Janne Tuovinen

Haitallisen JavaScript-koodin tunnistaminen koneoppimis-menetelmiä käyttäen

Tietotekniikan pro gradu -tutkielma

1. lokakuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Janne Tuovinen

Yhteystiedot: japetuov@student.jyu.fi

Ohjaajat: Timo Hämäläinen

Työn nimi: Haitallisen JavaScript-koodin tunnistaminen koneoppimismenetelmiä käyttäen

Title in English: Malicious JavaScript-code recognition using machine learning

Työ: Pro gradu -tutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 83+22

Tiivistelmä: Tutkimuksessa tutkitaan haitallisen JavaScript-ohjelmakoodin tunnistamista koneoppimismenetelmin opetettujen luokittelijoiden avulla. Tutkimusaiheen valintaan vaikuttivat seuraavat kolme tekijää. Tekijöistä ensimmäinen oli JavaScript-ohjelmointikielen rooli yhtenä keskeisimmistä, edelleen suosiotaan kasvattavista web-teknologioista. Toisena tekijänä oli JavaScriptin käytön yleistyminen tietoturvallisuutta vastaan suoritetuissa hyökkäyksissä. Kolmantena tekijänä puolestaan oli koneoppimismenetelmien roolin korostuminen haittaohjelmien tunnistamisessa. Teoreettiselta taustaltaan tutkimus liittyy haitallisen ohjelmakoodin staattiseen analyysiin ja haitallisen ohjelmakoodin tunnistamiseen koneoppimismenetelmin.

Tutkimuksessa keskitytään erityisesti tutkimaan luokittelijan opettamisessa käytettävien ominaispiirteiden ominaisuuksien ja opettamiseen valitun koneoppimismenetelmän vaikutusta luokittelijoiden suoriutumiseen haitallisten ja ei-haitallisten JavaScript-tiedostojen luokittelusta väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen. Tutkimuksen tulosten tuottamiseen käytetään tilastollisia menetelmiä ja silmämääräistä analyysia. Käytettyinä tilastollisina menetelminä ovat Kruskal-Wallisin testi, Dunnin testi ja keskiarvovertailu. Tutkimuksessa huomataan, että ominaispiirteiden ominaisuuksilla ja valitulla koneoppimismenetelmällä on yleisesti vaikutusta luokittelijoiden suoriutumiseen JavaScript-tiedostojen luokittelusta. Tutkimuksessa saadut tulokset ovat yleisesti samankaltaisia muiden samaan aihepiiriin liittyvien

tutkimusten tulosten kanssa väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen.

Tutkimuksen tuloksista on mahdollista tehdä ainakin seuraavat kolme johtopäätöstä. Ensimmäiseksi, luodessa haitallisen JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, kannattaa parhaan suorituskyvyn saavuttamiseksi, kokeilla eri mahdollisuuksia ainakin opetukseen ja testaamiseen käytettävien tiedostojen määrissä, käytettävien haitallisten tiedostojen määrässä opetusaineistossa, käytettävissä ominaispiirteissä, ominaispiirteiden valintaan käytettävissä menetelmissä ja käytettävässä koneoppimismenetelmässä. Toiseksi, pelkkien JavaScript-ohjelmakoodin staattisesta esitysmuodosta eristettyjen n-grammien avulla näyttää olevan mahdollista opettaa haitallisen JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, joiden suorituskyky kilpailee monimutkaisempien ominaispiirteiden avulla opetettujen luokittelijoiden kanssa. Kolmanneksi, jo suhteellisen pienillä tiedostomäärillä näyttää olevan mahdollista opettaa JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, joiden suorituskyky kilpailee suuremmilla aineistoilla opetettujen luokittelijoiden kanssa.

Avainsanat: Haitallinen JavaScript-ohjelmakoodi, haitallisen ohjelmakoodin tunnistaminen käyttäen koneoppimismenetelmiä, staattinen ohjelmakoodianalyysi

Abstract: This study studies recognition of malicious JavaScript-code using machine learning based classifiers. The choice of the research topic was influenced by facts that JavaScript has become a commonly used technology in attacks against people's information security and in the same time, machine learning is gradually increasing its role in malicious code recognition. The theoretical background of the research is related to research topics such as static code analysis and recognition of malicious code using machine learning.

The study focuses especially to study effects of a used machine learning method and effects of properties of input features to machine learning based classifiers ability to classify malicious and non-malicious JavaScript-code in terms of false positive rate, false negative rate and classification accuracy. The results of the study are produced using statistical methods and visual analysis. Statistical methods used are Kruskal-Wallis test, Dunn's test

and group average comparison. The study's results show that, in general, a chosen machine learning method and properties of input features have an effect to classifiers ability to classify malicious and non-malicious JavaScript-code. The results of the study are also similar to results of other studies studying recognition malicious JavaScript-code or recognition malicious code in general using machine learning.

Based on the results of the study, following three conclusions can be made. First, it's useful to try different possibilities in the size of a used training set, in the amount of malicious code samples in used training sets, in used input features, in a feature selection method and in a used machine learning method when training a classifier specialized for recognition of malicious JavaScript-code to find classifiers with the best classification abilities. Secondly, classifiers for malicious JavaScript-code recognition trained just with n-grams isolated from a static representation of JavaScript-source code seem to able to compete with classifiers trained with more complex feature sets in terms of classifiers classification abilities. Thirdly, classifiers for malicious JavaScript-code recognition trained with relatively small training sets seem to be able compete with classifiers trained with larger training sets in terms of classifiers classification abilities.

Keywords: Malicious JavaScript-code, malicious code recognition using machine learning, static code analysis

Esipuhe

Kiitos kesä2018, kiitos nöllipandemia, kiitos matkanelonen.

Helsingissä 01.10.2018

Janne Tuovinen

Taulukot

Sisältö

1 Johdanto

Tässä tutkimuksessa tutkitaan haitallisen JavaScript-ohjelmakoodin tunnistamista koneoppimismenetelmin opetettujen luokittelijoiden avulla. Tutkimuksen tarkoituksena on tutkia erityisesti haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen, opettamiseen ja testaamiseen käytettyjen n-grammien pituuden, opettamiseen ja testaamiseen käytettyjen n-grammien määrään, opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävän menetelmän sekä opettamiseen käytettävän koneoppimismenetelmän vaikutuksia luokittelijan kykyyn pystyä tunnistamaan haitallisia JavaScript-tiedostoja. Tutkimuksessa luokittelijoiden opettamiseen käytettyinä ominaispiirteinä toimivat ohjelmakoodin staattisesta esitystavasta muodostetut n-grammit. Tutkimusaiheen valintaan vaikuttivat kolme tekijää. Näistä tekijöistä ensimmäinen oli JavaScript-ohjelmointikielen rooli yhtenä keskeisimmistä, edelleen suosiotaan kasvattavista web-teknologioista. Toisena tekijänä oli JavaScriptin käytön yleistyminen tietoturvallisuutta vastaan suoritetuissa hyökkäyksissä. Kolmantena tekijänä toimi puolestaan koneoppimismenetelmien roolin korostuminen haittaohjelmien tunnistamisessa.

JavaScript-ohjelmointikielenä on teknologia, jonka roolia voidaan pitää nykymaailmassa erittäin merkittävä. Web-selainten kohdalla JavaScript on ollut ”de facto”-ohjelmointikieli jo melkein vuosikymmenen. Tänä aikana JavaScript on tehnyt tuloaan myös palvelin- ja työpöytäsovellus-ohjelmointiin. Näin ollen, JavaScriptiin liittyvät tietoturvauhkat ovat myös monipuolistuneet. Haitallisen koodin tuottajat pyrkivät nyt levittämään koodiaan myös paketinhallintajärjestelmien, ohjelmakirjastojen ja jopa ohjelmistojen kautta. Haitallinen JavaScript-koodi ei täten uhkaa enää vain huolimattomia webin selaajia, vaan kohdistuu yhä useammin myös ohjelmistojen kehittäjiin, JavaScriptiä käyttäviin yrityksiin sekä yritysten tuottamien ohjelmistojen loppukäyttäjiin. Ohjelmistokehittäjillä, paketinhallintajärjestelmien ylläpitäjillä ja yrityksillä ei myöskään ole aina kykyä eikä resursseja tunnistaa haitallista koodia sisältäviä paketteja, kirjastoja ja ohjelmistoja.

Haitallista ohjelmakoodia voidaan yleisesti ottaenkin pitää jo pitkäaikaisena uhkana tietoturvalla. Määritelmällisesti haitallinen ohjelmakoodi on mitä tahansa ohjelmakoodia, jonka suorittamisesta koituu jollekin taholle suoranaista haittaa tai sen suorittaminen vaarantaa jonkin tahon tietoturvan. Haitallinen ohjelmakoodi on luultavasti myös suurelle yleisölle yksi tutuimmista tietoturvaan kohdistuvista uhkista. Esimerkiksi tietokonevirus vaikuttaa olleen osa länsimaalaisen populaarikulttuurin käsitteistöä jo 1980-luvun lopulta saakka. Tästä huolimatta haitallisen ohjelmakoodin aiheuttamaa uhkaa ei ole saatu poistettua. Päinvastoin haitallista ohjelmakoodia tuotetaan enemmän kuin koskaan ja haitallisen koodin tuottajat käyttävät yhä luovempia tapoja levittääkseen ja saadakseen koodinsa suoritetuksi uhrien tietokoneilla. Koneoppiminen on puolestaan tämän hetken kuumimpia tietotekniikkaan liittyviä trendejä. Koneoppimisella ollaan saatu jo lupaavia tuloksia myös haittaohjelmien tunnistamisessa ja tulosten odotetaan jatkossa vain parantuvan.

Tässä tutkimuksessa tutkitaan koneoppimismenetelmien soveltamista ennen näkemättömän haitallisen JavaScript-koodin tunnistamiseen käyttämällä luokittelijoiden opettamiseen ja testaamiseen ominaispiirteet, jotka on johdettu suoraan JavaScript-ohjelmakoodin staattisesta esitystavasta. Tutkimuksessa käytetty ohjelman staattinen esitystapa on ohjelman alkuperäinen lähdekoodi. Tutkimuksen tarkoituksena on siis tutkia kappaleen alussa mainittujen parametrien eri arvoilla opetettujen luokittelijoiden suoriutumista JavaScript-tiedostojen luokittelusta väärien positiivisten luokitteluiden osuuden, väärien negatiivisten luokitteluiden osuuden ja luokittelun tarkkuuden suhteen.

Tutkimuksen tutkimuskysymykset ovat seuraavat:

1. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteella vaikutusta luokittelijan suorituskykyyn?

2. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien pituudella vaikutusta luokittelijan suorituskykyyn?

3. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien määrällä vaikutusta luokittelijan suorituskykyyn?

4. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävällä menetelmällä vaikutusta luokittelijan suorituskykyyn?

5. Onko haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan koneoppimismenetelmällä vaikutusta luokittelijan suorituskykyyn?

Tutkimus sisältää viisi lukua, joista ensimmäinen on tämä johdanto. Tutkimuksen toisessa luvussa käsitellään haitallista ohjelmakoodia ja haitallisen ohjelmakoodin tunnistamista niin yleisestä näkökulmasta kuin JavaScript-ohjelmakoodin näkökulmasta. Kolmannessa luvussa esitellään tutkimuksessa käytetty tutkimusaineisto ja tutkimusmenetelmä. Tutkimuksen neljännessä luvussa puolestaan kerrotaan aineiston analyysin yhteydessä saadut tulokset, pohditaan tutkimukseen liittyviä reliabiliteetteja ja validiteetteja sekä verrataan tutkimuksessa saatuja tuloksia aihepiiristä aikaisemmin tehtyjen tutkimusten tuloksiin. Tutkimus päättyy johtopäätöksiin, joissa kerrataan tutkimuksen vaiheet, vastataan edellä asetettuihin tutkimuskysymyksiin, pohditaan tutkimuksen onnistumista ja pohditaan mahdollisuuksia tutkimuksen pohjalta tehtävälle jatkotutkimukselle.

2 Haitallisen JavaScript-ohjelmakoodin tunnistaminen

Tässä luvussa käsitellään haitallista ohjelmakoodia ja sen tunnistamiseen käytettyjä menetelmiä. Luku aloitetaan aihepiirien käsittelyllä yleisessä kontekstissa, jonka jälkeen aihepiirejä käsitellään vain JavaScriptin kontekstissa. Luvun lopuksi käsitellään aiheista aikaisemmin tehtyjä tutkimuksia.

2.1 Haitallinen ohjelmakoodi

Haitallisen ohjelmakoodin voidaan katsoa määritelmällisesti olevan mitä vain ohjelmakoodia, jonka tarkoituksena on vahingoittaa koodin suorittavaa järjestelmää tai tämän järjestelmän käyttäjää (Deylami et al. 2016). Haitallisen ohjelmakoodin tarkoituksena voi olla esimerkiksi saada hallintaan tietokonejärjestelmiä ja järjestelmien verkkoresursseja, häiritä tietokoneiden operaatioita sekä kerätä henkilökohtaista tietoa järjestelmien käyttäjistä ilman käyttäjien hyväksyntää (Gandotra et al. 2014). Haitallista ohjelmakoodia luodaan useista syistä. Taustalla voivat muun muassa taloudelliset, poliittiset, rikolliset tai terroristiset motiivit (Moskovitch et al. 2008). Haittaohjelmien kirjoittamisesta on myös tullut entistä helpompaa, koska internetissä on runsaasti tarjolla kirjoittamista helpottavia työkaluja (Kolter & Maloof 2004).

Haittaohjelmat ovat käytökseltään erilaisia. Niiden käyttäytyminen vaihtelee aina yksinkertaisesta tiedostojen muokkaamisesta hyvinkin monimutkaisiin esimerkiksi verkon yli tehtäviin operaatioihin (Rieck et al. 2010). Haittaohjelmat voidaan jakaa kategorioihin ohjelmakoodin ominaisuuksien perusteella. Näitä ominaisuuksia voivat olla esimerkiksi ohjelmakoodin kyky replikoitua, monistua, suorittaa itsensä ja vaurioittaa systeemin käyttöjärjestelmää (Krishna et al. 2016). Tunnetuimpia haittaohjelmakategorioita ovat: virukset, madot, troijalaiset ja vakoiluohjelmat (Deylami et al. 2016). Haittaohjelmien luokitukset eivät kuitenkaan ole toisiaan pois sulkevia, vaan haittaohjelmat voivat olla ominaisuuksiensa perusteella sijoitettavissa useampaan kuin yhteen kategoriaan (Gandotra et al. 2014). Samasta alkuperästä peräisin olevien haittaohjelmien voidaan sanoa muodostavan haittaohjelmaperheen. Saman haittaohjelmaperheen jäsenet jakavat keskenään samoja käytösmalleja (Rieck et al. 2010).

Haittaohjelma leviävät useilla eri tavoilla. Aikaisemmin suosittu tapa haittaohjelmien levittämiseen oli etsiä haavoittuvuuksia järjestelmien verkkoa käyttävistä palveluista ja käyttää löydettyjä haavoittuvuuksia hyväksi esimerkiksi tietokonematojen avulla. Tämä on kuitenkin käynyt yhä vaikeammaksi teknologioiden kuten palomuurien kehityttyä (Ravi et al. 2014). Webin suosion räjähdettyä hyökkääjät ovat siirtyneet levittämään haittaohjelmiaan muun muassa web-selainten haavoittuvuuksia hyödyntäen (Ravi et al. 2014).

Suurin osa näitä haavoittuvuuksia hyödyntämään pyrkivistä hyökkäyksistä on suunnattu selainten muistin korruptoivia haavoittuvuuksia vastaan (Cova et al. 2010). Esimerkkinä tällaisista hyökkäyksistä ovat niin sanotut Drive by Download-hyökkäykset, joissa hyökkäyksen tarkoituksena on web-selainten haavoittuvuuksia hyödyntäen ladata web-sivuilla tietoisesti tai tietämättään vierailevien käyttäjien koneelle haittaohjelmia (Ravi et al. 2014). Saamalla web-selain suorittamaan omia komentojaan pääsee hyökkää käyttäjän palomuurin ohitse asentamaan haittaohjelmansa. JavaScript-koodilla on tällaisten hyökkäysten toteuttamisessa keskeinen rooli (Provos et al. 2007).

2.2 Haitallisen JavaScript-ohjelmakoodin ominaisuudet

JavaScript on dynaaminen asiakaspuolen ohjelmointikieli, joka on yksi kolmesta webin sisällön tuotantoon käytetystä pääteknologiasta HTML:n ja CSS:n ohella. Kaikki modernit web-selaimet tukevat JavaScriptin käyttöä. JavaScript on myös yleisimmin ja menestyksekkäimmin tietoturvaa vastaan tehdyissä hyökkäyksissä käytetty ohjelmointikieli. Haitallinen JavaScript-ohjelmakoodi asetetaan web-sivun yhteyteen, jolloin koodi tulee suoritetuksi kaikissa sivun ladanneissa selaimissa. Tällöin ohjelmakoodi pääsee turvatoimenpiteiden kuten palomuurien ja virustentorjuntaohjelmistojen ohi (Patil & Patil 2017). Suurin osa web-selainten käyttäjistä myös sallii JavaScriptin suorittamisen selaimissaan. Suojellakseen käyttäjiä selaimet pyrkivät rajoittamaan resursseja, joita JavaScriptillä on oikeus käyttää (Likarish & Jo 2009).

Eräs tärkeimmistä web-sivujen turvallisuutta edistävistä käytännöistä on niin sanottu ”Same Origin Policy” eli ”Saman alkuperän käytäntö”. Käytäntöä sovelletaan web-sivujen kaikkeen aktiiviseen sisältöön kuten JavaScript-koodiin. Käytännön mukaan web-sivulla

olevalla JavaScript-koodilla on oikeus lukea ja/tai kirjoittaa elementteihin, ikkunoihin ja dokumentteihin, jotka jakavat saman alkuperän kyseisin koodin kanssa (Johns 2008). Sama alkuperä määritellään resurssin sisältävän JavaScript-koodin hakemiseen käytetyn protokollan, verkkotunnuksen ja portin perusteella. Käytännön lisäksi selaimessa ajettavalla JavaScript-koodilla ei ole suoraa käyttöoikeutta järjestelmän lokaaliin tiedostojärjestelmään. Lokaaleihin tiedostoihin on mahdollista päästä käsiksi vain ”file://”-metaprotokollaan avulla (Johns 2008).

JavaScript-koodin avulla voidaan kuitenkin dynaamisesti liittää elementtejä mielivaltaisista lähteistä koodin säiliödokumentin dokumenttiobjektimalliin. Tämä poikkeus ja tosiasia, että saman alkuperän käytäntö pätee dokumentin tasolla, luovat aukon Saman alkuperän käytäntöön. JavaScriptillä ei edelleenkään ole suoraa pääsyä ulkoisiin kohteisiin, mutta sen avulla voidaan kuitenkin johtaa moninaista informaatiota liittämällä ulkoinen kohde sivun dokumenttiobjektimalliin puuhun. JavaScriptin avulla voidaan esimerkiksi vastaanottaa ja siepata tapahtumia, jotka syntyvät tällaisissa liittämisprosesseissa. Lisäksi liittämisprosessin päätyttyä ulkoinen kohde on osa dokumenttia, jolloin JavaScriptillä päästään käsiksi joihinkin kohteen ominaisuuksista, jolloin nämä ominaisuudet ovat luettavissa JavaScriptin-koodista käsin (Johns 2008).

Kuten jo edellä mainittiin, haittaohjelmia pyritään muun muassa levittämään web-sivujen avulla. Levittämiseen voidaan käyttää useita eri tapoja kuten: käyttäen sivuille asetettuja valheellisia mainoksia, käyttäen valheellisten blogeja, käyttäen sosiaalista mediaa, ohjaamalla käyttäjä valheelliselle latauslinkkejä sisältävälle sivulle tai näyttämällä käyttäjälle huijauskoodekkien asennuspyyntöjä. Haitallisen JavaScript-koodin löytäminen web-sivuilta esimerkiksi tutkimusaineistoksi on kuitenkin varsin haastavaa, koska ohjelmat ovat lyhyt ikäisiä. Sivujen ylläpitävät poistavat haitallisen ohjelmakoodin nopeasti, mikäli sellaista sivuillaan havaitsevat. Lisäksi haitallisen ohjelmakoodin kirjoittavat eivät halua paljastaa käyttämiänsä tekniikoita, joten haitallinen ohjelmakoodi voi tulla poistetuksi myös itse kirjoittajien toimesta (Likarish & Jo 2009).

Eräs JavaScript-ohjelmakoodin avulla selainten käyttäjiä vastaan tehtävien hyökkäysten kategoria on tiedusteluhyökkäykset. Tiedusteluhyökkäyksien tarkoituksena on selvittää

jonkin määritellyn olion olemassaolo. Tämä olio voi olla jokin ulkoinen resurssi tai selaimen käyttäjän tietokoneella oleva lokaali olio, kuten jokin lokaalisti pyörivä sovellus. Tiedusteluhyökkäyksissä käytetään hyödyksi JavaScript-kielen tapahtumien käsittelyyn tarkoitettua kehikkoa, jonka avulla voidaan kaapata useita tapahtumia sivujen latauksen liittyen. Tiedusteluhyökkäyksissä käytetään erityisesti ”onload”-, ”onerror”-, ja ”timeout”-tapahtumia (Johns 2008).

JavaScriptin avulla tiedusteluhyökkäys voidaan toteuttaa esimerkiksi seuraavasti. Ensin JavaScript-ohjelmakoodin avulla web-sivulle lisätään dynaamisesti jokin verkkoa hyödyntävä elementti. Elementti voi olla esimerkiksi kuva tai ”iframe”-elementti. Elementin lähteenä oleva URL-osoite laitetaan osoittamaan kohteeseen, jonka olemassaolosta halutaan saada tietoa. Onload-tapahtuman laukeaminen varmistaa kohteen olemassaolon. Onerror-tapahtuman laukeamisen yhteydessä saatava informaatio riippuu puolestaan kontekstista. Onerror-tapahtuma laukeaa, mikäli selaimen vastaanottama data ei sovi käytettyyn elementtiin tai jos verkkoyhteys katkaistiin. Onerror-tapahtuman tapauksessa JavaScript-konsolia voidaan käyttää lisäinformaation keräämiseen. Tiedusteluhyökkäyksien yhteydessä voidaan myös hyödyntää timeout-tapahtumia. Timeout-tapahtuman esiintyminen ennen mitään muuta tapahtumaa implikoi avointa verkkoyhteyttä, joka puolestaan implikoi, ettei resurssia ehkä ole olemassa (Johns 2008).

Selaimessa suoritettavan JavaScriptin-ohjelmakoodin avulla voidaan myös yrittää kerätä tietoa esimerkiksi ohjelmakoodia suorittavasta selaimesta, selainta pyörittävästä tietokoneesta ja selaimen käyttäjästä. Tämä tieto voi olla esimerkiksi käyttäjän selaushistoria, tietoa siitä onko selaimen käyttäjä kirjautuneena johonkin web-palveluun, tietoa käyttäjän asentamista selaimen lisäosia tai tietoa muista käyttäjän tietokoneen tunnusmerkeistä (Johns 2008). JavaScriptiä voidaan lisäksi käyttää tietoturvan kannalta puutteellisten ohjelmistojen havaitsemiseen käyttäjän tietokoneella, jonka jälkeen hyökkääjän on mahdollista käyttää hyväksi mahdollisesti havaitsemaansa puutetta käyttäjän tietoturvassa (Likarish & Jo 2009).

Selaimen selaushistoriaa voidaan yrittää selvittää esimerkiksi liittämällä web-sivulle linkkejä, jonka jälkeen selvitetään linkin CSS-attribuuteista, onko käyttäjä vieraillut sivulla.

Toinen tapa on lisätä web-sivulle kuva joltakin sivulta ja onload-tapahtuman avulla mitata sen lataukseen käytetty aika. Kuva on tallennettuna selaimen välimuistiin, mikäli käyttäjä on vieraillut lähiaikoina aiemmin sivulla, joka sisältää kyseisen kuvan. Tällöin kuvan latausaika on lyhyempi, kuin mikäli kuva jouduttaisiin lataamaan verkon kautta. JavaScriptin avulla voidaan yrittää myös päätellä kuinka kauan käyttäjältä kestää jonkin sivun lataamisessa. Tämä tapahtuu esimerkiksi lisäämällä web-sivulle kuva, jonka lähteen URL osoittaa haluttuun sivuun. Kaappaamalla kuvan lataamisen yhteydessä tällöin ilmenevä onerror-tapatuma sekä mittaamalla kuvan luontiprosessin ja onerror-tapahtuman välissä olevan aika, saadaan selvitettyä, kuinka kauan selaimella kesti sivun lataamisessa (Johns 2008).

Tietoa selaimen käyttäjän kirjautumisesta johonkin web-sovellukseen voidaan yrittää puolestaan selvittää lataamalla jokin HTML-sivu JavaScriptille varatun avainsanan sisään ja tulkitsemalla tästä aiheutuva JavaScriptin parsimiseen liittyvä virhe. Web-sovellukset vastaavat usein erilaisella HTML-sivulla riippuen siitä, onko selaimen käyttäjä kirjautuneena kyseiseen web-sovellukseen. Lisäksi HTML-sisällön tuottama JavaScriptin parsimiseen liittyvä virhe on usein riippuvainen HTML-sivun sisällöstä. Tällöin kaappaamalla parsimiseen liittyvän virheen ja tulkitsemalla virheeseen liittyvän viesti, voidaan mahdollisesti päätellä, onko selaimen käyttäjä kirjautuneena kyseiseen web-sovellukseen (Johns 2008).

Lisäksi JavaScript-ohjelmakoodilla on pääsy myös moniin web-selainta pyörittävän tietokoneen lokaaleihin resursseihin erityisten URL-skeemojen kautta. Näitä skeemoja ovat esimerkiksi ”file://” ja ”chrome://”. Kyseisiä skeemoja voidaan puolestaan hyödyntää tiedusteluhyökkäyksissä, joissa yritetään kalastella tietoa selaimen lokaalista kontekstista. URL-skeemoihin perustuvien hyökkäysten yleispätevyyttä kuitenkin rajoittaa se, että skeemoja käsitellään käytetystä selaimesta riippuen eri tavoin (Johns 2008).

Toinen JavaScript-koodin avulla toteutettavissa oleva hyökkäystyyppi on niin sanottu ”sivujen välinen palvelupyyntöväärennys” tai ”sessiolla ratsastaminen”. Tällaisessa hyökkäyksessä uhrin selaimesta tehdään JavaScript-koodin avulla HTTP-palvelupyyntö johonkin rajoitettuun resurssiin. Palvelupyyntö voidaan toteuttaa esimerkiksi piilottamalla web-sivulle kuva. Kuvan lähteeksi asetetaan jokin tilaa muuttava URL-osoite, joka osoittaa johonkin rajoitettuun resurssiin. Tällöin selain välittää resurssissa autentikointiin tarvittavat

tiedot automaattisesti palvelupyynnön mukana, mikäli uhri on jo valmiiksi kirjautuneena kyseessä olevaan resurssiin (Johns 2008).

Kolmas usein käsitelty JavaScript-koodin avulla toteutettavissa oleva hyökkäystyyppi on ”Drive by Download”-hyökkäys. Drive by download-hyökkäyksessä hyökkääjä sijoittaa aluksi selaimen muistiin koodia, joka on vastuussa varsinaisesta hyökkäyksestä. Tätä koodia kutsutaan yleensä ”Shell”-koodiksi. Shell-koodi on usein kirjoitettu konekielellä. Shell-koodin sijoittaminen selaimen muistiin tapahtuu niin sanotuin laillisin JavaScript-operaatioin, kuten sijoittamalla koodi merkkijonona johonkin muuttujaan. Hyökkäyksen seuraavassa vaiheessa hyökkääjä yrittää kaapata selaimen koodin suorittamisesta vastaavaan prosessin ja ohjata sen suorittamaan omaa koodiaan. Tämä tapahtuu käyttämällä hyväksi jotain haavoittuvuutta joko selaimessa itsessään tai jossain selaimen lisäosassa. Tavoitteena on usein pyrkiä ylikirjoittamaan selaimen funktio-osoitin esimerkiksi selaimen muistikeon ylivuodon avulla (Cova et al. 2010).

Hyökkäyksien piilottamiseen käytetään usein piilotettuja iframe-elementtejä tai JavaScript-koodin obfuskointia. Lisäksi hyökkäyksiin käytettyjä palvelimet tarjoilevat usein haitallisen JavaScript-ohjelmakoodin vain niissä tapauksissa, joissa käyttäjän web-selain ja käyttöjärjestelmä mahdollistavat jonkin kyseisen palvelimen tunteman hyökkäyksen. Palvelin saa tiedot käyttäjän selain ja käyttöjärjestelmästä HTTP-pyynnön mukana olevan ”User agent”-otsikon avulla (Ravi et al. 2014). Mikäli JavaScript-koodi ei havaitse käyttäjän koneella hyödynnettävissä olevaa haavoittuvuutta voidaan käyttäjää joskus yrittää huijata lataamaan ja suorittamaan haitallinen koodi manuaalisesti (Provos et al. 2007).

2.3 Haitallisen JavaScript-ohjelmakoodin obfuskointi

Tunnistamisen vaikeuttamiseksi haitallisen ohjelmakoodin kirjoittajat pyrkivät lisäämään koodinsa kompleksisuutta käyttäen erilaisia tekniikoita. Tällaisia tekniikoita ovat esimerkiksi polymorfismi, metamorfismi, koodin pakkaaminen ja obfuskointi. Tekniikoiden käyttämisen johdosta samankaltaisiin hyökkäyksiin perustuvista haittaohjelmista on löydettävissä useita versioita ja toteutuksia (Hanset et al. 2016).

Termi obfuskointi liittyy oleellisesti haitalliseen ohjelmakoodiin, erityisesti haitalliseen JavaScript-koodiin. Obfuskointi on kokoelma tekniikoita, joilla pyritään piilottamaan haittaohjelman todelliset aikeet ilman, että haittaohjelmaan lisätään haitallista käyttäytymistä (Idika et al. 2007). Usein obfuskointiin käytetään kerralla myös useampaa kuin yhtä obfuskointitekniikkaa (Xu et al. 2012). Uudet haittaohjelmat ovat usein aikaisemmin tunnettuja haittaohjelmia, jotka ovat joko uudelleen pakattu tai obfuskoitu (Tabish et al. 2009). Suurin osa haitallisesta JavaScript-ohjelmakoodista on nykyisin obfuskoitua (Xu et al. 2012). Haitallisen JavaScript-koodin obfuskoinnin suosion kasvuun ovat vaikuttaneet ainakin seuraavat kaksi tekijää. Ensinnäkin staattisia allekirjoitukseen perustuvia menetelmiä on helppo huijata obfuskoinnilla. Toiseksi JavaScript-kielen dynaamiset ominaisuudet tekevät koodin obfuskoinnista suhteellisen helppoa (Xu et al. 2013).

