Building multi-class e-commerce category classifier with Apache Spark, Apache Solr and Spring Boot in Java.

2016 Mihran Shahinian - {slowmihran} @ gmail com, https://github.com/logisticDigressionSplitter/SparkSpringClassifier

Overview

What:

The intent of this article is to demonstrate a solution that can be used to predict ‘best’ product category(ies) with confidence scores for each user query in real time. The predicted category(ies) will be used to boost products returned by a search engine for increased relevancy of the search results.

Why:

There are many outstanding frameworks and solutions capable of delivering the similar output. This solution is chosen to be self contained Spring Boot app that contains commodity components running on JVM and would likely get approved by architecture boards at companies with large IT organisations.

Disclaimers & Limitations:

- I am not able to provide full source code for the overall solution, but there are enough details for you to be able to build it on your own. I appreciate any feedback you might have or any improvements over the current approach.
- Multiclass classification is limited to number of the classes that can be used for categorization. I will leave it up to you to research or experiment, the practical number seems to be somewhere between 30-40 classes. In my case I was able to achieve 40-50 ms response times with 30 req/sec with my model being trained from ~1 million skus and 100,000 queries on a dual cpu virtual server with 6GB RAM.

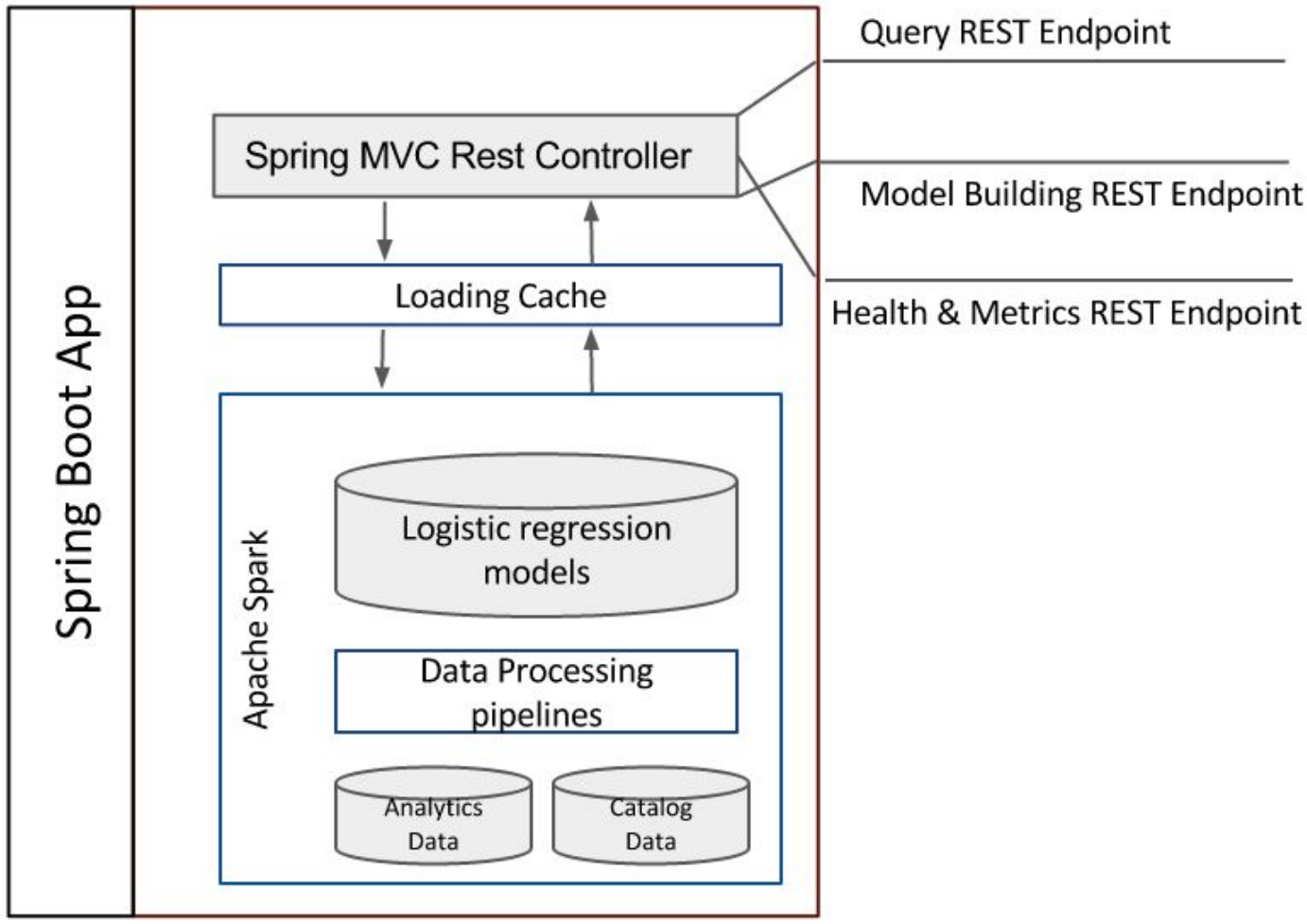
Technology Stack

Apache Spark 2.0.1	Datasets, Pipelines, ML, Mllib
Apache Solr 6.2.0	Tokenization, Search
Spring Boot 1.3.5.RELEASE	Spring MVC, Configuration Management
Google Guava	Loading Cache, Various utility methods
Java 8	Self explanatory

Please refer to the pom.xml for the comprehensive list of all dependencies.

Overall Architecture

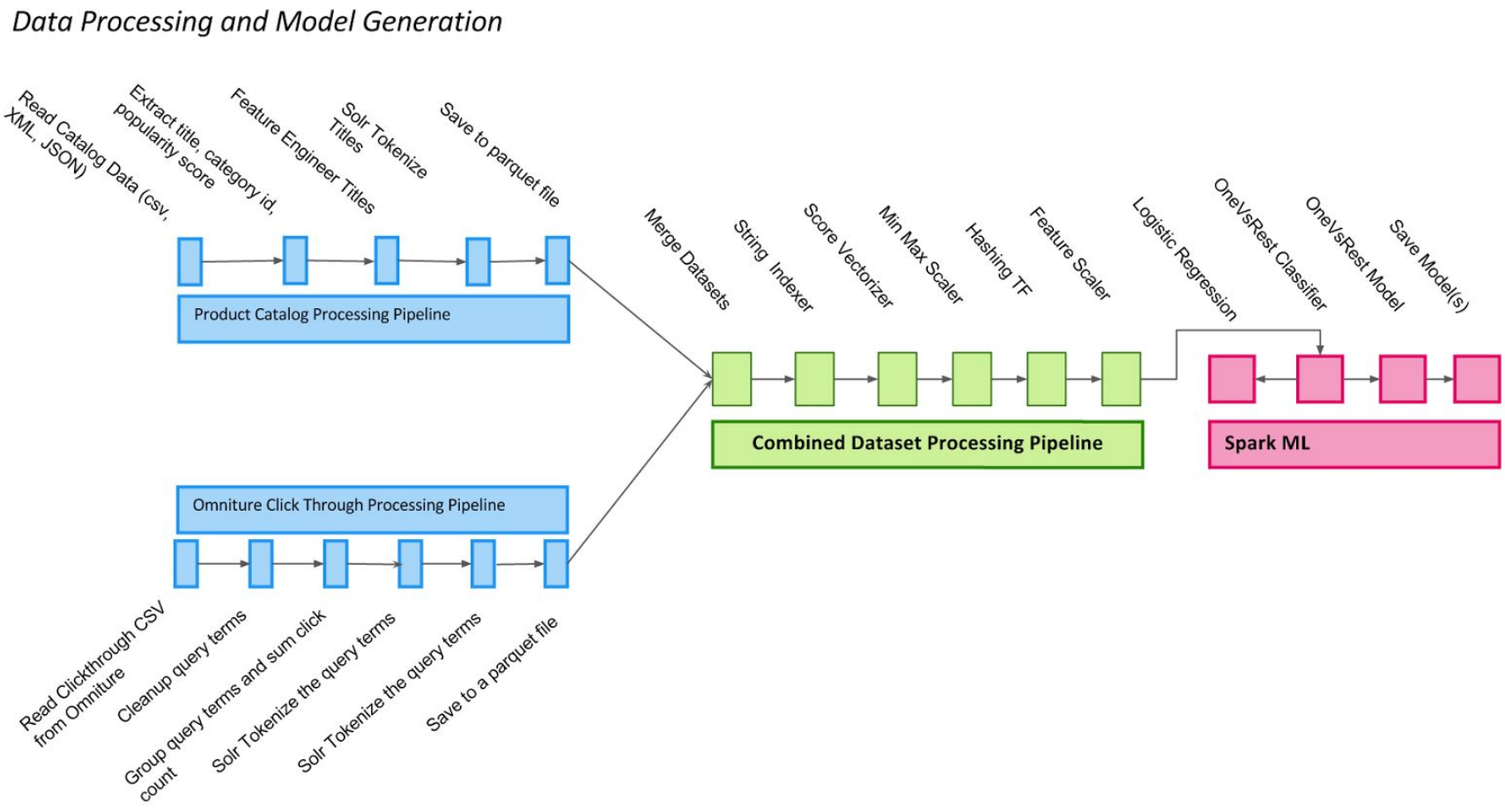
The application uses Spring Boot to manage configuration, provide a rest framework and application essentials such as health and performance metrics. Apache Spark 2.0 is embedded and runs inside Spring container to provide data training and classification capabilities. The application is assembled using maven. E.g ‘mvn package -P prod’. See pom.xml for full details. The data sources can be integrated to pull data directly from your analytics sources such as Omniture or Google Analytics and catalog index such as solr or some jdbc compliant data source.



