Abschlussbericht
MLOPS Jahresprojekt

Simon Jakschitz, Marcel Thomas, Alexander Wallner, Vincent Mattes, Kristian Paulic und Glenn Verhaag

Entscheidungen

Eine branch pro Team und Akite.

Um das deployment von pipelines und das Training von models zu vereinfachen, wurde je eine git branch pro Team pro Aktie angelegt. Dies hat den Vorteil, dass die branches verschiedene Konfigurationen besitzen können (LUDWIG config yml files, .env files etc.). Die Aufteilung in git branches sorgte zudem für eine bessere visuelle Übersicht und Nachvollziehbarkeit der Konfigurationen und des Codes. Überlegungen, das branching weiter in einzelne branches pro model pro Aktie pro Team aufzuteilen, wurden verworfen da dies zu unnötiger Komplexität führen würde. Einzelne models und deren Konfiguration sind zudem jederzeit per DVC-Versionierung innerhalb der Aktienbranch nachvollziehbar.

Beschränkung auf drei Aktien pro Team.

Da unsere models nicht den Anspruch besitzen, den gesamten Aktienmarkt abzubilden, wurde eine Beschränkung auf einzelne Aktien eingeführt. Beide Teams trainierten für jede dieser Aktien ein model zur Vorhersage des Schlusskurses am darauffolgenden Tag. Dabei einigten sich die Teams zunächst auf folgende Aktien:

- IBM
- GOOGLE
- APPLE
- TESLA
- SAP
- AMAZON

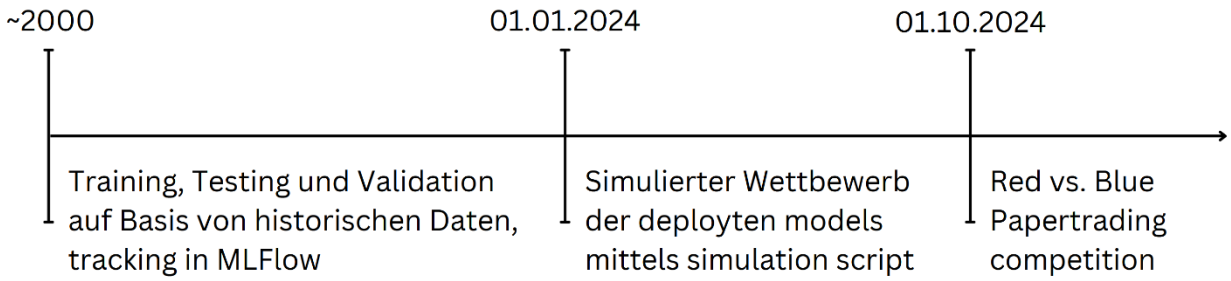
Die jeweiligen models wurden pro Team trainiert und auf dem bereitgestellten Server deployed. Die Menge an individuellen pipelines/laufenden Docker containern führte dabei

schnell zu Problemen. Pipeline steps wurden nur sehr langsam ausgeführt oder scheiterten mit out-of-memory exceptions. Eine Analyse der Probleme zeigte, dass konstant über 80% der CPU-Ressourcen des Servers im Einsatz waren. Die RAM-Nutzung lag konstant zwischen 90% und 95%, im Peak erreichte sie sogar ~100%. Da die Zeit für eine tiefergehende Analyse und eventuelle Optimierung der RAM-Nutzung durch die Pipelines fehlte, entschieden wir uns die Anzahl an models/Aktien zu reduzieren.
Die Teams einigten sich dabei auf folgende Aktien:

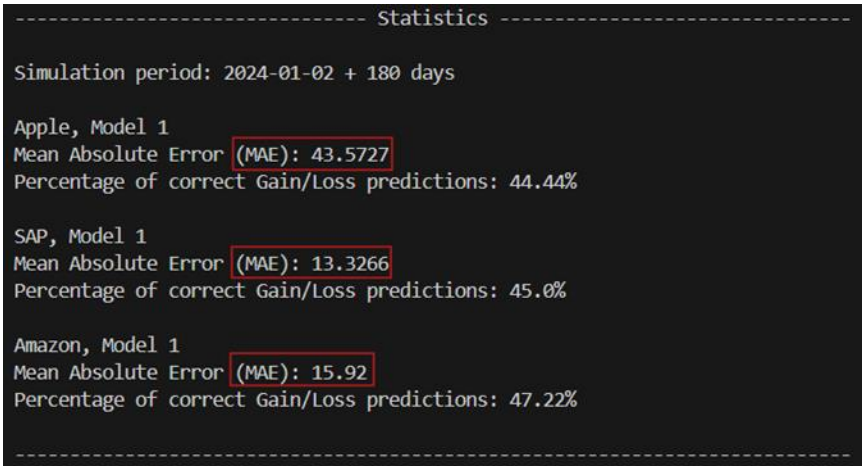
- APPLE
- AMAZON
- SAP

Red vs. Blue – Aufteilung in Simulation Phase und Papertrading Phase.

