

T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ÖZELLİKLERİ BAĞIMSIZ OLMAYAN VERİ KÜMELERİ
İÇİN NAİVE BAYES SINIFLANDIRICILAR

Yeliz HATİP

YÜKSEK LİSANS TEZİ
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Programı

Danışman
Doç. Dr. Ferkan YILMAZ

Haziran, 2022

T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

**ÖZELLİKLERİ BAĞIMSIZ OLMAYAN VERİ KÜMELERİ İÇİN NAİVE
BAYES SINIFLANDIRICILAR**

Yeliz HATİP tarafından hazırlanan tez çalışması 16.06.2022 tarihinde aşağıdaki jüri tarafından Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği Programı **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Doç. Dr. Ferkan YILMAZ
Yıldız Teknik Üniversitesi
Danışman

Jüri Üyeleri

Doç. Dr. Ferkan YILMAZ, Danışman
Yıldız Teknik Üniversitesi

Prof. Dr. Oya KALIPSIZ, Üye
Yıldız Teknik Üniversitesi

Doç. Dr. Atakan KURT, Üye
İstanbul Üniversitesi

Danışmanım Doç. Dr. Ferkan YILMAZ sorumluluğunda tarafımda hazırlanan ÖZELLİKLERİ BAĞIMSIZ OLMAYAN VERİ KÜMELERİ İÇİN NAİVE BAYES SINIFLANDIRICILAR başlıklı çalışmada veri toplama ve veri kullanımında gerekli yasal izinleri aldığımı, diğer kaynaklardan aldığım bilgileri ana metin ve referanslarda eksiksiz gösterdiğimi, araştırma verilerine ve sonuçlarına ilişkin çarpıtma ve/veya sahtecilik yapmadığımı, çalışmam süresince bilimsel araştırma ve etik ilkelerine uygun davrandığımı beyan ederim. Beyanımın aksinin ispatı halinde her türlü yasal sonucu kabul ederim.

Yeliz HATİP

İmza

Sevgili Aileme..



TEŞEKKÜR

Bu tezin içeriğinin oluşturulmasında ve yüzlerce ufak ayrıntının birbirleriyle ahenk içerisinde tezde yer almasında önemli katkılarda bulunan, kıymetli bilgi, birikim ve tecrübeleri ile bana yol gösterici ve destek olan, çalışmalarımı yönlendiren ve bütün içtenliğiyle yardımcı olan değerli tez danışmanım sayın Doç. Dr. Ferkan YILMAZ'a en içten saygılarımı ve teşekkürlerimi sunuyorum.

Hiç bir zaman maddi ve manevi desteklerini esirgemeyen canım aileme teşekkür ediyor ve onlara minnettar olduğumu bir kez daha dile getiriyorum.

Yeliz HATİP

İÇİNDEKİLER

SİMGE LİSTESİ	vii
KISALTMA LİSTESİ	viii
ŞEKİL LİSTESİ	x
TABLO LİSTESİ	xi
ÖZET	xii
ABSTRACT	xiii
1 GİRİŞ	1
1.1 Literatür Özeti	2
1.1.1 Yapı Uzantısı (Structure Extension)	3
1.1.2 Özellik Seçimi (Feature Selection)	4
1.1.3 Örnek Seçimi (Instance Selection)	5
1.1.4 Örnek Ağırlıklandırma (Instance Weighting)	5
1.1.5 İnce Ayar (Fine Tuning)	6
1.1.6 Özellik Ağırlıklandırma (Feature Weighting)	6
1.2 Tezin Amacı	6
1.3 Hipotez	7
2 NAİVE BAYES SINIFLANDIRMASI VE ÖZELLİK AĞIRLIKLANDIRMA	9
2.1 Kullback-Leibler Ölçüsü	11
2.2 Özellik Ağırlıklandırma Yöntemleri	12
3 BAĞIMLILIK METRİĞİ VE BAĞIMLI ÖZELLİK AĞIRLIKLI NAİVE BAYES	16
3.1 Bağımlılık Metriği	16
3.2 Bağımlı Özellik Ağırlıklandırmalı Naive Bayes Sınıflandırıcılar	22
4 DENEYSEL SONUÇLAR	26
5 SONUÇ VE ÖNERİLER	31

KAYNAKÇA	32
A DFW NB SINIFLAYICININ JAVA UYGULAMASI	36
A.1 DFW Naive Bayes Sınıflandırıcısının Java Gerçeklemesi	36
B CDFW NB SINIFLAYICININ JAVA UYGULAMASI	47
B.1 CDFW Naive Bayes Sınıflandırıcısının Java Gerçeklemesi	47
B.2 WEKA UCI 32 Veri Kümesi için CDFW Naive Bayes Test Kodu	58
TEZDEN ÜRETİLMİŞ YAYINLAR	63



SİMGE LİSTESİ

$c(\mathbf{a})$	$\mathbf{a} \in T$ örneğinin sınıflandırılmasıyla tahmin edilen sınıf
$\inf A$	A değişkeninin/kümesinin en büyük alt sınırı
$\sup A$	A değişkeninin/kümesinin en küçük üst sınırı
$\Pr(a_i)$	A_i niteliğinin $a \in \mathcal{A}_i$ değerini alma olasılığı, $\Pr(a_i) \equiv \Pr(A_i = a)$
a_i	A_i niteliğinin alabileceği değerlerden biri
$\mathcal{A}_i = \{a_{i1}, \dots, a_{in}\}$	A_i niteliğinin alabileceği değerlerin kümesi, burada $n = \mathcal{A}_i $
$\mathcal{C} = \{c_1, \dots, c_k\}$	C sınıf niteliğinin alabileceği değerlerin kümesi, burada $k = \mathcal{C} $
$\Pr(c)$	C sınıf niteliğinin $c \in \mathcal{C}$ değerini alma olasılığı, $\Pr(c) \equiv \Pr(C = c)$
q	K-mean kümeleme için seçilen küme sayısı
μ_n	n . dereceden bağımsızlık eksikliği
A_i	Nitelik (Nitelik rastgele değişkeni)
m	Nitelik sayısı/Özellik sayısı
$\mathbf{a} = \langle a_1, \dots, a_m \rangle$	Niteliklerin değerlerinden oluşan örnek vektör $\mathbf{a} \in T$
n	Örnek sayısı
W_i	Özellik ağırlığı
T	Rastgele eğitim numunesi

KISALTMA LİSTESİ

BNC	Bayes Ağ Sınıflandırıcı
CDFW	Küme-tabanlı DFW
CFS	Korelasyon-tabanlı Özellik Seçimi
CFW	Korelasyon-tabanlı Özellik Ağırlıklandırma
CLL	Koşullu Log Olabilirliği
CLLFW	Koşullu Log Olabilirliğe dayalı Özellik Ağırlıklandırma
DeepFW	Derin Özellik Ağırlıklandırma
DFW	Bağımlı Özellik Ağırlıklandırma
DEFW	Diferansiyel Evrim-tabanlı Özellik Ağırlıklandırma
DT	Karar Ağacı
DTFW	Karar Ağacı-tabanlı Özellik Ağırlıklandırma
ENB	Evrimsel Naive Bayes
FCS	Nitelik-Sınıf Önemi
FFS	Nitelik-Nitelik Önemi
FW	Özellik Ağırlıklandırma
FWNB	Özellik Ağırlıklı Naive Bayes
GR	Kazanç Oranı
GRFW	Kazanç Oranı-tabanlı Özellik Ağırlıklandırma
IWNB	Örnek Ağırlıklı Naive Bayes
KLMFW	Kullback-Leibler Ölçüm-tabanlı Özellik Ağırlıklandırma
LBR	Tembel Bayes Kuralı
LWNB	Yerel Ağırlıklı Naive Bayes
MI	Karşılıklı Bilgi

MSEFW	Ortalama Kare alınmış Hata-tabanlı Özellik Ağırlıklandırma
NB	Naive Bayes
SNNB	Seçici Komşuluk Naive Bayes
SSE	Karesel Hataların Toplamı
TAN	Ağaçla Güçlendirilmiş Naive Bayes
UCI	University of California Irvine
VIA	Değişken Önem Analizi



ŞEKİL LİSTESİ

Şekil 1.1	NB ve TAN Yapılarının Grafikselle Olarak Karşılaştırılması	3
-----------	--	---



TABLO LİSTESİ

Tablo 4.1	Deneylerimizde Kullanılan 32 UCI Veri Kümesi	27
Tablo 4.2	CDFW ve DFW için NB, CFW, KLMFW, GRFW, DTFW, DeepFW, DEFW, CLLFW ve MSEFW ile Doğruluk Karşılaştırmaları	29
Tablo 4.3	Wilcoxon Doğruluk Test Özeti	30



ÖZELLİKLERİ BAĞIMSIZ OLMAYAN VERİ KÜMELERİ İÇİN NAIVE BAYES SINIFLANDIRICILAR

Yeliz HATİP

Bilgisayar Mühendisliği Anabilim Dalı
Yüksek Lisans Tezi

Danışman: Doç. Dr. Ferkan YILMAZ

Naive Bayes (NB) sınıflandırması, yüksek verimliliği ve özelliklerin koşullu bağımsızlığına dayalı yapısal basitliği nedeniyle veri madenciliği ve makine öğreniminde en yaygın kullanılan algoritmalarındandır. Çalışmada, özellikler arasındaki bağımlılığı ölçmek için bir bağımlılık metriği sunuyoruz. NB’de daha az belirleyici özellikleri keşfetmek için Nitelik-Nitelik Önemi [Feature-Feature Significance (FFS)] ve Nitelik-Sınıf Önemi [Feature-Class Significance (FCS)] öneriyoruz. FFS ve FCS’den özellik ağırlıklarının nasıl alınacağını gösteriyoruz ve yeni bir Bağımlı Özellik Ağırlıklandırma [Dependent Feature Weighting (DFW)] NB öneriyoruz. Performansı daha da artırmak için niteliklerin homojen olmayan bağımlılığı nedeniyle ilgilenilen rastgele örneğin kümelenmesini ve koşullu bağımsızlık varsayımını hafifletmek için özellik ağırlıklandırma kullanılmasını öneriyoruz. Sonuç olarak, rastgele alt örneklerin DFW filtrelerini doğruluklarına göre ağırlıklandırmayı ve performansı arttırmak için birleştirmenin sonucu olarak Küme-tabanlı DFW [Cluster-based DFW (CDFW)] NB öneriyoruz. Deneysel sonuçlar, DFW filtreli NB’nin, geleneksel NB ve diğer özellik ağırlıklandırma tekniklerine göre performansı arttırdığını göstermektedir.

Anahtar Kelimeler: Özellik ağırlıklandırma, naive Bayes sınıflandırma, kümeleme-tabanlı bağımlılık, karşılıklı bağımlılık

NAIVE BAYES CLASSIFIERS FOR DATASETS WITH DEPENDENT ATTRIBUTES

Yeliz HATİP

Department of Computer Engineering
Master of Science Thesis

Supervisor: Assoc. Prof. Dr. Ferkan YILMAZ

Naive Bayes (NB) classification is one of the most extensively used algorithms in data mining and machine learning due to its high efficiency and structural simplicity based on conditional independence of attributes. In this paper, we present a dependence metric to quantify the dependence among attributes and class attributes and propose feature-feature significance (FFS) and feature class significance (FCS) to discover highly predictive attributes over less predictive ones in NB classification. We show how to get feature weights from FFS and FCS and propose a novel dependent feature weighted (DFW) NB classification. To increase performance further, we recommend clustering the random sample of interest due to the non-homogeneous dependence nature of features, and then using feature weighting to alleviate the conditional independence. As a consequence, we propose a cluster-based DFW (CDFW) NB as a result of weighting the DFW filters of random sub-samples by their accuracy and then merging them for performance augmentation. The experimental results show that the NB with DFW filter provides goods results while compared to the conventional NB and all other feature weighting techniques.

Keywords: Feature weighting, naive Bayes classification, cluster-based dependence, mutual dependence

Sınıflandırma, bilinmeyen değişkenler veya kategoriler hakkında tahminler yapmak için rastgele bir örneği istatistiksel özellikleriyle tanımlama sürecidir. Bir özellik vektörü tarafından temsil edilen farklı örnekler, doğru sınıf etiketlerini atayabilen bir sınıflandırıcıyı öğrenmeyi gerektiren makine öğrenimi ve veri analizinde temel bir konudur [1]. Sınıflandırma için algoritmaları öğrenmenin amacı sınıf etiketleriyle verilen örneklerden bir sınıflandırıcı oluşturmaktır. Sınıflandırıcı, bir dizi nitelik değeri tarafından açıklanan örnekler olası sınıf etiketini tahmin eder. Denetimli öğrenme algoritmaları (supervised learning algorithms), eğitim örneğinin istatistiksel yapısını kullanarak yeni örnekleri kategorize etmek için yaygın olarak kullanılmaktadır [2–10]. Veri madenciliği ve makine öğreniminde en temel yöntemlerden biri haline gelmiştir. Bayes ağları [2–4] bir öğrenme algoritmasının, sınıf etiketleriyle birlikte verilen bir eğitim rastgele örneğinden bir sınıflandırıcı oluşturmaya çalıştığı sınıflandırma problemlerini ele almak için kullanılır. Sonuç olarak bir sınıflandırma algoritmasının birincil amacı bir sınıflandırıcı oluşturmaktır ve mevcut sınıflandırma algoritmaları arasında Naive Bayes (NB) sınıflandırıcı sadece basitliği [2–8] ile değil, aynı zamanda yüksek performansıyla da mevcut sınıflandırma algoritmaları arasında öne çıkmaktadır [2–4, 6–8, 11]. NB, sınıflandırma için koşullu nitelik bağımsızlığı varsayımıyla basitleştirilmiş bir Bayes ağı kullanan basit, verimli ve etkili bir öğrenme algoritmasıdır [12].

NB algoritması en yaygın sınıflandırma algoritmalarından biridir. Birçok araştırmacı bu yaklaşımın teorik ve ampirik sonuçlarını incelemiştir. Birçok veri madenciliği uygulamasında yaygın olarak kullanılmaktadır ve uygulamalarda şaşırtıcı derecede iyi performans göstermektedir [13]. İyi performansının yanı sıra oluşturma ve yorumlama kolaylığı nedeniyle NB [14], gerçek dünya uygulamalarında sınıflandırma problemlerini ele almak için yaygın olarak kullanılmaktadır.

NB hala en iyi 10 veri madenciliği algoritmasından biridir [15]. NB, denetimli öğrenme için basit ve sık kullanılan bir yöntemdir. Herhangi bir sayıda nitelik veya sınıfla başa çıkmak için esnek bir yol sağlar ve olasılık teorisine dayanır. Tüm eğitim

girdilerini inceleyen asimptotik olarak en hızlı öğrenme algoritmasıdır. Modelin basit doğasına rağmen çok çeşitli problemlerde şaşırtıcı derecede iyi performans gösterdiği kanıtlanmıştır. Birçok gerçek dünya uygulamasında da hem ikili hem de çok sınıflı kategorilerde sınıflandırma zorluklarını çözmek için kullanılmakta ve dikkate değer bir şekilde iyi performans göstermektedir [16, ve buradaki referanslar]. Bununla birlikte, NB sınıflandırıcının hala iki büyük eksikliği vardır [17]. Bunlardan biri güvenilir olmayan olasılık tahminidir. Diğer ve belki de önemli olanı, nitelikler arasında koşullu bağımsızlığın gerçekçi olmayan varsayımdır. Koşullu bağımsızlık varsayımı gerçekte nadiren doğrudur. Bu durum da karmaşık nitelik bağımlılıklarına sahip uygulamaların performansına zarar verir [14]. NB algoritmasını kullanabilmek için niteliklerin birbirinden bağımsız olmaları gerekmektedir. Koşullu bağımsızlık varsayımından kaynaklanan hataları azaltmak için NB algoritmasında geçmişten günümüze birçok geliştirme ve iyileştirme yöntemleri önerilmiştir.

Çalışmanın Literatür Özeti bölümünde, literatürde koşullu bağımsızlık varsayımını hafifletmeye yardımcı olmak için NB'ye yönelik geliştirilen bir dizi iyileştirilmiş yaklaşımlara değinilmiştir.

1.1 Literatür Özeti

Niteliklerde bağımsızlık varsayımını azaltmak ve aynı zamanda NB modelinin basitliğini korumak için NB, araştırmacıların büyük ilgisini çekmiştir [18]. Koşullu bağımsızlık varsayımını hafifletmeye yardımcı olmak için literatürde NB'ye yönelik bir dizi iyileştirme yöntemleri önerilmiştir. İlgili iyileştirme yöntemleri genel olarak altı ana kategoriye ayrılabilir [15, 19]:

- Yapı Uzantısı (Structure Extension) [4, 8, 20–23]
- Özellik Seçimi (Feature Selection) [24–29]
- Örnek Seçimi (Instance Selection) [30–32]
- Örnek Ağırlıklandırma (Instance Weighting) [15, 33–35]
- İnce Ayar (Fine Tuning) [36–38]
- Özellik Ağırlıklandırma (Feature Weighting) [13, 19, 39–43]

Bu çalışmada dikkatimizi özellik ağırlıklandırma yöntemine odaklıyoruz. İyileştirme yöntemleri hakkında genel bir bilgi vermek amacıyla yöntemler, başlıklar halinde kısaca bahsedilmiştir. Ardından üzerinde duracağımız özellik ağırlıklandırma yöntemi ile devam edilmiştir.



