Hegdal, Emil Johannes Tillman
Hoel, Hans Kristian
Leiros, Vebjørn Fonstad

Beslutningsstøtte

Bacheloroppgave i Bachelor i Programmering [Spill | Applikasjoner]
Veileder: Hjelsvold, Rune

Mai 2021

Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU
Kunnskap for en bedre verden

Hegdal, Emil Johannes Tillman
Hoel, Hans Kristian
Leiros, Vebjørn Fonstad

Beslutningsstøtte

Bacheloroppgave i Bachelor i Programmering [Spill | Applikasjoner]
Veileder: Hjelsvold, Rune
Mai 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU
Kunnskap for en bedre verden

Abstract

Decision support is a web application developed for weather analysis. Using machine learning, the application will help a shift leader to analyze the weather and give recommendations for road maintenance such as plowing or salting. It is designed as a side application for **CarAdmin** owned by Electric Time Car (ETC).

The web application uses a dashboard and a map to provide an overview of the areas. In an area are weather data, recommendations and roads that the shift leader can rely on. This will help the shift leader to make a decision on when and where a mission should be made.

In the thesis we have used two different machine learning methods. The first divides and defines areas by zones with similar weather forecasts. The other provides recommendations for road maintenance based on weather data and previous decisions.

Sammendrag

Beslutningsstøtte er en webapplikasjon utviklet for væranalyse. Ved hjelp av maskinlæring skal applikasjonen hjelpe en vaktleder med å analysere været og gi anbefalinger for veivedlikehold som brøyting eller salting. Den er lagd som en side-applikasjon for **CarAdmin** som eies av Electric Time Car(ETC).

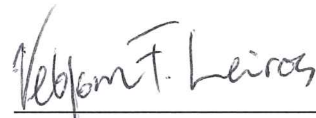
Webapplikasjonen bruker et dashbord og et kart for å gi en oversikt over områdene. I et område ligger værdata, anbefalinger og veier som vaktleder kan støtte seg på. Dette vil bidra til at vaktleder kan gjøre en beslutning på når og hvor en utkalling skal gjøres.

I oppgaven har vi benyttet oss av to forskjellige maskinlærings-metoder. Den første deler opp og definerer områder etter soner med like værprognoser. Den andre gir anbefalinger til vedlikehold ut fra værdata og tidligere beslutninger.

Forord

Vi ønsker å takke vår veileder prof. Rune Hjelsvold for hans oppfølging og hjelp gjennom hele prosjektet. En takk går også til prof. Ivar Farup som bidro til igangsetting av områdedefineringen.

Vi vil også takke Dag Solhaug og Øyvind Flatval fra Electric Time Car(ETC) for deres tilbakemeldinger og bidrag iløpet av prosjektet, og for å ha lagt ut denne bacheloroppgaven som gav oss muligheten til å sette oss inn i maskinlæring.



Emil Johannes Tillman Hegdal | Hans Kristian Hoel | Vebjørn Fonstad Leiros

Innhold

Abstract . . . . **iii**

Sammendrag . . . . **v**

Innhold . . . . **vii**

Figurer . . . . **xi**

Code Listings . . . . **xiii**

Terminologi . . . . **xv**

1 Introduksjon **1**

1.1 Introduksjon 1

1.2 Prosjekt- og effektmål 1

1.2.1 Prosjektmål 1

1.2.2 Effektmål 2

1.3 Avgrensning 2

1.3.1 Endring av kravspesifikasjon 3

1.4 Oppgaven 3

1.5 Målgruppe 3

1.6 Bakgrunn og Kompetanse 4

1.7 Øvrige Roller 4

1.8 Oppbygning 4

2 Kravspesifikasjon **7**

2.1 Use Cases 8
2.1.1 Use Case 1 - Vaktleder sjekker statistikk 8
2.2 User Stories 9
3 Teknisk Design **11**
3.1 Arkitektur 11
3.2 Komponenter innen systemet 12
3.2.1 RequestQueue 12
3.2.2 Innhenting av værdata 14
3.2.3 Maskinlæring 14
3.2.4 Områdedefinisjon 16
3.2.5 Analyseprosess 16
3.2.6 Spring 16
3.2.7 Databasen 17
3.3 Vær-API 19
3.3.1 Vue 20
4 Grafisk Design **21**
4.1 Prosess 21
5 Utviklingsprosess **25**
5.1 Ansvar og roller 25
5.2 Timeføring 26
5.2.1 Arbeidsfordeling 27
5.2.2 Kommunikasjon og arbeidsform 28
5.3 Arbeidsmetode 28
5.3.1 Tidsplan 28
5.4 Teknologi 31

5.4.1 GitLab . . . 31

6 Implemententering . . . 35

6.1 Maskinlæring . . . 35

6.1.1 Grupperingsalgoritmen . . . 35

6.1.2 Klassifisering . . . 42

6.2 Backend . . . 44

6.2.1 API . . . 44

6.2.2 Database . . . 46

6.2.3 Kø-System for Met-grensesnittet . . . 47

6.2.4 Innhenting av Værdata for en kommunegrense . . . 49

6.3 Frontend . . . 50

6.3.1 API Forespørsler . . . 50

6.3.2 Tabeller . . . 50

6.3.3 Beslutningskart/Dashbord . . . 51

7 Kvalitetssikring . . . 53

7.1 Kode validering . . . 53

7.1.1 SonarLint . . . 53

7.2 Versjonskontroll . . . 54

7.3 Gitlab Automatisert Pipeline . . . 54

8 Drøfting av Prosjektresultat . . . 55

8.1 Resultater . . . 55

8.1.1 Områdedefinering . . . 56

8.1.2 Klassifisering . . . 57

8.1.3 Frontend . . . 57

8.2 DevOps verktøy . . . 58

8.3 Frost API 58
8.4 Testing 59
8.5 Arkitektur og Rammeverk 59
8.6 Prosessen 59
8.7 Tid 60

9 Videreutvikling 61

9.1 Klassifiserer 61
9.1.1 Datapunkter 62
9.1.2 Hoeffding 62
9.2 Områdedefinering 63

10 Arbeidsevaluering 65

10.1 Hjemmekontor sammenliknet med arbeidsplass 65
10.2 Arbeidstimer 66

11 Konklusjon 69

Bibliografi 71

A Kontrakt 73

B Logger og referater 77

C Prosjektplan 85

D Oppdrag 103

Figurer

2.1 Use-Case diagram 8

3.1 Systemets overordnede arkitektur 11
3.2 RequestQueue sekvens diagram 13
3.3 Livsløp til Executor-tjenesten 14
3.4 Utprinten av treningsdata instanser 15
3.5 Enkelt eksempel på et beslutningstre 15
3.6 Prosess for område definering 16
3.7 Diagram over databasen 18

4.1 Rodekartet til CarAdmin 21
4.2 Første utkast av prototype 22
4.3 Siste utkast av prototype 23
4.4 Ferdig produkt 23

5.1 Timeplan diagram 26
5.2 Timeføring system 27
5.3 Gantt diagram 29
5.4 Produktkø 33

6.1 Tidsbruk av hver implementering 37

6.2 R-Tre kart over værdata fra Gjøvik Kommune ved siden av enkel tegning av R-Treets oppbygning 39

6.3 Statistikk av beslutningstre 43

6.4 Database struktur 46

7.1 Pipeline test 54

8.1 Resultat av FSDBSCAN, hver gruppering vises med sin egen farge . 56

10.1 Diagram over timefordeling 67

Code Listings

6.1 DBSCAN Algorithm[1] 36
6.2 Nabosøk i STDBSCAN 37
6.3 Fordeling av partisjoner 38
6.4 ExpandCluster[1] 40
6.5 regionQuery 40
6.6 rep_seeds_select[1] 41
6.7 Bygger url for API kall 44
6.8 ApiFrontController 45
6.9 Medlemsfunksjon fra Invoker som returnerer Invoker instans 47
6.10 Medlemsfunksjon fra Invoker som tar imot et kall 47
6.11 Medlemsfunksjon i Executor som implementerer vent dersom det trengs mellom kall 48
6.12 Medlemsfunksjon fra Executor som kjører selve kallet 48
6.13 Algoritme som deler et område inn i punkter 49
6.14 Sender med to knapper i kolonne med navn call og edit 51
6.15 Kalkulerings-funksjon for filtrering av roder 51

Terminologi

ETC	Electric Time Car, oppdragsgiver og produkteier
Koordinater	Med koordinater mener vi EUREF89[1] kartformat.
Veivedlikehold	Brøyting eller salting av veier
Rode	Vei i sammenheng med vedlikehold
Beslutning	Vaktleder sitt valg av veivedlikehold.
Anbefaling	Systemets anbefaling av veivedlikehold
Klassifiserer	Maskinlærings-delen som tar seg av å gi anbefalinger til utkalling.
Beslutningstre	Produsert av klassifiseringen basert på treningsdata for å gi anbefaling.
CarAdmin	Applikasjon/verktøy for styring av kjøretøys-flåter

Kapittel 1

Introduksjon

1.1 Introduksjon

I løpet av en norsk vintersesong varierer været mye. Dette, kombinert med endringer i temperatur-, vind- og solforhold, påvirker i stor grad kjøreforhold på norske veier. For å kunne holde veiene åpne og minimere trafikkulykker, er det avgjørende at veiene blir vedlikeholdt basert på værforholdene. Ved snøfall er brøyting nødvendig vedlikehold og ved underkjølt regn er det viktig at det saltes i god tid for å unngå at veiene blir islagte. Det vil være en vaktleder som har ansvaret for utsendingen av disse utkallingene.

1.2 Prosjekt- og effektmål

1.2.1 Prosjektmål

Oppgaven går ut på å lage en applikasjon som skal gi vaktlederen en anbefaling av nødvendig vedlikehold av veier basert på tilgjengelig værdata. Anbefalingene gitt i applikasjonen vil opplyse vaktleder om brøyting eller salting er nødvendig. Dette skal bli presentert i et beslutningsdashbord. Beslutningsdashbordet skal også presentere alle innsamlede data for områdene slik at vaktleder får det presentert på en oversiktlig måte. Den viktigste egenskapen applikasjonen må ha, er ifølge oppdragsgiveren å gi en anbefaling til vaktlederen når det er nødvendig å brøyte eller salte veiene.

Anbefalinger skal ha mulighet til å utvikle seg over tid og skal basere seg på tidligere utkallinger gjort av vaktleder.

I tillegg til applikasjonens hovedfunksjon (anbefaling til vaktleder), skal den også inneholde statistikk over tidligere vær, fremtidig vær og føreforhold. Statistikken presenteres på et eget dashbord i applikasjonen. Et kart som til enhver tid viser nåværende vær og føreforhold er også aktuelt. En funksjon som gir vaktlederen en enkel måte å kalle ut nødvendig kjøretøy vil være en viktig del av applikasjonen.

1.2.2 Effektmål

Med denne applikasjonen kan ETC utvide CarAdmin sine funksjonaliteter og åpner opp for å kunne utvide til flere kommuner.

En utkalling av kjøretøy baseres hovedsakelig på værvarsler og utplasserte kameraer. Jo større vaktlederens ansvarsområde er, jo vanskeligere er det å ha oversikt over hvilke veier som trenger vedlikehold. Ved hjelp av beslutningsstøtteverktøyet skal vaktlederen kunne øke produktiviteten og effektiviteten i arbeidet sitt. Beslutningsstøtte-applikasjonen samler inn værdata og gir en anbefaling på områder som trenger vedlikehold. Vaktlederen kan bruke applikasjonen for å se værmeldinger, utplasserte kameraer og tidligere rapporter fra forskjellige områder. Vaktlederen kan deretter bruke denne informasjonen til å gjøre en beslutning og kalle ut nødvendige kjøretøy mer effektivt.

1.3 Avgrensning

Et beslutningsstøtte-verktøy kan implementeres vha. maskinlærings algoritmer der man sammenlikner med tidligere erfaringer for å sikre gode beslutninger. Ved bruk av vår applikasjon kan en vaktleder få oversikt over vær- og føreforhold i en hel kommune. Vaktlederen kan derretter sammenlikne anbefalingene fra maskinlæringen med egne erfaringer og bruke dette til å ta en beslutning om utkalling.

For å kunne fullføre prosjektet i tide har det vært nødvendig å sette avgrensninger på prosjektet: Underveis i prosjektet ble vi enige med ETC i at vi ikke skulle lage publikumskart, utkalling eller avvikslogg siden de var allerede eksisterende funksjoner i Caradmin. Med disse avgrensningene åpnet det opp for å utvikle en smart løsning for områdedefineringen, som deler opp kommunen i områder utifra værdata målt over tid.

1.3.1 Endring av kravspesifikasjon

Den opprinnelige planen var at vi skulle få hjelp av en med doktorgrad med kompetanse innen analyse og maskinlæring til å strukturere, planlegge og veilede oss under utvikling av løsningen. Etter møter med ETC fikk vi vite at denne kompetansen ikke var tilgjengelig likevel, og dermed var det vårt ansvar å finne ut av hvordan analysen skulle gjennomføres.

Vi foreslo for ETC å slå sammen beslutningskartet og dashbordet til ett komponent for å spare plass og presentere den viktigste data på et sted. Dette ble vi da enige om å gjøre.

1.4 Oppgaven

ETC lagde denne bacheloroppgaven i tilknytning til konkurransen «Smart vinterveg»[1]. Konkurransen gikk ut på å levere det beste tilbudet av en smart og effektiv løsning for vedlikehold av vintervei.

Oppgaven går ut på å lage et beslutningsdashbord, der relevant data skal presenteres for vaktleder. Data som presenteres skal inneholde værprognoser, områder, sanntidsdata og kamera. Anbefalingene som blir gitt skal ta for seg værprognoser og tidligere erfaringer. Ut fra dette skal systemet gi en vaktleder støtte i sine oppgaver som går på utsending av oppdrag. Oppdragene dashbordet skal anbefale er enten salting eller brøyting av roder.

1.5 Målgruppe

Denne rapporten er rettet mot personer med kompetanse innen programmering og som ønsker et dykk inn i maskinlæring rettet mot klassifisering og gruppering av værdata. Oppgaven berører emner som web-teknologier, objekt orientert design og maskinlæring. Samtidig går vi innom brukergrensesnitt for å presentere anbefalinger og hvordan en kan hjelpe brukere å gjennomføre beslutninger.

[1] https://www.smartemjosbyer.no/smart-vinterveg/

1.6 Bakgrunn og Kompetanse

Vi går alle Programmering retning Applikasjoner[2] på NTNU Gjøvik, og to av oss har hatt IT1 og IT2 på videregående. Vi har gode kunnskaper innen emner som objekt orientert design, brukeropplevelses-design, www-teknologi og programmering.

Vi har særskilt kompetanse som egner seg godt til oppgaven. Vi har gjennom NTNU-relaterte prosjekter, utviklet web-applikasjoner som håndterer både brukergrensesnitt, databasebehandling og RESTful design. Disse prosjektene har også utviklet våre ferdigheter innen programmering, spesielt rettet mot objekt orientert design.

Innen emnet web-utvikling har vi erfaring med Php, Polymer, Node.js og liknende. Disse verktøyene er velutprøvde og populære valg i markedet. Disse har lært oss mye om gammeldagse og moderne utviklingsmetoder innen web-utvikling. Spesielt nyttig er den kompetansen vi har med komponent-basert utvikling mot brukergrensesnittet.

I faget Objekt Orientert Programmering[3] har vi lært om utviklingsmønstre som er praktiske i henhold til å løse problemer på en god og systematisk måte.

1.7 Øvrige Roller

Oppdragsgiveren for denne oppgaven er selskapet *Electric Time Car* der Dag Solhaug er daglig leder, og Øyvind Flatval er utvikler og medeier. De har begge vært kontaktpersoner for oss iløpet av prosjektet.

Vår veileder gjennom prosjektet har vært prof. Rune Hjelsvold.

1.8 Oppbygning

I rapporten kommer vi først til å gå inn i hvilke krav vi fikk fra ETC, så over i arkitekturen til systemet og de teknologiene som er blitt tatt i bruk. Etter det, går vi over prosessen vi gikk igjennom for å lage det grafiske brukergrensesnittet ut ifra Caradmin sin side. Videre snakker vi om rollefordeling i gruppen og hvordan vi jobbet. Så går vi i dybden på implementeringen av maskinlæring og forklarer i detalj hvordan de forskjellige delene av applikasjonen funker. Hvilke tiltak

[2]https://www.ntnu.no/studier/bprog
[3]Fagkode: IMT1082

vi gjorde for å forsikre kjørende og god kode, så drøfting av hva vi har lært og hvordan vi har blitt bedre. Så kommer videre arbeid for ETC slik at de kan sette seg inn og utvikle systemet videre. Til slutt avslutter vi med gruppeevaluering og konklusjon.

Kapittel 2

Kravspesifikasjon

Kravspesifikasjonen vi fikk av ETC:

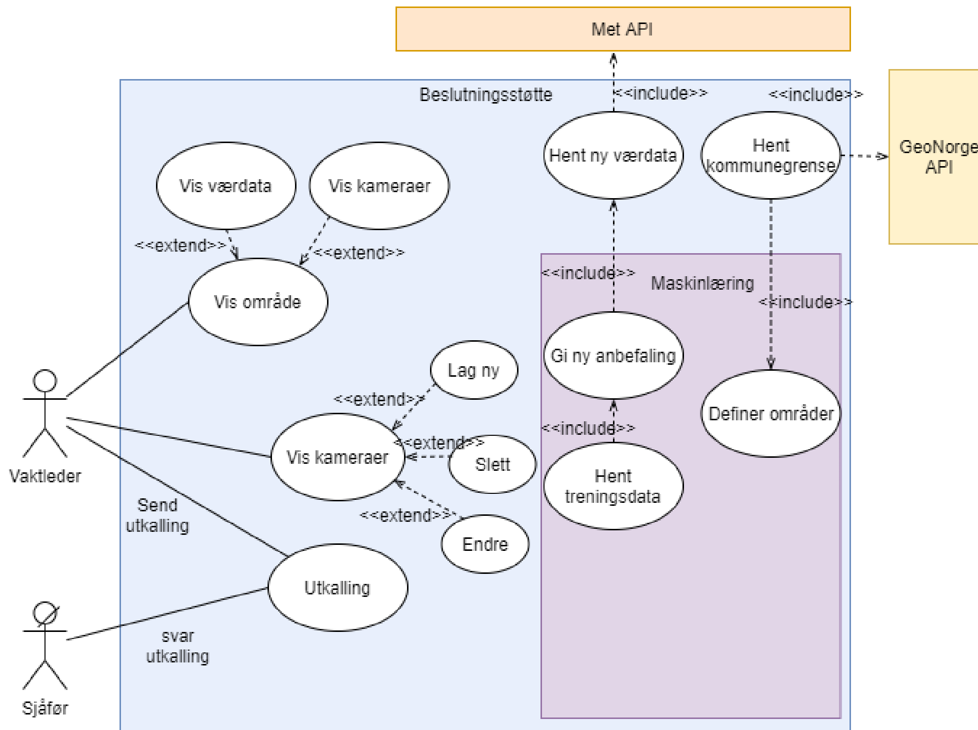
Systemanbefalinger for at applikasjonen skal kunne gi relevante anbefalinger til oppstart av veivedlikehold innebærer å samle og analysere informasjon fra ulike datakilder. De viktige datakildene er:

- Værprognoser for ulike områder
- Lokale værobservasjoner fra sensorer i ulike høydedrag
- Vintervedlikeholdstatus på veiene
- Avtalt tjenestekvalitet på roder
- Tilgjengelige veivedlikeholdsressurser
- Erfaringsdata (– lignede situasjoner)
- Vaktledernes eller sjåførrenes erfaringer
- Varsler
- Innmeldte observasjoner om vær- og føre-forhold fra privatpersoner
- Lokale kontrollobservasjoner, (f.eks friksjonsmålinger)
- Veivarsler fra vegvesenet

I tillegg til Beslutningskartet vil støtten for igangsetting av oppdrag ha sider for:

- Beslutningsdashbord
- Utkalling til oppdrag
- Tilbakemeldingslogg for utkallingene
- Rodeoversikt og planverktøy – ref besvarelse av krav 2.1.2
- Avviksoversikt – oversikt over alle innkomne vei og føre varsler og tilbakemeldinger fra systemets ulike bruker, inkludert publikum. Varseloversikten vil også ha mulighet for behandling av varslene.

2.1 Use Cases



Figur 2.1: Use-Case diagram

2.1.1 Use Case 1 - Vaktleder sjekker statistikk

For at en vaktleder skal kunne sjekke værstatistikken må han åpne dashbordet. Her ser han et kart som viser de forskjellige områdene. Vaktleder kan herfra klikke seg inn på et av områden på sidemenyen til kamera. Når han er inne på et område vil bare dette området bli vist i kartet. I sidemenyen vil vaktleder kunne se værdata og kameraer knyttet til områder.

Use Case 2 - Kamera Registrering

I applikasjonen skal vaktlederen ha mulighet til å registrere, slette og lagre kameraer i ulike områder. I applikasjonen kan vaktlederen klikke på vis kameraer. Her kan han se alle kameraer i systemet. Vaktlederen kan velge mellom å legge til nytt, slette eller endre på et kamera. For å registrere et kamera må vaktlederen legge inn et område, navnet på kamera og linken til kamerakoblingen.

2.2 User Stories

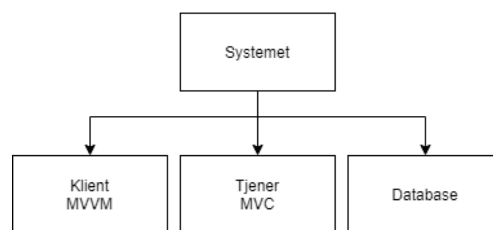
Jon B. er en vaktleder som ønsker å finne all statistikk og værdata oversiktlig på ett sted, slik at han lett kan gjøre en utkalling. Han kommer til å måtte gjøre mange utkallinger på en dag, så han setter pris på anbefalingene som blir presentert og får en ekstra godkjenning på beslutningen sin.

Kapittel 3

Teknisk Design

I dette kapittelet beskriver vi det tekniske designet for prosjektet. Vi forteller om arkitekturen vi brukte i applikasjonen, de forskjellige teknologiene vi brukte i prosjektet og løsninger vi har utviklet som kø håndteringen Request Queue.

3.1 Arkitektur



Figur 3.1: Systemets overordnede arkitektur

Systemet er delt opp i en klient tjener struktur, der tjeneren bygges opp etter en MVC[1] struktur og klienten etter en MVVM[2]. Grunnlaget bak er å sette opp en enkel struktur som vi er vant med, samtidig ha en enighet i hvor deler av systemet skal plasseres.

Tjeneren er satt opp i MVC strukturen. Ettersom backend er bygget for å servere JSON objekter er det relativt lite jobb å formatere dette. Dermed vil View-klassene

[1] Model-View-Controller
[2] Model-View-Viewmodel

være ganske små og representere en liten del av backenden. Det meste av koden ligger i Model og Controller der vi skal prosessere data og håndtere kall.

Klienten er satt opp i en MVVM struktur. Data hentes fra Model delen av arkitekturen, som i vårt system er backend. View er både brukergrensesnittet til applikasjonen og handlinger brukeren gjør. Vue står for Viewmodel delen av arkitekturen, og manipulerer dataen fra Model slik at den passer til View delene.

3.2 Komponenter innen systemet

3.2.1 RequestQueue

En del av systemet som står for innhentingen av data er avhengig av å følge retningslinjene til Meteorologisk Institutt[3] sitt API. Dermed er vi nødt til å definere et grensesnitt som håndterer alle kallene og sørger for at systemet ikke overskrider grensene.

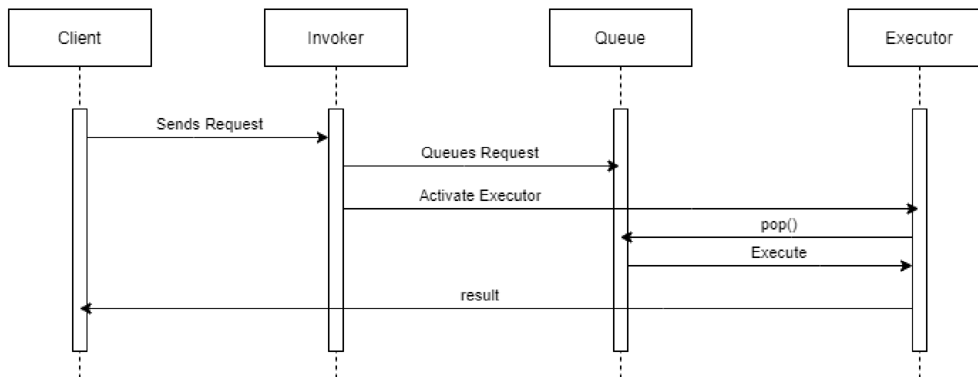
Hovedtankene bak grensesnittet vi designet var at det skulle være tilgjengelig globalt og håndtere opp til flere kall samtidig. Med dette i tankene hentet vi inspirasjon fra objekt orientert design mønstre og valgte å bruke Singleton- og Kommando-mønsteret[2].

