Machine Learning with pyspark

Module 3 | Chapter 2 | Notebook 5

In this lesson we will use `pyspark` to identify which hard drives are at risk of failure. We'll use the machine learning module `pyspark.ml`. By the end of this lesson you will have:

- Implemented one-hot encoding with `pyspark`
- Performed logistic regression with `pyspark`
- Evaluated your predictions with `pyspark`.

Preparing categorical columns

Scenario: You are an employee in a large data center and are tasked with investigating server hard drive failures. Thanks to the monitoring team's excellent work, you have the error data of the last quarter for all the hard drives that your data center has in operation - roughly 30000.

Your boss is extremely pleased with your latest results of your work. Now they want to use the data to find out whether it's possible to build a model that predicts whether a hard disk will fail in the next quarter or not.

They have already commissioned the data engineering team to clean and filter the data set for you. The data engineering team has stored the compiled data in the file *aggregated_HDD_Data.csv*.

First we'll create a `SparkSession` and store it as `spark`.

In [1]:
```
from pyspark.sql import SparkSession

#connect to Spark
spark = (SparkSession
          .builder
          .appName("ML mit SparkML")
          .getOrCreate())
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/usr/loc
al/spark-3.2.0-bin-hadoop3.2/jars/spark-unsafe_2.12-3.2.0.jar) to constructor java.ni
o.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsaf
e.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective a
ccess operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
Level).
24/05/02 15:12:55 WARN NativeCodeLoader: Unable to load native-hadoop library for you
r platform... using builtin-java classes where applicable
```

Now let's import the data, and specify the parameter `inferSchema=True`. This allows us to get the datatypes of the columns with the `my_spark_df.printSchema()` method.

Run the following Code cell:

In [2]:
```
df = spark.read.csv('aggregated_HDD_Data.csv', header=True, inferSchema=True)

df.printSchema()
```

```
[Stage 1:===================>                                        (1 + 2) / 3]
root
 |-- serial_number: string (nullable = true)
 |-- smart_read_error_rate_mean: double (nullable = true)
 |-- smart_spin_up_time_mean: double (nullable = true)
 |-- smart_start_stop_count_mean: double (nullable = true)
 |-- smart_reallocated_sectors_count_mean: double (nullable = true)
 |-- smart_seek_error_rate_mean: double (nullable = true)
 |-- smart_power_on_hours_count_mean: double (nullable = true)
 |-- smart_spin_up_retries_mean: double (nullable = true)
 |-- smart_power_cycle_count_mean: double (nullable = true)
 |-- smart_temperature_mean: double (nullable = true)
 |-- smart_current_pending_sectors_mean: double (nullable = true)
 |-- smart_off_line_uncorrectable_mean: double (nullable = true)
 |-- smart_udma_crc_error_rate_mean: double (nullable = true)
 |-- failure: integer (nullable = true)
 |-- model: string (nullable = true)
 |-- brand: string (nullable = true)
 |-- days_live: integer (nullable = true)
 |-- smart_read_error_rate_std: double (nullable = true)
 |-- smart_spin_up_time_std: double (nullable = true)
 |-- smart_start_stop_count_std: double (nullable = true)
 |-- smart_reallocated_sectors_count_std: double (nullable = true)
 |-- smart_seek_error_rate_std: double (nullable = true)
 |-- smart_power_on_hours_count_std: double (nullable = true)
 |-- smart_spin_up_retries_std: double (nullable = true)
 |-- smart_power_cycle_count_std: double (nullable = true)
 |-- smart_temperature_std: double (nullable = true)
 |-- smart_current_pending_sectors_std: double (nullable = true)
 |-- smart_off_line_uncorrectable_std: double (nullable = true)
 |-- smart_udma_crc_error_rate_std: double (nullable = true)
```

We can see that the following columns contain *strings*: `'serial_number'`, `'model'`, `'brand'`. These columns are categorical. `'failure'` contains our target variable. The other variables are continuous, and `'days_live'` indicates how many days the disk has been in operation. The remaining columns contain the averages and standard deviations of the "S.M.A.R.T." data.

We now want to one-hot encode the categorical columns `'brand'` and `'model'`. We'll use the machine learning module `'pyspark.ml` to do this. The functions we need for feature engineering are located in the submodule `'pyspark.ml.feature'`. However, this can only work with numerical values.

So we first need to convert the `str` values to numeric values. So we'll use label encoding, which you already implemented with `pandas` in *Preparing Categorical Features (Module 2, Chapter 2)*. But this time we're going to use Spark.

