Data Science for Business
What's Cooking Kaggle competition

Radics, Attila

March 7, 2016

Contents

1 What's Cooking

In this Kaggle competition the task is to predict a dish cuisine based on it's ingredients.
There are 20 cuisines in the training set, so this task is a 20 label classification problem.

It is important to note that I joined this competition after it has ended.

2 Data preparation

2.1 Loading

The data is stored in a JSON format. We can load it with the jsonlite::fromJSON function.

```
data.train = fromJSON(file.path(data.root, "train.json"))
data.test = fromJSON(file.path(data.root, "test.json"))
```

2.2 Cleaning

Some of the ingredients are containing special characters.
The CleanIngredients function removes every non-ASCII character, every number and replaces spaces with a single "_" character.

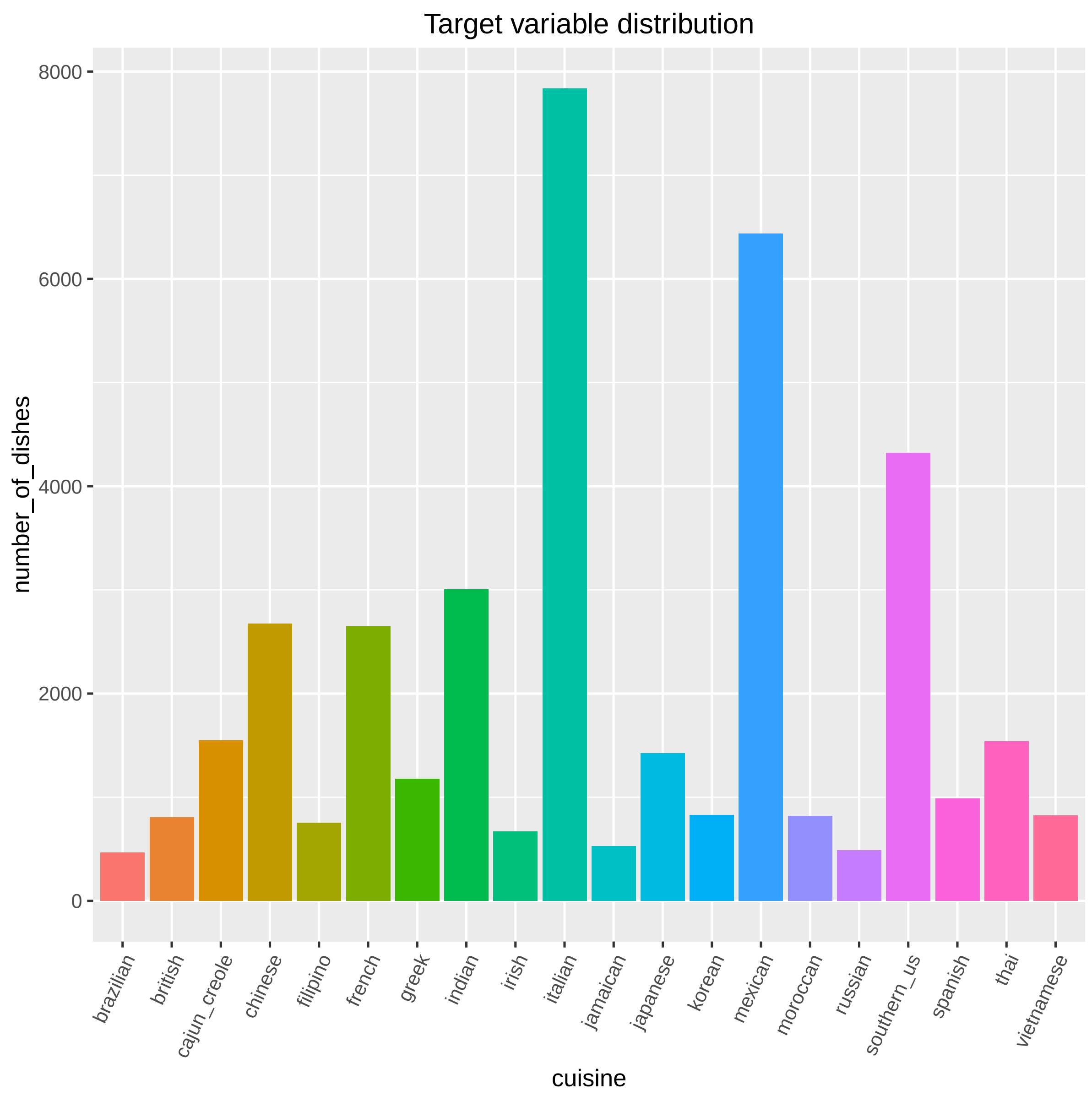
```
CleanIngredients = function(ingredients){
        ingredients = iconv(ingredients, "UTF-8", "ASCII", sub="")
        ingredients = tolower(ingredients)
        ingredients = gsub("[^a-z]", " ", ingredients)
        ingredients = gsub("\\s+", " ", ingredients)
        ingredients = gsub("^\\s+", "", ingredients)
        ingredients = gsub("\\s+$", "", ingredients)
        ingredients = lapply(ingredients, function(ingredient){
                                parts = strsplit(ingredient, " ")[[1]]
                                parts = paste0(wordStem(parts), collapse = "_")
                                gsub("_+", "_", parts)
                        })
        do.call("c", ingredients)
}
```

