Verslag Distributed Databases

Dylan Cluyse, Laura Renders, Liam Goethals

11 december 2022

Inhoudsopgave

Hoofdstuk 1

Inleiding

Tijdens het opleidingsonderdeel 'Distributed Databases' maakten wij kennis met Apache Spark. Spark biedt een data-analysetechnologie aan gericht op grootschalige gegevensverwerking. Deze technologie wordt aangeboden voor Java, Python, R en Scala. Voor dit opleidingsonderdeel gebruiken we Java als programmeertaal. Spark heeft pakketten die zich richten op andere aspecten binnen dataverwerking, één daarvan is *MLLib*. *MLLib* biedt *machine learning* technieken aan (ML). Wij kregen opdracht om via het platform Kaggle deel te nemen aan ML-gerichte competities, met het doel om zo meer vertrouwd te raken met ML en Spark.

In dit verslag nemen wij u mee in de wereld van Spark-ML. Wij tonen u met volle plezier drie concrete voorbeelden die verschillende regressie- en classificatiemethoden *highlighten*. U kan de gemaakte oefeningen digitaal terugvinden op deze link [1].

Deze opdracht was een proces van zelfstudie. De gehanteerde technieken zijn gebaseerd op de documentatie van Apache Spark [2] en van Spark MLLib [3].

Allereerst willen wij de tijdsduratie van een taxi-rit in downtown New York berekenen met regressie. Vervolgens detecteren wij kredietkaartfraude met behulp van binaire classificatie. Als derde oefening willen wij op basis van tekstinhoud achterhalen of een Tweet gerelateerd is aan een ramp. Tot slot geven wij u onze bevindingen mee van het MLLib pakket. Hier leggen we de aanpak van Spark en Sci-kit Learn, een pakket dat we in het opleidingsonderdeel Machine Learning zagen, parallel tegenover elkaar.

[1] GitHub repository: https://github.com/Dyashen/Distributed_Databases/tree/main/Kaggle
[2] Apache Spark documentatie: https://spark.apache.org/docs/latest/
[3] Spark MLLib documentatie: https://spark.apache.org/docs/latest/ml-guide.html

Hoofdstuk 2

Tijd van een taxi-verplaatsing voorspellen met regressie.

Probleemstelling

Als eerste opdracht maakten wij de ”New York City Taxi Trip Duration”wedstrijd. De competitie voorziet twee datasets: een training- en een testset. Het doel is om de totale tijdsduur van een taxirit in seconden te berekenen. Deze competitie kiest de inzending met de kleinste *Root Mean Square Log Error* (RMSLE). Volgende kolommen werden gegeven:

- Het ID en het verkopersnummer.
- Het aantal passagiers in een taxi.
- De longitude en latitude van het vertrek- en aankomstpunt.
- De pick-up en drop-off tijd.

De competitie kan u terugvinden op deze link. [1]. Wij namen inspiratie uit vooral de officiële documentatie en uit één notebook. De code, in *Python*, kan u hier [2] terugvinden.

Algemeen

Wij declareren *final* variabelen. Deze worden meermaals gebruikt en worden niet aangepast doorheen de applicatie. Deze variabelen hebben een andere waarde bij de volgende twee oefeningen, maar ze komen in ieder model aan bod. De verhouding is de ratio waarop we de trainingset splitsen. De label is de kolom waarop ons model zal voorspellen. De prediction houdt de naam van de prediction-kolom bij. We houden de metriek bij dat we nodig hebben. Als laatste geven wij de metriek mee. Voorlopig kijken we naar het model met de beste RMSE waarde, daarop berekenen we de RMSLE.

```
1 private static final double[] verhouding = { 0.8, 0.2 };
2 private static final String label = "target";
3 private static final String prediction = "prediction";
```

Uiteraard starten wij de applicatie met een Spark-object. Wij passen één instelling aan om zo de irrelevante CLI-informatie achterwege te laten.

```
1 private static SparkSession spark = SparkSession.builder()
2   .appName("DisasterTweet")
3   .master("local[1]")
4   .getOrCreate();
5
6 spark.sparkContext().setLogLevel("ERROR");
```

[1] Link naar competitie: http://bit.ly/3umBKN5
[2] Link naar gevolgde notebook: http://bit.ly/3F2Aofj

Data ophalen

De Kaggle-competitie bevat een training- en een testset. Dit in CSV-formaat. We kiezen ervoor om het schema vooraf op te bouwen. Dit geeft ons een snelheidsvoordeel, want met een *pre-built* schema moet Spark zelf niet de datatypes achterhalen. Voor de datums werken wij met een *Timestamp* type, want het bevat zowel de datum als het tijdstip wanneer een persoon werd opgehaald of afgezet. De code vindt u hieronder terug:

```
  private static Dataset<Row> getTraining() {

  List<StructField> fields = Arrays.asList(
    DataTypes.createStructField("id", DataTypes.StringType, false),
    DataTypes.createStructField("vendor_id", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_datetime", DataTypes.TimestampType, false),
    DataTypes.createStructField("dropoff_datetime", DataTypes.TimestampType, false),
    DataTypes.createStructField("passenger_count", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_longitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_latitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("dropoff_longitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("dropoff_latitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("store_and_fwd_flag", DataTypes.StringType, false),
    DataTypes.createStructField("trip_duration", DataTypes.DoubleType, false)
  );

  StructType schemaData = DataTypes.createStructType(fields);

  Dataset<Row> dataset = spark.read()
    .option("header", true)
    .schema(schemaData)
    .csv("src/main/resources/train.csv");

  return dataset
    .withColumn("hour", hour(col("pickup_datetime")))
    .withColumn("day", dayofweek(col("pickup_datetime")))
    .drop("id", "pickup_datetime", "dropoff_datetime");
}

private static Dataset<Row> getTest() {
  List<StructField> fields = Arrays.asList(
    DataTypes.createStructField("id", DataTypes.StringType, false),
    DataTypes.createStructField("vendor_id", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_datetime", DataTypes.TimestampType, false),
    DataTypes.createStructField("passenger_count", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_longitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("pickup_latitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("dropoff_longitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("dropoff_latitude", DataTypes.DoubleType, false),
    DataTypes.createStructField("store_and_fwd_flag", DataTypes.StringType, false)
  );

  StructType schema = DataTypes.createStructType(fields);

  Dataset<Row> dataset = spark.read()
    .option("header", true)
    .schema(schema)
    .csv("src/main/resources/test.csv");

  return dataset.withColumn("hour", hour(col("pickup_datetime")))
    .withColumn("day", dayofweek(col("pickup_datetime")))
    .drop("id", "pickup_datetime");
}
```

