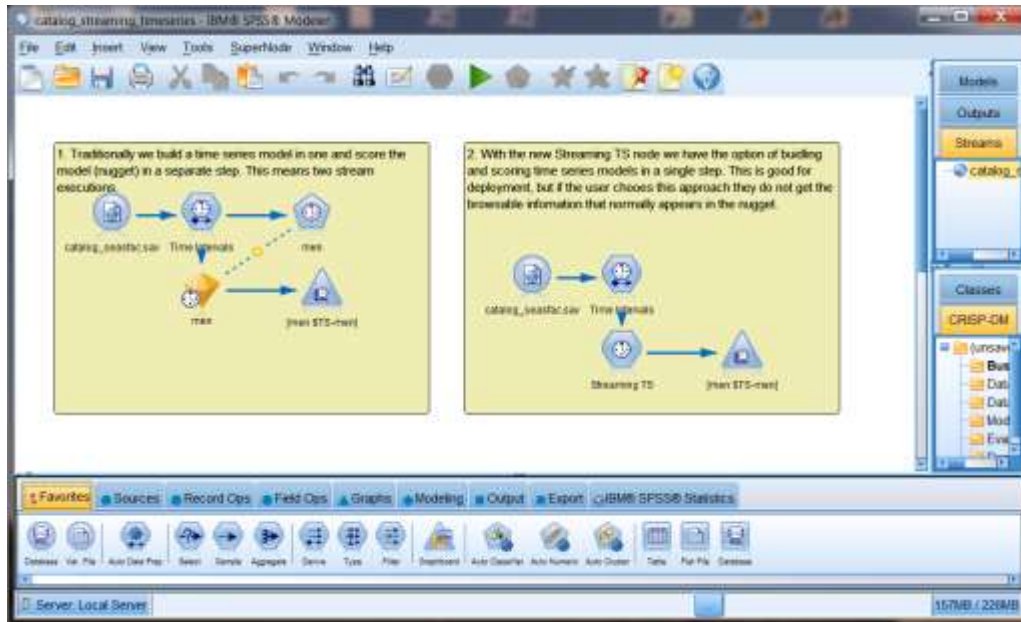


# IBM Predictive Modeling Service for Bluemix

## 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. 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, there is no model training branch in this design.

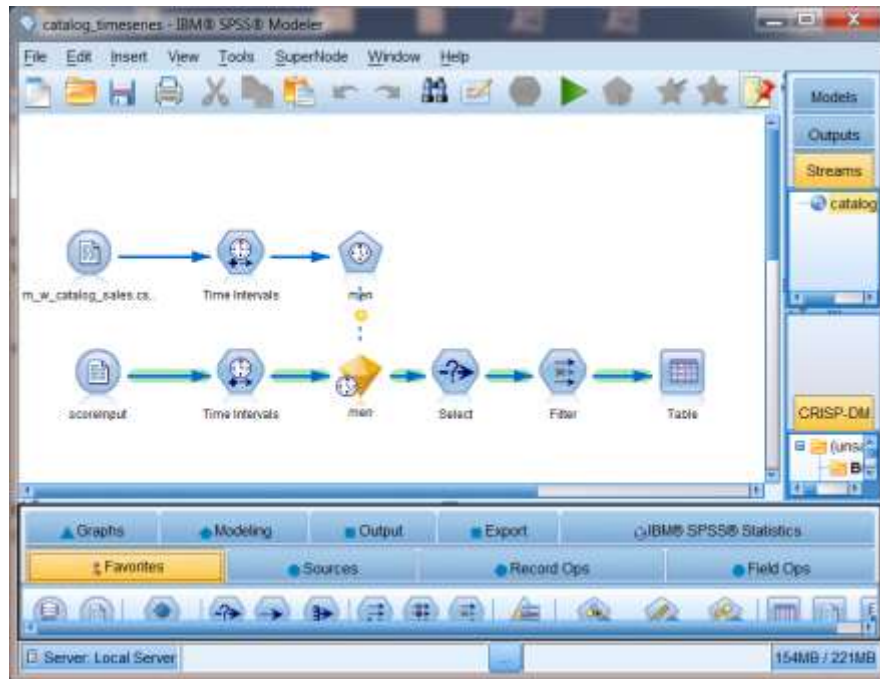
This sample uses two SPSS Modeler streams derived from this demo:

1. one that will use 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 screen snapshot above
2. one that will use 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 screen snapshot above

### *Predicting Sales of Men's Clothing:*

In the first version (catalog\_timeseries.str file) 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 scoring input, 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.

