Erkki Mäkinen (toim.)

Tietojenkäsittelytieteellisiä tutkielmia
Talvi 2017

TAMPEREEN YLIOPISTO

INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 53/2017

TAMPERE 2017

TAMPEREEN YLIOPISTO
INFORMAATIOTIETEIDEN YKSIKÖN RAPORTTEJA 53/2017
MAALISKUU 2017

Erkki Mäkinen (toim.)

Tietojenkäsittelytieteellisiä tutkielmia
Talvi 2017

ISBN 978-952-03-0406-5 (pdf)

ISSN-L 1799-8158
ISSN 1799-8158

K-lähimmän naapurin menetelmä ja sen muunnelmat

Pekka Rämö

Tiivistelmä.
K-lähimmän naapurin menetelmä on tiedonlouhinnassa käytetty yksinkertainen luokittelumenetelmä. Tässä tutkielmassa tarkastellaan alkuperäistä algoritmia sekä muutamaa sen muunnelmista.

Avainsanat ja -sanonnat: K-lähimmän naapurin menetelmä, Lähimmän naapurin menetelmä, k-nearest neighbor, KNN, LMKNN, PNN, LMPNN.

1. Johdanto

K-lähimmän naapurin menetelmä eli KNN (k-nearest neighbor method) [Cover and Hart 1967] on yksinkertainen instanssiperusteinen luokittelumenetelmä, jota hyödynnetään koneoppimisessa. Yksinkertaisuudestaan huolimatta tai siitä johtuen se on luokiteltu 10 parhaan tiedonlouhintamenetelmän joukkoon [Wu *et al.* 2008] ja siitä on luotu lukuisia muunnelmia.

KNN on ns. *laiska oppija* (lazy learner), joka tarvitsee luokitteluun vain arvon k, etäisyysmitan (distance measure) d ja opetusaineiston (training set) T. Lisäksi useat muunnelmat käyttävät luokittelussa painoa W, jolla priorisoidaan naapureita. Instanssit nähdään pisteinä euklidisessa avaruudessa, ja uuden instanssin luokitus tapahtuu etsimällä instanssin k lähintä naapuria ja laskemalla niiden enemmistöluokka. Etäisyysmittana käytetään yleensä *euklidista etäisyyttä* (Euclidean distance), mutta joissain tapauksissa, kuten dokumenttien luokittelussa, *kosinimitta* (cosine measure) on parempi vaihtoehto [Wu *et al.* 2008].

Vaikka k-lähimmän naapurin menetelmä on itsessään todella tunnettu, niin monet sen muunnelmista eivät ole. Tämän tutkielman tarkoituksena on tutustua k-lähimmän naapurin menetelmään ja muutamiin sen muunnelmista sekä vertailla näiden algoritmien luokittelutarkkuuksia ja suoritusaikoja. Tutkielman rakenne on seuraava: luvussa 2 käydään läpi aiheeseen liittyviä käsitteitä, luvussa 3 esitellään k-lähimmän naapurin menetelmän ja sen muunnelmien toimintaperiaatteet, luvussa 4 vertaillaan algoritmien luokittelutarkkuuksia ja suoritusaikoja ja luvussa 5 on yhteenveto tutkielmasta.

2. Käsitteitä

2.1. Laiska ja ahkera oppija

Koneoppimismenetelmät voidaan jakaa laiskoihin ja ahkeriin oppijoihin niiden opetus- ja luokitteluvaiheiden perusteella. Laiskojen oppijoiden opetusvaihe

koostuu pelkästään opetusaineiston tallentamisesta muistiin ja varsinainen laskenta suoritetaan vasta luokitteluvaiheessa. Luokitteluvaiheessa luokiteltavaa instanssia verrataan kaikkiin opetusaineiston instansseihin ja uusi luokka ennustetaan näiden perusteella. Laiskojen oppijoiden etuna on nopea opetusvaihe ja haittapuolena hidas luokitteluvaihe. K-lähimmän naapurin menetelmä on yksi esimerkki laiskasta oppijasta.

Ahkerat oppijat sen sijaan rakentavat opetusvaiheessa mallin, jonka avulla luokitus suoritetaan luokitteluvaiheessa. Ahkerien oppijoiden etuna on nopea luokitteluvaihe ja haittapuolena hidas opetusvaihe. Ahkeria oppijoita ovat esimerkiksi päätöspuut (decision trees) ja sääntöpohjaiset luokittelijat (rule-based classifiers).

2.2. Euklidinen etäisyys

Euklidinen etäisyys d_e on kahden pisteen välinen etäisyys euklidisessa avaruudessa. Tasossa eli kaksiulotteisessa avaruudessa pisteiden $p = (p_1, p_2)$ ja $q = (q_1, q_2)$ euklidinen etäisyys voidaan laskea Pythagoran lausetta hyödyntäen:

$$d_e(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

N-ulotteisessa avaruudessa pisteiden $p = (p_1, p_2, \dots, p_n)$ ja $q = (q_1, q_2, \dots, q_n)$ euklidinen etäisyys voidaan laskea kaavalla

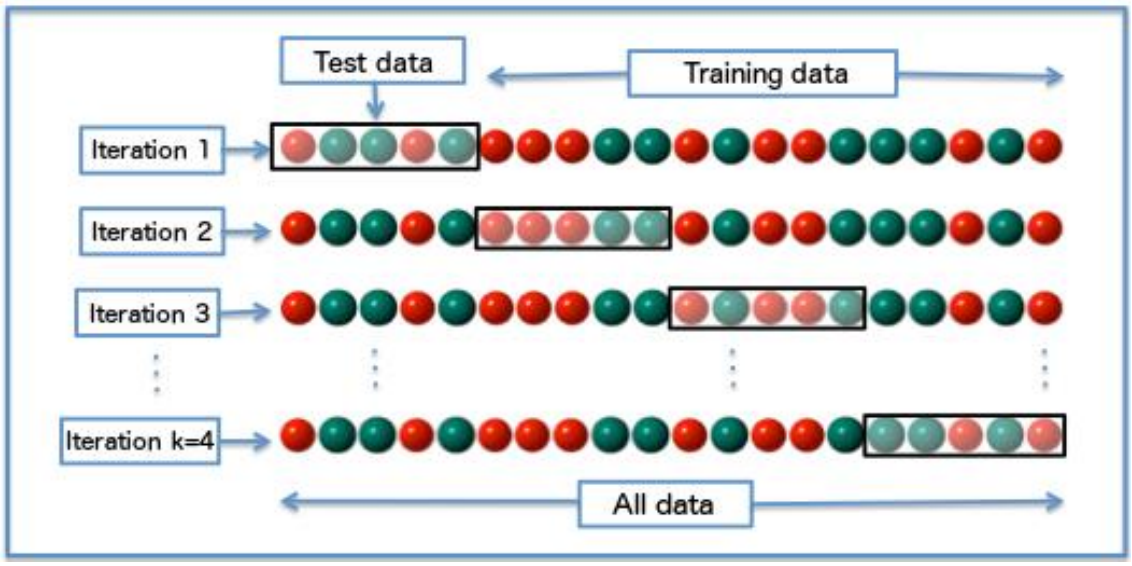
$$d_e(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2},$$

jossa n on koordinaattien eli ulottuvuuksien lukumäärä, p_i on pisteen p arvo koordinaatissa i ja q_i on pisteen q arvo koordinaatissa i.

2.3. K-kertainen ristiinvalidointi

K-kertainen ristiinvalidointi on suosittu validointimenetelmä, jolla voidaan arvioida luokittelumenetelmän tarkkuutta. Alkuperäinen opetusaineisto jaetaan k osajoukkoon niin, että yksi osajoukko toimii testausaineistona ja loput $k - 1$ osajoukkoa opetusaineistona. Testausaineiston instanssit luokitellaan opetusaineiston instanssien perusteella ja luokittelutarkkuus arvioidaan vertaamalla luokittelumenetelmän ennustamia luokkia instanssien todellisiin luokkiin. Tämä toimenpide toistetaan k kertaa, jolloin jokainen opetusaineiston instanssi tulee luokitelluksi täsmälleen yhden kerran. Lopullinen luokittelutarkkuus saadaan laskemalla saatujen luokittelutarkkuuksien keskiarvo.

Kuvassa 1 [Wikipedia 2017] on esimerkki k-kertaisesta ristiinvalidoinnista arvolla $k = 4$. Laatikoiden sisällä olevia instansseja käytetään testausaineistona ja loput aineistosta toimii opetusaineistona. Kuvasta näkee selkeästi sen, että jokainen instanssi tulee luokitelluksi yhden kerran.



Kuva 1. K-kertainen ristiinvalidointi arvolla k=4.

Jos opetusaineiston luokkajakauma on hyvin epätasainen, on suositeltua käyttää ositettua otantaa opetusaineiston jakamisessa. Tämä varmistaa sen, että suhteellinen luokkajako on suurin piirtein sama joka jaossa. Lisäksi, jos opetusaineisto sisältää instanssit jossain tietyssä järjestyksessä (esim. luokittain), on opetusaineiston järjestys hyvä sekoittaa ennen ristiinvalidoinnin tekemistä.

10-kertainen ristiinvalidointi on k-kertaisen ristiinvalidoinnin suosituin muoto, mutta myös muita k:n arvoja voidaan käyttää. Kun k:n arvona käytetään opetusaineiston instanssien lukumäärää, kyseessä on yksi-pois ristiinvalidointi, jossa jokainen instanssi luokitellaan koko opetusaineiston (pl. luokiteltava instanssi) perusteella. Yksi-pois ristiinvalidointia suositellaan käytettäväksi pienten aineistojen yhteydessä, missä opetusjoukon koko halutaan maksimoida.

3. Algoritmit

Tässä luvussa esitellään k-lähimmän naapurin menetelmän sekä muunnelmien LMKNN [Mitani and Hamamoto 2006], PNN [Zeng *et al.* 2009] ja LMPNN [Gou *et al.* 2014] toimintaperiaatteet. K-lähimmän naapurin menetelmän toimintaperiaate on esitelty samalla tavalla kuin Wu ja muut [2008] sen esittivät ja muunnelmat on esitelty käyttäen Gou ja muiden [2014] esitystapaa.

3.1. KNN

Olkoot T opetusaineisto ja luokiteltava instanssi $x = (x', \omega')$, jossa x' on luokiteltavan instanssin data ja ω' on sen luokka. Opetusaineisto koostuu instansseista $(x, \omega) \in T$, jossa x on instanssin data ja ω on instanssin luokka. Uuden instanssin x luokittelu tapahtuu seuraavasti:

1. Laske etäisyys x:n ja kaikkien opetusaineiston instanssien $(x, \omega) \in T$ välillä. Olkoon T_z osajoukko, joka sisältää instanssin x k lähintä naapuria.

2. Aseta luokiteltavan instanssin luokaksi lähimpien naapureiden enemmistöluokka:

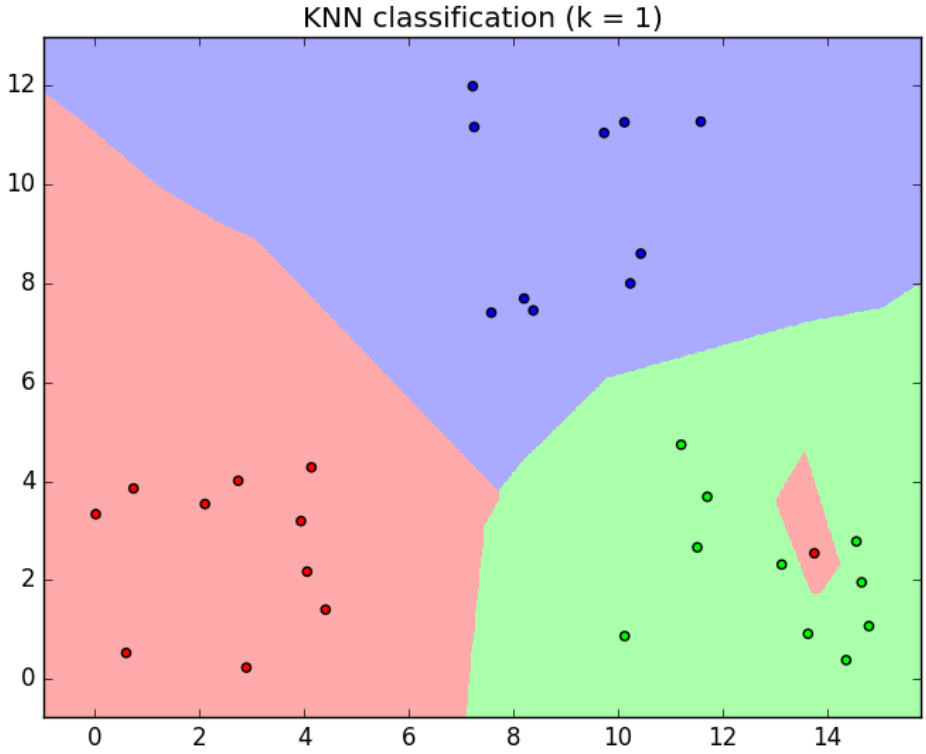
$$\omega' = \underset{v}{argmax} \sum_{(x_i, \omega_i) \in T_z} I(v = \omega),$$

jossa v on luokka, ω_i on i:n lähimmän naapurin luokka ja $I(\cdot)$ on indikaattorifunktio, joka palauttaa arvon 1, jos argumentti on tosi ja 0 muulloin.