Data Cleaning

De data is te *ruw*, en daarom maken wij een *clean*-functie aan. Zonder de *outliers* is het model nauwkeuriger. Wij passen het volgende stappenplan toe:

1. De *pick-up* en *drop-off* datetime wordt meegegeven als datetime-object. We veranderen de datetime-kolommen *pickuptime* naar *hour* en *day*. Zo krijgen we een pure numerieke waarde (bijvoorbeeld 1 tot en met 7 voor *day*).
2. We verwijderen de coördinaten-*outliers* uit de dataset. Hiervoor kijken we naar de rijen met een veel te hoge *longitude* en/of *latitude*.
3. We verwijderen de *distance-outliers* uit de dataset. Dit gebeurt na het berekenen van de distance. De functie hiervoor komt zo dadelijk aan bod.
4. We verwijderen alle rijen die géén passagiers meenemen. Alle rijen waar het aantal passagiers gelijk is aan 0 laten we vallen.
5. We *shufflen* de dataset. Dit wordt bij het splitsen nog eens gedaan, maar zo zijn we dubbel zeker dat er geen strikte volgorde is bij het splitsen.

```
private static Dataset<Row> clean(Dataset<Row> dataframe) {

  dataframe = dataframe.na().drop(); // alle rijen met null-waarden verwerpen

  double latMin = 40.6;
  double latMax = 40.9;
  double longMin = -74.25;
  double longMax = -73.7;

  // outliers verwerpen o.b.v. de longitude/latitude values
  dataframe = dataframe
    .where(col("pickup_longitude").$greater$eq(longMin))
    .where(col("dropoff_longitude").$greater$eq(longMin))
    .where(col("pickup_latitude").$greater$eq(latMin))
    .where(col("dropoff_latitude").$greater$eq(latMin))
    .where(col("pickup_longitude").$less$eq(longMax))
    .where(col("dropoff_longitude").$less$eq(longMax))
    .where(col("pickup_latitude").$less$eq(latMax))
    .where(col("dropoff_latitude").$less$eq(latMax));

  // outliers verwerpen o.b.v. de distance
  double rightOuter = dataframe
    .select(avg(col("distance")).plus(stddev(col("distance"))))
    .first()
    .getDouble(0);

  double leftOuter = dataframe
    .select(avg(col("distance")).minus(stddev(col("distance"))))
    .first()
    .getDouble(0);

  dataframe = dataframe
    .where(col("distance").$less$eq(rightOuter))
    .where(col("distance").$greater$eq(leftOuter));

  dataframe = dataframe
    .where(col("passenger_count").$greater(0)); // minstens één passagier in de taxi

  dataframe = dataframe
    .orderBy(rand()); // willekeurige volgorde

  return dataframe;
}
```

User-Defined Functions

De coördinaten van het vertrek- en aankomstpunt liggen te dicht bij elkaar. Daarnaast bieden deze vier waarden weinig inbreng aan het model. Daarom berekenen wij de afstand tussen de vertrek- en aankomstcoördinaten met de *haversine*-formule, ofwel de formule om de grootcirkelafstand te berekenen. Deze formule dient om de afstand op een bol oppervlak te berekenen. Hiervoor maken we een *user-defined-function* (UDF) aan dat vier parameters opvraagt: *de pickup longitude, pickup latitude, de dropoff-longitude en de dropoff-latitude*. We volgen de wiskundige formule, in drie aparte stappen. Wiskundige bewerkingen, indien mogelijk zoals de vierkantswortel, doen we met de Math-library van Java.

```
private static Dataset<Row> addDistance(Dataset<Row> dataframe) {
  return dataframe
    .withColumn("distance", call_udf("haversine",
      col("pickup_latitude"), col("pickup_longitude"),
      col("dropoff_latitude"), col("dropoff_longitude")))
    .drop("pickup_longitude", "dropoff_longitude", "pickup_latitude", "dropoff_latitude");
}

UDF4<Double, Double, Double, Double, Double> haversine = new UDF4<>() {
  public Double call(Double pickupLatitude, Double pickupLongitude, Double dropoffLatitude,
  Double dropoffLongitude) throws Exception {

    double radius = 6371 * 1.1; // Radius van de Aarde
    double lat1 = pickupLatitude;double lon1 = pickupLongitude; // pick-up
    double lat2 = dropoffLatitude;double lon2 = dropoffLongitude; // drop-off

    double deltaLat = Math.toRadians(lat2 - lat1);
    double deltaLon = Math.toRadians(lon2 - lon1);

    double a = Math.sin(deltaLat / 2) * Math.sin(deltaLat / 2) + Math.cos(Math.toRadians(lat1)
    ) * Math.cos(Math.toRadians(lat2)) * Math.sin(deltaLon / 2) * Math.sin(deltaLon / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    double d = radius * c;
    return d;
  }
};
spark.udf().register("haversine", haversine, DataTypes.DoubleType);
train = addDistance(train); test = addDistance(test);
```