Tunnettuja JavaScript-koodin obfuskointiin käytettyjä tekniikoita ovat ainakin sattumanvarainen obfuskointi, dataobfuskointi, uudelleen koodaaminen ja ohjelmakoodin loogisen rakenteen obfuskointi. Sattumanvaraisessa obfuskoinnissa koodiin lisätään elementtejä tai koodin sisältämiä elementtejä muutetaan sattumanvaraisesti muuttamatta koodin semantiikkaa. Dataobfuskoinnissa puolestaan muuttujia ja/tai vakioita muutetaan toisten muuttujien ja/tai vakioiden laskennallisiksi tuloksiksi. Dataobfuskointitekniikoita ovat esimerkiksi merkkijonojen halkaiseminen ja avainsanojen korvaaminen (Xu et al. 2012). Uudelleen koodauksessa alkuperäinen haitallinen ohjelmakoodi puolestaan uudelleen koodataan käyttäen esimerkiksi ASCII-, unicode- tai heksadesimaaliesitystapaa (Xu et al. 2012). Obfuskointiin käytetään yleisesti kombinaatioita välillä 0-9 olevista numeroista, heksadesimaaliarvoista ja sellaisista erikoismerkeistä kuten ”%” , ”(”,”)”, ”;”, ”#”, ”|”, ”[”, ”]”, ”{”, ”}” (Patil & Patil 2017). Uudelleen koodaukseen voidaan käyttää myös kustomoituja koodausfunktioita ja ohjelmakoodin suorituksen aikana koodauksen purkavia funktioita. Ohjelmakoodin loogisen rakenteen obfuskoinnissa puolestaan ohjelmakoodin rakennetta pyritään muuttamaan vaikuttamatta ohjelmakoodin alkuperäiseen semantiikkaan (Xu et al. 2012).

Xu ja kumppanit (2013) analysoivat yli 30 000 haitallista ja obfuskoitua JavaScript-koodia näytettä. Tutkijat huomasivat, että 71 prosenttia heidän näytteistään oli obfuskoitu käyttäen joko yhtä tai useampaa obfuskointitekniikkaa. Dataobfuskointia käytettiin 47 prosentissa näytteistä. Merkistöobfuskointia käytettiin 32 prosentissa näytteistä. Kustomoituja obfus-

koinnin purkamiseen käytettyjä funktioita käytettiin 23 prosentissa näytteistä. Salausta käytettiin 3 prosentissa tapauksia ja loogisen rakenteen obfuskointia käytettiin 11 prosentissa näytteistä.

Vuonna 2012 Xu ja kumppanit tutkivat obfuskoinnin vaikutusta tunnettujen antivirusohjelmistojen kykyyn tunnistaa haitallisia ohjelmia. Tutkijat aloittivat testaamalla haitallisen JavaScript-koodin tunnistamista valikoimillaan 20 antivirusohjelmistolla. Tuloksena oli keskimäärin noin 87 prosentin tunnistamisaste. Kun tutkijat käyttivät ohjelmakoodinäytteisiin sattumanvaraista obfuskointia, niin ohjelmistojen tunnistamisaste tippui noin 55 prosenttiin. Kun tutkijat käyttivät näytteisiin dataobfuskointia, niin ohjelmistojen tunnistamisaste tippua noin 46 prosenttiin. Kun näytteet obfuskointiin käyttämällä uudelleen koodausta ei valituista ohjelmistoista enää yksikään pystynyt tunnistamaan näytteitä haitallisiksi (Xu et al. 2012).

Xu ja kumppanit (2013) huomasit myös, että kahta yleistä operaatiota käytetiin usein edellä mainittujen tekniikoiden yhteydessä. Ensimmäinen oli haitallisen ohjelmakoodin palauttaminen puhtaaksi tekstiksi koodatusta merkkijonosta tai toiselta web-sivulta. Toinen oli palautetun koodin suorittaminen käyttäen dynaamista koodingenerointia ja suorituksen aikaista evaluointifunktiota. JavaScript-kielen dynaamiset ominaisuudet kuten dynaaminen koodingenerointi ja suorituksen aikainen evaluointi ovatkin usein obfuskointikäytäntöjen takana. JavaScriptin dynaamiset ominaisuudet mahdollista tekstin muuttamisen suoritettavaksi koodiksi. Esimerkiksi merkkijonojen manipulointi voidaan yhdistää funktioon, joka pystyy dynaamiseen koodingenerointiin tai suorituksen aikaiseen arviointiin. Tällaisia funktioita ovat muuan muassa document.write- tai eval-funktiot (Xu et al. 2012).

Dynaamisen koodingeneroinnin tai suorituksen aikaisen koodinevaluoinnin käyttäminen ei kuitenkaan suoraan tarkoita obfuskointia. Dynaamista koodingenerointia voidaan käyttää esimerkiksi ulkopuolisten JavaScript-kirjastojen lataamiseen. Suorituksen aikaista koodinevaluointia puolestaan voidaan käyttää esimerkiksi tilanteissa, joissa koodin suorittamiseen tarvittava informaatio on saatavissa vasta ohjelman suorituksen aikana (Xu et al. 2013). Wang ja kumppanit (2015) kuitenkin huomasivat, että 98 prosenttia heidän analysoimistaan ei-haitallisista JavaScript-ohjelmista ei käyttänyt eval-funktiota kuin maksimis-

saan kerran. Kun taas 87 prosenttia heidän analysoimista haitallisista JavaScript-ohjelmista käytti kyseistä funktiota enemmän kuin kaksi kertaa.

Dynaamisesta koodingenerointi vaikeuttaa koodin analysointia, koska kaikki lopulta suoritettavaksi päätyvä koodi ei ole saatavilla analyysihetkellä. Tämä pätee varsinkin staattisen analyysin tapauksessa (Kapravelos et al. 2013). Dynaamisen koodingeneroinnin tai suorituksen aikaisen koodinevaluoinnin mahdollistavien funktioiden argumentteina käytetään yleensä toisia funktiota. Tällä tavalla funktioiden argumentit yritetään piilottaa staattisesta näkökulmasta (Xu et al. 2013). Näin ollen, tällaisten funktioiden syötteiden tutkiminen voi olla hyödyllistä, kun halutaan arvioida JavaScript-koodin haitallisuutta (Xu et al. 2012). Valitettava kuitenkin obfuskoitu JavaScript-koodi täytyy usein suorittaa jollakin JavaScript-moottorilla ennenkö koodin todellinen toiminnallisuus saadaan selville (Kapravelos et al. 2013).

Obfuskoitu JavaScript-koodi ei kuitenkaan ole automaattisesti haitallista. Obfuskointia käytetään myös ei-haitallisen JavaScript-koodin suojamiseen esimerkiksi plagiarismilta. Ei-haitallisen koodin obfuskoinnissa päätarkoituksena on kuitenkin vain muuttaa koodi ihmisille lukukelvottomaksi (Xu et al. 2012). Tällaisissa tapauksissa suositaan yleensä satunnaistamiseen ja korvaamiseen perustuvia obfuskointitekniikoita. Ei-haitallista koodia obfuskoidessa pyritään usein välttämään dynaamiseen generointiin perustuvia obfuskointitekniikoita, koska koodin tekijät ovat huolissaan näiden tekniikoiden aiheuttamasta koodin suorituskyvyn laskusta. Lisäksi ei-haitallisen ohjelmakoodin tapauksessa funktiot ovat usein määritelty ennen niiden kutsumista. Kun taas haitallisen obfuskoidun ohjelmakoodin tapauksessa haitallisten funktioiden määrittely voi olla osittain tai kokonaan obfuskoitu, jotta niiden semantiikka pysyisi piilossa (Xu et al. 2013).

Edellä mainittujen obfuskointitekniikoiden lisäksi haittaohjelmien tekijät ovat pyrkineet luomaan haitalliseen koodin toimintoja, jotka pyrkivät havaitsemaan dynaamiset analyysisysteemit. Koodin havaitessa tällaisen systeemin koodi käyttäytyy eri tavoin kuin tilanteessa, jossa systeemiä ei havaita. Näin koodi pyrkii välttämään analysoitavaksi joutumiseen ja välttämään havaituksi tulemisen. Analysointia välttelevä ohjelmakoodi voi esimerkiksi

yrittää havaita ihmisinteraktiota tai yrittää havaita järjestelmässä olevia tunnettuja analyysityökaluja (Kapravelos et al. 2013).

2.4 Haittaohjelmien tunnistajat

Haitallisen ohjelmakoodin tunnistaja on systeemi, joka pyrkii tunnistamaan haitallisen ohjelmakoodin jotain menetelmää käyttäen. Tunnistaja voi sijaita isäntäkoneessa tai jossakin ulkoisessa resurssissa kuten esimerkiksi palvelimella (Deylami et al. 2016). Tunnetuimpia esimerkkejä tällaisista systeemeistä ovat antivirusohjelmistot. Vähemmän tunnettuja systeemejä ovat puolestaan ”hunajapöntöt” (Honeypot) ja ”hunaja-asiakkaat” (Honeyclient). Hunaja-asiakkaat ovat systeemejä, jotka tekeytyvät web-selaimiksi ja vierailevat websivuilla tarkoituksenaan analysoida sivuja esimerkiksi haitallisen JavaScript-koodin varalta. Tällaisilla työkaluilla pystytään myös analysoimaan koodin suorittamista aiheutuvia seuraamuksia (Kapravelos et al. 2013).

Haittaohjelman tunnistajan tärkeimpänä ominaisuutena voidaan pitää kykyä tunnistaa mahdollisimman paljon erilaisia haittaohjelmia ilman virheellisiä positiivisia havaintoja (Deylami et al. 2016). Vaikeaksi tämän tekevät haittaohjelmien määrä ja haittaohjelmien monimuotoisuus (Idika & Mathur 2007). Tämän lisäksi tunnistaja täytyy olla mahdollista päivittää (Deylami et al. 2016). Perinteisesti tunnistajien käyttämät menetelmät voidaan jakaa ohjelmakoodisigneeraukseen (signature-based) ja ohjelman tai järjestelmän käytöksessä esiintyvien anomalioiden (behavioral-based) tunnistamiseen perustuviin menetelmiin. Signeeraukseen perustuvassa tunnistamisessa ohjelmakoodista muodostetaan niin sanottu signeeraus, jota sitten verrataan tunnettujen haittaohjelmien signeerauksiin (Deylami et al. 2016). Signeeraus on tyypillisesti jokin koodisekvenssi. Anomalioihin tunnistamiseen perustuvat menetelmät puolestaan sisältävät opetusvaiheen ja tunnistamisvaiheen. Opetusvaiheen aikana menetelmät pyrkivät opettamaan mallin, joka kuvaa järjestelmän tai ohjelman normaalin käyttäytymisen. Tunnistamisvaiheessa analysoitavan järjestelmän tai ohjelman käyttäytymistä verrataan malliin. Näin pyritään havaitsemaan järjestelmän tai ohjelman epänormaali käyttäytyminen (Idika et al. 2007).

Menetelmätyypeillä on omat vahvuudet ja heikkoudet. Signeeraukseen perustuvien menetelmien vahvuutena on kyky tunnistaa tunnetut haittaohjelmat tarkasti suhteellisen minimaalisin systeemiresurssivaatimuksin (Deylami et al. 2016). Signeeraukseen perustuvat menetelmät kuitenkin hakevat signeeraukset tyypillisesti tietokannoista, joita ylläpitävät tyypillisesti haittaohjelmien tunnistamiseen ja haittaohjelman käytöksen konetunnistettavan signeerauksen muodossa ilmaisemaan kykenevät ihmiset. Tällaisten tietokantojen ylläpitäminen on kuitenkin työlästä ja vaatii ylläpitäjiltä erityistä asiantuntijuutta (Idika et al. 2007). Tästä johtuen on aina olemassa jokin aikaväli uuden haittaohjelman löytymisen ja allekirjoituksen saatavuuden välillä. Tämän lisäksi haittaohjelmat kuten tietokonemadot voivat myös muuttua levitessään, mikä tekee myös allekirjoituksiin perustuvissa menetelmistä vähemmän tehokkaita (Kruegel 2007). Signeeraukseen perustuvilla menetelmillä siis havaitaan tarkasti haittaohjelmat, jotka on jo aiemmin tunnistettu. Kyseisten menetelmien suurimpana heikkoutena voidaankin pitää niiden kykenemättömyys tunnistaa uusia ennen näkemättömiä haittaohjelmia (Deylami et al. 2016). Narayanan ja kumppanit (2013) huomauttavat kuitenkin, että signeeraukseen perustuvilla menetelmillä voidaan tunnistaa uusia haittaohjelmia tapauksissa, joissa koodi ja toiminnallisuus on jaettua jonkin jo tunnetun haittaohjelman kanssa. Näin ollen, signeeraukset perustuvat menetelmät pystyvät myös tunnistamaan tunnettujen haittaohjelmaperheiden uusia variantteja.

Ohjelmien tai järjestelmien käyttäytymisessä esiintyvien anomalioiden havaitsemiseen perustuvat menetelmät ovat puolestaan vahvuuksien ja heikkouksien suhteen kuin peilikuva signeerauksiin perustuvista menetelmistä. Anomalioiden havaitsemiseen perustuvien menetelmien vahvuutena on erinomainen kyky havaita uudet ennen näkemättömät haittaohjelmat, mutta samalla aikaa menetelmät tuottavat runsaasti virheellisiä positiivisia havaintoja (Deylami et al. 2016). Kyseisten menetelmien avulla voidaan havaita tehokkaasti myös jo olemassa olevista haittaohjelmista tehdyt muunnokset, sillä haittaohjelmien käyttäytyminen ja rakenne ovat usein vaikeammin muutettavia kuin esimerkiksi ohjelman lähdekoodiesitys (Kruegel 2007). Anomalioiden havaitsemiseen perustuvat menetelmät vaativat kuitenkin jatkuvaa tiedon keräämistä ohjelman tai järjestelmän suorituksessa, joten ne ovat signeeraukseen perustuviin menetelmiin nähden laskennallisesti vaativampia. Tämän

lisäksi anomalioiden tunnistamiseen käytettyjen mallien opettamiseen käytettyjen parametrien valitseminen voi olla joskus hankalaa (Deylami et al. 2016).

Ohjelmakoodisigneeraukseen ja anomalioiden tunnistamiseen perustuviin menetelmiin lisäksi on eriytettävissä vielä kolmas menetelmätyyppi: ohjelmien spesifikaatioihin perustuva tunnistaminen. Näitä menetelmiä käytettäessä ohjelmista luodaan spesifikaatiota, jotka kuvaavat ohjelmien tunnistamiseen tarvittavat yksityiskohdat. Tällaisessa kuvauksessa voidaan esimerkiksi kuvata ohjelman käyttäytyminen sen suorituksen aikana. Havaitsemisvaiheessa tarkkailun kohteena olevaa ohjelmaa verrataan sen spesifikaatioon. Spesifikaation perustuvan tunnistamisen vahvuutena on uusien ja ennen näkemättömien hyökkäysten tunnistaminen sekä vähäiset määrät virheellisiä positiivisia havaintoja. Menetelmätyypin heikkoutena on puolestaan yksityiskohtaisten spesifikaatioiden tekemisen vaikeus ja aikavaativuus. Tämän lisäksi anomalioiden tunnistaminen on usein spesifikaatioihin perustuvaa tunnistamista tehokkaampi keino havaita uudet ennen näkemättömät (Deylami et al. 2016).

2.5 Staattinen ja dynaaminen ohjelmakoodianalyysi

Signeeraukseen perustuvat menetelmät sekä ohjelman tai järjestelmän käytöksessä esiintyvien anomalioiden tunnistamiseen perustavat menetelmät voidaan molemmat jakaa vielä kolmeen alakategoriaan: staattiseen analyysiin perustuviin menetelmiin, dynaamiseen analyysiin perustuviin menetelmiin ja hybridianalyysiin perustuviin menetelmiin. Jako perustuu tapaan, jolla haitallista ohjelmakoodia analysoidaan (Deylami et al. 2016).

Haittaohjelmien analysointia ilman koodin suorittamista kutsutaan staattiseksi analyysiksi (Gandotra et al. 2014). Kun taas dynaamisessa analyysissa tarkasteltava ohjelmakoodi suoritetaan jollakin laitteistolla jossakin analyysin vaiheessa (Deylami et al. 2016). Dynaamisessa analyysissa koodi yleensä suoritetaan jossain eristetyssä ja hallitusta järjestelmässä samalla analysoiden koodin vuorovaikutusta järjestelmän kanssa. Tarkkailun alla ovat muuan muussa systeemin tiedostojärjestelmä, systeemirekisteri, järjestelmäprosessit ja verkkoresurssit sekä systeemissä tapahtuvat muutokset (Gandotra et al. 2014). Hybridiana-

lyysi on puolestaan jonkinlainen yhdistelmä staattista analyysia ja dynaamista analyysiä (Deylami et al. 2016).

Staattisessa analyysissa analysoitavat tiedostot ensin puretaan ja niiden mahdollinen salaus poistetaan. Varsinaisessa analyysissä puolestaan pyritään eristämään analysoitavasta ohjelmakoodista ominaispiirteitä, joiden avulla ohjelmakoodi voidaan mahdollisesti luokitella haitalliseksi (Damodaras et al. 2017). Perinteisesti staattiset menetelmät tutkivat analysoitavia ohjelmia konekielen tasolla (Tabish et al. 2009).

Korkealla tasolla nämä ominaispiirteet tarkoittavat esimerkiksi ohjelmoinnin rakennetta tai formatointia (Deylami et al. 2016). Yksityiskohtaisemmalla tasolla erityispiirteet voivat olla muuan muassa merkkijonosigneerauksia, tavusekvenssi n-grammeja, erityisiä kirjastokutsuja, tilakaavioita tai operaatiokoodin frekvenssijakaumia (Gandotra et al. 2014). Dynaamisessa analyysissa puolestaan pyritään analysoimaan järjestelmän tilaa ennen ohjelmakoodin suorittamista, suorittamisen aikana ja suorittamisen jälkeen. Tämän lisäksi pyritään analysoimaan ohjelmakoodin suoritusta ja sen suorituksen aikana tekemiä rajapintakutsuja (Deylami et al. 2016).

Dynaamista analyysia pidetään staattista analyysia tehokkaampana tapana tunnistaa haitallinen ohjelmakoodi, mutta se on samalla vaikeammin skaalattavissa, koska dynaaminen analyysi vaatii staattista analyysia enemmän resursseja (Gandotra et al. 2014). Ongelmana on, että API-kutsujen tarkkailu ohjelmien suorituksen aikana alentaa merkittävästi tietokoneen suorituskykyä (Tabish et al. 2009). Koneoppimis- ja tiedonlouhintamenetelmien luokittelumallien opettamisvaiheessa tämä ei kuitenkaan haittaa, mutta mallien arviointi vaiheessa dynaaminen analyysi on todennäköisesti epäkäytännöllinen, erityisesti tapauksissa, joissa joudutaan käymään läpi suuria määriä tiedostoja (Damodaras et al. 2017).

Staattisessa analyysissä tiedostojen analyysi voidaan puolestaan suorittaa, kun tietokoneen resurssit ovat vapaana. Analyysi voidaan myös suorittaa käyttäen ulkoisia resursseja, kuten pilvipalvelua (Tabish et al. 2009). Kuitenkin myös staattisen analyysin tehokkuus on suhteellista. Analyysimenetelmästä riippumatta käyvät tunnistamiseen käytetyt manuaaliset menetelmät tehottomiksi nykyisen haittaohjelmien määrän kasvuvauhdin edessä (Firdausi et al. 2010) (Gandotra et al. 2014).

2.6 Haittaohjelmien tunnistaminen koneoppimismenetelmin

Saxe ja Berlin (2015) nimesivät kolme trendiä, jotka ovat edistäneet koneoppimismenetelmien käyttöä haittaohjelmien tunnistamisessa. Ensinnäkin haittaohjelmanäytteiden saatavuus on parantunut. Toiseksi laskentatehon kustannukset ovat jo pitkään olleet laskussa. Tämän johdosta yhä suurempia aineistoja pystytään sovittamaan yhä monimutkaisempiin malleihin yhä nopeammin. Kolmanneksi koneoppiminen oppiaineena on kehittynyt niin, että tutkijoilla on käytössään yhä enemmän työkaluja tarkkojen ja skaalautuvien mallien luomiseen.

Koneoppimiseen perustuvien haittaohjelmientunnistajien tarkoituksena on algoritmeja käyttäen analysoida suuria määriä koodinäytteitä ja tunnistaa uudet ennen näkemättömät haitalliset ohjelmat opetusnäytteissä löydettyjen kaavojen perusteella (Schultz et al. 2001) (Krishna et al. 2016). Tarkoituksena on saada aikaan luokittelija, joka pystyy tarkasti luokittelemaan ohjelmakoodinäytteet joko haitallisiksi ja ei-haitallisiksi samalla minimoiden väärien positiivisten luokitteluiden osuuden (Gavrilut et al 2009). Koneoppimismenetelmiä voidaan käyttää sekä staattisessa että dynaamisessa analyysissa. Staattisessa analyysissä tunnistamismallin opettamiseen käytetään jotain ohjelmakoodin staattista esitystapaa (Shabtai et al. 2009). Dynaamiseen analyysissä puolestaan opettamiseen käytetty aineisto on esimerkiksi ohjelmakoodin suorittamisen ja käytöksen monitoroinnin tuloksista tuotettu raportti (Firdausi et al. 2010).

Koneoppimismenetelmät sisältävät aina kaksi päävaihetta: mallin opettamisen ja mallin testaamisen. Mallin opettamiseen käytetään aina lähtökohtaisesti eri dataa kuin sen testaamiseen. Usein käytettävä data myös esikäsitellään siten, että siitä erotetaan ominaispiirteitä, jotka mahdollistavat tehokkaamman luokittelun. Ominaispiirteiden eristämisellä pyritään vähentämään datan ulottuvuuksia ja parantamaan menetelmien toimivuutta. Mallin opettaminen tapahtuu käyttäen valittua menetelmää ja opettamiseen valittua esikäsiteltyä dataa. Mallin luokittelukykyä testataan testivaiheessa siihen valitulla esikäsitellyllä datalla. Testivaiheessa saatuja tuloksia arvioidaan käyttäen useita eri tunnuslukuja. Näitä vaiheita toistetaan, kunnes mallin suorituskykyyn ollaan tyytyväisiä (Shabtai et al. 2009).

Kokonaisuudessaan mallin luontiprosessi sisältää yleensä seuraavat vaiheet: aineiston hankkimisen ja tallentamisen, aineiston esikäsittelyn, mallin muodostamisen jotain soveltuvaa menetelmää käyttäen sekä mallin testaamisen ja testitulosten analysoinnin (Firdausi et al. 2010). Ennen koneoppimismenetelmien soveltamista haitallisen ohjelmakoodin tunnistamiseen on kuitenkin hyvä löytää vastaukset seuraaviin kysymyksiin: millaisessa muodossa käytetyt ohjelmakoodinäytteet esitetään, mitä ominaispiirteitä näistä valitaan ja mitä algoritmia käytetään mallin opettamiseen (Shabtai et al. 2009).

Koneoppimismenetelmiä käytettäessä haittaohjelmien tunnistamiseen eräänä haasteena on, kuinka eristää analysoitavasta koodista luokittelua edistäviä ominaispiirteitä (Sahs & Khan 2012). Kaikki ominaispiirteet eivät välttämättä paranna luokittelijan suorituskykyä. Lisäksi suuri määrä ominaispiirteitä saattaa aiheuttaa haasteita joidenkin koneoppimismenetelmien skaalautuvuudelle (Saxe & Berlin 2015). Ongelmana on, kuinka vähentää ominaispiirteiden määrään siten, että luokittelijan suorituskyky ei kärsi (Moskovitch et al. 2. 2008). Osa ominaispiirteistä saattaa vaikuttaa tarkkuuteen jopa negatiivisesti (Moskovitch et al. 2. 2008). Lisäksi haasteena on, että luokittelijoiden opettamiseen saatavissa olevien aineistojen näytteistä suurin osa on ei-haitallisia ohjelmakoodinäytteitä (Sahs & Khan 2012).

Mallin opettamiseen ja testaamiseen käytettyä aineistoa valittaessa on tärkeää, että mallin opettamiseen käytetty aineisto on tarpeeksi edustava. Esimerkiksi haitallisen ohjelmakoodin tapauksessa mallin opettamiseen voidaan helposti käyttää liian vanhoja koodinäytteitä. Lisäksi on hyvä ottaa huomioon aineiston ei-haitallisten ja haitallisten näytteiden prosentuaaliset osuudet (Shabtai et al. 2009). Haitallisen ohjelmakoodin osuuden voidaan olettaa olevan pieni verrattuna ei-haitalliseen koodiin (Moskovitch et al. 2. 2008). Esimerkiksi Shabtain ja kumppaneiden (2009) esittämän arvion mukaan haitallisten ohjelmien osuus kaikista ohjelmista on noin 10 prosenttia. Moskovitch ja kumppanit (2. 2008) kutsuvat tilannetta epätasapaino-ongelmaksi.

Ohjelmakoodinäytteet voidaan esittää esimerkiksi tavu-n-grammeina, operaatiokoodi-n-grammeina, siirrettävän ja suoritettavan tiedoston (PE-tiedoston) ominaispiirteinä, ohjelmakoodista löydettävinä merkkijonoina sekä tiivistelmänä ohjelmakoodissa käytettävistä funktio- ja API-kutsuista. Tiedoston esittäminen esimerkiksi tavusekvensseinä on hyvin

informatiivista, koska tavusekvenssit ilmaisevat ohjelman konekielenä (Schultz et al. 2001). Ohjelmakoodista on kuitenkin helposti eristettävissä suuria määriä ominaispiirteitä. Usein opettamiseen valitaankin jokin ominaispiirteiden osajoukko, johonka valittavat ominaispiirteet soveltuvat parhaiten kyseessä olevaan luokitteluongelmaan. Aineistoa parhaiten edustavien ominaispiirteiden valintaan on mahdollista käyttää useita erilaisia menetelmiä, joista esimerkkinä ovat Information Gain Ratio- ja Fisher Score-luvut. Myös mallien opettamiseen on tarjolla useita algoritmeja, joista esimerkkeinä yleisesti käytetyt Naive Bayes ja erilaiset päätöspuut (Shabtai et al. 2009).

Malleja käytettäessä on myös hyvä huomioida, että eri mallien tuottamat ennustukset poikkeavat usein toisistaan. Esimerkiksi riippuen opettamiseen käytetystä datasta tai eristetyistä ominaispiirteistä. Jokin malli voi olla toista parempi havaitsemaan tietyn tyyppisen haitallisen ohjelmakoodin ja päinvastoin (Shabtai et al. 2009). Menahem ja kumppaneiden (2009) mukaan eri tyyppisillä luokittelijoilla on tapana tuottaa eri tyyppisiä ”induktiivisia vääristymiä”. Usein onkin järkevää käyttää tunnistamisessa useita eri malleja ja yhdistää tulokset käyttäen jotain menetelmää. Tällä tavoin on mahdollista parantaa ennustamisen tarkkuutta kaiken tyyppiselle haitalliselle ohjelmakoodille (Shabtai et al. 2009). Myös Menahem ja kumppanit (2009) suosittelevat usean haittaohjelmien tunnistamiseen opetetun luokittelijan tuloksien yhdistämistä yhdeksi lopulliseksi luokittelutulokseksi.

Lisäksi kuten jo mainittu, uusia haittaohjelmia luodaan jatkuvasti, joten tunnistamiseen käytettyjä malleja tulisi myös säännöllisesti opettaa uudelleen uudella datalla ja päivitetyt mallit tulisi aina välittää malleja käyttävillä loppukäyttäjille mahdollisimman nopeasti (Schultz et al. 2001). Uutta aineistoa mallien opettamiseen on pystyttävä hankkimaan jatkuvasti, jotta mallit pysyvät ajan tasalla (Shabtai et al. 2009). Suurena haasteena valvotussa koneoppimisessa onkin opetusaineiston näytteiden lokeroiminen, joka on hidasta ja vaatii usein alan asiantuntijoiden työpanosta (Moskovitch et al. 2010). Kaikesta huolimatta malleista ei kuitenkaan koskaan saada täydellisiä ja onkin hyvä myös huomioida, että haitallinen ohjelmakoodi voidaan aina periaatteessa kirjoittaa siten, että jokin malli luokittelee sen väärin (Schultz et al. 2001).

2.7 Haitallisen JavaScript-ohjelmakoodin tunnistaminen

Haitallinen JavaScript-ohjelmakoodi on usein obfuskoitua. Obfuskoinnilla yritetään piilottaa haitalisen ohjelmakoodin aikeet. Obfuskointia käytetään myös ennalta tunnettujen hyökkäyksen piilottamiseen sekä ehkäisemään sääntöihin ja allekirjoituksiin pohjautuvia haittaohjelmientunnistajia tunnistamasta ohjelmakoodia haitalliseksi. Koneoppimismenetelmiin perustuvien menetelmien etuna on, että niillä tuotetut luokittelumallit luokittelevat ohjelmakoodin haitalliseksi, mikäli luokiteltava ohjelmakoodi muistuttaa opetukseen käytettyä haitallista aineistoa enemmän kuin opetukseen käytettyä ei-haitallista aineistoa. Tämä pätee usein myös aiempiin haittaohjelmiin perustuvaan obfuskoituun haitalliseen ohjelmakoodiin (Likarish & Jo 2009).

Haitallisen obsfuskoidun JavaScript-ohjelmakoodin tunnistamiseen käytetään usein koneoppimismenetelmiä, joiden opettamiseen käytetyt ominaispiirteet eristetään ohjelmien eri tasoisista esitystavoista kuten lähdekoodista, sanastollisia merkeistä tai ohjelman abstraktista syntaksipuusta. Koneoppimismenetelmien avulla opetetut mallit eivät välttämättä vangitse haitallisen obfuskoidun ohjelmakoodin tärkeimpiä tunnusmerkkejä (Xu et al. 2013). Kaikki koneoppimiseen perustuvat menetelmät epäonnistuvat joidenkin haittaohjelmien ja hyökkäysten tunnistamisessa, kun koodi ei sisällä niitä ominaispiirteitä, joiden avulla luokittelija on opetettu (Curtsinger et al. 2011).