Classifier App Logical Architecture

Building the Classification Model

Below is the workflow/architecture for merging two data sources (Catalog and Analytics datasets) and generating the multi class model.



Data Processing Pipeline

Let’s take a closer look at the key parts of this pipeline.

Training Data Sources:

Data Source	**Description**	**Fields**
Catalog Data	Sku information	Product Title, Category Name, Popularity Score
Analytics Data	Analytics Information	Search Term, Category of the Product Clicked on Most, Number of Clicks

For the Catalog data source we will extract product titles,category label and compute a product popularity score from your favorite metrics (number of transactions, conversion rate, etc). For the sake of simplicity we will save this data into a csv file.

Product Title (text)	Category (categoryId)	Popularity Score (rawScore)
Stainless Steel 1.6 Cu. Ft. Over-the-Range Microwave	Small Appliances	20
4.2 cu. ft. 5 burner Gas Range w/ Broil Stainless Steel	Appliances	10

Our next data source is analytics click-through data. For every search term we resolve the category for products that got most clicks cumulatively.

Search Term (text)	Category (categoryId)	Number of Clicks (rawScore)
Nest thermostat	Appliances	20000
netgear nighthawk	Wireless Routers	1000

The labels in parenthesis map to Spark Dataset columns. You can name them as you see fit.
Now that we have our data let’s go through putting this data through pipelines.

Spark Pipelines and Transformers

Descriptions for select transformations steps are listed below:

Read a csv or xml file	Read analytics clickthrough file or catalog data with xml/csv. We use databricks xml or csv extensions to accomplish this.. For more info take a look here https://github.com/databricks/spark-csv https://github.com/databricks/spark-xml	SparkSession sparkSession …. sparkSession.read() .format("com.databricks.spark.csv") .option("header", "true") .option("inferSchema", "true") .load(dataPath)
SolrSparkTokenizer Transformer	Ideally we want to use exact same	See this link for implementation

	tokenization method to preprocess our data as we use for search engine query tokenization. If we use different techniques we risk creating “false positive” predictions.	
MinMaxScaler	This is an existing Spark transformer that can be used to rescale a range of numbers to another range.	Let’s say you have one set of numbers that represent number of orders and are in the range of 1-400. Another range of numbers within 10-10000 is number of clicks for a product. We can use MinMax scale to map those for instance within the range of 1-30 and use some sort of formula to calculate popularity score.
Feature Scaler	This custom transformer takes vector that is generated by HashingTF and multiplies it out each vector entry by the score computed with MinMaxScaler essentially amplifying scores for tokens that have higher popularity score.	
Feature Engineering	Note: One of the critical tasks would be to preprocess (aka feature engineer) product titles and queries to keep only essential information that is good for training purposes. Example: Product title with “stainless steel microwave” would get rewritten to “microwave”, since “stainless steel” can describe bunch of other products in different categories. Example: Query “lens for canon eos camera” would get rewritten to “lens”	

