Analyse av oppgaver og oppgavesjangre i en digital eksamen i innledende programmering

Guttorm Sindre
Excited SFU og Institutt for datateknologi og informatikk (IDI), NTNU

Sammendrag

Et viktig mål for samstemt undervisning er at læringsaktiviteter i løpet av semesteret og vurderingen etterpå skal stemme best mulig overens med intenderte læringsutbytter i emnet. Utfordringen for eksamen er at den har mange til dels motstridende krav og ønsker, og særlig grunnundervisningen – slik som innledende emner i programmering – har utfordringer med store klasser. Ulike oppgavetyper vil ha forskjellige fordeler og ulemper på eksamen, og overgang til digital eksamen med verktøy som Inspera eller WiseFlow vil kunne påvirke disse fordelene og ulempene. Denne artikkelen gir først en generell gjennomgang av ulike oppgavetyper innen programmering, basert på internasjonal forskningslitteratur. Deretter gir den en konkret analyse av bruken av en del oppgavetyper i Inspera i en spesifikk eksamen (TDT4127 Programmering og numerikk ved NTNU, høst 2018), for å se på korrelasjoner og vanskegrad og hva dette kan si om bruken av ulike oppgavetyper. Analysen indikerer at auto-rettede oppgavesjangre med kodekomplettering i noen grad kan dekke samme behov som programmeringsoppgaver, men at det samtidig er potensial for forbedringer, både relatert til hvordan oppgavene ble brukt i eksamen, og den generelle verktøystøtten.

Innledning

Samstemt undervisning (eng. *constructive alignment*) [1] tilsier at læringsaktiviteter underveis i semesteret og vurdering (eksamen eller annet) begge bør samsvare best mulig med de intenderte læringsutbyttene i emnet. Mange faglærere baserer seg på tradisjoner i faget og egen intuisjon i utvikling av eksamensoppgaver, uten nødvendigvis å ha noen underliggende læringsteori som fundament [2]. Selv med pedagogisk fundament og grundig planlegging kan det være vanskelig å oppnå perfekt samsvar mellom eksamen og intenderte læringsutbytter. Dels vil karakterer påvirkes av *konstrukt-irrelevans* [3], dvs. ferdigheter eller egenskaper hos studentene som ikke er del av de intenderte læringsutbyttene. Det motsatte problemet er *konstrukt-underrepresentasjon* [4], dvs. at eksamen f.eks. pga. kort varighet eller andre begrensninger i rammebetingelser ikke klarer å vurdere alle intenderte læringsutbytter og må nøye seg med et utvalg.

Overgang fra papireksamen til digital eksamen kan ha en rekke fordeler både administrativt og faglig [5, 6]. Mange studenter foretrekker digital eksamen heller enn papir, på grunn av mindre slitsom skriving og bedre muligheter for redigering. For faglærere ligger én mulig fordel i økt bruk av oppgaver som kan rettes automatisk, dette kan både gi mer pålitelige karakterer og mindre stress med å rekke en 3-ukers sensurfrist. Potensielt kan digital eksamen også muliggjøre oppgavetyper som var vanskelig å få til på papir, med økt validitet for læringsutbytter som slike oppgaver passer til [7]. På den annen side kan mer bruk av auto-rettede oppgaver være en kilde til bekymring hvis slike oppgaver skulle ha lavere validitet i forhold til ønskede læringsutbytter enn de oppgavene man tidligere brukte på papireksamener, og det primære argumentet for oppgavevalget blir kostnadsbesparelsen. Det er derfor viktig å skaffe seg kunnskap om hva ulike oppgavetyper kan egne seg til.

Akkurat hvilke oppgavesjangre som er auto-rettede vil avhenge av verktøyet. Potensielt kan vanlige programmeringsoppgaver også gjøres auto-rettede. Denne artikkelen forholder seg imidlertid til det som er vanlig verktøystøtte i forbindelse med gjennomføring og retting av eksamener, som f.eks. Inspera Assessment som brukes ved NTNU og flere andre norske universiteter. Her er programmeringsoppgaver en manuelt

This paper was presented at the UDIT 2019 conference. For more information see http://www.udit.no/

rettet oppgavetype, det er ikke støtte for at kode kan kjøres mot testsuiter i sensurprosessen, og heller ikke støtte for at studentene kan kjøre koden underveis på eksamen. Det samme er tilfelle med WiseFlow, som enkelte andre norske universiteter bruker.

Forskningsspørsmålene for denne artikkelen er som følger:

- FS1: I hvilken grad kan ulike auto-rettede oppgavesjangre dekke samme læringsutbytter som programmeringsoppgaver?
- FS2: Har typiske auto-rettede oppgavesjangre (som flervalg, nedtrekk, innfylling, omstokking) større problem med konstrukt-irrelevans enn tradisjonelle programmeringsoppgaver?
- FS3: Kan auto-rettede oppgavesjangre fungere like godt som programmeringsoppgaver når det gjelder å skille mellom sterkere og svakere studenter?

Datamaterialet for artikkelen er fra en bestemt eksamen i ett bestemt emne, nemlig TDT4127 Programmering og numerikk ved NTNU, høsten 2018, som ble tatt av cirka 200 studenter. At det kun dreier seg om en enkelt eksamen, er selvsagt en svakhet ved artikkelen, det ville vært bedre å ha data fra mange eksamener, og aller helst i ulike programmeringsemner ved ulike universiteter. Funn fra en enkelt eksamen kan likevel være en interessant begynnelse og danne grunnlag for større sammenligninger senere.

Resten av artikkelen er strukturert som følger: Seksjon 2 gir en gjennomgang av forskningslitteratur om oppgavetyper i programmering, med fokus på nybegynneremner. Seksjon 3 gjør rede for konteksten for det aktuelle emnet og hvordan eksamen var bygget opp. Seksjon 4 forklarer forskningsmetoden som ble brukt i analyse av dataene. Seksjon 5 presenterer og analyserer resultatene, og seksjon 6 inneholder en avsluttende diskusjon og konklusjon.