Şekil 1.1 NB ve TAN Yapılarının Grafikselsel Olarak Karşılaştırılması

1.1.1 Yapı Uzantısı (Structure Extension)

İyileştirme yöntemlerinin ilki *Yapı Uzantısı (Structure Extension)* yöntemidir. NB'nin karşılaştığı temel sorun, koşullu bağımsızlık varsayımdır. Gerçek dünya veri madenciliği uygulamalarında bu varsayım gerçekçi değildir. Bu nedenle nitelik bağımlılıklarını açıkça temsil etmek için yönlendirilmiş yaylar (directed arcs) kullanarak NB'nin yapısını genişletmek, bu gerçekçi olmayan varsayımı gevşetmenin doğal bir yoludur [22, 23]. Yönlendirilmiş yaylar, birleşik olasılık dağılımını açıkça temsil etmek için kullanılabilir. Ortaya çıkan model aslında bir Bayes ağıdır. NB, Bayes ağlarının en basit şeklidir. NB'deki koşullu bağımsızlık varsayımının gerçekte nadiren doğru olduğu ve karmaşık nitelik bağımlılıkları olan uygulamalardaki performansına zarar vereceği açıktır. Bu nedenle Bayes ağlarının yapısını öğrenmek kaçınılmazdır. Ancak optimal yapıyı öğrenmek zor bir problemdir[44]. NB'nin grafiksel yapısı Şekil 1.1a'da gösterilmiştir. Bir Bayes ağı, yapısal bir model ve bir dizi koşullu olasılıktan oluşur. Yapısal model, düğümlerin nitelikleri temsil ettiği ve yayların nitelik bağımlılıklarını temsil ettiği yönlendirilmiş döngüsel olmayan bir grafikdir. Bayes ağları örneğin sınıf etiketini tahmin etmek için kullanılır. Her nitelik düğümü, ebeveyni olarak sınıf düğümüne sahiptir ancak nitelik düğümlerinden herhangi bir ebeveyni yoktur. Pratikte, Bayes ağlarının yapılarına kısıtlamalar getirilmesi gereklidir. Örneğin, Ağaçla Güçlendirilmiş NB [Tree-Augmented NB (TAN)] [4] öğrenmek, kabul edilebilir bir hesaplama karmaşıklığına ve NB'e göre önemli bir gelişmeye yol açar. Friedman ve ark. [4] koşullu karşılıklı bilgiye (conditional mutual information) dayalı bir TAN öğrenme algoritması önermektedir. TAN, sınıf düğümünün doğrudan tüm nitelik düğümlerine işaret ettiği ve bir nitelik düğümünün başka bir nitelik düğümünden yalnızca bir ebeveyni sahip olabileceği, genişletilmiş ağaç benzeri bir NB'dir [21]. Sınıf düğümünün tüm nitelik düğümlerinin ana düğümü olduğu, ancak nitelik düğümleri arasında yönlendirilmiş yaylarda herhangi bir sınırlamanın olmadığı (yönlendirilmiş herhangi bir döngü oluşturmamaları dışında) bir Bayes ağının özel bir durumudur [18]. TAN öğrenme algoritması, Şekil 1.1b'de gösterildiği gibi, bir nitelik düğümünün sınıf düğümü dışında en fazla bir ek üst düğüme sahip olabileceği NB'in bir uzantısıdır. Ek yay, nitelik düğümleri arasındaki etkileşimi temsil eder.

TAN öğrenmede kaçınılmaz bir konu, yapı öğrenmedir. Bu nedenle yapı öğrenmesinden kaçınan ve yine de nitelik bağımlılıklarını bir dereceye kadar temsil edebilen bir model arzu edilir.

Birçok yöntem, TAN gibi bağımsızlık varsayımını hafifleterek NB'nin sınıflandırma performansını iyileştirmeye çalışır [22]. Qiu ve ark. [8], artan yayları bulmak için Koşullu Log Olabilirliğinin [Conditional Log Likelihood (CLL)] kullanıldığı, geliştirilmiş bir CLL tabanlı SuperParent (SP) algoritması önerdi. Deneysel sonuçlarına göre SP algoritması, NB'den önemli ölçüde daha iyi performans göstermektedir [20]. Ancak bir dezavantajı, nispeten yüksek zaman karmaşıklığıdır.

1.1.2 Özellik Seçimi (Feature Selection)

İyileştirme yöntemlerinin ikincisi *Özellik seçimi (Feature selection)* yöntemidir. Özellik seçiminin amacı, veri kümesinden gereksiz ve/veya alakasız özellikleri kaldırarak ve yalnızca sınıflandırmada en belirleyici olanları seçerek NB sınıflandırıcının performansını iyileştirmektir [25]. Başlangıçtaki geniş özellik yelpazesinden uygun özelliklerin yalnızca küçük bir alt kümesini seçerek sınıflandırma verimliliğini artırmayı amaçlar [24]. Veri kümesinden gereksiz ve önemsiz özellikleri kaldırarak en uygun özellik kümesini bulmaya çalışır. Alakasız ve gereksiz özelliklerin kaldırılması, iyi bir metin gösterimi ve azaltılmış bir veri boyutu sağlar [29]. Modelin öğrenme döngüsünü hızlandırır ve tahmine dayalı modelin performansını artırır. Bu nedenle özellik seçiminin avantajı diğer birçok avantajın yanı sıra, veri boyutunu azaltmak, doğruluğu artırmak, alakasız verileri ve verilerdeki karmaşıklıkları ortadan kaldırmak ve içgörülerini geliştirmek için eğitimi hızlandırmak yer alır [45]. Literatürde özellik seçimine ilişkin çeşitli çalışmalar mevcuttur. Özellik seçim yöntemleri, bize hesaplama süresini azaltmanın, tahmin performansını iyileştirmenin ve makine öğrenimi veya örüntü tanıma [27] uygulamalarındaki verileri daha iyi anlamının bir yolunu sağlar. Tahmine en fazla katkısı olan özelliklerin etkin seçimi, bu yaklaşımdaki ana zorluk olarak kabul edilebilir. Bu nedenle birçok araştırmacı bu amaçla çalışmalar yapmıştır. Jiang ve ark. [26] Evrimsel NB [Evolutional NB (ENB)] adlı bir algoritma sundu. Algoritmaları, evrimsel bir arama süreci gerçekleştirerek, tüm özellikler alanı boyunca bir özellik alt kümesini seçer. Seçilen özellik alt kümelerini değerlendirmek için uygunluk fonksiyonu olarak kullanılan NB sınıflandırma doğruluğu gerçekleştirilir. Chandrashekar ve Sahin [46], özellik seçim yöntemlerine genel bir giriş sunmuş ve bunları filtre, sarmalayıcı ve gömülü olarak sınıflandırmıştır. Sarmalayıcılar, özellik alt kümelerini değerlendirmek için sınıflandırıcının kendisini kullanırken, filtreler verilerin özelliklerine dayalı olarak buluşsal yöntemler ile özellik alt kümelerini değerlendirir [18, 28].

1.1.3 Örnek Seçimi (Instance Selection)

Bir diğer iyileştirme yöntemi *Örnek seçimi (Instance selection)* yöntemidir. Veri kümesini yönetilebilir bir hacme indirmek için örnek seçimine yönelik yaklaşımlar uygulanabilmektedir ve bu durum öğrenme sürecini gerçekleştirmek için gerekli olan hesaplama kaynaklarının azalmasını sağlamaktadır. Örnek seçimi algoritmaları, öğrenme algoritmalarını uygulamadan önce gürültülü örnekleri kaldırmak için de uygulanabilir. Bu adım, sınıflandırma problemlerinde doğruluğu arttırabilmektedir [47].

Yerel öğrenme yaklaşımında NB modeli, eğitim veri kümesinin bir alt kümesi üzerine kuruludur. Yerel öğrenme, bir alt kümedeki koşullu bağımsızlık varsayımının olumsuz etkisinin tüm veri kümesindekinden daha küçük olduğu fikrini kullanır. Yerel öğrenme, birbirine gömülü yeni modellerin oluşturulmasına olanak tanır. NBTree [30], NB ve karar tablolarını birleştirirken, Yerel Ağırlıklı NB [Locally Weighted NB (LWNB)] [32], Tembel Bayes Kuralı [Lazy Bayesian Rule (LBR)] ve Seçici Komşuluk NB [Selective Neighborhood (SNNB)] [31] modelleri, k -en yakın komşular modeline (k -nearest neighbors model) gömülü NB modellerinin başında gelmektedir.

1.1.4 Örnek Ağırlıklandırma (Instance Weighting)

Bir diğer iyileştirme yöntemi *Örnek ağırlıklandırma (Instance weighting)* yöntemidir. Örnek ağırlıklandırmada her örneğe farklı bir ağırlık atayarak NB sınıflandırıcılarının ayırt edici özelliğinin artması amaçlanır [35]. [15]'de bugüne kadar NB de örnek ağırlıklandırma yaklaşımları konusunda çok az çalışma olduğunu belirtmişlerdir. Örnek ağırlıklandırma yaklaşımı, farklı eğitim örneklerine ayırt edici bir şekilde farklı ağırlıklar atayarak NB'yi iyileştirir. Ağırlık öğrenme yaklaşımı, filtre öğrenme ve sarmalayıcı öğrenme olarak iki kategoriye ayrılır [15]. Sarmalayıcı öğrenme, filtre öğrenimine kıyasla üstün bir performansa sahiptir, ancak filtre öğrenimi her zaman sarmalayıcı öğrenimden daha hızlı çalışır. Ayrıca sarmalayıcı öğrenmenin anlaşılabilirliği, filtre öğrenmeden çok daha zordur. Genel olarak, NB'yi iyileştirmeye yönelik sayısız yaklaşım arasında örnek ağırlıklandırma, garanti ettiğinden daha az ilgi görmüştür.

[34]'de Örnek Ağırlıklı NB [Instance Weighted NB (IWNB)] sınıflandırıcısı tanıtılmıştır. IWNB'de her eğitim örneği, öncelikle eğitim örneklerinin modu ile arasındaki benzerliğe göre ağırlıklandırılır ve ardından ağırlıklı eğitim örnekleri üzerine bir NB sınıflandırıcısı oluşturulur. Literatürde doğrudan örnek ağırlıklarını içeren sınıflandırıcılar da mevcuttur [33].

1.1.5 İnce Ayar (Fine Tuning)

Bir diğer iyileştirme yöntemi *İnce ayar (Fine tuning)* yöntemidir. İnce ayar yöntemi, NB'nin sınıflandırma doğruluğunu iyileştirmede etkili olduğu kanıtlanmıştır [48]. [36]'de El Hindi tarafından önerilen yöntem, gerekli olasılık terimlerinin daha iyi tahminlerini belirlemek için NB'yi ikinci bir ince ayar aşamasıyla güçlendirdiği belirtilmiştir. Aynı zamanda çalışmada sunulan ampirik sonuçlar, ince ayar yönteminin birçok alanda ortalama sınıflandırma doğruluğunu önemli ölçüde artırdığını göstermiştir.

[37]'de El Hindi ve ark. tarafından Bayes ağlarına ince ayar yapmak için bir ince ayar yöntemi önerildi. [38]'de, metin sınıflandırmasında NB tarafından kullanılan olasılık terimlerinde daha iyi tahminler bulmak için diferansiyel evrim kullanılmıştır.

1.1.6 Özellik Ağırlıklandırma (Feature Weighting)

İyileştirme yöntemlerinin sonuncusu *Özellik ağırlıklandırma (Feature weighting)* yöntemidir. Özellik ağırlıklandırma yaklaşımı, öngörücü özellikleri önemlerine göre farklı şekilde ağırlıklandırır ve ortaya çıkan model Özellik Ağırlıklı NB [Feature Weighting NB (FWNB)] olarak adlandırılır. FWNB, formüle özellik ağırlıklarını dahil eder.

Koşullu bağımsızlık varsayımını hafifletmeye yönelik birçok yaklaşım arasından özellik ağırlıklandırma, daha az tahmin edici olanlardan ziyade yüksek tahmin edici özelliklere daha fazla vurgu yapmıştır. [19]'da NB için yüksek öngörücü niteliklerin sınıfla yüksek oranda ilişkili olması gerektiğini (karşılıklı maksimum ilişki), ancak diğer niteliklerle ilişkisiz (minimum karşılıklı artıklık) olması gerektiği savunulmuştur. Bu önermeye dayanarak, NB için Korelasyon-tabanlı Özellik Ağırlıklandırma [Correlation-based Feature Weighting (CFW)] filtresi önerilmiştir. CFW'de bir özelliğin ağırlığı, özellik-sınıf korelasyonu (karşılıklı alaka) ile ortalama özellik-özellik korelasyonu (ortalama karşılıklı artıklık) arasındaki farkın sigmoid dönüşümüdür [19]. Başka bir ifade ile, CFW'de maksimum karşılıklı alaka ve minimum ortalama karşılıklı artıklığa sahip özellikler, yüksek öngörücü özellikleri olarak kabul edilir ve dolayısıyla daha büyük özellik ağırlıklarına sahiptir.

1.2 Tezin Amacı

Bu çalışmada özellik ağırlıklandırma yöntemine odaklanıyoruz. Mevcut tüm özellik ağırlıklandırma yöntemlerinin, NB'nin hesaplama karmaşıklığı ve/veya son modelin karmaşıklığı pahasına sınıflandırma doğruluğunu geliştirdiğini görüyoruz.

Literatür özetinde açıkladığımız, koşullu bağımsızlık varsayımını hafifletmeye yönelik geliştirilen iyileştirme yöntemleri, hesaplama karmaşıklığı pahasına özellik ağırlıkları elde ettiğinden, özellikler arasında koşullu bağımlılığın homojen olmayan doğasını hiç hesaba katmamaktadırlar. Ancak bu homojen olmayan doğa göz ardı edilmemelidir. Bu nedenle, özellikler arasındaki koşullu bağımlılığı kullanarak özellik ağırlıklandırma yoluyla NB sınıflandırmasının doğruluğunu artırmaya odaklanıyoruz ve başlıca katkılarımızı Hipotez bölümünde özetliyoruz.

1.3 Hipotez

Başlıca katkılarımızı şu şekilde özetliyoruz:

- NB sınıflandırıcıları, ilişkisiz niteliklerin mutlaka bağımsız olması gerekmediğinden, korelasyonsuzlukla ilgili zorluklar yaşar. Bu nedenle, özellik bağımlılığını ölçmek için bir bağımlılık metriği öneriyoruz. Ardından Nitelik-Sınıf Önemi [Feature Class Significance (FCS)] ve Nitelik-Nitelik Önemi [Feature Feature Significance (FFS)] tanıtıyoruz.
- Yüksek FCS'li özelliğin, sınıfın yüksek derecede tahmin edici olması, ancak diğer özelliklerin (en düşük FFS) olmaması gerektiğini gözlemleyerek, FCS ve FFS metriklerini kullanarak özellik ağırlıklarını öğrenen bir Bağımlı Özellik Ağırlıklandırma [Dependent Feature Weighting (DFW)] filtresi geliştirdik ve bir DFW NB sınıflandırıcısı önerdik. Öğrenilen ağırlıklar özelliklerin üsleri olarak atanır.
- Gerçek dünya uygulamalarında, rastgele örneklerin çeşitli kaynakları nedeniyle, rastgele örnek özellikleri arasındaki homojen olmayan bağımlılık kaçınılmazdır. K-mean gibi bir kümeleme yaklaşımı kullanarak, rastgele örneği rastgele alt örneklerle ayırmanızı ve ardından onların özellik ağırlıklarını DFW filtremizi kullanarak elde etmenizi öneririz. Buna göre, DFW filtrelerinin hedeflerini doğruluklarına göre ağırlıklandırmanın ve ardından performans geliştirme için birleştirmenin bir sonucu olarak, Küme-tabanlı DFW [Cluster-based Dependent Feature Weighting (CDFW)] filtresi adı verilen bir DFW filtresi türü öneriyoruz.

Çalışmanın geri kalanı aşağıdaki şekilde düzenlenmiştir. Bölüm 2'de, NB sınıflandırması detaylı olarak açıklanmıştır ve NB için özel olarak tasarlanmış son teknoloji FW yöntemleri hakkında bilgi verilmiştir. Bölüm 3'de, bağımlılık metriği açıklanmaktadır. FCS ve FFS'den özellik ağırlıklarının nasıl hesaplanacağı gösterilmektedir. Ardından DFW NB sınıflandırması açıklanmaktadır. Bölüm 4'de,

yöntemler arasında doğruluk karşılaştırmaları verilmiştir ve deneysel sonuçlar ayrıntılı olarak yorumlanmaktadır. Bölüm 5, çalışmayı sonuçlandırmaktadır.



2

NAIVE BAYES SINIFLANDIRMASI VE ÖZELLİK AĞIRLIKLANDIRMA

FW yöntemlerinde, her özelliğe önem düzeyine göre farklı bir ağırlık atanır. Uygun özellik ağırlıkları, NB'nin gerektirdiği nitelik bağımsızlık varsayımının ihlalinden kaynaklanan hatayı azaltabilir. Bir eğitim verisi seti birbiriyle bağımlı olan bir dizi niteliği içeriyorsa, özellik setine 1.0 olan ağırlıklar atanarak, nitelik bağımsızlık varsayımının ihlalinden kaynaklanan hata giderilebilir. Ayrıca özellik ağırlıklandırmadan bahsederken nitelik seçiminden sıklıkla söz edilir [14, 19]. Örneğin, niteliklerden biri olan A_i 'nin özellik ağırlığı 1.0 olarak ve A_i ile bağımlı olan diğer niteliklerin özelliklerinin ağırlığı 0.0 olarak ayarlanabilir. Bu durum eğitim verilerinden bağımlı niteliklerinin, yani özellik seçiminin kaldırılmasına eşdeğerdir. Nitelik seçimi, özellik ağırlıklarını 0 veya 1'e ayarlayarak özellik ağırlıklandırmanın özel bir durumu olarak görülebilir [13]. Fakat özellik ağırlıklandırma, nitelik seçiminden kesinlikle daha güçlüdür çünkü seçilen özelliklerin ağırlıklarını 1.0'a ve seçilmemiş özelliklerin ağırlıklarını 0.0'a ayarlayarak nitelik seçiminde aynı sonuçları elde etmek mümkündür. Özetle özellik ağırlıklandırma, her özelliğe sürekli bir pozitif ağırlık atar ve bu nedenle nitelik seçiminden daha esnek bir yaklaşımdır [19].

