MLOps på Google Cloud Platform
Driftsrapport

Ingvild Andersen, Jon Akselberg Langholm, Erik Flæsen Dalen

19. mai 2021

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfatter
12.05.2021	1.0	Førsteutkast	Erik F.D., Ingvild A. og Jon A. L.
13.05.2021	1.1	Mindre revideringer etter tilbakemelding fra veileder	Erik F.D., Ingvild A. og Jon A. L.
19.05.2021	1.2	Språkvask	Erik F.D., Ingvild A. og Jon A. L.

Innhold

Figurer

1 Innledning

1.1 Dokumentets hensikt

Dokumentet er skrevet i forbindelse med bacheloroppgaven 'MLOps på Google Cloud Platform' i faget IDRI3001, utarbeidet av Ingvild Andersen, Jon Akselberg Langholm og Erik Flæsen Dalen i samarbeid med NTNU og Statens vegvesen.

Bakgrunnen for oppgaven er at Statens vegvesen ønsker å utforske muligheten for å innføre en produksjonspipeline for maskinlæringsmodeller i Google Cloud Platform, samt hvilken løsning som ville vært mest effektiv for formålet. Løsningsforslag skal utarbeides som et såkalt 'Proof of Concept'. Driftrapporten gir leseren en detaljert innføring i den tekniske utførelsen av dette løsningsforslaget, samt problemer som kan oppstå underveis. Hensikten er at oppgavestiller skal kunne reprodusere og få detaljkunnskap om løsningen, samt muligheten til å drifte, implementere og utvikle den videre. Rapporten tar utgangspunkt i at leser allerede har et godt teknisk grunnlag, og har noe kjennskap til Google Cloud Platform.

Driftsrapporten tar utgangspunkt i designrapporten.[29] Løsningen er ikke ferdig testet før denne fasen av prosjektet, og det tas derfor forbehold om løsningsendringer underveis. Dette skjer i samarbeid med bedrift og veileder.

1.2 Avgrensning

Dette avsnittet beskriver oppgavens avgrensning, nærmere bestemt hva løsningen skal og ikke skal omfatte. Statens vegvesen ønsker et «Proof of Concept» på en produksjonspipeline i Google Cloud Platform for maskinlæringsmodeller. Avgrensningen avklarer prosjektgruppens ansvar, samt produksjonspipelinens ønskede funksjonalitet.

Oppgaven omfatter:

- Hente inn data
- Validere data
- Transformere data

- Versjonskontrollere modeller
- Overvåke modeller
- Validere modeller
- Levere modeller og medfølgende prediksjoner

Oppgaven omfatter ikke:

- Utvikling av maskinlæringsmodeller
- Drift av løsningen
- Implementasjon av produksjonsklar løsning
- Funksjonalitet for å hente inn data i sanntid (kontinuerlig trening)
- Opprette bruker eller prosjekt i Google Cloud Platform

Det ble i forstudierapporten gitt forbehold om at ønsket funksjonalitet kunne utelukkes om det skulle vise seg at oppgaven ble for stor. Etter research under designstadiet i oppgaven, viste det seg at det skulle være mulig å implementere all ønsket funksjonalitet. En gjennomgang av hvordan bachelorgruppen har løst alle krav oppdragsgiver hadde, kan finnes i kapittel 6. En avgrensning, eller nærmere avklaring, som ble gjort under designstadiet var at pipelinen kun vil være kompatibel med Tensorflow-modeller, da rammeverket pipelinen er bygget på bruker Tensorflow-biblioteker eksklusivt.

1.3 Oversikt over dokumentet

Første del av dokumentet (1) beskriver dokumentets hensikt (1.1), avgrensning (1.2), oversikt (1.3) og forkortelser og definisjoner (1.4). Videre presenteres leser en gjennomgang av teknologien som løsningen tar i bruk (2). Helt konkret beskrives løsningens overordende stadier (2.1), skyplattformen den kjører i (2.2) og underliggende funksjoner, rammeverk (2.3), Kubernetes (2.4) og orkestrator (2.5). I neste del av rapporten får leseren presentert en overordnet beskrivelse av løsningen (3) og hva som skal implementeres (4). Femte kapittel tar så leser med inn i løsningens implementasjonsfaser (5), som består av oppsett av testmiljø (5.1), utvikling (5.2), implementering i bedriften (5.3) og forslag til videre arbeid for

oppgavestiller (5.4). Til slutt gis det en konklusjon på rapporten i sin helhet (6). Det er også lagt til en appendiks (7) som inneholder kodegjennomgang (A).

1.4 Forkortelser og definisjoner

Se egen ordbok.[30]

2 Kort om teknologi

Følgende avsnitt presenterer en kort gjennomgang av teknologien som løsningen består av. Dette er hentet ut fra designrapporten[29] som ble utarbeidet tidligere i prosjektet, slik at driftsrapporten kan fungere som et enkeltstående dokument for leser. Ønskes det en mer detaljert gjennomgang enn det som presenteres her, så er dette å finne i designrapporten.[29] Prosjektgruppen kom frem til denne teknologien ved å ta utgangspunkt i krav fra Statens vegvesen, samt hensyn til hva som måtte til for å best mulig implementere prinsippene bak MLOps.[29] Disse prinsippene legger opp til automatisering hele veien fra maskinlæringsmodellene utvikles, til de leverer prediksjoner fra et endepunkt. Google Cloud Platform lagrer og kjører løsningen og lagrer dens underliggende data. Denne plattformen innehar en stor mengde maskinlæringsverktøy.

En automatisert maskinlæringspipeline er blant annet avhengig av komponenter, et rammeverk og en orkestrator. Rammeverket konfigurerer pipelinen, og orkestratoren kontrollerer kjøringen av denne. Den styrer og dirigerer rekkefølgen på komponentene og arbeidet som pipelinen består av. I denne løsningen består rammeverket av TensorFlow Extended (TFX), da det kan tilpasses modeller basert på populære TensorFlow. TFX er et ende-til-ende verktøy, og støtter opp under alt fra førprosessering av data til trening og levering av produksjonsklare modeller. Rammeverket støtter også opp under prinsippene bak MLOps. Løsningens orkestrator kalles Kubeflow Pipelines, som er en plattform for å bygge og kjøre ut portable og skalerbare maskinlæringspipeliner basert på Docker-konteinere. Orkestratoren er en ende-til-ende løsning som blant annet legger til rette for gjenbruk av kode, pipeliner og komponenter.

2.1 Overordede stadier

Løsningen skal som nevnt så godt det lar seg gjøre innføre prinsippene i MLOps. Den er bygget opp med utgangspunkt i følgende stadier:

1. Hente ut data
2. Analysere data
3. Forberede data
4. Trening av modellen
5. Evaluering av modellen
6. Validering av modellen
7. Levering av modellen
8. Overvåking av modellen

Videre gjennomgang av teknologiløsninger vil så nært som mulig gjenspeile disse stadiene.

2.2 Plattform: Google Cloud Platform (GCP)

En offentlig skytjeneste som leveres av Google. Tilbyr et stort antall datatjenester, og har spesielt mye å tilby innen maskinlæringsverktøy. Statens vegvesen er allerede kjent med GCP, og ønsket derfor en løsning som er integrerbar med denne plattformen. Det er her man finner prosjektets utviklings-, akseptansetest-, og produksjonsmiljø. For å kunne gjenskape løsningen slik det gjennomgås i dette dokumentet, behøver leseren en bruker hos Google Cloud Platform. Denne brukeren må være godkjent for betaling da det forekommer kostnader ved å kjøre løsningen i GCP.

2.2.1 Prosjekter og tilgangskontroll

For å kunne sette opp denne løsningen behøves et prosjekt i Google Cloud Platform. Slike prosjekter brukes på plattformen for å organisere ressurser i hvert sitt enkeltstående miljø. Det består av et gitt antall brukere, API-er med innstillinger for autentisering og monitorering, samt en oversikt over

prosjektets nåværende og estimerte fremtidige kostnad. Som bruker har man valget om å sette alle sine ressurser i et enkelt prosjekt, eller fordele dem over flere. En større organisasjon bør sikte på sistnevnte, da man kan sikre at brukere kun har tilgang til de prosjektene de er involverte i, slik at de ikke kan ødelegge ikke-relaterte bedriftsressurser om man gjør en feil innad eget prosjekt.[27] Prosjektets tilgangskontroll kalles Identity and Access Management (IAM). Tilgang kan gis til brukere og deres Google-kontoer, eller til Service Accounts. Disse tillater applikasjoner å autentisere seg på tvers av Google Cloud Platform sine ressurser og tjenester.[21]

2.2.2 Storage Buckets

All data som lagres i Google Cloud Platform må ligge i en såkalt *storage bucket*. Dette er konteinere som inneholder brukerens data. Det er ingen begrensning på hvor mange buckets man kan opprette, og man kan sette individuell adgangsbegrensning på hver av dem.[22] I denne gjennomgangen brukes buckets i hovedsak til lagring av maskinlæringsmodellens trenings- og testdata, samt artefakter. Sistnevnte er metadata fra kjøring av TFX-komponentene.