2.3 Exploration

Due to the nature of the problem there wasn't much room for exploratory analysis.

One thing we can explore is the cuisine distribution:

cuisine	number_of_dishes	percentage
greek	1175	2.95%
southern_us	4320	10.86%
filipino	755	1.9%
indian	3003	7.55%
jamaican	526	1.32%
spanish	989	2.49%
italian	7838	19.71%
mexican	6438	16.19%
chinese	2673	6.72%
british	804	2.02%
thai	1539	3.87%
vietnamese	825	2.07%
cajun_creole	1546	3.89%
brazilian	467	1.17%
french	2646	6.65%
japanese	1423	3.58%
irish	667	1.68%
korean	830	2.09%
moroccan	821	2.06%
russian	489	1.23%



We can see from the table that the distribution is not uniform, italian is the most common.
We can expect a similar distribution in the test set; the ”all italian submission” gives a score of 0.19268.

3 Feature engineering

3.1 All ingredients

The AllIngredients function creates a new ingredient vector containing both the original multi-part ingredients and their split versions.

```
AllIngredients = function(ingredients){
        ingredient.parts = do.call("c", strsplit(ingredients, "_"))
        c(ingredients, ingredient.parts)
}
```

3.2 Filter Ingredients

The FilterIngredients function helps us to select a subsection of the ingredients.
The first option lets us specify the minimum length of an ingredient (could be useful for filtering out *a*,

an, oz, etc...)

```
FilterIngredients = function(ingredients, min.length, min.occurance){
        if(min.length > 1){
                ingredients = ingredients[nchar(ingredients) >= min.length]
        }
        if(min.occurance > 1){
                ingredients = data.table("ingredient" = ingredients)
                ingredients = ingredients[, .N, by = ingredient]
                setkey(ingredients, ingredient)
                ingredients = ingredients[N > min.occurance][["ingredient"]]
        }
        sort(unique(ingredients))
}
```