[19]'da NB için yüksek öngörücü niteliklerin sınıfla yüksek oranda ilişkili olması gerektiğini (karşılıklı maksimum ilişki), ancak diğer niteliklerle ilişkisiz (minimum karşılıklı artıklık) olması gerektiği savunulmuştur. Bu önermeye dayanarak, NB için CFW filtresi önerilmiştir. CFW'de bir özelliğin ağırlığı, özellik sınıf korelasyonu (karşılıklı alaka) ile ortalama özellik-özellik korelasyonu (ortalama karşılıklı artıklık) arasındaki farkın sigmoid bir dönüşümüdür. Basit olması için, ayrık (kategorik) değişkenler dahil edilmiştir. Sürekli (sayısal) rastgele değişkenler, WEKA platformunda [49] uygulanan Fayyad-Irani'nin MDL yöntemi[50] kullanarak onları ayrık (kategorik) değişkenlere dönüştürülebilir. Daha sonra, her rastgele ayrık değişken çifti arasındaki korelasyonu ölçmek için karşılıklı bilgi miktarının/entropilerinin kullanıldığı özellik-sınıf korelasyonu ve özellik-özellik korelasyonu ölçütleri önerilmiştir ve ilgili tanımları açıklanmıştır[19].

n örnekle bir eğitim rastgele numunesi $T = \{X_1, X_2, \dots, X_n\}$ sağlandığında, her eğitim örneğinin bir sınıf özelliğinin yanı sıra m özellik içerdiğini varsayıyoruz. k 'ninci eğitim örneği, $A = \langle A_1, A_2, \dots, A_m \rangle$ ile rastgele bir $X = \langle A, C \rangle$ vektörü ile uygun bir şekilde temsil edelim; burada A_i , verisetinden i 'ninci niteliği ifade eder ve $\mathcal{A}_i$ örnek uzayında tanımlı olan rastgele değerlidir. C ise ilgili örneğin sınıfıdır ve örnek uzay $\mathcal{C}$ 'de sınıf rastgele özelliğidir. Bir Bayes Ağ Sınıflandırıcısı [Bayesian Network Classifier (BNC)], bir test örneğini $\mathbf{a} = \langle a_1, a_2, \dots, a_m \rangle$ 'yı $\forall_i a_i \in \mathcal{A}_i$ ile [2–4] aşağıdaki gibi sınıflandırır;

$$c(\mathbf{a}) = \arg \max_{c \in \mathcal{C}} \Pr(c) \Pr(a_1, a_2, \dots, a_m | c). \quad (2.1)$$

Burada BNC tarafından tahmin edilen sınıf etiketi $c(\mathbf{a})$ ile temsil edilmektedir. $\arg \max$ bütün sınıflar üzerinde arama yapar ve maximum değere ulaştıran sınıfı seçer. a_i, A_i niteliği için ilgilenilen değerdir, yani $\forall a \in \mathcal{A}_i, a_i \implies A_i = a$. Ayrıca, (2.1) denkleminde $\Pr(c)$ tüm eğitim rastgele örneğinde c sınıfının olasılığıdır ve $\Pr(a_1, a_2, \dots, a_m | c)$, verilen c sınıfı tarafından koşullu X 'in ortak olasılığıdır. $\Pr(a_1, a_2, \dots, a_m | c)$ 'nin tahmininin NP-hard (Non-deterministic Polynomial-time hard) [44] olduğunun, bir eğitim rastgele örneklerinden elde edilen $\Pr(c)$ 'den hesaplama açısından çok daha karmaşık olduğunun kanıtlandığını vurgulamakta yarar bulunmaktadır.

Yorum 2.1. Rastgele değişkenlerin birbirleriyle ilişkisiz olmaları, birbirleriyle bağımsız oldukları anlamına gelmez ancak birbirleriyle bağımsız olmaları, birbirleriyle ilişkisiz oldukları anlamına gelmektedir.

Yorum 2.1 ışığı altında, bütün $c \in \mathcal{C}$ sınıflar için $A_1, A_2, \dots, A_m$ nitelikleri ilişkisizlerse bağımsızlardır diyemeyiz fakat bağımsızlarsa aynı zamanda ilişkisizdir diyebiliriz. Tüm özelliklerin bağımsız olması durumunda, (2.2) denklemini aşağıdaki gibi yeniden yazabiliriz:

$$\Pr(a_1, a_2, \dots, a_m | c) = \Pr(a_1 | c) \Pr(a_2 | c) \cdots \Pr(a_i | c) \Pr(a_m | c). \quad (2.2)$$

Burada $\Pr(a_i | c)$, $c \in \mathcal{C}$ sınıfında a_i niteliğinin olasılığını (başka bir ifadeyle, a_i niteliğinin özelliğini) ifade eder, yani $c \in \mathcal{C}$ sınıfı için A_i niteliğinin $a \in \mathcal{A}_i$ değerini alma olasılığı, $\Pr(a_i | c) \equiv \Pr(A_i = a | C = c)$. Önceki olasılık $\Pr(c)$ ve koşullu olasılık $\Pr(a_i | c)$ değerleri verilen eğitim örneklerinden kolayca tahmin edilebildiğinden, NB'nin öğrenilmesi oldukça kolay ve etkilidir.

Ortak koşullu olasılığı $\Pr(a_1, a_2, \dots, a_m | c)$ ifadesinin (2.2)'deki gibi yazılması durumunda, yani niteliklerin bağımsız olması durumunda, NB algoritmasının oldukça sadeleşeceği aşikardır; (2.3) numaralı denklem ile NB sınıflandırıcı aşağıdaki gibi

yeniden ifade edilebilir:

$$c(a) = \arg \max_{c \in \mathcal{C}} \Pr(c) \prod_{i=1}^m \Pr(a_i|c). \quad (2.3)$$

Lüteratürde varolan özellik ağırlıklandırma algoritmalar [19, 40–42, 51], verilen niteliklerinin birbirinden tamamen bağımsız olduğu varsayımından kaynaklanan performans kaybını azaltmak veya mümkünse ortadan kaldırmaya çalışır. Öyle ki, tahmin edici özellikleri önemlerine göre farklı şekilde ağırlıklandırır. Sonuç olarak FWNB sınıflandırıcısı ortaya çıkmaktadır:

$$c(a) = \arg \max_{c \in \mathcal{C}} \Pr(c) \prod_{i=1}^m \Pr(a_i|c)^{W_i}, \quad (2.4)$$

Burada, $W_i \in \mathbb{R}^+$, A_i ($i = 1, 2, \dots, m$) niteliği için elde edilen $\Pr(a_i|c)$ özelliğinin ağırlığıdır. FWNB sınıflandırıcısı kullanabilmek için $\forall c \in \mathcal{C}$ için $\Pr(c)$ marjinal olasılıkları ve $\forall a_i \in \mathcal{A}_i$ için $\Pr(a_i|c)$ özelliklerinin yanında $W_1, W_2, \dots, W_m$ ağırlıklarının da bulunması gerekir. Burada önemli olan ve araştırmacıların giderek daha fazla dikkatini çeken konu, her bir özellik ağırlığının nasıl belirleneceğidir. Aşağıda NB için özel olarak tasarlanmış son teknoloji özellik ağırlıklandırma yöntemlerinden bahsedeceğiz.

2.1 Kullback-Leibler Ölçüsü

Kullback-Leibler ölçüsü veya basitçe KL ölçüsü, veri madenciliği alanında aynı koşul altında iki olasılık dağılımı arasındaki farkı ölçmek için yaygın olarak kullanılmaktadır. KL ölçüm konsepti, olasılık teorisi ve bilgi teorisinden gelmektedir. Açık bir ifadeyle, göreceli entropi, bilgi sapması ve ayırım için bilgi ile yakından ilişkili olan KL ölçüsü, $\forall i \neq j$, A_i ve A_j niteliklerinin dağılımları arasındaki bilgi farkının simetrik olmayan bir ölçüsüdür. Spesifik olarak, A_i niteliğinin $\Pr(A_i|A_j = a) \equiv \Pr(A_i|a_j)$ koşullu altındaki dağılımından A_i niteliğinin koşulsuz $\Pr(A_i)$ dağılımına olan KL ölçüsü, $KL(A_i|a_j)$ olarak gösterilir ve $A_j = a$ olduğu bilindiğinde ortadan kalkan bilginin bir ortalama ölçüsü olup aşağıdaki gibi ifade edilir:

$$KL(A_i|a_j) = - \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \log(\Pr(a_i)) + \underbrace{\sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \log(\Pr(a_i|a_j))}_{\text{Ortadan kalkan bilginin ölçüsü}}, \quad (2.5a)$$

$$= - \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \left\{ \log(\Pr(a_i)) - \log(\Pr(a_i|a_j)) \right\}, \quad (2.5b)$$

Sonuç olarak, $KL(A_i|a_j)$ aşağıdaki gibi kapalı yapıda yazılabilir:

$$KL(A_i|a_j) = \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \log \left(\frac{\Pr(a_i|a_j)}{\Pr(a_i)} \right). \quad (2.6)$$

Eğer, A_i ve A_j nitelikleri birbirinden bağımsızlarsa, $\Pr(a_i|a_j) = \Pr(a_i)$ olacağından $\log(\Pr(a_i|a_j)/\Pr(a_i)) = 0$ olur; yani $KL(A_i|a_j) = 0$ olacağından A_i ve A_j nitelikleri bağımsız olduklarında birbirine hiç benzemiyor (yani $\Pr(a_i)$ ve $\Pr(a_i|a_j)$ dağılımları birbirinin aynısıdır) diyebiliriz. Ek olarak, $\forall i \neq j$ için A_i niteliğinin dağılımından A_j niteliğinin dağılıma olan KL ölçüsü $KL(A_i|A_j)$ ifade gösterilsin. $KL(A_i|A_j)$ ifadesi $KL(A_i|a_j)$ ölçüsünün A_j niteliğine göre ortalamasıdır; aşağıdaki gibi tanımlanır:

$$KL(A_i|A_j) = \sum_{a_j \in \mathcal{A}_j} \Pr(a_j) KL(A_i|a_j), \quad (2.7a)$$

$$= \sum_{a_i \in \mathcal{A}_i} \sum_{a_j \in \mathcal{A}_j} \Pr(a_i) \Pr(a_j) \log \left(\frac{\Pr(a_i|a_j)}{\Pr(a_i)} \right), \quad (2.7b)$$

$$= - \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \log(\Pr(a_i)) + \underbrace{\sum_{a_i \in \mathcal{A}_i} \sum_{a_j \in \mathcal{A}_j} \Pr(a_i) \Pr(a_j) \log(\Pr(a_i|a_j))}_{\text{Ortadan kalkan bilginin ortalama ölçüsü}}. \quad (2.7c)$$

Sonuç olarak, A_j niteliğinin bilinmesiyle A_i niteliğinin bilgi miktarındaki ortalama azalma, FW tabanlı NB sınıflandırıcılarda birbirine bağımlı niteliklerin özellik ağırlıklarının belirlenmesinde kullanılabilir. $KL(A_i|A_j)$ ölçüsü A_i ve A_j niteliklerin dağılımları arasındaki *mesafeyi* ölçse de, bir mesafe ölçüsü değildir. Bunun nedeni, KL ölçüsünün bir metrik olmamasıdır. Simetrik değildir: A_i niteliğinden A_j niteliğine olan KL ölçüsü A_j niteliğinden A_i niteliğine olan KL ölçüsü ile aynı değildir. Bununla birlikte, $KL(A_i|A_j)$ negatif olmayan bir ölçüdür: $KL(A_i|A_j) \geq 0$. Ayrıca, eğer $KL(A_i|A_j) = 0$ ise $A_i \equiv A_j$ doğrudur.

2.2 Özellik Ağırlıklandırma Yöntemleri

Jiang ve diğerleri tarafından vurgulandığı gibi [19], Ferreira ve ark. [51] çalışmasında NB sınıflandırması için oluşturulmuş literatürdeki ilk özellik ağırlıklandırma algoritmasını önermişlerdir. Zhang ve Sheng [39] NB sınıflandırıcıda her bir özelliğin ağırlığını belirleme zorluğunu ele almış ve her bir özelliğin ağırlığının Kazanç Oranı [Gain Ratio (GR)] ile orantılı olduğu göstermişlerdir. Sonuç olarak, Kazanç Oranı-tabanlı Özellik Ağırlıklandırma [Gain Ratio-based Feature Weighting (GRFW)] NB sınıflandırıcısını önermişlerdir[39]. Daha yüksek kazanç oranına sahip bir özelliğin daha yüksek bir ağırlık alması gerektiğini göstermişlerdir. Bu nedenle her özelliğin ağırlığını, tüm özelliklerdeki ortalama kazanç oranına göre özelliğin kazanç oranına

dönüştürmüşlerdir. GRFW tabanlı NB sınıflandırıcının matematiksel ifadesi aşağıda verilmiştir:

$$W_i = \frac{GR(A_i)}{\frac{1}{m} \sum_{i=1}^m GR(A_i)}. \quad (2.8)$$

Burada m , özelliklerin sayısıdır. $GR(A_i)$, verilen eğitim örneklerini bölmek için A_i niteliğinin $\forall a_i \in \mathcal{A}_i$ için $\Pr(a_i|c)$ özelliğini kullanmanın kazanç oranıdır.

[40] çalışmasında Hall, karar ağacı temelli bir özellik ağırlıklandırma yöntemi olan Karar Ağacı-tabanlı Özellik Ağırlıklandırma [Decision Tree-based Feature Weighting (DTFW)] önermiştir. DTFW algoritmasında bir özelliğin ağırlığı, oluşturulmuş karar ağacında test edildiği minimum derinlikle ters orantılıdır, yani A_i niteliğinin $\forall a_i \in \mathcal{A}_i$ için $\Pr(a_i|c)$ özelliğinin ağırlığı şu şekilde atanmaktadır:

$$W_i = \frac{1}{T} \sum_{t=1}^T \frac{1}{\sqrt{d_{ti}}}. \quad (2.9)$$

Burada d_{ti} , A_i özelliğinin oluşturulmuş karar ağacı, t 'de test edildiği minimum derinliktir ve T oluşturulan karar ağaçlarının toplam sayısıdır.

Yukarıda bahsedilen özellik ağırlıklandırma yöntemlerinin tümü, FWNB çalıştırılmadan önce genel veri özelliklerine dayanan sezgisel yöntemlere göre özellik ağırlıklarını doğrudan hesapladığı için FW filtreleri olarak görülebilir. Bu tez çalışmasında da bu ağırlıklandırma yöntemlerinin her biri FW filtresinin bir çeşidi olarak düşünülecektir.

Yukarıdaki FW filtrelerine ek olarak, özellik ağırlıklarını bir bütün olarak optimize etmek için FW tabanlı NB sınıflandırıcının performans geri bildirimini kullanan bir başka özellik ağırlıklandırma yöntemi kategorisi vardır. Bu kategoriye özellik ağırlıklandırma sarmalayıcıları (feature weighting wrappers) diyoruz. Örneğin, Wu ve Cai [41], son sınıflandırıcının doğruluğunu arttırırken, özellik ağırlıklarını optimize etmek için diferansiyel evrim araması kullanan bir Diferansiyel Evrim-tabanlı Özellik Ağırlıklandırma [Differential Evolution-based Feature Weighting (DEFW)] sınıflandırıcısını tanıtmıştır.

[42] çalışmasında Zaidi ve ark. koşullu log olasılığını en üst düzeye çıkararak veya nihai modelin ortalama karesel hatasını en aza indirerek özellik ağırlıklarını optimize etmek için gradyan iniş aramalarını kullanan sırasıyla Koşullu Log Olabilirliğe dayalı Özellik Ağırlıklandırma [Conditional Log Likelihood-based Feature Weighting (CLLFW)] ve Ortalama Kare alınmış Hata-tabanlı Özellik Ağırlıklandırma [Mean Squared Error-based Feature Based (MSEFW)] önerdi.

Lee ve ark. [13], özellik ağırlıklarını hesaplamak için bilgi içeriğinden yararlanan bir Kullback-Leibler Ölçüm-tabanlı Özellik Ağırlıklandırma [Kullback-Leibler Measure-based Feature Weighting (KLMFW)] yöntemini önermişlerdir. KLMFW yöntemi, belirli bir özellik değeri a_i gözlemlendiğinde, sınıf değişkeni C 'ye belirli bir miktarda bilgi verdiğini varsayar ve daha sonra bir özellik değerinin a_i 'den kaynaklanan içerdiği bilgi miktarını hesaplamak için Kullback-Leibler ölçüsünü

$$KL(C|a_i) = \sum_{c \in \mathcal{C}} \Pr(c|a_i) \log \frac{\Pr(c|a_i)}{\Pr(c)}, \quad (2.10)$$

kullanır. Ardından, A_i niteliğinin $\Pr(a_i|c)$ özelliği için ağırlığı, A_i niteliğinin tüm nitelik değerleri boyunca Kullback-Leibler ölçümlerinin ağırlıklı ortalaması olarak tanımlamıştır. Özellik ağırlıklarının değerleri 0 ile 1 arasında olması gerektiğinden, hesaplanan özellik ağırlıkları sonrasında ağırlıkların tümünün ortalamasına göre normalleştirilir. Kullback-Leibler ölçüsü kullanılarak özelliklerin ağırlıkların belirlenmesi için takip edilen işlem basamakları aşağıda gösterilmiştir:

$$W_i = \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) KL(C|a_i), \quad (2.11a)$$

$$= \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) \sum_{c \in \mathcal{C}} \Pr(c|a_i) \log \frac{\Pr(c|a_i)}{\Pr(c)}, \quad (2.11b)$$

$$= \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \Pr(a_i) \Pr(c|a_i) \log \frac{\Pr(c|a_i)}{\Pr(c)}, \quad (2.11c)$$

$$= \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \Pr(a_i, c) \log \frac{\Pr(c|a_i)}{\Pr(c)}, \quad (2.11d)$$

$$= \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \Pr(a_i, c) \log \frac{\Pr(c|a_i) \Pr(a_i)}{\Pr(c) \Pr(a_i)}, \quad (2.11e)$$

$$= \frac{1}{Z} \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \Pr(a_i, c) \log \frac{\Pr(c, a_i)}{\Pr(c) \Pr(a_i)}. \quad (2.11f)$$

Burada, Z değişkeni bulunan ağırlıkların ortalama değeridir ve özellik ağırlıklarının değerlerini $[0, 1]$ aralığına taşımak için kullanılan bir normalleştirme sabitidir. Açık bir ifadeyle,

$$Z = \frac{1}{m} \sum_{i=1}^m w_i, \quad (2.12)$$

olarak yazılır. Burada w_i değişkeni, $w_i = \sum_{a_i \in \mathcal{A}_i} \Pr(a_i) KL(C|a_i)$.

