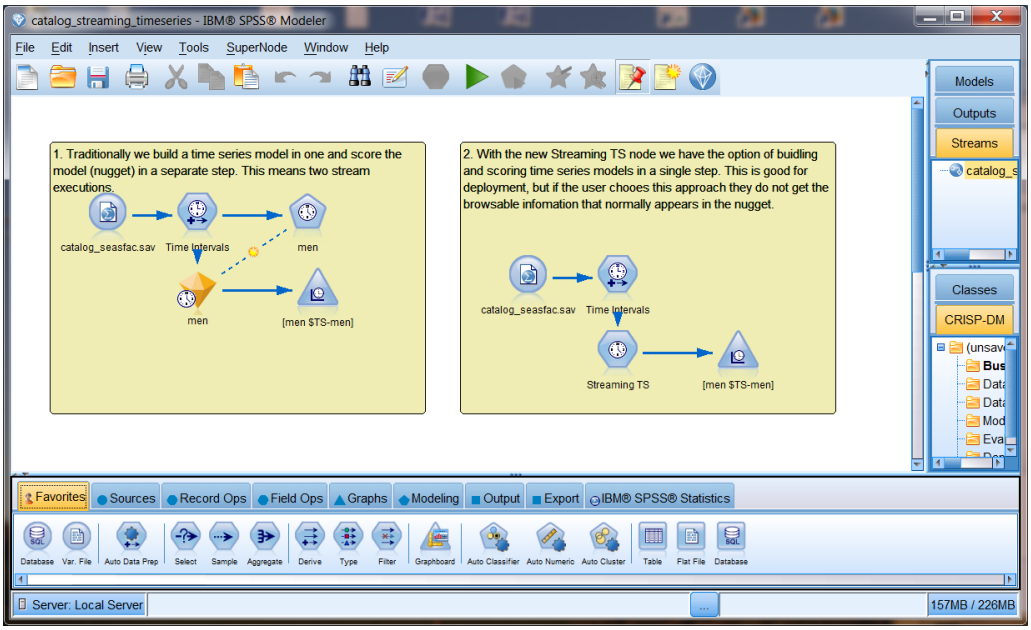
IBM Predictive Analytics Service for Bluemix
Example Application #2

Preparation:

In this example, we'll use two versions of the *catalog_streaming_timeseries.str* Modeler stream and training data from the SPSS Modeler Demos set shipped with the SPSS Modeler product.



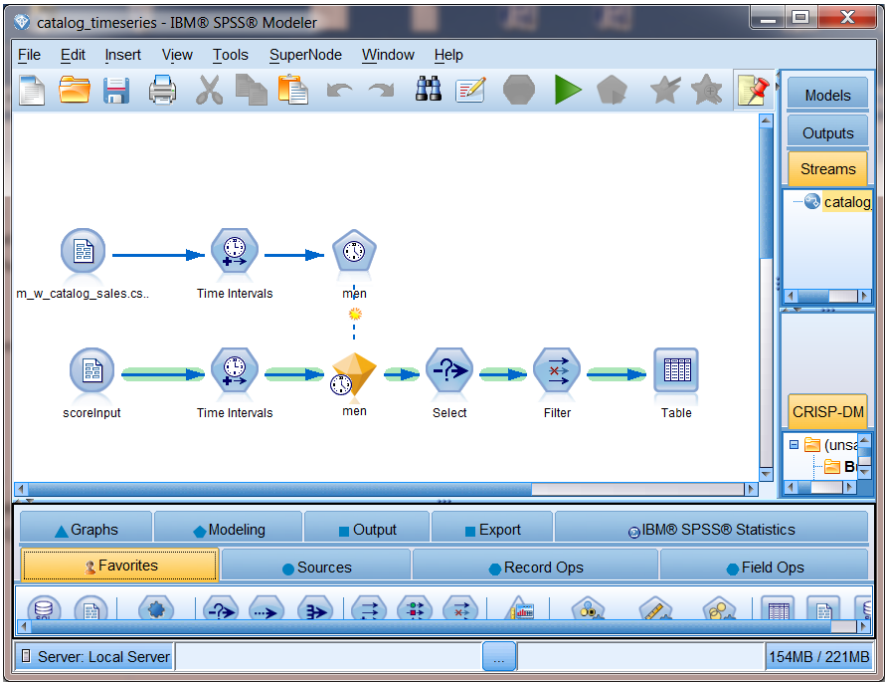
This demo contains two scoring branches of interest. In the previous screenshot, the traditional example on the left trains a Time Series model algorithm and uses the generated model nugget in its scoring branch. The streaming time series example on the right uses a Streaming TS node, which is self-learning in its scoring branch, and there is no model training branch in this design.