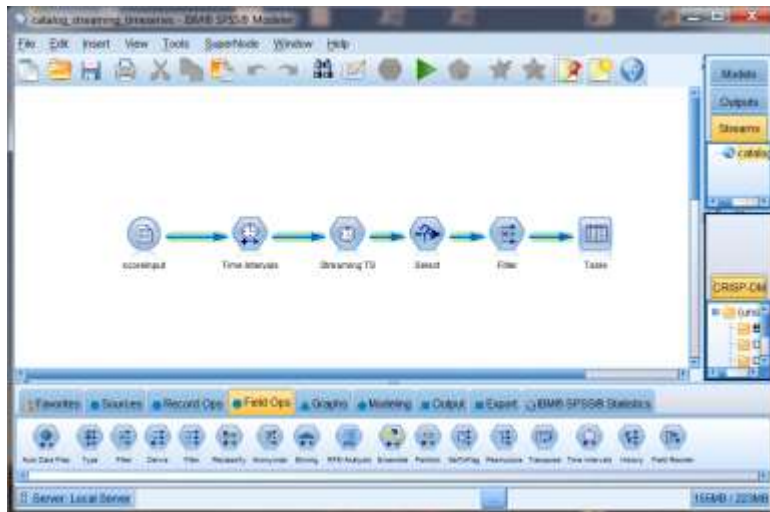
Field	Format	Justify	Column Width
\$T1 TimeIndex	####	Auto	Auto
\$T1 TimeLabel	YYYY-MM-DD	Auto	Auto
\$T1 Week	####	Auto	Auto
\$T1 Day	####	Auto	Auto
\$TS-men	#### ###	Auto	Auto
\$TSLCI-men	#### ###	Auto	Auto
\$TSUCI-men	#### ###	Auto	Auto

☒ View current fields
 ☐ View unused field settings

OK Run Cancel Apply Reset

## ***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 we saw in the first example but instead of a trained Time Series model we will use the self-learning Streaming TS node to forecast of 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 scoring input, 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.

Field	Format	Justify	Column Width
\$Ti TimeIndex	###	Auto	Auto
\$Ti TimeLabel	YYYY-MM-DD	Auto	Auto
\$Ti Week	###	Auto	Auto
\$Ti Day	###	Auto	Auto
\$TS-women	###	Auto	Auto
\$TSLCI-women	###	Auto	Auto
\$TSUCI-women	###	Auto	Auto

## Prepare to Develop the Application

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

## Develop 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.

### Adjust the NodeJS packages.json and app.js files.

First I'll go into the package.json file and update the name and description as well as adding the 'request' and 'body-parser' packages. I've dropped the Jade, we'll 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": {}
}
```

I've added 'http', 'path', 'body-parser' and 'request' to the 'express' I got in this sample application. If I ask the Node Package manager to pull down these packages by issuing the 'npm install' command I can watch these packages added to my desktop.

## Define 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 setup 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')));
```

## PM Service Connectivity, the Scoring Helper Service and NodeJS startup

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

To do this we'll use either 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'll also want to be able to call our provisioned instance of the PM 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 actually deployed in Bluemix.

```
var defaultBaseURL = 'http://174.36.238.2:8080/pm/v1';
var defaultAccessKey =
'"RN7dXYh3I1SN7bUERez7heVK1T/WlwsJ/NeKfDJRae0wt9nA0+UM71/6XsadECDqIhA7VGK7033Xo
CVgABt84wo7Io6/ltsqOs0i7k0j8lNI9jBxf8YqDOGTto+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 Modeling service will be 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 a helper services for the thin-client UI to use to make 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);
});
});
```

