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 data-analyse gericht op grootschalige gegevensverwerking. Deze technologie wordt aangeboden voor Java, Python, R en Scala. Voor dit opleidingsonderdeel werd Java gekozen als programmeertaal. Spark bevat enkele zijtakken dat zich richt op andere aspecten binnen dataverwerking, één daarvan is MLLib. MLLib is een pakket gericht op machine learning met Spark. Om meer kennis te vergaren kregen wij de groepsopdracht om via het platform Kaggle deel te nemen aan machine learning (ML) gerichte competities.

In dit verslag nemen wij u mee in de wereld van ML met gebruik van Spark. Wij willen u met volle plezier enkele concrete casussen tonen die gebruik maken van de verschillende regressie- en classificatiemethoden.

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.

Hoofdstuk 2

Tijd van een taxi-verplaatsing voorspellen met regressie.

Probleemstelling

Als eerste opdracht maakten wij de ”New York City Taxi Trip Duration”wedstrijd. Bij deze competitie krijgen wij twee datasets: een training- en een testset. Het doel is om de totale tijdsduur van een taxirit in seconden te berekenen. Volgende kolommen werden gegeven: het ID van de verkoper, het aantal passagiers in een taxi, het longitude en latitude van de plaats waar de passagier(s) werden opgehaald, de longitude en latitude van de plaats waar de passagiers werden gedropt, de pick-up en drop-off tijd. De data bevat echter foute datatypes en outliers.

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

Data ophalen

De Kaggle-competitie bevat een training- en een testset. Dit in CSV-formaat. We kiezen ervoor om de data in een schema te gebruiken met behulp van StructField. Dit geeft ons een snelheidsvoordeel. Het is even zoeken om dit correct te laten werken; we merken al snel dat de datetime kolommen niet van het datatype 'DateTime' horen te zijn maar wel van 'Timestamp'.

We worden regelmatig geconfronteerd met lege output van de testset dus lossen we dit op door de kolom *dropoffdatetime* uit het schema van de test set te halen. Nu krijgen voor beide sets de kolommen wel correct te zien.

De code om de twee datasets op te halen, inclusief het schema, vindt u terug op de volgende pagina:

```
  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), //
    TimestampType
  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 schema = DataTypes.createStructType(fields);

  Dataset<Row> dataset = spark.read().option("header", true).schema(schema).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

Onze data is nog te ruw en daarom maken wij een clean-functie aan. Dit moeten we veranderen naar een bruikbaar formaat voor het regressiemodel. Om de data te cleanen volgen wij het onderstaande stappenplan:

1. De *pick-up* en *drop-off* datetime wordt meegegeven als datetime-object. Verander de datetime-kolommen *pickuptime* naar *hour* en *day*. Zo behouden we een 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. Hier kijken we naar alle waarden
4. We verwijderen alle rijen die géén passagiers meenemen. Dus alle rijen waar het aantal passagiers gelijk is aan 0.

```
private static Dataset<Row> clean(Dataset<Row> dataset) {
  dataset = dataset.na().drop();

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

  dataset = dataset
    .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));

  double rightOuter = dataset
    .select(avg(col("distance")).plus(stddev(col("distance"))))
    .first()
    .getDouble(0);

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

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

  dataset = dataset.where(col("passenger_count").$greater(0));

  return dataset;
}
```

User-Defined Functions

De coördinaten van het vertrek- en aankomstpunt liggen te dicht bij elkaar. Dit biedt weinig waarde aan het model. Daarom gaan wij de afstand tussen de vertrek- en aankomstcoördinaten berekenen 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*. Hiervoor volgen we, in drie stappen, de wiskundige formule voor *haversine*. Wiskundige tussenstappen, zoals de vierkantswortel of het omzetten naar een radiaan, doen we met de Math-library van Java.

Wij maken gebruik van een *user-defined-function* (UDF) om de kolom te berekenen. De UDF schrijven we in de main-methode.