This sample uses two SPSS Modeler streams derived from this demo:

- One that uses a trained Time Series model to predict the sales of men's clothing – basically an as-is use of the scoring branch (on the left in the screenshot)
- One that uses the Streaming TS node to predict the sales of women's clothing – a slight modification to focus on women's sales in the scoring branch (on the right in the screenshot)

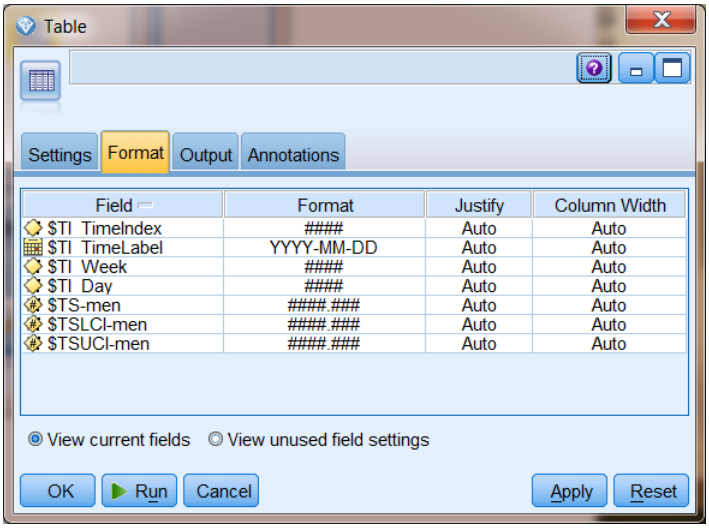
Predicting sales of men's clothing:

In the first version (*catalog_timeseries.str*), we've trained a Time Series model algorithm using the men's catalog sales. The scoring branch (highlighted in green) is the trained Time Series model and the processing involved in generating a forecast of sales of men's items from the catalog based on the scoring input of N sales periods.



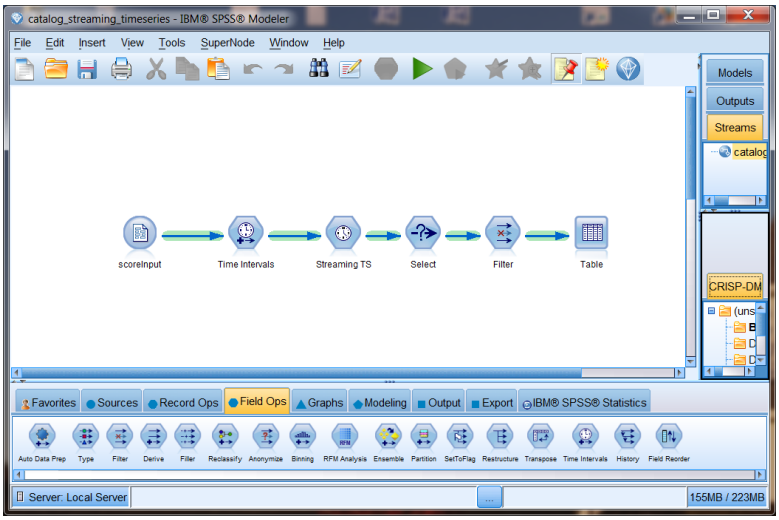
We will keep the inputs simple and require only the total sale of men's items from our catalog for each period as the scoring input. And we'll assume these are of the correct time interval.

