VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Teksto klasifikavimas

Text classification

Bakalauro darbas

Atliko: Karolis Deveikis
(parašas)

Darbo vadovas: lekt. dr. Rimantas Kybartas
(parašas)

Darbo recenzentas: asist. Linas Petkevičius
(parašas)

Vilnius – 2016

Santrauka

Šiame darbe nagrinėjamas teksto klasifikavimo uždavinys. Jo esmė – tekstą pagal jo turinį priskirti vienai iš kelių iš anksto žinomų kategorijų. Teksto klasifikavimas yra vienas iš natūralios kalbos apdorojimo uždavinių. Kaip ir kiti natūralios kalbos apdorojimo uždaviniai, jis dažnai sprendžiamas mašinų mokymosi algoritmais. Egzistuoja daug algoritmų, tinkamų teksto klasifikavimui. Dėl to kyla pasirinkimo problema. Siekiant šią problemą išspręsti, darbe analizuojami įvairūs požymių išskyrimo ir klasifikavimo algoritmai. Remiantis atlikta analize atliekamas eksperimentas, kurio metu išmatuojamas kiekvieno analizuoto algoritmo tikslumas. Pagal eksperimento rezultatus suformuluojamos išvados, galinčios padėti pasirinkti algoritmą.

Raktiniai žodžiai: natūralios kalbos apdorojimas, teksto klasifikavimas, mašinų mokymasis, požymių išskyrimas.

Summary

This paper investigates the problem of text classification. The task of text classification is to assign a piece of text to one of several categories based on its content. Text classification is one of the tasks of natural language processing. Like the others, it is often solved using machine learning algorithms. There are many algorithms suitable for text classification. As a result, a problem of choice arises. In an effort to solve this problem, this paper analyzes various feature extraction and classification algorithms. Based on the analysis, an experiment is performed to measure the classification accuracy of each algorithm analyzed in the paper. Based on the results of the experiment, a set of conclusions is made which may help in choosing an algorithm.

Keywords: natural language processing, text classification, machine learning, feature extraction.

Turinys

Įvadas

Teksto klasifikavimo uždavinys

Šiame darbe nagrinėjamas teksto klasifikavimo uždavinys. Teksto klasifikavimo uždavinio esmė – tekstą pagal jo turinį priskirti vienai iš kelių iš anksto žinomų kategorijų. Tipiškos teksto klasifikavimo taikymo sritys:

1. Nepageidaujamų elektroninių laiškų (angl. spam) atpažinimas [AKC+00].
2. Teksto autoriaus nustatymas [DAC+01].
3. Teksto kalbos nustatymas [Dun94].
4. Literatūros kūrinio žanro nustatymas [FK06].
5. Teksto autoriaus požiūrio nustatymas [PL04; PLV02].

Tekstas dažnai turi metaduomenis, pavyzdžiui, autoriaus asmens duomenis, publikavimo laiką arba siuntėjo adresą (elektroninių laiškų atveju). Remiantis metaduomenimis taip pat galima klasifikuoti tekstą. Šiame darbe toks klasifikavimo būdas nenagrinėjamas.

Teksto klasifikavimas yra vienas iš natūralios kalbos apdorojimo (angl. natural language processing) uždavinių. Natūralios kalbos apdorojimas yra disciplina, tirianti žmonių kalbų apdorojimą kompiuteriais. Pirmieji natūralios kalbos apdorojimo algoritmai buvo paremti išreikštinai užrašytomis, žmonių sudarytomis taisyklėmis. Šiais laikais natūralios kalbos apdorojimo uždaviniai dažnai sprendžiami mašinų mokymosi algoritmais.

Mašinų mokymasis

Mašinų mokymasis yra kompiuterių mokslo atšaka, artimai susijusi su dirbtiniu intelektu ir statistika. Mašinų mokymosi disciplina tiria algoritmus, kurie, panašiai kaip gyvi organizmai, sugeba mokytis iš jiems pateiktų duomenų. Mašinų mokymosi algoritmai dažnai naudojami spręsti uždaviniams, kuriems sunku sudaryti išreikštinį algoritmą.

Mašinų mokymosi algoritmai sukonstruoja funkciją, modeliuojančią jiems pateiktus duomenis. Pagal tai, kokiu būdu ši funkcija konstruojama, mašinų mokymosi algoritmai skirstomi į tris grupes [Lis96]:

1. **Mokymasis su mokytoju (angl. supervised learning)**.
   Apmokant mokymosi su mokytoju algoritmą, jam kaip pavyzdys pateikiama aibė funkcijos grafiko taškų. Algoritmas pagal tai sukonstruoja funkciją, artimą tai, iš kurios buvo gauti pavyzdiniai duomenys.
2. **Mokymasis be mokytojo (angl. unsupervised learning)**.
   Mokymosi be mokytojo algoritmui, priešingai nei mokymosi su mokytoju algoritmams, pateikiami tik funkcijos argumentai. Algoritmas turi aptikti juose esančius dėsningumus ir

sukonstruoti juos išreiškiančią funkciją. Tipiškas šiais algoritmais sprendžiamo uždavinio pavyzdys – duomenų grupavimas (angl. clustering) į iš anksto nežinomas grupes.

3. **Mokymasis skatinant (angl. reinforcement learning)**.
   Panašiai kaip ir mokymosi be mokytojo atveju, apmokant mokymosi skatinant algoritmą nėra žinomos norimos funkcijos reikšmės. Tačiau yra žinomas būdas įvertinti algoritmo išvedamų reikšmių teisingumą.

Klasifikavimo uždaviniams spręsti naudojami mokymosi su mokytoju algoritmai.

Darbo tikslas ir uždaviniai

Egzistuoja daug algoritmų, tinkamų teksto klasifikavimui. Iš to kyla klausimas: kurį iš jų pasirinkti? Bene svarbiausia klasifikavimo algoritmo charakteristika – jo tikslumas. Šio darbo tikslas – palyginti įvairių algoritmų tikslumą klasifikuojant tekstą. Šiame tikslui pasiekti keliami tokie uždaviniai:

1. Išanalizuoti algoritmus, naudojamus teksto klasifikavimui.
2. Remiantis teorinėje darbo dalyje atlikta algoritmų analize, eksperimentiniu būdu nustatyti klasifikavimo tikslumą, kurį galima pasiekti kiekvienu analizuotu algoritmu.
3. Remiantis eksperimento rezultatais palyginti algoritmus.

Temos analizės tvarka

Šis darbas susideda iš keturių dalių:

1. 1-ame skyriuje („Požymių išskyrimas“) aprašyti būdai tekstą transformuoti į skaitinį pavidalą, tinkamą klasifikavimui mašinų mokymosi algoritmais.
2. 2-ame skyriuje („Mašinų mokymosi algoritmai“) aprašyti trys klasifikavimui naudojami mašinų mokymosi algoritmai.
3. 3-ame skyriuje („Algoritmų palyginimas“) aprašytas atliktas eksperimentas ir pateikti jo rezultatai.
4. 4-ame skyriuje („Alternatyvūs sprendimai“) trumpai aptariami teksto klasifikavimo būdai, kurie šiame darbe nebuvo nagrinėjami.

Naudojamos priemonės

Eksperimentinėje darbo dalyje naudojamos šios priemonės:

1. Programavimo kalba Python 3 [VD95].
2. Matematinių skaičiavimų biblioteka NumPy [VCV11].
3. Įrankių moksliniams tyrimams biblioteka SciPy [JOP+16].

4. Mašinų mokymosi algoritmų biblioteka scikit-learn [PVG+11].
5. Dirbtinių neuroninių tinklų biblioteka scikit-neuralnetwork [CS15].
6. Natūralios kalbos apdorojimo algoritmų biblioteka NLTK [BKL09].
7. Natūralios kalbos apdorojimo algoritmų biblioteka Gensim [ŘS10].

1. Požymių išskyrimas

1.1. Problematika

Dauguma mašinų mokymosi algoritmų operuoja vektoriais, sudarytais iš realiųjų skaičių. Šių vektorių komponentės mašinų mokymosi kontekste dažnai vadinamos požymiais (angl. features). Norint klasifikuoti tekstą mašinų mokymosi algoritmais, reikalingi specializuoti požymių išskyrimo (angl. feature extraction) algoritmai, sugebantys tekstą transformuoti į skaitinį pavidalą, tinkamą klasifikavimui.