```
UDF4<Double, Double, Double, Double, Double> haversine = new UDF4<Double, Double, Double,
    Double, Double>() {
  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;

    double lat2 = dropoffLatitude;
    double lon2 = dropoffLongitude;

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

    // Haversine formula om de afstand tussen longitude en latitude te berekenen.
    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;
  }
};
```

Naast een kolom 'distance' voegen wij ook een kolom 'speed' toe. Dit is de snelheid in *miles* per hour. Voor deze berekening maken wij een tweede UDF met de afstand en de totale tijdsduur van een rit als parameter. Na het maken van de kolommen tonen wij, in de terminal, de gemiddelde snelheid en gemiddelde afstand per dag en uur. De verlopen tijd van een taxirit ontbreekt in de testset, dus we gebruiken deze kolom enkel als extra statistiek bij het analyseren van de data. We volgen hetzelfde principe als de *haversine*-formule.

```
UDF2<Double, Double, Double> speed = new UDF2<Double,Double,Double>() {
  public Double call(Double distance, Double time) throws Exception {
    return distance / (time/3600);
  }
};
```

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 omzetten naar een numerieke waarde. 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 zijn verschillend in bereik. Dit heeft een nadelig effect op ons model bij regressie. We moeten de waarden schalen. In Spark zijn er twee manieren: schalen met een *MinMaxScaler* en schalen met een *StandardScaler*. 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*, *Gradient Boost* en een gegeneraliseerd lineair model. 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. Dit model 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("features")
  .setOutputCol("scaledFeatures");

StandardScaler stdScaler = new StandardScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures");

LinearRegression linreg = new LinearRegression()
  .setLabelCol(label)
  .setFeaturesCol("scaledFeatures");

Pipeline pipelineLinReg = new Pipeline()
  .setStages(new PipelineStage[] { indexer, assembler, minmax, linreg });
```

Finetuning

De parameters van alle modellen moeten we finetunen. Dit probleem pakken wij aan door te werken met een ParamGridBuilder. Met deze tool geven wij meerdere waarden mee voor verschillende parameters. 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. Hieronder kiezen we het model dat de beste MAE heeft. Hieronder een voorbeeld van onze Random Forest Regressor (RFR). We geven mee dat we voor de pipeline dezelfde objecten gebruiken als hierboven. Het *tweaken* gebeurt binnenin de pipeline.

```
RandomForestRegressor rfr = new RandomForestRegressor().setLabelCol(label).setFeaturesCol("
    scaledFeatures");

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

ParamMap[] paramGridRFR = new ParamGridBuilder()
  .addGrid(rfr.maxDepth(), new int[] { 10, 20, 30 })
  .addGrid(rfr.numTrees(), new int[] { 40, 60, 80 })
  .build();

RegressionEvaluator regEval = new RegressionEvaluator()
  .setLabelCol(label)
  .setMetricName("mae");

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

CrossValidatorModel cvmRFR = cvRFR
  .fit(datasets[0]);

Dataset<Row> rfrTrain = cvmRFR
  .transform(datasets[1]);

```

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);
PipelineModel model = pipelineLinReg.fit(datasets[0]);
Dataset<Row> trainedLinReg = model.transform(datasets[1]);
```

Evaluatie en controle

Als tussentijdse controle toonden wij de inhoud van de dataframe. Zo controleren we de gemaakte transformaties. Daarnaast printen wij ook de correlatiematrix uit. De matrix geeft ons visueel weer welke features er interessant zijn voor ons model.

```
private static void printCorrelation(Dataset<Row> dataset) {
  Row r1 = Correlation.corr(dataset, "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 correlatiewaarde (R^2). De Kaggle-opdracht vroeg om de *root mean squared log error* of RMSLE mee te geven. Dit lukt echter niet direct binnen Spark. Met een *for*-lus kunnen wij voor iedere gevraagde metriek informatie ophalen.

```
  private static void printRegressionEvaluation(Dataset<Row> predictionsWithLabel) {
  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(predictionsWithLabel);

    System.out.printf("%s: \t%.5f \n", metricType, calc);
  }
}
```

Als laatste evaluatie willen we kijken naar de marge tussen de verschillende voorspelde waarden en de effectieve 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> predictionsWithLabel) {
  predictionsWithLabel = predictionsWithLabel.withColumn("margin",
  abs(col("prediction").minus(col("trip_duration"))));