3.3 Cuisine Membership

The IngredientCuisineMembership function creates a matrix.
Each row of the matrix corresponds to one unique ingredient and it creates one column for each cuisine and an extra Total column.
In each cell we can see the number of occurrence (*e.g. the cell in the "apple" row and "greek" column shows us how many times does a greek dish contains apple as an ingredient in our training set*)

```
IngredientCuisineMembership = function(data, all.ingredients){
        uniques = sort(unique(all.ingredients))
        cuisines = c("total", sort(unique(data$cuisine)))
        membership = matrix(0,
                        ncol = length(cuisines),
                        nrow = length(uniques))
        colnames(membership) = cuisines
        rownames(membership) = uniques
        lapply(cuisines, function(cuisine) {
                        if(cuisine == "total"){
                                by.cuisine = data$ingredients
                        } else {
                                by.cuisine = data$ingredients[data$cuisine == cuisine]
                        }
                        by.cuisine = do.call("c", by.cuisine)
                        by.cuisine = by.cuisine[by.cuisine %in% uniques]

                        by.cuisine = table(by.cuisine)
                        membership[names(by.cuisine), cuisine] <<-
                                as.vector(by.cuisine)
                        NULL
                })
        membership
}
```

3.3.1 Membership Ranks

From the membership matrix create a matrix containing the ranks by columns.

```
CalculateMembershipRanks = function(membership){
        colnames(membership) = paste0("rank_", colnames(membership))
        apply(membership, 2, rank)
}
```

3.3.2 Membership Exclusives

From the membership matrix create two boolean matrices.
The first ”only” matrix shows us if the given ingredient is exclusive for a cuisine.
The second ”never” matrix shows us if the given ingredient is never present in a cuisine.

```
CalculateMembershipExclusives = function(membership){
        total = membership[,1]
        only = membership[,-1]
        never = membership[,-1]

        colnames(only) = paste0("only_", colnames(only))
        only = only - do.call("cbind", rep(list(total - 1), ncol(only)))
        only[only < 0] = 0

        colnames(never) = paste0("never_", colnames(never))
        never = (never * -1) + 1
        never[never < 0] = 0
        cbind(only, never)
}
```

3.4 Normalization Parameters

The CalculateNormalizationParameters calculates the normalization parameters which can be used to transform each column to have mean of 0 and sd of 1.

```
CalculateNormalizationParameters = function(features){
        means = apply(features, 2, mean)
        sds = apply(features, 2, sd)
        return(list(mean = means, sd = sds))
}
```

The NormalizeFeatures function performs the normalization with the specified parameters.

```
NormalizeFeatures = function(features, normalization.parameters){
        means = normalization.parameters$mean
        sds = normalization.parameters$sd
        lapply(seq(features), function(i) {
                                features[[i]] <<- (features[[i]] - means[i]) / sds[i]
                        })
        features
}
```

3.5 Ingredients Matrix

The CalculateIngredientsMatrix function calculates a boolean matrix.
One row is created for each training element and the matrix has a column for each ingredient selected for training.
A cell in the matrix is 1 if in the current dish (row) the specified ingredient (column) is present and 0 otherwise.

```
CalculateIngredientsMatrix = function(data, feature.ingredients){
        ingredients.matrix = lapply(data$ingredients, function(ingredients) {
                                as.integer(feature.ingredients %in% ingredients)
                        })
        ingredients.matrix = do.call(rbind, ingredients.matrix)
        colnames(ingredients.matrix) = feature.ingredients
        ingredients.matrix
}
```