We will also filter the results to eliminate input records, limiting the output to only the forecast fields of interest as shown in this output example:



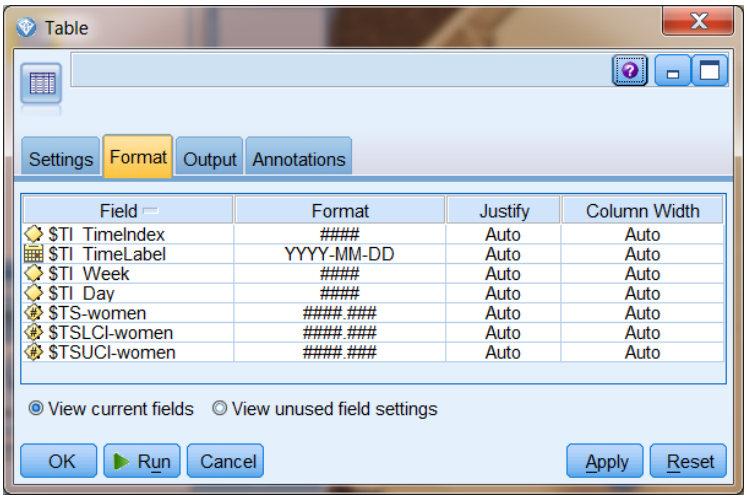
Predicting sales of women's clothing:

In the second version (a modified *catalog_streaming_timeseries.str* file), we'll use the Streaming TS node to "learn" how to forecast sales for women's clothing as we are passed input data. The scoring branch (highlighted in green) is of the same basic design as the first example, but instead of a trained Time Series model we will use the self-learning Streaming TS node to forecast sales of women's items from the catalog based on the scoring input of N sales periods.



We will keep the inputs simple and require only the total sale of women's items from our catalog for each period as the scoring input. And we will assume these are of the correct time interval.

We will also filter the results to eliminate input records, limiting the output to only the forecast fields of interest as shown in this output example:



Preparing to develop the application

See the *IBM Predictive Analytics service for Bluemix - General* document for an introduction to Bluemix application development.

Developing the NodeJS portion of the Single Page Application for this sample

We will be making some REST service calls, passing data in for scoring in our predictive model, and handling the score results, so we'll need some additional packages in our NodeJS application.

Adjusting the NodeJS packages.json and app.js files

First we'll open the *package.json* file and update the name and description as well as add the **request** and **body-parser** packages. We'll remove Jade and use AngularJS in this example.

Original:

```
{
        "name": "NodejsStarterApp",
        "version": "0.0.1",
        "description": "A sample nodejs app for Bluemix",
        "dependencies": {
                "express": "3.4.7",
                "jade": "1.1.4"
        },
        "engines": {
                "node": "0.10.26"
        },
        "repository": {}
}
```

Modified:

```
{
        "name": "SPSS-PM-sample-2",
        "version": "0.1.0",
        "description": "A Node.js app using Express delivering SPSS PM sample application 2",
        "dependencies": {
                "express": "~4.0.0",
                "request": "2.36.x",
        "body-parser": "~1.0.1"
         },
        "engines": {
                "node": "0.10.26"
        },
        "repository": {}
}
```

We've added **http**, **path**, **body-parser**, and **request** to the **express** section in this sample application. If we issue the **npm install** command to ask the Node Package manager to pull down these packages, we can watch these packages be added to the desktop.

Defining the routing entries for our services and Single Page Application

We'll be using some helper services in this NodeJS application, as well as serving up our Single Page Application (SPA), so we need to set up some routing information. Note that we'll serve up the SPA from *public* instead of *views*, so we'll drop the *views* directory and its Hello World content.

```
// ROUTES
// ============================================================= var router =
express.Router();   // get an instance of the express Router

// middleware to use for all requests
router.use(function(req, res, next) {
       next(); // make sure we go to the next routes and don't stop here
});

// TBD services and their routing…

// Register Service routes and SPA route ---------------
var rootPath = '/score';

// all of our service routes will be prefixed with rootPath
app.use(rootPath, router);

// SPA AngularJS application served from the root
app.use(express.static(path.join(__dirname, 'public')));
```

Predictive Analytics service connectivity, the scoring helper service, and NodeJS startup

Let's set things up so we can test and debug our NodeJS application from our desktop.