  Dataset<Row> range = predictionsWithLabel.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();
}
```

Conclusie

De opdracht vergde een stuk denkwerk. Voor wij aan dit regressieprobleem begonnen, probeerden wij enkele methoden en werkwijzen toe op vrij beschikbare datasets op Kaggle. Dit om de syntax en structuur van Spark gewoon te worden. Veel zaken uit de lessen kwamen aan bod tijdens de uitvoering. 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. Dit 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. Een tijdelijke oplossing was om de log te berekenen van de RMSE. Na zoekwerk leidden wij af dat het logaritme van de RMSE berekenenen een incorrecte berekening was.

Hoofdstuk 3

Credit Card fraude achterhalen met binaire classificatie.

Probleemstelling

Deze dataset werd eerder voor een andere competitie gebruikt, maar wij konden geen toegang verkrijgen tot deze competitie. Deze dataset bestaat uit ongeveer 25.000 rijen. We hebben hier twee klassen: niet-fraudulente en fraudulente transacties. We pakken dit aan met een binair classificatiemodel. De dataset kan u hier 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 één notebook, dat u op deze link 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 schema aan te maken.

```
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 moeten we aanpakken met een *clean*-methode. Zo zal ons model een grotere kans hebben op overfitten, want het model neemt aan dat de meeste transacties echt zal zijn. Als remedie maken we een *sub-sample* van de volledige dataframe. We spitsen ons toe op een 50:50 ratio tussen echte en fraudulente aankopen.

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

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. De array geven we mee aan 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 : data.columns()) {
  if (kolom.startsWith("V")) {
    arrFeatures[teller] = kolom;
    teller++;
  }
}
```

Het model bevat 28 features. Als uitprobeersel, en toepassing van een techniek uit de lessen *Business Intelligence*, maken wij in onze pipeline gebruik van PCA. Zo voorzien we *dimensionality reduction* op op onze features en we behouden daarmee enkel de nodige features. Voor de binaire classificatie keken wij naar drie verschillende classificatiemodellen: *Random Forest, Lineaire SVM en logistische regressie*. Het tunen van de parameters doen we met een ParamGridBuilder.

We splitsen de volledige dataset in twee delen: een training -en een testset.

```

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

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

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

LinearSVC svc = new LinearSVC();

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

ParamMap[] paramGridSVM = new ParamGridBuilder()
  .addGrid(svc.maxIter(), new int[] { 5, 10, 15 })
  .addGrid(svc.regParam(), new double[] { 0.1, 0.2, 0.3 })
  .addGrid(pca.k(), new int[] { 3, 6, 9 })
  .build();

CrossValidator cvSVM = new CrossValidator()
  .setEstimator(pipelineSVM)
  .setEvaluator(new BinaryClassificationEvaluator().setLabelCol(label))
  .setEstimatorParamMaps(paramGridSVM);

CrossValidatorModel cvmSVM = cvSVM.fit(trainingSet);

System.out.printf("Beste parameters: %s\n", cvmSVM.bestModel().params().toString());

Dataset<Row> predictionsSVM = cvmSVM.transform(testSet);
```

Evaluatie

Bij het evalueren van het classificatiemodel ging onze voorkeur uit naar de confusionmatrix. Zo hebben wij een zicht op hoe goed ons model de klassen kan voorspellen. Uit de confusion matrix kunnen wij ook de precision en recall berekenen. Deze functies zijn ingebouwd in Spark en dit hoeven wij niet manueel te berekenen. Als laatst berekent onze applicatie de nauwkeurigheid van het model.

Wij merken op dat zowel de Random Forest als de Lineaire SVM even goed scoort. Het logistische regressiemodel daarentegen komt net te kort. Onze eigen inbreng, PCA, heeft een kleine invloed gelaten op het model. Zo is de *precision* met een kleine fractie verhoogd.

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> predictions) {
  return new BinaryClassificationEvaluator().setLabelCol(label).evaluate(predictions);
}

private static void printCorrelation(Dataset<Row> dataset) {
  Row r1 = Correlation.corr(dataset, "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> predictions_and_labels) {
  Dataset<Row> preds_and_labels = predictions_and_labels.select(prediction, label).orderBy(
    prediction)
  .withColumn(label, col(label).cast("double"));

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

Met een group-by kunnen we een eenvoudige *confusion matrix* opstellen. In de main-methode spreken we de functies als volgt aan.