Wij willen ook de gemiddelde snelheid van een taxi in *miles* per hour (mph) bekijken. Voor deze berekening maken wij een tweede UDF aan met als input-parameters: de totale afgelegde afstand en de totale tijdsduur van een rit. De output is de gemiddelde snelheid van een taxi per rit als kommagetal. De testset ontbreekt de verlopen tijd van een taxirit, dus we gebruiken deze kolom enkel als extra statistiek bij het analyseren van de data.

```
private static Dataset<Row> addSpeed(Dataset<Row> dataframe) {
    return dataframe.withColumn("speed", call_udf("speed", col("distance"), col(label)));
  }

UDF2<Double, Double, Double> speed = new UDF2<>() {
  public Double call(Double distance, Double time) throws Exception {
    return distance / (time / 3600);
}};
spark.udf().register("speed", speed, DataTypes.DoubleType);
```

Als tussentijdse controle tonen wij, in de terminal, de gemiddelde snelheid en gemiddelde afstand per dag en uur. Zo controleren we de transformaties voor: *distance, speed, hour en day*. Hieronder ziet u de gemiddelde afstand/snelheid per dag of per uur.

```
train.groupBy("hour").mean("speed").orderBy("hour").show();
train.groupBy("day").mean("speed").orderBy("day").show();
train.groupBy("hour").sum("distance").orderBy("hour").show();
train.groupBy("day").sum("distance").orderBy("day").show();
```

Opbouwen van de pipeline

Voor de pipeline werken we met vier stappen:

1. We hebben een kolom met een vlag, ofwel een categorische tekstwaarde. Dit moeten we *one-hot-encoden* of *labelen*. In Spark gebruiken we een *StringIndexer*.
2. Spark MLLib verplicht dat we werken met een vector van feature-waarden. Alle kolommen omzetten naar één featurekolom doen we met een *VectorAssembler*.
3. De waarden verschillen sterk qua grootte, en dit heeft een nadelig effect op ons model bij regressie. We moeten de waarden schalen. In Spark zijn er twee manieren: *MinMaxScaling* en *StandardScaling*. Wij gebruiken hier *MinMaxScaler* wat de waarden tussen 0 en 1 zal plaatsen. De hoogste waarde zal dicht bij één aanleunen. De kleinste waarde zal dicht bij nul aanleunen.
4. Als laatste object geven wij het regressiemodel mee. Wij testen vier verschillende modellen: lineaire regressie, *Random Forest Regressor* (RFR), *Gradient Boost Regressor* (GBR) en een gegeneraliseerd lineaire regressie (GLR). Wij vonden enkel het lineaire regressiemodel terug in de notebooks, maar uit interesse waagden wij een poging met drie extra modellen.

Hieronder vindt u de pipeline voor het lineaire regressiemodel. De volgende code vindt u terug in de *main*-methode.

```
StringIndexer indexer = new StringIndexer()
  .setHandleInvalid("keep")
  .setInputCol("store_and_fwd_flag")
  .setOutputCol("flag_ind");

VectorAssembler assembler = new VectorAssembler()
  .setInputCols(new String[] {
    "vendor_id", "passenger_count", "hour", "day", "flag_ind", "distance"
  })
  .setOutputCol("features");

MinMaxScaler minMax = new MinMaxScaler()
  .setInputCol(assembler.getOutputCol())
  .setOutputCol("scaledFeatures");

LinearRegression linReg = new LinearRegression()
  .setLabelCol(label)
  .setFeaturesCol(minMax.getOutputCol());

Pipeline pipelineLinReg = new Pipeline()
  .setStages(new PipelineStage[] {
     indexer, assembler, minMax, linReg
  });

PipelineModel model = pipelineLinReg.fit(trainSetSplit);
Dataset<Row> trainedLinReg = model.transform(trainTestSplit);
```

Bij *machine learning* splitsen we de trainingsset in twee delen op. We stellen een vaste verhouding in door middel van een *final* variabele. Wij kiezen voor de verhouding [80% - 20%]. Met de ingebouwde *randomSplit* methode splitsen we de dataframe.

```
Dataset<Row>[] datasets = train.randomSplit(verhouding, 42);
Dataset<Row> trainSetSplit = datasets[0];
Dataset<Row> trainTestSplit = datasets[1];
```

Finetuning

Om het meest optimale model te vinden, moeten we finetunen. In Spark pakken wij dit aan met een *ParamGridBuilder*. Dit is het equivalent van werken met een *GridSearch* in *Sci-kit Learn*. Wij geven verschillende waarden per hyperparameter mee. Iedere combinatie zal worden getest. Hoe meer parameters, hoe langer de uitvoertijd van de applicatie.

We passen crossvalidatie toe op ons model. Het beste model kiezen gebeurt op basis van een metriek. Zoals eerder gezegd willen we de beste RMSE hebben. We geven mee dat we voor de pipeline dezelfde objecten gebruiken als hierboven. Het *tweaken* gebeurt binnenin de pipeline. Hieronder een voorbeeld van een RFR-pipeline.

```
RandomForestRegressor rfr = new RandomForestRegressor()
  .setLabelCol(label)
  .setFeaturesCol(minMax.getOutputCol());

ParamMap[] paramGridRFR = new ParamGridBuilder()
  .addGrid(rfr.maxDepth(), new int[] { 5, 10, 15 })
  .addGrid(rfr.numTrees(), new int[] { 10, 15, 20 })
  .build();

CrossValidator cvRFR = new CrossValidator()
  .setEstimator(rfr)
  .setEvaluator(regEval)
  .setEstimatorParamMaps(paramGridRFR);

Pipeline pipelineRFR = new Pipeline()
  .setStages(new PipelineStage[] { indexer, assembler, minMax, cvRFR });

PipelineModel pipelineModelRFR = pipelineRFR.fit(trainSetSplit);
Dataset<Row> trainedRFR = pipelineModelRFR.transform(trainTestSplit);
```

