NEXT GEN WORKSHOP CAR DIAGNOSTICS AT BMW.

POWERED BY APACHE SPARK.

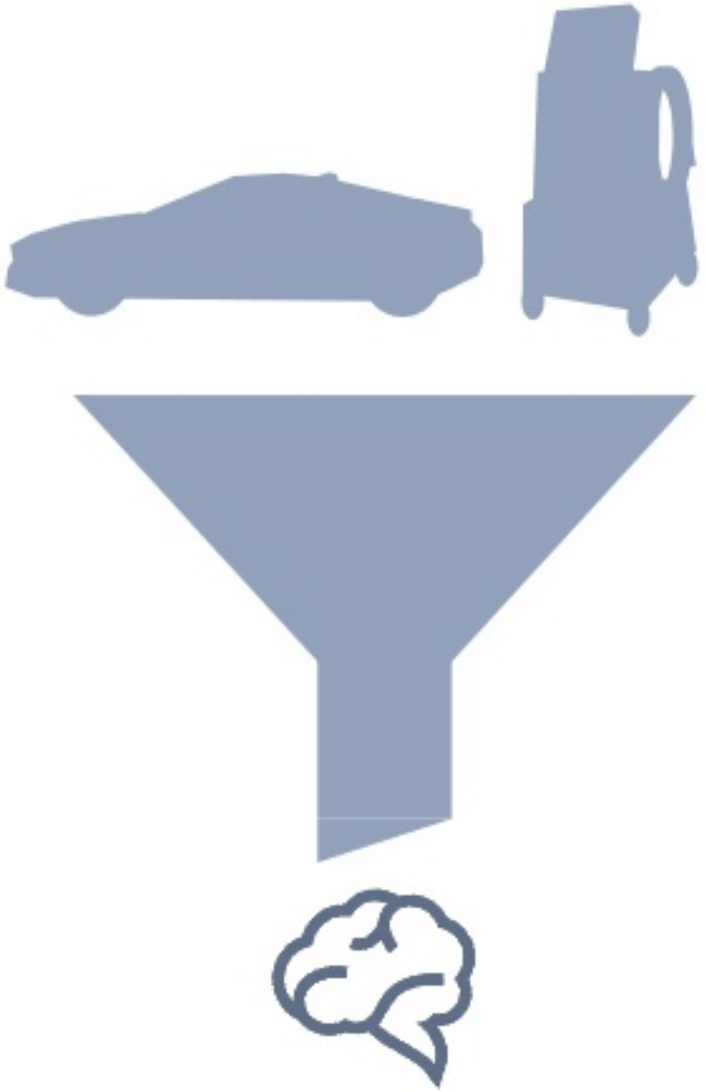
Schlegel, Kaminski | June 7th, 2017

BMW GROUP THE NEXT 100 YEARS

Rolls-Royce Motor Cars Limited

MOTIVATION.
IMPROVE CAR DIAGNOSTICS.

- Provide highly accurate and fast help to customers.
- Potential root causes, that make this hard:
  - Cars are getting more and more complex (hybridization, connectivity).
  - Less experienced workshop staff in evolving markets.
  - Static workshop diagnostics based on manually formalized expert knowledge can‘t cope with the vast number of possibilites.
- Idea: Shift from a manual to a **data driven** approach.