Um einen strukturierten Ablauf zu ermöglichen und bereits während der Entwicklung neuer Features eine Übersicht der model-performance zu erhalten, wurde der Red vs. Blue Wettbewerb in drei Phasen aufgeteilt:



Beide teams trainierten zunächst Models aus Basis von historischen Daten. Diese reichten zurück bis in die späten 1990er Jahre. Mittels des MLFlow tracking Servers wurden Experimente getrackt und als Basis für das (re-)training genutzt. Danach begann die Simulationsphase, in welcher der tatsächliche Wettbewerb mittels eines selbstentwickelten Python scripts simuliert wurde. Dafür wurden ebenfalls historische Daten, ab dem 01.01.2024, genutzt. Das script testet die performance eines oder mehrerer models pro Aktie und Team. Dafür wird der Mean Absolute Error (MAE), sowie der Anteil an predictions mit korrektem Sentiment betrachtet:



Abschließend folgte der tatsächliche Wettbewerb. Beide teams nutzten die Alpaca trading API um paper trading Käufe/Verkäufe auf Basis der predictions ihrer Models durchzuführen. Auf Grund von unerwarteten Schwierigkeiten und bugs während der Entwicklung der trading Logik, verzögerte sich der Beginn des Wettbewerbs um einige Wochen. Das drei-Phasen-Konzept ermöglichte es den Teams, ihre models zumindest auf den historischen Daten weiterhin zu trainieren und optimieren.

Challenges/Lessons learned

Spezifische Rollen ermöglichen erlauben maximal detailliertes Arbeiten – bieten aber auch Risiken.

Zu Beginn des Projekts wurden jeder Person einzelne Aufgaben bzw. Rollen für das komplette Projekt zugewiesen. Besonders im ersten Semester hatte das den Vorteil, dass wir schnell vorankamen, da die jeweiligen Aufgaben intensiv und zügig bearbeitet werden konnten. Über die Dauer des Projekts kam es jedoch immer wieder zu Problemen, etwa wenn ein Spezialist mal ausgefallen ist. Diese “Islands of Knowledge” haben uns, je länger das Projekt dauerte, immer mehr Aufwand bereitet, da wir uns nicht nur in die Tools und in das Backgroundwissen der anderen Bereiche einlesen mussten, sondern zusätzlich auch noch die erstelle Codebase verstanden werden musste.

Um diese “Islands of Knowledge” zu verhindern, und gleichzeitig die Vorteile der Spezialisierung zu behalten, wäre es nötig, das gewonnene Wissen festzuhalten und am besten auch aktiv zu teilen – in Form von Workshops oder praktischen Beispielen. Alternative könnte man auch Pairprogramming machen, um zumindest 2 Personen für eine spezifische Technologie/Anwendung zu schulen.

Pipelines sollten, sofern kein expliziter Grund dagegenspricht, jeden Schritt immer für eine beliebige Anzahl von Elementen ausführen können, um Skalierung zu ermöglichen.

Klar wird dieses Problem hoffentlich an unserer Pipeline und dem Beispiel des Serving. Aktuell funktioniert das Serving so, dass über eine Umgebungsvariable ein spezifisches Modell angegeben wird sowie ein Port, über den dieses Modell ansprechbar gemacht werden soll. Möchte man mehrere Modelle serven, so muss man nachträglich in der .env-Datei die jeweiligen Werte anpassen, ins Git-Repo pushen, von der deployten Pipeline aus pullen und den Serve-Step erneut ausführen. Das liegt daran, dass wir zu Beginn nicht den Fall beachtet haben, dass es mehr als ein Modell gleichzeitig geben könnte.

Sauberer und langfristig auch deutlich wartungsärmer wäre es gewesen, dem Pipelinestep als Parameter eine Sammlung von Modellkonfigurationen zu geben, die in einer Ausführung alle ausgeführt werden könnten. So könnte die Pipeline beliebig viele Modelle bedienen und müsste nicht x-fach parallel deployed werden.

Finalisieren und Debugging von erster Pipeline vor „Rollout“ auf weitere Pipelines

Versteckte und erst im späteren Verlauf der Pipeline auftretende Bugs im Code wurden erst nach Rollout und Deployment mehrerer nachfolgender Pipelines entdeckt. Dadurch wurde der Aufwand für eine nachträgliche Fehlerbehandlung erheblich erhöht und sämtliche Branches mussten geupdated, sowie entsprechende Pipelines neugestartet werden. Dies bietet ein großes Fehlerpotenzial da gleiche Fehler mehrfach korrigiert werden mussten.

Ein vollständiger Durchlauf der Pipeline unter verschiedenen Testbedingungen hätte Eingangs zwar mehr Aufwand bedeutet, die weitere Entwicklung jedoch einfacher gestaltet.