

Drawing-Recognition

Integrasjonsprosjekt - PROG2052

2021-12-10

Gruppe 2

Sindre Eiklid - 526361 - (sindreik@stud.ntnu.no)

Maren Skårestuen Grindal - 526359 - (marensq@stud.ntnu.no)

Rickard Loland - 707081 - (rickarl@stud.ntnu.no)

Innholdsfortegnelse

Ressurser	4
Introduksjon	4
Avgrensning	4
Motivasjon	5
Prosess	5
Kravspek	6
Teknologi	8
Verktøy	8
Eksterne bibliotek	9
Systemarkitektur	10
Kommunikasjon mellom moduler – ZeroMQ	11
Implementasjon	12
Innhenting av datasett	12
Convolutional Neural Network	13
Datasett og forhåndsbehandling av dette	13
Design av nettverks-modell	14
Tegneprogram	18
Brukergrensesnitt	18
Tegneprogrammet	22
Testing og kvalitetssikring	23
Utrulling	24
Emnebruk	25
Resultater	27
Refleksjoner rundt arbeidet	32
Timelog	34
Videreutvikling	36
Konklusjon	37
Literaturliste	38
Vedlegg	39
Prosjektplan	39

Statusrapport 1	53
Statusrapport 2	55
Kravspesifikasjon	57
Arkitekturskisse	60
Scrum board eksempler	62
Brukertester	63
Timelogger	67

Ressurser

Versjonskontrollssystem: [GitHub](#)

Introduksjon

Denne rapporten detaljerer arbeidet gjort rundt å skape et tegnespill der spiller får et ord de skal tegne, gjør sitt beste, og en kunstig intelligens (gjennom maskinlæring) forsøker å gjette hva tegningen er. Vi hadde tidligere spilt [Google's Quick, Draw!](#), og det spillet ble en stor inspirasjon for oss. Rapporten tar en grundig gjennomgang av alle deler av dette prosjektet, fra første idé, gjennom arbeid på hver del av spillets oppsett, og til slutt en konklusjon som sentrerer rundt den ferdige programvaren.

For å få til dette må vi ha en måte å hente inn bilder av alle slags gjenstander på. Som hyppige internett-brukere, gikk vår første tanke selvsagt til søkemotorer, og særlig Google. Slik lykke ville ha det, finnes det faktisk en API som henter resultater fra søker på Google's bilde-søkemotor - SerpAPI. Vår tanke var dermed å bruke SerpAPI til å søke Google Bilder for resultater til for eksempel 'banana', hente inn en viss mengde resultater som lenker til miniatyrbilder, og bruke disse lenkene til å laste ned disse bildene og trenne på dem.

Etter at vi har hentet ned bildene og trent modellen på dette, kan vi bruke modellen i vårt tegneprogram. For det tenkte vi å kode i C++ med bruk av OpenGL for å lage fungerende brukergrensesnitt, med vår erfaring rundt bruk av GLFW som inspirasjon.

Avgrensning

Vi satte ut for å lage et tegnespill, der en maskinlærings-modell forsøker å gjette hva brukeren tegner. Bruker blir gitt et vilkårlig begrep fra en kurert liste som modellen er trent på, og skal forsøke å tegne det. Når maskinen gjetter rett, fjerner vi det brukeren har tegnet, og gir bruker et nytt ord å tegne. Bruker har en viss mengde tid til å tegne bilder, og når tiden er ute slutter spillet, og bruker vil få vite sin poengsum, basert på antall ord maskinen gjettet rett. Deretter kan bruker starte et nytt spill.

Bruker kan tegne med farger, for å utheve farge-relaterte trekk. I tillegg til selve spillet, kan bruker se en forklaring på hvordan spillet funker, og avslutte spillet. Bruker kan ikke velge penselstørrelse.

Programmet skal ikke integrere flerspiller-funksjonalitet. Det skal heller ikke ha flere tegne-funksjonaliteter enn det som er listet over.

Motivasjon

Da vi startet planlegging av prosjektet, tenkte vi med en gang at et produkt som brukte AI var ideelt å satse på. Vi syntes AI er et spennende felt, og var ivrige til å ta det med videre gjennom utdanningen.

Med dette som utgangspunkt, tok vi noen sesjoner med brainstorming, der vi skrev ned flere ulike ideer om prosjektet basert delvis eller helt på AI. Vi gjorde brainstorming rundt blant annet Sjakk AI, Tegnegjenkjenningsspill og en applikasjon som kunne gjenkjenne tekst fra bilde og konvertere dette til et string format (mulighet for real-time translation, eller kopier-og-lim av tekst).

Vi ble enige om at Sjakk AI var for ambisiøst og at tekstgjenkjenning ville kreve mye ML trening. Da stod vi igjen med tegnegjenkjenningsspillet, som var inspirert av Google's Quick, Draw!.

Siden Quick, Draw! ikke er open source, ville vi ikke kunne hente mye fra deres løsning på dette, og vi ville også legge inn noen kvaliteter som er unike for oss. Vi landet på noen ulike ideer her, som for eksempel reglene i selve spillet, samt lage mer funksjonalitet innenfor tegne-aspektet - særlig bruk av farger. Dette gir både bruker mer mulighet til å tegne fritt, og også mer data å jobbe med til ML-modellen vår.

Vi var veldig ivrige til å jobbe med dette som vårt prosjekt, fordi det hørtes ut som en morsom måte å kombinere det vi tidligere hadde lært under studiet på en interessant og utfordrende måte. Vi ønsket også å lage et 'levende' datasett gjennom automatisk bilde-nedhenting, fordi dette hørtes veldig spennende ut etter å ha jobbet med kurerte datasett før. Det å lage et fungerende tegneprogram virket også veldig morsomt.

Prosess

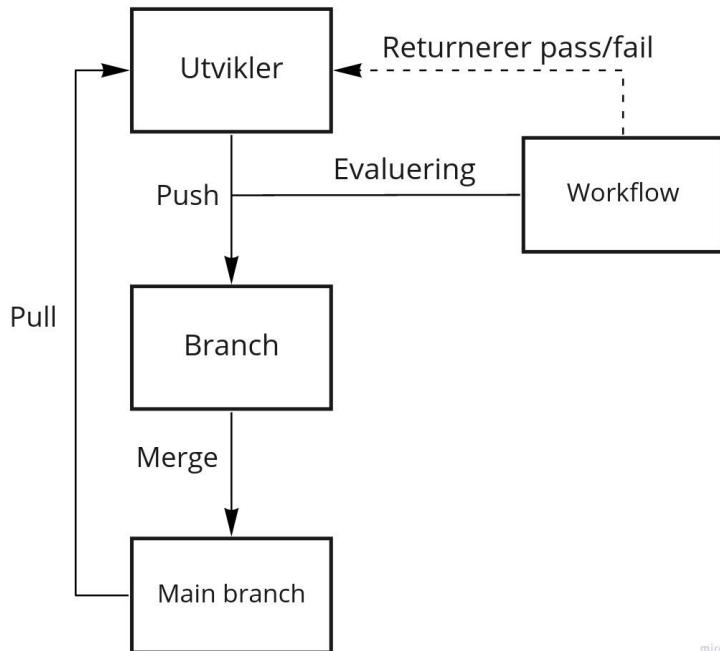
Vi brukte scrum-metodikk, med ukentlige sprinter (som vi senere gjorde om til annenhver uke). For å holde styr på oppgaver brukte vi GitHub issue-board, der vi la inn ulike deler av prosjektet som egne boards, og så mange oppgaver og funksjoner som vi kom på i hvert board. Vi brukte disse issuenene til å markere hva vi jobbet med til enhver tid.

Vi brukte også verktøyet Toggl til å holde styr på timebruken vår, både for logging til rapporten men også slik at vi kunne sammenligne med hverandre under arbeidet for å forsikre at alle gjorde sin del, og hjelpe hverandre med å passe på tidsbruk under møter og slikt.

Vi startet arbeidet med å jobbe på datainnhenting og CNN. Relativt raskt gjorde vi oss ferdig med datainnhenting, og gikk deretter over til arbeid på tegneprogrammet. Vi startet også rapportskriving relativt tidlig, da vi regnet med det ville innebære en del revisjoner og oppdateringer.

Bruk av GitHub og dens verktøy for kodehåndtering sto veldig sentralt i vår prosess også. Vi jobbet etter et prinsipp som kalles ‘trunk-based development’ (2), som kort sagt går ut på at vi lager en ny branch i vårt repo når vi starter arbeid på en ny funksjonalitet, jobber og ferdigstiller den funksjonaliteten på vår egen branch, og så merger vi den med main når den er ferdig utviklet og testet.

I tillegg til dette brukte vi GitHub’s workflow-verktøy til å kvalitetssikre og kjøre linting-tester på vår kode. Her er en modell av vår utviklingsprosess grovt sett:

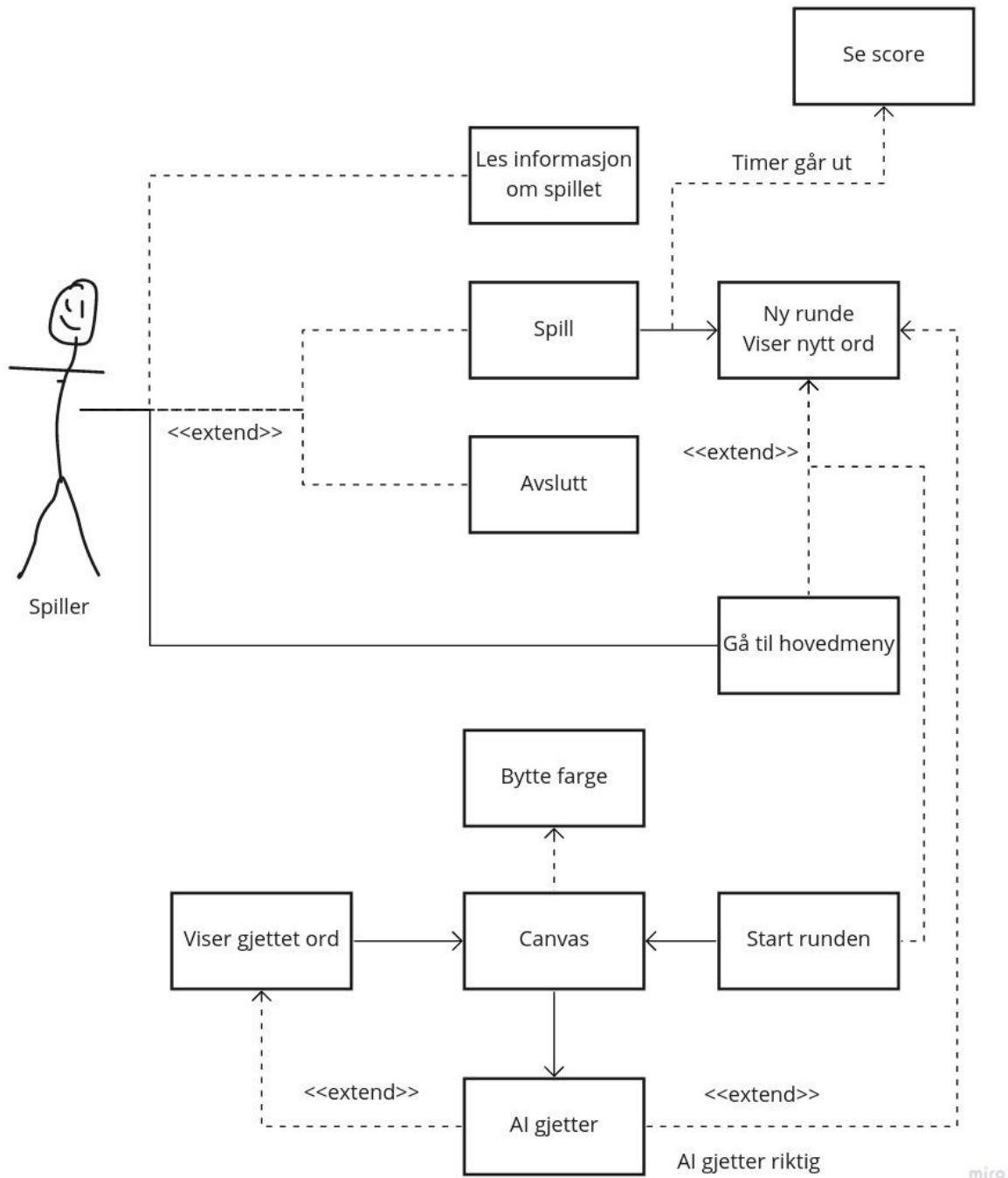


Kravspek

Programmet vi utvikler må være et funksjonelt tegneprogram, der bruker kan tegne med mus (evt. touchpad, touchscreen, digital penn, eller lignende) med farge. Bruker skal få begreper presentert som det er tenkt at bruker skal forsøke å tegne. Programmet skal kunne lese av tegningene til bruker, og gjette hva bruker har tegnet. Om den gjetter riktig må lerretet bli slettet, og nytt ord gis ut. Programmet må avslutte spill etter en viss mengde tid, og gi bruker poengsum. Bruker må kunne avslutte spill, se forklaring på hvordan en spiller, og faktisk spille spillet. Bruker må kunne se ord de skal tegne, ord som vår CNN gjetter, og hvor mye tid de har igjen.

Vårt prosjekt har relativt liten frontend med bare en type samhandling mellom bruker og software, men vi har likevel nok til å sette opp en bra use case. På den andre siden er det positivt at vårt system ikke er veldig sårbart med henhold til sikkerhet!

Vi kan forestille oss en mer utvidet versjon av spillet, der det ligger tilgjengelig på en server (f.eks. gjennom openstack) og kan spilles i browser. Denne versjonen vil ikke ha særlig større use case for en vanlig user, men vil ha en del flere krav til operasjon og spesielt sikkerhet - inndata fra bruker, IP-adresser, etc vil være tilgjengelig uten bra sikring, og angrep som for eksempel man-in-the-middle vil være mer seriøse affærer.



For mer om våre kravspesifikasjoner, se [Kravspek-dokumentet](#) lagt til som vedlegg.

Teknologi

Vi bestemte tidlig at vi ville utvikle prosjektet med GitHub som vårt organisering og versjonskontrollsysten. Vi endte også opp med å bruke GitHub's verktøy for å kjøre issue tracker, administrere product backlog og generell organisering av arbeidsoppdeling, etter å ha vurdert å bruke Jira og andre lignende verktøy - siden GitHub hadde en solid implementasjon som var enkel å bruke, og ikke trengte ekstra arbeid for oppsett eller tilkobling til systemet.

Verktøy

Navn	Formål	Bruk
GitHub	Tar seg av hosting og distribuering av vår kode	Prosjektorganisering
Visual Studio Code	Vår IDE for kodeutvikling i C++, Golang og Python i Linux	Koding av Back-end og Front-end
Visual Studio	IDE for kodeutvikling av C++ i Windows	Koding av Back-end og Front-end
GoLand	IDE for kodeutvikling av Golang i Windows	Koding av Back-end
Instagantt	Verktøy for design av Gantt-skjema	Prosjektorganisering
Google Docs	Verktøy for samarbeidende dokumentskriving	Prosjektrapport
Toggl	Verktøy for registrering av tidsbruk	Prosjektorganisering og rapport
GitHub issue tracker	Git verktøy for scrum-organisering og generell issue tracking	Prosjektorganisering
GitHub workflows	Git verktøy for kvalitetstesting av kode gjennom linting og kjøretester	Testing verktøy
Miro	Online whiteboard for design av modeller og wireframes	UML, wireframes, nettverksoppsett, etc
Discord	Online kommunikasjons-app	Prosjektorganisering, møter
CMake	Verktøy for organisering og kjøring av C++ kode	Prosjektorganisering
SerpAPI	API for innhenting av resultater fra Google søkemotor	Innhenting av datasett

Under utvikling brukte vi Visual Studio Code og Visual Studio som våre IDEer. Visual Studio ble brukt fordi noen gruppemedlemmer jobbet på Windows. VSC ble valgt på bakgrunn av dens lette og enkle UI og det faktum at den fungerer veldig bra for alle språkene vi tok i bruk gjennom prosjektarbeidet.

Eksterne bibliotek

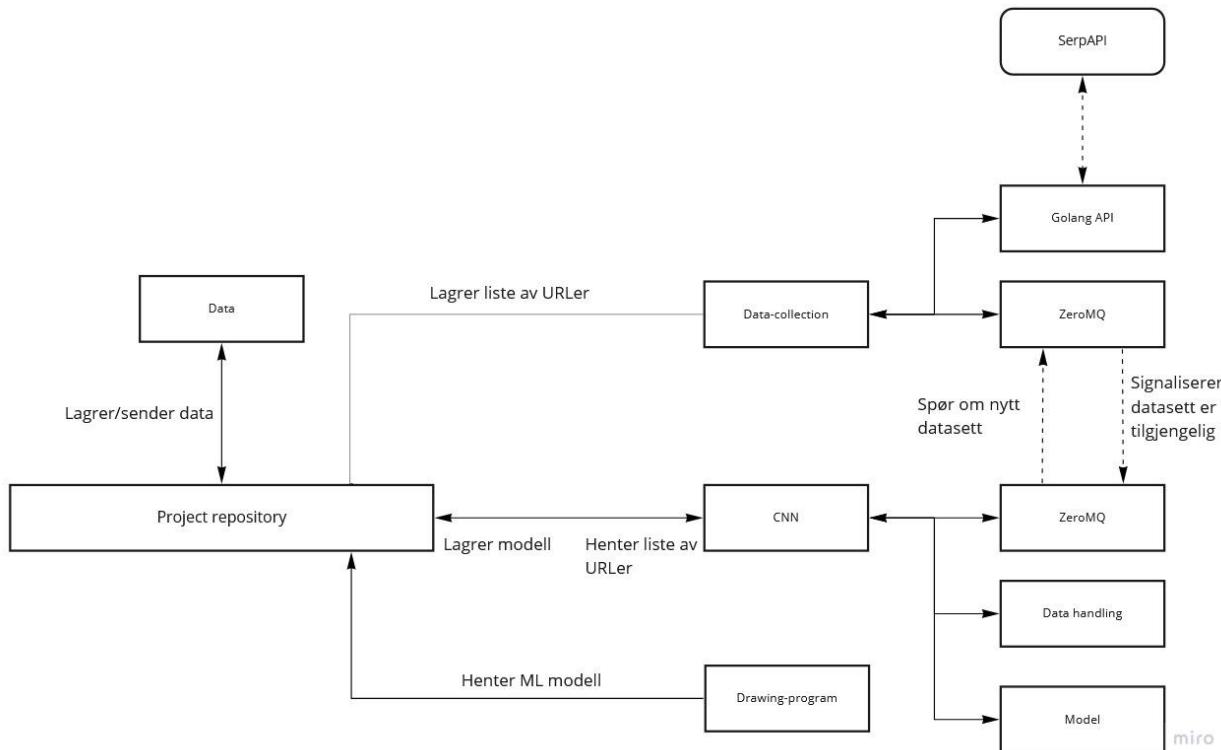
Navn	Bruk til
Tensorflow	Design og kjøring av CNN
OpenCV	Forhåndsbehandling av bilde
scikit	Bruk for forhåndsbehandling av bilder og CNN-modellens Klassifikasjonsrapport samt forvirringsmatrise
Keras	Definere lag i CNN-TF modellen
Numpy	Matematikk i CNN-TF
Matplotlib	Tegne graf av resultater fra vår CNN-modell
mlpack	Maskinlærings-bibliotek for C++
Armadillo	Databehandling IFT datatyper (matriser, rader), matematikk i mlpack
CURL	CURL er brukt til å laste ned bilder fra URL
pickle	Python object serialization
stb_image_write	Lagring av bruker-tegning for analyse
Glad	OpenGL loader-generator
GLFW	OpenGL kontekst og window management
GLM	Matematikk for grafikkprogramvare
Python C++ API	For kjøring av Python scripts i C++ kode
ZeroMQ	Kommunikasjon mellom moduler

Systemarkitektur

Helt fra starten av prosjektet, under skriving av prosjektplanen, bestemte vi oss for å dele prosjektet inn i flere moduler. Dette valget ble gjort med i hovedsak tre motivasjoner som bakgrunn:

- For det første visste vi at vi ville bruke flere ulike programmeringsspråk - til å begynne med Golang og C++ - i kodebasen vår. Vi visste også at bare datainnsamlingen ville absolutt trenge Golang, mens vi planla resten til å bruke C++. Det gav derfor mening å dele ut datainnsamlingen i egen modul.
- Ved å dele opp prosjektet i flere moduler, kunne vi jobbe med disse modulene isolert fra resten av kodebasen, som gjorde det mye enklere å teste, finne errorer og logiske feil, og sikre at implementasjonen av hver enkel modul blir riktig gjort.
- Som følge av dette blir det også mye enklere å dele opp arbeidet mellom prosjektmedlemmene - hver av oss kunne jobbe på ulike moduler og ikke tråkke på hverandres tær i kodingsprosessen. Kombinert med trunk-basert utvikling som fokuserer på å branche hvert issue, gjorde dette det mye enklere å kode sammen og reduserte merge issues.

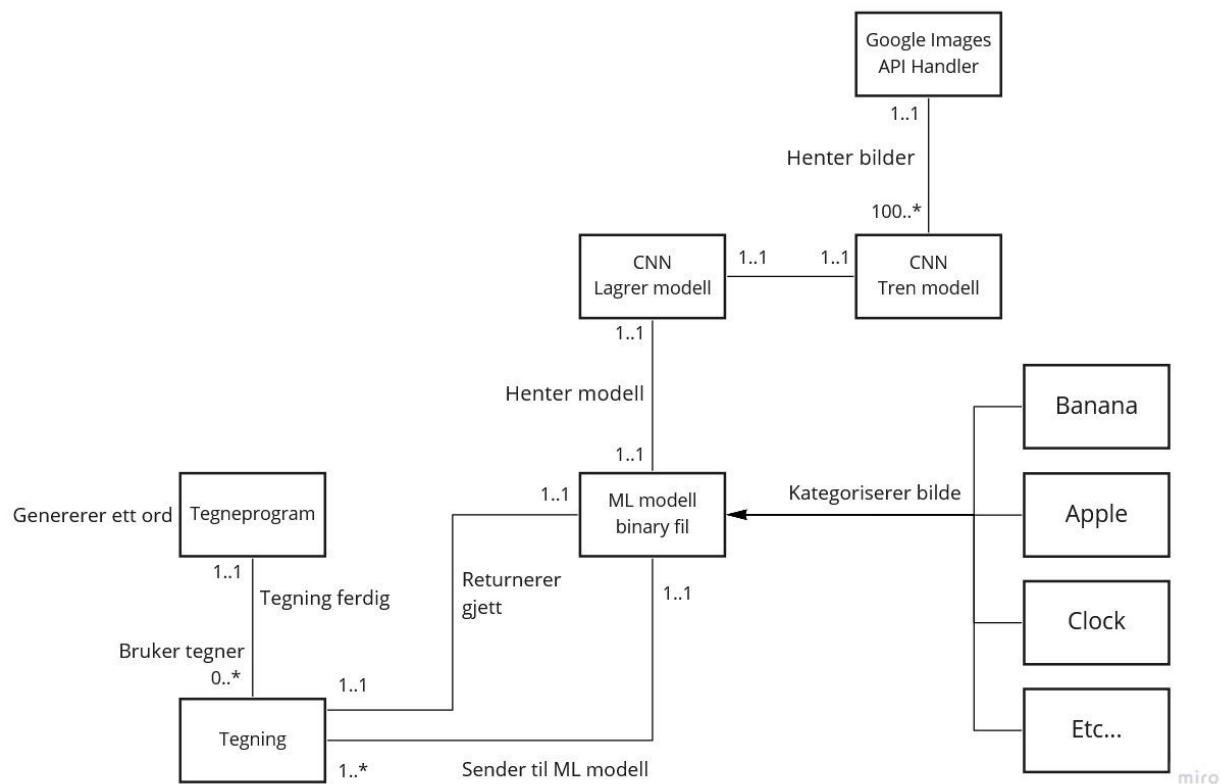
På bakgrunn av dette, og etter noen iterasjoner (som kan sees i vedlegget [arkitekturskisse](#)) endte vi opp med denne prosjektstrukturen:



Inndeling i moduler kommer derimot med en ulempe. Den første ulempen er logisk - det kan være vanskelig å forestille seg akkurat hvordan to moduler skal kobles sammen. Skal data-collection modulen sende array, liste eller bool til CNN modulen? Akkurat hvor skal modulene kobles? Vi støtte på en del slike små problemer til tross for at vi forsøkte å planlegge hvordan dette skulle se ut på forhånd. Vi måtte endre på og justere oppsettet inne i hver modul i kodeprosessen - men disse endringene ble ikke alltid justert for andre moduler som kobler til den modulen.

