Versionskontrolle für Machine-Learning-Projekte

Originalartikel

Backup

<html> » Betrieb » Künstliche Intelligenz Veit Schiele 11. August 2020<header></header><div class=„ce-textpic ce-left ce-intext“><div class=„ce-gallery ce-row ce-column“ data-ce-columns=„1“ data-ce-images=„1“><figure class=„image“><figcaption class=„image-caption“>© Adobe: dsdecoret</figcaption></figure> In diesem Artikel erfahren Sie, wie die Modellentwicklung für maschinelles Lernen (ML) systematisch organisiert werden kann. So kann die Leistung eines Modells verbessert werden, wenn die Parameter feiner abgestimmt oder wenn mehr Trainingsdaten verfügbar werden. Um die Verbesserung messen zu können, sollte nachverfolgt werden können, welche Daten für das Training in welcher Modelldefinition und -konfiguration (Parameter etc.) verwendet und welche Modellleistungen damit erzielt wurden. Dabei sollten sowohl die Daten wie auch der zugehörige Programmcode in einer Version erfasst werden. DVC (Data Version Control) wurde entwickelt, um Sie genau bei dieser Aufgabe zu unterstützen [1]. Durch die Implementierung einer DVC-Pipeline werden alle Daten geladen, vorverarbeitet, trainiert und die Leistung bewertet, wobei der Vorgang vollständig reproduzierbar und automatisierbar ist. Trainingsdaten, Modellkonfiguration, das Modell und Leistungsmetriken sind so versioniert, dass Sie bequem zu einer bestimmten Version zurückkehren und alle zugehörigen Konfigurationen und Daten überprüfen können. Außerdem bietet DVC einen Überblick über Metriken für alle Versionen Ihrer Pipeline, mit deren Hilfe Sie die beste Version ermitteln können. Zudem können Sie die Trainingsdaten, Modelle, Leistungsmetriken usw. mit anderen teilen und eine effiziente Zusammenarbeit ermöglichen. <h2>Warum DVC?</h2>
Git-annex speichert wie DVC große Dateien nicht im Git-Repository selbst, sondern in einem lokalen Schlüssel-Wert-Speicher und verwendet Hardlinks oder Symlinks, anstatt Dateien zu duplizieren [2].
Git-LFS verwendet Reflinks oder Hardlinks, um Kopiervorgänge zu vermeiden und so große Dateien effizienter verarbeiten zu können. DVC ist jedoch kompatibel zu deutlich mehr Remote-Speichern (S3, Google Cloud, Azure, SSH usw.) [3].
Andere Workflow-Management-Systeme wie MLflow sind meist sehr allgemein und nicht speziell für die Verwaltung von