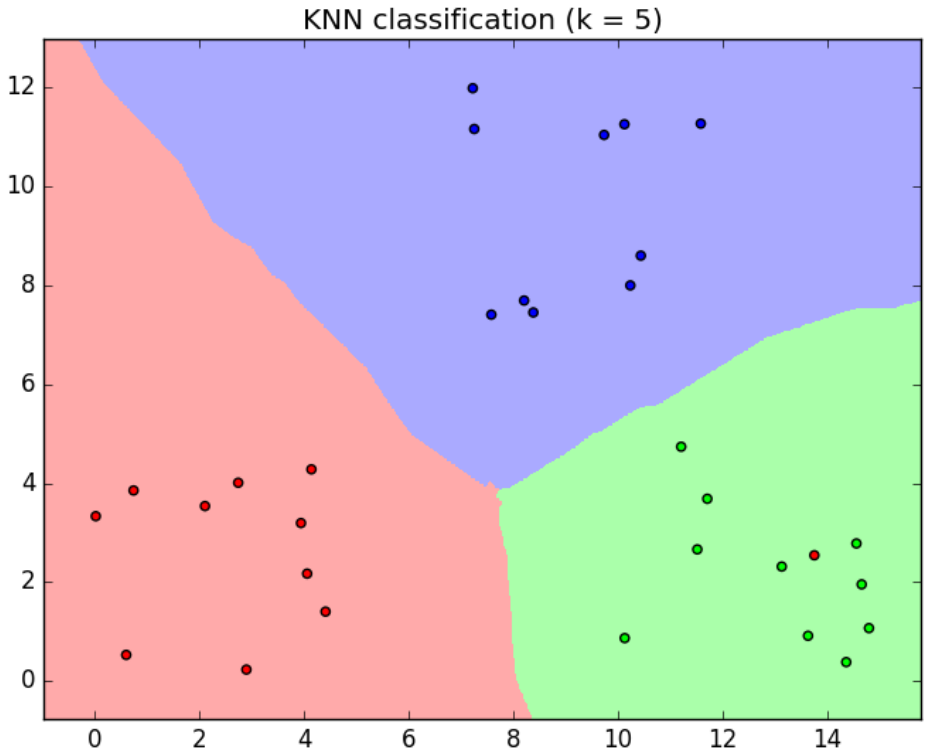
Kun $k = 1$, kyseessä on KNN:n erikoistapaus 1-NN eli lähimmän naapurin menetelmä (nearest neighbor method) [Cover and Hart 1967]. 1-NN suorittaa luokittelun pelkästään lähimmän naapurin perusteella ja asettaa uuden instanssin luokaksi sen lähimmän naapurin luokan. Vaikka 1-NN on vain yksi KNN:n erikoistapaus, sen nähdään myös olevan oma menetelmänsä.

Yksi KNN:n isoimmista ongelmista on arvon k määrittäminen. Liian pieni arvo altistaa luokittelun kohina-arvoille (noise values) ja liian suuri arvo sisällyttää liikaa instansseja muista luokista lähimpien naapurien joukkoon. Kuvissa 2, 3 ja 4 on esiteltynä luokkakohtaiset rajat, kun k:n arvot ovat 1, 5 ja n (opetusaineiston koko). Opetusaineiston instansseilla on kolme luokkaa (punainen, vihreä ja sininen) ja instanssit ovat piirrettyinä erivärisinä pisteinä kuviin.

Kuvasta 2 on helppo huomata, kuinka poikkeava havainto vaikuttaa luokitukseen k:n arvon ollessa pieni. Poikkeava havainto luo pienen luokittelualueen ympärilleen, joka voi aiheuttaa vääriä luokituksia. Kuvassa 3 k:n arvoa on kasvatettu, joka mitätöi poikkeavan havainnon vaikutuksen.

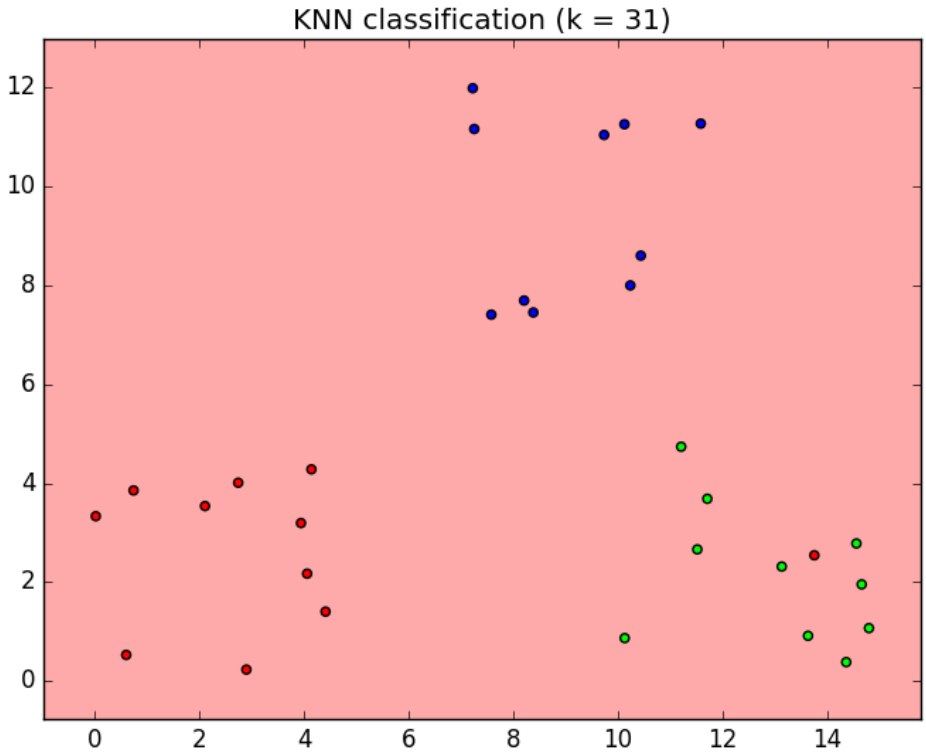


Kuva 2. KNN luokituksen luokkakohtaiset rajat arvolla k=1.



Kuva 3. KNN luokituksen luokkakohtaiset rajat arvolla k=5.

Mikäli k:n arvo on liian suuri, luokat, joiden instansseja on eniten opetusaineistossa, alkavat dominoida luokitusta. Äärimmäinen esimerkki tästä on kuvan 4 tilanne, jossa k:n arvo on sama kuin opetusaineiston koko. Tässä tilanteessa uusi instanssi saa luokakseen aina opetusaineiston enemmistöluokan.



Kuva 4. KNN luokituksen luokkakohtaiset rajat arvolla k=n.

Arvon k määrittämiseen ei ole mitään yksiselitteisesti parasta tapaa, mutta apuna voidaan käyttää esimerkiksi k-kertaista ristiinvalidointia vertailemalla luokittelutarkkuuksia eri k:n arvoilla (naapurien lukumäärillä) ja valitsemalla niistä paras.

Toinen iso ongelma on se, että kaikki naapurit ovat yhtä tärkeitä luokittelussa, vaikka menetelmän toimintaperiaatteen mukaisesti lähempänä olevien naapurien voidaan olettaa olevan tärkeämpiä luokittelun kannalta. Tätä ongelmaa varten menetelmästä kehitettiin painotettuja muunnelmia, joista etäisyyspainotettu k-lähimmän naapurin menetelmä (distance-weighted k-nearest neighbor, WKNN) [Dudani 1976] oli ensimmäinen. Painotetuissa muunnelmissa naapureille annetaan paino W, jota käytetään hyväksi enemmistöluokkaa laskiessa. Dudanin omassa versiossa WKNN johti parempaan luokittelutarkkuuteen, mutta joissain muissa tutkimuksissa WKNN ei ollut alkuperäistä KNN algoritmia parempi [Bailey and Jain 1978]. Bailey ja Jain [1978] lisäksi osoittivat, että opetusaineiston instanssien lukumäärän lähestyessä ääretöntä, alkuperäisen

lähimmän naapurin menetelmän luokittelutarkkuus on parempi kuin minkään painotetun version.

Muita KNN:n käyttöön liittyviä ongelmia ovat nk. dimensionaalisuuden kirous (the curse of dimensionality) ja vaihtelut attribuuttien arvoalueissa. Dimensionaalisuuden kirouksella viitataan siihen, että instanssien dimensioiden eli ulottuvuuksien lukumäärän kasvaessa suureksi kaikki instanssit ovat euklidisella etäisyydellä mitattuna kaukana toisistaan, joka tekee luokittelusta vaikeaa. K-lähimmän naapurin menetelmän muunnelmat eivät niinkään koeta ratkoa tätä ongelmaa, vaan suositeltu tapa on karsia turhat attribuutit pois opetusaineistosta ennen luokituksen suorittamista.

Vaihtelut attribuuttien arvoalueissa voivat taas aiheuttaa sen, että yksi attribuutti dominoi luokitusta yli muiden. Jos attribuutteja ovat esimerkiksi ihmisen pituus (cm) ja paino (g), on painojen arvoalue laajempi ja instanssien etäisyyttä laskiessa painon vaikutus on pituutta suurempi. Tätäkään ongelmaa ei yritetä ratkoa erilaisten muunnelmien tai etäisyysmittojen avulla, vaan yleinen käytäntö on normalisoida attribuutit ennen luokituksen suorittamista.

3.2. LMKNN

Olkoon $T = \{x_n \in \mathbb{R}^d\}_{n=1}^{N}$ opetusaineisto, jossa N on instanssien lukumäärä ja d attribuuttien lukumäärä. Instanssien luokkia on M kappaletta ja niistä käytetään merkintää $c_n \in \{\omega_1, \omega_2, \dots, \omega_M\}$. Olkoon $T_{\omega_i} = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^{N_i}$ osajoukko, joka koostuu opetusaineiston T instansseista, joiden luokka on ω_i ja joita on N_i kappaletta. Uuden instanssin x luokittelu tapahtuu seuraavasti:

1. Etsi k lähintä instanssia osajoukosta T_{ω_i} jokaiselle luokalle ω_i. Instanssien etäisyys toisistaan lasketaan käyttämällä euklidista etäisyyttä. Olkoon $T_{\omega_i}^k(x) = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^{k}$ osajoukko, joka sisältää uuden instanssin x lähimmät naapurit luokasta ω_i.

2. Laske paikallinen keskiarvovektori (local mean vector) $u_{\omega_i}^k$ jokaiselle luokalle ω_i osajoukon $T_{\omega_i}^k(x)$ avulla kaavalla
$$u_{\omega_i}^k = \frac{1}{k}\sum_{j=1}^{k} x_j^i \,.$$

3. Laske etäisyys $d(x, u_{\omega_i}^k)$ instanssin x ja luokkien paikallisten keskiarvovektorien välillä.

4. Instanssi x saa luokakseen lähimmän paikallisen keskiarvovektorin luokan

$$c_x = \underset{\omega_i}{argmin}\, d(x, u_{\omega_i}^k).$$

LMKNN eli paikalliseen keskiarvoon perustuva k-lähimmän naapurin menetelmä (local mean-based k-nearest neighbor) on yksinkertainen muunnelma alkuperäisestä algoritmista. Ideana on vähentää poikkeavien havaintojen vaikutusta luokitukseen ja näin parantaa luokittelutarkkuutta. Lähimpien naapureiden enemmistöluokan sijaan muunnelma laskee jokaisen luokan k lähimmästä naapurista paikallisen keskiarvovektorin, ja uusi instassi saa luokakseen lähimmän paikallisen keskiarvovektorin luokan.