2.8 Koneoppimismenetelmien soveltamin haittaohjelmien tunnistamiseen

Schultz ja kumppanit (2001) vertailivat signeeraukseen perustuvan tunnistamismenetelmän ja erilaisten koneoppimiseen perustuvien menetelmien kykyä tunnistaa ennen näkemätöntä haitallista ohjelmakoodia. Tutkijoiden käyttämät koneoppimismenetelmät olivat RIPPER, Naive Bayes ja Multi-Naive Bayes. Tutkimuksessa käytetyt koodinäytteet hankittiin useasta eri lähteestä. Koodinäytteet luokiteltiin haitallisiksi ja ei-haittalisiksi käyttäen kaupallisista antivirusohjelmaa. Koodinäytteet olivat tunnettuja ohjelmia, jotka kaupallisten antivirusohjelmien oletettiin tunnistavan.

Jokaista käytettyä menetelmää varten ohjelmakoodinäytteistä eristettiin erilaisia ominaispiirteitä. Signeeraukseen perustuvan menetelmän kohdalla käytetyt ominaispiireet olivat tavusekvenssejä. RIPPER-menetelmän kohdalla ominaispiirteinä toimivat ohjelmassa käytetyt dll-kirjastot, käytetyt dll-funktiokutsut ja käytettyjen dll-kirjastojen määrä yhdessä ohjelman dll-funktiokutsujen määrän kanssa. Naive Bayes-menetelmän kohdalla ominaispiirteinä käytettiin ohjelmakoodissa esiintyviä merkkijonoja. Kun taas Multi-Naive Bayes-menetelmän kohdalla ominaispiirteinä käytettiin jälleen tavusekvenssejä (Schultz et al. 2001).

Menetelmien suorituskyvyn arviointiin ja keskinäiseen vertailuun käytettyjen tulosten tuottamiseen tutkijat käyttivät 5-fold-menetelmää. Tuloksissa on huomattavissa, että kaikki tutkimuksessa käytetyt koneoppimiseen perustuvat menetelmät suoriutuivat käytettyä signeeraukseen perustuvaan menetelmään paremmin ennen näkemättömän haitallisen ohjelmakoodin tunnistamisessa. Korkein 97.11 prosentin tarkkuus saavutettiin käyttämällä Multi-Naive Bayes-menetelmää ja toiseksi korkein 96.88 prosentin tarkkuus saatiin Naive Bayes-menetelmällä. Molemmissa tapauksissa tarkkuus oli kaksinkertainen verrattuna käytettyyn signeeraukseen perustuvaan menetelmään.

Kolter ja Maloof (2004) puolestaan tutkivat n-grammien käyttöä luokittelijoiden opettamiseen käytettyinä ominaispiirteinä. Tutkijat keskittyivät erityisesti siihen mikä määrä n-grammeja ominaispiirteinä tuottaa parhaat tulokset luokittelijoiden opettamisessa. Tutkimuksessa käytettiin yhteensä 1651 haitallista ohjelmakoodinäytettä ja 1971 ei-haitallista ohjelmakoodinäytettä. Tutkimuksessa käytetyt koneoppimismenetelmät olivat tukivektorikone, tehostettu tukivektorikone, Naive Bayes, J48-päätöspuu ja tehostettu J48-päätöspuu. Kolter ja Maloof eristivät analysoitavista näytteistä n-grammeja, joista he sitten valitsivat 10, 20, . . ., 100, 200, . . ., 1000, 2000, . . ., 10 000 edustavinta n-grammia luokittelijoiden opettamiseen. Yleisesti ottaen parhaat tulokset saavutettiin valitsemalla ominaispiirteiksi 500 n-grammia. Luokittelijoista parhaat tulokset saavutettiin tehostetulla J48-päätöspuulla. Menetelmän tapauksessa sen suoriutumista kuvaavan ROC-käyrän alle jäävä pinta-ala oli 0.996.

Moskovitch ja kumppanit (1. 2008) tutkivat kronologisen opetuksen ja arvioinnin vaikutusta koneoppimisen menetelmien kykyyn tunnistaa haitallista ohjelmakoodia. Tutkijat opettivat luokittelijat aineistolla, joka sisälsi vuoteen ”k” asti kerättyjä näytteitä. Tämän jälkeen tutkijat arvioivat luokittelijoiden suoriutumista aineistolla, joka sisälsi vuonna ”k+1” kerättyjä näytteitä. Tutkivat suorittavat saman kahdella eri opetusaineistolla. Ensimmäisessä aineistossa haitallisen koodin osuus oli 50 prosenttia ja toisessa opetusaineistossa 16 prosenttia. Tutkimuksessa käytetty aineisto koostui haitallisista ja ei-haitallisista Windows EXE-tiedostoista. Aineisto sisälsi yhteensä 7688 haitallista ja 22 735 ei haitallista tiedostoa, jotka ajoittuivat vuosien 2000 ja 2007 välille.

Ominaispiirteinä tutkijat käyttivät ohjelmakoodista muodostettuja eri mittaisia n-grammeja. Ominaispiirteiden määrien rajoittamiseen tutkijat käyttivät eri menetelmiä. Käytettyinä koneoppimismenetelminä oli muiden muassa päätöspuu. Tutkijat huomasivat, että luokittelijan luokittelukyky paranee, kun ajassa mennään eteenpäin. Erityisesti kun haitallisen koodin osuus opetusaineistossa oli 16 prosenttia. Tutkijat huomasivat myös, että haitallista koodia 16 prosenttia sisältävällä aineistolla opetetut luokittelijat pärjäsivät yleisesti 50 prosenttia haitallista koodia sisältävillä aineistoilla opetettuja luokittelijoita paremmin (Moskvitch et al. 1. 2008).

Moskovitch ja kumppanit (2. 2008) tutkivat myös käytettyjen ominaispiirteiden ja opetusaineistossa olevien haitallisten näytteiden määrän vaikutusta luokittelijoiden suorituskykyyn. Tutkimuksessa käytetty aineisto koostui haitallisista ja ei-haitallisista Windows EXE-tiedostoista. Aineisto sisälsi yhteensä 7688 haitallista ja 22 735 ei haitallista tiedostoa. Tutkijat valitsivat ohjelmakooditiedoissa aina 5 500 niissä useimmin esiintyvää n-grammia. Tämän jälkeen tutkijat valitsivat aikaisemmin valituista n-grammeista 50, 100, 200, ja 300 n-grammia käyttäen seuraavia menetelmiä: dokumenttifrekvenssiä, Infromation Gain Ratio-lukua ja Fisher Score-lukua. Näitä ominaispiirteitä käyttäen tutkivat opettivat luokittelumalleja käyttäen neuroverkkoa, päätöspuita, Naive Bayes-menetelmää ja tukivektorikoneita.

Moskovitch ja kumppanit (2. 2008) huomasivat, että Fisher Score-luvun avulla valitut ominaispiirteet tuottivat parhaat tulokset kaikilla ominaispiirremäärillä. Yleisesti 300 n-

grammia tuotti parhaat tulokset. Käytetyistä menetelmistä puolestaan yleisesti parhaiten pärjäsivät neuroverkko ja päätöspuut saavuttaen 94 prosentin ylittävät tarkkuudet säilyttäen samalla alhaisen asteen väärissä positiivisissa luokitteluissa. Lisäksi tutkijat huomasivat, että paras tarkkuus saavutettiin opetusaineistoissa, jossa haitallisten näytteiden osuus oli 16.7 prosenttia.

Moskovitch ja kumppanit (3. 2008) tutkivat myös ohjelmankoodin operaatiokoodista eristettyjen ominaispiirteiden käyttämistä haittaohjelmien tunnistamiseen tarkoitettujen koneoppimismallien opettamisessa. Tutkijat perustelivat operaatiokoodin käyttämistä tunnistamisessa sillä, että iso osa haittaohjelmista kuuluu johonkin haittaohjelmaperheeseen ja nämä perheet ovat toiminnallisuuksiltaan samanlaisia, jolloin ne ovat myös operaatiokoodin osalta samankaltaisia. Tutkijat muodostivat käytettyjen näytteiden operaatiokoodista eri mittaisia n-grammeja. Ominaispiirteiden määrien rajoittamiseen tutkijat käyttivät dokumenttifrekvenssiä, Information Gain Ratio-lukua ja Fisher Score-lukua. Kokonaisuutena käytetty aineisto koostui yhteensä 5 677 haitallisesta ja 20 416 ei-haitallisesta tiedostosta. Tutkijat muodostivat luokittelijoiden opettamiseen käyttämänsä aineistot siten, että haittaohjelmien osuus vaihteli opetusaineistojen välillä. Käytetyt osuudet olivat: 5, 10, 15, 30 ja 50 prosenttia. Käytettyinä koneoppimismenetelminä olivat neuroverkot, päätöspuut, tehostetut päätöspuut, Naive Bayes (Moskovitch et a.l 3. 2008).

Yleisesti parhaat tulokset saavutettiin 2-grammeilla, joilla opetetut luokittelijat saavuttivat noin 87 prosentin tarkkuuden. Ominaispiirteiden määrään rajaamiseen käytetyistä menetelmistä yleisesti parhaiten pärjäsivät dokumenttifrekvenssin sekä Fisher Score-luku. Luokittelijoiden keskinäiseen vertailuun tutkijat käyttivät 2-grammeja, joista valittiin 300 edustavinta dokumenttifrekvenssiä käyttäen. Parhaiten luokittelijoista suoriutuivat päätöspuut ja tehostetut päätöspuut. Päätöspuiden avulla saavutettiin keskimäärin 93 prosentin tarkkuus ja 4 prosentin aste väärissä positiivisissa luokitteluissa. Kun taas tehostettujen päätöspuiden avulla saavutettiin keskimäärin 94.43 prosentin tarkkuus ja 3 prosentin aste väärissä luokitteluissa. Molemmilla menetelmien suoriutuminen luokittelusta oli huipussaan, kun haitallisen ohjelmakoodin osuus opetusaineistossa oli 10 ja 15 prosenttia (Moskovitch et a.l 3. 2008).

Ahmed ja kumppanit (2009) taas tutkivat spatiaalisten ja temporaalisten ominaispiirteiden käyttämistä haittaohjelmien tunnistamiseen tarkoitettujen luokittelijoiden opettamisessa. Spatiaaliset ominaispiirteet ovat yleensä API-kutsujen parametrien tilastollisia ominaisuuksia. Kun taas temporaaliset ominaispiirteet ovat API-kutsujen ajallisia kuvauksia. Tutkivat suorittivat yhteensä 416 haitallista ja ei-haitallista ohjelmaa Windows-käyttöjärjestelmässä ja muodostivat ohjelmien suorituksesta lokitiedostoja, jotka sisälsivät spatiaaliset ja temporaaliset ominaispiirteet yhteensä 237:stä käyttöjärjestelmän ydin ohjelmointirajapinnasta.

Tutkijoiden käyttämät koneoppimismenetelmät olivat k-lähin naapuri, J48-päätöspuu, Naive Bayes, RIPPER ja Sequential Minimal Optimization. Tutkijat onnistuivat saavuttamaan keskimäärin 89.8 prosentin tarkkuuden tunnistamisessa käyttäen spatiaalisia ominaispiirteitä ja keskimäärin 95.2 prosentin tarkkuuden käyttäen temporaalisia ominaispiirteitä. Kun tutkijat käyttivät mallien opettamiseen spatiaalisten ja temporaalisten ominaispiirteiden yhdistelmää tunnistamisen tarkkuuden keskiarvo nousi 96.3 prosenttiin (Ahmed et al. 2009).

Tabish ja kumppanit (2009) puolestaan kehittävät menetelmän ennen näkemättömien haittaohjelmien tunnistamiseen ja luokitteluun. Menetelmän koneoppimismenetelmänä toimi J48-päätöspuu, jonka toimintaa tehostettiin AdaBoostM1-algoritmilla. Mallien opettamiseen käytettiin ohjelmakoodin staattisesta tavutason kuvauksesta eritettäviä, tilastotieteeseen ja informaatioteoriaan perustuvia, ominaispiirteitä. Tutkijoiden käyttämä ei-haitallisten tiedostoja edustava aineisto piti sisällään tiedostoja kuudesta yleisesti käytössä olevasta tiedostotyypistä: DOC, EXE, JPG, MP3, PDF ja ZIP. Jokaisesta tiedostotyypistä mukana oli 300 tiedostoa, jolloin tiedostoja oli yhteensä 1 800. Haittaohjelmia edustavan aineiston tutkijat latasivat haittaohjelmatietokannasta. Saatavilla olevista näytteistä tutkijat valitsivat 10 311 32-bittiselle Windows-käyttöjärjestelmälle tarkoitettua haittaohjelmaa. Tutkijat jakoivat haittaohjelmanäytteet kuuteen luokkaan niiden ominaisuuksien perusteella.

Tutkijoiden kehittämä menetelmä voidaan arkkitehtuurinsa puolesta jakaa neljään moduuliin: tiedostot lohkoihin jakavaan moduuliin, ominaispiirteet tuottavaan moduuliin, kone-

oppimisesta vastaavaan moduuliin ja tulokset yhdistävään korrelaatiomoduuliin. Lohkoja tuottava moduuli jakaa tiedostot tavutasolla kiinteän kokoisiin lohkoihin. Ominaispiirteitä tuottava moduuli puolestaan tuottaa lohkoista 13 erilaista tilastotieteeseen ja informaatioteoriaan perustuvia ominaispiirrettä. Nämä ominaispiirteet laskettiin kaikille yhdestä neljään pituudeltaan olevalla n-grammille, joten ominaispiirteitä kertyi yhteensä 52 per lohko. Luokittelumenetelmän avulla jokainen tiedostosta muodostettu lohko luokiteltiin ei-haitalliseksi tai potentiaalisesti haitalliseksi. Korrelaatiomoduuli puolestaan tuotti lohkojen luokitteluiden avulla tiedoston lopullisen luokittelun. Tutkijat opettivat erillisen luokittelijat jokaista kuutta valitsemaansa haittaohjelmaluokkaa varten. Tutkijat saavuttivat menetelmällään 90 prosentin tunnistamisasteen kaikille haittaohjelmatyypeille (Tabish et al. 2009).

Santos ja kumppanit (2011) puolestaan tutkivat haittaohjelmien tunnistamista käyttäen operaatiokoodisekvenssejä ja yhden luokan tukivektorikonetta. Tutkijat käyttivät Information Gain Ratio-lukua ominaispiirteiden määrän rajaamiseen 1000 ominaispiirteeseen. Tutkijat testasivat luokittelijan suoriutumista ensin aineistolla, joka sisälsi vain näytteitä haitallisesta ohjelmakoodista. Tällöin onnistuttiin saavuttamaan taattu 83.42 prosentin tarkkuus, kun opetusaineisto muodostettiin käyttäen ainakin 600 haittaohjelmakoodinäytettä. Tämän jälkeen tutkijat testasivat luokittelijan suoriutumista, kun opetusaineisto muodostettiin pelkästään ei-haitallista koodia edustavista näytteistä. Tällöin tutkijat onnistuivat saavuttamaan 84.22 prosentin tarkkuuden, kun luokittelijan opettamiseen käytetty aineisto sisälsi 400 ohjelmakoodinäytettä. Luokittelun tarkkuus jatkoi kasvuaan aina pisteeseen, jolloin opetusaineisto muodostettiin käyttäen 600 ohjelmakoodinäytettä. Tällöin saavutettu tarkkuus oli 87.46 prosenttia. Tämän jälkeen tarkkuus kääntyi lievään laskuun, kun käytettyjen näytteiden määrää kasvatettiin.

Myös Narayanan ja kumppanit (2013) tutkivat koneoppimismenetelmin opetettujen luokittelijoiden suoriutumista haittaohjelmien tunnistamisesta. Tutkimuksessa menetelmien opettamiseen käytetyt ominaispiirteet olivat kuitenkin ohjelmakoodista eristetyistä heksadesimaaliallekirjoituksista muunnetut aminohappojäännösten kuvauksessa käytetyt kirjainsekvenssit ja sekvenssianalyysissä käytetyn sekvenssienlinjausmenetelmällä saadut kirjainsekvenssien linjatut versiot. Sekvenssianalyysi on esimerkiksi biologiassa käytetty mene-

telmä, jolla pyritään ymmärtämään kahden tai useamman sekvenssin välisiä suhteita. Sekvenssit ovat yleensä geneettistä informaatioita kuvaavia sekvenssejä kuten DNA. Menetelmää voidaan kuitenkin käyttää myös muihin sovelluskohteisiin kuin biologiaan.

Narayanan ja kumppaneiden (2013) tutkimuksessa käyttämät koneoppimismenetelmät olivat J48-päätöspuu, Naive Bayes ja neuroverkko. Tutkimuksessa käytettiin 60 tietokonemadon ja 60 tietokoneviruksen ohjelmakoodista eristettyjä heksadesimaaliallekirjoituksia. Tutkijat muunsivat heksadesimaaliallekirjoitukset aminohappojäännösten kuvauksessa käytetyiksi erilaisiksi kirjainsekvensseiksi ja sovelsivat sekvensseihin sitten sekvenssianalyysissä käytettyä sekvenssienlinjausmenetelmää. Tutkijat huomasivat, että käyttämällä sekvenssilinjausta virusten ja matojen luokittelun tarkkuus parani kaikilla kolmella koneoppimismenetelmällä.

Amalina ja kumppanit (2014) taas tutkivat Android-käyttöjärjestelmille suunnattujen haittaohjelmien tunnistamista koneoppimismenetelmin, eristäen luokittelijoiden opettamiseen käytetyt ominaispiirteet ohjelmien suoritusaikana verkkoresurssien käytöstä kerätystä informaatiosta. Käytettyinä koneoppimismenetelminä olivat J48-päätöspuu, Bayes-verkko, neuroverkko, k-lähin naapuri ja Random Forrest. Parhaiten tutkimuksessa käytetyistä menetelmistä menestyi J48-päätöspuu, joka saavutti keskimäärin 99.99 prosentin tarkkuuden ja 0.10 prosentin asteen väärissä positiivisissa luokitteluissa. Huonoiten menetelmistä tutkimuksissa menestyi neuroverkko alle 95 prosentin tarkkuudellaan ja yli 5 prosentin asteella väärissä positiivisissa luokitteluissa.

Le ja kumppanit (2014) puolestaan tutkivat haittaohjelmien tunnistamista geneettistä ohjelmointia käyttäen. Verrokkimenetelminä tutkijat käyttivät sellaisia koneoppimismenetelmiä kuten Naive Bayes, J48-päätöspuu ja tukivektorikone. Tutkimuksessa käytettiin vain staattisia ominaispiirteitä. Analysoitavat tiedostot käännettiin aluksi heksadesimaaliesitysmuotoon, jonka jälkeen niistä muodostettiin 4-grammeja. Näistä varsinaisiksi ominaispiirteiksi mukaan otettiin 50, 100 ja 200 4-grammia käyttäen termifrekvenssiä, dokumenttifrekvenssiä ja Fisher Score-lukua.

Le ja kumppanit (2014) tekivät tutkimuksessa kaksi koetta. Ensimmäinen koe suoritettiin aineistolla, jossa oli mukana 750 haitallista ohjelmakooditiedosta ja 750 ei-haitallista tie-

dosta. Toinen koe suoritettiin aineistolla, jossa oli 800 haitallista tiedostoa ja 1600 ei-haitallista tiedostoa. Molemmissa kokeissa aineistot jaettiin kahtia mallien opettamista ja testaamista varten. Useissa tehdyissä kokeissa geneettinen algoritmi oli parhaiten luokittelusta suoriutuva menetelmä, kun arviointi kriteerinä oli oikein luokiteltujen tiedostojen prosentuaalinen osuus kaikista luokitelluista tiedostoista. Tasapainoisilla aineistoilla parhaiten pärjäsivät dokumenttifrekvenssillä valituilla ominaispiirteillä opetetut mallit riippumatta menetelmästä. Tällöin geneettisellä ohjelmoinnilla ja 200 ominaispiirteellä tuotettu malli pystyi luokittelemaan oikein 93.7 prosenttia tiedostoista. Kun taas epätasapainoisilla aineistoilla tulokset jakaantuivat edellistä enemmän.

Saxe ja Berlin (2015) tutkivat haittaohjelmien tunnistamista käyttäen menetelmänä syvää neuroverkkoa, jonka opettamiseen käytetyt ominaispiirteet johdettiin haitallisten ja ei-haitallisten ohjelmien staattisista binaariesityksistä. Tutkijoiden käyttämä neuroverkko koostui syötekerroksesta, kahdesta piilokerroksesta ja tulostekerroksesta. Verkon opettamiseen tutkijat käyttivät Backpropagation-algoritmia ja gradienttipohjaista Adam-optimointimenetelmää. Tutkijat käyttivät neljän tyyppisiä ominaispiirteitä neuroverkon opettamiseen. Ensimmäisen tyypin ominaispiirteet perustuivat syötetiedostoille laskettuihin tavuentropiahistogrammeihin, jotka mallintavat tiedostojen tavujen jakaumaa. Toisen tyypin ominaispiirteet johdettiin ohjelmaan liitetyistä kirjastoista. Kolmannen tyypin ominaispiirteet johdettiin analysoitavan tiedoston PE-paketoinnista. Neljännen tyypin ominaispiirteet olivat yhdistelmiä kolmesta edellä mainitusta ominaispiirretyypistä.

Saxe ja Berlin (2015) arvioivat luokittelijan toimivuutta kahdella eri aineistolla. Kokonaisuutena tutkijoilla oli käytettävissä aineisto, joka sisälsi 350 016 haitalliseksi luokiteltua tiedostoa ja 81 910 ei-haitalliseksi luokiteltua tiedostoa. Tutkijat käyttivät 4-fold menetelmää neuroverkon opettamiseen ja tulosten validointiin. Näin tutkijat onnistuivat saavuttamaan menetelmällään 95.2 prosentin keskiarvon todellisissa positiivisissa luokitteluissa, kun sallittujen väärien positiivisten luokitteluiden osuus oli asetettu 0.1 prosenttiin.

Hansen ja kumppanit (2016) puolestaan tutkivat uusien ennen näkemättömien haittaohjelmien havaitsemista sekä luokittelua haittaohjelmaperheisiin käyttäen dynaamista analyysia ja koneoppimismenetelmiä. Tutkijat käyttivät haittaohjelmien tunnistamiseen ja luokitte-

luun Random Forest-menetelmää. Tutkijat kuvasivat haittaohjelmat ominaispiirteinä, jotka perustuivat ohjelmien tekemiin API-kutsuihin, API-kutsujen argumentteihin ja ohjelmien käyttäytymistä kuvaaviin allekirjoituksiin. Mallien opettamiseen käytetyt ominaispiireet valittiin käytten Information Gain Ratio-lukua. Tutkimuksessa käytetty aineisto sisälsi kokonaisuudessaan 837 haitallista ja 270 000 ei-haitallista ohjelmaa. Aineistosta 67 prosenttia käytettiin mallin opettamiseen ja loput 33 prosenttia käytettiin mallin testaamiseen. Haittaohjelmien tunnistamisessa tutkijat saavuttivat keskimääriin 98.1 prosentin asteen todellisissa positiivisissa havainnoissa ja 9.9 prosentin asteen väärissä positiivisissa luokitteluissa. Kun taas vastaavat luvut haittaohjelmien luokittelulle olivat 86.0 prosenttia ja 0.35 prosenttia.

Tobiyama ja kumppanit (2016) puolestaan kehittivät ohjelman prosessikäyttäytymiseen ja neuroverkkoihin pohjautuvan haittaohjelmien tunnistamismenetelmän. Menetelmä käyttää toistuvaa neuroverkkoa ominaispiirteiden eristämiseen. Menetelmä muodostaa näistä ominaispiirteistä kuvavektorin, joka puolestaan toimii syötteenä luokittelun toteuttavalle konvoluutioneuroverkolle. Tutkijat käyttivät 81 haitallisen ohjelman prosessilokitiedostoa ja 69 ei-haitallisen ohjelman prosessilokitiedostoa mallien opettamiseen ja validointiin. Haitallisen ohjelman prosessilokitiedostot hankittiin käyttämällä 26 haittaohjelmaa. Tutkijat saavuttivat menetelmällään parhaimmillaan AUC-luvun 0.96, kun opettamiseen ja testaamiseen käytettiin 5-fold menetelmää 150:llä kokoa 30 x 30 olevalla ominaispiirrekuvalla.

Damodaras ja kumppanit (2017) puolestaan tutkivat haittaohjelmien tunnistamista käyttäen Markovin piilomalleja. Tutkijat valitsivat opetusaineistoksi tiedostoja useisiin eri kategorioihin kuuluvista haittaohjelmista. Ei-haitallisia tiedostoja edustamaan valittiin Windowsin systeemitiedostoja. Valittuihin tiedostoihin sovellettiin dynaamisista ja staattista analyysia, joilla tiedostoista eristettiin API-kutsusekvenssejä sekä operaatiokoodisekvenssejä. Aluksi tutkijat opettivat ja testasivat ensin neljä mallia käyttäen API-kutsusekvenssejä, joiden suoriutumista he vertailivat keskenään. Tämän jälkeen tutkijat tekivät saman käyttäen eristämiään operaatiokoodisekvenssejä.

Ensimmäiset neljä mallia pyrkivät tunnistamaan haitallisen ohjelmakoodin käyttäen API-kutsusekvenssejä. Näistä ensimmäinen malli perustui dynaamisella analyysillä eristettyihin

sekvensseihin. Toinen malli perustui staattisella analyysillä eristettyihin sekvensseihin. Kolmas malli puolestaan opetettiin käyttäen dynaamisella analyysillä eristettyjä sekvenssejä ja mallin arviointiin staattisella analyysillä eristettyjä sekvenssejä. Neljäs malli puolestaan oli päinvastainen kolmanteen malliin nähden. Parhaiten haittaohjelmien tunnistamisesta suoriutui dynaamisella analyysillä eristetyillä sekvensseillä harjoitettu ja testattu malli (ensimmäinen), tämän jälkeen staattisella analyysillä eristetyillä sekvensseillä harjoitettu ja testattu malli (toinen). Yllättävää kyllä tunnistamisesta huonoiten suoriutuivat hybridimenetelmät (Damodaras et al. 2017).

Seuraavien neljän mallin kohdalla tukijat keskittyivät haittaohjelmien tunnistamiseen käyttäen operaatiokoodisekvenssejä. Mallien opettaminen ja testaaminen tapahtuivat samaan tapaan kuin API-kutsu-sekvenssien tapauksessa. Myös saadut tulokset olivat saman suuntaisia, kun API-kutsu-sekvenssien kohdalla. Tälläkin kertaa haittaohjelmien tunnistamisesta parhaiten suoriutui dynaamisella analyysillä eristetyillä sekvensseillä harjoitettu ja testattu malli, tämän jälkeen staattisella analyysillä eristetyillä sekvensseillä harjoitettu ja testattu malli. Jälleen huonoiten suoriutuivat jälleen hybridimenetelmät.

2.9 Haitallisen JavaScript-ohjelmakoodin tunnistaminen

Likarish ja Jo (2009) tutkivat haitallisen web-selaimissa suoritettavan JavaScript-koodin tunnistamista koneoppimismenetelmiä käyttäen. Tutkijat keräsivät käyttämänsä ei-haitallista JavaScript-koodia edustavan aineistonsa tunnetuilta web-sivuilta. Aineisto sisälsi kaiken kaikkiaan yhteensä 63 miljoonaa näytettä, joista tutkivat valitsivat varsinaisessa opetusaineistossa käytetyt näytteet manuaalisesti. Haitallista JavaScript-koodia edustavaa aineistoa varten tutkijat eristivät ensin koodinäytteitä ”mustille listoilla” päätyneiltä web-sivuilta, joista tutkijat sitten valikoivat opetusaineistonsa manuaalisesti. Käytetty opetusaineisto sisälsi lopulta 62 haitallista näytettä ja 50 000 ei-haitallista näytettä.

Tutkijat eristivät valituista tiedostoista yhteensä 65 ominaispiirrettä, joista 50 oli JavaScriptin avainsanoja ja symboleita. Nämä ominaispiirteet toimivat varsinaisina syötteinä luokittelumalleja opetettaessa. Tutkimuksessa käytetyt koneoppimismenetelmät olivat: Naive Bayes, ADTree, tukivektorikone ja Ripper. Mallit muodostettiin käyttäen 10-fold-

menetelmää. Kaikki luokittelijat suoriutuivat tällä aineistolla hyvin. Haitallinen ohjelmakoodi luokiteltiin haitalliseksi keskimäärin noin 90 prosentissa tapauksista ja ei-haitallinen ohjelmakoodi luokiteltiin ei-haitalliseksi keskimäärin noin 99.7 prosentissa tapauksissa. Tutkivat testasivat luokittelijoiden suorituskykyä vielä toisella erillisellä aineistolla. Tällöinkin luokittelijoiden suorituskyky oli hyvä, lukuun ottamatta Ripper-menetelmällä tuotettua luokittelijaa. Tutkijat nimesivät myös viisi ominaispiirrettä, jotka korreloivat eniten haitallisen selaimessa suoritettavan JavaScript-koodin kanssa. Nämä ominaispiirteet olivat: luettavuus ihmisen toimesta, ”eval”-avainsanan käyttökerrat, tyhjien välien osuus koko koodista, merkkijonojen keskimääräinen pituus ja rivillä olevien merkkien keskimääräinen määrä (Likarish & Jo 2009).

Cova ja kumppanit (2010) puolestaan kehittivät menetelmän nimeltä Jsand, joka tarkoituksena on koneoppimismenetelmää käyttäen oppia ei-haitallisen JavaScript-koodin ominaispiirteet. Menetelmä perustuu koodin suorittamisen emulointiin ja anomalioiden tunnistamiseen. Menetelmässä käytetty koneoppimismenetelmä on Naive Bayes. Valitsemiensa ominaispiirteiden avulla tutkijat pyrkivät löytämään analysoitavasta koodista osia, jotka viittaavat esimerkiksi selaimen uudelleen ohjaukseen jollekin toiselle sivulle, obfuskoinnin purkamiseen, ympäristön valmistamiseen hyökkäystä varten ja varsinaisiin hyökkäyksiin. Menetelmän opettamiseen ja testaamiseen käytetty haitallista koodia edustava aineisto sisälsi 823 tiedostoa ja ei-haitallista koodia edustava aineisto hankittiin käyttäen 11 215 ei-haitallista URL-osoitetta. Tutkijat saavuttivat menetelmää käyttäen 0.25 prosentin asteen väärissä negatiivisissa luokitteluissa.

