BDM: Project

Universitat Politècnica de Catalunya

June 13, 2021

A Classification pipelines

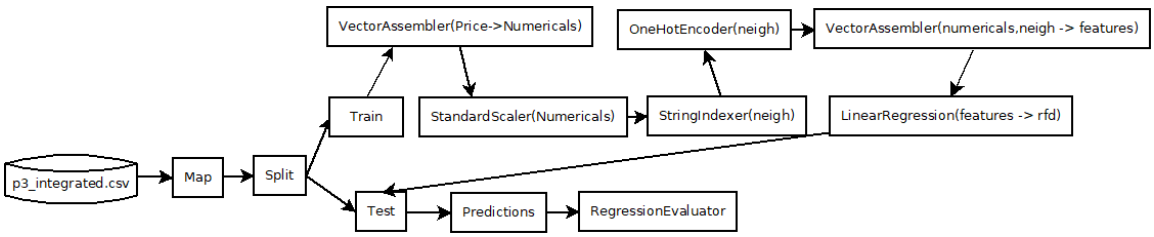

Figure 1: Classification pipeline

A.1 Model Construction

To construct the model we read the CSV provided and load it onto our Spark session, map the CSV to our desired names and split it into 30% test and 70% train, moving our "price" column into the "numericals" via the use of the VectorAssembler, so as to be able to apply the StandardScaler, which standarizes our "numericals" column for easier manipulation with ML algorithms. To be able to apply the OneHotEncoder, we apply a StringIndexer over our neighborhood variable and then apply the OneHotEncoder, leaving the one hot encoding of our neighborhoods in "neigh_cat". Finally, using a VectorAssembler step again, we fuse the one hot version of neighborhood with the scaled version of price into the "features" variable, we then apply the LinearRegression step to build a linear regression model, using the one hot encoding of neighborhood and scaled price to predict the RFD.

We train the model using the previously discussed pipeline, save it onto disk for future use, and test its performance, via making the model transform our test data into predictions. Having this predicted data, we make a new pipeline where we apply the RegressionEvaluator over the variable RFD and feed it our transformed test, obtaining the Mean Squared Error as our metric for the model's performance.

A.2 Prediction with model

To predict with the model, first we load the PipelineModel from disk, then we connect to the stream with Kafka, and, apply a transformation to the variable names from the observations coming from the stream, changing them to the exact same names as the model uses, and feed the PipelineModel this transformed RDD, causing it to generate a prediction.

B Streaming Algorithms

For our streaming algorithms, we implemented Heavy Hitters and Exponentially Decaying Window, however, due to using windows, unless the window size is 1, the behaviour is not identical to that of a pure stream, but an approximation, for efficiency's sake.

B.1 Heavy Hitters

```
def heavyHitters(c1: Counter, c2: Counter): Counter = {
  def go(c: Counter): Counter = {
    if (c.size <= maxSize) {
      c
    } else {
      val c1 =
        c.map { case (e, count) => (e, count - 1) }
          .filter { case (_, count) => count > 0 }
      go(c1)
    }
  }

  go(c1.unionWith(c2)(_ + _))
}

def run(stream: DStream[KafkaSample]): Unit = {
  stream
    .map { case KafkaSample(neigh, _) =>
      TreeMap((neigh, 1))
    }
    .reduceByWindow(heavyHitters, windowSize, windowSize)
    .map(_.map(_._1).toList)
    .print()
}
```

For heavy hitters we make use of the window to compute batches at once, this being more efficient, but unless window is of size 1, the behaviour is not as equal to that of heavy hitters with a pure stream. The way it is

performed is that each window is turned into a key value, with the value being the amount of repeated keys for that particular key, then, this is added to the existing heavy hitter RDD, where, if they share keys, the values are added, otherwise they key value is placed. Once this is done, until the amount of elements is equal to the maximum amount of allowed elements, we call a transformation that removes 1 from every value and filters those equal to 0, once the amount of elements is finally below the maximum, this process is stopped.

B.2 Exponentially Decaying Window

```
def go(newValues: Seq[Double], state: Option[Double]): Option[Double] = {
  if (newValues.length == 0) {
    state
  } else {
    val avg = newValues.sum / newValues.length
    state match {
      case None =>
        Some(avg)
      case Some(oldState) =>
        Some(oldState * (1 - c) + avg)
    }
  }
}

def run(stream: DStream[KafkaSample]): Unit = {
  stream
    .map { case KafkaSample(neigh, price) =>
      (neigh, price)
    }
    .updateStateByKey(go)
    .print()
}
```

For exponentially decaying window we make use of the window to calculate an average of the values of each neighborhood that arrived at the window, then we multiply the old state by 0.2 and add the new average. We do this to preserve the efficiency of processing windows, and to keep a behaviour that is similar enough to the pure stream Exponentially Decaying Window, and, if the window is changed to 1 element, it is the same exact behaviour.