Hvert kall er innkapslet i et objekt som implementerer Java Callable-grensesnittet[4]. Dette gjør at vi kan definere kall til tjenester som Meteorologisk institutt og utføre kallene når systemet har ledige ressurser.

Invoker-klassen er implementert i henhold til Singleton-mønsteret med en liten modifikasjon. Modifikasjonen gjør den synkron med flere tråder dersom den får flere kall. Invoker-klassen er ansvarlig for innlegg i køen og for at en tjeneste for kjøring av kall iverksettes dersom det blir lagt inn nye elementer.

For å sørge for at antall kall per sekund ikke overstrider det som er definert i retningslinjene har vi valgt at «kall-tjenesten» automatisk skal implementere venting mellom hvert kall. Grensesnittet til Meteorologisk Institutt tillater opp til 20 kall per sekund og har som krav at dersom det kommer mange kall skal de spres jevnt utover et tidsrom. Vi har satt «ventingen» mellom kallene godt under 20 kall per sekund og med dette lar vi også systemet spre kallene over tid. Ved å følge denne retningslinjen forsikrer vi oss om at systemet ikke blir utestengt fra grensesnittet.

[3]https://api.met.no/doc/TermsOfService
[4]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html



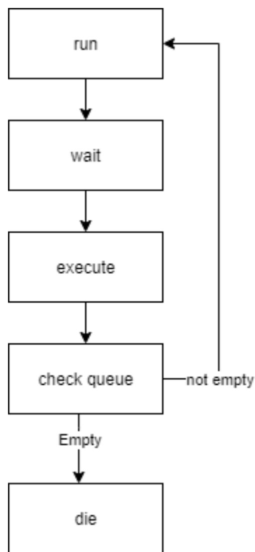
Figur 3.2: RequestQueue sekvens diagram

Vi definerte en «Interface Request». Dette tillot oss å implementere et kall og deretter plassere det i en kø. Fra Java's samtidighets-bibliotek brukte vi klassene FutureTask for at en klients kall kunne hentes etter at den var plassert i køen. Grensesnittet for køen ville lagre kallet i køen, samtidig som den startet en ny tråd som går gjennom hvert kall og kjører det. Når kallet var ferdig, kunne vi hente ut svaret ved hjelp av get-funksjoner i FutureTask.

Hvert kall blir innkapslet i Callable-grensesnittet til Java. Dette er en fordel ved bruk av Java's samtidighets-bibliotek. Når et kall sendes til Invoker-klassen, innkapsles det igjen i et FutureTask-objekt[5] som tillater å hente resultatet når det er klart. Dermed blir prosessen med å gjøre ett eller flere kall ganske enkelt. Vi kan sende flere kall, utføre andre prosesser i systemet og så hente svaret når det trengs.

Med dette grensesnittet unngår vi mye opptatt-venting ved at kall-tjenesten skrur seg selv av når køen er tom, og skrus på igjen når det kommer nye kall. I tillegg lar vi tjenesten kjøre andre prosesser mens den venter på svar.

[5] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html



Figur 3.3: Livsløp til Executor-tjenesten

Vi designer tjenesten slik at når den startes går den gjennom hvert steg i figur 3.3. Den skal vente om nødvendig før et kall utføres, og når et kall er gjennomført sjekkes køen før den eventuelt fortsetter.

3.2.2 Innhenting av værdata

3.2.3 Maskinlæring

For vårt program brukte vi klassifisering- og grupperingsalgoritmer. Grupperingsalgoritmene brukes for å dele inn en kommune i områder basert på værdata som er målt, mens klassifiseringen gir anbefaling for vedlikehold av disse områdene. Det betyr at alle roder innenfor et område vil få anbefalingen.

Klassifisering

Klassifisering innen maskinlæring prøver å finne sammenhenger mellom sett av treningsdata som har et gitt utfall. Basert på denne sammenhengen kan den gi en anbefaling på ny data, som ikke har et utfall. I vårt tilfelle er data værstatistikk, og utfallet er om det skal sendes ut til brøyting, salting eller ingenting. Vi brukte biblioteket Weka som inneholder algoritmer for data-analyse og predikativ modellering. For å gi en bedre oversikt over hvordan klassifiseringen fungerer, forklarer vi hvordan alle variablene henger sammen og hva vi mener med forskjellige begreper:

Beslutning blir gjort av en vaktleder når det blir sendt ut til veivedlikehold, dette blir lagret som treningsdata.

Anbefaling blir gjort av klassifisereren på ny værdata basert på treningsdataene.

Attributt definerer navn og verditype til et datapunkt i treningsdataene. Dette gjør at klassifisereren kan skille mellom forskjellig data. Vi bruker numeriske verdier for værdata bortsett fra beslutningen/utfallet som er en enum. Et eksempel på et attributt er mengde nedbør med numerisk verdi.

```
@attribute air_temperature numeric
@attribute relative_humidity numeric
@attribute dew_point numeric
@attribute precipitation_amount numeric
@attribute precipitation_max numeric
@attribute precipitation_min numeric
@attribute precipitation_probability numeric
@attribute air_pressure numeric
@attribute coordinates_high numeric
@attribute beslutning {BROYT,SALT,NONE}

@data
8.5,35.1,-4.7,0,0,0,0,1014.1,139,BROYT
8.5,35.1,-4.7,0,0,0,0,1014.1,139,BROYT
3.9,45.6,-8.2,0,0,0,0,1021.7,539,SALT
3.9,45.6,-8.2,0,0,0,0,1021.7,539,SALT
3.1,50.3,-6.5,0,0,0,0.3,1021.3,471,NONE
```

Figur 3.4: Utprinten av treningsdata instanser

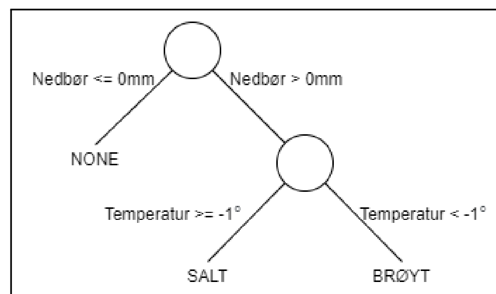
Instans er et sett med datapunkter for en værmåling som er separert med komma og samsvarer med attributtene som er gitt. Du ser eksempler på linjer med instanser under `@data` i figur 3.4.

Treningsdata er en liste instanser fra tidligere korrekte utsendelser. Dette blir lagret i databasen vår og vaktleder kan endre og slette elementer i den for å sikre at treningsdataene er riktig.

J48 er beslutningstre-algoritmen innen klassifisering som vi benytter oss av. Det er denne som bygger et beslutningstre ut fra treningsdata. J48 forventer at alle attributtene er numeriske utenom utfallet. Vi så på J48- og Hoeffding-beslutningstre algoritmene før vi kom fram til å bruke J48 på grunn av Weka sin gode dokumentasjon av denne algoritmen[3]. Dette var viktig for oss siden vi ikke har tidligere erfaring med maskinlæring og må støtte oss på selvlæring. Hoeffding algoritmen er god på store mengder data strømming og kan være verdt å bytte over til etterhvert som programmet får mer data.

Beslutningstre er resultatet man får av en beslutningstre-algoritme. Den vil bli satt opp til å passe best mulig til treningsdataene. Den bygger grener ut fra relevante attributter som til slutt ender i et utfall[4] . Nye anbefalinger blir gjort ved å sende værdata igjennom dette treet.

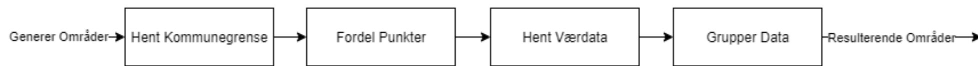
Klassifisereren blir kalt på hver gang det hentes inn nye værmeldinger fra et punkt. Når værdata hentes ut på et punkt hentes det værmeldinger for de neste 24 timene. Når dette skjer, går klassifisereren gjennom hver værmelding og gir den sin egen anbefaling. Vi hentet først ut for 48 timer men valgte å kutte ned til 24 timer siden målingene ble upresise og ikke oppdatert hver time. Dette ville gitt upresise treningsdata som ville ført til dårligere anbefalinger. Når applikasjonen skal gi en ny



Figur 3.5: Enkelt eksempel på et beslutningstre

anbefaling, sjekkes de nye værdataene mot beslutningstreet og lagrer anbefalingen i databasen.

3.2.4 Områdedefinisjon



Figur 3.6: Prosess for område definering

For å avgrense områder defineres de innenfor kommunegrenser. Først lages en MBR[6] som er parallell med aksene. Deretter plasseres punkter jevnt utover denne boksen, hvert punkt sjekkes først om den er innenfor kommunegrensen før den plasseres. Ut fra dette får vi et kart bestående av punkter innenfor en kommune. Disse koordinatene kan brukes for innsamling av værdata, som til slutt blir gruppert. Disse gruppene er det som representerer de områdene systemet skaper.

3.2.5 Analyseprosess

Analyseprosessen består av grupperingsalgoritmen og klassifisering. Grupperingsalgoritmen tar et sett med data fra en kommune og deler den opp i områder basert på hvor værdataene er likest over tid. Klassifiseringen tar inn dataene fra et målepunkt og gir en anbefaling til veivedlikehold av et område basert på treningsdata. Ved å kombinere disse to kan applikasjonen gi anbefalinger på forskjellige områder i en kommune.

3.2.6 Spring

Spring Boot er et Java rammerverk lagd for enkelt å bygge og kjøre applikasjoner. Spring Boot vil automatisk konfigurere systemet etter de bibliotekene som blir lagt inn. Den gjør det også lett å lage REST APIer, og får kommunisering av databasetransaksjoner til å gå via Java klasser. Det er spring vi bruker til å kommunisere med databasen.

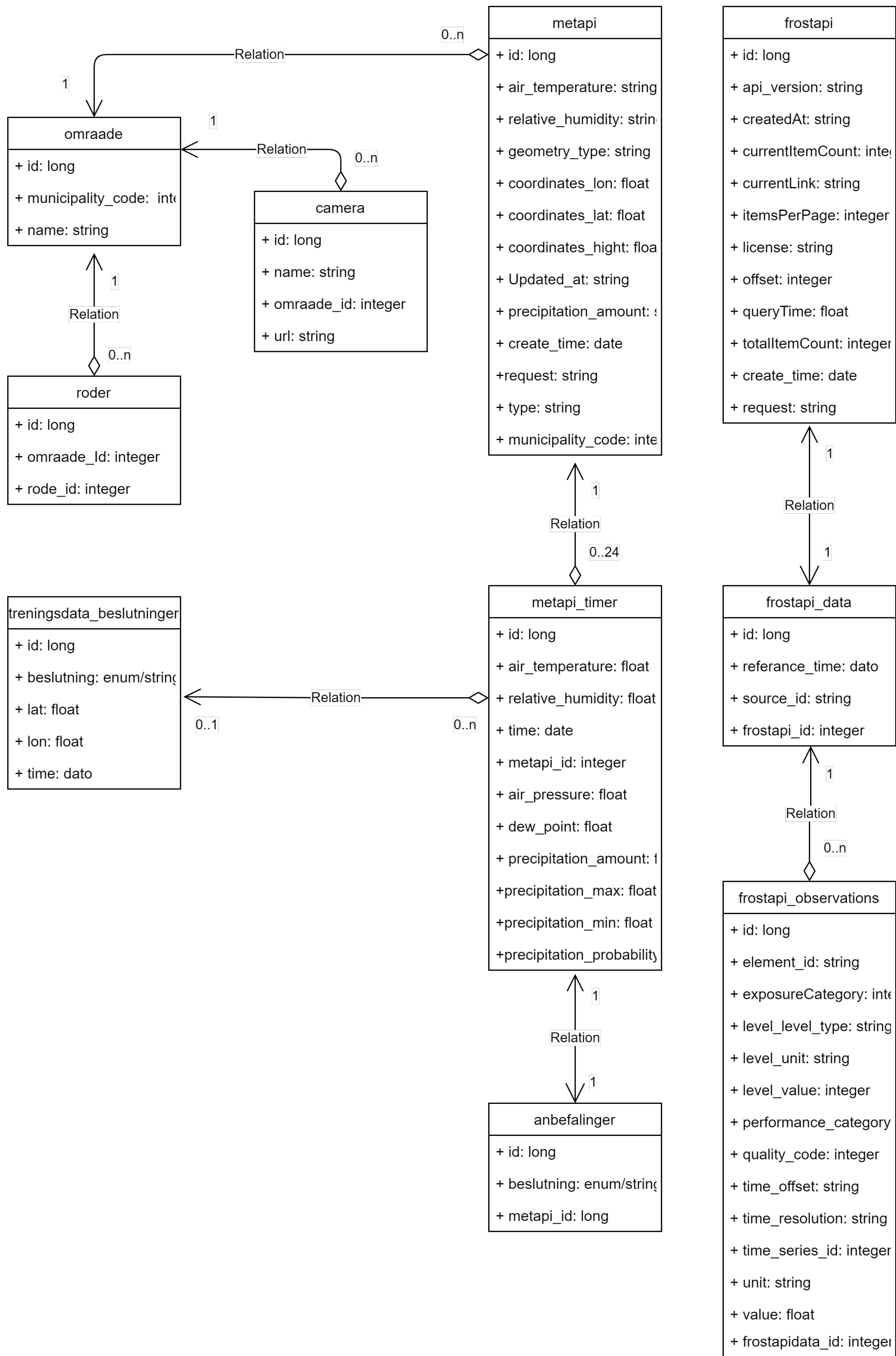
[6] minimum bounding rectangle, eller minst mulig rektangel rundt en geometrisk form

3.2.7 Databasen

Vi bruker en SQL-basert database. ETC ba om at vi brukte MariaDB[7] som database så vi valgte å bruke den. Årsaken til at ETC ba om at vi brukte MariaDB var at de bruker den for CarAdmin, og det ville da bli lettere for dem å videreutvikle vår applikasjon senere.

SQL-spørrespråket brukes til å kommunisere med relasjonsdatabaser. Med SQL kan man samhandle med databasen for å manipulere tabellene med data. Det finnes mange operasjoner for å påvirke data og tabellen. Noen av disse er Create, Drop, Select og Update.

[7]https://mariadb.org/



Figur 3.7: Diagram over databasen

Databasen inneholder tabeller for områder, roder, treningsdata, værdata og anbefalinger.

Område blir generert av grupperingsalgoritmen og har relasjon til kameraer, roder og værdata fra Metapi.

Kamera lagrer en lenke som viser kamera. Dette er rettet til å støtte Statens vegvesen sine veikameraer. Kameraene blir brukt av vaktleder for å få innblikk i situasjonen på veier innen et område.

Metapi inneholder generell informasjon om værmålingen som er blitt gjort som måleenheter brukt, posisjon det er målt fra og når den ble tatt. Den vil ha relasjon til 24 metapi_timer tabeller, som til sammen inneholder værdata for de neste 24 timene.

Metapi_timer inneholder værdata for en gitt time. Den inneholder data som blir presentert for vaktleder og brukt til treningsdata for klassifisereren som skal gi en anbefaling der.

Anbefaling lagrer anbefalinger som blir gjort av klassifisereren. Det blir gjort en anbefaling for hver Metapi_timer når den hentes inn, relasjonen mellom dem er en til en.

Treningsdata_beslutninger lagrer tid, sted og hvilken type beslutning som blir gjort når en utkalling blir sendt. Man kan finne alle værmeldingene som er satt til denne tiden og stedet for å hente treningsdata til klassifisereren.

Frostapi lagrer historisk værdata på samme måte som Metapi tabellen. Ved å legge til tidligere værdata kan man begynne å sette opp treningsdata til klassifisereren tidlig i prosessen.

3.3 Vær-API

For innsamling av værdata trengte vi APIer som leverer værdata. Vi har sett på flere leverandører av værdata, noen av de vi så på var APIene til Netatmo og Meteorologisk institutt.

Netatmo har to typer APIer som leverer værdata. Den ene er en global API som leverer sanntids temperaturer, men gikk ikke spesifikt inn på områder. Netatmo selger forskjellige enheter som kan måle nedbør, temperatur og vind til privatbruk. Disse enhetene kan man kjøpe og sette opp på ønsket sted. Har man disse produktene kan man koble de opp til en API og bruke data derifra. Disse dataene er også bare i sanntid.

Vi benyttet oss av to APIer fra det Meteorologiske instituttet. Den første var Frost API[5], som leverer historisk data, og den andre er Met API[6] som leverer sanntidsdata og værprognoser. Historisk værdata kan brukes som treningsdata for klassifiseringen, og sanntidsdata for været brukes for nye anbefalinger og presenteres til vaktleder. For å bruke denne APIen sendes det med koordinater for ønsket område og får returnert værdata og prognoser for de neste 24 timene for dette området.

Vi valgte å bruke Met og Frost APIene som leverandør av værdataene for applikasjonen.

3.3.1 Vue

Vue.js[8] er et progressivt og monolittisk rammeverk for javascript designet til å bygge brukergrensesnitt. Vue kan tilpasses trinnvis som gjør det mulig for ETC å implementere vue-komponenter inn i deres applikasjon hvis de ønsker. Kjernebibliotekene til Vue er rettet mot designet av nettsiden og kan lett integreres med flere biblioteker eller allerede eksisterende løsninger.

En av fordelene med Vue er at den er reaktivt. Dette vil si at data som presenteres i Vue automatisk oppdateres om den skulle bli endret. Det gjør at vi slipper å oppdatere dataene manuelt i html-elementene, som kommer godt til nytte når flere deler skal endre på samme data.

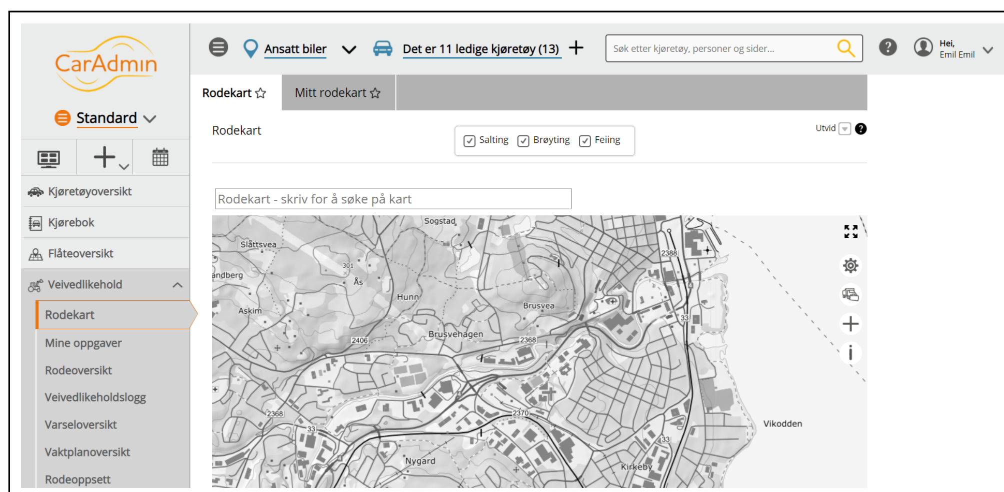
[8] https://vuejs.org/

Kapittel 4

Grafisk Design

4.1 Prosess

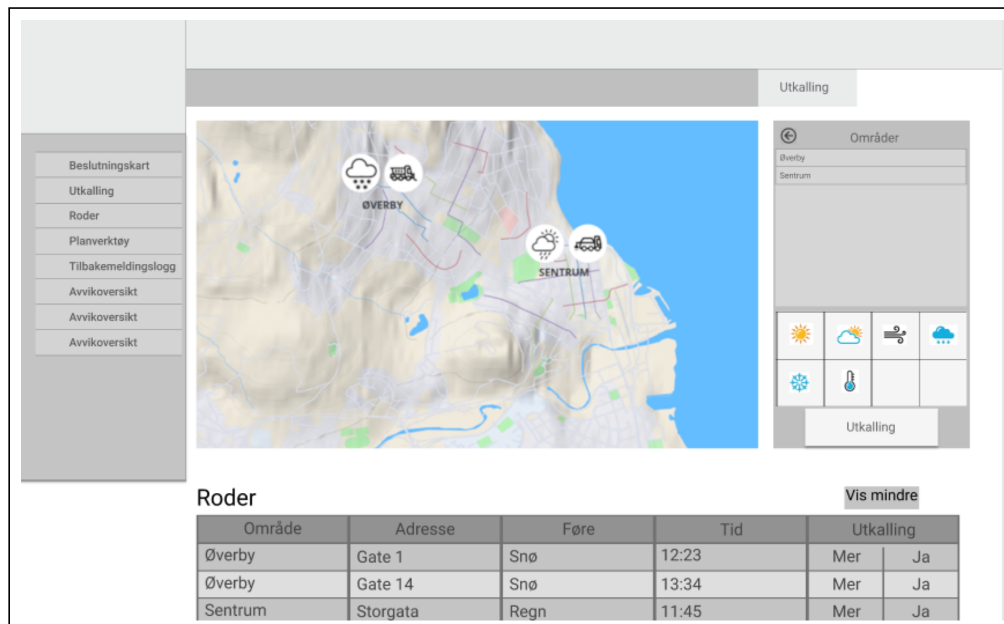
ETC stilte bare ett krav til det grafiske brukergrensesnittet til applikasjonen. Det var at det skulle gi samme brukeropplevelse som deres applikasjon CarAdmin. Vi fikk tilgang til CarAdmin slik at vi kunne teste applikasjonen og bli kjent med brukergrensesnittet. Basert på denne erfaringen laget vi en skisse til hvordan vi tenkte å sette opp brukergrensesnittet til vår applikasjon.



Figur 4.1: Rodekartet til CarAdmin

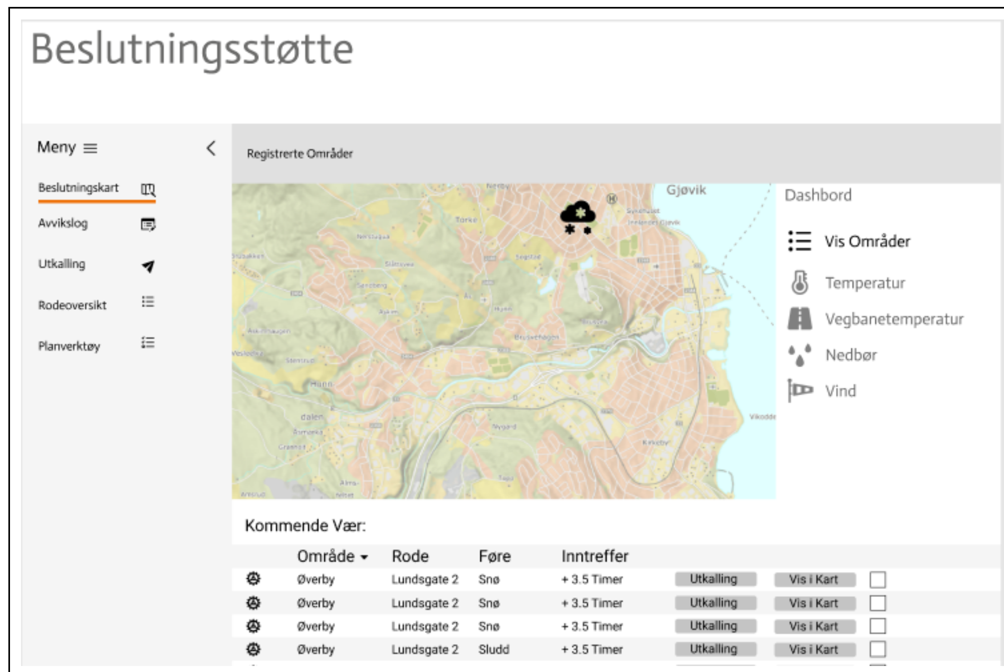
Nettsiden til CarAdmin har hovedfargene hvit, grå og oransje med blå og svart som sekundære farger. Den består av en sidemeny, header, underheader og hoveddel. I sidemenyen kan man navigere til de forskjellige sidene, headeren blir

brukt til søkefelt i applikasjonen, tilgang til profil og oversikt for bil og kjøretøy. Underheaderen gir mulighet til å bytte tabell-innhold og hoveddelen presenterer innholdet som varierer mellom tabeller og kart. I prototypen vår tok vi utgangspunkt i både headeren og sidemenyen som ble brukt i CarAdmin for å sette opp en side som navigeres igjennom på samme måte.

