Data Science produktiv

Ein neues datengetriebenes Preisbewertungstool

Arif Wider, Christian Deger

AutoScout24 ist eines der größten Portale für den Kauf und Verkauf von Fahrzeugen in ganz Europa. Dabei verfügt das Portal mit mehr als 2,4 Millionen Angeboten über eine große Menge an aktuellen und historischen Daten hinsichtlich der am Markt erzielten Fahrzeugpreise. Da sich AutoScout24 vom klassischen Anzeigenanbieter hin zu einem allumfassenden Dienstleister rund um den Fahrzeugkauf und -verkauf entwickelt, lag die Idee nahe, die existierenden Fahrzeug- und Preisdaten zu nutzen, um den Kunden ein datengetriebenes Produkt zur Hilfestellung bei Kaufentscheidung und Preisfindung anzubieten.

Vom selbstgehosteten Monolithen zu Cloud-basierten Microservices

Zeitgleich wurde bei AutoScout24 ein groß angelegter Umbau der technischen Infrastruktur vom bisherigen, selbst gehosteten, .NET-basierten und monolithisch geprägten System zu einer Cloud-gehosteten und JVM-basierten Microservice-Architektur begonnen [Tatsu].

Das Ziel dieses Umbaus besteht darin, Innovationen schneller realisieren und live bringen zu können. Dabei zerschneiden wir zunächst den Monolithen in Vertikale, die von einem autonomen Team entwickelt werden und jeweils *eine* Fachlichkeit abdecken. Diese vertikalen Schnitte sind sogenannte Self-Contained Systems [SCS]. Es ist den Teams überlassen, bei Bedarf eine Vertikale weiter in mehrere Services zu zerlegen. Falls die Domäne dem Team noch nicht gut bekannt ist, ist es aber häufig sinnvoll, nicht mit kleinen Services zu starten.

Mit der Neuentwicklung eines Preisbewertungstools ergab sich die Chance, die Fähigkeiten der neuen Microservice-Plattform und die Flexibilität dieser Vorgehensweise zu demonstrieren. So wurde entschieden, das Tool initial als einen vertikalen, mit Amazon Web Services (AWS) betriebenen und mit dem Play-Framework implementierten Microservice umzusetzen. Abbildung 1 zeigt die deutsche Einstiegsseite dieses neu entwickelten Preisbewertungstools.

Der Machine-Learning-Ansatz: Random Forest

Bisher hatte es bei AutoScout24 zwei Preisbewertungstools gegeben: Das eine war zwar automatisiert, nutzte aber nur die aktuellen Bestandsdaten von aktiven Fahrzeugangeboten. Das andere nutzte auch historische Angebotsdaten, war aber nicht voll automatisiert, sondern benötigte noch manuelle Eingriffe, um auf den aktuellen Datenstand gebracht zu werden.

Beide Tools nutzten lineare Regression als Machine-Learning-Ansatz. Diese suggeriert oft eine lineare Beziehung zwischen dem Fahrzeugpreis und bestimmten Faktoren, etwa dem Fahrzeugalter oder dem Kilometerstand. Dies entspricht jedoch häufig nicht der wirklichen Wertentwicklung.

Das neu zu entwickelnde Preisbewertungstool sollte nun historische Preisdaten berücksichtigen, automatisiert aktualisiert werden können und ein realistischeres, besser auf das Problem und die Datenbasis zugeschnittenes Preisvorhersageverfahren verwenden.