2.2.3 Virtual Private Cloud (VPC) network

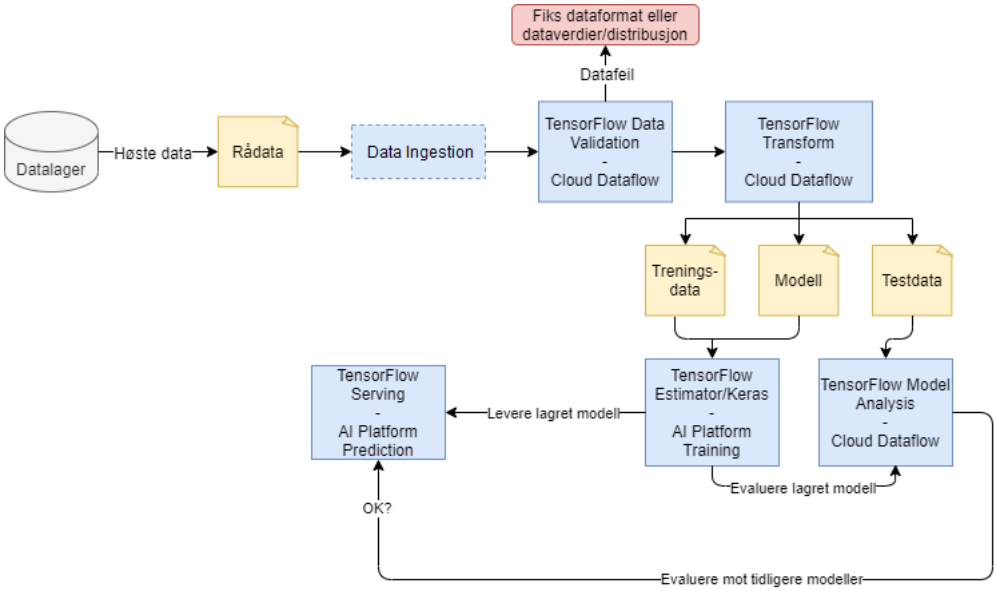
Et VPC-nettverk er et virtuelt privat nettverk som settes opp for at GCP-komponentene som inngår i løsningen skal kunne kommunisere med pipelinen, som ligger på et cluster i Kubernetes Engine. Et prosjekt kan inneha flere VCP-nettverk, og nye prosjekter starter med et standard nettverk som har et subnett i hver region.[23]

2.2.4 Google Cloud API

De ulike tjenestene i Google Cloud Platform har egne API-er som må aktiveres før de kan tas i bruk. GCP tilbyr også REST API-tjenester, som åpner for å kommunisere med GCP fra ønskede programmeringsspråk. Dette brukes blant annet i prosjektet for å opprette nødvendig infrastruktur gjennom et Pythonskript. Alle API-ene er av sikkerhetsmessige årsaker som standard deaktivert ved prosjektopprettelse.

2.3 Rammeverk: TensorFlow Extended (TFX)

TensorFlow Extended (TFX) tilbyr et rammeverk for å opprette pipeliner bestående av TFX-komponenter. Rammeverket støtter alt fra transformasjon av data til trening og levering av produksjonsklare modeller.[18] TFX har flere underliggende biblioteker som støtter opp under ønsket pipeline-funksjonalitet, slik som trening og utkjøring, samt integrasjon med GCP. Tjenestene som inngår i TensorFlow Extended kan legges nært opp mot de overordnede stadiene som er nevnt ovenfor (se figur 2).



Figur 2: TFX: Eksempel på flyt

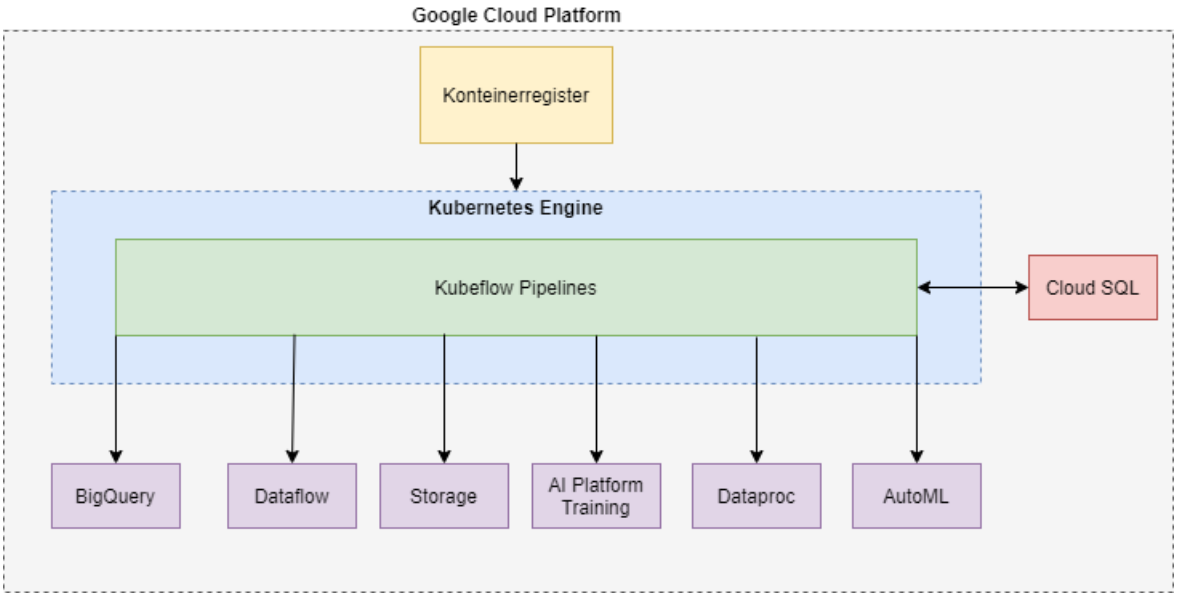
2.4 Kubernetes

Kubernetes er en plattform som kan kjøre ut, skalere og administrere applikasjoner som kjører i konteinere.[24] Videre kan man opprette noe som kalles Kubernetes cluster, som tillater slike konteinere å kjøre på tvers av virtuelle, fysiske og skybaserte miljø og maskiner. I denne løsningen kjøres det ut et cluster på Kubernetes Engine, som er en integrert tjeneste i Google Cloud Platform. Kubernetes Engine lar brukeren utnytte alle mulighetene ved Kubernetes direkte i GCP.[25] Løsningens orkestrator, som beskrives i neste avsnitt, kjører på et slikt cluster.

2.5 Orkestrering: Kubeflow Pipelines (KFP)

Orkestratoren behøves for å koble sammen de ulike komponentene i produksjonspipelinen. Dette er det som sørger for at komponentene kjøres automatisk, i ønsket sekvens, etter planlagte triggere. Kubeflow stammer fra Kubernetes, og har en underliggende tjeneste kalt Kubeflow Pipelines som spesielt er utviklet for maskinlæring. Hver komponent i KFP kan kjøre enten i Kubeflow eller i Google Cloud Platform. Argo Workflows[19] brukes for å orkestrere Kubernetes-ressursene, og KFP har en Python SDK[20] for å definere og manipulere komponentene. KFP kan om ønskelig kjøres tett integrert med maskinlæringstjenestene i GCP.

3 Oversikt over løsning



Figur 3: Kubeflow Pipelines integrert med Google Cloud Platforms ML-verktøy

Teknologiene som er beskrevet i kapittel 2 skal sys sammen for å samarbeide. De individuelle komponentene må også konfigureres, slik at disse til slutt utgjør et system som tilfredsstiller oppdragsgivers behov. Disse er:

- Kunne kjøre ut en maskinlæringsmodell i GCP.
- Kunne kjøre ut nye versjoner av en maskinlæringsmodell i GCP.