Daten in ML-Projekten entwickelt worden [4]. DAGsHub ist ein DVC-Äquivalent, jedoch nur für Github [5]. <h2>Ein Beispielprojekt</h2> Dieser Artikel führt Sie durch ein Beispielprojekt mit folgenden Phasen: Repositories erstellen
Datenpipelines definieren
Reproduzieren
Pipeline visualisieren
Daten teilen <header><h3>1. Repositories erstellen</h3></header> 1. Zunächst wird ein Repository mit einer Versionsverwaltung Ihrer Wahl vorbereitet. In unserem Beispiel ist dies Git, DVC kann jedoch auch mit jeder anderen Versionsverwaltung zusammenarbeiten. <pre>$ git init</pre> 2. In diesem Repository DVC initiieren: <pre>$ dvc initYou can now commit the changes to git.…</pre> 3. Initiales Git-Repository einchecken: <pre>$ git status neue Datei: .dvc/.gitignore neue Datei: .dvc/config$ git add .dvc $ git commit -m „Initial repo“</pre> 4. Daten mit DVC verwalten: <pre>$ mkdir data$ dvc get https://github.com/iterative/dataset-registry get-started/data.xml \ -o data/data.xml$ dvc add data/data.xml</pre> 5. Datenänderungen mit Git synchronisieren: <pre>$ git add data/data.xml.dvc data/.gitignore$ git commit -m „Add raw data“</pre> 6. Entfernten Datenspeicher konfigurieren: Sie können DVC-Daten und -Modelle mit dvc push auch außerhalb Ihres lokalen Caches an einem entfernten Ort speichern, damit sie später auch in anderen Umgebungen abgerufen werden können. Üblicherweise sind dies entfernte Cloud-Services (S3, Azure Blob Storage, Google Cloud Storage), aber auch SSH, HDFS, und HTTP sind möglich. Um das Beispiel möglichst einfach nachvollziehbar zu halten, wählen wir hier einen lokalen Speicherort: <pre>$ sudo mkdir -p /var/dvc-storage$ dvc remote add -d local /var/dvc-storageSetting 'local' as a default remote.$ git commit .dvc/config -m „Configure local remote“[master efaeb84] Configure local remote 1 file changed, 4 insertions(+)</pre> 7. Speichern und teilen: Mit dvc push kopieren Sie Dateien aus Ihrem lokalen DVC-Cache in den zuvor konfigurierten Remote-Storage. <pre>$ dvc push</pre> 8. Überprüfen: Sie können dies überprüfen, z. B. mit: <pre>$ ls -R /var/dvc-storagea3 f1/var/dvc-storage/a3:04afb96060aad90176268345e10355/var/dvc-storage/f1:5a7474cd26c014ce0cf7a8a3d50516.dir</pre> Beachten Sie, dass beide Versionen der Daten gespeichert sind und übereinstimmen sollten mit dvc/cache. <div class=„ce-image ce-left ce-above ce-gallery ce-row ce-column“><figure class=„image“><figcaption class=„image-caption“>Abb. 1: Repositories erstellen. © Cusy GmbH, 2020</figcaption></figure> <header><h3>2. Datenpipelines definieren</h3></header> Die Versionierung großer Daten für Data Science ist ein Schritt in die richtige Richtung, aber noch nicht ausreichend, wenn Daten gefiltert, transformiert oder zum Trainieren von ML-Modellen verwendet werden sollen. Daher erfasst DVC auch die Abfolge der Prozesse bei der Manipulation der Daten. Damit können die Ergebnisse später genau so reproduziert werden, wie sie entstanden sind. Für unser Beispiel soll die Pipeline aus den folgenden fünf Phasen bestehen: Vorbereiten
Aufteilen von Trainings-