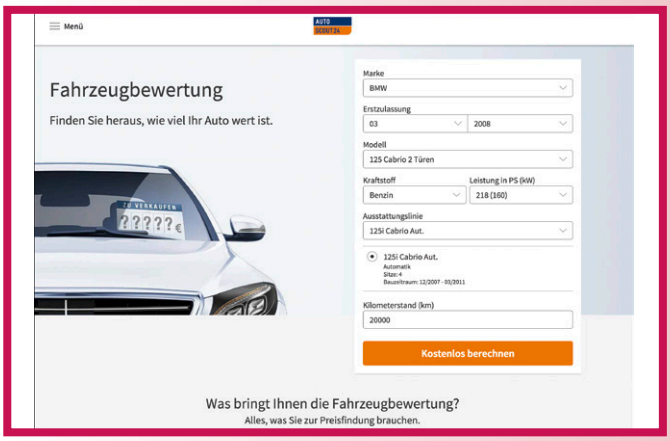

Abb. 1: Einstiegsseite des neuen Preisbewertungstools

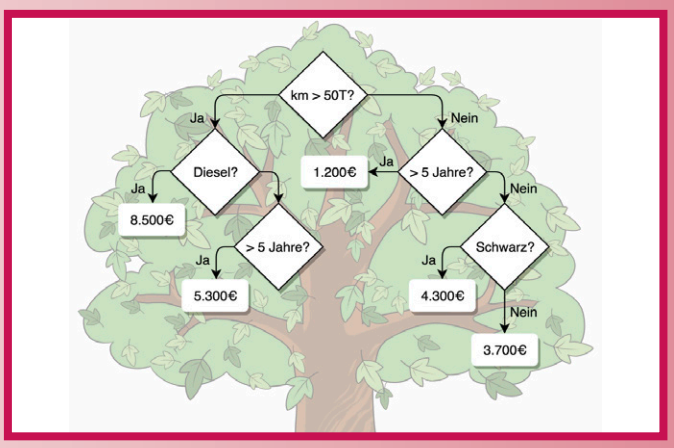

Abb. 2: Ein exemplarischer Entscheidungsbaum zur Preisbewertung

Nachdem verschiedene Machine-Learning-Ansätze von unserem Data Science Team evaluiert wurden, stellte sich *Random Forest* als das Verfahren heraus, welches die mit Abstand besten Ergebnisse für die gegebene Fragestellung erzielte. Random Forest ist ein supervisierter und auf Entscheidungsbäumen basierter Machine-Learning-Ansatz, der im Vergleich zu anderen Entscheidungsbaum-basierten Ansätzen effektiv einem Overfitting entgegenwirkt: Dadurch dass viele Entscheidungsbäume aufgrund zufällig gewählter Teildatensätze generiert werden, verringert sich die Wahrscheinlichkeit, dass das produzierte Vorhersagemodell ausschließlich auf dem zum Lernen verwendeten Datensatz gute Ergebnisse liefert [WikiRF]. Abbildung 2 veranschaulicht exemplarisch, wie ein einzelner Entscheidungsbaum in dem generierten Vorhersagemodell strukturiert ist.

Mittels der auf Datenanalysen zugeschnittenen Programmiersprache *R* und in R verfügbaren Random-Forest-Bibliotheken entwickelte unser Data Science Team eine erste Version eines Modellberechnungsskripts, welches automatisiert die Rohfahrzeugdaten der letzten Jahre aus unserer Datenbasis verarbeitet, diese bereinigt und daraufhin ein Preisvorhersagemodell erzeugt.