Iš teksto išskiriant požymius susiduriama su keliomis problemomis:

1. Tekstas yra sudarytas iš simbolių. Simboliai yra ne skaičiai, o kategoriniai duomenys.
2. Daugumos mašinų mokymosi algoritmų įvestis yra fiksuoto ilgio vektoriai. Tuo tarpu teksto ilgis yra nepastovus.
3. Esant dideliam požymių skaičiui, mašinų mokymosi algoritmai sunaudoja daug kompiuterinių resursų. Todėl pageidaujama, kad požymių nebūtų daug. Tačiau natūraliose kalbose naudojama daug skirtingų žodžių ir gramatinių formų. Be to, teksto simbolių aibė yra didžiulė – naujausioje Unicode versijoje yra daugiau nei milijonas simbolių [Con15].

Verta pastebėti, kad to paties požymių išskyrimo algoritmo išvestis gali būti klasifikuojama įvairiais mašinų mokymosi algoritmais.

1.2. Teksto paruošimas požymių išskyrimui

Prieš išskiriant iš teksto požymius, dažnai atliekamos kelios transformacijos, supaprastinančios jo struktūrą:

1. Skaidymas į žodžius.
2. Žodžių kanonizavimas.
3. Nereikšmingų žodžių šalinimas.

1.2.1. Skaidymas į žodžius

Pirmoji transformacija – skaidymas į žodžius (angl. tokenization). Daugumoje kalbų ši transformacija triviali, tačiau yra ir išimčių, pavyzdžiui, kinų ir japonų kalbos, kuriose žodžiai tarpais neskiriami. Šios transformacijos metu žodžiai taip pat atskiriami nuo tekste esančių skyrybos ženklų. Skyrybos ženklai gali būti laikomi savarankiškais žodžiais arba iš teksto pašalinami.

1.2.2. Žodžių kanonizavimas

Žodžių kanonizavimas (angl. canonicalization) – teksto žodžių transformavimas į standartinę formą. Yra keli kanonizavimo būdai:

1. Rašybos taisymas [KMM00].

2. Raidžių dydžio suvienodinimas.
3. Šaknies nustatymas (angl. stemming), kurio metu pašalinama žodžio galūnė, priesagos ir, kai kuriais atvejais, priešdėliai.
4. Lemavimas (angl. lemmatization), kurio metu žodis suvedamas į pagrindinę jo formą, pavyzdžiui, bendratį (jei tai veiksmažodis) arba vienaskaitą (jei tai daiktavardis). Norint atlikti lemavimą, reikalingas būdas nustatyti žodžio kalbos dalį.

Pagrindinė kanonizavimo paskirtis – sumažinti požymių skaičių.

1.2.3. Nereikšmingų žodžių šalinimas

Šios transformacijos metu iš sudaryto žodžių sąrašo pašalinami itin dažnai sutinkami arba savarankiškos prasmės neturintys žodžiai (angl. stop words). Šiai transformacijai atlikti sudaromi specialūs nereikšmingų žodžių sąrašai. Į juos paprastai įtraukiami:

1. Įvardžiai.
2. Artikeliai.
3. Prielinksniai.
4. Jungtukai.

1.3. Požymių išskyrimo algoritmai

1.3.1. Žodžių maišo modelis

Žodžių maišo (angl. bag-of-words) modelyje tekstas laikomas multiaibe, kurios elementai yra žodžiai. Pagal žodžių maišo modelį tekstas t vektoriumi paverčiamas tokiu būdu:

1. Sudaromas žodynas A iš n žodžių:
$$A = \{a_1, a_2, \ldots, a_n\}.$$
2. Suskaičiuojama, kiek kartų tekste t pasikartoja kiekvienas žodis iš žodyno A. Pasikartojimų skaičiai pažymimi $c(t, a_1), c(t, a_2), \ldots, c(t, a_n)$.
3. Sudaromas n-matis vektorius:
$$(c(t, a_1), c(t, a_2), \ldots, c(t, a_n)).$$
Šis vektorius ir yra teksto skaitinis atitikmuo.

1-ame žingsnyje minimas žodynas paprastai sudaromas iš turimų mokymo duomenų.

Žodžių maišo modelis ir jo variacijos – ko gero plačiausiai naudojamas požymių išskyrimo algoritmas teksto klasifikavime. Tačiau šis modelis turi porą trūkumų:

1. Skaičiuojant žodžius prarandama informacija apie žodžių tvarką tekste.
2. Nekreipiama dėmesio į žodžių prasmę. Viena iš to pasekmių yra tai, kad sinonimai laikomi visiškai skirtingais žodžiais.

1.3.2. n-gramų maišo modelis

Paprastas būdas patobulinti žodžių maišo modelį – skaičiuoti ne tik žodžių, bet ir n-gramų dažnumus. n-grama vadinama seka iš n žodžių, einančių iš karto vienas po kito.

Palyginus su paprastu žodžių maišo modeliu, n-gramų maišo modelis turi vieną esminį privalumą. Sekos elementai turi tvarką, todėl n-gramų maišo modelis išsaugo dalį informacijos apie žodžių tvarką tekste.

Turint n žodžių žodyną, galima sudaryti n^k k-gramų. Todėl naiviai pritaikius šį modelį tekstų rinkiniui, susiduriama su požymių sprogimo (angl. feature explosion) problema – požymių skaičius smarkiai išauga. Šią problemą galima spręsti keliais būdais:

1. Apriboti n-gramų ilgį.
2. Sumažinti gaunamų požymių vektorių matą naudojant dimensijų mažinimo (angl. dimensionality reduction) metodus.

1.3.3. Tf-idf statistika

Žodžių maišo modelyje vietoj žodžio pasikartojimų skaičiaus gali būti naudojamos ir kitokios statistikos. Viena iš tokių statistikų yra tf-idf.

Tf-idf yra angliškos frazės „term frequency-inverse document frequency“ santrumpa. Ši statistika atsižvelgia ne tik į žodžio pasikartojimų skaičių, bet ir į žodžio svarbą tekstų aibei. Svarbesniais žodžiais laikomi tie, kurie aptinkami mažesniame kiekyje tekstų. Tai paremta intuicija, kad dažnai pasitaikantys žodžiai yra prasta priemonė tekstams atskirti [Spa72].

Tf-idf statistikai apskaičiuoti apibrėžiami du dydžiai:

1. **Sąvokos dažnumas (angl. term frequency)**.
   Jam apskaičiuoti yra įvairių būdų, tačiau dažniausiai naudojamas tiesiog žodžio pasikartojimų skaičius tekste:
   $$\mathrm{tf}(t, a) = c(t, a),$$
   čia t – tekstas, a – žodis, kuriam skaičiuojama statistika.
2. **Sąvokos atvirkštinis dažnumas dokumentuose (angl. inverse document frequency)**.
   Paprasčiausiu atveju ji apibrėžiama taip:
   $$\mathrm{idf}(T, a) = \ln \frac{|T|}{|p(T, a)|}, \quad p(T, a) = \{t : t \in T \wedge a \in t\},$$
   čia T – tekstų aibė, a – žodis, kuriam skaičiuojama statistika, $p(T, a)$ – aibės T poaibis, į kurį įeina tik tie tekstai, kuriuose yra žodis a.
   Ši statistika pirmą kartą buvo paminėta [Spa72] kaip priemonė dokumentų paieškos tikslumui padidinti. Tiesa, ten ji buvo vadinama sąvokos specifiškumu (angl. term specificity).

Žodžio a tf-idf statistika lygi šių dviejų dydžių sandaugai:

$$\mathrm{tfidf}(T, t, a) = \mathrm{tf}(t, a) \cdot \mathrm{idf}(T, a),$$

čia T – tekstų aibė, t – jai priklausantis tekstas.

Iš tf-idf statistikos apibrėžimo seka dvi išvados:

1. Žodžio tf-idf statistika tuo didesnė, kuo rečiau jis sutinkamas tekstuose.
2. Jei žodis aptinkamas visuose tekstuose, jo tf-idf statistika lygi nuliui.