Oppgavesjangre i innledende programmering

Det er gjort flere arbeider med forsøk på å kategorisere eksamensoppgaver innen innledende programmering, typisk kalt CS1 i internasjonale artikler. Petersen et al. [8] gjorde en sammenligning av 15 eksamenssett fra diverse amerikanske universiteter. I snitt hadde disse eksamenene 59% vekt på å skrive kode, 13% på å lese og forstå kode, 10% på forståelse av programmeringskonsepter, og 18% på ikke-programmering (en del av emnene hadde noe annet pensum i tillegg til ren programmering, litt à la f.eks. IT Grunnkurs ved NTNU). Av 59% skriving av kode, var det meste (54%) programmeringsoppgaver, og resten (5%) kortsvarsoppgaver (f.eks. fylle inn manglende kode). Simon et al. [9] så på 20 eksamener, de fleste fra australske universiteter, og fant 48% skriving av kode, 18% sporing av kode (f.eks. «Hva blir skrevet ut av dette programmet?»), 8% forklaring av kode (mer dyptgående enn sporing), 5% feilfinning, 2% modifisering av kode, samt oppgaver som ikke direkte involverte programmering (10% gjengi faktakunnskap, 7% programdesign, 2% testing). Det er ikke gjort noen tilsvarende sammenlignende analyse av norske programmeringseksamener, men forfatterens egen erfaring både som eksaminator og sensor tilsier et lignende bilde her, med følgende vanlige oppgavetyper:

- *Skriving av kode*, dvs. hvor oppgaveteksten forklarer hva (en del av) et program skal gjøre, og det er kandidatens oppgave å skrive kode som tilfredsstiller de gitte kravene. Dette later til å være den dominerende oppgavetypen.
- *Kodeforståelse*, dvs. hvor det er gitt noe programkode og kandidaten skal forklare hva denne gjør. Avhengig av oppgaveformuleringen kan dette enten være å

forklare koden instruksjon for instruksjon, eller bare hva sluttresultatet blir (f.eks. «Hva blir utskriften fra dette programmet?»).

- *Teorioppgaver*, som ikke etterspør praktiske ferdigheter men kunnskap om programmeringskonsepter eller mekanismer. Eksempel for Python kunne være «Hva er forskjellen på lister og tupler?»; «Hva er fordeler og ulemper med mengder (*set*) vs. lister (*list*)?»
- *Feilfinning*, hvor det er gitt noe kode, pluss forklaring av hva koden var ment å gjøre, men koden inneholder feil som kandidaten skal gjøre rede for, eventuelt også foreslå korreksjoner.
- *Fullføring av kode*, hvor det er gitt forklaring av hva koden skal gjøre, pluss at deler av ferdig kode er presentert. To varianter av dette er oppgaver med innfylling (hvor noe er tatt bort / skjult i koden og skal settes inn) og omstokking, hvor kodelinjer er vist i sin helhet men i feil rekkefølge, og det er kandidatens oppgave å sette dem riktig. I forskningslitteratur kalles oppgaver med omstokking gjerne «Parsons' problems» [10], oppkalt etter den som først begynte å undersøke bruk av denne oppgavetypen i programmering [11].

Ulike oppgavetyper kan adressere ulike typer læringsutbytter og vil dermed utfylle hverandre i et eksamenssett. Både programmering, kodeforståelse og feilfinning samsvarer med evner som har klar jobbmessig nytteverdi, siden en programmerer ikke bare må kunne skrive kode, men også forstå og vedlikeholde kode skrevet av andre. Collofello [12] påpekte allerede i 1989 at undervisning ved universitetene har en tendens til å fokusere mer på nyutvikling av programvare (såkalt *green field development*), mindre på vedlikehold, selv om vedlikehold gjerne vil være en sentral aktivitet i den første jobben mange får etter endt utdanning. De øvrige spørsmålskategoriene kan sies å være mindre autentiske, men har likevel en klar misjon både for læring og vurdering. Selv om man neppe blir stilt et eksplisitt spørsmål om fordeler og ulemper med lister vs. mengder i en arbeidssituasjon, er det viktig å ha denne kunnskapen for å velge riktig datastruktur i utvikling av systemer. I praktisk arbeid blir man heller ikke stilt overfor kode hvor enkelte uttrykk er skjult eller linjer er stokket om. Slike oppgaver har imidlertid en mulig pedagogisk fordel i form av lavere kognitiv last for studenten [13]. Når man skal skrive all koden selv, må man mestre mange konsepter på en gang, og på flere ulike nivåer (riktig syntaks, datastruktur, algoritme, …). Kompletterings- og omstokkingsoppgaver kan gjøre det mulig å fokusere på én ting av gangen, som er gunstig i en læringssituasjon for nybegynnere. På en eksamen kan slike oppgaver gjøre det mulig for svake studenter å vise at de kan *noe*, selv om de ikke får til de mer ambisiøse oppgavene med å skrive større sammenhengende kode [14, 15]. Omstokkingsoppgaver fritar studentene fra det syntaktiske bryet med å huske akkurat hvordan hver enkelt linje skal skrives, og kan dermed fokusere mer på konstruktive ferdigheter, i form av evne til å sette sammen byggeklosser til en fungerende helhet [11, 13]. Både komplettering og omstokking kombinerer programmering og kodeforståelse, da man for det første må forstå den koden som allerede står der, og være i stand til å fullføre det som mangler.

Kontekst og eksamensdesign

Emnet TDT4127 Programmering og Numerikk ble gitt første gang høsten 2018. Konteksten for emnet er litt spesiell, da målgruppen er masterstudenter som blir tatt opp i 4.årskurs på ulike siv.ing.studier ved NTNU, etter tidligere å ha gått 3 år på ingeniørhøgskole, men med for lite IT-fag fra tidligere studier til å tilfredsstille kravene for siv.ing.grad. Dette innebærer at de ikke kan programmere fra før, så programmeringsdelen av emnet (som utgjør 2/3) starter med helt grunnleggende konsepter á

la et innføringskurs i programmering på bachelornivå. På den annen side har disse studentene hatt en del matematikk, så numerikkdelen (1/3) kan starte på et høyere nivå.