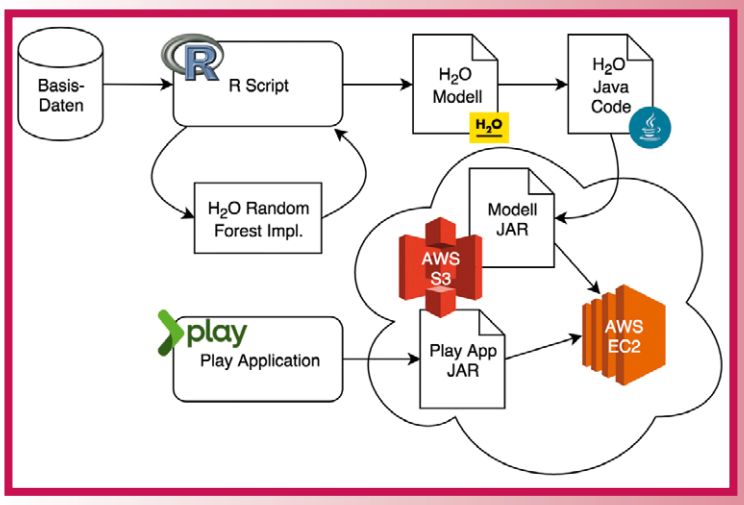

Abb. 3: Erzeugung und Deployment des Preisvorhersagemodells

Vom Preisvorhersagemodell zum Preisbewertungsprodukt

Die Anforderungen, die an dieses Preismodell gestellt werden, unterscheiden sich maßgeblich von denen, die an ein Preisbewertungsprodukt im Live-Betrieb gestellt werden. Während das Preismodell eine akkurate Preisvorhersage liefern muss, sollte das entsprechende Produkt darüber hinaus kurze Antwortzeiten, eine hohe Verfügbarkeit und gute Skalierbarkeit hinsichtlich hoher Nutzerzahlen bieten. Außerdem sollten mittelfristig mehrere Länder und Nutzersegmente unterstützt werden können.

Da das Preismodell immer auch die aktuelle Marktlage abbilden und auch grundsätzlich kontinuierlich verbessert werden sollte, müssen Modellverbesserungen schnell in das Preisprodukt integriert werden können, das heißt, die Cycle Time sollte hier möglichst kurz sein. Dies bedeutet, dass das in R entwickelte Preismodell möglichst eins zu eins und vor allem automatisiert in das produktive System überführt werden können muss.

Eine naheliegende Lösung wäre, das Preismodell direkt in einer R Runtime über einen entsprechenden Service zur Verfügung zu stellen, welcher dann von einer Play-Frontend-Anwendung über ein REST-API aufgerufen wird. Leider ist zumindest die Open-Source-Variante der R Runtime nicht multithreading fähig, was die Skalierung bezüglich vieler paralleler Nutzeranfragen massiv erschwert. Darüber hinaus würde diese Lösung beinhalten, dass man sich mit dem Autoscaling von R-Runtime-Instanzen, deren Speichermanagement und deren Failure-Handling auseinandersetzen müsste.

Wir entschieden uns daher mit der Verwendung von H_2O [H2O] für eine alternative Herangehensweise. H_2O ist eine quelloffene Java/Scala-basierte Predictive Analytics Engine, die eine gute Integration mit den Apache-Projekten Hadoop und Spark bietet, aber auch Anbindungen für andere im Big-Data-Umfeld populäre Programmiersprachen wie etwa Python und R anbietet. Da H_2O ebenfalls eine Random-Forest-Implementierung anbietet, konnten wir ein Random-Forest-basiertes Preisvorhersagemodell mittels H_2O berechnen. Ein solches Vorhersagemodell kann nun mit der H_2O Engine in einem Cluster ausgeführt und über ein API angesprochen werden.

Alternativ bietet H_2O die Möglichkeit, ein mit H_2O berechnetes Vorhersagemodell komplett in Java-Quellcode zu übersetzen. Im Fall unserer Random-Forest-Preismodelle haben die daraus kompilierten JAR-Dateien allerdings mit mehreren Gigabyte pro Land eine erhebliche Größe. Dies liegt daran, dass die darin codierten Entscheidungsbäume Logik und Daten des Modells vereinen. Genauigkeit und Größe des Modells hängen zusammen, da beides maßgeblich durch die gewählte Maximaltiefe der Entscheidungsbäume bestimmt wird.

Insgesamt bietet dieser Ansatz aber den großen Vorteil, dass ein solches als JAR vorliegendes Preismodell zusammen mit einer (ebenfalls als JAR vorliegenden) Play-Webanwendung auf ein und derselben Amazon-EC2-Maschine und in einer einzigen JVM ausgeführt werden kann. Dies verringert den Wartungsaufwand deutlich, da nur der Speicherverbrauch dieser JVM konfiguriert und überwacht werden muss. So können wir außerdem die ohnehin genutzten Skalierungsmechanismen von JVM (Thread-Pools, Concurrency) und AWS verwenden: Das Play-Framework setzt vollständig auf Javas Unterstützung für non-blocking I/O und integriert dazu passend einen Netty-Webserver; AWS bietet mit Elastic Load Balancern (ELBs) und Autoscaling Groups (ASGs) die Möglichkeit, automatisch lastabhängig neue EC2-Maschinen mit derselben Webanwendung hochzufahren und die Last auf diese Maschinen zu verteilen. Abbildung 3 zeigt, wie ein Preisvorhersagemodell berechnet, übersetzt und zusammen mit der Webanwendung deployt wird.