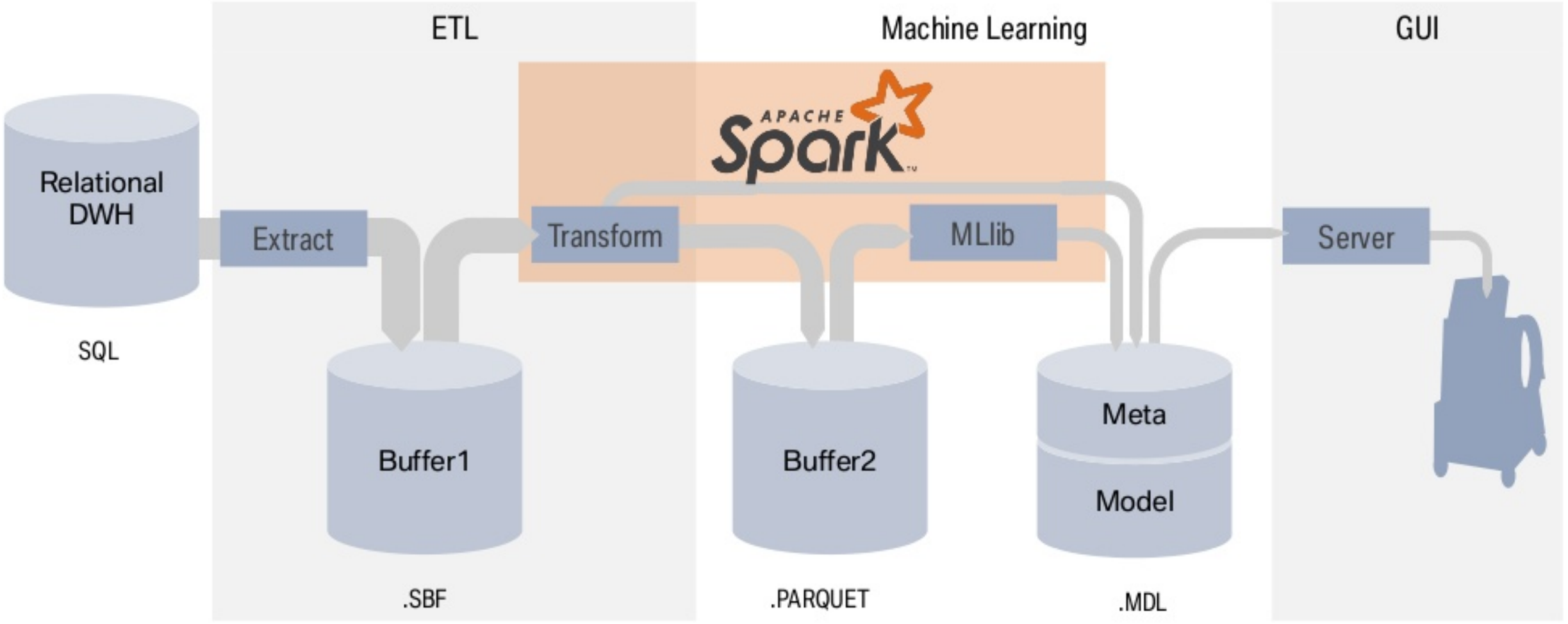


APPROACH.
THE DATASET.

- Input:
  - Diagnostic trouble codes (DTCs).
  - Measurement values.
  - Many more.
  - But: Pipeline is data-agnostic.
- Potential outputs:
  - Parts to switch.
  - Actions to take.
  - Workshop protocols to execute. (Includes parts, actions and checks).
- Data is generated every time a car is „read out“ in the workshop.
- Data needs to be transformed.

ID	KEY	VALUE
1	MV_1	0.15

APPROACH.
SIMPLIFIED OVERVIEW.



APPROACH.
ETL.

1. Extraction using extendedSQL:
   - `-- $ESQL_text("AND MY_COL ") $ESQL_spreadNumArea(1:100:10)` will generate 10 subqueries.
   - Offload computational complexity from the database to our server.
2. Transformation:
   - Feature engineering using UDFs and pipeline stages, e.g.
     - the location of purchase into latitudinal and longitudinal.
     - string indexing of vehicle properties like position of steering wheel.
   - Transformation from readout-key-value into ML suited format:

ID	KEY	VALUE
1	MV_1	0.15
2	MV_1	0.13
2	DTC_2	False
2	SP_1	0

ID	MV_1	DTC_2	SP_1
1	0.15		
2	0.13	False	0

   - This step requires massive amounts of computing power. **Other frameworks fail or are much slower**.

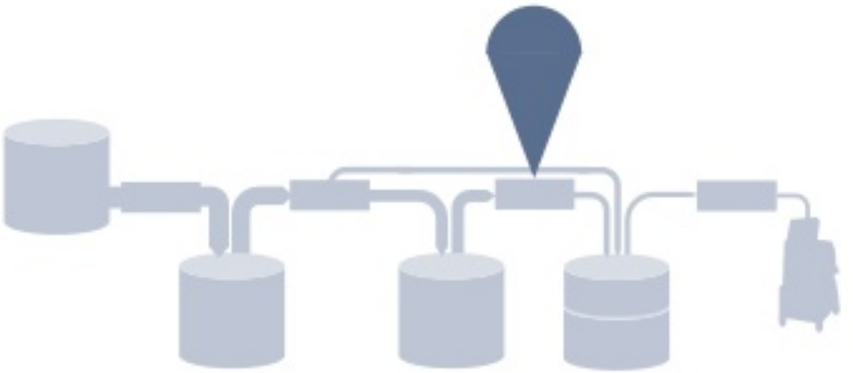
APPROACH.
THE DATASET.