Den andre ulempen vi hadde med arkitekturoppsettet vårt var ganske enkelt i kommunikasjonen mellom språk. Fra starten av hadde vi planlagt minst to språk, der vi ville hente inn data med Golang, og så jobbe på det med C++. For å få til dette, trengte vi en metode for å kommunisere mellom disse to språkene.

Kommunikasjon mellom moduler – ZeroMQ



Over har vi domenemodellen for vårt produkt, som viser hvordan modulene skal kobles sammen. Fra starten av dette prosjektet visste vi at vi ønsket å kode det i flere ulike språk, og dele det inn i slike ulike moduler – dette var allerede på plass i prosjektplan-dokumentet. Særlig visste vi at vi ville bruke C++ til spillet, og Golang for API til datainnhenting. Vi tenkte i den tid også å bruke bare C++ for koding av maskinlærings-delen av prosjektet. I løpet av arbeidet

utvidet vi også bruken av dette verktøyet ved å legge til Python for definering og trening av ML-modellen.

For å få til dette, ble det nødvendig å implementere en kommunikasjonsmåte for disse ulike modulene. Noe data kunne greit ta mellomveien til disk – for eksempel datasettet som blir funnet av Golang og skal brukes av Python - men selv disse trenger et slags signal på at datasettet er ferdig nedlastet. Andre ting bør kommuniseres direkte – vi kunne kanskje lagret listen over begreper vi vil hente inn datasett for i en fil fra C++ og lese den i Golang, men det er mye enklere og mer effektivt å sende direkte uten mellomsteg.

For å implementere dette, bestemte vi oss for å bruke ZeroMQ. Vi hadde litt erfaring med dette fra før, men ikke særlig mye – og ingenting i Golang, C++ eller Python. Vi tenkte likevel at dette ville gå veldig greit, og satte av relativt lite tid til det.

Dessverre går ikke alt som planlagt. Problemet vi støtte på var koblet til C++-implementasjonen av ZeroMQ. Vi startet arbeide med en utdatert bibliotek-implementasjon av ZeroMQ, men oppdaget raskt at vi måtte endre til en nyere implementasjon. Denne matchet ikke dokumentasjonen på ZeroMQ's offisielle dokumentasjon, ei heller hadde den tilstrekkelig dokumentasjon selv, så vi måtte gjøre litt prøving og feiling for å få denne til å fungere.

Når vi til slutt fikk byttet til det mer oppdaterte biblioteket, gikk implementasjonen av ZeroMQ veldig greit, og vi utvidet til slutt også ZeroMQ til å brukes for å sende data mellom tegneprogrammet og vår CNN-modell. Vi hadde derimot fremdeles problemer med ZeroMQ relatert til build-prosessen - vi fikk ikke dette til å fungere på Windows, og endte opp med å prioritere kjøring i Linux-miljø. Vi valgte ikke å fikse dette, da vi uansett kunne bruke Linux som utviklingsplattform og hadde stort tidspress på dette tidspunktet.

Implementasjon

Innhenting av datasett

For innhenting av datasett lagde vi vår egen API i Golang. Som diskutert over, ville vi hente inn bilder automatisk fra Google Images gjennom SerpAPI ved å lagre lenkene til disse bildene i egne filer. Måten vi satte det opp på var at for hvert ord vi ønsket å trenne på, sendte vi de inn fra Python til Golang. Der hentet vi inn de 100 første resultatene på Google og lagret URLene til thumbnailene i en tekstfil. Når vår API var ferdig med å hente inn disse resultatene for alle ordene som ble sendt inn, ga den et signal tilbake til vårt C++ program at datainnhenting er ferdig.

For kommunikasjon med SerpAPI brukte vår Golang modul bare enkle request metoder, og bestemte hvilken URL den skal sende requester til gjennom å kombinere ordene vi sender inn med en ferdigbygd URL-string basert på SerpAPI's gitte parameter. Dette betyr at dersom

SerpAPI endrer formatet for sine API-endpunkt, må vi inn å endre disse manuelt for at disse skal fortsette å fungere. Dette er en risiko med alle eksterne APIer.

Convolutional Neural Network

Datasett og forhåndsbehandling av dette

Som nevnt i introduksjonen hentet vi inn datasett automatisk fra Google Bilder. Det betyr at ikke alle resultatene ville være veldig gode. Bildene har også ganske ulik metadata. Vi kom opp med en del ideer for å hjelpe på dette.

- For det første må alle bilder ha like dimensjoner for at vårt nettverk skal kunne trenere på dem. Vi valgte å satse på lik bredde og høyde - 128px ganger 128px - og endrer størrelsen på alle bildene i datasettet vårt til dette. Vi valgte å endre til en ganske liten størrelse fordi vi heller ville skalere ned store bilder enn opp små - det gjør at trening og kjøring av vår modell går mye raskere fordi den har færre datapunkter å se på, men kan også tape noe treffsikkerhet.
 - Som en tilleggsfunksjon kunne vi også forkaste alle bilder der dimensjonene avviker kraftig fra vårt kvadratiske mål - for eksempel der høyde eller bredde er mer enn 1.5 ganger så stor som den andre dimensjonen. Alternativt kunne vi tatt dette enda lengre og først regnet gjennomsnitts-dimensjonen av vårt totale datasett, og brukt den som standard for vårt prosjekt. Deretter forkaste bilder med stort avvik fra denne dimensjonen.
- Deretter kan vi normalisere farge-verdiene på våre bilder, slik at de er noenlunde like sterke. Dette vil gjøre at datasettet blir mer konsistent, og vil kunne jevne ut individuelle forskjeller mellom bilder i hver klasse. Vi valgte ikke å bruke ressurser på å implementere dette i vårt prosjekt, delvis fordi vi tenker at det å differensiere på små forskjeller i farge mellom klasser blir lite relevant.
- Det er en form for forhåndsbehandling vi kan gjøre før vi i det hele tatt laster ned datasettet - nemlig av selve frasene vi sender til serpAPI for å bestemme google bilde-søket som skal gjøres! Med andre ord, endringer av hva vi søker på. Vi kan for eksempel søke på 'apple', 'red apple' eller 'isolated red apple drawing'. Ulike søk vil gi ulike resultater - det vi kan forsøke å finne er adjektiv som konsistent gir bedre resultater om vi sender de med i frasene til SerpAPI.
- Vi vurderte å isolere elementene vi faktisk er interessert i for hvert bilde. Dette gjøres vanligvis i en prosess som kalles 'masking'. En slik prosess går ut på at vi først finner denne delen av bildet vi helst vil ha gjennom å sammenligne med et eksempel - så om vi vil ha en bil, har vi et eksempel som bare er en bil uten bakgrunn. Vi sammenligner dette eksempelet med hvert bilde i datasettet, og på denne måten finner vi den delen av hvert bilde som ligner mest på eksempelet - vanligvis vil det være da faktisk en bil i hvert bilde. Dersom flere biler er på bildet, vil den bilen som intelligansen synes ligner mest bli valgt. Det neste vi gjør er å bruke formen til bilen vi fann i bildet til å lage en mask til dette bildet, der alt rundt bilen blir satt svart, og bilen til hvit. Til slutt ganger vi sammen mask med det originale bildet, som vil føre til at alle regioner som er hvit i masken (der bilen

er) får sine originale farger tilbake (ganger verdien med 1), og de andre regionene (bakgrunnsstøy) fortsatt er helt svart (ganger verdien med 0). Vi har dermed isolert bare bilen i bildet, og støy i form av farger eller linjer fra bak- eller forgrunnselementer er eliminert!

- Vi tenkte også på å fjerne særlig avvikende bilder fra datasettet. Dette er noe utfordrende uten å manuelt gripe inn, men vi hadde en idé til dette. Vi forklarte nettop masking, og hvordan det brukes til å fjerne bakgrunn fra bilder. Men vi kan bruke deler av prosessen til noe lignende! Først trener vi noen ganger på datasettet og holder styr på hvilke bilder som konsistent blir gjettet riktig. Det mest konsistente bildet er valgt som 'stadarden' for det individuelle datasettet - et 'standardeple', en 'standardbil', osv. Når vi vet hvilket bilde dette er, kan vi etter masking-prosessen sammenligne noen kvaliteter i dette bilde med de andre bildene i datasettet, og forkaste de bildene som avviker særlig mye. Å finne ut akkurat hvilke kvaliteter vi vil se på, og hvor stort avvike skal være, er derimot ikke enkelt. I tillegg vil denne fremgangsmåten ikke fungere dersom et dårlig eksempel skulle bli valgt som standard - dog tror vi dette sjeldent vil være et problem.

Den forhåndsbehandlingen vi valgte til slutt gikk altså ut på å bruke adjektiver ved definisjon av nye begreper til datasett, samt endre bilde-dimensjoner etter nedlasting slik at alle er like. Vi diskuterer bakrunnen til dette nærmere under gjennomgang av resultater, men dette er ikke bare på grunn av manglende tid og ressurser.

Design av nettverks-modell

Når datasettet var klart til å trenes på, var neste steg å konstruere en modell. Vår første plan var å gjøre all design, trening og bruk av vårt nettverk i C++. Vi lette tidlig etter potensielle biblioteker for det formålet, og kom fram til at mlpack ville være en bra kandidat. Det virket som et bibliotek med akkurat de trekkene som vi trengte for å lage nettverket vårt. Vi gikk raskt i gang med å sette opp en modell og forsøke å trenen den greit på de nedlastede datasettene våre.

Dessverre møtte vi veldig mange problemer med mlpack som gjorde prosessen vanskelig. Det tok for det første, ganske lang tid å installere og sette det opp i CMake slik at ville linke til programmet vårt, mer tid enn vi hadde sett for oss under planlegging. I tillegg støttet vi på et fatalt problem når vi først hadde suksess med installasjon og oppsett av mlpack - den nektet å trenen nettverket på mer enn en tråd.

For en liten bakgrunn om trening av CNN: ideelt sett bør nettverk trenes på GPU. Dette er fordi slike nettverk er designet med flere lag, der hvert lag har mange noder. Datasettet som jobbes på blir likt mellom alle nodene, og hver node tar seg av sin lille del av datasettet. Deretter deler hver node opp sitt datasett slik at det igjen blir fordelt til hver node i det neste laget. Slik blir datasettet kryss-polinert gjennom mange lag i nettverksprosessen. Men dette innebærer også at treningen helst må kunne utføre mange beregninger samtidig.

Selv om en prosessor er kraftig, kan den bare gjøre en ting per tråd til enhver tid. Dette kan vi se på f.eks. Windows Task Manager, under 'Logical processors'. Det betyr at på det meste ville den sterkeste maskinen vi har tilgang til kunne utføre 12 beregninger samtidig. Dette er ganske tregt sammenlignet med en GPU som er datamaskinens mester av parallelisering og kan klare mange flere beregninger (6). Vi vil derfor ideelt sett trenre vår modell på GPU, men kunne muligens ta til takke med en multi-threaded CPU om nødvendig.

Det er da kanskje ikke veldig overraskende at å bruke bare en eneste tråd til dette til sammenligning ikke er tilstrekkelig. De ble mange lange kvelder med debugging og leting etter løsninger, blant annet gjennom posting av issues på den offisielle [mlpack GitHub](#), men vi klarte ikke å finne en løsning. Vi trodde lenge at problemet var relatert til linking av [BLAS](#) som er løsningen mlpack bruker til å kjøre selve de matematiske operasjonene som utgjør treningen, men oppdaget til slutt at mlpack klarte å bruke BLAS - og alle prosessor-kjernene våre - til gjetting av datasett. Den nektet bare å bruke mer enn en til den mer tid-krevende treningsprosessen.

Istedentfor å stange hodet mot veggen, valgte vi å flytte oss over til noe som var mer kjent; TensorFlow i Python. Dette har vi erfaring med fra AI faget og vi visste også hvordan vi kunne få kjørt modell treningen på NVIDIA GPUer. Det tok mindre tid å skrive om programmet til Python enn hele debugging problemet med mlpack.

Til tross for at vi endte opp med å løse denne oppgaven gjennom to separate programmer som kommuniserer sammen, ble det likevel en mye mer effektiv løsning fordi den bruker maskinens ressurser fullt ut, og kan trenre enten på GPU om [CUDA](#) er installert, eller alle tråder dersom CPU må brukes. Det er likevel en stor forskjell på GPU og CPU multi-threading - vår ferdig utviklet modell tar bare 48 sekunder å fullføre på GPU, **med** kryssvalidering. Kryssvalidering går ut på å trenre modellen flere ganger - vi valgte fem - og sammenligne resultatet for hver modell. Dette sjekker om modellen vår er konsistent, eller om det er problemer med datasettet som forårsaker stor variasjon i resultatet. Kryssvalidering bruker også veldig mye ressurser, og det tok hele 33 minutter å kjøre på en 8-kjerners CPU! Dette er et praktisk eksempel på hvor mye bedre egnet GPUer er for trening av modeller.

Modellen vi bygde baserer seg på convolutional lag. Måten disse lagene funker på er relativt komplisert, men kan forenkles slik: vi sender inn bilder, samt spesifikke parameter til disse lagene. Disse parameterne kan deles inn i to deler: nevral-nettverk parameter, og convolutional parameter. I den første kategorien har vi parametre som antall noder i hvert lag, som bestemmer hvor kompleks det laget er. Som allerede nevnt over, deles datasett etter antall noder, så jo flere noder per lag, jo mer ressurs-krevende vil modellen være. Og det samme gjelder for antall lag i modellen. Samtidig vil vi ikke ha for få noder - flere noder gir bedre konsistens fra lag til lag, da data blir mer grundig spredd ut i modellen.

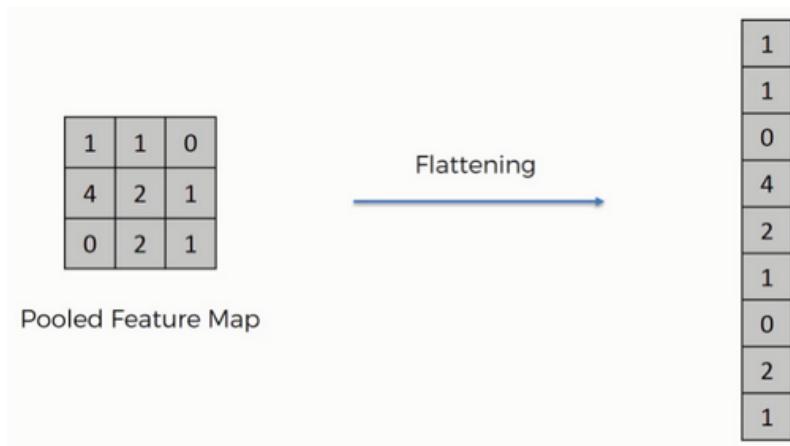
CPU1	3.1%
CPU2	5.0%
CPU3	7.1%
CPU4	6.1%
CPU5	100.0%
CPU6	4.1%
CPU7	6.1%
CPU8	6.1%
CPU9	6.1%
CPU10	5.1%
CPU11	5.1%
CPU12	6.0%

Den andre kategorien av parametre er spesifikke til convolutional lag. Disse lagene er spesielt godt egnet til flerdimensjonale datasett, og passer derfor perfekt for våre bilder. Parametre her inkluderer size og stride. Disse parametre bestemmer hvor stor del av datasettet som skal ses på om gangen - for våre bilder er det en x ganger y seksjon av piksler, for eksempel 3x3, og hvor langt modellen skal flytte seg mellom hver del. Om vi for eksempel ser på 3x3 pixler samtidig, men stride er 1, vil den bare flytte senteret av 3x3 ruten en pixel, og alle pixler utenom de ytterste to radene i hver dimensjon blir evaluert tre ganger. Setter vi stride til 3 derimot, vil alle pixler bare ses på en gang.

```
# layer 1
keras.layers.convolutional.Conv2D(filters=32, kernel_size=3, input_shape=dict.SHAPe, activation='relu'),
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2)),
# layer 2
keras.layers.convolutional.Conv2D(filters=32, kernel_size=3, activation='relu'),
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2)),
# layer 3
keras.layers.convolutional.Conv2D(filters=64, kernel_size=3, activation='relu'),
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2)),
# flatten
keras.layers.core.Flatten(),
# layer 4
keras.layers.core.Dense(units=64, activation='relu'),
keras.layers.core.Dropout(rate=0.5),
# layer 5
keras.layers.core.Dense(units=dict.DATASET_AMOUNT, activation='sigmoid')
```

Koden over viser hvordan vi har bygd opp modellen vår. Det starter med 3 convolutional lag med 2x2 max-pooling. I en convolutional lag er det 3 parameterer som går igjen; filters, kernel_size, og activation. filters parameteret spesifiserer dybden i hvert lag, også kalt noder, mens kernel_size parameteret spesifiserer størrelsen på hvert filter. Med activation parameteret kan vi definere hvilken funksjon vi vil bruke for å kalkulerer weight sum på en input node til en output node (4). Max-pooling reduserer bilde størrelsen utifra poolsize, 2x2 halverer da bilde størrelsen og gjøre neste steg mye raskere.

Etter convolutional lagene har vi en flatten funksjon som konverterer et convolutional lag til et dense lag. Det vil si at den gjør en matrise om til en kolonne som illustrert på bilde under.



Bilde er hentet fra [SuperDataScience](#)

Etter å ha flatet ut convolutional lag har vi to dense lag med et nytt parameter; units. Det har den samme funksjonen som filter parameteret og definerer dybden/noder. Etter det første dense lag har vi en dropout funksjon. Dette blir brukt for å fjerne noen tilfeldige noder og er viktig for at modellen ikke skal bli overfitted. Det vil si at en modell er trenet for mye på trenings datasette og ender opp med å gi dårlig nøyaktighet på andre bilder. Siste lag må ha dybde på størrelsen med antall klasser som vi har trenet på (apple, banana, cheese, osv...) og vil ende opp med å holde prediksjonsverdien for hver klasse.

Modellbygging går ut på trial-and-error, det vil si at det ikke finnes en korrekt metode for å bygge modeller, men at man må prøve seg fram. Vi fant denne modellen til å gi bra resultat, men også kompakt nok til å kunne trenes på kort tid. Dette er viktig når man konstruerer en modell ettersom man alltid må sjekke resultatet etter hver endring for å se forskjell og treningstiden har mye å si når man skal teste flere parametere.

Vi håpte på å kunne importere modellen trenet i TensorFlow til mpack, og kjøre gjettingen derfra. Dessverre funket ikke dette fordi mpack ikke hadde implementert denne funksjonen enda. Vår endelige løsning ble å heller å ta i bruk ZeroMQ enda en gang, og kjøre vår gjetting i Python gjennom bruk av Python bibliotek for C++.

```
void Model::initScript() {
    Py_Initialize();
    PyRun_SimpleString(("exec(open(\"" + filename + "\").read()).c_str()");
    Py_Finalize();
}
```

Når vår CNN er ferdig å behandle bildet og gjette hvilken klasse det passer i, gjør den seg klar til å sende det tilbake, og vi bruker en klient-funksjon i C++ til å be om resultatet:

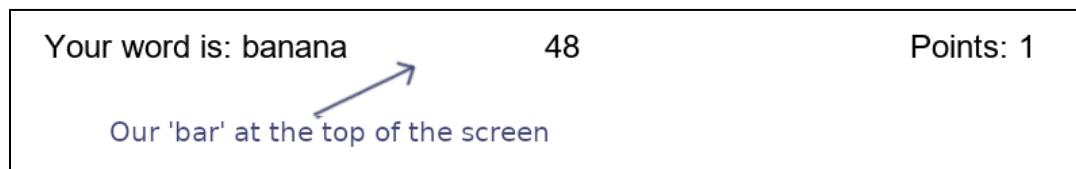
```
void Model::ping(const std::string cmd) {
    // setup connection
    zmq::context_t ctx(1);
    zmq::socket_t sock(ctx, zmq::socket_type::req);
    sock.connect("tcp://localhost:5959");
    // ping server
    sock.send(zmq::buffer(cmd), zmq::send_flags::dontwait);
    // get prediction
    zmq::message_t msg;
    zmq::recv_result_t res = sock.recv(msg);
    prediction = msg.to_string();
}
```

Tegneprogram

Brukergrensesnitt

For vårt brukergrensesnitt ønsket vi noe ganske lett og enkelt, og ideen med hvit bakgrunn og ingen toolbars eller oppdeling av skjermen i seksjoner sto sentralt fra starten av. Som nevnt ble vi inspirert til å gjøre dette prosjektet av Quickdraw, og vi hentet også ideer derfra. For elementer som farge-bruk, tok vi ideer fra MSPaint som tross alt også er et kjent og kjært tegneprogram. Dette hadde to fordeler: for det første fikk vi en enkel UI som er grei å bruke, og for det andre er det god sjanse for at våre brukere er kjent med lignende brukergrensesnitt fra før.

I tillegg til inspirasjon fra eksisterende programvare, brukte vi også noe tid på å vurdere prinsipper rundt design og hvordan det ville påvirke vårt program. For eksempel brukte vi ideene om constraints og affordances til å bestemme hvordan vi skulle sette opp brukergrensesnittet, og hvilken del av skjermen vi skal lese av for gjetting. Tanken vår var for eksempel at om vi hadde informasjon i en linje på toppen av skjermen som ordet som skal skrives og hva som skal gjettes, vil dette gi både en logisk constraint og til en viss grad kulturell constraint til bruker (3). Logisk fordi bruker vil kunne se at den delen av vinduet er i bruk og derfor trolig ikke vil være en god plass å tegne, kulturell fordi bruker gjerne er vant med at det funker på denne måten:

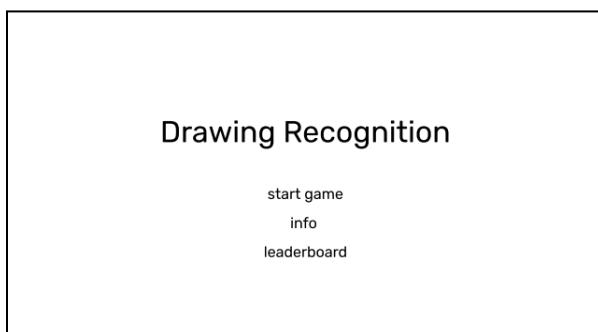


Før vi startet med å kode tegneprogrammet, gikk vi gjennom en prosess med design og testing av brukergrensesnittet. Dette var for å få et bedre innblikk i hvordan programmet ville bli brukt. Denne prosessen ble delt inn i to faser:

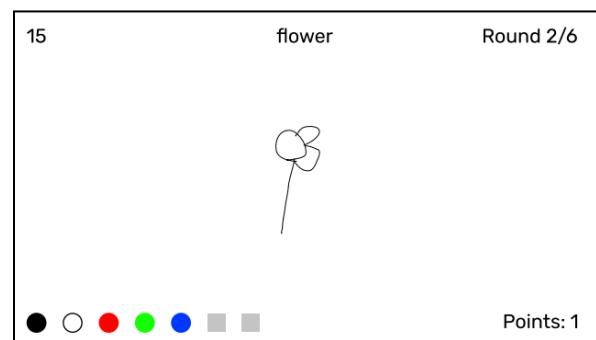
1. Layout og navigasjon: UI-elementene, som knapper og tegneverktøy, må være plassert på et sted der det virker naturlig for brukeren. Navigasjonen skal ikke være knotete, altså det skal ikke være mange ledd for å komme seg til en del av applikasjonen til en annen.
2. Design: Designet skal lages på en måte slik at det verken er distraherende eller utydelig for brukeren.

På slutten av hver fase utførte vi brukertesting på det vi hadde lagd i fasen. Vi hadde 4 testpersoner, som var med i begge fasene. Disse personene var en blanding av kvinner og menn. De hadde forskjellige nivåer av erfaring med både spill og tegneprogram fra før av. Vi fikk derfor testet et bredt spekter av mennesker. Programmet vårt har ikke en spesiell målgruppe vi sikter mot, så feedback fra en variert gruppe personer var veldig nyttig under denne testingen.