Jiang ve ark. [43], özellik ağırlıklı frekansları derinlemesine hesaplayarak koşullu olasılıkları tahmin eden bir Derin Özellik Ağırlıklandırma [Deep Feature Weighting (DeepFW)] yöntemi önerdi. DeepFW, tüm özellikler alanından en iyi özellik alt kümesini seçmek için korelasyon tabanlı özellik seçimi [Correlation-based Feature

Selection (CFS)] filtresi [52] kullanır ve ardından seçilen özellik alt kümesindeki özelliklere daha büyük ağırlıklar ve diğerlerine daha küçük ağırlıklar atar. Basit olması için seçilen özelliklerin ağırlıklarını 2 ya da 1'e göre ayarlamışlardır. Formül aşağıda gösterilmiştir:

$$W_i = \begin{cases} 2, & \text{eğer } A_i \text{ seçilmişse,} \\ 1, & \text{aksi halde.} \end{cases} \quad (2.13)$$

Mevcut diğer tüm özellik ağırlıklandırma yöntemlerinden farklı olarak DeepFW, öğrenilen özellik ağırlıklarını yalnızca NB'nin sınıflandırma formülüne değil, aynı zamanda koşullu olasılık tahminlerine de dahil eden tek yöntemdir.

Son zamanlarda, Jiang ve ark. [19], hem özellik-sınıf korelasyonunu (karşılıklı alaka) hem de ortalama özellik-özellik arası korelasyonu (ortalama karşılıklı artıklık) hesaba katan CFW NB sınıflandırıcı geliştirdi. Bu ağırlıklar [19]'de özellik-özellik ve özellik-sınıf korelasyonlarının kullanılmasıyla hesaplanır.

Yukarıda bahsettiğimiz NB için özel olarak tasarlanmış son teknoloji özellik ağırlıklandırma yöntemlerinin ardından Bölüm 3'de, bağımlılık metriği ve Bağımlı Özellik Ağırlıklandırma [Dependent Feature Weighting (DFW)] NB sınıflandırması öneriyoruz. Deneysel sonuçlar, DFW filtreli NB'nin, geleneksel NB ve diğer tüm özellik ağırlıklandırma teknikleri ile karşılaştırıldığında performansı arttırdığını gösterir.

BAĞIMLILIK METRİĞİ VE BAĞIMLI ÖZELLİK AĞIRLIKLI NAIVE BAYES

Çalışmamızda öncelikle nitelik-nitelik ve nitelik-sınıf niteliği arasındaki bağımlılığı ölçmek için bir bağımlılık metriği sunuyoruz. NB'in koşullu bağımsızlık varsayımını hafifletmeye yardımcı olmak için literatürde NB'ye yönelik geliştirilen bir dizi iyileştirme yöntemlerinden biri olan özellik ağırlıklandırma, her $\Pr(a_i|c)$ özelliğe önem düzeyine göre farklı bir ağırlık atar:

$$c(\mathbf{a}) = \arg \max_{c \in \mathcal{C}} \Pr(c) \prod_{i=1}^m \Pr(a_i|c)^{W_i}, \quad (3.1)$$

burada $W_i \in \mathbb{R}^+$, A_i ($i = 1, 2, \dots, m$) nitelik değerinin ağırlığıdır. Uygun özellik ağırlıkları, NB'nin gerektirdiği nitelik bağımsızlık varsayımının ihlalinden kaynaklanan hatayı azaltabilmektedir. NB sınıflandırmasında belirleyici özellikleri keşfetmek için FFS ve FCS öneriyoruz. FFS ve FCS'den özellik ağırlıklarının nasıl bulunacağını gösteriyoruz ve yeni bir DFW NB sınıflandırması öneriyoruz.

3.1 Bağımlılık Metriği

Tüm $i \neq j \in \mathbb{N}^{[1,m]}$ için A_i ve A_j , rastgele örnek T 'nin iki niteliği olsun. Bu noktada iki önemli çıkarımımız bulunmaktadır.

- $i \neq j$ için A_i ve A_j karşılıklı olarak bağımsız kabul edildiğini varsayalım. Bu durumda ortak olasılıkları $\Pr(a_i, a_j)$, marjinal olasılıkları $\Pr(a_i)$ ve $\Pr(a_j)$ olarak yeniden yazılabilir. Bu durumda tüm $i \neq j$ için $\Pr(a_i, a_j) = \Pr(a_i)\Pr(a_j)$ 'dir. Dolayısıyla, tüm $n \in \mathbb{N}$ için,

$$\Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n = 0. \quad (3.2)$$

- $i \neq j$ için A_i ve A_j karşılıklı olarak bağımsız değilse, ortak olasılıkları $\Pr(a_i, a_j)$

marjinal olasılıkları $\Pr(a_i)$ ve $\Pr(a_j)$ ile yazılamaz. Tüm $n \in \mathbb{N}$ için,

$$\Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n = \mu_n(a_i, a_j), \quad (3.3)$$

ile sonuçlanır. Burada $\mu_n(a_i, a_j)$, A_i ve A_j nitelikleri arasındaki karşılıklı bağımlılığın neden olduğu bağımsızlık eksikliğidir. Bağımsızlarsa, $\forall i \neq j$, $\mu_n(a_i, a_j) = 0$ elde edilir.

Teorem 3.1. $\mu_n(a_i, a_j)$ bağımsızlığının olmaması şu şekilde sınırlandırılmıştır:

$$-1 \leq 4\mu_n(a_i, a_j) \leq 1. \quad (3.4)$$

İspat. Toplam Olasılık Yasasını (Law of Total Probability) [53, Thm. 2.5] kullanarak, $\forall a_i \in \mathcal{A}_i, \Pr(a_i)$ ve $\forall a_j \in \mathcal{A}_j, \Pr(a_j)$ marjinal olasılıklarını aşağıdaki gibi yazabiliriz:

$$\Pr(a_i) = \sum_{\alpha \in \mathcal{A}_j} \Pr(a_i, \alpha), \text{ ve } \Pr(a_j) = \sum_{\alpha \in \mathcal{A}_i} \Pr(\alpha, a_j). \quad (3.5)$$

Bu iki ifadenin çarpımı

$$\Pr(a_i) \Pr(a_j) = \sum_{\alpha \in \mathcal{A}_i} \sum_{\beta \in \mathcal{A}_j} \Pr(\alpha, a_j) \Pr(a_i, \beta), \quad (3.6)$$

şeklinde yazılır ve en küçük üst sınırı

$$\sup \Pr(a_i) \Pr(a_j) = \frac{1}{|\mathcal{A}_i| |\mathcal{A}_j|}, \quad (3.7a)$$

$$\leq \frac{1}{4}, \text{ çünkü } |\mathcal{A}_i| \geq 2 \text{ ve } |\mathcal{A}_j| \geq 2, \quad (3.7b)$$

olarak bulunur. Burada $\forall i \in \{1, 2, \dots, m\}$, $\mathcal{A}_i$ kümesi A_i niteliğinin alabileceği değerlerin kümesidir. $|\mathcal{A}_i|$ ise $\mathcal{A}_i$ kümesinin eleman sayısını (büyüklüğünü) ifade etmektedir. Eğer $|\mathcal{A}_i| = 1$ olursa A_i niteliği bütün örnekler için aynı değere sahip olacağından rastgele bir değişken olmaz ve dolayısıyla sınıflandırmada kullanılması gereksizdir. (3.5)'den $\Pr(a_i, a_j)$ bileşik olasılığı aşağıdaki gibi yazılabilir:

$$\Pr(a_i, a_j) = \Pr(a_i) - \sum_{a \in \mathcal{A}_j \setminus \{a_j\}} \Pr(a_i, a), \quad (3.8a)$$

$$= \Pr(a_j) - \sum_{a \in \mathcal{A}_i \setminus \{a_i\}} \Pr(a, a_j). \quad (3.8b)$$

Buradan, (3.7) yardımıyla, $\sup \Pr(a_i, a_j) = \frac{1}{2}$ ve $\inf \Pr(a_i, a_j) = 0$ bulunur. Dolayısıyla, $\mu_0(a_i, a_j) = 0$ ve

$$-\frac{1}{4} \leq \mu_1(a_i, a_j) \leq \frac{1}{4}. \quad (3.9)$$

Sonuç olarak, $\forall n \in \mathbb{N}_+$, $|\mu_n(a_i, a_j)| \leq |\mu_1(a_i, a_j)|$ olacağından, Teorem 3.1'in ispatı tamamlanmış olur. ■

Teorem 3.2. *Tüm $i \neq j \in \mathbb{N}^{[1,m]}$ için A_i ve A_j , rastgele örnek T 'nin iki niteliği olsun. Ortak olasılığı $\Pr(a_i, a_j)$ marjinal olasılıkları $\Pr(a_i)$ ve $\Pr(a_j)$ ile aşağıdaki gibi sınırlıdır:*

$$\inf \Pr(a_i, a_j)^2 = 0, \quad (3.10)$$

$$\sup \Pr(a_i, a_j)^2 = \frac{1}{\inf |\mathcal{A}_i| |\mathcal{A}_j|} = \frac{1}{4}. \quad (3.11)$$

İspat. Öncelikle $\inf \Pr(a_i, a_j) = 0$ olduğu aşıkardır ve $(\inf \Pr(a_i, a_j))^2 = \inf \Pr(a_i, a_j)^2$ olduğundan $\inf \Pr(a_i, a_j)^2 = 0$ 'dır. $\sup \Pr(a_i, a_j)^2$ ifadesini bulabilmek için (3.8a) ve (3.8b) kullanarak

$$\Pr(a_i, a_j)^2 = \Pr(a_i, a_j) \Pr(a_i, a_j), \quad (3.12a)$$

$$= \left(\Pr(a_i) - \sum_{a \in \mathcal{A}_j \setminus \{a_j\}} \Pr(a_i, a) \right) \left(\Pr(a_j) - \sum_{a \in \mathcal{A}_i \setminus \{a_i\}} \Pr(a, a_j) \right), \quad (3.12b)$$

ifadesini elde ederiz. Buradan, $\Pr(a_i, a_j)^2$ için en küçük üst sınırını

$$\sup \Pr(a_i, a_j)^2 = \sup \Pr(a_i, a_j) \sup \Pr(a_i, a_j), \quad (3.13a)$$

$$= \left(\sup \Pr(a_i) - \inf \sum_{a \in \mathcal{A}_j \setminus \{a_j\}} \Pr(a_i, a) \right) \times \left(\sup \Pr(a_j) - \inf \sum_{a \in \mathcal{A}_i \setminus \{a_i\}} \Pr(a, a_j) \right), \quad (3.13b)$$

$$= \left(\sup \Pr(a_i) - \sum_{a \in \mathcal{A}_j \setminus \{a_j\}} \inf \Pr(a_i, a) \right) \times \left(\sup \Pr(a_j) - \sum_{a \in \mathcal{A}_i \setminus \{a_i\}} \inf \Pr(a, a_j) \right), \quad (3.13c)$$

$$= \sup \Pr(a_i) \sup \Pr(a_j), \quad (3.13d)$$

$$= \sup \Pr(a_i) \Pr(a_j), \quad (3.13e)$$

$$= \frac{1}{\inf |\mathcal{A}_i| |\mathcal{A}_j|}, \quad (3.13f)$$

$$= \frac{1}{4}, \text{ çünkü } |\mathcal{A}_i| \geq 2 \text{ ve } |\mathcal{A}_j| \geq 2, \quad (3.13g)$$

olarak bulunur. Sonuç olarak, Teorem 3.2'in ispatı tamamlanmış olur. ■

Teorem 3.3. $\mu_n(a_i, a_j)$ bağımsızlığının olmaması ifadesinin en büyük alt sınırı $\inf \mu_n(a_i, a_j)$ ve en küçük üst sınırı $\sup \mu_n(a_i, a_j)$ aşağıdaki gibidir:

$$\inf \mu_n(a_i, a_j) = -|\mathcal{A}_i|^{-n} |\mathcal{A}_j|^{-n} \geq -4^{-n}, \quad (3.14)$$

$$\sup \mu_n(a_i, a_j) = |\mathcal{A}_i|^{-n} |\mathcal{A}_j|^{-n} (|\mathcal{A}_i|^{\frac{n}{2}} |\mathcal{A}_j|^{\frac{n}{2}} - 1) \leq 4^{-n} (2^n - 1). \quad (3.15)$$

İspat. (3.3)'de verilen bağımsızlık eksikliği ifadesinin en büyük alt sınırı yani $\inf \mu_n(a_i, a_j)$ aşağıdaki gibi elde edilir:

$$\inf \mu_n(a_i, a_j) = \inf \{ \Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n \}, \quad (3.16)$$

$$= \inf \Pr(a_i, a_j)^n - \sup \Pr(a_i)^n \Pr(a_j)^n. \quad (3.17)$$

Teorem 3.2'de verilen (3.10) ifadesinin ve (3.7a) ifadesinin birlikte kullanılmasıyla,

$$\inf \mu_n(a_i, a_j) = \inf \Pr(a_i, a_j)^n - \sup \Pr(a_i)^n \Pr(a_j)^n, \quad (3.18a)$$

$$= (\inf \Pr(a_i, a_j))^n - (\sup \Pr(a_i) \Pr(a_j))^n, \quad (3.18b)$$

$$= -(\sup \Pr(a_i) \Pr(a_j))^n = -|\mathcal{A}_i|^{-n} |\mathcal{A}_j|^{-n}, \quad (3.18c)$$

elde edilir. Dolayısıyla, $\inf \mu_n(a_i, a_j) \geq -4^{-n}$ çünkü $|\mathcal{A}_i| \geq 2$ ve $|\mathcal{A}_j| \geq 2$. (3.13e) denkleminin kullanılmasıyla $\Pr(a_i, a_j) \leq \sqrt{\Pr(a_i) \Pr(a_j)}$ sonucunu elde ederiz. Yani, en küçük üst sınır $\sup \mu_n(a_i, a_j)$ ifadesi elde edilirken $\sup \Pr(a_i, a_j)$ ve $\inf \Pr(a_i) \Pr(a_j)$ birbiriyle sınırlıdır. $\sup \mu_n(a_i, a_j)$ ifadesi

$$\sup \mu_n(a_i, a_j) = \sup \{ \Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n \}, \quad (3.19a)$$

$$= \sup \{ \sup \Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n \}, \quad (3.19b)$$

gibi elde edilir. Burada, (3.13e) denkleminin kullanılmasıyla elde ettiğimiz $\Pr(a_i, a_j) \leq \sqrt{\Pr(a_i) \Pr(a_j)}$ ifadesini dikkate aldığımızda

$$\sup \mu_n(a_i, a_j) = \sup \{ \sup \Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n \}, \quad (3.20a)$$

$$\stackrel{(a)}{=} \sup \{ \Pr(a_i)^{\frac{n}{2}} \Pr(a_j)^{\frac{n}{2}} - \Pr(a_i)^n \Pr(a_j)^n \}, \quad (3.20b)$$

$$= \sup \Pr(a_i)^n \Pr(a_j)^n (\Pr(a_i)^{-\frac{n}{2}} \Pr(a_j)^{-\frac{n}{2}} - 1), \quad (3.20c)$$

$$= |\mathcal{A}_i|^{-n} |\mathcal{A}_j|^{-n} (|\mathcal{A}_i|^{\frac{n}{2}} |\mathcal{A}_j|^{\frac{n}{2}} - 1), \quad (3.20d)$$

Burada, (a) adımımda (3.13e) denkleminde $\sup \Pr(a_i, a_j)^2 = \sup \Pr(a_i) \Pr(a_j)$ ifadesi kullanılmıştır. Dolayısıyla, (3.13f) sayesinde $\sup \mu_n(a_i, a_j) \leq 4^{-n}(2^n - 1)$ olur çünkü $|\mathcal{A}_i| \geq 2$ ve $|\mathcal{A}_j| \geq 2$. Böylece, Teorem 3.3'in ispatı tamamlanmış olur. ■

Tanım 3.1. $\mathcal{X}$ kümesi üzerinde tanımlı olan bir fonksiyon $d: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$ ($\mathbb{R}_+$ pozitif reel sayılar kümesini ifade eder) $\forall x, y \in \mathcal{X}$ için aşağıdaki ölçütleri sağlıyorsa bir semimetriktir:

i) $d(x, y) \geq 0$ (negatif olmama),

ii) $d(x, y) = 0$ ancak ve ancak $x \equiv y$ (kesinlik),

iii) $d(x, y) = d(y, x)$ (simetri).

Tanım 3.2. $\mathcal{X}$ kümesi üzerinde tanımlı bir semimetrik $d: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$, $\forall x, y, z \in \mathcal{X}$ için $d(x, z) \leq d(x, y) + d(y, z)$ (toplamsallık / üçgen eşitsizliği) ölçütünü sağlıyorsa bir metriktir.