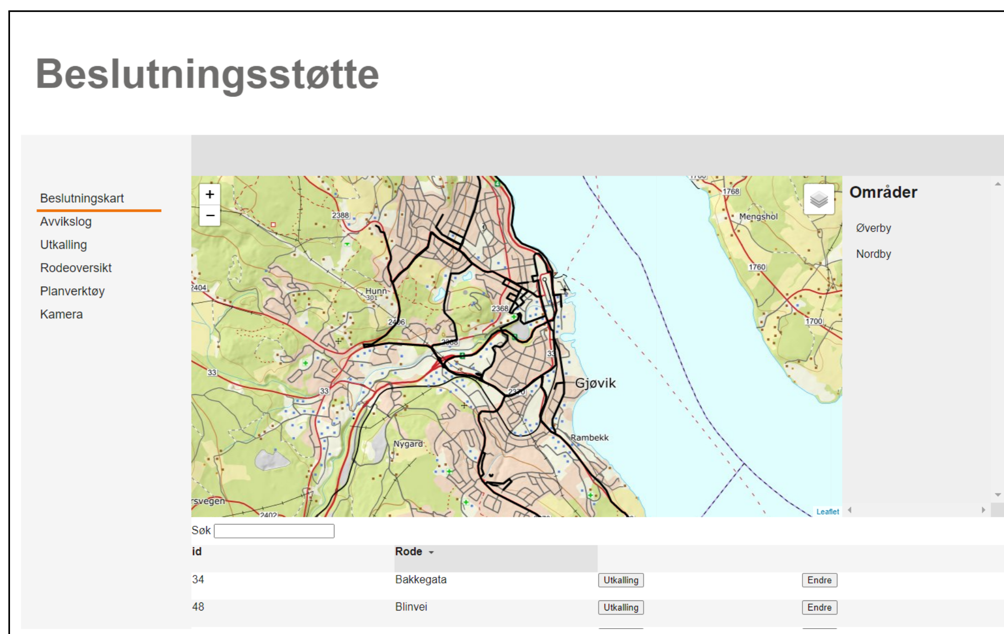


Figur 4.2: Første utkast av prototype

Gjennom flere møter med ETC tilpasset vi prototypen til deres ønsker. De ville ha en bedre indikasjon på når beslutningen bør tas og mer data i tabellene. Når det gjaldt selve utseende og oppbygging av applikasjonen var de fortsatt åpne for utvikling så vi fortsatte å jobbe med det. Vi satte headeren over sidemenyen og la inn logo på applikasjonen. Vi ville ikke legge til ny funksjon her siden CarAdmin allerede har fylt ut sin header. Vi gjorde fargene i applikasjonen lysere og fjernet unødvendige detaljer for å rette fokuset mot de viktige detaljene på siden, nemlig dashbordet.



Figur 4.3: Siste utkast av prototype



Figur 4.4: Ferdig produkt

Kapittel 5

Utviklingsprosess

I utviklingsprosessen vil du kunne lese om valg og deling av ansvarsroller og oppgaver. Samtidig vil du få et innblikk i hvordan vi har jobbet og bruken vår av hjelpemidler iløpet av prosjektet. Vi vil gå igjennom hjelpemidler som kommuniseringsverktøy og planverktøy.

5.1 Ansvar og roller

I gruppearbeid er det viktig å vite hvem som har hvilke ansvar og roller. For oss var det viktig å få på plass rollene tidlig slik at alle visste hvem som hadde ansvar for de ulike oppgavene gjennom hele prosjektet. Vi hadde en diskusjon om hvem som skulle ha de forskjellige rollene. Vi ble enig om at Vebjørn skulle være prosjektleder og teknologiansvarlig, Emil design- og sikkerhetsansvarlig og Hans Kristian oppfølgings- og dokumentasjonsansvarlig.

Som prosjektleder og teknologiansvarlig er Vebjørn sin rolle å ha overordnet ansvar for møter, planer og annet relatert til prosjektet. Han skulle være scrummaster og lede sprint-møtene. Siden Vebjørn styrte alle møtene skal han ha overordnet ansvar for å holde seg oppdatert på teknologien vi skulle bruke i prosjektet.

Emil hadde hovedansvaret for design av brukergrensesnittet og implementering av applikasjonen. I tillegg hadde han ansvar for at applikasjonens sikkerhet samsvarte med oppgavebeskrivelsen. Han måtte gå over koden jevnlig for å sørge for at dette blir fulgt.

Oppfølgings- og dokumentasjonsansvarlig sin oppgave er å passe på at gruppen følger fremdriftsplanen, fører timer for arbeid og har øverste ansvar for dokumentasjonen. Det vil si at Hans Kristian skal dokumentere og skrive referater

fra gruppe-, veilednings-, bedriftsmøter og andre aktuelle hendelser.

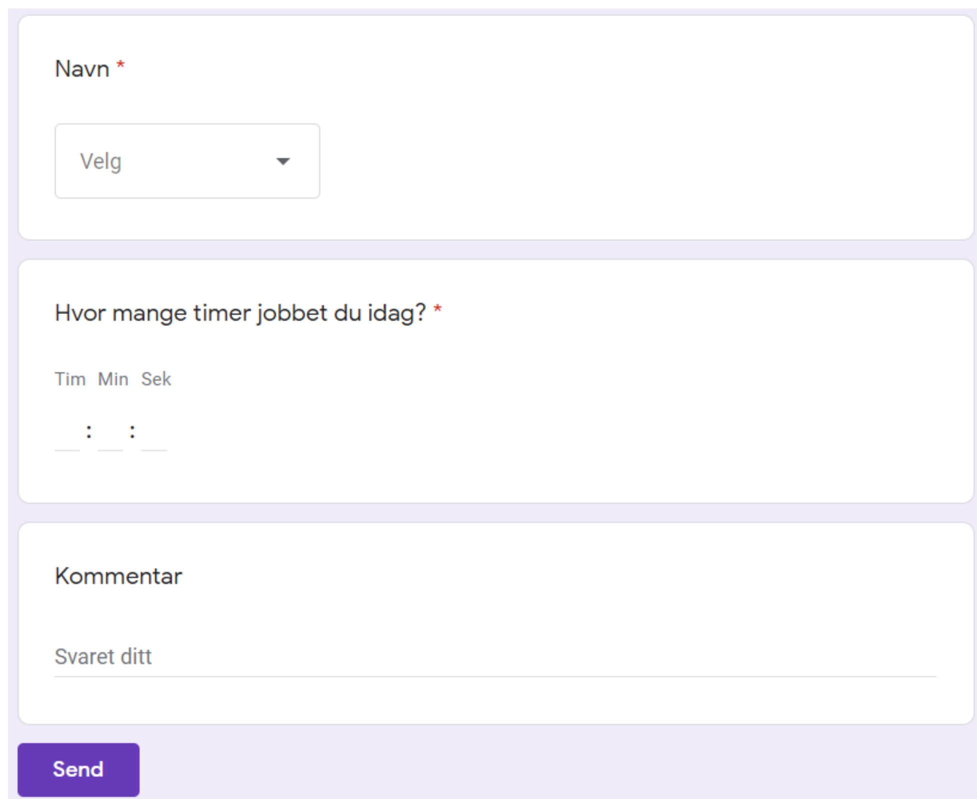
5.2 Timeføring

Under bacheloren er det viktig å føre timene man jobber. For å kunne se hvor mye tid vi har brukt på de forskjellige prosessene og totaltsett. Vi hadde satt oss som mål å jobbe 30 timer i uken hver, etter å høre fra faglærer og veileder om at det var anbefalt arbeidsmengde for en bacheloroppgave.

	Man	Tirs	Ons	Tors	Fre
8					
9	Planlegging	Veiledningsmøte	- Arbeid -		- Arbeid -
10	Rapport til Veiled	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -
11	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -
Pause 30 min	-	-	-	-	-
12	- Arbeid -	- Arbeid -		- Arbeid -	- Arbeid -
13	- Arbeid -	- Arbeid -		- Arbeid -	- Arbeid -
14	Møte ETC		- Arbeid -		- Arbeid -
15	- Arbeid -		- Arbeid -		Oppsummering
16					

Figur 5.1: Timeplan diagram

Vi bestemte oss for å fordele arbeidstimene fra 09:00 til 16:00 mandag til fredag med en 30 minutters lunsjpause. I timeplanen har vi brukt fargekoding for å lettere holde oversikt på når vi er tilgjengelige. Grønn betyr at alle tre i gruppen er tilgjengelige for å arbeide . De timene som er markert med gul, er når Vebjørn har forelesninger og ikke kan jobbe. Til slutt har vi brukt rød for å markere de timene alle har forelesning og ikke kan jobbe med prosjektet. Skulle det komme en dag en av oss ikke får jobbet alle timene, vil personen måtte ta dette igjen på et annet tidspunkt.



Figur 5.2: Timeføring system

For å føre timene har vi laget et Google spørreskjema som er koblet opp til et Google regneark. Vi skulle fylle ut dette skjemaet når vi var ferdig med å jobbe for dagen. I skjemaet fyller man inn navn, antall timer jobbet og eventuelt en kommentar. Informasjonen fra skjemaet blir lagret i regnearket.

Vi kunne godt også brukt andre applikasjoner for å holde styr på timeføring. F.eks finnes det mange mobilapplikasjoner som gjør jobben for deg. Vi valgte bare å gjøre dette fordi det ble lettere å få innsyn i timene som ble ført.

5.2.1 Arbeidsfordeling

Under oppstartfasen ble vi enige om at alle skal være med å finne teknologi og ressurser vi skulle bruke videre i prosjektet. Alle hjalp til med å sette opp prosjektet før vi begynte på hver vår del. Selv om vi hadde ansvar for forskjellige deler, bidro vi på de andre delene.

5.2.2 Kommunikasjon og arbeidsform

Koronapandemien har hatt en stor påvirkning på hvordan vi arbeidet og kommuniserte under bachelor oppgaven. Det har påvirket hvordan vi jobbet som en gruppe og hvordan vi holdt møtene våres med veileder og ETC. Vi har i liten grad møttes fysisk pga. begrensningene pandemien har satt. Vi har måttet jobbe mer individuelt og ble avhengige av gode kommunikasjonsverktøy for et best mulig samarbeid. Vi valgte å bruke kommunikasjonsverktøyene *Microsoft Teams*[1] og *Discord*[2].

Innad i gruppen foretrakk vi å bruke Discord fordi vi er mest vant med dette verktøyet. Med Discord kan vi ha samtaler både med og uten video, vi kan sende meldinger og dele skjerm. Skjermdeling er praktisk når vi trengte å hjelpe hverandre med enkelte oppgaver. Vi brukte Discord innad i gruppen da vi ikke hadde mulighet til å møte opp fysisk, som var hele utviklingsfasen utenom et par unntak.

Microsoft Teams ble brukt til å kommunisere med veileder og ETC. Dette er et verktøy de fleste bruker, og det er lett å sende innkallinger til møter på epost. Teams støtter skjermdeling som var praktisk for å vise applikasjonen vår til ETC.

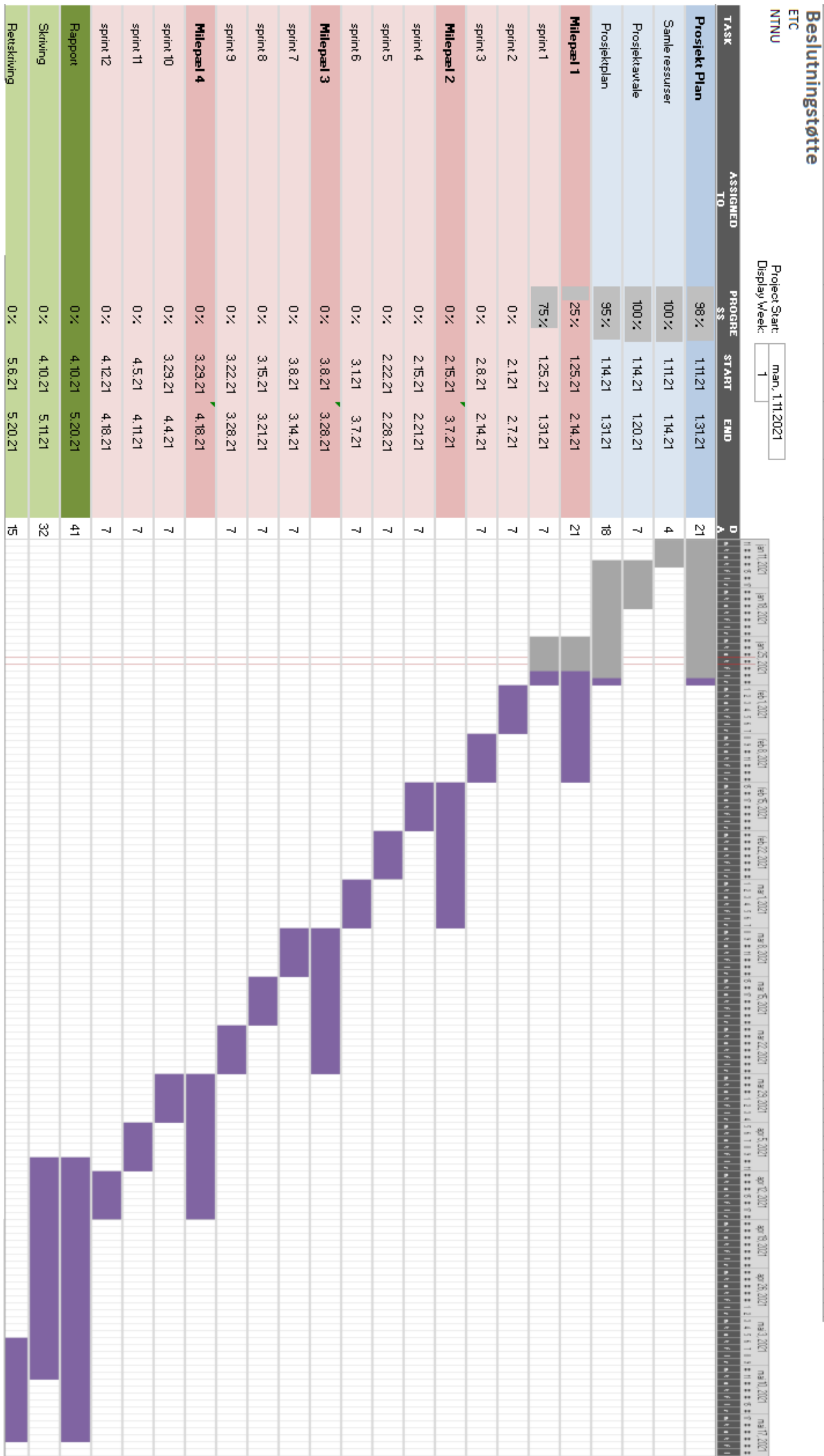
5.3 Arbeidsmetode

5.3.1 Tidsplan

Utviklingsprosessen vår trengte en tidsplan. Vi valgte å bruke et Gantt-diagram for å illustrere planen. Et Gantt diagram viser start- og slutt-datoer for oppgaver over en tidsperiode. Her har vi satt opp seks tidsperioder. Den første var å lage prosjektplan, så hadde vi fire milepæler og til slutt rapportskriving. Tidsperiodene er på tre uker hver og avsluttes med en milepæl, som er et delmål i utviklingsprosessen.

Den første milepælen var 25.januar da vi startet å utvikle selve produktet. Deretter jobbet vi gjennom de fire milepælene. Planen var å være ferdig med milepælene innen 18.april. Det vil si at da skulle vi være ferdige med hele løsningen og ville da ha litt over en måned på å skrive rapporten.

[1] https://www.microsoft.com/nb-no/microsoft-teams/group-chat-software
[2] https://discord.com/



Figur 5.3: Gantt diagram

Her er utdypingen av hva milepælene innebar:

Milepæl 1) 15. Februar

- **Prototype** - Prototypen er laget i Figma. Her skal det være ferdig resultat av en UI og UX model av applikasjonen.
- **Web Applikasjon** - En enkel tjener som leverer dokumenter (html, css og javascript filer) til klienter. Implementerer MySQL
  Fungerende database som kan lese og skrives til med simpel UI på websiden
- **Rate limit løsning** - Lagre den data som er ømfintlig for rate-limitering fra de tredjeparts grensesnittene vi bruker. Data som blir lagret vil bestå av værdata som senere skal fores inn i maskinlærings-algoritmen.
- **Område Definering** - Automatisk definering av områder i en valgt kommune. Skal defineres basert på Grunnkretser der vær-prognoser er enormt like over lang tid. Navngiving av et område skjer automatisk basert på grunnkretsene og skal være redigerbart i tilfellet navnet ikke passer lokale normer.

Milepæl 2) 8. mars

- **Vær-prognoser** - Hente og vise vær-prognoser i form av diagrammer for et valgt område. Dette vil være et datapunkt som kan velges fra dashbordet.
- **Rodeoversikt** - Hente roder inn basert på områdene som er definert fra milepæl 1. Oversikten skal inneholde område, rodenavn, en utkalling-knapp, en beskrivelse av føret når vedlikehold bør utføres, et tidspunkt for når føret inntreffer og en "vis i kartknapp som viser hvor området ligger.
- **Interaktivt Kart** - Kartet skal automatisk gå inn mot et av de registrerte områdene. Et område skal kunne velges slik at man senere kan bruke dette for å vise data basert på området som er klikket på.
- **Utkalling** - Det skal være mulig å gjøre utkallinger. Det vil si at en vaktleder skal få opp forslag om oppdrag som bør gjøres, og da ha muligheten til å tildele dette til en sjåfør.
- **Innlogging** - Hente grensesnitt for innlogging fra CarAdmin for å verifisere vaktledere. UI vil bli oppdatert utifra hvilken bruker som er innlogget.

Milepæl 3) 29. Mars

- **Dashbord** - Vise div statistikk fra områder man trykker på. Historisk data, ikke framtidsdata.
- **Beslutningskart** - Er en videre utviklet versjon av det interaktive kartet. Her skal man kunne se forskjellige områder har blitt merkert med ikoner

for værvarsel. Man kan klikke på ikonet for å få full rapport om været her og da ta en beslutning om å sende en utkalling.

- **Vedlikeholds-prognose** - Gjennom maskinlæring av vekta datapunkter skal vi levere en prognose for hva slags vedlikehold som skal utføres. Prognosene gjøres for hvert område, f.eks Øverby eller Sentrum.
- **Publikumskart** - Vise statuser på roder/områder. Dette kartet vil være tilgjengelig offentlig for alle og vil vise statuser som: Ikke begynt, Oppdrag startet, Oppdrag ferdig.
- **Avvik** - Gjør det mulig å legge inn avviksrapporter for både brukere og publikum, vaktleder kan i tillegg se og redigere innsendte avvik.

Milepæl 4) 18. April

- **Analysemotor** - Analysemotoren skal være ferdig. Med dette vil det si at den samler data fra Met og potensielt andre ressurser og sammenlikner dette med de observasjoner som er gjort og gir en anbefaling til vaktleder.
- **Web Applikasjon** - En fullstendig nettside som presenterer kart og tabeller på en oversiktlig måte og som er lett for en vaktleder å bruke.
- **Dokumentasjon** - Sammen med et ferdig prosjekt skal vi levere dokumentasjon av koden som er skrevet. Denne leveres som en nettside generert av Doxygen med forklaringer rundt de objekter og funksjoner som trenger det. Dokumentasjonen vil også inneholde overordna diagrammer som viser prosessene i systemet: analysemotoren, områdedefinering, rodeprioritering.

5.4 Teknologi

5.4.1 GitLab

Gitlab er en komplett utviklingsplattform, som leverer all funksjonalitet en trenger for å lage et produkt.[3].

Milepæler

Gitlab har en egen funksjon som heter milepæl. Her kan vi legge inn alle milepælene vi har. Til hver milepæl kan vi legge til oppgaver og vekte dem. Hvordan en oppgave er vektet påvirker hvor stor andel av en milepæl den tilsvarer. Milepælene vises som fra 0 - 100% fullført. Etter hvert som man fullfører en oppgave blir

[3]https://about.gitlab.com/

prosenten oppdatert. Milepælene brukte vi til å se og holde oversikt over progresjonen vår.

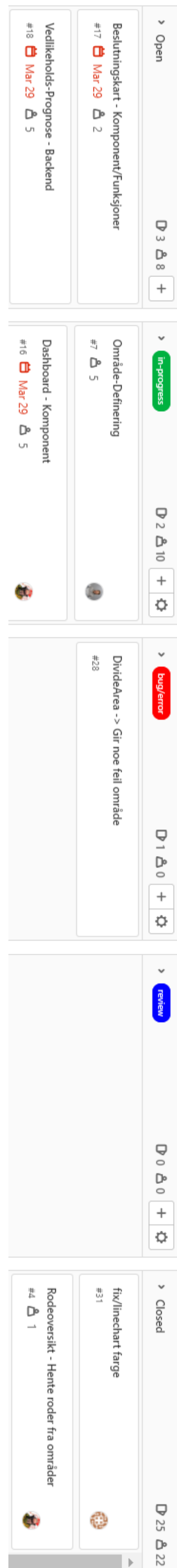
Git

Gitlab støtter Git som er et distribuert versjonskontrollsystem. Git gjør det lett for oss å jobbe på hver vår pc og slå sammen koden underveis. Med Git er det enkelt å flette filene sammen.

Git har en egenskap der man kan lage grener ut fra hovedprosjektet. Grenen er en egen versjon av hovedprosjektet der man kan lage og slette elementer uten at det påvirker selve prosjektet. Når man er fornøyd med endringen i grenen, kan man sette opp en forespørsel om å flette grenen inn i hovedprosjektet. Da må en annen i gruppen se over endringene og godkjenne dem før de blir flettet inn.

Produktkø

Produktkø er en del av planverktøyet til Gitlab. Med produktkøen får vi enkelt satt opp en oversikt over oppgaver som skal gjøres. Vi delt dem inn i de fem kategoriene: Open, in-progress, bug/error, review og Closed.



Figur 5.4: Produktkø

Open er den den første kategorien. Her la vi inn alle oppgavene vi skulle gjøre. Når vi startet på en av oppgavene, ble oppgaven flyttet over til **in-progress**, og den ble tildelt personen som skulle jobber med den.

I **bug/error**-kategorien legger vi inn problemer og feil som blir oppdaget i koden og måtte rettes. Når vi hadde gjort ferdig en oppgave eller rettet en feil, la vi dem inn i **review**-kategorien og lagde en forespørsel om å slå grenen inn i hovedgrenen der den skulle ligge fram til en annen i gruppen hadde sett over og godkjent endringen. Koden ble så lagt inn i hovedgrenen, og oppgaven ble lagt inn i **Closed**. Her ligger alle oppgaver som er fullførte.

Kapittel 6

Implementering

Implementerings kapittelet beskriver implementering av applikasjonen. Her går vi dypere inn i funksjonaliteten i de tre hovedgruppene: maskinlæring, frontend og backend. I maskinlæring vil du få lese mer om grupperingsalgoritmen, DBSCAN-algoritmen og klassifisering. I backend går vi dypere inn på funksjonaliteten knyttet til å hente, lagre og behandle data. I frontenden beskriver vi funksjonaliteten til kartet, dashbordet og tabeller.

6.1 Maskinlæring

Av maskinlæring er det to ledd vi har implementert, den ene er grupperingsalgoritmen og den andre klassifisering. Vi ønsker automatisk genererte områder som representerer områder med samme type vær. De anbefalingene som gis skal forbedre seg over tid basert på tidligere erfaringer. For å automatisk generere områder har vi da valgt å gå inn i grupperingsalgoritmer for å definere områder med lik værdata. Utfordringen her er å la algoritmen vite forskjell på geografisk distanse og generell distanse mellom værdata. Klassifiseringen er mer rett frem og vi ønsker å ha et beslutningstre som lærer av beslutninger som er gjort tidligere.

6.1.1 Grupperingsalgoritmen

Formålet med grupperingsalgoritmen er å få ut områder med relativt lik værmåling. Dette gjøres fordi vi antar at små forskjeller i geografi og vær kan utgjøre forskjeller ved beslutningene applikasjonen skal gjøre. For eksempel skal algoritmen finne områder der det er kjøligere, eller områder der det faller mer snø enn andre. Med dette kan systemet gjøre bedre prioriteringer på hvor utkallinger skal

sendes. Denne typen klyngeanalyse har vi funnet lite offentlig kildekode til, som betydde at vi måtte støtte oss på annen offentliggjort forskning innen emnet.

Vi har brukt en del tid på å finne en passende algoritme for å gruppere datapunktene våre. En del av vurderingen bak valgene vi har gjort er med tanke på vår begrensede erfaring innen maskinlæring, dermed blir populære og velutprøvde algoritmer prioritert over moderne eller nyere metoder. Valget ble mellom k-means og DBSCAN, der begge ble delvis implementert og utprøvd før vi valgte en spesifikk algoritme å implementere.

Disse to algoritmene har styrker og svakheter ovenfor hverandre som gjør det vanskelig å peke ut en uten å teste de på vår data. K-means har tendenser mot konvekse eller sirkulære former, samt antall grupperinger må defineres på forhånd. DBSCAN derimot klarer å definere områder med varierte former og finner antall grupperinger av seg selv. En annen svakhet ved K-means er at den tar i bruk alle datapunkter, mens DBSCAN klarer å definere punkter som støy dersom de ikke passer godt nok inn i en gruppering. En siste forskjell er at K-means fungerer godt på store datasett, mens DBSCAN fungerer dårlig uten en mer avansert implementering[7].