```
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();
```

Hoofdstuk 4

De inhoud van tweets detecteren met binaire NLP-classificatie.

Bron: https://www.kaggle.com/c/nlp-getting-started

Probleemstelling

Sociale media is een actueel onderwerp. Dubbelzinnige zinnen spelen een grote rol op sociale media. Voor deze opdracht moeten we, op basis van gegeven Engelstalige Tweets, achterhalen of een Tweet gaat over 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.

Data ophalen

Deze oefening verschilt niet qua aanpak. Wij krijgen de training -en testset in CSV-formaat. Dit lezen wij in met het Spark-object. Hier geven wij een schema mee.

Data Cleaning

Het *cleanen* van de data voeren we buiten de pipeline uit. We passen zowel de test- als trainingset aan.

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.

```
Dataset<Row> dataset = getTrainingData();
Dataset<Row> testset = getTestData();

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

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

Pipeline

We krijgen rauwe tekstdata binnen. Deze bevat niet-alfanumerieke karakters, overbodige spaties en stopwoorden. Deze woorden willen wij liefst mijden in ons model. Om de tekstdata om te zetten naar bruikbare data voor ons model maken wij gebruik van een pipeline. Het classificatiemodel komt pas ná de pipeline 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. Hiervoor gebruiken we een *CountVectorizer*-object.

We halen zowel de training- als de testset door de pipeline. Zo staan beide datasets klaar om een classificatiemodel op te laten draaien. We bekomen twee dataframes: *convertedTraining* en *convertedTesting*.

```
RegexTokenizer regexTokenizer = new RegexTokenizer()
    .setInputCol("str_only")
    .setOutputCol("words")
    .setPattern("\\W");

StopWordsRemover stopWordsRemover = new StopWordsRemover()
    .setInputCol("words")
    .setOutputCol("filtered");

CountVectorizer countVectorizer = new CountVectorizer()
    .setInputCol("filtered")
    .setOutputCol("features");

Pipeline pipeline_training = new Pipeline()
    .setStages(new PipelineStage[] { regexTokenizer, stopWordsRemover, countVectorizer });

PipelineModel modelTraining = pipeline_training.fit(dataset);
Dataset<Row> convertedTraining = modelTraining.transform(dataset);

PipelineModel modelTesting = pipeline_training.fit(testset);
Dataset<Row> convertedTesting = modelTesting.transform(testset);
```

Als volgende stap laten we een model los op de getransformeerde data. We maken gebruik van een *paramgrid* om de hyperparameters te *finetunen*. Zoals daarnet bouwen we een crossvalidatiemodel op, want zo vinden we de beste combinatie voor een model.

```
System.out.printf("\n\nLog-Reg\n");

LogisticRegression lr = new LogisticRegression()
    .setFeaturesCol("features")
    .setLabelCol(label);

ParamMap[] paramGridLogReg = new ParamGridBuilder()
    .addGrid(lr.maxIter(), new int[] { 400 })
    .addGrid(lr.threshold(), new double[] { 0.7, 0.8, 0.9 })
    .build();

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

CrossValidatorModel cvmLogReg = cvLogReg.fit(datasets[0]);

System.out.println(cvmLogReg.paramMap().toString());
System.out.printf("Beste model: %s\n", "");

Dataset<Row> cvPredictionsLogReg = cvmLogReg.transform(datasets[1]);
printConfusionMatrixEssence(cvPredictionsLogReg);
```

Evaluatie

Het model evalueren doen we, net zoals bij het vorige classificatieprobleem, met een *confusion matrix*.

Hoofdstuk 5

Bevindingen ML met Spark.

In het tweede jaar Data Engineering maakten wij kennis met machinaal leren. Tijdens de lessen werkten wij exclusief met Scikit-Learn. Het verschil tussen Spark Machine Learning en Scikit Machine Learning is dat Spark Machine Learning een framework is voor het bouwen van machine learning modellen die gebruik maken van distributed computing om te kunnen werken met grote hoeveelheden data, terwijl Scikit Machine Learning een open source bibliotheek is die gebruik maakt van Python om machine learning modellen te bouwen die werken met kleinere hoeveelheden data.

Als tweede punt ontbreekt Spark de nodige aanschouwelijkheid. Data visualiseren is niet op een directe manier mogelijk. Tijdens de opdrachten werkten we vooral met de functies van Spark, bijvoorbeeld het tellen van het aantal klassen om te kijken of de data *skewed* is. Outliers achterhalen was een stuk moeilijker, maar ook hiervoor werkten we met het gemiddelde en de standaardafwijking.

Hoofdstuk 6

Conclusie

Voor deze opdracht hebben wij een eerste keer