1.3.4. Pastraipų vektoriai

2013-aisiais metais grupė kompanijoje Google dirbančių mokslininkų išleido darbą pavadinimu „Distributed Representations of Words and Phrases and their Compositionality“ [MSC+13]. Jame jie aprašė algoritmą word2vec, skirtą žodžiams transformuoti į vektorius, koduojančius jų prasmę. word2vec yra mokymosi be mokytojo metodais paremtas algoritmas, pasižymintis keliais ypatumais:

1. Jis žodžius transformuoja į taip vadinamą paskirstytą reprezentaciją (angl. distributed representation). Joje kiekvienas žodis koduojamas keliais ar net visais požymiais. Tuo tarpu anksčiau minėtame žodžių maišo modelyje kiekvienam žodžiui tenka po lygiai vieną požymį.
2. Panašios prasmės žodžius algoritmas atvaizduoja į panašius vektorius.
3. Sudaryti vektoriai koduoja sąryšius tarp žodžių. Pavyzdžiui, apytiksliai teisinga tokia lygybė:

$$\text{doc2vec}(\text{"Madrid"}) - \text{doc2vec}(\text{"Spain"}) + \text{doc2vec}(\text{"France"}) \approx \text{doc2vec}(\text{"Paris"}).$$

2014-aisiais metais kompanijoje Google dirbantys mokslininkai išleido darbą pavadinimu „Distributed Representations of Sentences and Documents“ [LM14]. Jame jie word2vec algoritmą apibendrino kintamo ilgio tekstams. Naująjį algoritmą jie pavadino pastraipų vektorių algoritmu (angl. Paragraph Vector). Priešingai nei sako pavadinimas, šis algoritmas gali būti taikomas ne tik pastraipoms, bet ir ištisiems tekstams. Šis algoritmas tekstą transformuoja į vektorių, koduojantį jo prasmę. Gautą vektorių galima panaudoti teksto klasifikavimui.

Kadangi pastraipų vektorių algoritmas paremtas mokymosi be mokytojo metodais, tai prieš požymių išskyrimą jį patį reikia apmokyti, pateikiant jam didelį kiekį teksto. Tam gali būti naudojami turimi mokymo duomenys.

1.3.5. Kalbos dalių statistikos

Finn ir Kushmerick [FK06] kaip metodą požymiams išskirti įvardina tekste naudojamų kalbos dalių skaičiavimą. Pagal šį metodą tekstas vektoriumi paverčiamas tokiu būdu:

1. Nustatoma kiekvieno tekste esančio žodžio kalbos dalis.
2. Suskaičiuojama, kiek tekste yra kiekvienos kalbos dalies žodžių.
3. Iš gautų skaičių vektorius sudaromas taip, kaip žodžių maišo modelyje.

Sudėtingiausias šio metodo žingsnis – pirmasis. Jam reikalingas algoritmas, sugebantis nustatyti žodžio kalbos dalį. Kalbos dalies nustatymas yra dar vienas iš natūralios kalbos apdorojimo uždavinių. Finn ir Kushmerick tam naudoja [Bri94] aprašytą algoritmą, kuris anglų kalboje skiria 36 kalbos dalis.

1.3.6. Teksto statistikos

Finn ir Kushmerick [FK06] mini dar vieną metodą požymiams išskirti – paprastų teksto statistikų skaičiavimą. Pagal šį metodą į teksto požymių vektorių įtraukiami tokie dydžiai:

1. Vidutinis žodžio ilgis tekste.
2. Vidutinis sakinio ilgis tekste.
3. Žodžių skaičius tekste.
4. Tam tikrų specialiai atrinktų žodžių pasikartojimo skaičiai tekste, kaip kad žodžių maišo modelyje. Tikslus šiems požymiams sudaryti naudojamų žodžių sąrašas pateiktas [FK06]. Verta pastebėti, kad dauguma jame esančių žodžių paprastai įtraukiami į nereikšmingų žodžių sąrašus (žr. poskyrį 1.2.3.).
5. Įvairių skyrybos ženklų pasikartojimų skaičius.

2. Mašinų mokymosi algoritmai

Šiame skyriuje aprašyti trys klasifikavimui tinkami mašinų mokymosi algoritmai. Visi jie priklauso mokymosi su mokytoju algoritmų grupei.

2.1. Logistinė regresija

Logistinė regresija (angl. logistic regression) yra regresijos modelis, naudojamas prognozuoti įvykio tikimybei esant tam tikroms sąlygoms. Logistinėje regresijoje sukonstruojama tokio pavidalo funkcija, išreiškianti įvykio tikimybę:

$$f(\boldsymbol{x}) = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}},$$

čia n yra požymių skaičius, $x_1, \ldots, x_n$ – požymių vektoriaus $\boldsymbol{x}$ elementai, o $w_0, \ldots, w_n$ – svoriai, parenkami pagal mokymo duomenis.

Logistinės regresijos algoritmo apmokymu vadinamas funkcijos f išraiškoje esančių svorių $w_0, \ldots, w_n$ parinkimas pagal mokymo duomenis. Norint tai padaryti, reikalinga svorių kainos funkcija, leidžianti įvertinti neatitikimą tarp modeliuojamos funkcijos ir sukonstruotos funkcijos. Apmokant logistinę regresiją naudojama tokia kainos funkcija [SJ12]:

$$J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \mathrm{Cost}\big(f_{\boldsymbol{w}}\big(\boldsymbol{x}^{(i)}\big), \boldsymbol{y}^{(i)}\big), \quad \mathrm{Cost}(f_{\boldsymbol{w}}(\boldsymbol{x}), y) = \begin{cases} -\ln(f_{\boldsymbol{w}}(\boldsymbol{x})), & \text{kai } y = 1, \\ -\ln(1 - f_{\boldsymbol{w}}(\boldsymbol{x})), & \text{kai } y = 0, \end{cases}$$

čia:

- $\boldsymbol{w} = (w_0, \ldots, w_n)$ – konstruojamos funkcijos svoriai,
- m – mokymo duomenų kiekis,
- $f_{\boldsymbol{w}}$ – konstruojama funkcija su svoriais $\boldsymbol{w}$,
- $\boldsymbol{x}^{(i)}$ – i-asis požymių vektorius iš mokymo duomenų aibės,
- $\boldsymbol{y}^{(i)}$ – i-ojo mokymo duomenų aibės elemento klasė, išreikšta skaičiumi (0 arba 1).

Svorių $w_0, \ldots, w_n$ reikšmės randamos minimizavus kainos funkciją J. Minimizavimui paprastai naudojamas gradiento metodas (angl. gradient descent). Bendru atveju gradiento metodas suranda lokalų minimumą, tačiau logistinės regresijos kainos funkcija turi tik vieną (globalų) minimumą [SJ12].

Logistinę regresiją nesunku pritaikyti binariniam klasifikavimui. Pažymėkime klases raidėmis A ir B. Pasitelkę logistinę regresiją galime sukonstruoti funkciją, kuri įvertina tikimybę, kad požymių vektorius priklauso klasei A. Gautą tikimybę lyginame su norima slenkstine reikšme (paprasčiausiu atveju ji gali būti lygi $\frac{1}{2}$). Jei tikimybė didesnė už slenkstinę reikšmę, laikome, kad požymių vektorius priklauso klasei A. Priešingu atveju laikome, kad požymių vektorius priklauso klasei B.

Logistine regresija duomenis klasifikuojant į daugiau nei dvi klases, naudojamas metodas „vienas prieš visus“ (angl. one-vs-all). Jo veikimo principas yra toks:

1. Kiekvienai klasei atskirai apmokoma logistinės regresijos funkcija, sugebanti atpažinti tos klasės narius.

2. Norint nustatyti požymių vektoriaus klasę, su kiekviena sudaryta funkcija įvertinama jo priklausymo atitinkamai klasei tikimybė.
3. Požymių vektorius priskiriamas tai klasei, su kurios funkcija buvo gautas didžiausias įvertis.

2.2. Dirbtinis neuroninis tinklas

Dirbtinis neuroninis tinklas yra mašinų mokymosi algoritmas, sukurtas mėgdžiojant gyvūnų nervų sistemų struktūrą. Pirmą kartą jis buvo paminėtas W. S. McCulloch ir W. Pitts darbe [MP43].

Smulkiausias dirbtinio neuroninio tinklo vienetas – neuronas. Neuronas yra darinys, skaičiuojantis tokio pavidalo funkciją:

$$g(\boldsymbol{x}) = k\left(w_0 + \sum_{i=1}^{n} w_i x_i\right),$$

čia n – neurono įvesčių skaičius, $x_1, \ldots, x_n$ – įvesčių vektoriaus $\boldsymbol{x}$ elementai, $w_0, \ldots, w_n$ – svoriai, parenkami pagal mokymo duomenis, o k – aktyvacijos funkcija (angl. activation function). Dažnai naudojamos tokios aktyvacijos funkcijos:

1. Logistinė funkcija:
$$k(x) = \frac{1}{1 + e^{-x}}.$$
2. Hiperbolinis tangentas.
3. Išlyginta tiesinė funkcija (angl. rectified linear unit):
$$k(x) = \begin{cases} 0, & \text{kai } x < 0, \\ x, & \text{kai } x \geq 0. \end{cases}$$

Tačiau bendru atveju aktyvacijos funkcija gali būti bet kokia netiesinė funkcija. Verta pastebėti, kad pavienis neuronas su logistine aktyvacijos funkcija ekvivalentus logistinei regresijai.