Putting together the pipeline

Below is the code listing for wiring together the pipeline as described. So far we have been using org.apache.spark.ml packages once we save OneVsRest model we need to load it from Spark org.apache.spark.mllib for querying. The suffix “_ColumnName” is added intentionally to underline that these map to columns in Spark’s dataset.

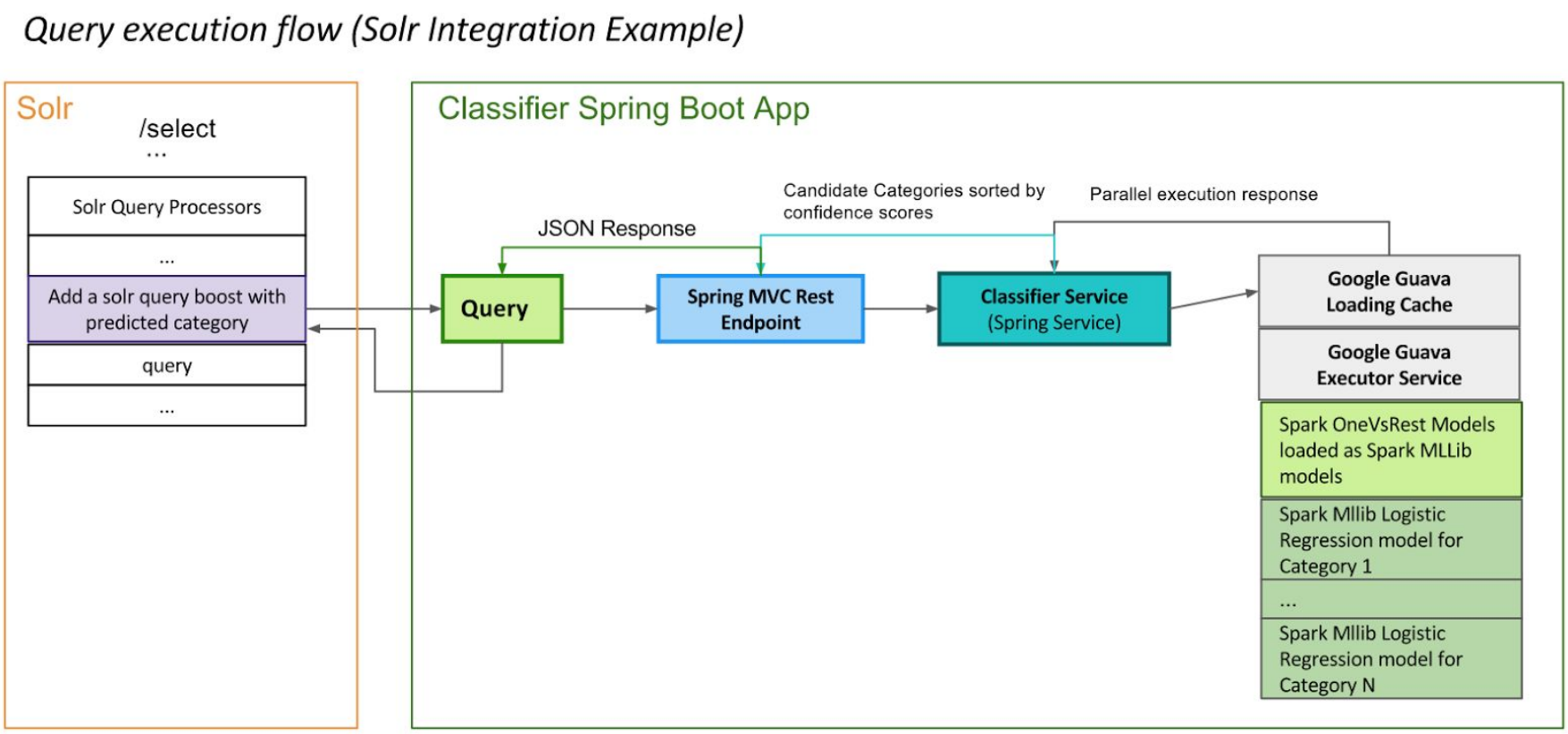
```
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.classification.LogisticRegressionModel;
import org.apache.spark.ml.classification.OneVsRest;
import org.apache.spark.ml.classification.OneVsRestModel;
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator;
import org.apache.spark.ml.feature.HashingTF;
import org.apache.spark.ml.feature.IDF;
import org.apache.spark.ml.feature.IndexToString;
import org.apache.spark.ml.feature.MinMaxScaler;
import org.apache.spark.ml.feature.StringIndexer;
import org.apache.spark.ml.feature.StringIndexerModel;

// Preprocessed catalog dataset
Dataset<Row> catalogDataset = getProductsCatalog(sparkSession,"data/source/parquet/catalog.parquet");

// Preprocessed analytics dataset
Dataset<Row> analyticsDataset = OmnitureData.getOmnitureDataSet(sparkSession, "data/source/parquet/analytics.parquet");

Dataset<Row> combinedDataSet = catalogDataset.union(analyticsDataset);

// Split dataset into training and validation
Dataset<Row>[] splits = joined.randomSplit(new double[] {0.7,0.3});

Dataset<Row> trainingSet = splits[0];

StringIndexer() stringIndexerModel = new StringIndexer().setInputCol("categoryId_ColumnName")
                                    .setOutputCol("categoryIndex_ColumnName")
                                    .fit(trainingSet);
```

```
// we need to vectorize our raw score
VectorAssembler scoreVector = new VectorAssembler().setOutputCol("scoreVectorized_ColumnName")
                                                   .setInputCols(new String[] {"rawScore_ColumnName");

// Custom Transformer
SolrSparkTokenizer tokenizer = new SolrSparkTokenizer().setInputCol("text_ColumnName").setOutputCol("tokenized_ColumnName");

HashingTF hashingTF = new HashingTF().setInputCol("tokenized_ColumnName")
                                     .setOutputCol("features_ColumnName")
                                     .setNumFeatures(20000);