Der dargestellte Ansatz führt außerdem zu Preisberechnungszeiten von nur wenigen Millisekunden, da der aus dem Vorhersagemodell generierte Java-Code fast ausschließlich aus sehr großen if-else-Verzweigungen besteht. Dadurch müssen bei der Berechnung keinerlei Objekte erstellt werden und die Heap-Auslastung bleibt somit konstant niedrig. Auf der anderen Seite erfordert das Laden der ungewöhnlich vielen großen Klassen viel Arbeitsspeicher. Sind diese jedoch erstmal vollständig geladen, verändert sich die Speicherauslastung im Betrieb fast überhaupt nicht mehr, was den Wartungs- und Monitoringaufwand weiter verringert und den Umgang mit Lastveränderungen weiter vereinfacht.

Insgesamt ermöglichten uns die Verwendung von H_2O und die soeben dargestellte Herangehensweise, bereits nach kurzer Zeit mit einem ersten einfachen Preisbewertungsprodukt in einem ersten Land und für ein Nutzersegment produktiv live

zu gehen. Für den Live-Gang in weiteren Ländern entschieden wir uns dann, den Service, welcher die Weboberfläche implementiert, vom Service für die Modellberechnung zu trennen und letzteren als unabhängigen (aber immer noch mit Play implementierten) Webservice mit REST-Schnittstelle zu deployen.

Der Hauptgrund für diesen Service-Split waren vor allem die unterschiedlichen Iterationsgeschwindigkeiten für die Weboberfläche und das Vorhersagemodell: Während sich letzteres bald nur noch selten änderte, wurden Verbesserungen an der Weboberfläche mehrmals täglich ausgerollt. Die Kopplung von Weboberfläche und Preismodellservice führte hier zu unnötig langen Deploymentzeiten. Darüber hinaus ermöglichte die Trennung, einen Preismodellservice pro Land und Nutzersegment zu verwenden und das Vorhersagemodell somit über mehrere Maschinen zu partitionieren. Dass gleichzeitig AWSs Autoscaling eine Replikation der Vorhersagemodelle umsetzt, ermöglichte uns, komplett auf ein Cluster-Datenbanksystem zu verzichten, obwohl die Größe aller Preismodelle zusammen die Speicherkapazität einer einzelnen Maschine weit überschreitet.

Wie geht man mit Gigabytes an Klassendefinitionen um?

Da sämtliche für die Preisbewertung benötigten Daten im Vorhersagemodell enthalten sind, braucht unsere Preisbewertungsanwendung für die Preisberechnung auf keinerlei Datenbank zuzugreifen. Stattdessen müssen allerdings teilweise über zehn Gigabyte an Klassendefinitionen in den Arbeitsspeicher geladen werden.

Dies führt zu einem für die JVM recht ungewöhnlichen Nutzungsszenario: Es wird fast überhaupt kein Speicherplatz für den Heap benötigt, aber umso mehr für die Class-Metadaten. Letztere werden seit Java 8 im Metaspace verwaltet und können damit, falls nötig, sämtlichen zur Verfügung stehenden Hauptspeicher verwenden. Dadurch ist für dieses Szenario in Java 8 keine spezielle Speicherkonfiguration mehr notwendig; mit Java 7 mussten wir noch die PermGen-Größe entsprechend anpassen.

Es stellte sich als vorteilhaft heraus, das JIT-Kompilieren aller Modell-Klassen und auch derer Vorhersagemethoden früh aktiv zu erzwingen, da dies einige Minuten in Anspruch nehmen kann und nicht erst beim Abarbeiten erster User-Anfragen durchgeführt werden kann. Durch diesen Warm-up wird bereits während des Service-Start-ups die maximale Speicherauslastung klar, sodass ein möglicher Speichermangel früh erkannt werden kann.

Beim Wechsel von Java 7 zu Java 8 zeigte sich jedoch, dass das seit Java 8 standardmäßig eingeschaltete Feature der Tiered Compilation [JSE7HotSpot], welches eigentlich die Start-up-Zeiten von Serveranwendungen beschleunigen soll, die Zeit für das initiale Klassenladen und deren Kompilierung um mehr als das Zehnfache erhöhte, sodass wir dies ab Java 8 explizit durch das Setzen der Option `-XX:-TieredCompilation` ausschalten mussten.

