Recipe Data Analysis using Pyspark

Sangyun Kang

Industrial and System Engineering, Lehigh University

Nov 3, 2019

Abstract

In this task, I will collect recipe data and use the '`modified`' file which cleaned list of ingredients as compared to the '`original`' file. Before analyzing more deeply, I try to get familiar with the data by understanding it well. In Task 1, I fix a dictionary of ingredients. Then basically each ingredient will become one dimension and we can represent each ingredient array as a sparse vector (of 0s and 1s). Locations of ones will correspond to the ingredients in a given recipe and map each ingredient array into these sparse vectors. I use '`CountVectorizer`' to convert the list of ingredients to sparse vectors. '`CountVectorizer`' converts the list of tokens above to vectors of token counts. In Task 2 I cluster the given data and characterize each cluster. I observed that in 1 cluster you have majority of recipes from certain categories, high ratings, low calories or such. In this task, I found out how to represent the quality of the clustering. And, because the range of features are so large, clustering the original data set is so biased and the result is not good. So, it is needed to normalize the data somehow to have more meaningful clusters. After normalizing the data set, I can get the meaningful results. In Task 3, I process the data to come up with some classes and create corresponding labels. Because the given data is unlabeled, I made three different targets: Rating labels(0, 1, 2, 3, 4, 5, 6, 7), Protein labels(Low protein: 0, High protein: 1), Fat labels (Low fat: 0, High fat: 1). After finishing the data pre-process, I split the data randomly into training and testing part (80% training, 20% testing). And, I train different kinds of Machine learning models:Logistic Regression, Naive Bayes, Decision tree, Random Forest, and Linear Support Vector Machine. The testing accuracy of your model is not high in the first time because the target variables are imbalanced and features are needed to normalized. For Logistic regression and SVM, I can check the lrModel.coefficientMatrix from the lrModel and find the large values. The large values are corresponding to the importance features. Their meaning is that the larger value is, the more it can be important. The testing accuracy be improved further by using Decision tree models. Specifically, Random forest and Gradient-Boosted tree are better suited for certain types of input data. And, after normalizing the data set through Normalizer, StandardScaler, and MinMaxScaler, the result is improved further.

1 Introduction

We have recipe data set with unlabeled data. In this task, my objective is to get some meaningful insight from unstructured data set. I explore the data set and check the relationships between the features through statistical information and data visualizations. Using clustering each features, I identify the characteristic of each clustering and can suggest the food company or food-related people manage their resource more efficiently. Through this task, I try to answer the question: What story does it tell about the relationships between the variables and get some meaningful insights/discover the patterns from the recipe data set.

2 Data understanding (Data descriptive)

Our data set is related with recipe for 18,187 with 11 features. Original recipe data set has 20,130 rows and modified recipe data set has 18,187. And, there are 11 features: calories, categories, date, desc, directions, fat, ingredients, rating, sodium, title

```
print rddData_modified_DF.columns ## Column Names
print "Row Count: ", rddData_modified_DF.count() ## Row Count
print "Column Count: ", len(rddData_modified_DF.columns) ## Column Count
```

```
['calories', 'categories', 'date', 'desc', 'directions', 'fat', 'ingredients', 'protein', 'rating', 'sodium', 'titl
e']
Row Count:  18187
Column Count:  11
```

Figure 1: The representation of the dataset in PySpark

Variable	Type	Description
calories	Numerical	Amount of Calories
categories	Text	Categories e.g Sandwich, Bean, Fruit, Tomato, turkey
date	Date	1996-9-1 $\sim$ 2016-12-13
desc	Text	Descriptions
directions	Text	Cooking instructions
fat	Numerical	Amount of fat e.g 0-1722763
ingredients	Text	List of ingredients e.g kosher salt, extravirgin olive oil
protein	Numerical	Amount of protein e.g 0-236489
rating	Numerical	Range: 1-5
sodium	Numerical	Amount of Sodium:0-27675110
title	Categorical	Name of food e.g Lentil, Apple, and Turkey Wrap

Table 1: List of attributes of dataset

As shown in Table 1, we can know that features have different ranges and normalization will be required for doing further analysis. For example, the large values of attributes influence the result more due to its larger value. But, this doesn't necessarily mean it is more important as a predictor. And, text variable is needed to handle properly for analysis(e.g using sparse vector).

2.1 Modified data sets

First row '`ingredients`': [kosher salt, extravirgin olive oil]
Second row '`ingredients`': [bay leaves, salt, minced parsley, dried currants, sugar, cracked peppercorns, canned chicken broth, red wine vinegar, olive oil, lettuce leaves, ground nutmeg, butter, eggs, pepper, whipping cream, cloves, tawny port]

2.2 Original data set

First row '`ingredients`': [4 cups low-sodium vegetable or chicken stock, 1 cup dried brown lentils, 1/2 cup dried French green lentils, 2 stalks celery, chopped, 1 large carrot, peeled and chopped, 1 sprig fresh thyme, 1 teaspoon kosher salt, 1 medium tomato, cored, seeded, and diced, 1 small Fuji apple, cored and diced, 1 tablespoon freshly squeezed lemon juice, 2 teaspoons extra-virgin olive oil, Freshly ground black pepper to taste, 3 sheets whole-wheat lavash, cut in half crosswise, or 6 (12-inch) flour tortillas", 3/4 pound turkey breast, thinly sliced, 1/2 head Bibb lettuce]
Second row '`ingredients`': [1 1/2 cups whipping cream, 2 medium onions, chopped, 5 teaspoons salt, 3 bay leaves, 3 whole cloves, 1 large garlic clove, crushed, 1 teaspoon pepper, 1/8 teaspoon ground nutmeg, Pinch of dried thyme, crumbled, 8 large shallots, minced, 1 tablespoon butter, 1 pound trimmed boneless center pork loin, sinew removed cut into 1-inch chunks, well chilled, 3 eggs, 6 tablespoon all purpose flour, 1/4 cup tawny Port, 3 tablespoons dried currants, minced, Lettuce leaves, Cracked peppercorns, Minced fresh parsley, Bay leaves, French bread baguette slices, 3 tablespoons olive oil, 2 large red onions, halved, sliced, 3 tablespoons dried currants, 3 tablespoons red wine vinegar, 1 tablespoons canned chicken broth, 2 teaspoons chopped fresh thyme or 3/4 teaspoon dried, crumbled, 1/2 teaspoon sugar]
I compared these two data sets. Stare at the data for more hours. The difference between original data sets and modified data sets is ingredients. The modified data set extract more important ingredients from original data set. The '`modified`' file has cleaned list of ingredients as compared to the '`original`' file. And, "original" file has more data set than '`modified`' file.

3 Exploratory Data Analysis

3.1 Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing a description method. The describe function automatically computes basic statistics for all continuous variables. Any null values are automatically skipped in these statistics. This will show:
- the mean
- the standard deviation (std)
- the minimum value
- the maximum value

```
+-------+-----------------+  +-------+--------------------+  +-------+-----------------+
|summary|         calories|  |summary|                date|  |summary|              fat|
+-------+-----------------+  +-------+--------------------+  +-------+-----------------+
|  count|            14478|  |  count|               18187|  |  count|            14430|
|   mean|6633.144426025694|  |   mean|                null|  |   mean|363.93562023562026|
| stddev|375092.3609403792|  | stddev|                null|  | stddev|21372.496653092345|
|    min|              0.0|  |    min|1996-09-01T20:47:...|  |    min|              0.0|
|    max|     3.0111218E7|  |    max|2016-12-13T13:00:...|  |    max|         1722763.0|
+-------+-----------------+  +-------+--------------------+  +-------+-----------------+

+-------+-----------------+  +-------+-----------------+   +-------+-----------------+
|summary|          protein|  |summary|           rating|   |summary|           sodium|
+-------+-----------------+  +-------+-----------------+   +-------+-----------------+
|  count|            14450|  |  count|            18177|   |  count|            14477|
|   mean|92.37702422145328|  |   mean|3.7204984320845025|  |   mean| 6585.688195068039|
| stddev|3514.183797992332|  | stddev|1.3262101416442342|  | stddev|348705.81156770256|
|    min|              0.0|  |    min|              0.0|   |    min|              0.0|
|    max|         236489.0|  |    max|              5.0|   |    max|        2.767511E7|
+-------+-----------------+  +-------+-----------------+   +-------+-----------------+
```

Figure 2: The statistics of features in recipe data set

As shown in Table 1, we can know that features have different ranges. '`calories`' column is the highest value compared to protein and rating. Therefore, the analysis can be biased from them. And, the period is from 1996-09-01 to 2016-12-13.