Haitallisen ohjelmakoodin tunnistamisen lisäksi Cova ja kumppanit (2010) analysoivat menetelmäänsä käyttäen mitkä haitallisen koodin ominaispiirteet vaikuttivat eniten menetelmän käyttämään anomalia-arvioon. Tuloksena oli, että suoraan hyökkäyksiin liittyvät ominaispiirteet olivat ylivoimaisesti suurimalta osin (88 prosenttisesti) vastuussa anomalia-arvoista. Lisäksi ohjelmakoodi, josta oli erotettavissa useisiin hyökkäyksiin liittyviä ominaispiirteitä, saivat anomalia-arvot suorastaan räjähtämään. Koodin obfuskaatioon liittyvät ominaispiirteet olivat vastuussa vain 2.7 prosentista lopullisesta anomalia-arvosta.

Myös Curtsinger ja kumppanit (2011) kehittivät Naive Bayes-menetelmään pohjaavan tunnistamismenetelmän nimeltä Zozzle. Menetelmän luokittelija saa syötteinä JavaScript-koodin abstraktista syntaksipuusta johdettuja ominaispiirteitä. Menetelmä on analyysin osalta staattinen, mutta menetelmällä on myös dynaaminen komponentti obfuskoinnin varalta. Komponentti kerää ja prosessoi suorituksen aikana luotua JavaScript-koodia. Zozzle kykenee havaitsemaan haitallisen JavaScript-koodin tehokkaasti ja tutkijat kykenivätkin saavuttamaan menetelmällä 0.00003 prosentin asteen väärissä positiivisissa luokitteluissa.

Ravi ja kumppanit (2014) puolestaan kehittävät web-selaimiin JSGuard nimisen lisäosan, avulla pyritään tunnistamaan haitalliset web-sivut. Lisäosa käyttää sivujen analysointiin sekä staattista että dynaamista analyysiä. Lisäosan staattinen analyysi etsii haavoittuvuuksia analysoitava sivun lähdekoodista. Dynaaminen analyysi puolestaan tutkii muutoksia sivun DOM:in rakenteessa ja dynaamisen JavaScript-koodin käyttöä. JSGuard tekee päätöksen sivun haitallisuudesta vertaamalla tekemiään huomioita ennalta määriteltyihin sääntöihin.

Tutkijat määrittelivät edellä mainitut säännöt analysoimalla useiden JavaScript-hyökkäysten kaavoja. Obfuskoidun koodin havaitsemiseen lisäosa käyttää erilaisia tunnuslukuja, jotka lasketaan analysoitavan sivun JavaScript-koodista. Näitä lukuja ovat esimerkiksi valkoisen tilan osuus merkkijonoissa, avainsanojen osuus kaikista sanoista, merkkijonojen pituus, ja ei-printattavien ASCII-merkkien osuus merkkijonoissa. Lisäosa suoriutui kohtalaisen hyvin sivujen luokittelusta, luokitellen vain 0.72 prosenttia ei-haitallisista sivuista haitallisiksi ja 9.5 prosenttia haitallisista sivuista ei-haitallisiksi. Menetelmä ei myöskään aiheuttanut merkittäviä lisäkustannuksia web-sivujen lataukseen, kasvattaen sivujen lataamiseen käytettyä aikaa keskimäärin vain 180 millisekuntia (Ravi et al. 2014).

Wang ja kumppanit (2015) taas puolestaan kehittävät menetelmän nimeltä Jsdc. Menetelmä käyttää koneoppimismenetelmiä haitallisen JavaScrip-ohjelmakoodin havaitsemiseen ja luokitteluun 8 eri luokkaan perustuen haitallisessa koodissa käytettyyn hyökkäysvektoriin. Ohjelmakoodin luokitteluun haitalliseksi ja ei-haitalliseksi käytetyt koneoppimismenetelmät olivat ADTree, Random Forest, J48-päätöspuu ja Naive Bayes. Havaittujen haittaohjelmien luokitteluun hyökkäysvektoriin perustuviin luokkiin käytetyt koneoppimismene-

telmät olivat puolestaan Random Tree, Random Forest, J48-päätöspuu ja Naive Bayes. Tutkijat johtivat menetelmien yhteydessä käytetyt ominaispiirteet tekstimuodossa esitetystä lähdekoodista, analysoitavan ohjelman rakenteesta ja käytetyistä epäilyttäväksi luokitelluista funktiokutsuista.

Ominaispiirteinä toimivat esimerkiksi sanojen pituus, eri pituiset n-grammit, merkkien frekvenssit, kommentointityyli, entropia, käytetyt HTML-elementit, käytetyt DOM-operaatiot, käytetyt API-kutsukaavat ja ohjelman abstraktista syntaksipuusta johdetut ominaispiirteet. Lisäksi tutkijat määrittelivät ylimääräisiä ominaispiirteitä haittaohjelmien hyökkäysvektoriin perustuvaa luokittelua varten. Tutkijoiden käyttämä opetusaineisto sisälsi yhteensä 942 eri tavoin hankittua haitallista koodinäytettä ja 20 000 suosituilta websivuilta hankittua ei-haitallista näytettä. Mallien opettamiseen tutkijat käyttivät 10-fold-menetelmää. Parhaat tulokset tutkijat saivat käyttäen luokitteluun Random Forest-menetelmään. Menetelmää käyttäen saavutettiin 99.532 prosentin tarkkuus, 0.212 prosentin aste väärissä positiivisissa luokitteluissa ja 0.829 prosentin aste väärissä negatiivisissa luokitteluissa. Menetelmistä huonoiten luokittelusta selviytyi Naive Bayes-menetelmä. Menetelmän avulla saavutettiin kuitenkin niinkin korkea tarkkuus kuin 98.224 prosenttia. Naive Bayes-menetelmällä saavutettiin 1.127 prosentin aste väärissä positiivisissa luokitteluissa ja 4.528 aste väärissä negatiivisissa luokitteluissa (Wang et al. 2015).

Myöskin Patil ja Patil (2017) tutkivat haitallisen JavaScript-ohjelmakoodin tunnistamista koneoppimismenetelmiä käyttäen. Tutkimuksessa käytettyjä koneoppimismenetelmiä olivat Naive Bayes, J48-päätöspuu, Random Forrest, tukivektorikone, AdaBoost, REP-puu ja AD-puu. Tutkimuksessa käytetty aineisto puolestaan koostui 2225 haitallisesta ohjelmakoodinäytteestä ja 4500 ei-haitallisesta näytteestä. Luokittelijoiden opettamiseen tutkijat käyttivät yhteensä 77 ominaispiirrettä, joista 32 oli vanhoja aikaisemmissa tutkimuksissa käytettyjä ominaispiirteitä ja 45 uusia tutkijoiden itse määrittämiä ominaispiirteitä. Ominaispiirteinä toimivat esimerkiksi eval-funktiokutsujen määrä, setTimeout-funktiokutsujen määrä, search-funktiokutsujen määrä ja split-funktiokutsujen määrä. Luokittelijat opetettiin aineistoilla, jotka sisälsivät 50 prosenttia haitallisia näytteitä ja 50 prosenttia ei-haitallisia näytteitä.

Yleisesti ottaen kaikki Patilin ja Patilin (2017) käyttämät luokittelijat pystyivät tunnistamaan haitalliset näytteet tarkasti. Pelkästään kirjallisuudesta ennestään tunnetuilla ominaispiirteillä opetetut luokittelijat saavuttivat 95.5-99.9 prosentin tarkkuuden, 0.063-0.001 prosentin asteen virheellisissä positiivisissa luokitteluissa ja 0.009-0.0001 prosentin asteen virheellisissä negatiivisissa luokitteluissa. Parhaiten ennestään tunnetuilla ominaispiirteillä luokittelusta selvisi AD-puu. Tutkijat huomasivat, että uudet ominaispiirteet lisäsivät luokittelijoiden tarkkuutta kaikkien muiden menetelmien paitsi AD-puun kohdalla. Kun tutkijat lisäsivät ominaispiirteisiin mukaan heidän itse määrittelemät uudet ominaispiirteet, luokittelijat saavuttivat 97.5-99.9 prosentin tarkkuuden, 0.03-0.0001 prosentin asteen virheellisissä positiivisissa luokitteluissa ja 0.013-0.0001 prosentin asteen virheellisissä negatiivisissa luokitteluissa. Tällöin luokittelusta puolestaan selvisi parhaiten tukivektorikone. Tutkijat vertasivat saamiaan tuloksiaan myös tunnettuihin antivirusohjelmistoihin. Kaikki tutkimuksessa opetetut luokittelijat suoriutuivat antivirusohjelmistoja paremmin aineistossa mukana olleiden haitallisten JavaScript-ohjelmakoodinäytteiden tunnistamisesta.

3 Tutkimusmenetelmä

Tässä luvussa kuvataan tutkimuksessa käytetyn tutkimusmenetelmän yksityiskohdat ja perustellaan valintoihin johtaneet päätökset. Luku alkaa ohjatun koneoppimisprosessin yleisellä kuvauksella. Tämän jälkeen lukua jatketaan tutkimuksessa käytetyn aineiston ja sen hankintaprosessin kuvauksella. Tämän jälkeen kuvataan tutkimuksessa käytetyt ominaispiirteet, jonka jälkeen esitellään tutkimuksessa käytetyt koneoppimismenetelmät. Luku päättyy luokittelijoiden opetuksen ja testauksen kuvaukseen.

3.1 Ohjattu koneoppiminen yleisesti

Ohjatussa koneoppimisessa tavoitteena on algoritmeja käyttäen tuottaa ulkoisesti syötetyistä instansseista yleisiä hypoteeseja, joidenka avulla voidaan tehdä ennustuksia tulevaisuudessa tavattavasti instansseista. Hypoteesi on malli, jonka avulla uusia ennalta tuntemattomia instansseja voidaan jakaa ennalta tunnettuihin luokkiin. Ohjattu koneoppimisprosessi sisältää yleensä seuraavan kaltaiset vaiheet: ratkaistavan ongelman määrittelyn; vaadittavan aineiston tunnistamisen ja hankkimisen; aineiston esikäsittelymenetelmien valinnan; opetus- ja testiaineiston valinnan; mallin muodostamiseen käytettävän algoritmin valinnan; mallin opettamisen ja testaamisen; sekä päätöksen mallin hyväksymisestä. Prosessin aikana on mahdollista palata uudelleen mihin vain prosessin vaiheista (Kotsiantis 2007).

3.2 JavaScript-ohjelmakoodiaineisto ja sen kerääminen

Seuraavaksi kuvataan tutkimuksessa käytetyt JavaScript-ohjelmakoodiaineistot sekä perustellaan, miksi kyseiset aineistot valittiin ja miten ne kerättiin. Ensin esitellään tutkimuksessa käytetty haitallista JavaScript-ohjelmakoodia edustava aineisto ja sen kerääminen. Tämän jälkeen esitellään ei-haitallista JavaScript-koodia edustava aineisto ja sen kerääminen. Tutkimuksessa käytettävä aineisto on kokonaisuudessaan web-selaimilla suoritettavaksi tarkoitettua JavaScript-ohjelmakoodia.

3.2.1 Haitallinen JavaScript-ohjelmakoodiaineisto

Tutkimuksessa käytetty haitallista JavaScript-ohjelmakoodi edustava aineisto on peräisin seuraavasta GitHub-säiliöstä: `https://github.com/geeksonsecurity/js-malicious-dataset`. Aineisto sisältää yhteensä 1156 haitalliseksi väitettyä HTML-tiedostoa ja 1357 haitalliseksi väitettyä JavaScript-tiedostoa. Säiliö sisältää useita tunnettuja JavaScript-haittaohjelmia kuten RIG ja Angler exploitation kit. Lisäksi säiliössä on runsaasti tuntemattomampia haittaohjelmia. Säiliön näytteet ovat kaikki saman käyttäjän lisäämiä ja viimeksi säilöön on lisätty näytteitä vuoden 2017 tammikuussa.

Kyseinen aineisto valittiin tähän tutkimukseen edustamaan haitallista JavaScript-ohjelmakoodia, koska se oli helposti saatavilla. Tutkimuksen resurssit (käytettävissä oleva aika ja asiantuntemus) eivät mahdollistaneet laajempaa aineistohankintaprosessia. Tutkimuksen resurssit eivät myöskään mahdollistaneet aineiston tarkkaa läpikäyntiä näytteiden haitallisuuden varmistamiseksi, joten tutkimuksessa oletetaan, että säiliön tiedostot ovat haitallisia. Samaa ainestoa ovat myös käyttäneet ainakin Patil ja Patil (2017) haitallisen JavaScript-koodin tunnistamiseen liittyvässä tutkimuksessaan.

Osasta näytteistä on olemassa erikseen sekä obfuskoitu-versio että versio, josta obfuskointi on poistettu. Tiedostot, joista obfuskointi on poistettu ovat merkitty `deobfuscated` avainsanalla. Osaan näytteiden tiedostonimistä on myös merkitty päivämäärä, jolloin kyseinen näyte on ilmestynyt. Osasta tiedostonimistä käy myös ilmi, mikäli näyte on ilmestynyt jonkin web-sivun etusivulla. Tällöin näytteet on merkitty avainsanoilla `lp` tai `landingpage`. Web-sivuille injektoidut JavaScript-tiedostot ovat puolestaan merkitty avainsanalla `injected`. Tuntemattomat näytteet ovat sijoitettu erikseen `misc` kansioon.

Tässä tutkimuksessa aineistosta valittiin käytettäväksi kuitenkin vain 150 ensimmäistä JavaScript-tiedostoa, joista obfuskointia ei oltu erikseen purettu. Käytettyjen tiedostojen määrä valittiin siten, että haitallisten tiedostojen suhde ei-haitallisiin tiedostoihin on tutkimuksen koeasetelmissa maksimissaan 15/100. Ei-haitallisten tiedostojen määrä jouduttiin rajaamaan 1000 näytteeseen tutkimusta varten kirjoitetun ohjelman puutteiden ja tutkimukseen käytetyn laitteiston keskusmuistin aiheuttamien rajoitteiden takia. Näin ollen, rajoittui myös käytettyjen haitallisten tiedostojen määrä 150 näytteeseen. Tutkimuksessa päätettiin

käyttää ensimmäisiä tiedostoja, koska tiedostojen valinnan satunnaistamisella ei koettu pystyttävän saavuttavan mitään lisäarvoa. Koko aineisto sisälsi vain muutamia JavaScript-tiedostoja, joista obfuskointi oli purettu. Täten tutkimuksessa päätettiin käyttää vain tiedostoja, joista obfuskointia ei ole poistettu.

3.2.2 Ei-haitallinen JavaScript-ohjelmakoodiaineisto

Tutkimuksessa käytetty ei-haitallista JavaScript-ohjelmakoodia edustava aineisto puolestaan kerättiin hyödyntämällä Amazon Alexa Top Sites-listalla julkaistuja web-sivuja. Lista on ladattavissa osoitteesta: `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`. JavaScript-tiedostojen URL-osoitteet kerättiin tätä tarkoitusta varten kirjoitetulla Python-ohjelmalla. Ohjelma kutsui Amazon Alexa Top Sites-listalla olevia web-sivuja erottaen niistä sivujen sisältämien JavaScript-tiedostojen URL-osoitteet. Ohjelma tallensi osoitteet erilliseen tekstitiedostoon. Ohjelmalla kerättiin yhteensä 70 000 JavaScript-tiedoston URL-osoitetta. Osoitteet eivät olleet peräisin peräkkäisiltä websivuilta, koska ohjelma ei onnistunut samaan JavaScript-tiedostojen osoitteita kaikilta web-sivuilta.

JavaScript-tiedostojen sisällön haku puolestaan suoritettiin tarkoitusta varten kirjoitetulla erillisellä Python-ohjelmalla. Ohjelma kävi läpi tiedostojen URL-osoitteita sisältävää listaa, yrittäen hakea ja tallentaa tiedostojen sisältöä erillisiin UTF-8 koodattuihin tekstitiedostoihin. 70 000 osoitteen listalta ohjelma onnistui tallentamaan 11 091 JavaScript-tiedoston sisällön erilliseksi tekstitiedostoksi. Kuitenkin aineistosta käytettiin tutkimukseen vain ensimmäistä 1000 tiedostoa. Valittujen tiedostojen järjestyksen satunnaistamisella ei koettu saavutettavan mitään etua myöskään tässä tapauksessa, sillä aineistossa olevat peräkkäiset tiedostot ovat harvoin peräisin samalta web-sivulta.

3.3 Tutkimuksessa käytettävät ominaispiirteet ja niiden eristäminen

Tutkimuksessa käytetään pelkästään staattisia ominaispiirteitä. Käytettävät ominaispiirteet ovat tiedostojen tekstimuodossa olevista esityksistä eristettyjä n-grammeja. N-grammit valittiin tutkimuksessa käytetyiksi ominaispiirteiksi, koska ne ovat yksi yleisemmin käytetyistä ominaispiirteistä. Perinteisesti haitallisen ohjelmakoodin tunnistamisessa on käytetty

ohjelmasigneerauksiin perustuvaa menetelmää, joten on perusteltua käyttää saman tyylisiä merkkijonoja myös koneoppimista hyödyntävien tunnistamismenetelmien syötteinä. Eräs vaihtoehto tällaisiksi syötteiksi ovat n-grammit, jotka ovat koko ohjelmakoodin alueelta muodostettavia kiinteän mittaisia merkkijonoja (Abou-Assaleh et al. 2004).

3.3.1 N-grammit

N-grammien luonteeseen kuuluu, etteivät ne vangitse vain statistiikkaa kyseisen mittaisista merkkijonoista vaan ne vangitsevat implisiittisesti pitempien merkkijonojen frekvenssit. N-grammit voivat vangita koodista ominaisuuksia, jotka ovat esimerkiksi tunnusomaisia jollekin käytetylle työkalulle, koodigeneraattorille, kääntäjälle tai ohjelmointiympäristölle. Lisäksi n-grammit voivat vangita koodista ominaisuuksia, jotka ovat ominaisia jollekin tietylle koodinkirjoittajalle tai koodaustyylille. (Abou-Assaleh et al. 2004). Tutkimuksessa käytettävät n-grammit ovat 2-grammit ja 3-grammit. Nämä pituudet valittiin tutkimukseen, koska samankaltaisia pituuksia on käytetty myös aikaisemmissa aiheeseen liittyvissä tutkimuksissa. Lisäksi tutkimuksessa käytettävissä olevan laitteiston muistirajoitteet eivät antaneet mahdollisuutta tutkia pitempiä n-grammeja.

3.3.2 Ominaispiirteiden valinta

N-grammien ongelmana on, että niitä syntyy tiedostoista eristettäessä helposti liian paljon, jotta niitä kaikkia voitaisiin käyttää koneoppimismenetelmien syötteinä. Tässä tutkimuksessa n-grammien määrään rajoittamiseen käytetään Tf-idf-lukua, Fisher Score-lukua ja Information Gain Ratio-lukua. Nämä luvut valittiin, koska ne ovat laajalti käytettyjä muissa aiheeseen liittyvissä tutkimuksissa, joissa ne ovat myös tuottaneet hyviä tuloksia.

Tf-idf-luku termille t saadaan, kun otetaan luonnollinen logaritmi dokumenttien lukumäärän sekä termin t sisältävien dokumenttien lukumäärän osamäärästä ja kerrotaan tämä termin t sisältävien dokumenttien lukumäärällä (Manning et al. 2008). Tässä tutkimuksessa luvusta käytetään hieman muokattua versioita, jossa termin t sisältävien dokumenttien lukumäärään summataan yksi osamäärää laskettaessa. Tällä tavoin vältytään nollalla jakamiselta. Fisher Score-lukujen avulla on puolestaan ideana löytää ominaispiirteet, jotka mak-

simoivat datapisteiden etäisyydet eri luokkien välillä samalla minimoiden datapisteiden etäisyyden luokkien sisällä (Gu et al 2011). Tutkimuksessa Fisher Score-luku termille t lasketaan jakamalla termin t positiivisten havaintojen joukossa olevien esiintymisten lukumäärän keskiarvon ja termin t negatiivisten havaintojen joukossa olevien esiintymisten lukumäärän keskiarvon erotus termin t positiivisten havaintojen joukossa olevien esiintymisten lukumäärän keskihajonnan ja termin t negatiivisten havaintojen joukossa olevien esiintymisten lukumäärän keskihajonnan summalla (Le et al. 2014). Information Gain Ratio-luku puolestaan esittää kuinka tarkasti termi t ennustaa luokkien jakaumaa. Esimerkiksi päätöspuiden opettamiseen käytetty algoritmi C4.5 käyttää Information Gain-lukua (Mori et al. 2002). Tutkimuksessa kaikki edellä mainitut luvut laskettiin tarkoitusta varten kirjoitetulla Python-ohjelmalla.

Tutkimuksessa mallien syötteinä käytettyjen n-grammien lukumäärät ovat 50, 100, 200, 400, 800 ja 1600. Määrät valittiin tutkimukseen, koska samankaltaisia lukumääriä on käytetty myös aikaisemmissa aiheeseen liittyvissä tutkimuksissa. Ominaispiirteinä käytetyt n-grammit valitaan Tf-idf-luvun, Fisher Score-luvun tai Information Gain Ratio-luvun suuruuden avulla. Suurimmat arvot saaneet n-grammit otetaan mukaan ominaispiirteiksi.

3.4 Tutkimuksessa käytettävät koneoppimismenetelmät

Seuraavaksi esitellään tutkimuksessa käytetyt ja vertailtavat koneoppimismenetelmät. Kyseiset koneoppimismenetelmät ovat päätöspuu, monikerroksinen neuroverkko ja geneettinen ohjelmointi. Jokaisen menetelmän esittelyä seuraa perustelu sille, miksi kyseinen menetelmä valittiin mukaan tutkimukseen.

3.4.1 Päätöspuut

Päätöspuu on puu, joka luokittelee instanssin perustuen sen ominaispiirteiden arvoihin. Päätöspuun jokainen solmu edustaa jotain luokiteltavan instanssin ominaispiirteistä ja jokainen haara edustaa solmun odotusarvoa. Instanssit luokitellaan alkaen juurisolmusta ja jaetaan luokkiin perustuen niiden ominaispiirteiden arvoihin. Optimaalisen binaarisen pää-

töspuun rakentaminen on NP-täydellinen ongelma, mutta puun muodostamiseen on tarjolla tehokkaita heuristisia algoritmeja (Kotsiantis 2007).

Päätöspuut voidaan jakaa karkeasti kahteen ryhmään: yksimuuttujaisiin ja monimuuttujaisiin. Yksimuuttujapäätöspuissa päätöksen solmuissa perustuu yhden muuttujan arvoihin, kun taas monimuuttujapäätöspuissa päätöksenteko voi perustua useamman muuttujan arvoihin. Päätöspuut tarjoavat monia etuja muihin koneoppimismenetelmiin nähden: käyttäjien on helppo ymmärtää päätöksenteon perusteita; päätöspuut voivat käsitellä monen muotoista syötteitä, kuten nominaalista, numeerista ja tekstimuodossa olevaa dataa; päätöspuiden avulla voidaan käsitellä virheellisiä aineistoja ja aineistoja, jotka sisältävät puuttuvia arvoja (Bhargava et al. 2013).

Menetelmä valikoitui mukaan tutkimukseen, koska useissa aikaisemmissa haitallisen ohjelmakoodin tunnistamista koneoppimismenetelmin tutkivissa tutkimuksissa on käytetty J48-algoritmin avulla rakennettuja päätöspuita. Usein J48-päätöspuut ovat myös onnistuneet tunnistamisessa hyvin muihin menetelmiin verrattuna. Tässä tutkimuksessa päätöspuu toteutettiin Scikit Learn-kirjastoa käyttävällä Python-ohjelmalla. Scikit Learn-kirjastossa ei kuitenkaan ole J48-algoritmin toteutusta, joten päätöspuut rakennetaan käyttäen CART-algoritmia. CART on yleisesti päätöspuiden muodostamiseen käytetty algoritmi, jossa puun solmujen epäpuhtautta arvioidaan Gini-indeksin avulla (Loh 2013).

3.4.2 Monikerroksiset keinotekoiset neuroverkot

Yksinkertaisessa tapauksessa monikerroksiset keinotekoiset neuroverkot ovat verkkoja, joissa keinotekoisia neuroneja on liitetty yhteen erilaisin kuvion. Keinotekoiset neuronit saavat tyypillisesti syötteenä jonkin vektorin X ja vektorin W tulojen summan. Perustuen sitten johonkin tämän summan raja-arvoon neuronin tuloste on joko 0 tai 1. Monikerroksisen neuroverkon neuronit ovat yleensä jaettu kolmeen kerrosluokkaan: syötekerrokseen, piilokerroksiin ja tulostekerrokseen. Syötekerroksen jokainen neuroni edustaa jotain luokiteltavan instanssin ominaispiirrettä (Kotsiantis 2007).

Eteenpäin syötetyissä neuroverkoissa syötesignaali kulkee vain yhteen suuntaan, syötekerroksesta kohti tulostekerrosta. Edellisen kerroksen tulosteet toimivat syötteinä seuraavalla

kerrokselle. Kokonaan kytketyssä neuroverkossa kaikki edellisen kerroksen neuronit vaikuttavat seuraavan kerroksen syötteeseen. Kun neuroverkkoja käytetään luokitteluun, päätös instanssin luokan osoittamisesta tehdään verkon tulostekerroksen tulosteen perusteella jotain apufunktiota käyttäen (Kotsiantis 2007).

Monikerroksiset neuroverkot valikoituvat yhdeksi tässä tutkimuksessa tarkasteltavaksi menetelmäksi valikoituvat, koska näiden ympärillä on viime vuosien aikana ollut paljon kohua. Tämän lisäksi neuroverkkoja on sovellettu monentyyppisten ongelmien ratkaisuun. Näin ollen, onkin mielenkiintoista testata neuroverkkojen soveltuvuutta myös haitallisen JavaScript-koodin tunnistamiseen. Lisäksi neuroverkkoja on aikaisemmissa tutkimuksissa käytetty jonkin verran myös haitallisen ohjelmakoodin tunnistamiseen, joten menetelmän avulla saadaan myös vertailukohta aikaisempiin tutkimuksiin. Tässä tutkimuksessa käytetty neuroverkko on kolme kerroksinen täydellisesti kytketty eteenpäin syötetty neuroverkko. Verkon neuronien kertoimien optimointi tapahtuu käyttäen Backpropagation-algoritmia. Backpropagation on tunnettu, neuroverkkojen opetuksessa laajalti käytetty algoritmi (Kotsiantis 2007).

Tutkimuksessa neuroverkon toteuttaminen tapahtui Keras-kirjastoa käyttävän Python-ohjelman avulla. Mallia ohjelmaan saatiin osoitteesta: `https://www.kaggle.com/parthsuresh/binary-classifier-using-keras-97-98-accuracy`. Keras-kirjaston avulla neuroverkon ominaisuuksien määrittäminen on tehty helpoksi. Käytetyssä verkossa oli kolme täysin yhdistettyä kerrosta. Syötekerroksen dimensioiden määrä riippua aina käytettävien ominaispiirteiden määrästä. Sisäkerros oli jokaisessa tapauksessa aina kymmenulotteinen. Sisäkerros käyttää ”ReLu”-aktivointia. Tulostekerros puolestaan on aina yksiulotteinen ja käyttää ”Sigmoid”-aktivointia. Kerrosten solmujen kerrointen alustamiseen käytettiin satunnaisia arvoja normaalijakaumasta. Tässä tapauksessa neuroverkkojen opettamiseen Keras käytti Tensoflow-kirjastoa. Opettamiseen käytetty algoritmi oli gradientin laskeutumiseen pohjautuvaan Adam-algoritmi. Hävikkifunktiona toimi binäärinen ristientropia. Verkon validointiin käytettiin luokittelun tarkkuutta. Verkon opettamiseen käytettiin aina kymmenen epookkia, joissa gradientin päivitys suoritettiin aina 32 näytteen jälkeen.

3.4.3 Geneettinen ohjelmointi

Geneettinen ohjelmointi on jatke evoluutioteoriaan perustuviin laskennallisiin tekniikoihin kuuluville geneettisille algoritmeille. Geneettiset algoritmit esittävät ratkaisun kiinteän mittaisina binäärisinä merkkijonoina, kun taas geneettisen ohjelmoinnin ratkaisut ovat vaihtelevan pituisia tietokoneongelmia. Nämä ohjelmat ovat funktio- ja terminaalikokoelmia sekä ohjeita näiden suorittamiseen (Hansen et al. 2007).

Geneettinen ohjelmointi ratkaisee annetun ongelman ilman, että käyttäjän tarvitsee tietää tai määritellä ratkaisun muotoa tai rakennetta etukäteen. Geneettisessä ohjelmoinnissa tietokoneohjelmapopulaatiota kehitetään sukupolvesta toiseen stokastisesti toivoen, että uusi sukupolvi selviytyy ratkaistavasta ongelmasta edellistä sukupolvea paremmin. Ohjelmien keskinäinen vertailu perustuu niin sanottuun sopivuusarvoon, joka on kvantifioitu määre ohjelman kyvylle ratkaista haluttu ongelma. Ohjelmat, joilla on korkea sopivuusarvo, valitaan tuottamaan uusi sukupolvi ohjelmia. Uusia ohjelmia tuotetaan käyttäen primaarisia geneettisiä operaatioita kuten ohjelmien ominaisuuksien sekoittamista sekä ominaisuuksien mutaatioita (Poli et al. 2008).

Geneettinen ohjelmointi valittiin mukaan tähän tutkimukseen, koska se vaikutti mielenkiintoiselta ja erilaiselta menetelmältä verrattuna muihin valittuihin menetelmiin. Geneettinen ohjelmointi on ainoa tässä tutkimuksessa tarkasteltavista menetelmistä, jossa mukana on stokastinen elementti muualla kuin alkuarvojen arvauksessa. Lisäksi tutkimuksessa esitellyn aikaisemman tutkimuksen perusteella haitallisen ohjelmakoodin tunnistamista geneettistä ohjelmointia käyttäen on tutkittu vähemmän kuin esimerkiksi käyttäen päätöspuita, neuroverkkoja tai Bayesilaiseen todennäköisyyteen perustuvia menetelmiä. Näin ollen, ottamalla geneettinen ohjelmointi mukaan tutkimuksella on parempi mahdollisuus tuottaa uutta tietoa haittaohjelmien tunnistamisesta yleisesti koneoppimismenetelmin eikä vain JavaScript-ohjelmakoodin kontekstissa.