Neuroninis tinklas susideda iš neuronų, sugrupuotų į n tarpusavyje sujungtų sluoksnių. i-ojo sluoksnio neuronų įvestys sujungiamos su visų $(i-1)$-ojo sluoksnio neuronų išvestimis. Tam pačiam sluoksniui priklausantys neuronai nėra sujungiami.

Norint apskaičiuoti neuroninio tinklo išvestį su požymių vektoriumi $\boldsymbol{x}$, 1-ojo sluoksnio neuronų įvestys prilyginamos $\boldsymbol{x}$ elementams. Apskaičiuojamos 1-ojo sluoksnio neuronų išvestys. Iš jų sudaromas naujas vektorius y_1, turintis tiek elementų, kiek yra neuronų 1-ajame sluoksnyje. Analogiška operacija kartojama su tolimesniais sluoksniais, laikant i-ojo sluoksnio išvestis $(i+1)$-ojo sluoksnio įvestimis. Neuroninio tinklo išvestis yra paskutinio (n-ojo) jo sluoksnio išvestis y_n.

Paskutinis neuroninio tinklo sluoksnis vadinamas išvesties sluoksniu (angl. output layer). Visi iki jo esantys sluoksniai vadinami paslėptais sluoksniais (angl. hidden layers). Paprasčiausiu atveju paslėptų sluoksnių gali nebūti, t.y., neuroninis tinklas gali būti tik iš vieno sluoksnio. Taip pat kartais skiriamas įvesties sluoksnis (angl. input layer), kuris nėra sudarytas iš neuronų, ir kurio

išvestys tiesiog prilyginamos požymių vektoriaus elementams. Neuroninio tinklo sluoksnių skaičius, kiekviename iš jų esančių neuronų skaičius ir naudojamos aktyvacijos funkcijos kartu paėmus apibūdina taip vadinamą neuroninio tinklo architektūrą.

Neuroninio tinklo apmokymu vadinamas tinklą sudarančių neuronų svorių parinkimas. Ilgą laiką nebuvo žinomas algoritmas, galintis greitai apmokyti dirbtinius neuroninius tinklus. 1974-aisias metais P. Werbos savo daktaro disertacijoje [Wer74] aprašė taip vadinamą atgalinio skleidimo (angl. backpropagation) algoritmą. Jis leidžia apskaičiuoti neuroninio tinklo kainos funkcijos gradientą, kurį toliau galima panaudoti optimizuojant tinklo svorius gradiento metodu. Pagrindinis šio mokymo metodo trūkumas tas, kad jis suranda lokalų minimumą.

Kadangi neuroninio tinklo išvestis yra ne skaliaras, o vektorius, neuroninį tinklą nesunku pritaikyti klasifikavimui ne tik į dvi, bet ir į daugiau klasių. Tam naudojamas taip vadinamas one-hot kodavimas:

1. Klasėms priskiriami numeriai nuo 1 iki n, kur n – klasių skaičius.
2. Priklausymas klasei, kurios numeris i, išreiškiamas n-mačiu vektoriumi, kurio visos komponentės, išskyrus i-ąją, lygios nuliui, o i-oji – vienetui.

Kai klasifikuojama į n klasių, neuroninio tinklo išvesties sluoksnis turi turėti n neuronų. Jų aktyvacijai paprastai naudojama taip vadinama softmax funkcija, kuri viso sluoksnio išvestis normalizuoja taip, kad jų suma būtų lygi vienetui. Tokio sluoksnio išvestis galima interpretuoti kaip priklausymo klasėms tikimybes.

2.3. Atraminių vektorių mašina

Atraminių vektorių mašina – mašinų mokymosi algoritmas, kurį praeito amžiaus 7-ajame dešimtmetyje išrado V. Vapnik ir A. Červonenkis. Šiais laikais paprastai naudojamas patobulintas jo variantas, aprašytas [CV95].

Klasifikavimo uždavinį galima suformuluoti taip: požymių erdvėje atrasti paviršių, skiriantį skirtingoms klasėms priklausiančius taškus. Tokių paviršių yra be galo daug. Intuityvu manyti, kad geriausias iš jų yra tas, kuris išlaiko didžiausią atstumą nuo jam artimiausių taškų. Tuo remiantis ir buvo sukurtos atraminių vektorių mašinos. Atraminių vektorių mašinos veikimo principas yra toks:

1. Jei taškus galima atskirti hiperplokštuma, atraminių vektorių mašina suranda tokią hiperplokštumą, kuri išlaiko didžiausią atstumą nuo jai artimiausių taškų.
2. Jei taškų neįmanoma atskirti hiperplokštuma, atraminių vektorių mašina suranda tokią hiperplokštumą, kuri minimizuoja neteisingai suklasifikuotų taškų atstumų nuo jos sumą.

Klasifikuojant taškus ne visada galima atskirti hiperplokštuma. Tačiau kartais tai galima padaryti transformavus juos į daugiau dimensijų turinčios erdvės taškus. Naudojant atraminių vektorių mašiną, nebūtina šios transformacijos atlikti išreikštinai. Atraminių vektorių mašinos algoritmą

galima suformuluoti taip, kad požymių vektoriai į matematines išraiškas įeitų tik kaip taip vadinamos branduolio funkcijos argumentai. Paprasčiausiu atveju branduolio funkciją atstoja vektorių skaliarinė sandauga. Tačiau ją galima pakeisti sudėtingesne funkcija, kuri minėtąją transformaciją atlieka neišreikštinai (t.y., neapskaičiuodama papildomų koordinačių). Tai vadinama branduolio gudrybe (angl. kernel trick).

Kaip ir logistinė regresija, atraminių vektorių mašinos savaime duomenis gali klasifikuoti tik į dvi klases. Esant daugiau nei dviems klasėms, naudojamas „vienas prieš visus“ metodas.

3. Algoritmų palyginimas

Siekiant palyginti ankstesniuose skyriuose aprašytus algoritmus, buvo atliktas eksperimtas. Šiame skyriuje aprašyta eksperimento atlikimo metodika bei pateikti eksperimento metu gauti rezultatai.

3.1. Eksperimento aprašymas

Kriterijus, pagal kurį lyginami klasifikavimo algoritmai – klasifikavimo tikslumas. Algoritmui klasifikavimo tikslumas skaičiuojamas taip:

$$\text{tikslumas} = \frac{|\{\text{teisingai suklasifikuoti testavimo duomenys}\}|}{|\{\text{visi testavimo duomenys}\}|}.$$

Kitaip tariant, algoritmo klasifikavimo tikslumas – dalis testavimo duomenų aibės elementų, kuriems tas algoritmas klasę priskyrė teisingai.

Eksperimento metu buvo lyginami tokie požymių išskyrimo algoritmai:

1. Žodžių maišo modelis.
2. n-gramų maišo modelis.
3. Žodžių maišo modelis su tf-idf statistikomis vietoj pasikartojimų skaičiaus.
4. n-gramų maišo modelis su tf-idf statistikomis vietoj pasikartojimų skaičiaus.
5. Pastraipų vektorių algoritmas.
6. Kalbos dalių statistikos.
7. Teksto statistikos.

Žodynai 1–4 algoritmams buvo sudaryti iš transformuotų mokymo duomenų. Kiekvieno žodyno dydis buvo apribotas iki 5000 elementų. Pastraipų vektorių algoritmas buvo apmokytas naudojant transformuotus mokymo duomenis. Jis buvo išmokytas sudaryti 300 požymių turinčius vektorius.

Klasifikavimo tikslumas priklauso ne tik nuo algoritmo, bet ir nuo to, kaip tekstas transformuojamas prieš apdorojimą požymių išskyrimo algoritmu. Todėl su 1–5 algoritmais matavimai buvo atlikti keturis kartus:

1. Su nepakeistais tekstais.
2. Su tekstais, iš kurių pašalinti nereikšmingi žodžiai.
3. Su tekstais, kurių žodžiai kanonizuoti šaknies nustatymo būdu.
4. Su tekstais, kurių žodžiai iš pradžių kanonizuoti, o tada pašalinti nereikšmingi žodžiai.

Visų išvardintų požymių išskyrimo algoritmų išvestis buvo klasifikuojama trimis skirtingais mašinų mokymosi algoritmais:

1. Logistine regresija.

2. Dirbtiniu neuroniniu tinklu.

3. Atraminių vektorių mašina.

Eksperimento metu visuose atvejuose buvo naudojamas tokios pačios architektūros neuroninis tinklas. Buvo pasirinkta tokia architektūra:

1. Vienas paslėptas sluoksnis su 30 neuronų. Jų aktyvacijos funkcija – hiberbolinis tangentas.
2. Išvesties sluoksnyje tiek neuronų, kiek yra klasių duomenų aibėje. Jų aktyvacijos funkcija – softmax.

