Programming Assignment 2

Deadline: Sunday March 31st 2024

Delivery method : Upload to elearn a tarball with your code and your report.

Million Song Pipeline In this assignment we are going to train a model that predicts the release year of the song given a set of audio features.

1 Data Collection

1.1 Dataset

As dataset use file dataset.csv of the homework tarball.

1.2 Load the raw data into spark

Load the raw data into Spark by creating an RDD[String]. Suppose that we name the resulted RDD as baseRdd.

1. How many data points are there?
2. Use *take* and print the top five datapoints

1.3 Convert datapoints into LabeledPoints

A labeled point is a local vector, either dense or sparse, associated with a label/response. In MLlib, labeled points are used in supervised learning algorithms. MLlib uses a double to store a label, so we can use labeled points in both regression and classification. To see how to create a LabeledPoint visit `https://spark.apache.org/docs/latest/mllib-data-types.html`

1. Write a function that takes as input a comma separated string, similar to a line of the raw data, converts it to a list of Doubles and returns a LabeledPoint. The label (i.e. the release year) is the first value of the string and the features are the rest. Represent the features as a dense vector (refer again to the documentation above).
2. Use the function that you just created and convert baseRdd into an rdd, called parsedPointsRdd which is of type RDD[LabeledPoint].
3. Print the label of the first element of parsedPointsRdd.
4. Print the features of the first element of parsedPointsRdd.
5. Print the length of the features of the first element of parsedPointsRdd.
6. Find the smallest and largest labels in parsedPointsRdd.

1.4 Shift Labels

In learning problems, it is often natural to shift labels such that they start from zero. So:

1. Create shiftedPointsRdd by transforming parsedPointsRdd by substracting from all labels of the latter the smallest label that you found above.
2. What is the min and max label of shiftedPointsRdd?

1.5 Create Training, validation and test sets

1. Use method randomSplit of RDDs and generate RDDs with name trainData, valData and testData. You are going to generate all three datasets with a single call to that function if you pass as argument the following values:

   ```
   val weights = Array(.8, .1, .1)
   val seed = 42
   val Array(trainData, valData, testData) = ?
   ```

2. Cache all three of the generated tables.

3. Print the element count of each dataset. Does their sum equal to the element count of shiftedPointsRdd?

2 Create a Baseline Model

2.1 The average label model

A very simple -but natural- baseline model is one that produces the same prediction, independent of the given data point, using the average label of the training set as the constant prediction value.

1. Compute the average (shifted) song year on the training set.

2. Write a function called baseLineModel that takes as input a LabeledPoint and returns a “prediction”, which in this case is always the average song year that you calculated above.

2.2 Use MLlib to calculate RMSE

Implement function calcRmse. It should take as input an RDD[Double, Double] and it should return a Double. The first element of each RDD element is a prediction and the second is the corresponding label.

For this implementation, you are going to instantiate a RegressionMetrics object of MLlib; rootMeanSquaredError is one of its methods. See: `https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html`

2.3 Training, validation and test RMSE

1. Use the baseline model and create RDDs predsNLabelsTrain, predsNLabelsVal, predsNLabelsTest from trainData, valData and testData respectively. The new RDDs should be of type RDD[Double, Double] with prediction as the first element and label as the second one.

2. Print the RMSE of each set

3 Linear Regression with Gradient Descent

Now we are going to train a model using Gradient Descent (GD) for Linear Regression (LR). Remember that the update rule of GD for LR is: $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \alpha^{(i)} \sum_{j=1}^{N} (\mathbf{w}^{(i)^T} \mathbf{x}_j - \mathbf{y}_j)\mathbf{x}_j$. For this exercise ignore the intercept with the y-axis (i.e. the dimension of the output parameters should be equal to the dimension of features).

3.1 Gradient Summand

Implement a function that implements the summand of this update. The summand is $(\mathbf{w}^T\mathbf{x} - y) \cdot \mathbf{x}$. Name the function as gradientSummand. Given that this function uses linear algebra operations (dot product and sum of vectors), we are going to use the DenseVector object of the breeze library. The function should take as inputs a DenseVector of weights and a LabeledPoint. To implement the dot product use method *dot* of the DenseVector. Here we show the signature of the function and a couple of test cases:

```
import breeze.linalg.DenseVector
def gradientSummand(weights: DenseVector, lp: LabeledPoint)

val example_w = DenseVector(1.0, 1.0, 1.0)
val example_lp = LabeledPoint(2.0, Vectors.dense(3, 1, 4))
gradientSummand(example_w, example_lp)
//gradient_summand = (dot([1 1 1], [3 1 4]) - 2) * [3 1 4] = (8 - 2) * [3 1 4] = [18 6 24]
```

```
val example_w = DenseVector(.24, 1.2, -1.4)
val example_lp = LabeledPoint(3.0, Vectors.dense(-1.4, 4.2, 2.1))
gradientSummand(example_w, example_lp)
//gradient_summand = [1.7304, -5.1912, -2.5956]
```