Tutkimuksessa geneettisen ohjelmoinnin toteuttaminen tapahtui Gplearn-kirjastoa käyttävän Python-ohjelman avulla. Malli ohjelmaan saatiin osoitteesta: `https://gplearn.readthedocs.io/en/stable/examples.html#example-1-symbolic-regressor`. Parhaan ratkaisun löytämiseen käytettiin enintään 20

sukupolvea. Jokainen sukupolvi sisältää 5000 ohjelmaa, joista 20 sopivinta otetaan mukaan seuraavaan sukupolveen. Todennäköisyys sille, että sukupolven seuraavalla kierrokselle selvinnyt ohjelma vaihdetaan satunnaiseen ohjelmaan, oli 0.7. Lisäksi asetettiin eri todennäköisyyksiä erilaisille mutaatioille, joita suoritetaan seuraavaan sukupolveen selvinneille ohjelmille. Suurien ohjelmien sopivuutta rankaistiin kertoimella 0.01. Arvioitiin käytettävien näytteiden enimmäismäärä oli 90 prosenttia kaikista opettamiseen kulloinkin tarjolla olevista näytteistä.

3.5 Mallien opettaminen ja testaaminen

Tutkimuksessa käydään läpi kaikki seuraavaksi luokiteltavilla muuttujilla muodostettavissa olevat kombinaatiot. Muuttujat ovat haitallisten näytteiden suhde ei-haitallisiin näytteisiin, n-grammin pituus, n-grammien määrä, n-grammien valintaan käytetty menetelmä ja käytetty koneoppimismenetelmä. Tutkimuksessa käytetyt haitallisten näytteiden suhteet ei-haitallisiin näytteisiin olivat: 50/1000, 100/1000 ja 150/1000. Käytetyt n-grammien pituudet olivat puolestaan: 2 ja 3.

Kun n-grammin pituus oli kaksi, kaikille haitallisten näytteiden suhteille ei-haitallisiin näytteisiin 50/1000, 100/1000 ja 150/1000 aineistosta saatiin luotua 10 274 uniikkia n-grammia. Kun taas n-grammin pituutena oli kolme, niin suhteelle 50/1000 aineistosta saatiin luotua 331 185 uniikkia n-grammia, suhteelle 100/1000 saatiin 332485 uniikkia n-grammia ja suhteelle 150/1000 saatiin luotua uniikkia 336 563 n-grammia. Näistä luokittelussa käytettäväksi valittiin 50, 100, 200, 400, 800 ja 1600 ominaispiirrettä käyttäen eri menetelmiä. Käytetyt menetelmät olivat: Tf-idf-luku, Fisher Score-luku ja Information Gain Ratio-luku. Käytetyt koneoppimismenetelmät olivat: keinotekoinen neuroverkko, CART-päätöspuu ja geneettinen ohjelmointi.

Kaikissa tapauksissa mallien opettaminen ja testaaminen tapahtuivat käyttäen 4-fold-menetelmää. 4-fold-menetelmän käyttämiseen päädyttiin, koska se on varsin yleisesti koneoppimismenetelmin luotavien luokittelijoiden opettamiseen ja testaamiseen käytetty menetelmä. Tutkimuksessa jokaisessa tehdystä kokeesta tallennettiin oikein luokiteltujen haitallisten näytteiden lukumäärä, oikein luokiteltujen ei-haitallisten näytteiden lukumäärä,

väärin luokiteltujen haitallisten näytteiden lukumäärä, väärin luokiteltujen ei-haitallisten näytteiden lukumäärä, todellisten positiivisten luokittelujen osuus, todellisten negatiivisten luokittelun osuus, väärien positiivisten luokittelujen osuus, väärien negatiivisten luokittelun osuus ja luokittelun tarkkuus. Luvut olivat aina 4-fold menetelmän neljän kierroksen keskiarvoja jokaiselle kierrokselle saaduista vastaavista luvuista. Näin saatiin yhteensä 324 havaintoriviä, jotka toimivat tämän tutkimuksen varsinaisena tutkimusaineistona.

3.6 Tutkimusaineiston analysointi

Tässä luvussa kuvataan kuinka tutkimuksen aineistoa analysointiin. Tutkimuksessa on tarkoituksena tutkia edellisessä kappaleessa mainittujen muuttujien saamien arvojen vaikutusta haitallisten ja ei-haitallisten JavaScript-koodinäytteiden luokitteluun erikoistuneen luokittelijan suorituskykyyn. Kyseiset muuttujat olivat haitallisten näytteiden suhde ei-haitallisiin näytteisiin, n-grammin pituus, n-grammien määrä, n-grammien valintaan käytetty menetelmä ja käytetty koneoppimismenetelmä. Tutkimuksessa aineiston analyysi tapahtuu useassa vaiheessa. Ensin aineistoa tarkastellaan tilastollisia menetelmiä apuna käyttäen. Tämän jälkeen aineistoa tarkastellaan silmämääräisesti. Tämän jälkeen arvioidaan tulosten reliabiliteettia ja validiteettia. Lopuksi vertaillaan tilastollisten menetelmien ja silmämääräisen tarkastelun avulla saatuja tuloksia aikaisempien tutkimusten tuloksiin.

Tutkimuksessa käytetyt varsinaiset tilastolliset menetelmät ovat Kruskal-Wallisin testi ja Dunnin testi. Kruskal-Wallisin testin avulla selvitetään poikkeavatko kulloinkin vertailtavat joukot toisistaan. Dunnin testiä käytetään Kruskal-Wallisin testin jälkitarkasteluun. Dunnin testin avulla selvitään miten joukot eroavat toisistaan. Tämän lisäksi joukkojen eroavaisuuksien tarkasteluun käytetään joukkojen eri muuttujille laskettavia keskiarvoja.

Kruskal-Wallisin testi on parametriton joukkojen sijoitukseen perustuva testi, jolla voidaan testata kolmen tai useamman joukon samankaltaisuutta (arvojen samankaltaisuutta joukkojen välillä). Testin kohdalla ei tarvitse olettaa, että joukkojen arvot olisivat normaalijakautuneita. Testi on kuitenkin tarkoitettu suhteellisen pienille otoskoille ja sen luotettavuus heikkenee suuremmilla otoskoilla (Meyers & Seaman 2011).

Tässä tutkimuksessa Kruskal-Wallisin testiä käytetään kaikissa joukkojen ensimmäisissä vertailussa, vaikka joukkojen määrä on yhden testin kohdalla vain kaksi ja otoskoko (324 havaintoa) on yleisesti suuri. Tutkimuksessa Kruskal-Wallisin testin käyttöön päädyttiin testin oletusten yksinkertaisuuden vuoksi. Tutkimuksessa hyväksytään joukkojen määrästä ja otoskoosta aiheutuvat epätarkkuudet. Epätarkkuuksia uskotaan pystyttävän paikkaavan aineistolle tehtävällä silmämääräisellä tarkastelulla. Dunnin testi puolestaan valittiin tutkimukseen mukaan, koska se on sopiva ja suosittu menetelmä Kruskal-Wallisin testin jälkeiseen ryhmien keskinäiseen vertailuun (Dinno 2015).

Tilastollisia analyysit tehdään aina kerralla yhden muuttujan suhteen eli analyyseissa on tarkoituksena tutkia yhden muuttujan saamien eri arvojen vaikutusta luokittelusta suoriutumiseen, kun muut muuttujat saavat arvoikseen kaikki mahdolliset kombinaatiot niille mahdollista arvoista. Tällöin tarkoituksena on tutkia muuttujan arvojen keskimääräistä vaikutusta luokittelusta suoriutumiseen. Esimerkiksi n-grammien pituutta tarkastellessa, aineisto jaetaan kahteen joukkoon. Ensimmäisessä joukossa n-grammin pituus saa arvon 2 ja toisessa ryhmässä arvon 3. Näiden joukkojen alkiot ovat sitten kaikkien muiden muuttujien arvojen muodostamat kombinaatiota eli kummassakin joukossa on tässä tapauksessa 162 havaintoa (aineistossa on yhteensä 324 havaintoa).

Kulloinkin tarkastelussa olevien joukkojen eroavaisuuksia toisistaan tarkasteltiin tekemällä aina Kruskal-Wallisin testi väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelun osuuden ja luokittelun tarkkuuden suhteen. Hypoteesiksi Kruskal-Wallisin testille asetettiin, etteivät joukot poikkea toisistaan. Mikäli huomattiin, että hypoteesi kumoutui (Kruskal-Wallisin testin p-arvo oli pienempi kuin 0.05), niin ryhmien keskinäisiä eroja tarkasteltiin Dunnin testin avulla. Tutkimuksessa tilastollisista tarkasteluista raportoidaan kaikki tehdyt Kruskal-Wallisin testit ja testien p-arvot (Kruskal-Wallisin testin H-lukua ei raportoida). Samassa yhteydessä raportoidaan tarkastelun yksityiskohdat (tarkastelun tarkoitus, ryhmien määrät ja ryhmien koot). Lisäksi tilanteissa, joissa päädytään ryhmien vertailuun Dunnin testillä, raportoidaan myös Dunnin testin tulokset. Ryhmien keskinäisistä vertailuista raportoidaan ainoastaan ne tapaukset, joiden välillä on Dunnin testin mukaan merkittävää eroa. Näissä tapauksissa raportoidaan myös, millaisesta erosta on ky-

symys. Tilastolliset tulokset raportoidaan lausemuotoon kirjoitettuna tekstinä. Lisäksi raportoidaan ryhmien keskiarvoja, joiden avulla havainnollistetaan ryhmien välisiä eroja.

Silmämääräiset tarkastelun tarkoituksena tutkimuksessa on täydentää tilastollista analyysiä. Tarkastelun tarkoituksena on yrittää löytää puutteellisia tai virheellisiä tulkintoja tilastollisista analyyseistä sekä yrittää löytää aineistosta nähtävissä olevia säännönmukaisuuksia ja epäsäännönmukaisuuksia. Tarkastelua varten ei tutkimuksessa nimetä tutkimushypoteeseja. Tarkastelun tulokset raportoidaan lausemuotoon kirjoitettuna tekstinä. Reliabiliteetin ja validiteetin arvioinnin raportointi tapahtuu myös lausemuotoon kirjoitettuna tekstinä. Lopuksi tämän tutkimuksen tuloksia vertaillaan tutkimuksessa esiteltyjen aikaisempien tutkimusten tuloksiin.

4 Tutkimuksen tulokset

Tässä luvussa esitetään tutkimuksen tulokset. Luvun aluksi esitellään tilastollisia menetelmiä apuna käyttäen saadut tulokset, jonka jälkeen esitetään silmämääräisellä tarkastelulla saadut tulokset. Näiden jälkeen esitetään yhteenveto tilastollisin menetelmin ja silmämääräisellä tarkastelulla saaduista tuloksista. Tämän jälkeen esitetään arviot tutkimusmenetelmän ja tulosten reliabiliteetista ja validiteetista. Luvun lopuksi tutkimuksen tuloksia vertaillaan aikaisempien tutkimusten tuloksiin.

4.1 Tilastollisin menetelmin saadut tulokset

Tässä luvussa esitetään tilastollisin menetelmin saadut tulokset. Aluksi käydään läpi haitallisten näytteiden ja ei-haitallisten näytteiden suhteen vaikutus luokittelusta suoriutumiseen. Tämän jälkeen käydään läpi n-grammien pituuden vaikutus, jonka jälkeen lukua jatketaan n-grammien määrän vaikutuksen esittelyllä. Tämän jälkeen tarkastellaan n-grammien valintaan käytetyn menetelmän vaikutusta. Luvun lopuksi tarkastellaan käytetyn koneoppimismenetelmän vaikutusta luokittelusta suoriutumiseen.

4.1.1 Haitallisten näytteiden ja ei-haitallisten näytteiden suhde

Haitallisten näytteiden ja ei-haitallisten näytteiden määrien suhteen vaikutusta luokittelusta suoriutumiseen tutkittaessa tarkasteltiin kolmea ryhmää. Ensimmäisen ryhmän opetusaineistossa oli mukana 50 haitallista JavaScript-koodinäytettä. Toisen ryhmän opetusaineistossa oli puolestaan mukana 100 haitallista JavaScript-koodinäytettä. Kun taas kolmannen ryhmän opetusaineistossa oli mukana 150 haitallista JavaScript-koodinäytettä. Kaikissa tapauksissa ryhmien opetusaineisto sisälsi 1000 ei-haitallista JavaScript-koodinäytettä. Jokaisen ryhmän koko oli 108 näytettä.

Tarkasteltaessa eroja luokkien välillä väärien positiivisten luokittelujen osuudessa suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p = 0.000047$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Ryhmien välisten erojen tarkastelua jatketaan Dunnin testin ja ryhmien keskiarvojen avulla. Dunnin testin

avulla huomataan, että ero 50 haistallista näytettä ja 150 haistallista näytettä sisältävien ryhmien sekä ero 100 haitallista näytettä ja 150 haitallista näytettä sisältävien ryhmien välillä ovat tilastollisesti merkitseviä. Dunnin testin p-arvot ovat näissä tapauksissa 0.0018 ja $p < 0.0001$. Ryhmien keskiarvot väärien positiivisten luokittelujen osuuksille ovat järjestyksessään: 0.004825, 0.007496 ja 0.016703. Keskiarvoista huomataan, että väärien positiivisten luokittelujen osuus näyttää kasvavan, kun haitallisten näytteiden suhteellista osuutta kasvatetaan, vaikkakin osuudet ovat edelleenkin matalia. Kasvun voidaan arvioida johtuvan yksinkertaisesti syystä, että haitallisia luokitteluja tehdään enemmän, kun haitallisia näytteitä on aineistossa enemmän mukana.

Tarkasteltaessa eroja luokkien välillä väärien negatiivisten luokittelujen osuuden suhteen saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p < 0.000001$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Täten ryhmien välisiä eroja tarkastellaan Dunnin testin ja ryhmien keskiarvojen avulla. Dunnin testin avulla huomataan, että ero 50 haistallista näytettä ja 100 haistallista näytettä sisältävien ryhmien sekä ero 100 haitallista näytettä ja 150 haitallista näytettä sisältävien ryhmien välillä ovat tilastollisesti merkitseviä. Dunnin testin p-arvot ovat näissä tapauksissa $p < 0.0001$ ja $p = 0.0016$. Ryhmien keskiarvot väärien negatiivisten luokittelujen osuuksille ovat järjestyksessään: 0.474752, 0.363622, 0.239278. Keskiarvoista huomataan, että väärin negatiivisten luokittelujen osuus näyttää laskevan, kun haitallisten näytteiden suhteellista osuutta kasvatetaan. Laskun voidaan arvioida johtuvan syystä, että ero haitallisten ja ei-haitallisten näytteiden välillä luokittelijoissa selkeytyy, kun haitallisten näytteiden suhteellista osuutta kasvatetaan 5 prosentista 10 prosenttiin tai 5 prosentista 15 prosenttiin.

Tarkasteltaessa eroja luokkien välillä luokittelijan tarkkuudessa saadaan Kruskal-Wallisin testin p-arvoksi 0.158911. Tämän vuoksi perushypoteesi säilyy, eikä ryhmien välillä ole muuttujan suhteen nähtävissä merkittävää eroa. Kokonaisuudessa haitallisten näytteiden ja ei-haitallisten näytteiden määrien suhteen vaikutus luokittelijan suoriutumiseen jättää ristiriitaisen kuvan, toisaalta väärien positiivisten luokitteluiden suhteellinen osuus kasvaa ja toisaalta väärien negatiivisten luokitteluiden suhteellinen osuus laskee, kun haitallisten näytteiden suhteellista osuutta kasvatetaan opetus- ja testiaineistossa. Kuitenkin johtopää-

töksenä voidaan sanoa, että luokittelun suorituskyky paranee, kun haitallisten näytteiden suhteellista osuutta kasvatetaan 5 prosentista 10 prosenttiin tai 5 prosentista 15 prosenttiin.

4.1.2 N-grammien pituus

N-grammien pituuden vaikutusta luokittelijan suorituskykyyn tutkittaessa tarkasteltiin kahta ryhmää. Ensimmäisessä ryhmässä n-grammin pituus oli kaksi merkkiä ja toisessa ryhmässä n-grammin pituus oli kolme merkkiä. Kummankin ryhmän koko oli 162 näytettä.

Tarkasteltaessa eroja luokkien välillä väärien positiivisten luokittelujen osuudessa, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p = 0.012537$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Täten tarkastellaan ryhmien eroja Dunnin testin ja ryhmien keskiarvojen avulla. Ero ryhmien välillä on tilastollisesti merkitsevä. Dunnin testin p-arvo on 0.0125. N-grammin pituudella kaksi keskiarvoksi väärien positiivisten luokittelujen osuuksille saadaan 0.010143. N-grammin pituudella kolme vastaavaksi keskiarvoksi saadaan 0.009206. Keskiarvoista huomataan, että väärien positiivisten luokittelujen osuus näyttää laskevan, kun n-grammin pituutta kasvatetaan, vaikkakin keskiarvot ovat molemmissa tapauksissa matalat.

Tarkasteltaessa eroja ryhmien välillä väärien negatiivisten luokittelujen osuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi 0.138324. Tämän vuoksi perushypoteesi säilyy, eikä ryhmien välillä ole muuttujan suhteen nähtävissä merkittävää eroa. Kun tarkastellaan ryhmien eroja luokittelun tarkkuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p = 0.015142$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Ryhmien välisten erojen tarkastelua jatkettiin Dunnin testin ja ryhmien keskiarvojen avulla. Ero ryhmien välillä on Dunnin testin mukaan tilastollisesti merkitsevä. Dunnin testin p-arvo on 0.0151. Keskiarvoksi luokittelun tarkkuudelle saadaan 0.957887, kun n-grammin pituus kaksi ja 0.966646, kun n-grammin pituus on kolme. Keskiarvoista huomataan, että luokittelun tarkkuus kasvaa, kun n-grammin pituutta kasvatetaan.

Kokonaisuudessa n-grammin pituuden vaikutuksessa luokittelijan suoriutumiseen jää suhteellisen selkeä kuva tehtyjen analyysien perusteella, vaikkakin väärien negatiivisten luo-

kitteluiden osuudessa ei havaittu pituuksien välillä merkittävää eroa. Kuitenkin väärien positiivisten luokitteluiden osuus laski ja luokittelun tarkkuus kohosi, kun n-grammin pituutta kasvatettiin. Johtopäätöksenä voidaan sanoa, että luokittelun suorituskyky paranee, kun n-grammin pituutta kasvatetaan kahdesta kolmeen.

4.1.3 N-grammien määrä

N-grammien määrän vaikutusta luokittelusta suoriutumiseen tutkittaessa tarkasteltiin kuutta ryhmää. Ryhmät poikkeavat toisistaan näytteistä valittujen edustavien n-grammien määrässä. Ryhmien n-grammien määrät ovat: 50, 100, 200, 400, 800 ja 1600. Jokaisen ryhmän koko oli 54 näytettä.

Tarkasteltaessa eroja ryhmien välillä väärien positiivisten luokittelujen osuuden suhteen saadaan Kruskal-Wallisin testin p-arvoksi 0.744016. Tämän vuoksi perushypoteesi säilyy, eikä ryhmien välillä ole muuttujan suhteen nähtävissä merkittävää eroa. Kun tarkastellaan ryhmien eroja väärien negatiivisten luokittelujen osuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p = 0.015396$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Dunnin testin mukaan erot ovat merkitseviä ryhmien 50 n-grammia ja 800 n-grammia, 50 n-grammia ja 1600 n-grammia, 100 n-grammia ja 800 n-grammia sekä 100 n-grammia ja 1600 n-grammia välillä. Dunnin testin p-arvojen ollessa järjestyksessään: 0.0028, 0.0039, 0.0243 ja 0.0315. Keskiarvojen ollessa näille ryhmille järjestyksessään: 0.450061, 0.432100, 0.363613, 0.353897, 0.290520 ja 0.265112. Keskiarvojen perusteella puolestaan näyttää, että väärien negatiivisten luokittelujen osuus laskee keskimäärin, kun käytettyjen n-grammien määrää kasvatetaan.

Tarkastellessa ryhmien eroja luokittelun tarkkuuden suhteen huomataan, että Kruskal-Wallisin testi saa p-arvokseen alle 0.05 ($p = 0.000033$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Ryhmien välisten erojen tarkastelua jatketaan Dunnin testin ja ryhmien keskiarvojen avulla. Dunnin testin mukaan erot merkitseviä ryhmien 50 n-grammia ja 400 n-grammia, 50 n-grammia ja 800 n-grammia, 50 n-grammia ja 1600 n-grammia, 100 n-grammia ja 400 n-grammia, 100 n-grammia ja 800

n-grammia, 100 n-grammia ja 1600 n-grammia, 200 n-grammia ja 800 n-grammia sekä 200 n-grammia ja 1600 n-grammia. P-arvojen ollessa järjestyksessään: 0.0091, 0.0004, $p < 0.0001$, 0.0366, 0.0026, 0.0002, 0.0353 ja 0.0053. Keskiarvot ovat puolestaan ryhmille järjestyksessään: 0.953411, 0.954882, 0.960636, 0.963793, 0.969820 ja 0.971057. Keskiarvojen perusteella puolestaan näyttää, että luokittelun tarkkuus kasvaa keskimäärin, kun käytettyjen n-grammien määrää kasvatetaan.

Kokonaisuudessa n-grammien määrän vaikutuksessa luokittelijan suoriutumiseen jää tehtyjen analyysien perusteella selkä kuva, vaikka väärien positiivisten luokitteluiden osuudessa ei havaittu pituuksien välillä merkittävää eroa. Kuitenkin väärien positiivisten luokitteluiden osuus laski ja luokittelun tarkkuus koheni aina, kun luokittelussa käytettyjen n-grammien määrää kasvatettiin. Tehtyjen analyysien perusteella näyttäisi, että luokittelijoiden suorituskyky parani suhteellisen tasaisesti, kun n-grammien määrää kasvatettiin 50 n-grammista aina 1600 n-grammiin aina kaksinkertaistaen määrä jokaisella kierroksella.

4.1.4 N-grammien valintaan käytetty menetelmä

N-grammien valintaan käytetyn menetelmän vaikutusta luokittelusta suoriutumiseen tutkittaessa tarkasteltiin kolmea ryhmää. Ensimmäisen ryhmässä n-grammit valittiin käyttäen Tf-idf-lukua. Toisen ryhmässä n-grammit puolestaan valittiin käyttäen Fisher-Score-lukua. Kun taas kolmannen ryhmässä n-grammit valittiin käyttäen Information Gain Ratio-lukua. Jokaisen ryhmän koko oli 108 näytettä.

Tarkasteltaessa eroja ryhmien välillä väärien positiivisten luokittelujen osuuden suhteen saadaan Kruskal-Wallisin testin p-arvoksi 0.378117. Tämän vuoksi perushypoteesi säilyy, eikä ryhmien välillä ole muuttujan suhteen nähtävissä merkittävää eroa. Kun taas tarkastellaan ryhmien eroja väärien negatiivisten luokittelujen osuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p = 0.000027$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Ero ryhmien Tf-Idf-luku ja Information Gain Ratio-luku sekä ryhmien Fisher Score-luku ja Information Gain Ratio-luku välillä ovat tilastollisesti merkitseviä. Dunnin testin p-arvojen ollessa $p < 0.0001$ ja 0.0088. Myös ryhmien Tf-Idf-luvun ja Fisher Score-luvun välisen eron tarkastelu saa p-arvokseen

0.0511. Tämän voidaan katsoa osoittavan, että ero myös näidenkin ryhmien välillä on Dunnin testin mukaan merkitsevä. Keskiarvoksi väärien negatiivisten luokittelujen osuuksille ryhmät saavat järjestyksessään: 0.487085, 0.361834, 0.228733. Keskiarvoista huomataan, että väärien negatiivisten luokittelujen osuus pienenee, kun n-grammien valintaan käytettävää menetelmää vaihdetaan Tf Idf-luvusta Fisher Score-lukuun ja, kun menetelmää vaihdetaan Fisher Score-luvusta Information Gain Ratio-lukuun.

Kun ryhmien välisiä eroja tarkastellaan luokittelun tarkkuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p < 0.000001$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Ero ryhmien Tf-idf-luku ja Fisher Score-luku, ryhmien Tf-idf-luku ja Information Gain Ratio-luku sekä ryhmien Fisher Score-luku ja Information Gain Ratio-luku välillä ovat tilastollisesti merkitseviä. Dunnin testin p-arvojen ollessa: 0.0018, 0.0016 ja $p < 0.0001$. Keskiarvoksi muuttujalle ryhmät saavat järjestyksessään: 0.946371, 0.963651 ja 0.976778. Keskiarvoista huomataan, että luokittelijoiden tarkkuus paranee keskimäärin, kun n-grammien valintaan käytettävää menetelmää vaihdetaan Tf-idf-luvusta Fisher Score-lukuun ja, kun menetelmää vaihdetaan Fisher Score-luvusta Information Gain Ratio-lukuun.

Kokonaisuudessa n-grammien valintaan käytetyn menetelmän vaikutuksessa luokittelijan suoriutumiseen jää suhteellisen selkeä kuva tehtyjen analyysien perusteella, vaikkakin väärien positiivisten luokitteluiden osuudessa ei havaittu menetelmien välillä merkitsevää eroa. Kuitenkin väärien negatiivisten luokittelujen osuus laski ja luokittelun tarkkuus kohosi, kun n-grammien valintaan käytetty menetelmä vaihdettiin ensin Tf-idf-luvusta Fisher Score-lukuun ja edelleen Fisher Score-luvusta Information Gain Ratio-lukuun. Johtopäätöksenä voidaan sanoa, että n-grammien valintaan käytetyllä menetelmällä on väliä luokittelijan suorituskyvyn kannalta ja että, huonoiten menetelmistä pärjäsi Tf Idf-luku ja parhaiten Information Gain Ratio-luku.

4.1.5 Käytetty koneoppimismenetelmä

Luokittelussa käytetyn koneoppimismenetelmän vaikutusta luokittelusta suoriutumiseen tutkittaessa tarkasteltiin kolmea ryhmää. Ensimmäisen ryhmässä käytetty menetelmä oli

CART-päätöspuu. Toisen ryhmässä käytetty menetelmä oli puolestaan keinotekoinen neuroverkko. Kolmannessa ryhmässä käytetty menetelmä oli geneettinen ohjelmointi. Jokaisen ryhmän koko oli 108 näytettä.

Tarkasteltaessa eroja ryhmien välillä väärien positiivisten luokittelujen osuudessa, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p < 0.000001$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa, joten ryhmien välisiä eroja tarkasteltiin Dunnin testin ja ryhmien keskiarvojen avulla. Erot ryhmien CART-päätöspuu ja keinotekoinen neuroverkko, CART-päätöspuu ja geneettinen ohjelmointi sekä keinotekoinen neuroverkko ja geneettinen ohjelmointi välillä ovat kaikki Dunnin testin mukaan merkitseviä. Testin p-arvojen ollessa järjestyksessä: $p < 0.0001$, $p < 0.0001$ ja 0.0003. Keskiarvon ollessa CART-päätöspuulle 0.013274, keinotekoiselle neuroverkolle 0.002571 ja geneettiselle ohjelmoinnille 0.013179. Keskiarvoista huomataan väärien positiivisten luokittelujen osuuden olevan keskimäärin varsin alhainen kaikille tutkimuksessa mukana olevilla tapauksille ja siten myös kaikille koneoppimismenetelmille. Varsinaista eroa ryhmien välillä on kuitenkin vaikea havaita osuuksien ollessa niin pieniä, vaikka Dunnin testin mukaan eroa olisikin olemassa.

Tarkasteltaessa eroja ryhmien välillä väärien negatiivisten luokittelujen suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p < 0.000001$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää. Erot ryhmien CART-päätöspuu ja keinotekoinen neuroverkko, CART-päätöspuu ja geneettinen ohjelmointi sekä keinotekoinen neuroverkko ja geneettinen ohjelmointi välillä ovat Dunnin testin mukaan merkitseviä. Testin p-arvojen ollessa järjestyksessä: $p < 0.0001$, $p < 0.0001$ ja 0.0005. Ryhmien keskiarvot väärien positiivisten luokittelujen osuuksille ovat järjestyksessään: 0.107985, 0.406197 ja 0.563470. Keskiarvoista huomataan, että väärien negatiivisten luokitteluiden osuus on keskimäärin pienin, kun luokitteluun käytetty koneoppimismenetelmä on CART-päätöspuu, ja toiseksi pienen, kun käytetty koneoppimismenetelmä on keinotekoinen neuroverkko ja suurin, kun käytetty koneoppimismenetelmä on geneettinen ohjelmointi.

Kun tarkastellaan koneoppimismenetelmien eroja luokittelun tarkkuuden suhteen, saadaan Kruskal-Wallisin testin p-arvoksi alle 0.05 ($p < 0.000001$). Tämän vuoksi hylätään hypoteesi, ettei ryhmien välillä ole muuttujan suhteen merkittävää eroa. Erot ryhmien CART-päätöspuu ja keinotekoinen neuroverkko, CART-päätöspuu ja geneettinen ohjelmointi sekä keinotekoinen neuroverkko ja geneettinen ohjelmointi välillä ovat Dunnin testin mukaan kaikki merkitseviä. Testin p-arvojen ollessa järjestyksessä: 0.0004, $p < 0.0001$ ja $p < 0.0001$. Keskiarvon ollessa CART-päätöspuulle 0.979906, keinotekoiselle neuroverkolle 0.963254 ja geneettiselle ohjelmoinnille 0.943639. Keskiarvoista huomataan, että luokittelun tarkkuus on keskimääräisesti suurin CART-päätöspuulle, toiseksi suurin keinotekoiselle neuroverkolle ja matalin geneettiselle ohjelmoinnille.

Kokonaisuudessa käytetyn koneoppimismenetelmän vaikutuksessa luokittelijan suoriutumiseen voidaan tehtyjen analyysien perusteella sanoa, että tutkittujen menetelmien välillä oli huomattavissa selkeää eroa. Tutkituista menetelmistä parhaiten kokonaisuutena suoriutui CART-päätöspuu ja huonoiten geneettinen ohjelmointi.