Mitani ja Hamamoto [2006] mainitsevat, että LMKNN:n vakautta poikkeavia havaintoja kohtaan on vaikea todistaa teoreettisesti, mutta empiirisissä tutkimuksissa LMKNN osoittautui vakaaksi niitä kohtaan. Lisäksi, alkuperäisissä kokeissa LMKNN oli useimmiten parempi kuin 1-NN-, KNN-, ED-, Parzen- ja ANN-algoritmit luokittelutarkkuudessa opetusaineiston koosta tai attribuuttien lukumäärästä riippumatta [Mitani and Hamamoto 2006].

3.3. PNN

Olkoon $T = \{x_n \in \mathbb{R}^d\}_{n=1}^N$ opetusaineisto, jossa N on instanssien lukumäärä ja d attribuuttien lukumäärä. Instanssien luokkia on M kappaletta ja niistä käytetään merkintää $c_n \in \{\omega_1, \omega_2, \dots, \omega_M\}$. Olkoon $T_{\omega_i} = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^{N_i}$ osajoukko, joka koostuu opetusaineiston T instansseista, joiden luokka on ω_i ja joita on N_i kappaletta. Uuden instanssin x luokittelu tapahtuu seuraavasti:

1. Etsi k lähintä instanssia osajoukosta T_{ω_i} jokaiselle luokalle ω_i. Instanssien etäisyys toisistaan lasketaan käyttämällä euklidista etäisyyttä. Olkoon $T_{\omega_i}^k(x) = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^k$ osajoukko, joka sisältää uuden instanssin x lähimmät naapurit luokasta ω_i. Olkoot niiden etäisyydet nousevassa järjestyksessä $d\left(x, x_1^i\right), d\left(x, x_2^i\right), \dots, d\left(x, x_{k-1}^i\right)$ ja $d\left(x, x_k^i\right)$.

2. Aseta lähimmille naapureille painot niiden etäisyyksien perusteella niin, että j:nnen lähimmän naapurin x_j^i paino luokassa ω_i on:
$$W_j^i = \frac{1}{j} \quad j = 1, \dots, k.$$

3. Etsi pseudolähimmät naapurit jokaiselle luokalle ω_i. Olkoon x_i^{PNN} luokan ω_i pseudolähin naapuri instanssille x. Etäisyys instanssin x ja pseudolähimmän naapurin välillä lasketaan kaavalla
$$d(x, x_i^{PNN}) = W_1^i \times d\left(x, x_1^i\right) + W_2^i \times d\left(x, x_2^i\right) + \cdots + W_k^i \times d\left(x, x_k^i\right).$$

4. Aseta uuden instanssin x luokaksi lähimmän pseudolähimmän naapurin luokka:

$$c_x = \underset{\omega_i}{argmin}\, d(x, x_i^{PNN}).$$

PNN eli pseudolähimmän naapurin menetelmä (pseudo nearest neighbor) pohjautuu WKNN ja LMKNN muunnelmiin. Jokaisesta luokasta etsitään k lähintä naapuria, joiden painotetut etäisyydet lasketaan yhteen, ja uusi instanssi saa luokakseen pienimmän summan tuottavat luokan. Huomattava on, että PNN käyttää erilaista painoa kuin WKNN ja muunnelmien samankaltaisuus tulee vain siitä, että molemmat käyttävät painoja luokituksen apuna.

Alkuperäisissä kokeissa PNN oli useimmiten parempi kuin KNN- ja WKNN-algoritmit. Suuria opetusaineistoja käytettäessä PNN oli myös parempi kuin LMKNN-algoritmin, mutta pieniä opetusaineistoja käytettäessä ei. Zeng ja muut [2009] huomauttavat, että PNN:än suorituskykyä pieniä opetusaineistoja käytettäessä tulisi tutkia enemmän.

3.4. LMPNN

Olkoon $T = \{x_n \in \mathbb{R}^d\}_{n=1}^N$ opetusaineisto, jossa N on instanssien lukumäärä ja d attribuuttien lukumäärä. Instanssien luokkia on M kappaletta ja niistä käytetään merkintää $c_n \in \{\omega_1, \omega_2, \dots, \omega_M\}$. Olkoon $T_{\omega_i} = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^{N_i}$ osajoukko, joka koostuu opetusaineiston T instansseista, joiden luokka on ω_i ja joita on N_i kappaletta. Uuden instanssin x luokittelu tapahtuu seuraavasti:

1. Etsi k lähintä instanssia osajoukosta T_{ω_i} jokaiselle luokalle ω_i. Instanssien etäisyys toisistaan lasketaan käyttämällä euklidista etäisyyttä. Olkoon $T_{\omega_i}^k(x) = \left\{x_j^i \in \mathbb{R}^d\right\}_{j=1}^k$ osajoukko, joka sisältää uuden instanssin x lähimmät naapurit luokasta ω_i etäisyyksien perusteella asetettuna nousevaan järjestykseen.

2. Laske paikallinen keskiarvovektori $\bar{x}_j^i$ jokaisen luokan j lähimmälle naapurille kaavalla

$$\bar{x}_j^i = \frac{1}{j}\sum_{i=1}^{j} x_j^i.$$

Olkoon $\bar{T}_{\omega_i}^k(x) = \left\{\bar{x}_j^i \in \mathbb{R}^d\right\}_{j=1}^k$ osajoukko, joka sisältää k paikallista keskiarvovektoria luokalle ω_i ja olkoot $d\left(x, \bar{x}_1^i\right), d\left(x, \bar{x}_2^i\right), \dots, d\left(x, \bar{x}_{k-1}^i\right)$ ja $d(x, \bar{x}_k^i)$ niiden etäisyydet instanssista x.

3. Aseta painot paikallisille keskiarvovektoreille niiden etäisyyksien perusteella. J:nnen lähimmän paikallisen keskiarvovektorin $\bar{x}_j^i$ paino $\bar{W}_j^i$ lasketaan kaavalla

$$\bar{W}_j^i = \frac{1}{j} \quad j = 1, \dots, k.$$

4. Etsi paikallisiin keskiarvovektoreihin perustuvat pseudolähimmät naapurit jokaiselle luokalle ω_i. Olkoon $\bar{x}_i^{PNN}$ luokan ω_i paikallisiin keskiarvovektoreihin perustuva pseudolähin naapuri instanssille x. Etäisyys x:n ja $\bar{x}_i^{PNN}$:n välillä lasketaan kaavalla

$$d(x, \bar{x}_i^{PNN}) = \bar{W}_1^i \times d\left(x, \bar{x}_1^i\right) + \bar{W}_2^i \times d\left(x, \bar{x}_2^i\right) + \cdots + \bar{W}_k^i \times d\left(x, \bar{x}_k^i\right).$$

5. Aseta uuden instanssin x luokaksi lähimmän paikallisiin keskiarvovektoreihin perustuvan pseudolähimmän naapurin luokka:

$$c = \underset{\omega_i}{argmin}\, d(x, \bar{x}_i^{PNN}).$$

LMPNN eli paikalliseen keskiarvoon perustuva pseudolähimmän naapurin menetelmä on yhdistelmä aiemmin esitellyistä LMKNN- ja PNN-muunnelmista. Toimintaperiaatteeltaan se on lähes identtinen PNN:n kanssa, mutta pseudolähimpiä naapureita laskiessa käytetään paikallisia keskiarvovektoreita tavallisten naapurien sijasta. Alkuperäisissä kokeissa LMPNN tuotti parhaan luokittelutarkkuuden keskiarvon ja oli parempi kuin KNN-, WKNN-, LMKNN-, CFKNN- ja PNN-algoritmit [Gou *et al.* 2014].

4. Algoritmien vertailu

Algoritmien vertailu toteutettiin Python-ohjelmointikielellä hyödyntäen scikit-learn-koneoppimiskirjastoa [Pedregosa *et al.* 2011]. Opetusaineistoina käytettiin scikit-learnin sisältämiä Iris- [Lichman 2013] ja Digits-aineistoja [Lichman 2013]. Algoritmien toteutukset ohjelmoitiin itse tätä tutkielmaa varten, mutta 10-kertainen ristiinvalidointi tehtiin scikit-learnin sisältämällä toteutuksella.

4.1. Aineistot

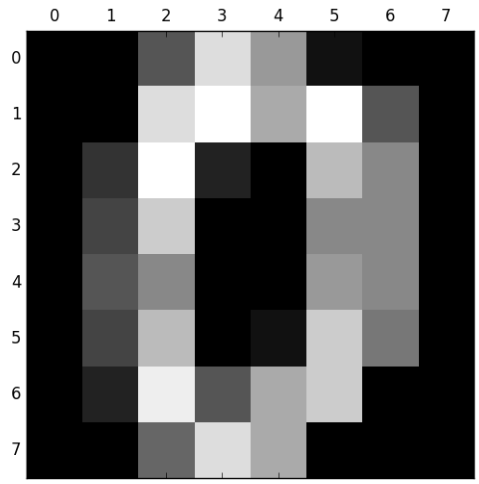
Tässä tutkielmassa käytetyt kaksi aineistoa ovat hyvin erilaisia kooltaan sekä koostumukseltaan. Taulukossa 1 on esiteltynä joitain ominaisuuksia käytetyistä aineistoista.

Aineisto	Instanssien lkm	Attribuuttien lkm	Luokkien lkm
Iris	150	4	3
Digits	1797	64	10

Taulukko 1. Tietoja käytetyistä aineistoista.

Iris-aineisto on luokittelumenetelmien kanssa hyvin usein käytetty aineisto, joka sisältää tiedot 150 kukan lehtien pituuksista. Kukkien alalajeja eli instanssien luokkia on kolme ja jokaisesta luokasta on 50 instanssia aineistossa. Attribuutteja on yhteensä neljä ja ne ilmoittavat verholehden ja terälehden pituuden ja leveyden senttimetreinä.

Digits-aineisto koostuu 1797 8x8-resoluutioisesta kuvasta, jotka esittävät käsinpiirrettyjä numeroita. Luokittelussa kuvia käsitellään 64-ulotteisina piirrevektoreina (feature vector), jossa jokainen attribuutti sisältää kokonaislukuarvon väliltä 0-16. Kokonaislukuarvo kertoo pikselin tummuuden siten, että 0 tarkoittaa täysin mustaa ja 16 täysin valkoista pikseliä. Kuvassa 5 on esimerkki Digits-aineiston sisältämästä kuvasta.



Kuva 5. Esimerkkikuva Digits-aineiston instanssista.

4.2. Luokittelutarkkuudet

Luokittelutarkkuuksia arvioitiin 10-kertaisen ristiinvalidoinnin avulla. Aineistot sekoitettiin ennen ristiinvalidoinnin suorittamista, sillä Iris-aineisto sisälsi instanssit luokkien mukaan järjestettynä ja Digits-aineiston instanssien järjestyksestä ei ollut tietoa. Attribuuttien normalisointia ei suoritettu, koska attribuuttien arvoalueissa ei ollut suuria eroja kummassakaan aineistossa.

Laskenta suoritettiin käyttäen samaa ristiinvalidointijakoa jokaisen algoritmin kohdalla. Luokittelutarkkuus laskettiin kaavalla

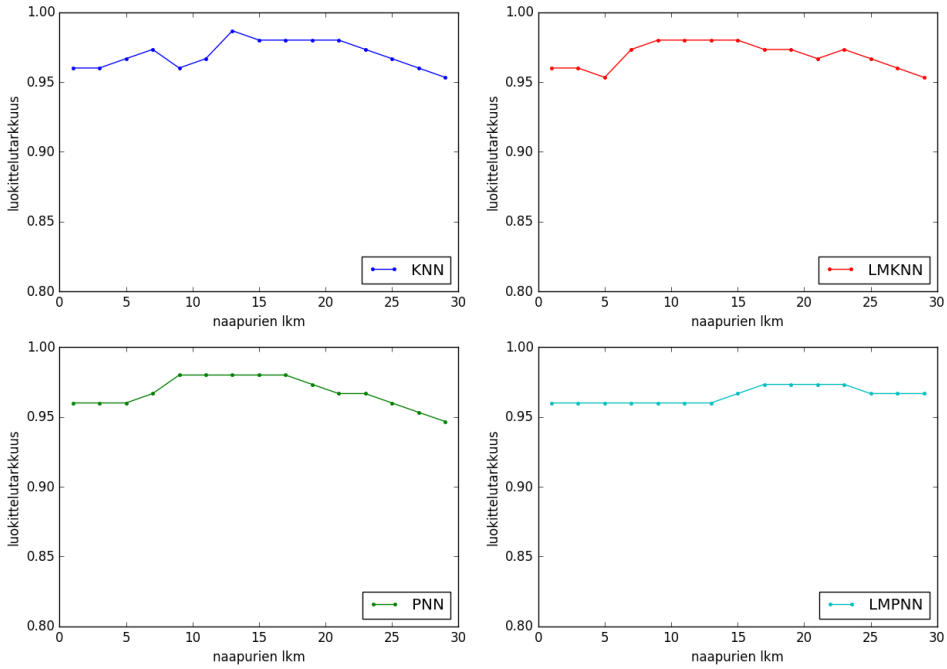
$$luokittelutarkkuus = \frac{oikein\ luokitellut\ instanssit}{kaikki\ instanssit},$$

missä $kaikki\ instanssit$ tarkoittaa testiaineiston instanssien lukumäärää ja $oikein\ luokitellut\ instanssit$ niiden testiaineiston instanssien lukumäärää, jotka algoritmi luokitteli oikein. Saadut tulokset pyöristettiin neljän desimaalin tarkkuuteen ja lopullinen luokittelutarkkuus saatiin laskemalla ristiinvalidointijakojen tuottamien tarkkuuksien keskiarvot. Tarkkuudet laskettiin 15 eri k:n arvolla siten, että eri k:n arvoilla laskemiset suoritettiin samassa ristiinvalidointijaossa. Taulukossa 2 on lueteltuna Iris- ja Digits-aineistojen luokittelutarkkuudet.