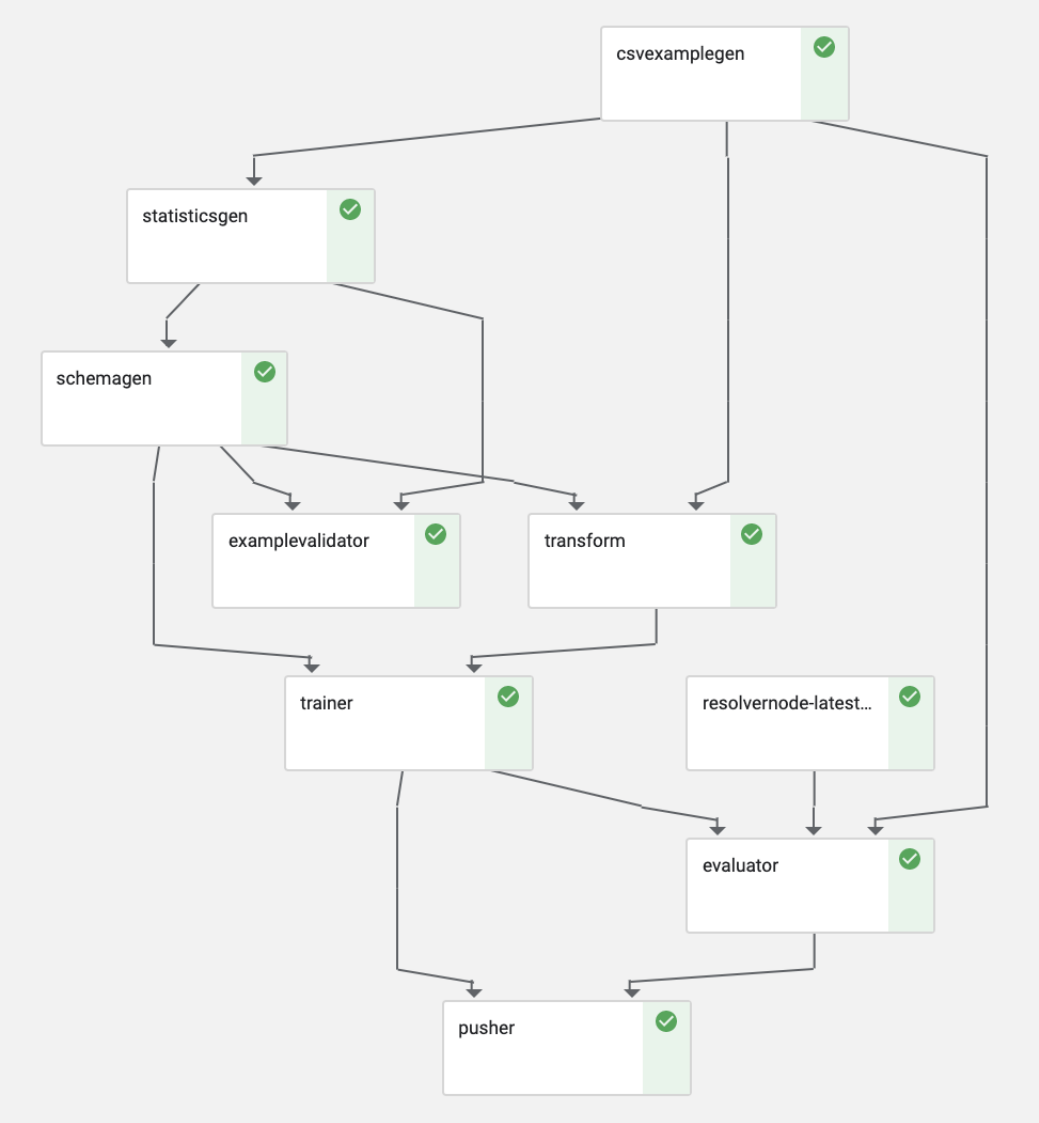
- Kunne eksponere en maskinlæringsmodell for input og bruke den fra andre applikasjoner.
- Finne ut hvordan man kan overvåke en maskinlæringsmodell og dens evne til å predikere riktig.
- Finne ut hvordan man kan skalere en pipeline med hensyn på kapasitet og ytelse.

Figur 3 viser hvordan alle komponentene henger sammen. På øverste nivå ligger GCP med innebygde funksjoner som blir brukt til forskjellige jobber i pipelinen. På GCP kjøres også Kubernetes engine. I Kubernetes engine ligger Kubeflow Pipelines som kjører pipelinen som er definert med Python-kode som bruker TFX-biblioteker. Orkestratoren Kubeflow Pipelines sørger for at de enkelte nodene i pipelinen kjører i riktig rekkefølge. Pilene som peker på de lilla boksene, illustrerer at de individuelle komponentene av pipelinen kjører sine jobber ved hjelp av forskjellige funksjoner som er innebygget i GCP.

Oppgaven som skal løses for å implementere dette systemet er todelt; en del av oppgaven er å sette opp arkitekturen rundt dette systemet som tillater de forskjellige komponentene å samarbeide. Den andre delen av oppgaven er å lage selve produksjonspipelinen som tar i bruk funksjonene i GCP.

Den første delen av oppgaven som går ut på å sette opp arkitekturen i GCP hvis løsning er beskrevet i kapittel A.1 og i vedlagt README.[31]

Del to av oppgaven som omfatter oppsett av selve produksjonspiplinen består av ren Python-kode. Denne koden definerer hvilke jobber som skal gjøres, hvordan disse skal gjøres, og rekkefølgen på dem. Figur 4 viser flyten i denne pipelinen.



Figur 4: TFX-komponentene satt sammen i Kubeflow

Csvexamplegen, eventuelt *bigqueryexamplegen*, tar inndata for pipelinen. *Statisticsgen* tar imot dataen og genererer statistikk. *Schemagen* lager skjemaer for dataen basert på statistikken. *Examplevalidator* sjekker at dataen stemmer overens med skjemaet. *Transform* utfører førprosessering på data før trening. *Trainer* trener en maskinlæringsmodell. *Resolvernode* henter gjeldende modell. *Evaluator* sammenligner ytelsen til den nye modellen med den gamle på data fra *examplegen*. *Pusher* laster eventuelt opp den nye modellen om den tilfredsstiller kriterier stilt av *evaluator*.

En grundigere gjennomgang av flyten i pipelinen, begrunnelse, og forklaring

av de individuelle komponentenes rolle kan finnes i designrapporten.[29]

4 Hva skal implementeres

I bacheloroppgaven skal det implementeres en rekke komponenter som til sammen utgjør en pipeline som tilfredsstiller all funksjonalitet som er beskrevet i kapittel 1.2. Statens vegvesen benytter allerede GCP, så funksjonaliteten skal kjøre der.

Komponenter som må implementeres eller være klare for at pipelinen skal kjøre er følgende:

- En bucket i Google Cloud Storage
- Et VPC-nettverk
- Et Kubernetes-cluster
- Kubeflow Pipelines deployet til det overnevnte Kubernetes-clusteret
- Nødvendige Google Cloud Platform API-er må være aktivert

Etter ønske, kan BigQuery benyttes, men dette er ikke påkrevd for å kjøre pipelinen. Mer om dette i kapittel 5.4. Etter at disse komponentene er klargjorte, kan pipelinens kildekode kjøres og modifiseres.

Kapittel 2.2 har forklart disse forskjellige komponentene nærmere.

5 Implementasjonsfaser

5.1 Oppsett av testmiljø

Å sette opp et testmiljø kan gjøres på to måter - ved bruk av Jupyter Notebooks eller på lokal datamaskin. Det er smak og behag som avgjør hvilken metode man velger, men det anbefales likevel å bruke Jupyter Notebooks, da oppsettet er mest testet der og systemet kommer med nødvendige preinstallerte programmer. Spesielt dersom lokal datamaskin kjører Windows anbefales Jupyter Notebooks, da prosjektgruppen har

arbeidet på de Unix-baserte operativsystemene macOS Big Sur og Linux Debian Testing (Bullseye) når de ikke har brukt Jupyter Notebooks.

For oppsett av testmiljø/PoC henvises det til repositoryets README, som ligger vedlagt.[31]

Skriptet for oppsett av infrastruktur er forklart i appendiks A.1.

5.2 Utvikling

Mesteparten av arbeidet i utføringsfasen ligger her. Etter at alle komponenter lå til rette, kunne arbeidet med å utvikle pipelinen i Jupyter-miljøet starte. Jupyter-miljøet har alle avhengigheter installert fra før av, noe som forenkler arbeidet. Det har i hovedsak blitt utviklet to Python-filer som definerer pipelinen. `pipeline.py` definerer sammenhengen mellom alle komponentene i pipelinen, og `configs.py` inneholder alt av konfigurasjon. I utgangspunktet skal det ikke være nødvendig å endre på noe i `pipeline.py` med mindre man vil heller vil bruke BigQuery fremfor CSV som datainput.

Koden som ble utviklet i denne fasen gjennomgås i appendiks A.2 og A.3.

5.3 Implementering i bedriften

At det ikke er mulig å kopiere hele GCP-miljøer, kompliserer arbeidet med å ta i bruk produktet til bachelorgruppen. Målet med dette dokumentet og med README-filen[31] til github-repositoryet[1] er at bachelorgruppens produkt skal være raskt og enkelt å ta i bruk hos Statens vegvesen. Den eneste forutsetningen er at man bruker Google Cloud Platform.

5.3.1 Tilpasning av pipeline til modell

Pipelinen som er utviklet av bachelorgruppen har blitt utviklet for å være generisk, slik at den potensielt kan tilpasses en hvilken som helst modell. Det er tre ting som må skreddersys til den modellen som skal utvikles i pipelinen. Videre endringer av pipelinen er mulig, men ikke strengt nødvendig. De tre tingene som må være spesifikke for modellen er:

- Transform

- Trainer
- Evaluator

5.3.1.1 Transform

Transform utfører førprosessering på inndataen før den blir brukt videre til trening og evaluering av modellen. Her må dataviter konfigurere slik at de attributter som blir brukt videre kommer med. Her utleder man gjerne også attributter som ikke allerede eksisterer som egne felter i inndataen. Det utvikles en modulfil som inneholder en funksjon som kalles av komponenten når komponenten kjører i pipelinen. Se A.3.0.3 for nøyere beskrivelse av hvordan denne skal konfigureres.

5.3.1.2 Trainer

Trainer er komponenten som utfører selve treningen. Denne må naturligvis også skreddersys til modellen den skal trene. På lik linje med Transform sin konfigurasjonsfil, skal Trainers konfigurasjonsfil også lagres i Google Cloud Storage og inneholde en funksjon som kalles av komponenten når pipelinen kjører. Nærmere instruksjoner er å finne i A.3.0.4.

5.3.1.3 Evaluator

Den siste komponenten som må tilpasses er Evaluator. Konfigurasjonen av Evaluator ligger som et objekt i `configs.py`. Her må man definere hvilke attributter de to modellene skal sammenlignes på, og hvor god en modell må være for å bli deployet. Nærmere forklaring ligger i A.3.0.6

5.4 Videre arbeid

Bacheloroppgaven har fullført et «Proof of Concept» og implementert den funksjonaliteten som Statens vegvesen ønsket seg. Pipelinen har allikevel forbedringspotensiale. Det er funksjonalitet som kan legges til pipelinen, men som ikke har blitt utviklet da gruppen ikke hadde tid til dette, og prioriterte heller å få på plass den funksjonaliteten som ble bedt om fra starten av.