Darüber hinaus war auch die Standardeinstellung für die Garbage Collection (GC) nicht gut für unser spezielles Nutzungsszenario geeignet. Ohne spezielle Einstellungen lag die Mark-Sweep-Zeit einer GC-Ausführung weit über eine Sekunde, was im Produktivbetrieb natürlich nicht tolerierbar ist, insbesondere, weil sich die Antwortzeiten ansonsten im kleinstelligen Millisekundenbereich bewegen.

Eine Umstellung auf den erst seit Java 8 Update 40 für den Produktiveinsatz freigegebenen Garbage-First GC [G1] brachte Abhilfe, unter anderem da hier die maximale Dauer für eine GC-Ausführung konfiguriert werden kann, auch wenn dadurch der GC entsprechend häufiger ausgeführt wird. Konkret erhöhte bei uns die Umstellung von ParallelGC zu G1 zwar die Häufigkeit der GC-Ausführungen um den Faktor vier, verringerte aber die durchschnittliche Dauer einer GC-Ausführung fast um den Faktor 100.

Noch mehr Speicher als für das Laden der Klassen wird jedoch für das Kompilieren der teilweise über 30 GB großen Java-Quellcode-Pakete benötigt. Hier mussten wir auf die leistungsfähigste verfügbare EC2-Instanzkonfiguration `c4.8xlarge` mit 64 GB Hauptspeicher und 36 CPU-Kernen zurückgreifen.

Data Science und Continuous Delivery

Bisher wurden bei AutoScout24 Data-Science-Methoden hauptsächlich genutzt, um unternehmensintern Entscheidungen zu leiten. Der Ansatz, ein auf Nutzerdaten basierendes Vorhersagemodell praktisch eins zu eins in den Produktivbetrieb mit vielen User-Anfragen zu bringen, stellte daher unser Data Science Team vor einige Herausforderungen. Bisher arbeiteten diese hauptsächlich explorativ, um Antworten auf Business-Fragen zu finden. Nun musste dauerhaft die korrekte Funktionsweise des Vorhersagemodells bei dessen kontinuierlicher Integration in den Live-Betrieb sichergestellt werden.

Konzepte, die im Software-Engineering für solche Anforderungen entwickelt wurden, wie etwa Continuous Delivery, Test-Driven Development und Consumer-Driven Contracts, sind im Data-Science-Betrieb bisher wenig verbreitet. Daher passierte es anfangs beispielsweise häufig, dass sich die Schnittstelle des Vorhersagemodells ohne Vorwarnung änderte – beispielsweise indem Modellparameter umbenannt wurden – und das Modell dadurch nicht mehr automatisiert in den Produktivbetrieb überführt werden konnte.

Wir entwickelten deshalb zusammen mit unserem Data Science Team sowohl Testsuites für das in R implementierte Modellgenerierungsskript als auch Consumer-Driven Contracts (CDCs), die das vom Preisbewertungsservice erwartete