Een alternatief voor crossvalidatie is *trainvalidationsplit* (TVS). Het verschil is dat TVS hier één datasetpaar maakt, terwijl crossvalidatie afhankelijk is van het aantal *folds*. Dit gebruiken we bij GLR en GBR. Hieronder ziet u de pipeline voor het GLR-model.

```
GeneralizedLinearRegression glr = new GeneralizedLinearRegression()
  .setFamily("gaussian")
  .setLink("identity")
  .setLabelCol(label)
  .setFeaturesCol(minMax.getOutputCol())
  .setMaxIter(10).setRegParam(0.3);

ParamMap[] paramGridGLR = new ParamGridBuilder()
  .addGrid(glr.maxIter(), new int[] { 15, 20, 25 })
  .addGrid(glr.regParam(), new double[] { 0.6, 0.9 })
  .build();

TrainValidationSplit trainValidationSplitGLR = new TrainValidationSplit()
  .setEstimator(glr)
  .setEvaluator(regEval)
  .setEstimatorParamMaps(paramGridGLR)
  .setTrainRatio(verhouding[0]);

Pipeline pipelineGLR = new Pipeline().setStages(new PipelineStage[] { indexer, assembler,
    minMax, trainValidationSplitGLR});

PipelineModel pipelineModelGLR = pipelineGLR.fit(trainSetSplit);
Dataset<Row> trainedGLR = pipelineModelGLR.transform(trainTestSplit);
```

Evaluatie en controle

Het achterhalen van de interessante features doen we met een *correlation matrix*.

```
private static void printCorrelation(Dataset<Row> dataframe) {
  Row r1 = Correlation.corr(dataframe, "features").head();
  System.out.printf("\n\nCorrelation Matrix\n");
  Matrix matrix = r1.getAs(0);
  for (int i = 0; i < matrix.numRows(); i++) {
    for (int j = 0; j < matrix.numCols(); j++) {
      System.out.printf("%.2f \t", matrix.apply(i, j));
    }
    System.out.println();
  }
}
```

Met Crossvalidatie voorspelt het model met de meest optimale combinatie hyperparameters. Wij vragen de volgende metrieken op: *Mean Squared Error* (MSE), *Root Mean Squared Error* (RMSE), *Mean Absolute Error* (MAE) en de R^2-waarde. Als laatst berekenen we de RMSLE door de log-functie te gebruiken op de RMSE. Met een *for*-lus kunnen wij voor iedere gevraagde metriek informatie ophalen.

```
private static void printRegressionEvaluation(Dataset<Row> dataframe) {
  String[] metricTypes = { "mse", "rmse", "r2", "mae" };
  System.out.printf("\n\nMetrics:\n");
  for (String metricType : metricTypes) {
    RegressionEvaluator evaluator = new RegressionEvaluator()
      .setLabelCol(label)
      .setPredictionCol(prediction)
      .setMetricName(metricType);

    double calc = evaluator.evaluate(dataframe);
    System.out.printf("%s: \t%.5f \n", metricType, calc);
  }

  RegressionEvaluator evaluator = new RegressionEvaluator()
    .setLabelCol(label)
    .setPredictionCol(prediction)
    .setMetricName(metric);

  double calc = evaluator.evaluate(dataframe);
  double log = Math.log(calc);
  System.out.printf("%s: \t%.5f \n", "rmsle", log);
}
```

Tussen de *fit* en de *transform* printen we de meest optimale hyperparameters uit. De beste combinatie wordt achterhaald door of het *crossvalidatiemodel*, ofwel de *traintestvalidation*. Hieronder een voorbeeld van de *traintestvalidation*. We moeten rekening houden met de volgorde waarin we de parameters toevoegen aan de ParamGridBuilder. Die volgorde komt overeen met de index van de parameters in de *ParamMap-array*.

```
System.out.printf("Ideale parameters:\nMax Iter: %s\nReg params: %s",
  trainValidationSplitGLR.getEstimatorParamMaps()[0].get(glr.maxIter()).toString(),
  trainValidationSplitGLR.getEstimatorParamMaps()[1].get(glr.regParam()).toString()
);
```

Hieronder een voorbeeld van hoe we de optimale hyperparameters ophalen bij een crossvalidatiemodel.

```
System.out.printf("Ideale parameters:\nMax depth: %s\nNumber of trees: %s",
  cvRFR.getEstimatorParamMaps()[0].get(rfr.maxDepth()).toString(),
  cvRFR.getEstimatorParamMaps()[1].get(rfr.numTrees()).toString()
);
```

Als laatste evaluatie willen we kijken naar de marge tussen de verschillende voorspelde waarden en de effectieve waarden. Zo weten we hoe sterk het model afwijkt en ook in hoeveel gevallen het in die mate afwijkt van de echte waarden. We werken met de volgende bereiken:

1. 0-50 seconden
2. 50-200 seconden
3. 200-500 seconden
4. 500-5000 seconden
5. 5000+ seconden

```
private static Dataset<Row> getRangeDataFrame(Dataset<Row> dataframe) {
  dataframe = dataframe.withColumn("margin",
    abs(col("prediction").minus(col("trip_duration"))));

  Dataset<Row> range = dataframe
    .withColumn("range",
       when(col("margin").leq(50), "0-50")
      .when(col("margin").leq(200), "50-200")
      .when(col("margin").leq(500), "200-500")
      .when(col("margin").leq(5000), "500-5000")
      .otherwise("5000+"));
  return range.groupBy("range").count();
}
```

Inzending

Voor de opdracht gaan we alle voorspellingen opslaan en koppelen aan het ID. We hergebruiken deze code, dus we plaatsen dit in een functie. Bij het opslaan geven we een savemode mee. De *Overwrite* zorgt ervoor dat we geen foutmelding krijgen als de map al bestaat. Het resultaat wordt telkens overgeschreven. We geven u mee dat deze code herhaalt wordt voor alle oefeningen. In het verslag vermelden we enkel hier deze functie.

```
private static void createSubmission(Dataset<Row> dataframe, String fnaam) {
  String output = "src/main/resources/" + fnaam + ".csv";
  dataframe = dataframe.select("id","prediction");
  dataframe.write().mode(SaveMode.Overwrite).csv(output);
}

// In de main-methode
createSubmission(predictedLogReg, "submissionLogReg");
```

Conclusie

De opdracht vergde een stuk denkwerk. Om te oefenen, probeerden wij werkwijzen uit de documentatie toe te passen op vrij beschikbare datasets op Kaggle. Dit om te wennen aan de syntax en structuur. Enkele zaken uit de lessen kwamen hier aan bod. Zo moesten we twee *UDF*'s implementeren om twee kolommen te berekenen, waaronder een kolom met de grootcirkelafstand. Op basis van *groupby*'s kunnen we regelmatig data-analyse uitvoeren. Deze werkwijze hielp ons om outliers terug te vinden.