Code listing 6.1: DBSCAN Algorithm[1]

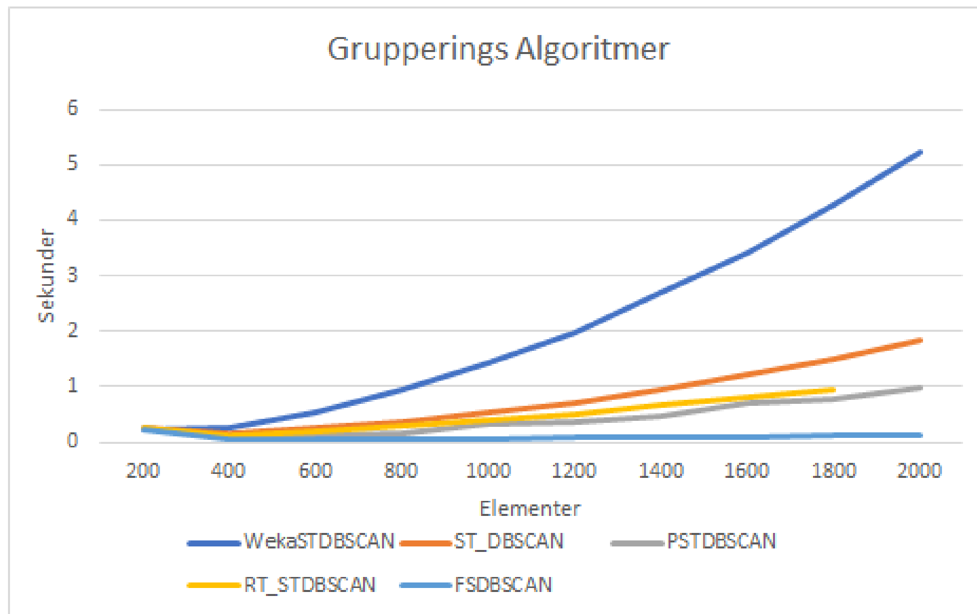
```
int clusterId = 0;
for (int i = 0; i < tree.size(); i++) {
    var point = treeElementAt(i);

    if (point.value().cid == UNLABELED)
        if (ExpandCluster(point, clusterId, eps))
            ++clusterId;
}
```

DBSCAN algoritmen 6.1 er en tetthetsbasert algoritme. Tettheten bestemmes av om et punkt overstiger et minimums antall naboer[1]. Algoritmen over vil utvide en klynge når den finner et punkt som ikke allerede er angitt en gruppe. Vi har testet forskjellige implementasjoner av algoritmen, der de vi har jobbet med er i hovedsak inspirert av STDBSCAN[8]. Det er slik at de fleste DBSCAN algoritmer er henholdsvis like, men forskjellene utspiller seg størst i henting av naboer. Dette gjelder da hvilke grenseverdier som skal brukes og hva de skal måles opp mot.

En implementasjon av ExpandCluster går vi inn i detalj på senere. Oppgaven dens er å gå utifra et punkt og finne naboer. Deretter markere de som en del av en gruppering og fortsette veksten til nye punkter ikke lenger defineres som naboer.

[1]MinPts



Figur 6.1: Tidsbruk av hver implementering

WekaSTDBSCAN [2] var vår første implementasjon av STDBSCAN[8], dette var et forsøk på å implementere funksjonen inn mot grensesnittet til Weka. Som en kan se i Figur 6.1 brukte denne lang tid og hadde høyt ressursforbruk. Algoritmens tidsforbuk viser seg at den er tilnærmet lik $O(n^2)$ som er enormt dårlig når det kommer til å prossesere store datamengder. Ved en nærmere titt fant vi ut at den brukte mye minne i systemet. Vi tenkte at en måte å forbedre algoritmen på var å minimere overhead i algoritmen.

STDBSCAN var en implementasjon som i all hovedsak er direkte lik forrige, men datastrukturen er redusert til bruk av en enkel `HashMap<>` der attributtnavn er nøkkelordet til en verdi. Samtidig ble noe mer jobb gjort ved filtrering av naboer basert på tidslikhet som også reduserte kalkulasjoner på distanser. Ved å gjøre dette klarte vi å forbedre algoritmen med et ganske stort steg. Likevel var algoritmen treg ved store datamengder ettersom den har en $O(nlogn)$ kompleksitet ved seg.

Code listing 6.2: Nabosøk i STDBSCAN

```
private List<ml_Instance> RetrieveNeighbours(ml_Instance instance) {
    return initial_instances.parallelStream()
            .filter( e -> {
                return non_spatial_distance(instance, e) < Eps1
                && spatial_distance(instance, e) < Eps2;
            })
            .collect(Collectors.toList());
```

[2]Spatio-Temporal Density Based Spatial Clustering of Applications with Noise

```
}
```

Begge STDBSCAN algoritmene implementerer et ekstra krav i nabosøket i forhold til standrad DBSCAN. I tillegg skulle de inneholde en tidsdistanse, men den hadde vi lite bruk for i vår kontekst.

PSTDBSCAN [3] ble vårt neste forsøk i å redusere kompleksiteten i algoritmen. Denne gikk ut på å parallellisere algoritmen ved å splitte datasettet utover fler tråder, for å så sette de sammen igjen til slutt. Dermed utnytte mer av maskinens kraft. Vår implementasjon var tungt inspirert av PDBSCAN hvor nettopp dette ble gjort. Samtidig blir alle filtreringer av lister parallellisert for å sørge for at innhenting av naboer går fortere. Med denne implementasjonen forbedret vi tidsforbruket av applikasjonen, mens minnebruk og prosessbruk gikk opp. Siden systemet fortsatt hadde $O(nlogn)$ kompleksitet ville den fortsatt øke drastisk med større datasett. Med dette vil den da stoppe systemet fra å kunne svare på andre kall ettersom alle ressurser brukes av algoritmen.

Code listing 6.3: Fordeling av partisjoner

```
/* Partition Data */
chunk_size = Math.max(2, instances.size() / availableProcessors());
// creating partitions of equal chunk size
for (int start = 0, end = 0;
end < instances.size();
start = start + chunk_size) {
    end = Math.min(instances.size(), start + chunk_size);
    partitions.add(instances.subList(start,end));
}
```

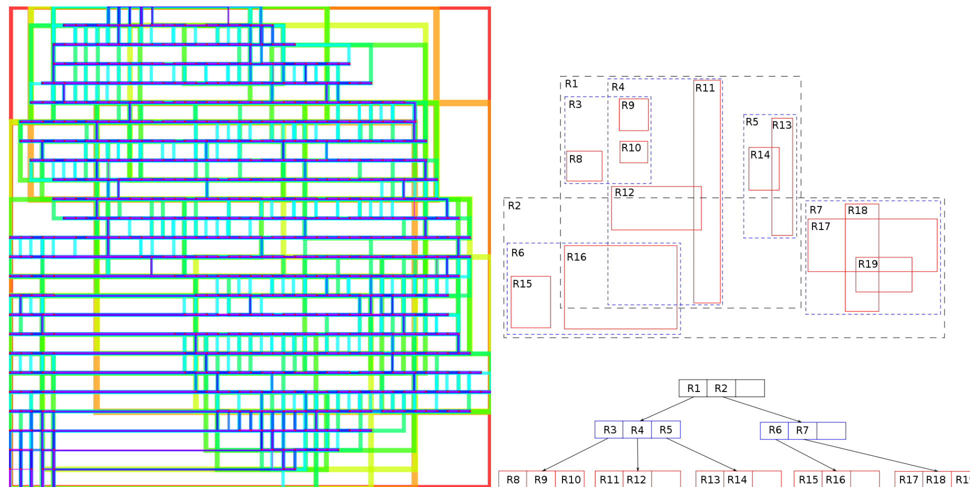
I utsnittet 6.3 viser vi til hvordan datasettet blir fordelt utover fler partisjoner. Hver av disse partisjonene blir så gruppert hver for seg på hver sin tråd, de punktene som ikke er med i partisjonen markeres som 'ikke sett'. Dette får algoritmen bruk for senere når de skal slås sammen.

Denne implementasjonen er basert på brukeren snithish fra github[9] og alle punktene som er markert 'ikke sett' samles. Deretter spleises de sammen via bruk av java `Lock`.

RTSTDBSCAN [4] var vårt første forsøk på å forbedre datastrukturen algoritmen henter fra. R-Tre er den foretrukkede datastrukturen når det kommer til denne algoritmen fordi den forbedrer prosessen med å hente inn data massivt.

R-Treet forbedrer søk av punkter drastisk over bruk av vanlige lister. Treet går ut på å indeksere punkter i noder, der en node kan inneholde et gitt antall punkter.

[3] Parallell-STDBSCAN
[4] R-Tre STDBSCAN



Figur 6.2: R-Tre kart over værdata fra Gjøvik Kommune ved siden av enkel tegning av R-Treets oppbygning

Hver node kan representeres med en avgrensningsboks. I Figur 6.2 kan man se slike avgrensningsbokser rundt hvert datapunkt vi har for kommunen Gjøvik. I dette bildet vil f.eks rød boks representere alle punktene, mens blått representerer et mindre utvalg av punkter. Denne strukturen gjør at søk kan ignorere punkter som er uaktuelle. Typiske bruk av R-Tre kan f.eks være å finne alle restauranter innen 5km.

R-Treet er et balansert tre, altså alle blad-noder er på samme dybde. Når et punkt legges til starter den på toppen av treet, og går rekursivt gjennom nodene. Så finner punktet den noden som krever minst endring av avgrensningsboksen og fortsetter ned treet til den finner siste node. Der legges punktet inn om det er plass, ellers splittes noden og treet balanseres på nytt[10].

Søk i et R-Tre har normalt en kompleksitet på $O(logn)$, men i værste tilfelle $O(n)$, dette er langt bedre enn å søk gjennom lister. Ved listebruk er det slik at man må traversere først hele listen, og per ledd traversere alle på nytt for å finne alle naboene. Dermed er kompleksiteten og tids-forbruken eksponentiell eller $O(n^2)$.

Implementasjonen av R-Tre i oppgaven er hentet fra GitHub og er implementert av David Moten[11]. Grunnen til at denne ble benyttet er at det fantes få alternativer, men likevel var denne godt implementert og funksjonell. Vi hverken tid eller ressurser til å implementere vårt eget R-Tre. I tilegg støttet den funksjoner for å visualisere treet som var nyttig i testing.

FSDBSCAN [5] er den algoritmen vi resulterte med etter utprøving og testing av andre algoritmer. Den implementerer i hovedsak FDBSCAN [1] som utnytter konseptet 'region growing' hvor hver gruppering finner fler av sine naboer før den går videre. Vår implementasjon tar med et konsept fra STDBSCAN der vi henter naboer basert på forskjellige grenseverdier som måles mot forskjellige attributter. Mer spesifikt da koordinater måles mot koordinater og værdata mot værdata. Dermed har vi kalt vår algoritme for FSDBSCAN ettersom distanse måles to ganger med forskjellige hensikter.

Code listing 6.4: ExpandCluster[1]

```
var candidateSeeds = regionQuery(point, eps);

if(candidateSeeds.size() < MinPts) {
    point.value().cid = NOISE;
    return false;
} else {
    Instance.changeIds(candidateSeeds, clusterId);
    var repSeeds = rep_seeds_select(candidateSeeds, point);

    while ( !repSeeds.isEmpty() ) {
        var currentP = repSeeds.get(0);
        var result = regionQuery(currentP, eps);

        if (result.size() >= MinPts ) {
            var repResultP = rep_seeds_select(result, currentP);
            for (var p : repResultP)
                if (p.value().cid == UNLABELED)
                    repSeeds.add(p);
            for (var p : result)
                if (p.value().cid == UNLABELED || p.value().cid == NOISE )
                    p.value().cid = clusterId;
        }

        repSeeds.remove(currentP);
    }
}
return true;
```

Algoritmen i utsnitt 6.4 henter først naboene til startpunktet for grupperingen. Dersom et punkt ikke har nok naboer markeres den som støy og går til neste punkt. Ellers markerer den alle naboene etter samme grupperings id og henter en representativ gruppe av grensepunkter, for å fortsette ekspansjonen av grupperingen. Den går så gjennom alle disse punktene med samme hensikt som starten av funksjonen. Å sjekke at de har nok naboer, markerer de som støy eller klynge og går videre. Hvert nye ytre punkt i denne delen går gjennom samme søk for å finne representative ytre datapunkter for å ekspandere grupperingen.

Code listing 6.5: regionQuery

```
regionQuery(Entry<Instance, Point> point, double eps) {
    return this.tree.search(point.geometry(), eps)
```

[5] Fast Spatial DBSCAN

```
                .toList()
                .toBlocking()
                .single()
                .parallelStream()
                .filter(e -> distance(e.value(), point.value()) < eps2)
                .collect(Collectors.toList());
}
```

Naboene til et punkt hentes via et søk i R-Treet. R-Treet tar imot to parametre hvor den først er punktet som skal søkes rundt, og deretter en radius på selve søket. Verdien `eps` brukes i algoritmen som grenseverdi for geografisk distanse, det er også geografisk lokasjon R-Treet er indeksert etter.

Etter R-Treet returnerer nærliggende punkter måles de opp mot distanse verdiene mellom de resterende attributtene. Vi bruker en enkel distanse funksjon som beregner Euklids distanse. Denne distansen må være innen for `eps2` grenseverdien.

Code listing 6.6: rep_seeds_select[1]

```
List<Entry<Instance, Point>> rep_seeds = new ArrayList<>();
for (int i = 0; i < rep_MinPts; i++) {
    double maxDist = 0;
    double minDist = Double.MAX_VALUE;
    Entry<Instance,Point> maxPoint = null;
    for (Entry<Instance, Point> p : candidate_seeds )
    {
        if( i == 1)
            minDist = distance(p.value(),point.value());
        else {
            // minDist = min{dist(p,q) | q is element of rep_seeds }
            for (Entry<Instance, Point> q : rep_seeds) {
                var d = distance(p.value(),q.value());
                if(d < minDist)
                    minDist = d;
            }
        }
        if ( minDist >= maxDist ) {
            maxDist = minDist; maxPoint = p;
        }
    }
    if(maxPoint != null)
        rep_seeds.add(maxPoint);
}

return rep_seeds;
```

Utsnittet 6.6 viser hvordan en finner representative grensepunkter. Den går gjennom kandidatpunktene som er naboene til punktet `Point` og finner de punktene med størst distanse fra `Point`. `rep_MinPts` representer det antallet representative punkter et punkt har. Disse returneres til `ExpandCluster` som da igjen finner nabopunktene videre. I vår implementasjon har vi valgt tallet tre som antall representative punkter og det ser ut til å fungere greit.

Når algoritmen er ferdig kan grupperingene hentes ut via en `getAssignments()` funksjon som returnerer gruppe identifikasjonen til hver indeks i treet. Deretter lagres de som områder ved å kjøre en konveks omhylnings funksjon på grupperingene. Deretter lagres de som områder ved å kjøre Jarvis' algoritme[12] som generer en konveks omhylning[13] av punktene.

Parameterene i algoritmen ble justert frem til vi kom til en passende mengde grupperinger etter vår mening. Vi kom frem til at `eps` skulle være 0.05 som ga en grei geografisk avstand, `eps2` ble satt til 3 `MinPts`til 4, og `rep_Minpts`til 3. Ved endringer i distanse grensene fikk vi fort veldig mange grupperinger eller ingen. Dermed fant vi ut at den er ømfintlig for små endringer og det kan være vanskelig å finne perfekte verdier. Vi valgte å ikke fokusere særlig mer på å finjustere parametrene da algoritmen ble ferdig implementert sent i arbeids-prosessen.

6.1.2 Klassifisering

Målet for denne applikasjonen er å gi anbefaling til hvor og når man bør brøyte og salte veier, også der det ikke er åpenbart at det burde bli sendt ut. Klassifisering av data er en teknikk innen maskinlæring der man finner et mønster i beslutninger fra treningsdata, som i vårt tilfelle er utkalling til brøyting, salting eller ingenting. Det er mange algoritmer innen klassifisering som er gode på å løse og finne mønster og sammenheng i forskjellige scenarier. Vi valgte å bruke algoritmen J48 som setter opp et beslutningstre ut fra treningsdata som kan brukes til å gi anbefalinger for ny værdata.

Treningsdata

Anbefalingen som blir gitt er bare like god som treningsdataene den får inn. For å ha kontroll på treningsdataene har vi en egen tabell i databasen vår som lagrer posisjon til område, vedlikehold og tid. Hver gang en vaktleder sender en utkalling vil den bli lagret i treningsdataene. For å sikre at treningsdataene er riktig er det mulig å slette eller endre data i situasjoner der utkallingen er feil. Når klassifisereren henter ut treningsdata kobler den det til alle værmeldinger som er satt til denne tiden. Siden vi henter inn nye oppdaterte værmeldinger for 24 timer fram i tid, hver time, vil hver treningsdata bli lenket opp til 24 forskjellige værmeldinger. Dette er nøkkelen til at maskinlæringen kan finne mønster som er vanskelig for en vaktleder å finne, og gi en anbefaling som kan skille seg fra hva som normalt ville blitt kalt ut.

Klassifiserer

Funksjonen `addClassifier(Long id)` blir kalt på automatisk når ny værdata blir lagt til suksessfult og sender med id til værdata som parameter. Så vil den hente treningsdata og sende det inn i et J48 beslutningstre. Værdata blir så hentet ut av databasen ut ifra id og det gis en anbefaling på hver værvarsel frem i tid som lagres i beslutning-tabellen hvor det kan hentes og presenteres til vaktleder.

```
air_temperature <= 3.9
|   coordinates_high <= 471: NONE (4.0)
|   coordinates_high > 471: SALT (9.0)
air_temperature > 3.9
|   air_pressure <= 1019: BROYT (8.0)
|   air_pressure > 1019
|   |   coordinates_high <= 172: BROYT (2.0)
|   |   coordinates_high > 172: NONE (6.0)

Number of Leaves  :     5

Size of the tree :   9


Results
======

Correctly Classified Instances          29              100      %
Incorrectly Classified Instances         0                0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               29

.
```

Figur 6.3: Statistikk av beslutningstre

Beslutningstreet øverst i figur 6.3 er resultatet av tilfeldig treningsdata sendt inn i klassifisereren. Anbefalingen gitt fra dette treet vil ikke være korrekt. Dette er bare for å vise et eksempel på hvordan treet blir bygget opp. Klassifisereren ser på treningsdata den har fått inn, og bygger grener fra datapunktene som den ser henger sammen med en beslutning. Den bygger et beslutningstre hver gang den

skal gi en anbefaling, som gjør at treet kommer til å endre seg når ny treningsdata blir lagt til. Dermed lærer den.

I figur 6.3 viser den også hvor mange korrekte klassifiserte instanser som er i beslutningstreet. Denne statistikken peker til hvor mye av treningsdataen som faktisk passer til treet som er generert. I vårt tilfelle vil vi ligge mellom 80-100% siden vi forventer at det kommer værdata og utkallinger som ikke alltid vil stemme. Dette er med antakelsen av at mesteparten av treningsdata er korrekt, ligger man under 80% risikerer man å ignorere riktig data. Det er viktig at klassifisereren har rom til å kunne se bort ifra disse avvikene og heller fokusere på flertallet av treningsdata. Man kan også vektlegge de viktige attributtene som nedbør og temperatur slik at disse blir mer prioritert. For å finjustere hvordan beslutningstreet blir bygget trenger man mye god treningsdata. Vi har skrevet utdypende om dette i Videre-arbeid 9.1

6.2 Backend

I backend går vi inn på behandling av data ved hjelp av interne og eksterne APIer. For å håndtere begrensninger på API kall har vi laget et kø-system. I tillegg vil du få et innblikk i hvordan databasen vår er satt opp med Spring Boot.

6.2.1 API

Ekstern API

Ekstern API er de APIene vi henter data fra. Dette gjelder Met og Frost APIene.

Det er flere deler som innebærer i å hente data fra Met. En viktig del av det er hvilke endepunkt man bruker for å kalle på den. Met APIen har flere forskjellige endepunkter som gir forskjellige type data om ønskede områder. Vi ønsket å få all data for et område ut fra koordinater. Da var complete[6] endepunktet riktig for oss. For den leverer all data lagret for ønsket koordinat.

Code listing 6.7: Bygger url for API kall

```
public static String buildURL(Float lat, Float lon){
    DecimalFormat f = new DecimalFormat("#.####");
    String api =
    "https://api.met.no/weatherapi/locationforecast/2.0/complete?lat=%s&lon?%s";

    return String.format(api, f.format(lat), f.format(lon));
}
```

[6]https://api.met.no/weatherapi/locationforecast/2.0/complete

Complete endepunktet til Met tar lengde- og breddegrader som parametere. Endepunktet godtar koordinater med opptil fire desimaler. Blir det for mange desimaler vil ikke Met godta det, grunnet at det blir for spesifikke koordinater. For å følge kravene til endepunktet lagde vi funksjonen buildURL", se utsnittet 6.7. Denne funksjonen tar imot lengde- og breddegrad i form av flyttall. Inni funksjonen blir flyttallene konvertert til string format og lagt inn i url-en til API kallet.

Dataene vi får fra Met kommer i Json format. For å lettest mulig lagre dataene valgte vi å lage en klasse som tilsvarer formatet på Json strukturen til dataene. Da kan vi ved hjelp av Gson gjøre om Json data til et Java objekt. Gson er en Json omformer som brukes til å omformere Json data til Java objekt eller motsatt[14]. Data vi henter blir så lagret til databasen for videre bruk.

Met APIen leverer data for flere dager framover i tid. Etter de første 24 timene framover i tid, begynner værprognosen å bli upresis og mangelfull. Grunnet dette har vi valgt å bare lagre og bruke data for de neste 24 timene.

Intern API

Systemet vårt inneholder interne APIer som er koblet opp mot databasen. De interne APIene er koblingen mellom frontend og backend i systemet. Med disse APIene kan en hente, endre, slette eller lagre ny data til databasen. Vi har forskjellige endepunkter som er koblet opp til de forskjellige tabellene i databasen. Vi lagde en fil som heter ApiFrontController, denne filen er kontrolleren over alle endepunktene. Det er her du får tilgang til alle endepunktene.

Code listing 6.8: ApiFrontController

```
@RestController
@CrossOrigin(origins = "http://localhost:8081")
@RequestMapping("/api")
@ResponseBody
public class ApiFrontController {
    @GetMapping("/forecast")
    public ResponseEntity<String> forecast(@RequestParam float lat,
    @RequestParam float lon) {
        return ForecastView.get(MetController.get(lat, lon));
    }

    @PostMapping("/camera")
    public ResponseEntity<String> addCamera(
    @RequestBody TableCamera newCamera,
    @RequestParam Integer auth){
        return CameraView.get(
        CameraController.addCamera(newCamera, auth));
    }

    @GetMapping("/camera")
    public ResponseEntity<String> getAllCamera(){
        return CameraView.get(CameraController.getAllCamera());
```

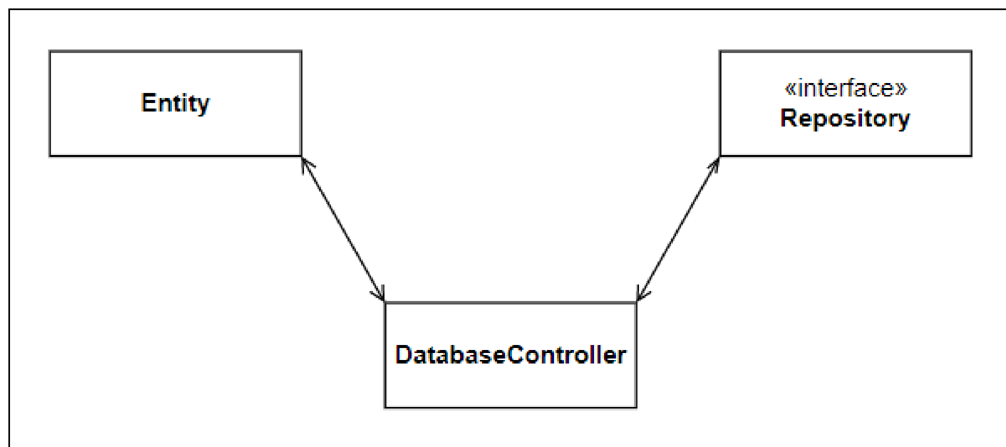
```
    }

    @GetMapping("/camera/{id}")
    public ResponseEntity<String> getIdCamera(@PathVariable Long id){
        return CameraView.get(CameraController.getIdCamera(id));
    }
```

Utsnittet 6.8 viser et lite utdrag fra ApiFrontController. Her kan du se at vi har et eget endepunkt til filen som er "/api". Skal man videre til en av de andre endepunktene, må man for eksempel legge på "/camera"og sende med riktig parametere for ønsket data. De forskjellige endepunktene tar imot forskjellige forespørsler som POST, PUT, GET og DELETE. GET forespørsel er for å hente data, POST er for å lagre data, PUT for å endre data og DELETE for å slette data. Når du gjør en forespørsel til en av APIene vil du få tilbake en respons. Responsen sir om forespørselen din ble fullført eller ikke.