Variable	count	mean	stddev	min	max
calories	14478	6633.14	375092.36	0.0	3.0111218E7
fat	14430	363.93	21372.49	0.0	1722763
protein	14450	92.37	3514.18	0.0	236489
rating	18177	3.72	1.32.	0.0	5
sodium	14477	6585.68	348705.81	0.0	2.767511E7

Table 2: Summary statistics for numeric variables

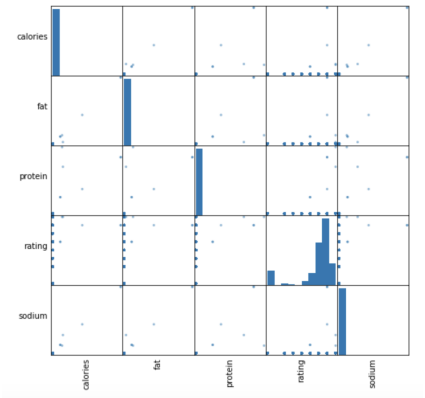


Figure 3: Correlation between numerical features

In this task, I find out the correlations between variables. It is concluded that there aren't highly correlated numeric variables. Therefore, we will keep all of them for the model. However, '`categories`', '`date`', '`desc`', and '`directions`' columns are not really useful, we will remove these four columns. Categorical variable '`title`' will be changed into numerical valuables through StringIndexer method.

3.2 Preparing Data for Machine Learning

Real data are rarely clean and complicated. Normally, they tend to be incomplete, noisy, and inconsistent and it is super important task to prepossess the data by handling missing values. Let's identify missing values in the dataset.

calories	fat	ingredients	protein	rating	sodium	title
426.0	7.0	kosher salt,extra...	30.0	2.5	559.0	Lentil, Apple, an...
403.0	23.0	bay leaves,salt,m...	18.0	4.375	1439.0	Boudin Blanc Terr...
165.0	7.0	chicken broth,oni...	6.0	3.75	165.0	Potato and Fennel...
null	null	onion,anchovy pas...	null	5.0	null	Mahi-Mahi in Toma...
547.0	32.0	ground nutmeg,sou...	20.0	3.125	452.0	Spinach Noodle Ca...
948.0	79.0	lettuce leaves,ma...	19.0	4.375	1042.0	The Best Blts
null	null	null	null	4.375	null	Ham and Spring Ve...
null	null	granulated sugar,...	null	3.75	null	Spicy-Sweet Kumqu...
170.0	10.0	minced garlic,sug...	7.0	4.375	1272.0	Korean Marinated ...
602.0	41.0	dijon mustard,may...	23.0	3.75	1696.0	Ham Persillade wi...

only showing top 10 rows

Figure 4: Dataframe in recipe data set

Variable	Count
calories	3709
fat	3757
protein	586
rating	3737
sodium	10
title	3710

Table 3: List of missing data

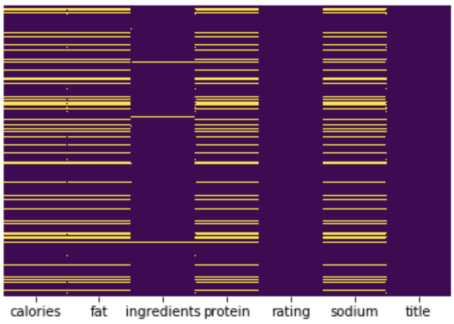


Figure 5: Heatmap of missing data

As shown in the figure 4, we can find the null values in data set. Missing values can distort the real pattern hidden in the data so it is needed to handle properly. I looked into how many missing values are in each of these columns. There are various methods to solve the null values issue. Next steps, I will talk about how to handle the missing values.

1. Ignore the data row

This is a quick solution and typically is preferred in cases where the percentage of missing values is relatively low (less than 5%). This is a simple method to drop data containing the missing values.

2. Replacing With Mean/Median/Mode

We can calculate the mean, median or mode of the feature and replace it with the missing values. This is an approximation which can add variance to the data set. But the loss of the data can be negated by this method which yields better results compared to removal of rows and columns. Replacing with the above three approximations are a statistical approach of handling the missing values.

For the analysis, I choose the method to drop the missing rows because the missing values are not too much and the variance is so high that imputing the mean/median can affect the results. I changed blank into null and deleted them through dropna() function. After dealing with missing values, the number of data set (18187) is changed into 17591.

4 Feature engineering

In most of tasks, I use Dataframe for analyzing the data set. And, after choosing features to analyze and handling the missing values, I change the text variables into a sparse vectors for using this meaningful data. We usually work with structured data but unstructured text data can have vital content for analysis. The next step is Feature Engineering. PySpark has made it so easy that we do not need to do much for extracting features. here are the steps:

1. Tokenize the ingredient

Tokenization is the process of taking text (such as a sentence) and breaking it into individual terms (usually words). In '`ingredients`' column, there are bunch of ingredients and it is needed to parse this data. Let's tokenize the messages and create a list of words of each ingredient.

```
from pyspark.ml.feature import RegexTokenizer
tokenizer = RegexTokenizer(inputCol="ingredients", outputCol = "ingredient", pattern=",")
ingredientData3 = tokenizer.transform(dataset3)
ingredientData3.show(10)
```

```
+--------+----+--------------------+-------+------+------+-----+--------------------+
|calories| fat|         ingredients|protein|rating|sodium|label|          ingredient|
+--------+----+--------------------+-------+------+------+-----+--------------------+
|   426.0| 7.0|kosher salt,extra...|   30.0|   2.5| 559.0|  0.0|[kosher salt, ext...|
|   403.0|23.0|bay leaves,salt,m...|   18.0| 4.375|1439.0|  0.0|[bay leaves, salt...|
|   165.0| 7.0|chicken broth,oni...|    6.0|  3.75| 165.0|  0.0|[chicken broth, o...|
|   547.0|32.0|ground nutmeg,sou...|   20.0| 3.125| 452.0|  1.0|[ground nutmeg, s...|
|   948.0|79.0|lettuce leaves,ma...|   19.0| 4.375|1042.0|  1.0|[lettuce leaves, ...|
|   170.0|10.0|minced garlic,sug...|    7.0| 4.375|1272.0|  0.0|[minced garlic, s...|
|   602.0|41.0|dijon mustard,may...|   23.0|  3.75|1696.0|  1.0|[dijon mustard, m...|
|   256.0| 5.0|olive oil,ground ...|    4.0|  3.75|  30.0|  0.0|[olive oil, groun...|
|   766.0|48.0|allpurpose flour,...|   12.0| 4.375| 439.0|  1.0|[allpurpose flour...|
|   174.0|12.0|canned beef broth...|   11.0| 4.375| 176.0|  0.0|[canned beef brot...|
+--------+----+--------------------+-------+------+------+-----+--------------------+
only showing top 10 rows
```

Figure 6: Tokenization of ingredients

2. Apply CountVectorizer

CountVectorizer converts the list of tokens above to vectors of token counts. We can see the documentation description for details at `https://spark.apache.org/docs/latest/ml-features#countvectorizer`.

```
from pyspark.ml.feature import CountVectorizer
count = CountVectorizer(inputCol="ingredient", outputCol="rawFeatures")
model3 = count.fit(ingredientData3)
featureizedData3 = model3.transform(ingredientData3)
featureizedData3.show(3)
```

```
+--------+----+--------------------+-------+------+------+-----+--------------------+--------------------+
|calories| fat|         ingredients|protein|rating|sodium|label|          ingredient|         rawFeatures|
+--------+----+--------------------+-------+------+------+-----+--------------------+--------------------+
|   426.0| 7.0|kosher salt,extra...|   30.0|   2.5| 559.0|  0.0|[kosher salt, ext...|(1748,[7,10],[1.0...|
|   403.0|23.0|bay leaves,salt,m...|   18.0| 4.375|1439.0|  0.0|[bay leaves, salt...|(1748,[0,1,2,5,19...|
|   165.0| 7.0|chicken broth,oni...|    6.0|  3.75| 165.0|  0.0|[chicken broth, o...|(1748,[9,11,20,52...|
+--------+----+--------------------+-------+------+------+-----+--------------------+--------------------+
only showing top 3 rows
```

Figure 7: countVectorizer of ingredients

3. StringIndexer: Assign indices to each category in our categorical columns.

StringIndexer encodes a string column of labels to a column of label indices. The indices are in [0, numLabels), ordered by label frequencies, so the most frequent label gets index 0. In our case, the label column (Category) will be encoded to label indices, from 0 to 5. Because '`title`' column is a categorical variable, it has to be changed into a numerical variable. StringIndexer helps change into numerical one.

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler
label_stringIdx = StringIndexer(inputCol = "title", outputCol = "title_")
model = label_stringIdx.fit(rescaledData3)
rescaledData4 = model.transform(rescaledData3)
```

```
rescaledData4.show(2)
```

```
+--------+----+--------------------+-------+------+------+--------------------+-----+--------------------+--------------------+
--------------------+------+
|calories| fat|         ingredients|protein|rating|sodium|               title|label|          ingredient|         rawFeatures|
sparsevector|title_|
+--------+----+--------------------+-------+------+------+--------------------+-----+--------------------+--------------------+
--------------------+------+
|   426.0| 7.0|kosher salt,extra...|   30.0|   2.5| 559.0|Lentil, Apple, an...|  0.0|[kosher salt, ext...|(1748,[7,10],[1.0...|
(1748,[7,10],[2.5...|8776.0|
|   403.0|23.0|bay leaves,salt,m...|   18.0| 4.375|1439.0|Boudin Blanc Terr...|  0.0|[bay leaves, salt...|(1748,[0,1,2,5,19...|
(1748,[0,1,2,5,19...|2231.0|
+--------+----+--------------------+-------+------+------+--------------------+-----+--------------------+--------------------+
--------------------+------+
only showing top 2 rows
```

Figure 8: StringIndexer of title column

4. VectorAssembler

VectorAssembler is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees. It helps create a feature vector from all categorical and numerical features and we call the final vector as '`features`'. So, I combine all features to cluster or build the models.