Tüm $i \neq j \in \mathbb{N}^{[1, m]}$ için A_i ve A_j nitelikleri arasında tanımlı olan *bağımsızlık eksikliğinin karesi* $\mu_n^2(a_i, a_j)$, Tanım 3.1’da verilen (i) negatif olmama, (ii) kesinlik, (iii) simetri ölçütlerini sağladığı kolaylıkla gösterilebilir:

- i) $\mu_n^2(a_i, a_j) \geq 0$ (negatif olmama),
- ii) $\mu_n^2(a_i, a_j) = 0$ ancak ve ancak A_i ve A_j nitelikleri bağımsızda (kesinlik),
- iii) $\mu_n^2(a_i, a_j) = \mu_n^2(a_j, a_i)$ (simetri).

Dolayısıyla, *bağımsızlık eksikliğinin karesi* $\mu_n^2(a_i, a_j)$ semimetriktir. Metrik olduğunu söyleyebilmemiz için, Tanım 3.2’da verilen son ölçütün sağlandığını göstermeliyiz, yani bütün $i \neq j \neq k \in \mathbb{N}^{[1, m]}$ indisleri için $\mu_n^2(a_i, a_k) \leq \mu_n^2(a_i, a_j) + \mu_n^2(a_j, a_k)$ eşitsizliğinin olduğunu ispatlamamız gerekmektedir. Dolayısıyla, (3.3) yardımıyla

$$\left(\Pr(a_i, a_k)^n - \Pr(a_i)^n \Pr(a_k)^n \right)^2 \leq \left(\Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n \right)^2 + \left(\Pr(a_j, a_k)^n - \Pr(a_j)^n \Pr(a_k)^n \right)^2, \quad (3.21)$$

olduğunu göstermeliyiz. Hem A_i niteliğinin A_j niteliğiyle ve A_j niteliğinin A_k niteliğiyle bağımsız olduklarını kabul ettiğimizde, A_i ve A_k nitelikleri bağımsızdır diyemeyiz. Dolayısıyla, $\mu_n^2(a_i, a_k) > 0$ iken $\mu_n^2(a_i, a_j) = 0$ ve $\mu_n^2(a_j, a_k) = 0$ olur; yani (3.21) denklemi sağlanmadığından *bağımsızlık eksikliğinin karesi* $\mu_n^2(a_i, a_j)$ semimetriktir ama metrik değildir.

Yukarıda bahsedilen çıkarımlardan, A_i ve A_j niteliklerinin birbirine ne ölçüde bağımlı olduğunu belirlemek için toplam benzerliği ölçmek adına yaygın olarak kullanılan Karesel Hataların Toplamı [Sum of Squared Errors (SSE)] adı verilen istatistiksel bir teknikten yararlanabiliriz. $D(A_i; A_j)$ belirli bir bağımlılık derecesini gösterdiğini varsaydığımızda,

$$D(A_i; A_j) \propto \sum_{a_i \in \mathcal{A}_i} \sum_{a_j \in \mathcal{A}_j} \sum_{n=0}^{\infty} \mu_n^2(a_i, a_j), \quad (3.22)$$

çıkarımını yapabiliriz. Aşağıda, bağımlılık metriklerini kapalı biçimde öneriyoruz. Burada, $\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j)$ toplamının yakınsak olduğunu ispatlamamız gerekmektedir. Bu toplamın terimlerinin pozitif olduğundan dolayı, öyle bir $\{b_0, b_1, \dots, b_n, \dots\} > 0$, seti olabilir ki

$$\lim_{n \rightarrow \infty} \frac{\mu_n^2(a_i, a_j)}{b_n} \neq 0, \quad (3.23)$$

olması durumunda $\sum_{n=1}^{\infty} b_n$ yakınsıyorsa $\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j)$ toplamıda yakınsayacaktır. Yakınsak olduğunu ispatlayabilmek için

$$\sum_{n=0}^{\infty} \sup_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) < \infty, \quad \text{ve} \quad \sum_{n=0}^{\infty} \inf_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) < \infty, \quad (3.24)$$

olduğunu göstermemiz yeterli olacaktır. Çok iyi bilinen, Mutlak Yakınsaklık (Absolute Convergence) teoremini [54, Bölüm 0.21] kullanarak

$$\sum_{n=0}^{\infty} \inf_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) = \sum_{n=0}^{\infty} \left| \inf_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) \right| = \sum_{n=0}^{\infty} \left| -\frac{1}{4^n} \right|^2 = \frac{16}{15} < \infty, \quad (3.25)$$

$$\sum_{n=0}^{\infty} \sup_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) = \sum_{n=0}^{\infty} \left| \sup_{\substack{a_i \in \mathcal{A}_i \\ a_j \in \mathcal{A}_j}} \mu_n^2(a_i, a_j) \right| = \sum_{n=0}^{\infty} \left| \frac{1}{2^n} - \frac{1}{4^n} \right|^2 = \frac{4}{45} < \infty, \quad (3.26)$$

Sonuç olarak, $\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j)$ toplamı yakınsaktır. Yani, (3.22) denklemini kullanarak bağımlılık derecesini tanımlayabiliriz.

Teorem 3.4 (Bağımlılık Metriği). $D(A_i, A_j)$ ile gösterilen, rastgele T örneğinin iki niteliği A_i ve A_j arasındaki bağımlılığın bir ölçüsü şu şekilde verilir:

$$D(A_i; A_j) = \sum_{a_i \in \mathcal{A}_i} \sum_{a_j \in \mathcal{A}_j} \left(\frac{2 \Pr(a_i, a_j)}{\Pr(a_i) \Pr(a_i, a_j) \Pr(a_j) - 1} + \frac{\Pr(a_i, a_j)}{1 - \Pr(a_i)^2 \Pr(a_j)^2} + \frac{\Pr(a_i, a_j)}{1 - \Pr(a_i, a_j)^2} \right), \quad (3.27)$$

Benzer şekilde, A_i niteliği ile C sınıf niteliği arasındaki $D(A_i, C)$ ile gösterilen bir bağımlılık ölçüsü şu şekilde verilir:

$$D(A_i; C) = \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \left(\frac{2 \Pr(a_i, c)}{\Pr(a_i) \Pr(a_i, c) \Pr(c) - 1} + \frac{\Pr(a_i, c)}{1 - \Pr(a_i)^2 \Pr(c)^2} + \frac{\Pr(a_i, c)}{1 - \Pr(a_i, c)^2} \right), \quad (3.28)$$

Burada $\mathcal{A}_i$ ve $\mathcal{C}$, sırasıyla A_i niteliğinin ve C sınıf niteliğinin örnek uzaylarıdır.

İspat. Bağımsızlık eksikliği ifadesi $\sup \mu_n(a_i, a_j)$ 'nin karesi, (3.3) denklemi ve binom açılımını [54, Eq. (1.111)] birlikte kullanarak, $\mu_n^2(a_i, a_j)$ ifadesi

$$\mu_n^2(a_i, a_j) = (\Pr(a_i, a_j)^n - \Pr(a_i)^n \Pr(a_j)^n)^2, \quad (3.29)$$

$$= \Pr(a_i, a_j)^{2n} - 2 \Pr(a_i, a_j)^n \Pr(a_i)^n \Pr(a_j)^n + \Pr(a_i)^{2n} \Pr(a_j)^{2n}, \quad (3.30)$$

şeklinde yazılabilir. Dolayısıyla, bütün $n \in \mathbb{N}$ için $\mu_n^2(a_i, a_j)$ 'lerin toplamını, yani $\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j)$ ifadesi aşağıdaki gibi

$$\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j) = \sum_{n=0}^{\infty} \Pr(a_i, a_j)^{2n} - 2 \sum_{n=0}^{\infty} \Pr(a_i, a_j)^n \Pr(a_i)^n \Pr(a_j)^n + \sum_{n=0}^{\infty} \Pr(a_i)^{2n} \Pr(a_j)^{2n} \quad (3.31)$$

yazabiliriz. Burada her bir toplam ifadesinin terimleri, olasılık aksiyomları gereği,

$$0 \leq \Pr(a_i, a_j)^{2n} \leq 1, \quad (3.32)$$

$$0 \leq \Pr(a_i, a_j)^n \Pr(a_i)^n \Pr(a_j)^n \leq 1, \quad (3.33)$$

$$0 \leq \Pr(a_i)^{2n} \Pr(a_j)^{2n} \leq 1, \quad (3.34)$$

sınırlıdır. $\forall |x| < 1$ için $\sum_{n=0}^{\infty} x^n = (1 - x)^{-1}$ [54, Eq. (0.112)] yardımıyla, (3.31)

$$\sum_{n=0}^{\infty} \mu_n^2(a_i, a_j) = \frac{1}{1 - \Pr(a_i, a_j)^2} + \frac{1}{1 - \Pr(a_i)^2 \Pr(a_j)^2} + \frac{2}{\Pr(a_i, a_j) \Pr(a_i) \Pr(a_j) - 1} \quad (3.35)$$

olarak elde edilir. A_i ve A_j nitelikleri arasında bağımlılığın olması durumu bileşik olasılıklarının yüksek olduğunda etkili olacaktır. Bu durumda A_i ve A_j nitelikleri arasındaki bağımlılık

$$D(A_i; A_j) = \sum_{a_i \in \mathcal{A}_i} \sum_{a_j \in \mathcal{A}_j} \Pr(a_i, a_j) \sum_{n=0}^{\infty} \mu_n^2(a_i, a_j), \quad (3.36)$$

şeklinde tanımlanır. Benzer şekilde A_i niteliği ile C sınıfı arasındaki bağımlılıkta aşağıdaki gibi yazılır.

$$D(A_i; C) = \sum_{a_i \in \mathcal{A}_i} \sum_{c \in \mathcal{C}} \Pr(a_i, c) \sum_{n=0}^{\infty} \mu_n^2(a_i, c), \quad (3.37)$$

Sonuç olarak, (3.36) ve (3.37) denklemlerinde (3.35) ifadesinin yerleştirilmesiyle, sırasıyla (3.27) ve (3.28) elde edilerek Teorem 3.4'in ispatı tamamlanmış olur. ■

3.2 Bağımlı Özellik Ağırlıklandırılmalı Naive Bayes Sınıflandırıcılar

Özellik ağırlıklandırma filtresi, her bir özelliğe önemine göre ayrı bir ağırlık atar; bu, özellik uzayındaki eksenleri, her bir özelliğin önemine göre belirlenen germe

derecesi ile germeye eşdeğerdir. NB performansını artırmak için eksenleri germek, daha önemli özelliklere ilham verir ve daha az önemli özellikleri bastırır. Son derece kestirimci bir özelliğin, diğer özelliklerden bağımsız kalırken sınıf özelliğine güçlü bir şekilde bağımlı olması gerektiği göz önüne alındığında, önemini FCS ile ortalama FFS arasındaki fark olarak tanımlarız ve ağırlığının aşağıdaki teoremden tanımlanan bu anlamla orantılı olması gerektiğini öneriyoruz.

Teorem 3.5 (Bağımlı Özellik Ağırlıkları). *DFW filtresi, tüm $i = 1, 2, \dots, m$ için özellik ağırlıklarının $W_i = 1/(\exp(-S_i) + 1)$ olarak seçilmesini önerir. Burada, S_i değişkeni A_i özelliğinin sınıflandırmadaki önemini ifade etmektedir ve*

$$S_i = S(A_i; C) - \frac{1}{m-1} \sum_{j=1 \wedge j \neq i}^m S(A_i; A_j) \quad (3.38)$$

olarak verilmektedir. Burada m , sınıf özelliği dışındaki özelliklerin sayısıdır. Ayrıca $S(A_i; C)$ ve $S(A_i; A_j)$ sırasıyla aşağıdaki şekilde tanımlanan nitelik-sınıf önemini ve nitelik-nitelik önemini ifade eder.

$$S(A_j; C) = \frac{D(A_i; C)}{\frac{1}{m} \sum_{k=1}^m D(A_k; C)}, \quad (3.39)$$

$$S(A_i; A_j) = \frac{D(A_i; A_j)}{\frac{1}{m(m-1)} \sum_{k=1}^m \sum_{\ell=1 \wedge \ell \neq k}^m D(A_k; A_\ell)}. \quad (3.40)$$

İspat. Tüm $i \neq j \in \mathbb{N}^{[1,m]}$ için A_i ve A_j rastgele değişkenleri, rastgele örnek T 'nin iki niteliği olsun ve C rastgele değişkeni de ilgili sınıfı olsun. Eğer, A_i niteliği A_j niteliğine göre son derece kestirimci bir nitelikse, A_i 'nin C olan bağımlılığı A_j 'nin C ile olan bağımlılığından büyük olacaktır, yani $D(A_i, C) \geq D(A_j, C)$ olacaktır. Dolayısıyla, C sınıf değişkeninin bütün niteliklerle olan bağımlılığının ortalama miktarı

$$D(C) = \frac{1}{m} \left(D(A_1; C) + D(A_2; C) + \dots + D(A_m; C) \right), \quad (3.41)$$

olarak yazılır. A_i niteliğinin C sınıf değişkeni için FCS

$$S(A_i; C) = \frac{D(A_i; C)}{D(C)}, \quad (3.42)$$

olarak tanımlanır. Eğer $S(A_i; C) > 1$ ise A_i niteliği C sınıfı için önemli aksi halde önemsizdir. (3.41) ifadesinin (3.42)'de yerine konmasıyla (3.39) elde edilir. Böylece, Teorem 3.5'in ispatının ilk aşaması tamamlanmış olur. İspatın ikinci aşamasında,

rastgele örnek T 'nin farklı iki nitelikleri arasındaki bağımlılığın ortalama miktarı

$$D(T) = \frac{1}{m(m-1)} \sum_{k=1}^m \sum_{\ell=1 \wedge \ell \neq k}^m D(A_k; A_\ell), \quad (3.43)$$

olarak yazılır. A_i niteliğinin A_j niteliği için FFS değeri

$$S(A_i; A_j) = \frac{D(A_i; A_j)}{D(T)}, \quad (3.44)$$

olarak tanımlanır. (3.43) ifadesi (3.44)'de yerine konarak (3.40) elde edilir ve Teorem 3.5'in ispatının ikinci aşaması tamamlanmış olur. İspatın son aşamasında, $\forall i \in \mathbb{N}^{[1,m]}$ için A_i niteliğinin sınıflandırmadaki önemini ifade eden S_i elde edilecektir. Eğer $i \neq j \neq k$ için $S(A_i; A_j) > S(A_i; A_k)$ ise A_i ile A_j arasındaki bağımlılık A_i ile A_k arasındaki bağımlılıktan daha büyüktür. Fakat, NB sınıflandırıcının başarımının yüksek olabilmesi için $A_1, A_2, \dots, A_m$ nitelikleri arasındaki bağımlılığın olmaması gerekmektedir[14]. Yüksek derecede önemli bir nitelik A_i olsun. Bu A_i niteliği diğer niteliklerden bağımsız kalırken büyük ölçüde sınıfa C bağımlı olması gerektiğinden, A_i niteliğinin önemi S_i , FCS ve ortalama FFS arasındaki farkla orantılı olmalıdır. Sonuç olarak, S_i ifadesi (3.38) olarak bulunur. S_i 'nin değeri negatif olabileceğinden, ancak özellik ağırlıklı NB'nin ihtiyaç duyduğu ağırlıklar pozitif olduğundan, S_i 'nin değerini $(0, 1)$ aralığına dönüştürmek için tipik bir lojistik sigmoid işlevi kullanırız. Sonuç olarak, A_i 'nin ağırlığının nihai biçimi $W_i = 1/(\exp(-S_i) + 1)$ olarak belirlenir ve Teorem 3.5'in ispatı tamamlanmış olur. ■

Tüm özellik ağırlıklarını elde etmek için Teorem 3.3'ü kullanarak, sınıflandırma formülü (3.9) için aynı olan DFwNB sınıflandırıcısını öneriyoruz.

$$c(\mathbf{a}) = \arg \max_{c \in \mathcal{C}} \widehat{\Pr}(c) \prod_{i=1}^m \widehat{\Pr}(a_i|c)^{W_i}, \quad (3.45)$$

burada $\widehat{\Pr}(c)$ ve $\widehat{\Pr}(a_i|c)$, c sınıfının tahmin edilen önceki olasılığı ve tahmini koşullu olasılıktır, sırasıyla m -estimation kullanarak [42, Eq. (20)] ve CFw [19, Eqs. (15) ve (16)]. Şimdi, bağımlı özellik ağırlıklandırmamız (kısaca DFw) için ayrıntılı öğrenme algoritması, Algoritma 1 olarak kısaca gösterilmiştir ve Bölüm A'da WEKA kütüphanesine bir ek modül olacak şekilde JAVA dilindeki gerçekleştirilmesi muhtemel okuyucularla paylaşılmıştır. Rastgele örnek numunelerinin (instance) bir dizi kaynaktan gelebileceği göz önüne alındığında, rastgele örneğin özellikleri arasındaki homojen olmayan bağımlılık, gerçek dünya uygulamalarında doğal bir durumdur. Daha sonra, K-mean[7] gibi bir kümeleme tekniği kullanarak, eğitim rastgele örneğini T n sayıda örnekle eğitim rastgele alt örneklerine bölmeyi savunuyoruz $T_1, T_2, \dots, T_q$