4.2 Tutkimusaineiston silmämääräinen tarkastelu

Tutkimuksessa käytetyssä aineistossa oli yhteensä 324 havaintoriviä. Aineistossa virheellisten positiivisten luokitteluiden osuus oli kokonaisuudessaan matala, suurimmaksi osaksi osuus oli 2 prosentin luokkaa. Parhaiten virheellisten positiivisten luokittelujen suhteen pärjänneet kokoonpanot olivat: kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 150/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli keinotekoinen neuroverkko; kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 100/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli keinotekoinen neuroverkko; ja kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 100/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli CART-päätöspuu. Kaikille näille kokoonpanoille väärien positiivisten luokittelujen osuus oli 0.

Aineistossa oli 30 näytettä, jossa virheellisten positiivisten luokittelujen osuus oli suurempi kuin 2 prosenttia. Näiden joukossa haitallisten ja ei-haitallisten näytteiden suhde oli suurimmaksi osaksi 150 / 1000 ja käytettyjen n-grammien pituus oli tässä joukossa suurimmaksi osaksi kaksi. Käytettyjen ominaispiirteiden määrät olivat joukossa melko tasaisesti jakautuneet. Joukossa ominaispiirteiden valintaa käytetty menetelmä oli puolestaan suurimmalta osin Tf-idf-luku ja käytetty koneoppimismenetelmä oli geneettinen ohjelmointi. Lisäksi aineistossa oli 20 havaintoriviä, jossa väärien positiivisten luokittelujen osuus oli 0 ja väärien negatiivisten luokittelujen aste oli 100 prosenttia. Näistä osuuksista on pääteltävissä, että tapauksissa luokittelijat luokittelivat kaikki havainnot negatiiviksi ja ovat näin käyttökelvottomia. Näistä suurimmassa osassa n-grammien valintaan käytettymenetelmä oli Tf-Idf-luku ja käytetty koneoppimismenetelmä oli joko keinotekoinen neuroverkko tai geneettinen ohjelmointi.

Virheellisten negatiivisten luokittelujen osuudet vaihtelivat aineistossa tasaisesti 0.03 ja 1 välillä. Parhaiten negatiivisten luokittelujen osuudet pärjänneet kokoonpanot olivat: kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 150/1000, n-grammin pituus oli 3, n-grammien määrä oli 400, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli CART-päätöspuu; kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 150/1000, n-grammin pituus oli 3, n-grammien määrä oli 100, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli CART-päätöspuu; ja kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 150/1000, n-grammin pituus oli 2, n-grammien määrä oli 800, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli CART-päätöspuu. Kaikille näille kokoonpanoille näillä väärien negatiivisten luokittelujen osuus oli alle 0.039 prosenttia.

Aineistosta on löydettävissä 90 havaintoriviä, joille väärien negatiivisten luokitteluiden osuus on alle 0.1. Näiden joukossa haitallisten ja ei-haitallisten näytteiden suhde oli suurimmaksi osaksi 150/1000. Tässä joukossa käytetty koneoppimismenetelmä saattoi olla

CART-päätöspuu, keinotekoinen neuroverkko tai CART-päätöspuu, mutta useimmiten kuitenkin CART-päätöspuu. Tässä joukossa n-grammin pituudet, ominaispituuksien määrät ja n-grammien valintaan käytetyt menetelmät jakautuivat tasaisesti. Lisäksi kuten jo aikaisemmin mainittiin, aineistossa oli 20 havaintoriviä, jossa väärien positiivisten luokittelujen osuus oli 0 ja väärien negatiivisten luokittelujen aste oli 100 prosenttia. Tällöin luokittelijat luokittelivat kaikki havainnot negatiiviksi ja ovat näin käyttökelvottomia.

Aineistossa tarkkuus oli yli 85 prosenttia kaikille havaintoriveille ja yli 90 prosenttia peräti 304 havaintoriville. Parhaiten tarkkuuden suhteen pärjänneet kokoonpanot olivat: kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 50/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli keinotekoinen neuroverkko; kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 100/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli keinotekoinen neuroverkko; ja kokoonpano, jossa haitallisten ja ei-haitallisten näytteiden suhde oli 100/1000, n-grammin pituus oli 3, n-grammien määrä oli 1600, n-grammien valintaan käytetty menetelmä oli Information Gain Ratio-luku ja käytetty koneoppimismenetelmä oli CART-päätöspuu. Näille kaikille kokoonpanoille tarkkuus on yli 99.32 prosenttia.

4.3 Yhteenveto tilastollisesta ja silmämääräisestä tarkastelusta

Tutkimusaineistolle tehty tilastollinen tarkastelu aloitettiin luokittelijan opetusaineistossa olevien haitallisten ja ei-haitallisten näytteiden suhteen vaikutuksesta luokittelijan tarkkuuteen. Tarkastelun tuloksena oli, että luokittelijan suorituskyky paranee, kun haitallisten näytteiden suhteellista osuutta kasvatetaan 5 prosentista 10 prosenttiin tai 5 prosentista 15 prosenttiin. Tutkimusaineistoa silmämääräisesti tarkastellessa huomattiin myös, että parhaiten luokittelusta väärien positiivisten luokittelujen osuuden suhteen, väärien negatiivisten luokittelujen osuuden suhteen sekä luokittelun tarkkuuden suhteen suoriutuneet luokittelijat olivat opetettu aineistoilla, joissa haitallisten ja ei-haitallisten näytteiden suhde oli joko 100/1000 tai 150 /1000. Kokoonpanojen joukossa oli kuitenkin enemmän 100/1000-suhteella opetettuja luokittelijoita.

N-grammin pituuden vaikutuksen tilastollisen tarkastelun tuloksena todettiin, että luokittelun suorituskyky paranee, kun n-grammin pituutta kasvatetaan kahdesta kolmeen. Aineiston silmämääräinen tarkastelu vahvisti tätä kuvaa. Parhaiten luokittelusta väärien positiivisten luokittelujen osuuden suhteen, väärien negatiivisten luokittelujen osuuden suhteen sekä luokittelun tarkkuuden suhteen suoriutuneet luokittelijat olivat, lukuun ottamatta yhtä, kaikki opetettu aineistoilla, joissa n-grammin pituus oli 3.

Koneoppimismenetelmä	**Väärien positiivisten luokittelujen osuus**	**Väärien negatiivisten luokittelujen osuus**	**Luokittelun tarkkuus**
Keinotekoinen neuroverkko	0	0.0977	0.9922
CART-päätöspuu	0.0070	0.0806	0.9882
Geneettinen ohjelmointi	0.0020	0.3199	0.9745

Taulukko 1. Information Gain Ratio-luvun avulla valituilla 1600 3-grammilla opetettujen luokittelijoiden keskimäärinen suorituskyky

N-grammien määrän vaikutuksen tilastollisen tarkastelun tuloksena puolestaan todettiin, että luokittelijoiden suorituskyky parani suhteellisen tasaisesti, kun n-grammien määrää kasvatettiin 50 n-grammista aina 1600 n-grammiin, aina kaksinkertaistaen määrä jokaisella kierroksella. Silmämääräinen tarkastelu tuki johtopäätöstä, vaikkakin ei niin suoraviivaisesti, kuten esimerkiksi n-grammin pituuden tapauksessa. Parhaiten luokittelusta väärien positiivisten luokittelujen osuuden suhteen ja luokittelun tarkkuuden suhteen suoriutuneet luokittelijat olivat kaikki opetettu aineistolla, jossa n-grammien määrä oli 1600. Kun taas parhaiten luokittelusta väärien negatiivisten luokittelujen osuuden suhteen opetettu aineistoilla, joissa n-grammien määrät olivat 100, 400 ja 800.

N-grammin pituus	N-grammien määrä	Koneoppimismenetelmä	Väärien positiivisten luokitteluiden osuus	Väärien negatiivisten luokitteluiden osuus	Luokittelun tarkkuus
2	100	Keinotekoinen neuroverkko	0.0047	0.5144	0.9508
2	100	CART-päätöspuu	0.0138	0.0955	0.9802
2	100	Geneettinen ohjelmointi	0.0089	0.8540	0.9203
3	100	Keinotekoinen neuroverkko	0.0017	0.5388	0.9519
3	100	CART-päätöspuu	0.0226	0.1367	0.9709
3	100	Geneettinen ohjelmointi	0.0127	0.4532	0.9553
2	400	Keinotekoinen neuroverkko	0.0037	0.3552	0.9667
2	400	CART-päätöspuu	0.0136	0.1070	0.9796
2	400	Geneettinen ohjelmointi	0.0106	0.7594	0.9294
3	400	Keinotekoinen neuroverkko	0.0022	0.3762	0.9648
3	400	CART-päätöspuu	0.0082	0.0723	0.9872
3	400	Geneettinen ohjelmointi	0.0126	0.4532	0.9553
2	800	Keinotekoinen neuroverkko	0.0020	0.3655	0.9675
2	800	CART-päätöspuu	0.0104	0.0883	0.9837
2	800	Geneettinen ohjelmointi	0.0170	0.5230	0.9450
3	800	Keinotekoinen neuroverkko	0.0019	0.2185	0.9837
3	800	CART-päätöspuu	0.0103	0.0731	0.9848
3	800	Geneettinen ohjelmointi	0.0103	0.4747	0.9543

Taulukko 2. Information Gain Ratio-luvun avulla valituilla 100, 400 ja 800 n-grammilla opetettujen luokittelijoiden suorituskyky

N-grammien valintaan käytetyn menetelmän vaikutuksen tilastollisen tarkastelun tuloksena puolestaan todettiin, että n-grammien valintaan käytetyllä menetelmällä on väliä luokittelijan suorituskyvyn kannalta. Lisäksi todettiin, että huonoiten menetelmistä pärjäsi Tf-idf-luku ja parhaiten Information Gain Ratio-luku. Aineiston silmämääräinen tarkastelu tuki

vahvasti tätä johtopäätöstä. Parhaiten luokittelusta väärien positiivisten luokittelujen osuuden suhteen, väärien negatiivisten luokittelujen osuuden suhteen sekä luokittelun tarkkuuden suhteen suoriutuneet luokittelijat olivat, lukuun ottamatta yhtä, kaikki opetettu aineistoilla, joissa käytetyt n-grammit valittiin käyttäen Information Gain Ratio-lukua.

Tutkimusaineistolle tehty tilastollinen tarkastelu päätettiin valitun koneoppimismenetelmän vaikutuksen tarkasteluun. Tarkastelun tuloksena oli, että tarkasteltujen koneoppimismenetelmien välillä oli huomattavissa selkeää eroa. Tutkituista menetelmistä parhaiten kokonaisuutena suoriutui CART-päätöspuu ja huonoiten geneettinen ohjelmointi. Aineiston silmämääräinen tarkastelu tuki tätä johtopäätöstä vain osittain. Parhaiten luokittelusta väärien positiivisten luokittelujen osuuden suhteen ja luokittelun tarkkuuden suhteen suoriutuneet luokittelijat käyttivät suurimmalta osin koneoppimismenetelmänä keinotekoista neuroverkkoa, vaikka joukossa oli myös luokittelijoita, jotka käyttivät CART-päätöspuuta. Parhaiten luokittelusta väärien negatiivisten luokittelujen osuuden suhteen suoriutuneet luokittelijat puolestaan käyttivät kaikki koneoppimismenetelmänä CART-päätöspuuta.

4.4 Tulosten reliabiliteetti ja validiteetti

Seuraavaksi arvioidaan tutkimuksessa käytetyn tutkimusaineiston, tutkimusmenetelmän ja tutkimuksessa saatujen tulosten reliabiliteettia ja validiteettia. Arvioinnin alkuun kerrataan lyhyesti tutkimuksessa tutkittavat asiat ja tutkimuksen tuloksista selviävät asiat. Tämän jälkeen jatketaan tutkimusaineiston ja tutkimusmenetelmän reliabiliteetin arvioinnilla. Arviointia jatketaan tutkimusaineiston ja tutkimusmenetelmän validiteetin arvioinnilla. Arviointi päättyy tulosten validiteetin arviointiin.

Tässä tutkimuksessa tutkittiin haitallisen JavaScript-koodin tunnistamista koneoppimismenetelmin ja erityisesti haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen, opettamiseen ja testaamiseen käytettyjen n-grammien pituuden, opettamiseen ja testaamiseen käytettyjen n-grammien määrään, opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävän menetelmän ja opettamiseen käytettävän koneoppimismenetelmän vaikutusta luokittelijan suorituskykyyn. Tutkimuk-

sessa vertailtiin eri parametrein opetettuja luokittelijoita suoriutumista JavaScript-tiedostojen luokittelusta väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen.

Tutkimuksen varsinaisena tutkimusaineistona toimi eri parametrein opetettujen luokittelijoiden luokittelukykyä mittaavat suureet: väärien positiivisten luokittelujen osuus, väärin negatiivisten luokittelujen osuus ja luokittelun tarkkuus. Tutkimuksessa etsittiin aineistosta eroja, joiden avulla voitiin tehdä johtopäätöksiä, siitä millä parametrien arvoilla opetetut luokittelijat selviytyvät parhaiten haitallisen JavaScript-koodin tunnistamisesta. Tutkimusmenetelmä puolestaan sisälsi kaksi vaihetta erojen löytämiseksi. Vaiheet olivat tilastollinen analyysi ja silmämääräinen analyysi.

Seuraavaksi pohditaan tutkimusaineiston ja tutkimusmenetelmän reliabiliteettia toistettavuuden ja luotettavuuden näkökulmasta. Tutkimuksessa käytetty tutkimusaineisto muodostettiin erikseen tätä tutkimusta varten. Aineiston muodostamiseen käytettiin tutkimusta varten kirjoitettua koodia, saatavilla olevia avoimen lähdekoodin piiriin kuuluvia ohjelmistopaketteja ja tutkimusta varten hankittuja haitallisia ja ei-haitallisia JavaScript-koodinäytteitä. Tutkimusaineiston uudelleen muodostaminen edellä mainittuja asioita käyttäen on mahdollista ja lisäksi suhteellisen helppoa.

Aineiston tilastolliseen analyysiin käytettiin yleisesti käytössä olevia tilastollisia menetelmiä. Menetelmien toteuttamiseen kirjoitettiin koodia ja käytettiin avoimen lähdekoodin piiriin kuuluvia ohjelmistopaketteja. Tilastollisen analyysin tulosten tulkinnassa puolestaan pyrittiin soveltamaan yleisesti käytettyjä käytäntöjä. Näin ollen, myöskin tutkimuksessa tehty tilastollinen analyysi on toistettavissa. Tutkimuksessa tehty silmämääräinen analyysi puolestaan perustui tutkijan henkilökohtaiseen näkemykseen aineistosta ja analyysissä etsittävistä asioista. Analyysissa pyrittiin kuitenkin löytämään asioita, joidenka tulkinnat olisivat mahdollisimman yksiselitteisiä. Analyysiprosessia ei kuitenkaan dokumentoitu riittävän tarkasti. Näin ollen, aineiston silmämääräisen analyysin toistaminen ei ole yhtä helppoa kuin tilastollisen analyysin. Tästäkin huolimatta tutkimusmenetelmä on yleisesti ottaen toistettava siten, että toistettaessa se tuottaa samankaltaisia tuloksia kuin tässä tutkimuksessa raportoitiin.

Tutkimusmenetelmän voidaan myös katsoa tutkivan ja vertailevan suhteellisen luotettavasti edellä mainittujen parametrien eri arvoilla opetettujen luokittelijoiden suoriutumista JavaScript-tiedostojen luokittelusta tutkimuksen rajoitteet huomioon ottaen. Huomioitavina rajoitteina ovat: pienet näytekoot, pienet haitallisten näytteiden suhteelliset osuudet opetus- ja testiaineistoissa sekä lyhyet n-grammin pituudet. Muilta osin tutkimuksessa käytettiin tutkimusmenetelmää, joka on hyvin samankaltainen suhteessa muihin haitallisen ohjelmakoodin tunnistamista koneoppimismenetelmin tutkivien tutkimusten tutkimusmenetelmiin. Samankaltaisuus on nähtävissä tutkimuksessa käytetyissä haitallisissa tiedostoissa, tarkasteltavista muuttujista (parametreistä), luokittelijoiden opettamiseen ja testaamiseen käytetystä k-fold-menetelmässä, luokittelijoiden vertaamiseen käytetyistä tunnusluvuista sekä vertailemiseen käytetyissä tilastollisissa menetelmissä. Tästä huomataan myös, että kaikki tutkimusmenetelmässä pyrkii eri parametrein opetettujen luokittelijoiden suoriutumisen keskinäiseen vertailemiseen. Täten tutkimusmenetelmän voidaan katsoa myös mittaavan juuri sitä ilmiötä, jota tutkimuksessa haluttiinkin tutkia eli tutkimusmenetelmän voidaan katsoa olevan validi.

Pohditaan vielä lopuksi oikeuttavatko tutkimuksessa käytetty tutkimusaineisto, tutkimusmenetelmä ja tulokset tutkimuskysymyksiin esitetyt vastaukset eli ovatko tutkimuksen tulokset valideja. Saatujen tulosten perusteella tutkimuksessa esitetään että, luokittelijan opetuksessa ja testaamisessa käytettävällä opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteella, opettamiseen ja testaamiseen käytettävällä n-grammin pituudella, opettamiseen ja testaamiseen käytettävällä n-grammien määrällä, opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävällä menetelmällä ja opettamiseen käytettävällä koneoppimismenetelmällä on vaikutusta haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan suorituskykyyn.

Tutkimuksessa käytetyn tutkimusaineiston muodostamisessa pyrittiin huomioimaan seuraavat asiat: käytetyt haitalliset näytteet olivat oikeasti haitallisia, käytetyt ei-haitalliset näytteet olivat oikeasti ei-haitallisia sekä että käytetyt n-grammien valintamenetelmät ja käytetyt koneoppimismenetelmät toimivat oikein. Haitallisten tiedostojen haitallisuus pyrittiin varmistamaan käyttämällä muissakin tutkimuksissa käytettyjä haitallisia tiedostoja. Ei-haitallisten tiedostojen haitattomuus puolestaan pyrittiin varmistamaan käyttämällä suo-

situilla web-sivuilla käytössä olevia JavaScript-tiedostoja. N-grammien valintaan käytetyille menetelmille taas kirjoitettiin tutkimusta varten omat toteutukset, joidenka toimivuus pyrittiin varmistamaan kirjoittamalla toteutuksille testit. Tutkimuksessa käytettyjen koneoppimismenetelmien toimivuus puolestaan pyrittiin varmistamaan käyttämällä tunnettujen avoimen lähdekoodin piirin kuuluvien kirjastojen toteutuksia tutkimukseen valituista menetelmistä ja seuraamalla näiden dokumentaatioista löytyviä esimerkkejä menetelmien käytöstä.

Kuitenkin on aina mahdollista, että tutkimusaineiston muodostamisessa on sattunut virheitä, jotka vaikuttavat tulosten validiteettiin. Tällainen virhe voi esimerkiksi liittyä käytettyihin tiedostoluokkiin siten, että molemmissa tiedostoluokissa on ollut mukana toisen luokan edustajia. N-grammien valintaan käytetyt menetelmät puolestaan toteutettiin tutkimusta varten erikseen. Toteutuksien testaamisesta huolimatta on aina mahdollisuus, että esimerkiksi menetelmien matemaattisia kaavoja on toteutuksen yhteydessä luettu väärin. Myös käytettyjen koneoppimismenetelmien toimivuuteen, erityisesti geneettisen ohjelmoinnin toimivuuteen, liittyy kysymyksiä, jotka realisoituessaan heikentävät tulosten validiteettia. Geneettinen ohjelmointi tuotti välillä luokittelijoita, jotka suoriutuivat muiden menetelmien tuottamiin luokittelijoiden nähden erittäin heikosti. Tämän vuoksi onkin perustultua epäillä, osattiinko valittua geneettisen ohjelmoinnin toteutukseen käytettyä ohjelmistokirjastoa käyttää tutkimuksessa oikein. Lisäksi yleisesti koneoppimismenetelmien toimivuuteen on saattanut vaikuttaa se, että tutkimuksessa koneoppimismenetelmien opettamisessa ja testaamisessa käytetyt tiedostomäärät olivat myös vähäisiä. Kaikilla edellä mainituilla asioilla on tulosten validiteettia heikentävä vaikutus.

Tutkimusaineistoon sovellettavan tutkimusmenetelmän suunnittelussa puolestaan pyrittiin yhdistelemään menetelmiä, jotka tukisivat toisiaan ja tuottaisivat tutkimuskysymyksiin vastaamiseen kannalta mahdollisimman helppotulkintaisia tuloksia. Tutkimuksessa päädyttiin käyttämään tilastollisia menetelmiä ja tukemaan näillä saatuja tuloksia aineiston silmämääräisellä analyysillä. Tilastollisiksi menetelmiksi valikoituivat Kruskal-Wallisin testi, Dunnin testi ja keskiarvovertailu. Kruskal-Wallisin testi ja Dunnin testi valittiin tutkimukseen, koska ne ovat yleisesti käytettyjä menetelmiä vastaavanlaisissa ryhmien vertailuissa. Keskiarvovertailu otettiin tutkimukseen mukaan, jotta eroavaisuuksien mittakaavoista saa-

taisiin selkeämpi kuva. Silmämääräisellä analyysilla pyrittiin puolestaan tarjoamaan vahvistusta tilastollisilla menetelmillä saaduille tuloksille ja löytämään aineistossa olevia ääripäitä.

Myös tutkimusmenetelmään liittyy asioita, jotka heikentävät menetelmällä tuotettujen tulosten validiteettia. Ensimmäiseksi joissakin tehdyissä tilastollisissa testeissä Kruskal-Wallisin testin ja Dunnin testin käyttöön liittyviä oletuksia relaksoitiin, esimerkiksi vertailtavien ryhmien määrän ja koon suhteen, jotta menetelmiä pystyttiin käyttämään kaikissa halutuissa tilastollisissa analyyseissä. Myös keskiarvovertailun käyttö Kruskal-Wallisin testin ja Dunnin testin yhteydessä voidaan nähdä kyseenalaisena, koska mainitut testit eivät perustu keskiarvojen vertailuun. Lisäksi silmämääräisesti tehtävään analyysin liittyy aina inhimillisenvirheen mahdollisuus ja tosiasia, että asioiden tulkinta on aina jossakin määrin riippuvaista tulkitsijasta. Kaikki tässä kappaleessa luetellut asiat voidaan myös nähdä tulosten validiteettia heikentävinä asioina.

Tutkimuksessa saadut tulokset ovat edellä esitetyistä validiteettia heikentävistä asioista riippumatta suhteellisen selkeitä ja helppo tulkintaisia. Kuten edellä mainittiin, tutkimusmenetelmän valinnassa pyrittiin tuottamaan tuloksia, joidenka avulla tutkimuksen tutkimuskysymyksiin vastaaminen voitaisiin tehdä mahdollisimman suoraviivaisesti rajatuin tulkinnan mahdollisuuksin. Tutkimuksessa onnistuttiin tässä kohtuullisen hyvin. Tuloksista on helposti mahdollista löytää vastaus luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen, opettamiseen ja testaamiseen käytettyjen n-grammin pituuden, opettamiseen ja testaamiseen käytettyjen n-grammien määrään, opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävän menetelmän ja opettamiseen käytettävän koneoppimismenetelmän vaikutuksesta luokittelijan kykyyn tunnistaa haitalliset JavaScript-tiedostot. Täten voidaan tutkimusaineiston luontiin ja käytettyyn tutkimusmenetelmään liittyvistä kysymyksistä huolimatta kuitenkin katsoa, että muodostettu tutkimusaineisto ja siitä tutkimusmenetelmällä saadut tulokset oikeuttavat tutkimuskysymyksiin esitettävät vastaukset.

4.5 Tulosten vertaaminen aikaisempiin tutkimuksiin

Seuraavaksi vertaillaan tutkimuksessa saatuja tuloksia aikaisemmissa haitallisen ohjelmakoodin tunnistamista koneoppimismenetelmin käsitelleissä tutkimuksissa saatuihin tuloksiin. Vertailu aloitetaan vertailemalla tässä tutkimuksessa ja aikaisemmissa tutkimuksissa tehtyjä havaintoja luokittelijoiden opetusaineistossa olevien haitallisten ja ei-haitallisten koodinäytteiden suhteen, n-grammin pituuden, n-grammien määrän, ominaispiirteiden valintaan käytetyn menetelmän ja käytetyn koneoppimismenetelmän vaikutuksesta luokittelusta suoriutumiseen. Tämän jälkeen vertaillaan aikaisemmissa tutkimuksissa ja tässä tutkimuksessa saatuja tuloksia luokittelijoiden tekemien väärien positiivisten luokitteluiden osuuden, väärien negatiivisten luokitteluiden osuuden ja luokittelijoiden tarkkuuden suhteen.

Luokittelijoiden opettamisessa käytetyn aineiston sisältämien haitallisten ja ei-haitallisten näytteiden suhteen vaikutusta luokittelijan suoriutumiseen ovat tutkineet aiemmin ainakin Moskvitch ja kumppanit (1., 2., 3. 2008). Kaikissa näissä tutkimuksissa tutkittiin haitallisen JavaScript-koodin tunnistamiseen sijaan haitallisten Windows-suoritettavien tunnistamista koneoppimismenetelmin. Tästä huolimatta Moskovitchin ja kumppaneiden saamat tulokset ovat mielenkiintoisia, sillä heidän saamansa tulokset ovat tämän tutkimuksen tulosten kanssa hyvin samankaltaisia.

Tutkiessaan kronologisen opetuksen ja arvioinnin vaikutusta koneoppimismenetelmin opetettujen luokittelijoiden kykyyn tunnistaa haitallista ohjelmakoodia Moskovitch ja kumppanit (1. 2008) huomasivat, että luokittelijat, jotka opetettiin 16 prosenttia haitallista koodia sisältävällä aineistolla, pärjäsivät yleisesti 50 prosenttia haitallista koodia sisältävillä aineistoilla opetettuja luokittelijoita paremmin. Kun Moskovitch ja kumppanit (2. 2008) tutkivat varsinaisesti opetusaineistossa olevien haitallisten näytteiden määrän vaikutusta luokittelijoiden suorituskykyyn. Moskovitch ja kumppanit saivat vahvistusta edellisen tutkimuksen tulokselle. Moskovitch ja kumppanit huomasivat, että paras tarkkuus luokittelijoille saavutettiin opetusaineistolla, jossa haitallisten näytteiden osuus oli 16.7 prosenttia.

Samana vuonna Moskovitch ja kumppanit (3. 2008) tutkivat myös ohjelmankoodin operaatiokoodista eristettyjen ominaispiirteiden käyttämistä haittaohjelmien tunnistamiseen tar-

koitettujen luokittelijoiden opettamisessa. Samassa yhteydessä he tutkivat myös haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen vaikutusta luokitteluun. Heidän käyttämät haitallisten näytteiden osuudet olivat: 5, 10, 15, 30 ja 50 prosenttia opetusaineistoista. Myös tämän tutkimuksen yhteydessä tutkijat huomasivat, että parhaat tulokset saavutettiin luokittelijoilla, joiden opettamiseen ja testaamiseen käytetyissä aineistoissa haitallisen ohjelmakoodin osuus opetusaineistossa oli 10 tai 15 prosenttia.

Moskovitch ja kumppaneiden tutkimuksissaan (1., 2., 3. 2008) tekemät havainnot luokittelijoiden opettamiseen ja testaamiseen käytetyn aineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen vaikutuksesta luokittelijoiden suoriutumiseen ovat samankaltaisia, kun tässä tutkimuksessa tehdyt havainnot. Myös tässä tutkimuksessa huomattiin, että luokittelijoiden suorituskyky paranee, kun haitallisten näytteiden osuutta luokittelijoiden opetusaineistossa kasvatetaan 5 prosentista 10 prosenttiin tai 5 prosentista 15 prosenttiin. Tutkimusten tulokset ei ole keskenään täysin verrattavissa, sillä tässä tutkimuksessa ei tutkittu aineistoja, joissa haitallisten näytteiden osuus opetus- ja testausaineistosta olisi ollut yli 15 prosenttia. Lisäksi eroina edellä mainittujen tutkimusten ja tämän tutkimuksen välillä ovat muuan muassa: millaisia haitallisia tiedostoja haluttiin tunnistaa, tutkimuksissa käytetyt ominaispiirteet, tutkimuksissa ominaispiirteiden valintaan käytetyt menetelmät ja tutkimuksissa käytetyt koneoppimismenetelmät.

Luokittelijoiden opettamiseen ja testaamiseen käytettyjen n-grammien pituuden vaikutusta luokittelijoiden suoriutumiseen ovat puolestaan aiemmin tutkineet ainakin Moskovitch ja kumppanit (1. 2008). Tutkiessaan ohjelmakoodin operaatiokoodista eristettyjen ominaispiirteiden käyttämistä haittaohjelmien tunnistamiseen tarkoitettujen luokittelijoiden opettamisessa tulivat tutkijat tarkastelleeksi myös operaatiokoodista muodostettujen n-grammien pituuden vaikutusta luokittelijan suorituskykyyn. Tutkijat huomasivat, että yleisesti parhaat tulokset saavutettiin n-grammeilla, joiden pituus oli kaksi. Tämä tulos poikkesi hieman tämän tutkimuksen havainnosta, jossa luokittelun suorituskyky parani, kun n-grammin pituutta kasvatettiin kahdesta kolmeen. Kyseisen tutkimuksen ja tämän tutkimuksen välillä on kuitenkin selviä eroja, kuten jo edellä mainittiin.

Luokittelijoiden opettamiseen ja testaamiseen käytettyjen n-grammien määrän vaikutusta luokittelijoiden suoriutumiseen ovat tutkineet ainakin Kolter ja Maloof (2004) sekä Le ja kumppanit (2014). Kolter ja Maloof (2004) tutkivat n-grammien käyttöä luokittelijoiden opettamiseen käytettyinä ominaispiirteinä. Tutkijat keskittyivät erityisesti siihen mikä määrä n-grammeja ominaispiirteinä tuottaa parhaat tulokset luokittelijoiden opettamisessa. He eristivät analysoitavista ohjelmakoodi näytteistä n-grammeja, joista he sitten valitsivat 10, 20, . . ., 100, 200, . . ., 1000, 2000, . . ., 10 000 edustavinta n-grammia luokittelijoiden opettamiseen. Kolter ja Maloof huomasivat, että yleisesti ottaen parhaat tulokset saavutettiin käyttämällä 500 n-grammia luokittelijoiden opettamiseen ja testaamiseen. Le ja kumppanit (2014) puolestaan huomasivat tutkiessaan haittaohjelmien tunnistamista geneettistä ohjelmointia käyttäen, että parhaiten suoriutui 200 n-grammilla opetettu luokittelija. Le ja kumppanit käänsivät ensin analysoitavat tiedostot heksadesimaaliesitysmuotoon, jonka jälkeen niistä muodostettiin 4-grammeja. Näistä varsinaisiksi ominaispiirteiksi mukaan otettiin 50, 100 ja 200 4-grammia eri menetelmiä käyttäen.