MinMaxScaler mmScaler = new MinMaxScaler().setInputCol("scoreVectorized_ColumnName")
                                          .setOutputCol("scoreMinMaxScaled")
                                          .setMax(40.0)
                                          .setMin(1.0);
 // Custom Transformer
 FeatureScoreScaler featureScoreScaler = new FeatureScoreScaler("features_ColumnName",
                                                 "scoreMinMaxScaled");

  // Pipeline
  Pipeline processingPipeline = new Pipeline().setStages(new PipelineStage[] {stringIndexerModel,
                                                               scoreVector,
                                                               mmScaler,
                                                               hashingTF,
                                                               featureScoreScaler
   });




      Dataset<Row> mlReady = processingPipeline.transform(trainingSet)
                                     // drop overhead
                                          .drop("tokenized_ColumnName")
                                          .drop("rawScore_ColumnName");



      // Logistic Regression - adjust params as you see fit
      LogisticRegression lr = new LogisticRegression().setFeaturesCol("features_ColumnName")
                                           .setLabelCol("categoryIndex_ColumnName")
                                           .setMaxIter(140)
                                           .setRawPredictionCol("confidenceScore")
                                           .setFitIntercept(true);

      // One vs Rest
      OneVsRest oneVsRestClassifier = new OneVsRest().setClassifier(lr)
                                         .setPredictionCol("prediction_ColumnName")
                                         .setLabelCol("categoryIndex_ColumnName");
      // Train model and Save
      OneVsRestModel ovrModel = oneVsRestClassifier.fit(mlReady);
      ovrModel.write()
            .overwrite()
            .save("some directory path");

      // Omitted for brevity ...
      // Evaluate the model with remaining 30% of data
      // tune to increase accuracy
      // See Spark documentation for cross validation
```

Next we will look at the query time classification with the model we have generated.