Algoritma 1. Bağımlı Özellik Ağırlıklandırma

Girdi: Bir eğitim veri seti

```
1  FOR her nitelik  $A_i$  ( $i = 1, 2, \dots, m$ ) için DO
2     $\forall a_i \in \mathcal{A}_i$  için  $\text{Pr}(a_i)$  marjinal olasılıklarını hesapla
3  END FOR
4  FOR her çift nitelik  $A_i$  ve  $A_j$  ( $j \neq i$ ) için DO
5     $\forall a_i \in \mathcal{A}_i$  ve  $\forall a_j \in \mathcal{A}_j$  için  $\text{Pr}(a_i, a_j)$  bileşik olasılıklarını hesapla
6  END FOR
7  FOR her nitelik  $A_i$  ( $i = 1, 2, \dots, m$ ) için DO
8     $D(A_i, C)$ 'yi (3.28) denklemiyle hesapla
9  END FOR
10 FOR her çift nitelik  $A_i$  ve  $A_j$  ( $j \neq i$ ) için DO
11    $D(A_i, A_j)$ 'yi (3.27) denklemiyle hesapla
12 END FOR
13 FOR her nitelik  $A_i$  ( $i = 1, 2, \dots, m$ ) için DO
14    $S(A_i; C)$ 'yi (3.39) denklemiyle hesapla
15 END FOR
16 FOR her çift nitelik  $A_i$  ve  $A_j$  ( $j \neq i$ ) için DO
17    $S(A_i; A_j)$ 'yi (3.40) denklemiyle hesapla
18 END FOR
19 FOR her nitelik  $A_i$  ( $i = 1, 2, \dots, m$ ) için DO
20    $S_i$ 'yi (3.38) denklemiyle hesapla
21   Özellik ağırlığını  $W_i = 1/(\exp(-S_i) + 1)$  olarak hesapla
22 END FOR
```

ÇIKTI: Bütün özelliklerin ağırlıkları W_i ($i = 1, 2, \dots, m$)

öyle ki $T = T_1 \cup T_2 \cup \dots \cup T_q$. Teorem 3.3'ün yardımıyla, her rastgele alt örnek için özellik ağırlıklarını hesaplıyoruz ve $\mathbf{a} = \langle a_1, a_2, \dots, a_m \rangle$ örneğini $q \in \mathbb{N}^+$ rastgele alt örneklerin sayısı olarak sınıflandıran CDFW NB sınıflandırıcısı öneriyoruz.

$$c(\mathbf{a}) = \arg \max_{c \in \mathcal{C}} \sum_{k=1}^q \rho_{(k)} \hat{\text{Pr}}_{(k)}(c) \prod_{i=1}^m \hat{\text{Pr}}_{(k)}(a_i|c)^{W_{ki}}, \quad (3.46)$$

Burada $\hat{\text{Pr}}_{(k)}(c)$ ve $\hat{\text{Pr}}_{(k)}(a_i|c)$, T_k için sırasıyla c sınıfının tahmini önsel olasılığı ve tahmini koşullu olasılığıdır. W_{ki} , Teorem 3.3 tarafından T_k için hesaplanan A_i niteliğinin özellik ağırlığıdır. Ayrıca, $\rho_{(k)}$, k . DFWNB sınıflandırıcı kullanılarak T_k 'de doğru sınıflandırılan örneklerin sayısını temsil eden σ_k ile $\rho_{(k)} = \sigma_k/n$ tarafından sağlanan T_k için doğru sınıflandırma olasılığıdır. Pratik uygulamalarda, seçilen küme sayısı q , rastgele örnekten rastgele örneğe değişir. Uygulamamızda, $q=2$ ayarının iyi sonuçlar verdiğine dair bir gözlemimiz var. Bölüm B'da WEKA kütüphanesine bir ek modül olacak şekilde CDFW NB sınıflandırıcısının JAVA dilindeki gerçekleştirme kodları muhtemel okuyucularla paylaşılmıştır. Bir sonraki bölümde, deneysel sonuçlarımızı sunuyoruz.

4 DENEYSEL SONUÇLAR

Bu bölümde, öncelikle deneylerimizde kullandığımız 32 UCI veri kümesini tanıtıyoruz. 32 UCI veri kümesinden yararlanarak, DFW ve CDFW ile NB, CFW, KLMFW, GRFW, DTFW, DeepFW, DEFW, CLLFW ve MSEFW yöntemleri arasında doğruluk karşılaştırmaları yapıyoruz. Oluşturduğumuz Doğruluk Karşılaştırmaları tablosundaki elde edilen değerler ile ilgili yorumlarımızı yapıyoruz. Ardından Wilcoxon Doğruluk Test Özeti tablosunu oluşturuyoruz ve deneylerimizin sonuçlarını açıklıyoruz.

Tablo 4.1’de, deneylerimizde kullandığımız 32 UCI veri kümesi listelenmektedir. Tabloda veri kümelerinin sırasıyla veri kümesi adları, örnek sayısı, nümerik özellik sayısı, nominal özellik sayısı, sınıf sayısı ve kayıp değerlerinin yüzde değerleri bulunmaktadır. Kayıp değerler (%) sütunu, bir veri kümesinin eksik değerlere sahip girdilerinin (özellik sayısı X örnek sayısı) yüzdesini belirtmektedir [55].

Sınıflandırma doğruluğu açısından CDFW ve DFW’yi NB, CFW, KLMFW, GRFW, DTFW, DeepFW, DEFW, CLLFW ve MSEFW yöntemleri ile karşılaştırmak için deneyimizi gerçekleştirdik. Sınıflandırma doğruluğu açısından doğruluk karşılaştırma sonuçları, Tablo 4.2’de sunulmuştur. Burada sınıflandırma doğruluğu tahminleri, 10 ayrı tabakalı 10 katlı çapraz doğrulama çalışması ve özellik başına ağırlıklı filtreler aracılığıyla elde edilir. 32 veri kümesinin tamamındaki ortalamalar ve DFW ve CDFW bazında *Win / Tie / Loss* (*W/T/L*) istatistikleri tablonun alt satırlarında belirtilmiştir. Tüm veri kümelerindeki her bir özellik ağırlıklandırma yönteminin ortalaması (aritmetik ortalama), diğer istatistiklere ilave olarak performans ölçütü sağlamaktadır. DFW rakipleriyle karşılaştırıldığında, DFW’nin *W* veri kümelerinde kazandığı, *T* veri kümelerinde eşitlik kurulduğu ve *L* veri kümelerinde kaybettiği anlamına gelir ve değerler DFW *W/T/L* satırında belirtilir. CDFW rakipleriyle karşılaştırıldığında elde edilen değerler bir alt satırda bulunan CDFW *W/T/L* satırında belirtilir. Tablo 4.2’de bazı değerlerin sağına $\circ$ ve $\bullet$ sembolleri eklenmiştir. Bu semboller sırasıyla DFW ve CDFW filtrelerimizin rakiplerine kıyasla istatistiksel olarak daha düşük performans gösterdiğini ifade eder.

Tablo 4.1 Deneylerimizde Kullanılan 32 UCI Veri Kümesi

Veri Kümesi Adı	Örnek Sayısı	Nümerik Özellik Sayısı	Nominal Özellik Sayısı	Sınıf Sayısı	Kayıp Değer (%)
anneal	898	6	32	5	0.0
anneal.ORIG	898	6	32	5	63.3
audiology	226	0	69	24	2.0
autos	205	15	10	7	1.1
balance-scale	625	4	0	3	0.0
breast-cancer	286	0	9	2	0.3
breast-w	699	9	0	2	0.3
colic	368	7	15	2	23.8
colic.ORIG	368	7	20	2	18.7
credit-a	690	6	9	2	0.6
credit-g	1000	7	13	2	0.0
diabetes	768	8	0	2	0.0
Glass	214	9	0	7	0.0
heart-c	303	6	7	5	0.2
heart-h	294	6	7	5	20.4
heart-statlog	270	13	0	2	0.0
hepatitis	155	6	13	2	5.6
hypothyroid	3772	23	6	4	6.0
ionosphere	351	34	0	2	0.0
iris	150	4	0	3	0.0
letter	20000	16	0	26	0.0
lymph	148	3	15	4	0.0
mushroom	8124	0	22	2	1.4
segment	2310	19	0	7	0.0
sick	3772	7	22	2	6.0
soybean	683	0	35	19	9.8
splice	3190	0	61	3	0.0
vehicle	846	18	0	4	0.0
vote	435	0	16	2	5.6
vowel	990	10	3	11	0.0
waveform-5000	5000	40	0	3	0.0
zoo	101	1	16	7	0.0

Tablo 4.2’de CDFW sütununun sağında (q) simgesi yer almaktadır. CDFW(q), alt simgesi $q \in \mathbb{N}^+$, CDFW filtresinde K-mean kümeleme için seçilen küme sayısını belirtir. İlgili (q) değeri kadar küme sayısı uygulandığında elde edilen en yüksek performans değeri CDFW(q) sütununda yerini almıştır. Tablo 4.2’yi istatistiksel olarak incelediğimizde aşağıdaki sonuçları gözlemlemekteyiz:

- CDFW filtresinin ortalama doğruluğu 86,79 iken, DFW filtresinin ortalama doğruluğu 85,75’tir. NB filtresinin ortalama doğruluğu ise 83.95’dir. CDFW ve DFW, NB’de dahil olmak üzere ortalama değer bakımından tüm rakiplerinden daha iyi performans göstermektedir. Her ikisinin de en yakın rakipleri MSEFW ve CLLFW olarak görülmektedir. Ancak bu iki filtrenin yüksek hesaplama karmaşıklığı ve kompleks yapısı göz önünde bulundurulmalıdır.

- DFW filtresi NB ile karşılaştırıldığında, 26 adet veri kümesinde daha iyi sınıflandırma doğruluğuna sahiptir ve hiçbir veri kümesinde daha kötü bir performansa sahip olmamıştır. CDFW filtresi de NB ile karşılaştırıldığında, 27 adet veri kümesinde daha iyi sınıflandırma doğruluğuna sahipken aynı şekilde hiçbir veri kümesinde daha kötü bir performansa sahip olmamıştır. Ayrıca, $W/T/L$ istatistikleri dikkate alındığında, her ikisi de diğer rakiplerinden önemli ölçüde daha iyi performans göstermektedir.
- CDFW filtresi DFW ile karşılaştırıldığında 15 veri kümesinde, CDFW yöntemi beklendiği gibi DFW yönteminden büyük ölçüde daha iyi performans gösterir. Bu da rastgele örnek özellikleri arasındaki homojen olmayan bağımlılık yapısının sınıflandırmada ele alınması gerektiğini gösterir. 17 adet veri kümesinde ise eşit seviyede performansa sahip oldukları görülmüştür.
- DFW filtresi, NB, CFW, KLMFW, GRFW ve DeepFW ile karşılaştırıldığında sırasıyla 26, 17, 26, 27 ve 24 veri kümesinde performansı daha iyidir. Aynı filtreler ile karşılaştırıldığında hiçbir veri kümesinde performansı daha kötü değildir. DTFW, CLLFW ve MSEFW'ye göre sadece iki veri kümesinde ve DEFW'nin sadece bir veri kümesinde daha düşük performansa sahip olmuştur.
- CDFW filtresi, NB, CFW, KLMFW, GRFW, DTFW, DeepFW ve DEFW ile karşılaştırıldığında sırasıyla 27, 22, 28, 29, 21, 27 ve 22 veri kümesinde performansı daha iyidir. Aynı filtreler ile karşılaştırıldığında hiçbir veri kümesinde performansı daha kötü değildir. CLLFW ve MSEFW'ye göre sadece birer veri kümesinde daha düşük performansa sahip olmuştur.
- KLMFW ve GRFW, tüm özellik ağırlıklandırma yöntemleri içerisinde ortalamaları en düşük iki yöntem olduklarını görüyoruz. Her iki yöntem de NB'den bile daha kötü performans gösterdiğini görüyoruz. KLMFW ve GRFW tarafından sağlanan iyileştirmeler, sınıflandırma doğruluğu açısından çok sınırlı olduğu yorumunu yapabiliriz.
- Güçlü ve yüksek dereceli özellik bağımlılıklarının gözlemlendiği “letter” ve “mushroom” veri kümelerinde DFW'nin performansını gözlemledik. Bu veri kümelerinde, DFW'nin sınıflandırma doğruluklarının (sırasıyla %75.01, %99.05) NB'nin sınıflandırma doğruluklarından (sırasıyla %73.71, %95.30) belirgin şekilde daha iyi olduğunu görebiliriz.

Tablo 4.2 CDFW ve DFW için NB, CFW, KLMFW, GRFW, DTFW, DeepFW, DEFW, CLLFW ve MSEFW ile Doğruluk Karşılaştırmaları

Veri Kümesi	CDFW (q)	DFW	NB	CFW	KLMFW	GRFW	DTFW	DeepFW	DEFW	CLLFW	MSEFW
anneal	98.16 (2)	97.53	96.16	98.33	87.85	97.14	90.27	97.42	98.06	98.60	98.69
anneal.ORIG	94.70 (1)	94.72	93.10	94.42	85.77	88.74	93.59	91.66	93.72	93.29	93.69
audiology	72.31 (1)	72.31	67.37	71.68	70.82	71.62	70.11	73.83	72.75	78.08 ◦●	78.08 ◦●
autos	83.62 (2)	83.36	77.22	82.59	75.28	72.77	75.23	76.18	76.78	80.43	80.09
balance-scale	86.76 (1)	86.76	71.08	86.99	71.27	71.32	72.38	71.99	69.26	71.08	71.08
breast-cancer	71.89 (2)	71.68	72.80	72.22	70.63	72.24	72.29	71.89	71.91	71.00	71.35
breast-w	97.15 (1)	97.15	97.23	97.14	97.43	97.30	97.20	97.31	96.74	96.84	96.51
colic	83.63 (1)	83.63	80.61	83.45	83.81	81.80	84.29	83.15	82.45	84.13	83.72
colic.ORIG	73.10 (1)	73.10	71.56	69.01	70.98	72.50	75.79	74.46	74.05	73.87	73.87
credit-a	86.82 (1)	86.81	86.14	86.58	85.99	85.54	87.23	86.54	86.65	86.59	86.23
credit-g	75.20 (1)	75.20	75.46	76.17	68.28	69.78	75.49	74.28	75.16	75.61	75.59
diabetes	78.13 (1)	78.13	76.60	78.45	76.81	77.82	78.72	78.70	77.88	78.54	78.48
glass	73.72 (2)	72.97	72.52	73.14	73.01	70.99	73.74	70.84	75.89	73.92	73.82
heart-c	83.83 (2)	83.21	83.60	83.17	83.90	82.98	83.20	82.71	82.41	83.11	83.73
heart-h	83.69 (1)	83.69	84.27	83.46	81.99	82.77	83.18	83.44	81.23	84.94	84.39
heart-statlog	83.82 (2)	83.36	83.31	83.36	84.33	84.37	83.11	83.33	83.70	83.48	84.74
hepatitis	85.09 (2)	85.54	84.93	85.41	83.69	83.81	85.52	84.41	86.26	86.74	86.61
hypothyroid	99.10 (1)	99.10	98.64	98.83	96.27	97.12	99.07	98.06	99.10	99.37	99.37
ionosphere	92.54 (2)	91.14	90.50	91.17	90.80	91.48	92.22 ◦	91.20	91.34	92.85	92.73
iris	96.79 (1)	96.79	95.72	94.40	94.33	94.33	94.93	94.47	94.47	94.60	94.33
letter	80.32 (5)	75.01	73.71	75.22	74.27	74.30	73.87	74.99	74.78	75.52	75.55
lymph	84.13 (1)	84.13	84.94	84.74	80.07	80.22	79.40	82.56	84.62	84.30	84.48
mushroom	99.51 (3)	99.05	95.30	99.19	98.52	98.92	97.97	98.91	99.15	99.90	99.90
segment	95.31 (3)	94.05	91.45	93.62	90.20	91.14	93.63	93.09	94.80	95.15	95.24
sick	97.26 (1)	97.26	97.11	97.33	96.55	96.55	97.47	96.83	97.61	97.46	97.47
soybean	93.03 (1)	93.03	91.58	93.16	91.74	90.60	92.64	92.47	92.99	93.92	93.75
splice	95.68 (1)	95.68	95.36	96.19	94.50	94.00	96.03	95.99	95.83	96.05	96.28
vehicle	73.12 (5)	65.82	64.49	64.89	61.25	62.66	63.61	62.60	66.02	68.20	68.57
vote	95.91 (2)	92.61	90.16	92.11	93.01	93.01	95.40 ◦	92.62	95.10 ◦	95.56 ◦	95.52 ◦
vowel	83.49 (10)	72.61	70.12	74.21	63.25	67.47	65.15	67.42	67.05	68.22	68.19
waveform-5000	84.31 (3)	83.46	80.65	83.02	79.64	80.39	81.06	80.96	83.73	84.58	84.65
zoo	95.00 (1)	95.00	92.64	94.66	93.87	94.17	88.04	91.30	95.75	95.85	95.75
Average	86.79	85.75	83.95	85.57	82.82	83.43	84.12	84.24	84.91	85.68	85.70
DFW W/T/L	---	---	26/6/0	17/15/0	26/6/0	27/5/0	20/10/2	24/8/0	14/17/1	9/21/2	8/22/2
CDFW W/T/L	---	15/17/0	27/5/0	22/10/0	28/4/0	29/3/0	21/11/0	27/5/0	22/10/0	14/17/1	14/17/1

KLMFW, GRFW, DTFW, DeepFW, DEFW, CLLFW ve MSEFW filtrelerinin doğrulukları karşılaştırma amacıyla [19]'dan alınmıştır.

CDFW (q), alt simgesi $q \in \mathbb{N}^+$, CDFW filtresinde K-mean kümeleme için seçilen küme sayısını belirtir.