6.2.2 Database

Databaseen er knyttepunktet for all data i systemet. Både interne og eksterne APIer er koblet opp mot databasen.



Figur 6.4: Database struktur

Figur 6.4 viser strukturen for håndtering av data i applikasjonen. I applikasjonen brukte vi Spring Boot sin struktur for å kommunisere med databasen. Den kobler sammen databasen med Java klasser ved å bruke `Entity`, `Repository` og `DatabaseController` klassene.

Entity klassen initialiserer og definerer dataene som skal være i tabellene. Inni klassen setter man datavariabler og relasjoner mellom andre tabeller. Det er denne som lager konstruktor- og hentefunksjoner for variablene, som gjør at variablene i klassen samsvarer med det i databasen. Hvis man skal håndtere data fra databasen får man det som en `Entity` klasse

Repository grensesnittet kommuniserer med databasen. Det er en `Repository` klasse for hver `Entity` klasse. Det er her funksjonene for å hente, lagre og slette data finnes ved hjelp av SQL-spørring. Den sammenkobler `Entity` klassene med SQL-spørring

DatabaseController er en singulær klasse som binder sammen `Entity` og `Repository`. Den står for all håndtering av data ved å lage et nytt objekt av `Entity` klassen, fyller den med data og lagrer den med grensesnittet. Hver tabell som blir laget får sin egen `Entity` klasse og `Repository` grensesnitt. Kontrolleren er fleksibel og kan bli utvidet med nye funksjoner for data håndtering etter behov.

6.2.3 Kø-System for Met-grensesnittet

Code listing 6.9: Medlemsfunksjon fra Invoker som returnerer Invoker instans

```
/* Member of Invoker class */
public static Invoker getInstance() {
    Invoker result = instance;
    if (result == null) {
        synchronized (mutex) {
            result = instance;
            if (result == null)
                instance = result = new Invoker();
        }
    }
    return result;
}
```

Kø systemet vårt implementeres som en global klasse der `Invoker` fungerer som et grensesnitt for køen. Denne er implementert slik at den kan aksesseres av fler tråder uten å låse seg eller miste kontroll på data. Utsnittet 6.9 viser hvordan man henter en instans av grensesnittet. Klassen er implementert etter Singleton mønsteret og dermed lagrer den en instans av seg selv i `instance` variabelen. For å sikre at ikke fler enn en instanse initialiseres introduseres en lås `mutex` som sørger for at alle tråder som opererer på denne funksjonen kun initialiserer en instans. Etter den blir initialisert returneres den og programmet kan benytte seg av køsystemet.

Code listing 6.10: Medlemsfunksjon fra Invoker som tar imot et kall

```
/* Member of Invoker class */
public Future<String> request(Request task) {
    FutureTask<String> taskf = new FutureTask<>(task);
    queue.add(taskf);

    startRunner();
    return taskf;
}
```

`Invoker`klassen tar imot et kall via `requst(Request task)` som da enkapsulerer oppgaven i `FutureTask`objektet. Denne sendes videre til køen og `Executor`tjenesten startes. Når en sender et kall returneres et objekt med en gang som kan hente resultatet av kallet når det er klart.

Code listing 6.11: Medlemsfunksjon i Executor som implementerer vent dersom det trengs mellom kall

```
/* Member of Executor class */
private void executionWaitTime() {
    if(Invoker.getLastExecution() != null) {
        var waitTime = 200; // ms
        var ts1 = new Timestamp(Invoker.getLastExecution().getTime() + waitTime);
        var ts2 = new Timestamp(System.currentTimeMillis());

        if(ts1.compareTo(ts2) > 0) {
            try {
                Thread.sleep((ts1.getTime() + waitTime) - ts2.getTime());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

For å sørge for at antall kall per sekund til Meteorologisk Institutt ikke overstrider 20 kall i sekundet har vi implementert et vent mellom kall. Vår første implementasjon av denne var enkel og benyttet seg bare av `wait` funksjonen i Java, dette er sløsing av ressurser ettersom det er ingen garanti for at det finnes et nytt kall som er klart for å kjøres.

Dermed endret vi til at hvert kall oppdaterte et tidsstempel som `Executor` tjenesten kunne sjekke. Her finner vi ut hvor lang tid er gått siden forrige kall, differansen mellom det og nåtid brukes til å sette tråden i dvale inntil nok tid er gått. Variabelen `waitTime` inneholder antall milli-sekunder programmet skal vente mellom hvert kall, denne er satt til 200 som vil si at programmet kjører 5 kall i sekundet. Dette er da godt innenfor de rammene som Meteorologisk Institutt sitt grensesnitt har.

Code listing 6.12: Medlemsfunksjon fra Executor som kjører selve kallet

```
/* Member of Executor class */
private void execute() {
    Invoker.queue.pop().run();
```

```
    Invoker.setLastExecution(new Timestamp(System.currentTimeMillis()));
}
```

I utsnittet 6.12 kjøres selve kallet. Her hentes `FutureTask` objektet fra køen og deretter kjøres. Når den er ferdigkjørt oppdateres tidsstempelet. Ettersom selve kallet implementerer `Callable<String>` returnerer den resultatet av kallet til `FutureTask` som lagrer resultatet i en variabel som hentes via en `get()` funksjon.

6.2.4 Innhenting av Værdata for en kommunegrense

Code listing 6.13: Algoritme som deler et område inn i punkter

```
public static ArrayList<Point> get(double separation, ArrayList<Point> border) {
    ArrayList<Point> points = new ArrayList<>();
    Polygon polygon = new Polygon(border);
    Rectangle rect = PolygonBoundingBox.get(polygon);
    // Rows
    if (boundingBox != null) {
        for(double w = rect.x; w < rect.x + rect.length; w += separation)
        {
            for(double h = rect.y; h < rect.y + rect.height; h += separation)
            {
                Point p = new Point(w,h);
                if(PolygonPointCollider.checkInside(polygon, p))
                    points.add(p);
            }
        }
    }

    return points;
}
```

For å samle inn data for et avgrenset område må vi definere en metode for å jevnt fordele koordinater innad grensen. For å komme frem til dette definerer vi et aksebundet rektangel rundt grensen[7]. Denne lages ved å finne størst og minst x og y verdi for grensen. Punkter plasseres deretter utover dette rektangelet med jevnt mellomrom. Avstanden mellom punktene bestemmer presisjonen av grupperingsalgoritmen.

```
public static boolean checkInside(Polygon poly, Point p) {
    if(poly.size() < 3)
        return false;
    Line exline = new Line(p, new Point(Double.MAX_VALUE, p.getY()));
    int count = 0;
    int i = 0;
    do {
        Line side = new Line(poly.get(i), poly.get((i + 1)% poly.size()) );
        if(isIntersect(side, exline)) {
            if(direction(side.p1, p, side.p2) == 0)
                return onLine(side, p);
```

[7] MBR, Minimum Bounding Rectangle

```
            count++;
        }
        i = (i+1)%poly.size();
    } while(i != 0);
    return count % 2 != 0;
}
```

Før et koordinat er lagt inn i en liste av punkter vi skal hente data på sjekkes det om den er innenfor grensen. Dermed blir punktene som returneres jevnt fordelt innenfor grensen. Alle punktene som nå er bekreftet innenfor grensen blir så omgjort til kall for innhenting av værdata. Hvert kall lagrer resultatet i databasen.

6.3 Frontend

Her kommer vi til å forklare hvordan vi både henter data, manipulerer det og sender videre til komponenter for å presenteres på en god måte.

6.3.1 API Forespørsler

For å hente inn data fra databasen lagde vi funksjoner for å sende forespørsler for å hente, sette, endre og slette data. Vi installerte disse inn i Vue der den blir instansiert. Dette gjør at alle Vue komponenter har tilgang til å bruke funksjonene og få returnert dataene den forventer. Alle funksjonene er asynkrone siden det kan ta tid å få svar fra databasen. Dette er ikke et problem siden Vue er et reaktivt program og vil automatisk oppdatere relevante dataset når de blir endret.

6.3.2 Tabeller

All data som presenteres i applikasjonen blir sendt som lister med objekter, derfor utviklet vi en tabell-mal i Vue fra bunnen av. Denne malen har tre variabler: *kolonner*, *radElementer* og *sorteringsNøkkel*. Kolloner variabelen er en liste med kolonne-objekter, en kolonne inneholder navn, navn på elementet og sorteringteknikk (alfabetisk er satt som standard). For at dataene skal presenteres må navn på element i kolonnene samsvare med dataene i radElementer. Denne måten å lage en tabell på gjør at den er både lett å bruke på forskjellige lister med data og man kan enkelt justere data som presenteres. Vue har det som heter kalkuleringfunksjoner, disse vil oppdatere seg når relevant data endres. Vi implementerte en kalkulering-funksjon som returnerer en liste av elementene som blir vist i tabellen. Denne funksjonen går igjennom variablene for søk og sortering og bruker liste-funksjonene `slice`, `sort` og `some` og returnerer listen.

Code listing 6.14: Sender med to knapper i kolonne med navn call og edit

```
    <template v-slot:call="item">
      <button v-bind:item="item.item">Utkalling</button>
    </template>
    <template v-slot:edit="item">
      <button v-bind:item="item.item">Endre</button>
    </template>
```

Ved å bruke `slot` funksjonaliteten til Vue kan man sende med html elementer i en `template`. Ved å navngi den med samme navn som kolonnen blir den satt inn i hver rad i tabellen. Html elementet får tilsendt data til elementet i raden, som gjør det mulig å lage funksjoner som redigerer data eller sender den videre.

6.3.3 Beslutningskart/Dashbord

Den største og viktigste delen av frontenden er dashbordet. Det er her vaktleder vil få opp beslutningsstøtten og har mulighet til å få mer data fra forskjellige områder. Dashbord-siden er delt opp i fire hoved-komponenter: Kart, menybar, rodetabell og informasjonspanel.

Håndtering og distribuering av data

Dashbord-siden henter inn all relevant data på oppstart og distribuerer den til de komponentene som trenger. Grunnen til at vi gjør det på denne måten er at vi bruker dashbordet som et knutepunkt for de andre elementene. Dette gjør at hvis vaktleder f.eks trykker inn på et område via sidemenyen vil dataene for rodene som skal vises i kartet også oppdateres. Dette løste vi ved at dashbordet har variabler som lagrer hvilke område som er valgt, samt hvilke data innenfor område som skal vises. Dataene blir sendt igjennom en kalkulerings-funksjon som filtrerer den riktige data ut fra variablene og returnerer den til komponentene.

Code listing 6.15: Kalkulerings-funksjon for filtrering av roder

```
computed:{
    chosenRoder: function(){

      if(!this.showRoder.length)
        return this.roder

      let roder = []
      for(let i = 0; i < this.showRoder.length; i++)
        for(let j = 0; j < this.roder.length; j++) {
          if ( this.showRoder[i] === this.roder[j].id ) {
            roder.push(...[], this.roder[j])
          }
        }
      return roder
```