Import `StringIndexer` from `pyspark.ml.feature`. This can encode `str` values as numeric values. Each instance of `StringIndexer` can only encode one column. So we need a different instance each of the columns `'brand'` and `'model'`. We won't use the `'serial_number'` column, because it just identifies each hard disk individually.

When instantiating, we pass the name of the column to be encoded to the `inputCol` parameter and the name of the resulting encoded column to `outputCol`. In `pyspark.ml` there are a lot of *transformers* which work in a very similar way to those in `sklearn`.

In the following code cell we'll import and instantiate `StringIndexer` for the `'brand'` column. We'll fit it to the data with `my_spark_transformer.fit()`. We'll store the fitted `StringIndexer` again. Then we'll add the encoded column to our spark `DataFrame` with `my_spark_transformer.transform()`.

In [3]:
```
from pyspark.ml.feature import StringIndexer

brand_indexer = StringIndexer(inputCol="brand", outputCol="brand_indexed")  # initiali
brand_indexer = brand_indexer.fit(df)  # fit indexer to dataframe
df = brand_indexer.transform(df)  # encode brand
```

Now instantiate `StringIndexer` for the `'model'` column yourself. Fit it to the data. Remember to store the outputs in each case.

In [4]:
```
model_indexer = StringIndexer(inputCol="model", outputCol="model_indexed")  # initiali
model_indexer = model_indexer.fit(df)  # fit indexer to dataframe
df = model_indexer.transform(df)  # encode brand
```

Now display the first 5 values of the columns `['model', 'model_indexed']`.

In [5]:
```
df['model', 'model_indexed','brand_indexed','brand'].show()
```

```
+------------------+------------+------------+-------+
|             model|model_indexed|brand_indexed|  brand|
+------------------+------------+------------+-------+
| TOSHIBA DT01ACA300|        24.0|         4.0|TOSHIBA|
| TOSHIBA DT01ACA300|        24.0|         4.0|TOSHIBA|
| TOSHIBA DT01ACA300|        24.0|         4.0|TOSHIBA|
| TOSHIBA DT01ACA300|        24.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
|TOSHIBA MD04ABA400V|        13.0|         4.0|TOSHIBA|
+------------------+------------+------------+-------+
only showing top 20 rows
```

As you can see, the model names have been converted into numbers. In our case, for example, `'TOSHIBA DT01ACA300'` became `24`.

We no longer need the unencoded columns. We can remove them with `my_spark_df.drop()`, a bit like you would with a `pandas.DataFrame`. However, you can't pass the column names as a list, and you don't need to specify an axis.

Remove the `'brand'` and `'model'` columns. Store the returned value as `df`.

In [6]:
```
df = df.drop('brand','model')
```

Now we have the categorical values as numbers. Now `OneHotEncoder` from `pyspark.ml.feature` can use these to encode each category in a separate column.

We'll initialize this with the following parameters:

```
OneHotEncoder(inputCols=list,   # list with names of categorical columns
              outputCols=list)  # list with names of new columns
```

Import und initialize `OneHotEncoder`. Give `inputCols` a list of the names you used as `outputCol` for the `StringIndexer` and pass `outputCols` the names `['model_onehot', 'brand_onehot']`. Store the result as `encoder`.

Fit `encoder` to `df` and then use it on `df`. Remember to store the outputs in each case. Then remove the columns you used for `inputCols`.

In [7]:
```
from pyspark.ml.feature import OneHotEncoder
```

```
encoder = OneHotEncoder(inputCols=['brand_indexed', 'model_indexed'],   # list with nc
                        outputCols=['brand_onehot', 'model_onehot'])  # list with names of new c
encoder = encoder.fit(df)
df = encoder.transform(df)

df = df.drop('brand_indexed', 'model_indexed')
```

There are still some columns in `df` that we shouldn't use for training. `'failure'` is the target variable, `'day_live'` anticipates the target variable and `'serial_number'` doesn't give us any information.

Unlike `sklearn`, Spark doesn't like to have the feature matrix and the target vector in different DataFrames. By default, Spark wants a column in the `DataFrame` named `'label'` as the target vector. We can rename a column with the `my_spark_df.withColumnRenamed()` method. Run the following cell to rename `'failure'` as `'label'`:

In [8]:
```
# rename target col to label -> spark default for target
df = df.withColumnRenamed("failure", "label")
```

Spark wants to have the features in a single column called `'features'`. Each value in this column consists of a vector of the actual column values.

In the following code cell, we first define the columns that `'features'` should contain. Then we use `VectorAssembler` to merge these columns into one. `VectorAssembler` doesn't have the `my_transformer.fit()` method, only `my_transformer.transform()`.

