Getting started with Tensorflow and Java-Spring

June 29, 2018

by Enrique Llerena Domínguez (https://blog.mimacom.com/author/enriquedominguez/)

Java (https://blog.mimacom.com/tag/java/)

Spring (https://blog.mimacom.com/tag/spring/)

Machine Learning (https://blog.mimacom.com/tag/machine-learning/)

Introduction

One of the goals I set for this year is to explore Machine Learning (ML), so after having done a couple of courses here and there, I decided to do a -rather simple- starting project, where I could deal with some of the basic stages of the ML: Get the data, prepare it, choose a model, train it, evaluate it, export it, and make the predictions available for use. For this first project, I chose:

- Tensorflow (https://www.tensorflow.org/) as the framework,
- Python (v3.5), with Jupyter Notebooks as the model generating part,
- Java, using Spring Boot as the prediction serving part.

Environment setup

Note: These steps were executed in Windows. The full code for this post can be found at https://github.com/ellerenad/Getting-started-Tensorflow-Java-Spring (https://github.com/ellerenad/Getting-started-Tensorflow-Java-Spring)

Training part

- Download Anaconda (https://www.anaconda.com/download/).
- Create an environment using:

```
conda create -n tensorflow pip python=3.5
```

> In this example of the conda create command, "tensorflow" corresponds to the name of the environment.

(https://www.mimacom.com/en/about/#offices-anchor)

- Activate your recently created environment

```
activate tensorflow
```

- Install Jupyter, Tensorflow, and other required packages:

```
pip install --ignore-installed --upgrade jupyter
pip install --ignore-installed --upgrade tensorflow
pip install --ignore-installed --upgrade scipy
pip install --ignore-installed --upgrade pandas
pip install --ignore-installed --upgrade sklearn
```

- Execute Jupyter Notebook

```
jupyter notebook
```

From now on, to run your notebooks, you just need to open the Anaconda Prompt and execute:

```
activate tensorflow
jupyter notebook
```

Using the jupyter notebook webapp (by default opened at http://localhost:8888 (http://localhost:8888)), create a new notebook, or use the following, found on the Github repository (https://github.com/ellerenad/Getting-started-Tensorflow-Java-Spring): `./training/TF_iris_data.ipynb`

Serving part

- Install Java SDK
- Install Maven
- Use the following parent POM and dependencies:

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Camden.SR7</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.tensorflow</groupId>
        <artifactId>tensorflow</artifactId>
        <version>1.8.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

> Note: This POM shows just the required dependencies, and is not in the required final form.

Describing the problem and the data set

We will use the famous Iris Data Set, where different types of Iris flowers are classified based on some of its features, like the length and width of its petal and sepal, resulting into 3 different categories: *Setosa*, *Versicolour*, or *Virginica*. We will use supervised training and a neural network classifier. More information about the data set at the scikit learn website (http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html) and Wikipedia (https://en.wikipedia.org/wiki/Iris_flower_data_set).

Describing the -rather basic- architecture

As previously discussed, we have 2 components: The training component, written in Python, and the server component, written in Java. The output of the former is a trained and evaluated model, which is the one of the inputs of the latter. This is possible because we are using the Tensorflow framework on both components.

Describing the training component

In this component we will perform the following steps:

1. Get the data
2. Prepare the data
3. Partition the data into train and evaluation/test sets
4. Format the data as required by Tensorflow
5. Train the model
6. Evaluate the model
7. Export the model

(https://www.mimacom.com/en/about/#offices-anchor)

Now, let's proceed with the code: (Finally!)

Prepare the data and train the model

Import the modules:

```
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn import datasets
```

Define *"constants"* for the names of the features:

```
FEATURE_SEPAL_LENGTH = 'SepalLength'
FEATURE_SEPAL_WIDTH = 'SepalWidth'
FEATURE_PETAL_LENGTH = 'PetalLength'
FEATURE_PETAL_WIDTH = 'PetalWidth'
LABEL = 'label'
```

Get the data:

```
# load the data set
iris = datasets.load_iris()
```

Since the data comes originally with all the examples ordered, we need to shuffle it to get a meaningful test set. To achieve this, we first need to add the target (the label each set of measures correspond to) to the data, and then shuffle it:

```
iris_data_w_target = [];

# add the target to the data
for i in range(len(iris.data)):
    value = np.append(iris.data[i], iris.target[i])
    iris_data_w_target.append(value)
```