Apmokant neuroninius tinklus reguliarizavimui buvo naudojamas išmetimo (angl. dropout) metodas, aprašytas [SHK+14].

Apmokant neuroninius tinklus pradiniai svoriai jiems parenkami atsitiktinai. Be to, neuroninio tinklo kainos funkcija dažnai turi lokalių minimumų. Todėl tokį patį tinklą apmokius kelis kartus galima gauti nevienodą klasifikavimo tikslumą. Siekiant sumažinti šios problemos įtaką eksperimento rezultatams, su kiekvienu požymių išskyrimo algoritmu buvo apmokyti 5 neuroniniai tinklai. Palyginimui naudojamas jų klasifikavimo tikslumų vidurkis.

Eksperimento metu buvo naudojamos atraminių vektorių mašinos su radialine bazine branduolio funkcija. Duomenų vektoriai, pateikiami atraminių vektorių mašinai, buvo normalizuojami taip, kad kiekvieno požymio vidurkis būtų lygus nuliui, o standartinis nuokrypis – vienetui. Eksperimento metu buvo pastebėta, kad kai kuriais atvejais atraminių vektorių mašina pasiekia didesnį tikslumą, kai duomenys nėra normalizuoti. Todėl buvo nuspręsta į palyginimą įtraukti ir atraminių vektorių mašiną, kuriai pateikiami nenormalizuoti duomenys.

Eksperimentui naudotos programos kodas pateiktas 1-ame priede („Programos kodas“).

3.2. Duomenų aibės

Eksperimento metu buvo naudojamos dvi duomenų aibės:

1. **Filmų apžvalgų duomenų aibė**.
   Ji sudaryta iš 2000 filmų apžvalgų, suskirstytų į teigiamas ir neigiamas. Pirmą kartą duomenų aibė buvo panaudota [PLV02]. Šiame darbe naudojama jos versija, publikuota kartu su [PL04].
2. **Brown duomenų aibė**.
   Tai yra 500 angliškų tekstų rinkinys, kurį praeito amžiaus 7-ajame dešimtmetyje surinko H. Kučera ir W. N. Francis iš Brown universiteto Jungtinėse Amerikos Valstijose. Jame tekstai pagal tematiką suskirstyti į 15 kategorijų. Šiame darbe naudojama naujausia rinkinio versija, publikuota kartu su [FK79].

Abi duomenų aibės sudarytos iš angliškų tekstų, todėl algoritmuose naudojami anglų kalbos žodžių sąrašai ir anglų kalbai skirtas kalbos dalies nustatymo algoritmas.

Eksperimento metu abi duomenų aibės buvo padalintos į mokymo ir testavimo duomenis santykiu 80:20. Šis padalinimas buvo atliktas vieną kartą. Kitaip tariant, visuose atvejuose mašinų mokymo algoritmai buvo apmokyti požymiais, išskirtais iš tų pačių tekstų.

3.3. Rezultatai

Filmų apžvalgų duomenų aibės klasifikavimo tikslumo įverčiai pateikti 1-oje lentelėje. Brown duomenų aibės klasifikavimo tikslumo įverčiai pateikti 2-oje lentelėje. Abiejose lentelėse didesni įverčiai paryškinti ryškesne spalva.

3.4. Rezultatų analizė

Iš filmų apžvalgų duomenų aibės klasifikavimo rezultatų matome, kad:

1. Didžiausias tikslumas buvo 86,5 %. Jis buvo pasiektas pritaikius n-gramų maišo algoritmą su tf-idf statistika kanonizuotam ir nereikšmingų žodžių neturinčiam tekstui. Klasifikavimui buvo panaudota atraminių vektorių mašina.
2. Naudojant žodžių maišo modelį ir jo variacijas buvo pasiektas didesnis tikslumas nei kitais algoritmais.
3. Mašinų mokymo algoritmo pasirinkimas nedarė didelės įtakos klasifikavimo tikslumui.
4. Visais atvejais tiksliau klasifikavo atraminių vektorių mašina, kuriai buvo pateikiami normalizuoti duomenys.

Iš Brown duomenų aibės klasifikavimo rezultatų matome, kad:

1. Didžiausias tikslumas buvo 60 %. Jis buvo pasiektas dviem atvejais:
    1.1. Pritaikius n-gramų maišo modelį kanonizuotam ir nereikšmingų žodžių neturinčiam tekstui.
    1.2. Pritaikius pastraipų vektorių algoritmą neapdorotam tekstui.

    Abiem atvejais klasifikavimui buvo panaudota logistinė regresija.
2. Pastraipų vektorių algoritmas leido pasiekti didelį klasifikavimo tikslumą su visais mašinų mokymo algoritmais.
3. Žodžių maišo ir n-gramų maišo modeliai su tf-idf statistika davė prastesnių rezultatų nei modeliai su pasikartojimų skaičiumi.
4. Naudojant žodžių maišo ir n-gramų maišo modelius tiksliau klasifikavo atraminių vektorių mašina, kuriai buvo pateikiami nenormalizuoti duomenys.

Iš abiejų duomenų aibių klasifikavimo rezultatų matome, kad:

1. Iš požymių išskyrimo algoritmų prasčiausiai pasirodė kalbos dalių statistikų ir teksto statistikų algoritmai.

1 lentelė. Filmų apžvalgų duomenų aibės klasifikavimo tikslumo įverčiai procentais

	Žodžių kanonizavimas	**Nereikšmingų žodžių šalinimas**	**Logistinė regresija**	**DNT (5 tinklų vidurkis)**	**AVM be normalizavimo**	**AVM su normalizavimu**
Žodžių maišas	Ne	Ne	80,25	84,15	76,75	81
	Ne	Taip	84,25	83,35	74	82
	Taip	Ne	83,75	83,1	77,5	82,5
	Taip	Taip	83	81,55	76,25	81,5
n-gramų maišas	Ne	Ne	80,75	83,85	77,75	84,25
	Ne	Taip	81,25	83,7	75,75	82,5
	Taip	Ne	80,75	84,15	78,25	82
	Taip	Taip	83	83,8	78,25	81,75
Žodžių maišas su tf-idf statistika	Ne	Ne	82,5	84,2	69,25	83
	Ne	Taip	84,25	82,95	72,25	84
	Taip	Ne	83	82,2	70	83,75
	Taip	Taip	84,5	82,1	74	83,25
n-gramų maišas su tf-idf statistika	Ne	Ne	83	84,7	68,75	84,25
	Ne	Taip	83,25	83,65	72,5	85,75
	Taip	Ne	81,25	83,5	69	84,75
	Taip	Taip	83,75	83,75	74,25	86,5
Pastraipų vektoriai	Ne	Ne	75,25	75,35	74,5	76,5
	Ne	Taip	78,75	78,6	77,5	80,75
	Taip	Ne	77	75,35	72,75	77,5
	Taip	Taip	78,5	78,75	78	79,5
Kalbos dalių statistikos	Ne	Ne	62	61,5	51,5	62,25
Teksto statistikos	Ne	Ne	69,5	65,05	57	65,75

2 lentelė. Brown duomenų aibės klasifikavimo tikslumo įverčiai procentais

	Žodžių kanonizavimas	**Nereikšmingų žodžių šalinimas**	**Logistinė regresija**	**DNT (5 tinklų vidurkis)**	**AVM be normalizavimo**	**AVM su normalizavimu**
Žodžių maišas	Ne	Ne	50	50,2	42	30
	Ne	Taip	53	51,8	43	29
	Taip	Ne	54	47,2	40	25
	Taip	Taip	59	46,8	45	24
n-gramų maišas	Ne	Ne	50	53,6	36	38
	Ne	Taip	57	48,8	42	32
	Taip	Ne	54	53,2	36	41
	Taip	Taip	60	46,8	44	34
Žodžių maišas su tf-idf statistika	Ne	Ne	43	47	16	27
	Ne	Taip	44	51	16	27
	Taip	Ne	43	48,8	16	26
	Taip	Taip	44	48	16	25
n-gramų maišas su tf-idf statistika	Ne	Ne	41	49,2	16	37
	Ne	Taip	46	51,2	16	29
	Taip	Ne	43	54,2	16	37
	Taip	Taip	46	50	16	32
Pastraipų vektoriai	Ne	Ne	60	54,6	29	58
	Ne	Taip	54	54,2	30	59
	Taip	Ne	54	52,4	28	56
	Taip	Taip	54	50,6	26	57
Kalbos dalių statistikos	Ne	Ne	35	44,2	16	47
Teksto statistikos	Ne	Ne	29	45	16	43

2. Nereikšmingų žodžių šalinimas daugeliu atvejų padidino klasifikavimo tikslumą.
3. Žodžių kanonizavimas klasifikavimo tikslumą kai kuriais atvejais padidino, o kai kuriais – sumažino.