In [9]:
```
# select all columns that we want to use as features
feature_cols = [col for col in df.columns if col not in ['serial_number', 'days_live',

# import and initialize VectorAssembler
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=feature_cols,
                            outputCol="features")

# Now let us use the transform method to transform our dataset
df = assembler.transform(df)
```

We have not provided you with an extra test data set in this lesson. To be able to test our model with unknown data despite this, we'll divide the data into a training set and a test set. In *Introduction to Natural Language Processing* (Module 2, Chapter 4) you used the `train_test_split()` function from `sklearn.model_selection` to do this.

In `pyspark` you can split the complete record into two sets with the command `my_spark_df.randomSplit()`. You determine the relative size of these partial data sets with the `weights` parameter. Pass this a list with the desired percentages as a `float` (i.e., numbers between `0` and `1`). The first entry regulates the proportion that the training set ends up with. The second controls the proportion of the test data set. So both entries must add up to `1`.

As the name of the method already suggests, a random generator selects which rows of the data set end up in which sub data set. Whenever the random number generator comes into play, we recommended that you set a *random_seed* to keep the random selection reproducible. With `pyspark.ml`, the corresponding parameter is simply called `seed`. This corresponds to the `random_state` parameter, which you know from `sklearn`.

Use `my_spark_df.randomSplit()` to split `df` into two DataFrames with proportions of `0.9` (90%) and `0.1` (10%). Assign the value `42` to `seed`. The output is `tuple` with both the sub DataFrames. Assign `df_train, df_test` as variables for this `tuple`.

In [10]:
```
#from sklearn.model_selection import train_test_split
df_train, df_test = df.randomSplit([0.9, 0.1])
```

Congratulations: You've encoded categorical columns with Spark and seen that Spark also has transformers too, just like `sklearn`! The data is now ready for an initial model.

Logistic regression with `pyspark`

Our dataset has a great deal of working hard disks and very few malfunctions. So the classes are very unbalanced. Just how unbalanced are they exactly?

Use an SQL query to get the number of classes in the `'label'` column of `df_train`. Note that you must first register `df_train` as an SQL table. Call this Table `'train_set'`.

In [11]:
```
df_train.createOrReplaceTempView('train_set')

my_query = """
            SELECT COUNT(label) as count , label
            FROM train_set
            Group by label
            """
spark.sql(my_query).show()
```

```
24/05/02 15:13:03 WARN package: Truncated the string representation of a plan since i
t was too large. This behavior can be adjusted by setting 'spark.sql.debug.maxToStrin
gFields'.
[Stage 9:===================>                                       (1 + 2) / 3]
+-----+-----+
|count|label|
+-----+-----+
|  294|    1|
|29702|    0|
+-----+-----+
```

We only have 296 hard disk failures (`'label'=1`) and 29772 hard disks that did not fail. So only about 1% of the data is in the minority class.

We'll need this result again later. Repeat the query, but do not use `my_spark_df.show()` to display the result. Instead use `my_spark_df.toPandas()` to convert it into a `pandas.DataFrame`. Store this as `df_train_classes`. Make sure that you also include the column `'label'` next to the number of data points in the classes, so `df_train_classes` should have two columns.

In [12]:
```
df_train_classes = spark.sql(my_query).toPandas()
```

You've already learned how to deal with unbalanced target categories in *Module 2, Chapter 3, Unbalanced Target Categories*. To avoid that only the majority class is considered relevant, we have the 3 options:

- *Oversampling* the minority class
- *Undersampling* the majority class
- Give more weight to the incorrect classification of minority data points when training the model

After *undersampling* we would only have 600 data points left, which is relatively few. On the other hand, oversampling would make our training set almost 100% larger. This will significantly increase the computing time and memory requirements. Therefore, in this case the 3rd option is the most promising. It saves us resources and we don't lose any data points.

Unlike in `sklearn`, the class weightings are not automatically done for us. Instead, we need to define a new column containing the weight of each data point.

We calculate the weights using the following formula from the data in the `'label'` column:

\begin{equation*} \mathrm{weight}_\mathrm{class} = \frac{1}{\mathrm{ratio}_\mathrm{class}} = \frac{\mathrm{count}_{all}}{\mathrm{count}_\mathrm{class}} \end{equation*}

To calculate the weights, we need the total number of all data points. You can find this with the `my_spark_df.count()` method. Save the total number of data points in `df_train` as `df_train_count`.

In [13]:
```
df_train_count = df_train.count()

#   297|     1|
# 29674|     0|
```

We've already stored the number of data points per class in `df_train_classes`. So we can continue with the calculation straight away.