Emnet hadde omtrent 200 studenter høsten 2018, og karakter var basert på en avsluttende digital eksamen. Inspera Assessment tilbød på eksamenstidspunktet 17 ulike oppgavetyper, hvorav 10 auto-rettede og 7 manuelle. Eksamen i TDT4127 hadde 20 oppgaver totalt, og *Tabell 1* viser en oversikt over oppgavetyper og temaer som var dekket. 60,5% av vekten var på auto-rettede oppgaver. De manuelt rettede oppgavene var i all hovedsak av typen Programmering. Av plasshensyn er det ikke mulig å diskutere hver av de 20 oppgavene i detalj. Siden flervalg, sant/usant og manuelt rettede kortsvar og programmeringsoppgaver er velkjente oppgavetyper, fokuserer vi på øvrige oppgavetyper som har vært mindre vanlig å bruke ved norske universitet.

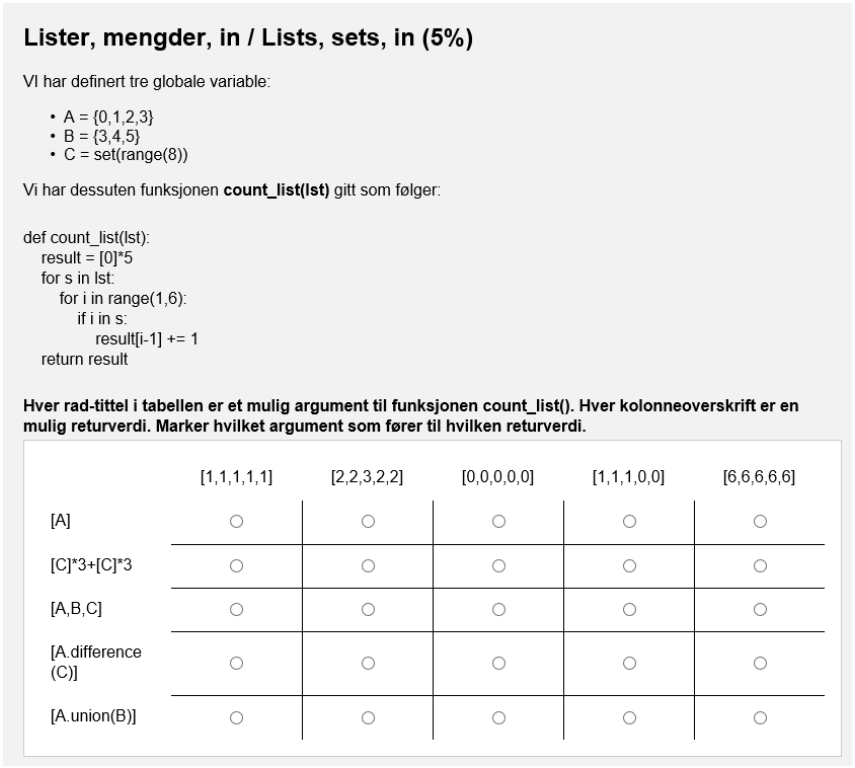
Tabell 1. Eksamensoppgaver i rekkefølge gitt i settet

Nr	Vekt%	Sjanger	Pensumdel	Inspera-Type	Retting
1	5	Teori	Programmering	Flervalg	Auto
2	10	Teori	Numerikk	Flervalg	Auto
3	3	Forstå kode	Numerikk	Sant/usant	Auto
4	3	Finn feil	Numerikk	Tekstfelt	Manuell
5	4	Finn feil	Programmering	Flervalg	Auto
6	5	Forstå kode	Programmering	Paring	Auto
7	5	Fullfør kode	Programmering	Dra og slipp	Auto
8	5	Fullfør kode	Programmering	Nedtrekk	Auto
9	5	Fullfør kode	Programmering	Nedtrekk	Auto
10	5	Skriv kode	Programmering	Programmering	Manuell
11	5	Fullfør kode	Programmering	Dra og slipp	Auto
12	5	Skriv kode	Numerikk	Programmering	Manuell
13	5	Fullfør kode	Numerikk	Dra og slipp	Auto
14	5	Skriv kode	Numerikk	Programmering	Manuell
15	5	Fullfør kode	Numerikk	Nedtrekk	Auto
16	5	Skriv kode	Numerikk	Programmering	Manuell
17	5	Skriv kode	Numerikk	Programmering	Manuell
18	4	Skriv kode	Programmering	Programmering	Manuell
19	3,5	Fullfør kode	Programmering	Fyll inn tekst	Auto
20	7,5	Skriv kode	Programmering	Programmering	Manuell

Figur 1 viser en kodeforståelsesoppgave gitt som Paring i Inspera. Poenget med å bruke akkurat denne oppgavetypen var at den kunne presenteres forholdsvis konsist men likevel ha N (her 5) spørsmål som studentene kan svare rett eller galt på, og dermed gi en viss granularitet. Man kunne ha oppnådd samme granularitet med 5 flervalgsspørsmål, men måtte da ha brukt plass og tid på å presentere distraktorer for hvert spørsmål. Alternativt kunne man ha gitt oppgaven slik at studentene selv måtte skrive returverdiene fra funksjonen, enten som auto-rettet fyll-inn-tekst, eller manuelt rettet kortsvar, men begge disse har ulemper: Manuelt rettet kortsvar blir mer tidkrevende å rette og mer sårbart for subjektivt skjønn, og fyll-inn-tekst gjør at studenter kan ende opp med null poeng for nesten riktig svar (f.eks. én taste- eller tankefeil i en liste på 5 tall, eller ett tilfelle hvor man uheldigvis taster punktum i stedet for komma). Med paring har man fordelen av auto-retting samtidig som man slipper å lage distraktorer, og antall kombinasjoner er såpass stort at oppgaven blir forholdsvis lite sårbar for ren gjetning.