3.2 Use weights to make predictions

Implement function *getLabeledPrediction* that takes a DenseVector of weights and a labeledPoint and returns a (prediction, label) tuple. Make a prediction by computing the dot product between weights and the feautures of a labeledPoint.

3.3 Implement Gradient Descent and Monitor Error Progress

Write a function that implements Gradient Descent for Linear Regression and name it lrgd. The function takes as inputs an RDD of LabeledPoint and an integer that denotes the number of iterations. The function returns the model parameters w, and an array of length equal to the number of iterations that contains the trainning error per iteration. As a starting point, you are provided with a partial implementation of function lrgd. For this step you should:

1. Fill in the gaps of function lrgd that you were provided.
2. Use the training set and the lrgd function to train your model. Use 50 iterations. Print the per-iteration rmse.
3. Does Gradient Descent converge?
4. If it does not, which parameter out of numIters (number of iterations), alpha is responsible? Why?
5. Fix the responsible parameter, and for the parameter set that produces an algorithm that converges, print α, number of iterations, training error per iteration (first 50 if numIters > 50) and the model parameters (weights).

3.4 Evaluate the model

Calculate the RMSE on the validation set.

4 Train using MLlib and grid search

4.1 MLlib Linear Regression

Use LinearRegression to train a model with elastic net regularization and an intercept. This method returns a LinearRegression model. In this section we are going to use the more up to date library spark.ml. This library works with Dataframes. You are provided with the proper conversions of trainData, testData and valData into Dataframes. You are also provided with a code snipet that 1) instantiates a LinearRegression object and 2)it fits a model using 50 iterations, regularization parameter of 0.1 and allows the use of y-intercept. The snippet also shows you how to evaluate with RMSE the model that you just generated. For documentation see `https://spark.apache.org/docs/latest/ml-classification-regression.html#linear-regression` and `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.regression.LinearRegression`.

1. Print the model coefficients and intercept.
2. Print the RMSE on the validation set.
3. Transform the validation set and print the first 10 predictions

4.2 Grid Search

We are already outperforming the baseline on the validation set, but let's see if we can do better. Perform grid search to find a good regularization parameter. Try regParam values $1e-10, 1e-5, 1$.

1. What is the RMSE of the best model?
2. What regularization parameter achieves that?

5 Add interactions between features

5.1 Add 2-way interactions

So far, we have used the features as they were provided. Now, we will add features that capture the two way interactions between our existing features. Ignore for now the optimization that reduces dimensions. You are provided with the transformation of the trainData to an RDD of LabeledPoints that contain quadratic features.

1. Transform crossTrainDataRDD (see code snippets) into a Dataframe.
2. Transform the validation and tests into Dataframes with quadratic features.

5.2 Build interaction model

Build the new model using MLlib. Use 500 iterations, regulation parameter of $1e-10$ and allow intercept.

5.3 Evaluate interaction model

Use the validation data to find the RMSE of the new model.

5.4 Finally Evaluate the interaction model on test data

Our next step is to evaluate the new model on the test dataset. Note that we haven't used the test set to evaluate any of our models. Because of this, our evaluation provides us with an unbiased estimate for how our model will perform on new data. If we had changed our model based on viewing its performance on the test set, our estimate of RMSE would likely be overly optimistic.

1. Print the RMSE for both the baseline model and our new model. With this information, we can see how much better our model performs than the baseline model.
2. Print the first 50 predictions on the test set (hint! Use the *transform* method of LinearRegressionModel). To isolate the column "label" of a LabeledPoint originated DataFrame type:

   ```
   myDataframe.select("label")
   ```

5.5 Use a pipeline to create the interaction model

The final step is to create the interaction model using a Pipeline. Note that Spark contains the PolynomialExpansion transformer which will automatically generate interactions for us. In this section, you'll need to generate the PolynomialExpansion transformer and set the stages for the Pipeline estimator. Make sure to use a degree of 2 for PolynomialExpansion. The pipeline should contain two stages: the polynomial expansion and the linear regression. You are given most of the code for this step. You are expected to fill the missing parts. In the end, print the RMSE of the test set.