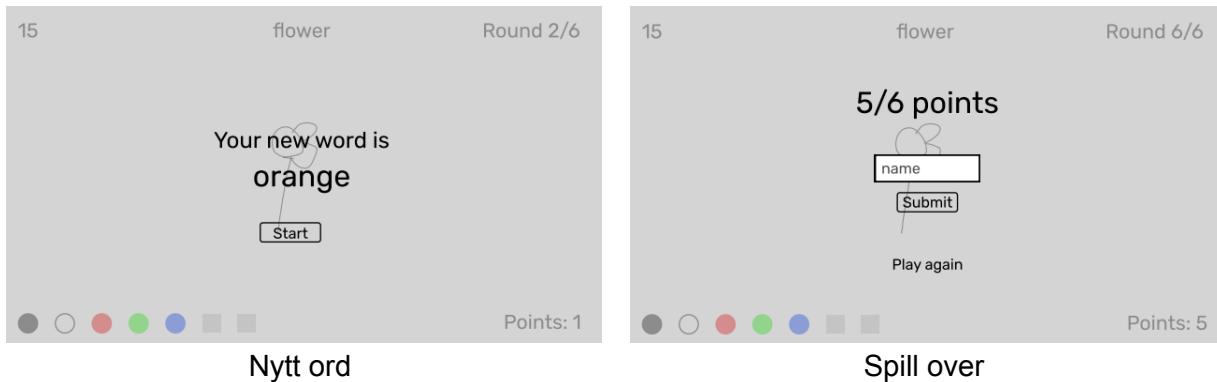
Vi gikk for en testmetode der vi ga testpersonene en oppgave de skulle utføre alene utifra wireframes/prototypen. De brukte sine egne enheter til dette. De ble så stilt spørsmål relatert til hvordan de opplevde utførelsen av oppgaven. Spørsmålene var åpne, men vi stilte oppfølgingsspørsmål for å få avklaring på meningene deres hvis noe var uklart. Ved å gjøre det på denne måten ga vi testpersonene en frihet til å utforske prototypen i sin egen hastighet og sitt eget miljø. Altså i en situasjon de reelt kunne brukt det ferdige programmet i. Derfor hadde de også et godt grunnlag til å svare på spørsmålene i intervjufasen etter, og hadde muligens fått et annet inntrykk av prototypen enn om vi hadde overvåket prosessen. En ting vi kunne gjort annerledes var å stille flere oppfølgingsspørsmål når de kun sa at noe fungerte "greit" eller var "bra". Dette kunne hjelpt oss med å bedre forstå hva som fungerte bra i prototypen, så vi kunne brukt det til videre utvikling og implementering av brukergrensesnittet.



Hovedmeny



Spillrunde



Vi startet fase 1 med å lage wireframes (vist i bildene over), altså et enkelt oppsett av hvor de forskjellige elementene skulle være plassert. Som sagt tidligere var fokuset her layout og navigasjon, så vi brukte enkel tekst og kun farger der det var behov (se "Spillrunde" i bildene over). Oppgaven testpersonene ble gitt var: "Start et spill, fullfør runden, og legg til poengsummen din på leaderboardet". Alle klarte det, og hadde ingen forslag til forbedringer til selve navigasjonen. Én tilbakemelding vi fikk var at i "Spillrunde" burde timeren og ordet bytte plass, ettersom ordet blir vist før runden, og ikke er like viktig at er i midten.

Hovedmeny

Drawing Recognition

Start game

About

High scores

Exit

Spill

Your word is: flower 15 Round 2/6

... is it a flower?

Om spillet

About

You have 30 seconds to draw the word that shows up on the screen. For each drawing the game guesses what are, you will get a point.

Back

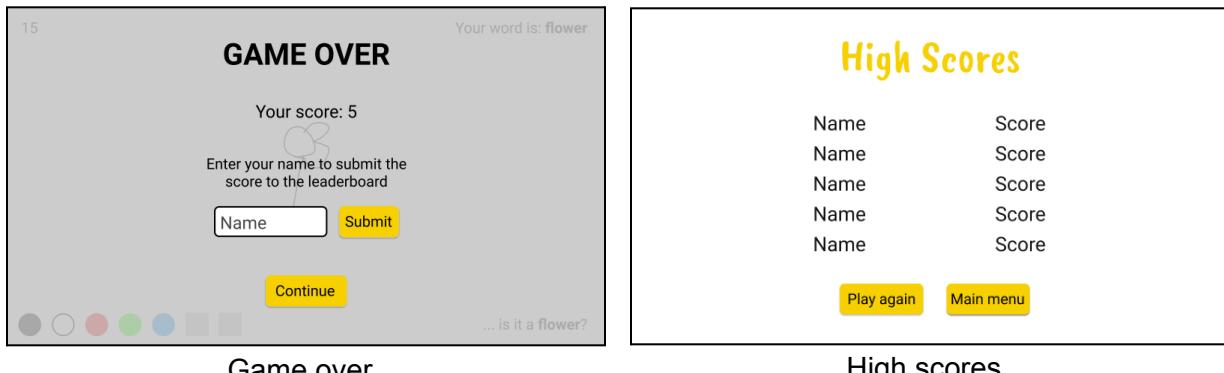
Nytt ord

Yay, your drawing was guessed correctly!
Total points: 1

Your new word is
ORANGE

Start Main menu

... is it a flower?



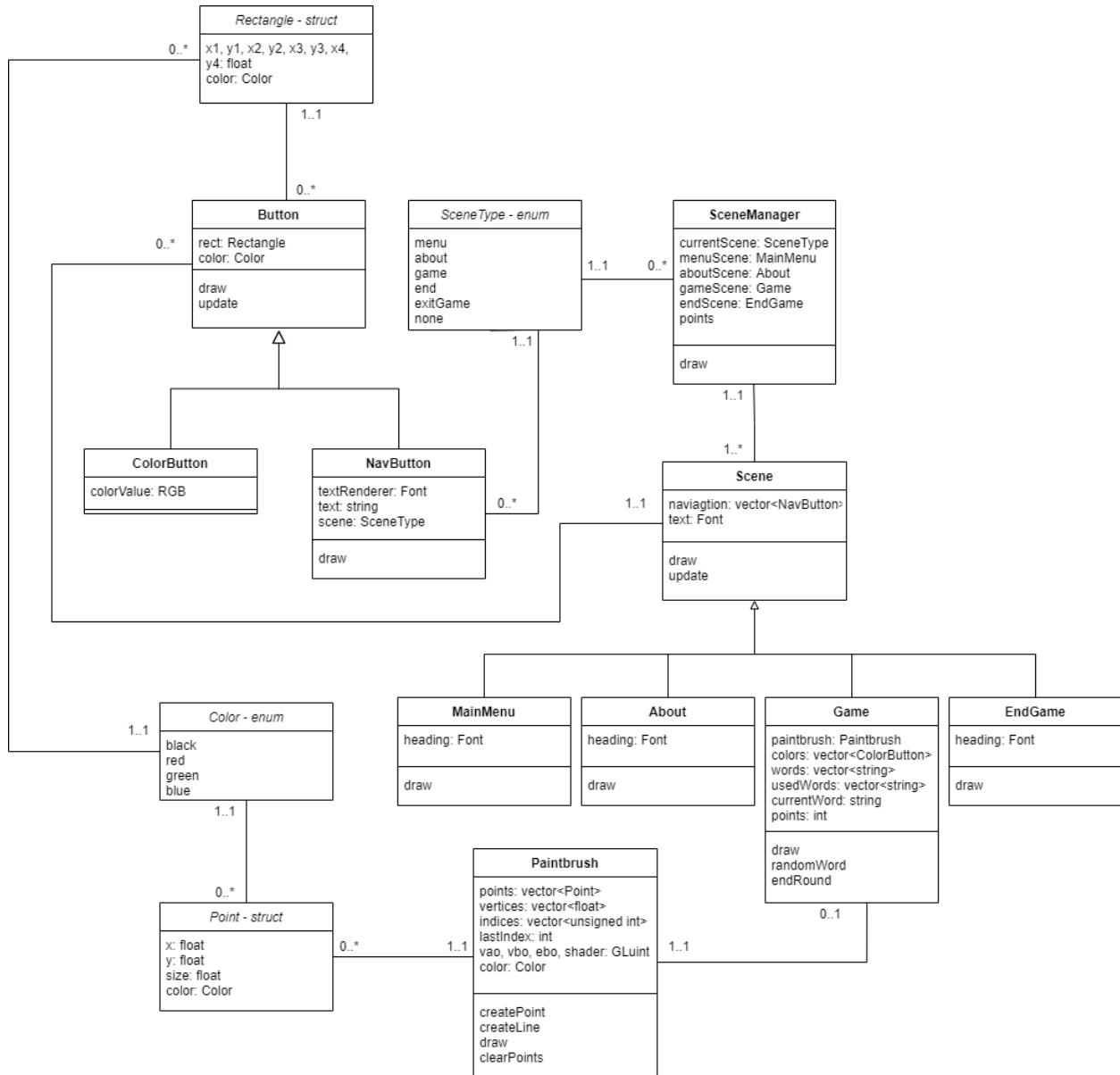
I fase 2 ble det laget en prototype, altså en mer ferdig versjon av programmet enn i fase 1. Vi så først på tilbakemeldingene fra brukertestingene i fase 1, for å få layout og navigasjonen mer på plass. Vi byttet plasseringen til ordet og timeren, ettersom vi fikk tilbakemelding om at det kunne være vanskelig å få med seg timeren da den var plassert på siden. Ordet ble vist på en egen skjerm først, og det var derfor mindre viktig å ha det i midten. I tillegg til dette gjorde vi ordet **bold**, slik at det fremdeles skulle være enkelt å få det med seg.

Da navigasjonen og layouten var på plass, begynte vi å se mer på designet. Målet var å sjekke om de grafiske elementene virket distraherende eller utydelig for brukeren. Vi gikk for et enkelt design ettersom ingen av oss har noe særlig erfaring med design i OpenGL fra før av. Vi innførte derfor kun elementer som font, farge og enkle knapper. Vi valgte en gul farge som en rød tråd gjennom hele applikasjonen, ettersom gult ofte er forbundet med kreativitet.

Etter dette hadde vi en ny brukertesting, der testpersonene ble gitt denne oppgaven: "Start et spill og avbryt det etter første ord-runde er ferdig". Den nye navigasjons-delen (se "Nytt ord" i bildene over) fungerte bra i følge testerne. I forhold til designet fikk vi tilbakemelding om at den gule fonten hadde for dårlig lesbarhet når den er plassert på den hvite bakgrunnen.

Den siste endringene vi innførte før vi startet implementasjonen av selve programmet var å gjøre gulfargen mørkere. Da fikk den en god kontrast mot den hvite bakgrunnen, og dermed bedre lesbarhet.

Tegneprogrammet



Før vi startet på kodingen av tegneprogrammet, lagde vi et klassediagram for å få en bedre oversikt over hva som trengtes. Her er det satt opp en oversikt over hvordan klassene samhandler med hverandre. Programmet er delt inn i ulike scener som bestemmer hva som skjer på skjermen til en gitt tid. Alle scenene arver fra en baseklasse (`Scene`) som inneholder et felles system for blant annet navigasjonsknapper og tekst. Vi har for eksempel en meny-scene (`MainMenu`) som tegner en overskrift og alle knappene som skal vises på menyen. Vi har også en klasse (`SceneManager`) for å styre alle de ulike scenene. Denne bestemmer hvilken scene som skal tegnes, og sjekker om den nåværende scenen skal byttes ut.

Scenenes UI er bygd opp av to ulike elementer, knapper og tekst. Disse er delt inn i klasser for å gjøre designet så modulært som mulig. Knappene (alle klasser relatert til Button) er igjen delt inn i to forskjellige klasser som begge arver fra samme baseklasse. Dette ble innført fordi programmet består av både navigasjonsknapper og knapper for å endre farge på penselen.

Tegneprogrammet ble kodet i C++ med OpenGL som brukes til grafikkprogrammering. I tillegg brukte vi GLFW, som er et bibliotek for å håndtere vinduer og input fra brukeren, og GLM som brukes til matematikk. Vi brukte også FreeType, som er et bibliotek som renderer tekst på bitmaps. Både GLFW og GLM er ting vi var kjent med fra før av, men FreeType og tekstrendering generelt var nytt, og det trengtes derfor en del kompetanseinnhenting før vi kunne ta dette i bruk.

For implementasjon av penselen prøvde vi å finne et bibliotek vi kunne bruke. Dette viste seg å være en vanskelig oppgave. For det første var det ikke så mange tegnefunksjoner som lå ute på nettet. For det andre var det vi fant uferdig eller hadde bugs som påvirket penselen. Vi endte derfor opp med å implementere vår egen tegnefunksjonalitet, noe som fungerte bra. En negativ side med dette var at penselen ser "taggete" ut da den har en stor tykkelse, fordi vi ikke fikk tid til å implementere interpolering.

For å ta bilde av brukerens tegninger bruker vi stb_image_write. Dette er et enkeltfilbibliotek (header fil) som lagrer bilde av det som skjer på skjermen hvert andre sekund, slik at det kan sendes til analyse. Det blir så sendt et ord tilbake med hva modellen tror tegningen er. Dette fortsetter til tegningen gjettes eller nedtellingen går ut.

Testing og kvalitetssikring

Prosjektet vårt bruker en kodebase som ikke er egnet til testing-suites og enhetstesting, blant annet maskinlæring, OpenGL og IO-funksjoner, som gjorde at vi dessverre hadde liten mulighet til å utvikle slike tester. Til tross for det hadde vi muligheten til å bruke en rekke med verktøy og rutiner til å kvalitetssikre vår kode og det ferdige produktet.

Siden vi manglet mulighet til å gjøre enhetstesting utførte vi en veldig stor del av manuell testing, for å sikre at alt fungerte som det skulle. Vi gjorde også slik testing for hver modul når den var ferdig bygget, og integrasjonstesting etter V-modellen når vi var klare til å kombinere de ulike modulene vi hadde bygget. Vi støtte her på noe problemer med importering av biblioteker der mlpack og OpenCV bibliotekene ikke kom overens, som førte til at vi måtte ta en runde med omstrukturering av prosjektet. Heldigvis løste det seg da vi endret til å bruke Python for hele CNN-løsningen vår.

For å minimere antall tester vi trengte å gjennomgå, gjorde vi granulær testing av moduler og individuell kode hver for oss, på det vi selv hadde skrevet. Når vi hadde en del funksjonalitet implementert, gikk vi gjerne gjennom det i møter, og ba hverandre gå inn i den branchen på GitHub, hente ned den nye koden, og teste flere av oss om vi hadde problemer. I tillegg til

generell testing av om koden funget som den skulle, lot det oss også teste hvordan det funget på ulike operativsystemer. Vi måtte ofte endre noe på koden eller CMake-oppsett på grunn av ulikheter mellom Gnome-basert Linux og Windows 10.

Vi oppdaget en del ulike problemer gjennom dette arbeidet. For eksempel oppdaget vi at vår CNN hadde veldig god treffsikkerhet i å gjette korrekt klasse, men for mange begreper hadde den **også** høy gjettingsprosent for en eller flere andre klasser. Treffsikkerhet er hvor stor sjanse det er for at et bilde blir korrekt gjettet som den klassen den er. Presisjon komplementerer treffsikkerhet, og sier hvor bra modellen er til å gjette at bildet ikke passer inn i klasser det ikke hører til i (altså falske positive treff). Med andre ord oppdaget vi gjennom manuell testing at vår modell hadde bra treffsikkerhet, men dårlig presisjon. Vi kunne da videre jobbe på modellen med dette som basis.

Vi gjorde også en liten brukertest på brukergrensesnittet til tegneprogrammet ved hjelp av wireframes og prototype. I et større prosjekt ville vi brukt mer tid på dette, men vi fikk gjort en test som vi kunne bruke som basis til utviklingen av det ferdige brukergrensesnittet.

På GitHub satte vi opp workflows, som hjalp med både testing og kvalitetssikring. Workflows setter opp en virtuell maskin som kjører ett sett kommandoer for hver commit. Dette kan brukes til å passe på at koden som blir pushet opp faktisk fungerer, og tester kodekvaliteten ved å kjøre enhetstester og ved bruk av eksterne linting checkers. Linting checker hjelper oss med å skrive konsistent kode til tross for at vi var flere som skrev på samme fil. Dette er viktig for å gjøre koden vår mer lesbar og enklere å gjenoppta til senere tid. Før merging med main branch ventet vi alltid til alle workflowene var OK for å forsikre at koden på hoved bransjen alltid var kjørbar. Helt mot slutten av utviklingsprosessen sluttet workflowen - som sjekket om Drawing-program modulen fungerte - å fungere. Dette skjedde pga. ZeroMQ, og vi valgte å ikke bruke tid på å debugge dette rett før innleveringsfristen.

Utrulling

Arbeidet med utrulling var noe mer utfordrende enn forventet. Et stort problem her var ulike development-verktøy for utvikling på Linux og Windows, som var vanskelig å samkjøre. Særlig gjaldt dette bibliotek og andre verktøy som vi ville bruke, men var vanskelig å få implementert på Windows.

Vi endte til slutt opp med å distribuere programmet gjennom at bruker kompilerer til en enkel binærfil som kan kjøre hele greien. Dette må gjøres fordi kompilering med CMake lenker ulike biblioteker til programmet, men kopierer de ikke til build-folder - derfor vil den ikke finne bibliotekene på samme plass som de var om den ble bygget på en annen masking. Vi skriver mer om dette under videreutvikling.

Siden vi ikke bruker noen slags hosting-løsning for programmet, ei heller har vi implementert online leaderboard på openstack eller lignende, har vi ikke rullet ut programmet med en nettbasert løsning.

Emnebruk

Gjennom prosjektet tok vi i bruk elementer fra mange ulike emner i bachelor-programmet. Naturlig nok ble det spesielt mye fokus på AI og Grafikkprogrammerings-fagene. Men mange andre fag ble også brukt mye gjennom arbeidet vi gjorde i denne oppgaven. En liste over de ulike fagene vi føler ble særlig bra representert i prosjektet er vist i tabellen under.

Emne	Elementer vi skal bruke/sette oss dypere inn i
AI	Bildegjenkjenning – sentralt for produktet vi leverer. En stor del av det totale prosjektarbeidet involverte dette faget. Særlig var det bearbeiding av bilder og bruk av CNN til å klassifisere bilder som ble hentet fra dette faget.
Grafikkprogrammering	Vi brukte lære fra Grafikkprogrammering til å lage en tegnefunksjon og utforme et grafisk grensesnitt ved hjelp av OpenGL for vår frontend. Dette involverte både selve tegne-delen av programmet, samt også vising av UI-elementer som knapper, tekst og lignende. Vi lærte også grunnleggende bruk av CMake i dette faget.
Cloud Technologies	Tas i bruk gjennom innhenting av bilder via Google Images API, som får tak i datasettet vårt og passer på at det er tilgjengelig som forventet. Det er disse bildene som skal bli brukt for å trenne vår AI.
Programvareutvikling	Selve prosjektprosessen. Vi brukte flere konsepter som står sentralt i dette faget under arbeidet, som for eksempel Scrum-oppsett og sprinter, testing og lignende. Vi tar også med planlegging og rapportskriving.
Designtenkning	Brukergrensesnitt. Et fokus herfra er å huske empati for brukeren når vi jobber med utformingen av tegneprogrammet. Veldig i fokus under utviklingen av denne delen av prosjektet.
Spillprogrammering	Spilldesign og ideer rundt hvordan vi kan gjøre tegnespillet morsommere og mer brukervennlig. Herfra har vi for det meste hentet prinsipper og ideer om hva som gjør spill morsomme, hvordan best lære bruker i hvordan de skal bruke

	spillet, og hva vi bør passe på å teste i forhold til brukers kreativitet - de vil kanskje ikke bruke spillet på den tiltenkte måten.
Programmerings-fokuserte fag (Objekt-orientert, Avansert)	Backend. Det meste vil bli kodet i C++ og Golang, som er programmeringsspråk vi har lært i tidligere programmeringsemner. Konsepter vi lærte i disse fagene står sentralt i arbeidet vi gjorde, særlig i forhold til kvalitetssikring og generelt oppsett av repo, arbeidsmetoder, osv. Ideen om bruk av ZeroMQ.

Vi føler at vi lærte mye gjennom dette prosjektet. Dette inkluderer samarbeid i grupper, oppsett av prosjekt på GitHub, systemarkitektur og planlegging, forberedende arbeid som kravspesifisering, og selvsagt kodingen vi gjorde. Noen mer spesifikke eksempler inkluderer:

- Vi lærte mye mer om bruk av CMake og generelt oppsett og kjøring av C++-programmer.
- Som nevnt, mye rundt forberedende prosess i kodings-prosjekt, særlig rundt krav, arkitektur og avgrensning.
- Vi lærte mye om arbeid med git, blant annet [trunk based development](#), oppsett av issue board med automatisering, linking til issues under commits og oppsett av workflows på GitHub for testing/linting.
- Vi fikk mer trening i og forbedring av arbeid i grupper med scrum, og samarbeid med kodingsprosjekter generelt.
- Vi lærte hvordan vi kan sette sammen og distribuere en kodebase basert på separate moduler skrevet i flere ulike språk.

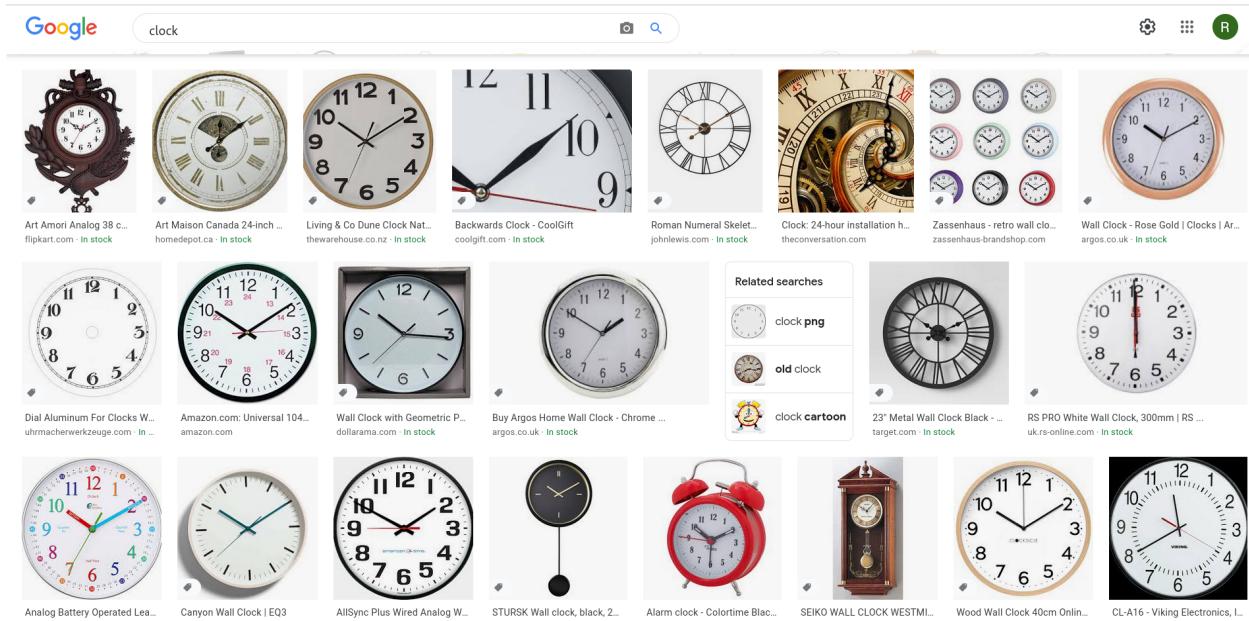
En ting vi føler vi kunne gjort en enda bedre jobb med var å hjelpe hverandre sette oss inn i og få oversikt over koden de andre i gruppen skrev. Med unntak av deler av maskinlærings-modellen, var det lite pair-programming og kodegjennomgang av hverandres kode gjennom prosjektet. Dette gjorde at for selve detalj-kodingen vi gjorde, fikk hver av oss for det meste mer læring i den delen av prosjektet som vi jobbet i, men mindre i andre deler av kodebasen. For eksempel, mer trening med maskinlæring, men ikke innenfor grafikkprogrammering og arbeid med OpenGL. Heldigvis fikk vi likevel alle nyttig læring ut av den kodingen vi gjorde, og fullt utbytte av læring fra de andre delene av prosjektet som nevnt over.