To do this, we'll either use the port and host we get from Bluemix in the VCAP_APP_PORT and VCAP_APP_HOST environment variables, or the environment variable set on our local system, or some defaults.

```
var port = (process.env.VCAP_APP_PORT || process.env.PORT || 3000);
var host = (process.env.VCAP_APP_HOST || process.env.HOST || 'localhost');
```

We also want to be able to call our provisioned instance of the Predictive Analytics services for Bluemix, so we'll define some defaults for the service instance URL and access_key and initialize a data structure we'll modify using the VCAP_SERVICES information if we are deployed in Bluemix.

```
var defaultBaseURL = 'http://174.36.238.2:8080/pm/v1';
var defaultAccessKey =
'"RN7dXYh3I1SN7bUERez7heVKlT/WlwSj/NeKfDJRae0wt9nA0+UM71/6XsadECDqIhA7VGK7033Xo
CVgABt84wo7Io6/ltsqOs0i7k0j8lNI9jBxf8YqDOGTo+qpTLwzRqXP+5vfE4jhLDBIIf4BdQ=="';

var env = { baseURL: defaultBaseURL, accesskey: defaultAccessKey };
```

Now let's look at the VCAP_SERVICES information Bluemix sets in our environment. The IBM Predictive Analytics service is identified by a 'pm-20' label and, from that, we are interested in the credentials –

where the URL of our service instance and the **access_key** to be used as set in the **bind** event are communicated to us. We'll store this information in a simple data structure for later use.

```
// VCAP_SERVICES contains all the credentials of services bound to
// this application. For details of its content, please refer to
// the document or sample of each service.
var services = JSON.parse(process.env.VCAP_SERVICES || "{}");
var service = (services['pm-20'] || "{}");
var credentials = service.credentials;
if (credentials != null) {
            env.baseURL = credentials.url;
            env.accesskey = credentials.access_key;
}
```

Next we'll define helper services for the thin-client UI to use for making score requests:

```
// score request
router.post('/', function(req, res) {
      var scoreURI = env.baseURL + '/score/' + req.body.context + '?accesskey='
+ env.accesskey;
      try {
            var r = request({ uri: scoreURI, method: "POST", json:
req.body.input });
            req.pipe(r);
            r.pipe(res);
      } catch (e) {
            console.log('Score exception ' + JSON.stringify(e));
    var msg = '';
    if (e instanceof String) {
       msg = e;
    } else if (e instanceof Object) {
      msg = JSON.stringify(e);
    }
    res.status(200);
    return res.send(JSON.stringify({
        flag: false,
        message: msg
       }));
       }

      process.on('uncaughtException', function (err) {
          console.log(err);
      });
});
```

The last step in the NodeJS work is to start things up:

```
// START THE SERVER with a port reminder when run on the desktop
// ============================================================
app.listen(port, host);
console.log('App started on port ' + port);
```

Developing the HTML, CSS, and AngularJS portion of the Single Page Application for this sample

HTML used

To make things simple, our SPA for this example will be defined in *index.html*. We won't discuss the CSS used here. If interested, you can read through this in the sample bundle. We are going to set things up to be able to call either version of our Time Series designs from this one page application.

First we define our application in AngularJS terms and set the main controller:

```
<body ng-app="tsSample" ng-controller="AppCtrl" >
```

Then we define the main form that dominates this simple UI. The main thing to note here are the **ng-model** definitions binding these HTML elements to a data model managed by the controller in the Angular MVC pattern.