Query Time Classification

The diagram below depicts query execution workflow. It consists of a Solr component querying a rest endpoint that we have configured in our spring boot application to query the model OneVsRest model we have saved in the previous step. The key here is not to use the transform(Dataset<?> dataset) method on ml.classification.OneVsRestModel rather than load model for each classification class individually and query in parallel. This avoids code generation overhead on creating the Dataset<?> for each query.



The steps for the execution flow are as follows.

1. Load the OneVsRest model from disk and get a handle to it.
2. For each incoming query:
    a. Tokenize it using the same tokenizer that was used to build the OneVsRest model.
    b. Run the tokenized input through HashingTF with the same feature count as it was used for generating OneVsRest model
    c. Query all individual OneVsRest class models in parallel with the vector generated from HashingTF
    d. Process results, compute the confidence score for each model and filter out prediction based on confidence threshold.
3. Wrap the results with a Loading Cache.

The general code listing for the approach is below.

```
// Load the previously generated model from disk
org.apache.spark.ml.classification.OneVsRestModel myOneVsRestModel = loadModelFromDisk(String somePath);

// Number of classes our model contains.
Integer classCount = myOneVsRestModel.models().length;

// Use Google Guava executor service to configure a service to query in parallel all the models.
ListeningExecutorService executorService = MoreExecutors.listeningDecorator(Executors.newCachedThreadPool());

// Get an instance to hashingTF - note we need to use the same number of features we used during model generation
org.apache.spark.mllib.feature.HashingTF hashingTF = new HashingTF(20000);


/** Here is what your method to classify queries would do */

// 1) Get an instance of the same solrAnalyzer that was used to analyze text for model and tokenize the incoming query
// and tokenize the input query
List<String> tokenized = solrAnalyzer.tokenize(query);

// 2) Generate feature vector using hashingTF
```

```
Vector queryVector = hashingTF.transform(tokenized).asML();

// 3) Create a ClassificationTask that will encapsulate the classification task for each model to feed to the executorService
    // sigmoid function for confidence score
    public static Double sigmoid(Double x) {
      return 1.0 /
           (1.0 +
            Math.exp(-x));
    }


public class ClassifyTask implements Callable<ClassificationResult> {
                        // call the model for given class index
                        @Override
                        public ClassificationResult call() throws Exception {
                                LogisticRegressionModel lrModel = (LogisticRegressionModel) myOneVsRestModel.models()[classId];
                                Vector rawPrediction = lrModel.predictRaw(vector);
                                Double confidenceScore = (1 - sigmoid(rawPrediction.apply(0))) * 100;
                        }
 }

// For each class add classification task to the list of tasks and pass it to the executorService to execute in parallel.
List<ClassifyTask> classificationTasks = Lists.newArrayList();
for (int i = 0; i < classCount; i++) {
     ClassifyTask task = ClassifyTask.builder()
                        .query(query)
                        .tokenized(tokenized)
                        .vector(queryVector)
                        .classId(i)
                        .build();
    classificationTasks.add(task);
 }

// predict query label for all the classes
List<Future<ClassificationResult>> futureResults = executorService.invokeAll(classificationTasks
                                                        100,
                                                        TimeUnit.MILLISECONDS);
// 4) Iterate through futureResults, sort them by confidence score, filter out results based on acceptable threshold
// of confidence scores and return the top classification results

/** End Classification method **/
```

Overall implementation steps

1. Start a Spring Boot project (Spring Starter from Eclipse STS) See reference Maven pom.xml file.
2. Pull the common parameters such as HashingTF number of features, loading cache size, model directories into application.yml
3. Configure Spring MVC REST endpoints.
    a. /query - would take the end-user query and return the predicted category(-ies) that can be used in Solr (any search engine) to boost products in those category(-ies).
    b. /build - this would trigger model rebuilding from potentially updated dataset
    c. /reload - reload the model from disk
4. Annotate REST endpoints to keep track of execution time (com.codahale.metrics.annotation.Timed;)
5. Classify queries and boost your categories to improve relevancy (e.g. use bq parameter in Solr)