I tillegg lærte vi også mye om multiplatform-utvikling, og særlig da Linux. Siden noen medlemmer jobbet i Fedora og andre i Windows, måtte begge gruppene være påpasselige rundt dette, og fikk fokus på dette området som vi ikke har hatt i tidligere prosjekter. Som nevnt tidligere så endte dette med å skape problemer for oss.

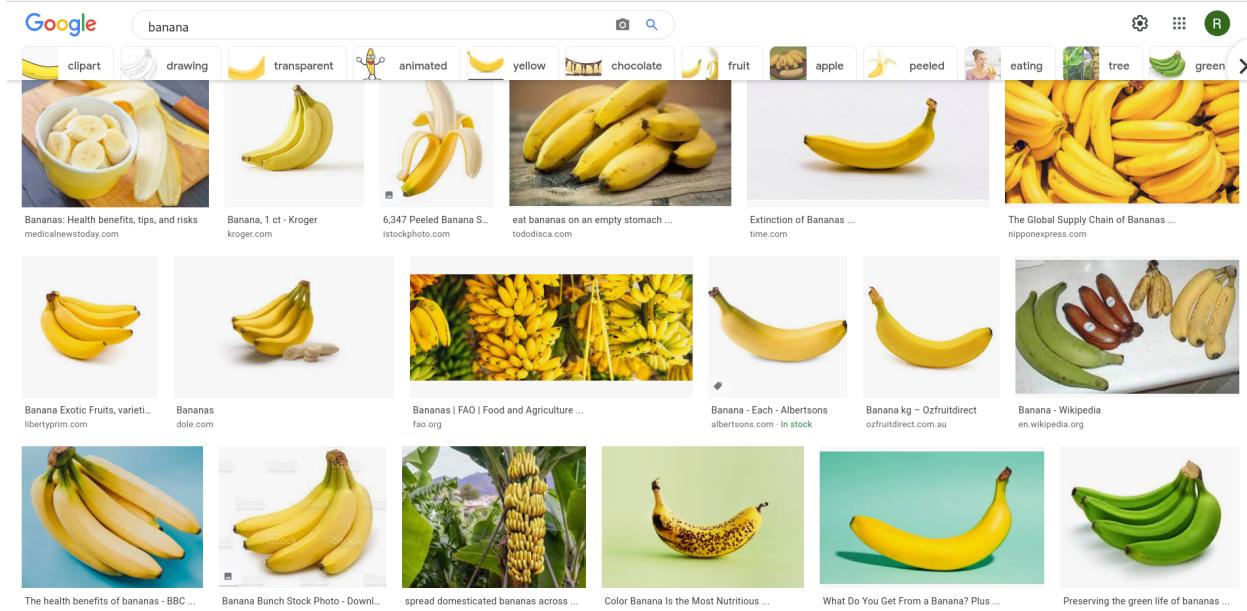
Resultater

Vi var ved starten av arbeidet veldig interessert i to aspekter ved vårt endelige produkt. Først om vi ville klare å designe og kode et tegneprogram som funker, med solid brukergrensesnitt - et presentabelt produkt. Deretter om ML-modellen vår ville klare å identifisere og dele mellom de ulike klassene på en grei måte. Selv om vi ikke fikk perfekte resultater, er vi godt fornøyde med hvordan dette gikk.

En ting som var interessant - dog ikke uventet - var forskjellen i resultatet av modellen vår basert på de frasene som ble sendt inn. Dersom vi bare sendte inn enkle ord uten utdypelse, som for eksempel 'banana' eller 'apple', ble resultatene veldig dårlige. En rask titt på datasettet forklarer det fint - et søk på 'clock' kan for eksempel gi dette resultatet:

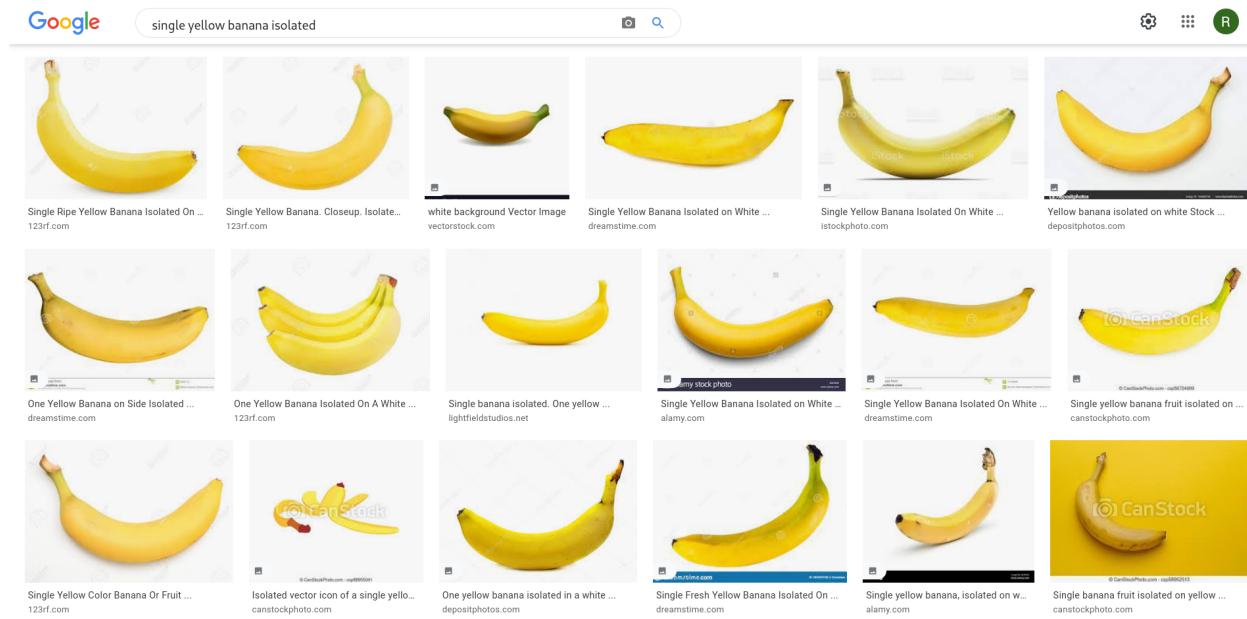


De fleste er bilder av en kjøkkenklokke slik vi tenkte det. Men det er også en del som ikke passer med de andre - eldre vegg-klokker, et surrealistisk maleri, en klynge med klokker, osv. Dette eksempelet er ikke det verste - la oss sammenligne med et søk på 'banana':



Som vi ser er resultatet fylt med hele banan-klaser, som vår dumme ML-modell ikke enkelt kan forbinde med enkle bananer. I tillegg er det grønne og noen få røde bananer i resultatene. Dette gjør at modellen har store problemer med å skille ut definerende trekk i hver klasse, som gjør det mye vanskeligere å deretter skille mellom ulike klasser under treningen.

Dersom vi derimot bruker noen adjektiver for å presisere hvilke typer bilde vi vil ha, får vi mye bedre resultater. Et adjektiv som konsistent hjalp var 'isolated' for å få bare en banan eller et eple istedenfor grupper - vi forsøkte også 'single', 'one', osv for å finne det beste adjektivet. Vi hadde også god suksess med å legge inn farge som adjektiv. Presisering av 'red apple' eller 'yellow banana' var god hjelp med å luke bort bilder som "ødela" datasettet, som for eksempel umodne bananer eller grønne og gule epler. På denne måten ble farge en mye mer pålitelig faktor for modellen i arbeidet med å skille klasser fra hverandre. Se for eksempel det mer konsistente resultatet fra et google image søk for 'single yellow banana isolated' på neste side.



Uten adjektiv, hadde modellen vår en ganske dårlig treffsikkerhet mellom 50 og 60%. Den var veldig upålittig, og mange klasser ble konsistent blandet sammen - for eksempel epler og jordbær, eller klokker og spøkelser. Dette er på grunn av form og farge for det første eksempelet, og fargeprofil i det andre - både klokkene og spøkelsene er for det meste hvite bakgrunner med svarte linjer over, og formene er ikke ulike nokk til at vår modell klarte å separere disse.

Med adjektiv lagt til, skjøt treffsikkerheten til modellen i været - uten noen andre endringer gikk den over 70%, og med en liten økning i antall omganger modellen kjørte, samt tregere learning-rate som gjør at modellen går grundigere gjennom datasettet, fikk vi resultater opp mot 90% treffsikkerhet. Dette er mye bedre resultat enn vi forventet fra datasett laget fra google-bilder. Siden datasettene ikke er kurerte er det flere bilder under hver klasse som ikke ligner på resten i det hele tatt - for eksempel bananklaser i et datasett med neste bare enkle bananer. I tillegg gjør vi ingen masking, og bakgrunner blir derfor også trent på, som vil svekke modellens kapasitet til å skikkelig definere hver klasse.

Dette er en type preprocessing som må brukes med omhu. Det er veldig lett å kombinere ord som gir et helt annet resultat enn du forestilte deg. For eksempel sender vi gjerne med 'drawing' for å få tegnebilder av det begrepet vi er interessert i - men om vi vil ha bilder av penner eller blyanter, kan vi ikke sende in 'pencil drawing' eller lignende!

Dette løste mye med modellen, men vi har likevel et problem her - når vi snakker om treffsikkerhet relatert til maskinlæring (accuracy), er det hvor god maskinen er til å korrekt klassifisere noe - altså hvor sikker den er på at elementet ligger i den klassen. Men det sier ingenting om modellen klarer å si at de elementer som IKKE er i den klassen faktisk ikke er det.

Med andre ord garanterer det ikke for at modellen, i tillegg til å være skråsikker på at et vannmelon er en vannmelon, ikke også kan være sikker på at vannmelonen er en jordbær. Dette gir oss derfor et lite problem - selv om modellen vår er 90% sikker på at alle vannmelon-tegninger den får sendt inn er et vannmelon, hjelper det lite om den er 92% sikker på at mange er jordbær og derfor bare gjetter jordbær på disse:

[strawberry.jpg](#): [0.0173, 0.4532, 0.1886, 0.9999, 0.1352, 0.9941, 0.7099, 0.0075, 0.7701, 0.2161, 0.0063]

Som vi ser på eksempelet over her, finner vår modell at to klasser - indeks 3 og 5 (går utifra at array starter på 0) - blir evaluert som veldig nær bildet av jordbær. I dette eksempelet er de klassene 'strawberry' og 'watermelon'. Begge to er gjettet med over 99% sikkerhet, som selvsagt er et problem for vannmelon-klassen, det vil si at modellen kan veldig enkelt ta feil av de to.

Vi tenkte ut to mulige løsninger til dette: først, forbedre treffsikkerheten til modellen gjennom forhåndsbehandling, større og mer komplisert model, osv. Dette vil være mye arbeid, gjøre treningen til modellen mye tregere, og risikerer å bare forbedre modellen litt - særlig ved større antall gjenstander/konsepter å gjette på, vil dette gjerne forverres igjen.

Den andre løsningsideen vi hadde baserte seg på spillets funksjonalitet. Modellen vår skal gjette på brukerens tegninger ganske ofte, og oppdatere med ny gjett på skjermen. Når den gjetter riktig, slutter spillet. Det er med andre ord viktigst at modellen TIL SLUTT gjetter det riktige ordet, og ikke at det gjetter det med en gang.

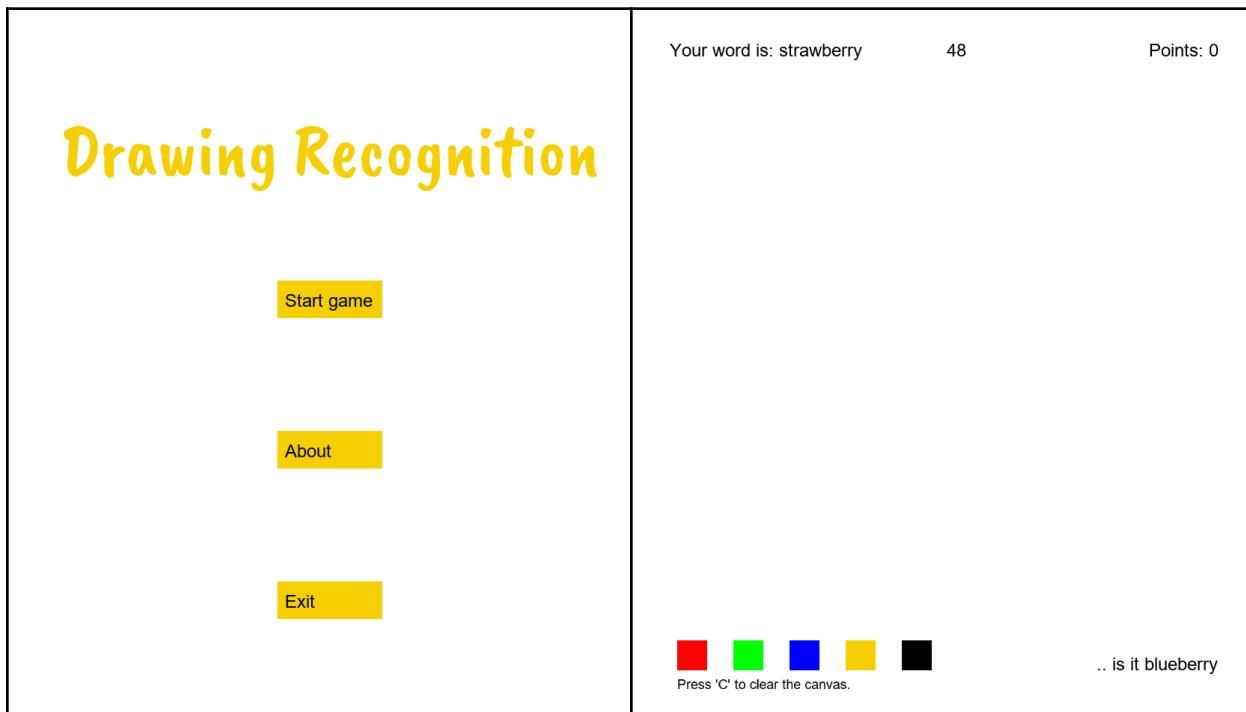
Vi får modellens klassifikasjoner som tall på treffsikkerhet, der det største tallet er det maskinen er mest trygg på. Men vi kan som nevnt ha flere resultater med ganske høy treffsikkerhet. Vår løsning var at, dersom vi får samme klassen fra vår modell flere ganger på rad, sender vi isteden klassen med nest-høyest treffsikkerhet. På denne måten er vi sikker på at modellen gjetter flere ulike ord, og ikke blir kjørt inn i feilspor med gjetting. Vi kunne også tatt dette videre og fortsatt å gå nedover treffsikkerhets-listen, men bestemte at det ville gjøre det for enkelt - modellen ville alltid gjette rett, og også ganske fort.

I tillegg prioriterte vi å legge inn ord med ganske unike og lett gjenkjennelig former og farger. For eksempel fjernet vi gjennom arbeidet 'clock' og 'ghost' fordi disse gjerne var bare svart/hvit, og hadde ikke konsistente former. Vi oppdaget også her at det ofte var vanskelig å finne gode eksempler på google images, da mange bilder gjerne hadde flere elementer, eller av en eller annen grunn var av helt andre ting enn det vi søkte på. Dette er helt klart en svakhet med vår datasett-løsning, men også en vi forutså helt fra starten.

Vi fikk ganske mye bedre resultat enn vi forventet ved start av prosjektet - vi hadde forventet resultater ned mot 50% treffsikkerhet, men vi oppnådde resultater på opp mot 80%+ med testdata, og det er fremdeles mer arbeid med forhåndsbehandling som kan gjøres. Det spørs likevel om ikke kurerte datasett vil gi enda bedre treffsikkerhet totalt sett. I tillegg er denne

løsningen mer sårbar ved tillegg av nye datapunkter - velger vi et nytt ord å legge til med dårlig datasett, kan vi ødelegge hele modellen. Det kan derfor diskuteres hvilken fremgangsmåte er best - dersom en treffsikkerhet rundt 80 - 90% er akseptabelt, vil vår løsning med lett og enkel nedhenting av data være et godt valg. Større treffsikkerhet kan oppnås med kurert datasett.

Om vi hadde brukt kurerte datasett tror vi at programme ville funket mye mer konsistent, og treffsikkerheten ville krype opp godt over 90%. Men vi angrer ikke for at vi gikk for automatisk innhentet datasett - ikke bare fikk vi testet hvordan automatisk innhenting ville påvirke modellen, samt implementere API-løsningen i prosjektet, men vi sparte også mye tid ved å ikke trenge å finne eksisterende datasett eller lage våre egne manuelt.



Bilde over representerer brukergrensesnittet i det ferdige produktet. Måten vi har satt opp programmet gjør at det vil starte å gjette med en gang. Litt av problemet vi oppdaget da var at programmet automatisk vil gjette den klassen som ligner mest på en hvit bakgrunn, før spiller får starte å tegne. Vi vurderte dette problemet i gruppe, og bestemte at beste løsning vil være å ikke bruke denne klassen som noe spiller skal gjette på - ellers ville den automatisk gjette korrekt uten tegning.

I vår prosjektplan hadde vi implementasjon av online leaderboard som funksjonalitet vi ønsket å legge til etter at selve MVP for tegneprogrammet var ferdig. Dessverre måtte vi til slutt bestemme oss for å ikke implementere dette i vårt ferdige produkt. Det var i hovedsak to grunner til dette:

- Resten av prosjektet tok mer tid enn vi hadde sett for oss, og derfor manglet vi tiden vi trengte til å implementere dette.

- Vi tenkte ikke på komplikasjonen som følger med internett-tilkobling for leaderboardet i forhold til det å lage en kjørbar binærfil for tegnespillet vårt, som vi oppdaget ville gjøre det mye vanskeligere å sette opp binærfilen. Det vil innebære tilgang til internet som åpner opp produktet til flere angrepssvinkler; Man In The Middle (MITM), Distributed Denial of Service(DDoS), osv...

Siden vi oppdaget at vi ville ha store problemer med å implementere leaderboard med bakgrunn i disse to utviklingene gitt den tiden vi hadde til arbeid, bestemte vi oss for å ikke implementere leaderboard. Derfor ble det også en del endringer fra prototype til ferdig produkt. Siden vi fjernet leaderboard, måtte også denne delen av tegneprogrammet fjernes, og brukergrensesnittet ble noe endret for å tilpasse dette. Det var heller ikke lett å innføre det planlagte brukergrensesnittet med OpenGL, og deler av programmet (knapper og tekst) ble derfor seende annerledes ut enn først planlagt. Ellers følger vårt produkt veldig nært prototypen vi laget under designdelen av arbeidet. Særlig er spillet veldig tett opp mot vår originale idé.

Refleksjoner rundt arbeidet

Alt i alt er vi veldig fornøyd med hvordan prosjektet gikk, ikke bare sett med resultatet i betrakning, men også prosessen totalt sett. Gruppen jobbet bra sammen, kommuniserte fint, og gjorde en konsistent innsats gjennom prosjektpérioden.

Oppgavefordelingen var god, men kunne nok forbedres noe. Noen oppgaver ble veldig brede og store, andre små og enkle. Dette gjorde at det var vanskelig å bruke vårt issue-board til å bestemme fremgangen på prosjektet, noe som ble enda tydligere da vi måtte utvikle flere iterasjoner av vår CNN-løsning, og bruke en del tid på å løse import-problemer med den. I tillegg førte moduloppdelingen til at hvert gruppemedlem gjerne jobbet med sin egen del av kodebasen, og det ble vanskeligere å senere flytte til en annen del som de ikke hadde jobbet med. Dette løste vi typisk gjennom pair-programming.

I arbeidet gjennom prosjektet brukte vi scrum-utvikling. Vi valgte en ganske fri organisering av gruppen ved å blant annet ikke ha en scrum master. Dette funket greit, men vi så også fordelen av å ha noen til å organisere møter og bestemme hvordan agendaen for scrum-møtene skulle settes opp og ta endelige bestemmelser.

Vi delte prosjektet inn i egne biter og utviklet de separat. Denne oppdelingen hadde i hovedsak to fordeler. For det første unngikk vi å trække på hverandres tær, og minimerte risikoen for at flere av oss jobbet mot samme mål og kastet bort arbeidstid på denne måten - bruk av issue board hjalp også her. I tillegg gjorde det at vi kunne utvikle modularisert kode, som var til stor hjelp i forhold til testing og kvalitetssikring av kode - vi kunne ferdigutvikle interne deler av hver modul før vi koblet de ulike modulene i prosjektet sammen.

Dette hadde både fordeler og ulemper. Alle gruppemedlemmene fikk jobbe med kode, og vi gjorde en god jobb med å jobbe effektivt og ikke bli sittende å vente på at et annet

gruppemedlem gjorde ferdig en kodebit slik at vi kunne fortsette. Men det gjorde også at hver person gjerne stod for en 'del' av koden, og dersom f.eks. et gruppedel ble ferdig med sin modul ble det veldig vanskelig å sette seg inn i og hjelpe med utvikling av andre moduler som det medlemmet ikke hadde gjort noe arbeid med. Heldigvis var dette ikke et stort problem da vi gjorde jobben med å sette oss inn i kodebasen når det ble nødvendig, og hadde også mye arbeid å gjøre innenfor rapport-skriving og annet arbeid utenfor koding.

Måten vi styrte fremgangen i prosjektet på var gjennom issue board på GitHub. Vi delte hver modul av prosjektet til egen milestone, og la til issues for hver milestone. Vi assignet oss til de vi jobbet med, og oppdaterte status til de ulike issuene. I scrum-møtene åpnet vi de relevante milestones som vi jobbet med, og diskuterte fremgang/ressursbruk. Eksempel på et slikt issue board midt i prosessen:

Column	Issue Title	Assignee	Labels	Status
To do	Midtstille tekst	#132 opened by maren-sg	wontfix	
To do	Lag felles vao, vbo, ebo for buttons	#115 opened by maren-sg		
To do	Interpolation	#119 opened by rickarl	Cpp enhancement wontfix	
In progress	Integrator ML modellen	#117 opened by sindre0830		
In progress	Sette opp prosjektet med CMake	#68 opened by maren-sg		
In progress	State managing	#107 opened by maren-sg		
In progress	Fikse fullscreen	#103 opened by maren-sg		
Done	Lage brukergrensesnitt i OpenGL	#33 opened by maren-sg		
Done	Design OpenGL	#28 opened by rickarl	Cpp	
Done	Design kodelbase	#26 opened by rickarl	Cpp enhancement	
Done	Kompetanseinnhenting	#27 opened by rickarl	Cpp enhancement	
Done	Legg til drawing-program workflow	#120 opened by sindre0830		

Nye issues ble lagt til i 'To do' kolonnen, og vi la til de fristene og andre detaljer vi kom på. Når vi startet arbeid på en issue, ble det flyttet til 'In progress' og viste hvem som jobbet med det. I starten av utviklingsprosessen utnyttet vi en Review-kolonne for å sammen gå gjennom det som var gjort i samme modul før vi satte det til Done. Men etter hvert som vi jobbet i separate moduler, på egne branches, så vi ikke lenger poenget med det og fjernet det. Eksempel med review er lagt til i Scrum Board eksempler. Når det var gjort, gjorde vi merge på branchen vi jobbet i med sammen til main, og lukket automatisk issuet gjennom det - som også flyttet det til 'Done'.