Tämän tutkimuksen tulokset poikkesivat Kolter ja Maloofin (2004) sekä Le ja kumppaneiden (2014) tutkimuksissa tekemistä havainnoista. Tässä tutkimuksissa havaittiin, että edellisiä tutkimuksia suuremmat määrät n-grammeja luokittelijoiden opettamissa ja testaamisessa vaikuttivat positiivisesti luokittelijan suorituskykyyn. Tämän tutkimuksen, Kolter ja Maloofin (2008) tutkimuksen sekä Le ja kumppaneiden (2014) tutkimuksen havainnot eivät ole keskenään täysin verrattavissa, sillä edellä mainittujen tutkimusten ja tämän tutkimuksen tutkimusasetelmissa on nähtävissä selkeitä eroja. Eroina ovat muuan muassa: millaisia haitallisia tiedostoja tutkimuksissa haluttiin tunnistaa, tutkimuksissa käytetyt ominaispiirteet, tutkimuksissa ominaispiirteiden valintaan käytetyt menetelmät ja tutkimuksissa käytetyt koneoppimismenetelmät.

Luokittelijoiden opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettyjen menetelmien vaikutusta luokittelijoiden suoriutumiseen ovat puolestaan tutkineet ainakin Moskovitch ja kumppanit (2. 3. 2008) sekä Le ja kumppanit (2014). Moskovitch ja kumppanit (2. 2008) huomasivat tutkiessaan käytettyjen ominaispiirteiden ja opetusaineistossa olevien haitallisten näytteiden määrän vaikutusta luokittelijoiden suorituskykyyn sekä tutkiessaan ohjelmankoodin operaatiokoodista eristettyjen ominaispiirteiden käyttämistä

haittaohjelmien tunnistamiseen tarkoitettujen luokittelijoiden opettamisessa (1. 2008), että dokumenttifrekvessin ja Fisher Score-luvun avulla valituilla ominaispiirteillä opetetut ja testatut luokittelijat suoriutuivat luokittelusta parhaiten. Molemmissa näissä tutkimuksissa valintaan käytetyt menetelmät olivat dokumenttifrekvenssi, Information Gain Ratio-luku ja Fisher Score-luku. Le ja kumppanit (2014) puolestaan havaitsivat, että Fisher Score-luvun avulla valitut ominaispiirteet toimivat parhaiten geneettisen ohjelmoinnin tapauksessa. Kyseisessä tutkimuksessa valintaan käytetyt menetelmät olivat termifrekvenssi, dokumenttifrekvenssi ja Fisherin Score-luku.

Moskovitchin ja kumppaneiden (2., 3. 2008) sekä Le ja kumppaneiden tekemät havainnot poikkesivat tässä tutkimuksessa tehdyistä havainnoista. Näissä tutkimuksissa parhaat tulokset saavuttiin joko dokumenttifrekvenssin tai Fisher Score-luvun avulla. Kun taas tässä tutkimuksessa Tf Ifd-luvun avulla valituilla ominaispiirteillä opetetut ja testatut luokittelijat suoriutuivat luokittelusta huonoiten. Kun taas parhaiten suoriutuivat luokittelijat, joidenka opettaminen ja testaaminen tapahtui Information Gain Ratio-luvun avulla valituilla n-grammeilla. Tämän tutkimuksen, Moskovitchin ja kumppaneiden (2., 3. 2008) tutkimusten sekä Le ja kumppaneiden (2014) tutkimuksen tulokset eivät ole myöskään keskenään täysin verrattavissa, sillä tutkimusten asetelmissa on nähtävissä selkeitä eroja. Eroina ovat muuan muassa: millaisia haitallisia tiedostoja tutkimuksissa haluttiin tunnistaa, tutkimuksissa käytetyt ominaispiirteet, tutkimuksissa ominaispiirteiden valintaan käytetyt menetelmät ja tutkimuksissa käytetyt koneoppimismenetelmät.

Käytetyn koneoppimismenetelmän vaikutusta haitallisen ohjelmakoodin tunnistamiseen tarkoitettujen luokittelijoiden suoriutumiseen ovat tutkineet ainakin Kolter ja Maloof (2004) ja Moskovitch ja kumppanit (2. 2008). Kolter ja Maloof (2004) huomasivat tutkiessaan n-grammien käyttöä luokittelijoiden opettamiseen käytettyinä ominaispiirteinä, että parhaat tulokset haitallisen ohjelmakoodin tunnistamisessa saavutettiin tehostetulla J48-päätöspuulla. Tutkimuksessa käytetyt koneoppimismenetelmät olivat tukivektorikone, tehostettu tukivektorikone, Naive Bayes, J48-päätöspuu ja tehostettu J48-päätöspuu. Moskovitch ja kumppanit (2. 2008) puolestaan havaitsivat tutkiessaan käytettyjen ominaispiirteiden ja opetusaineistossa olevien haitallisten näytteiden määrän vaikutusta luokittelijoiden suorituskykyyn, että tutkimuksessa käytetyistä koneoppimismenetelmistä yleisesti parhai-

ten pärjäsivät neuroverkot ja päätöspuut. Tutkimuksessa käytetyt koneoppimismenetelmät olivat keinotekoinen neuroverkko, päätöspuu, Naive Bayes ja tukivektorikone.

Kolter ja Maloof (2004) sekä Moskovitchin ja kumppaneiden (2. 2008) tekemät huomiot ovat samanlaisia, kun tässä tutkimuksessa tehdyt havainnot. Kolter ja Maloof sekä Moskovitchin ja kumppanit huomaisivat, että päätöspuut ja keinotekoiset neuroverkot pärjäsivät muita käytettyjä koneoppimismenetelmiä paremmin haitallisen ohjelmakoodin tunnistamisessa. Tässäkin tutkimuksessa havaittiin, että tutkituista menetelmistä parhaiten kokonaisuutena suoriutui CART-päätöspuu. Tämän lisäksi tutkimuksessa havaittiin, että myös käyttämällä keinotekoisia neuroverkkoja on mahdollista luoda suorituskykyisiä luokittelijoita. Tämän tutkimuksen, Kolter ja Maloofin (2004) tutkimuksen sekä Moskovitchin ja kumppaneiden (2. 2008) tutkimuksen tulokset eivät ole kuitenkaan keskenään täysin verrattavissa, sillä tutkimusten asetelmissa on myös nähtävissä selkeitä eroja. Eroina ovat muuan muassa: millaisia haitallisia tiedostoja haluttiin tunnistaa, tutkimuksissa käytetyt ominaispiirteet, tutkimuksissa ominaispiirteiden valintaan käytetyt menetelmät ja tutkimuksissa käytetyt koneoppimismenetelmät.

Lopuksi vertaillaan tässä tutkimuksessa ja aikaisemmissa tutkimuksissa saatuja tuloksia luokittelijoiden väärien positiivisten luokitteluiden osuuden, väärien negatiivisten luokitteluiden osuuden ja luokittelijoiden tarkkuuden suhteen. Ensin vertaillaan tämän tutkimuksen tuloksia yleisesti haitallisen ohjelmakoodin tunnistamista tutkivissa tutkimuksissa saatuihin tuloksiin. Tämän jälkeen keskitytään vertaamaan tutkimuksen tuloksia haitallisen JavaScript-koodin tunnistamiseen koneoppimismenetelmin keskittyvissä tutkimuksissa saatuihin tuloksiin.

Tässä tutkimuksessa onnistuttiin opettamaan useita luokittelijoita, joilla saavutettu väärien positiivisten luokittelujen osuus tutkimusaineistolla testattaessa oli 0 prosenttia. Tutkimuksessa onnistuttiin myös opettamaan useita luokittelijoita, joilla saavutettu virheellisesti luokiteltujen negatiivisten havaintojen osuus oli alle 4 prosenttia. Paras tutkimuksessa saavutettu väärien negatiivisten luokittelujen osuus oli 2.567 prosenttia. Lisäksi tutkimuksessa onnistuttiin opettamaan muutamia luokittelijoita, joidenka luokittelun tarkkuus ylitti 99.3 prosenttia. Korkein saavutettu luokittelijan tarkkuus tutkimuksessa oli 99.36 prosenttia.

Tutkimuksessa kaikissa väärien positiivisten luokitteluiden osuuden, väärien negatiivisten luokitteluiden osuuden sekä tarkkuuden suhteen parhaiten suoriutuneissa luokittelijoissa käytetty koneoppimismenetelmä oli joko CART-päätöspuu tai keinotekoinen neuroverkko.

Schultz ja kumppanit (2001) vertailivat erään signeeraukseen perustuvan tunnistamismenetelmän ja erilaisten koneoppimiseen perustuvien menetelmien kykyä tunnistaa ennen näkemätöntä haitallista ohjelmakoodia. Heidän käyttämänsä koneoppimismenetelmät olivat RIPPER, Naive Bayes ja Multi-Naive Bayes. Korkein tutkimuksessa saavutettu tarkkuus luokittelulle oli 97.11 prosenttia. Tämä saavutettiin käyttämällä Multi-Naive Bayes-menetelmää. Ahmed ja kumppanit (2009) puolestaan tutkivat spatiaalisten ja temporaalisten ominaispiirteiden käyttämistä haittaohjelmien tunnistamiseen tarkoitettujen luokittelijoiden opettamiseen. Tutkijoiden käyttämät koneoppimismenetelmät olivat k-lähin naapuri, J48-päätöspuu, Naive Bayes, RIPPER ja Sequential Minimal Optimization. Tutkimuksessa onnistuttiin saavuttamaan 96.3 prosentin keskiarvo luokittelun tarkkuudessa, kun luokittelijoiden opettamiseen käytettiin spatiaalisten ja temporaalisten ominaispiirteiden yhdistelmää. Santos ja kumppanit (2011) taas tutkivat haittaohjelmien tunnistamista käyttäen operaatiokoodisekvensseistä johdettuja ominaispiirteitä ja yhden luokan tukivektorikonetta koneoppimismenetelmänä. Tutkimuksessa saavutettu luokittelun tarkkuus oli 87.46 prosenttia.

Amalina ja kumppanit (2014) tutkivat Android-käyttöjärjestelmille suunnattujen haittaohjelmien tunnistamista koneoppimismenetelmin. Luokittelijoiden opettamiseen käytetyt ominaispiirteet eritettiin tutkimuksessa ohjelmien suoritusaikana verkkoresurssien käytöstä kerätystä informaatiosta. Käytettyinä koneoppimismenetelminä olivat J48-päätöspuu, Bayes-verkko, neuroverkko, k-lähin naapuri ja Random Forrest. Parhaiten tutkimuksessa käytetyistä menetelmistä menestyi J48-päätöspuu, joka saavutti keskimäärin 0.10 prosentin asteen väärissä positiivisissa luokitteluissa ja 99.99 prosentin tarkkuuden. Hansen ja kumppanit (2016) puolestaan tutkivat uusien ennen näkemättömien haittaohjelmien havaitsemista sekä luokittelua eri haittaohjelmaperheisiin käyttäen dynaamista analyysia ja koneoppimismenetelmiä. Tutkijat käyttivät haittaohjelmien tunnistamiseen ja luokitteluun Random Forest-menetelmää. Tutkimuksessa onnistuttiin saavuttamaan keskimääriin 0.099 prosentin aste väärissä positiivisissa luokitteluissa.

Tässä tutkimuksessa saadut tulokset ja yleisesti koneoppimismenetelmien avulla haitallisen ohjelmakoodin tunnistamista tutkivien tutkimusten tulokset ovat siis samankaltaisia, erityisesti luokittelun tarkkuuden suhteen. Tässä tutkimuksessa paras saavutettu luokittelijan tarkkuus oli 99.36 prosenttia. Kun taas edellä läpikäydyissä tutkimuksissa luokittelijoiden tarkkuudet olivat välillä 87.46 ja 99.9 prosenttia. Myöskään näiden tulosten vertaamien ei ole täysin mielekästä, koska tutkimusten tutkimusasetelmat ovat useassa suhteessa hyvin erilaisia. Tuloksia vertailtiin kuitenkin, jotta nähtäisiin, miten tässä tutkimuksessa saadut tulokset ovat linjassa muihin haitallisen ohjelmakoodin tunnistamista koneoppimismenetelmin tutkineisiin tutkimuksiin.

Tutkimus	**Tiedostotyyppi**	**Väärien positiivisten luokitteluiden osuus**	**Väärien negatiivisten luokitteluiden osuus**	**Luokittelun tarkkuus**
Tuovinen (2018)	JavaScript	0	2.567	99.36
Schultz ja kumppanit (2001)	Muu	-	-	97.11
Santos ja kumppanit (2011)	Muu	-	-	-
Amalina ja kumppanit (2014)	Muu	0.10	-	99.99
Hansen ja kumppanit (2016)	Muu	0.099	-	-
Curtsinger ja kumppanit (2011)	JavaScript	0.00003	-	-
Cova ja kumppanit (2010)	JavaScript	-	0.25	-
Ravi ja kumppanit (2014)	JavaScript	0.72	9.5	-
Wang ja kumppanit (2015)	JavaScript	0.2123	0.8293	99.5322
Patil ja Patil (2017)	JavaScript	0.0001	0.0001	99.9

Taulukko 3. Haitallisen ohjelmakoodin tunnistamista koneoppimismenetelmin tutkivien tutkimusten tuloksia

Vertaillaan lopuksi tämän tutkimuksen tuloksia muihin haitallisen JavaScript-koodin tunnistamiseen koneoppimismenetelmin keskittyviin tutkimuksiin. Likarish ja Jo (2009) tutkivat haitallisen web-selaimissa suoritettavan JavaScript-koodin tunnistamista koneoppimismenetelmiä käyttäen. Tutkimuksessa käytetyt koneoppimismenetelmät olivat Naive Bayes, ADTree, tukivektorikone ja RIPPER. Tutkimuksessa onnistuttiin parhaimmillaan

luokittelemaan haitallinen ohjelmakoodi haitalliseksi noin 90 prosentissa tapauksista ja ei-haitallinen ohjelmakoodi ei-haitalliseksi noin 99,7 prosentissa tapauksissa. Curtsinger ja kumppanit (2011) puolestaan kehittivät Naive Bayes-menetelmään pohjaavan luokittelijan nimeltä Zozzle. Luokittelija saa syötteinä JavaScript-koodin abstraktista syntaksipuusta johdettuja ominaispiirteitä. Luokittelijalla saavutettu väärien positiivisten luokitteluiden osuus oli 0.00003 prosenttia. Cova ja kumppanit (2010) puolestaan kehittivät luokittelijan nimeltä Jsand, jonka toiminta perustuu koodin suorittamisen emulointiin ja anomalioiden tunnistamiseen. Menetelmässä käytetty koneoppimismenetelmä on myöskin Naive Bayes. Luokittelijan avulla saavutettu virheellisten negatiivisten luokitteluiden osuus oli 0.25 prosenttia.

Ravi ja kumppanit (2014) puolestaan kehittivät haitallisten web-sivujen tunnistamiseen erikoistuneen luokittelijan nimeltä JSGuard. Luokittelija käyttää sivujen analysointiin sekä staattista että dynaamista analyysiä. Kyseisen luokittelijan avulla saavutetut väärien positiivisten luokitteluiden ja väärien negatiivisten luokitteluiden osuudet olivat 0.72 prosenttia ja 9.5 prosenttia. Wang ja kumppanit (2015) puolestaan käyttivät eri koneoppimismenetelmiä haitallisen JavaScript-ohjelmakoodin havaitsemiseen ja luokitteluun 8 eri luokkaan haitallisessa tiedostossa käytetyn hyökkäysvektorin mukaan. Luokittelussa käytetyt koneoppimismenetelmät olivat ADTree, Random Forest, J48-päätöspuu ja Naive Bayes. Parhaat tulokset tutkijat saivat käyttäen luokitteluun Random Forest-menetelmään, jolloin saavutetut väärien positiivisten luokitteluiden osuus, väärien negatiivisten luokitteluiden osuus ja luokittelun tarkkuus olivat 0.2123 prosenttia, 0.8293 prosenttia ja 99.5322 prosenttia.

Myös Patil ja Patil (2017) tutkivat haitallisen JavaScript-ohjelmakoodin tunnistamista koneoppimismenetelmin. Tutkimuksessa käytetyt koneoppimismenetelmät olivat Naive Bayes, J48-päätöspuu, Random Forrest, tukivektorikone, AdaBoost, REP-puu ja AD-puu. Yleisesti ottaen kaikki tutkimuksen luokittelijat pystyivät tunnistamaan haitalliset näytteet tarkasti. Pelkästään kirjallisuudesta ennestään tunnetuilla ominaispiirteillä opetetut luokittelijat saavuttivat 0.001-0.063 prosentin osuuden väärissä positiivisissa luokitteluissa, 0.0001-0.009 prosentin osuuden väärissä negatiivisissa luokitteluissa ja 95.5-99.9 prosentin tarkkuuden. Kun ominaispiirteisiin lisättiin mukaan tutkijoiden määrittelemät uudet omi-

naispiirteet, luokittelijat saavuttivat 0.0001-0.03 prosentin osuuden väärissä positiivisissa luokitteluissa, 0.0001-0.013 prosentin osuuden väärissä negatiivisissa luokitteluissa ja 97.5-99.9 prosentin tarkkuuden.

Edellä läpikäydyissä tutkimuksissa väärien positiivisten luokitteluiden osuudet olivat välillä 0.0001-0.72 prosenttia, väärien negatiivisten luokitteluiden osuudet olivat välillä 0.0001-0.72 prosenttia, tarkkuudet olivat välillä 87.46-99.9 prosenttia. Kun taas tässä tutkimuksessa vastaavat luvut olivat parhaimmillaan 0 prosenttia, 2.567 prosenttia ja 99.36 prosenttia. Täten tässä tutkimuksessa saadut tulokset ja edellä esitettyjen haitallisen JavaScript-koodin tunnistamiseen liittyvien tutkimusten tulokset ovat myös samankaltaisia, erityisesti väärien positiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen. Tulosten vertaaminen ei taaskaan ole täysin suoraviivaista, koska tutkimusten tutkimusasetelmat ovat erilaisia. Eroina ovat muuan muassa: tutkimuksissa käytetyt ominaispiirteet, tutkimuksissa käytetyt ominaispiirteiden valintaan käytetyt menetelmät ja tutkimuksissa käytetyt koneoppimismenetelmät.

5 Yhteenveto ja johtopäätökset

Tässä tutkimuksen viimeisessä luvussa kerrataan tutkimuksen vaiheet ja esitetään tutkimuksen tekemisen myötä syntyneet johtopäätökset. Luku aloitetaan kertaamalla tutkimuksen tavoitteet. Tämän jälkeen vastataan tutkimuksessa asetettuihin tutkimuskysymyksiin. Tämän jälkeen esitellään tutkimuksen tuloksien pohjalta tehdyt johtopäätökset. Luvun lopuksi arvioidaan mitä tutkimuksessa olisi voinut tehdä eri tavalla ja pohditaan mahdollisuuksia tutkimuksen pohjalta tehtävällä jatkotutkimukselle.

Tutkimuksessa tutkittiin haitallisen JavaScript-koodin tunnistamista koneoppimismenetelmin. Erityisesti tutkittiin haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan opetuksessa sekä testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteen, opettamiseen ja testaamiseen käytettyjen n-grammien pituuden, opettamiseen ja testaamiseen käytettyjen n-grammien määrään, opettamiseen ja testaamiseen käytettyjen n-grammien valintamenetelmän sekä opettamiseen käytettävän koneoppimismenetelmän vaikutusta luokittelijan suorituskykyyn. Tutkimuksessa vertailtiin näiden parametrien eri arvoilla opetettujen luokittelijoiden suoriutumista JavaScript-tiedostojen luokittelusta väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen. Luokittelijoiden opettamiseen käytettyinä ominaispiirteinä toimivat ohjelmakoodin staattisesta esitystavasta eristetyt n-grammit.

Tutkimuksessa etsittiin vastausta seuraaviin tutkimuskysymyksiin:

1. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteella vaikutusta luokittelijan suorituskykyyn?
2. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien pituudella vaikutusta luokittelijan suorituskykyyn?

3. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien määrällä vaikutusta luokittelijan suorituskykyyn?

4. Onko haitallisen JavaScript-koodin tunnistamista varten opetetun luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävällä menetelmällä vaikutusta luokittelijan suorituskykyyn?

5. Onko haitallisen JavaScript-koodin tunnistamiseen opetetun luokittelijan koneoppimismenetelmällä vaikutusta luokittelijan suorituskykyyn?

Tutkimus tekeminen sisälsi useita vaiheita. Tutkimus alkoi kiinnostavan aihepiirin valinnalla. Aihepiiriksi valikoitui haitallisen JavaScript-koodin tunnistaminen koneoppimismenetelmin. Tutkimuksen tekeminen jatkui aihepiiriin liittyvän kirjallisuuden etsimisellä ja lukemisella. Kirjallisuudesta selvitettiin, millaista tutkimusta aihepiiristä oli aiemmin tehty sekä haettiin mallia tutkimuskysymyksiin ja tutkimuksen tekemiseen. Kooste luetusta kirjallisuudesta esitettiin tutkimuksen toisessa luvussa.

Tutkimuskysymysten selkeytymisen jälkeen selvitettiin millaisella aineistolla ja millaisilla menetelmillä tutkimuskysymyksiin pystytään vastaamaan. Aineistoksi päätettiin rakentaa matriisi, joka kuvaa valittujen parametrien eri arvoilla opetettujen luokittelijoiden suoriutumista JavaScript-tiedostojen luokittelusta väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen. Luodun aineiston analyysiin käytettiin tilastollisia menetelmiä ja silmämääräistä analyysia. Silmämääräisillä analyyseillä pyrittiin tukemaan tilastollisin menetelmin tehtyjä analyyseja. Tilastollisiksi menetelmiksi valikoituivat Kruskal-Wallisin testi, Dunnin testi ja keskiarvovertailu. Aineiston muodostaminen ja siihen sovelletut analyysimenetelmät kuvattiin tarkemmin tutkimuksen kolmannessa luvussa.

Seuraavaksi tutkimuksessa suoritettiin varsinaiset tilastolliset analyysit, silmämääräiset analyysit ja näiden keskinäinen vertailu. Tämän jälkeen arvioitiin käytetyn tutkimusmenetelmän ja tutkimuksen tulosten reliabiliteettia ja validiteettia. Kyseisen arvioinnin jälkeen tutkimuksen tuloksia vertailtiin tutkimuksen kirjallisuuskatsauksessa referoitujen tutkimusten tuloksiin. Tutkimuksessa huomattiin, että tutkittavien parametrien arvoilla oli yleisesti

vaikutusta luokittelijoiden suoriutumiseen JavaScript-tiedostojen luokittelusta. Lisäksi todettiin, että tässä tutkimuksessa saadut tulokset ja aihepiiriin liittyvien muiden tutkimusten tulokset ovat yleisesti samankaltaisia väärien positiivisten luokittelujen osuuden, väärien negatiivisten luokittelujen osuuden ja luokittelun tarkkuuden suhteen. Siitäkin huolimatta, että tulosten vertaaminen ei ole täysin mielekästä johtuen eri tutkimusten erilaisista tutkimusasetelmista. Tässä tutkimuksessa saadut tulokset ja tulosten vertailu aikaisempien tutkimusten tuloksiin kuvattiin tarkemmin tutkimuksen neljännessä luvussa.

Tutkimuksen tulosten perusteella voidaan tutkimuskysymyksiin vastata seuraavasti:

1. Luokittelijan opetuksessa ja testaamisessa käytettävän opetusaineiston haitallisten ja ei-haitallisten ohjelmakoodinäytteiden suhteella on vaikutusta luokittelijan suorituskykyyn. Tutkimuksessa huomattiin, että luokittelijan suorituskyky paranee yleisesti, kun haitallisten näytteiden suhteellista osuutta aineistossa kasvatetaan 5 prosentista 10 prosenttiin tai 5 prosentista 15 prosenttiin.

2. Luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien pituudella on vaikutusta luokittelijan suorituskykyyn. Tutkimuksessa huomattiin, että luokittelijan suorituskyky paranee yleisesti, kun n-grammien pituutta kasvatetaan kahdesta kolmeen.

3. Luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien määrällä on vaikutus luokittelijan suorituskykyyn. Tutkimuksessa huomattiin, että luokittelijan suorituskyky paranee yleisesti, kun n-grammien määrää kasvatettiin 50 n-grammista aina 1600 n-grammiin.

4. Luokittelijan opettamiseen ja testaamiseen käytettyjen n-grammien valintaan käytettävällä menetelmällä on vaikutusta luokittelijan suorituskykyyn. Tutkimuksessa huomattiin, että yleisesti paras suorituskyky saavutettiin Information Gain Ratio-luvun avulla valituilla n-grammeilla.

5. Luokittelijan koneoppimismenetelmällä on vaikutusta luokittelijan suorituskykyyn. Tutkimuksessa mukana olleista koneoppimismenetelmistä yleisesti parhaat tulokset saavuttiin, kun koneoppimismenetelmänä käytetiin CART-päätöspuuta.

Tutkimuksen tuloksista on mahdollista tehdä useita johtopäätöksiä. Ensimmäinen johtopäätös on, että luodessa haitallisen JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, kannattaa parhaan suorituskyvyn saavuttamiseksi, kokeilla eri mahdollisuuksia ainakin opetukseen ja testaamiseen käytettävien tiedostojen määrissä, käytettävien haitallisten tiedostojen määrässä aineistossa, käytettävissä ominaispiirteissä, ominaispiirteiden valintaan käytettävissä menetelmissä ja käytettävässä koneoppimismenetelmässä. Toinen johtopäätös on, että pelkkien JavaScript-ohjelmakoodin staattisesta esitysmuodosta eristettyjen n-grammien avulla näyttäisi olevan mahdollista opettaa haitallisen JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, joiden suorituskyky kilpailee monimutkaisempien ominaispiirteiden avulla opetettujen luokittelijoiden kanssa. Kolmas johtopäätös on, että jo suhteellisen pienillä tiedostomäärillä näyttäisi olevan mahdollista opettaa JavaScript-koodin tunnistamiseen erikoistuneita luokittelijoita, joiden suorituskyky kilpailee suuremmilla aineistoilla opetettujen luokittelijoiden kanssa.

Tutkimuksesta löytyy kuitenkin myös asioita, joita voisi mahdollisesti tehdä toisin tutkimusmenetelmän ja tulosten reliabiliteetin sekä validiteetin parantamiseksi. Tutkimuksessa luokittelijoiden opettamiseen ja testaamiseen käytetyt aineistot olivat huomattavan pieniä, maksimissaan yhteensä 1150 tiedostoa. Lisäksi opetus- ja testiaineistojen haitallisten tiedostojen prosentuaaliset osuudet olivat tutkimuksessa pieniä. Näillä asioilla on voinut olla vaikutusta tutkimuksessa saatuihin tuloksiin. Tämän mahdollisuuden poistamiseksi olisikin ollut hyvä, jos tutkimukseen käytettävissä olevat resurssit olisivat mahdollistaneet myös suurempien opetus- ja testiaineistojen sekä suurempien haitallisten tiedostojen osuuksien käytön luokittelijoiden opettamisessa. Luokittelijoiden suorituskykyä olisi voinut myös testata ristiinvalidoinnin lisäksi esimerkiksi siten, että opetusaineistojen sisältämien haitallisten tiedostojen osuus olisi ollut eri suuri kuin testiaineiston sisältämien haitallisten tiedostojen osuus. Lisäksi tutkimuksessa olisi voinut tutkia tarkemmin geneettisen ohjelmoinnin tuottamien oudoilta vaikuttavien tulosten taustalla olevia syitä.

Tutkimuksen aikana syntyi myös useita ideoita mahdolliselle tutkimuksen pohjalta tehtävälle jatkotutkimukselle. Edellisessä kappaleessa mainitut, tämän tutkimuksen kehittämiseen liittyvät asiat huomioon ottaen, ensimmäinen luonnollinen suunta olisi jatkaa tutkimusta käyttämällä luokittelijoiden opettamiseen ja testaamiseen myös suurempia tiedosto-

määriä sekä suurempia haitallisen ohjelmakoodin osuuksia opetusaineistoissa. Tämän lisäksi tutkimusta voitaisiin jatkaa käyttämällä luokittelijoiden opettamiseen myös nykyistä pitempiä n-grammeja. Tutkimuksen heikkouksia täydentävien ideoiden lisäksi voisi olla mielenkiintoista kääntää JavaScript-tiedostot jonkin JavaScript-moottorin tavukoodiksi, käyttää näitä tavukoodiesityksiä n-grammien lähteenä ja verrata tämän tutkimuksen tuloksia tavukoodiesityksistä johdetuilla n-grammeilla saatuihin tuloksiin. Lisäksi voisi olla mielenkiintoista tutkia olisiko luokittelijoiden suorituskykyä mahdollista parantaa soveltamalla sekvenssianalyysitekniikoita luokittelijoiden opettamiseen käytettävien ominaispiirteiden valintaa. Lisäksi hieman tutkimuksen ulkopuolelta nousevana suuntana jatkotutkimukselle voisi olla tutkia, onko tutkimuksessa saavutettuja tuloksia mahdollista parantaa käytettyjen koneoppimismenetelmien hyperparametrien optimoinnilla.