We'll modify the title to indicate the predictive model we'll be scoring with:

```
<div class="title">
        <img alt="" src="img/pm24x24.png" />
        IBM Predictive Analytics service <b>"{{modelType[mIdx]}}"</b> scoring applicaiton</div>
```

We'll put a radio button set at the top of the screen to permit selection of the men's or women's forecasts. Note that the radio buttons will be setting the value of the **mIdx** variable:

```
Catalog Sales:
        <input type="radio" ng-model="mIdx" value="0">Men's Clothing</input>
        <input type="radio" ng-model="mIdx" value="1">Women's Clothing</input>
```

Then we'll permit the input of however many sales periods are represented by our **sales** data object:

```
<div ng-repeat="sval in sales track by $index">
        Sales Input: {{$index +1}}  <input type="numeric" required value="{{sval}}"></input><br /><br />
</div>
```

Finally, we'll include a button to kick off the scoring request:

```
<button type="button" class="btn.lg" ng-click="score()">
        <i class="glyphicon glyphicon-cloud-upload"></i> Score Now </button>
```

Certain countries block Google, in which case you'd have to deploy AngularJS yourself. In our example here, we just pull it in dynamically via reference and then we pull in the controllers and services we define for this application:

```
    <!-- load angular via CDN -->
    <script
src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.10/angular.js"></script>
```

```
	<script src="//angular-ui.github.io/bootstrap/ui-bootstrap-tpls-
0.11.0.js"></script>

	<!-- our scripts -->
	<script src="js/app.js"  type="text/javascript" ></script>
	<script src="js/srv.js"  type="text/javascript" ></script>
```

The controller

Our controller module will use some dialog and data services from another module, so we'll note these dependencies now:

```
Var AppCtrl = ['$scope', 'dialogServices', 'dataServices',
function AppCtrl($scope, dialogServices, dataServices) {
```

As noted earlier, the radio buttons in our SPA will modify the **mIdx** variable in our controller's scope:

```
	$scope.mIdx = 0;
```

This variable value will select the **context ID** of our model to be used in scoring – where we deploy the trained TimeSeries model algorithm version as **catalogTS** and the Streaming TS node version as **catalogSTS**.

```
	$scope.context = ['catalogTS', 'catalogSTS'];
```

This variable value will be used to select the input field name of **men** for scoring our **catalogTS** model and **women** for our **catalogSTS** model, as well as our indicator of which model is being used in the title of our UI.

```
	$scope.fld = ['men', 'women'];

	$scope.modelType = ['Time Series', 'Streaming TS'];
```

We initialize the data model used by the UI with some reasonable values. If desired, you can use more than 4 periods of sales input.

```
	// init UI data model
	$scope.sales = [10000, 20000, 30000, 40000];
```

There is one button on this UI telling us to score with the data set on our form. This request will be performed in an asynchronous fashion, and we'll react to the results when they come in by displaying the information returned in a model dialog OR by showing whatever error messages are returned in an error dialog. Note that we pass along the context_id, input field name, as well as the values you have entered for the four sales periods in the UI.

```
	$scope.score = function()	{
		dataServices.getScore(
			$scope.context[$scope.mIdx],
			$scope.fld[$scope.mIdx],
			$scope.sales)
		.then(
			function(rtn) {
```

```
                        if (rtn.status == 200){
                                // success
                                $scope.showResults(rtn.data);
                        } else {
                                //failure
                                $scope.showError(rtn.data.message);
                        }
                },
                function(reason) {
                        $scope.showError(reason);
                }
        );
}

$scope.showResults = function(rspHeader, rspData) {
        dialogServices.resultsDlg(rspHeader, rspData).result.then(); }

$scope.showError = function(msgText) {
        dialogServices.errorDlg("Error", msgText).result.then(); }
}]
```

The dialog and service call helpers

The data services are our **getScore** helper that builds the input data structure that will be passed. This data model is defined by the scoring branch of the SPSS Modeler file we looked at earlier. Note that the **tabular input data source name** and all other names must match the scoring branch as defined in the deployed predictive model.

```
sampleSrv.factory("dataServices", ['$http',
function($http)        {
        this.getScore  = function(context, fld, sales) {
                /* create the scoring input object */
                var s = [];
                for (i = 0; i < sales.length; i++)
                        s[i] = [ sales[i] ];

                var input = {
                        tablename: 'scoreInput',
                        header: [ fld ],
                        data: s
                };

                /* call scoring service to generate results */
                return $http({ method: "post",
                                  url: "score",
                          data: { context: context, input: input }
                             })
                        .success(function(data, status, headers, config) {
                                return data;
                        })
                        .error(function(data, status, headers, config) {
                                return status;
                        });
        }
        return this;
}]);
```

The dialog services were already discussed, but we should look at the data model bindings and the partial HTML template used here to illustrate how we handle single row and multiple row returns. The basic work done here is to take the **score results** object we get back from a successful score request and make its contents easy for the HTML template to process. The two **resolve** items make the field names available by an **rspHeader** reference and the **0..N** result rows available by an **rspData** reference.

```
sampleSrv.factory("dialogServices", ['$modal',
function($modal) {

	this.resultsDlg = function (r) {
		return $modal.open({
			templateUrl: 'partials/scoreResults.html',
			controller:  'ResultsCtrl',
			size:  'lg',
			resolve: {
				rspHeader: function () {
					return r[0].header;
				},
				rspData: function   () {
					return r[0].data;
				}
			}
		});
	}

	return this;
}]);
```

The HTML template used for score results is interesting. We use **ng-repeat** on the number of fields in the **rspHeader** to add column headers to the table.

```
<thead>
  <tr>
    <th ng-repeat="fname in rspHeader">{{fname}}</th>
  </tr>
</thead>
```

Then we use **ng-repeat** on the rows we got back and a lower level **ng-repeat** on the number of columns in each row to display all values for all rows in the response.

```
<tbody>
  <tr ng-repeat="row in rspData">
    <td ng-repeat="fval in row track by $index">{{fval}}</td>
  </tr>
</tbody>
```

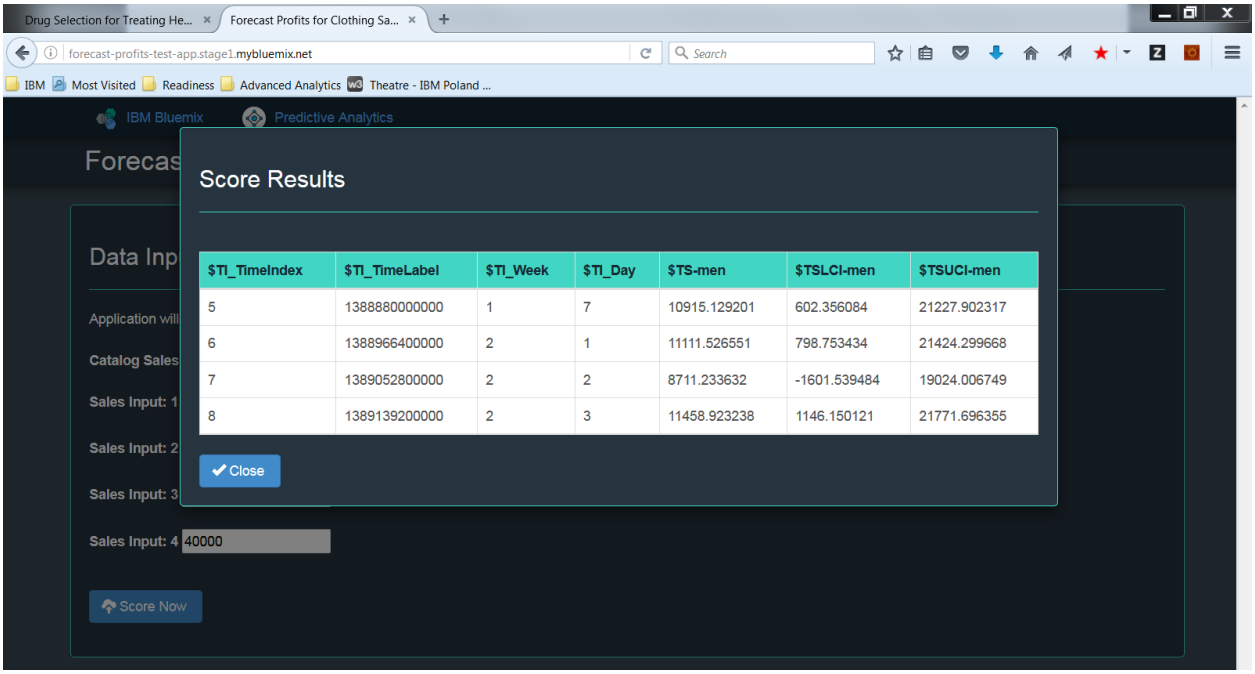
That covers the AngularJS UI implementation for our Single Page Application.