Tablo 4.3 Wilcoxon Doğruluk Test Özeti

	CDFW(<i>q</i>)	DFW	NB	CFW	KLMFW	GRFW	DTFW	DeepFW	DEFW	CLLFW	MSEFW
CDFW(<i>q</i>)	–	□	□	□	□	□	□	□	□	□	□
DFW	■	–	□	□	□	□	□	□	□	–	–
NB	■	■	–	■	□	□	■	■	■	■	■
CFW	■	■	□	–	□	□	□	□	□	■	■
KLMFW	■	■	■	■	–	■	■	■	■	■	■
GRFW	■	■	■	■	□	–	■	■	■	■	■
DTFW	■	■	□	■	□	□	–	■	■	■	■
DeepFW	■	■	□	■	□	□	□	–	■	■	■
DEFW	■	■	□	■	□	□	□	□	–	■	■
CLLFW	■	–	□	□	□	□	□	□	□	–	–
MSEFW	■	–	□	□	□	□	□	□	□	–	–

Bu bulgular, hem her bir nitelik çifti arasındaki FFS'yi hem de her bir nitelik ile sınıf niteliği arasındaki FCS'yi yakalamanın, NB özellik ağırlıklarını belirlemek için oldukça önemli olduğunu ifade etmektedir.

Tablo 4.2'de sunulan doğruluk sonuçlarını baz alarak, her bir algoritma çiftini karşılaştırmak adına Wilcoxon doğruluk testi Tablo 4.3'de oluşturulmuştur. Oluşturulan Wilcoxon doğruluk testi tablosunda *içi-dolu kare* (■), sütundaki algoritmanın ilgili satırdaki algoritmadan daha iyi başarıma sahip olduğunu, *içi-boş kare* □ ise satırdaki algoritmanın ilgili sütundaki algoritmadan daha başarılı olduğunu ifade etmektedir. "–" simgesi ise, her iki algoritmanın da performans değerlerinin birbirlerine çok yakın olduğunu, bu nedenle herhangi bir üstünlük ilişkisinin kurulmadığını ifade etmektedir. Tablo 4.2 üzerinden özellik ağırlıklandırma yöntemlerinin karşılaştırma sonuçlarını incelediğimizde aşağıdaki sonuçları gözlemlemekteyiz:

- CDFW algoritmamız 10 *içi-dolu kare* ile tüm algoritma çiftleri arasında en başarılı algoritma olduğunu görmekteyiz. CDFW'e 7 *içi-dolu kare* ile en yakın performansa sahip DFW, tüm algoritma çiftleri arasında en başarılı ikinci algoritma olduğunu görmekteyiz.
- DFW ile aynı başarı oranına sahip CLLFW ve MSEFW algoritmanın bulunduğunu görüyoruz. Ancak bu iki algoritmanın yüksek hesaplama karmaşıklığı ve kompleks yapıya sahip olduğu göz önünde bulundurulmalıdır.
- Hem CDFW hem de DFW algoritmalarımız, 2 *içi-dolu kareye* sahip NB'e göre önemli ölçüde daha yüksek performansa sahip olduklarını görüyoruz. Bu bakımdan CDFW ve DFW, NB'e göre büyük kazanç sağlamaktadır.
- GRFW sahip olduğu sadece 1 *içi-dolu kare* ile ve KLMFW'nin de hiçbir *içi-dolu kareye* sahip olmamasından dolayı performansları en düşük iki algoritma olduklarını görüyoruz. Her iki algoritmanın da NB'den bile daha kötü performans gösterdiğini görüyoruz.

5 SONUÇ VE ÖNERİLER

Bu çalışma, NB'in koşullu bağımsızlık varsayımından kaynaklanan hataları azaltmak adına NB algoritmasında geçmişten günümüze geliştirilmiş yöntemlerden biri olan özellik ağırlıklandırma yöntemi için DFW NB olarak yeni bir özellik ağırlıklandırma yöntemi sunar. Özelliklerin birbirleriyle olan bağımlılığını ve özellik sınıf özelliği arasındaki bağımlılığı ölçmek için bağımlılık metriği öneriyoruz. NB sınıflandırmasında hangi özelliklerin daha fazla belirleyici, hangilerinin daha az belirleyici olduklarını ortaya çıkarmak için FFS ve FCS kavramlarını sunuyoruz. FFS ve FCS'den özellik ağırlıklarının nasıl bulunacağını gösteriyoruz ve yeni bir DFW NB sınıflandırması öneriyoruz. Özelliklerin homojen olmayan bağımlılığı göz önüne alındığında, rastgele örneklerin kümelenmesini ve ardından koşullu bağımsızlık varsayımını azaltmak için özellik ağırlıklandırmasının kullanılmasını savunuyoruz. Rastgele alt örneklerden DFW sınıflandırma yöntemlerinin doğruluğunu birleştirerek CDFW NB sınıflandırması öneriyoruz. DFW ve CDFW filtrelerimizi kullanan NB, geleneksel NB'den daha iyi performans gösterdiğini deneylerimizde gözlemliyoruz. Aynı zamanda çalışmalarda kullanılan diğer tüm özellik ağırlıklı NB'lerden daha iyi performans gösterdiğini görüyoruz ve deney sonuçlarını detaylarıyla yorumluyoruz.

- [1] Y. Ren, L. Wang, X. Li, M. Pang J. Wei, “Stochastic optimization for bayesian network classifiers,” *Applied Intelligence*, **pages** 1–21, 2022.
- [2] G. H. John P. Langley, “Estimating continuous distributions in Bayesian classifiers,” *in Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence* 1995, **pages** 338–345.
- [3] J. Pearl, “Belief updating by network propagation,” *Proba. Reasoning in Intelligent Systems: Networks of Plausible Inference*, **pages** 143–190, 1988.
- [4] N. Friedman, D. Geiger M. Goldszmidt, “Bayesian network classifiers,” *Machine Learning*, **jourvol** 29, **number** 2, **pages** 131–163, 1997.
- [5] D. J. Hand K. Yu, “Idiot’s Bayes – Not so stupid after all?” *International Statistical Review*, **jourvol** 69, **number** 3, **pages** 385–398, 2001.
- [6] X. Wu **and others**, “Top 10 algorithms in data mining,” *Knowledge and Info. Systems*, **jourvol** 14, **number** 1, **pages** 1–37, 2008.
- [7] X. Wu V. Kumar, *The Top Ten Algorithms in Data Mining*, **jourser** Chapman & Hall/CRC Data Mining and Knowledge Discovery Series. CRC Press, 2009.
- [8] C. Qiu, L. Jiang C. Li, “Not always simple classification: Learning superparent for class probability estimation,” *Expert Systems with Applications*, **jourvol** 42, **number** 13, **pages** 5433–5440, 2015.
- [9] P. Tan, M. Steinbach, A. Karpatne V. Kumar, *Introduction to Data Mining*, Second. Pearson Education, Inc., 2019.
- [10] J. Han, J. Pei M. Kamber, *Data Mining: Concepts and Techniques*. San Francisco, CA: Elsevier, 2011.
- [11] R. Blanquero, E. Carrizosa, P. Ramirez-Cobo M. R. Sillero-Denamiel, “Variable selection for Naive Bayes classification,” *Computers & Operations Research*, **jourvol** 135, **page** 105456, 2021.
- [12] J. Hernandez-Gonzalez, I. Inza J. A. Lozano, “Learning Bayesian network classifiers from label proportions,” *Pattern Recognition*, **jourvol** 46, **number** 12, **pages** 3425–3440, 2013.
- [13] C.-H. Lee, F. Gutierrez D. Dou, “Calculating feature weights in naive Bayes with Kullback-Leibler measure,” *in Proceedings of IEEE International Conference on Data Mining* Vancouver, Canada, 2011, **pages** 1146–1151.
- [14] L. Jiang, L. Zhang, L. Yu D. Wang, “Class-specific attribute weighted naive Bayes,” *Pattern recognition*, **jourvol** 88, **pages** 321–330, 2019.
- [15] W. Xu, L. Jiang L. Yu, “An attribute value frequency-based instance weighting filter for naive Bayes,” *Journal of Experimental & Theoretical Artificial Intelligence*, **jourvol** 31, **number** 2, **pages** 225–236, 2019.

- [16] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Computer Science*, **jourvol 2, number 3, pages 1–21**, 2021.
- [17] H. Zhang, L. Jiang C. Li, "Collaboratively weighted Naive Bayes," *Knowledge and Information Systems*, **jourvol 63, number 12, pages 3159–3182**, 2021.
- [18] L. Yu, L. Jiang, D. Wang L. Zhang, "Toward naive Bayes with attribute value weighting," *Neural Computing and Applications*, **jourvol 31, number 10, pages 5699–5713**, 2019.
- [19] L. Jiang, L. Zhang, C. Li J. Wu, "A correlation-based feature weighting filter for Naive Bayes," *IEEE Transactions on Knowledge and Data Engineering*, **jourvol 31, number 2, pages 201–213**, 2018.
- [20] E. J. Keogh M. J. Pazzani, "Learning augmented Bayesian classifiers: A comparison of distribution-based and classification-based approaches," *in AIStats Citeseer*, 1999.
- [21] L. Jiang, H. Zhang Z. Cai, "A novel bayes model: Hidden naive bayes," *IEEE Transactions on knowledge and data engineering*, **jourvol 21, number 10, pages 1361–1371**, 2008.
- [22] N. Li, Y. Yu Z.-H. Zhou, "Semi-naive exploitation of one-dependence estimators," *in 2009 Ninth IEEE International Conference on Data Mining IEEE*, 2009, **pages 278–287**.
- [23] J. Wu, S. Pan, X. Zhu, P. Zhang C. Zhang, "Sode: Self-adaptive one-dependence estimators for classification," *Pattern Recognition*, **jourvol 51, pages 358–377**, 2016.
- [24] P. Langley S. Sage, "Induction of selective Bayesian classifiers," *in Uncertainty Proceedings 1994 Elsevier*, 1994, **pages 399–406**.
- [25] C. a. Ratanamahatana D. Gunopulos, "Feature selection for the naive bayesian classifier using decision trees," *Applied artificial intelligence*, **jourvol 17, number 5-6, pages 475–487**, 2003.
- [26] L. Jiang, H. Zhang, Z. Cai J. Su, "Evolutional naive bayes," *in Proceedings of the 2005 International Symposium on Intelligent Computation and its Application, ISICA 2005*, **pages 344–350**.
- [27] H. Peng, F. Long C. Ding, "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Transactions on pattern analysis and machine intelligence*, **jourvol 27, number 8, pages 1226–1238**, 2005.
- [28] L. Jiang, Z. Cai, H. Zhang D. Wang, "Not so greedy: randomly selected naive Bayes," *Expert Systems with Applications*, **jourvol 39, number 12, pages 11 022–11 028**, 2012.
- [29] B. Tang, S. Kay H. He, "Toward optimal feature selection in naive Bayes for text categorization," *IEEE transactions on knowledge and data engineering*, **jourvol 28, number 9, pages 2508–2521**, 2016.
- [30] R. Kohavi **and others**, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid.," *in Kdd volume 96*, 1996, **pages 202–207**.

- [31] Z. Xie, W. Hsu, Z. Liu M. L. Lee, "Snnb: A selective neighborhood based naive Bayes for lazy learning," *in Pacific-Asia Conference on Knowledge Discovery and Data Mining Springer*, 2002, **pages** 104–114.
- [32] E. Frank, M. Hall B. Pfahringer, *Locally weighted naive Bayes. In proceedings of the conference on uncertainty in artificial intelligence*, 2003.
- [33] C. Elkan **and others**, "Boosting and naive Bayesian learning," *in Proceedings of the international conference on knowledge discovery and data mining* 1997.
- [34] L. Jiang, Z. Cai D. Wang, "Improving naive Bayes for classification," *International Journal of Computers and Applications*, **jourvol** 32, **number** 3, **pages** 328–332, 2010.
- [35] L. Jiang, D. Wang Z. Cai, "Discriminatively weighted naive Bayes and its application in text classification," *International Journal on Artificial Intelligence Tools*, **jourvol** 21, **number** 01, **page** 1 250 007, 2012.
- [36] K. El Hindi, "Fine tuning the Naïve Bayesian learning algorithm," *AI Communications*, **jourvol** 27, **number** 2, **pages** 133–141, 2014.
- [37] A. Alhussan K. El Hindi, "Selectively fine-tuning Bayesian network learning algorithm," *International Journal of Pattern Recognition and Artificial Intelligence*, **jourvol** 30, **number** 08, **page** 1 651 005, 2016.
- [38] D. M. Diab K. M. El Hindi, "Using differential evolution for fine tuning naïve Bayesian classifiers and its application for text classification," *Applied Soft Computing*, **jourvol** 54, **pages** 183–199, 2017.
- [39] H. Zhang S. Sheng, "Learning weighted naive Bayes with accurate ranking," *in Proceedings of the 4th IEEE International Conference on Data Mining (ICDM'04)* Brighton, United Kingdom, 2004, **pages** 567–570.
- [40] M. Hall, "A decision tree-based attribute weighting filter for Naive Bayes," *in International Conference on Innovative Techniques and Applications of Artificial Intelligence Springer*, 2006, **pages** 59–70.
- [41] J. Wu Z. Cai, "Attribute weighting via differential evolution algorithm for attribute weighted naive bayes (wnb)," *Journal of Computational Information Systems*, **jourvol** 7, **number** 5, **pages** 1672–1679, 2011.
- [42] N. A. Zaidi, J. Cerquides, M. J. Carman G. I. Webb, "Alleviating Naive Bayes attribute independence assumption by attribute weighting," *Journal of Machine Learning Research*, **jourvol** 14, **pages** 1947–1988, 2013.
- [43] L. Jiang, C. Li, S. Wang L. Zhang, "Deep feature weighting for Naive Bayes and its application to text classification," *Engineering Applications of Artificial Intelligence*, **jourvol** 52, **pages** 26–39, 2016.
- [44] D. M. Chickering, "Learning Bayesian networks is NP-complete," *in Learning from Data Springer*, 1996, **pages** 121–130.
- [45] E. O. Abiodun, A. Alabdulatif, O. I. Abiodun, M. Alawida, A. Alabdulatif R. S. Alkhawaldeh, "A systematic review of emerging feature selection optimization methods for optimal text classification: the present state and prospective opportunities," *Neural Computing and Applications*, **pages** 1–28, 2021.
- [46] G. Chandrashekar F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, **jourvol** 40, **number** 1, **pages** 16–28, 2014.

- [47] *Instance Selection*, 2019.
- [48] K. El Hindi, "A noise tolerant fine tuning algorithm for the Naïve Bayesian learning algorithm," *Journal of King Saud University-Computer and Information Sciences*, **jourvol** 26, **number** 2, **pages** 237–246, 2014.
- [49] I. H. Witten E. Frank, "Data mining: practical machine learning tools and techniques with Java implementations," *Acm Sigmod Record*, **jourvol** 31, **number** 1, **pages** 76–77, 2002.
- [50] U. Fayyad K. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," 1993.
- [51] J. Ferreira, D. Denison D. Hand, "Weighted Naive Bayes modelling for data mining," *CiteSeer Scientific Literature Digital Library*, 2001.
- [52] M. A. Hall, "Correlation-based feature selection of discrete and numeric class machine learning," 2000.
- [53] J. Haigh, *Probability models*, Second. Springer Science & Business Media, 2013.
- [54] I. S. Gradshteyn I. M. Ryzhik, *Table of integrals, series, and products*, 8th. Academic press, 2014.
- [55] E. Frank, M. Hall B. Pfahringer, "Locally weighted naive bayes," *arXiv preprint arXiv:1212.2487*, 2012.

A

DFW NB SINIFLAYICININ JAVA UYGULAMASI

DFW Naive Bayes Sınıflandırıcısı için geliştirilmiş Java kodu bu bölümde verilmiştir.