Figur 2 viser et utsnitt av den første omstokkingsoppgaven (nr 7 i settet). Siden innrykk er semantisk signifikant i Python, brukes en todimensjonal variant av slike oppgaver [16]. Av plasshensyn inkluderes ikke den foregående forklaringen av hva koden skal gjøre, bare selve svarområdet. Studentens oppgave her var å trekke kodelinjene fra

venstre og inn til riktig posisjon i rutenettet. Bildet viser hvordan svarområdet vil ha sett ut for en student som var kommet litt i gang med å svare på oppgaven: Studenten har trukket de to første kodelinjene til korrekt posisjon, mens det fortsatt gjenstår å finne ut hvordan de øvrige linjene skal plasseres. Siden denne oppgavetypen gir en viss risiko for at studentene kan bli hardt straffet for småfeil (f.eks. nesten alt rett, men en feilplassert linje gjør at det meste er blitt forskjøvet ett hakk; eller alt i rett rekkefølge med ett innrykk for langt inn), ble svarene sett igjennom manuelt i tillegg til auto-rettingen. To av de cirka tohundre studentene fikk justert poengsummen sin i positiv retning som følge av slike tilfeller.



Lister, mengder, in / Lists, sets, in (5%)

VI har definert tre globale variable:

- A = {0,1,2,3}
- B = {3,4,5}
- C = set(range(8))

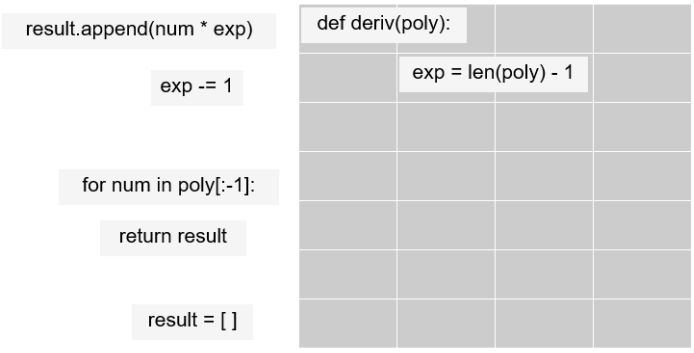
Vi har dessuten funksjonen **count_list(lst)** gitt som følger:

```
def count_list(lst):
    result = [0]*5
    for s in lst:
        for i in range(1,6):
            if i in s:
                result[i-1] += 1
    return result
```

Hver rad-tittel i tabellen er et mulig argument til funksjonen count_list(). Hver kolonneoverskrift er en mulig returverdi. Marker hvilket argument som fører til hvilken returverdi.

	[1,1,1,1,1]	[2,2,3,2,2]	[0,0,0,0,0]	[1,1,1,0,0]	[6,6,6,6,6]
[A]	○	○	○	○	○
[C]*3+[C]*3	○	○	○	○	○
[A,B,C]	○	○	○	○	○
[A.difference (C)]	○	○	○	○	○
[A.union(B)]	○	○	○	○	○

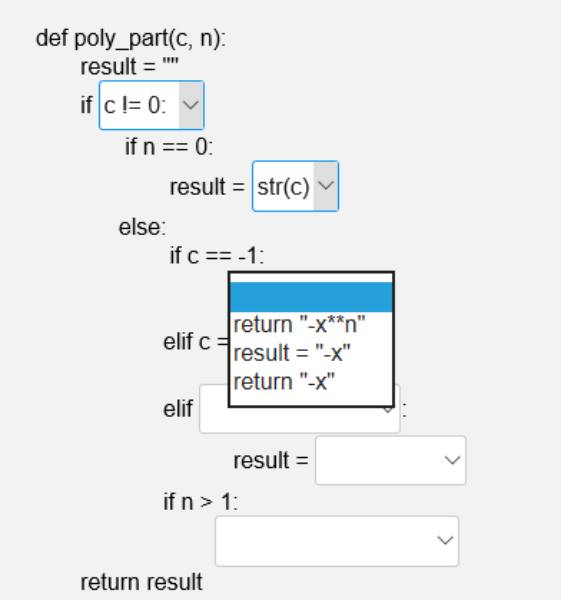
Figur 1. Kodeforståelsesoppgave som **Paring** i Inspera



Figur 2. Omstokkingsoppgave som **Dra og slipp** i Inspera

Figur 3 viser en nedtrekksoppgave, igjen uten den forklarende teksten. Her er det gitt en funksjon med noen fragmenter av koden utelatt. Hvert av disse feltene gir ved museklikk en meny med alternativ for hva som kan fylles inn. Oppgavetypen har dermed mye til felles med flervalg. Nedtrekk er mer hensiktsmessig enn flervalg her, fordi (a) det

tar mindre plass på skjermen siden svaralternativer «skjules» inni feltene og (b) når man har valgt alternativ, ser man valgt svar i koden heller enn annet sted på skjermen. Hva man skal svare lenger nede, vil gjerne avhenge av tidligere valg. Nedtrekk gjør det enklere for studenten å se sammenheng i koden når alternativer er valgt enn hva flervalg ville gjøre.