3.6 Final features

The final features are calculated using the CalculateFeatures function.

```
CalculateFeatures = function(data, all.ingredients, feature.ingredients,
                membership = NULL, membership.rank = NULL,
                membership.exclusive = NULL, normalization.parameters = NULL) {

        if(is.null(membership)){
                membership = IngredientCuisineMembership(data, all.ingredients)
                membership.rank = CalculateMembershipRanks(membership)
                membership.exclusive = CalculateMembershipExclusives(membership)
        }

        features = lapply(data$ingredients, function(ingredients) {
                                idx = which(rownames(membership) %in% ingredients)
                                exclusives = membership.exclusive[idx, , drop = FALSE]
                                ranks = membership.rank[idx, , drop = FALSE]

                                c(
                                                apply(exclusives, 2, sum),
                                                apply(ranks, 2, min),
                                                apply(ranks, 2, max),
                                                apply(ranks, 2, mean),
                                                apply(ranks, 2, sum)
                                )
                        })
        features = as.data.frame(do.call(rbind, features))

        setDT(features)
        prefix = c(
                        rep("sum_", ncol(membership.exclusive)),
                        rep("min_", ncol(membership.rank)),
                        rep("max_", ncol(membership.rank)),
                        rep("mean_", ncol(membership.rank)),
                        rep("sum_", ncol(membership.rank)))
        setnames(features, paste0(prefix, colnames(features)))

        if(is.null(normalization.parameters)){
                normalization.parameters = CalculateNormalizationParameters(features)
        }
        features = NormalizeFeatures(features, normalization.parameters)

        features = cbind(
                        features,
                        CalculateIngredientsMatrix(data, feature.ingredients)
        )

        params = list(
                        all.ingredients = all.ingredients,
                        feature.ingredients = feature.ingredients,
                        membership = membership,
                        membership.rank = membership.rank,
                        membership.exclusive = membership.exclusive,
                         normalization.parameters = normalization.parameters)

        list(features = features, params = params)
}

CalculateFeaturesWrapper = function(data, params){
        CalculateFeatures(data,
                         params$all.ingredients,
                         params$feature.ingredients,
                         params$membership,
                         params$membership.rank,
                         params$membership.exclusive,
                         params$normalization.parameters)
}
```

3.6.1 Example usage

I used the following to create the training, validation and test sets for my most recent models.

```
#Data cleaning
data.train$ingredients = lapply(data.train$ingredients, CleanIngredients)
data.test$ingredients = lapply(data.test$ingredients, CleanIngredients)
data.train$ingredients = lapply(data.train$ingredients, AllIngredients)
data.test$ingredients = lapply(data.test$ingredients, AllIngredients)

#Discarding some features
all.ingredients = do.call("c", data.train$ingredients)
feature.ingredients = FilterIngredients(all.ingredients, 1, 1)

#Separating training and validation sets
idx = sample(seq(nrow(data.train)), nrow(data.train) * 0.8)
train = data.train[idx,]
valid = data.train[-idx,]

#Training set
params = CalculateFeatures(train, all.ingredients, feature.ingredients)
train = params$features
params = params$params
train[, cuisine := as.factor(data.train$cuisine[idx])]

#Validation set
valid = CalculateFeaturesWrapper(valid, params)
valid = valid$features
valid[, cuisine := as.factor(data.train$cuisine[-idx])]

#Test set
test = CalculateFeaturesWrapper(data.test, params)
test = test$features
```

4 Modeling

I used the H2O package for modelling.

4.1 Preparation

To use H2O first we need to initialize the H2O server and upload the data.

```r
#Start H2O server
h2o.init(max_mem_size = "10g", nthreads = 8)

#Upload frames
train.hex = as.h2o(train, "train")
valid.hex = as.h2o(valid, "valid")
test.hex = as.h2o(test, "test")
```

4.2 Grid search

I used grid search to select the best model parameters for this problem.

This is the most recent grid search, only focusing on deeplearning models. In previous versions I tried several random forests and GBMs. I found the training of GBMs too slow for this problem and the random forests did not perform well with my engineered features.

With the grid searches I found out that the best activation function for this problem is the RectifierWithDropout, the ideal number of nodes in the first hidden layer is around 500 and between 75 - 150 in the second layer. The hidden dropout ratio parameter is good with the default 0.5 but the input dropout should be increased from 0. Setting the l1 parameter around 1e-5 also significantly increased the model's performance on the validation set.

```r
grid.dl = h2o.grid(
                "deeplearning",
                model_id = "dl",
                x = seq(ncol(train.hex)-1),
                y = "cuisine",
                training_frame = train.hex,
                validation_frame = valid.hex,
                hyper_params = list(
                                hidden = list(
                                                c(750, 250),
                                                c(500, 125),
                                                c(500, 250, 50)
                                ),
                                input_dropout_ratio = c(0, 0.3),
                                l1 = c(0, 1e-3)
                ),
                activation = "RectifierWithDropout",
                l2 = 0,
                overwrite_with_best_model = TRUE,
                use_all_factor_levels = TRUE,
                sparse = TRUE,
                epochs = 1000,
                stopping_rounds = 5,
                stopping_tolerance = 0.025,
                loss = "CrossEntropy",
                distribution = "multinomial",
                stopping_metric = "misclassification"
)
```