Replace the row names of `df_train_classes` with the corresponding class values in `'label'`. Then divide `df_train_count` by the column with the number of data points in `df_train_classes`. In my case that would be the `'count(label)'` column. The result should be a `Series` where the weights for class `0` are at index `0` and the weights for class `1` are at index `1`. Store the `Series` as the variable `weights`.

In [14]:
```
df_train_classes.index = df_train_classes.loc[:, 'label']
weights = df_train_count / df_train_classes.loc[:, 'count']
```

The weights are approximately 1.0 (class `0`) and 103.4 (class `1`).

To save your weights in a new column, run the following code cell. Here we'll create the new `'weights'` column in `df_train` with `df_train.withColumn()`. The values of the new column depend on the values of `'label'`. If the value there is 0, we insert the first entry of `'weights`'. If this is not the case, we insert the second entry.

This allows us to use the function `when()` from `pyspark.sql.functions`. We pas this a condition for the values of a column and a value that is used when the condition is met. With `.otherwise()` we can specify a value if the condition is not met. `when()` then goes through each value in the column, checks the condition and inserts the corresponding value into the new column.

In [15]:
```
from pyspark.sql.functions import when
df_train = df_train.withColumn("weights",
              when(df_train["label"] == 0, weights.loc[0]).otherwise(weights.loc[1]))
```

Now `df_train` contains everything we need to train our model. The `'features'`, the `'labels'` and the `'weights'`.

So we are finally ready to use a model in Spark. We'll use `LogisticRegression` to predict the failures. The classification models are located in `'pyspark.ml.classification`.

Import `LogisticRegression`, instantiate it as `model` and pass it the parameter `weightCol='weights'`. This way it will use our balanced class weighting.

In [16]:
```
from pyspark.ml.classification import LogisticRegression
model  = LogisticRegression(weightCol='weights')
```

Now fit the model to the data by applying the `my_model.fit()` method to `df_train`. Store the fitted model again under `model`. Then generate the predictions for `df_test`. In `pyspark` the method for this is called `my_model.transform()`. Store the output value as `df_test_pred`.

In [17]:
```
#Fitting the model
model = model.fit(df_train)

#Predictions in pyspark
df_test_pred = model.transform(df_test)
```

`df_test_pred` is a `DataFrame` again, which contains the `'prediction'` column as well as the features:

In [18]:
```
df_test_pred.select('prediction').show(10)
```

```
+----------+
|prediction|
+----------+
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
+----------+
only showing top 10 rows
```

How good was our model? Besides `my_model.transform()` we also have the `my_model.evaluate()` method. It generates some metrics that we can use for evaluation. Apply it to `df_test` and store the return value as `pred_summary`.

In [23]:
```
#evaluate
pred_summary = model.evaluate(df_test)
```

For example, `pred_summary` now has the attributes `my_summary.accuracy`, `my_summary.recallByLabel` and `my_summary.areaUnderROC`. They return the corresponding metrics as a number. Print the three metrics.

In [26]:
```
print(pred_summary.accuracy)
print(pred_summary.recallByLabel)
print(pred_summary.areaUnderROC)
```

```
0.9410889616185659
[0.9413357400722022, 0.918918918918919]
0.9614555891631729
```

The values look very good on first glance. We got an accuracy of over 95% and an ROC-AUC-score of 92%. For the *recall* we got 95% and 80%. This means that we identified 95% of functioning hard disks correctly and 80% of faulty ones. Conversely, this means that we incorrectly classify 5% of functioning hard drives as faulty. In fact, only 1% of the hard disks in the data are defective. So with this model we would replace about 5 times as many hard drives as is necessary.

To know whether it's worth it, we would have to balance the cost of a failure and the cost of an unnecessary replacement. Feature engineering and hyperparameter tuning might be able to improve the model even further.

Now you've gained an insight into working with Spark in Python. If your job involves very large data sets that require distributed processing, it may be worthwhile continuing to work with Spark. You can find the documentation for the Spark-Python API here. A lot of the things you have implemented with `sklearn` in the previous modules can also be implemented easily with Spark.

Now close the `SparkSession` again.

In [27]:
```
spark.stop()
```

Congratulations: You've implemented your first machine learning model with `pyspark`! The `pyspark` API has some similarities to `sklearn`, which you already know well. Your boss thanks you for the model and would like to improve it in another project.

Remember:

- You will find everything you need for machine learning with Spark under `pyspark.ml`
- You can implement *label encoding* and *one-hot encoding* with `StringIndexer` and `OneHotEncoder` respectively
- Convert your features to a column of vectors with `VectorAssembler(inputCols, outputCol)`
- You fit a model in `pyspark` with `my_model.fit()` and generate predictions with `my_model.transform()`
- Evaluate your model for a data set with `my_model.evaluate()`

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