Det første som må gjøres for å ta i bruk pipelinen, forutenom å sette opp infrastrukturen i GCP, er å tilpasse pipelinen til den konkrete modellen som

skal trenes. Kapittel 5.3.1, samt de relevante underkapitlene i A.3 beskriver hvordan dette gjøres.

Etter kommunikasjon med maskinlæringsteamet senere i prosjektet, har det blitt etterspurt muligheten for å kjøre pipelinen lokalt for å enklere kunne teste pipelinen i det den tilpasses til modellen. En `local_runner.py` som gjør dette, ble ferdigstilt av maskinlæringsteamet 23. april 2021.

Slik pipelinen er konstruert, lages det en ny modell for hver gang pipelinen kjøres. En mulig forbedring er å gi en trent modell som input til pipelinen for å kjøre pipelinen med en «varm start». Trainer-komponenten har et valgfritt parameter for modell,[10] så det vil være her man gjør det. Trainer kan også ta hyperparametere som input, noe som pipelinen ikke støtter i skrivende stund.

Pipelinen er laget slik at det tar csv-filer som datainput. Dette har vært tilstrekkelig for å utvikle og teste pipelinen. Teamet hos Statens vegvesen som jobber med trafikkdata som vi har vært i kontakt med, har også sin inndata som csv-filer, så dette har vært nyttig for dem. Derimot skal Saga-plattformen få mye data i *BigQuery* og bruke dette ekstensivt. Pipelinen bør også tilpasses dette for å fungere bedre som en del av Saga-plattformen. Skal pipelinen bruke *BigQuery* som input fremfor csv, må *CsvExampleGen*-komponenten erstattes med *BigQueryExampleGen*.[26] Bruken av *BigQueryExampleGen* krever også at en spørring blir gitt til `create_pipeline()` i `pipeline.py`. `kubeflow_runner.py` og eventuelt `local_runner.py` må også tilpasses for å gi dette parameteret når funksjonen kalles. Utover dette kreves ingen videre tilpasning av pipelinen for å bruke *BigQuery*.

Overvåkning av pipelinen og modellers ytelse har ikke blitt direkte utviklet av bachelorgruppen, men arbeidet med å kjøre pipelinens komponenter som jobber på GCP muliggjør dette. All logging av *Dataflow*-jobber og treningsjobber i *AI Platform* blir logget til GCP i *Logging*-verktøyet. Bachelorgruppen er ikke kjent med å skrive spørringer i dette verktøyet, men det vil herfra være mulig å hente ut data om det man lurer på angående pipelinens kjøring og modellenes ytelse.

Nok en mulig forbedring av pipelinen, vil være å kjøre den automatisk. Pipelinen kjøres nå manuelt ved å kjøre en TFX-kommando. Det kan for eksempel være ønskelig å kjøre pipelinen automatisk hver gang et visst antall

data har blitt lagt til i *BigQuery* siden sist kjøring av pipelinen. Eventuelt kan man lage en jobb som kjører pipelinen med et fast intervall. Etter gruppens forståelse av GCP, kan dette implementeres ved å lage en *Cloud Function*. Det skal være nok å konfigurere denne jobben til kjøre en enkelt TFX-kommando, gitt at Service Accounten som kjører jobben er autentisert.

6 Konklusjon

Bachelorgruppen fikk i oppgave å undersøke hvordan en maskinlæringspipeline kan se ut og hvordan den kan brukes hos Statens vegvesen. Produktet gruppen skulle jobbe mot var et 'Proof of Concept' av en slik pipeline. Sammen med oppdragsgiver fikk partene konkretisert en rekke behov som SVV hadde angående pipelinen.

Arbeidet som har gått med i driftsfasen av bacheloroppgaven har gått med til å realisere funksjonalitet som tilfredsstilte behovene, hvor samtlige behov har blitt tilfredsstilt.

Punktene under er behovene oppdragsgiver hadde. Disse er hentet fra forstudierapporten.[28]

- *Kunne kjøre ut en maskinlæringsmodell i GCP.*

  Pusher-komponenten i pipelinen kjører ut modeller til GCP.

- *Kunne kjøre ut nye versjoner av en maskinlæringsmodell i GCP.*

  Når Pusher-komponenten kjører ut en ny modell til GCP, blir den nye modellen satt som gjeldende i GCP, mens den gamle fortsatt er tilgjengelig.

- *Kunne eksponere en maskinlæringsmodell for input og bruke den fra andre applikasjoner.*

  Modeller som har blitt kjørt ut til GCP, ligger eksponert for input gjennom API-kall. Denne funksjonaliteten er innebygd i AI Platform på GCP.

- *Finne ut hvordan man kan overvåke en maskinlæringsmodell og dens evne til å predikere riktig.*

Det har blitt lagt fokus på å kjøre komponentene i pipelinen som jobber på GCP. Dette gjør at loggingen fra de forskjellige komponentene samles på ett sted, og at man gjennom loggingverktøyet i GCP blant annet kan overvåke maskinlæringsmodellens ytelse.

- *Finne ut hvordan man kan skalere en pipeline med hensyn på kapasitet og ytelse.*

  At orkestratoren til pipelinen kjører på Kubernetes, gjør det enkelt å gjøre om på clusterstørrelse. Kubernetes kan skaleres enkelt etter behov. En fordel med at komponentene i pipelinen kjører som jobber på GCP, er at disse automatisk kan skaleres med workers i *Dataflow*. Man kan manuelt sette antall workers og maksimum antall workers om man ønsker.

Pipelinen som er beskrevet i dette dokumentet svarer til de behovene oppdragsgiver har ønsket. Det 'Proof of Concept' som skulle utvikles har blitt utviklet, og bachelorgruppen leverer prosjektet videre til Statens vegvesen.

Referanser

[1] GitHub: *Bacheloroppgave for Statens vegvesen*, 2021.
Hentet fra:
`https://github.com/efdalen/bachelor2021`.
Lastet ned: 14. april 2021

[2] GitHub: *Bacheloroppgave for Statens vegvesen - Release Innlevering av bachelorarbeid*, 2021.
Hentet fra:
`https://github.com/efdalen/bachelor2021/releases/tag/1`.
Lastet ned: 11. mai 2021

[3] Google Cloud: *Global Locations - Regions & Zones*, 2021.
Hentet fra:
`https://cloud.google.com/about/locations/#europe`.
Lastet ned: 20. april 2021

[4] Google Cloud: *Specifying pipeline execution parameters*, 2021.
Hentet fra:
`https://cloud.google.com/dataflow/docs/guides/specifying-exec-params#setting-other-cloud-dataflow-pipeline-options`.
Lastet ned: 20. april 2021

[5] Google Cloud: *Introducing Cloud Dataflow Shuffle*, 2017.
Hentet fra:
`https://cloud.google.com/blog/products/gcp/introducing-cloud-dataflow-shuffle-for-up-to-5x-performance-improvement-in-`
Lastet ned: 20. april 2021

[6] Google Cloud: *Machine types*, 2021.
Hentet fra:
`https://cloud.google.com/compute/docs/machine-types`.
Lastet ned: 20. april 2021

[7] Tensorflow: *tfx.components.CsvExampleGen*, 2021.
Hentet fra:
`https://www.tensorflow.org/tfx/api_docs/python/tfx/components/`

CsvExampleGen.
Lastet ned: 20. april 2021

[8] Tensorflow: *tfx.components.SchemaGen*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/components/SchemaGen.
Lastet ned: 20. april 2021

[9] Tensorflow: *tfx.components.Transform*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/components/Transform.
Lastet ned: 20. april 2021

[10] Tensorflow: *tfx.components.Trainer*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/components/Trainer.
Lastet ned: 20. april 2021

[11] Tensorflow: *Module: tfx.extensions.google_cloud_ai_platform.trainer.executor*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/extensions/google_cloud_ai_platform/trainer/executor.
Lastet ned: 20. april 2021

[12] Tensorflow: *Creating a Custom TFX Executor*, 2019.
Hentet fra:
https://blog.tensorflow.org/2019/09/creating-custom-tfx-executor_19.html.
Lastet ned: 20. april 2021

[13] Tensorflow: *LatestBlessedModelResolver*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/dsl/experimental/latest_blessed_model_resolver/

LatestBlessedModelResolver?hl=ru.
Lastet ned: 20. april 2021

[14] Tensorflow: *tfx.components.Evaluator*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/components/Evaluator.
Lastet ned: 20. april 2021

[15] Github: *TFX Chicago Taxi Pipeline*, 2021.
Hentet fra:
https://github.com/tensorflow/tfx/tree/master/tfx/examples/chicago_taxi_pipeline.
Lastet ned: 20. april 2021

[16] Tensorflow: *Tensorflow Model Analysis*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/tutorials/model_analysis/tfma_basic.
Lastet ned: 20. april 2021

[17] Tensorflow: *tfx.components.Pusher*, 2021.
Hentet fra:
https://www.tensorflow.org/tfx/api_docs/python/tfx/components/Pusher.
Lastet ned: 20. april 2021

[18] V. Tatan, "Intro to ML Ops: Tensorflow Exntended (TFX)",towards datascience,Apr.2020.[Online]. Hentet fra:
https://towardsdatascience.com/intro-to-ml-ops-tensorflow-extended-tfx-39b6a
Lastet ned: 23.02.2021

[19] Argo Project, GitHub, *Argo Documentation*, 2021. Hentet fra:
https://argoproj.github.io/argo-workflows/.
Lastet ned: 01.03.2021

[20] Kubeflow, *Building Pipelines with the SDK*, 2020. Hentet fra:
https://www.kubeflow.org/docs/pipelines/sdk/.
Lastet ned: 01.03.2021.