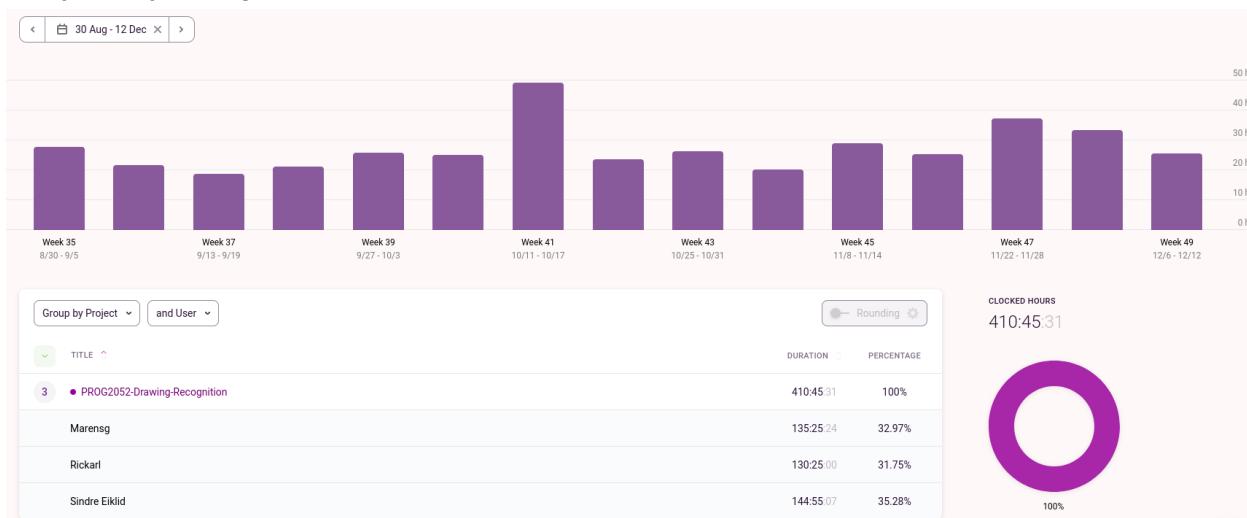
Dette funket greit, men ikke perfekt. Det store problemet var i å bli vant med det - noen gruppemedlemmer var flinke til å lage nye issuer på funksjonalitet, bug fixing og endringer som skulle gjøres, andre gjorde ikke det og fortsatte å jobbe på en stor bit kode som ett enkelt issue. Dette skapte ikke problemer i vårt prosjekt, men gjør issue-boardet vårt litt rotete og til en litt dårligere logg enn det kunne være, da størrelsen på hvert issue i forhold til arbeidsmengde ikke er like konsistent som vi skulle ha ønsket.

Vi delte prosjektet inn i tre deler som vi jobbet med hver for seg: innhenting av datasett, CNN og tegneprogram. Vi jobbet ikke sekvensielt med disse, men vi startet de heller ikke samtidig - først så vi på datasett, så startet vi med CNN, og vi begynte arbeidet med tegneprogrammet sist. Dette var ganske enkelt bare fordi vi trengte mer enn en person til å jobbe med hver modul, og vi ønsket ikke å hoppe frem og tilbake så mye. Derfor hadde vi først overlapp mellom datainnhenting og CNN, og når datainnhentingen var nesten ferdig, startet vi på tegneprogram og hadde overlapp mellom det og CNN frem til slutten av prosjektet.

Timelog

Vi brukte [Toggl](#) til å føre timebruk i prosjektet. Dette lot oss både logge bruk av tid til ulike arbeidsoppgaver, og se hva de andre i gruppen jobbet med og hvor stor arbeidsmengde de hadde hver uke. Etter å ha brukt tjenesten gjennom prosjektet ser vi at vi kunne gjort en enda bedre jobb med dette verktøyet, ved å sette opp mer robuste tags som speilet direkte de ulike milestones som vi definerte i issue tracker. Dette er noe vi kommer til å ta lærdom fra til fremtidig arbeid.

En fordel med Toggl er at det også lett lar oss visualisere og hente ut timebruk i prosjektet uke for uke gjennom semesteret. Dette gjorde at vi lett kunne se fremdriften i arbeidet og passe på at vi jobbet jevnt og trutt:



Rapporten fra Toggl viser at vi hadde noe sporadisk arbeidsoppsett, der flere uker hadde færre arbeidstimer enn ideelt, mens særlig uke 41 var en tung uke med mange timers arbeid. Men vi jobbet også hver uke, og har ikke ekstrem variasjon. Det skal også nevnes at vi ikke alltid husket å legge til arbeid i Toggl, og rapporten er derfor ikke helt perfekt - den dekker kanskje rundt 90% av arbeidstiden. Vi hadde som mål å bruke ca 8.5 timer per person per uke, og endte opp med rundt 9.1. I tillegg ser vi at alle i gruppen jobbet bra og endte opp med cirka samme timebruk i faget!

Vi ville logge eksakt timebruk for ulike deler av arbeidet, men vi støtte på et problem med Toggl fordi alt vi hadde lagt til om oppdeling i ulike milepåler ble automatisk fjernet da det gratis plusabonnementet til Toggl gikk ut. På grunn av problemet, har vi kun mulighet til å se timebruk for hver av våre fire 'tags' - Koding, Skriving, Design og Kompetanseinnhenting. Disse har også noe overlap da vi gjerne brukte både skriving og design som tag under prosjektplanarbeid, eller koding og kompetanseinnhenting ved starten av kodingsarbeid, så sum timebruk totalt for alle fire er over 100%/410 timer. Dette gjør dessverre at detaljert timebruk ikke er tilgjengelig, men vi har lært mer om hvordan vi bruker verktøyet Toggl, og hva som er viktig å passe på rundt timelogging generelt.

Toggl rapporterte at vi brukte ca 15 timer på design, men det faktiske tallet er nok en del timer større da vi gjorde designarbeid under prosjektplanlegging som ikke ble skikkelig tagget - kanskje ~25 timer totalt. Kompetanseinnhenting er logget til 101 timer, og er som forventet særlig fokusert rundt starten av prosjektet. Koding ligger på 210 timer, rundt halvparten av den totale tidsbruken så langt, som vi føler speiler erfaringen vi har hatt rundt prosjektet godt. Skriving endte på 158 timer og er tungt fokusert på starten og slutten av prosjektet, ved skriving av prosjektplan og rapport i plenum. Den fikk også et stort hopp i uke 43 da rapportskriving startet. For flere detaljer rundt vår tidsbruk under prosjektet, sjekk [Timelogger](#) under vedlegg.

I forhold til risikovurderingen vi gjorde på starten av prosjektet, endte den opp med å treffe ganske ok. Vurderingen traff ganske bra i forhold til ZeroMQ, som ble vurdert som relativt høy risiko. Vi endte opp med å ha noe problemer med implementasjonen av ZeroMQ, dog ikke akkurat av samme årsak som vi så for oss på den tiden. Istedenfor var det den største risikoen vi ikke hadde beregnet som også stakk kjepper i hjulene for ZeroMQ: utviklingsplattformer.

Forskjell mellom Windows og Linux, særlig for linking av bibliotek, gav oss store problemer gjennom de siste fasene av prosjektet, og var ikke noe vi hadde forutsett under planlegging. I tillegg til dette brukte vi også mer tid på å implementere CNN enn vi hadde forutsett, mye på grunn av at vi helst ville forsøke å få det til i C++ uten bruk av Python. Alt i alt kunne vår risikovurdering klart ha truffet bedre, men vi er fornøyd med at vi traff ganske bra med de fleste vurderingene sett med etterpåklokskapens øyner.

Gjennom prosjektet lærte vi en del om å sette og definere oppnåelig omfang for prosjekt gjennom våre erfaringer med leaderboard. Vi lærte også å skikkelig definere minimum viable product, og at det å fokusere på MVP først og ikke bli distraheret av andre ideer er smart.

Videreutvikling

Vi har flere ideer om hvor prosjektet kunne utvikles videre utenfor omfanget av prosjektet. Det aller første vi ville jobbet med er forbedringer i koden vi har skrevet. For eksempel ville vi gjerne endre litt på tegnespillets UI, slik at når bruker gjetter et ord blir det fremhevet. Vi ville også gjøre tegne-delen enda bedre med mer flytende strøk - vi endte opp med å øke pensel-størrelsen mot slutten av prosjektet, som hjelper med tegningen men gjør at det ser litt mindre fint ut. Dette er forbedringer vi måtte hoppe over under utvikling på grunn av tidspress.

Den første og kanskje klareste videreutviklingen i forhold til ny funksjonalitet vil kanskje være å legge til leaderboard-funksjonaliteten vi vurderte under planleggingen av prosjekt. Dette vil legge inn et element av spiller-mot-spiller funksjonalitet, og gi enhver bruker mulighet til å sammenligne seg selv med andre brukere. Det vil også lage spillet litt mer sosialt.

Med tanke på forbedringer av eksisterende funksjonalitet, kunne vi tenkt oss å for eksempel legge inn mulighet for å fortsette å trenere ML-modellen vår på bruker-lagde bilder. Måten vi tenker dette kunne gjøres på vil være å lage docker-bilde av selve maskinlærings-programmet vårt, og legge det opp som server på en cloud-løsning (for eksempel openstack). Bildene brukerne lagrer sendes gjennom API til denne serveren, som lagrer de i nye datasett i en database - enten Firebase eller lignende, eller eventuelt bare lokalt.

Når vi har mottatt et visst antall nye elementer, laster vi inn den eksisterende modellen vi bruker, og trener den videre på det nye datasettet flere ganger. Vi velger så den modellen med best resultater som vår nye modell, og lagrer den gamle som backup. Denne videre-treningen kan gjøres på to måter: enten lagrer vi alle datasett vi har brukt og bare legger til nye datapunkter, og trening av ny modell gjøres fra starten av. Dette vil være veldig grundig, men tvinger oss til å arkivere bruker-tegninger langt fremme i tid, og enten gjøre det samme eller laste ned flere ganger for google-images datasettene. Vi ønsker ikke å gjøre slik vedvarende lagring av bildedata, så vi tenker heller å laste inn eksisterende modell og trenere den videre med bare ny data som lagt frem av Jason Brownlee i hans artikkel for Machine Learning Mastery (1).

Utfordringen her blir da å legge den nye modellen til vårt tegneprogram og oppdatere med ny executable. En måte å gjøre dette på er å få server til å bygge programmet på nytt etter ny modell er valgt, og lagre executable etter det - deretter laste det nykompliserte programmet opp på en eller annen service der brukerne kan laste det ned, og gi lenke til denne.

Vi har også tanker rundt videre funksjonalitet som kan legges inn i tegneprogrammet. For eksempel tenker vi at det ville være morsomt å legge til flere vanskelighetsgrader bruker kan velge imellom, som for eksempel kan bestemmes av hvor vanskelig det er å tegne noe som vår maskin kan gjette. For å få til dette ville vi måtte gå gjennom vår database med begreper, og kvalifisere de basert på to kriterier: hvor vanskelig er det for en person å tegne dette med PC-mus, og hvor bra treffsikkerhet har vår modell på dette begrepet. Det første vil bestemme hvor enkelt det er for brukeren å kommunisere de trekkene ved begrepet som de vil fremheve til

vårt program, og det andre bestemmer hvor god sjanse modellen vår har til å gjette riktig, gitt at den får noenlunde korrekte trekk sendt inn av bruker.

Ettersom vi hadde en del problemer med å kjøre programmet på windows, kan det være relevant å se på Docker. Det er en container løsning som virtualiserer produktet vårt ved å kjøre det på OS nivå som gjør at det er kjørbart på alle plattformer (5).

Konklusjon

Vi endte opp med et sluttprodukt som dekket alt det vi hadde mål om i vårt utkast til MVP, selv om vi kanskje ikke fikk finpusset det helt opp som vi ønsket. Gjennom arbeidet med oppgaven forstod vi at vi ikke hadde estimert arbeidet som måtte gjøres helt korrekt, og endte opp med en stor utfordring. Særlig støtte vi på problemer rundt implementasjonen av vår CNN i C++, og med konflikter mellom utviklings- og utrullings-plattformer.

Til tross for disse utfordringene, er vi veldig fornøyde med egen innsats, og stolte av den resulterende programvaren. Tegnespillet funker bra, selv om det har noen finurligheter og småting vi gjerne skulle ha fikset opp. Både innhenting av datasett og trening/bruk av CNN-modell er også funksjonelt. Vi har hatt mye læring gjennom faget, ikke bare i form av kode-kunnskaper og det å sette sammen et stort prosjekt, men også det å arbeide i gruppe, effektiv bruk av tid, organisering og strukturering av kodebase, arbeidsprosesser og mye mer.

Literaturliste

1. Brownlee, Jason. How to Update Neural Network Models With More Data [Internett]. San Francisco: Machine Learning Mastery; 2021 [hentet 25. november 2021]. Tilgjengelig fra: <https://machinelearningmastery.com/update-neural-network-models-with-more-data/>
2. Hammant, Paul et al. Trunk-based Development: Introduction [Internett]. United Kingdom: Trunk-based Development; 2021 [hentet 30. november 2021] Tilgjengelig fra: <https://trunkbaseddevelopment.com/>
3. Norman, D. The Design of Everyday Things. Revised & Expanded utg. New York: Perseus Book Group; 2013. s. 128-32.
4. Brownlee J. How to Choose an Active Function for Deep Learning [Internett]. San Francisco: Machine Learning Mastery; 2021 [hentet 7. desember 2021]. Tilgjengelig fra: <https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/>
5. Docker. What is a Container? [Internett]. Palo Alto: Docker; ukjent [hentet 9. desember 2021] Tilgjengelig fra: <https://www.docker.com/resources/what-container>
6. Lim, S. GPU vs CPU Deep Learning: Training Performance of Convolutional Networks [Internett]. Phoenix: phoenixNAP; 2018 [hentet 28. november 2021] Tilgjengelig fra: <https://phoenixnap.com/blog/gpu-deep-learning>

Vedlegg

Prosjektplan

Mål

Bakgrunn

Da vi startet planlegging til prosjektet, tenkte vi med en gang at et produkt som brukte AI var ideelt å satse på. Det var ganske enkelt fordi vi visste at bare et fåtall av studentene i faget hadde bestått AI-valgfaget fra vårsemesteret, og to av de er i vår gruppe. Dermed ble AI en særegen kompetanse for oss, som vi vil kapitalisere på. I tillegg synes vi AI er et spennende felt, og var ivrige til å ta det med videre gjennom utdanningen.

Med dette som utgangspunkt, tok vi noen sesjoner med brainstorming, der vi skrev ned flere ulike ideer om prosjektet basert delvis eller helt på AI. Vi gjorde brainstorming rundt blant annet Sjakk AI, Tegnegjenkenningspill og en applikasjon som kunne gjenkjenne tekst fra bilde og konvertere dette til et string format (mulighet for real-time translation, eller copypaste av tekst).

Vi ble enige om at Sjakk AI var for ambisiøst og at tekstgjenkjenning ville kreve mye ML trening. Da stod vi igjen med tegne gjenkennings spillet. Denne ideen ble inspirert av Google Quick Draw, men vår versjon er ikke identisk, med blant annet eget system for datasett-innhenting til tidlig opplæring og et rudimentært konkurranse-element i form av leaderboard utenfor spillet.

Prosjektmål

Effektmål

- Brukeren lærer å tegne under kort tid
- Brukeren blir underholdt
- Bruker kan sammenligne score med andre brukere og se hvor en rangerer relativt til andre

Resultatmål

- Utvikle et spill med ferdig tegne- og gjenkenningsfunksjon.
- Få Convolutional Neural Network modellen til å gjenkjenne bilder innenfor spesifikke kategorier (Dyr, frukt, etc.).
- Klare å automatisere danning av datasett for maskinlæring gjennom Google Images API.
- Klare å bruke tegninger laget i spillet til å videre trenere opp vår AI.

Læringsmål

- Lære Scrum i en større sammenheng. Bruke dette til å forbedre gruppearbeid.
- Effektiv datainnhenting for modelllæring.
- Convolutional Neural Network modell kodet i C++ (Har kun brukt Python tidligere).
- Selv-læring av modell (Sende opp det brukeren tegner til en server, og trenere modellen på de bildene. Vil gjøre modellen bedre jo mer den blir brukt).
- Utvide kunnskapene våre i grafikkprogrammering ved å lage en tegnefunksjon og et grafisk grensesnitt.

- Kombinere deler av kunnskapen vi har innhentet gjennom studieløpet.

Omfang

Oppgavebeskrivelse

Planen vår er å lage et tegnegjenkjenningsspill. Det går ut på at AI skal gjette hva spilleren tegner. Brukeren får en gitt tid på å tegne så mange som mulig tilfeldige ord som mulig. Det blir gitt poeng for hvert ord AI klarer å gjette riktig. Den totale poengsummen oppnådd vil deretter bli lagret i et leaderboard, som holder oversikt over hvilke brukere som har den beste poengsummen.

Avgrensning

Vi skal starte smått med trening av AI-en. I første fase vil vi kun benytte bilder innen en bestemt kategori, for eksempel frukt, som kan utvides etter hvert som vi har tid. Vi vil også være låst bak rate-limit til APIen så vi må muligens gå for ferdiglagde datasett.

Selv om vi har planlagt leaderboard, så er ikke dette nødvendig for å nå MVP og kan bli kuttet ut hvis vi ikke har tid til å implementere.

Det er fortsatt noe usikkert om vi skal lage en tegnefunksjon med farger eller ikke. Inkludering av farger vil gjøre det lettere for AI å skille mellom for eksempel en melon og en appelsin. Samtidig er det mulig dette blir vanskeligere å designe enn svart-hvitt tegninger i spill-applikasjonen vår. Svart-hvitt er også mye lettere å trenne AI på, da mindre informasjon fører til mye raskere opptrenings. Vi skal se hvordan dette vil påvirke arbeidsmengden. Heldigvis er det ikke mye ekstraarbeid å kode en AI som kan jobbe med både svart-hvitt og farger.

Vi trenger et fungerende spill for å ha et fullverdig prosjekt, men vi kan teste og jobbe med resten av prosjektet gjennom å manuelt sende bilder til AI. Dette gjør at spillet trygt kan avgrenses fra resten av prosjektet.

Fagområde

Kjernen av prosjektet vi arbeider med er kombinasjonen av tre ulike mål, oppgaver som vi vil forsøke å løse. Den første oppgaven som vi var interessert i å jobbe med er automatisk innhenting av datasett for trening av kunstig intelligens på bilder. Gjennom bruk av Google Images API og vår egen API som tar denne i bruk, har vi som mål om å lage et bibliotek som lar bruker taste inn et emne og antall ønskede bilder, og deretter henter inn lenker til thumbnails som passer dette emnet. Jo enklere emnet er, jo mer treffsikker vil vår API være - for eksempel vil 'eple' trolig gi veldig bra treff, der 'her majesty's royal guard' trolig ikke vil være like pålitelig.

Den andre oppgaven er selve trening av AI på disse datasettene. Dette er et relativt enkelt konsept, men kan være relativt komplisert å løse.

Den tredje oppgaven er design og koding av et tegnespill som kan kobles til vår ferdigtrente AI, slik at den kan gjette på tegningene som sendes inn. Vi planlegger at Alen skal trenes på noen

spesifikke emner, og tegningene også vil høre inn under disse emnene. Et eksempel av hvordan dette kan fungere er at vi trener AI på bilder av ting som er relativt like, og gir disse som emner som skal tegnes av bruker. Eksempel her kan være eple og melon, eller tre og brokkoli.

Vi skal benytte oss av kunnskap fra et bredt spekter av de tidligere emnene våre. Elementer vi drar inn vil bli hentet fra disse fagene spesielt:

Emne	Elementer vi skal bruke/sette oss dypere inn i
AI	Bildegjenkjenning.
Grafikkprogrammering	Lage en tegnefunksjon og utforme et grafisk grensesnitt ved hjelp av OpenGL.
Cloud Technologies	Innhenting av bilder via Google Images API. Det er disse bildene som skal bli brukt for å trenere AI.
Programvareutvikling	Selve prosjektprosessen. Sette oss mer inn i bruk av Scrum i en større sammenheng. Planlegging og rapportskriving.
Designtenkning	Lage et brukergrensesnitt. Huske empati for brukeren når vi jobber med utformingen.
Datamodellering og databasesystemer	Lagre brukeres poengsummer så de kan brukes til et leaderboard.
Spillprogrammering	Spilldesign.
Programmeringsfokuserte fag	Backend. Det meste vil bli kodet i C++ og Golang, som er programmeringsspråk vi har lært i tidligere programmeringsemner.

Ettersom vi skal lage et spill, blir hovedfokuset liggende på AI og grafikk.

Prosjektorganisering

Ansvarsforhold og roller

Vi har valgt å ikke ha en scrum master. Vi er en såpass liten gruppe, og tror selv at det kommer til å fungere bedre uten. Vi har derfor ikke unike roller i dette prosjektet, og har alle ansvar for arbeidet som blir gjort både personlig, og sammen i gruppen. Siden vi er en liten gruppe på 3 som kjenner hverandre, og møtes i person for gruppearbeid minst en gang i uken, mener vi dette bør gå bra.

Vi tenker å omfavne agile development, og jobbe smidig både hver for oss og sammen på de ulike arbeidsoppgavene i hver scrum. Vi har delt prosjektet inn i egne biter og utvikler de hver for seg i rekkefølge. Slik har vi ikke et gruppemedlem som jobber med AI, og et annet medlem som jobber med koding av selve spillet.

Dette har både fordeler og ulemper. Blant annet gjør det at hver av oss får erfaring med koding av hver del av prosjektet, og at ingen setter seg fast på sin oppgave og ikke får hjelp av andre fordi de jobber med noe helt annet og har ikke satt seg inn i den relevante kode. Men det betyr også at arbeidseffektivitet ikke er optimalisert, da det er sjanser for å 'trække på tærne' til andre i gruppen. Vi tenker å organisere arbeidet slik fordi det virker mest logisk for oppsettet med 1-ukes sprinter.

Dersom vi ser at arbeid på samme del av prosjektet under hver sprint ikke funker bra, vil vi justere dette. Det er viktig at vi finner et oppsett som fungerer for gruppen slik at vi kommer i mål med prosjektet og får fullført et produkt som vi er fornøyde med.

For estimering av oppgavens tidsbruk skal vi forsøke å benytte [Planning Poker](#). Ved å bruke dette vil alle få frem meningen sin, og vi forhindrer at alle kaster seg på det første som ble sagt.

Vi skal sette opp en sprint backlog til hver sprint. Her vil alle arbeidsoppgaver som skal gjøres i løpet av sprinten være. Etter hvert som vi blir ferdige med arbeidsoppgavene våre, vil vi derfor kunne plukke ut flere på en lett måte. Når man velger en arbeidsoppgave, så skal vi bruke branching for å hindre merge-conflict også koordinerer vi merging med main branch på discord. Dette vil gjøre utviklingen mye enklere.

Rutiner og regler i gruppa

- Ca 7 timer i uken - ser an hvordan fremgangen av prosjektet ligger an og justerer
- Møte fysisk på starten av sprinten (minst en gang i uken)
- Eget ansvar for issue tracking
- Eget ansvar for time logging
- Om en ikke vil kunne nå et møte, vil melding bli gitt over discord
- Dersom en i gruppa faller bak skjema i forhold til arbeidstimer, vil det bli tatt opp i gruppemøte først. Om det fortsetter, vil fagleder bli kontaktet.
- Om vi sitter fast med den delen av prosjektet som vi jobber i sprint på, og ikke kan få hjelp innen den nåværende sprint (f.eks noe relatert til spillprogrammering, [IMT3603](#)), vil vi bytte til arbeid med en annen del av prosjektet og utsette dette til neste sprint eller når det passer best.

