What eat ? - Professional project - Software Engineering Master degree

Eulalie Gaquer
eulalie.gaquer@etu.u-bordeaux.fr
Jason Fachan
jason.fachan@etu.u-bordeaux.fr

August 31, 2020

Contents

1 Introduction

During the second year of the Software Engineering master degree we are asked to develop a professional project.

We were free to choose our own subject and wanted to develop something meaningful and useful. We thought a lot about it, and one night we realized that we had the same conversation nearly daily "What do you want to eat ?". This is a question that most people have to answer when they want to go to the restaurant or order take out. We then decided that it was a good idea for a mobile application. We choose to develop an app that would ask questions to the user about food (i.e. Do you want to eat vegetables ?) and based on the answers of the user would suggest the best meal.

Our team is made up of two members : Fachan Jason and Gaquer Eulalie.

2 State of the art

The first thing we did for this project was to look for similar applications. We found some quiz applications for finding a character or an object but nothing related to food. The best example of what kind of application we wanted to accomplish is Akinator.

https://fr.akinator.com/.

Figure 1: The main page of the web application Akinator

Akinator is a web application which purpose is to find the character (or object) you are thinking about. It can either be a fictional or non fictional character. Akinator (who is personified by a genie) will ask a set of questions and at the end proposes you the character he thinks you are thinking about.

Figure 2: One question from the web game Akinator

What makes Akinator so popular is the extreme accuracy of his answers and the gigantic database of characters he seems to "know".

3 Literature search

Since our application is very close of the principle of Akinator, we decided to search what kind of algorithm it implements. We did not find a precise answer since the creator of Akinator never reveals it but it gave us some good points to start.

We discovered that this kind of project are based on classification algorithms and decision trees. We selected two classification algorithms which seemed to match our needs : the 20 questions algorithm and the C4.5 algorithm.

3.1 The 20 questions algorithm

The 20 questions algorithm is based on a old parlor game which encourages deductive reasoning and creativity.
In the traditional game, one player is chosen to be the answerer. That person chooses a subject but does not

reveal it to the others. All other players are questioners. They each take turns asking a question which can be answered with a simple "Yes" or "No". The questioners have 20 questions to find the subject otherwise they loose the game.

The algorithm is based on a binary search algorithm. In each iteration, a question is asked, which should eliminate roughly half of the possible word choices. With a total of N words, we should expect to get an answer after log2(N) questions.

With 20 question, we should optimally be able to find a word among $2^{20} = 1$ million words.

3.2 The C4.5 algorithm

C4.5 is an algorithm used to generate a decision tree which can be used for classification.
C4.5 builds decision trees from a set of training data, using the concept of information entropy.
In information theory, the entropy of a random variable is the average level of "information", "surprise", or "uncertainty" inherent in the variable's possible outcomes.

At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets. The splitting criterion is the normalized information gain (the amount of information gained about a random variable from observing another random variable). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurses on the partitioned sub-lists.

4 First steps

After some researches on the algorithms we could implement, we needed to think about the technical aspects of the project (what framework do we want to use, do we need a database, etc). We also needed to think about what features we wanted to develop.

We wrote the project specifications on a very basic document since our group was only made of two people and we did not have a client. We decided to focus the majority of our development around the main idea of the project which is the suggestion of a meal to eat but we also wanted to add additional features :

- The possibility for the users to access a recipe of the suggested meal.
- The possibility for the users to find restaurant serving the suggested meal.
- The possibility for the users to see all the meals stored in the application.

To develop our application we wanted to use *Flutter*.
Flutter is a mobile framework developed by Google which allows developers to code mobile applications for both android and ios devices. The code is written in dart which is a C-style syntax language which can compile to either native code or JavaScript.
We already had some experience using *Flutter* since we used it to develop our client application during our internship. We knew that using it for this project would save us a lot of time.

When we first started to work on this project we were also having the course *Data Analysis and Machine Learning*.
Since this course was closely related to our project, we made the choice to wait for the course to finish before starting our project.

This was a good idea since it allowed us to better understand how machine learning works.
In the TD sessions we also learnt about *Colaboratory* (Colab). *Colab* is a service provided by Google which allows its users to write and execute Python code in the navigator without having to configure a development environment. It also works very well since it allows the users to access an external power of calculation (GPU).
We decided to use *Colab* to develop our model and train it.

During our researches for an algorithm we encountered *Tensorflow*. *Tensorflow* is a free and open-source software library for dataflow developed by Google. It is used for machine learning applications such as neural networks. We dig into detail into this program but could not find a use for anything non related to image recognition. We then decided that it was better to look for another solution.
After some more researches, we found *Scikit-learn* which is a free software machine learning library for Python.
We saw that with *Scikit-learn* we could build a decision tree model to train our dataset which was exactly what we needed.

Once we chose the tools we wanted to use to develop the project, we needed to decide how we were going to provide a list of food for our model to work on.
We first tried to find complete culinary database. We wanted these database to provide the name of the dish but also a list of ingredients and some additional information like the origin of the food (Chinese food, etc). We did not find anything that we were satisfied with. Every database was always missing some information so we decided to build our own database.

We saved the meals displayed in our application and the questions and answers in a spreadsheet using *Google spreads sheets* and we compiled around 150 meals.

Link to the the meals spreadsheet.

5 Development

Once we had set up our tools, we could start to dig into more details and specific parts of the project.

We first started to fill the meals spreadsheet with meals and added new questions when we did not have a way to distinguish two (or more) meals.
We compiled around 150 meals which is a good start for our application even if it's not much to run an algorithm.
We also wrote around 30 questions to be sure that the application can differentiate all the meals. We have 3 possible answers to each questions : "no", "yes" and "yes and no". This last answer is a choice if the meal can be made with or without a particular ingredient.

