to underfit starting with the eighth parameter combination because lambda begins to increase, which greatly simplifies the model.

We also have the ROC curve graph, which shows the model's performance along lambda, with each line representing a different alpha value. On the Y axis, we see the ROC value. We can see in this graph that the model is sensitive to small lambda and alpha values and that the model tends to lose effectiveness at higher values of these two parameters due to the underfitting caused by them.

With the model trained and ready to be used, we can move on to making predictions on the test dataset and evaluating those predictions.

```
# We make predictions from the objective data in the evaluation set
predictions_glmnet_class <- predict(glmnet_model, newdata = x_test)

# Because we converted the objectives into factors, we must specify which is the positive class
POSITIVE_CLASS <- "X1"

# Calculate the confusion matrix
cm_glmnet <- confusionMatrix(data = predictions_glmnet_class, reference = y_test, positive = POSITIVE_CLASS)
print(cm_glmnet)

cat("The accuracy of the glmnet model is", round(cm_glmnet$overall["Accuracy"]*100),"%","\n")
cat("La Sensitivity del modelo glmnet es del", round(cm_glmnet$byClass["Sensitivity"]*100),"%","\n")
```

```
Confusion Matrix and Statistics

          Reference
Prediction X0 X1
        X0 58 17
        X1 19 95

               Accuracy : 0.8095
                 95% CI : (0.7462, 0.8629)
    No Information Rate : 0.5926
    P-Value [Acc > NIR] : 1.598e-10

                  Kappa : 0.6039

 Mcnemar's Test P-Value : 0.8676

            Sensitivity : 0.8482
            Specificity : 0.7532
         Pos Pred Value : 0.8333
         Neg Pred Value : 0.7733
             Prevalence : 0.5926
         Detection Rate : 0.5026
   Detection Prevalence : 0.6032
      Balanced Accuracy : 0.8007

       'Positive' Class : X1

The accuracy of the glmnet model is 81 %
La Sensitivity del modelo glmnet es del 85 %
```

The glmnet model achieves an overall accuracy of approximately 81% on the test set. While this general performance is strong, the client's primary goal is for the model to correctly identify at least 80% of the recipes that will genuinely generate high traffic.

Therefore, a more critical metric for this specific business objective is Sensitivity (also known as Recall or True Positive Rate). Sensitivity measures the proportion of actual high-traffic recipes that our model successfully predicts as high-traffic. For this task, our model demonstrates a Sensitivity of nearly 85%.

This Sensitivity score of 85% directly addresses and surpasses the client's 80% target for identifying popular recipes. It indicates that the model is highly effective at finding the high-traffic recipes we want to promote, which is more directly aligned with the client's request than the overall accuracy alone

```
#####################################################################################
# Training the model with the entire data set, to improve its capacity
#####################################################################################

# Joining the data
full_train <- rbind(x_train, x_test)
full_target <- c(y_train, y_test)
```

Here we simply combine the training and test predictor variables, and we did the same with the target variables.

Then I proceed to define the new control parameters and alpha and lambda, which are much smaller this time. For the control parameters, we won't use cross-validation; the rest is the same as in the previous control parameters we used for this model. And for the alpha and lambda values, we simply used the values that produced the best results in the previous training.

```
# seting the control parametres
fit_control_final <- trainControl(
  method = "none",                    # No cross-validation
  summaryFunction = twoClassSummary,  # For ROC, Sensitivity, Specificity
  classProbs = TRUE,                  # Important! Required for ROC and probabilities
  allowParallel = TRUE                # Allows parallelization if we have multiple cores
)
# obtaining the best parameters
glmnet_grid_best <- expand.grid(alpha = glmnet_model$bestTune$alpha, lambda = glmnet_model$bestTune$lambda)
```

We proceed to train the model

```
# Training the modelentrenando el modelo
glmnet_model_final <- train(
  x = full_train,
  y = full_target,
  method = "glmnet",
  trControl = fit_control_final,
  tuneGrid = glmnet_grid_best,                     # Custom grid
  metric = "ROC",                                  # Optimize by ROC AUC
  preProcess = c("center", "scale")
)

# have an overview of the model's metrics
print(glmnet_model_final)
```

```
glmnet

947 samples
 14 predictor
  2 classes: 'X0', 'X1'

Pre-processing: centered (14), scaled (14)
Resampling: None
```

Summary and Recommendations

The developed glmnet model is well-prepared to enhance the recipe selection process on the Tasty Bytes homepage. It demonstrates a strong capability to identify recipes that will generate high traffic, achieving a ***Sensitivity of 85%*** for predicting these popular recipes.

Considering the historical data point that a popular recipe can boost overall site traffic by up to 40%, implementing this model is projected to yield a significant uplift in web traffic, ***estimated at approximately 34%*** (0.8482 Sensitivity * 40% historical increase). This effectively addresses the dual goals of maximizing popular content display and minimizing the showcase of less engaging recipes.

Therefore, the deployment of this machine learning model offers a data-driven strategy to consistently select engaging content and drive substantial business benefits. We recommend proceeding with its implementation. Looking ahead, to further refine and enhance our predictive capabilities, we recommend the following future steps:

- Continuous Monitoring and Retraining: Regularly monitor the model's performance in production using key metrics (Sensitivity, Precision, actual traffic lift) and retrain it periodically with new data to adapt to evolving user preferences and recipe trends.
- Feature Enrichment: Explore incorporating additional data points that were hinted at in the recipe examples but not available in the current dataset, such as cost per serving, detailed ingredients (potentially using NLP técnicas for feature extraction), and preparation time. These could provide richer signals for predicting popularity.
- A/B Testing and Refinement: Implement A/B testing frameworks to compare the model-driven recipe selection against other strategies, allowing for continuous refinement of the model or a hybrid approach.
- Addressing Underperforming Categories: For categories like 'Beverages' or 'Breakfast' that showed lower inherent traffic generation, consider specific analyses or even separate, tailored models if these categories remain a business priority for homepage features.

✅ When you have finished...

- Publish your Workspace using the option on the left
- Check the published version of your report:
  - Can you see everything you want us to grade?
  - Are all the graphics visible?
- Review the grading rubric. Have you included everything that will be graded?
- Head back to the Certification Dashboard to submit your practical exam report and record your presentation