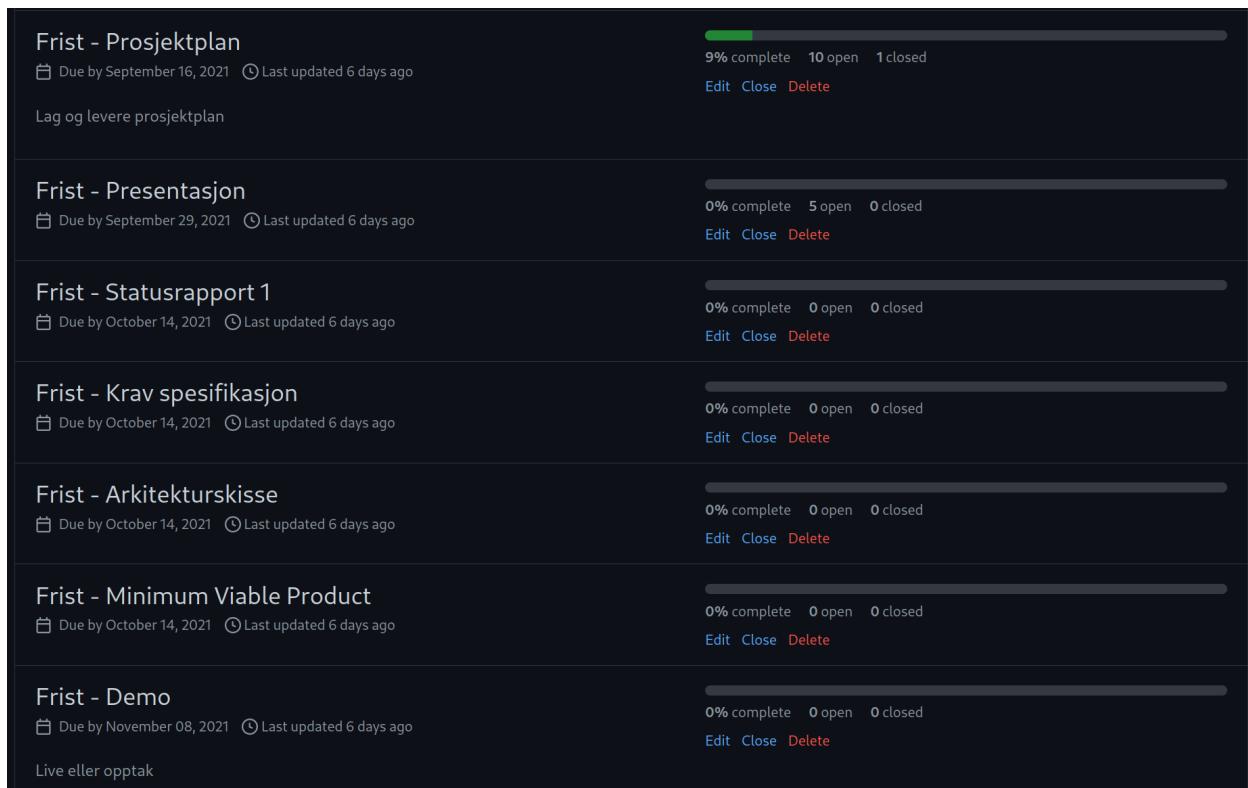
Planlegging Oppsett av Scrum

Helt i starten av prosjektarbeidet planlegger vi å ha en sprintlengde på 1 uke. Etter hvert som vi får satt oss ordentlig inn i prosjektet, og ser hvor lang tid prosesser faktisk tar, vil dette kunne utvides.

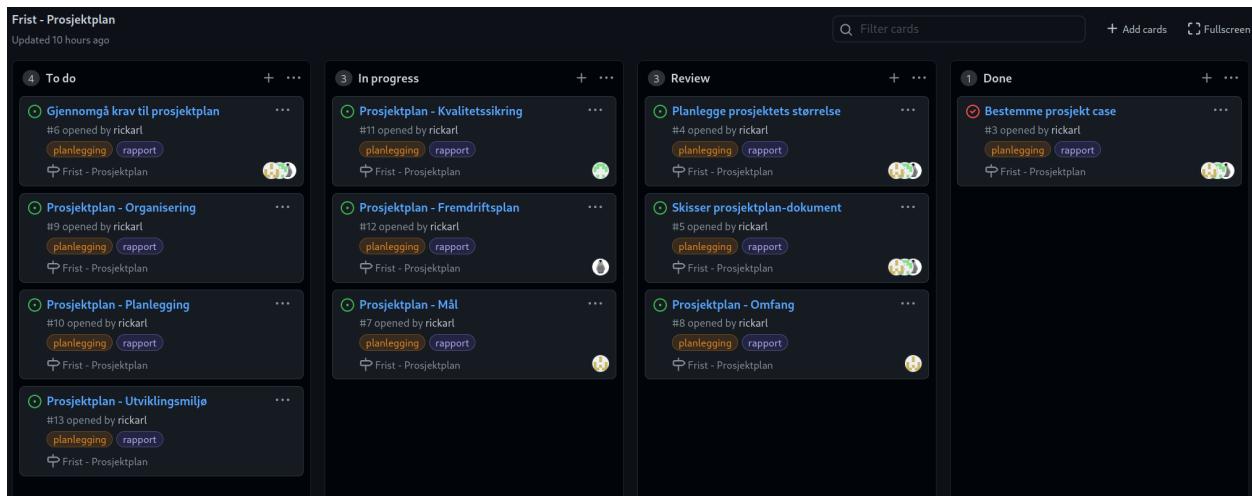
Ettersom vi skal ha møter/arbeidsøkter felles på tirsdager og onsdager, vil diverse scrum-møter settes opp på disse dagene. Dagene vi har planlagt med felles jobbing er rett etter hverandre på starten av uken. Derfor tar vi både sprint review og sprintplanlegging på tirsdager. Vi har valgt å kombinere sprint review-møte og sprint retrospective-møte ettersom vi er en liten gruppe med begrenset tid hver uke. Vi skal også forsøke å ha daglige scrum-møter på de dagene vi jobber felles. Ved bruk av dette kan vi se hvordan fremgangen til hvert enkelt gruppemedlem er, og kartlegge problemer som oppstår underveis.

Issue Tracking (Product Backlog)

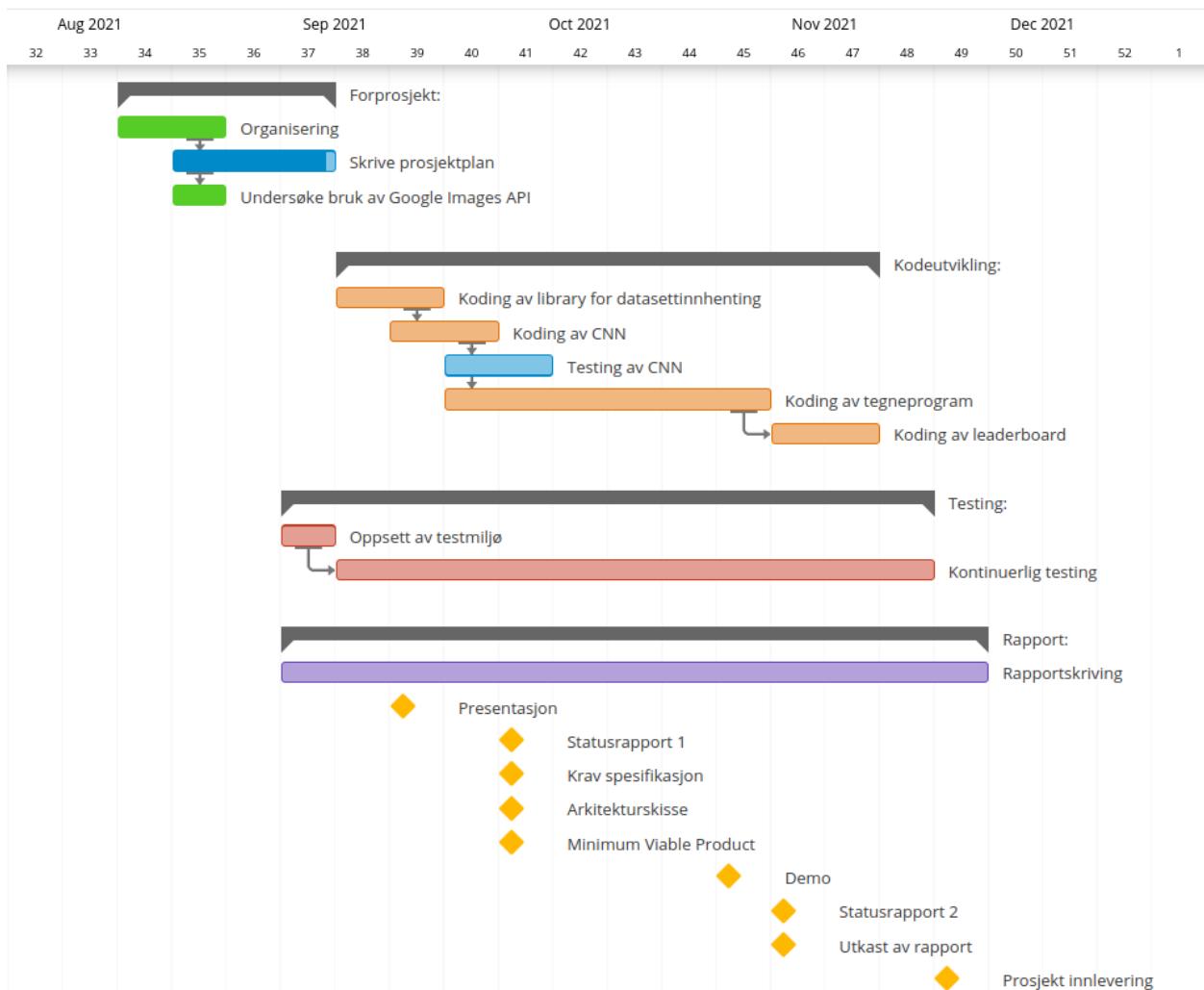
Vi bruker [GitHub Projects](#) for å plotte Product Backlog. Dette gir oss en god oversikt og tillater integrasjon med [GitHub Milestones](#) for å se progresjonen.



To do kolonnen representerer arbeidsoppgaver som ikke har blitt startet på. **In progress** viser hvilke oppgaver som blir jobbet på (Her må vi tildele oppgaven til oss selv slik at vi vet hvem som jobber på det). **Review** viser ferdige oppgaver som skal bli gjennomgått mot slutten, dette blir en slags QA (Quality Assurance). Det siste er **Done**, dette viser ferdige arbeidsoppgaver.



Oversikt over prosjektperioden



Testing blir gjort på alle *pure functionality* for å opprettholde defensiv programmering. Vi kommer til å bruke testing bibliotek som vi finner underveis.

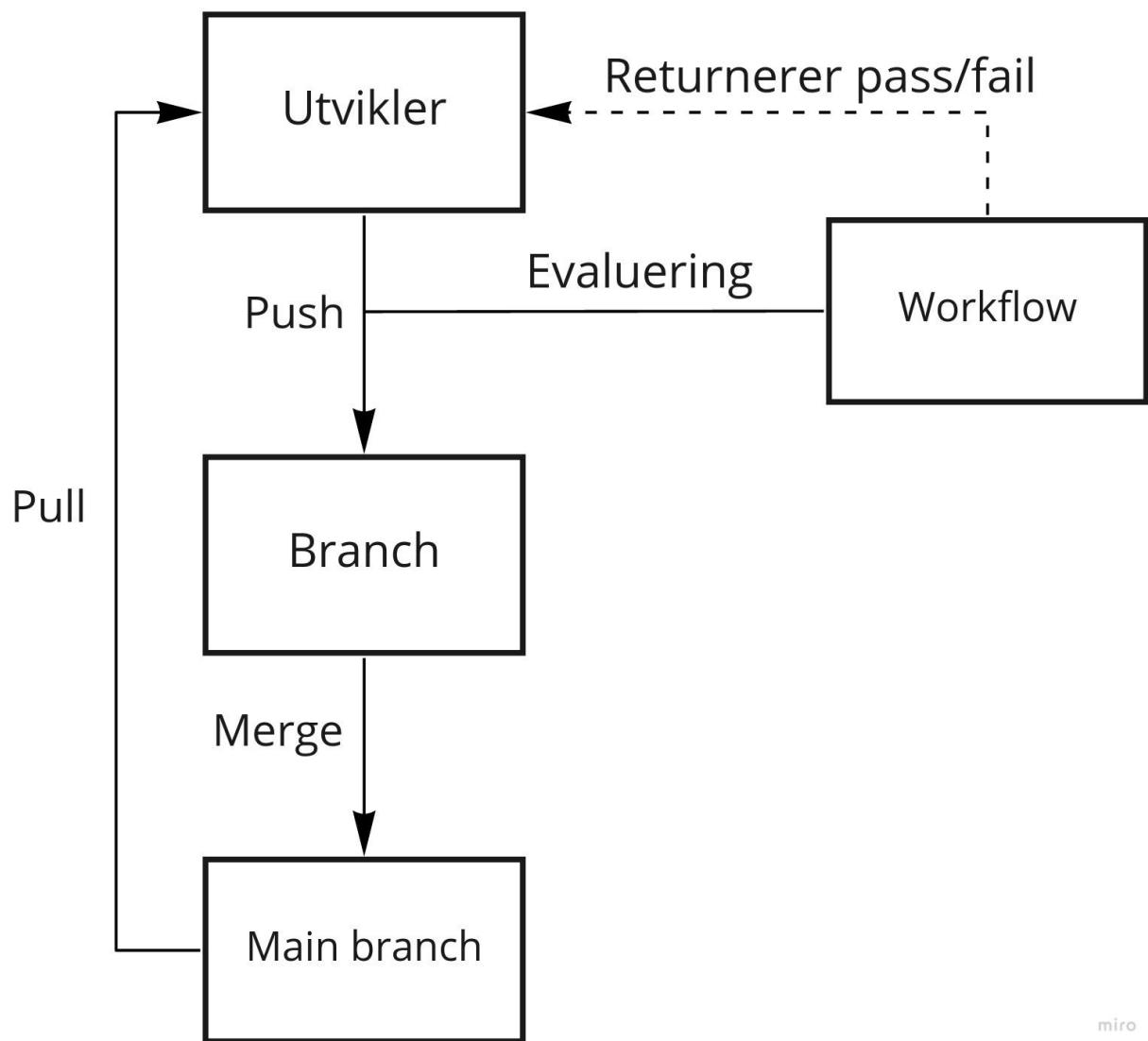
Milepæler i prosjektperioden

Vi har delt inn selve utviklingsperioden i 4 hoveddeler:

1. Bibliotek for datainnsamling: Bibliotek for datainnsamling skal gjøre modell trening mer effektiv og dynamisk.
2. Convolutional Neural Network: Convolutional Neural Network modell
3. Tegneprogram: Tegneprogram laget med OpenGL bibliotek for C++
4. Leaderboard: Leaderboard som kan hentes fra en server via API
5. Rapport: Rapport for prosjektet

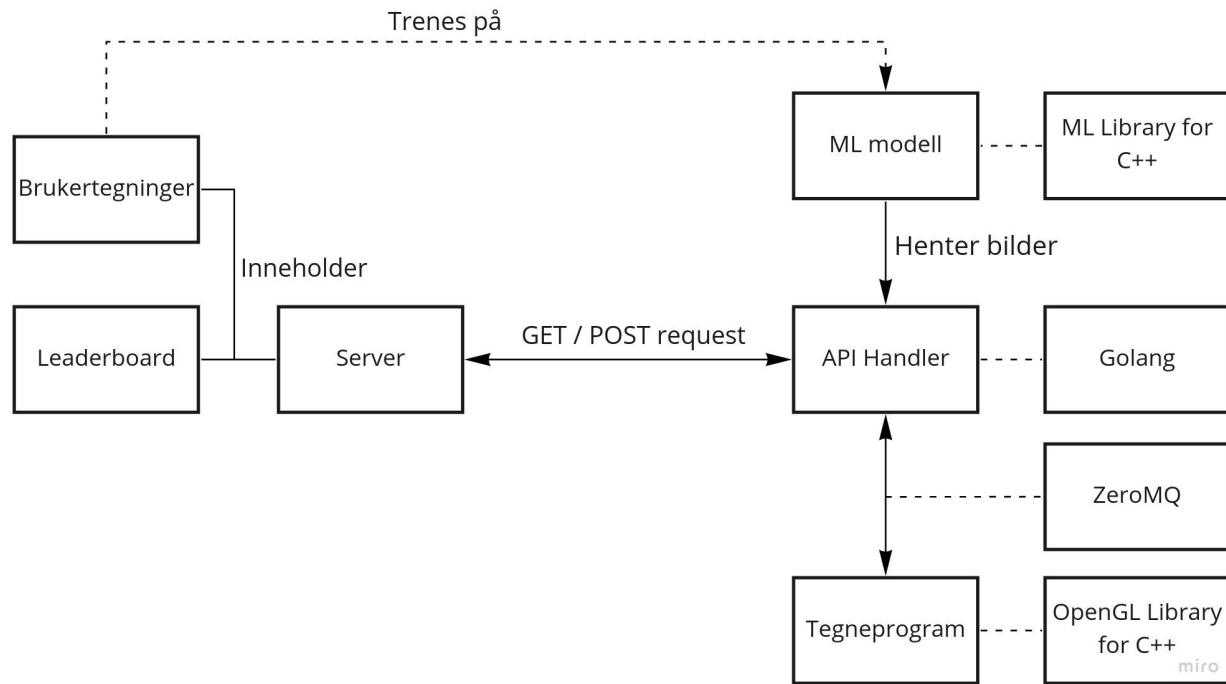
Skisse over utviklingsmiljø

Versjonskontrollsysten



miro

Nettverksoppsett



Kvalitetssikring

Plan for verktøybruk

Dokumentasjon

Vi kommer til å bruke en rekke ulike verktøy for dokumentasjon av vårt arbeid og fremgang i faget. En midlertidig liste av verktøy som vi vil bruke er:

- Rapport-innleveringer: Google Docs, samt GIMP/Paint for bilderedigering
- Status og fremgang: Issue board/prosjektplan på GitHub
- Logging av arbeid: Google Sheets, møtereferater på GitHub wiki
- Supplerende materialer: GitHub wiki
- Kode: Kodekommentering, git push/pull-kommentering
- Gantt-skjema laget med Instagantt

Verktøybruk

I tillegg til verktøyene allerede nevnt under dokumentasjon, planlegger vi å bruke følgende:

- Microsoft Visual Studio Code som IDE
- Kode skrives i C++ og Golang, vi planlegger bruk av ZeroMQ for å kommunisere mellom
- Planlegger å bruke Tensorflow og Keras biblioteker for C++ til design av AI
- Vi skal bruke OpenGL bibliotek for C++ til å lage selve tegneprogrammet
- Datalagring blir gjort veldig enkelt i JSON format

- Vi har ikke avklart om vi skal bruke biblioteker til selve spillet, og skal undersøke det
- En eller annen slags persistent cloud storage for leaderboards, TBD
- GitHub Workflows for automatisk testing av commits (Vil redusere feil i main branch)

Kildekode

For øyeblikket har vi ikke noen planlagte kilder for kode - siden vår erfaring med keras/tensorflow i AI er i python, vil vi ikke gjenbruke noe av vår egen kode eller den koden vi lånte i tidligere semester.

Når det er sagt kan vi gjøre noen sikre spådommer på hva vi kan få bruk for. Vi vil garantert komme til å bruke kildekode fra Keras og Tensorflow-bibliotek samt trolig lån av eksempelkode for disse. Vi vil også helt sikkert gjenbruke kildekode for selve spillet vårt. I den ferdige rapporten vil vi lenke til all kode som brukes både i kode-kommentering samt i denne delen av rapport-dokumentet.

Risikovurdering

I dette prosjektet blir det mange ulike deler som må settes sammen til et ferdig prosjekt. Vi har jobbet sammen for å forsøke å finne trolige risikoer som vi må ta høyde for i prosjektet.

Vi vurderer både sjansen for at en risiko kan materialiseres, og hvor sterk påvirkning det kan få på prosjektet. Vi deler disse inn i fire kategorier:

Styrke	Sjanse	Påvirkning
Høy	Stor sjanse - vil trolig inntrefte	Risikerer å velte hele prosjektet
Medium	Reel sjanse - bør ta høyde for	Vil ha stor påvirkning på frister og plan
Lav	Liten sjanse - kan kanskje inntrefte	Relativt liten påvirkning
Svært lav	Usannsynlig - kan ignorere	Minimal påvirkning

1. Google Image API

Vi vil bruke denne til innsamling av data til dataset for å trenere opp vår AI. Vi har funnet ut at den kan brukes gratis, og med greit antall bilder per måned for 3 personer. Men vi har ikke kodet program som faktisk bruker denne, og vet ikke hvordan det går.

Vi har tidligere jobbet med ulike APIer i for eksempel Cloud, og har ok tro på at arbeid med denne APIen skal gå fint. Der problemet kanskje kan inntrefte er selve nedlastingen av bildene og om vi møter på problemer med rate limiting eller lignende fra Google's API.

Sjanse: Medium

Påvirkning: Lav

Løsning: Vi kan finne fullstendige datasett manuelt, og bruke de i stedet. Kan også forsøke å trenere AI på brukerlagde bilder etter hvert.

2. AI-kvalitet

Siden datasettene vi bruker trolig vil hovedsakelig være bilder av virkelige ting (for eksempel epler eller hunder) kan den ha vanskeligheter med å sammenligne med tegninger. Dette er særlig sant dersom vi reduserer informasjonen i bilder, for eksempel ved å bruke grayscale eller rescaler bildene til å bli svært små.

Litt dårlig performance er ikke problem, men i verste fall kan den få en gjenkjennings-prosent nede på kanskje 10 - 20%. I så fall vil ikke spillet funke så veldig bra.

Sjanse: Lav

Påvirkning: Lav

3. Koding av spill

Vi planlegger et veldig basic tegnespill. Vi har ikke hatt spillprogrammering ennå, ei heller erfaring med å kode lignende spill før. Det er mulig dette blir utfordrende. Den store risikoen her er at vi ikke får til å kode et slikt spill. Vi har dog stor tro på at dette ikke kommer til å inntreffe, da spillet vi lager ikke er særlig komplisert.

Sjanse: Svært lav

Påvirkning: Medium

4. UI-design

Vi har ikke designet UI i tidligere programmeringsprosjekter, bare i designtenking som ikke hadde kode-relatert arbeid. Knytting av UI til spillet kan bli en ny utfordring. I tillegg må vi passe på at UI som vises til bruker, jobber godt sammen med kodebasen slik at bruker ikke erfarer bugs med for eksempel plassering av mus eller andre problemer som kan påvirke tegningen.

Sjanse: Lav

Påvirkning: Lav

5. Utvidelse av case

Eksempler her kan være online multiplayer eller spill-moduser. Siden dette vil for eksempel innebære en online løsning for distribuering av spillet, vil det gi mange utfordringer som vil kunne være vanskelig eller umulig å løse. Fordelen her er at det ikke er en risiko for vår kjerne-case, og derfor vil tillate oss å fremdeles produsere et bra produkt.

Sjanse: Medium

Påvirkning: Svært lav

6. Fordeling og utførelse av prosjekt

Her er det en risiko for at en eller flere av prosjektmedlemmene ikke klarer å utføre sin del av prosjektarbeidet. Dersom dette skulle inntrefte, vil dette bety mer arbeid for de andre medlemmene av gruppen, eller eventuelt at arbeid blir utsatt på ubestemt tid. Påvirkningen av en slik risiko er veldig vanskelig å estimere, da mange ulike årsaker som har ulik innvirkning faller under denne risikogruppen. I beste fall kan det være småsyke, der studenten kan jobbe hjemmefra uten problem. I verste fall vil det innebære en student som hopper av faget og forlater prosjektet helt.

Vi tror ikke de mer katastrofale scenarioene har særlig stor sjanse for å inntrefte - vi er studenter som kjenner hverandre greit, og har mulighet til å jobbe hjemmefra gjennom korona smitte eller lignende. Mindre pauser i prosjektplanen vår er mer sannsynlig.

Sjanse: Lav

Påvirkning: varierer

7. Koding av AI

Gruppen har tidligere erfaring med design og koding av AI i Python, men har ikke tidligere erfaring med dette i C++, som vi planlegger å bruke i dette prosjektet. Vi vil bruke C++ både for å øke vår erfaring og kompetanse, samt for å redusere bruk av ZeroMQ eller lignende teknologier for kommunikasjon mellom programmeringsspråk. Det er en risiko for at vi støter på problemer i denne delen av prosjektet, dersom det skulle vise seg vanskelig å bruke AI-relaterte biblioteker som for eksempel Keras og Tensorflow.

Sjanse: Lav

Påvirkning: Medium

8. Bruk av ZeroMQ/andre kommunikasjonsteknologier

Gruppen har veldig liten erfaring med ZeroMQ og andre teknologier som muliggjør kommunikasjon mellom flere programmeringsspråk. I tillegg er den erfaringen vi har bare for andre programmeringsspråk. Bruk av slik teknologi vil være sentral for fullførelsen av prosjektet. Dersom vi møter problemer ved implementasjon av slik teknologi, vil det kunne skape problemer.

Sjanse: Medium

Påvirkning: Medium

Løsning: Det er ingen enkel måte å omgå dette problemet - vi vil prioritere læring av ZeroMQ for Golang og C++ ganske høyt når vi når denne delen av prosjektet.

9. Testing

Vi kommer til å trenge tester for flere ulike komponenter av programmet. Dersom vi får problemer under testing med for eksempel oppsett av testing environment, mange feil under

tester, eller at arbeidemengen med design og kode av tester blir for stort, vil det ha påvirkning på fremgangen i prosjektet.

Sjanse: Medium

Påvirkning: Medium

Løsning: Vi vil forsøke å innhente kompetanse innen testing i C++ som vi mangler, og sette av tid til arbeid med testing i hver sprint, for å forsøke å holde prosjektet a jour i forhold til testing.