Last step in the NodeJS work, 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);
```

# Develop the HTML, CSS and AngularJS portion of the Single Page Application for this Sample

## HTML Used

Our SPA for this example will be defined in index.html to make things simple. We won't go into the CSS used here, you can read through this in the sample bundle if you are interested. 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">
  
  IBM Predictive Modeling service <b>{{modelType[mldx]}}</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 'mldx' variable.

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

Then will 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>&nbsp;Score Now&nbsp;</button>
```

Certain countries have Google blocked and so you'll have to deploy AngularJS yourself. In our example here will 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, it could be more than 4 periods of sales input if you like.

```

// 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 async fashion and we'll react to the results when they come in by popping the information returned up in a model dialog OR showing whatever error messages we get back 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 help see 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 a ‘rspHeader’ reference and the 0..N result rows available by a ‘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 the ng-repeat on the number of fields in the ‘rspHeader’ to put column headers in 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>
```

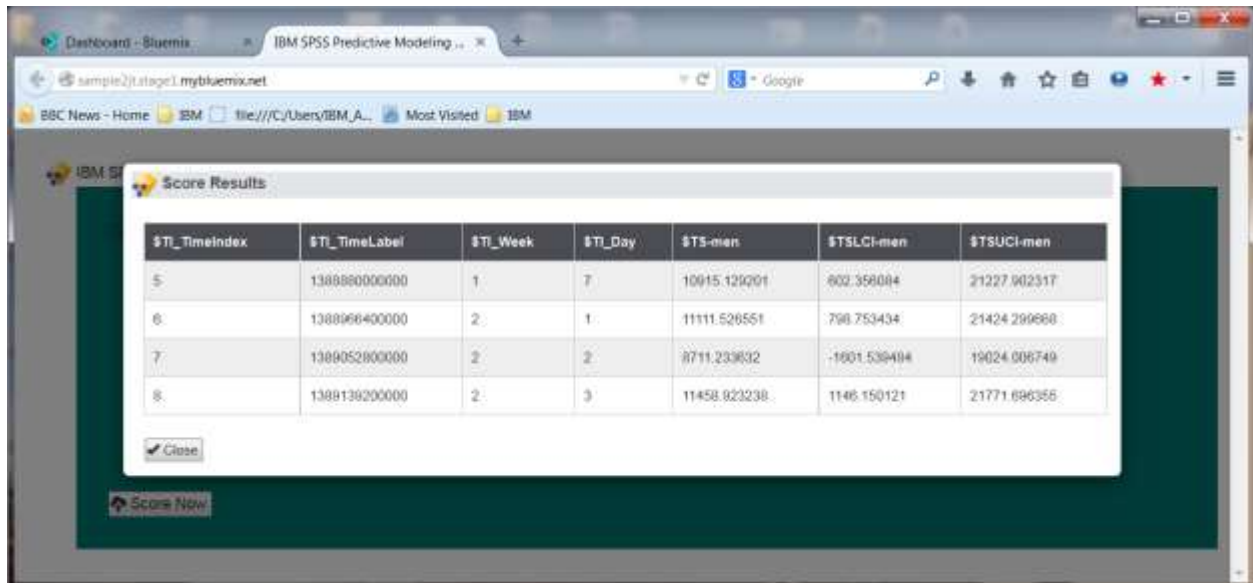
That’s about it for the AngularJS implementing the UI of our Single Page Application.

## Test the Single Page Application for this Sample

See the ‘IBM Predictive Modeling 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 and open the browser to localhost:3000 and score with our provisioned PM service on Bluemix. This should look the same and function the same as your application running on Bluemix.