Iš gautų rezultatų galima daryti tokias išvadas:

1. Nėra vieno geriausio algoritmo teksto klasifikavimui. Skirtingomis sąlygomis didžiausią tikslumą pasiekia skirtingi algoritmai.
2. Klasifikuojant tekstą į dvi klases, pasirinktas mašinų mokymo algoritmas nėra svarbus. Visais pasiekiamas panašus tikslumas.
3. Klasifikuojant tekstą į daugiau nei dvi klases, didžiausią tikslumą pasiekti leidžia logistinė regresija.
4. Klasifikuojant tekstą į dvi klases, didžiausią tikslumą pasiekti leidžia žodžių maišo modelis ir jo variacijos.
5. Klasifikuojant tekstą į daugiau nei dvi klases, pastraipų vektorių algoritmas leidžia pasiekti didelį tikslumą bet kuriuo mašinų mokymo algoritmu.
6. Klasifikuojant tekstą atraminių vektorių mašina ir taikant žodžių maišo modelį, kai kuriais atvejais didesnis tikslumas gaunamas duomenų nenormalizavus.
7. Nereikšmingų žodžių pašalinimas iš teksto prieš požymių išgavimą padidina klasifikavimo tikslumą.
8. Žodžių kanonizavimas klasifikavimo tikslumą kai kuriais atvejais gali padidinti, o kai kuriais – sumažinti.

4. Alternatyvūs sprendimai

Šiame darbe išnagrinėti ne visi teksto klasifikavimo būdai. Darbe buvo orientuojamasi į požymių išskyrimo algoritmus, kurių išvestis – fiksuoto ilgio skaičių vektoriai. Šiame skyriuje trumpai aprašyta pora alternatyvių teksto klasifikavimo būdų.

4.1. Teksto branduoliai

Atraminių vektorių mašinos ir kiti branduolio metodai pasižymi įdomia savybe: jiems pateikiamų duomenų nebūtina paversti skaičių vektoriais. To pasiekti galima pasinaudojus branduolio gudrybe ir įvedus branduolio funkciją, kuri duomenų aibės elementus apdoroja tiesiogiai, nepaversdama jų vektoriais.

Teksto branduolys (angl. string kernel) – branduolio funkcija, kuri operuoja ne skaičiais, o tekstais. Teksto branduoliai pirmą kartą buvo paminėti [LSS+02]. Panašiai kaip vektorių skaliarinė sandauga apskaičiuoja vektorių panašumą, teksto branduolys apskaičiuoja tekstų panašumą. Naudojant teksto branduolį, tekstus galima klasifikuoti jų nepavertus skaičių vektoriais.

4.2. Algoritmai, sudarantys kintamo ilgio požymių vektorius

Yra mašinų mokymosi algoritmų, kurie sugeba apdoroti kintamo ilgio įvestį. Tokio algoritmo pavyzdys – rekurentinis neuroninis tinklas (angl. recurrent neural network). Klasifikavimui naudojant tokį mašinų mokymosi algoritmą, požymius iš teksto galima išgauti alternatyviu būdu:

1. Tekstą suskaidyti į mažesnius vienetus. Tai gali būti simboliai, žodžiai ar net sakiniai.
2. Kiekvieną vienetą atskirai transformuoti į požymių vektorių. Jei transformuojami vienetai yra simboliai, tam galima naudoti one-hot kodavimą. Jei transformuojami vienetai yra žodžiai, tam galima naudoti poskyryje 1.3.4. minėtą word2vec algoritmą.

Rezultatai ir išvados

Atliekant darbą buvo pasiekta šių rezultatų:

1. Išanalizuoti būdai iš teksto išskirti požymius.
2. Išanalizuoti trys mašinų mokymosi algoritmai, tinkami klasifikavimui.
3. Remiantis teorine darbo dalimi, atliktas eksperimentas. Jo metu išmatuotas kiekvieno analizuoto algoritmo klasifikavimo tikslumas.
4. Remiantis eksperimento rezultatais atliktas algoritmų palyginimas. Padarytos išvados, galinčios padėti pasirinkti algoritmą.

Iš eksperimento rezultatų galima daryti tokias išvadas:

1. Nėra vieno geriausio algoritmo teksto klasifikavimui. Skirtingomis sąlygomis didžiausią tikslumą pasiekia skirtingi algoritmai.
2. Klasifikuojant tekstą į dvi klases, pasirinktas mašinų mokymo algoritmas nėra svarbus. Visais pasiekiamas panašus tikslumas.
3. Klasifikuojant tekstą į daugiau nei dvi klases, didžiausią tikslumą pasiekti leidžia logistinė regresija.
4. Klasifikuojant tekstą į dvi klases, didžiausią tikslumą pasiekti leidžia žodžių maišo modelis ir jo variacijos.
5. Klasifikuojant tekstą į daugiau nei dvi klases, pastraipų vektorių algoritmas leidžia pasiekti didelį tikslumą bet kuriuo mašinų mokymo algoritmu.
6. Klasifikuojant tekstą atraminių vektorių mašina ir taikant žodžių maišo modelį, kai kuriais atvejais didesnis tikslumas gaunamas duomenų nenormalizavus.
7. Nereikšmingų žodžių pašalinimas iš teksto prieš požymių išgavimą padidina klasifikavimo tikslumą.
8. Žodžių kanonizavimas klasifikavimo tikslumą kai kuriais atvejais gali padidinti, o kai kuriais – sumažinti.

Šaltiniai

[AKC+00] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinos, George Paliouras, and Constantine D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. *arXiv preprint cs/0006013*, 2000. URL: https://arxiv.org/pdf/cs/0006013.pdf (visited on 2016-05-28).

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly Media, Inc., 2009. URL: http://14.139.183.250:8080/bitstream/handle/123456789/120/Natural%20Language%20Processing%20with%20Python%20(2009).pdf?sequence=1 (visited on 2016-05-28).

[Bri94] Eric Brill. Some advances in transformation-based part of speech tagging. *arXiv preprint cmp-lg/9406010*, 1994. URL: https://arxiv.org/pdf/cmp-lg/9406010.pdf (visited on 2016-05-28).

[Con15] The Unicode Consortium. *The Unicode standard, version 8.0.0*, 2015.

[CS15] Alex J. Champandard and Spyridon Samothrakis. sknn: deep neural networks without the learning cliff. nucl.ai Conference 2015. 2015.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. URL: http://csee.wvu.edu/~xinl/library/papers/comp/ML/svm.pdf (visited on 2016-05-28).

[DAC+01] Olivier De Vel, Alison Anderson, Malcolm Corney, and George Mohay. Mining e-mail content for author identification forensics. *ACM sigmod record*, 30(4):55–64, 2001. URL: https://eprints.qut.edu.au/8019/1/8019.pdf (visited on 2016-05-28).

[Dun94] Ted Dunning. *Statistical identification of language*. Computing Research Laboratory, New Mexico State University, 1994. URL: https://www.researchgate.net/profile/Ted_Dunning/publication/2263394_Statistical_Identification_of_Language/links/0deec51cb2675ae546000000.pdf (visited on 2016-05-28).

[FK06] Aidan Finn and Nicholas Kushmerick. Learning to classify documents according to genre. *Journal of the American Society for Information Science and Technology*, 57(11):1506–1518, 2006. URL: https://www.sfu.ca/~mtaboada/lot/readings/Finn_Kushmerick_2006.pdf (visited on 2016-05-28).

[FK79] W. Nelson Francis and Henry Kučera. Brown corpus manual. *Brown university*, 1979. URL: http://www.hit.uib.no/icame/brown/bcm.html (visited on 2016-05-28).

[JOP+16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: open source scientific tools for Python. 2001–2016. URL: https://www.scipy.org/ (visited on 2016-05-28).

[KMM00] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on TFIDF ranking. In *Proceedings of the 23rd annual international ACM SIGIR conference on research and development in information retrieval*. ACM, 2000, pp. 357–359. URL: https://people.cs.pitt.edu/~behrang/pubs/sigir2000.pdf (visited on 2016-05-28).

[Lis96] Pierre Lison. An introduction to machine learning, 1996. URL: http://folk.uio.no/plison/pdfs/talks/machinelearning.pdf (visited on 2016-05-28).