Vurdering

Generelt er prosjektet vi har valgt et lavrisiko-prosjekt. Siden den ikke trenger noe brukerdata annet enn et valgfritt kallenavn for leaderboard, trenger vi ikke å ta høyde for behandling av privat data eller lignende. På samme måte bruker ikke prosjektet noe sensitiv data, ei heller integrerer det teknologier eller applikasjoner som trenger login, passord eller lignende.

I tillegg er MVP generelt veldig greit, og en del av kompleksiteten i prosjektet ligger i utvidelse fra hoved-case. På denne måten er risikoen for problemer ved disse minimert, da påvirkningen kan minimaliseres så lenge vi viderefører kjerneproduktet og kan lene oss på det om nødvendig. I tillegg har vi flere 'backup' strategier vi kan bruke om nødvendig for å nå MVP - dog håper vi å unngå disse.

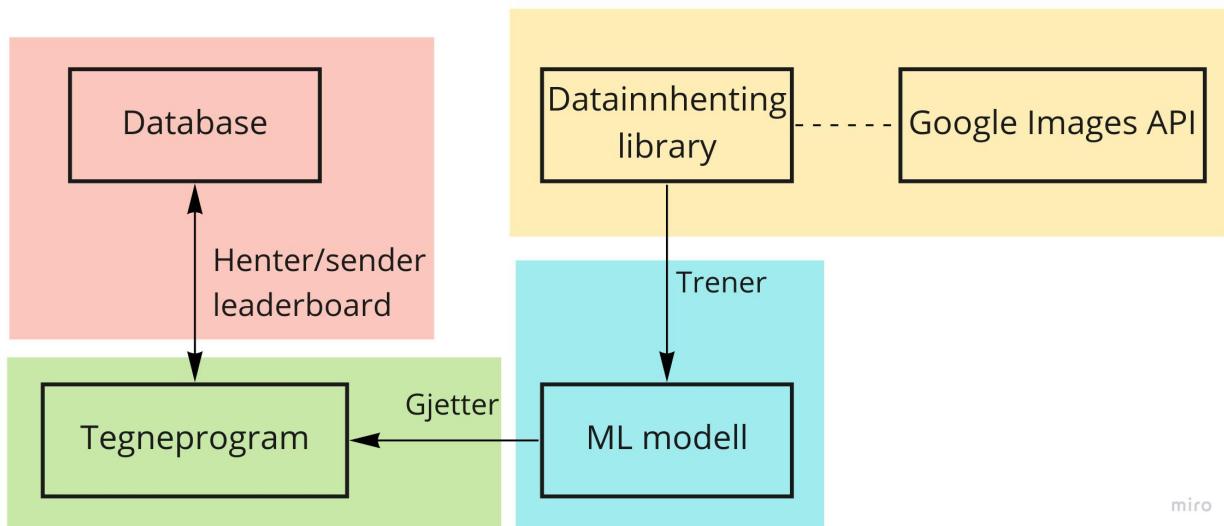
		Påvirkning		
Sjanse	Svært lav	Lav	Medium	Høy
Svært lav			3	
Lav	6?	2, 4, 6?	7, 6?	6?
Medium	5	1	8, 9	
Høy				

Slik vi analyserer prosjektet vårt, er det bruk av ZeroMQ som utgjør den største risikoen for prosjektet. Dette er fordi ingen i gruppen har særlig erfaring med teknologien - særlig for våre valgte språk - og fordi det ikke er noen enkel måte å 'jobbe rundt' bruken av denne - det er et krav for prosjektets suksess at vår nettverks-kode i Golang og spill/AI-koden i C++ kan kommunisere sammen. Vi mener derfor at det vil være naturlig å prioritere kompetanseinnhenting og design/testing av denne teknologien relativt tidlig i prosjektet, slik at vi kan få dette på plass og eliminere risikoen, eventuelt sette av mer tid til løsninger av dette.

Fremdriftsplan

Det første vi gjorde i prosjektet etter at vi bestemte konseptet vi vil forsøke å gjennomføre, var å finne fram til vårt MVP - det minste vi vil trenge for å faktisk ha noe å presentere. Vi bestemte at for dette vil selve bildegenkjenning-modellen passe for oss. Om vi manuelt sender den bildet og kjører bildegenkjenning på det har vi et fungerende, dog skuffende, resultat.

Etter å ha funnet frem til dette, delte vi prosjektet inn i mindre ‘biter’ som vi planlegger å jobbe med sekvensielt - men denne planen kan selvsagt endre seg ettersom vi ser om vi trenger tre personer til å jobbe med hvert steg. Denne inndelingen kan ses på bildet under:



Den første delen av prosjektet vi vil fullføre er bibliotek for datainnhenting. Vi tenker å hente data fra Google Images API, og må kode et program som tar seg av dette. Dette vil være vår første sprint etter at prosjektplanen er ferdiggjort.

Del to består av selve bildegenkjenning-Alen. Her vil vi også bruke noe tid til kompetanseinnhenting, da vi vil kode den i C++, men ingen av gruppemedlemmene har erfaring med koding av AI i C++, så grundig gjennomgang av Keras og Tensorflow i C++ vil være nødvendig her.

Del tre består av kodingen av selve spillet, samt presentasjonen av det i form av UI. Vi estimerer at dette arbeidet vil være det mest tidkrevende i prosjektet, men siden vi ikke har tidligere erfaring med koding av spill eller lignende applikasjoner, gjetter vi bare her. Siden det er vanskelig å estimere arbeidsmengde vil vi uansett ha så mye tid som mulig til denne delen av prosjektet, og håper å kunne sette i gang med del tre innen midten av Oktober, om ikke før.

Del fire består bare av oppsett av et leaderboard som registrerer hvem som klarer å best tegne slik at Alen klarer å treffte når den gjetter. I tillegg muligens noe arbeid med ZeroMQ eller

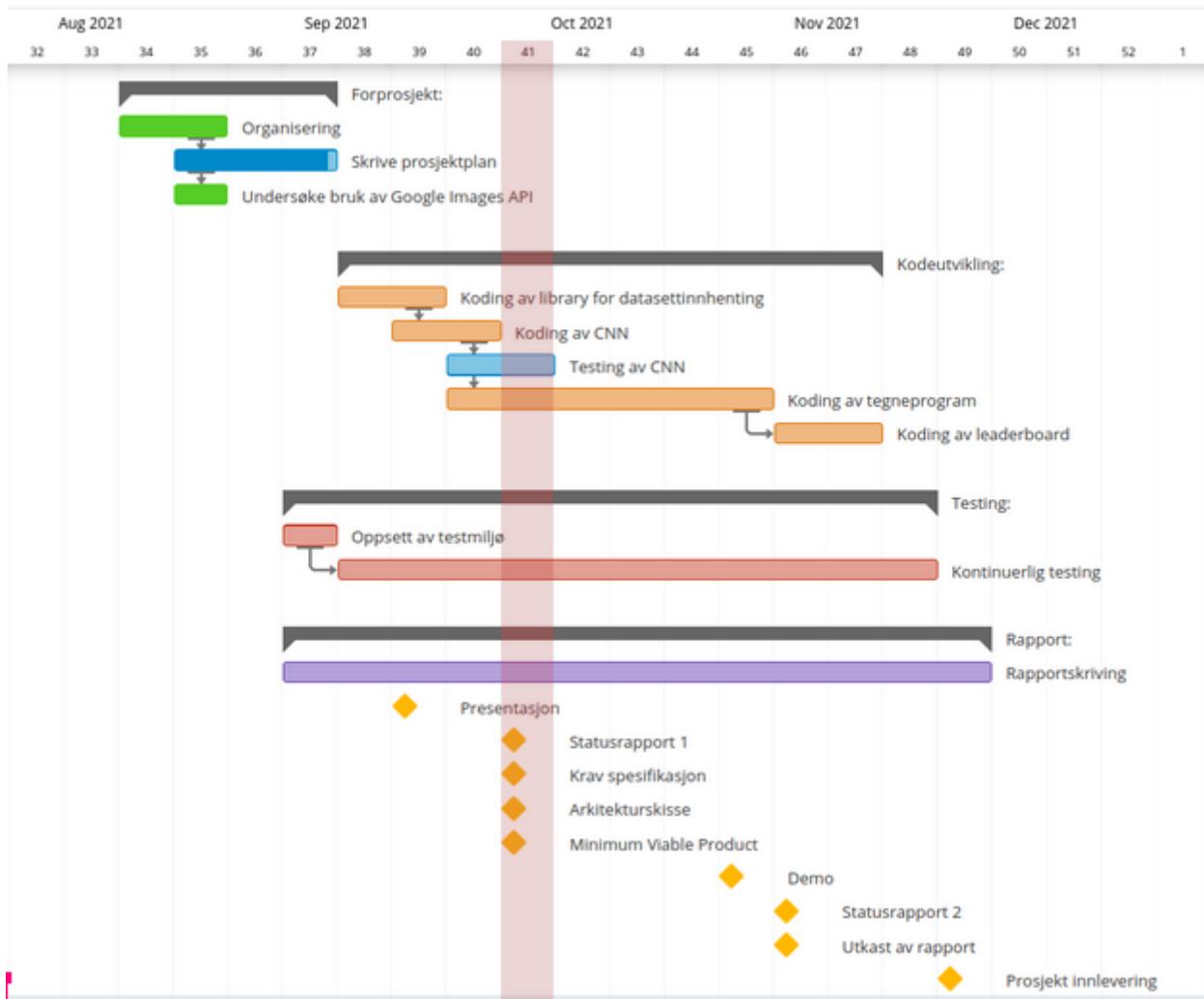
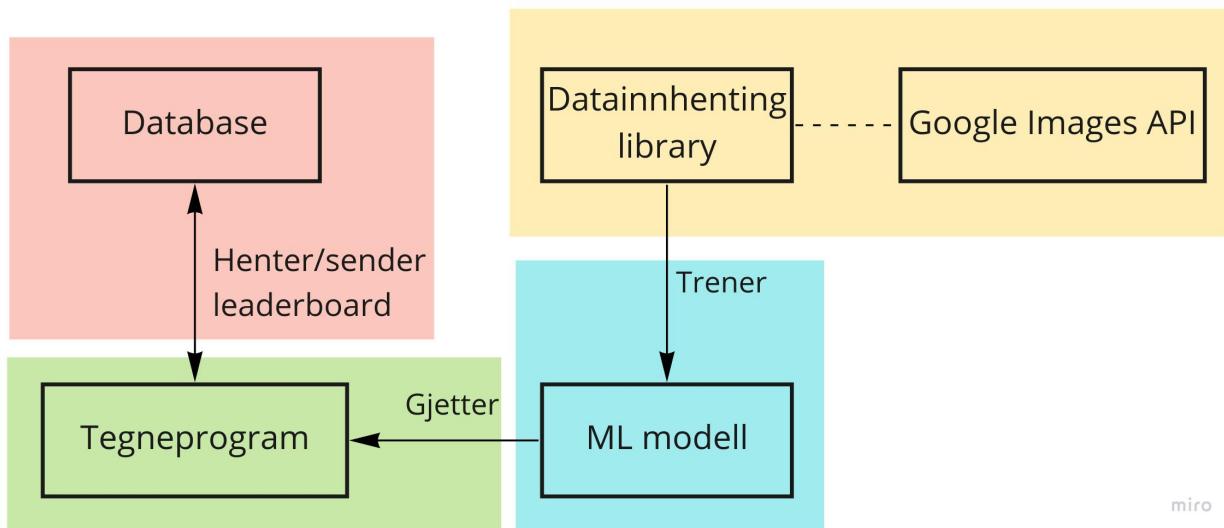
lignende for å sette opp skikkelig kommunikasjon. Vi tror ikke denne delen vil innebære veldig mye arbeid, og dette er også den funksjonaliteten som har minst påvirkning på resultatet vårt.

Dette er alle de fire obligatoriske delene av vårt prosjekt. Dersom vi blir ferdig med disse i god tid før fristen er det flere utvidelser og forbedringer vi tenker å implementere. Ett eksempel er å utnytte brukernes tegninger til videre trening av AI, som vil blant annet kreve at vi lagrer og sender disse til en server som Alen ligger på.

I tillegg til de fire delene, vil både start og slutt av prosjekt innebære skriving av henholdsvis plan og rapport, som vi tenker vil ta ca en uke hver å gjennomføre.

I første omgang har vi en sprint for uken 01.09 - 08.09 som går ut på å fullføre så mye som mulig av prosjektplanen, samt sette opp GitHub, timelogging, møtenotater og lignende slik at bruk av disse blir raskere og enklere gjennom resten av prosjektet.

Statusrapport 1



Datainnhenting library

Vi er helt ferdige med koding av datainnhenting library.

ML modell

Fremgangen på ML-modellen har gått greit. Vi er ikke helt ferdig per 12.10, men vi jobber mot å bli ferdig innen MVP-frist 19.10. Vi tror ikke dette vil være et problem. Det er flere årsaker til at vi er litt forsinket på denne delen av prosjektet:

- Vi ble litt forsinket under kompetanseinnhenting til ZeroMQ.
- Vi måtte bruke en del tid enn forventet til kompetanseinnhenting for CNN i C++ - til slutt landet vi på og fikk installert MLPack, men det var nesten en uke bak skjema.

Tegneprogram

Vi har startet med å finne library til tegneprogrammet, og dette vil jobbes videre med fremover. Etter det vi har funnet ut fra innledende forskning på tegneprogrammer i C++, er vi fremdeles usikker på hvor vanskelig dette vil være - eksempler virker gjerne veldig enkle, eller veldig kompliserte, og vi har ikke satt oss skikkelig inn i hvilket kompleksitetsnivå vi må legge oss på.

Vi er også litt forsinket her, siden ML modell tok lenger tid enn forventet. Til tross for dette, har vi fremdeles tillit til prosessen vår, og tror vi vil komme i mål etter plan.

Leaderboard

Som beskrevet i Gantt skjema, har vi ikke startet på denne milepælen enda.

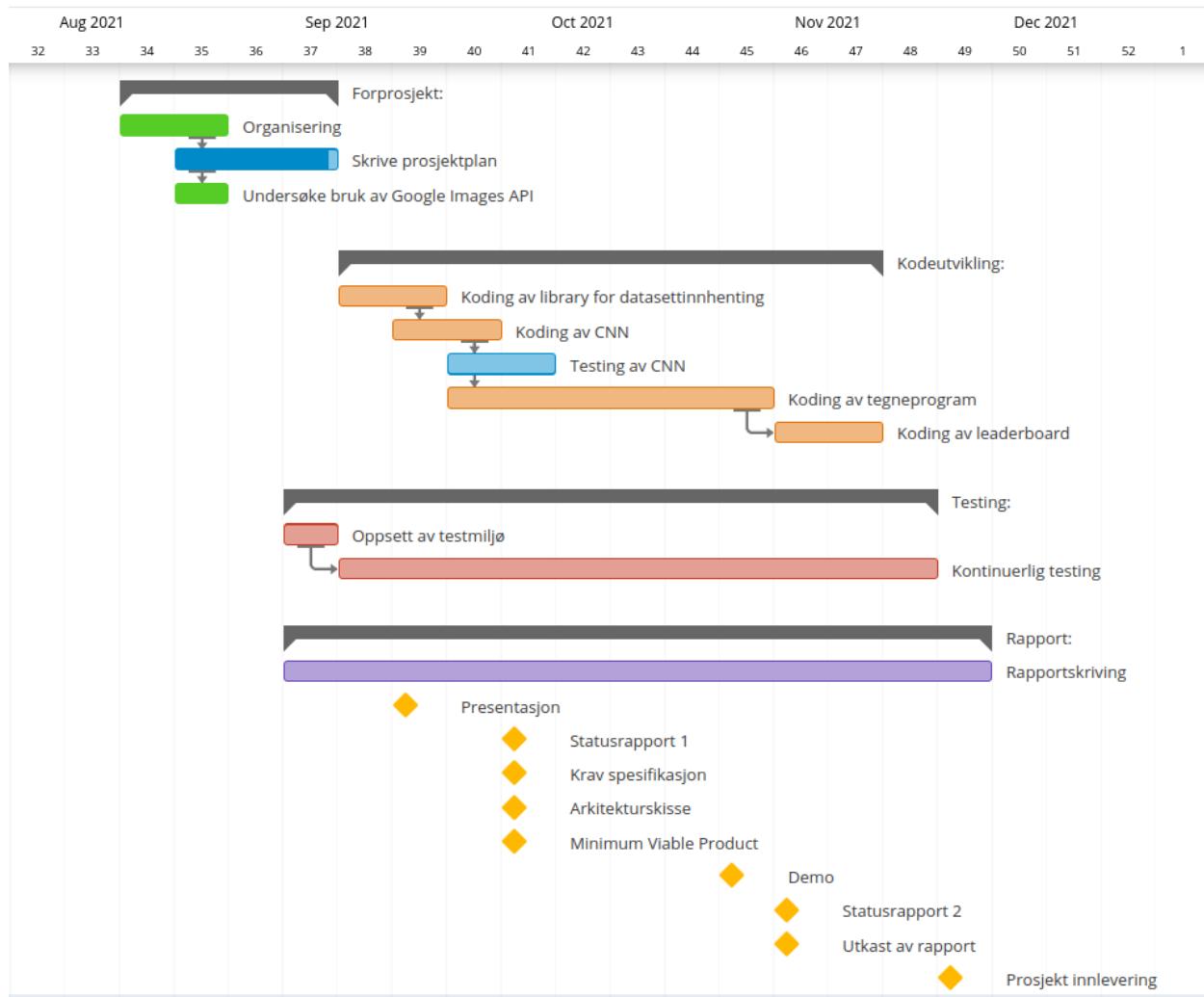
Testing

Vi har per i dag ikke laget noen tester. For datainnhenting library kunne vi ikke lage noen tester i frykt for at vi skulle bruke opp kvoten vår på Google Images API. Vi har heller ikke lagd noe pure functionality(funksjoner som ikke utnytter I/O funksjoner) enda som kan testes på.

Rapport

Vi har ikke startet med rapportskrivning enda, utover å ta notater, og holde GitHub-wiki oppdatert.

Statusrapport 2



ML model

Helt ferdig, skal legge til flere ord som brukeren kan tegne.

Tegneprogram

Vi har gjort god fremgang på tegneprogrammet, men vi er litt bak skjema her. Det største problemet vi har hatt er at vi har støtt på konflikter med imports i CMake, som har ført til både feilsøking og refaktorering av noe kode for å bruke nye løsninger. Selv om vi er noe bak skjema, gjør vi fremdeles fremgang jevnt og trutt.

De fleste bitene er på plass, så nå jobber vi med å flette alt sammen ved hjelp av å lage forskjellige "scenes". Det er disse som bestemmer hva som blir vist på skjermen til enhver tid.

Noen eksempler er hovedmenyen og spillrunden. Når dette er på plass må det legges inn selve spillfunktionaliteten, altså utdeling av poeng, ord og timeren.

Leaderboard

Vi har bestemt oss for å droppe dette ettersom vi må fokusere på å få ferdig tegneprogrammet.

Testing

Fordi vårt program ikke er egnet for enhetstesting på grunn av ‘non-pure’ funksjonalitet, har vi fokusert på andre typer kvalitetssikring. Vi har kvalitet testet alle commitsene ved bruk av GitHub Workflows. Her sjekker vi koden vår opp mot linting standarder og om programmet kjører før vi merger branchen med main. Vi har også hatt brukertester på wireframes før vi startet på tegne programmet.

Rapport

Vi har gjort veldig bra fremgang på rapportskrivingen, og vår foreløpige mal er lagt til. Det mangler noen deler som skal hentes inn/forbedres fra tidligere leveranser som kravspek og arkitektur, og vi har heller ikke skrevet ferdig rundt implementasjon av tegneprogrammet og resultater/konklusjon. Vi har også noe tekst som må trimmes vekk og flyttes på.

Status samlet sett

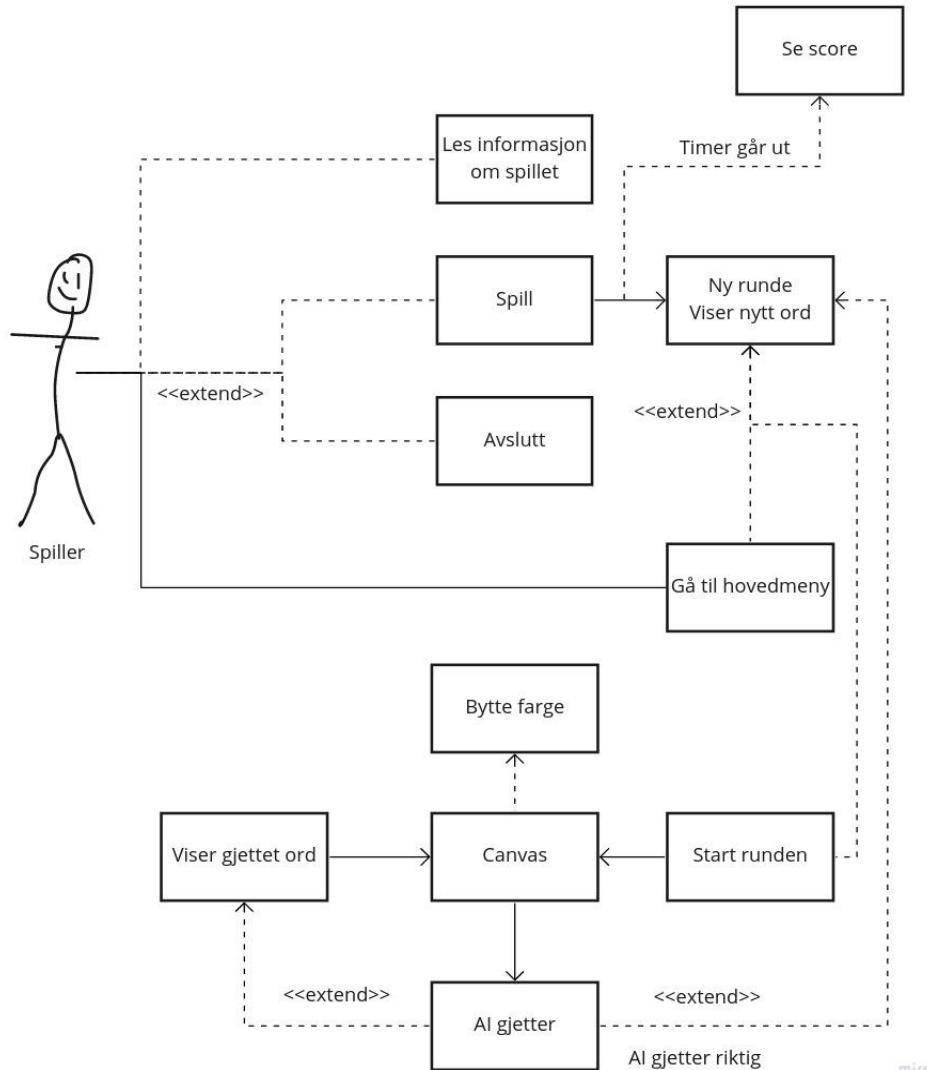
I forhold til planen vi satte ved starten av arbeidet, har leaderboardet gått ut, og slik sett har vi fått åpnet en del tid nå på slutten som vi ellers ikke ville ha hatt. Fremgangen på tegneprogrammet er litt bak skjema, mens vi er foran skjema på rapportskriving. Alt i alt mener vi at vi er på god vei til å fullføre tegneprogrammet, men det kommer til å stå på om vi har flere problemer under sammenkobling av moduler IFT CMake eller å lage en fungerende executable.

Kravspesifikasjon

Fordi vårt prosjekt har relativt liten frontend med bare en type samhandling mellom bruker og software, har vi dessverre veldig små use og misuse cases. På den andre siden er det positivt at vårt system ikke er veldig sårbart med henhold til sikkerhet!

Vi kan forestille oss en mer utvidet versjon av spillet, der det ligger tilgjengelig på en server (f.eks. gjennom openstack) og kan spilles i browser. Denne versjonen vil ikke ha særlig større use case for en vanlig user, men vil ha en del flere krav til operasjon og spesielt sikkerhet - ikke bare poeng i leaderboard men også inndata fra bruker, IP-adresser, etc vil være tilgjengelig uten bra sikring, og angrep som for eksempel man-in-the-middle vil være mer seriøse affærer.