[21] Google Cloud, *Projects*, 2021. Hentet fra:
`https://cloud.google.com/storage/docs/projects`.
Lastet ned: 05.03.2021

[22] Google Cloud, *Key terms*, 2021. Hentet fra:
`https://cloud.google.com/storage/docs/key-terms`.
Lastet ned: 22.04.2021

[23] Google Cloud, *VCP network overview*, 2021. Hentet fra:
`https://cloud.google.com/vpc/docs/vpc`.
Lastet ned: 22.04.2021

[24] Kubernetes, What is Kubernetes?, 2021. Hentet fra:
`https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`.
Lastet ned: 22.04.2021

[25] Google Cloud, *Google Kubernetes Engine Documentation*, 2021. Hentet fra:
`https://cloud.google.com/kubernetes-engine/docs`.
Lastet ned: 22.04.2021

[26] Tensorflow: *BigQueryExampleGen*, 2021.
Hentet fra:
`https://www.tensorflow.org/tfx/api_docs/python/tfx/extensions/google_cloud_big_query/example_gen/component/BigQueryExampleGen`.
Lastet ned: 26. april 2021

[27] Google Cloud: *Best practices for enterprise organizations*, 2021.
Hentet fra:
`https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations#project-structure`.
Lastet ned: 14. mai 2021

[28] J. A. Langholm, I. Andersen og E. F. Dalen, ”Forstudierapport”, *NTNU*, Februar 2021. [Vedlegg].
Hentet fra:
Vedlegg i innlevering av oppgave.

[29] J. A. Langholm, I. Andersen og E. F. Dalen, ”Designrapport”, *NTNU*, Mars 2021. [Vedlegg].
Hentet fra:
Vedlegg i innlevering av oppgave.

[30] J. A. Langholm, I. Andersen og E. F. Dalen, ”Ordbok”, *NTNU*, Mai 2021. [Vedlegg].
Hentet fra:
Vedlegg i innlevering av oppgave.

[31] J. A. Langholm, I. Andersen og E. F. Dalen, ”README”, *NTNU*, Mai 2021. [Vedlegg].
Hentet fra:
Vedlegg i innlevering av oppgave.

[32] Github: *Is there a way to disable cache...*, 2021.
Hentet fra:
`https://github.com/kubeflow/pipelines/issues/4857`.
Lastet ned: 19. mai 2021

7 Appendiks

A Gjennomgang av kode

Koden som blir fremvist under er hentet fra Github-repositoryet som bachelorgruppen har brukt under arbeidet med oppgaven.[1] Versjonen av koden som ligger i rapporten er den samme som fins i utgave *1 - Innlevering av bachelorarbeid* på GitHub.[2] En full kopi av utgaven kan lastes ned derfra.

Grunnet begrensninger i -biblioteket som anvendes i denne rapporten for å fremstille kode, kan det være mindre feil i linjenumre. Av og til har forfatter måttet tvinge linjeskift, og da blir dette registrert som en ny linje, noe som gjør at linjene under blir forskjøvet relativt til det kunstige linjeskiftet. Der de forklarende avsnittene refererer til konkrete linjenumre, er linjenumrene oppdatert til de linjenumre som står i rapporten. Avsnittene er ment til å leses sammen med kodeboksene i rapporten, og ikke med den faktiske kildekoden slik den fremstår på GitHub, da det enkelte steder vil være små forskjeller i linjenumre. Hver kodeboks' første linje er derimot synkron med kildekoden, og ikke alle kodebokser har kunstige linjeskift som skaper problemer.

A.1 template/pipeline/gcp_infrastructure.py

Prosjektgruppen har skrevet et skript for å sette opp nødvendig infrastruktur i GCP. Skriptet er skrevet i Python og benytter GCPs REST API-er og Pythonmoduler. Dette skriptet bør bare kjøres en gang, når man skal sette opp prosjektet i GCP, og helst samtidig som man gjennomgår README..[31]

```
1 import google.auth
2 from google.cloud import storage
3 from configs import GCS_BUCKET_NAME, PIPELINE_NAME,
  ↪  CLUSTER_SCALING
4 import cluster_creation
5 import network_creation
6 import json
7 import requests
```

```
8  import google.auth.transport.requests
9  import os
10 import subprocess
11 import sys
12 import time
```

Linje 1-12 inneholder importsetninger for å inkludere nødvendige Pythonmoduler og andre filer. Linje 4-5 (cluster_creating og network_creation) inneholder variabler av typen Dict, som underveis i skriptet oppdateres og konverteres til JSON for å nyttes som payload til REST API-ene.

```
14 # Get default project and authentication
15 try:
16     _, GOOGLE_CLOUD_PROJECT = google.auth.default()
17 except google.auth.exceptions.DefaultCredentialsError:
18     GOOGLE_CLOUD_PROJECT = ''
```

Linje 14-18 henter valgt prosjekts navn og et token for autentisering.

```
20 ### BUCKET CREATION ###
21 # Creates bucket with necessary subdirectories
22 print("Building GCS bucket...")
23 try:
24     client = storage.Client(project=GOOGLE_CLOUD_PROJECT)
25     try:
26         bucket = storage.Bucket(client, name=GCS_BUCKET_NAME)
27         bucket.create(location='eu')
28         print("Created bucket {} in project
           ↪  {}.".format(GCS_BUCKET_NAME, GOOGLE_CLOUD_PROJECT))
29     except Exception as e:
30         if e.code == 409:
31             print("Bucket already exists.")
32         else:
```

```
33              raise
34      try:
35          bucket = client.get_bucket(GCS_BUCKET_NAME)
36          path_modules = bucket.blob('input/modules/')
37          path_data = bucket.blob('input/data/')
38          path_output = bucket.blob('output/')
39          path_modules.upload_from_string('')
40          path_data.upload_from_string('')
41          path_output.upload_from_string('')
42          print("Built bucket hierarchy.")
43      except Exception as e:
44          print(e)
45  except Exception as e:
46      print(e)
```

Linje 20-46 omhandler opprettelse av en GCS-bucket med korrekt hierarki. Navn på bucket er definert i linje 26 og er hentet fra `configs.py`. Videre er feilhåndtering dersom bucketen allerede eksisterer. Linje 36-41 lager mappestrukturen i bucketen.

```
48  # Updloading necessary files to correct subdirectories
49  try:
50      client = storage.Client(project=GOOGLE_CLOUD_PROJECT)
51      bucket = client.get_bucket(GCS_BUCKET_NAME)
52      preprocessing = bucket.blob('input/modules/preprocessing.py')
53      trainer = bucket.blob('input/modules/trainer.py')
54      data = bucket.blob('input/data/data.csv')
55      preprocessing.upload_from_filename(
56          './modules/preprocessing.py'
57      )
58      print("Uploaded preprocessing.py to bucket
        ↪ {}/input/modules/preprocessing.py".format(
59          GCS_BUCKET_NAME
60      ))
```

```
61      trainer.upload_from_filename('./modules/trainer.py')
62      print("Uploaded trainer.py to bucket
        ↪  {}/input/modules/trainer.py".format(GCS_BUCKET_NAME))
63      data.upload_from_filename('./data/data.csv')
64      print("Uploaded data to bucket
        ↪  {}/input/data/data.csv".format(GCS_BUCKET_NAME))
65  except Exception as e:
66      print(e)
```

Linje 50-63 sørger for å laste opp nødvendige filer og data til GCS for bruk av pipelinen. Her lastes kode for preprosessering og trening opp, samt dataen som skal brukes til trening og testing.

```
64  ### Network creation ###
65  POST_URL = "https://www.googleapis.com/compute/v1/projects/{}
    ↪  /global/networks".format(GOOGLE_CLOUD_PROJECT)
66  POST_PAYLOAD = network_creation.network
67  POST_PAYLOAD["selfLink"] = "projects/{}
    ↪  /global/networks/default".format(GOOGLE_CLOUD_PROJECT)
68  POST_PAYLOAD = json.dumps(POST_PAYLOAD)
69
70  # Creating credential token
71  creds, project = google.auth.default(scopes=[
72      "https://www.googleapis.com/auth/cloud-platform"
73  ])
74  auth_req = google.auth.transport.requests.Request()
75  creds.refresh(auth_req)
76
77  # REST API call to create network
78  print("Building network...")
79  response = requests.post(POST_URL, data = POST_PAYLOAD,
    ↪  headers={'Authorization': 'Bearer {}'.format(creds.token)})
80  resp_json = json.loads(response.text)
81
```

```
82 # Catching HTTP errors
83 if response.status_code != 200:
84     if resp_json["error"]["errors"][0]["reason"] ==
       ↪  "alreadyExists":
85         print("Network already exists.")
86     else:
87         print("Network creation failed. \n")
88         print(response.status_code)
89         print(response.text)
90 else:
91     print("Network is deploying...")
```