```
numericCols = ['calories', 'fat', 'protein', 'rating', 'sodium']
#numericCols = ['calories', 'fat', 'protein', 'rating', 'sodium', 'sparsevector']
assemblerInputs = numericCols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features_")
```

Figure 9: vectorAssembler of all features

5. Pipeline

We use Pipeline to chain multiple Transformers and Estimators together to specify our machine learning workflow. A Pipeline's stages are specified as an ordered array. I applied the feature functions individually but when I googled how to do in one step, the answer is '`Pipeline`'. It is a brilliant method!

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler
label_stringIdx = StringIndexer(inputCol = "rating", outputCol = "label")
```

```
#pipeline = Pipeline(stages=[regexTokenizer, stopwordsRemover, countVectors, label_stringIdx, normalizeVectors])
pipeline = Pipeline(stages=[regexTokenizer, stopwordsRemover, countVectors, label_stringIdx])
# Fit the pipeline to training documents.
pipelineFit = pipeline.fit(df)
df_new = pipelineFit.transform(df)
df_new.show(10)
```

Figure 10: Pipeline method

c.f) We can apply OneHotEncoderEstimator() to convert categorical columns to onehot encoded vectors. But, I didn't use this technique in this data set.

After a few steps above, we finally get our sparse vector for ingredients. Each ingredient will become one dimension and we can represent each ingredient array as a sparse vector (of 0s and 1s). Locations of ones will correspond to the ingredients in a given recipe. The sparse vector allows fast row access and matrix-vector multiplications compared to dense vectors.

```
+--------------------------------------------------------------------------------------------------------------------------------
---------+
|features
|
+--------------------------------------------------------------------------------------------------------------------------------
---------+
|(813,[6,8],[1.0,1.0])
|
|(813,[0,1,2,5,23,35,44,55,61,77,121,123,186,223,259,406],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0])|
|(813,[10,12,20,56],[1.0,1.0,1.0,1.0])
|
|(813,[6,13,20,262],[1.0,1.0,1.0,1.0])
|
|(813,[31,61],[1.0,1.0])
|
|(813,[29,406],[1.0,1.0])
```

Figure 11: Sparse vectors for ingredients

5 Clustering

This task is about task 2 and about K-means clustering. K-means is a type of unsupervised learning and one of the popular methods of clustering unlabelled data into k clusters.

5.1 Step 1: Assemble your features

In contrast to most ML packages out there, Spark ML requires your input features to be gathered in a single column of your dataframe, usually named features; and it provides a specific method for doing this, VectorAssembler: VectorAssembler is a transformer that combines a given list of columns into a single vector column. I assembled calories, fat, protein, rating, and sodium for clustering.

```
#Step 1 - assemble your features
from pyspark.ml.feature import VectorAssembler

vecAssembler = VectorAssembler(inputCols=['calories','fat','protein','rating','sodium'], outputCol="features")
df_new4 = vecAssembler.transform(df_new3)
df_new4.show()
```

```
+--------+--------------------+----+--------------------+-------+------+------+--------------------+
|calories|          categories| fat|         ingredients|protein|rating|sodium|            features|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+
|   426.0|[Sandwich, Bean, ...| 7.0|[kosher salt, ext...|   30.0|   2.5| 559.0|[426.0,7.0,30.0,2...|
|   403.0|[Food Processor, ...|23.0|[bay leaves, salt...|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|
|   165.0|[Soup/Stew, Dairy...| 7.0|[chicken broth, o...|    6.0|  3.75| 165.0|[165.0,7.0,6.0,3....|
|   547.0|[Cheese, Dairy, P...|32.0|[ground nutmeg, s...|   20.0| 3.125| 452.0|[547.0,32.0,20.0,...|
|   948.0|[Sandwich, Food P...|79.0|[lettuce leaves, ...|   19.0| 4.375|1042.0|[948.0,79.0,19.0,...|
|   170.0|[Beef, Ginger, Sa...|10.0|[minced garlic, s...|    7.0| 4.375|1272.0|[170.0,10.0,7.0,4...|
|   602.0|[Salad, Mustard, ...|41.0|[dijon mustard, m...|   23.0|  3.75|1696.0|[602.0,41.0,23.0,...|
|   256.0|[Milk/Cream, Dair...| 5.0|[olive oil, groun...|    4.0|  3.75|  30.0|[256.0,5.0,4.0,3....|
|   766.0|[Cake, Chocolate,...|48.0|[allpurpose flour...|   12.0| 4.375| 439.0|[766.0,48.0,12.0,...|
|   174.0|[Garlic, Sauté, L...|12.0|[canned beef brot...|   11.0| 4.375| 176.0|[174.0,12.0,11.0,...|
|   134.0|[Sauce, Mustard, ...| 3.0|[sugar, dijon mus...|    4.0| 3.125|1394.0|[134.0,3.0,4.0,3....|
|   382.0|[Soup/Stew, Garli...|31.0|[minced garlic, o...|    5.0| 4.375| 977.0|[382.0,31.0,5.0,4...|
|   146.0|[Bread, Milk/Crea...| 5.0|[strawberry prese...|    4.0| 1.875| 160.0|[146.0,5.0,4.0,1....|
|   890.0|[Onion, Pork, Veg...|68.0|[black peppercorn...|   59.0| 4.375|1027.0|[890.0,68.0,59.0,...|
|   107.0|[Cheese, Fish, Pe...| 7.0|[minced garlic, o...|    5.0|   5.0| 344.0|[107.0,7.0,5.0,5....|
|   421.0|[Potato, High Fib...|33.0|[olive oil, veget...|   10.0|   5.0| 383.0|[421.0,33.0,10.0,...|
|   345.0|[Salad, Ginger, N...|19.0|[salt, sugar, water]|   11.0| 4.375| 423.0|[345.0,19.0,11.0,...|
|   279.0|[Nut, Bake, Cockt...|30.0|[pecan halves, fi...|    3.0|  3.75| 206.0|[279.0,30.0,3.0,3...|
|    95.0|[Bread, Condiment...| 7.0|[salt, olive oil,...|    1.0|   0.0| 103.0|[95.0,7.0,1.0,0.0...|
|   215.0|[Egg, Herb, Veget...|20.0|[freshly ground p...|    6.0|  3.75| 250.0|[215.0,20.0,6.0,3...|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+
only showing top 20 rows
```

Figure 12: vectorAssembler for clustering

As perhaps already guessed, the argument inputCols serves to tell VectoeAssembler which particular columns in our dataframe are to be used as features.

5.2 Step 2: fit your KMeans model

```
from pyspark.ml.clustering import KMeans

kmeans = KMeans(k=2, seed=1)  # 2 clusters here (Just for checking)
model = kmeans.fit(df_new4.select('features'))
```

Figure 13: Fit KMeans clustering

select('features') here serves to tell the algorithm which column of the dataframe to use for clustering - remember that, after Step 1 above, your original features are no more directly used.

5.3 Step 3: transform your initial dataframe to include cluster assignments

When we see the prediction, shows the cluster assignment in our recipe data set. We can further modify the transformed dataframe with select statements, or even drop the features column.

```
transformed = model.transform(df_new4)
transformed.show(3)
```

```
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
|calories|          categories| fat|         ingredients|protein|rating|sodium|            features|prediction|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
|   426.0|[Sandwich, Bean, ...| 7.0|[kosher salt, ext...|   30.0|   2.5| 559.0|[426.0,7.0,30.0,2...|         0|
|   403.0|[Food Processor, ...|23.0|[bay leaves, salt...|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|         0|
|   165.0|[Soup/Stew, Dairy...| 7.0|[chicken broth, o...|    6.0|  3.75| 165.0|[165.0,7.0,6.0,3....|         0|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
only showing top 3 rows
```

```
transformed.select('calories','fat','protein','rating','sodium','features','prediction').show(4)
```

```
+--------+----+-------+------+------+--------------------+----------+
|calories| fat|protein|rating|sodium|            features|prediction|
+--------+----+-------+------+------+--------------------+----------+
|   426.0| 7.0|   30.0|   2.5| 559.0|[426.0,7.0,30.0,2...|         0|
|   403.0|23.0|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|         0|
|   165.0| 7.0|    6.0|  3.75| 165.0|[165.0,7.0,6.0,3....|         0|
|   547.0|32.0|   20.0| 3.125| 452.0|[547.0,32.0,20.0,...|         0|
+--------+----+-------+------+------+--------------------+----------+
only showing top 4 rows
```