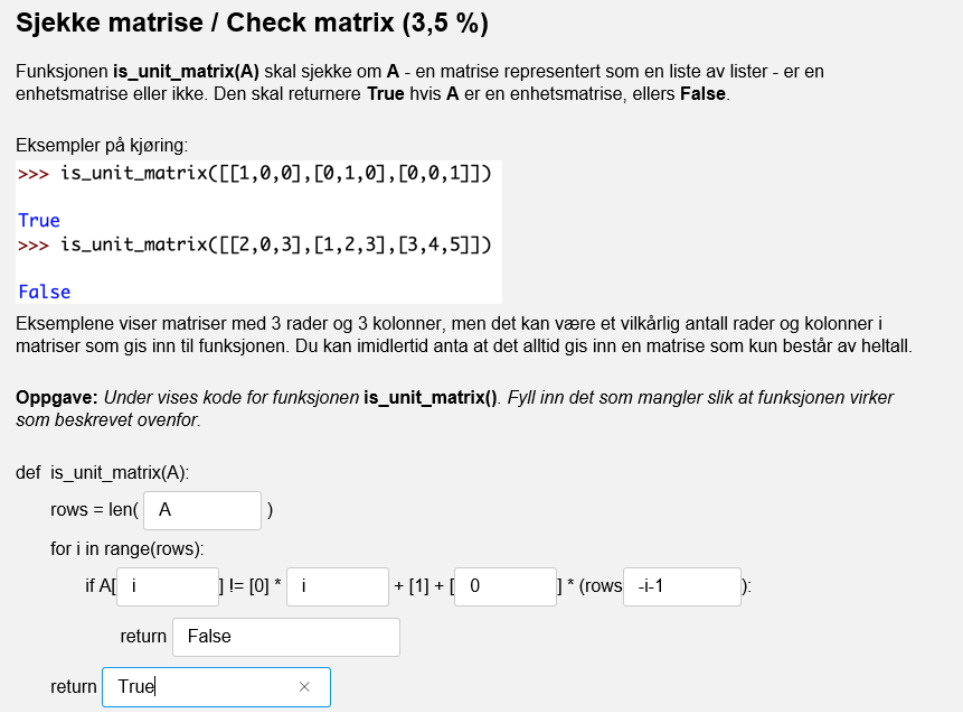


Figur 3. Fullføringsoppgave som **Nedtrekk** i Inspera

Et alternativ til nedtrekk er «Fyll inn tekst», vist i Figur 4 (oppgave 19 i settet). Bildet viser hvordan oppgaven ville se ut for en student som akkurat har fylt inn alle rutene med korrekte svar. Denne auto-rettede sjangeren er kun hensiktsmessig hvis det som skal fylles inn, er kort (som her: stort sett enkle variabelnavn eller konstanter), siden det ellers vil være lett for at små tastefeil gjør at studenter som har tenkt riktig likevel taper poeng. Det er også viktig for oppgavestiller å være oppmerksom på situasjoner hvor flere ulike uttrykk kan være riktige. I feltet lengst til høyre var det satt opp 4 mulige korrekte svar: `-i-1` ; `-1-i` ; `-(i+1)` ; `-(1+i)`, og for alle oppgavene var det satt opp å ignorere evt. blanke tegn. Selv om denne oppgaven var auto-rettet ble den i tillegg sett kjapt igjennom manuelt for å se om noen studenter som hadde lav score burde få delvis uttelling (f.eks. uheldigvis skrevet I med stor bokstav, eller true, false med små, som riktignok er feil i Python men likevel burde gi delvis uttelling hvis det var plassert i riktig rute.

Forskningsmetode

Forskningen i artikkelen bygger på en statistisk analyse av eksamensresultatene. Det fins en del litteratur om typiske analyser man kan gjøre av eksamensresultater i etterkant, f.eks. [17], primært fokusert på flervalgsoppgaver og lignende. En oppgaves *vanskegrad* kan finnes ved å se på studentenes gjennomsnittlige score på oppgaven (i prosent av oppnåelig maks-score). En vanlig analyse for eksamensspørsmål er å se på *korrelasjon* mellom score på det enkelte spørsmålet og totalscore, hvor man gjerne tenker jo høyere jo bedre, iallfall så lenge det er snakk om at de ulike oppgavene skal måle samme eller nært relaterte læringsutbytter. For våre to første forskningsspørsmål er det spesielt interessant å se på følgende når det gjelder korrelasjoner:



Figur 4. Fullføringsoppgave som **Fyll inn tekst** i Inspera

- Knyttet til FS1: Hvordan er korrelasjonen mellom auto-rettede oppgaver med programkomplettering (dra-og-slipp, nedtrekk, fyll-inn som vist i Fig 2-4) og vanlige programmeringsoppgaver? Hvis korrelasjonen er høy, f.eks. like god som internt mellom programmeringsoppgaver, vil dette tyde på at de kan måle samme læringsutbytter.
- Knyttet til FS2: Hvordan er intern korrelasjon for spesifikke oppgavetyper vs. ekstern korrelasjon mellom disse oppgavene og andre oppgavetyper? Anta f.eks. at det fins en ferdighet som noen studenter har og andre ikke har med å være spesielt god til å løse omstokkingsoppgaver (uavhengig av om man er god i programmering eller ikke). Hvis en slik ferdighet har betydelig innvirkning på resultatet, ville man forvente at dra-og-slipp-oppgaver har en vesentlig høyere intern korrelasjon enn ekstern korrelasjon, fordi studenter som har denne ferdigheten knyttet til omstokking, vil gjøre det bedre på disse oppgavene, andre vil gjøre det dårligere.

Med hensyn på FS 3, å undersøke hvor godt oppgaver skiller mellom sterke og svake studenter, brukes typisk det som kalles en diskrimineringsindeks (*item discrimination index*). Det er foreslått mange ulike slike indekser [18], i denne artikkelen brukes den klassiske indeksen D foreslått av Kelley [19], hvor man sammenligner de beste 27% og de svakeste 27% av studentene, dvs. D = (Høy – Lav) / Total, hvor Høy er snittscore på deloppgaven blant de 27% beste av studentene mhp totalscore, Lav er snittscore på samme deloppgave blant de 27% svakeste av studentene, og Total er snittet som alle studentene oppnådde på deloppgaven. D vil ligge mellom -1 og +1, hvor +1 vil bety at deloppgaven skiller perfekt (alle de sterke studentene klarte oppgaven, alle de svake feilet), 0 vil bety at den ikke skilte i det hele tatt, og -1 ville innebære at den skilte omvendt, dvs. alle svake studenter fikk til deloppgaven mens sterke studenter feilet. Særlig en negativ verdi for D vil vanligvis indikere at det er noe tvilsomt ved oppgaven, for eksempel en villedende formulering som gjør at sterke studenter forledes til å svare feil, mens svake studenter som har forstått mindre av pensum derimot ikke skjønner det

distraherende hintet og svarer riktig. Det er også regnet som et svakhetstegn hvis en oppgave har høy vanskegrad men likevel lav (om enn positiv) D, dette kan indikere at det er litt tilfeldig hvem som svarer riktig og galt, heller enn å være basert på læringsutbyttet. Det er imidlertid verdt å merke seg at indeksen D også er gjenstand for noe kritikk, blant annet at man typisk vil oppnå høyest D for spørsmål som er i midtsjiktet når det gjelder vanskegrad [20]. En eksamen kan også trenge noe lette oppgaver for å skille mellom E og F, og noe vanskelig for å skille mellom A og B. Man kan derfor ikke blindt fokusere på at alle deloppgaver skal ha høyest mulig D-verdi, men lavere D-verdi bør henge sammen med et bevisst valg om at den aktuelle oppgaven var ment å være spesielt lett eller spesielt vanskelig.