```
    }
}
```

For å spare på minne så hentes informasjon som værdata og kamera til et område først når vaktleder trykker inn på det.

Kapittel 7

Kvalitetssikring

I dette kapittelet går vi inn på det vi gjorde for å kvalitetssikre koden vår.

7.1 Kode validering

Når vi har utviklet et element for prosjektet og skal flette det inn i utviklingsgrenen, setter vi opp en forespørsel om å flette elementet inn i grenen. For at forespørselen skal bli godkjent må en av de to andre i gruppen gå over forespørselen. Da skal de se over koden og sjekke at den funker som den skal og at den ikke ødelegger for den allerede eksisterende koden. Hvis koden funker som den skal blir den godtatt og flettet inn i utviklingsgrenen, hvis ikke så må den bli fikset før vi kan flette den inn.

7.1.1 SonarLint

Etterhvert fant vi ut at vi ikke var fornøyd med tilbakemeldingene Intellij ga oss på koden våres. Vi ønsket bedre tilbakemeldinger, eller varsler på dårlige metoder. Dermed installerte vi SonarLint mot slutten av prosjektet for å gå over. Denne kom med mange varsler på fler av filene våre om at f.eks enkelte liste-klasser ikke burde brukes, eller hvor det potensielt kunne være andre sikkerhetshull.

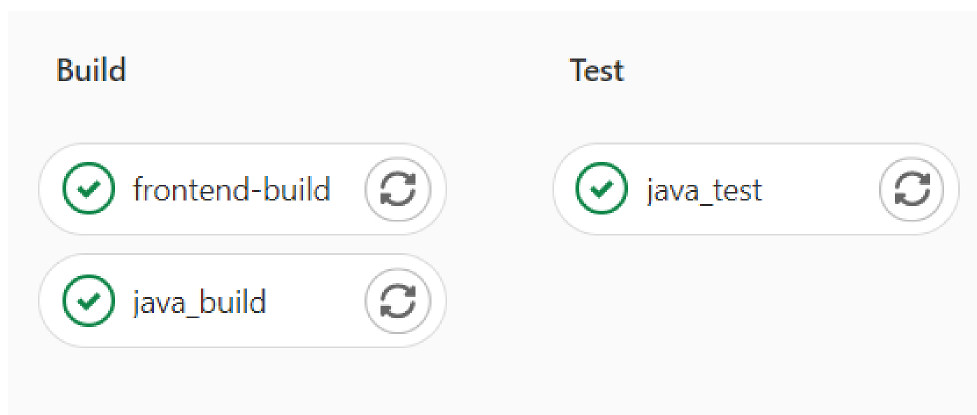
7.2 Versjonskontroll

Vi brukte Git for å opprettholde kontroll på versjonene av systemet. Git er et versjonskontrollsystem som tilatter utviklere å lage fler grener av programmet og flette de sammen når enkelte funksjoner er ferdige. Samtidig kan en markere versjoner med 'commits' for å ha punkter å falle tilbake til dersom man skulle gjøre noe galt.

7.3 Gitlab Automatisert Pipeline

NTNU's Gitlab har offentlige tjenere som bygger og kjører programmet vårt ved nye oppdateringer. Vi har lagt som krav at dersom en gren skal legges inn i hovedgrenen eller utviklingsgrenen skal den få en godkjenning fra pipeline. Den kan også konfigureres til å kjøre tester på programmet og ut ifra disse igjen gi godkjenning eller ikke. Foreløpig er den ikke konfigurert til å kjøre særlig testing.

Hovedtanken bak bruken av pipelines er å unngå det tilfelle at programmet kjører på en datamaskin, men ikke på en annen. Dette er et problem vi har hatt i tidligere prosjekter iløpet av studietiden. Vi har lært at et slikt problem kan føre til unødvendig ressursbruk ved å måtte fikse det. Mistanken om dette problemet ble realisert og pipelinen reddet oss fra et par dårlige sammenspleisinger av grener. Kostnaden av å sette dette opp mot å fikse eventuelle problemer under sammenspleising er vanskelig å måle, men basert på erfaringer mener vi at den har spart oss en del tid. Den største fordelen er økt trygghet i at programmet kjører for alle når man jobber hver for seg. Dette har vært til stor hjelp under covid-pandemien, der tett samarbeid ikke alltid er like lett.



Figur 7.1: Pipeline test

Kapittel 8

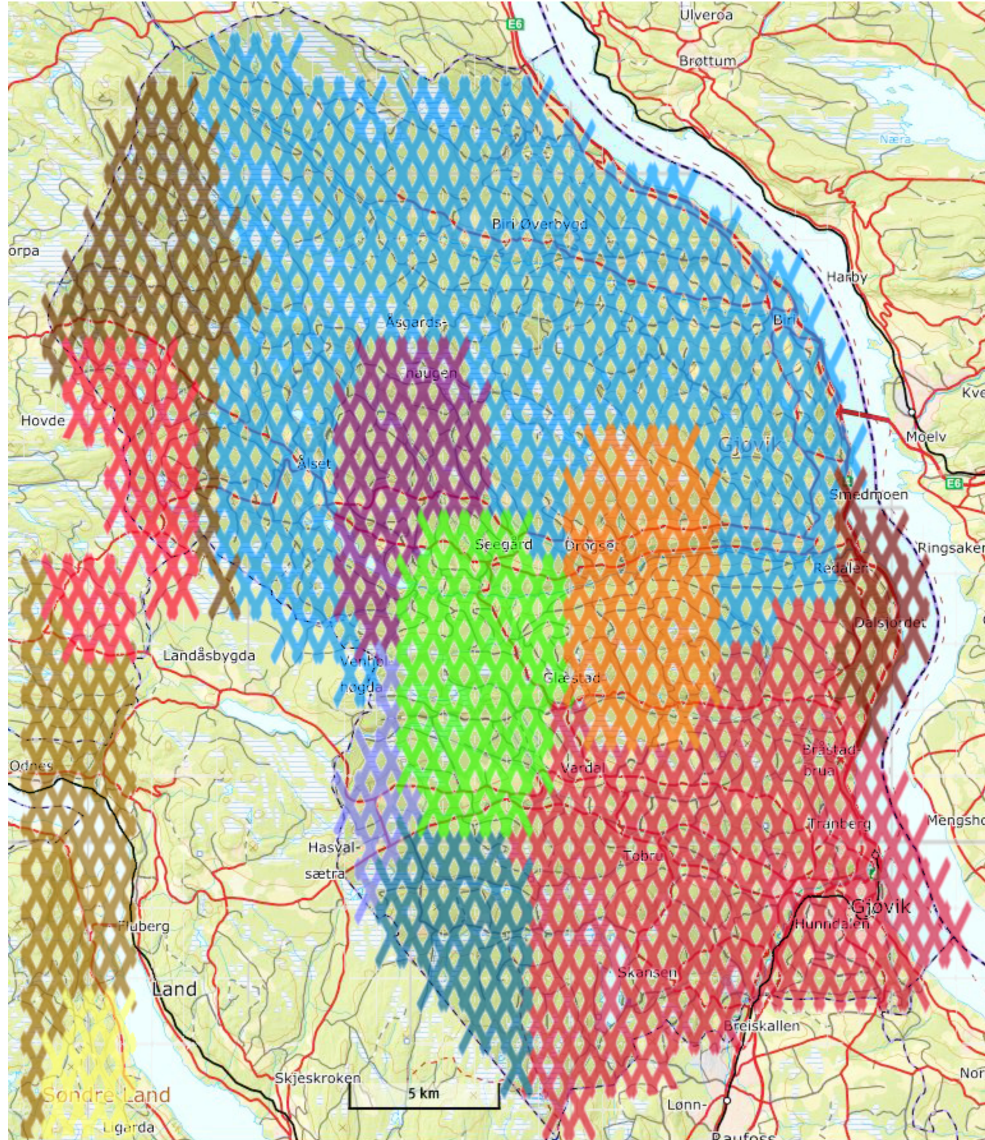
Drøfting av Prosjektresultat

I drøftings kapitlet skal vi ta opp problemstillinger som har dukket opp under prosjektet og det vi har lært av det. Her vil dere få et innblikk inn i våre vurderinger av prosjektet.

8.1 Resultater

.

8.1.1 Områdedefinering



Figur 8.1: Resultat av FSDBSCAN, hver gruppering vises med sin egen farge

Figur 8.1 viser hvordan grupperingene har utformet seg for Gjøvik Kommune. En kan se fem former som utformer seg som sirkulære og potensielt dårlig gruppert. Dette er da gul, rød, lilla, grønn og oransje. En kan lure på hva disse områdene representerer, en tanke er at det kan være data fra Meteorologisk Institutt som inneholder feilberegninger. Om det er slik må en kanskje revurdere metoden som er tatt i bruk for å innhente datapunkter.

Vi regner med at Meteorologisk Institutt sin data er nokså presise, dermed mener vi at en mer reell mulighet er at parametrene våre burde finjusteres mer. Samtidig kan en vurdere vekting av attributtene, heve frem temperatur forskjeller og nedbør er en mulighet.

En feil som ikke er blitt nevnt så langt i rapporten er rektangelet nederst til venstre i Figur 8.1. Denne kommer av en feil i funksjonen som sjekker om et punkt er innenfor en geometri eller ikke. Denne ble regnet som lite viktig i henhold til andre problemer og oppgaver vi måtte fullføre. Konsekvensen er at det er grupperinger som er utenfor det definerte området vi ønsket å bruke.

Ettersom grupperingene inneholder relativt lite data kan det hende at de grupperingene som er gjort er lite representative over lang tid. For å unngå dette burde man samle inn mer data over lengre tid for å danne mer robuste grupperinger. Dette er en del av problemstillingen til hvorfor vi brukte så lang tid på å implementere DBSCAN algoritmen ordentlig, vi trengte at den fungerte bra på større datasett. Derav fikk vi en ganske god algoritme, som burde vært testet på enda større datasett.

8.1.2 Klassifisering

Treningsdata

Vi skulle gjerne hatt et datasett med god treningsdata vi kunne gitt til maskinlæringen vår. Da kunne vi finjustert den og gitt et enda bedre grunnlag for ETC når de skal starte å bruke den. Dette skjedde ikke, på grunn av flere faktorer: Den første faktoren er at vi ble ferdig med innhenting av data etter at snø-sesongen var ferdig. Vi fikk da ikke gode målinger å lage treningsdata av. Hadde vi hatt værdata ville det også vært tidkrevende å manuelt sette opp hver utkalling. Siden vi ikke har erfaring som vaktledere ville våre estimerte utkallinger ført til unøyaktig værdata for klassifisereren. En løsning på dette kunne vært å høre med ETC om å få en liste over utkallinger de har gjort, og lagd treningsdata ut av den. Nå som vi har fått erfaring med maskinlæring, vet vi hvor viktig treningsdataene er for å finstille klassifisereren. Hadde vi visst dette da vi startet, ville vi prioritert å fått laget treningsdata tidligst mulig for å kunne videreutviklet maskinlærings-prosessen i størst mulig grad.

8.1.3 Frontend

Vi ble fornøyde med implementasjon av GUI-et, men vi brukte mye tid på utviklingen av den som kunne gått andre steder. Siden ETC var mest interessert

i backenden av prosjektet burde vi redusert unødvendig arbeid på frontend. Vi kunne ha implementert inn Bootstrap som ville hjulpet oss med både utseende, funksjon og oppbygningen av siden for å sette av mer tid på resten av prosjektet. Vi fikk ikke brukertestet applikasjonen pga. korona. Med brukertester kunne vi fått tilbakemeldinger på potensielle mangler med brukergrensesnittet vårt, og muligheten til å tilpasse det etter vaktleder sine ønsker.

8.2 DevOps verktøy

Grunnlaget for valg av et slikt verktøy startet med at vi ønsket å være på samme platform som ETC. Dette var fordi vi ønsket at de skulle ha lett tilgang på prosjektet underveis og samtidig kunne se produktkøen. Dermed fikk vi tilgang på Azure DevOps, men ikke tilgang til alle funksjonene med en gang. ETC gjorde en prioritering på å sette opp sin egen versjon av Gitlab istedet, etter ønske fra andre bachelor-grupper. Dermed ble vi invitert til Gitlab istedet, prosessen med å få alle inn tok litt tid. Vi gjorde en kjapp prioritering på å overføre alt til NTNU sin Gitlab. Det ferdige prosjektet skal lastes opp til ETC sin Gitlab når vi er ferdige. Fordelen ved å bruke Gitlab over Azure DevOps er at vi har mye erfaring fra Gitlab, og kan dermed sette opp pipelines raskere og være generelt tryggere i arbeidet.

8.3 Frost API

Da vi startet prosjektet, undersøkte vi og fant ut at Frost APIen til Met var en god kilde for historiske værdata. Det så ut som at den hadde all værdata vi trengte, for å kunne bruke den som læringsdata til maskinlæringen vår. Det første vi gjorde i starten av prosjektet, var å begynne å implementere henting av data fra Frost. Vi hadde fått laget funksjon for henting av data fra APIen. Vi lagret det i databasen og hadde mulighet til å hente den for internt bruk i systemet. Vi oppdaget at værdata kun kom fra værstasjoner og ikke fra koordinater. Vi sjekket hvor de nærmeste værstasjonene var i forhold til Gjøvik. Det var to stasjoner i nærheten, men de var fortsatt for langt unna til å gi relevant data for Gjøvik. Datapunktene vi fikk ut av Frost API samsvarte heller ikke med alle datapunktene fra Met API. Det å bruke Frost API som treningsdata til klassifiseringen var derfor ikke mulig, siden den ikke får sammenliknet det med nye værmålinger.

Vi burde ha undersøkt hvilke data Frost APIen leverte og hvor den kom fra. Hadde vi sett dette tidligere ville vi spart mye tid. Funksjonaliteten for Frost APIen ligger fortsatt i prosjektet, om ETC skulle finne en bruk for den senere, som f.eks grupperingsalgoritmen.

8.4 Testing

Vi lagde ingen tester for systemet. Det hadde spart oss tid i lengden å lage tester enn å sjekke det manuelt. Det ville også gitt en oversikt over om alle delene kjører som de skal.

8.5 Arkitektur og Rammeverk

Vi brukte mye tid og energi på å finne ut hvordan systemet skulle struktureres og hvilke rammeverk den skulle bestå av. Tanken bak valgene var i starten basert på et ønske om å beholde så mye som mulig i et system. Altså at frontend og backend skulle være i det samme systemet.

Derfor startet vi med å implementere Thymeleaf[1] med Spring. Vi fant fort ut at vi kom til å få mye nytte av ferdiglagde komponenter og andre moderne frontend metodikker. Dermed skrapte vi Thymeleaf.

Vaadin var vårt neste steg som tillot bindinger mellom Java og litElements. LitElements er et rammeverk til node.js som tillater komponent-basert webutvikling. Dette virket som et solid valg ettersom Vaadin tilbyr verktøy for å kjapt og enkelt sette opp brukergrensesnitt. Samtidig kunne man manipulere sidene fra Java koden i spring ved bruk av bindingene. Vi fant ingen dokumentasjon på hvordan man kunne sette opp egne komponenter på en god måte. Dermed falt denne også bort.

Vi endte med å benytte et moderne rammeverk som heter Vue.js. Denne tillater i likhet med LitELements å lage komponenter til nettsider. Vi har hatt erfaring med LitElements tidligere, men etter litt utforskning fant vi ut at Vue virker lettere å implementere. Dette var det siste leddet som bestemte hvordan systemet skulle bli delt opp. Vi er fornøyde med valget vårt siden Vue er god på å manipulere data for å få det til å passe en forventet oppbygging.

Frontend og backend ble dermed adskilt, som skapte en naturlig klient/tjener arkitektur.

8.6 Prosessen

Iløpet av bacheloroppgaven har vi fått erfart at det er vanskelig å skulle lede og organisere et prosjekt samtidig som man utvikler det. Det er mye arbeid som går

[1] Malmotor for java: https://www.thymeleaf.org/index.html

inn i begge deler og vi merket hvor viktig prosjektplanen vår var, og støttet oss på den når det dukket opp usikkerhet rundt veivalg.

Med vår smidige arbeidsmetodikk var vi forberedt på at det ville komme endringer, dette førte til at vårt originale tidskjema ikke lenger passet. Etter en stor endring i kravspesifikasjonen kunne vært lurt å ta seg tid til å endre på milepælene, slik at den tilsvarer nåværende prosjektkrav. For å kunne passe planen til det de nye endringene og få maksimert tidsbruken. Samtidig føler vi at vi har vært flinke til å jobbe smidig og endre jobbingen etter planendringene.

Det vi kommer til å ta med oss videre er å lage en god plan før man starter et prosjekt. Det er også viktig å være forberedt på at det kan bli store endringer i planen, da kan det være greit å oppdatere den.

8.7 Tid

Nå som vi er i sluttfasen av bachelor oppgaven, kan vi se tilbake og si at vi underestimerte hvor lang tid det ville ta å utvikle maskinlæringen for områdedefinering. Vi hadde opprinnelig estimert at den skulle bli ferdig i første milepæl, men endte opp med å bli i siste. Grunnen til dette var at vi opprinnelig skulle ha en enkel måte å dele opp i områder på, men ble enige med ETC at vi skulle ha en smartere løsning på det. Her merket vi hvor viktig det var å sette opp avgrensninger på prosjektet slik at vi fikk prioritert riktig.

Kapittel 9

Videreutvikling

Her vil vi gå igjennom arbeid vi mener er viktigst for å videreutvikle prosjektet. Siden det er backenden som kommer til å bli tatt i bruk av ETC videre, er det dette vi har fokusert på. Vi har satt opp et godt rammeverk for maskinlæringen, for å utvikle denne videre går det mest ut på å finjustere variabler.

9.1 Klassifiserer

Det er mange parametere og variabler man kan justere i klassifisereren for å gjøre den mer treffsikker, fleksibel for feildata eller mer generaliserende (ikke for spesifikk). Hvis beslutningstreet er for detaljert vil det bestå av unødvendig mange steg og attributter som ikke nødvendigvis har noe med beslutningen å gjøre. Man kan da beskjære treet, slik at man reduserer antall grener, og vil bare bestå av de grenene med mest data til å støtte det opp. På den andre siden, hvis treet som blir bygget er kort så kan man justere treet slik at det bygger et mer avansert et. Dette gjøres ved å sende med alternativer til treet i en liste med strenger. Her skal vi presentere de variablene[15] vi mener er viktigst å se på for videreutvikling av klassifisereren når man har fått inn mye og god treningsdata:

Selvtillits Faktor -C <float>
Denne er satt til 0.25 som standard og bestemmer hvor mye treet skal beskjæres. Hvis man er sikker på at treningsdata både er god og representerer alle mulige scenarier kan man sende med et høyt tall, som vil gjøre at treet blir dypere og mer spesifikt.[1] Lave tall vil gjøre at klassifisereren beskjærer mer av treet. Denne variabelen påvirker hvordan treet blir i stor grad og vil være en viktig del av

[1] https://www.schankacademy.com/demos/data-analytics/xt/lib/docs/0/j48_parameters.pdf

finjusteringen av klassifisereren.

Minimum nummer av objekter -M <int>
Denne definerer hvor mange instanser av data det må være før det kan bygges en beslutningsgren. Dette er en god måte å passe på at alle grenene som blir skapt har nok data bak seg til å begrunne det. Hvis denne er for lav kan det komme data i beslutningstreet som ikke nødvendigvis har noe med utkallingen å gjøre.

Antall deler -N <int>
Klassifisereren tar deler at treningsdataene og bruker det til å bygge treet og en mindre del av dataene til å beskjære ned på treet der dataene ikke samsvarer. Denne variabelen bestemmer hvor stor andel av dataene som skal brukes til å beskjære treet. Denne er satt til tre som standard, som betyr at en 1/3 av dataene blir brukt til å beskjære og resten til å bygge opp, setter man den til fem blir 1/5 av dataene til beskjæring osv. For å redusere unødvendige detaljer i treet kan man sette denne til en høyere verdi for å få mer av dataene til å bygge opp treet.

9.1.1 Datapunkter

Man kan se på flere datapunkter å legge til i klassifisereren som kan hjelpe med å gi en smartere og mer detaljert anbefaling. Et datapunkt å prioritere til videreutvikling er trafikkmeldinger, slik at anbefalingen kan prioritere veiene med mest trafikk (eller finne ut at det ikke er nødvendig siden det er så mye trafikk). Dette vil også bidra til viderutvikling av ny anbefaling for grusing av vei, siden man ikke vil gruse veier med mye trafikk. Vegvesenet sin DATEX-II API leverer trafikkmeldinger for ulykker og trafikkregulering[2] og kan gi en indikasjon på hvor det burde sendes ut til vedlikehold oftere.

9.1.2 Hoeffding

Denne algoritmen for klassifisering er verdt å se nærmere på når treningsdata øker. Den bruker en teknikk som verifiserer hvor viktig hvert attributt er, som gjør at den yter bra på store mengder data.

[2]https://www.vegvesen.no/om+statens+vegvesen/om+organisasjonen/apne-data/datex/publikasjoner/trafikkmeldinger

9.2 Områdedefinering

Det kan være en del arbeid videre med grupperingsalgoritmen. For det første burde man analysere tettheten av datapunktene, hvorvidt de trenger å ha kort distanse mellom, eller lengre. Vi har gjort en antagelse om at forskjeller i geografi og værdata kan utgjøre forskjell på veiarbeidet, men om disse rekker spille mye inn på et område på størrelse med Gjøvik kommune vet vi lite om. Det kan hende at disse områdene vi definerer egentlig inneholder så små forskjeller at de ikke rekker utspille seg i en reell beslutning. En annen mulighet er også å finne ut hvilke parametre som passer best inn i algoritmen for å få best resultat ut. Vi har f.eks to distanse verdier `eps1` og `eps2` som representerer først geografisk-distanse og så attributt-distanse. Foreløpig er `eps1` satt til 0.05 som viste seg å gi greie resultater. Samtidig har `eps2` en verdi på 2, som også viste seg å gi resultater der vi fikk greie grupperinger. Disse tallene finnes det lite annet grunnlag for enn at de har gitt et resultat. Dermed burde en analysere i hvor stor grad disse er nyttige. Det samme gjelder `MinPts` og `rep_MinPts` som bestemmer antall naboer et punkt må ha for å ikke bli markert støy. Endring av alle disse variablene vil endre formen og antallet grupperinger som blir produsert. Det finnes også en rekke potensielle forbedringer av algoritmen, den vi implementerte kan vise seg å være svak på enormt store dataset. Disse forbedringene kan være, å implementere partisjonering av grupperingen. Der en kan gruppere hver partisjon og deretter slå de sammen. En slik implementasjon finnes allerede i prosjektet vårt, men vi rakk ikke implementere denne med FSDBSCAN.

Kapittel 10

Arbeidsevaluering

I kapittel 5.2.2 nevnte vi hva vi gjorde som følge av korona-situasjonen, her kommer vi til å gå mer i dybden på hvordan disse valgene har påvirket oss og hva vi har fått ut av det. Vi vil også gå over hvordan vi som gruppe har samarbeidet, hva vi er fornøyde med og hva vi kunne gjort bedre.

Vi har samme utdannings-bakgrunn men har forskjellige interesser og styrker som utfyller hverandre godt. Vebjørn jobbet med områdedefinering (som ble en mye større del enn vi hadde forventet) og hjalp til med backend, Hans Kristian Jobbet backend og hjalp til frontend mens Emil tok seg av frontend og klassifiseringen. Denne oppdelingen av arbeidet gjorde at alle fikk jobbet med det de følte seg tryggest på, det ble god flyt i arbeidet og vi slapp å vente på at andre skulle implementere sine deler. Maskinlæringen var det eneste området ingen av oss hadde erfaring med. Det hadde vært til stor hjelp å hatt med en fra data ingeniør til denne delen av prosjektet, men likevel lærte vi fort og fikk løst maskinlæringsdelen på en god måte. Vi har jobbet sammen på gruppe i flere prosjekter tidligere og har bidratt til godt samarbeid ut prosjektet

10.1 Hjemmekontor sammenliknet med arbeidsplass

Vi savnet det å ha en egen arbeidsplass og merket at det var vanskelig å ikke ha et klart skille mellom soverom/fritidsrom og arbeidsrom. Det førte til at noen dager ikke ble like produktive som de kunne ha blitt og vi var generelt mer slitne og reduserte enn vanlig. I denne perioden merket vi hvor viktig tidsskjemaet vårt var og hvordan det drev oss til å jobbe videre når motivasjonen ellers var lav. Vi holdt sosiale arrangementer(over nett) innad i gruppen utenfor arbeidstiden for å øke lagånden. Dette førte til bedre kommunikasjon og humør på alle, og var nødvendig i en ellers repetitiv hverdag.

I et grupperom har man mulighet til å diskutere med hverandre og idémyldre på tavlen. Dette merket vi spesielt under planleggingen og prosjektet, da vi kun jobbet over nett de første 3-4 månedene. Det er også lettere å spørre om hjelp uten å avbryte noen i en god flyt eller tilby å hjelpe noen som man ser sitter fast.

Vi fikk jobbet på skolen de siste ukene av prosessen og merket store forbedringer av å jobbe sammen. Alle møtte opp til arbeid opplagte og klare for arbeidsdaget og både motivasjonen og psyken ble bedre etter at vi fikk komme oss ut av huset. Samarbeidet og planleggingen av rapporten fikk en helt annen flyt og fortgang og vi klarte å legge fra oss arbeidet på slutten av dagen.

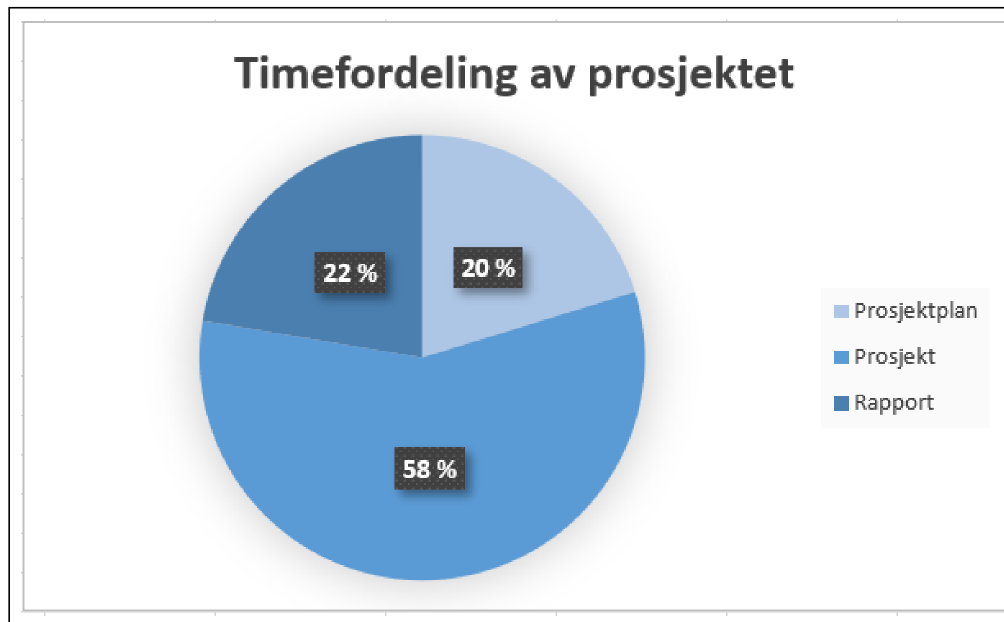
Korona situasjonen har ført til utfordringer for gruppen vår. Flere av oss har havnet i karantene iløpet av prosjektet. Dette har ført til at vi var nødt til å endre på planene våres. Det har vært flere ganger hvor vi hadde planlagt å møte på skolen for å jobbe, men endte med å jobbe over nettet grunnet at en av oss hadde blitt satt i karantene.

Vi i gruppen hadde et ønske om å få dratt på besøk til ETC iløpet av prosjektet, for å kunne vise og diskutere oppgaven. Dessverre fikk vi ikke muligheten til å gjøre det, på grunn av sånn korona situasjonen var.

ETC holder til rett ved campus men vi fikk ikke hatt noen møter ansikt til ansikt med dem pga. smittevern.

10.2 Arbeidstimer

Gjennom prosjektet har vi for det meste klart å holde tidsplanen vår. Det er enkelte uker hvor vi har jobbet mer enn planen, også er det enkelte uker hvor vi har jobbet litt mindre en planen.



Figur 10.1: Diagram over timefordeling

Vi i gruppen har totalt brukt 1779,5 timer på bacheloren fra start til slutt. Av disse timene har 360 av dem gått på prosjektplan og utforsking. 1022 av timene har vi brukt på å utvikle produktet vårt. De resterene 397,5 timen har gått på å skrive rapporten. Over en periode på 19 uker gir 1779.5 timer et gjennomsnitt på 31,2 timer per person i uken. Ut i fra planen vår om å ha 30 timer jobbing i uken, har vi klart å holde målet. Emil hadde totalt 587,5 timer, Hans Kristian hadde total 600,5 timer og Vebjørn hadde totalt 591,5 timer arbeid.

Kapittel 11

Konklusjon

Målet for denne oppgaven var å lage en smart og effektiv løsning for å hjelpe vaktleder med å gjøre beslutninger på utkalling av veivedlikehold. Vi har satt opp et godt rammeverk for ETC, som de kan bruke i deres applikasjon for å gi støtte til vaktleders beslutning. I applikasjonens hovedside får vaktleder opp kart og dashbord med alle områdene til en kommune. Her kan vaktleder få opp værdata, kameraer og en værprognose som gir en anbefaling på vedlikehold tilknyttet området. Mye av arbeidet gikk inn i grupperingsalgoritmen, som ble en større del enn vi først hadde planlagt. Områdene blir generert av vår grupperingsalgoritme for å best mulig passe anbefalingen som blir gjort. Ved å dele opp områdene slik, får vaktleder en liste over veiene i området knyttet til hver anbefaling. Vi vinklet oppgaven vekk fra funksjonaliteter som allerede var eksisterende i CarAdmin som utkalling, publikumskart og avvikslogg for å rekke å utvikle maskinlæringen for områdedefineringen og vedlikeholdsanbefalingen. Ingen i gruppen hadde erfaring med maskinlæring før vi startet dette prosjektet, og det har vært en bratt læringskurve å utvikle både klassifisereren og områdedefineringen. Vi er fornøyde med vinklingen vi gjorde på oppgaven, den treffer godt på å bidra til beslutningsstøtte for en vaktleder.

Bibliografi