Create a Pandas Data Frame to operate with, and shuffle the data:

```
columns_names = [FEATURE_SEPAL_LENGTH, FEATURE_SEPAL_WIDTH, FEATURE_PETAL_LENGTH, FEATURE_PETAL_WIDTH, LABEL]

df = pd.DataFrame(data = iris_data_w_target, columns = columns_names )

# shuffle our data
df = df.sample(frac=1).reset_index(drop=True)
```

Having done the shuffling of the data, we can partition it into training and evaluation/test sets. We will reserve 20% of the original set for evaluation/testing, whilst the model will be trained with the rest 80%:

```
test_len = (len(df) * 20)//100;
training_df = df[test_len:]
test_df = df[:test_len]
```

After that, we format the data for Tensorflow. So far, we have stored our data in a Pandas Data Frame, which represents a data table, but Tensorflow expects to receive a map, were the keys are the names of the features, and the correspondent values are arrays storing the same data as the columns from our Pandas Data Frame. So, we first declare the columns we will be using, and then we create the map using the Pandas Data Frame with our training data.

```
iris_feature_columns = [
    tf.contrib.layers.real_valued_column(FEATURE_SEPAL_LENGTH, dimension=1, dtype=tf.float32),
    tf.contrib.layers.real_valued_column(FEATURE_SEPAL_WIDTH, dimension=1, dtype=tf.float32),
    tf.contrib.layers.real_valued_column(FEATURE_PETAL_LENGTH, dimension=1, dtype=tf.float32),
    tf.contrib.layers.real_valued_column(FEATURE_PETAL_WIDTH, dimension=1, dtype=tf.float32)
]

x = {
    FEATURE_SEPAL_LENGTH : np.array(training_df[FEATURE_SEPAL_LENGTH]),
    FEATURE_SEPAL_WIDTH  : np.array(training_df[FEATURE_SEPAL_WIDTH]),
    FEATURE_PETAL_LENGTH : np.array(training_df[FEATURE_PETAL_LENGTH]),
    FEATURE_PETAL_WIDTH  : np.array(training_df[FEATURE_PETAL_WIDTH])
}
```

Then, we instantiate the model and train it. We will use a Neural Network Classifier, with 5 nodes and 5 hidden layers, which has an output of 3 different classes.

```
classifier = tf.estimator.DNNClassifier(
       feature_columns = iris_feature_columns,
       hidden_units = [5, 5],
       n_classes = 3)


# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x = x,
    y = np.array(training_df[LABEL]).astype(int),
    num_epochs = None,
    shuffle = True)

# Train model.
classifier.train(input_fn = train_input_fn, steps = 4000)
```

(https://www.mimacom.com/en/about/#offices-anchor)

Evaluate the model

Once we have the model trained, we proceed to evaluate it using the evaluation/test set we separated earlier:

```
x = {
    FEATURE_SEPAL_LENGTH : np.array(test_df[FEATURE_SEPAL_LENGTH]),
    FEATURE_SEPAL_WIDTH  : np.array(test_df[FEATURE_SEPAL_WIDTH]),
    FEATURE_PETAL_LENGTH : np.array(test_df[FEATURE_PETAL_LENGTH]),
    FEATURE_PETAL_WIDTH  : np.array(test_df[FEATURE_PETAL_WIDTH])
}

# Define the training inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
    x = x,
    y = np.array(test_df[LABEL]).astype(int),
    num_epochs = 1,
    shuffle = False)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("Test Accuracy: ", accuracy_score)
```

Output:

```
INFO:tensorflow:Saving dict for global step 4000: accuracy = 1.0, average_loss = 0.03394517, global_step = 4000, loss = 1.018355
Test Accuracy:  1.0
```