Det kan sies å være en metodisk svakhet at utarbeidelse og gjennomføring av eksamen ikke kunne gjøres primært ut fra forskningsmessige målsetninger, da det viktigste var å gi studentene en adekvat eksamen i forhold til læringsutbytter og praktiske rammer. Vi kunne ha oppnådd en bedre sammenligning av ulike oppgavetyper med et eksperiment spesifikt designet for formålet – for eksempel delt studentene i ulike tilfeldige grupper og latt en gruppe få som vanlig programmeringsoppgave samme problemstilling som en annen gruppe fikk som en auto-rettet oppgavetype, og vice versa. Et slikt eksperiment har imidlertid andre utfordringer, som å få tak i nok deltakere, finne tid og motivere studentene til å gjøre sitt beste. Eksamen har fordelen at deltakerne kommer «gratis» og i utgangspunktet er motivert for å gjøre sitt beste. Selv om eksamen ikke kunne designes primært ut fra forskningshensyn, kan det dessuten legges til at faglærerne var bevisste på å prøve ut flere forskjellige oppgavetyper for å se hvordan de fungerte, og siden det var første gang emnet ble gitt, fantes det ikke noen tidligere tradisjon for eksamen som man følte seg forpliktet til å følge.

Resultater

Tabell 2 viser de 20 oppgavene sortert fra den med høyest snittscore til lavest snittscore. En god del av programmeringsoppgavene er i den vanskelige enden av spekteret, men ikke kun: Den aller enkleste oppgaven var også programmering, samt noen i midtsjiktet. Den aller siste oppgaven (20) var ment å være spesielt vanskelig, for å skille A-kandidater fra resten. Kolonnen CORREL viser korrelasjonen mellom hver enkelt deloppgave og totalscore. Grønt indikerer høy korrelasjon (>0,6), lysegrønt 0,5-0,6 og gult og oransje de med korrelasjon hhv. 0,4-0,5 og <0,4. Kolonnen D viser oppgavens diskrimineringsindeks. D over 0,4 (grønn) er ansett som meget bra, 0,3-0,4 (lysegrønn) som brukbart, mens 0,2-0,3 og særlig <0,2 er sett på som tvilsomt, disse er derfor farget gul og rød. Kodekompletteringsoppgaver sett under ett (dvs. dra-og-slipp, nedtrekk og fyll-inn) hadde en intern korrelasjon på **0,37** mens programmeringsoppgaver hadde en intern korrelasjon på **0,50**. Korrelasjonen mellom kompletteringsoppgaver og programmeringsoppgaver var **0,42**, altså noe lavere enn den interne korrelasjonen for programmeringsoppgaver (mhp FS1), men ikke mye. Disse tallene er lavere enn det som ses i CORREL-kolonnen i tabellen, dette er fordi det er færre oppgaver som sammenlignes når man ser på slike underkategorier, dermed blir korrelasjonen redusert på grunn av tilfeldige utslag av prestasjoner på enkeltoppgaver, mens slike tilfeldige utslag gjør seg mindre gjeldende i en sammenligning med totalscore (alle 20 deloppgaver) som gjort i CORREL-kolonnen.

Tabell 2. Oppgavenes vanskegrad, korrelasjon med totalscore, og diskriminering

Nr	IA-Type	MEAN	STDEV	CORREL	D
14	Programmering	85	20	0,51	0,25
4	Kortsvar	82	24	0,59	0,36
7	Dra og slipp	81	22	0,58	0,33
2	Flervalg	74	19	0,56	0,27
15	Nedtrekk	73	20	0,54	0,26
13	Dra og slipp	71	30	0,41	0,34
1	Flervalg	69	20	0,33	0,16
8	Nedtrekk	65	26	0,59	0,36
6	Paring	62	35	0,59	0,53
16	Programmering	60	29	0,72	0,50
5	Flervalg	60	27	0,44	0,35
17	Programmering	58	30	0,67	0,51
19	Fyll inn tekst	58	28	0,63	0,43
3	Sant / usant	57	50	0,31	0,37
18	Programmering	55	31	0,74	0,57
10	Programmering	50	23	0,62	0,35
9	Nedtrekk	48	23	0,56	0,31
11	Dra og slipp	44	28	0,60	0,38
12	Programmering	43	29	0,67	0,47
20	Programmering	30	30	0,72	0,48

Tabell 3 ser spesielt på de «nye» programmeringsrelaterte oppgavetypene og hvordan disse korrelerer internt og eksternt. Vi har her delt dem i to kategorier, dra-og-slipp-oppgaver for seg (3 oppgaver) og nedtrekk + fyll-inn (3+1 oppgaver). Motivasjonen for å putte fyll-inn-tekst i bås med nedtrekk er at disse to oppgavetypene har mange fellestrekk, jfr. Fig 3 og Fig 4. Begge innebærer å fylle inn manglende fragmenter i gitt kode, forskjellen er bare at for nedtrekk får man alternativer å velge mellom, mens for fyll-inn må man skrive inn det som mangler uten noen hint. Dra-og-slipp (Fig 2) er derimot vesensforskjellig fra dette siden all koden er gitt, men at linjene må plasseres riktig i rutenettet. Som man kan se av de uthevede tallene er den interne korrelasjonen (hvordan disse oppgavene korrelerer med andre deloppgaver i samme bås) mindre enn korrelasjonen som de har med vanlige programmeringsoppgaver. De korrelerer også bedre med programmering enn med kodeforståelse og teori-oppgaver.