[1] A. Zhou, S. Zhou, J. Cao, Y. Fan og Y. Hu, «Approaches for scaling DBSCAN algorithm to large spatial databases,» *Journal of Computer Science and Technology*, årg. 15, nr. 6, s. 509–526, nov. 2000, ISSN: 1000-9000, 1860-4749. DOI: 10.1007/BF02948834. adresse: http://link.springer.com/10.1007/BF02948834 (sjekket 12.05.2021).

[2] (). «Design Patterns | Object Oriented Design,» adresse: https://www.oodesign.com/ (sjekket 16.05.2021).

[3] *Use weka in your java code - Weka Wiki*. adresse: https://waikato.github.io/weka-wiki/use_weka_in_your_java_code/#traintest-set (sjekket 15.05.2021).

[4] J. Brownlee, *How To Use Classification Machine Learning Algorithms in Weka*, en-US, jul. 2016. adresse: https://machinelearningmastery.com/use-classification-machine-learning-algorithms-weka/ (sjekket 15.05.2021).

[5] *Frost API*. adresse: https://frost.met.no/index.html (sjekket 18.05.2021).

[6] *Locationforecast*. adresse: https://api.met.no/weatherapi/locationforecast/2.0/documentation (sjekket 18.05.2021).

[7] D. Sinha. (). «Difference between K-Means and DBScan Clustering - GeeksforGeeks,» adresse: https://www.geeksforgeeks.org/difference-between-k-means-and-dbscan-clustering/ (sjekket 12.05.2021).

[8] D. Birant og A. Kut, «ST-DBSCAN: An algorithm for clustering spatial–temporal data,» *Data & Knowledge Engineering*, Intelligent Data Mining, årg. 60, nr. 1, s. 208–221, 1. jan. 2007, ISSN: 0169-023X. DOI: 10.1016/j.datak.2006.01.013. adresse: https://www.sciencedirect.com/science/article/pii/S0169023X06000218 (sjekket 12.05.2021).

[9] N. Sankaranarayanan, *snithish/DM-DBScan*, original-date: 2019-11-10T10:05:04Z, 1. des. 2019. adresse: https://github.com/snithish/DM-DBScan (sjekket 18.05.2021).

[10] A. Guttman, «R-trees - a dynamic index structure for spatial searching,» s. 11,

[11] D. Moten, *davidmoten/rtree*, original-date: 2014-08-26T12:29:14Z, 6. mai 2021. adresse: `https://github.com/davidmoten/rtree` (sjekket 12.05.2021).

[12] (13. jul. 2013). «Convex hull | set 1 (jarvis's algorithm or wrapping),» GeeksforGeeks. Section: Geometric, adresse: `https://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/` (sjekket 18.05.2021).

[13] *Konveks omhylning*, i *Wikipedia*, Page Version ID: 18384466, 3. apr. 2018. adresse: `https://no.wikipedia.org/w/index.php?title=Konveks_omhylning&oldid=18384466` (sjekket 18.05.2021).

[14] *google/gson*, original-date: 2015-03-19T18:21:20Z, 19. mai 2021. adresse: `https://github.com/google/gson` (sjekket 19.05.2021).

[15] *J48 (weka-dev 3.9.5 API)*. adresse: `https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/J48.html` (sjekket 19.05.2021).

Tillegg A

Kontrakt

NTNU

Norges teknisk-naturvitenskapelige universitet

Vår dato Vår referanse

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

Electric Time Car

Dag Solhaug (oppdragsgiver), og

Vebjørn Fonstad Leirod

Hans Kristian Hoel

Emil Johannes Tillmann Hegdal (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 11.01.2021 til 20.05.2021 .

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon, reiser og nødvendig overnatting på steder langt fra NTNU i Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle beståtte bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv NTNU Open.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

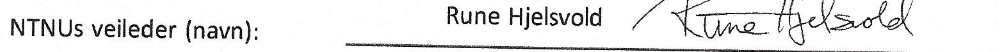
9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn): Rune Hjelsvold



Oppdragsgivers kontaktperson (navn): ______________________

Student(er) (signatur): ______________________ dato 19.1.2021

______________________ dato ________

______________________ dato 19.01.2021

______________________ dato 19.01.2021

Oppdragsgiver (signatur): ______________________ dato 21.01.2021

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): ______________________ dato ________

Tillegg B

Logger og referater

Referat fra planmøte for uke 3

- Skrive ferdig prosjekt avtalen til onsdag 20.01-2021
- Starte forfult på prosjekt planen
- Neste uke skal vi starte med væranalyse
- Vi skal forberede størsmå og informasjon til veilednigsmøte med veilederen vår Rune
- Vi skal gjøre ferdig prototypen for prosjekte og vise den for klienten vår ETC på møte på onsdag
- Alle skal lese gjennom en tidligere bacheloroppgave fram til mandag 25.Jan

Veiledningsmøte 19. Jan 2021

- Avgrensning - hvis oppgaven blir for stor så avgrense for å få tid til å gjøre andre ting
  - Potensiell avgrensing - Maskinglæring
  - Litt fortidelig nå å sette avgrensing
  - Må vurdere bruk for avgrensing underveis / senere ut i prosjektet
- Vi må diskutere i rapporten om avvik
- Hvis man ender med endringer eller andre oppdateringer i forhold til planen må det dokumenteres i rapporten
- Risiko:
  - Gjort dårlig valg tidligere
  - Valgte API-er svikter
  - Ting vi ikke lett kan fikse
  - Teknologi problemer
  - Tenke gjennom og være forberedt på det som kan skje

    - Der det er stor risiko - Kanskje komme med tidelig alternativ løsning på det

- Skille mellom A, B og C er analytiske evne
    - Tenke å sammenligne forskjellige ting selv
    - Ikke bare det oppdragsgiver sier vi skal se på
    - Tenke selv

- Hvordan får vi vist mest mulig bredde på kompetanse
    - Blir maskinlæring for komplisert?
    - Gjerne få vist mye forskjellige kompetanse
    - Ikke la maskin læring ta for mye tid av prosjektet
    - Viktig å ha forskjellige temaer / kunnskaper for å vise bredde

Oppsummering av uke 3

- Vi har blitt ferdig med prosjektplanen
- Vi har fått levert prosjekt avtalen
- Vi har oppdater og ferdigstilt prosjektplan etter veileders innspill
- Vi har startet på prototypen

Referat fra planmøte uke 4

- Ferdigstille prototypen
- Ferdigstille docker
- Få satt opp serveren
- Begynne på client siden
- Starte på henting av data fra API
- Monolitic aproche

Referat fra møte med ETC 25. Jan

- Om applikasjonen
    - Den skal kunne vise fremtidig værvarsler
    -

- Vi forklarte hvorfor vi har valgt å slå sammen kart og dashbord i en side isteden for å ha de på to forskjellige sider

Veiledningsmøte 26. Jan 2021

- Vi burde ha flere mileperler
- Mileperlene vår trenger å være mer utdypende
- Hvilke endringer kan vi gjøre for at vi kommer i mål

- Skrive litt om ETC, skrive litt om ut fordringer
- Skrive om effektmålet
- Vise at vi har skjønt det oppdragsgiver vil
- Hva er det realistisk at vi leverer?
- Vi bør være mer utdypende på kartet
- Hva vi kaller metodikken er ikke så viktig, Det er hvordan vi beskriver det
- Tydeliggjør hva vi skal gjøre i milepælene. Ikke gjøre de for store
- Ha gode tester der det er mye logikk
- Si mer om standarder vi skal følge
- Viktigste med prosjektplanen er at den hjelper oss med å må få et godt prosjekt

Referat fra møte med ETC 8. Feb

- Vi skal droppe publikumskart, rodeoversikt og utkalling
    - Rodeoversikt og utkalling kan hentes fra eksisterende applikasjon
- Vi skal lage læringsdata selv
- Kode API-er sånn at de er lette å vidreutvikle
- Mileperle 2 - utveksle data mellom oss og ETC
- Bruke Maria DB
- Lage en bestilling på data vi trenger fra ETC

Veiledningsmøte 16. Feb 2021

- Det vi trenger å gjøre er å diskutere hvordan vi forholder oss til endringer
    - Henvise til møteprotokoller / endringsvalg
    - Ikke nødvendigvis skriv alle veivalg i rapporten
    - Skriv de store veivalgene
    - Henvis til de mindre
    - Vise hvordan vi har holdt oss til planen
    - Vise krav spec og change log
    - Dokumentere men ikke nødvendigvis i sluttrapporten
    - Mye i notatene
- Diskuter i rapporten om hvorfor vi endrer tjenester
- Trenger ikke nødvendigvis å modellere store om råder over hvor det ikke er veier
- Kategorisere områder
- Lite å bekymre seg for der det er få folk/ en vei
- Sjekke om veier er lange
- Gjøre en sjekk om det er noe vei på område
- Ivar Farup driver med bilde behandling og algoritmer – kan være smarte å snakke med han

- På et eller annet tidspunkt er det kanskje nødvending med plan endring?

Referat fra planmøte uke 8

- Hva har vi fått gjort:
    - Jobber fortsatt med områdedefinering
    - Databasen er fikset
    - Ryddet opp i prosjektet
- Hva skal vi gjøre vidre
    - Jobbe videre med områdedefinering
    - Hente data fra veivesenet
    - Utkalling
    - Sende krav til ETC om ressursser

Veiledningsmøte 23. Feb 2021

- Erling er en geomatiker vi kan høre med
- Vi skal gå videre med område inndeling
- Bildeprosesseringene kan bli ganske tunge
- Kanskje endre oppøsing for å gjøre det lettere
- Se hvordan vi ligger an i forhold til planen om noen uker
- Skrive et refleksjonsnotat, Vi bør begynne å tenke på:
    - Hvordan gruppa vår jobber?
    - Når er produktiviteten best?
    - Hva er det som skal til for at jeg klarer meg?

Referat fra møte med ETC 1. Mars

- Vi trenger ikke å jobbe med rettighetsmodulene, for det er en oppgave i seg selv
- Vi skal legge opp til at det er lett å legge inn rettigheter senere
- Vi skal få test data fra det de har allerede

Veiledningsmøte 9. Mars 2021

- Vi må diskuter og beskrive godt for veivalgene vi har gjort
- Er greit å få ned småpunkter og utkast til rapporten underveis så vi ikke glemmer det
- Vi bør få fram alt arbeid i rapporten. Skrive om arbeidet rundt databasen og maskinlæring
- Kan være interessant å skrive om hva og hvordan casing funker i rapporten

  - Kan være interessant for rapporten
  - Rune mener database løsningen vår ligner på en casing løsning
- Hva skal cashes, hvor lenge man skal bruke det og hvor lenge man lagrer det
- Forklare hvorfor vi byttet fra Azzure til GitLab
  - Årsakene til valg av dette
  - Pipelines
  - Rettigheter
  - Tilgang
- Områdedefinering er noe av det støreste med prosjektet
- Legger områdedefinering i milepæle 3
  - Litt endring på feilvurdering av plan
- Klient og tjener utvikling, slå de sammes så tidelig som mulig
- Høre med etc hva slags kriterier det er for sortering?
- Si noe om sprint oppsettingen i sluttrapporten
- Skrive opp litt i rapporten etter mileperle 2 ‘
  - Hovederfaringer
  - Kunnskaper
  - Prosess
  - Erfaringer

Oppsummering av uke 10

- Vi har startet på dashbordet
- Vi har fikset kommune inndeling
- Fått satt opp forskjellige sider
- Jobber med å fikse request systemet
- Jobber med å prøve å korte ned singelton filene
- Har laget RestApi-er
- Har fikset Api call
- Har fikset og oppdater databasen og database diagramet
- Har skrevet litt på rappoten

Referat fra planmøte uke 11

- Få løst en del småproblemer
- Fikse et interaktive kartet
- Begynne å samle data til maskinlæring
- Få unnagjort endel oppgaver

Referat fra møte med ETC 22. Mars

- Implementere å kunne legge til kamera for kunder
  - URL til kamerar
  - Kunne legge in kamerar som viser veier
  - Vise iste over kamerar
- Kan legge inn kravspesifikasjon i rapporten, men send til etc så de kan se over den

Veiledningsmøte 23. Mars

- Hvis leseren sitter med inntrykk av dårlig norsk får de dårlig inntrykk
- Må skrive om hvordan vi har tenkt/ hvorfor vi har det
- Prøv å anstrenge oss til å skrive mer
- Sensor vei ikke mer enn de som står i rapporten
- Fjerne unødvendige fraser
- Skriv kompakt
- Husk kommaregler og orddeling
- Må innlede med noe om hva ETC er
- Effektmål er det overodnetet målet
- Kan kopiere fra prosjektplan
- Se lit på arkitekttur
- Illustrere Request Queue
- Forklare hvorfor vi ønsket å gjøre noe

Veiledningsmøte 13. April

- Få noen til å gå over norsken
- Kan irritere folk hvis det blir for mye feil
- Effektmål er litt for prosjektrettet
- Putte problemmålet på toppen
- Avgrensning
- Må velge hva vi trenger å gå inn i dybden på
- Utviklingsprosess
  - Hvordan vi har jobber
  - Noen av teknologivalgene er arkitektur

Veiledningsmøte 4. Mai

- Skrive om utviklingsprosessen
- Se på beskrivelsen av læringsutbytte
- Skrive mer om maskinlæring i implementering
- Drøfte hva vi lørte og erfrate
- Hva er hovedkonklusjonen

- hva er vi mest stolte av, både produkt og prosess
- Stort mulighetsrom
- Vær mer konkret på use casene
- Arkitekturdiagram
- Det vi bruker plass på må være noe leseren kan skjønne
- Sensor trenge å få en føles av hvor mye vi har gjort

Veiledningsmøte 12. Mai

- Må være lett å les, Kunne få god lesefart
- Skrive om klassifisering tidligere
- Forklare tall
- Skrive mer om eksempel
- Ligger mye verdi i maskinlæring
- Få arbeide tydelig fram
- Få fram den store insikten
- Skrive om feil ved anbefaling
- Bruke use case til å kommunisere hva vi har gjort
- Vi er ekspertene på hva vi har gjort

Tillegg C

Prosjektplan

DEPARTMENT OF COMPUTER SCIENCE

BACHELOR

Prosjektplan

Forfatter:
Hans Kristian Hoel
Vebjørn Fonstad Leiros
Emil Johannes Tillman Hegdal

Januar 18, 2021

Innholdsfortegnelse

1 Begreper **1**

2 Mål og rammer **1**

2.1 Bakgrunn 1

2.2 Prosjektmål 1

2.2.1 Resultatmål 1

2.2.2 Effektmål 2

2.3 Rammer 2

3 Omfang ved programvareutvikling ved øvrige oppgaver **2**

3.1 Fagområde 2

3.2 Avgrensning og foreløpige teknologivalg 3

3.2.1 Maskinlæring 3

3.2.2 Kart 3

3.2.3 Publikumskart 3

3.2.4 Innsamling av data 3

3.2.5 UI-Design 4

3.2.6 Rammeverk 4

4 PROSJEKTORGANISERING **6**

4.1 Ansvarsforhold og roller 6

4.2 Rutiner og regler i gruppa 6

5 ORGANISERING AV KVALITETSSIKRING **7**

5.1 Dokumentasjon, standardbruk og kildekode 7

5.2 Risikoanalyse (identifisere, analysere, tiltak, oppfølging) (Teknologisk, Forretningsmessig, Prosjektgruppemessig) 8

5.2.1 Kompleksitet ved Dependencies 8

5.2.2 APIer fra tredjeparter endres/svikter 8

5.2.3 Gruppen blir utsatt for sykdom 8

6 PLANLEGGING, OPPFØLGING OG RAPPORTERING 8
6.1 Hovedinndeling av prosjektet og valg av SU-modell 8
6.1.1 Milepæler 9
6.2 Plan for statusmøter og beslutningspunkter i perioden 12
7 Gantt-Skjema over prosjektplan 12

1 Begreper

ETC - Electric Time Car, Oppdragsgiver
Veiarbeid eller Vedlikehold av vei - Her menes brøyting eller salting av vei.
Met - Meteorologisk Institutt

2 Mål og rammer

2.1 Bakgrunn

Smart Vinterveg er et prosjekt mellom Mjøsbyene for å finne løsninger for vinterdrift av veger og eiendommer. Det er flere selskaper som kjemper om innovasjonskontrakten, og blant disse er ETC. ETC utvikler og drifter admininistrasjonsverktøyet Caradmin[1]. Denne applikasjonen tilbyr kjøretøysoppfølging som vedlikehold- og skadeoppfølging, elektronisk kjørebok, nøkkelhåndtering og veivedlikehold.

Som den del av prosjektet har vi følgende avsnitt som beskriver grunnlaget for prosjektet: "*En vaktleder som har ansvaret for å sette i gang veivedlikehold på vinterveier trenger et beslutningsgrunnlag å handle etter. Eks er det meldt underkjølt regn mellom 03 og 04 i Gjøvik. Må saltbilene begynne saltingen allerede fra kl 23 kvelden før for å rekke å salt før det underkjølte regnet inntreffer for å forhindre såpeglatte veier.*"

2.2 Prosjektmål

Oppgaven går ut på å lage en applikasjon som skal gi en anbefaling basert på værdatakilder til en vaktleder. Anbefalingene som gis vil bestå av brøyting eller salting, viktigste oppgaver sorteres først. Det vil, ifølge oppdragsgiver være brøyting.

Det er viktig at anbefalingene som gis har mulighet for å lære seg å bli mer presis over tid. Den skal kunne anbefale etter tidligere kjente tilfeller der værmelding tok feil og etter presiseringer fra diverse innsendte raporter/avviks-raporter.

I tillegg til denne grunn funksjonen skal vi kunne vise diverse statistikk på været og forhold som er målbart til vaktleder. Denne statistikken skal ha et eget dashbord, men et kart som viser nåværende situasjon er også aktuelt. Samtidig kunne gi vaktleder en enkel mulighet for utkalling av kjøretøy.

2.2.1 Resultatmål

Prosjektets utvikling skal samsvare med Milepælene vi har satt i 6.1.1. Med dette skal prosjektets sluttfase inneholde en webapplikasjon tilknyttet en analysemotor som leverer beslutninger til vaktledere.

[1] https://caradmin.no/

Web applikasjonen skal inneholde: beslutningskart, utkalling, rodeoversikt, vedlikeholdsprognoser, vær-prognoser, avvikslogg og publikumskart.

Web applikasjonen skal være brukervennlig. Det skal være lett for vaktleder å navigere, lese informasjon og gi utkallinger til sjåførene.

2.2.2 Effektmål

Applikasjonen skal automatisere deler av vaktleders arbeid ved analyse av områder. Ettersom applikasjonen automatisk sjekker områdene istedet for et menneske minker vi risiko for at vaktleder overser et område. Applikasjonen skal også anbefale vedlikehold der det ikke er åpenbart at det trengs. Tilsammen vil dette øke trafikkflyt for trafikanter og gjøre arbeidsdagen lettere for arbeiderne.

2.3 Rammer

Java: Siden Caradmin er skrevet i Java er det sterkt ønsket at vi skriver vår applikasjon i Java også slik at ETC kan videreutvikle den og eventuelt implementere den i Caradmin hvis ønsket.

Rettighetsmodul: Oppdragsgiver har tilgjengeliggjort en rettighetsmodul som vil være bakgrunn for tilgangskontrollen som applikasjonen skal inneholde. Omfanget av modulen har vi ikke fått oversikt over, dermed vil dette bli en diskusjonssak senere i prosjektet.

Utkalling: I prosjektbeskrivelsen står det at en utkalling skal skje via et "push-varsel". Dette er en funksjon som ETC allerede har implementert i Caradmin, som vår nettside skal kunne kalle på. Hvordan vi implementerer dette går vi igjennom med ETC senere inn i prosjektet.

3 Omfang ved programvareutvikling ved øvrige oppgaver

3.1 Fagområde

Vi tenker det faller naturlig inn med noe maskinlæring i en applikasjon som dette. Vi har ikke vært borte i mye maskinlæring iløpet av tiden våres her på NTNU, så dette vil være bakgrunn for ny-læring hos oss.

Mye av prosjektet består ennå av Web-Utvikling som er noe vi har vært en del borti. Vi skal lage en brukervennlig applikasjon som viser til dataene som analysemotoren / maskinlærings-algoritmen kommer frem til. Det som blir nytt for oss her er det interaktive kartet som skal implementeres.

3.2 Avgrensning og foreløpige teknologivalg

3.2.1 Maskinlæring

Deler av oppgaven vår går ut på å hente og analysere værdata. Her skal vi kunne sammeligne værdata og ut i fra dette gi anbefalinger om hva som burde gjøres med veiene for å forhindre at de blir glatte. Dette kan bli en veldig stor og komplisert oppgave å utvikle. Spesielt når ingen av oss har noe særlig erfaring innen for maskinglæring. Derfor er vi nødt til å sette en begrensning på hvor stort og komplisert vi kan gjøre dette, for å få tid til å gjøre de andre delene av oppgaven. Hvor stor denne avgrensningen kommer til å vær vet vi ikke enda. Men det finner vi ut av senere i prosessen. Er vanskelig å si hvor komplisert og tid-krevene det kommer til å være nå.

3.2.2 Kart

Vi har sett på både Kartverktet og Openstreetmap sine kart API-er. Kartverket er en norsk tjeneste for levering av forskjellige typer kart og statistikker over Norge. Kartverket har forskjellige typer API-er som leverer forskjellige typer kart og egenskaper til kartet. Openstreetmap leverer også kart API-er. De har ikke bare kart for Norge men også for hele verden. Openstreetmap er åpen for alle, som gjør at de som vil har muligheten til å hjelpe til å utvikle programmet. Man har også muligheten til å legge inn nye sted og områder, for å videreutvikle kartes kunnskaper.

Både Kartverket og Openstreetmap er gratis tjenester å bruke. Vi har kommet fram til å bruke Kartverket for levering av kart. Dette er fordi Kartverket leverer et mer detaljert og nøyaktig kart over Norge, og siden Caradmin kun er laget for å brukes i Norge har vi ikke bruk for kart over hele verden. I tillegg har ETC kommet med et ønske om at Kartverket skal brukes da de mener dette er noe av det beste man får tilgang på for bruken, noe vi er enige i, etter å ha sett på alternativene.

3.2.3 Publikumskart

Publikumskartet skal vise mye av dataen som beslutningskartet viser, men siden denne skal være tilgjengelig for offentligheten er den strippet for all privat informasjon. I første omgang vil vi fokusere på å få ferdig funksjonene til vaktleder, så hvis det blir knapt med tid blir denne nedprioritert.

3.2.4 Innsamling av data

I oppgaven skal vi samle inn en god del data. Samtidig har vi fått en god del forslag på disse, som f.eks: Netatmo[2], Met, Vegvesenet[3], Kartverket og rapporter

[2]Et selskap som leverer små værstasjoner som kobles opp mot deres API. En del sanntidsdata ligger tilgjengelig, mens noe data krever at man eier stasjonen.

[3]Leverer værdata, reisetider, kamerabilder og trafikkmeldinger via et API

fra trafikanter eller brukere. Vi velger å prioritere enkelte av disse som særdeles viktig for applikasjonens grunnfunksjoner og det er disse vi kommer til å fokusere på i hoveddelen av prosjektet. De dataene vi anser som særdeles viktig er: værdata fra Meteorologisk Institutt, kartdata ifra Kartverket/Geonorge og rapporter fra brukere eller trafikanter.

Både Netatmo og Met gir oss sanntidsdata, mens Met gir også tilgang på spådd vær. En styrke ved Netatmo er at deres moduler kan gi oss tilgang på mer nøyaktig sanntidsdata og live bilde fra stasjonen. Denne styrken ser vi på som en mindre prioritert feature, men som absolutt kan legges til ved prosjektets senere faser dersom vi ser vi har tid til det.

Vegvesenet tilbyr mange datapunkter via DATEXII[4]. Denne API'en krever en søknad for tilgang. Planen er å sende en slik søknad slik at vi da også får tilgang på denne API'en. Ettersom den ikke umiddelbart er tilgjengelig vil vi støtte oss i hovedsak på offentlige API'er. I tillegg finner vi ikke dokumentasjon på at DATEXII formatet inneholder spådd vær, dette senker da prioriteten iht prosjektets grunnmål. De datapunktene vi har funnet i DATEXII som utskiller seg fra det vi allerede har tilgjengelig er: vegbanetemperatur, dugg- eller rimpunkts-temperatur, trafikkhastighet og diverse trafikkmeldinger.

Både Netatmo og Vegvesenet tilbyr sanntidstemperaturmålinger og vindmålinger samt også sanntidsbilder. Ettersom vegvesenet allerede har mange stasjoner ute langs norske veger blir denne API'en vektet over Netatmo dersom vi får tilgang på den.

Rapporter ifra brukerne og trafikantene ser vi på særdeles viktig ettersom dette blir hovedvektleggingen i maskinlæringsprosessen. Beslutningsstøtten skal ha mulighet for å anbefale en prosess der det ikke er så åpenbart at det kunne trenges. Disse rapportene vil da være med på å justere senere anbefalinger. Disse rapportene vil også fungere som redundans dersom maskinlærings algoritmen kommer med en dårlig anbefaling eller en vaktleder ikke har sett at et område trengte vedlikehold.

3.2.5 UI-Design

ETC har sagt at de ønsker at designet på applikasjonen skal baseres på utseende og brukeropplevelsen som applikasjonen deres Caradmin. Dette gjør at vi er ganske åpne på hvordan vi designer applikasjonen vår. Vi kommer da til å basere oppsettet av applikasjonen vår på deres UI-design.