Linje 65-80 omhandler oppretting av et VPC-nettverk. Her brukes et REST API, som kalles med en POST-forespørsel bestående av JSON-data om nettverkskonfigurasjon. I linje 67 sikres hvilket prosjekt nettverket opprettes i. Linje 71-75 oppdaterer et token, nødvendig for autentisering av forespørselen.

```
91 ### CLUSTER CREATION ###
92 POST_PAYLOAD = {}
93 POST_URL = ""
94
95 # Defines which JSON data to POST to which URL
96 if CLUSTER_SCALING == 'standard':
97     POST_PAYLOAD = cluster_creation.standard
98     POST_URL =
       ↪  "https://container.googleapis.com/v1beta1/projects/{}
       ↪  /zones/europe-west3-a/clusters
       ↪  ".format(GOOGLE_CLOUD_PROJECT)
99 else:
100    POST_PAYLOAD = cluster_creation.autopilot
101    POST_URL = "https://container.googleapis.com/v1/projects/{}
       ↪  /locations/europe-west3/clusters
       ↪  ".format(GOOGLE_CLOUD_PROJECT)
```

```
102
103 # Set variables for cluster creation
104 CLUSTER_NAME = PIPELINE_NAME + '-cluster'
105 CLUSTER_NAME = CLUSTER_NAME.replace('_','-')
106
107 # Alters payload to match GCP project and adds relevant cluster
    ↪ name
108 POST_PAYLOAD["cluster"]["name"] = CLUSTER_NAME
109 POST_PAYLOAD["cluster"]["network"] =
    ↪ "projects/{}/global/networks/default
    ↪ ".format(GOOGLE_CLOUD_PROJECT)
110 POST_PAYLOAD["cluster"]["subnetwork"] =
    ↪ "projects/{}/regions/europe-west3/subnetworks/default
    ↪ ".format(GOOGLE_CLOUD_PROJECT)
111 # Dict to JSON for POST data type
112 POST_PAYLOAD = json.dumps(POST_PAYLOAD)
113
114 # Creating credential token
115 creds, project = google.auth.default(scopes=["
    ↪ https://www.googleapis.com/auth/cloud-platform"])
116 auth_req = google.auth.transport.requests.Request()
117 creds.refresh(auth_req)
118
119 # REST API call to build cluster
120 print("Building cluster...")
121 response = requests.post(POST_URL, data = POST_PAYLOAD,
    ↪ headers={'Authorization': 'Bearer {}'.format(creds.token)})
122 resp_json = json.loads(response.text)
123
124 # Waiting for network to be fully configured
125 while response.status_code == 400:
126     response = requests.post(POST_URL, data = POST_PAYLOAD,
        ↪ headers={'Authorization': 'Bearer
        ↪ {}'.format(creds.token)})
```

```
127     resp_json = json.loads(response.text)
128 
129 # Catching HTTP errors when creating cluster
130 if response.status_code != 200:
131     if resp_json["error"]["status"] == "ALREADY_EXISTS":
132         print("Cluster already exists.")
133     else:
134         print("Cluster build failed. \n")
135         print(response.status_code)
136         print(response.text)
137 else:
138     print("Cluster build completed successfully. Cluster
      ↪  deployment in progress. Continue with KFP deployment.")
```

Linje 92-127 sørger for å opprette et cluster i Kubernetes Engine. Dette er nødvendig for å rulle ut Kubeflow Pipelines i GCP. Linje 96-101 er til dags dato ikke brukbar, da skaleringskonfigurasjon av clusteret må være standard for å være kompatibelt med Kubeflow Pipelines. Kodesnutten ligger likevel inne for at man i fremtiden kan nytte automatisk skalering ved behov. Linje 104-105 definerer navnet på clusteret og sørger for at understreker byttes ut med bindestreker, da det er eneste tillatte spesialtegn. Linje 108-111 sørger for at clusteret blir konfigurert riktig, ved å oppdatere payloaden. Linje 121 utfører POST-forespørselen med nødvendig payload. Gjennom skriptet tar det kort tid fra forsepørselen om å opprette nettverk til å deploye cluster skjer. Det er derfor sannsynlig at nettverket ikke er ferdig konfigurert før clusteret forsøkes å deployes. Derfor sørger linje 125-127 for å vente til nettverket er ferdig konfigurert før clusteret deployes. Resten av koden er feilhåndtering.

A.2 template/pipeline/pipeline.py

Denne filen definerer definerer strukturen og flyten til pipelinen. Komponentene blir en etter en definert med konfigurasjon importert fra *configs.py* og med outputen til tidligere komponenter som parametere. En forklaring av flyten og strukturen til pipelinen finnes i kapittel 3 og i

designrapporten.[29]

Begynnelsen av filen inneholder importsetninger av biblioteker som senere brukes i filen.

```
54 def create_pipeline(
55     pipeline_name: Text,
56     pipeline_root: Text,
57     metadata_connection_config:
       ↪  Optional[metadata_store_pb2.ConnectionConfig] = None,
58     beam_pipeline_args: Optional[List[Text]] = None,
59     enable_cache: bool = False,
60 
61     csv_example_gen_args: Optional[Dict[Text, Any]] = None,
62     schema_gen_args: Optional[Dict[Text, Any]] = None,
63     transform_args: Optional[Dict[Text, Any]] = None,
64     trainer_args: Optional[Dict[Text, Any]] = None,
65     model_resolver_args: Optional[Dict[Text, Any]] = None,
66     evaluator_args: Optional[Dict[Text, Any]] = None,
67     pusher_args: Optional[Dict[Text, Any]] = None,
68 ) -> pipeline.Pipeline:
```

Dette er alle parametrene som gis til selve funksjonen som definerer pipelinen. Disse defineres i `configs.py` og gis via `kubeflow_runner.py`.

```
54     components = []
```

Denne listen vil bli fylt med de individuelle TFX-komponentene som defineres senere i filen.

```
72     # Brings data into the pipeline or otherwise joins/converts
       ↪  training data.
73     example_gen = CsvExampleGen(**csv_example_gen_args)
74     components.append(example_gen)
```

Dette er den første komponenten som defineres i pipelinen. Om man heller vil bruke BigQuery som input, må man erstatte linje 73 og heller bruke `bigqueryexamplegen(query=query)` der `query` er en streng som er spørringen som skal gjøres mot bigquery. Utover dette kreves ingen andre tilpasninger i pipelinen for å bruke BigQuery fremfor csv.

```
76      # Computes statistics over data for visualization and example
        ↪ validation.
77      statistics_gen =
        ↪ StatisticsGen(examples=example_gen.outputs['examples'])
78      components.append(statistics_gen)
```

Genererer statistikk over inputdataen. Dette brukes senere for å validere dataen. Man kan også bruke output herfra for å analysere dataen.

```
80      # Generates schema based on statistics files.
81      schema_gen = SchemaGen(
82          statistics=statistics_gen.outputs['statistics'],
83          **schema_gen_args)
84      components.append(schema_gen)
```

Lager et skjema for dataen. Skjemaet beskriver dataen med typer, kategorier og rammer. Skjemaet er automatisk generert og stemmer ikke nødvendigvis 100%, så dataviter bør se over og eventuelt modifisere. Skjemaet kan skrives helt manuelt, men det er gjerne greiest å modifisere et som kommer fra denne komponenten slik man vil ha det. Skjemaet er å finne som klartekst i filen `<pipeline_root>/SchemaGen/schema/<artifact_id>/schema.pbtxt`. Ved senere kjøring kan det være ønskelig å heller bruke et skjema man har laget selv. Det automatisk genererte skjemaet er bare gjetning fra en datamaskin, og hvis man skal forsikre seg om at man har et korrekt skjema, bør man ha laget det selv slik at det ikke oppstår problemer en dag man møter på søppeldata. Dette er ikke funksjonalitet som bachelorgruppen har prioritert å utvikle, så slik pipelinen står i skrivende stund, genereres skjema automatisk hver gang man kjører pipelinen.

```
86      # Performs anomaly detection based on statistics and data
        ↪ schema.
87      example_validator = ExampleValidator(
88          statistics=statistics_gen.outputs['statistics'],
89          schema=schema_gen.outputs['schema'])
90      components.append(example_validator)
```

Validerer at inndataen passer til skjemaet og oppdager eventuelle uregelmessigheter. Komponenten kan kjenne igjen «skew» mellom treningsdata og evalueringsdata og data drift. Om man ikke får rensket dataen, vil det bli problemer når man senere skal trene modellen.

```
92      # Performs transformations and feature engineering in
        ↪ training and serving.
93      transform = Transform(
94          examples=example_gen.outputs['examples'],
95          schema=schema_gen.outputs['schema'],
96          **transform_args)
97      components.append(transform)
```

Transform utfører førprosessering på dataen. Transform trenger en `preprocessing_fn()`. Denne må defineres i en Python-modul som så gis til Transform og brukes i førprosesseringen. Konfigurasjon av denne komponenten må gjøres av dataviter som kjenner modellen og vet hvilke krav den stiller til inndata og formen på de.

```
99      trainer = Trainer(
100         examples=transform.outputs['transformed_examples'],
101         transform_graph=transform.outputs['transform_graph'],
102         schema=schema_gen.outputs['schema'],
103         **trainer_args
104     )
105     components.append(trainer)
```