Tabell 3: Kodekompletteringsoppgaver, intern og ekstern korrelasjon

Sjanger	Intern korrelasjon	Vs. hver-andre	Vs. prog-rammering	Vs. kode-forståelse	Vs. teori
Dra-og-slipp	**0,36**	0,35	**0,37**	0,32	0,31
Nedtrekk/fyll-inn	**0,41**		**0,44**	0,33	0,34

Diskusjon og konklusjon

Det mest positive funnet er knyttet til FS2. Når korrelasjonen internt mellom dra-og-slipp-oppgaver viste seg å være mindre enn korrelasjonen disse har med programmeringsoppgaver, er det ingen ting som tyder på at det fins en særegen evne akkurat for å løse slike omstokkingsoppgaver, eller iallfall har ikke slike evner hatt noen nevneverdig innvirkning på resultatet, og tilsvarende funn ble også gjort for nedtrekk og

innfylling (jfr. Tabell 3). Dette betyr ikke at disse oppgavetypene er fullstendig fri for konstrukt-irrelevans –det er det knapt noen oppgavetype som er. For eksempel vil alle oppgaver hvor det fins en del tekst å lese, påvirkes av kandidatenes generelle leseferdigheter, og alle oppgaver hvor det er en del å skrive vil påvirkes av skriveferdigheter; for en papireksamen vil håndskrift spille en rolle, for digital eksamen hastighet på tastaturet. Det man imidlertid kan slutte, er at det iallfall ikke er noe tegn på høyere konstrukt-irrelevans for denne oppgavetypen enn for andre oppgavetyper.

Relatert til FS1 og FS3 ser vi at kompletteringsoppgavene jevnt over hadde brukbar korrelasjon med totalresultatet – men litt dårligere enn programmeringsoppgavene (FS1), og at de også jevnt over skilte noe dårligere enn programmeringsoppgavene (FS3), selv om D-verdiene var brukbare for de fleste oppgavene. Korrelasjonen mellom kompletteringsoppgaver og programmeringsoppgaver (0,42) er noe lavere enn vi kunne ha ønsket, da man helst bør over 0,5 for å snakke om en sterk korrelasjon [21] – særlig hvis man tenker seg å bruke kompletteringsoppgaver som delvis erstatning for programmeringsoppgaver for å måle de samme læringsutbyttene. (Dog kan det legges til at programmeringsoppgavers interne korrelasjon bare så vidt nådde 0,5 så forskjellen var ikke stor). En mulig forklaring relatert til FS1 kan være at kompletteringsoppgaver måler relaterte men likevel litt andre ferdigheter enn rene programmeringsoppgaver. En annen mulig forklaring kan være at det var den konkrete bruken av oppgavetypene dra-og-slipp og nedtrekk i den aktuelle eksamen som ga dårligere korrelasjon og D-verdier, ikke iboende egenskaper ved disse sjangrene som sådan. Som tabell 1 viser, bortsett fra en enkelt programmeringsoppgave som var spesielt lett (nr 14) var de øvrige i den vanskeligere halvdelen av spekteret, mens sjangrene dra-og-slipp og nedtrekk i større grad befant seg i den lettere halvdelen. For nedtrekk kan en årsak være at noen studenter har scoret en del poeng på gjetning, særlig siden det ikke ble gitt minuspoeng for gale svar. Likeledes vil dra-og-slipp-oppgaver kanskje gjøre det lettere for svake studenter å score noe poeng enn hva som vil være tilfelle med rene programmeringsoppgaver. Å tippe helt vilt i et 7x4 rutenett som vist i Figur 2 vil sjelden gi nevneverdig uttelling, men de færreste tipper helt vilt. Selv studenter med begrensede programmeringsferdigheter vil gjerne skjønne iallfall at funksjonshodet skal øverst og `return` nederst. I en oppgave som denne ville man da stå igjen med 5 linjer som må pusles i riktig rekkefølge mellom disse to, og så sant man setter på riktig marg, er det brukbare sjanser for å score noe på tipping. At score delvis påvirkes av gjetning vil bidra til å redusere korrelasjoner og gi lavere D-verdier. Som tabell 1 viste, skilte oppgaver med dra-og-slipp og nedtrekk jevnt over noe dårligere enn vanlige programmeringsoppgaver (jfr. forskningsspørsmål nr 3), mens den ene fyll-inn-tekst-oppgaven, hvor det ikke på samme måte er mulighet for tipping, hadde korrelasjon og D-verdi i det mørkegrønne området. Mulige forbedringstiltak for kommende eksamener er derfor:

- Gi minuspoeng for gale svar på nedtrekksoppgaver (f.eks. hvis det er tre alternative kodefragmenter å velge mellom, gi +1 for riktig svar, -0.33 for galt svar, 0 for intet svar). Vel å merke er dette et omdiskutert tema [22], hvor noen hevder at minuspoeng skaper flere problemer enn det løser (f.eks. at det straffer studenter som er mindre risikovillige), versus andre som mener at den økte påliteligheten er en langt større fordel enn disse ulempene.
- Gjøre (noen av) dra-og-slipp-oppgavene vanskeligere ved å innføre distraktorer, dvs. at i tillegg til kodelinjene som skal pusles på plass, står det flere kodelinjer som ikke skal brukes i det hele tatt. I øvinger tidlig i et emne vil dette kanskje ikke være anbefalt, da man har funnet at det kan gi redusert læringseffekt for nybegynnere [23], men dette behøver ikke forhindre at det kan brukes på en

eksamen ved semesterslutt. Dog bør man da også ha brukt det på noen øvinger mot slutten av semesteret, så studentene er kjent med formatet.

Hvis man gjør dra-og-slipp- og nedtrekksoppgaver vanskeligere, vil imidlertid eksamen totalt sett lett bli for vanskelig, med mindre man gjør andre tiltak i tillegg. Ett mulig tiltak kan derfor være å ha noe større innslag av enkle programmeringsrelaterte oppgaver tidlig i settet, i tråd med anbefalingen fra Zingaro et al. [14] at man bør starte med *«single concept questions»*, enten som kortsvar eller flervalg.

Selv om kompletteringsoppgaver korrelerer bra med programmeringsoppgaver, virker det ikke tilrådelig å gå helt bort fra oppgaver hvor studentene skal skrive kode som tilfredsstiller gitte krav. Det er denne ferdigheten de ultimat skal lære, og hvis det ikke ble testet på eksamen, ville en del studenter kanskje ikke prioritere dette lenger, men bare øve på innfyllingsoppgaver.