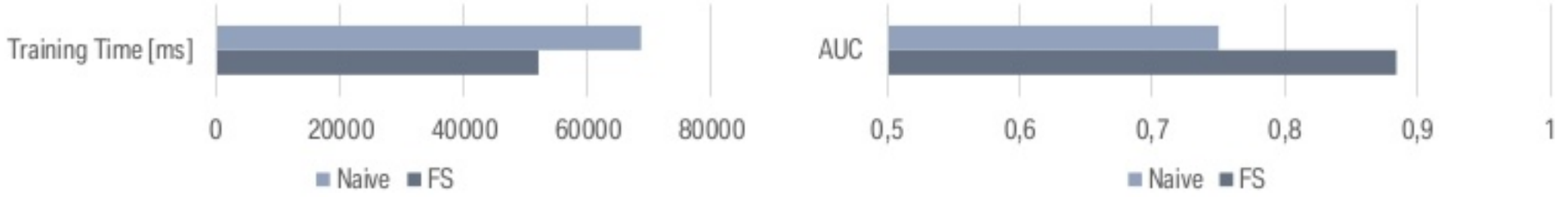
ID	MV_1	MV_2	MV_3	MV_4	SC_IP	SC_1	SC_2	DTC_PU	DTC_1	DTC_2	CP	SP_1
1	0.15	3	20	-0.06	2	77	27	true		true	v.10	0
2	0.13	36	73	-0.01						false	v.10	0
3	0.13	4	16	-0.02		45	1		false	false	v.10	0
4	0.16	14	54	-0.02						false	v.10	0
5	0.15	34	73	-0.07		80	22		false	false	v.10	0
6	0.16	50	33	-0.02		61	93	false	false	false	v.11	0
7		4	27	-0.09		59	91			false	v.10	0
8		48	20	-0.07		32	31			false	v.10	1
9	0.16	60	72	-0.01	1.9	96	53	true	false	true	v.10	0
10	0.11	14	88			73	14			false	v.10	0

- High dimensional featurespace (5000 features +).
- Heterogeneos features.
- High sparsity.
- High class imbalance.
- No „negative“ readouts.

APPROACH.
MODELLING.

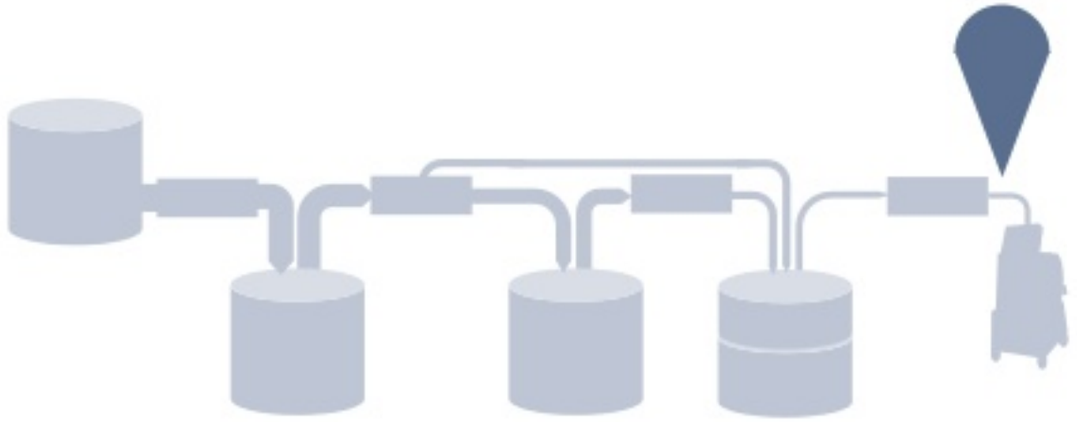
- Modelling itself is pretty straightforward to do with Spark.
- Serializing and loading is the crux of the matter:
  - Model data.
    - Serialization split into two stages:
      - 1st stage: `StringIndexer`, `LabelIndexer`, `VectorAssembler`, `ColumnDropper`, and scaling.
      - 2nd stage: `VectorIndexer`, random forest, and logistic regression.
      - Reasons: Code generations fails (exceeding the 1600k class limit, see PR16648), reusability.
  - Metadata: Additional information to assess model quality (like true/false positive rate, AUC, etc.).
- Feature reduction: Information gain (self), correlation (`mllib.stat.Statistics`), chi² (Spark).