Figure 14: dataframe including cluster assignments

5.4 Step 4: Determine the optimal number of clusters for k-means clustering

As k increases, the sum of squared distance tends to zero. Imagine we set k to its maximum value n (where n is number of samples) each sample will form its own cluster meaning sum of squared distances equals zero.

```
## Checking the Elbow Point (WSSSE)
for k in range(2,9):
    kmeans = KMeans(featuresCol='features',k=k)
    model = kmeans.fit(df_new4.select('features'))
    wssse = model.computeCost(df_new4.select('features'))
    print("With K={}".format(k))
    print("Within Set Sum of Squared Errors = " + str(wssse))
    print('--'*30)
```

```
With K=2
Within Set Sum of Squared Errors = 4.66271751557e+14
------------------------------------------------------------
With K=3
Within Set Sum of Squared Errors = 1.51035663941e+14
------------------------------------------------------------
With K=4
Within Set Sum of Squared Errors = 1.45003780914e+13
------------------------------------------------------------
With K=5
Within Set Sum of Squared Errors = 6.74420614369e+11
------------------------------------------------------------
With K=6
Within Set Sum of Squared Errors = 1.60652698445e+11
------------------------------------------------------------
With K=7
Within Set Sum of Squared Errors = 1.48793394874e+11
------------------------------------------------------------
With K=8
Within Set Sum of Squared Errors = 1.48793355234e+11
------------------------------------------------------------
```

Figure 15: Checking the Elbow Point (WSSSE)

The result is terrible. The errors are so huge it is needed to normalize the features. And, various ranges of features produce bad clustering.

```
model_k3.transform(df_new4.select('features')).groupBy('prediction').count().show()
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1|    1|
|         6|    1|
|         3|    1|
|         5|    1|
|         4|    2|
|         2|    1|
|         0|14413|
+----------+-----+
```

Figure 16: The number of clusters

It is time to normalize the feature which has large ranges. I use three kinds of normalization methods, compare those methods, and choose the best method among them. Normalization is a technique applied as part of data preparation and it change numerical values in the dataset to a common scale.

- Normalizer

```
from pyspark.ml.feature import Normalizer
normalizeVectors = Normalizer(inputCol="features", outputCol="features_norm", p=2)
NormalizedData = normalizeVectors.transform(df_new4)
print("Normalized using L^2 norm")
NormalizedData.show()
```

```
Normalized using L^2 norm
+--------+--------------------+----+--------------------+-------+------+------+--------------------+--------------------
-----+
|calories|          categories| fat|         ingredients|protein|rating|sodium|            features|       features
_norm|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+--------------------
-----+
|   426.0|[Sandwich, Bean, ...| 7.0|[kosher salt, ext...|   30.0|   2.5| 559.0|[426.0,7.0,30.0,2...|[0.605543710021
32...|
|   403.0|[Food Processor, ...|23.0|[bay leaves, salt...|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|[0.269626921843
34...|
|   165.0|[Soup/Stew, Dairy...| 7.0|[chicken broth, o...|    6.0|  3.75| 165.0|[165.0,7.0,6.0,3....|[0.706464427373
80...|
|   547.0|[Cheese, Dairy, P...|32.0|[ground nutmeg, s...|   20.0| 3.125| 452.0|[547.0,32.0,20.0,...|[0.769776287835
49...|
|   948.0|[Sandwich, Food P...|79.0|[lettuce leaves, ...|   19.0| 4.375|1042.0|[948.0,79.0,19.0,...|[0.671835557396
40...|
```

Figure 17: Normalizer

Normalizer is a Transformer which transforms a dataset of Vector rows, normalizing each Vector to have unit norm. This normalization can help standardize your input data and improve the behavior of learning algorithms.

- StandardScaler

```
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                        withStd=True, withMean=True)
# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(df_new4)
# Normalize each feature to have unit standard deviation.
scaledData = scalerModel.transform(df_new4)
scaledData.show()
```

```
+--------+--------------------+----+--------------------+-------+------+------+--------------------+--------------
-----+
|calories|          categories| fat|         ingredients|protein|rating|sodium|            features|      scaledFea
tures|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+--------------
-----+
|   426.0|[Sandwich, Bean, ...| 7.0|[kosher salt, ext...|   30.0|   2.5| 559.0|[426.0,7.0,30.0,2...|[-0.01658444536
12...|
|   403.0|[Food Processor, ...|23.0|[bay leaves, salt...|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|[-0.01664564067
42...|
|   165.0|[Soup/Stew, Dairy...| 7.0|[chicken broth, o...|    6.0|  3.75| 165.0|[165.0,7.0,6.0,3....|[-0.01727887913
09...|
|   547.0|[Cheese, Dairy, P...|32.0|[ground nutmeg, s...|   20.0| 3.125| 452.0|[547.0,32.0,20.0,...|[-0.01626250480
13...|
```

Figure 18: standardScaler

StandardScaler transforms a dataset of Vector rows, normalizing each feature to have unit standard deviation and/or zero mean. The model can then transform a Vector column in a dataset to have unit standard deviation and/or zero mean features.

- MinMaxScaler

```
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.linalg import Vectors

minMaxscaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")

# Compute summary statistics and generate MinMaxScalerModel
minMaxscalerModel = minMaxscaler.fit(df_new4)

# rescale each feature to range [min, max].
minMaxscaledData = minMaxscalerModel.transform(df_new4)
print("Features scaled to range: [%f, %f]" % (minMaxscaler.getMin(), minMaxscaler.getMax()))
minMaxscaledData.select("features", "scaledFeatures").show()
```

```
Features scaled to range: [0.000000, 1.000000]
+--------------------+--------------------+
|            features|      scaledFeatures|
+--------------------+--------------------+
|[426.0,7.0,30.0,2...|[1.41475512548180...|
|[403.0,23.0,18.0,...|[1.33837163279147...|
|[165.0,7.0,6.0,3....|[5.47968534517600...|
|[547.0,32.0,20.0,...|[1.81659871746137...|
|[948.0,79.0,19.0,...|[3.14832830741021...|
|[170.0,10.0,7.0,4...|[5.64573641624194...|
|[602.0,41.0,23.0,...|[1.99925489563391...|
|[256.0,5.0,4.0,3....|[8.50181483857610...|
|[766.0,48.0,12.0,...|[2.54390240873019...|
|[174.0,12.0,11.0,...|[5.77857727309469...|
```

Figure 19: MinMaxScaler

MinMaxScaler transforms a dataset of Vector rows, rescaling each feature to a specific range (often [0, 1]). After comparing the errors in each normalization, it can be concluded that minMaxScaler is the best method to normalize the data set. That is, among the normalization

methods, minmax method is the best and cluster the features more clearly. And, in the Figure 20 above the elbow is at k=12 indicating the optimal k for this dataset is 12 but I choose the best k for 6 because the error is a quite low and it is easy to classify each cluster easily.

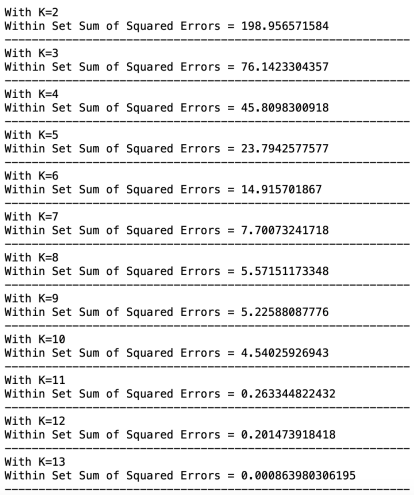


Figure 20: Checking the Elbow Point (WSSSE) in MinMaxScaler

I performed k-means-clustering through choosing the optimal k for 6. Let's characterize each cluster.

- Cluster 0: High fat, olive oil, extravirgin olive oil, kosher salt(ingredient), Milk/Cream, Rum, Alcoholic, Egg, Christmas, Brandy, Winter, Christmas Eve, Nutmeg, Gourmet, Drink (category), Medium protein, High Sodium, high rating
- Cluster 1: Low fat , extravirgin olive oil, kosher salt(ingredient), Bon Appétit, Bitters, Gin, Alcoholic, Cocktail Party, House and Garden, Drink (category), Low protein, Low sodium, zero rating
- Cluster 2: Medium fat, olive oil, extravirgin olive oil, sugar(ingredient), Oscars, Cheddar, Cream Cheese, Bacon, Pecan, Tailgating, Family Reunion, Poker/Game Night, Jalapeño, Party (category), Low protein, medium sodium, low rating
- Cluster 3: Low fat, olive oil, extravirgin olive oil, vegetable oil(ingredient) Chocolate, Fruit, Dessert, Bake, Cherry, Summer, Bon Appétit (category): Related with Bake, Low protein, low sodium
- Cluster 4: Medium fat, sugar, lemon juice, sugar, water(ingredient), Pastry, Gourmet (category), High protein, medium sodium, five rating
- Cluster 5: Low fat, vegetable oil, olive oil, lemon juice(ingredient), no special category, Low protein, medium sodium, five rating