und
Testdaten
Merkmalsextraktion
Trainieren
Evaluieren Das Ergebnis dieser Pipeline sind die Leistungsmetriken des trainierten Modells. Das gesamte Schema sieht so aus: Rohdaten → aufteilen → Trainings-/Testdaten → extrahieren → Features → trainieren → Modell → evaluieren → Metriken 2.1. Vorbereiten: Als Vorbereitung auf die Datenpipeline unseres Beispiels benötigen wir zunächst eine virtuelle Python-Umgebung: <pre>$ python3 -m venv venv$ source venv/bin/activate$ pip install -r requirements.txt</pre> In unserem Beispiel enthält die requirements.txt-Datei die vier Python-Pakete pandas, sklearn, scikit-learn und scipy. Beachten Sie jedoch bitte, dass DVC sprachunabhängig und nicht an Python gebunden ist. Niemand kann Sie davon abhalten, die Phasen in Bash, C oder in einer anderen Lieblingssprache oder einem anderen Framework wie Spark, PyTorch usw. zu implementieren. 2.2. Aufteilen von Trainings- und Testdaten: Mit dvc run können Sie einzelne Verarbeitungsphasen erstellen, wobei jede Phase durch eine mit Git verwaltete Quellcode-Datei sowie die Abhängigkeiten und Ausgabedaten beschrieben wird. Alle Phasen zusammen bilden dann die DVC-Pipeline. Unsere erste Phase soll die Daten in Trainings- und Testdaten aufteilen: <pre>$ dvc run -n split -d src/split.py -d data/data.xml -o data/splitted \python src/split.py data/data.xml</pre>
-n split gibt den Namen mit der Beschreibung der Verarbeitungsphase an.
-d src/split.py und -d data/data.xml geben die Abhängigkeiten (dependencies) an. Wenn sich später eine dieser Daten ändert, erkennt DVC, dass die Ergebnisse neu berechnet werden müssen.
-o data/splitted gibt das Verzeichnis an, in das die Ergebnisse geschrieben werden sollen. In unserem Fall sollte sich der Arbeitsbereich geändert haben in:
<pre> ├── data │ ├── data.xml │ ├── data.xml.dvc │ └── splitted+│ ├── test.tsv+│ └── train.tsv+│+├── dvc.lock+├── dvc.yaml ├── requirements.txt └── src └── split.py</pre>
python src/split.py data/data.xml ist der Befehl, der in dieser Verarbeitungsphase ausgeführt wird. Die resultierende dvc.yaml-Datei sieht dann so aus:
<pre> stages: split: cmd: python src/split.py data/data.xml deps: - data/data.xml - src/split.py outs: - data/splitted</pre> In dvc.lock werden hingegen die MD5-Hashwerte gespeichert, anhand derer DVC erkennen kann, ob Änderungen an den Dateien vorgenommen wurden: <pre> split: cmd: python src/split.py data/data.xml deps: - path: data/data.xml md5: a304afb96060aad90176268345e10355 - path: src/split.py md5: ffa32f4104c363040f27d2bd22db127d outs: - path: data/splitted md5: 1ce9051bf386e57c03fe779d476d93e7.dir</pre> Da die Daten im Ausgabeverzeichnis nie mit Git versioniert werden sollten, hat dvc run dies auch bereits in die data/.gitignore-Datei geschrieben: <pre> /data.xml+ /splitted</pre> 2.3. Merkmalsextraktion Die nächste Verarbeitungsphase können Sie nun erstellen, indem die Ausgabe der vorhergehenden als Abhängigkeit definiert wird, in unserem Beispiel mit: <pre>dvc run -n featurize -d src/featurization.py -d data/splitted \ -o data/features python src/featurization.py data/splitted data/features</pre> Sie können diese Verarbeitungsphase jedoch auch parametrisieren. Hierfür erstellen wir in unserem Beispiel die Datei params.yaml mit folgendem Inhalt: <pre> max_features: 6000 ngram_range: lo: 1 hi: 2</pre> Der Aufruf fügt dann dem obigen Befehl noch -p <filename>:<params_list> hinzu: <pre>$ dvc run -n featurize -d src/featurization.py -d data/splitted \ -p params.yaml:max_features,ngram_range.lo,ngram_range.hi -o data/features \ python