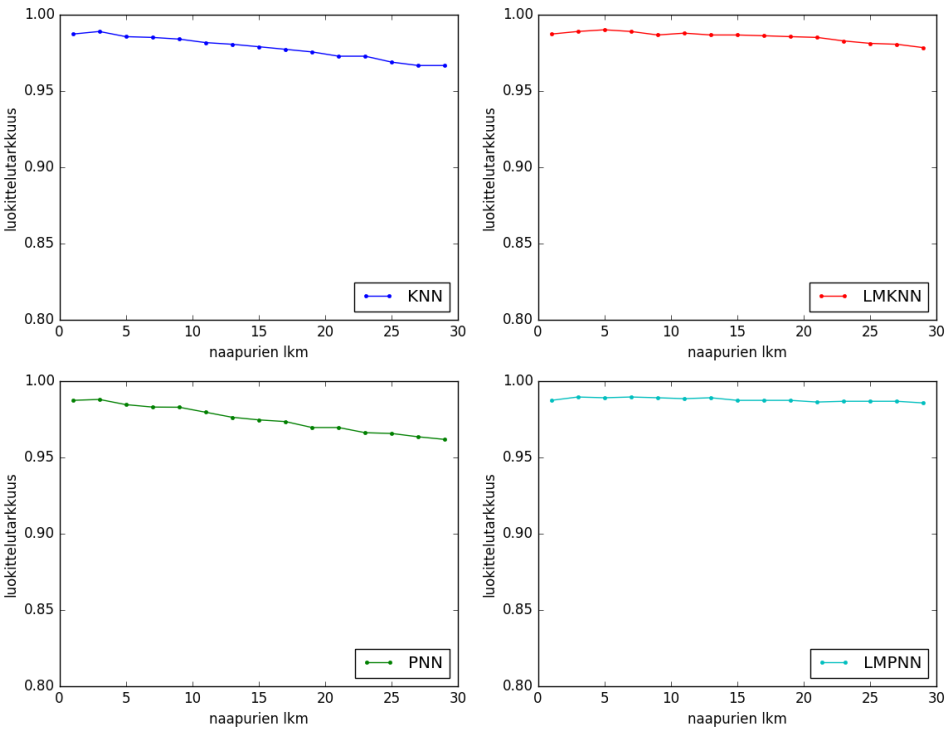
	Iris-aineisto				**Digits-aineisto**			
k	**KNN**	**LMKNN**	**PNN**	**LMPNN**	**KNN**	**LMKNN**	**PNN**	**LMPNN**
1	**0,9600**	**0,9600**	**0,9600**	**0,9600**	**0,9872**	**0,9872**	**0,9872**	**0,9872**
3	**0,9600**	**0,9600**	**0,9600**	**0,9600**	0,9889	0,9889	0,9878	**0,9894**
5	**0,9667**	0,9533	0,9600	0,9600	0,9855	**0,9900**	0,9844	0,9889
7	**0,9733**	**0,9733**	0,9667	0,9600	0,9850	0,9889	0,9828	**0,9894**
9	0,9600	**0,9800**	**0,9800**	0,9600	0,9839	0,9866	0,9827	**0,9889**
11	0,9667	**0,9800**	**0,9800**	0,9600	0,9816	0,9878	0,9794	**0,9883**
13	**0,9867**	0,9800	0,9800	0,9600	0,9805	0,9866	0,9761	**0,9889**
15	**0,9800**	**0,9800**	**0,9800**	0,9667	0,9789	0,9866	0,9744	**0,9872**
17	**0,9800**	0,9733	**0,9800**	0,9733	0,9772	0,9861	0,9733	**0,9872**
19	**0,9800**	0,9733	0,9733	0,9733	0,9755	0,9855	0,9694	**0,9872**
21	**0,9800**	0,9667	0,9667	0,9733	0,9727	0,9850	0,9694	**0,9861**
23	**0,9733**	**0,9733**	0,9667	**0,9733**	0,9727	0,9827	0,9660	**0,9866**
25	**0,9667**	**0,9667**	0,9600	**0,9667**	0,9688	0,9811	0,9655	**0,9866**
27	0,9600	0,9600	0,9533	**0,9667**	0,9666	0,9805	0,9633	**0,9866**
29	0,9533	0,9533	0,9467	**0,9667**	0,9666	0,9783	0,9616	**0,9855**
ka	**0,9698**	0,9689	0,9676	0,9653	0,9781	0,9855	0,9749	**0,9876**

Taulukko 2. Iris- ja Digits-aineistojen luokittelutarkkuudet.

Luokittelutarkkuus ei vaihdellut kovinkaan paljoa eri menetelmien tai k:n arvojen välillä. Huomattava on, että arvolla $k = 1$ kaikkien menetelmien luokittelutarkkuus on sama, koska ne toimivat tällä arvolla kuten 1-NN ja palauttavat saman naapurin. Jos tarkkuudet piirtää kuvaajaan, ovat menetelmien väliset erot hieman selkeämmät. Kuvat 6 ja 7 sisältävät kuvaajat aineistojen luokittelutarkkuuksista.



Kuva 6. Iris-aineiston luokittelutarkkuudet.



Kuva 7. Digits-aineiston luokittelutarkkuudet.

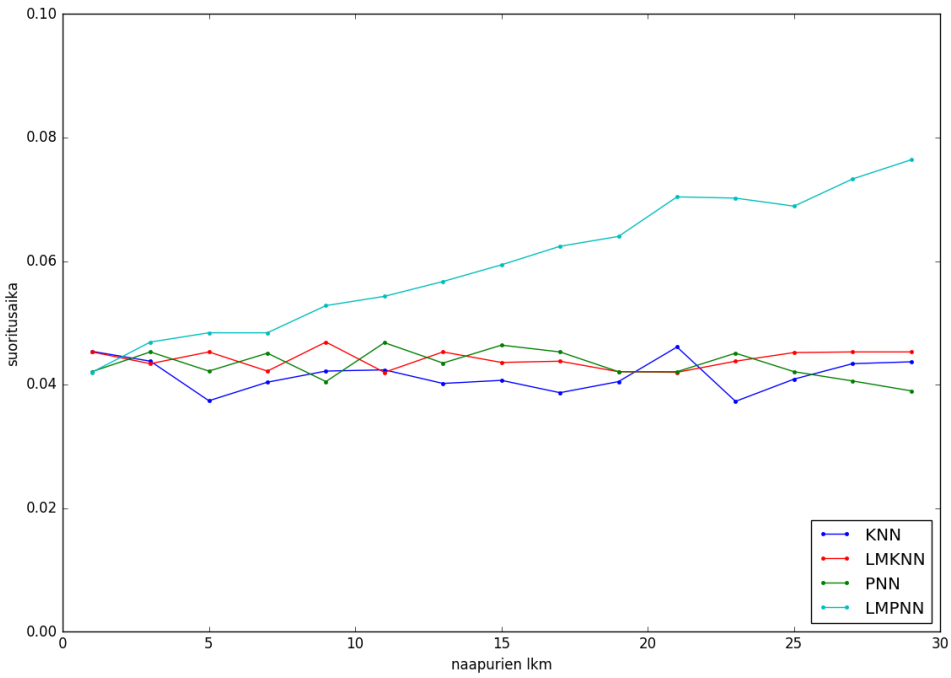
Iris-aineistossa luokittelutarkkuuksissa on huomattavissa pientä vaihtelua jokaisen menetelmän kohdalla, mutta LMPNN:n luokittelutarkkuus vaihtelee vähiten. Taulukosta 2 huomaamme, että KNN tuottaa parhaimman keskiarvon

kaikkien arvojen k suhteen, mutta erot menetelmien välillä eivät ole kovinkaan suuret. Digits-aineistossa menetelmien KNN ja PNN luokittelutarkkuudet alenevat selkeästi arvon k kasvaessa. Myös LMKNN:n luokittelutarkkuuden kanssa on huomattavissa pieni lasku, mutta LMPNN:n luokittelutarkkuus pysyy jokseenkin samana k:n arvosta riippumatta.

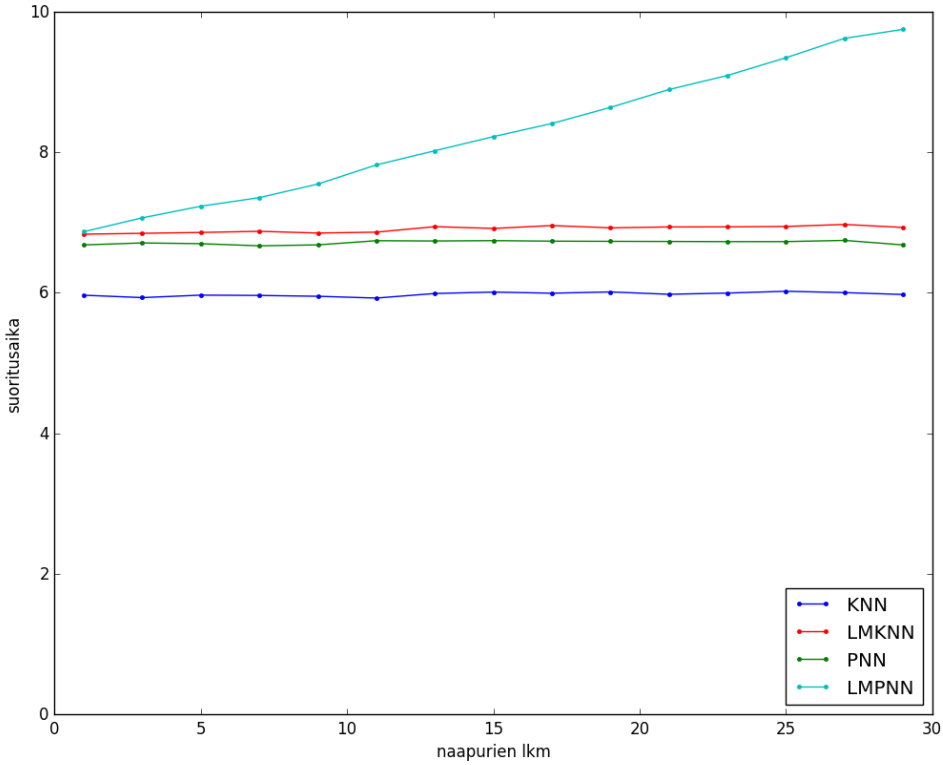
Kuvaajista käy ilmi, että LMPNN tuottaa muita menetelmiä vankemman luokittelutarkkuuden arvon k vaihteluiden suhteen. Muita johtopäätöksiä kuvaajista on vaikea päätellä, ja yleisesti ottaen kaikki menetelmät suoriutuivat molempien aineistojen luokittelusta hyvin. Muunnelmat LMKNN ja LMPNN osoittautuivat hieman paremmaksi kuin alkuperäinen k-lähimmän naapurin menetelmä Digits-aineiston kanssa, mutta Iris-aineistossa huomattavia eroja ei ollut. Tämä voi johtua osin siitä, että alkuperäinen k-lähimmän naapurin menetelmä suoriutuu hyvin kummankin aineiston luokittelusta, joten parannettavaa on hyvin vähän. Menetelmien eroja olisi hyvä testata sellaisten aineistojen kanssa, joiden luokittelusta k-lähimmän naapurin menetelmä ei suoriudu niin hyvin.