3.2.6 Rammeverk

UI-Rammeverk Både Angular, React og Vue.js er kjente rammeverk innen utvikling av UI på nettsider. React er avhengig av å kompilere og kjøre i nodejs, noe vi ikke skal gjøre i vår applikasjon, så denne utgår og da gjenstår Angular og Vue. For å

[4]Format for veitrafikk meldinger, værdata og liknende. Brukt av Norge og store deler av europa

finne ut hvilken som passer best for å utvikle applikasjonen så leste vi oss opp på begge. Angular og Vue har mange likheter, men her fokuserer vi på forskjellene:

Angular er foreløpig det mest populære rammeverket og har dermed også mer dokumentasjon enn Vue. Angular er også til en viss grad mer fleksibel enn Vue og kan lettere skaleres siden den har mer prefabrikerte funksjoner og bruker Typescript, dette kommer på prisen av både størrelse og læringskurve. Siden Vue sparer tid ved å gjøre både design og API utviklingen lettere er det dette rammeverket vi går for.

Java Web-Rammeverk For vårt Java Web-rammeverk så vi på både Spring og JavaServer Faces(JSF) som gode kandidater. JSF er komponent-basert, som vil si at den bruker gjenbrukbare UI-komponenter som representer f.eks tekstfelt og knapper for å simplifisere design prosessen. Spring er handling-basert og lager klasser som har funksjoner som representerer handlingene til brukeren og kan derfor ta lenger tid å sette seg inn i. Vi har likevel valgt å bruke Spring, til tross for sin bratte læringskurve. Dette er fordi Spring gjør opp for det ved å simplifisere og effektivisere integrering av API og sikkerhet, noe JSF ikke gjør.

Java Template Engine Vi ønsker å bruke en template engine som er passende for Spring-rammeverket. Her har vi sett på to forskjellige rammeverk, dette er da Thymeleaf[5] og Groovy[6]. Prioriteten her er å finne et rammeverk som lar oss skrive på den måten vi er mest vant med. Groovy har egne syntakser for å skrive et HTML dokument, mens Thymeleaf har noe som ligner mer på vanlig HTML. Det er denne syntaksen og støtten for xml, html, javascript og css filer som gjør at vi da velger å bruke Thymeleaf i prosjektet. Thymeleaf ser også ut til å være bedre enn Groovy på følgende punkter: jar-fil størrelse, oppstartstid, raskere, bedre dokumentasjon og større bruksområde[7].

Maskinlærings-Rammeverk Her har vi sett på to forskjellige rammeverk som ser ut til å passe prosjektet bra. Dette er Shogun[8] og Weka[9], førstnevnte er et ML-rammeverk laget i c++ med et Java interface. Andre er et ML-rammeverk basert på Java. Sistnevnte inneholder GUI-funksjoner og diverse "lette" metoder for å starte en maskinlæringsprosess som gjør den mer aktuell for oss med relativt lite ML-erfaring. Ettersom sistnevnte virker mer brukervennlig for oss er det denne vi kommer til å fokusere på.

[5]https://www.thymeleaf.org/
[6]http://groovy-lang.org/templating.html$_{t}he_{m}arkuptemplateengine$
[7]https://springhow.com/spring-boot-template-engines-comparison/
[8]https://www.shogun-toolbox.org/
[9]https://www.cs.waikato.ac.nz/ml/weka/

4 PROSJEKTORGANISERING

4.1 Ansvarsforhold og roller

Vebjørn : Prosjektleder og teknologi-ansvarlig

I dette prosjektet skal Vebjørn ha ansvar som leder for gruppen, det vil si at han har overordna ansvar for møter, planer og annet relatert til prosjektet. Han vil da også ha ansvar som Scrum-master og lede sprint-møtene. Ettersom han skal lede disse får han også ansvar for å holde seg oppdatert på teknologien som er i bruk i prosjektet.

Emil : Design- og sikkerhetansvarlig

Emil har hoved-ansvaret for design av UI og implementering av applikasjonen. Han har også ansvaret for at koden følger gode sikkerhetsstandarder. Det aller viktigste er at applikasjonens sikkerhet samsvarer med oppgavebeskrivelse. Dette vil også si at Emil må gå igjennom koden jevnlig og gjerne også godkjenne pull-requests i git.

Hans Kristian : Oppfølging- og Dokumentasjonsansvarlig

Hans Kristian får ansvaret for oppfølging av kode-standarder og dokumentasjon. Dette vil si at han i hovedsak har ansvaret for at koden blir dokumentert riktig og at koden følger en felles syntaks slik at den er leselig for alle. Han har også ansvaret for å føre notater ved alle møter.

4.2 Rutiner og regler i gruppa

	Man	Tirs	Ons	Tors	Fre
8					
9	Planlegging	Veiledningsmøte	- Arbeid -		- Arbeid -
10	Rapport til Veiled	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -
11	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -	- Arbeid -
Pause 30 min	-	-	-	-	-
12	- Arbeid -	- Arbeid -		- Arbeid -	- Arbeid -
13	- Arbeid -	- Arbeid -		- Arbeid -	- Arbeid -
14	Møte ETC		- Arbeid -		- Arbeid -
15	- Arbeid -		- Arbeid -		Oppsummering
16					

Figure 1: Timeplan

Arbeidstid: Vi planlegger å jobbe 9-16 hver dag i uken der det er rom for det, dette er for å nå målet vårt om 30 timer i uken. Med planen våres har vi kun 25,5 time med fellesarbeid. Vi har satt som regel at dersom en ikke klarer å nå målete på 30 timer så må en selv ta igjen tiden på egen fritid. Vi har satt av lunsj kl 12 for å gi oss et avbrekk og tenkepause som vi mener er sunt i en slik arbeidsprosess. Timene loggføres i et Google spørreskjema, dette gjør at vi kan lett føre diverse statistikker i løpet av semesteret.

Arbeidsmetode: Vi har fått tilgang på Azure hvor vi planlegger å plassere arbeidsoppgaver. Disse legges inn som "Work items" hvor vi kan lett koble de opp mot bugs, tester, kommentarer og git-brancher. Work items kategoriseres etter status for

å gi en oversikt over oppgavene. Statuser vil være, "to-do", "active", "resolved" og "closed". "to-do" er oppgaver som ennå ikke er startet på, "active" er oppgaver som er under arbeid, "resolved" vil være oppgaver som er løst og avventer gjennomgang sist vil "closed" brukes til avsluttede punkter som er ferdig gjennomgått.

Vi vurderte andre systemer for å ha kontroll på backloggen til prosjektet, f.eks teams med oppkobling mot Azure og Trello med oppkobling mot git. Ønsket for å bruke teams kom fra at vi ønsker å samle mesteparten av informasjonen våres på ett sted for å unngå kompleksitet i "byråkratiet" vårt. Funksjonaliteten med å koble opp mot Azure var noe som ikke var tilgjengelig for våre brukere. Trello var også vurdert, men etter vi fikk tilgang til flere av Azure sine tjenester via oppdragsiver falt den også bak.

Rutiner for møter og planlegging kommer i neste seksjon.

5 ORGANISERING AV KVALITETSSIKRING

5.1 Dokumentasjon, standardbruk og kildekode

Alt av planlegging og møter skal bli dokumentert av dokumentasjonsansvarlig og bli lagt inn i teams-driven til gruppen på en oversiktlig og ryddig måte.

Kildekoden lagres i Azure Repos for å forsikre at alle på gruppen jobber med nyeste versjon av programmet, samt at ETC får tilgang til å se progresjonen til gruppa. Det gjør det også lettere å kunne jobbe hver for oss, som vi må forvente å gjøre i perioder der korona tiltakene strammes.

Doxygen vil bli brukt som et automatisert dokumentasjons verktøy. Denne applikasjonen kan generere forskjellige typer dokumenter for å vise dokumentasjonen, blant annet PDF- og HTML-sider. En bonus ved dette verktøyet er at den kan markere relasjoner mellom klasser og funksjoner med avhengihets-grafer, arve-grafer og sammarbeids-grafer. Doxygen støtter JavaScript og Java. Det er noe forskjell i hva som står i deres GitHub dokumentasjon og på nettsiden angående hvor stor grad Java er støttet. Dersom det viser seg at Doxygen ikke fungerer bra nok til Java bruker vi Javadoc istedet. Denne genererer også HTML sider, men med ulikt format.

Vi definerer kodestandarder i et annet dokument, dokumentet legges til som vedlegg i denne teksten.

5.2 Risikoanalyse (identifisere, analysere, tiltak, oppfølging) (Teknologisk, Forretningsmessig, Prosjektgruppemessig)

5.2.1 Kompleksitet ved Dependencies

Ved bruk av tredjeparts systemer vil applikasjonen kreve noe større grad av vedlikehold. Vi må sørge for at systemene som tilknyttes er oppdatert og fortsetter å utføre de oppgavene vi trenger de til. Her tilknytter vi teknologi-ansvarlig, som vil følge opp systemene og oppdatere de dersom det er nødvendig eller hensiktsmessig. Teknologi-ansvarlig vil også prøve å begrense tredjeparts systemer slik at unødig kompleksitet unngås.

5.2.2 APIer fra tredjeparter endres/svikter

For at programmet vårt skal kunne kjøre som forventet er den avhengig av å hente data fra flere forskjellige sider via API. Hvis det skulle skjedd noe med disse sidene, som at den endrer APIen den sender eller at serveren går ned, er det viktig at det blir gitt beskjed til brukeren at denne dataen ikke har blitt mottatt. Ved å lage egne funksjoner som har ansvar for å hente og lese inn dataen gjør det det lettere å endre den uten at resten av systemet blir påvirket og åpner opp for muligheten til å endre til en ny API hvis nødvendig.

5.2.3 Gruppen blir utsatt for sykdom

Ved et tilfelle der noen i gruppa blir smittet med korona og deretter møter opp på scrum møtet kan det påvirke arbeidet for de neste to sprintene. Alle på gruppa må følge de nyeste korona restriksjonene som er gitt og bare møtes når situasjonen tillater det, vi må også passe på å få nok søvn og trening for å redusere risikoen for andre sykdommer.

6 PLANLEGGING, OPPFØLGING OG RAPPORTERING

6.1 Hovedinndeling av prosjektet og valg av SU-modell

Bakgrunnen for valg av prosessrammeverket er bygget opp av de ressursene vi har tilgjengelig, de erfaringer vi har gjort fra tidligere arbeid og vårt ønske om ukentlig tilbakemelding fra både veileder og selskapet ETC. Ut ifra dette mener vi det er naturlig med en agil arbeidsmetode. (Foreløpig SCRUM, om vi ikke finner et annet navn som passer bedre)

Planen er å kjøre ukentlige sprinter hvor planleggingen av sprinten skjer tidlig i uken i samarbeid med veileder og ETC. Tredjepartene bidrar til at vi holder stødig kurs i

løpet av prosjektet, men selve planleggingen og utførelsen er opp til oss. Ellers kjøres korte statusmøter iløpet av uken med et siste oppsummeringsmøte ved slutten av uken.

Prosjektleder tar ansvar som SCRUM-Master under planleggingsmøtet på mandagene. De oppgavene som dukker opp på disse møtene fordeles på oss i gruppen basert på ønsker og ferdigheter vi sitter med.

6.1.1 Milepæler

Vi velger å dele opp prosjektets implementering i milepæler, hvor hvert mål viser til hvor stor grad av applikasjonen som er blitt implementert og hvor stor grad den er funksjonell.

Foreløpig får hver milestein 3 sprinter som tilsvarer 3 uker. Dette er i samsvar med de sprintene vi har satt opp i Gantt skjemaet. Vi ønsker å understreke at vi har en åpenhet rundt å legge til 1 eller 2 ekstra uker mot slutten dersom det skulle være nødvendig. Dermed er tiden per milestein noe løst definert.

Milepæl 1) 15. Februar

- **Prototype** - Prototypen er laget i Figma. Her skal det være ferdig resultat av en UI og UX model av applikasjonen.
- **Web Applikasjon** - En enkel tjener som leverer dokumenter (html, css og javascript filer) til klienter. Implementerer MySQL

  Fungerende database som kan lese og skrives til med simpel UI på websiden
- **Rate limit løsning** - Lagre den data som er ømfintlig for rate-limitering fra de tredjeparts grensesnittene vi bruker. Data som blir lagret vil bestå av værdata som senere skal fores inn i maskinlærings-algoritmen.
- **Område Definering** - Automatisk definering av områder i en valgt kommune. Skal defineres basert på Grunnkretser der vær-prognoser er enormt like over lang tid. Navngiving av et område skjer automatisk basert på grunnkretsene og skal være redigerbart i tilfellet navnet ikke passer lokale normer.

Målet med første milepæl er å få ferdig en prototype som vi kan utvikle etter for å forsikre oss om at vi møter forventningene og kravene til ETC. Samtidig ønsker vi å få ferdig grunnleggende punkter for utvikling videre, som f.eks en enkel web-tjener og algoritme for definering av områdene som skal brukes i systemet senere.

Som en del av det å få frem disse enkle og grunnleggende elementene skal vi også få inn automatiserte test verktøy for å sørge for at applikasjonen fungerer på tvers av utvikling og platformer. Dermed legger vi ved en release pipeline som skal teste og kjøre koden som blir levert i Azure.

Milepæl 2) 8. mars

- **Vær-prognoser** - Hente og vise vær-prognoser i form av diagrammer for et valgt område. Dette vil være et datapunkt som kan velges fra dashbordet.
- **Rodeoversikt** - Hente roder inn basert på områdene som er definert fra milepæl 1. Oversikten skal inneholde område, rodenavn, en utkalling-knapp, en beskrivelse av føret når vedlikehold bør utføres, et tidspunkt for når føret inntreffer og en "vis i kart" knapp som viser hvor området ligger.
- **Interaktivt Kart** - Kartet skal automatisk gå inn mot et av de registrerte områdene. Et område skal kunne velges slik at man senere kan bruke dette for å vise data basert på området som er klikket på.
- **Utkalling** - Det skal være mulig å gjøre utkallinger. Det vil si at en vaktleder skal få opp forslag om oppdrag som bør gjøres, og da ha muligheten til å tildele dette til en sjåfør.
- **Innlogging** - Hente grensesnitt for innlogging fra CarAdmin for å verifisere vaktledere. UI vil bli oppdatert utifra hvilken bruker som er innlogget.

Andre milepæl går ut på å få inn de grunnleggende funksjonene ved applikasjonen. Dette vil hovedsaklig bestå av lage en oversikt over de viktigste områdene/rodene som trenger vedlikehold. Her vil vi sortere rodene etter hvor ille vær-prognosen er, mens selve analysen vil vi gå dypere inn på ved neste milestein.

Oversikten skal også inneholde en funksjon for utkalling. Selve utkallingen gjøres gjennom CarAdmin, dermed vil potensielt knappen være tom til vi har fått mer informasjon fra ETC. På grunn av dette vil knappen utvikles åpent for å sende informasjon om utkallingen fra vårt system uten at vi har en spesifikk mottaker.

Innloggingen vil også basere seg på CarAdmin sine tjenester. Vi utvikler dermed innloggingen med tanke på at det CarAdmin brukere som logger inn, dersom denne ikke er tilgjengelig utvikler vi en test versjon av innloggingen som ikke inneholder passord. Applikasjonen vil likevel utvikles slik at den er begrenset basert på brukergrupper.

Et siste element ved denne milepælen er et interaktivt kart. Målet er å utvikle et kart som inneholder funksjoner for å stille kartet inn mot et område, velge et område og plassere trykkbare ikoner på kartet.

Milepæl 3) 29. Mars

- **Dashbord** - Vise div statistikk fra områder man trykker på. Historisk data, ikke framtidsdata.
- **Beslutningskart** - Er en videre utviklet versjon av det interaktive kartet. Her skal man kunne se forskjellige områder har blitt merkert med ikoner for værvarsel. Man kan klikke på ikonet for å få full rapport om været her og da ta en beslutning om å sende en utkalling.

- **Vedlikeholds-prognose** - Gjennom maskinlæring av vekta datapunkter skal vi levere en prognose for hva slags vedlikehold som skal utføres. Prognosene gjøres for hvert område, f.eks Øverby eller Sentrum.
- **Publikumskart** - Vise statuser på roder/områder. Dette kartet vil være tilgjengelig offentlig for alle og vil vise statuser som: Ikke begynt, Oppdrag startet, Oppdrag ferdig.
- **Avvik** - Gjør det mulig å legge inn avviksrapporter for både brukere og publikum, vaktleder kan i tillegg se og redigere innsendte avvik.

Ved tredje milepælet så åpner det for at offentligheten har oversikt over vedlikeholdet og har mulighet til å gi tilbakemeldinger for å bidra til maskinlæringen. Dashbordet vil gi en bedre oversikt over statistikken som har blitt hentet inn, og vil være koblet opp mot det interaktive kartet.

Avvikene registreres av både offentligheten slik beskrevet over og av brukerne i systemet. De registreres i samme system, men vil ikke nødvendigvis bli vist sammen. Dersom en rapport strider sterkt mot en prognose skal et varsel sendes til vaktleder.

I denne milepælen skal vedlikeholds-prognoser gis. Denne baserer seg da på dataene nevnt tidligere. Disse prognosene erstatter tidligere prioritering ved rodeoversikten slik at områder med verst prognoser kommer fremst.

Milepæl 4) 16. April

- **Analysemotor** - Analysemotoren skal være ferdig. Med dette vil det si at den samler data fra Met og potensielt andre ressurser og sammenlikner dette med de observasjoner som er gjort og gir en anbefaling til vaktleder.
- **Web Applikasjon** - En fullstendig nettside som presenterer kart og tabeller på en oversiktlig måte og som er lett for en vaktleder å bruke.
- **Dokumentasjon** - Sammen med et ferdig prosjekt skal vi levere dokumentasjon av koden som er skrevet. Denne leveres som en nettside generert av Doxygen med forklaringer rundt de objekter og funksjoner som trenger det. Dokumentasjonen vil også inneholde overordna diagrammer som viser prosessene i systemet: analysemotoren, områdedefinering, rodeprioritering.

Ved fjerde og siste milepæl skal hele applikasjonen være ferdig og leveres med et eget dokument med dokumenteringen for koden. Nå skal alle komponenter fungere og være satt sammen på riktig måte i applikasjonen.

Dokumentasjonen skal være oversiktlig og lett å forstå. Det den skal gi god nok informasjon for at en utvikler som er uvitende til denne applikasjonen. Lett skal kunne les seg opp på dokumentasjonen og forstå hvordan applikasjonen funger. Ut i fra dokumentasjonen skal det våre lett for utvikleren å videre utvikle applikasjonen.

Analysemotor skal være fult fungerende og levere sine anbefalinger til applikasjonen. Dette vil si at analysemotoren kan ta imot data fra Met og andre vær kilder. Med

dissa dataene skal analysemotoren kunne sammenligne vær datane for nåtiden og framtiden. Den skal ut i fra det kunne gi anbefalinger for veiarbeidsoppdrag som burde bli gjort etter hvordan været kommer til å bli framover på dagen. Vaktleder skal ha muligheten til å godta anbefaling og sende en utkalling til et oppdrag. Eller ikke godta det om det skulle være feil eller han oppdager annet vær ut i fra personlig observasjoner og erfaring.

I denne milepælen skal arbeidsgiver få et helt produkt som de skal kunne bruke. Produktet skal fungere etter forventet kravspesifikasjon for produktet.

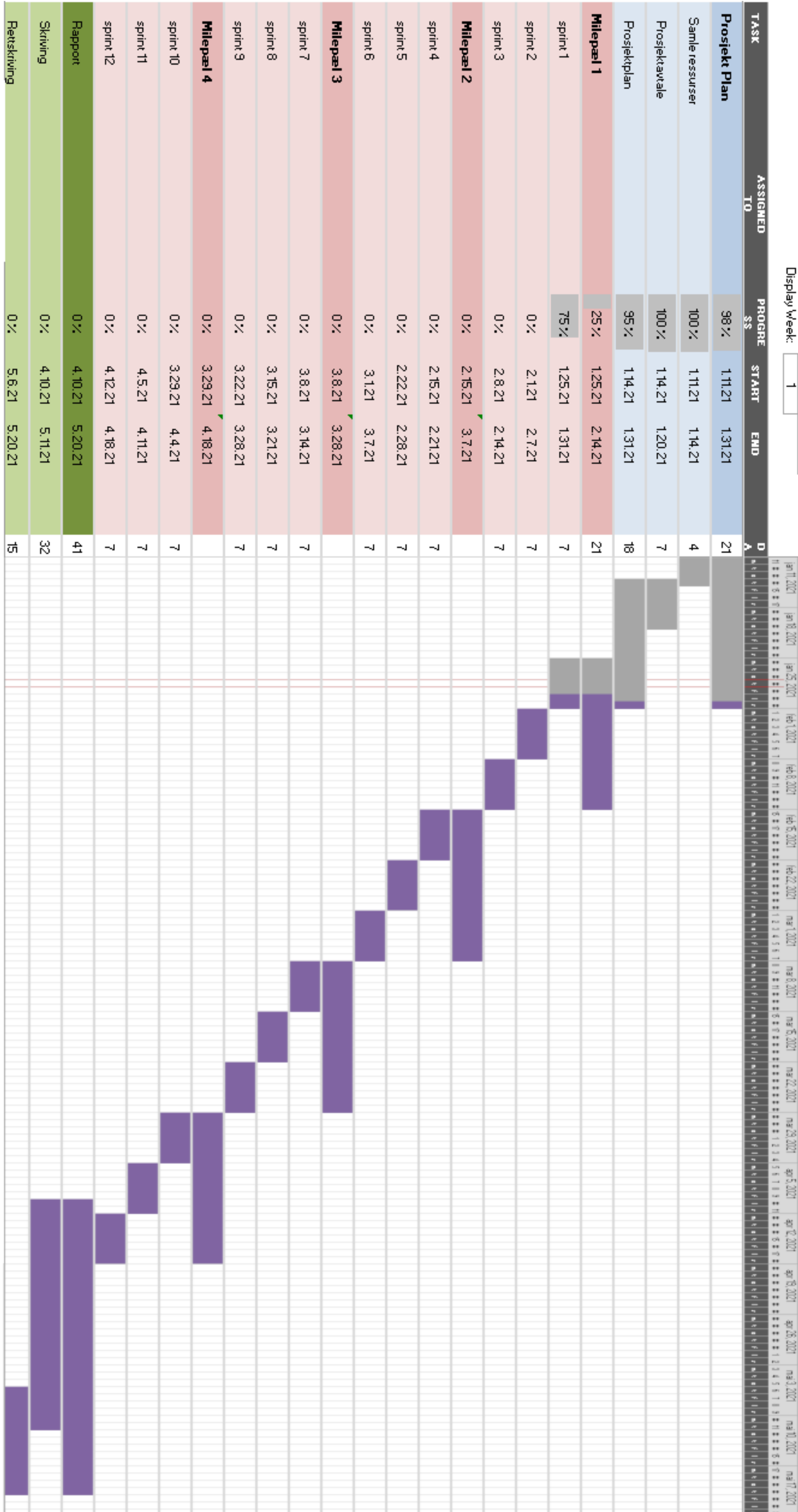
6.2 Plan for statusmøter og beslutningspunkter i perioden

Sprint-Planning foregår på mandager med gruppen, her lager vi er grov plan for uken. Hovedpunkter ved vært møte er: Hva kan vi levere iht målene våres denne uken, og hvordan kan vi levere det. Det er i all hovedsak under dette møtet at de store beslutningene tas, men med en viss åpenhet rundt at beslutninger også kan tas iløpet av uken. Planlagt tid til dette møtet er cirka en time.

Som en del av den agile-metoden kjører vi også daglige korte møter på cirka et kvarter for å oppsummere hva som er blitt gjort, hva som skal gjøres og punkter man sliter med. Dette gjør at vi får godt overblikk over status på prosjektet og nåværende sprint samtidig som vi får god mulighet for å hjelpe hverandre med enkelt emner.

Vi avsetter siste timen på fredager for å oppsummere uken, oppdatere arbeidsloggen og demonstrere arbeidet som er gjort. Dette gjør det lettere å planlegge neste ukes sprint da vi får innblikk i både funksjonalitet og status for prosjektet. Etterhvert i prosjektets livsløp vil også produkteier, altså ETC, også få tilgang på demoer som blir produsert. Med dette får produkteier mulighet for innspill ved de møtene vi har med dem.

7 Gantt-Skjema over prosjektplan



TASK	ASSIGNED TO	PROGRESS	START	END	DAYS
Prosjekt Plan		98%	1.11.21	1.31.21	21
Samle ressurser		100%	1.11.21	1.14.21	4
Prosjektavtale		100%	1.14.21	1.20.21	7
Prosjektplan		95%	1.14.21	1.31.21	18
Milepæl 1		25%	1.25.21	2.14.21	21
sprint 1		75%	1.25.21	1.31.21	7
sprint 2		0%	2.1.21	2.7.21	7
sprint 3		0%	2.8.21	2.14.21	7
Milepæl 2		0%	2.15.21	3.7.21	
sprint 4		0%	2.15.21	2.21.21	7
sprint 5		0%	2.22.21	2.28.21	7
sprint 6		0%	3.1.21	3.7.21	7
Milepæl 3		0%	3.8.21	3.28.21	
sprint 7		0%	3.8.21	3.14.21	7
sprint 8		0%	3.15.21	3.21.21	7
sprint 9		0%	3.22.21	3.28.21	7
Milepæl 4		0%	3.29.21	4.18.21	
sprint 10		0%	3.29.21	4.4.21	7
sprint 11		0%	4.5.21	4.11.21	7
sprint 12		0%	4.12.21	4.18.21	7
Rapport		0%	4.10.21	5.20.21	41
Skriving		0%	4.10.21	5.11.21	32
Retskriving		0%	5.6.21	5.20.21	15

Figure 2: Gantt-skjema

Tillegg D

Oppdrag

Oppdragsgiver

Oppdragsgiver: ETC, Electric Time Car AS
Kontaktperson: Dag L Solhaug
Adresse: Studieveien 2, 2815 Gjøvik
Telefon: +47 901 01 344
Epost: dag.solhaug@electrictimecar.com

Beslutningsstøtte

ETC er et lokalt innovativt IT-selskap som leverer nyskapende løsninger for helhetlig kjøretøyoppfølging, kalt CarAdmin. Nå samarbeider vi med NTNU ifm Smarte vinterveier for å vinne en Innovasjonskontrakt med Mjøsbyene.

Innovasjonspartnerskapet Smart vinterveg er et prosjekt hvor vi finner fremtidens løsninger for vinterdrift av veger og eiendommer. https://www.smartemjosbyer.no/

Oppgaven

En vaktleder som har ansvaret for å sette i gang veivedlikehold på vinterveier trenger et beslutningsgrunnlag å handle etter. Eks er det meldt underkjølt regn mellom 03 og 04 i Gjøvik. Må saltbilene begynne saltingen allerede fra kl 23 kvelden før for å rekke å salt før det underkjølte regnet inntreffer for å forhindre såpeglatte veier.

Oppgaven går ut på å lage et beslutningsdashbord hvor det samle og presenteres innsamlede relevante data for veivedlikehold, som erfaringer, værforhold med værprognoser, sanntidsdata og observasjoner.
De innsamlede dataene skal automatisk gi forslag til hvilke veivedlikehold som anbefales.

Nærmere spesifisering og avgrensing gjøres sammen med oppdragsgiver så snart oppgaven er tildelt da denne informasjonen kan være sensitiv ift pågående anbud.

Dere som tar denne oppgaven, vil få en utvikler hos oss til å hjelpe dere gjennom oppgaven slik at dere på en ordentlig måte kommer raskt og greit inn i vår system.

Programmering
ETC vil bistå prosjektet i gjennomføringen av oppgaven. Jevnlige møter og oppgavefordelinger avtales underveis.

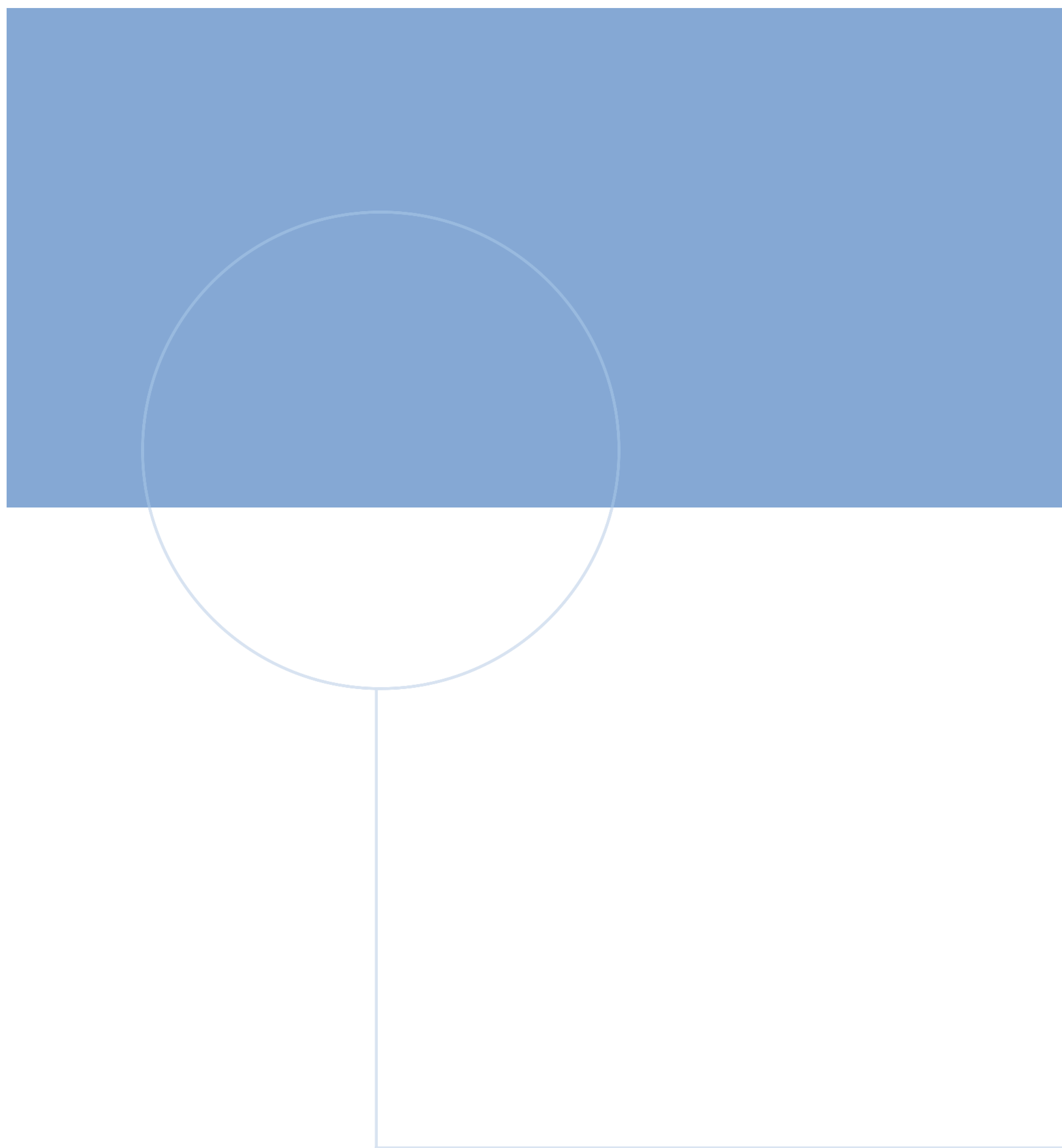
Studentgruppen vil gjennom prosjektet få erfaring med:

- Java / HTML / Ajax / SQL
- Database - MariaDB
- Server side programmering
- Web/Klientprogrammering
- Prosjektet utvikles som en modul i vår løsning CarAdmin

Oppgaven passer best for en gruppe på 3 - 4 personer som kan Java.

ETC har gjennomført bacheloroppgave ved HiG/NTNU siden 2004 hvor flere av oppgaven har fått topp karakter. Alle bacheloroppgaver har resultert i ansettelser i ETC, eller ansettelser som følge av referanse fra ETC etter endt bacheloroppgave.

Vi er på utkikk etter å ansette Java programmerere, lokal tilhørighet vektlegges.



NTNU
Kunnskap for en bedre verden