Testing the Single Page Application for this sample

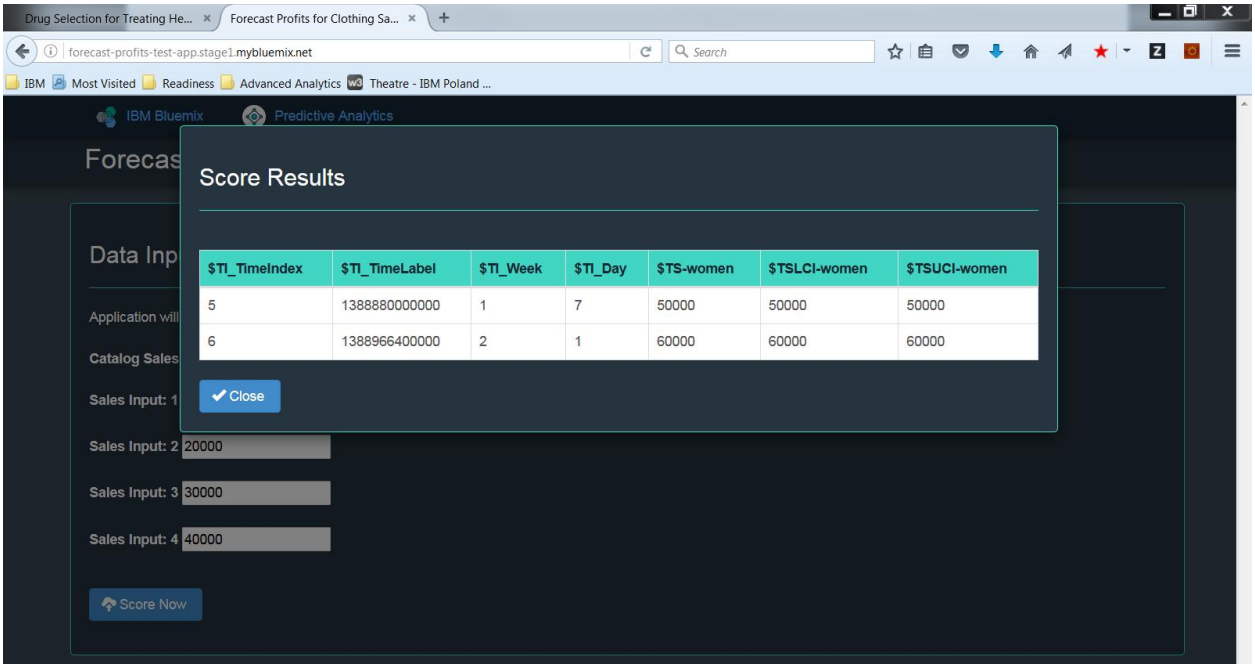
See the *IBM Predictive Analytics service for Bluemix - General* document for an introduction to Bluemix application testing.

If running from your desktop, start your NodeJS example with a **node app.js** command, open a browser to **localhost:3000**, and score with our provisioned Predictive Analytics service on Bluemix. This should look the same and function the same as your application running on Bluemix.

For men's clothing sales forecasts:



For women's sales forecasts:



Your data analyst may want to feed the Streaming TS node a series of scoring inputs to get experience in the self-learning aspect of this algorithm. Note that the learning curve will start all over again once you stop and re-start your application.