[LM14] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014. URL: https://arxiv.org/pdf/1405.4053.pdf (visited on 2016-05-28).

[LSS+02] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *The journal of machine learning research*, 2:419–444, 2002. URL: http://machinelearning.wustl.edu/mlpapers/paper_files/LodhiSSCW02.pdf (visited on 2016-05-28).

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. URL: https://www.informatics.indiana.edu/jbollen/I501F13/readings/mccullochpitts1943.pdf (visited on 2016-05-28).

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems 26*. C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors. Curran Associates, Inc., 2013, pp. 3111–3119. URL: https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf (visited on 2016-05-28).

[PL04] Bo Pang and Lillian Lee. A sentimental education: sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2004, p. 271. URL: https://arxiv.org/pdf/cs/0409058.pdf (visited on 2016-05-28).

[PLV02] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on empirical methods in natural language processing-volume 10*. Association for Computational Linguistics, 2002, pp. 79–86. URL: https://arxiv.org/pdf/cs/0205070.pdf (visited on 2016-05-28).

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, et al. Scikit-learn: machine learning in Python. *The journal of machine learning research*, 12:2825–2830, 2011. URL: https://arxiv.org/pdf/1201.0490.pdf (visited on 2016-05-28).

[ŘS10] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 workshop on new challenges for NLP frameworks*. ELRA, Valletta, Malta, 2010-05-22, pp. 45–50. URL: https://is.muni.cz/publication/884893/en (visited on 2016-05-28).

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. URL: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf (visited on 2016-05-28).

[SJ12] Ingmar Schuster and Patrick Jähnichen. Classification using logistic regression, 2012. URL: http://asv.informatik.uni-leipzig.de/uploads/document/file_link/530/TMI05.2_logistic_regression.pdf (visited on 2016-05-28).

[Spa72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972. URL: http://nlp.cs.swarthmore.edu/~richardw/papers/sparckjones1972-statistical.pdf (visited on 2016-05-28).

[VCV11] Stefan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011. URL: https://arxiv.org/pdf/1102.1523.pdf (visited on 2016-05-28).

[VD95] Guido Van Rossum and Fred L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995. URL: http://ft-sipil.unila.ac.id/dbooks/Python%20Reference%20Manual.pdf (visited on 2016-05-28).

[Wer74] Paul Werbos. Beyond regression: new tools for prediction and analysis in the behavioral sciences, 1974.

1 priedas. Programos kodas

Šiame priede pateiktas eksperimentui naudotos programos kodas.

1.1 priedas. Failas `main.py`

```
#!/usr/bin/env python3

import os
import json

import nltk

from utils      import identity, flatten, flatmap
from dataset    import brown, movie
from preproc    import make_word_list, make_doc, filter_stopwords, stem, \
    stem_filter_stopwords, make_pos_list, make_sentence_list, PreprocStep
from features   import bag_of_words, bag_of_ngrams, tfidf, tfidf_of_ngrams, \
    doc2vec, statistics, FeatureStep
from classifier import build_logistic, build_nn, build_svm_raw, \
    build_svm_normalized, ClassifyStep

datasets = [brown, movie]

def make_list_preproc_steps(data):
    return [PreprocStep('list_nothing',   data, make_word_list(identity)),
            PreprocStep('list_stop',      data, make_word_list(filter_stopwords)),
            PreprocStep('list_stem',      data, make_word_list(stem)),
            PreprocStep('list_stem_stop', data, make_word_list(stem_filter_stopwords))]
list_preproc_steps = flatmap(make_list_preproc_steps, datasets)

def make_doc_preproc_steps(data):
    return [PreprocStep('doc_nothing',   data, make_doc(identity)),
            PreprocStep('doc_stop',      data, make_doc(filter_stopwords)),
            PreprocStep('doc_stem',      data, make_doc(stem)),
            PreprocStep('doc_stem_stop', data, make_doc(stem_filter_stopwords))]
doc_preproc_steps = flatmap(make_doc_preproc_steps, datasets)

pos_preproc_steps = [PreprocStep('pos', data, make_pos_list)
                     for data
                     in datasets]

sentence_preproc_steps = [PreprocStep('sentence', data, make_sentence_list)
                          for data
                          in datasets]

def make_list_feature_steps(preproc_step):
    return [FeatureStep('words',        preproc_step, bag_of_words),
            FeatureStep('ngrams',       preproc_step, bag_of_ngrams),
            FeatureStep('tfidf',        preproc_step, tfidf),
            FeatureStep('tfidf_ngrams', preproc_step, tfidf_of_ngrams)]
list_feature_steps = flatmap(make_list_feature_steps, list_preproc_steps)

vec_feature_steps = [FeatureStep('vec', preproc_step, doc2vec)
                     for preproc_step
                     in doc_preproc_steps]

pos_feature_steps = [FeatureStep('pos', preproc_step, bag_of_words)
```

```
                    for preproc_step
                    in pos_preproc_steps]

sentence_feature_steps = [FeatureStep('stats', preproc_step, statistics)
                          for preproc_step
                          in sentence_preproc_steps]

all_feature_steps = flatten([list_feature_steps,
                             vec_feature_steps,
                             pos_feature_steps,
                             sentence_feature_steps])

def make_cases(feature_step):
    nn_cases = [ClassifyStep('nn{}'.format(number), feature_step, build_nn)
                for number
                in range(1, 6)]
    other_cases = [ClassifyStep('log',      feature_step, build_logistic),
                   ClassifyStep('svm_raw',  feature_step, build_svm_raw),
                   ClassifyStep('svm_norm', feature_step, build_svm_normalized)]
    return nn_cases + other_cases
cases = {case.label: case for case in flatmap(make_cases, all_feature_steps)}

def main():
    metrics = {label: case.metrics() for (label, case) in cases.items()}
    os.makedirs('results', exist_ok=True)
    with open('results/metrics.json', 'w') as json_file:
        json.dump(metrics, json_file, sort_keys=True, indent=4)

if __name__ == '__main__':
    main()
```

1.2 priedas. Failas `utils.py`

```
import functools
import itertools
import os
import pickle

def identity(value):
    return value

def flatten(iterable):
    return itertools.chain.from_iterable(iterable)

def flatmap(function, iterable):
    return flatten(map(function, iterable))

class MethodCache:
    def __init__(self, directory):
        self.directory = directory

    def __call__(self, method):
        def load_or_call(wrapped_self):
            file_name = '{}.{}'.format(wrapped_self.label, method.__name__)
            file_path = os.path.join(self.directory, file_name)
            try:
                return self.load(file_path)
            except Exception:
                value = method(wrapped_self)
```

```
                self.dump(file_path, value)
                return value
        return lazy(load_or_call)

    @staticmethod
    def load(file_path):
        with open(file_path, 'rb') as in_file:
            return pickle.load(in_file)

    def dump(self, file_path, value):
        self.ensure_exists()
        with open(file_path, 'wb') as out_file:
            pickle.dump(value, out_file, pickle.HIGHEST_PROTOCOL)

    def ensure_exists(self):
        os.makedirs(self.directory, exist_ok=True)

lazy = functools.lru_cache(1)

cache = MethodCache('cache/')
```

1.3 priedas. Failas `dataset.py`

```
import itertools

import nltk
from sklearn.preprocessing import LabelEncoder

from utils import flatmap, cache

def load_or_download(corpus, identifier):
    try:
        corpus.ensure_loaded()
    except LookupError:
        nltk.download(identifier)

class DataSet:
    def __init__(self, corpus, identifier):
        self.corpus = corpus
        self.label = identifier
        load_or_download(corpus, identifier)

    @cache
    def train_fileids(self):
        test = set(self.test_fileids())
        return list(itertools.filterfalse(test.__contains__,
                                          self.corpus.fileids()))

    @cache
    def test_fileids(self):
        grouped = itertools.groupby(self.corpus.fileids(),
                                    self.corpus.categories)
        def tail_slice(group):
            fileids = list(group[1])
            count = round(len(fileids) * 0.2)
            return fileids[:count]
        return list(flatmap(tail_slice, grouped))

    def make_labels(self, fids):
```

```
        categories = self.corpus.categories
        doc_categories = [categories(fid)[0] for fid in fids]
        encoder = LabelEncoder().fit(categories())
        return encoder.transform(doc_categories)

    @cache
    def train_labels(self):
        return self.make_labels(self.train_fileids())

    @cache
    def test_labels(self):
        return self.make_labels(self.test_fileids())

brown = DataSet(nltk.corpus.brown, 'brown')

movie = DataSet(nltk.corpus.movie_reviews, 'movie_reviews')
```