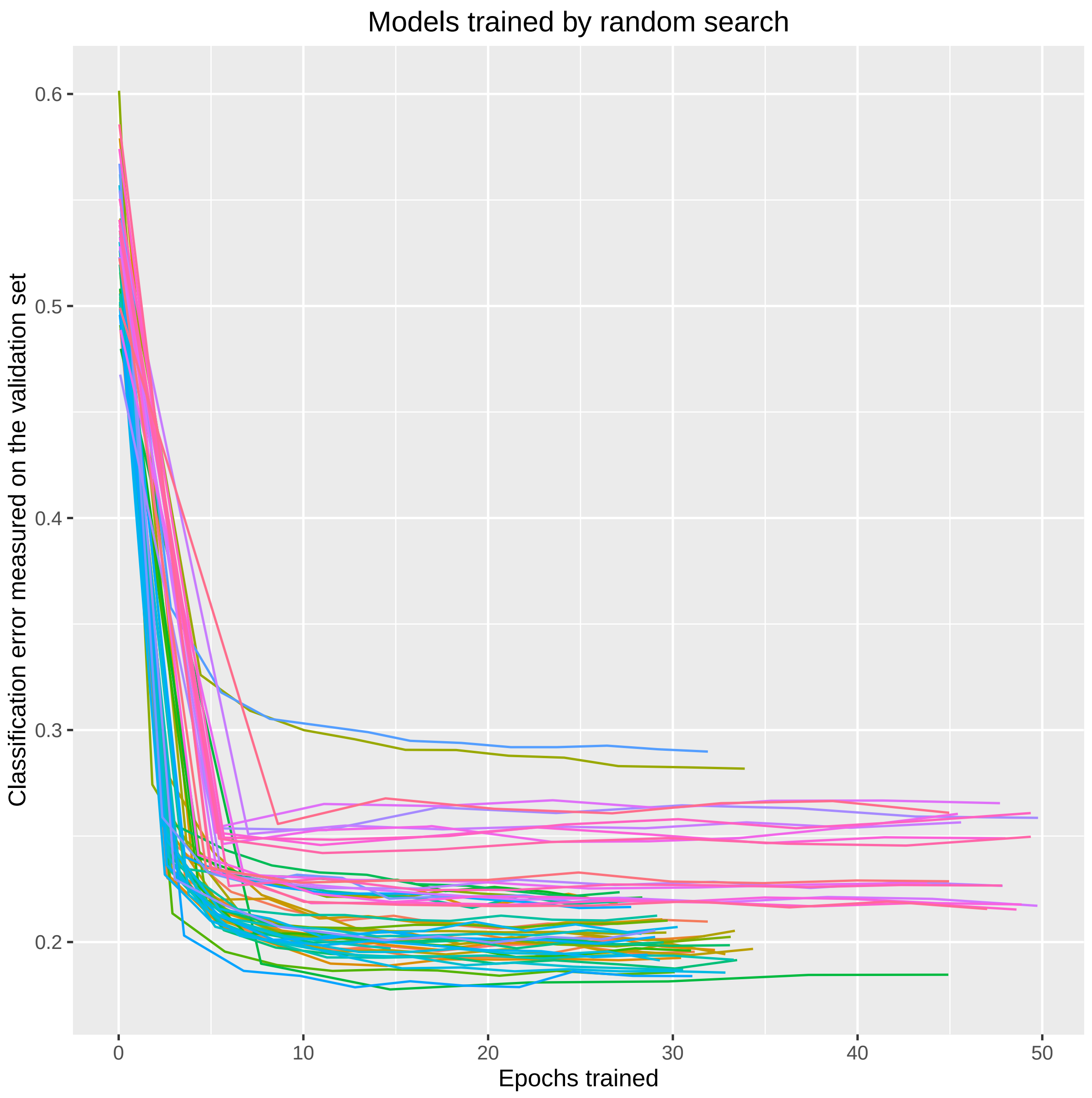
4.3 Random search

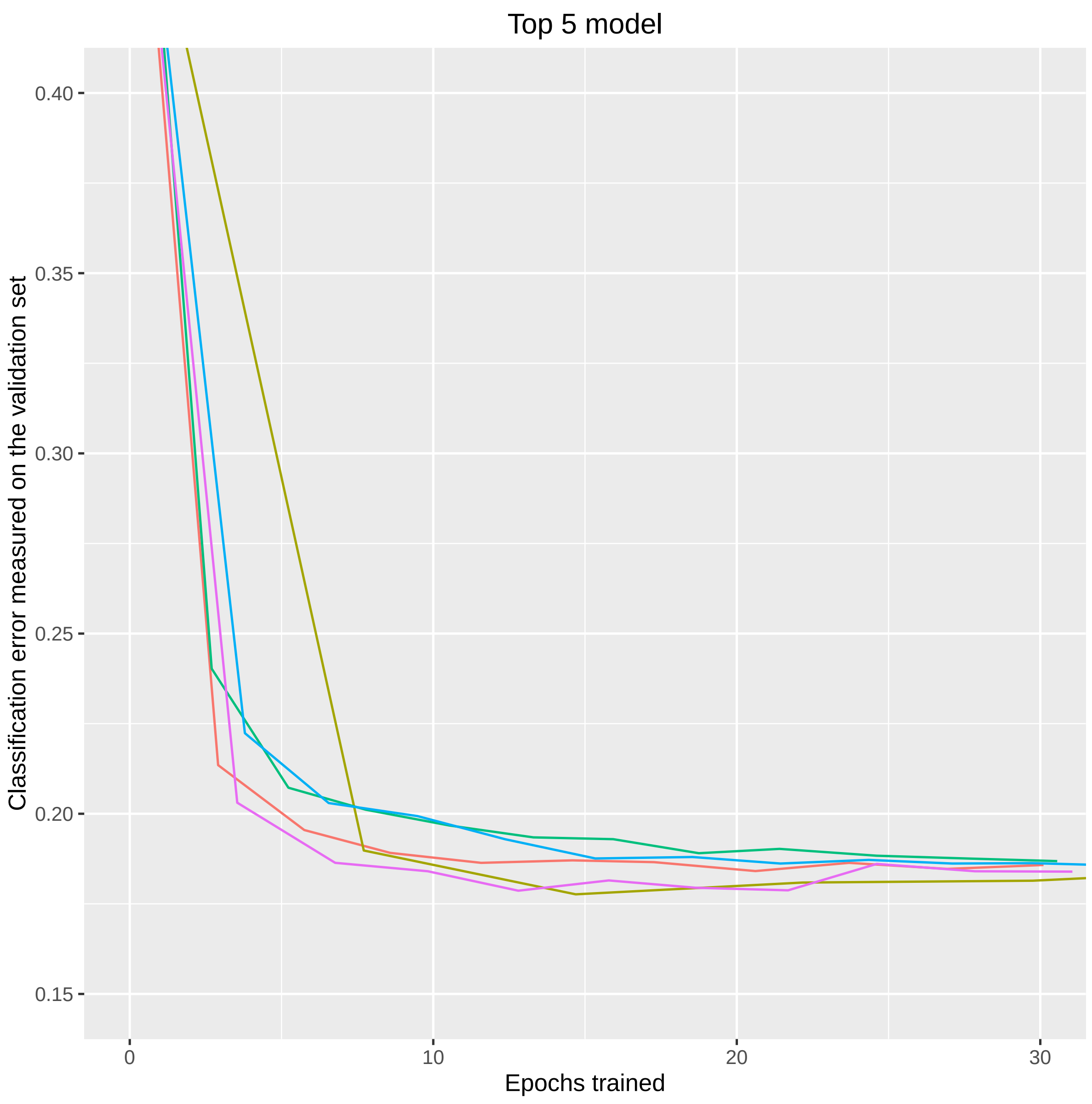
The grid searched helped to narrow down the search, after this I started to randomly search for best parameters within this parameter space.

```r
while(file.exists(paste0(data.root, "run.txt"))){
        h2o.init(max_mem_size = "10g", nthreads = 6)
        Sys.sleep(5)

        labelled.hex = as.h2o(train, "labelled")

        splits = h2o.splitFrame(labelled.hex,
                        destination_frames = c("train", "valid"))

        hidden.layers = as.integer(c(runif(1, 450, 600), runif(1, 75, 150)))
        input.dropout.ratio = round(runif(1, 0.2, 0.5), 3)
        l1.regularization = round(runif(1, 1e-05, 1e-03), 6)

        model.id = paste0("dl_layers(",
                        paste0(hidden.layers, collapse = "-"),
                        ")_input.dropout(", input.dropout.ratio,
                        ")_l1(", l1.regularization ,")")

        if(file.exists(paste0(data.root, "Models/", model.id))) {
                next
        }

        model = h2o.deeplearning(
                        model_id = model.id,
                        x = seq(ncol(labelled.hex)-1),
                        y = "cuisine",
                        training_frame = splits[[1]],
                        validation_frame = splits[[2]],
                        hidden = hidden.layers,
                        input_dropout_ratio = input.dropout.ratio,
                        l1 = l1.regularization,
                        activation = "RectifierWithDropout",
                        l2 = 0,
                        overwrite_with_best_model = TRUE,
                        use_all_factor_levels = TRUE,
                        sparse = TRUE,
                        epochs = 150,
                        score_interval = 60,
                        score_validation_samples = 0,
                        stopping_tolerance = 0.025,
                        score_duty_cycle = 0.2,
                        fast_mode = TRUE,
                        loss = "CrossEntropy",
                        distribution = "multinomial",
                        ignore_const_cols = FALSE,
                        shuffle_training_data = TRUE,
        )
        h2o.saveModel(object = model,
                        path = file.path("file:///", data.root, "Models"))

        h2o.shutdown(F)
        Sys.sleep(55)
}
```