4.3. Suoritusajat

Menetelmien suoritusajat tallennettiin luokittelutarkkuuksia laskiessa. Jokainen suoritusaika on siis 10-kertaisen ristiinvalidoinnin tuottama keskiarvo. Menetelmiä ohjelmoidessa niiden optimointiin ei käytetty juurikaan aikaa, joten tulosten voidaan olettaa olevan vain suuntaa antavia. Menetelmien suoritusajat sekunteina ovat esiteltynä kuvissa 8 ja 9.



Kuva 8. Menetelmien suoritusajat Iris-aineistossa.



Kuva 9. Menetelmien suoritusajat Digits-aineistossa.

Suoritusajoissa on huomattavia eroja sekä aineistojen, että menetelmien välillä. Iris-aineistossa kaikki suoritusajat ovat alle 0,1 sekunnin, mutta Digits-aineistossa korkein suoritusaika on melkein 10 sekuntia. Tämä on selitettävissä sillä, että Digits-aineisto sisältää yli 10 kertaisen määrän instansseja ja instanssien attribuutteja on 16-kertainen määrä. Korkea attribuuttien määrä hidastaa kaikkien euklidisten etäisyyksien laskemista, joka huonontaa suoritusaikaa runsaasti.

Digits-aineistossa menetelmien erot näkyvät selkeimmin. KNN:n suoritusaika on kaikkein matalin, mutta vain alle sekunnin vähemmän kuin PNN:n ja LMKNN:n. LMPNN on ainoa menetelmä, jonka suoritusaika kasvaa naapurien lukumäärän kasvaessa. Tämä tulee ilmi molemmista aineistoista. Syy tähän löytyy menetelmien toimintaperiaatteista. KNN-, PNN- ja LMKNN-menetelmillä suurin osa laskennasta tapahtuu laskiessa instanssien välisiä etäisyyksiä ja järjestäessä nämä etäisyydet. Kun etäisyydet ovat laskettu ja järjestetty, ei ole suurta eroa sillä, kuinka montaa naapuria käytetään enemmistöluokan määrittämiseen. LMPNN sen sijaan laskee paikallisia keskiarvovektoreita naapurien lukumäärän verran, joka selittää suoritusajan kasvun arvon k kasvaessa.

5. Yhteenveto

Tässä tutkielmassa tutustuttiin k-lähimmän naapurin menetelmään sekä kolmeen siitä luotuun muunnelmaan. Menetelmien toimintaperiaatteet käytiin hyvin yksityiskohtaisesti läpi ja niiden luokittelutarkkuuksia ja suoritusaikoja vertailtiin keskenään käyttäen kahta aineistoa. Luokittelutarkkuuksissa ei ollut kovinkaan suuria eroja menetelmien välillä, mutta LMPNN osoittautui kaikkein vakaimmaksi menetelmäksi naapurien lukumäärän vaihteluiden suhteen. Suoritusajoissa LMPNN oli hitain menetelmä ja ainut menetelmä, jonka suorituskyky huononee naapurien lukumäärän kasvaessa. Muiden menetelmien suoritusajat olivat suurin piirtein samat naapurien lukumäärästä riippumatta.

Tutkielmassa käytettyjä menetelmiä olisi hyvä tutkia useampien aineistojen kanssa. Käytetyt kaksi aineistoa olivat sellaisia, joiden luokittelusta alkuperäinen k-lähimmän naapurin menetelmä suoriutuu hyvin, joten näiden aineistojen kanssa luokittelutarkkuuden parantaminen muunnelmien avulla voi olla vaikeaa. Useampien aineistojen käyttö voisi myös antaa osviittaa siitä, onko LMPNN:n suoritusajan kasvu hyväksyttävissä menetelmän vakauden nimissä.

Viiteluettelo

T. Bailey and A. K. Jain. 1978. A Note on Distance-Weighted k-Nearest Neighbor Rules. *IEEE Transactions on Systems, Man, and Cybernetics* 8, 4, 311-313.

T. Cover and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 21-27.

S. A. Dudani. 1976. The Distance-Weighted k-Nearest-Neighbor Rule. *IEEE Transactions on Systems, Man, and Cybernetics* 6, 4, 325-237.

J. Gou, Y. Zhan, Y. Rao, X. Shen, X. Wang and W. He. 2014. Improved pseudo nearest neighbor classification. *Knowledge-Based Systems* 70, 361-375.

M. Lichman. 2013. UCI Machine Learning Repository: Digits-dataset. http:// archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits. Checked 24.2.2017.

M. Lichman. 2013. UCI Machine Learning Repository: Iris-dataset. http://archive.ics.uci.edu/ml/datasets/Iris. Checked 24.2.2017.

Y. Mitani and Y. Hamamoto. 2006. A local mean-based nonparametric classifier. *Pattern Recognition Letters* 27, 1151-1159.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michael, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 2825-2830.

Wikipedia. 2017. Cross-validation. https://en.wikipedia.org/wiki/Cross-validation_(statistics). Checked24.2.2017.

X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z. Zhou, M. Steinbach, D. J. Hand and D. Steinberg. 2008. Top 10 algorithms in data mining. *Knowledge and Information Systems* 14, 22-24.

Y. Zeng, Y. Yang and L. Zhao. 2009. Pseudo nearest neighbor rule for pattern classification. *Expert Systems with Applications* 36, 3587-3595.

Katsaus mobiilisovellusten monialustakehitykseen

Ilmo Setälä

Tiivistelmä.
Laajentuneen laitekannan ja mobiililaitteiden teknisen kehityksen myötä mobiilisovellusten määrä on kasvanut nopeasti viime vuosina. Alustojen erilaisuudesta johtuen sovelluksen kehittäminen monelle alustoille aiheuttaa koodin hajautumista ja ongelmaa ratkaisemaan onkin kehitetty erilaisia menetelmiä ja työkaluja, jotka mahdollistavat koodipohjan jakamista alustojen kesken. Monialustakehityksellä ja sitä toteuttavilla menetelmillä ja työkaluilla on erilaisia etuja ja heikkouksia, joiden perusteella sovellusprojektille optimaalisen valinnan tekeminen on kriittistä projektin onnistumisen kannalta.

Avainsanat ja -sanonnat: Monialusta, mobiilisovellukset, sovelluskehitys, alustariippumattomuus.

1. Johdanto

Mobiiliteknologiat ovat kehittyneet nopeasti viime vuosien ajan. Mobiililaitteet ovat laajentuneet puhelimien ja tablettitietokoneiden lisäksi kattamaan muun muassa kelloja ja silmälaseja. Laitteiden suoritustehot ovat parantuneet, säilytystila kasvanut ja näytöt tarkentuneet. Nämä parannukset ovat mahdollistaneet kehittäjille monimutkaisten ja näyttävien sovellusten kehittämisen. Näiden kehitysten myötä sovellusten määrä onkin kasvanut huomattavasti viime vuosina [Statista 2017]. Erilaisten laitteiden lisäksi käytössä on erilaisia mobiilialustoja, suosituimpina Android, iOS ja Windows Phone [IDC 2017]. Nämä sovellusalustat eivät kuitenkaan ole yhteensopivia, ja siksi tärkeimpien alustojen tukeminen vaatiikin sovelluksen kehittämisen jokaiselle alustalla erikseen. Tämä aiheuttaa laajat osaamisvaatimukset kehittäjille ja vie runsaasti aikaa mobiilisovelluksia kehitettäessä ja myöhemmin päivitettäessä koodipohjan jakautuessa usealle alustalle [Joorabchi *et al.* 2013]. Ratkaisuksi on kehitetty erilaisia menetelmiä, jotka pyrkivät mahdollistamaan alustariippumattoman sovelluskehityksen siten yksinkertaistaen kehitysprosessia, mutta tuoden kuitenkin saataville natiivialustojen ominaisuuksia projektin vaatimusten toteuttamiseen tarvittavalla laajuudella.

Nämä menetelmät voidaan jakaa pääpiirteittensä mukaan kahteen erilliseen kategoriaan: web-tekniikoihin pohjautuviin (web-based) ja natiiveja rajapintoja (native API) hyödyntäviin sovelluskehyksiin [Boushehrinejadmoradi *et al.* 2015].

Web-tekniikoihin pohjautuvat sovellukset voidaan jakaa kahteen aliryhmään: web-sovelluksiin (web app) ja hybridisovelluksiin (hybrid), joista ensimmäinen jaetaan pääasiassa verkkosivustojen tavoin verkkoselaimella käsiteltäväksi ja jälkimmäinen asentamalla kohdelaitteeseen, jälkimmäisen mahdollistaessa laajemmin laitteen natiiviominaisuuksien käyttämisen [Angulo and Ferre 2014]. Eri kategorioilla on omat käyttökohteensa; yksinkertaistettuna voitaisiin määritellä web-tekniikoihin perustuvien sovellusten sopivan yksinkertaisempiin ja laitteelta vähemmän tehoja ja natiiviominaisuuksia vaativiin sovelluksiin. Natiiveja rajapintoja hyödyntävien menetelmien avulla voidaan luoda nopeampia ja lähes kaikkia laitteen ominaisuuksia ja ulkoasuelementtejä hyödyntäviä sovelluksia [Angulo and Ferre 2014]. Monialustakehityksen erilaisia menetelmiä ja projektiin sopivan menetelmän valintaa tarkastellaan tarkemmin luvussa 2.

Monialustakehityksen eri menetelmiä toteuttavia sovelluskehyksiä on lukuisia. Useat sovelluskehykset nousevat positiivisesti esille tietyillä ominaisuuksillaan, kuten yksinkertaisuudellaan, hyvällä suorituskyvyllään, monikäyttöisyydellään tai laajalla tuellaan [Dhillon and Mahmoud 2014]. Sovelluskehystä valittaessa projektin kannalta tarpeellisimpiin ominaisuuksiin tuleekin kiinnittää huomiota. Uusia sovelluskehyksiä ja vanhojen laajennuksia kehitetään nopeaan tahtiin ja uuden käyttäjän voikin olla hankalaa löytää sopivaa sovelluskehystä lukuisten eri vaihtoehtojen joukosta [Smeets and Aerts 2016]. Luvussa 3 tarkastellaan muutamia erilaisia sovelluskehyksiä (Sencha Touch, Apache Cordova, NativeScript ja Xamarin) ja käsitellään projektin kannalta sopivan sovelluskehyksen valintaa.

Monialustakehityksen valitseminen sovellusprojektiin voi tuottaa monia etuja: resurssien säästöä, nopeamman projektin käynnistymisen vanhoja taitoja hyödyntäessä ja sovelluksen koodin yhteiskäyttöä eri alustoilla. Etujen saavuttaminen vaatii kuitenkin tarkkaa sopivan menetelmän ja sovelluskehyksen valintaa, jotta saadaan vähennettyä tai vältytään monialustakehityksen haasteilta ja ongelmilta [Dhillon and Mahmoud 2014]. Ongelmiksi voivat nousta suorituskykyongelmat, UI-elementtien eroaminen natiivielementeistä ja siten syntyvät käytettävyyshaitat, sovelluskehysten päivityssykli ja uusi välitaso ohjelmakoodin ja suoritettavan sovelluksen välillä, jonka sovelluskehys tuo mukanaan, ja mahdollisesti myös virheitä tai yhteensopivuusongelmia. Monialustakehityksen mahdollisuuksia ja ongelmia tarkastellaan laajemmin luvussa 4.

2. Erilaiset toteutusmenetelmät

Monialustakehityksen mahdollistavat menetelmien jaotteluun ei ole vakiintunutta tapaa. Tässä tutkielmassa menetelmät on jaoteltu kahteen pääkategoriaan: web-tekniikoihin pohjautuviin ja natiiveja rajapintoja hyödyntäviin menetelmiin

[Boushehrinejadmoradi *et al.* 2015]. Tämän lisäksi web-tekniikoihin pohjautuvat menetelmät on jaettu kahteen alempaan kategoriaan mukaillen Anguloa ja Ferrea [2014]: web- ja hybridisovelluksiin.

Nämä erilaiset lähestymistavat monialustakehitykseen soveltuvat erilaisiin sovellusprojekteihin ja sopivan menetelmän valintaa tuleekin harkita tarkasti projektin vaatimuksien ja kehittäjien taustojen pohjalta, sillä jokaisella menetelmällä on omat mahdollisuutensa ja heikkoutensa. Onnistuneella menetelmän valinnalla on vahva vaikutus myös itse sovellusprojektin onnistumiseen [Mercado *et al.* 2016].