Kjører selve treningen av modellen. Dette må konfigureres av dataviter på lik linje med Transform, da denne er spesifikk for modellen. Denne kan også ta en ferdig modell som input for å kjøre «varm start» ved å sette `base_model`-parameteret. Dette gjør man hvis man ikke ønsker å trene en modell helt fra «scratch».

```
107     # Get the latest blessed model for model validation.
108     model_resolver = ResolverNode(**model_resolver_args)
109     components.append(model_resolver)
```

Henter den for øyeblikket gjeldende versjonen av modellen fra *Cloud Storage*, slik at den kan sammenlignes med den nye fra Trainer.

```
111     evaluator = Evaluator(
112         examples=example_gen.outputs['examples'],
113         model=trainer.outputs['model'],
114         baseline_model=model_resolver.outputs['model'],
115         **evaluator_args,
116     )
117     components.append(evaluator)
```

Evaluator sammenligner den nye modellen med den gamle. Den som er best blir gitt videre slik at den kan sendes til endepunkt. Evaluator behøver ikke konfigurasjon, og vil uten konfigurasjon bare sammenligne total ytelse på de to. Med konfigurasjon kan man be komponenten se på spesielle tilfeller.

```
119     # Checks whether the model passed the validation steps and
        ↪  pushes the model
120     # to a file destination if check passed.
121     pusher = Pusher(
122         model=trainer.outputs['model'],
123         model_blessing=evaluator.outputs['blessing'],
124         **pusher_args,
125     )
126     components.append(pusher)
```

Pusher sender modellen til endepunkt.

```
128     return pipeline.Pipeline(
129         pipeline_name=pipeline_name,
130         pipeline_root=pipeline_root,
131         components=components,
132         enable_cache=enable_cache,
133         metadata_connection_config=metadata_connection_config,
134         beam_pipeline_args=beam_pipeline_args,
135     )
```

Disse siste linjene i `pipeline.py` returnerer pipelinen til `kubeflow_runner.py` som kan ses på som hovedfilen. Pipeline-objektet blir kompilert og siden gitt til Kubeflow Pipelines der den kjører.

A.3 template/pipeline/configs.py

Den forrige filen definerte bare sammenhengen mellom de forskjellige TFX-komponentene og var veldig vag i hvordan disse konkret skulle fungere. All konfigurasjon av komponentene samt øvrig konfigurasjon ligger i filen `template/pipeline/configs.py`. Et viktig unntak er skriptene som kjøres av Transform- og Trainer-komponentene.

Linje 1 til 41 inneholder importsetninger av biblioteker som blir brukt senere.

```
42 # Navnet brukes for å identifisere denne pipelinen. Navnet brukes
    ↪  konkret i KFP og i GCS.
43 PIPELINE_NAME = 'dev-pipeline'
```

Som kommentaren sier, brukes dette navnet for å identifisere pipelinen. Dette vil bli brukt senere i konfigurasjonsfilen, og også av `pipeline.py` når pipelinen kompileres.

```
45 # Brukes for å definere config for cluster.
46 # 'standard' or 'autopilot'
47 CLUSTER_SCALING = 'standard' # Kan også bruke autopilot, sett da
   ↪ "autopilot"
```

Definerer skaleringen av Kubernetes-clusteret. Ved standard, skalerer man clusteret manuelt, mens autopilot skalerer automatisk. Det er per nå ikke støtte for kjøring av Kubeflow Pipelines på autopilot-cluster da autopilot er under etablering hos GCP.

```
49 # GCP related configs.
50
51 # Following code will retrieve your GCP project. You can choose
   ↪ which project
52 # to use by setting GOOGLE_CLOUD_PROJECT environment variable.
53 try:
54   import google.auth  # pylint: disable=g-import-not-at-top
55   try:
56     _, GOOGLE_CLOUD_PROJECT = google.auth.default()
57   except google.auth.exceptions.DefaultCredentialsError:
58     GOOGLE_CLOUD_PROJECT = ''
59 except ImportError:
60   GOOGLE_CLOUD_PROJECT = ''
```

Denne kodesnutten henter navnet på GCP-prosjektet slik at man slipper å skrive dette selv.

```
62 # Specify your GCS bucket name here. You have to use GCS to store
   ↪ output files
63 # when running a pipeline with Kubeflow Pipeline on GCP or when
   ↪ running a job
64 # using Dataflow. Default is
   ↪ '<gcp_project_name>-kubeflowpipelines-default'.
65 # This bucket is created automatically when you deploy KFP from
   ↪ marketplace.
```

```
66
67  # Using pipeline name to name GCS bucket
68  bucket_name = PIPELINE_NAME
69
70  # String to lowercase
71  bucket_name = bucket_name.lower()
72
73  # For norwegian letters, use ae, o, a
74  bucket_name = bucket_name.replace('æ', 'ae')
75  bucket_name = bucket_name.replace('ø', 'o')
76  bucket_name = bucket_name.replace('å', 'a')
77
78  # Removing all characters but a-z, 0-9, whitespace, '_', '-' and
    ↪  '.'
79  bucket_name = re.sub("[^a-z0-9\s\_\-\.]+", "", bucket_name)
80
81  # Swapping whitespace with '-'
82  bucket_name = re.sub("\s", "-", bucket_name)
83
84  # Remove repeating '-' (dashes) and dash after '.' and '_'
85  iter = list(bucket_name)
86  bucket_name = ''
87  for i in range (0,len(iter)):
88      if iter[i] == '-' and iter[i-1] in '.-_':
89          iter[i] = ''
90      bucket_name += str(iter[i])
91
92  GCS_BUCKET_NAME = GOOGLE_CLOUD_PROJECT + '-' + bucket_name
```

Et navn på bucketen avledes fra pipelinenavnet. Eventuelle spesialtegn som ikke kan være i et bucketnavn lukes ut slik at det endelige bucketnavnet er på riktig format.

```
94  # TFX pipeline produces many output files and metadata. All
    ↪  output data will be
95  # stored under this OUTPUT_DIR.
96  OUTPUT_DIR = os.path.join('gs://', GCS_BUCKET_NAME, 'output')
97  INPUT_DIR = os.path.join('gs://', GCS_BUCKET_NAME, 'input')
98
99  # The last component of the pipeline, "Pusher" will produce
    ↪  serving model under
100 # SERVING_MODEL_DIR.
101 SERVING_MODEL_DIR = os.path.join(OUTPUT_DIR, 'serving_model')
102
103 # Specifies data file directory. DATA_PATH should be a directory
    ↪  containing CSV
104 # files for CsvExampleGen in this example. By default, data files
    ↪  are in the
105 # GCS path: `gs://{GCS_BUCKET_NAME}/input/data/`.
106
107 DATA_PATH = os.path.join(INPUT_DIR, 'data')
108 PREPROCESSING_PATH = os.path.join(INPUT_DIR,
    ↪  'modules/preprocessing.py')
109 TRAINER_PATH = os.path.join(INPUT_DIR, 'modules/trainer.py')
```

Diverse stier som brukes i pipelinen defineres her; både for input og output. Stiene for preprocessing og trainer peker på konfigurasjonen for disse to komponentene. Det er disse filene som må skrives av dataviteren og være tilpasset modellen som skal kjøre i pipelinen.

Spesifikt for oppgavens proof of concept, blir filene som skal ligge i disse stiene lastet opp av skriptet `gcp_infrastructure.py`

```
111 # Setter region til europa. Pga begrensninger i GCP, kan vi ikke
    ↪  kjøre alle tjenester i samme region. Deploying av modeller
    ↪  vil skje i europe-west1, mens resten skjer i europe-west4.
112 GOOGLE_CLOUD_REGION = 'europe-west4'
```

```
113 PUSHER_REGION = 'europe-west1'
```

De forskjellige regionene hvor pipelinen kjører. Ingen region i Europa støtter all funksjonaliteten alene,[3] så pipelinen kjører per dags dato både i Belgia (europe-west1) og i Holland (europe-west4).

```
115 ENABLE_CACHE = False
```

Skrur av mellomlagring av komponenters output. Om en komponent har blitt kjørt tidligere med identisk input, blir den ikke kjørt igjen, og outputen fra den tidligere kjøringen blir heller brukt. Dette kan man også skru av for KFP. Det har oppstått problemer med doble artefakter, så dette er avskrudd. Dette innebærer at enkelte komponenter fikk dobbel input, og ikke kunne kjøre. Et problem med Kubernetes gjorde at bachelorgruppen uheldigvis måtte skru dette av i KFP.[32] Det vil være fordelaktig å komme tilbake til dette senere for å rette opp i feilen.

```
117 #Argumenter til dataflow
118 DATAFLOW_BEAM_PIPELINE_ARGS = [
119     '--project=' + GOOGLE_CLOUD_PROJECT,
120     '--runner=DataflowRunner',
121     '--temp_location=' + os.path.join('gs://', GCS_BUCKET_NAME,
    ↪  'tmp'),
122     '--region=' + GOOGLE_CLOUD_REGION,
123 
124     # Temporary overrides of defaults.
125     '--disk_size_gb=50',
126     '--experiments=shuffle_mode=auto',
127     '--machine_type=e2-standard-8',
128     ]
```

Argumenter som gis til Dataflow på GCP.[4]

- `--project` sier hvilket prosjekt det gjelder.