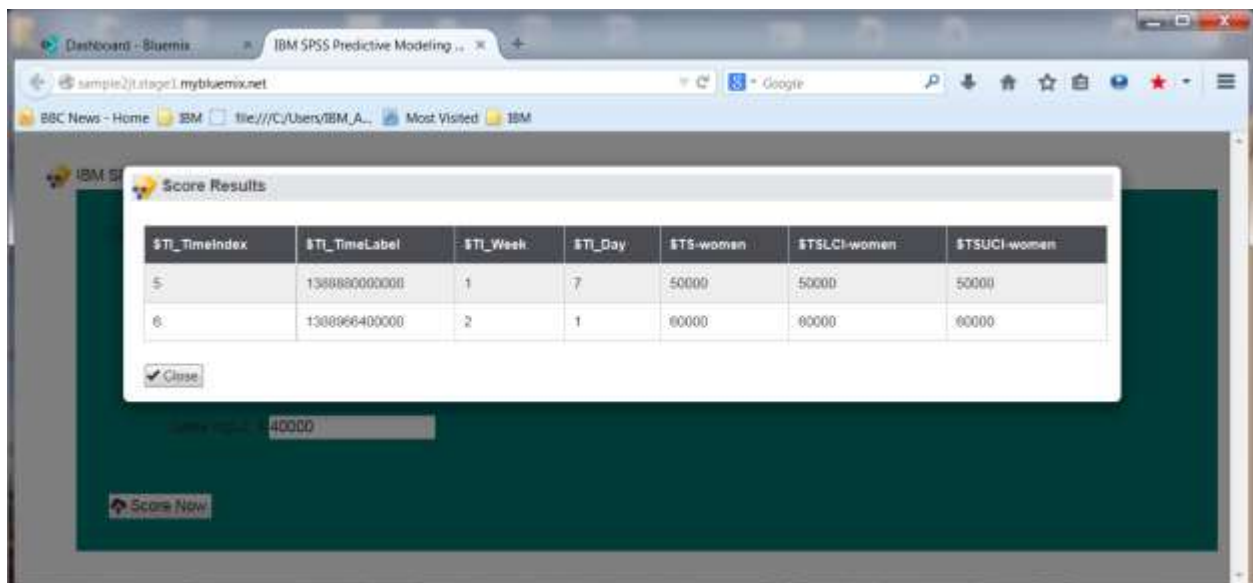
For Men's Clothing Sales forecasts:



The screenshot shows a web browser window with the URL 'sample2/latage1.mybluemix.net'. The page displays a 'Score Results' modal window with a table of data. The table has seven columns: \$TI\_TimeIndex, \$TI\_TimeLabel, \$TI\_Week, \$TI\_Day, \$TS-men, \$TSLCI-men, and \$TSUCI-men. There are four rows of data, indexed 5 through 8. Below the table is a 'Close' button. In the background, a 'Score Now' button is visible.

\$TI_TimeIndex	\$TI_TimeLabel	\$TI_Week	\$TI_Day	\$TS-men	\$TSLCI-men	\$TSUCI-men
5	1388880000000	1	7	10915.129201	802.358084	21227.902317
6	1388966400000	2	1	11111.526551	790.753434	21424.289680
7	1389052800000	2	2	8711.23832	-1801.538404	19024.006749
8	1389139200000	2	3	11458.823238	1146.150121	21771.696355

For Women's Sales forecasts:



The screenshot shows a web browser window with the URL 'sample2/latage1.mybluemix.net'. The page displays a 'Score Results' modal window with a table of data. The table has seven columns: \$TI\_TimeIndex, \$TI\_TimeLabel, \$TI\_Week, \$TI\_Day, \$TS-women, \$TSLCI-women, and \$TSUCI-women. There are two rows of data, indexed 5 and 6. Below the table is a 'Close' button. In the background, a 'Score Now' button is visible.

\$TI_TimeIndex	\$TI_TimeLabel	\$TI_Week	\$TI_Day	\$TS-women	\$TSLCI-women	\$TSUCI-women
5	1388880000000	1	7	50000	50000	50000
6	1388966400000	2	1	60000	60000	60000

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: the learning curve will start all over again once you stop and re-start your application.