De resultaten van ons model vielen in lijn met andere *notebooks* die deze technieken gebruikten. Notebooks dat regressiemodellen gebruikten, zoals *XGBoost* wordt momenteel niet door Spark aangeboden. Daarnaast beschikten wij niet over een *built-in* methode om de RMSLE te berekenen. Wij losten dit tijdelijk op door de log van de RMSE te berekenen. Na zoekwerk leidden wij af dat het logaritme van de RMSE berekenen een deelse oplossing was. *Sci-kit Learn* biedt hiervoor een *built-in* function aan.

Hoofdstuk 3

Kredietkaartfraude bestrijden met binaire classificatie.

Probleemstelling

Tijdens het zoeken van een classificatieprobleem stuiten wij ons op deze dataset. Deze dataset bestaat uit circa 25.000 rijen. We hebben hier twee klassen: echte en fraudulente kredietkaarttransacties. We pakken dit probleem aan met een binair classificatiemodel. De dataset kan u hier [1] terugvinden. Voor deze oefening baseerden wij ons vooral op de nodige officiële documentatie om een classificatieprobleem in Spark aan te pakken. Wij namen inspiratie uit het notebook dat u hier [2] kan terugvinden.

Data ophalen

Voor deze opdracht krijgen we één dataset, namelijk *creditcard.csv*. We hebben hier meer dan 30 features. Daarom kiezen we ervoor om geen apart schema aan te maken, maar we gebruiken de *inferschema* optie van Spark. Zo achterhaalt Spark zelf het datatype van een kolom. *Spark* herkent hier alle nodige features als een *double*.

```
private static Dataset<Row> getData() {
  return spark.read()
    .option("header", true)
    .option("inferSchema", true)
    .csv("src/main/resources/creditcard.csv");}
```

Data Cleanen

De oorspronkelijke dataset is *skewed*. Bij het bestuderen van de dataset merken we op dat 99.83% van alle transacties echt zijn, terwijl enkel 0.71% wél fraudulent is. Dit pakken we aan met een *clean*-methode. Zonder deze methode hebben we een grotere kans op *overfitting*, want het model neemt aan dat de meeste transacties echt zullen zijn. We lossen dit op door een *sub-sample* te maken van de volledige dataframe. We voorzien een 50:50 ratio tussen echte en fraudulente transacties.

```
private static Dataset<Row> createSubSample(Dataset<Row> dataframe) {
  dataframe = dataframe.sample(1.0);
  dataframe = dataframe.orderBy(rand());
  Dataset<Row> nietFraudulent = dataframe.where(col(label).equalTo(1));
  long lengte = nietFraudulent.count();
  Dataset<Row> fraudulent = dataframe.where(col(label).equalTo(0)).limit((int) lengte);
  dataframe = nietFraudulent.union(fraudulent);
  return dataframe.sample(1.0);}
```

[1] Link naar competitie: http://bit.ly/3iM6zIB
[2] Link naar gevolgde notebook: https://bit.ly/3Y9Kmo6

Pipeline

Alle nodige features starten met een 'V'-teken. Met behulp van een *for-loop* en de *startswith* methode kunnen wij de features zo aan een array van Strings toevoegen. Het alternatief is 28 features manueel ingeven bij het opstellen van de assembler. De *assembler* zal de 28 kolommen omzetten naar één vector van features.

```
String[] arrFeatures = new String[28];
int teller = 0;
for (String kolom : creditCardSubSample.columns()) {
  if (kolom.startsWith("V")) {
    arrFeatures[teller] = kolom;
    teller++;
  }
}
```

Het model bevat 28 features. Zelf weten we niet welke er noodzakelijk zijn. Als experiment, en toepassing van een techniek uit de lessen *Business Intelligence*, passen wij PCA toe aan de pipeline. Zo voorzien we *dimensionality reduction* op op onze features. Zo behouden we enkel de features nodig voor het model. Voor de binaire classificatie keken wij naar drie verschillende classificatiemodellen: *Random Forest, Lineaire SVM en logistische regressie*. Hyperparameters tunen doen we met een *ParamGridBuilder*.

```
VectorAssembler assembler = new VectorAssembler()
  .setInputCols(arrFeatures)
  .setOutputCol("features");

MinMaxScaler minmax = new MinMaxScaler()
  .setMax(1.0).setMin(0.0)
  .setInputCol(assembler.getOutputCol())
  .setOutputCol("scaledFeatures");

PCA pca = new PCA().setInputCol(minmax.getOutputCol()).setOutputCol("pcaFeatures").setK(3);

RandomForestClassifier rfc = new RandomForestClassifier()
  .setLabelCol(label)
  .setFeaturesCol(pca.getOutputCol());

ParamMap[] paramGridRFC = new ParamGridBuilder()
  .addGrid(rfc.maxDepth(), new int[] { 3, 7, 9 })
  .addGrid(rfc.numTrees(), new int[] { 40, 60, 80 }).addGrid(pca.k(), new int[] { 3, 6, 9 })
  .build();

CrossValidator cvRFC = new CrossValidator().setEstimator(rfc)
  .setEvaluator(new BinaryClassificationEvaluator().setLabelCol(label))
  .setEstimatorParamMaps(paramGridRFC)
  .setNumFolds(3);

Pipeline pipelineRFC = new Pipeline().setStages(new PipelineStage[]
  { assembler, minmax, pca, cvRFC }
);

PipelineModel pipelineModelRFC = pipelineRFC.fit(trainSetSplit);

System.out.printf("Ideale parameters:\nMax depth: %s\nNum trees: %s\nk: %s",
  cvRFC.getEstimatorParamMaps()[0].get(rfc.maxDepth()).toString(),
  cvRFC.getEstimatorParamMaps()[1].get(rfc.numTrees()).toString(),
  cvRFC.getEstimatorParamMaps()[2].get(pca.k()).toString()
);

Dataset<Row> predictionsRFC = pipelineModelRFC.transform(trainTestSplit);
createSubmission(predictionsRFC, "submissionRFC");
```