Food	FoodId	Does it contain cheese ?	Does it contain dought (ie : bread) ?	Does it contain vegetables ?	Is it healthy ?	Is there a soup ?	Is it cooked in hoven ?	Is it a hot meal ?	Vegetarian ?	Is there sea food or fish ?
Pizza	0	yes	yes	yes	no	no	yes	yes	no	yes and no
Hamburger	1	yes	yes	yes	no	no	no	yes	no	no
Pasta	2	yes	no	yes	yes and no	no	no	yes	yes and no	yes and no
Hot dog	3	yes and no	yes	no	no	no	no	yes	no	no
Salad	4	yes and no	no	yes	yes	no	no	no	yes and no	no
Ramen	5	no	no	yes	yes and no	yes	no	yes	no	no
Gratin dauphinois	6	no	no	no	no	no	yes	yes	yes	no
vichy carrots	7	no	no	yes	no	no	no	yes	yes	no
Minestrone	8	yes	no	yes	yes	yes	no	yes	yes and no	no
Pho	9	no	no	yes	yes	yes	no	yes	no	no
Couscous	10	no	no	yes	yes	no	no	yes	yes	no
Paella	11	no	no	yes	no	no	no	yes	no	yes
Vegetable soup	12	no	no	yes	yes	yes	no	yes	yes	no
Spaghetti bolognese	13	yes and no	no	yes	no	no	no	yes	no	no
Fried Spring Rolls	14	no	no	yes	no	no	no	yes	yes and no	no
Chop Suey	15	no	no	yes	yes	no	no	yes	no	yes and no
Tartiflette	16	yes	no	no	no	no	yes	yes	no	no
Tonkatsu	17	no	no	yes and no	no	no	no	yes	no	no
Samosas	18	no	no	yes	no	no	no	yes	yes and no	no
Chili con carne	19	no	no	yes	no	no	no	yes	no	no
Burritos	20	yes and no	no	yes	no	no	no	yes	no	no
Fajitas	21	yes and no	yes	yes	no	no	no	yes	yes and no	no
Enchiladas	22	yes and no	no	yes	no	no	yes	yes	no	no
veal stew	23	no	no	yes	no	no	no	yes	no	no
Bánh bao	24	no	yes	no	no	no	no	yes	no	no
Lasagna	25	yes	no	yes	no	no	yes	yes	no	no
Fried noodles	26	no	no	yes	no	no	no	yes	yes and no	no
Korean BBQ	27	no	no	no	no	no	no	yes	no	no
Crumble of tomatoes	28	no	yes	yes	yes	no	yes	yes	yes	no
Stir mine	29	no	no	yes	no	no	no	yes	no	yes
Cantonese rice	30	no	no	yes	yes	no	no	yes	no	no
Sushis	31	no	no	yes	no	no	no	no	no	yes
Kebab	32	no	yes	yes	no	no	no	yes	no	no

Figure 3: The meals spreadsheet in Google spread sheets

5.1 The model

To buid the core of our algorithm, we needed to implement a decision tree. Instead of doing it from scratch, we decided to use *Scikit-learn* which is a Python library for machine learning.
Scikit-learn allowed us to easily use the class *DecisionTreeClassifier()* which build an object representing a classification tree (discrete values). Here the decision tree acts like a predictive model to go from our questions about the meals (basically the branches of the tree) to the more adequate meal (the leaves of the tree).

```
[ ]  model = tree.DecisionTreeClassifier()
```

```
[ ]  model.fit(inputTree, targetId)
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

the score represent the accuracy of the tree, since some foods have the same answer the score may not be 100%

```
[ ]  model.score(inputTree,targetId)
```

```
0.9721448467966574
```

Figure 4: The creation of the decision tree in *Colab*

Here, we first create our object *TreeClassifier*.

```
[ ]  model = tree.DecisionTreeClassifier()
```

Then we train the tree on our dataset.

```
[ ]  model.fit(inputTree, targetId)

     DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                            max_depth=None, max_features=None, max_leaf_nodes=None,
                            min_impurity_decrease=0.0, min_impurity_split=None,
                            min_samples_leaf=1, min_samples_split=2,
                            min_weight_fraction_leaf=0.0, presort='deprecated',
                            random_state=None, splitter='best')
```

And finally, we compute the score of the model.

```
[ ]  model.score(inputTree,targetId)

     0.9721448467966574
```

The score represents the accuracy of the tree. The best score is 1 since it means that all the data (meals) in the dataset are distinguishable. On this example, our score is 0.98 which means that with the exact same set of questions and answers, the tree can produce two (or more) different outputs (meals). To prevent it, we can add more questions to the set (on the condition that these questions have different answers for the two or more meals).

We then need to save our tree to be able to use it in the application. To do it, we export it in the json format.

```
[ ]  lazy = LazyExport(model)
     lazy.save("what_eat/assets/decisionTree.json",indent='  ',force_override=True)
     treeDict = lazy.build()
     print(dict)
```

To be able to export the tree, we use another Python library : *Sklite*. *Sklite* allows us to export models from *Scikit-learn* into *json* format. This library has been built purposely for the dart language (the language of *Flutter*).

Once this is done we need to access all these files from our application. We cannot let the *Colab* document in our Google Drive folder since the application will not be able to reach it.
The solution we came with was to create a *Firebase* database to store our model and our dataset.
Firebase is a set of hosting services developed by Google and working for any type of application (Android, iOS, Javascript, ...). It is used with NoSQL and allows its users to use real-time database, notifications or authentication services using Google, Facebook, Twitter and Github.

We first needed to link the *Colab* document and our *Firebase* database. We did it in *Colab*.

We had to provide to *Colab* our *Firebase* admin credentials. This way, the *Admin SDK* of *Firebase* will allow the connection between *Colab* and our database.