- `--runner` sier hvilken runner som skal brukes. `DataflowRunner` bør helst brukes. Alternativet er `Directrunner` som vil være å kjøre lokalt.
- `--temp_location` sier hvor midlertidige filer skal lagres.
- `--region` sier hvilken GCP-region Dataflow skal kjøre.
- `--disk-size` angir diskstørrelsen til hver Compute Engine worker.
- `--experiments=shuffle_mode` angir bruken av Dataflow Shuffle, som er en optimalisering av Dataflowjobber.[5]
- `--machine-type` angir VM-typen for Compute Engine workers.[6]

Det er andre parametere man kan spesifisere her; blant annet antall workers man vil ha til jobben.[4]

```
130 #Trening på GCP
131 GCP_AI_PLATFORM_TRAINING_ARGS = {
132     'project': GOOGLE_CLOUD_PROJECT,
133     'region': GOOGLE_CLOUD_REGION,
134 }
```

Oppgir prosjekt og region for trening som skjer på AI Platform i GCP.

```
136 #Serving på GCP
137 GCP_AI_PLATFORM_SERVING_ARGS = {
138     'model_name': PIPELINE_NAME.replace('-','_'),  # '-' is not
    ↪  allowed.
139     'project_id': GOOGLE_CLOUD_PROJECT,
140     'regions': [PUSHER_REGION],
141 }
```

Argumenter til serving av modeller på GCP. Modeller gis det samme navnet som pipelinen for ryddighet. Modeller pushes til Belgia (europe-west1), da det ikke støttes i Holland (europe-west4).

```
143 #Parametere til TFX-komponenter
```

Videre vil det komme konfigurasjon av de forskjellige komponentene som er beskrevet i `pipeline.py` i kapittel A.2. Konfigurasjonen er i form av objekter som inneholder parametere til de forskjellige funksjonene som definerer TFX-komponentene. Merk at objektene ikke inneholder de paramtere som er output fra andre komponenter.

A.3.0.1 CsvExampleGen

```
145 CSV_EXAMPLE_GEN_ARGS = {
146     'input_base': DATA_PATH,
147 }
```

Argumenter til CsvExampleGen.[7] Denne komponenten behøver bare en sti. `input_base` er stien til en mappe der det ligger en eller flere csv-filer.

A.3.0.2 SchemaGen

```
149 SCHEMA_GEN_ARGS = {
150     'infer_feature_shape': True,
151 }
```

Argumenter til SchemaGen.[8] `infer_feature_shape` angir hvorvidt komponenten skal legge inn formene til dataens attributter automatisk, slik at man ikke trenger å definere skjemaet selv.

A.3.0.3 Transform

```
153 TRANSFORM_ARGS = {
154     #sti til konfigurasjonsfil for transform (foerprosessering)
155     'module_file': PREPROCESSING_PATH,
156 }
```

Argumenter til Transform.[9] `module_file` er stien til en fil som må inneholde en funksjon som heter `preprocessing_fn()`. Man kan se på denne funksjonen som hovedfunksjonen til førprosesseringen. Funksjonen må ha denne signaturen:

```
def preprocessing_fn(inputs: Dict[Text, Any]) -> Dict[Text, Any]:
```

Typen til både inputdicten og outputdicten må være av `tf.Tensor` eller `tf.SparseTensor`
Modulfilen som inneholder denne funksjonen skal ligge i en GCS bucket med stien som defineres på linje 108.

A.3.0.4 Trainer

```
158 #treningssteg
159 TRAIN_NUM_STEPS = 100
160
161 #evalueringssteg
162 TRAINER_EVAL_NUM_STEPS = 15
163
164 TRAINER_ARGS = {
165     #sti til konfigurasjonsfil for trainer
166     'module_file': TRAINER_PATH,
167
168     #får trening til å skje på GCP
169     'custom_executor_spec': executor_spec.ExecutorClassSpec(
        ↪ ai_platform_trainer_executor.GenericExecutor),
170     'custom_config': {
171         ai_platform_trainer_executor.TRAINING_ARGS_KEY:
            ↪ GCP_AI_PLATFORM_TRAINING_ARGS
172     },
173     'train_args':
        ↪ trainer_pb2.TrainArgs(num_steps=TRAIN_NUM_STEPS),
174     'eval_args':
        ↪ trainer_pb2.EvalArgs(num_steps=TRAINER_EVAL_NUM_STEPS)
```

```
175 }
```

Konfigurasjon av treningen.[10]

- `TRAIN_NUM_STEPS` angir antall steg i treningen. Dette blir siden gitt på linje 173.
- `TRAINER_EVAL_NUM_STEPS` angir antall steg i evalueringen i treningen. Dette blir siden gitt på linje 174.
- `module_file` er stien til filen som definerer treningen. Når man bruker `GenericExecutor`, slik som pipelinen er definert, må filen ha en funksjon som heter `run_fn()` med denne signaturen:

  ```
  def run_fn(trainer.fn_args_utils.FnArgs)
  ```

  Den trente modellen må lagres til stien i `FnArgs.serving_model_dir`.
- `custom_executor_spec` spesifieserer hvilken executor som skal utføre treningen. Her er den definert med `GenericExecutor`. Man har to valg her.[11] Eventuelt kan man lage sin egen.[12]
- `custom_config` inneholder prosjekt og region slik de ble definert på linje 132 og 133.

A.3.0.5 ResolverNode

```
178 MODEL_RESOLVER_ARGS = {
179     'instance_name': 'latest_blessed_model_resolver',
180     'resolver_class':
        ↪  latest_blessed_model_resolver.LatestBlessedModelResolver,
181     'model': Channel(type=Model),
182     'model_blessing': Channel(type=ModelBlessing),
183 }
```

Argumenter for `ResolverNode` som henter gjeldende modell.[13]

A.3.0.6 Evaluator

```
185 # Change this value according to your use cases.
186 EVAL_ACCURACY_THRESHOLD = 0.6
187
188 # Uses TFMA to compute a evaluation statistics over features of a
    ↪ model and
189 # perform quality validation of a candidate model (compared to a
    ↪ baseline).
190 EVAL_CONFIG = tfma.EvalConfig(
191     model_specs=[
192         # This assumes a serving model with signature
            ↪ 'serving_default'. If
193         # using estimator based EvalSavedModel, add
            ↪ signature_name: 'eval' and
194         # remove the label_key.
195         tfma.ModelSpec(label_key='tips')
196     ],
197     metrics_specs=[
198         tfma.MetricsSpec(
199             # The metrics added here are in addition to those
                ↪ saved with the
200             # model (assuming either a keras model or
                ↪ EvalSavedModel is used).
201             # Any metrics added into the saved model (for example
                ↪ using
202             # model.compile(..., metrics=[...]), etc) will be
                ↪ computed
203             # automatically.
204             # To add validation thresholds for metrics saved with
                ↪ the model,
205             # add them keyed by metric name to the thresholds
                ↪ map.
206             metrics=[
207                 tfma.MetricConfig(class_name='ExampleCount'),
```

```
208                 tfma.MetricConfig(class_name='BinaryAccuracy',
209                   threshold=tfma.MetricThreshold(
210                       value_threshold=tfma.GenericValueThreshold(
211                           lower_bound={'value':
                                ↪ EVAL_ACCURACY_THRESHOLD}),
212                       change_threshold=
                            ↪ tfma.GenericChangeThreshold(
213                           direction=tfma.MetricDirection.
                                ↪ HIGHER_IS_BETTER,
214                           absolute={'value': -1e-10})))
215             ]
216         )
217     ],
218     slicing_specs=[
219         # An empty slice spec means the overall slice, i.e. the
            ↪ whole dataset.
220         tfma.SlicingSpec(),
221         # Data can be sliced along a feature column. In this
            ↪ case, data is
222         # sliced along feature column trip_start_hour.
223         tfma.SlicingSpec(feature_keys=['trip_start_hour'])
224     ]
225 )
226
227 EVALUATOR_ARGS = {
228     'eval_config': EVAL_CONFIG,
229 }
```

Konfigurasjon av `Evaluator`.[14] Konfigurasjonen her er et eksempel og er hentet fra TFXs taxi-eksempel.[15] Konfigurasjonen her bruker i stor grad Tensorflow Model Analysis.[16] Denne konfigurasjonen er spesifikk for modellen, og må utvikles av dataviter.
`EVAL_ACCURACY_THRESHOLD` angir terskelverdi for minste akseptable nøyaktighet.

A.3.0.7 Pusher

```
231 PUSHER_ARGS = {
232     'push_destination': pusher_pb2.PushDestination(
233         filesystem=pusher_pb2.PushDestination.Filesystem(
234             base_directory=SERVING_MODEL_DIR
235         )
236     ),
237     'custom_executor_spec': executor_spec.ExecutorClassSpec(
        ↪  ai_platform_pusher_executor.Executor),
238     'custom_config': {
239         #får serving til å skje på GCP
240             ai_platform_pusher_executor.SERVING_ARGS_KEY:
241                 GCP_AI_PLATFORM_SERVING_ARGS
242         },
243 }
```

Argumenter til `Pusher`.[17] Denne er konfigurert til å både lagre modellen i en bucket, samt deploye den til AI Platform Models, der den er er klar for eksponering mot tredjepartsapplikasjoner.

- `push_destination` angir stien i GCS der modellen skal lagres. Denne stien er definert på linje 101.
- `custom_executor_spec` definerer hvilken executor som skal kjøre pushingen. Her er denne konfigurert med en som pusher til GCP AI Platform.
- `custom_config` inneholder konfigurasjonen fra linje 137 som inneholder modellnavn, prosjekt og region.