Lähteet

Abou-Assaleh, T., Cercone, N., Keseli, V., Sweidan, R. (2004). N-gram-based Detection of New Malicious Code. COMPSAC '04 Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02 41-42. https://doi.org/10.1109/CMPSAC.2004.1342667

Ahmed, F., Hameed, H., Shafig, M., Z., Farooq, M. (2009). Using spatio-temporal information in API calls with machine learning algorithms for malware detection. AISec '09 Proceedings of the 2nd ACM workshop on Security and artificial intelligence. 55-62. https://doi.org/10.1145/1654988.1655003

Amalina, F., Feizollah, A., Anuar, N., Gani, A. (2014). Evaluation of machine learning classifiers for mobile malware detection. Journal Soft Computing. 20. 1. 343-357. https://doi.org/10.1007/s00500-014-1511-6

Dr. Bhargava. N., Sharma, G., Bhargava, R., Mathuria, M. (2013). Decision Tree Analysis on J48 Algorithm for Data Mining. International Journal of Advanced Research in Computer Science and Software Engineering 3. 6. 1114-1119. http://www.academia.edu/4375403/Decision_Tree_Analysis_on_J48_Algorithm_for_Data_Mining

Cova, M., Kruegel, C., Vigna, G. (2010). Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. Proceedings of the 19th international conference on World wide web. 281-290. https://doi.org/10.1145/1772690.1772720

Curtsinger, C., Livshits, B., Zorn, B., Seifert, C. (2011) ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. Proceeding SEC'11 Proceedings of the 20th USENIX conference on Security 3-3. https://www.usenix.org/legacy/event/sec11/tech/full_papers/Curtsinger.pdf

Damodaran, A., Troia, F.D., Visaggio, C.A. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. Journal of Computer Virology and Hacking Techniques 13, 1-12. https://doi.org/10.1007/s11416-015-0261-z

Deylami, H.M., Munivandi, R.C., Ardekani, I.T., Sarrafradeh, A. (2016). Taxonomy of malware detection techniques: A systematic literature review. 14th Annual Conference on Privacy, Security and Trust. doi:10.1109/PST.2016.7906998

Dinno, A. (2015). Nonparametric pairwise multiple comparisons in independent groups using Dunn's test. The Stata Journal 15. 1. 292-300. https://www.stata-journal.com/sjpdf.html?articlenum=st0381

Firdausi, I., Lim, C., Erwin, A., Nugroho, A. (2010). Analysis of Machine Learning Techniques Used in Behavior-Based Malware Detection. Advances in Computing, Control, and Telecommunication Technologies, International Conference on. 0. 201-203. https://www.researchgate.net/publication/232627329_Analysis_of_Machine_learning_Techniques_Used_in_Behavior-Based_Malware_Detection

Gandotra, E., et al. (2014) Malware Analysis and Classification: A Survey. Journal of Information Security, 5, 56-64. http://dx.doi.org/10.4236/jis.2014.52006

Gavrilut, D., Cimpoes, M., Anton, D., Ciortuz, L. (2009). Malware Detection Using Machine Learning. Proceedings of the International Multiconference on Computer Science and Information Technology. 4. 735–741. https://pdfs.semanticscholar.org/1536/bffa0b497ff6cddd65f3cab5f98c6e9bb025.pdf

Gu, Q., Li, Z., Han, J. (2011). Generalized Fisher score for feature selection. Proceeding UAI'11 Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence. 266-273. https://arxiv.org/ftp/arxiv/papers/1202/1202.3725.pdf

Hansen, J., V., Lowry, P., B., Meservy, R., D., McDonald, D., M. (2007). Genetic programming for prevention of cyberterrorism through dynamic and evolving intrusion detection. Journal Decision Support Systems archive 43. 4. 1362-1374. https://doi.org/10.1016/j.dss.2006.04.004

Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen. J. M. (2016). An approach for detection and family classification of malware based on behavioral analysis. Computing, Networking and Communications (ICNC), 2016 International Conference on. https://doi.org/10.1109/ICCNC.2016.7440587

Idika, N., Mathur, A. P. (2007) A Survey of Malware Detection Techniques. http://profsandhu.com/cs5323_s17/im_2007.pdf

Kapravelos, A., Shoshitaishvili, Y., Cova, M., Vigna, G. (2010). Revolver: an automated approach to the detection of evasive web-based malware. Proceedings of the 22nd USENIX conference on Security. 637-652. https://dl.acm.org/citation.cfm?id=2534821

Kotsiantis, S., B. (2007) Supervised Machine Learning: A Review of Classification Techniques. Proceeding Proceedings of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies. 3-24. https://datajobs.com/data-science-repo/Supervised-Learning-[SB-Kotsiantis].pdf

Krishna, G. B., Radha, V., Venu Gopala Rao, K. (2016). Review of Contemporary Literature on Machine Learning based Malware Analysis and Detection Strategies. Global Journal of Computer Science and Tecchnology. 5. 10-16. https://computerresearch.org/index.php/computer/article/view/1410

Kruegel, C. (2007). Behavioral and Structural Properties of Malicous Code. In: Christodorescu M., Jha S., Maughan D., Song D., Wang C. (eds) Malware Detection. Advances in Information Security. 27. https://doi.org/10.1007/978-0-387-44599-1_4

Le, T., A., Chu T., H., Nguyen, Q., U., Nguyen, X., H. (2014). Malware Detection Using Genetic Programming. Computational Intelligence for Security and Defense Applications (CISDA), 2014 Seventh IEEE Symposium on. https://doi.org/10.1109/CISDA.2014.7035623

Likarish, P., Jung, J., Jo, I. (2009). Obfuscated malicous javascript detection using classification techniques. 2009 4th International Conference on Malicious and Unwanted Software. https://doi.org/10.1109/MALWARE.2009.5403020

Loh, W. (2008). Classification and Regression Tree Methods. Encyclopedia of Statistics in Quality and Reliability. 315–323. https://www.stat.wisc.edu/~loh/treeprogs/guide/eqr.pdfts

Kolter, J., Z., Maloof, M., A. (2004). Learning to Detect Malicious Executables in the Wild. Proceeding KDD '04 Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. 470-478. https://doi.org/10.1145/1014052.1014105

Manning, C., C., Raghavan, P., Schütze, H. (2009). Introduction to Information Retrieval. Cambridge University Press. 118-119 https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf

Menahem, E., Shabtai, A., Rokach, L., Elovici, Y. (2009). Improving malware detection by applying multi-inducer ensemble. Computational Statistics & Data Analysis archive. 53. 4. 1483-1494. https://doi.org/10.1016/j.csda.2008.10.015

Meyers, J.P., Seaman, M. A. (2013) Comparison of the Exact Kruskal-Wallis Distribution to Asymptotic Approximations for All Sample Sizes up to 105. The Journal of Experimental Education 81. 2. 139-156. http://dx.doi.org/10.1080/00220973.2012.699904

Mori, T., Kikuchi, M., Yoshida, K. (2002). Term Weighting Method based on Information Gain Ratio for Summarizing Documents retrieved by IR systems. Journal of Natural Language Processing. 9. 4. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.3450&rep=rep1&type=pdf

Moskovitch, R., Fehrer, C., Elovici, Y. (2008) [1]. Unknown Malcode Detection – A Chronological Evaluation. Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on. 267-268. https://doi.org/10.1109/ISI.2008.4565078

Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y. (2008) [2]. Unknown Malcode Detection via Text Categorization and the Imbalance Problem. Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on. 156-161. https://doi.org/10.1109/ISI.2008.4565046

Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., Elovici, Y. (2008) [3]. Unknown malcode detection using OPCODE representation. European Conference on Intelligence and Security Informatics. 204–215. https://doi.org/10.1007/978-3-540-89900-6_21

Moskovitch, R., Nissim, N., Elovici, Y. (2010). Acquisition of malicious code using active learning. https://www.researchgate.net/profile/Robert_Moskovitch/publication/228953558_Acquisition_of_malicious_code_using_active_learning/links/5419e1ca0cf203f155ae148f/Acquisition-of-malicious-code-using-active-learning.pdf?origin=publication_detail

Narayanan, A., Chen, Y., Pang, S., Tao, B. (2013). The Effects of Different Representations on Static Structure Analysis of Computer Malware Signatures. The Scientific World Journal. https://doi.org/10.1155/2013/671096

Patil, J., B., Patil, D., R. (2017). Detection of Malicious JavaScript Code in Web Pages. Indian Journal of Science and Technology 10. 19. 1-12 http://www.indjst.org/index.php/indjst/article/download/114828/80248

Poli, R., Langdon, W., B., Mcphee, N. (2008). A Field Guide to Genetic Programming. https://www.researchgate.net/profile/Nicholas_Mcphee/publication/216301261_A_Field_Guide_to_Genetic_Programming/links/004635225dcdfc2097000000/A-Field-Guide-to-Genetic-Programming.pdf

Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N. (2007) The ghost in the browser analysis of web-based malware. Proceeding HotBots'07 Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets. 4-4. https://dl.acm.org/citation.cfm?id=1323132

Quinlan, J.R. (1993). C4.5: Programs for machine learning. Morgan Kaufmann, San Francisco

Ravi, K. K., Mallesh, M., Jyostna, G., Eswar P. R. L., Samavedam, S. S. (2014). Browser JS Guard: Detects and Defends against Malicious JavaScript Injection based Drive by Download Attacks: A measurement study. Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the. https://doi.org/10.1109/ICADIWT.2014.6814705

Rieck, K., Trinius, P., Willems, C., Holz, T. (2010). Automatic analysis of malware behavior using machine learning. Journal of Computer Security archive 19. 4. 639-668. http://www.mlsec.org/malheur/docs/malheur-jcs.pdf

Sahs, J., Khan, L. (2012). A Machine Learning Approach to Android Malware Detection. Proceedings - 2012 European Intelligence and Security Informatics Conference, EISIC 2012. 141-147. https://doi.org/10.1109/EISIC.2012.34

Santos, I., Brezo, F., Sanz, B., Laorden, C., Bringas, P., G. (2011). Using Opcode Sequences in Single-Class Learning to Detect Unknown Malware. IET Information Security 5. 4. 220-227. https://doi.org/10.1049/iet-ifs.2010.0180

Saxe, J., Berlin, K. (2015). Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on. 11-20. https://doi.org/10.1109/MALWARE.2015.7413680

Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C. (2009). Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey, Inform. Secur. Tech. Rep. (2009), doi:10.1016/j.istr.2009.03.003

Schultz, M, G., Eskin, E., Zadok, F., Stolfo, S. (2001). Data Mining Methods for Detection of New Malicious Executables. Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy. 38-49. https://www.researchgate.net/publication/3897868_Data_Mining_Methods_for_Detection_of_New_Malicious_Executables

Tabish, S. M., Shafiq, M. Z., Farooq, M. (2009) Malware detection using statistical analysis of byte-level file content. Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Information. 23-31. https://doi.org/10.1145/1599272.1599278

Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T. (2016). Malware Detection with Deep Neural Network Using Process Behavior. Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual. 577-582. https://doi.org/10.1109/COMPSAC.2016.151

Wang, J., Xue, Y., Liu, Y., Tan, T., H. (2015) JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification. Proceeding ASIA CCS '15 Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security 109-120. https://doi.org/10.1145/2714576.2714620

Xu, W., Zhang, F., Zhu, S. (2013). The power of obfuscation techniques in malicious JavaScript code: A measurement study. Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on. https://doi.org/10.1109/MALWARE.2012.6461002

Xu, W., Zhang, F., Zhu, S. (2013) JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. Proceeding CODASPY '13 Proceedings of the third ACM conference on Data and application security and privacy. 117-128. https://doi.org/10.1145/2435349.2435364

Zhao, Y., Zhang, Y. (2008). Comparison of decision tree methods for finding active objects. Advances in Space Research. 41. 12. 1955-1959. https://doi.org/10.1016/j.asr.2007.07.020

Liitteet

A main_result_provider.py lähdekoodi

```
import numpy as np

import pandas as pd

import sys

sys.path.insert(0, 'C:/Users/jantu/Dropbox/thesis/source_code/ngram_providers')

sys.path.insert(1, 'C:/Users/jantu/Dropbox/thesis/source_code/feature_providers')

sys.path.insert(2, 'C:/Users/jantu\Dropbox/thesis/source_code/logging_provider')


import ngram_file_distribution_provider

import df_idf_feature_provider

import fisher_score_feature_provider

import information_gain_provider

import logging_provider

import cart_result_provider

import ann_result_provider

import gp_result_provider


CSV_FILE_PATH = 'xxx'
```

```
def get_file_name():
    return 'main_result_provider.py'


def get_ngram_file_distribution_provider_opt(opt_dict, ngram_len):
    logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name + ' called with ' + str(opt_dict) + str(ngram_len))
    provider_opt = opt_dict.copy()
    provider_opt['ngram_len'] = ngram_len
    return provider_opt


def flush_feature_providers_caches():
    df_idf_feature_provider.flush_cache()
    fisher_score_feature_provider.flush_cache()
    information_gain_provider.flush_cache()


def get_features_data_frame_dicts_as_dict(js_ngram_freq_data_frame_dict, feature_amount):
    logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name + ' called with ' + str(feature_amount))
    feature_data_frame_dicts_as_dict = {}
    feature_data_frame_dicts_as_dict['td_idf_features'] = df_idf_feature_provider.get_df_idf_features(js_ngram_freq_data_frame_dict, feature_amount)
```

```
    feature_data_frame_dicts_as_dict['fisher_score_features'] = fisher_score_feature_provider.get_fisher_score_features(js_ngram_freq_data_frame_dict, feature_amount)

    feature_data_frame_dicts_as_dict['information_gain_features'] = information_gain_provider.get_information_gain_features(js_ngram_freq_data_frame_dict, feature_amount)

    return feature_data_frame_dicts_as_dict


def append_series_to_data_frame(data_frame, series):

    logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name)

    if data_frame is None:

        return pd.concat([series], axis=1)

    else:

        return pd.concat([data_frame, series], axis=1)


def save_data_frame_as_csv_to_disk(data_frame, csv_name):

    logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name + ' called with ' + str(csv_name))

    data_frame.to_csv(CSV_FILE_PATH + csv_name + '.csv')

    print(csv_name)

    print(data_frame)
```

```
def get_and_write_all_results(opt_dict):
    logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name + ' called with ' + str(opt_dict))
    for ngram_len in opt_dict['ngram_lenghts']:
        ngram_file_distribution_provider_opt = get_ngram_file_distribution_provider_opt(opt_dict, ngram_len)
        js_ngram_freq_data_frame_dict = ngram_file_distribution_provider.get_ngram_distributions_of_js_files_as_data_frames(ngram_file_distribution_provider_opt)
        flush_feature_providers_caches()

        for feature_amount in opt_dict['feature_amounts']:
            feature_data_frame_dicts_as_dict = get_features_data_frame_dicts_as_dict(js_ngram_freq_data_frame_dict, feature_amount)
            cart_result_data_frame = None
            ann_result_data_frame = None
            gp_result_data_frame = None

            for dict_key in feature_data_frame_dicts_as_dict.keys():
                logging_provider.write_to_log_file(get_file_name() + ' ' + sys._getframe().f_code.co_name + ' feature name ' + str(dict_key))
                tmp_result_data_frame = cart_result_provider.get_cart_tree_model_results_data_frame_using_k_fold(feature_data_frame_dicts_as_dict[dict_key], opt_dict['fold_amount'])
```

```
            cart_result_data_frame = append_series_to_data_frame(cart_result_data_frame,
tmp_result_data_frame.mean().rename(dict_key))


            tmp_result_data_frame =
ann_result_provider.get_ann_results_data_frame_using_k_fold(feature_data_frame_dicts_
as_dict[dict_key], opt_dict['fold_amount'])

            ann_result_data_frame = append_series_to_data_frame(ann_result_data_frame,
tmp_result_data_frame.mean().rename(dict_key))


            tmp_result_data_frame =
gp_result_provider.get_gp_results_data_frame_using_k_fold(feature_data_frame_dicts_as
_dict[dict_key], opt_dict['fold_amount'])

            gp_result_data_frame = append_series_to_data_frame(gp_result_data_frame,
tmp_result_data_frame.mean().rename(dict_key))


        save_data_frame_as_csv_to_disk(cart_result_data_frame, 'cart_ngram_' +
str(ngram_len) + '_feature_' + str(feature_amount))

        save_data_frame_as_csv_to_disk(ann_result_data_frame, 'ann_ngram_' +
str(ngram_len) + '_feature_' + str(feature_amount))

        save_data_frame_as_csv_to_disk(gp_result_data_frame, 'gp_ngram_' +
str(ngram_len) + '_feature_' + str(feature_amount))


if __name__ == '__main__':

    opt_dict = {}
```

```
    opt_dict['mal_js_amount'] = 150
    opt_dict['non_mal_js_amount'] = 1000
    opt_dict['ngram_lenghts'] = [3]
    opt_dict['ngram_amount'] = 10000000000

    #ann_opt_dict['feature_amounts'] = [50, 100, 200, 400, 800, 1600]
    opt_dict['feature_amounts'] = [50, 100, 200, 400, 800, 1600]
    opt_dict['fold_amount'] = 4
    #opt_dict['ngram_len'] = 2

    logging_provider.write_to_log_file('program started with ' + str(opt_dict))
    get_and_write_all_results(opt_dict)
    logging_provider.write_to_log_file('program ended')
```

B statistics_provider.py lähdekoodi

```
import numpy as np
import pandas as pd
from scipy.stats import mstats
import scikit_posthocs as sp

np.set_printoptions(suppress=True)
```

```
RESULT_CSV_PATH = 'xxx'
RESULT_STATISTICS_PATH = 'xxx'


cached_results_df = None


def get_all_result_rows_as_dataframe():
    global cached_results_df
    if cached_results_df is None:
        cached_results_df = pd.read_csv(RESULT_CSV_PATH)
    return cached_results_df


def erase_statistics_file():
    open(RESULT_STATISTICS_PATH, 'w').close()


def write_to_statistics_file(written_string):
    with open(RESULT_STATISTICS_PATH, 'a') as file:
        file.write(str(written_string))


def get_rows_from_results_data_frame_that_meet_condition(column_name, condition):
    df = get_all_result_rows_as_dataframe()
```

```
    return df.loc[df[column_name] == condition]


def get_data_frame_column_values_as_list(df, column_name):
    return df[column_name].tolist()


def get_kruskal_wallis_test_results_data_frame_by_column_name(column_name, *args):
    np_arrays = []
    array_names = []
    for arg in args:
        arg_df = arg[1]
        np_array = np.array(arg_df[column_name].tolist())
        np_arrays.append(np_array)
        array_names.append(arg[0])
    np_arrays_tuple = tuple(np_arrays)


    H, p = mstats.kruskalwallis(np_arrays_tuple)
    write_to_statistics_file('\nH={0:f} and p={1:f} for column name: '.format(H, p) + column_name + '\n')


    if p < 0.05:
        df_for_dunn = pd.DataFrame(np.array(np_arrays).T, columns=array_names)
```

```
        get_means_for_df(column_name, df_for_dunn)
        df_for_dunn = df_for_dunn.melt(var_name='groups', value_name='values')
        get_posthoc_dunn_results(column_name, df_for_dunn)

def get_means_for_df(column_name, df):
    write_to_statistics_file('means for ' + column_name + ':\n')
    write_to_statistics_file(df.mean())
    write_to_statistics_file('\n')

def get_posthoc_dunn_results(column_name, x):
    write_to_statistics_file('dunn posthoc test for ' + column_name + ':\n')
    write_to_statistics_file(sp.posthoc_dunn(x, val_col='values',
group_col='groups').round(4))
    write_to_statistics_file('\n')

def get_kruskal_wallis_test_results_for_tp_rate(*args):
    get_kruskal_wallis_test_results_data_frame_by_column_name('tp_rate', *args)

def get_kruskal_wallis_test_results_for_tn_rate(*args):
    get_kruskal_wallis_test_results_data_frame_by_column_name('tn_rate', *args)
```

```
def get_kruskal_wallis_test_results_for_fp_rate(*args):
    get_kruskal_wallis_test_results_data_frame_by_column_name('fp_rate', *args)

def get_kruskal_wallis_test_results_for_fn_rate(*args):
    get_kruskal_wallis_test_results_data_frame_by_column_name('fn_rate', *args)

def get_kruskal_wallis_test_results_for_acc(*args):
    get_kruskal_wallis_test_results_data_frame_by_column_name('acc', *args)

def get_kruskal_wallis_test_results_for_mal_non_mal_ratios():
    df_50 = get_rows_from_results_data_frame_that_meet_condition('mal_amount', 50)
    df_100 = get_rows_from_results_data_frame_that_meet_condition('mal_amount', 100)
    df_150 = get_rows_from_results_data_frame_that_meet_condition('mal_amount', 150)

    write_to_statistics_file('Kruskal Wallis results for mal and non mal ratios: ' + '\n')

    get_kruskal_wallis_test_results_for_tp_rate(('50', df_50), ('100', df_100), ('150', df_150))
    get_kruskal_wallis_test_results_for_tn_rate(('50', df_50), ('100', df_100), ('150', df_150))
    get_kruskal_wallis_test_results_for_fp_rate(('50', df_50), ('100', df_100), ('150', df_150))
```

```
    get_kruskal_wallis_test_results_for_fn_rate(('50', df_50), ('100', df_100), ('150', df_150))

    get_kruskal_wallis_test_results_for_acc(('50', df_50), ('100', df_100), ('150', df_150))


    write_to_statistics_file('__________' + '\n\n')


def get_kruskal_wallis_test_results_for_ngram_lenghts():
    df_2 = get_rows_from_results_data_frame_that_meet_condition('ngram_len', 2)
    df_3 = get_rows_from_results_data_frame_that_meet_condition('ngram_len', 3)


    write_to_statistics_file('Kruskal Wallis results for ngrams lengths:' + '\n')


    get_kruskal_wallis_test_results_for_tp_rate(('2', df_2), ('3', df_3))
    get_kruskal_wallis_test_results_for_tn_rate(('2', df_2), ('3', df_3))
    get_kruskal_wallis_test_results_for_fp_rate(('2', df_2), ('3', df_3))
    get_kruskal_wallis_test_results_for_fn_rate(('2', df_2), ('3', df_3))
    get_kruskal_wallis_test_results_for_acc(('2', df_2), ('3', df_3))


    write_to_statistics_file('__________' + '\n\n')


def get_kruskal_wallis_test_results_for_feature_amounts():
```

```
    df_50 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 50)

    df_100 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 100)

    df_200 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 200)

    df_400 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 400)

    df_800 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 800)

    df_1600 = get_rows_from_results_data_frame_that_meet_condition('feat_amount', 1600)


    write_to_statistics_file('Kruskal Wallis results for feature amounts: ' + '\n')


    get_kruskal_wallis_test_results_for_tp_rate(('50', df_50), ('100', df_100), ('200', df_200), ('400', df_400), ('800', df_800), ('1600', df_1600))

    get_kruskal_wallis_test_results_for_tn_rate(('50', df_50), ('100', df_100), ('200', df_200), ('400', df_400), ('800', df_800), ('1600', df_1600))

    get_kruskal_wallis_test_results_for_fp_rate(('50', df_50), ('100', df_100), ('200', df_200), ('400', df_400), ('800', df_800), ('1600', df_1600))

    get_kruskal_wallis_test_results_for_fn_rate(('50', df_50), ('100', df_100), ('200', df_200), ('400', df_400), ('800', df_800), ('1600', df_1600))

    get_kruskal_wallis_test_results_for_acc(('50', df_50), ('100', df_100), ('200', df_200), ('400', df_400), ('800', df_800), ('1600', df_1600))


    write_to_statistics_file('_____' + '\n\n')
```

```
def get_kruskal_wallis_test_results_for_feature_provider_methods():
    df_tf_idf = get_rows_from_results_data_frame_that_meet_condition('feat_method', 'tf_idf')
    df_fs = get_rows_from_results_data_frame_that_meet_condition('feat_method', 'fs')
    df_ig = get_rows_from_results_data_frame_that_meet_condition('feat_method', 'ig')

    write_to_statistics_file('Kruskal Wallis results for feature providing methods:' + '\n')

    get_kruskal_wallis_test_results_for_tp_rate(('tf_idf', df_tf_idf), ('fs', df_fs), ('ig', df_ig))
    get_kruskal_wallis_test_results_for_tn_rate(('tf_idf', df_tf_idf), ('fs', df_fs), ('ig', df_ig))
    get_kruskal_wallis_test_results_for_fp_rate(('tf_idf', df_tf_idf), ('fs', df_fs), ('ig', df_ig))
    get_kruskal_wallis_test_results_for_fn_rate(('tf_idf', df_tf_idf), ('fs', df_fs), ('ig', df_ig))
    get_kruskal_wallis_test_results_for_acc(('tf_idf', df_tf_idf), ('fs', df_fs), ('ig', df_ig))

    write_to_statistics_file('_____' + '\n\n')

def get_kruskal_wallis_test_results_for_machine_learning_methods():
    df_ann = get_rows_from_results_data_frame_that_meet_condition('ml_method', 'ann')
    df_cart = get_rows_from_results_data_frame_that_meet_condition('ml_method', 'cart')
    df_gp = get_rows_from_results_data_frame_that_meet_condition('ml_method', 'gp')
```

```
    write_to_statistics_file('Kruskal Wallis results for machine learning methods:' + '\n')

    get_kruskal_wallis_test_results_for_tp_rate(('ann', df_ann), ('cart', df_cart), ('gp', df_gp))

    get_kruskal_wallis_test_results_for_tn_rate(('ann', df_ann), ('cart', df_cart), ('gp', df_gp))

    get_kruskal_wallis_test_results_for_fp_rate(('ann', df_ann), ('cart', df_cart), ('gp', df_gp))

    get_kruskal_wallis_test_results_for_fn_rate(('ann', df_ann), ('cart', df_cart), ('gp', df_gp))

    get_kruskal_wallis_test_results_for_acc(('ann', df_ann), ('cart', df_cart), ('gp', df_gp))

    write_to_statistics_file('_____' + '\n\n')

if __name__ == '__main__':
    erase_statistics_file()
    get_kruskal_wallis_test_results_for_mal_non_mal_ratios()
    get_kruskal_wallis_test_results_for_ngram_lenghts()
    get_kruskal_wallis_test_results_for_feature_amounts()
    get_kruskal_wallis_test_results_for_feature_provider_methods()
    get_kruskal_wallis_test_results_for_machine_learning_methods()
```

C Tilastollisen analyysin tuloste

sdfs

Kruskal Wallis results for mal and non mal ratios:

H=19.909716 and p=0.000047 for column name: fp_rate

means for fp_rate:

50 0.004825

100 0.007496

150 0.016703

dtype: float64

dunn posthoc test for fp_rate:

	100	150	50
100	-1.0000	0.0018	0.2337
150	0.0018	-1.0000	0.0000
50	0.2337	0.0000	-1.0000

H=43.568229 and p=0.000000 for column name: fn_rate

means for fn_rate:

50 0.474752

100 0.363622

150 0.239278

dtype: float64

dunn posthoc test for fn_rate:

	100	150	50
100	-1.0000	0.0016	0.0006
150	0.0016	-1.0000	0.0000
50	0.0006	0.0000	-1.0000

H=3.678827 and p=0.158911 for column name: acc

_____

Kruskal Wallis results for ngrams lengths:

H=6.233328 and p=0.012537 for column name: fp_rate

means for fp_rate:

2 0.010143

3 0.009206

dtype: float64

dunn posthoc test for fp_rate:

	2	3
2	-1.0000	0.0125
3	0.0125	-1.0000

H=2.196504 and p=0.138324 for column name: fn_rate

H=5.899908 and p=0.015142 for column name: acc

means for acc:

2 0.957887

3 0.966646

dtype: float64

dunn posthoc test for acc:

	2	3
2	-1.0000	0.0151
3	0.0151	-1.0000

_____

Kruskal Wallis results for feature amounts:

H=2.713739 and p=0.744016 for column name: fp_rate

H=14.033855 and p=0.015396 for column name: fn_rate

means for fn_rate:

50 0.450061

100 0.432100

200 0.363613

400 0.353897

800 0.290520

1600 0.265112

dtype: float64

dunn posthoc test for fn_rate:

	100	1600	200	400	50	800
100	-1.0000	0.0315	0.4516	0.2139	0.4596	0.0243
1600	0.0315	-1.0000	0.1622	0.3641	0.0039	0.9193
200	0.4516	0.1622	-1.0000	0.6241	0.1356	0.1339
400	0.2139	0.3641	0.6241	-1.0000	0.0474	0.3130
50	0.4596	0.0039	0.1356	0.0474	-1.0000	0.0028
800	0.0243	0.9193	0.1339	0.3130	0.0028	-1.0000

H=28.249760 and p=0.000033 for column name: acc

means for acc:

50 0.953411

100 0.954882

200 0.960636

400 0.963793

800 0.969820

1600 0.971057

dtype: float64

dunn posthoc test for acc:

	100	1600	200	400	50	800
100	-1.0000	0.0002	0.3646	0.0366	0.6028	0.0026
1600	0.0002	-1.0000	0.0053	0.1087	0.0000	0.4951
200	0.3646	0.0053	-1.0000	0.2368	0.1536	0.0353
400	0.0366	0.1087	0.2368	-1.0000	0.0091	0.3565
50	0.6028	0.0000	0.1536	0.0091	-1.0000	0.0004
800	0.0026	0.4951	0.0353	0.3565	0.0004	-1.0000

_____

Kruskal Wallis results for feature providing methods:

H=1.945105 and p=0.378117 for column name: fp_rate

H=21.018916 and p=0.000027 for column name: fn_rate

means for fn_rate:

tf_idf 0.487085

fs 0.361834

ig 0.228733

dtype: float64

dunn posthoc test for fn_rate:

	fs	ig	tf_idf
fs	-1.0000	0.0088	0.0511
ig	0.0088	-1.0000	0.0000
tf_idf	0.0511	0.0000	-1.0000

H=39.273781 and p=0.000000 for column name: acc

means for acc:

tf_idf 0.946371

fs 0.963651

ig 0.976778

dtype: float64

dunn posthoc test for acc:

	fs	ig	tf_idf
fs	-1.0000	0.0016	0.0018
ig	0.0016	-1.0000	0.0000
tf_idf	0.0018	0.0000	-1.0000

Kruskal Wallis results for machine learning methods:

H=118.347111 and p=0.000000 for column name: fp_rate

means for fp_rate:

ann 0.002571

cart 0.013274

gp 0.013179

dtype: float64

dunn posthoc test for fp_rate:

	ann	cart	gp
ann	-1.0000	0.0	0.0003
cart	0.0000	-1.0	0.0000
gp	0.0003	0.0	-1.0000

H=143.633959 and p=0.000000 for column name: fn_rate

means for fn_rate:

ann 0.406197

cart 0.107985

gp 0.563470

dtype: float64

dunn posthoc test for fn_rate:

	ann	cart	gp
ann	-1.0000	0.0	0.0005
cart	0.0000	-1.0	0.0000
gp	0.0005	0.0	-1.0000

H=96.183110 and p=0.000000 for column name: acc

means for acc:

ann 0.963254

cart 0.979906

gp 0.943639

dtype: float64

dunn posthoc test for acc:

	ann	cart	gp
ann	-1.0000	0.0004	0.0
cart	0.0004	-1.0000	0.0
gp	0.0000	0.0000	-1.0

_____