```
{
  "type": "service_account",
  "project_id": "project-food-892bf",
  "private_key_id": "e3914e7af66707db692ed728359eded7e24cbfe8",
  "private_key": "-----BEGIN PRIVATE KEY-----\nMIIEvAIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiAgEAAoIBAQCl6mx13TfqGVED\nFfAjxWcaeF4GKx4I8yK/Dg9pwZTdsBwmH
  "client_email": "firebase-adminsdk-u27ki@project-food-892bf.iam.gserviceaccount.com",
  "client_id": "109572968032968272543",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-u27ki%40project-food-892bf.iam.gserviceaccount.com"
}
```


Figure 5: The credentials of the *Firebase* database

Then we had to connect the *Colab* file to the *Firebase* database. Our database is declared as "*db*".

```
[ ]  # Use the application credentials to connect to the database
     cred = credentials.Certificate("cert.json")
     firebase_admin.initialize_app(cred, {
         'projectId': "project-food-892bf",
     })
     db = firestore.client()
```

Figure 6: The connection to the *Firebase* database

Finally, we used this connection to store the tree into *Firebase*. We stored the information in different parts : the tree in *json* format, the depth of the tree (i.e. the maximum number of questions to return a meal), the number of leaves of the tree (i.e. meals) and the time when the tree was uploaded to the database.

```
[ ]  model_ref = db.collection(u'model')
     print(treeDict)
     model_ref.document(u"tree").set({
         u'jsonTree' : u"{}".format(treeDict),
         u'depth' : int(model.get_depth()),
         u'leaves' : int(model.get_n_leaves()),
         u'time' : datetime.datetime.now()
         }, merge=True)
```

Figure 7: The storage of the tree in the *Firebase* database

5.2 The dataset

Having the spreadsheet on Google drive is a convenient way to share it but to be able to use it with our project, we needed to export it as a *csv* file. Once done, we stored the *csv* file into our *Github* repository

and implemented into *Colab* a github connexion. That way, the model built in *Colab* was able to run on the data stored into *Github*.

First we needed to import the data stored into the *csv* file (*foods.csv*). *Scikit-learn* cannot read these data directly. To be able to use them, we needed to convert them into dataframes.
Dataframes are some kind of Python dictionary where the keys are the names of the columns and the values are the series.
Dataframes are special types from the Python library *Pandas*. *Pandas* is a special tool that allows the developers to easily manipulate data to analysed.

```
[ ]  raw_data = pd.read_csv("database/foods.csv", skiprows= [0])
     df = pd.read_csv("database/foods.csv", skiprows= [0])
     df.sort_values(by=['FoodId'])
     df.head(11)
```

Figure 8: Importation of the dataset from the *csv* file

We also needed to store our meals into the *Firebase* database. We needed to store both the answers to the questions for each meals but also the names and ids of them.
To have an efficient training for the tree, we will later only keep the ids of the meals. To retrieve their names in order to communicate them to the users we kept them into our database.

```
def updateFoodToFirebase(dataframe):
  foodDict = {}
  ToChange = []
  for i, foodId in enumerate(dataframe['FoodId']):
    food = dataframe.loc[dataframe["FoodId"] == foodId]
    id = food["FoodId"].values[0]
    label = food['Food'].values[0].capitalize()
    foodDict[id] = {}

    foodDict[id]["FoodId"] = id
    foodDict[id]["label"] = label
    foodDict[id]["inside"] = False

  docs = food_ref.stream()
  for doc in docs:
    foodDict[int(doc.id)]["inside"] = True
    foodData = doc.to_dict()
    #if the label is modified we update the data
    if (foodDict[int(doc.id)]["label"] != foodData["label"]):
      ToChange = ToChange + [int(doc.id)]

  for i, id in enumerate(foodDict.keys()):
    if (foodDict[id]["inside"] == False or id in ToChange):
      foodData = {}
      foodData["label"] = foodDict[id]["label"]
      food_ref.document(u"{}".format(id)).set(foodData)
      print("{} : {}  modified".format(id, foodDict[id]["label"]))


updateFoodToFirebase(df)
```

Figure 9: Function to store the meals into the *Firebase* database

Since the decision tree is a binary tree, we needed to manipulate our dataset to obtain only "yes" and "no" possibilities (for now we also have a "yes and no" option).
To do it, we needed to duplicate the row with a yes and no option.

For example , for this array :

id	Question 1	Question 2
0	yes and no	yes
1	no	yes and no

We have two "yes and no" options that we want to get rid of, so we will duplicate the array like this :

id	Question 1	Question 2
0	yes	yes
0	no	yes
1	no	yes and no
1	no	yes and no

For each column with a “yes and no” option, the column will be duplicate to provide one row with the “yes” option and another with the “no” option.

We did not find how to only duplicate the row we needed to. So our algorithm duplicate all the rows of the array and then delete all the duplicate. So we end up with this result :

id	Question 1	Question 2
0	yes	yes
0	no	yes
1	no	yes and no

And we continue to do so for every column of the dataset.

```
[ ]  print("start : df of size {}".format(df["Food"].size))
     for index, column in enumerate(df.columns) :
       yes = pd.DataFrame()
       for i, columns in enumerate(df.columns):
         yes[columns] = df[columns]
       yes.loc[yes[column] == 'yes and no', column] = "yes"
       df.loc[df[column] == 'yes and no', column] = "no"
       df = df.append(yes)
       df = df.drop_duplicates(df.columns, ignore_index=True)
       print("{} : df of size {}".format(index, df["Food"].size))


     print("end : df of size {}".format(df["Food"].size))

     df.describe(exclude=[np.number])
```

Figure 10: The function to double the ”yes and no” lines and get rid of the duplicates

This was not the most effective way to do it but it worked and we were sure not to eliminate important information by mistake.

Once this was done, we needed to verify that every row in the dataset was unique, i.e. that for every meal there is a unique set of answers.