I used '`filter`' function and filter out each cluster and each feature. Then, check which features are the most and characterize each cluster easily. We can check more details in my python codes' output. Because of large ranges in each feature, normalization is very powerful in this data set.

```
from pyspark.ml.clustering import KMeans
#from pyspark.ml.evaluation import ClusteringEvaluator

kmeans = KMeans().setK(6)
model = kmeans.fit(minMaxscaledData.select('features'))
transformed = model.transform(minMaxscaledData).drop('features')
transformed.filter(transformed.prediction==0).show(5)
```

```
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
|calories|          categories| fat|         ingredients|protein|rating|sodium|           features_|prediction|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
|   403.0|[Food Processor, ...|23.0|[bay leaves, salt...|   18.0| 4.375|1439.0|[403.0,23.0,18.0,...|         0|
|   948.0|[Sandwich, Food P...|79.0|[lettuce leaves, ...|   19.0| 4.375|1042.0|[948.0,79.0,19.0,...|         0|
|   170.0|[Beef, Ginger, Sa...|10.0|[minced garlic, s...|    7.0| 4.375|1272.0|[170.0,10.0,7.0,4...|         0|
|   766.0|[Cake, Chocolate,...|48.0|[allpurpose flour...|   12.0| 4.375| 439.0|[766.0,48.0,12.0,...|         0|
|   174.0|[Garlic, Sauté, L...|12.0|[canned beef brot...|   11.0| 4.375| 176.0|[174.0,12.0,11.0,...|         0|
+--------+--------------------+----+--------------------+-------+------+------+--------------------+----------+
only showing top 5 rows
```

```
model.transform(minMaxscaledData.select('features')).groupBy('prediction').count().show()
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1| 1144|
|         3| 3800|
|         5|  183|
|         4| 1839|
|         2| 1429|
|         0| 6025|
+----------+-----+
```

Figure 21: Kmeans in MinMaxScaler

```
from pyspark.sql.functions import col, when

transformed.filter(transformed.prediction==0).groupBy("ingredients") \
    .count() \
    .orderBy(col("count").desc()) \
    .show(20,False)
```

```
+---------------------------------+-----+
|ingredients                      |count|
+---------------------------------+-----+
|olive oil                        |51   |
|extravirgin olive oil            |27   |
|sugar,water                      |19   |
|lemon juice                      |18   |
|sugar                            |18   |
|salt                             |15   |
|mayonnaise                       |13   |
|lemon juice,olive oil            |13   |
|lemon juice,extravirgin olive oil|11   |
|water                            |9    |
|heavy cream                      |9    |
|kosher salt                      |9    |
|salt,sugar                       |8    |
|coarse kosher salt               |8    |
|salt,olive oil                   |7    |
|lemon juice,sugar                |6    |
|salt,allpurpose flour            |6    |
|vegetable oil                    |6    |
|stick butter                     |6    |
|olive oil,dry white wine         |6    |
+---------------------------------+-----+
only showing top 20 rows
```

Figure 22: Characterize each cluster

6 Model Training and Evaluation

Since this is a classification problem with binary/multi-class response, the classification method includes Logistic regression, Naive bayes, Decision tree (Random forest, Gradient boosted tree), Support vector machines (grid search) and neural network algorithms. All my models are supervised machine learning algorithms. And, because the given data is unlabeled, I need to come up with some classes and create corresponding labels. Regarding creating labels in this task, I use the multiple rating, high-protein content vs low-protein content, high-fat content vs low-fat content as a target of my models. I used different targets in order to check all the cases and learn the techniques to improve the models. Therefore, I build the classification model based on each target and predict the outcome (e.g predict if the food contain high-protein content or low-protein content, predict

the rating of foods based on the ingredients, sodium, fat, and so on.) Before building the models, we need to split data into 80% training, 20% testing. We build the models based on training data set and validate our models through test data set. This task is super important because it helps us avoid over-fitting and accurately measure how well our model generalizes to new data. In this task, I attempt to improve the testing accuracy by using any other algorithms. This is because it is possible that certain algorithms are better suited for certain types of input data. In this task, Decision tree models (including Random forest, Gradient Boosted tree) perform the best. (Over 94%) The reason is that Decision trees are able to handle both continuous/categorical variable and there are missing values in the data set. Decision trees are robust for this data set.

```
# Partition Training & Test sets
# set seed for reproducibility
trainingData, testData = dataset.randomSplit([0.7, 0.3], seed = 100)
print("Training Dataset Count: " + str(trainingData.count()))
print("Test Dataset Count: " + str(testData.count()))
```

Figure 23: Split data into training and test data

It is important to set seed for the randomSplit() function in order to get same split for each run. And, I will build the binary and multi-class models based on training data set and evaluate my models through test data set.

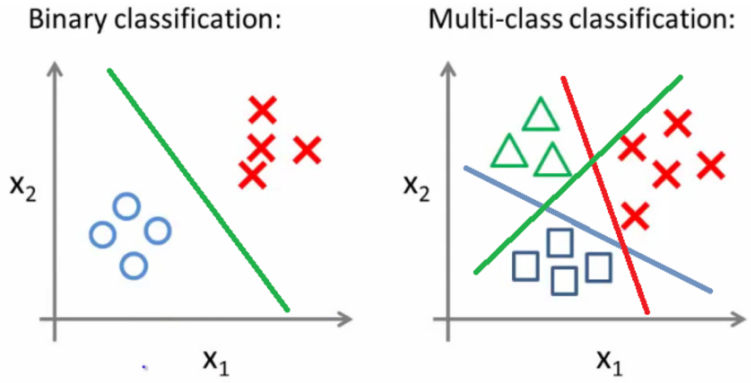


Figure 24: Binary and Multi-class classification from geeksforgeeks website

I has already performed the feature engineering like below:

1. Handle the missing values (Drop the missing rows)
2. Feature Selection (Choose the features to build the models)
3. Change text variables ("Ingredient") into a sparse vector
4. Change categorical column "Title" into numerical variables
5. Normalize the variables

When I built the models, I can use Logistic regression, Naive Bayes, Decision tree, Random forest, Gradient Boosted Tree, LinearSVM, (Multilayer perceptron) for Binary classification.
For Multi-class classification, there are Multinomial logistic regression, Naive Bayes, Decision tree, Random forest, LinearSVM, (Multilayer perceptron)

In this task, I used 'calories', 'fat', 'protein', 'rating', 'sodium', 'sparse vector' of ingredients, and 'title' feature for building my models. I created the target with labeled and check if the target is imbalanced or not because the accuracy is not good if the target is imbalanced. So, I modify the threshold to classify the label and choose the best threshold to split the similar amount of labels. And, I also performed Hyper-parameter Tuning to improve my models.

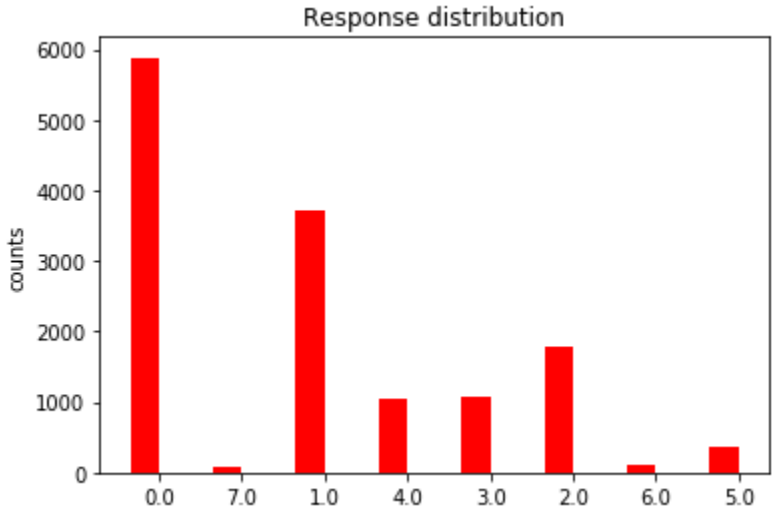


Figure 25: Multi-class classification

```
from pyspark.sql.functions import col, when
data = data.withColumn('label',when(col('protein')>=20,1.0).otherwise(0))
```

```
data.groupBy("label") \
    .count() \
    .orderBy(col("count").desc()) \
    .show()
```

```
+-----+-----+
|label|count|
+-----+-----+
|  0.0| 9356|
|  1.0| 4666|
+-----+-----+
```

```
from pyspark.sql.functions import col, when
data = data.withColumn('label',when(col('protein')>=30,1.0).otherwise(0))
```

```
data.groupBy("label") \
    .count() \
    .orderBy(col("count").desc()) \
    .show()
```

```
+-----+-----+
|label|count|
+-----+-----+
|  0.0|10603|
|  1.0| 3419|
+-----+-----+
```

Figure 26: Imbalanced data set based on threshold

As shown in the Figure 26, unbalanced data set is a very common in real world. In binary/multi-class classification problems, accuracy can be misleading if one class is much more common than another, this is when the classes are unbalanced. The distribution looks quite skewed. There are some techniques to handle the imbalanced data sets: Collecting more data and resampling your dataset (Over-sampling and down-sampling) In this case, I used a method to change the threshold before classifying the labels for targets. After changing the threshold, we can get the balanced target variables, which help us improve our models' output. After solving the imbalanced data set issues, my models' accuracy has improved further.(e.g logistic regression: 34% $\rightarrow$ 81%). Imbalanced dataset can distort prediction so using down/over-sampling make the number of different labels become the similar number or using value transformation(e.g log transformation) can change biased value distribution into normalized distribution. It can help us build the unbiased models and generalize our model to predict all the cases.