Evaluatie

We hebben een confusionmatrix nodig om het classificatiemodel te evalueren. Zo zien wij hoe goed het model de klassen kan voorspellen. Met de confusion matrix kunnen wij ook de precision en recall berekenen. Deze functies zijn ingebouwd in Spark, dus dit hoeven wij niet manueel te berekenen. Als laatste berekenen we de nauwkeurigheid van het model. Hieronder vindt u de twee methoden om de *confusion matrix* weer te geven, en ook de methode om de nauwkeurigheid van het model op te halen.

```
  private static double getAreaROCCurve(Dataset<Row> dataframe) {
  return new BinaryClassificationEvaluator()
    .setLabelCol(label)
    .evaluate(dataframe);
}

private static void printCorrelation(Dataset<Row> dataframe) {
  Row r1 = Correlation.corr(dataframe, "pcaFeatures").head();
  System.out.printf("\n\nCorrelation Matrix\n");
  Matrix matrix = r1.getAs(0);
  for (int i = 0; i < matrix.numRows(); i++) {
    for (int j = 0; j < matrix.numCols(); j++) {
      System.out.printf("%.2f \t", matrix.apply(i, j));
    }
    System.out.println();
  }
}

private static void printConfusionMatrixMetrics(Dataset<Row> dataframe) {
  dataframe = dataframe
    .select(prediction, label)
    .orderBy(prediction)
    .withColumn(label, col(label).cast("double"));

  MulticlassMetrics metrics = new MulticlassMetrics(dataframe);
  System.out.printf("Precision: %.5f \n", metrics.weightedPrecision());
  System.out.printf("Recall: %.5f \n", metrics.weightedRecall());
  System.out.printf("Recall: %.5f \n", metrics.accuracy());
}
```

Met een *group-by* kunnen we een *confusion matrix* opstellen. In de *main*-methode spreken we de functies als volgt aan. Hieronder tonen we de evaluatie bij het logistische regressiemodel. De code is gelijkaardig aan de andere modellen.

```
System.out.printf("Ideale parameters:\nThreshold: %s\nMax Iterations: %s\nk: %s",
  cvLogReg.getEstimatorParamMaps()[0].get(lr.threshold()).toString(),
  cvLogReg.getEstimatorParamMaps()[1].get(lr.maxIter()).toString(),
  cvLogReg.getEstimatorParamMaps()[2].get(pca.k()).toString()
); // beste hyperparameters opvragen

System.out.printf("\n\nLogistische Regressie\n");
printConfusionMatrixMetrics(predictionsLogReg);
System.out.printf("Area ROC curve: %.4f\n", getAreaROCCurve(predictionsLogReg));
predictionsRFC.groupBy(col(label), col(prediction)).count().show(); // confusion matrix
```

Bevindingen

Wij merken op dat zowel de Random Forest als de Lineaire SVM even sterk scoort. Het logistische regressiemodel daarentegen komt net iets te kort. Zo zijn er véél meer *False Positives* vergeleken met de False Negatives. Dit wilt zeggen dat het model veel sneller een transactie goed zal keuren, terwijl het eigenlijk een fraudulente transactie is.

Het logistische regressiemodel is optimaal op een threshold van 75%, hoogstens vijf iteraties en een k van 3. Het lineaire SVM-model is optimaal op hoogstens vijf iteraties met een *reg params* van 0.2 en een k van 3. Het RFC-model is optimaal op een maximale diepte van 3, 60 bomen en een k van 3. Onze eigen inbreng, PCA, heeft een kleine invloed gelaten op het model. Zo is de *precision* met een kleine fractie verhoogd.

Hoofdstuk 4

De inhoud van tweets detecteren met binaire NLP-classificatie.

Probleemstelling

Sociale media is een actueel onderwerp. Dubbelzinnige teksten spelen een grote rol. Voor deze opdracht moeten we, op basis van gegeven Engelstalige Tweets, achterhalen of een Tweet gerelateerd is aan een ramp of niet. De dubbelzinnigheid van een zin is een obstakel voor ons model. Zo kan een *tweet* zeggen *'look at the sky it was ablaze'* terwijl het woord *ablaze* een andere context heeft. In deze zin is er een spreekwoordelijke betekenis. De tekst is ruw. Zo zijn er woorden en symbolen dat ons model kan hinderen. De link naar de competitie kan u hier [1] terugvinden. Wij baseerden ons vooral op de documentatie.

Data ophalen

Deze oefening verschilt niet in aanpak. Wij krijgen de training- en testset mee in CSV-formaat. Dit lezen wij in met het Spark-object. Net zoals bij de regressie-oefening bouwen wij een schema op.

```
private static Dataset<Row> getTraining() {
  List<StructField> fields = Arrays.asList(
    DataTypes.createStructField("id", DataTypes.IntegerType, false),
    DataTypes.createStructField("keyword", DataTypes.StringType, false),
    DataTypes.createStructField("location", DataTypes.StringType, false),
    DataTypes.createStructField("text", DataTypes.StringType, false),
    DataTypes.createStructField("target", DataTypes.DoubleType, false)
  );
  StructType schemaData = DataTypes.createStructType(fields);
  return spark.read()
    .option("header", true)
    .schema(schemaData)
    .csv("src/main/resources/train.csv");
}

private static Dataset<Row> getTest() {
  List<StructField> fields = Arrays.asList(
    DataTypes.createStructField("id", DataTypes.IntegerType, false),
    DataTypes.createStructField("keyword", DataTypes.StringType, false),
    DataTypes.createStructField("location", DataTypes.StringType, false),
    DataTypes.createStructField("text", DataTypes.StringType, false)
  );
  StructType schemaData = DataTypes.createStructType(fields);
  return spark.read()
    .option("header", true)
    .schema(schemaData)
    .csv("src/main/resources/test.csv");
}
```

[1] Link naar competitie: https://www.kaggle.com/c/nlp-getting-started

Data Cleaning

.

Voor we beginnen met teksttransformaties moeten we eerst de *dataset* cleanen.

1. We droppen alle rijen die *null*-waarden bevatten.
2. We voegen een *string-only* kolom toe aan het dataframe. We behouden enkel de alfabetische karakters. We gaan er van uit dat we enkel met het Latijns alfabet te maken hebben. Arabische en Kanji-symbolen worden bijvoorbeeld weggefilterd.
3. De data is licht *skewed*. We willen een 50:50 verhouding hebben. Dit passen we aan met de *createSubSample* methode. Deze is gelijkaardig aan de gelijknamige methode uit de vorige oefening.
4. We splitsen de trainingset in twee: een *trainSetSplit* en een *trainTestSplit*. Ter controle printen we de lengte uit van beide dataframes.

```
train = train.select(col("id"), col("text"), col(label));
train = train.na().drop();
train = train.withColumn("text", regexp_replace(col("text"), "\\d+", ""));

test = test.select(col("id"), col("text"));
test = test.na().drop();
test = test.withColumn("text", regexp_replace(col("text"), "\\d+", ""));

train = createSubSample(train);
train.groupBy(label).count().show();

Dataset<Row>[] datasets = train.randomSplit(verhouding, 42);
Dataset<Row> trainSetSplit = datasets[0];
Dataset<Row> trainTestSplit = datasets[1];

System.out.printf("Lengte trainsetsplit: %s\n", trainSetSplit.count());
System.out.printf("Lengte testsetsplit: %s\n", trainTestSplit.count());
```

Pipeline

We krijgen rauwe tekstdata binnen. Deze bevat niet-alfanumerieke karakters, overbodige spaties en stopwoorden. Deze woorden willen wij mijden in ons model. Om de tekstdata om te zetten naar bruikbare data voor ons model maken wij gebruik van een pipeline. Op het einde van de pipeline komt het classificatiemodel aan bod.

1. We behouden enkel de alfanumerieke karakters. Alles met spaties en symbolen verwerpen we. Hiervoor gebruiken we een *RegexTokenizer* object.
2. Vervolgens willen we de stopwoorden verwijderen. Dit doen we met een *StopWordsRemover*. Als de tweets in een andere taal waren, bijvoorbeeld Nederlands, dan hadden wij eerst de stopwoorden uit die taal moeten opladen. Vervolgens moeten wij die set van stopwoorden koppelen aan het model. Dit hoeven wij niet te doen.
3. De gefilterde woorden moeten we, net zoals aparte featurekolommen bij het regressiemodel, gaan omzetten naar één featurekolom. Dit zal een vector zijn van alle woorden. In dit verslage gebruiken wij twee verschillende transformers: *CountVectorizer* of *HashingTF*. Hieronder ziet u een voorbeeld van een pipeline met HashingTF.
4. Het einde van de rit: een classificatiemodel trainen en testen. Bij deze oefening testen we twee verschillende modellen uit: logistische regressie én een *Random Forest Classifier*. Voor beide classificatiemodellen werken we met crossvalidatie.

Dit is de eerste pipeline. Hier werken we met Logistische regressie, inclusief crossvalidatie. De features worden naar een vector omgezet met *HashingTF*.

```
RegexTokenizer tokenizer = new RegexTokenizer()
  .setInputCol("text")
  .setOutputCol("words")
  .setPattern("\\W");

StopWordsRemover remover = new StopWordsRemover()
  .setInputCol(tokenizer.getOutputCol())
  .setOutputCol("filtered");

HashingTF htf = new HashingTF()
  .setInputCol(remover.getOutputCol())
  .setOutputCol("hashedFeatures");

LogisticRegression lr = new LogisticRegression()
  .setFeaturesCol(hashingTF.getOutputCol())
  .setLabelCol(label)
  .setMaxIter(10).setRegParam(0.001);

ParamMap[] paramGridLogReg = new ParamGridBuilder()
  .addGrid(lr.maxIter(), new int[] { 30, 40, 50})
  .addGrid(lr.threshold(), new double[] { 0.8, 0.85 })
  .build();

CrossValidator cvLogReg = new CrossValidator()
  .setEstimator(lr)
  .setEvaluator(new BinaryClassificationEvaluator().setLabelCol(label))
  .setEstimatorParamMaps(paramGridLogReg);

Pipeline pipelineLogReg = new Pipeline()
  .setStages(new PipelineStage[]
    { tokenizer, remover, htf, cvLogReg }
  );
```

Hieronder ziet u ons tweede model: een *RFC*-model dat *CountVectorizer* gebruikt om de features naar één vector om te zetten.

```
RandomForestClassifier rfc = new RandomForestClassifier()
  .setLabelCol(label)
  .setFeaturesCol(vectorizer.getOutputCol()).setSeed(42);

CountVectorizer vectorizer = new CountVectorizer()
  .setInputCol(remover.getOutputCol())
  .setOutputCol("features")
  .setVocabSize(10000)
  .setMinDF(5);

ParamMap[] paramGridRFC = new ParamGridBuilder()
  .addGrid(rfc.maxDepth(), new int[] { 5, 7, 10 })
  .addGrid(rfc.numTrees(), new int[] { 40, 80 })
  .build();

CrossValidator cvRFC = new CrossValidator()
  .setEstimator(rfc)
  .setEvaluator(new BinaryClassificationEvaluator().setLabelCol(label))
  .setEstimatorParamMaps(paramGridRFC).setNumFolds(5);

Pipeline pipelineRFC = new Pipeline()
  .setStages(new PipelineStage[] { tokenizer, remover, vectorizer, cvRFC});
```

Evaluatie

We evalueren het model op basis van de volgende metrieken:

- Nauwkeurigheid
- De *precision* en *recall*.
- *Area under ROC curve*
- De confusion matrix printen we uit met door een *groupby* uit te voeren op het dataframe met de voorspelde waarden.

Net zoals bij de voorbije twee oefeningen printen we de ideale hyperparameters uit.

```
private static double getAreaROCCurve(Dataset<Row> dataframe) {
  return new BinaryClassificationEvaluator().setLabelCol(label).evaluate(dataframe);
}

private static void printConfusionMatrixMetrics(Dataset<Row> dataframe) {
  System.out.println();
  dataframe = dataframe.select(prediction, label).orderBy(prediction)
  .withColumn("target_d", col(label).cast("double")).drop(col(label));

  MulticlassMetrics metrics = new MulticlassMetrics(dataframe);
  System.out.printf("Precision: %.5f \n", metrics.weightedPrecision());
  System.out.printf("Recall: %.5f \n", metrics.weightedRecall());
  System.out.printf("Nauwkeurigheid: %.5f \n", metrics.accuracy());
}

// in main-methode
PipelineModel pipelineModelLogReg = pipelineLogReg.fit(trainSetSplit);

System.out.printf("Ideale parameters:\nMax Iter: %s\nReg params: %s",
  cvLogReg.getEstimatorParamMaps()[0].get(lr.maxIter()).toString(),
  cvLogReg.getEstimatorParamMaps()[1].get(lr.threshold()).toString()
);

Dataset<Row> predictedLogReg = pipelineModelLogReg.transform(trainTestSplit);
printConfusionMatrixEssence(predictedLogReg);
System.out.printf("Area ROC Curve: %.4f\n", getAreaROCCurve(predictedLogReg));

predictedLogReg.groupBy(col(label), col(prediction)).count().show();
```

Bevindingen

Het logistische regressiemodel is optimaal op 30 maximale iteraties en een *reg params* van 0.85. De diepte van het RFC-model moet hoogstens 4 zijn met 30 bomen om een optimaal resultaat te bekomen. Het finetunen deden we door *trial & error*. We vergroten of verkleinen de parameters stelselmatig.

Beide modellen scoren hier even sterk. Noch het verschil tussen de *HashingTF* en *CountVectorizer*, noch het verschil tussen logistische regressie en *Random Forest Classification* valt hier op. Er is een aanvaardbaar evenwicht tussen de *false negatives* en *false positives*.

Hoofdstuk 5

Conclusie

Algemene bevindingen

In het tweede jaar Data Engineering doken wij voor het eerst in de wereld van *machine learning*. Tijdens de lessen werkten wij exclusief met de *Python-library Scikit-Learn*. Spark MLlib was voor onze groep een nieuwe ervaring. Tijdens het maken van deze oefeningen merkten wij vier algemene punten op, waaronder drie gebreken en één voordeel.

De eerste kwaal is het debuggen. De luxe van de Python notebooks hebben wij hier niet meer. Een klein deel van de code uitvoeren, bijvoorbeeld het herladen van een dataset of het aanpassen van een model, is niet mogelijk. Alles moet vanaf nul worden opgestart.

Als tweede punt ontbreekt Spark de nodige aanschouwelijkheid. Data visualiseren is niet mogelijk op een terminal en dit is nadelig om de ruwe data te analyseren. Tijdens de opdrachten gebruikten we de analytische functies van Spark, bijvoorbeeld het tellen van het aantal per klasse, om te kijken of de data *skewed* is. Wij benaderen dit in de regressieoefening met een *case-when*. Daarnaast werkten wij met het gemiddelde en de standaardafwijking om alles buiten een bereik weg te laten. Alles binnen Spark met Java blijft *terminal*-gebaseerd. Er bestaan alternatieven om grafieken te genereren. Na onderzoek kwamen wij uit op *GraphFrames*. Met dit pakket kan je grafieken creëeren binnen Java. Dit is een deelse oplossing, want het is niet even eenvoudig als bij *Matplotlib* met Python.

Toch komen de sterktepunten van Apache ook boven water. Spark en Python Pandas staan op een relatief gelijk niveau. Ze bieden een even krachtige oplossing voor data-analyse en data-manipulatie. Na verder onderzoek blijkt Spark toch een krachtige tool voor *Machine Learning*. Zo is dit pakket ideaal om meerdere statistische berekeningen parallel op gigantische datasets uit te voeren. *Sci-kit Learn* is eerder geschikt voor middelgrote datasets. Dit is iets wat wij als toekomstige *data-engineers* best in het achterhoofd houden.

Conclusie

Met het maken van drie distincte ML-oefeningen hebben wij een beter zicht gekregen op Spark MLlib. Dit pakket is een must wanneer je op de volgende twee vragen positief antwoord:

1. Wil ik zware rekenkrachtige toepassingen draaien op een gigantische hoeveelheid data?
2. Is alle data verspreid over een *distributed system*?

We kunnen besluiten dat Spark Mllib een krachtig en veelzijdig platform is voor dataverwerking en *machine learning*. Het biedt gebruikers de mogelijkheid om grote hoeveelheden gegevens snel en efficiënt te verwerken. Daarnaast biedt het een rijk aanbod aan machine learning-algoritmen en -hulpmiddelen. Hoewel er enkele uitdagingen en beperkingen zijn, het kan een waardevol hulpmiddel zijn wanneer je gegevens op grote schaal wil verwerken en analyseren.