```
[ ]  duplicates = df[df.duplicated(df.columns[2:],keep=False)]

     duplicates.describe(exclude=[np.number])
     drop = []
     for column in duplicates.columns[2:]:
       if len(duplicates[column].unique().tolist()) == 1:
         drop.append(column)
     duplicates = duplicates.drop(columns= drop)
     value = duplicates[duplicates.columns[2]].copy()
     for column in duplicates.columns[3:]:
       value += duplicates[column]
     duplicates['value'] = value
     duplicates = duplicates.sort_values("Food")
     duplicatesGroup = duplicates.groupby("value")
     if len(duplicatesGroup.groups) > 0 :
       print('you need a new question for differenciate :')
       for group in duplicatesGroup.groups :
         same = duplicatesGroup.get_group(group)
         names = same['Food'].tolist()
         print('\t- {}'.format(names))
     else :
        print('Great ! you have enouth questions to differentiate all the food')
```

Figure 11: The function verify that each row of the dataset is unique

After that, we converted all the answer into integer. The “yes” option became a 1 and the “no” option became a 0.

```
[ ]  for i, column in enumerate(df.columns):
       if(i != 0):
         dataset_bin[column] = df[column]
         dataset_bin.loc[dataset_bin[column] == 'no', column] = 0
         dataset_bin.loc[dataset_bin[column] == 'yes', column] = 1
```

Figure 12: The function to convert each String option into a binary one

Finally, we deleted the labels (meals names) from the dataset to keep our data as clear and minimal as possible.
This is important because we only want our model to train on the questions and answers of the dataset and not other non relevant information.

```
[ ]  inputTree = dataset_con.drop(['Food','FoodId'], axis=1)
     inputTree
```

Figure 13: We delete the labels of the meals from the dataset

Once the model and dataset were clear and set up. We could (in our application) use our model to predict

a meal. The tree will return the id of the meal he finds and we can look into the *Firebase* database which meal it corresponds to.

5.3 The Flutter code

Now that our dataset was cleaned and our tree built, we could start to work on linking our flutter application with our model.

To do it, we used the package *Sklite* for *Flutter*. This package allowed us to re-built the tree from the *json* file stored in the *Firebase* database into our *Flutter* application.

```
treeClassifier = Future.value(firestore).then((FirebaseFirestore instance){
  return instance.collection(modelCollectionName).doc(modelDocumentName).get().then((DocumentSnapshot doc){
    print("doc getted ");
    Map<String,dynamic> treeData = doc.data();
    String jsonString = treeData["jsonTree"].replaceAll("'", "\"");
    Map<String,dynamic> jsonTree = jsonDecode(jsonString);
    print('tree ${jsonTree.keys}');

    var tree =  DecisionTreeClassifier.fromMap(jsonTree);
    return tree;
  });
});
```

Figure 14: Dart function to build the tree from the *json* file

Once this was done, we did the same for the questions and answers.
We will use these questions to obtain a "list" of answers defining a meal. For example, the list : 0,1,1,0,0,0,1,0,1,1 will match the meal "pizza". We will then submit this list to our tree which will return the id of the meal. So here if the id of pizza is 15, the tree will return 15.

```
questions = Future.value(firestore).then((FirebaseFirestore instance){
  return instance.collection("questions").get().then((docs){
    List<dynamic> questionsDoc = docs.docs;
    Map<String,dynamic> questions = {};
    nbQuestions = questionsDoc.length;
    input = new List(nbQuestions);
    for(int i=0; i< nbQuestions;i++){

      QueryDocumentSnapshot currentDoc = questionsDoc[i];
      questions[currentDoc.id] = currentDoc.data();

    }
    print("question : ${questions.keys.length} ok");
    return questions;
  });
});
```

Figure 15: Dart function to retrieve the questions

When we retrieved the answers, we also stored the list of all the meals in a global variable. We will later skim this list to obtain only one meal, the one to return to the user.

```
foods = Future.value(firestore).then((FirebaseFirestore instance){
  return instance.collection("foods").get().then((docs){
    List<dynamic> foodDocs = docs.docs;
    possibleFoodId = [];
    Map<String,dynamic> foods = {};
    nbFood = foodDocs.length;
    for(int i=0; i< nbFood; i++){
      QueryDocumentSnapshot currentDoc = foodDocs[i];
      foods[currentDoc.id] = currentDoc.data();
      possibleFoodId.add(int.parse(currentDoc.id));
    }
    print("foods : ${foods.keys.length} ok");

    return foods;
  });
});
```

Figure 16: Dart function to retrieve the answers

After that we needed to determine in which order the questions (obtained with *Firebase*) will be asked to the user and more precisely which question should be chosen after the previous question.
This part is very important since it allowed us to optimize the quickness of our algorithm. We needed it to quickly find the more suitable meal based on the questions and their answers.

To do so, we compared each questions and most importantly their answers. We needed to take the questions whose answers's division were the closest to 50/50 (50% of the answers were "yes" and 50% of the answers

were “no”). We chose this division because it was the most efficient. Indeed, with this division we could cut roughly 50% of the remaining possibilities (i.e. meals). In fact, if we decided to took a division of 30% of “yes” and 70% of “no”, if the user chose “yes”, we could eliminate a larger part of the remaining possibilities but if he chose “no”, we eliminated a smaller part.

However, it was possible to eliminate all the possibilities and end up without any meals left. In that case, the next question chosen will be the one with the higher information gain (computed by *Scikit-learn* in *Colab*).

```
Future nextQuestion() async {
  print("getting next question ... ");
  Map<String, dynamic> myQuestions = await Future.value(this.questions);
  int next_key = 0;
  double importance = 0.0;
  double distribution = 0.0;
  //get the next question more important not ansered
  myQuestions.forEach((key,value) {
    Map<String, dynamic> questionData = value;
    double feature_importance = questionData['importance'];
    dynamic yesValues = questionData['yes'];
    dynamic noValues = questionData['no'];
    yesValues.removeWhere((value) =>  ! possibleFoodId.contains((value)));
    noValues.removeWhere((value) =>  ! possibleFoodId.contains((value)));
    int nb_yes= yesValues.length;
    int nb_no = noValues.length;
    double feature_distribution = 0.0;
    if(nb_no > nb_yes && nb_no>0)
      feature_distribution = nb_yes / nb_no;
    if(nb_yes > nb_no && nb_yes>0)
      feature_distribution = nb_no / nb_yes;


    print('${nb_yes} yes and ${nb_no} no');

    if (input[int.parse(key)] == null && ((feature_distribution != 0 && feature_distribution > distribution)
        || (feature_distribution == 0 && feature_importance > importance) )){
      distribution = feature_distribution;
      importance = feature_importance;
      next_key = int.parse(key);
    }
  });
  Map<String, dynamic> feature = myQuestions[next_key.toString()];
  String question = feature['name'];
  print(question);
  print("\nthe question : $question \nhave an importance of $importance and a current distribution of $distribution\n");

  return setState(() {
    currentFeatureIndex = next_key;
    currentQuestion = question;
  });
```

Figure 17: Dart function to order the questions

When a question is chosen, the user will respond to it (with “yes” or “no”). We then proceed to delete all

the possibilities which do not match the answer. For example if the question is “Do you want french food” and the answer is “yes”, we will delete all the meals for which the answer to this question is “no”.

One of the possibility was that a particular set (or subset) of answers only match one meal. In that case, we did not need to ask all the questions of the dataset to return the meal.

```
Future setValue(double response) async {
  print('set $currentQuestion to ${response == 1 ? "yes" : "no" }');
  input[currentFeatureIndex] = response;
  Map<String, dynamic> myFoods = await Future.value(this.foods);
  Map<String, dynamic> myQuestions = await Future.value(this.questions);
  Map<String, dynamic> currentQuestionData = myQuestions[currentFeatureIndex.toString()];
  if(response == 1){
    //if the response of the question is yes we remove the "no" of the possible responses
    possibleFoodId.removeWhere((element)=> currentQuestionData["no"].contains(element));
  }else{
    //if the response of the question is no we remove the yes of the possible responses
    possibleFoodId.removeWhere((element)=> currentQuestionData["yes"].contains(element));
  }
  infos = "the current possible food are :\n ${possibleFoodId.map((e) => myFoods[e.toString()]["label"]).toList().join(", ")}";
  if(possibleFoodId.length == 1){
    found(possibleFoodId[0]);
  }else{
    testValue();
  }
}
```

Figure 18: Dart function to retrieve the answers

We were to repeat this step until there were only one meal left in the list of possible meals and we then returned it.
This can be very quick (i.e. only ask a small number of questions) if the answers given only define one meal.

Since our questions and answers do not cover all the possible combinations, it is possible that we end up with no meal for our set of answers. In that case, we need to find another relevant question to ask to still end up with a meal.

These cases are when we take the predictions into account.

The predictions are a list of all the possible outputs for the list of answers. These are built before the user respond to all the questions. For example, if the user needs to answer 3 questions but had only answer one question so far (“yes”), the predictions will take his answer into account and generate all the possible outputs based on it. So we will have :

- yes, yes, yes
- yes, yes, no
- yes, no, yes
- yes, no, no

With the last two questions completed by the predictions.

When the user answer a new question, the predictions are re-generate to match the new list of answers. With our previous example, if the user answer “no” to the second question, the predictions become :

- yes, no, yes
- yes, no, no

```
Map<int, double> results = {};
predictions.forEach((element) {
  int output = tree.predict(element);
  if (results[output] == null) {
    results[output] = (100 / nb_case);
  } else {
    results[output] += (100 / nb_case);
  }
});
print(results);
if (results.length == 1) {
  found(results.keys.first);

}else{
  proba = results;
  sortProba();
  nextQuestion();
}
```

Figure 19: Dart function which calculates if the lists of predictions all return the same meal

For each set of possible answers in our list of predictions, we will look at what meal it can possibly match. If all the set of answers return the same meal, we will return it.
Otherwise (if the set of answers match two or more meals) we will need to answer one more question. We repeat it until we have a match (for a meal).

For example :

set of answers	Q3	Q4	Predictions of the model
no yes no	yes	no	Meal 1
no yes no	yes	yes	Meal 1
no yes no	no	no	Meal 1
no yes	no no	yes	Meal 2

In that case, the set of answers does not return the same meal. So we need to ask another question to the user.
We will consider that the answer to the third question to be “yes”.

set of answers	Q3	Q4	Predictions of the model
no yes no	yes	no	Meal 1
no yes no	yes	yes	Meal 1

Now, all the set of answers return the same meal (Meal 1) so we do not need to respond to question 4 and we can return Meal 1.

Now that we have our result (meal), we can return it to the user. We will return from the *Firebase* database the name of the meal and the pictures which describe it by providing its id to the database.

6 The application

6.1 The home page

When the user launched the application, it opens the home page. The home page is very simple and composed of two main buttons : “Find a meal” and “List all the meals”.

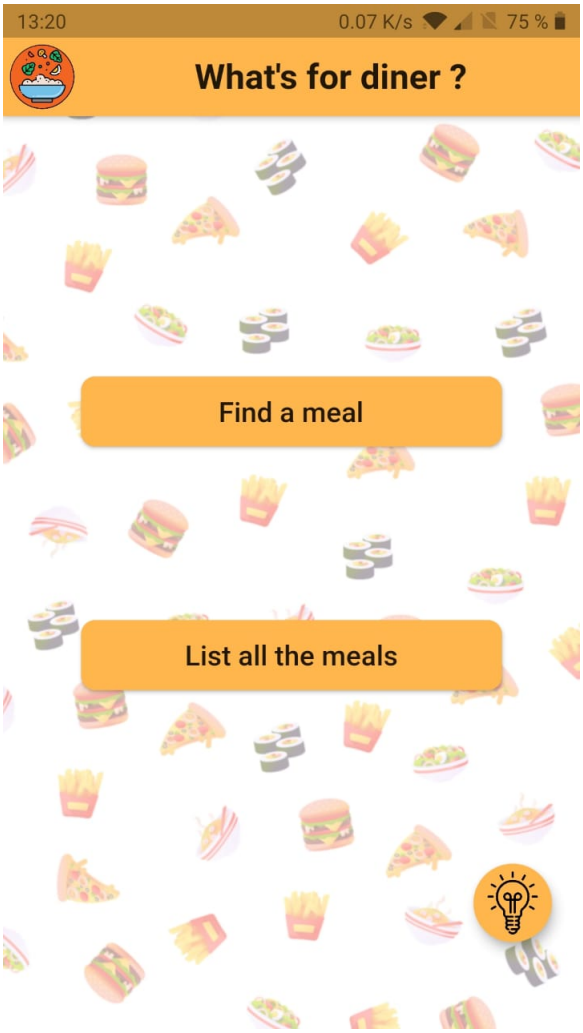

Figure 20: The home page of the application

The first button allows the user to access the meal quiz where he will find out what he should cook or order.

The second button opens another page where are listed all the meals of the application.

The last button (with the light-bulb) is a switch with allows the user to switch to a dark background if he feels that the light one is too bright.

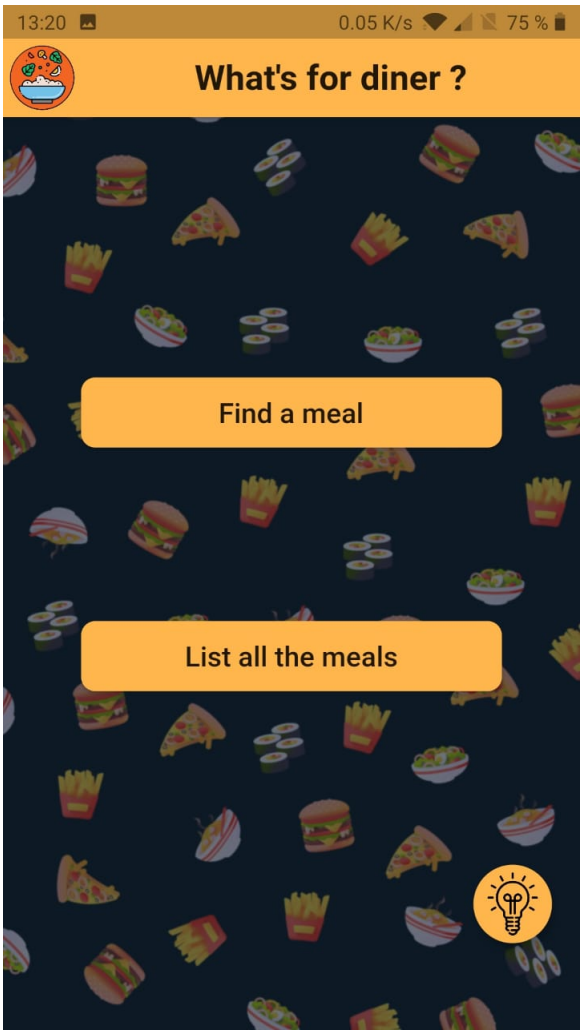

Figure 21: The home page of the application with the dark background

6.2 The quiz page

After pressing the button “find a meal”, the user access the quiz page. In this page he will have to answer some questions about the food he wants to eat in order for the application to suggest him a meal.

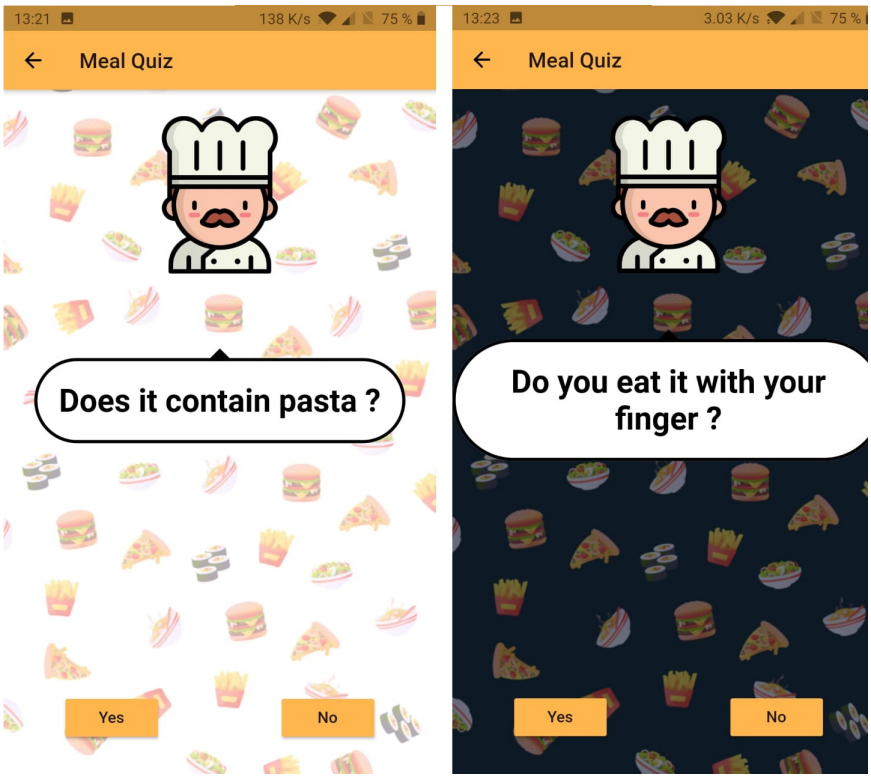

Figure 22: The page quiz with both the light and dark backgrounds

If the user chooses the dark background on the home page, the background will be applied to all the pages he visits until he presses the button again (to switch back to the light theme).

6.3 The result page

Once the user finish to answer the questions, the result page open.
The result page display the meal selected by the algorithm to the user and some pictures to illustrate the meal.

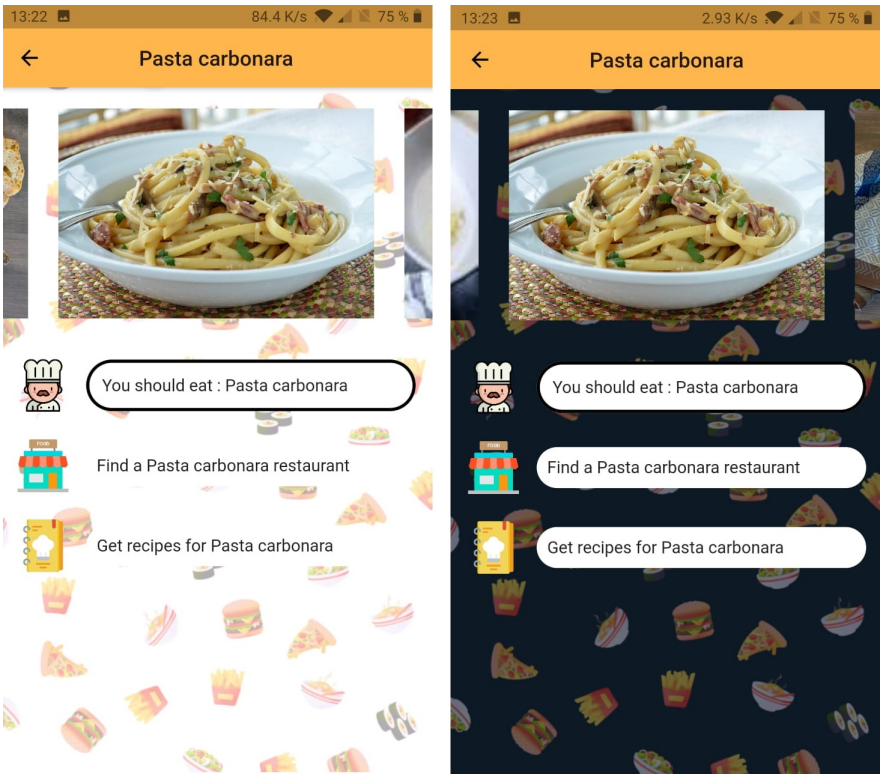

Figure 23: The result page with both the light and dark background

On this page, the user has three choices. If the meal does not suit him, he can go back to the home page and take the quiz again. If the meal does suit him, he can either look for a restaurant that cook this meal or access a recipe to cook it himself.

If he chooses the restaurant, he needs to click on the "Find a *name of the meal* restaurant" and he will be taken to a map which display all the restaurants serving this kind of meal.

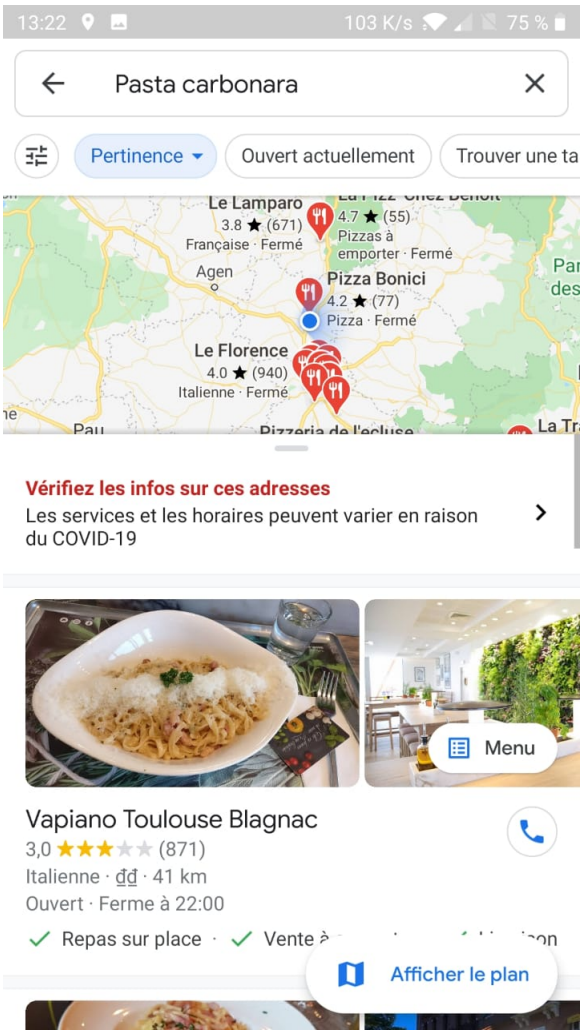

Figure 24: The map showing the user the restaurants cooking the meal

If he prefers to cook himself, he can press the "get recipes for *name of the meal*" option and he will access an internet page, displaying a list of recipes for this meal.

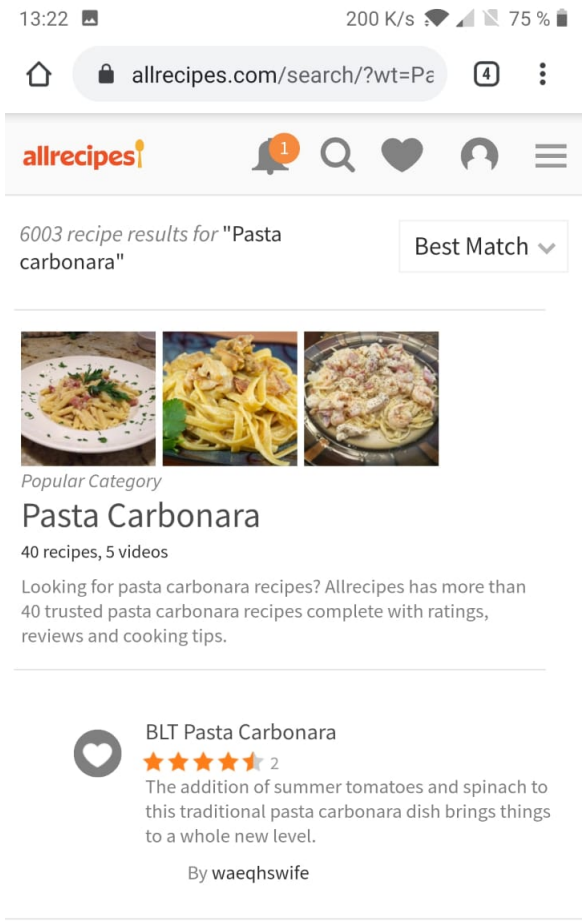

Figure 25: The internet page displaying a recipe of the meal

6.4 All the meals page

If the user wants to see all the meals stored in the application, he can access the page “list all the meals”.

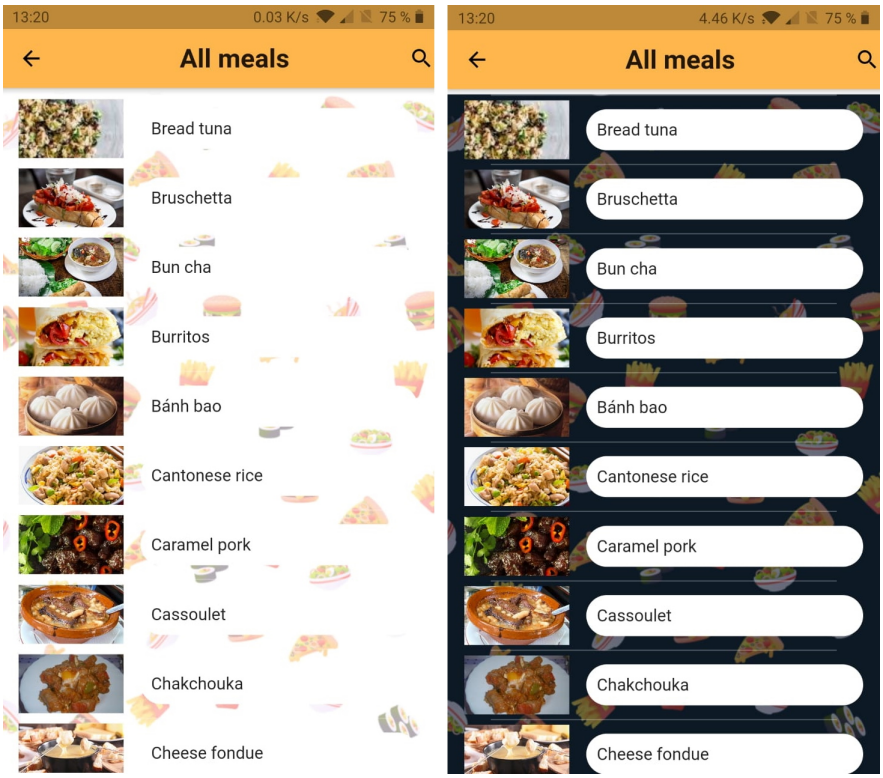

Figure 26: The page which display all the meals with both light and dark backgrounds

In this page the user can scroll through all the meals of the application or he can search for a particular meal.

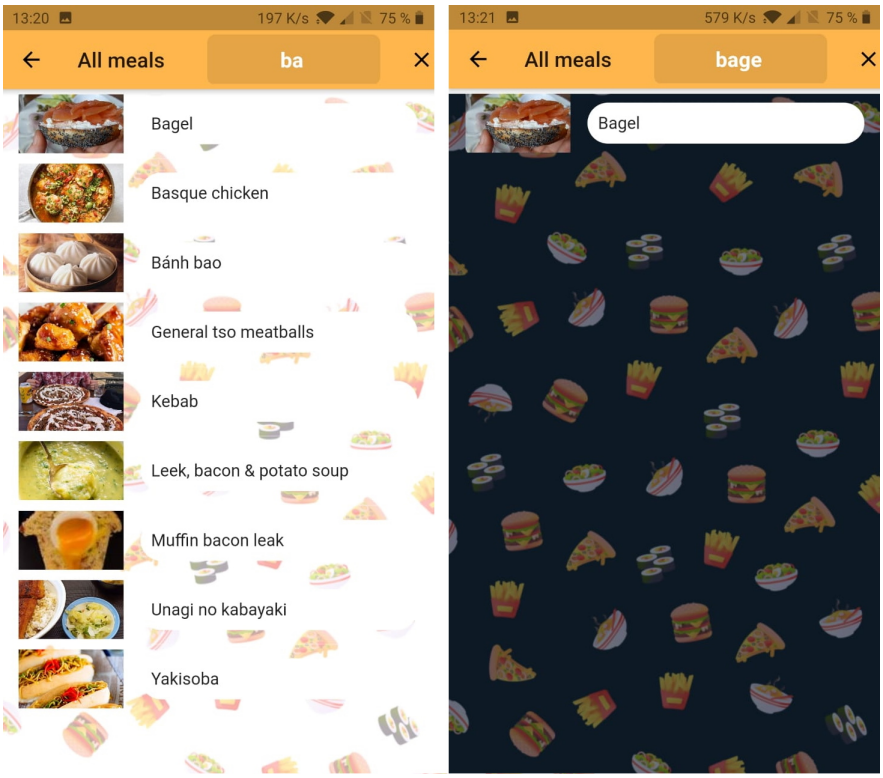

Figure 27: The user can search for a particular meal

7 Conclusion

This project allowed us to discover a new aspect of computer science that we only scratch last year which is machine learning.
We were able to directly put our new knowledge (gained from the *Data Analysis and Machine Learning* course) into a practical project.
In addition to theoretical skills, we also learnt technical ones. This project was also the occasion to discover and deepen new tools such as *Colab* or specialised Python libraries (*Scikit-learn*, ...).
Finally we were glad to develop a useful application.