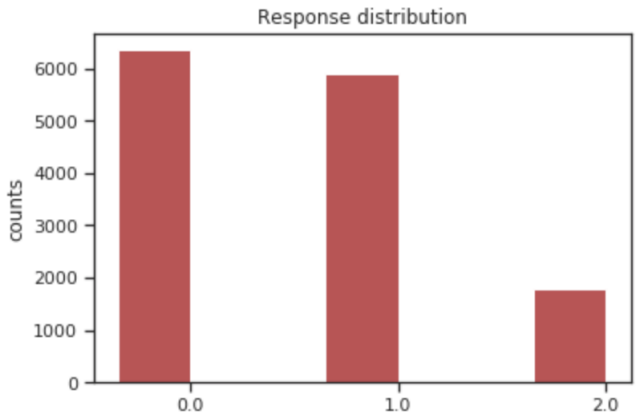


Figure 27: Balanced data set based on threshold

6.1 Logistic Regression and Results

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. The logit function takes input values in the range 0 to 1 and transforms them to values over the entire real number range. Because many data sets in this case are unbalanced, a logistic model using the full set of variables could perform badly compared with a set that has near-zero variance predictors removed. In spark.ml logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multi-class outcome by using multinomial logistic regression. Logistic regression techniques are different between the rdd and dataframe. Also, depending on binary/multinomial logistic regression, the analysis methods are various.

1) Target: Rating labels (0, 1, 2)

I used 'calories', 'fat', 'protein', 'rating', 'sodium' features to build a multinomial logsitic regression. I classify the rating labels through StringIndexer and it automatically generates the eight labels of rating. Actually, the rating has eight different values: 5.0 (1786 EA), 4.375(5884 EA), 3.75(3717 EA), 3.125 (1033 EA), 2.5 (353 EA), 1.875 (68 EA), and 1.25 (105 EA). Therefore, in the first time, accuracy is around 20%. The values in rating are fixed so changing the threshold to change the range of labels has limitations. Despite of this limitation, I experimented a lot of methods to increase the accuracy through changing the feature selections/rating labels, balancing the number of labels, normalizing the feature variables. And, I built the best models to predict the three labels of rating by using 'calories', 'fat', 'protein', 'rating', and 'sodium'. And, we check the lrModel.coefficientMatrix from the lrModel (or analogous values in the SVM model) and find the large values below:

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
# Fit the model
lrModel = lr.fit(trainingData)

# Print the coefficients and intercept for multinomial logistic regression
print("Coefficients: \n" + str(lrModel.coefficientMatrix))
print("Intercept: " + str(lrModel.interceptVector))
```

```
Coefficients:
3 X 5 CSRMatrix
(0,3) -0.2245
Intercept: [0.9837181944314755,0.12001569429435821,-1.1037338887258337]
```

Figure 28: Multinomial Logistic Regression

Coefficients: 3 X 5 CSRMatrix means that there are 3 classes and 5 features('calories', 'fat', 'protein', 'rating', 'sodium') used to build the models. And, there are coefficent values which means the contributions to the labels.

2) Target: Protein labels (Low protein: 0, High protein: 1)

I used 'calories', 'fat', 'protein', 'rating', 'sodium', 'sparsevector' of ingredent for building the models. For this target, I used the binary target, not multi-class targets to compare these two methods.

```
from pyspark.sql.functions import col, when
data = data.withColumn('label',when(col('protein')>=10,1.0).otherwise(0))
```

```
data.groupBy("label") \
    .count() \
    .orderBy(col("count").desc()) \
    .show()
```

```
+-----+-----+
|label|count|
+-----+-----+
|  0.0| 7209|
|  1.0| 6813|
+-----+-----+
```

Figure 29: The number of labels on Balanced data

I changed the threshold to get the balanced data set and built my best logistic model.

IDF down-weighs features which appear frequently in a corpus. This generally improves performance when using text as features since most frequent, and hence less important words, get down-weighed. That is, we can give the weights for text features. It is a way to normalize their values and it works well.

```
from pyspark.ml.feature import IDF
#IDF down-weighs features which appear frequently in a corpus.
#This generally improves performance when using text as features since most frequent,
# and hence less important words, get down-weighed.
idf = IDF(inputCol="rawFeatures", outputCol="sparsevector")
idfModel = idf.fit(featureizedData)
rescaledData = idfModel.transform(featureizedData)
rescaledData.select("label","sparsevector").show(10) # We need only the label and features columns #for ML models
```

```
+-----+--------------------+
|label|        sparsevector|
+-----+--------------------+
|  1.0|(1748,[7,10],[2.5...|
|  1.0|(1748,[0,1,2,5,19...|
|  0.0|(1748,[9,11,20,52...|
|  1.0|(1748,[29,59],[3....|
|  1.0|(1748,[30,400],[3...|
|  0.0|(1748,[1,6,26,38]...|
|  1.0|(1748,[4,7,21,30]...|
|  0.0|(1748,[2,19,59],[...|
|  1.0|(1748,[1,5,8,10,1...|
|  1.0|(1748,[2,14,35,86...|
+-----+--------------------+
only showing top 10 rows
```

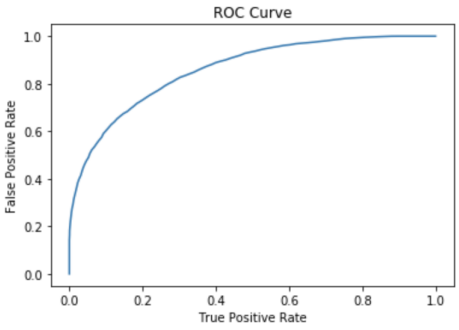
Figure 30: IDF down-weighs features

```
print("Coefficients: \n" + str(lrModel.coefficientMatrix))
print("Intercept: " + str(lrModel.interceptVector))
```

```
Coefficients:
DenseMatrix([[ 2.74930829e-07,  4.63021691e-06,  5.45653501e-05, ...,
              -4.55429747e-01, -6.04806620e-01,  0.00000000e+00]])
Intercept: [-0.6438865765242009]
```

Figure 31: Logistic Regression Model's Coefficient

We can obtain the coefficients by using Logistic Regression Model's attributes.




```
Training set areaUnderROC: 0.859193703337
```

Figure 32: ROC curve in Logistic Regression

For Training data set, Area Under ROC is 0.8591. For Test data set, Area Under ROC is 0.7533. In order to improve the logistic regression without changing the selected features, I used Hyperparameter Tuning and Test Area Under ROC becomes 0.7836.

3) Target: Fat labels (Low fat: 0, High fat: 1) & (Low fat: 0, Medium fat: 1, High fat: 2)
In this task, I attempted to build the models based on the fat labels. Using the amount of fat columns, I classify the labels and made two different targets. (Binary target and Multi-class target). I experimented the logistic models and figure out how to improve their models depending on different kinds of techniques.

a) Binary logistic regression model

```
from pyspark.sql.functions import col, when
data = data.withColumn('label',when(col('protein')>=10,1.0).otherwise(0))
```

```
data.groupBy("label") \
    .count() \
    .orderBy(col("count").desc()) \
    .show()
```

```
+-----+-----+
|label|count|
+-----+-----+
|  0.0| 7209|
|  1.0| 6813|
+-----+-----+
```

Figure 33: The number of labels on Balanced data

I used 'calories', 'fat', 'protein', 'rating', 'sodium', 'title' for building a binary logistic model. In the first time, the model's test accuracy is not high. So, I use 'normalizer' methods to normalize the features. This works very well.

```
# Normalize each Vector using $L^1$ norm.
from pyspark.ml.feature import Normalizer
normalizer = Normalizer(inputCol="features_", outputCol="features", p=1.0)
rescaledData5 = normalizer.transform(rescaledData5)
print("Normalized using L^1 norm")
rescaledData5.select('features').show(3,False)
```

```
Normalized using L^1 norm
+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
|features                                                                                                                                                      |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
|[0.04346717004234478,7.142492729962757E-4,0.0030610683128411815,2.5508902607009846E-4,0.057037906229274016,0.8954645171164737]                               |
|[0.09785412935927398,0.00558472698576501 6,0.004370655901903057,0.0010623121983792151,0.34940965793547213,0.5417185176192066]                               |
|[0.05023213334348124,0.002131060202450719,0.0018266230306720 45,0.001141639394170028,0.05023213334348124,0.8944364106857448]                                |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
only showing top 3 rows
```

Figure 34: Normalize the features

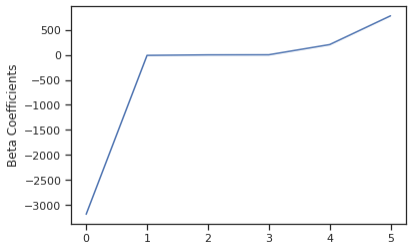
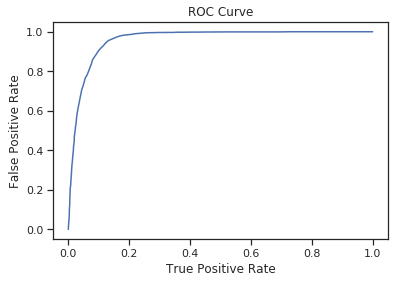


Figure 35: Logistic Regression Model's Coefficient

'title' coefficient is high so this value contributes the prediction a lot. Coefficients: DenseMatrix([[-1.24497140e+01, 7.76993362e+02, 2.03013598e+02, -3.18219011e+03, -2.04617066e+00, -4.17755824e-01]]) Intercept: [-0.7099391215126974]




```
Training set areaUnderROC: 0.960185661289
```

Figure 36: ROC curve in Logistic Regression

For Training data, Area Under ROC is 0.960168014871 and for Test data, Area Under ROC is 0.965465563966. This model fit the data set well.

b) Multinomial logistic regression model

I used 'calories', 'fat', 'protein', 'rating', 'sodium' for building a binary logistic model. Before building the models, I normalized the features and tried to classify the balanced labels.

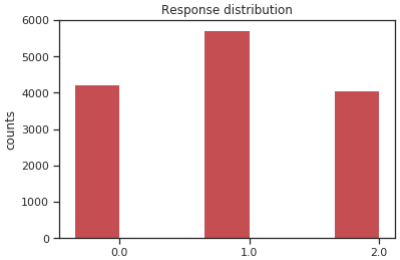


Figure 37: The number of "fat" labels

We can check the coefficient depending on each labels. Coefficients are 3 X 5 CSRMatrix. It means that there are 3 labels with 5 features. As shown in the figure 38, we can check which factor contributes to each label. Test Area Under ROC is 0.8430108544286352, which is reasonable to predict the target variable.