2.1. Web-tekniikoihin pohjautuvat menetelmät

Web-tekniikoihin pohjautuvat menetelmät hyödyntävät web-kehityksessä käytössä olevia tekniikoita, kuten HTML5, CSS ja JavaScript. Näissä menetelmissä etuna ovat useille kehittäjille tutut ja helposti lähestyttävät tekniikat, mahdollisesti sovellukseen liittyvällä verkkosivustolla käytetyn koodin hyödyntäminen ja nopea käyttöönotto. Web-tekniikoihin pohjautuvat menetelmät eivät kuitenkaan mahdollista täyttä laitteen suorituskyvyn ja ominaisuuksien hyödyntämistä eivätkä saavuta käytettävyydessä natiivin sovelluksen sulavuutta eivätkä siten ole yhtä mieluisia käyttäjille [Angulo and Ferre 2014].

Web-tekniikoihin pohjautuvat menetelmät eivät vaadi kehittäjältä tuntemusta erillisistä alustoista, vaan mahdollistavat kokonaisten sovellusten toteuttamisen kohdealustoja tuntematta [Xanthopoulos and Xinogalos 2013]. Tämä nousee esille etuna varsinkin mobiilisovelluksen ollessa yksinkertainen ja ilman alustojen natiiviominaisuuksia toteutettavissa. Tässä tutkielmassa web-tekniikoihin pohjautuvat menetelmät jaetaan kahteen alakategoriaansa niiden tarjonnan ja natiiviominaisuuksien käytön perusteella: web-sovelluksiin ja hybridisovelluksiin, ensin mainittujen mahdollistaessa ainoastaan web-standardien mukaisten natiiviominaisuuksien käytön ja jälkimmäisten myös laajemmin riippuen menetelmän toteuttavasta sovelluskehyksestä [Angulo and Ferre 2014].

2.1.1. Web-sovellukset

Web-sovellukset tarjoillaan mobiililaitteille useimmiten verkkosivujen tavoin, eikä niitä asenneta itse laitteeseen, vaikka sekin on mahdollista hyödyntäen alustojen web-sivustoupotteita (WebView). Web-sovelluksilla on siten mainituista menetelmistä rajatuimmat mahdollisuudet käyttää laitteen ominaisuuksia, vaikkakin uusien HTML-standardien julkaisun myötä mahdollisuuksia on pyritty parantamaan [Dhillon and Mahmoud 2014]. HTML5 APIn mukaisesti saataville on tullut ominaisuuksia laajemminkin laite- ja selainkohtaisesta tuesta riippuen, kuten valo- ja videokuvaus, ilmoitukset, Bluetooth ja kosketuseleet [What Web

Can Do Today 2017]. Kehitystyön ohessa huomiota tulee kuitenkin kiinnittää näiden HTML5-ominaisuuksien tukeen eri alustoilla ja alustojen tukemissa selaimissa. Tuki uusille ominaisuuksille kuitenkin tulee todennäköisesti tulevaisuudessa laajenemaan ja natiiviominaisuuksien hyödyntäminen helpottumaan [Xanthopoulos and Xinogalos 2013].

Web-sovelluksien etuna ovat web-kehityksestä hyödynnettävien työkalujen suora käytettävyys ja siten saavutettava hyöty kehitystyön nopeudessa. Web-sovelluksen kehitys mobiililaitteelle palvelun pääverkkosivuston ohessa on myös tärkeä etu mahdollistaessaan päivityssyklin yhtenäisyyden ja säästäessään runsaasti kehitysaikaa. Web-sovellukset eivät kuitenkaan mahdollista natiivien ulkoasuelementtien käyttöä, mikä vaikuttaakin heikentävästi sovelluksen käyttökokemukseen [Angulo and Ferre 2014] ja suorituskykyyn. Web-sovellukset tarvitsevat lisäksi toimiakseen verkkoyhteyden, niiden käyttö selaimen kautta vähentää toistuvaa käyttäjäkuntaa, eivätkä ne mahdollista notifikaatioiden käyttöä taustalta [Litayem *et al.* 2015], joten menetelmänä niiden sopivuus rajoittuu pääasiassa harvoin käytettyihin palveluihin.

2.1.2. Hybridisovellukset

Hybridisovellukset pyrkivät korjaamaan web-sovelluksien puutteita, kuten mobiililaitteiden natiiviominaisuuksien hyödynnettävyyttä ja alustojen natiivien ulkoasuelementtien mukailtavuutta. Hybridisovellusten kehittäminen tapahtuu web-tekniikoita käyttäen, mutta niitä ei tarjoilla verkkosivuina, vaan ne asennetaan mobiililaitteisiin sovelluksina. Tämä mahdollistaa natiiviominaisuuksien käyttämisen erinäisten rajapintojen kautta riippuen sovelluskehyksen tuen laajuudesta. [Smeets and Aerts 2016]

Hybridisovellukset säilyttävät etunaan web-kehityksen työkalujen saatavuuden tuoden kuitenkin lisänä myös alustakohtaisia ominaisuuksia ja parantaen käyttökokemuksen sulavuutta ja suorituskykyä. Tämä mahdollistaa perinteisiä web-sovelluksia monimutkaisempien ja, riippuen käyttötapauksesta, myös nopeampien mobiilisovelluksien kehittämisen. Hybridisovelluksia kehitettäessä ollaan kuitenkin riippuvaisia sovelluskehyksen tarjoamista natiiviominaisuuksista, mikä nousee esille mahdollisena ongelmana varsinkin uusien natiiviominaisuuksien ollessa kyseessä [Xanthopoulos and Xinogalos 2013]. Hybridisovelluksissa on myös web-sovelluksiin verrattuna suurempi oppimiskynnys sovelluskehysvaatimuksensa vuoksi.

Verrattuna natiivisovelluksiin hybridisovellukset tuovat uuden tason käyttäjän ja sovelluksen väliin, sillä hybridisovelluksia ei suoriteta natiivisti vaan oh-

jelmakoodi suoritetaan alustan web-toteutuksen kautta, jolloin ohjelman toiminta on hitaampaa. Tämä voikin aiheuttaa varsinkin runsaasti grafiikkaa tarvitsevissa sovelluksissa suorituskykyongelmia. [Litayem *et al.* 2015]

2.2. Natiivien alustojen rajapintoja hyödyntävät menetelmät

Toisena mobiilin monialustakehityksen pääkategoriana ovat natiivien alustojen rajapintoja hyödyntävät menetelmät. Nämä menetelmät tuovat saataville lähes kaikki natiivien alustojen ominaisuudet ja hyödyntävät sovelluksen kohdealustan rajapintoja mahdollistaen lähes natiivia vastaavan nopeuden ja natiivien ulkoasuelementtien käyttämisen. Osa näistä menetelmistä kääntää koko sovelluksen alustalle natiivisti suoritettaviksi (generated) [Xamarin 2017] ja osa kutsuu sovelluksen suorituksen aikana natiiviominaisuuksia rajapintojen kautta luoden kuitenkin sovelluksen käyttöliittymän natiivisti (interpreted) [Mercado *et al.* 2016]. Yleinen lähestymistapa sovelluskehyksille tämän menetelmän toteuttamiseen on natiivialustojen toteutusten jako komponentteihin, jolloin koodia käännettäessä käytetään sovelluskehyksen määrittelemää kohdealustan komponenttitoteutusta, kuten natiivia listanäkymää. Komponentit tuovat näkyville rajapinnan, jolloin taustalla olevalla toteutuksella ei ole merkitystä ja komponentteja voidaan käyttää kohdealustaa tietämättä [El-Kassas *et al.* 2015]. Jaetun toteutuksen lisäksi natiiveja rajapintoja hyödyntävät menetelmät tarjoavat usein mahdollisuuden kirjoittaa erillisiä sovelluksen osia halutulle laitealustalle jaettujen osien lisäksi, jolloin kehittäjä voi itse määritellä haluamansa natiivikomponentin käytön olematta riippuvainen sovelluskehyksen valitsemasta toteutuksesta.

Jotkin menetelmään perustuvat sovelluskehykset mahdollistavat jopa natiivialustoille luotujen kirjastojen tuomisen käyttöön. Näille natiivikirjastoille luodaan omat sovelluskehyksen kanssa yhteensopivat moduulinsa, jonka jälkeen ne on mahdollista liittää sovellukseen ja käyttää rajapinnan avulla. [Xamarin 2017]. Natiivikirjastojen hyödyntäminen on tärkeä ominaisuus, sillä natiivikirjastot ovat suorituskykyisiä, helpottavat ja nopeuttavat ohjelmistokehitystä ja natiivikirjastoja on laajasti saatavilla eri tarkoituksiin. Natiivikirjastojen siirtäminen sovelluskehyksen moduuliksi ei kuitenkaan välttämättä suju ongelmitta ja mahdollisiin ongelmiin tuleekin varautua natiivikirjastoihin tukeuduttaessa, minkä lisäksi natiivikirjastojen hyödyntäminen joudutaan tekemään alustakohtaisesti.

Natiiveja rajapintoja hyödyntävät menetelmät vaativat uuden ohjelmistokehyksen opiskelemisen, taustalla olevien alustojen vähintään jonkinasteisen tuntemisen ja siten web-tekniikoihin perustuviin menetelmiin verrattuna suuremman panostuksen mobiilisovelluksen kehittämiseen lähtiessä. Pyrittäessä kuitenkin mahdollisimman lähelle natiivia suorituskykyä ja ulkoasua, ovat nämä

menetelmät tärkeässä osassa mahdollistaessaan natiivien ominaisuuksien käytön. Riippuen sovelluskehyksestä, voi sovellus suorituskyvyssään saavuttaa jopa erillisesti natiivialustoille toteutettujen sovellusten suorituskyvyn. Angulo ja Ferre [2014] havaitsivat monialustakehitystä hyödyntävän sovelluksen ulkoasun käyttäjätesteissä saavan vain hieman heikommat tulokset natiiviin verrattuna, eron ollessa kuitenkin suurempi iOS- kuin Android-alustalla. Natiivin ulkoasun tason saavuttaminen riippuu vahvasti myös käytettävän sovelluskehyksen ominaisuuksista ja toteutuksesta.

2.3. Oikean menetelmän valinta

Aiemmin esitellyt menetelmät ovat hyvin erilaisia lähtövaatimuksiensa ja mahdollistamiensa sovellustoteutusten osalta. Tämän vuoksi sopivan menetelmän valinta onkin hyvin tärkeää mobiilisovellusprojektin suunnitteluvaiheessa. Sopivimman menetelmän onnistunut valinta voi säästää hyvinkin paljon aikaa kehitysvaiheessa, kuten myös sovelluksen jatkokehityksessä ja tukemisessa.

Sopivan menetelmän valinnassa tulee ottaa huomioon sovelluksen vaatimat laite- ja järjestelmäominaisuudet, kuten sensorien käytön mahdollisuudet ja tallennusmahdollisuudet, suorituskykyvaatimukset ja ulkoasulta vaadittavat ominaisuudet [Le Goaer and Waltham 2013]. Laite- ja järjestelmäominaisuuksien osalta huomioon tulee ottaa esimerkiksi HTML5-standardin mahdollistamat mobiililaitteen ominaisuudet harkittaessa web-sovelluksen tekoa [Humayoun *et al.* 2013] ja tarkistaa myös menetelmän toteuttavien sovelluskehysten tuet tarvittaville natiiviominaisuuksille. Suorituskyvyn tarkastelu mobiilialustoilla on tärkeää käyttäjien ollessa kärsimättömiä ja suorituskyky tuleekin esille varsinkin graafisesti monimutkaisemmissa sovelluksissa [Le Goaer and Waltham 2013]. Yksinkertaisemmissa sovelluksissa suorituskyky ei monissa tapauksissa nouse kovinkaan suureen rooliin.

Sovellukselle suunniteltu ulkoasu on myös tärkeää menetelmää valittaessa. Natiivisti toteutetut sovellukset ovat käyttäjille sulavampia ja siten myös miellyttävämpiä käyttää kuin monialustatyökaluja käyttäen luodut [Angulo and Ferre 2014]. Natiiveja rajapintoja hyödyntävät menetelmät tuovat kuitenkin natiivit ulkoasuelementit saataville ja voivat siten saavuttaa lähes natiivin käytettävyyden. Sovelluksen ulkoasu voi olla suunniteltu myös alustojen ulkoasuohjeistuksia noudattamatta tai sisältää natiivista poikkeavia ulkoasuelementtejä, jolloin ulkoasun painoarvo menetelmän valinnassa on vähäisempi kuin ohjeistuksia noudatettaessa.