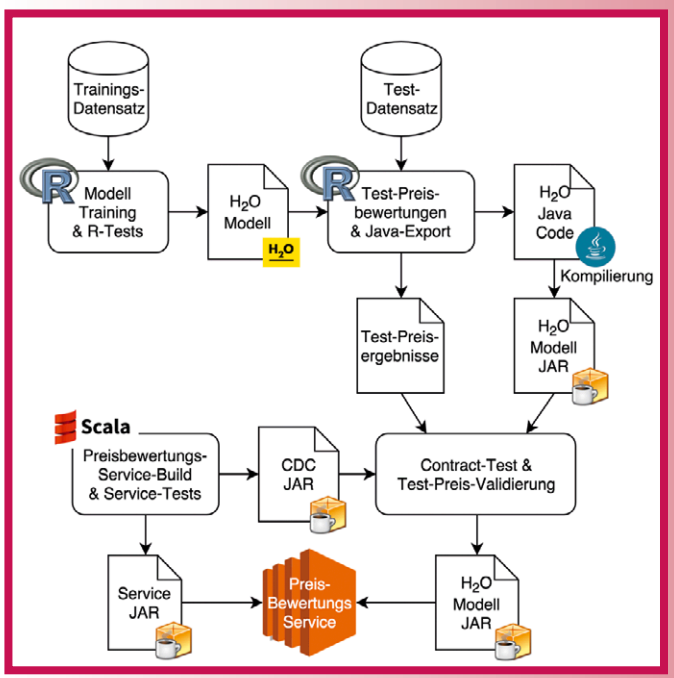

Abb. 4: Test- und Validierungsschritte beim Modelldeployment

Verhalten des Preismodells vor jedem Deployment automatisch sicherstellen. Darüber hinaus führten wir umfangreiche Ende-zu-Ende-Tests ein, die sicherstellen, dass die Webanwendung dieselben Preise ausgibt, wie das ursprünglich generierte Preismodell. Dafür wird zunächst das mit H_2O erzeugte Preismodell mit einer großen Anzahl von Test-Preisbewertungen aufgerufen. Die Eingabedaten für diese Bewertungen stammen dabei aus einem Testdatensatz, welcher nicht für das Training des Preismodells genutzt wurde.

Die Ergebnisse dieser Test-Preisbewertungen dienen zwei Zwecken: Erstens kann durch einen Vergleich mit den tatsächlichen Preisen im Testdatensatz die Modellqualität ermittelt werden. Zweitens können die Ergebnisse mit denen verglichen werden, die man durch ein Aufrufen des zu Java-Bytecode konvertierten Preismodells erhält. Abbildung 4 zeigt die verschiedenen Test- und Validierungsschritte, die während des Modelldeployments durchgeführt werden.

Die dargestellten Maßnahmen erlauben es uns, Modellverbesserungen direkt voll automatisiert in den Produktivbetrieb zu übernehmen, sodass unsere Nutzer unmittelbar von diesen profitieren können.

Literatur und Links

[G1] Oracle, The Garbarge-First Garbage Collector, `http://www.oracle.com/technetwork/ java/javase/tech/g1-intro-jsp-135488.html`

[H2O] `http://www.h2o.ai`

[JSE7HotSpot] Oracle, Java 7 Performance Enhancements, `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`

[SCS] InnoQ, SCS: Self-Contained Systems, 2015, `http://scs-architecture.org`

[Tatsu] `http://inside.autoscout24.com/project_tatsu/2015/01/04/autoscout24-changes-technology-aws-linux-jvm/`

[WikiRF] Wikipedia, Random Forest, 2016, `https://en.wikipedia.org/wiki/Random_forest`

Dr. Arif Wider ist Berater und Entwickler bei ThoughtWorks Deutschland. Dort entwickelt er skalierbare Webanwendungen, unterrichtet Scala und berät zu Big-Data-Themen. Vor seiner Tätigkeit bei ThoughtWorks forschte er zu Themen im Bereich Datensynchronisation, domänenspezifische Sprachen und wie letztere in Scala eingebettet werden können. Er war bei Autoscout24 bereits früh am Technologietransformationsprojekt „Tatsu“ beteiligt, insbesondere an der Migration von .NET/C# zu JVM/Scala. An der Entwicklung des neuen Preisbewertungstools war er von Anfang an maßgeblich beteiligt.
E-Mail: awider@thoughtworks.com

Christian Deger ist Coding Architect bei AutoScout24. Er ist von Anfang an intensiv an dem Projekt „Tatsu“ beteiligt, das die existierende AutoScout24 IT in die nächste Generation von skalierbaren Webplattformen transformiert. Er hat bei AutoScout24 als Softwareentwickler angefangen, wurde später zum Teamleiter, bis ihn aktuelle Herausforderungen in seine heutige technische Rolle zurückgeholt haben. Nach vielen Jahren in der Softwareentwicklung ist Christian Deger immer noch begeistert von technologischen Entwicklungen, neuen Ideen und neuen Konzepten. Seine derzeitigen Schwerpunkte drehen sich um “You build it, you run it”, AWS, Microservices und Event Sourcing.