```
print("Coefficients: \n" + str(lrModel2.coefficientMatrix))
print("Intercept: " + str(lrModel2.interceptVector))
```

```
Coefficients:
DenseMatrix([[ 4.41108128e+00, -1.66138806e+02, -3.18405528e+01,
               2.27905187e+02,  7.54387884e-03],
             [ 3.34855879e-01,  3.77222732e+01,  5.38688985e+00,
              -3.45833350e+01, -2.85229815e-01],
             [-4.74593716e+00,  1.28416533e+02,  2.64536629e+01,
              -1.93321852e+02,  2.77685936e-01]])
Intercept: [0.5916314979622412,0.054603071650848586,-0.6462345696130898]
```

Figure 38: Logistic Regression Model's Coefficient

6.2 Decision Tree and Results

This decision trees are a type of supervised machine learning where the data is continuously split according to a certain parameter. The tree can be explained by two entities: decision nodes and leaves. The leaves are the outcomes. And the decision nodes are where the data is split. Decision trees are able to handle both continuous and categorical variables and Decision trees provide a clear indication of which fields are most important for prediction or classification. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multi-class classification, do not require feature scaling, and are able to capture non-linearity and feature interactions.

1) Target: Rating labels (0, 1, 2)

I used 'calories', 'fat', 'protein', 'rating', 'sodium'. Test Area Under ROC is 1.0. One simple decision tree can performed poor because it is too weak given the range of different features. The prediction accuracy of decision trees can be improved by Ensemble methods, such as Random Forest and Gradient-Boosted Tree. DT will find the most significant independent variable to create a group.

2) Target: Protein labels (Low protein: 0, High protein: 1)

I used 'calories', 'fat', 'protein', 'rating', 'sodium', 'sparsevector' of ingredent for building tree models. Test Area Under ROC is 1.0. For binary classification in this data set, Decision tree models are very powerful to predict the target as we can see the output from Pyspark.

3) Target: Fat labels (Low fat: 0, High fat: 1) & (Low fat: 0, Medium fat: 1, High fat: 2)

For binary classification, I used 'calories', 'fat', 'protein', 'rating', 'sodium', 'title' and Test Area Under ROC is 0.929304013466. For multi-class classification, I used 'calories', 'fat', 'protein', 'rating', 'sodium' and Test Area Under ROC is 0.7293630383365266 but accuracy of decision trees can be improved by Random Forest models or Gradient Boosted tree.

6.3 Random Forest and Results

A random forest is an ensemble of decision tress. In order to minimize the over-fitting of decision tree, we try to use Random forest and get better accuracy. This method is good for prediction but a little bit difficult to interpret. Because we have the binary category, random forest is a good classification method. Random forest will grow a big tree without trimming, then, take majority vote of the results of all the trees.

1) Target: Rating labels (0, 1, 2)

I used 'calories', 'fat', 'protein', 'rating', 'sodium'. Test Area Under ROC: 1.0. Random forest is an ensemble of decision tree so Random Forest build multiple decision trees and merges them together and use bagging method.

2) Target: Protein labels(Low protein: 0, High protein: 1)

rfModel.featureImportances can show which features contribute the target a lot. And, this values is very important to select the feature after building the models. When we build the models with high dimensional features, the result can be biased but we can reduce the features through feature selection in random forest. Test Area Under ROC: 0.997323427522.

3) Target: Fat labels (Low fat: 0, High fat: 1) & (Low fat: 0, Medium fat: 1, High fat: 2)
For Binary classification, Test Area Under ROC is 0.977142868502.
As shown in the Figure, feature 1(calories) and feature 3(rating) have high feature importance.

```
1 print rfModel.featureImportances
(6,[0,1,2,3,4,5],[0.07977527198733389,0.5773409545417787,0.03237120444964862,0.2675954459617585,0.007156854830464412,0.03576026
822901588])
```

Figure 39: Random Forest Feature Importance

For multinomial classification, Test Area Under ROC is 0.853870280827628.
As shown in the Figure, feature 2(protein) and feature 4(sodium) have high feature importance.

```
1 print rfModel2.featureImportances
(5,[0,1,2,3,4],[0.015043347598503454,0.4673667134472255,0.026888619582859163,0.47706315022091356,0.013638169150498486])
```

Figure 40: Random Forest Feature Importance

6.4 Linear Support Vector Machines and Results

Support Vector Machine is a supervised machine learning algorithm where it outputs an optimal hyperplane which categorizes the new examples. As well as performing linear classification, SVMs can efficiently perform a non-linear classification using the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.
- SVM's are very good when we have no idea on the data.
- It has a regularization parameter, which makes the user avoid 'over-fitting'.
- Choosing a "good" kernel function is not easy.
- Need to be patient while building SVMs on large data sets. So, it take a lot of time for training.

A support vector machine constructs a hyperplane in a high-dimensional space, which can be used for classification. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data points of any class. LinearSVC in Spark ML supports binary classification with linear SVM. Therefore, I used Linear Support Vector Machine only in the binary classification case.

(1) Target: Protein labels(Low protein: 0, High protein: 1)
'Test Area Under ROC', 0.776822219581915.
print the coefficients and intercept for linear SVC
print("Coefficients: " + str(lsvcModel.coefficients))
print("Intercept: " + str(lsvcModel.intercept))
we also get the coefficients which means that the larger value is, the larger feature importance is. It is similar to binary logistic regression model.

(2) Target: Fat labels (Low fat: 0, High fat: 1)

As shown above, linear support vector machine has analogous values in the logistic model and find the large values. Test Area Under ROC is 0.8768974097868399.

```
from pyspark.ml.classification import LinearSVC

lsvc = LinearSVC(maxIter=10, regParam=0.1)

# Fit the model
lsvcModel = lsvc.fit(trainDF)

# Print the coefficients and intercept for linear SVC
print("Coefficients: " + str(lsvcModel.coefficients))
print("Intercept: " + str(lsvcModel.intercept))
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator()
predictions = lsvcModel.transform(testDF)
print('Test Area Under ROC', evaluator.evaluate(predictions))
```