src/featurization.py data/splitted data/features</pre> Die dvc.yaml-Datei wird dann ergänzt um: <pre> + featurize: + cmd: python src/featurization.py data/splitted data/features + deps: + - data/splitted + - src/featurization.py + params: + - max_features + - ngram_range.lo + - ngram_range.hi + outs: + - data/features</pre> Schließlich müssen noch dvc.lock, dvc.yaml und data/.gitignore im Git-Repository aktualisiert werden: <pre>$ git add dvc.lock dvc.yaml data/.gitignore</pre> Mit dvc params erhalten Sie weitere Informationen zu den Parametrisierungsoptionen. 2.4. Trainieren Die Trainingsphase wird erstellt mit: <pre>$ dvc run -n train -d src/train.py -d data/features -o model.pkl \ python src/train.py data/features model.pkl</pre> 2.5. Evaluieren <pre>$ dvc run -n evaluate -d src/evaluate.py -d model.pkl -d data/features \ -M auc.json python src/evaluate.py model.pkl data/features auc.json</pre> evaluate.py liest Features aus der features/test.pkl-Datei aus und berechnet den AUC-Wert des Modells. Diese Metrik wird dann in die auc.json-Datei geschrieben. Wir verwenden die -M-Option, um die Datei als Metrik in der dvc.yaml-Datei zu markieren: <pre> + evaluate: cmd: python src/evaluate.py model.pkl data/features auc.json + deps: + - data/features + - model.pkl + - src/evaluate.py + metrics: + - auc.json: + cache: false</pre> Die Ergebnisse können Sie sich anzeigen lassen mit dvc metrics: <pre> $ dvc metrics show auc.json: 0.514172</pre> Um nun unsere erste Version der DVC-Pipeline abzuschließen, fügen wir die Dateien und ein Tag dem Git-Repository hinzu: <pre> $ git add dvc.yaml dvc.lock auc.json $ git commit -m 'Add stage ‹evaluate›' $ git tag -a 0.1.0 -m „Initial pipeline version 0.1.0“</pre><header><h3>3. Reproduzieren</h3></header> Wie Sie sehen konnten, macht DVC das Erstellen einer Pipeline sehr einfach. Der wirkliche Vorteil ist jedoch, dass das Reproduzieren der Ergebnisse oder Teilen davon, d. h. das erneute Ausführen von Stufen ggf. mit geänderten Bedingungen, sehr einfach ist. Hierfür steht Ihnen dvc repro zur Verfügung: <pre> $ dvc repro Verifying data sources in stage: 'data/data.xml.dvc' Stage 'split' didn't change, skipping Stage 'featurize' didn't change, skipping Stage 'train' didn't change, skipping Stage 'evaluate' didn't change, skipping</pre> Sie können nun z. B. Parameter in der params.yaml-Datei ändern und anschließend die Pipeline erneut durchlaufen: <pre>$ dvc reproStage 'data/data.xml.dvc' didn't change, skippingStage 'split' didn't change, skipping Running stage 'featurize' with command: python src/featurization.py data/splitted data/features...Stage 'train' didn't change, skippingStage 'evaluate' didn't change, skippingTo track the changes with git, run: git add dvc.lock</pre> In unserem Fall hatte die Änderung der Parameter also keinen Einfluss auf das Ergebnis. Beachten Sie jedoch, dass DVC Änderungen an Abhängigkeiten und Ausgaben über MD5-Hashwerte erkennt, die in der dvc.lock-Datei gespeichert sind. <header><h3>4. Pipeline visualisieren</h3></header> Damit sich auch für andere schnell die Struktur der Pipeline erschließt, kann diese sehr einfach mit dvc dag visualisiert werden: $ dvc dag
+————————+ | data/data.xml.dvc |
+————————+
       *
   +——-+
| split |
+——-+
     *
+————+ | featurize
|
+————+
**   ** +——-

+   *| train | **+——-+   *   **   **   +———-+ | evaluate | +———-+ <header><h3>5. Daten teilen</h3></header> Sie können nun Ihren Code und Ihre Trainingsdaten einfach mit anderen Teammitlgiedern teilen. Sofern die Teammitglieder ebenfalls auf unser lokales DVC-Repository in /var/dvc-storage zugreifen können, können Sie die Ergebnisse unseres Beispiels reproduzieren mit: <pre> $ git clone https://github.com/veit/dvc-example.git $ cd dvc-example $ dvc pull -TR A data/data.xml 1 file added $ ls data/ data.xml data.xml.dvc</pre><header><h2>Fazit</h2></header> Mit DVC können Sie sprachunabhängig reproduzierbare ML-Pipelines definieren und zusammen mit den zugehörigen Trainingsdaten, Konfigurationen, Leistungsmetriken usw. versioniert speichern. Dabei arbeitet DVC mit allen modernen Versionsverwaltungen zusammen und unterstützt viele verschiedene Speicherarten wie S3, Google Cloud, Azure, SSH usw. Damit strukturiert DVC nicht nur die Datenhaltung, sondern durch einzelne, atomare Phasen der DVC-Pipeline bleiben Änderungen in den Daten auch transparent und nachvollziehbar. Insgesamt erleichtert und effektiviert dies die Arbeit an ML-Projekten erheblich. <div id=„c32318“ class=„frame quellen frame-type-text frame-layout-0“><header> Quellen </header> DVC git-annex Git Large File Storage (LFS) MLflow DAGsHub Github: DVCBeispielprojekt: Github: dvc-example <header> Autor </header><div class=„tx-pwteaser-pi1 main odd“><h3>Veit Schiele</h3>Veit Schiele ist erfahrener Trainer für Analysten, Wissenschaftler und Ingenieure bei der Entwicklung von Analyse- und Forschungssoftware sowie Autor des Jupyter-Tutorial und des PyViz-Tutorial.>> Weiterlesen <header> Das könnte Sie auch interessieren </header> Kommentare (0) </html>