Assignment 5B

Spark Streaming

In this question, we will work with Spark Structured Streaming *(https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html)* to process a constant stream of data in real(-ish) time. (Note that we are **not** using the older DStream-style streaming where you work with RDDs.)

We will get our streaming data from Kafka *(https://kafka.apache.org/)* on our cluster. A process is already running that produces randomly-generated (x, y) points as plain text messages encode as text, separated by spaces like this: "`-740.844 -10829.371`".

The code that produces the messages is simple: it generates random points near a line. This will simulate a stream of sensor data. [It's also possible it will die for reasons I don't foresee: if you aren't getting messages, bug Greg.]

We will imagine a situation where storing all of the incoming data isn't practical or necessary. The sensible thing to do in this case it to do some filtering/aggregating/summarization on the data as it comes in and stores only those results.

We will do a simple linear regression *(https://en.m.wikipedia.org/wiki/Simple_linear_regression)* on the data (not using machine learning tools: just calculating the simple linear regression formula) and store the slope (which the Wikipedia page calls $\hat{\beta}$) and intercept ($\hat{\alpha}$) of the values. (That's maybe not a particularly sensible summary of the data to store, but it's a not-totally-trivial calculation, and we want that to make sure we can keep up.) A reasonable formula to calculate the slope and intercept:

$$\hat{\beta} = \frac{\sum xy - \frac{1}{n}\sum x \sum y}{\sum x^2 - \frac{1}{n}(\sum x)^2}$$
$$\hat{\alpha} = \frac{\sum y}{n} - \hat{\beta}\frac{\sum x}{n}$$

Technical Notes

There are messages being produced on three topics: `xy-1`, `xy-10`, and `xy-100`, which get one, ten, and 100 messages per second, respectively. If you want to see what's going on, you can try a simple consumer just to see the messages and confirm that they're coming through.

Also please write your streaming job with a modest **timeout** so you don't actually create an infinitely-running job:

```
stream = streaming_df.….start()
stream.awaitTermination(600)
```

Call your program `read_stream.py`. Take a command-line argument for the topic to listen on. You'll also have to load the Spark Kafka package on the command line.

When running Streaming, there is *a lot* of logging output from Spark. You can throw away the logging **and error** output and put the above together like this:

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.2.0 read_stream.py xy-10 2>/dev/n
```

Working with Streams

All of the cluster nodes are in the Kafka cluster, so you can start by connecting to any of them. In Spark Structured Streaming, that is:

```
messages = spark.readStream.format('kafka') \
        .option('kafka.bootstrap.servers', '199.60.17.210:9092,199.60.17.193:9092') \
        .option('subscribe', topic).load()
```

Once you create a Streaming DataFrame (like `messages` above), you can specify calculations on it almost exactly *(https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#unsupported-operations)* like any other DataFrame.

A Kafka DataFrame contains a column `'value'` which must be cast to a string to see the message:

```
lines  = messages.select(messages['value'].cast('string'))
```

The goal for this question is to aggregate to a DataFrame with a single row containing a slope and intercept.

Usually, data like this would be sent to a database, but output is one area that seems under-developed in Structured Streaming.

For this question, we will just output to the console (`.format('console')`), so we can read-off the slope and intercept. [?]

Spark ML: Colour Prediction

We have collected some data on RGB colours. When creating the experiment, I showed the user an RGB colour on-screen and gave options for the 11 English basic colour terms *(https://en.wikipedia.org/wiki/Color_term)* . The result is >2800 data points mapping RGB colours (each component 0–255) to colour words. You can find it in the usual places as `colour-words-1`.

Let's actually use it for its intended purpose: training a classifier. This data set is likely small enough that you can do this question entirely locally: the cluster probably won't actually speed things up.

Create a Spark program `colour_predict.py` that takes the input path on the command line. Our output paths will be hard-coded. You may want to do the same trick as before to ignore the stderr output:

```
spark-submit colour_predict.py colour-words-1  2>/dev/null
```

For this question, you **must create your classifiers as machine learning pipelines *(https://spark.apache.org/docs/2.2.0//ml-pipeline.html)*** . That is, each model will be a Pipeline *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.Pipeline)* instance containing each step in the workflow to classifying the colours. [In this specific problem, it might feel like you're doing redundant calculations, and you probably are, but we're forcing you to figure out how to put together a pipeline here.]

Provided Pieces

You can download a ZIP file with some pieces for this question.

The `colour_tools.py` file contains code for the colour space conversion, and to produce a nice plot of the predictions your models make. Both are described below when they're relevant.

The `colour_predict_hint.py` contains a skeleton of a solution to give you a little structure to work from. In particular, it gets the input/output things taken care of so you can move one to something more interesting.

Data Prep

The features in the input data are red, green, and blue values in columns `'R'`, `'G'`, `'B'`. The classifiers all need a single column containing a vector of values. i.e. three columns will have to become a vector of length three. The VectorAssembler *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.VectorAssembler)* transformer can do this for us.

The targets in the data are strings: `'red'`, `'black'` and so on. The classifiers in Spark insist on predicting numeric values. The StringIndexer *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.StringIndexer)* transformer can convert these values to numbers.

The transformers that convert the features and targets into the shape the estimators need will have to be in each of the pipelines we create.

Evaluating the Classifiers

We'll need a way to evaluate the models we build on the testing data. A MulticlassClassificationEvaluator *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.evaluation.MulticlassClassificationEvaluator)* can take the test targets and predicted targets and come up with a score we can use as a first approximation of “goodness” of a classifier.

Basic Classifiers

There are many classifiers in pyspark.ml *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#module-pyspark.ml.classification)* , but we will focus on two: RandomForestClassifier *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.classification.RandomForestClassifier)* and MultilayerPerceptronClassifier *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.classification.MultilayerPerceptronClassifier)* (i.e. non-recurrent neural network).

Create an instance of each of these to work on this problem. Get the best scores you can out of them.

There isn't much to be configured with the random forest classifier: you could perhaps tweak the number of trees or depth.

For the multilayer perceptron, you must specify the shape of the layers. The number of nodes in the first and last layers are fixed by the data. Experiment a little to find the best hidden layers. (Hint: this isn't a deep learning situation.)

Once you have working classifiers, you can uncomment the call to `plot_predictions` and have a look at the resulting plots. They will be written to files on the driver filesystem (since they are created by matplotlib, they will become files and never go to HDFS).

Feature Engineering

The problem of mapping RGB ↔ word is fundamentally one of human perception, but RGB colour space isn't about perception: it's about computer displays. The LAB colour space *(https://en.wikipedia.org/wiki/Lab_color_space)* is designed to encode something about human vision: maybe we can create better input features.

The provided function `rgb2lab_query` produces a SQL query that converts RGB colours (in columns `'R'`, `'G'`, `'B'`) to LAB colours (columns `'lL'`, `'lA'`, `'lB'`). That might be convenient to combine with a SQLTransformer *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.SQLTransformer)* .

Create pipelines for both the random forest and MLP classifiers that transform to LAB colours and classify based on those.

Now that you have four pipelines, compare the scores. Tweak everything to get the best results you can from each scenario without taking excessive processing time. [❓]

Predicting Weather: How Hard Can It Be?

Want to take conversations about the weather to the next level? How about using it as a segway to share your thoughts on model selection and hyperparameter tuning!

The task

Building on what we've learned about Spark ML pipelines and feature engineering, this exercise revisits the familar weather dataset from Assignment 3B with a slight twist. This time, please work on data in the folders provided on HDFS as `tmax-{1,2,3,4}` and test out Spark's regression algorithms.

We would like you to **build a predictor for the maximum temperature** and **tune the hyperparameters of your chosen regression estimator** to see how good Spark gets at predicting the weather.

Provided pieces

To enable you to focus on algorithm comparison and tuning, we've provided a ZIP file with some helper code (or check HDFS `/user/sbergner/A5_weather.zip`) to get you started. It might take some time to familiarize yourself with the provided code (see also `weather_tools.py`), but overall it is meant to save you time when trying out different regressors.

Perform the TODO items in `weather_hint.py` and save your version as `weather_predict.py`:

a) create a feature transformer that calculates the day of year for the date and outputs it as column `doy`,

b) construct a pipeline with stages that provide all the features for an estimator that is input to the function `make_weather_trainers`.

This enables you to run the code with a LinearRegression and get some prediction output. Now use this framework to c) **find better estimators** try different Regressor types and explore a number of different parameter settings using the grid builder. If you manage to make predictions that achieve a coefficient of determination `r2>0.75` or a root mean-squared error `rmse < 5` degrees on one of the larger datasets, you've done well.

Questions

In a text file `answers.txt`, answer these questions:

1. What is your best guess for the slope and intercept of the streaming points being produced?

2. Is your streaming program's estimate of the slope and intercept getting better as the program runs? (That is: is the program aggregating all of the data from the start of time, or only those that have arrived since the last output?)
3. In the colour classification question, which pipeline gave the best results? What was the score and what parameters did you use to get it?
4. Why is your temperature predictor better than the one in the example? What strategy did you use to find it?

Submission

Submit your code and `answers.txt` to the CourSys activity Assignment 5B.

Updated Tue Nov. 14 2017, 15:11 by sbergner.