```
Coefficients: [0.4898529416304575,65.10000241945255,53.43151618569487,-154.94442490197116,-0.760450789914536,-1.06814612882392
3]
Intercept: -0.0673243085841
('Test Area Under ROC', 0.8768974097868399)
```

Figure 41: Linear Support Vector Machine

7 Conclusion/Recommendation

During this task, I spent a lot of time performing feature engineering. Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work. It is essential to the application of machine learning and helps in increasing the accuracy of the model. 80 percent of a data scientist's valuable time is spent simply finding, cleansing, and organizing data, leaving only 20 percent to actually perform analysis. It is really fundamental in creating the right features. I will make the sparse vector to use the text variable. And, I attempt to select the important features to build the best models and also change the parameter in my models to get better accuracy. And, I split the date into discrete components so the decision trees were able to make better guesses. Therefore, Decision tree and Random forest models' accuracy is better than others. According to this feature engineering, our analysis can be more accurate and build better models. Most of work consists of cleaning the data and it is very hard to handle the unstructured data such as Text. Compared to the feature engineering, building machine learning (ML) models is much easier and took less time.

To improve/advance our analysis of data set, deeper data exploration and better feature engineering are needed. Because time is limited, there are other methods that I didn't try. For handling the missing values, I didn't impute the missing values and just deleted them. Because the range is so various, imputation of mean method can be worse. If I replace the missing values with mean/median/median, the result can be changed. I will attempt to impute the values in the future work. There are deleted values such as date, category, etc. This is because I thought these features are hard to analyze and they are not meaningful values. But, if they are included to predict the classification models. The results can be different or improved further. For example, I can create the derived variables such as year, month, day, hour, minute, etc by using 'Date' variable. And, if I change the floating points (change double data type into Integer), my codes can be operated fast. I built the neural network models to predict the labels but it did work well. Therefore, I will build better neural network models through using more advanced techniques for the future work. The target can be as simple as vegetarian vs non-vegetarian, recipes using a certain ingredient vs recipes without that ingredient. In the future work, I build more advanced prediction models through using different target variables.

During this task, I felt that there is no free lunch. Machine learning algorithms make us analyze the data set easily. However, there are various kinds of methods to predict the target variables through using the independent variables. And, we confront with important choices in many parts: feature selection, handling unbalanced data set, building the models. According to the feature selection and machine learning algorithms, the outcome is very different and I got different outcomes. Therefore, there is no perfect ML algorithm. When we face with real world data, it is more difficult to solve the cases. We will keep in mind that there is no such thing as a free lunch and try to choose the optimal model after many trials.

Appendix

A Data Visualization

I will show data visualizations about our features here. There are some outliers in each features. In this task, I didn't handle the outliers but if we handle the outliers through value transformation, the result can be improved. More graphs are found in my python file.

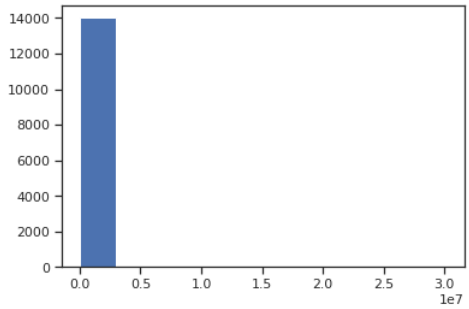


Figure 42: Histogram in calories

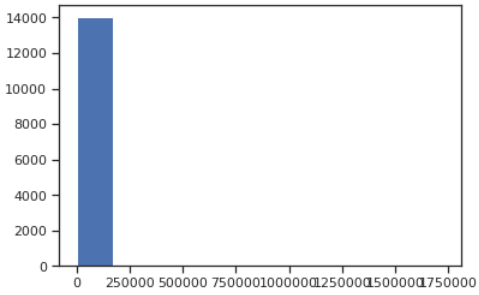


Figure 43: Histogram in fat

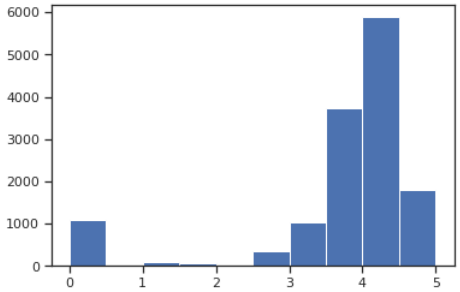


Figure 44: Histogram in rating

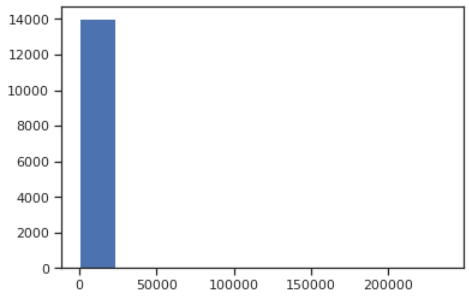


Figure 45: Histogram in protein

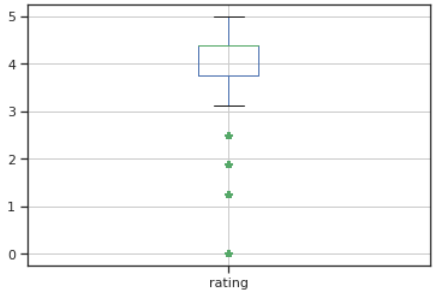


Figure 46: Check outlier in boxplot

B Correlation

In this part, I figure out that there are correlations between features in the data set. This is because if two variables are highly correlated, Multicollinearity can occur and it is hard to interpret the accurate result.

	calories	fat	protein	rating	sodium
calories	1.000000	0.996527	0.803336	0.007580	0.996518
fat	0.996527	1.000000	0.770938	0.006955	0.986456
protein	0.803336	0.770938	1.000000	0.012385	0.819283
rating	0.007580	0.006955	0.012385	1.000000	0.008008
sodium	0.996518	0.986456	0.819283	0.008008	1.000000

Figure 47: Correlation Table

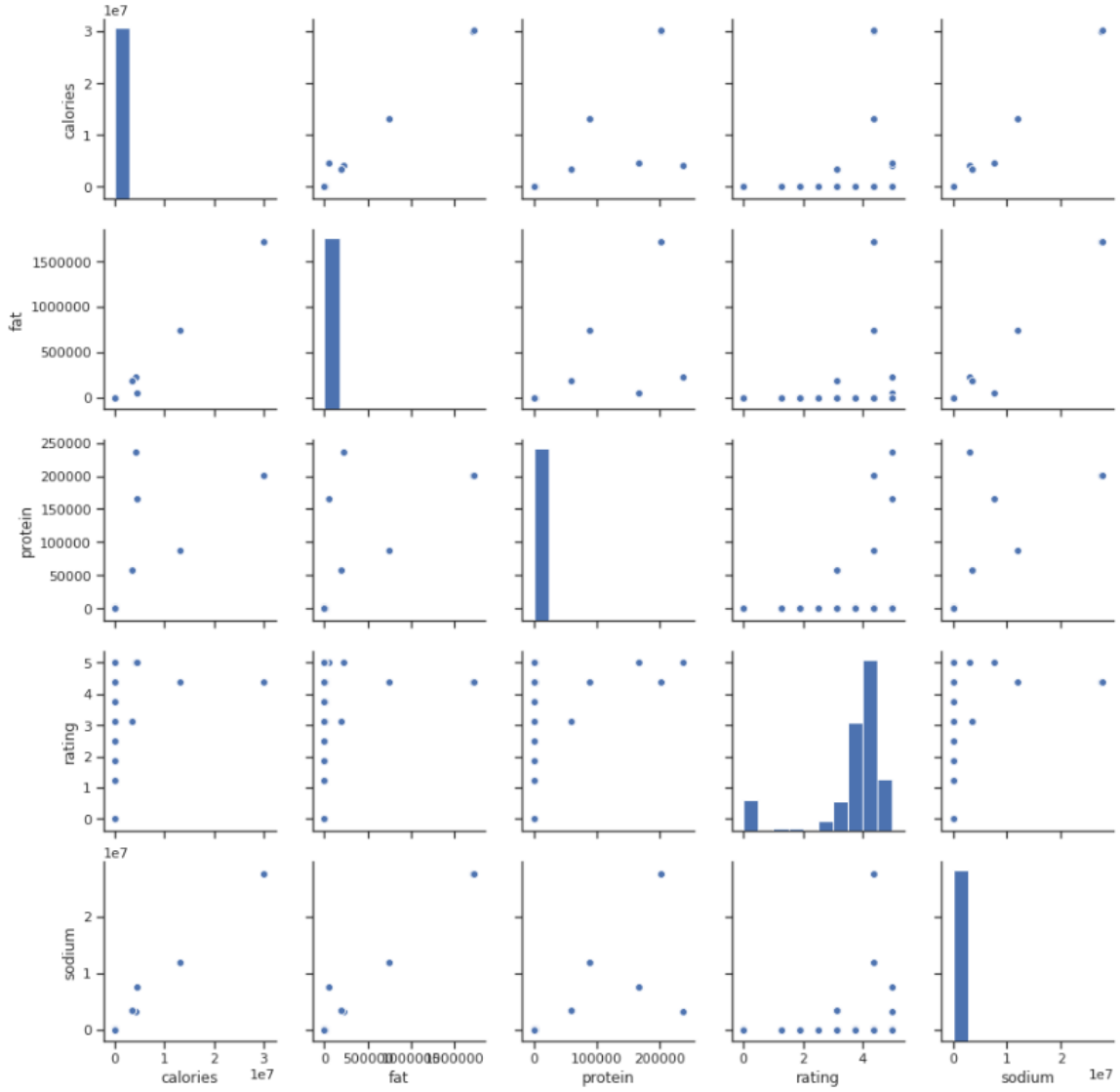


Figure 48: Correlation Plot

C Multilayer perceptron classifier

Multilayer perceptron classifier (MLPC) is a classifier based on the feedforward artificial neural network. MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node's weights w and bias b and applying an activation function.

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Split the data into train and test
splits = rescaledData5.randomSplit([0.7, 0.3], 1234)
train = splits[0]
test = splits[1]

# specify layers for the neural network:
# input layer of size 6 (features), two intermediate of size 3 and 2
# and output of size 2 (classes)
layers = [6, 3, 2, 2]

# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128, seed=1234)

# train the model
model = trainer.fit(train)

# compute accuracy on the test set
result = model.transform(test)
predictionAndLabels = result.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

```
Test set accuracy = 0.786375504391
```

Figure 49: Neural Network for binary classification

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Split the data into train and test
splits = rescaledData8.randomSplit([0.7, 0.3], 1234)
train = splits[0]
test = splits[1]

# specify layers for the neural network:
# input layer of size 5 (features), two intermediate of size 5 and 4
# and output of size 3 (classes)
layers = [5, 8, 7, 3]

# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128, seed=1234)

# train the model
model = trainer.fit(train)

# compute accuracy on the test set
result = model.transform(test)
predictionAndLabels = result.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

```
Test set accuracy = 0.771421789699
```

Figure 50: Neural Network for multi-class classification