1.4 priedas. Failas `preproc.py`

```
import itertools

import nltk
from gensim.models.doc2vec import TaggedDocument

from utils   import cache
from dataset import load_or_download

load_or_download(nltk.corpus.stopwords, 'stopwords')
stopwords = set(nltk.corpus.stopwords.words('english'))

stemmer = nltk.stem.SnowballStemmer('english')

def pos_tag(tokens):
    try:
        return nltk.pos_tag(tokens)
    except LookupError:
        nltk.download('averaged_perceptron_tagger')
        return nltk.pos_tag(tokens)

def make_word_list(preproc):
    return lambda corpus, fid: preproc(corpus.words(fid))

def make_doc(preproc):
    return lambda corpus, fid: TaggedDocument(preproc(corpus.words(fid)), [fid])

def filter_stopwords(words):
    return list(itertools.filterfalse(stopwords.__contains__, words))

def stem(words):
    return list(map(stemmer.stem, words))

def stem_filter_stopwords(words):
    return filter_stopwords(stem(words))

def make_pos_list(corpus, fid):
    tagged_words = pos_tag(corpus.words(fid))
    return [pos for (word, pos) in tagged_words]

def make_sentence_list(corpus, fid):
```

```
    try:
        return corpus.sents(fid)
    except:
        nltk.download('punkt')
        return corpus.sents(fid)

class PreprocStep:
    def __init__(self, label, data_set, preprocessor):
        self.label = '{}.{}'.format(data_set.label, label)
        self.data_set = data_set
        self.preprocessor = preprocessor

    @cache
    def train_input(self):
        return [self.preprocessor(self.data_set.corpus, fid)
                for fid
                in self.data_set.train_fileids()]

    @cache
    def test_input(self):
        return [self.preprocessor(self.data_set.corpus, fid)
                for fid
                in self.data_set.test_fileids()]
```

1.5 priedas. Failas `features.py`

```
import numpy
from gensim.models.doc2vec           import Doc2Vec
from nltk.tokenize                   import WordPunctTokenizer
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

from utils import cache, flatten

def join_words(word_lists):
    return list(map(' '.join, word_lists))

def join_sentences(sentence_lists):
    return join_words(map(flatten, sentence_lists))

class BagOfWordsModel:
    def __init__(self, vectorizer_type, max_n):
        self.vectorizer_type = vectorizer_type
        self.max_n = max_n

    def fit(self, train_word_lists):
        tokenizer = WordPunctTokenizer()
        vectorizer = self.vectorizer_type(analyzer='word',
                                          tokenizer=tokenizer.tokenize,
                                          ngram_range=(1, self.max_n),
                                          max_features=5000,
                                          dtype=numpy.float64)
        word_strings = join_words(train_word_lists)
        return vectorizer.fit(word_strings)

    def transform(self, vectorizer, word_lists):
        word_strings = join_words(word_lists)
        return vectorizer.transform(word_strings).todense()

bag_of_words = BagOfWordsModel(CountVectorizer, 1)
```

```
34
35  bag_of_ngrams = BagOfWordsModel(CountVectorizer, 2)
36
37  tfidf = BagOfWordsModel(TfidfVectorizer, 1)
38
39  tfidf_of_ngrams = BagOfWordsModel(TfidfVectorizer, 2)
40
41  class Doc2VecModel:
42      def fit(self, train_docs):
43          return Doc2Vec(train_docs, size=300, window=8, min_count=5)
44
45      def transform(self, model, docs):
46          feature_list = [model.infer_vector(doc.words) for doc in docs]
47          return numpy.matrix(feature_list)
48
49  doc2vec = Doc2VecModel()
50
51  class StatisticsModel:
52      VOCABULARY = set("""because been being beneath can can certainly completely
53      could couldn did didn do does doesn doing don done downstairs each early
54      enormously entirely every extremely few fully furthermore greatly had hadn
55      has hasn haven having he her herself highly him himself his how however
56      intensely is isn it its itself large little many may me might mighten mine
57      mostly much musn must my nearly our perfectly probably several shall she
58      should shouldn since some strongly t that their them themselves therefore
59      these they this thoroughly those tonight totally us utterly very was wasn we
60      were weren what whatever when whenever where wherever whether which
61      whichever while who whoever whom whomever whose why will won would wouldn
62      you your ! " # % & ' '' () * + , - -- . : ; = ? ` ``""".split())
63
64      def fit(self, train_sentence_lists):
65          tokenizer = WordPunctTokenizer()
66          vectorizer = CountVectorizer(analyzer='word',
67                                       tokenizer=tokenizer.tokenize,
68                                       vocabulary=StatisticsModel.VOCABULARY,
69                                       dtype=numpy.float64)
70          word_strings = join_sentences(train_sentence_lists)
71          return vectorizer.fit(word_strings)
72
73      @staticmethod
74      def calc_stats(sentence_list):
75          word_list = list(flatten(sentence_list))
76          mean_word_length = numpy.mean(list(map(len, word_list)))
77          mean_sentence_length = numpy.mean(list(map(len, sentence_list)))
78          word_count = len(word_list)
79          return [mean_word_length, mean_sentence_length, word_count]
80
81      def transform(self, vectorizer, sentence_lists):
82          word_strings = join_sentences(sentence_lists)
83          count_matrix = vectorizer.transform(word_strings).todense()
84          stats_matrix = numpy.matrix(list(map(self.calc_stats, sentence_lists)))
85          return numpy.concatenate((count_matrix, stats_matrix), axis=1)
86
87  statistics = StatisticsModel()
88
89  class FeatureStep:
90      def __init__(self, label, preproc_step, feature_extractor):
91          self.label = '{}.{}'.format(preproc_step.label, label)
92          self.preproc_step = preproc_step
93          self.feature_extractor = feature_extractor
```

```
        self.data_set = preproc_step.data_set

    @cache
    def model(self):
        return self.feature_extractor.fit(self.preproc_step.train_input())

    @cache
    def train_features(self):
        return self.feature_extractor.transform(self.model(),
                                                self.preproc_step.train_input())

    @cache
    def test_features(self):
        return self.feature_extractor.transform(self.model(),
                                                self.preproc_step.test_input())
```

1.6 priedas. Failas `classifier.py`

```
from sklearn.linear_model  import LogisticRegression
from sklearn.metrics       import accuracy_score
from sklearn.pipeline      import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm           import SVC
from sknn.mlp              import Classifier, Layer

from utils import cache

def build_logistic(category_count):
    return LogisticRegression()

def build_nn(category_count):
    scaler = StandardScaler()
    layers = [Layer('Tanh', units=30),
              Layer('Softmax', units=category_count)]
    classifier = Classifier(layers=layers, dropout_rate=0.5)
    return make_pipeline(scaler, classifier)

def build_svm_raw(category_count):
    return SVC()

def build_svm_normalized(category_count):
    scaler = StandardScaler()
    svm = SVC()
    return make_pipeline(scaler, svm)

class ClassifyStep:
    def __init__(self, label, feature_step, classifier_builder):
        self.label = '{}.{}'.format(feature_step.label, label)
        self.feature_step = feature_step
        self.classifier_builder = classifier_builder
        self.data_set = feature_step.data_set

    def category_count(self):
        return len(self.data_set.corpus.categories())

    @cache
    def classifier(self):
        classifier = self.classifier_builder(self.category_count())
        return classifier.fit(self.feature_step.train_features(),
```

```
                                self.data_set.train_labels())

    @cache
    def train_predictions(self):
        return self.classifier().predict(self.feature_step.train_features())

    @cache
    def test_predictions(self):
        return self.classifier().predict(self.feature_step.test_features())

    @cache
    def metrics(self):
        train_labels = self.data_set.train_labels()
        test_labels = self.data_set.test_labels()
        return {'feature_count': self.feature_step.train_features().shape[1],
                'class_count': self.category_count(),
                'train_total': train_labels.shape[0],
                'train_matches': accuracy_score(train_labels,
                                                self.train_predictions(),
                                                normalize=False).item(),
                'train_score': accuracy_score(train_labels,
                                              self.train_predictions()).item(),
                'test_total': test_labels.shape[0],
                'test_matches': accuracy_score(test_labels,
                                               self.test_predictions(),
                                               normalize=False).item(),
                'test_score': accuracy_score(test_labels,
                                             self.test_predictions()).item()}
```

1.7 priedas. Failas `requirements.txt`

```
numpy==1.11.0
scipy==0.17.1
scikit-learn==0.17.1
scikit-neuralnetwork==0.7
nltk==3.2.1
gensim==0.12.4
```