Now, we can also do some more manual testing of our model, if required. To do so, we take some arbitrary records from the original data set, including its respective target, and feed them to the model.

```
x = {
    FEATURE_SEPAL_LENGTH : np.array([5.0, 6.7, 7.4]),
    FEATURE_SEPAL_WIDTH  : np.array([3.5, 3.1, 2.8]),
    FEATURE_PETAL_LENGTH : np.array([1.3, 4.4, 6.1]),
    FEATURE_PETAL_WIDTH  : np.array([0.3, 1.4, 1.9])
}

expected = np.array([0, 1, 2])

# Define the training inputs
predict_input_fn = tf.estimator.inputs.numpy_input_fn(
    x = x,
    num_epochs = 1,
    shuffle = False)

predictions = classifier.predict(input_fn = predict_input_fn)

for pred_dict, expec in zip(predictions, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]

    print('\nPrediction is "{}" (certainity {:.1f}%), expected "{}"'.format(class_id, 100 * probability, expec))
```

> Note: Since these records are hardcoded, it is highly probable (80%, to be precise ;)) they are part of the training data set. To limit overfitting, it is important not to test the model with the data it was trained with. If this testing step is important for you, please consider improving this piece of code.

Output:

```
Prediction is "0" (certainity 100.0%), expected "0"

Prediction is "1" (certainity 98.0%), expected "1"

Prediction is "2" (certainity 99.8%), expected "2"
```

Export the model

Now that we have evaluated our model, we proceed to export it. To do that, we need to define a function describing the input it will receive, and then call to the `export_savedmodel` method of the classifier itself.

```

def serving_input_receiver_fn():
    serialized_tf_example = tf.placeholder(dtype = tf.string, shape = [None], name = 'input_tensors')
    receiver_tensors      = {'predictor_inputs' : serialized_tf_example}
    feature_spec          = {FEATURE_SEPAL_LENGTH : tf.FixedLenFeature([25], tf.float32),
                             FEATURE_SEPAL_WIDTH  : tf.FixedLenFeature([25], tf.float32),
                             FEATURE_PETAL_LENGTH : tf.FixedLenFeature([25], tf.float32),
                             FEATURE_PETAL_WIDTH  : tf.FixedLenFeature([25], tf.float32)}
    features              = tf.parse_example(serialized_tf_example, feature_spec)
    return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)


model_dir = classifier.export_savedmodel(export_dir_base = "stored_model",
                             serving_input_receiver_fn = serving_input_receiver_fn,
                             as_text = True)
print('Model exported to '+ model_dir.decode())
```


The following output means we have properly exported the model, and it is at `.\stored_model\1530093489`

```
INFO:tensorflow:SavedModel written to: b"stored_model\\temp-b'1530093489'\\saved_model.pbtxt"
Model exported to stored_model\1530093489
```

Having exported our trained model, we are now ready for loading it in Java for the server component :D

Describing the process in java

In this component we will perform the following steps:

1. Publish two GET endpoints to retrieve predictions: The predicted class und a set of probabilities per class.
2. Load the previously saved model.
3. Feed the input to the model and fetch the prediction.
4. Integration testing.
5. Examples of usage

Defining the domain objects

We need two domain objects: The Iris and the possible types of Iris, represented by an enum: IrisType:

```
public class Iris {

    private float petalLength;
    private float petalWidth;
    private float sepalLength;
    private float sepalWidth;

    public Iris() {
    }

    public Iris(float petalLength, float petalWidth, float sepalLength, float sepalWidth) {
        this.petalLength = petalLength;
        this.petalWidth = petalWidth;
        this.sepalLength = sepalLength;
        this.sepalWidth = sepalWidth;
    }
    // ...
    // (Getters and setters omitted)
}
```

```
public enum IrisType {
    SETOSA,
    VERSICOLOUR,
    VIRGINICA
}
```

Exposing the endpoints

Here we expose the two required GET endpoints. Both expect as parameters the features of the Iris:

- petalLength
- petalWidth
- sepalLength
- sepalWidth

The Spring framework will read them from the URL and inject the Iris object in the method.

- The `/iris/classify/class` endpoint returns the predicted class, *Setosa*, *Versicolour*, or *Virginica*
- The `/iris/classify/probabilities` returns the probabilities the input has for each given class to appear

```
@RestController
public class IrisController {

    @Autowired
    IrisClassifierService irisClassifierService;

    @GetMapping(value = "/iris/classify/class")
    public IrisType classify(Iris iris) {
        return irisClassifierService.classify(iris);
    }

    @GetMapping(value = "/iris/classify/probabilities")
    public Map<IrisType, Float> classificationProbabilities(Iris iris) {
        return irisClassifierService.classificationProbabilities(iris);
    }

}
```

(https://www.mimacom.com/en/about/#offices-anchor)

> On "Examples of Usage" section we will show examples of CURL get requests.

Loading the Tensorflow model

As you might have noticed, we have a service where the classification logic is encapsulated.

```
public interface IrisClassifierService {

    /**
     * Method to fetch a classification from the model
     * @param iris the data to classify
     * @return the predicted type
     */
    IrisType classify(Iris iris);

    /**
     * Method to fetch from the model the probabilities of all the types
     * @param iris the data to classify
     * @return A map relating the type with its predicted probabilities
     */
    Map<IrisType, Float> classificationProbabilities(Iris iris);
}
```

In this implementation of the service, we will use the Tensorflow framework to load the previously trained model and feed the inputs to fetch the outputs. Here we use the SavedModelBundle.load() method to load the model, and create a session out of it. Such session will be used later to interact with the model.

```
public class IrisTensorflowClassifierService implements IrisClassifierService {

    private final Session modelBundleSession;
    private final IrisType[] irisTypes;

    //...

    @Autowired
    public IrisTensorflowClassifierService(@Value("${irisml.savedModel.path}") String savedModelPath,
                                           @Value("${irisml.savedModel.tags}") String savedModelTags) {
        this.modelBundleSession = SavedModelBundle.load(savedModelPath, savedModelTags).session();
        this.irisTypes = IrisType.values();
    }

//...

}
```

Feeding the input to and fetching the prediction from the Tensorflow model

Before we can feed the input to the model, we need to build it: The Tensorflow framework uses Tensors (https://www.tensorflow.org/programmers_guide/tensors) to do this. Here we see how it is built.

```
 public class IrisTensorflowClassifierService implements IrisClassifierService {
    //...
    private static Tensor createInputTensor(Iris iris){
        // order of the data on the input: PetalLength, PetalWidth, SepalLength, SepalWidth
        // (taken from the saved_model, node dnn/input_from_feature_columns/input_layer/concat)
        float[] input = {iris.getPetalLength(), iris.getPetalWidth(), iris.getSepalLength(), iris.getSepalWidth()};
        float[][] data = new float[1][4];
        data[0] = input;
        return Tensor.create(data);
    }
    //...
}
```

> Notice the importance of the order of the parameters on the array. This order was obtained from the node `dnn/input_from_feature_columns/input_layer/concat` on the `saved_model.pbtxt` file. In that node, we can see how the name of the parameters match with those described on the `serving_input_receiver_fn` on the export of the model at the (Python) training section, and in this case, the order happens to be alphabetical.

Once we have a standard way to build the input for the model, we proceed to feed them and fetch a prediction. The different kinds of predictions are returned when we *query* for an operation. In this case, we have two fetch operations. We also need to define the input.

```
public class IrisTensorflowClassifierService implements IrisClassifierService {
   //...
    private final static String FEED_OPERATION = "dnn/input_from_feature_columns/input_layer/concat";
    private final static String FETCH_OPERATION_PROBABILITIES = "dnn/head/predictions/probabilities";
    private final static String FETCH_OPERATION_CLASS_ID = "dnn/head/predictions/class_ids";
   //...
}
```


> The exact names of the fetch and feed operations were found on the saved_model.pbtxt file

Now, we can fetch the predicted class for the given input:

```
@Service
public class IrisTensorflowClassifierService implements IrisClassifierService {
    // ...
    @Override
    public IrisType classify(Iris iris) {
        int category = this.fetchClassFromModel(iris);
        return this.irisTypes[category];
    }

     private int fetchClassFromModel(Iris iris){
        Tensor inputTensor = IrisTensorflowClassifierService.createInputTensor(iris);

        Tensor result = this.modelBundleSession.runner()
               .feed(IrisTensorflowClassifierService.FEED_OPERATION, inputTensor)
               .fetch(IrisTensorflowClassifierService.FETCH_OPERATION_CLASS_ID)
               .run().get(0);

        long[] buffer = new long[1];
        result.copyTo(buffer);
        return (int)buffer[0];
     }
    // ...
}
```

And below we see how we fetch the predicted probabilities for each possible class, and build a map to return.

```
@Service
public class IrisTensorflowClassifierService implements IrisClassifierService {
// ...

    @Override
    public Map<IrisType, Float> classificationProbabilities(Iris iris){
        Map<IrisType, Float> results = new HashMap<>(irisTypes.length);
        float[][] vector = this.fetchProbabilitiesFromModel(iris);
        int resultsCount = vector[0].length;
        for (int i=0; i < resultsCount; i++){
            results.put(irisTypes[i],vector[0][i]);
        }
        return results;
    }

    private float[][] fetchProbabilitiesFromModel(Iris iris) {
        Tensor inputTensor = IrisTensorflowClassifierService.createInputTensor(iris);

        Tensor result = this.modelBundleSession.runner()
                .feed(IrisTensorflowClassifierService.FEED_OPERATION, inputTensor)
                .fetch(IrisTensorflowClassifierService.FETCH_OPERATION_PROBABILITIES)
                .run().get(0);

        float[][] buffer = new float[1][3];
        result.copyTo(buffer);
        return buffer;
    }

// ...
}
```

> Notice how the buffer is a matrix, whose second dimension matches the dimension of the expected output.

Performing integration test

Having done the required services, we can do some integration testing. First, the easiest: The `/iris/classify/class` endpoint to get the class given the Iris features. For both endpoints, we feed the same numbers we used on the training (Python) manual testing section, and we expect the endpoint to return the same classes contained in a response with status OK.

```
public class IrisControllerTest extends BaseControllerTest {

    @Test
    public void classify() throws Exception {

        String urlTemplate = "/iris/classify/class?petalLength=%.1f&petalWidth=%.1f&sepalLength=%.1f&sepalWidth=%.1f";

        // Locale.US to make sure the numbers are with period instead of comma.
        String urlRequest = String.format(Locale.US, urlTemplate,1.3f, 0.3f,5.0f, 3.5f);
        MvcResult mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertEquals(IrisType.SETOSA.toString(), mvcResult.getResponse().getContentAsString().replace("\"",""));

        urlRequest = String.format(Locale.US, urlTemplate,4.4f, 1.4f, 6.7f, 3.1f);
        mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertEquals(IrisType.VERSICOLOUR.toString(), mvcResult.getResponse().getContentAsString().replace("\"",""));

        urlRequest = String.format(Locale.US, urlTemplate,6.1f, 1.9f,7.4f, 2.8f);
        mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertEquals(IrisType.VIRGINICA.toString(), mvcResult.getResponse().getContentAsString().replace("\"",""));

    }
    // ...
}
```

(https://www.mimacom.com/en/about/#offices-anchor)

Then, we test the `/iris/classify/probabilities` endpoint, which retrieves the a map of the probabilities for each class. Since the exact number can be slightly different depending on the instance of the model, we will assert the following:

- The class with the highest probabilities is the one we expect.
- The amount of entries is the same as the amount of possible classes.
- All the possible classes have an entry
- All entries have a class and a probability.

Here we see the assertions:

```
public class IrisControllerTest extends BaseControllerTest {
// ...
    private void assertProbabilitiesResponse(MockHttpServletResponse mockHttpServletResponse, IrisType expectedType) throws Unsu
        // Extract the probabilities response
        Gson gson = new Gson();
        LinkedTreeMap<String, Float> probabilities;
        probabilities = (LinkedTreeMap<String, Float>) gson.fromJson(mockHttpServletResponse.getContentAsString(), Map.class);
        // Assert
        assertEquals(expectedType.toString(), getPredictedType(probabilities));
        assertProbabilities(probabilities);
    }

    private String getPredictedType(LinkedTreeMap<String, Float> probabilities) {
        // The predicted type is the one with the highest probabilities
        String predictedType = probabilities.entrySet().stream().max(Map.Entry.comparingByValue()).get().getKey();
        return predictedType;
    }

    private void assertProbabilities(LinkedTreeMap<String, Float> probabilities) {
        // The same amount of entries in the map as the possible values
        assertEquals(probabilities.size(), IrisType.values().length);

        // All the types have a probability value
        for(IrisType irisType: IrisType.values()){
            assertTrue(probabilities.containsKey(irisType.toString()));
        }

        // All the entries have a value
        probabilities.entrySet().stream().forEach(probabilityEntry -> {
            assertTrue(probabilityEntry.getKey() != null);
            assertTrue(probabilityEntry.getValue() != null);
        });
    }
// ...
}
```

And here we perform the calls and evaluate the results:

```
public class IrisControllerTest extends BaseControllerTest {
// ...
 @Test
    public void classificationProbabilities() throws Exception {

        String urlTemplate = "/iris/classify/probabilities?petalLength=%.1f&petalWidth=%.1f&sepalLength=%.1f&sepalWidth=%.1f";

        // Locale.US to make sure the numbers are with period instead of comma.
        String urlRequest = String.format(Locale.US, urlTemplate, 1.3f, 0.3f, 5.0f, 3.5f);
        MvcResult mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertProbabilitiesResponse(mvcResult.getResponse(), IrisType.SETOSA);

        urlRequest = String.format(Locale.US, urlTemplate, 4.4f, 1.4f, 6.7f, 3.1f);
        mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertProbabilitiesResponse(mvcResult.getResponse(), IrisType.VERSICOLOUR);

        urlRequest = String.format(Locale.US, urlTemplate, 6.1f, 1.9f, 7.4f, 2.8f);
        mvcResult = this.mockMvc.perform(get(urlRequest))
                .andExpect(status().isOk()).andReturn();

        assertProbabilitiesResponse(mvcResult.getResponse(), IrisType.VIRGINICA);
    }
// ...
}
```

Packaging the application, executing it, and curl examples with output:

First, create the jar file with maven (on the /serving folder), and then, execute the jar:

```
mvn clean package
java -jar ./target/tensorflowdemo-0.0.1-SNAPSHOT.jar
```

The default port is 7373, but is configurable using the application.yml. Again, we are using the same manual testing values:

Example for `/iris/classify/class` endpoint, where we expect `Versicolour`:

```
curl -GET "localhost:7373/iris/classify/class?petalLength=4.4&petalWidth=1.4&sepalLength=6.7&sepalWidth=3.1"
```

Output:

```
"VERSICOLOUR"
```

Example for `/iris/classify/probabilities` endpoint, where we expect `Setosa` to have the highest probabilities:

```
curl -GET "localhost:7373/iris/classify/probabilities?petalLength=1.3&petalWidth=0.3&sepalLength=5.0&sepalWidth=3.5"
```

Output:

```
{"SETOSA":0.999987,"VIRGINICA":3.4298865E-15,"VERSICOLOUR":1.2982294E-5}
```

Conclusions

Using Tensorflow, Java, and Python, we have demonstrated with a simple project how to perform the basic steps to train, evaluate, and export a model, so it can be later used by another application, in this case a Java Spring Boot application, exposing a REST endpoint to fetch predictions.

The full code can be found at https://github.com/ellerenad/Getting-started-Tensorflow-Java-Spring (https://github.com/ellerenad/Getting-started-Tensorflow-Java-Spring)

Thanks for reading!

About the author: Enrique Llerena Domínguez

Passionate in code and in life. Likes football (both american and the real one ;)). Goal oriented. Keep movin', keep movin'!

COMMENTS

ALSO ON **MIMACOM**

(https://www.mimacom.com/en/about/#offices-anchor)

Autocomplete with Elasticsearch - Part …

a year ago • 3 comments

You know searches from Google or DuckDuckGo. As you start typing the …

Building … API … Liferay DXP with …

a year ago • 2 comments

With the arrival of Liferay DXP and JAX-RS services, we discovered a whole …

Code Folding for JSON Data

3 months ago • 1 comment

In Pair Programming sessions, we often exchange tips and tricks. …

…ing Certific… With BOSH

a year ago • 1 comme…

In any on-premise deployment there a… usually a bunch of

3 Comments **mimacom** **Disqus' Privacy Policy** 1 **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Alejandro Piedra Abarca • 10 months ago

JOIN US

Are you creative and passionate about software development? Do you think unconventionally and act with initiative? Do you want to achieve great things within our team?

Check out Our Job Offers (https://www.mimacom.com/en/jobs/)