Valinnan eri osa-alueet tasapainoisesti huomioon otettaessa voidaan tehdä onnistunut valinta sovellusprojektissa käyttöön otettavasta teknologiasta. Epä-

onnistunut menetelmän valinta tai heikko toteutus voivat johtaa huonoon sovellustoteutukseen ja käyttäjätyytymättömyyteen. Esimerkiksi Facebookin vaihtaessa hybridimenetelmien käytöstä alustakohtaisiin natiiveihin toteutuksiin negatiivisten käyttäjäarvostelujen määrä putosi merkittävästi [Mercado *et al.* 2016].

3. Monialustakehityksen mahdollistavia sovelluskehyksiä

Monialustakehityksen mahdollistavia sovelluskehyksiä on lukuisia, jotka kaikki pyrkivät ratkaisemaan ongelman omalla lähestymistavallaan. Nämä sovelluskehykset käyttävät eri ohjelmointikieliä, hyödyntävät eri kehitysympäristöjä ja tukevat alustoja vaihtelevalla laajuudella [Dhillon and Mahmoud 2014]. Osa soveluskehyksistä on kokonaan ilmaisia, osa maksullisia ja jotkut tarjoavat useita erilaajuisia lisenssivaihtoehtoja. Sopivaa sovelluskehystä valittaessa huomiota kannattaa kiinnittää sen tarjoamiin mahdollisuuksiin ja kehitystiimin osaamiseen ohjelmointikielten ja kehitysympäristöjen osalta. Tässä luvussa perehdytään tarkemmin seuraaviin sovelluskehyksiin: Sencha Touch, Apache Cordova, Xamarin ja NativeScript.

3.1. Web-sovelluksien kehittämisen sovelluskehykset – Sencha Touch

Sencha Touch on Senchan kehittämä JavaScript-käyttöliittymäkirjasto kohteenaan mobiilit web-sovellukset. Sencha Touch tarjoaa työkaluja, joiden avulla kehittäjät voivat luoda alustojen natiivia ulkoasuelementtejä mukailevia näkymiä. Sencha Touchin HTML5 UI-komponentit ovat tarjolla seuraaville tuetuille alustoille: iOS, Android, BlackBerry ja Windows Phone. UI-komponentit pyrkivät mukailemaan alustakohtaisia teemoja mahdollisimman tarkasti. [Sencha 2017]

Sencha Touch tukee myös mobiililaitteiden näytön eri tiloja, tilojen muutoksia ja näyttöjen vaihtelevia kokoja. Muita tarjolla olevia ominaisuuksia ovat HTML5-pohjainen suorituskykyinen graafikirjasto ja datan käsittelyyn tarjottava paketti, joka mahdollistaa datan haltuunoton useista eri backend-lähteistä. Sencha Touch -projektien perusmalli noudattaa MVC-mallia (Model View Controller pattern), jossa mallit (models) esittävät erilaisia sovelluksen dataobjekteja, näkymät (views) huolehtivat datan esittämisestä käyttäjälle hyödyntäen UI-komponentteja ja kontrollerit (controller) hoitavat vuorovaikutuksen käyttäjän ja applikaation välillä. Näiden osien lisäksi käytössä ovat myös varastot (stores), jotka lataavat datan sovellukseen yleensä mallien perusteella, ja profiilit (profiles), jotka mahdollistavat muutoksia alustakohtaisesti kohdistuen käyttöliittymiin. [Sencha 2017]

Talvella 2017 Sencha Touch oli vapaasti ilmaiseksi käytettävissä tietyin rajoituksin; laajennettu lisenssi vaaditaan, jos sovellus halutaan lisätä Applen App Storeen tai jos sovellus toimii kehittäjien työkaluna. Laajempi Sencha Ext JS -

paketti sisältää kehittäjäkohtaiset lisenssimaksut, mutta tuo samalla laajempia datan analysointi ja visualisointi mahdollisuuksia, teemoja ja lisätyökaluja kehitystyön nopeuttamiseen. [Sencha 2017]

3.2. Hybridisovelluksien kehittämisen sovelluskehykset – Apache Cordova

Ilmainen avoimen lähdekoodin Apache Cordova tarjoaa työkalut mobiilisovellusten luomiseen käyttäen HTML5, CSS3 ja JavaScriptiä. Cordovalla luodut mobiilisovellukset ovat hybridejä eli ne asennetaan kohdelaitteeseen, ne käyttävät alustojen web-näkymiä (WebView) sisältönsä esittämiseen ja natiivien APIen käyttöä on mahdollistettu. Cordova tukee yleisimpien Androidin, iOSin ja Windows Phonen lisäksi myös harvinaisempia alustoja kuten Bada, Firefox OS, LG webOS, Nokia Symbian OS, Tizen ja Ubuntu Touch. Kaikille alustoille ei kuitenkaan tarjota samoja natiiviominaisuuksia APIen kautta käytettäväksi. [Apache 2017]

Avoimen lähdekoodin sovelluskehyksenä Apache Cordovaan on tehty runsaasti liitännäisiä, työkaluja ja palveluja. Jotkin sovelluskehykset myös käyttävät Apache Cordovaa pohjanaan, jonka päälle luodaan sovelluskehyksen omia uusia ominaisuuksia tai tarjoavat lisätyökaluja Cordova-kehitykseen. Yksiä tunnetuimmista laajennuksista ovat Adobe Phonegap, joka lisää Cordovan päälle helppokäyttöiset työpöytäsovellukset mobiilisovellusten nopeaan kehittämiseen ja käyttöönottoon, ja Ionic Framework, joka pyrkii laajentamaan kehitysmahdollisuuksia tuomalla saataville uusia komponentteja, toiminnallisuuksia ja teemoja. Phonegap tukee pääosin samoja alustoja kuin Cordova ja Ionic tukee iOS ja Android alustoja. [Apache 2017, Adobe 2017]

3.3. Natiivien alustojen rajapintoja hyödyntävät sovelluskehykset

3.3.1. Xamarin

Xamarin on samannimisen ohjelmistoyhtiön, joka siirtyi helmikuussa 2016 Microsoftin omistukseen, vuodesta 2011 kehittämä sovelluskehys. Xamarin sisältää C#-pohjaisia työkaluja eri alustoille, kuten Android, iOS ja Mac. Xamarinin kehityksessä eri alustat voivat käyttää yhteistä C#-koodipohjaa ja luoda ulkoasuelementit alustakohtaisesti tai vaihtoehtoisesti hyödyntää Xamarin.Forms-työkalun tarjoamia valmiita kontrolleja, jotka kohdistetaan natiivielementteihin sovelluksen kääntämisen yhteydessä. C#:n sijaan Xamarinin yhteydessä voidaan käyttää myös funktionaalista F#-ohjelmointikieltä, mutta dokumentaatiot kohdistuvat pääosin C#-esimerkkeihin. [Xamarin 2017]

Xamarin tarjoaa täyden pääsyn iOS ja Android API:hin, mikä mahdollistaakin käytännössä kaiken natiivisti toteutettavissa olevan toteutuksen myös Xamarinia käyttäen. Alustariippumattoman koodin jakamiseen Xamarin käyttää

.NET Frameworkin PCL-kirjastoja (Portable Class Library), jolloin kaikki varsinainen taustalogiikka voidaan siirtää alustariippumattomaan kirjastoon ja pitää alustariippuvaiset osiot omissa projekteissaan. Xamarin on yksi harvoista sovelluskehyksistä, joka kääntää ohjelmakoodin suoraan natiivialustalle eikä ainoastaan kutsu suorituksen aikana natiiveja rajapintoja. [Xamarin 2017]

Xamarin suosittelee MVC-mallia kehitykseen, mutta saatavilla on myös muita, esimerkiksi MVVM-mallia tukevia lisäpaketteja. Muita Xamarinin suosittelemia kehitysmalleja mobiilikehitykseen ovat Singleton, jossa tietyistä objekteista voidaan luoda vain yksi ainoa instanssi, Provider, jossa alustakohtaiset muutokset kirjoitetaan alustoille yhteistä rajapintaa vasten ja Async, jossa pidempään suoritettavat tehtävät siirretään erilliseen säikeeseen ja toiminta voi jatkua pidempiaikaisen tehtävän ollessa käynnissä taustalla. [Xamarin 2017]

Xamarinin lisensointi muuttui Microsoftin omistajuuden yhteydessä liitetyksi Visual Studio -lisenssiin. Organisaatiotason kaupalliseen käyttöön tarvitaan voimassa oleva Visual Studio Professionalin tai Enterprisen tilaus, muuhun käyttöön riittää ilmainen Visual Studio Community Edition. [Xamarin 2017]

3.3.2. NativeScript

Telerikin vuodesta 2014 kehittämä NativeScript mahdollistaa natiiveja ulkoasukomponentteja hyödyntävien sovellusten luomisen käyttämällä web-teknologioita CSS, JavaScript, TypeScript ja Angular. NativeScript eroaa kuitenkin Cordova-pohjaisista sovelluskehyksistä, sillä se ei käytä alustakohtaisia WebView-näkymiä vaan sovellus kutsuu ajon aikana natiivialustojen rajapintoja mahdollistaen myös nopeamman uusien alustaominaisuuksien käyttöönoton [Smeets and Aerts 2016]. NativeScript ei siten mahdollista saman sovelluksen suoraa käyttöä sekä verkkosivustona että asennettavana sovelluksena, kuten monet Cordova-pohjaiset ratkaisut. Hyötynä kuitenkin on sovellusten parempi suorituskyky, joka nousee esille varsinkin graafisesti vaativissa sovelluksissa, ja suurempi natiiviominaisuuksien hyödyntämismahdollisuus. [NativeScript 2017]

NativeScriptiin on rakennettu vahva tuki Angular-ohjelmistokehykselle ja mahdollista onkin rinnakkainen kehitys Angular-pohjaista web-sivustoa ja mobiilisovellusta tehdessä. NativeScript-kehitys eroaa web-kehityksestä omilla XML-pohjaisilla näkymäpohjillaan, joihin määritellään käytettävät natiivielementit. Rinnakkaisesti kehitettäessä voidaan web-sivuston sivupohjat tehdä normaalisti HTML-tiedostoihin ja NativeScriptin XML-tiedostoihin. NativeScriptin avulla voidaan päästä lähelle natiivien sovellusten suorituskykyä säilyttämällä kuitenkin yhä web-tekniikoiden totutut ominaisuudet. [NativeScript 2017]

NativeScript on avoimen lähdekoodin sovelluskehys ja siten ilmainen käyttää myös kaupallisissa sovelluksissa. Telerik tarjoaa myös maksullisia Telerik Platform -lisätyökaluja NativeScript-kehitykseen, kuten uusia UI-komponentteja, testaus- ja analysointityökaluja ja työkaluja datan hallintaan [Telerik 2017]. Uutuudestaan huolimatta NativeScript on herättänyt runsaasti kiinnostusta ja saavuttanut suosiota lähestymistavallaan [Smeets and Aerts 2016].

3.4. Sopivan sovelluskehyksen valinta

Projektin tarpeille sopivan monialustakehityksen menetelmän valinnan jälkeen tulee vielä valita projektille sopiva sovelluskehys. Sovelluskehyksen onnistunut valinta on tärkeää, sillä valinnalla voi olla suuri positiivinen tai negatiivinen vaikutus projektin onnistumiseen [Dhillon and Mahmoud 2014]. Sovelluskehyksen valinnassa huomiota tulee kiinnittää joihinkin samoihin kriteereihin kuin menetelmää valittaessa, kuten tarpeellisten alustakohtaisten ominaisuuksien saatavuuteen. Projektin vaatimusten mukaan tulee tarkastella sovelluskehyksen mahdollistavan kaiken tarpeellisen, kuten tuetut alustat ja ominaisuudet, sovelluskehyksen käyttämät ohjelmointikielet ja mahdollinen koodin ristiinkäytettävyys verkkosovelluksen kanssa.

4. Monialustakehityksen mahdollisuuksia ja ongelmia

Monialustakehitys pyrkii ratkaisemaan eri alustojen vaatimien erillisten sovellusprojektien aiheuttamia ongelmakohtia tarjoamalla jaettavaa koodipohjaa. Hyötyinä monialustakehitys tuo jaetun koodipohjan myötä kehitystyön nopeutumisen, joka säästää kehitystiimin aikaa ja resursseja [El-Kassas *et al.* 2015]. Monialustakehitys myös mahdollistaa alustojen ulkopuolisten taitojen hyödyntämisen riippuen lähestymistavan ja sovelluskehyksen valinnasta ja jaettu koodipohja yhtenäistää sovellusversioiden päivityssykliä. Hyötyjen ohella monialustakehitys ei kuitenkaan tule aivan ongelmitta, vaan sen käyttöönotossa ja hyödyntämisessä on myös suorituskykyongelmia, riippuvuutta sovelluskehityksen päivityksistä ja sen tarjoamista mahdollisuuksista ja natiiviominaisuuksien hyödyntämisen osittain rajalliset mahdollisuudet.