Funksjonelle krav



Operasjonelle krav

Hovedprinsippe i produktet vårt blir brukt lokalt og krever ingen support fra oss, leaderboard krever at vi opprettholder connection mellom bruker og server.

Leaderboard-serveren har ikke særlig høye krav til tilgjengelighet (jamført CIA-prinsipper), men vi vil likevel at den ikke skal være nede i lange perioder, eller for mye i løpet av en måned.

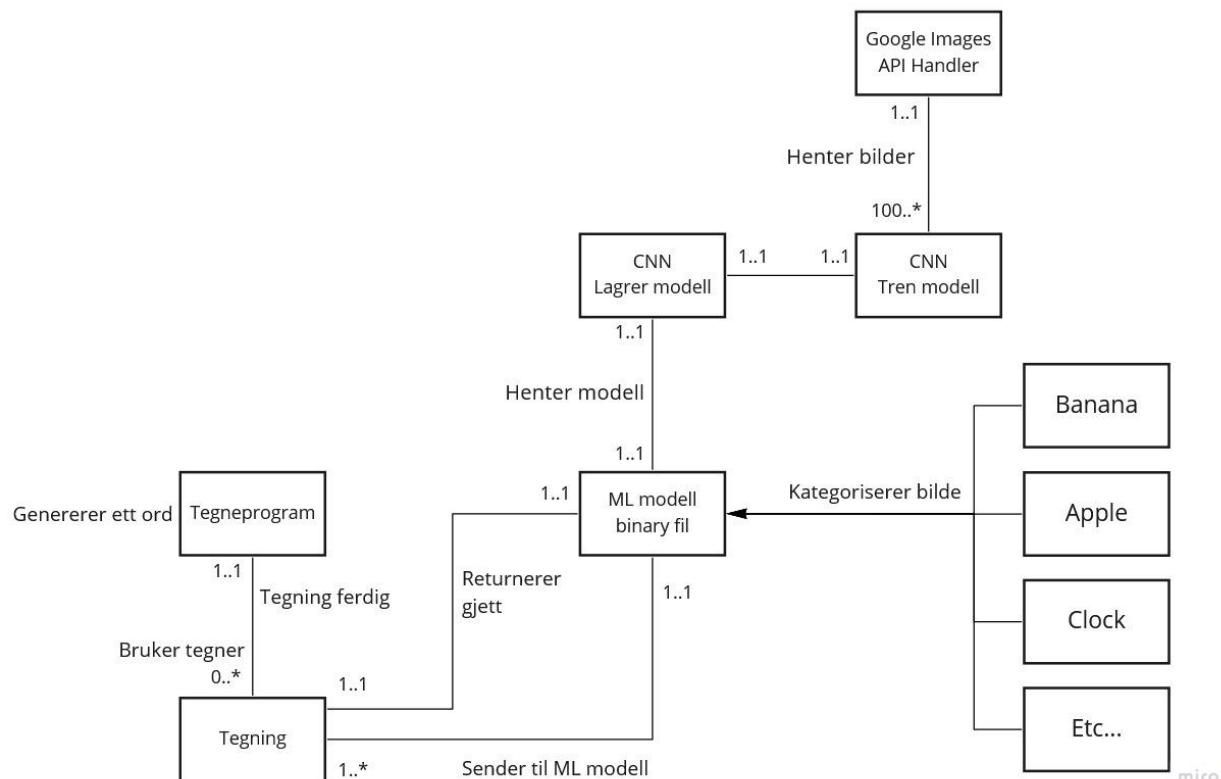
Siden vi ikke har full kontroll over serveren der vi vil laste opp leaderboard, tenker vi at å satse på et såkalt ‘two nines’ nivå på tilgjengelighet, altså 99%, passer best for dette. Vi vil måtte passe på å overvåke leaderboard-server og starte på nytt dersom den skulle stoppes for vedlikehold eller lignende.

Sikkerhetskrav og misbrukshåndtering

Utifra CIA prinsippene ser vi for oss at *confidentiality* ikke vil være et relevant prinsipp å forholde oss til ettersom vi ikke lagrer/sender noe sensitiv data. *Integrity* er relevant i forhold til juksing, hvor brukeren endrer på sin egen poengsum før den blir sendt opp. *Availability* er relevant for både ‘se leaderboard’ og ‘legg inn sluttpoeng i leaderboard’ hvor man krever fungerende server connection. Her vil DDOS være relevant hvor en angriper kan ta ned systemet.

Vi ser ikke for oss at risikoen eller innvirkningen på disse angrepene vil være noe særlig høyt. Angriperen kan ikke tjene noe på disse angrepene, og DDOS koster også angriperen penger.

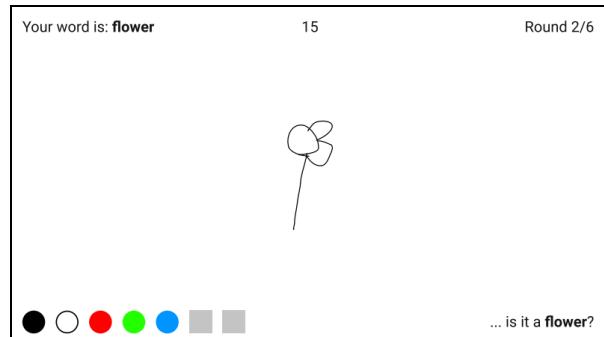
Domenemodell



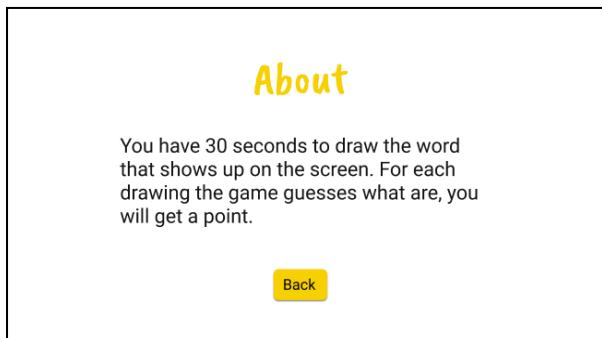
Grensesnittkrav



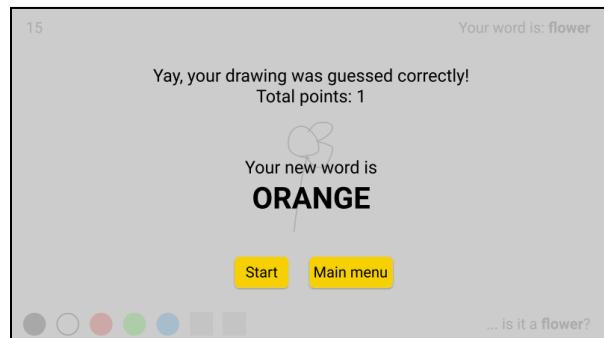
Hovedmeny



Spill



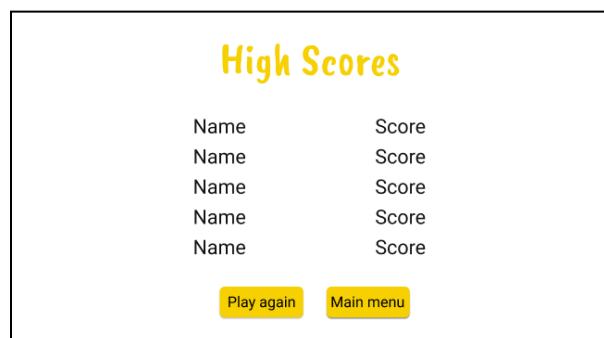
Om spillet



Nytt ord



Game over



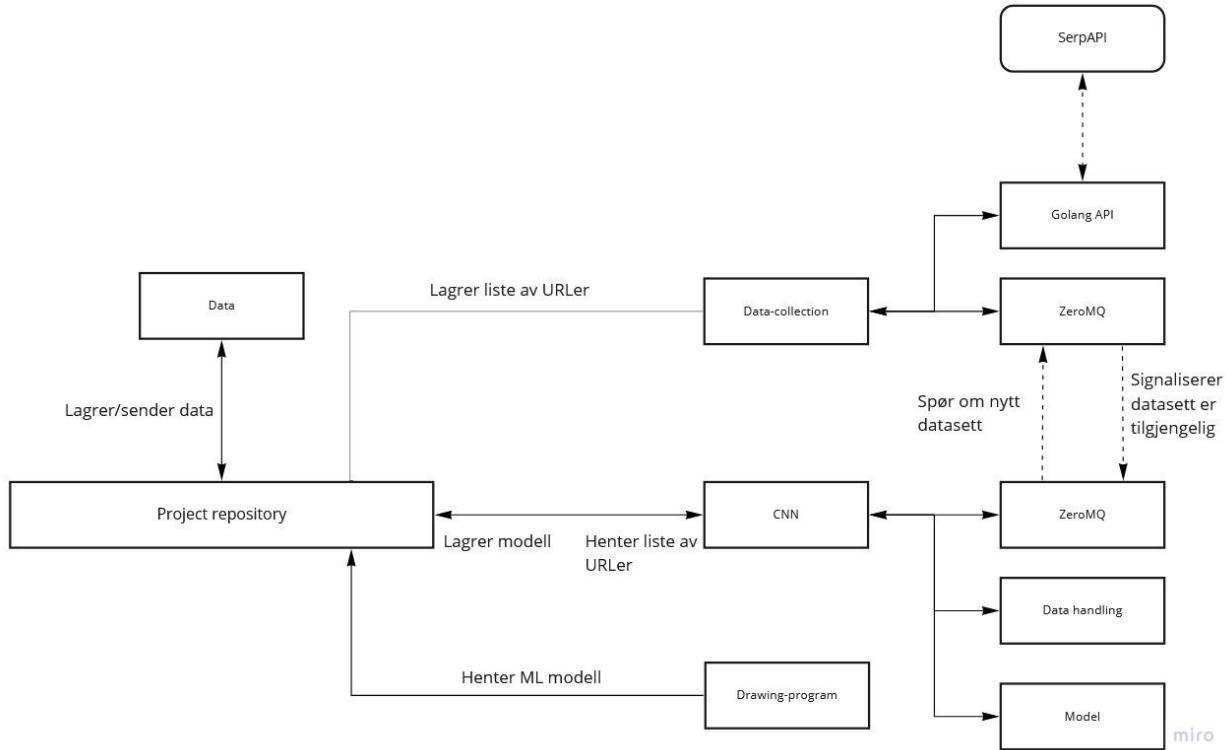
High scores

Over er prototypen til tegneprogrammet. [Interaktiv versjon](#).

Siden vi bare har begrenset front-end interaktivitet, og hverken bruker eller blir brukt av noen eksterne APIs eller lignende, har vi relativt enkle grensesnittkrav.

Arkitekturskisse

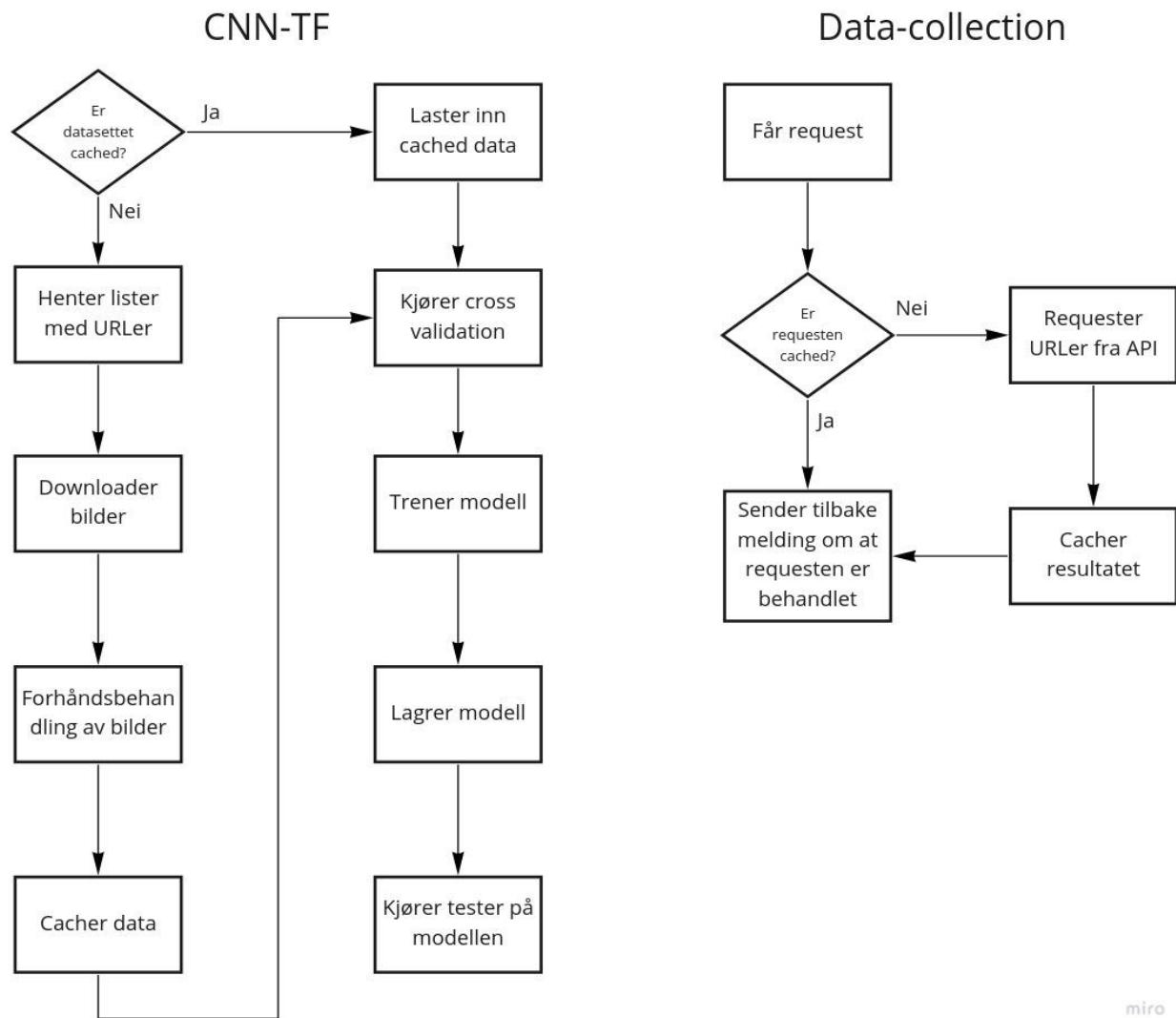
Prosjektstruktur



Vår repo er delt inn i fire deler, som vist over. CNN tar seg av kunstig intelligens, og trener den etter datasett. Data-collection henter inn datasett og sender de over til CNN. Drawing-program tar seg av selve spillingen, og bruker de trenede modellene i CNN. Resultater blir sendt til Leaderboard. Dersom vi får tid til å implementere selv-læring sender Drawing-program også tegninger til CNN for videre trening.

All data blir lagret i Data direktoriet som brukes av alle modulene (utenom leaderboard).

Nettverksstruktur



miro

Scrum board eksempler

Eksempelboard - rapport

Milestone - Rapport
Updated 2 minutes ago

Filter cards

2 To do + ...

12 In progress + ...

3 Review + ...

0 Done + ...

Automated as To do Manage

Automated as In progress Manage

Automated as Done Manage

Brukertester

Intervju 1

Oppgaven: Start et spill, «fullfør» det, og legg til poengsummen din på leaderboard.

1. Hvilke steg ville du tatt for å løse denne oppgaven?

Her gjorde alle det samme: «Start game» -> Tegne blomsten -> «Start» -> Skriv inn navn og «Submit»

2. Hvordan opplevde du navigeringen?

Person 1:

Enkelt og rett fram. Knapper og tekst var der jeg forventet.

Person 2:

Navigeringen var enkel, og det var lett å finne frem.

Person 3:

Lett å forstå, og oversiktlig.

Person 4:

Ser greit ut, den var naturlig.

3. Hva synes du om området fargene og verktøyet er plassert?

Person 1:

Det ga mening. Det har ikke så mye å si om de er plassert der eller på høyre siden av skjermen.

Person 2:

Det er fint det var plassert i et hjørne ettersom det er naturlig å se ned dit. Det burde være enten der det er, eller øvre høyre hjørne.

Person 3:

Plassert på et naturlig sted.

Person 4:

Den er plassert der man er vandt til at den er plassert. Den burde enten være der eller vertikalt langs siden.

4. Hvordan var det å finne informasjon om runden du er i?

Person 1:

Jeg slet litt med å skjønne at det var en timer, men regner med at det hadde endret seg hvis den hadde gått nedover.

Når du får et nytt ord, gjør det klarere hva som er ordet. Teksten over er nesten lik i størrelse.

Person 2:

Du får jo ordet før du kommer inn i runden, så kanskje ha nedtellingen der ordet er så det blir lettere å se.

Person 3:

Alt ga mening.

Person 4:

Skjønte hva jeg skulle gjøre, det jeg så er de minst viktige tingene (runde og poeng).

Intervju 2

Oppgave: Start et spill, tegn første ord, og avbryt spillet etter du har tegne ordet.

1. Hvilke steg ville du tatt for å løse denne oppgaven?

«Start game» -> Tegne en blomst -> Trykk på «Meny» når det nye ordet kommer

2. Hva er inntrykket ditt av designet? Har noe av det dårlig lesbarhet, eller er distraherende?

Person 1:

Alt virker greit.

Person 2:

På skjermen med neste ord, ville jeg hatt knappene under hverandre. Det ser rart ut når de står ved siden av hverandre.

Den gule fonten blir litt lys på den hvite bakgrunnen.

Person 3:

Det er oversiktlig, man ser alt i et blikk.

Person 4:

Knappene burde være over hverandre i stedet for ved siden av hverandre, dette er mest naturlig.

3. Hvordan var det å finne informasjon om runden du er i?

Person 1:

Det var fint at ordet og timeren hadde byttet plass, lettere å få med seg nedtellingen nå.

Person 2:

Det gikk greit.

Person 3:

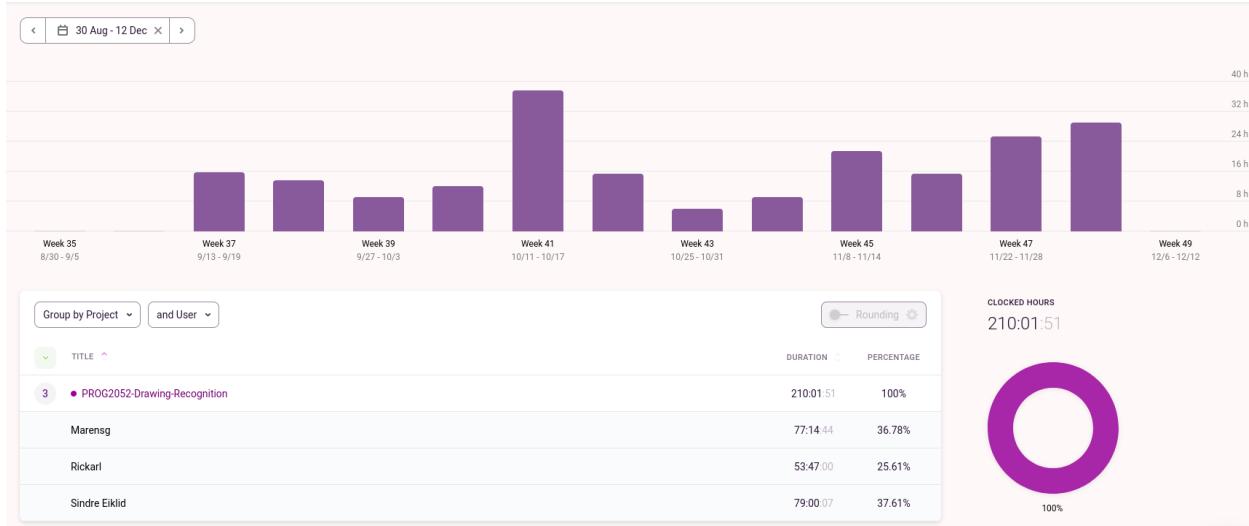
Det var greit, man ser alt lett med en gang.

Person 4:

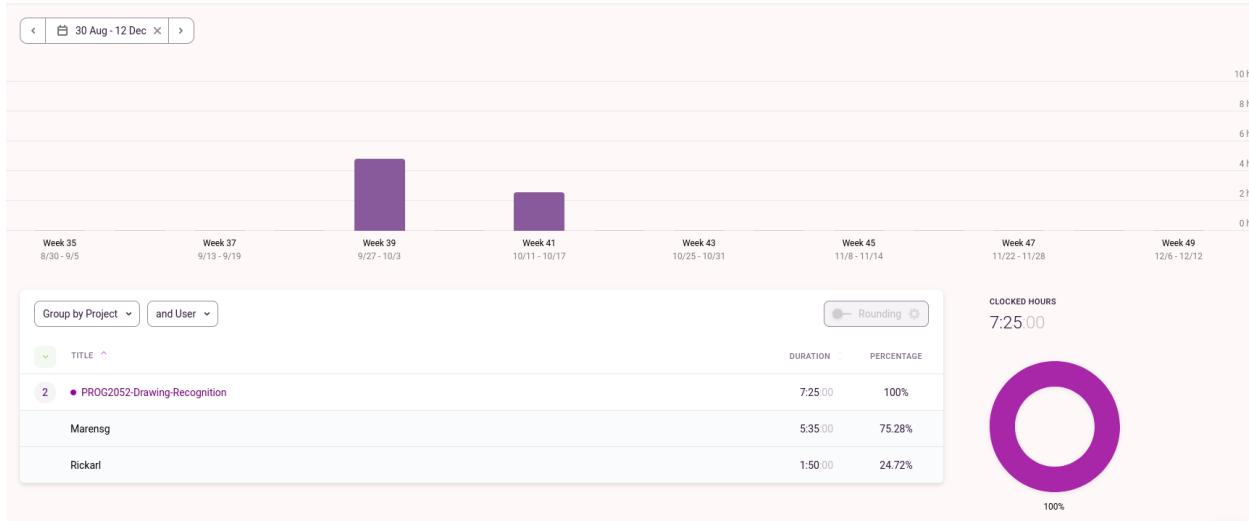
Fungerer bra. Nedtellingen burde kanskje vise 00:15, for å tydeliggjøre at det er sekunder, men det spørs jo hvor lang tid man skal ha.

Timelogger

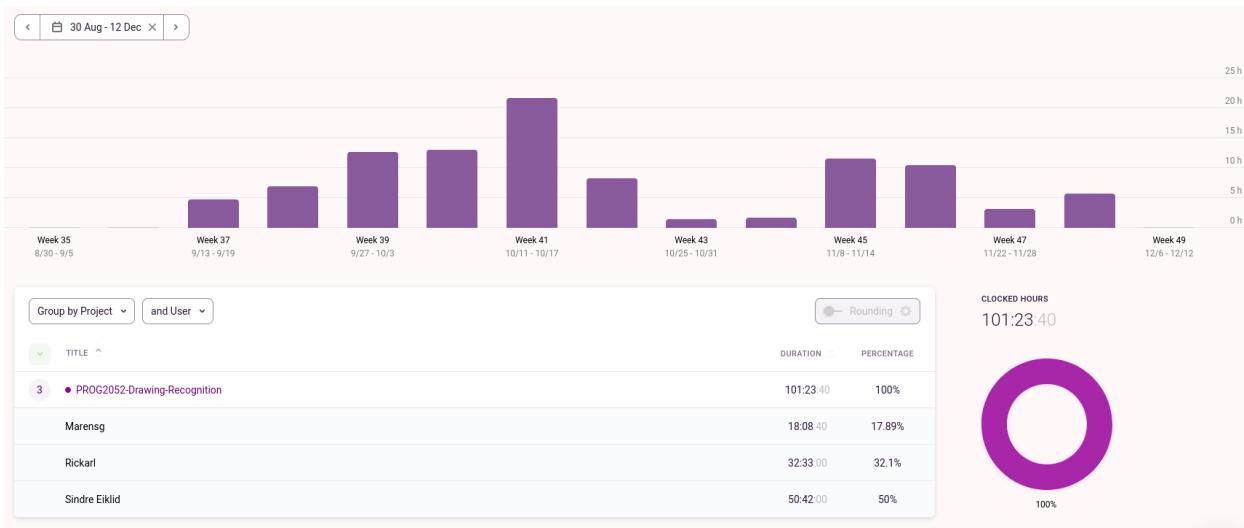
Koding:



Design:



Kompetanseinnhenting:



Rapportskriving:

