**Chapter 5: Understanding Classifiers**

**Title Page**

Hi and welcome to the Zero to Cognitive Series. This tutorial is designed to introduce you to building cognitive applications using the IBM Bluemix Cloud. We will use the IBM Cognitive Solution Advisor as our foundation for this tutorial. This is Chapter 4 where we figure out how Watson figures out what we’re saying. Welcome to Natural Language Classifiers.

**Page 2**

Some parts of working with classifiers will look very similar to things you’ve already done. Natural Language Classifiers are another Watson service, so we’ll go through the process of adding that service and capturing the credentials for the service.

We’ll create a classifier file and, from that, create an object that Watson will use to classify things the way WE want them classified. Then we’ll use that classifier in our application so that we can see the results of our work.

But first …

**Page 3**

What is a classifier? A classifier is a word or phrase which logically links to something else. For example, A&D, A&D Business, aerospace prime contractors, air travel, aircraft industry, asset availability all relate to the Aerospace and Defense industry. Of course, air travel also relates to the Travel and Transportation industry while asset availability relates to many other industries. Because some words and phrases relate to multiple industries in this example, depending on a single word or phrase to identify an industry is ineffective. We, you and I, provide a file with classifiers, including those which are used in multiple industries. Watson uses that information to then figure out what a given sentence or paragraph is most likely linked to.

Watson NLC is designed to work with short, less than an thousand characters, paragraphs. The description at the top of this page is 797 characters long, to give you a visual reference on how long is 1000 characters.

**Page 4**

We’re going to add two visual elements to our application today.

* A button to invoke the Watson NLC service
* A pop-up page to display the results. This will introduce you to a new visual element in your web page, tables, and a new way to display temporary information – a pop-up.

**Page 5**

To add a service into your app, we’ll do the same thing we did to enable Speech to Text.

1. Step 1, log in to your bluemix dashboard and go to your app
2. Step 2, select “Add a Service or API”
3. Step 3, select Watson Natural Language Classifier
4. Step 4, note the specific name of your service (Natural Language Classifier-xx)
5. Step 5, display and copy your credentials

**Page 6**

Open your env.json file and copy in the new credentials for Natural Language Classifier

Open your manifest.yml file and add a third service “Natural Language Classifier -xx” where the dash xx is what you wrote down in Step 4 from the previous page

**Page 7**

Classification is logically a different concept from moving back and forth between speech and text. In keeping with our Principles of cohesion and separation of concerns (cohesion = keep things together that logically relate to each other; separation of concerns = keep things separate that don’t logically relate to each other) we will create a new service in our router to support classification and we will put the code to support classification into a new file.

1. Step 1 open the router.js file and note the following:
   1. Line 5 is new and defines a variable called classifier which loads a file called classifier.js from the features folder
   2. Line 13 is new and specifies that when a browser issues a POST command directed to anything that starts with ‘/api/understand/classifyInd\*' to invoke the classifyInd function from within the classifier.js file.
2. Step 2, open the classifier.js file in the Chapter 5 features folder
   1. Line 2 adds the cfenv module, which we’ll later use with the NLC credentials
   2. Line 4 creates a static variable with the name of the service we’re using
   3. Line 5 is familiar, it’s the file where we’re placing all of the Watson credentials
   4. Line 7 uses cfenv to get the application environment so that in
   5. Line 9 we can get the service credentials for Watson NLC
   6. Line 10 creates an object with all of the nlc information bundled together
   7. Line 11 specifies which classifier we’re going to use (we haven’t created it yet)
   8. Lines 12-16 export the function referred to in line 13 from the router.js file and is empty.
   9. Lines 18-20 define another empty function call nlc\_res, short for nlc results. We’ll use this to format the results from the nlc call into json for use by the browser.

**Page 8**

1. Step 3, open the classifier-complete.js file from the answers folder
   1. Line 14 gets the text sent from the browser. Although we’re not doing it here, it would be a really good idea to check to make sure this field is not empty as that causes an error when you connect with NLC
   2. Line 15 defines a javascript object
   3. Lines 16-25 define a javascript anonymous function which
   4. Line 17 calls the nlc service with the provided text and classifier. When the nlc function returns,
   5. Line 21 a new function is called which
   6. Line 22 initiates a send back to the browser and
   7. Line 23 invokes the formatting function to handle getting the data back to the browser
   8. Lines 28-39 define the json formatting function
   9. Line 29 defines an array
   10. Line 30 checks to see if the nlc function returned successfully
   11. Lines 31-32 record the error and populate the array with the error information
   12. Lines 33-35 otherwise populate the array with the nlc results and
   13. Lines 36-37 print out the results of the nlc call and
   14. Line 38 send it back to the browser
2. Step 4, copy these functions from the ‘complete’ file into the classifier.js file in the features folder.

**Page 9**

But wait, we don’t have a classifier. So let’s go build one. Well, let’s take a classifier file that already exists and create a Watson classifier from it. We’ll use a file I created for another project to figure out what industry someone is talking about. The file we will use is in the Documentation folder and is called “industry.csv”

1. Step 1: open the industry.csv file in your favorite editor
   1. What you’ll see in this file is a long list of items, each of which has a phrase, followed by a comma, followed by a classifier. As you read through the list, you’ll quickly identify that many of the terms associated with the first industry, Aerospace and Defense, are also associated with other industries. You change how a classifier works by changing the content of this file.
2. Step 2: open a terminal window
3. Step 3: type the command shown on this page to create the classifier. Username and password are the NLC credential user name and password.
4. Step 4: record the classifier id returned from this command
   1. Copy that into line 11 of the classifier.js file in the features folder
5. Step 5, using the username, password and classifier id from the previous step, execute the curl command shown to determine the status of your classifier. Your application will not work until this command returns a ‘classification complete, classifier ready to use’ response.

It’s not necessary to wait for the classification to complete, however, before going on with the tutorial, so we’ll finish up our coding and come back and check on the status of the classifier again a little later on.

**Page 10**

We only have three things left to do: activate the classify button in the browser, get the text from the browser and send it to the classifier and, finally, display the results. Let’s get to it.

**Page 11**

Just as we used the principle of separation of concerns to tell us to create some new javascript files, we will do the same thing with our html. Interacting with the core Watson services will still be done in our main file, but we also want to display the results from the classification in a table-based pop up page. It’s much simpler to keep these pages separate.

1. Step 1: open the index.html page
   1. Line 32: You will see that a new line has been added which creates an empty ‘div’ with an id of ‘modal’. We will use this to create our pop-up.
   2. Lines 49-51: These three lines create a new row which holds a button with an id of ‘classifySpeech’ and a label of ‘Classify Speech.’
   3. Line 75: a new javascript file has been added called z2c-NLC.js, which is where all the code will reside to support the nlc functions.

**Page 12**

Because we’ve added a new button to the home page, we need to activate it just as we have activated the other parts of the page.

1. Step 1: open the z2c-speech.js file
   1. Line 8 adds a new variable, called displayNLC which links to the HTML element ‘classifySpeech’
   2. Lines 71-75 activate the classifySpeech button
   3. Line 73 identifies the html file with the structure of the pop-up page and
   4. Line 74 calls the new function ‘checkNLC’, passing it the name of the html file and stt-out, which you may recall from the 3rd tutorial as the place where the speech to text stored it’s information. This means that whatever you say into your microphone is what will be analyzed.

**Page 13**

Let’s build the functions we need to read and write NLC information

1. Step 1: open the z2c-NLC.js file where you’ll see two empty functions:
   1. checkNLC, identified in the initPage function and
   2. displayNLC, which will be used to activate the pop-up window and populate it with the NLC results.
2. Step 2: Open the displayNLC.html file
   1. The structure of this file is very similar to what you’ve seen in the index.html file.
   2. Line 4 defines the header for the page
   3. Line 5 creates a row
   4. Line 6 defines the width of the row (from IBM design language)
   5. Line 7 starts the definition of an html table with an id of ‘industryResult”
   6. Line 8 identifies the headings for the table columns
   7. Line 9 ends the table
   8. Lines 13-15 create a new row with a single button with an id of ‘close\_NLC’ which will be used to close the window when you’re done looking at this information.
3. Step 3: open the pageStyles\_complete.css file
   1. Lines 3-13 describe how to create the black background for the pop-up window
   2. Lines 15-23 (a div within ‘modalDialog’) define how to display the contents of the pop-up window.

**Page 14**

1. Step 4: open the z2c-NLC-complete.js file from the answers folder
   1. Lines 2-22 define the ‘checkNLC’ function
      1. Line 4 creates a javascript object, this will be used to pass information in the post command
      2. Line 5 creates a keyword in the options object called cquery and gives it the text from the \_source div provided during the calling process
      3. Line 6 is really very simple. It’s using the services of jQuery and can be read in English from left to right:
         1. When (you’re done) getting \_display and posting to the classifyInd api execute this function starting at the end of this line and give to that function the results of the get and post calls.
         2. Get(\_display) gets the displayNLC.html file
         3. Post(etc) sends the text to nlc and returns a ‘stringified’ json object.
      4. Line 8 creates an object linked to the modal div in the index.html file
      5. Line 9 appends the display-NLC.html file to that div
      6. Lines 10-11 resize that div and display it. This makes it visible and makes the html elements inside displayNLC.html accessible to javascript
      7. Line 12 gets the first part of the returned array and stores it in \_data
      8. Line 13 extracts the classes objects from the returned information. Json.parse is called twice because, on the sending side, json.stringify was called twice. I’ll leave this as an exercise to you to simplify should you so desire.
      9. Line 14 calls the display function, passing in the html tag to use for displaying and the data to display.
      10. Lines 15-19 activate the close button in this pop-up window.

**Page 15**

* 1. Lines 23-39 create a function to display the NLC results
     1. Line 25 creates a variable from \_target
     2. Line 26 empties the table of content, if any were present
     3. Line 28 retrieves the number of classifiers returned
     4. Line 29 creates and sets a counter variable to 0
     5. Lines 30-37 execute a loop which terminates when all classifiers have been processed.
     6. Lines 32-35 create an anonymous function inside the loop. This enables each element in the loop to be processed correctly. While it might seem simpler to just use the code on line 14, it actually won’t return the correct results in javascript. The current counter (\_idx) and the classifier array (\_results) are passed in as variables to this anonymous function in line 35 and renamed for use inside the function in line 32
     7. Lines 33-34 format one row of the table
     8. Line 36 increments the counter
     9. Line 38 ends the table definition.

1. Step 5: Copy both of these functions into z2c-NLC.js, replacing the existing and empty functions.
2. Step 6: save the file

**Page 16**

1. Step 1: If you’ve not already done so, Save all of your files.
2. Step 2: Open a command prompt and navigate to Chapter05
   1. Execute the command npm install
   2. Execute the command node index.js and note the port number displayed by the newly started application. On my system, you’ll see it displayed as port 6008
   3. Open your browser and direct it to the correct URL, for example: <http://localhost:6008>
   4. Test the application, talk to it, give it some text to use, tell it to classify what you said.
3. Step 3: send your information to github
   1. Git status (tells you what files you’ve changed)
   2. Git add …. For each changed file
   3. Git commit –m ‘your commit message goes here’
   4. Git push … sends your data back to your git repository
4. Step 4: send your code to cloud foundry
   1. Cf login (uses your ibm id and password)
   2. Cf push Z2C (this will replace your existing Z2C app with the one from Chapter04)
   3. Log in to Bluemix and navigate to your Z2C application
   4. Click on the highlighted URL, this will open a new browser window or tab. Test the application.
5. Step 5: Woohoo!! You’ve successfully built your third Bluemix application.

In our next session, we will add in cookies and session management and use the results of your hard work so far to create a custom dialog with Watson. Why cookies and session management? Because there’s nothing in place right now which tells Watson how to remember what you’ve done in the past. We will do some very simple things in our next tutorial. See you soon!