4.1. Mahdollisuuksia ja etuja

Mobiilisovellusten kehityksen perinteinen ongelmakohta on samantyyppisen koodin jakautuminen ja toisto eri alustoilla. Suuri osa sovelluskoodista olisi kuitenkin alustariippumatonta, mutta sen jakaminen ei suoraan onnistu alustojen käyttämistä eri ohjelmointikielistä johtuen. Monialustakehityksen myötä koodia voidaan kuitenkin jakaa eri alustoille. Tämä säästää kehittäjien aikaa sekä itse sovellusprojektia ennen uusiin teknologioihin perehdyttäessä, sovellusprojektin

aikana ja jatkokehityksessä ja vähentää myöhemmin tarvittavien korjausten määrää [El-Kassas *et al.* 2015]. Joissain tapauksissa sovellus voidaan myös tuoda alustoille, joille se ei olisi resurssien puolesta ollut mahdollista ilman monialustakehitystä. Eri mobiilialustoille jakautuvan koodin lisäksi samaa koodipohjaa voi esiintyä esimerkiksi web-sovelluksessa, joten myös tämän koodipohjan rinnakkainen hyödyntäminen mahdollistuu [Smeets and Aerts 2016].

Alustakohtaisen osaamisen puuttuessa kehittäjät voivat valita oman osaamisensa pohjalta sopivan sovelluskehyksen ja hyödyntää tuntemiaan ohjelmointikieliä. Kynnys mobiilisovelluksen tekemiseen pienenee, kun tekniikat ovat tuttuja ja tuottava työ pääsee nopeammin käyntiin kuin täysin uusia asioita opetellessa. Työn laatu on todennäköisesti myös parempaa laajemmasta kokemuksesta johtuen.

4.2. Haasteita ja ongelmakohtia

Monialustakehityksessä on kuitenkin myös huomioonotettavia ongelmakohtia. Suorituskykyongelmat ovat mahdollisia varsinkin suuria datamääriä käsittelevissä tai graafisesti vaativissa sovelluksissa, kuten peleissä. Mahdollisten suorituskykyongelmien esiintulo riippuu kuitenkin valitusta toteutusmenetelmästä ja sovelluskehyksestä ja sopivilla valinnoilla voidaankin saavuttaa lähes natiivia sovellusta vastaava suorituskyky [Mercado *et al.* 2016].

Mobiilialustojen kehitys on nopeaa ja päivityssykli tiuha samalla kehitysalustojen laajentuessa ja monimutkaistuessa. Tästä johtuen kehittäjät joutuvat päivittämään sovelluksiaan tukemaan uusia käyttöjärjestelmäversioita säännöllisesti. Monialustakehitystä hyödynnettäessä päivityksiin tulee kuitenkin uusi riippuvuustaso, sovelluskehyksen päivitystahti. Suosituimmat sovelluskehykset ovat hyvin tuettuja ja päivittyvät nopeaan tahtiin, mutta kuitenkin alustojen natiivia päivitystahtia jäljessä. Sovelluskehyksen lisätessä uuden tason ohjelmistokehitykseen tulevat mukana myös sovelluskehyksen omat bugit, joiden aiheuttamat ongelmat voivat viedä runsaastikin kehitysaikaa. Nopea päivitystahti aiheuttaa ongelmia myös puutteellisen versiotuen, vanhentuneiden API:en ja tarvittavien uusien laitteistoversioiden takia [Majchrzak and Grønli 2016]. Suosituimmilla ja tuetuimmilla sovelluskehyksillä päivitystahti on kuitenkin usein nopea ja varsinkin kriittisten virheiden korjaus nopeaa aktiivisen kehittäjäyhteisön ansiosta.

Hybrideille sovelluksille haasteen saattaa luoda web-näkymien hajautuminen alustaversioiden runsaan määrän takia [Joorabchi *et al.* 2013]. Vanhempien sovellusversioiden web-näkymät ovat suorituskyvyltään heikompia ja niissä on myös ollut kriittisiä virheitä, jotka saattavat estää sovellusta käyttämästä jotain

tarjolla olevaa ominaisuutta tietyllä alustaversiolla. Ratkaisuksi on kuitenkin esitetty monille alustoille tarjolla olevaa Crosswalk WebView -näkymää, joka korjaa yhteensopivuusongelmia, tuo saataville laajasti HTML5-ominaisuuksia ja säännöllisen päivityssyklin [Smeets and Aerts 2016].

Projektin suunnitteluvaiheessa lisäongelmia tuo sopivan monialustakehityksen menetelmän ja sovelluskehyksen valinta [Smeets and Aerts 2016]. Riippuen vaatimuksista ja kehittäjien kokemuksesta valinta voi olla yksinkertainenkin. Valinnan tärkeys tulee ottaa huomioon, sillä huonon valinnan tapahtuessa monialustakehityksessä saavutettua ajallista hyötyä voidaan menettää uusien teknologioiden opiskeluun tai sovelluksen kannalta tarpeellisten natiiviominaisuuksien puutteelliseen tukeen. Natiivikirjastojen tuki voikin osoittautua ongelmalliseksi joillakin sovelluskehyksillä. Kehitystyö voi viivästyä tai jumittua tarpeellisen ominaisuuden puutteellisen, virheellisen tai kokonaan puuttuvan tuen johdosta. Ohjeiden ja dokumentaation laajuus todennäköisimmin on myös heikompaa kuin natiivialustoilla. Natiivin kehityksen mahdollistavilla sovelluskehyksillä natiivialustoille suunnatut ohjeet voidaan kuitenkin usein helpostikin mukauttaa käytettävään sovelluskehykseen.

5. Yhteenveto

Mobiililaitekannan ja -alustojen kehittymisen myötä sovellusten määrä on kasvanut runsaasti ja nopealla tahdilla. Usealla alustalle samanaikaisesti sovelluksen kehittämisen aiheuttama sovelluskoodin hajautuminen vaatii ratkaisuja. Monialustakehitys ja sen mahdollistama koodin yhteiskäyttö eri alustoilla esittelee mahdollisen ja useisiin käyttötapauksiin toimivan ratkaisun.

Monialustakehityksen erilaiset menetelmät kehittyvät nopeaa vauhtia ja ovat riippuvaisia myös laitetuesta, varsinkin web-sovelluksien ollessa kyseessä HTML5-pohjaiset ratkaisut tarvitsevat laitteilta ja valmistajilta tuen standardin mahdollistaville mobiililaiteominaisuuksille. Tulevaisuudessa web-tekniikoihin pohjautuvat ratkaisut todennäköisesti kasvattavat suosiotaan niiden helppokäyttöisyyden myötä. Useat uudet tutkimukset vaikuttivatkin keskittyvän lähinnä web- ja hybridisovelluksien sovelluskehyksien vertailuun. Tämäkin kuvastaa web-tekniikoihin pohjautuvien ratkaisujen kasvavaa suosiota.

Monialustakehityksen mahdollistavia sovelluskehyksiä on monia ja niiden määrän yhä lisääntyessä hankaloituu sopivan sovelluskehyksen valinta. Onnistunut sovelluskehyksen valinta on sovelluksen onnistuneen toteutuksen kannalta tärkeää ja siksi ennen valintaa vaihtoehtoja tulee punnita tarkkaan muun muassa projektin eri vaatimuksien ja kehittäjien pohjaosaamisen kannalta. Sovelluskehyksiä vertailevia artikkeleja löytyy useita, mutta sopivaa valintaa teh-

dessä pitää tarkistaa niiden väitteiden paikkansa pitävyys sovelluskehyksien nopeasta kehityssyklistä johtuen. Sovelluskehyksien tehokkuus kehittyy ja ominaisuudet lisääntyvät, aiheuttaen monien vertailujen päätelmien vanhentumisen.

Monialustakehitys on potentiaalinen vaihtoehto suunniteltaessa monelle eri alustalle kehitettävää sovellusta. Tehtäessä onnistunut menetelmän ja sovelluskehyksen valinta voidaan päästä hyviin lopputuloksiin, säästäen samalla resursseja monialustakehityksen hyödyistä johtuen. Kuitenkin taustalla on myös riski epäonnistuneen valinnan tai projektin aikana muuttuneiden vaatimusten osalta, jolloin monialustakehitys voi osoittautua epäonnistuneeksi valinnaksi.

Viiteluettelo

Adobe. 2017. Adobe PhoneGap. http://phonegap.com/. Checked 4.2.2017.

Amatya, S. and Kurti, A. 2014. Cross-platform mobile development: challenges and opportunities. *ICT Innovations* 231, 219-229.

Angulo, E., and Ferre, X. 2014. A case study on cross-platform development frameworks for mobile applications and UX. In: *Proc. of the XV International Conference on Human Computer Interaction*, Article 27, 8 pages.

Apache. 2017. Apache Cordova. https://cordova.apache.org/. Checked 4.2.2017.

Boushehrinejadmoradi, N., Ganapathy V., Nagarakatte S. and Iftode, L. 2015. Testing Cross-Platform Mobile App Development Frameworks. In: *Proc. of the 30th International Conference on Automated Software Engineering*, 441-451.

Dhillon, S. and Mahmoud, QH. 2014. An Evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45, 1331–1357.

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., and Wahba, A. M. 2015. Taxonomy of Cross-Platform Mobile Applications Development Approaches. *Ain Shams Engineering Journal*, 28 pages.

Heitkotter, H., Majchrzak, T.A., and Kuchen, H. 2013. Cross-platform model-driven development of mobile applications with md2. In: *Proc. of the 28th Annual ACM Symposium on Applied Computing*, 526-533.

Humayoun, S. R., Ehrhart, S., and Ebert, A. 2013. Developing mobile apps using cross-platform frameworks: a case study. In: *Proc. of the International Conference on Human-Computer Interaction* 8004, 371-380.

IDC. 2016. Smartphone OS Market Share, 2016 Q3. https://www.idc.com/promo/smartphone-market-share/os. Checked 15.2.2017.

Joorabchi, M. E., Mesbah, A. and Kruchten, P. 2013. Real Challenges in Mobile App Development. In: *Proc. of ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 15-24.

Le Goaer, O., and Waltham, S. 2013. Yet another DSL for cross-platforms mobile development. In: *Proc. of the First Workshop on the Globalization of Domain Specific Languages*, 28-33.

Litayem, N., Dhupia, B., and Rubab, S. 2015. Review of Cross-Platforms for Mobile Learning Application Development. *International Journal of Advanced Computer Science and Applications* 6, 45-48.

Majchrzak, T. and Grønli, T-M. 2016. Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks. In: *Proc. of the 50th Hawaii International Conference on System Sciences*, 10 pages.

Mercado, I. T., Munaiah, N., and Meneely, A. 2016. The impact of cross-platform development approaches for mobile applications from the user's perspective. In: *Proc. of the International Workshop on App Market Analytics*, 43-49.

NativeScript. 2017. About NativeScript. https://www.nativescript.org/about. Checked 28.1.2017.

NativeScript. 2017. NativeScript Documentation. https://docs.nativescript.org. Checked 28.1.2017.

Sencha. 2017. Sencha Touch. https://www.sencha.com/products/touch/. Checked 12.2.2017.

Sencha. 2017. Sencha Touch 2.4 Documentation. https://docs.sencha.com/touch/2.4/. Checked 12.2.2017.

Smeets, R. and Aerts, K. 2016. Trends in Web Based Cross Platform Technologies. *International Journal of Computer Science and Mobile Computing 5*, 190-199.

Statista. 2017. Number of Available Apps in the Apple App Store. https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/. Checked 15.2.2017.

Telerik. 2017. NativeScript Platform. http://www.telerik.com/nativescript. Checked 28.1.2017.

What Web Can Do Today. 2017. Can I rely on the Web Platform features to build my app? An overview of the device integration HTML5 APIs. https://whatwebcando.today/. Checked 10.1.2017.

Xamarin. 2017. Xamarin Platform. https://www.xamarin.com/platform. Checked 11.2.2017.

Xamarin. 2017. Xamarin Developer Center. https://developer.xamarin.com/. Checked 11.2.2017.

Xanthopoulos, S., and Xinogalos, S. 2013. A comparative analysis of cross-platform development approaches for mobile applications. In: *Proc. of the 6th Balkan Conference in Informatics*, 213-220.