4.4 Model evaluation

The models trained with the random search are compared by their performance on the validation set. The best models parameters are selected based on their calssification power.





4.5 Final models

I selected the 3 best parameter combinations based on the gridsearch results and used these parameters to train 10-10 models for 15-20-25 epochs.

Model parameters for the 3 best models:

1. model
   - hidden layers: 489-123
   - input dropout ratio: 0.26
   - l1 regularization: 1e-05
2. model
   - hidden layers: 493-84
   - input dropout ratio: 0.29
   - l1 regularization: 5e-04
3. model
   - hidden layers: 576-123

- input dropout ratio: 0.255
- l1 regularization: 1.1e-05

5 Scoring history

My submission scores:

1. 19.268% - all italian baseline
2. 58% - only feature engineering
3. 70% - with most important ingredients
4. 75% - with all ingredients
5. 78% - with multiple models
6. 79.214% - multiple models trained with best parameter settings found

Comparing my solution to the best submission I found that some people managed to score in the top 2% (score of 81.818%) using only a ”brute force” approach.

Their data cleaning was essentially identical to mine, meanwhile they did not do any additional feature engineering, beside using all available ingridients as features (like I did with my Ingredients Matrix).

This suggest that I could’ve achieved better results by focusing only tweaking the deeplearning parameters instead feature engineering OR that with my feature engineering I overfit the training data.