Når det gjelder erfaringer med selve Inspera-verktøyet, er en del av oppgavene ganske tidkrevende å lage, dette gjelder særlig dra-og-slipp-oppgaver hvor det i nåværende versjon av Inspera er svært fiklete å få til et pent rutenett. Her har imidlertid Kvannli og Jørgensen [24] laget et prototypverktøy for å generere slike oppgaver fra gitt løsningskode til QTI v.2.1-formatet som kan lastes opp til Inspera. Faglærer trenger derfor bare skrive løsningskoden i Python og auto-generere rutenettet med dra-objekter og ferdig fasit. Andre utfordringer er at man kunne ha ønsket mulighet til å legge til mer fleksible poengscoringsregler for dra-og-slipp-oppgaver. På sikt vil det være en klar fordel om eksamensverktøy som Inspera og WiseFlow gir bedre støtte for programmeringsoppgaver, både i studenters utførelse av eksamen (integrert kodeeditor, mulighet for å kjøre og teste) og i faglæreres retting (mulighet for autoretting ved testsuiter samt andre typer automatisk analyse av kode). Åpne arkitekturer med godt dokumenterte API'er og en tankegang i retning av *digitale økosystemer* [25, 26] kunne lettere gjøre det mulig for universitetsmiljøer å lage egne plug-ins for spesielle behov, heller enn å vente på at en ønsket utvidelse kommer høyt nok opp på prioriteringslisten til leverandøren.

Referanser

1. Biggs, J., *Constructive alignment in university teaching.* HERDSA Review of higher education, 2014. **1**(1): p. 5-22.
2. Sheard, J., et al., *Assessment of programming: pedagogical foundations of exams*, in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 2013, ACM: Canterbury, England, UK. p. 141-146.
3. Haladyna, T.M. and S.M. Downing, *Construct-irrelevant variance in high-stakes testing.* Educational Measurement: Issues and Practice, 2004. **23**(1): p. 17-27.
4. Spurgeon, S.L., *Evaluating the unintended consequences of assessment practices: Construct irrelevance and construct underrepresentation.* Measurement and Evaluation in Counseling and Development, 2017. **50**(4): p. 275-281.
5. Hillier, M. and A. Fluck, *Arguing again for e-exams in high stakes examinations.* Electric dreams. proceedings ascilite, 2013: p. 385-396.
6. Sindre, G. and A. Vegendla, *E-exams and exam process improvement*, in *UDIT 2015*. 2015, Bibsys OJS: Ålesund.
7. Fluck, A. and M. Hillier. *Innovative assessment with eExams*. in *Australian Council for Computers in Education Conference, Brisbane*. 2016.
8. Petersen, A., M. Craig, and D. Zingaro. *Reviewing CS1 exam question content*. in *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011. ACM.

9. Simon, et al. *Introductory programming: examining the exams*. in *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*. 2012. Australian Computer Society, Inc.
10. Denny, P., A. Luxton-Reilly, and B. Simon. *Evaluating a new exam question: Parsons problems*. in *Proceedings of the fourth international workshop on computing education research*. 2008. ACM.
11. Parsons, D. and P. Haden. *Parson's programming puzzles: a fun and effective learning tool for first programming courses*. in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 2006. Australian Computer Society, Inc.
12. Collofello, J. *Teaching practical software maintenance skills in a software engineering course*. in *Proc. Twentieth SIGCSE Technical Symposium on Computer Science Education*. 1989. ACM.
13. Van Merriënboer, J.J. and M.B. De Croock, *Strategies for computer-based programming instruction: Program completion vs. program generation.* Journal of Educational Computing Research, 1992. **8**(3): p. 365-394.
14. Zingaro, D., A. Petersen, and M. Craig. *Stepping up to integrative questions on CS1 exams*. in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012. ACM.
15. Luxton-Reilly, A. and A. Petersen. *The compound nature of novice programming assessments*. in *Proceedings of the Nineteenth Australasian Computing Education Conference*. 2017. ACM.
16. Ihantola, P. and V. Karavirta, *Two-dimensional parson's puzzles: The concept, tools, and first observations.* Journal of Information Technology Education, 2011. **10**: p. 119-132.
17. Tavakol, M. and R. Dennick, *Post-examination analysis of objective tests.* Medical Teacher, 2011. **33**(6): p. 447-458.
18. Oosterhof, A.C., *Similarity of various item discrimination indices.* Journal of Educational Measurement, 1976: p. 145-150.
19. Kelley, T.L., *The selection of upper and lower groups for the validation of test items.* Journal of educational psychology, 1939. **30**(1): p. 17.
20. Pyrczak, F., *VALIDITY OF THE DISCRIMINATION INDEX AS A MEASURE OF ITEM QUALITY 1*. Journal of Educational Measurement, 1973. **10**(3): p. 227-231.
21. Cohen, J., *Statistical Power Analysis for the Behavioural Sciences (2nd ed.)*. 2nd ed. 1988, Hillsdale, NJ: Lawrence Earlbaum and Associates.
22. Lesage, E., M. Valcke, and E. Sabbe, *Scoring methods for multiple choice assessment in higher education–Is it still a matter of number right scoring or negative marking?* Studies in Educational Evaluation, 2013. **39**(3): p. 188-193.
23. Harms, K.J., J. Chen, and C.L. Kelleher. *Distractors in Parsons problems decrease learning efficiency for young novice programmers*. in *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 2016. ACM.
24. Jørgensen, J. and S. Kvannli, *Efficient generation of Parsons problems for digital programming exams in Inspera*, in *Department of Computer Science*. 2019, NTNU: Trondheim.
25. Uden, L., I.T. Wangsa, and E. Damiani. *The future of E-learning: E-learning ecosystem*. in *Digital EcoSystems and Technologies Conference, 2007. DEST'07. Inaugural IEEE-IES*. 2007. IEEE.
26. Behrens, J.T. and K.E. DiCerbo, *Technological implications for assessment ecosystems: Opportunities for digital technology to advance assessment*. 2013.