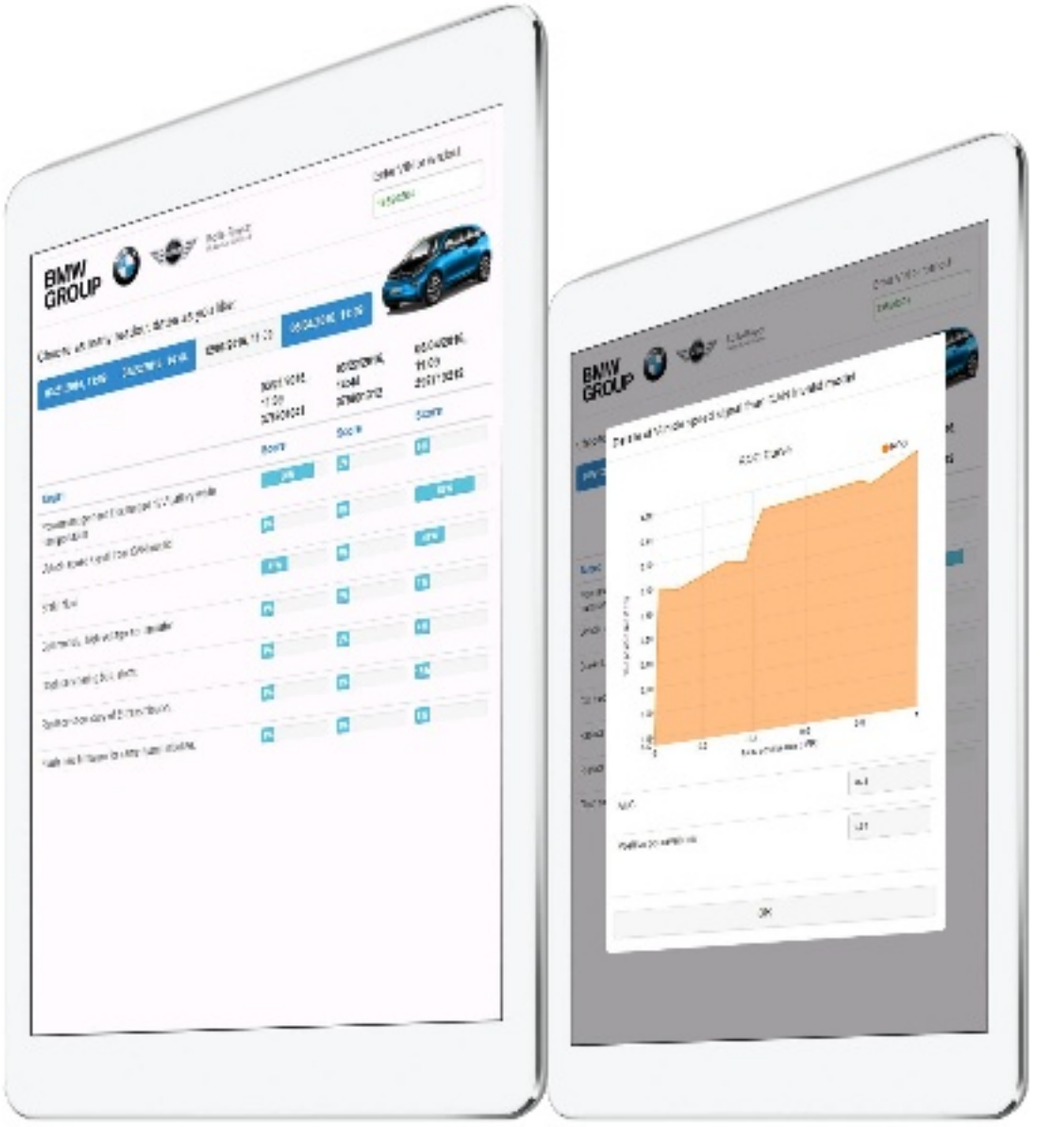




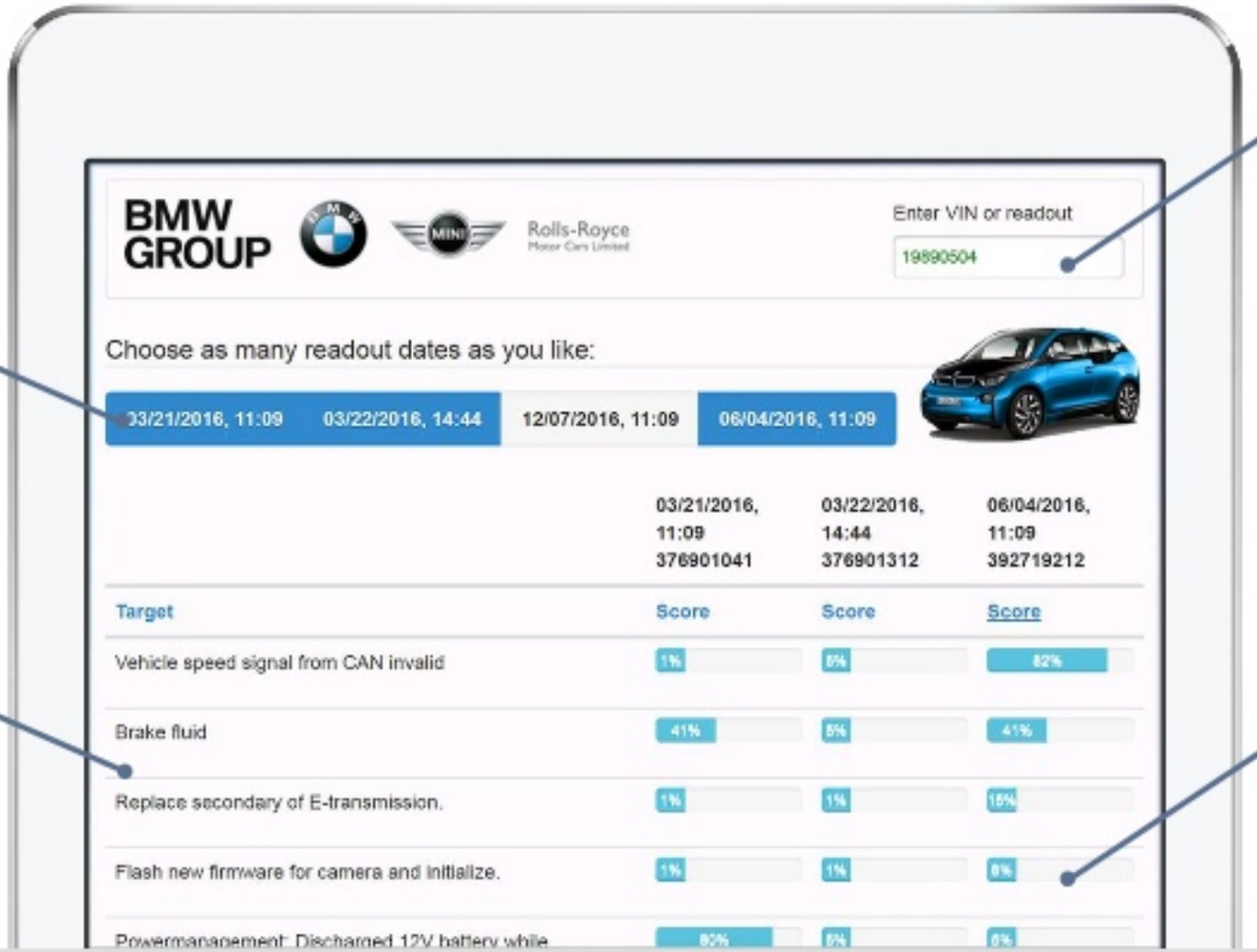
APPROACH.
USER INTERFACE.

- Backend:
  - Scala.
  - Play Framework.
- Frontend:
  - AngularJS.
  - Bootstrap.
- Used by workshop staff and BMW engineers.





APPROACH.
USER INTERFACE DETAIL.



Search for VIN or readout ID

Chose readout IDs to compare

List of potential faults

Ranked by model output

PITFALLS.

- Vast number of columns are hard to take - even for Spark. Feature subset selection / transformation (e.g. PCA), strongly recommended.
- Exact same code runs in shell and using spark-submit but not when being launched from Debugger of IDE (e.g. UDF).
- Watch your logs: Especially high dimensional input data can grow your log file to insane sizes when a stack trace is printed out (2mb data caused over 100GB of log during couple of hours) → Wasting drive space, **harming overall** performance.
- Getting number of partitions right:
  - Partitioning (using `.repartition()`) is absolutely necessary that Spark parallelizes well and is not done automatically.
  - But: Reducing the number of partitions can also increase your performance.
    - If a task takes less than 100ms, use less partitions.

#Partitions: 2
Computation time: ~10%
Total Time: 82ms

#Partitions: 1120
Computation time: ~1%
Total Time: 4000ms

- Toolchain: Ubuntu (less hacky), IntelliJ IDEA (great integration and VCS), Scala (functional paradigm, harmonizes well).

CONCLUSION & OUTLOOK.

- Conclusion:
  - Spark is great for crunching large datasets…
  - … once you know the pitfalls.
- Next steps:
  - Roll out code to larger cluster.
  - Implement more filter methods to reduce the featurespace.
  - Implement techniques to cope with imbalanced dataset (SMOTE, ADAYSN, etc.).
  - Develop strategies to deal with missing values.
  - Evaluate Spark Streaming.

THANKS

FOR YOUR ATTENTION AND TIME.

Bernhard Schlegel
bernhard.bb.schegel@bmw.de