A.1 DFW Naive Bayes Sınıflandırıcısının Java Gerçeklemesi

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 * CDWNB.java
 * Copyright (C) 2022 Gieliz Chatip and Ferkan Yilmaz
 */

package weka.classifiers.bayes;

import java.security.InvalidParameterException;
import java.util.HashMap;

import weka.classifiers.AbstractClassifier;
import weka.core.Attribute;
import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;

/**
 * <!-- globalinfo-start --> Constructs Dependence Weighted Naive Bayes
 * (CDWFNB).<br/>
 * <br/>
 * For more information refer to:<br/>
 * <br/>
 * G. Chatip and F. Yilmaz, "A Dependent Feature Weighting Filter for Naive Bayes
 * Classifier," in The 30th IEEE Conference on Signal Processing and
 * Communications Applications Conference (SIU 2022), Safranbolu Kastamonu,
 * Turkiye.
 */
```

```

* <p/>
* <!-- globalinfo-end -->
*
* <!-- technical-bibtex-start --> BibTeX:
*
* <pre>
* &#64;inproceedings{Chatip2022,
*   author = "Chatip, G. and Yilmaz, F.",
*   title = "A Dependent Feature Weighting Filter for Naive Bayes Classifier",
*   booktitle = "The 30th IEEE Conference on Signal Processing and Communications
*   Applications Conference (SIU 2022), Safranbolu, Kastamonu, Turkiye",
*   year = "2022"
* }
* </pre>
* <p/>
* <!-- technical-bibtex-end -->
*
* <!-- options-start --> Valid options are:
* <p/>
*
* <pre>
* -D
* If set, classifier is run in debug mode and
* may output additional info to the console
* </pre>
*
* <!-- options-end -->
*
* @author Gieliz Chatip (gieliz.chatip@ieee.org)
* @author Ferkan Yilmaz (ferkan.yilmaz@ieee.org)
* @version $Revision: 001 $
*/
public class DFWNB extends AbstractClassifier implements TechnicalInformationHandler {

    /** for serialization */
    private static final long serialVersionUID = -7758292156835559818L;

    /** The core implementation of dependent feature weighting (DFW) classifier */
    private class DFWClassifier {
        /** The number of each class value occurs in the dataset */
        private double[] m_ClassCounts;

        /** The number of each attribute value occurs in the dataset */
        private double[] m_AttCounts;

        /** The number of two attributes values occurs in the dataset */
        private double[][] m_AttAttCounts;

        /** The number of class and two attributes values occurs in the dataset */
        private double[][][] m_ClassAttAttCounts;

        /** The number of instances in the dataset */
        private int m_NumInstances;

        /** The 2D array of the mutual dependence of each pair attributes */
        private double[][] m_mutualDependence;

        /**
         * The array of pure (raw) mutual dependence between each attribute and class
         */
        private double[] m_PureWeight;

        /** The array of mutual dependence between each attribute and class */
        private double[] m_Weight;

        /**
         * Initializes the classifier
         *
         * @param numClasses The number of class labels
         * @param totalAttValues The total number of attribute values
         */
        public void initialize(int numClasses, int totalAttValues) {
            m_NumInstances = 0;

```

```

// allocate space for counts and frequencies
m_ClassCounts = new double[numClasses];
m_AttCounts = new double[totalAttValues];
m_AttAttCounts = new double[totalAttValues][totalAttValues];
m_ClassAttAttCounts = new double[numClasses][totalAttValues][totalAttValues];
}

/**
 * Calculate the attribute counts and the mutual counts of two attributes.
 *
 * @param instances Set of instances serving as training data
 */
public void calculateCounts(Instances instances) {
    for (int k = 0; k < instances.numInstances(); k++) {
        m_NumInstances++;
        // The histogram of distinct class values
        int classVal = (int) instances.instance(k).classValue();
        m_ClassCounts[classVal]++;
        int[] valIndex = new int[m_NumAttributes];
        int[] attIndex = new int[m_NumAttributes];
        for (int i = 0; i < m_NumAttributes; i++) {
            valIndex[i] = (int) instances.instance(k).value(i);
            attIndex[i] = m_StartAttIndex[i] + valIndex[i];
            m_AttCounts[attIndex[i]]++;
        }
        for (int Att1 = 0; Att1 < m_NumAttributes; Att1++) {
            for (int Att2 = 0; Att2 < m_NumAttributes; Att2++) {
                m_AttAttCounts[attIndex[Att1]][attIndex[Att2]]++;
                m_ClassAttAttCounts[classVal][attIndex[Att1]][attIndex[Att2]]++;
            }
        }
    }
}

/**
 * Calculate the weights
 */
public void calculateWeights() {
    // compute mutual information between each pair attributes (including class)
    m_mutualDependence = new double[m_NumAttributes][m_NumAttributes];
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        for (int att2 = att1 + 1; att2 < m_NumAttributes; att2++) {
            if (m_NumInstances != 0) {
                m_mutualDependence[att1][att2] = mutualDivergence(att1, att2);
                m_mutualDependence[att2][att1] = m_mutualDependence[att1][att2];
            } else {
                m_mutualDependence[att1][att2] = 0;
                m_mutualDependence[att2][att1] = 0;
            }
        }
    }

    double ave = 0;
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        ave += m_mutualDependence[att1][m_ClassIndex];
    }
    ave /= (m_NumAttributes - 1);

    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        if (ave != 0)
            m_mutualDependence[att1][m_ClassIndex] /= ave;
    }

    double mean = 0;
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        for (int att2 = 0; att2 < m_NumAttributes; att2++) {
            if (att2 == m_ClassIndex || att2 == att1)

```

```

        continue;
        mean += m_mutualDependence[att1][att2];
    }
}
mean /= ((m_NumAttributes - 1) * (m_NumAttributes - 2));
for (int att1 = 0; att1 < m_NumAttributes; att1++) {
    if (att1 == m_ClassIndex)
        continue;
    for (int att2 = 0; att2 < m_NumAttributes; att2++) {
        if (att2 == m_ClassIndex || att2 == att1)
            continue;
        if (mean != 0)
            m_mutualDependence[att1][att2] /= mean;
    }
}

double[] aveMutualDivergence = new double[m_NumAttributes];
for (int att1 = 0; att1 < m_NumAttributes; att1++) {
    if (att1 == m_ClassIndex)
        continue;
    for (int att2 = 0; att2 < m_NumAttributes; att2++) {
        if (att2 == m_ClassIndex || att2 == att1)
            continue;
        aveMutualDivergence[att1] += m_mutualDependence[att1][att2];
    }
    aveMutualDivergence[att1] /= (m_NumAttributes - 2);
}

m_PureWeight = new double[m_NumAttributes];
for (int att1 = 0; att1 < m_NumAttributes; att1++) {
    if (att1 == m_ClassIndex)
        continue;
    m_PureWeight[att1] = m_mutualDependence[att1][m_ClassIndex] -
        aveMutualDivergence[att1];
}
m_Weight = m_activation.apply(m_PureWeight);
}

/**
 * compute the mutual divergence between each pair attributes (including class)
 *
 * @param args att1, att2 are two attributes
 * @return the mutual information between each pair attributes
 */
private double mutualDivergence(int att1, int att2) {
    int attIndex1 = m_StartAttIndex[att1];
    int attIndex2 = m_StartAttIndex[att2];
    double[] PriorsAtt1 = new double[m_NumAttValues[att1]];
    double[] PriorsAtt2 = new double[m_NumAttValues[att2]];
    double[][] PriorsAtt1Att2 = new double[m_NumAttValues[att1]][m_NumAttValues[att2]];
    double eps = Math.ulp(1.0);
    double mutualDependence = eps;
    for (int i = 0; i < m_NumAttValues[att1]; i++) {
        PriorsAtt1[i] = m_AttCounts[attIndex1 + i] / m_NumInstances;
    }

    for (int j = 0; j < m_NumAttValues[att2]; j++) {
        PriorsAtt2[j] = m_AttCounts[attIndex2 + j] / m_NumInstances;
    }

    for (int i = 0; i < m_NumAttValues[att1]; i++) {
        for (int j = 0; j < m_NumAttValues[att2]; j++) {
            PriorsAtt1Att2[i][j] = m_AttAttCounts[attIndex1 + i][attIndex2 + j] /
                m_NumInstances;
        }
    }

    for (int i = 0; i < m_NumAttValues[att1]; i++) {
        for (int j = 0; j < m_NumAttValues[att2]; j++) {
            double dependence = 1.0 / (1.0 - Math.pow(PriorsAtt1[i] * PriorsAtt2[j], 2.0) + eps)
                + 1.0 / (1.0 - Math.pow(PriorsAtt1Att2[i][j], 2.0) + eps)
                - 2.0 / (1.0 - PriorsAtt1[i] * PriorsAtt1Att2[i][j] * PriorsAtt2[j] + eps);

```

```

        if (dependence < 0) {
            if (Math.abs(dependence) > 1.0e-6) {
                System.err.println("Inconsistent dependence value!");
            }
            dependence = Math.abs(dependence);
        }
        dependence *= PriorsAtt1Att2[i][j];
        mutualDependence += dependence;
    }
}

return mutualDependence;
}

/**
 * Calculates the class membership probabilities for the given test instance
 *
 * @param instance the instance to be classified
 * @return predicted class probability distribution
 * @exception Exception if there is a problem generating the prediction
 */
public double[] distributionForInstance(Instance instance) throws Exception {
    // Definition of local variables
    double[] probs = new double[m_NumClasses];
    // store instance's att values in an int array
    int[] attIndex = new int[m_NumAttributes];
    for (int att = 0; att < m_NumAttributes; att++) {
        attIndex[att] = m_StartAttIndex[att] + (int) instance.value(att);
    }

    // calculate probabilities for each possible class value
    for (int classVal = 0; classVal < m_NumClasses; classVal++) {
        probs[classVal] = (m_ClassCounts[classVal] + 1.0 / m_NumClasses) / (m_NumInstances + 1.0);
        for (int att = 0; att < m_NumAttributes; att++) {
            if (att == m_ClassIndex)
                continue;
            double probGivenClass = 0;
            probGivenClass += m_ClassAttAttCounts[classVal][attIndex[att]][attIndex[att]];
            probGivenClass += 1.0 / m_NumAttValues[att];
            probGivenClass /= m_ClassCounts[classVal] + 1.0;
            double prob = Math.pow(probGivenClass, m_Weight[att]);
            probs[classVal] *= prob;
        }
    }

    return probs;
}

/**
 * Classifies the given test instance. The instance has to belong to a dataset
 * when it's being classified. Note that a classifier MUST implement either this
 * or distributionForInstance().
 *
 * @param instance the instance to be classified
 * @return the predicted most likely class for the instance or
 *         Utils.missingValue() if no prediction is made
 * @exception Exception if an error occurred during the prediction
 */
public double classifyInstance(Instance instance) throws Exception {
    double[] dist = distributionForInstance(instance);
    Utils.normalize(dist);
    if (dist == null) {
        throw new Exception("Null distribution predicted");
    }
    switch (instance.classAttribute().type()) {
        case Attribute.NOMINAL:
            double max = 0;
            int maxIndex = 0;

            for (int i = 0; i < dist.length; i++) {
                if (dist[i] > max) {
                    maxIndex = i;
                }
            }

```

```

        max = dist[i];
    }
}
if (max > 0) {
    return maxIndex;
} else {
    return Utils.missingValue();
}
}
case Attribute.NUMERIC:
case Attribute.DATE:
    return dist[0];
default:
    return Utils.missingValue();
}
}
}

private DFWClassifier m_Classifier;

/** The number of values for each attribute in the dataset */
private int[] m_NumAttValues;

/** The number of values for all attributes in the dataset */
private int m_TotalAttValues;

/** The number of classes in the dataset */
private int m_NumClasses;

/** The number of attributes including class in the dataset */
private int m_NumAttributes;

/** The index of the class attribute in the dataset */
private int m_ClassIndex;

/** The starting index of each attribute in the dataset */
private int[] m_StartAttIndex;

/** The number of distinct values for each attribute in the dataset */
private int[] m_AttSize;

private interface Function {

    double apply(double alpha) throws InvalidParameterException;

    double[] apply(double[] alpha) throws InvalidParameterException;
}

/** Activation enumeration class */
public static enum Activation implements Function {
    ACTIVATION_NONE(Activation.NONE), ACTIVATION_SIGMOID(Activation.SIGMOID),
    ACTIVATION_TANH(Activation.TANH);

    public final static int NONE = 0;
    public final static int SIGMOID = 1;
    public final static int TANH = 2;

    /**
     * Comparator to be used to compare the activation functions by their indexing
     */
    public static class Comparator implements java.util.Comparator<Activation> {
        public int compare(Activation lhsObj, Activation rhsObj) {
            return Integer.compare(lhsObj.value(), rhsObj.value());
        }
    }
}

private int m_value;

/** Constructor */
private Activation(int value) {
    m_value = value;
}

/** The indexing value of activation function */

```

```

public int value() {
    return m_value;
}

/**
 * Apply the activation function on input value.
 *
 * @param the double value to be applied the activation
 * @return the double value resulted from that the activation is applied on the
 *         input.
 */
@Override
public double apply(double in) throws InvalidParameterException {
    switch (this) {
        case ACTIVATION_NONE:
            return in;
        case ACTIVATION_SIGMOID:
            return 1.0 / (1.0 + Math.exp(-in));
        case ACTIVATION_TANH:
            return 0.5 * (1.0 + Math.tanh(in));
        default:
            throw new InvalidParameterException("Unknown activation type.");
    }
}

/**
 * Apply the activation function on all elements of input array.
 *
 * @param the double array vector to be applied the activation
 * @return the double array resulted from that the activation is applied on all
 *         elements of the input array.
 */
@Override
public double[] apply(double[] in) throws InvalidParameterException {
    int len = in.length;
    double[] out = new double[len];
    for (int i = 0; i < len; i++) {
        out[i] = apply(in[i]);
    }
    return out;
}

@Override
public String toString() {
    switch (this) {
        case ACTIVATION_NONE:
            return "none";
        case ACTIVATION_SIGMOID:
            return "sigmoid";
        case ACTIVATION_TANH:
            return "tanh";
        default:
            throw new java.lang.RuntimeException("Unknown activation type!");
    }
}

/** Activation function has been chosen as sigmoid function. */
private Activation m_activation = Activation.ACTIVATION_SIGMOID;

/**
 * Returns a string describing this classifier.
 *
 * @return a description of the data generator suitable for displaying in the
 *         explorer/experimenter gui
 */
public String globalInfo() {

    return "Constructs Correlation-based Feature Weighted Naive Bayes (CFWNB).\n\n"
        + "For more information refer to:\n\n" + getTechnicalInformation().toString();
}

/**

```

```

* Returns an instance of a TechnicalInformation object, containing detailed
* information about the technical background of this class, e.g., paper
* reference or book this class is based on.
*
* @return the technical information about this class
*/
public TechnicalInformation getTechnicalInformation() {
    TechnicalInformation result;
    result = new TechnicalInformation(Type.ARTICLE);
    result.setValue(Field.AUTHOR, "Chatip, G. and Yilmaz, F.");
    result.setValue(Field.TITLE, "A Dependent Feature Weighting Filter for Naive Bayes "
        + "Classifier");
    result.setValue(Field.BOOKTITLE,
        "The 30th IEEE Conference on Signal Processing and Communications Applications "
        + "Conference (SIU 2022)");
    result.setValue(Field.MONTH, "May");
    result.setValue(Field.YEAR, "2022");
    result.setValue(Field.ADDRESS, "Safranbolu Kastamonu, Turkiye");
    result.setValue(Field.ABSTRACT, "Naive Bayes (NB) classification is one of the most "
        + "extensively used algorithms in data mining and machine "
        + "learning due to its high efficiency and structural "
        + "simplicity based on conditional independence of attributes. "
        + "In this paper, we present a dependence metric to quantify the "
        + "dependence among attributes and class attributes and propose "
        + "feature-feature significance (FFS) and feature-class "
        + "significance (FCS) to discover highly predictive attributes "
        + "over less predictive ones in NB classification. We show how "
        + "to get feature weights from FFS and FCS and propose a novel "
        + "dependent feature weighted (DFW) NB classification. To increase "
        + "performance further, we recommend clustering the random "
        + "sample of interest due to the non-homogeneous dependence "
        + "nature of features, and then using feature weighting to "
        + "alleviate the conditional independence. As a consequence, "
        + "we propose a cluster-based DFW (CDFW) NB as a result of "
        + "weighting the DFW filters of random sub-samples by their accuracy "
        + "and then merging them for performance augmentation. The "
        + "experimental results show that the NB with DFW filter provides "
        + "good results when compared to the conventional NB and all other " + "feature
        weighting techniques.");
    result.setValue(Field.KEYWORDS,
        "Feature weighting, naive Bayes classification, cluster-based dependence, mutual
        dependence.");
    result.setValue(Field.PUBLISHER, "IEEE");
    return result;
}

/**
 * Returns default capabilities of the classifier.
 *
 * @return the capabilities of this classifier
 */
public Capabilities getCapabilities() {
    Capabilities result = super.getCapabilities();
    result.disableAll();

    // attributes
    result.enable(Capability.NOMINAL_ATTRIBUTES);

    // class
    result.enable(Capability.NOMINAL_CLASS);
    result.enable(Capability.MISSING_CLASS_VALUES);

    return result;
}

/** Table for transition from class key top class name */
HashMap<Double, String> m_value2unique = new HashMap<Double, String>();

/** Table for transition from class name top class key */
HashMap<String, Double> m_unique2Value = new HashMap<String, Double>();

private void determineUniqueNames(Instances instances) {
    for (Instance current : instances) {

```

```

        double key = current.value(instances.classIndex());
        if (!Utils.isMissingValue(key)) {
            if (!m_value2unique.containsKey(key)) {
                // String unique_name = current.toString(instances.classIndex());
                String unique_name = current.stringValue(instances.classIndex());
                m_value2unique.put(key, unique_name);
                m_unique2Value.put(unique_name, key);
            }
        }
    }
}

/** From class key to class name */
public String toClassName(double classKey) {
    return m_value2unique.get(classKey);
}

/** From class name to class key */
public double toClassKey(String uniqueName) {
    return m_unique2Value.get(uniqueName);
}

/**
 * Generates the classifier.
 *
 * @param instances set of instances serving as training data
 * @exception Exception if the classifier has not been generated successfully
 */
public void buildClassifier(Instances instances) throws Exception {

    // can classifier handle the data?
    getCapabilities().testWithFail(instances);

    // remove instances with missing class
    instances = new Instances(instances);
    instances.deleteWithMissingClass();

    // Determine unique names of classes
    determineUniqueNames(instances);

    // reset variable
    m_NumClasses = instances.numClasses();
    m_ClassIndex = instances.classIndex();
    m_NumAttributes = instances.numAttributes();
    // m_Props.m_NumInstances = instances.numInstances();
    m_TotalAttValues = 0;

    // allocate space for attribute reference arrays
    m_StartAttIndex = new int[m_NumAttributes];
    m_NumAttValues = new int[m_NumAttributes];

    // allocate space for attribute reference arrays
    m_AttSize = new int[m_NumAttributes];

    // set the starting index of each attribute and the number of values for
    // each attribute and the total number of values for all attributes (not
    // including class).
    for (int i = 0; i < m_NumAttributes; i++) {
        m_AttSize[i] = instances.attribute(i).numValues();
    }

    // set the starting index of each attribute and the number of values for
    // each attribute and the total number of values for all attributes (not
    // including class).
    for (int i = 0; i < m_NumAttributes; i++) {
        m_StartAttIndex[i] = m_TotalAttValues;
        m_NumAttValues[i] = instances.attribute(i).numValues();
        m_TotalAttValues += m_NumAttValues[i];
    }

    m_Classifier = new DFWClassifier();
    // Allocate space for counts and frequencies

```

```

        m_Classifier.initialize(m_NumClasses, m_TotalAttValues);
        // Calculate the counts
        m_Classifier.calculateCounts(instances);
        // Calculate the weights
        m_Classifier.calculateWeights();
    }

    /** Auxiliary min function */
    public static double min(double... array) {
        if (array == null || array.length == 0) {
            return Double.NaN;
        }
        if (array.length == 1) {
            return array[0];
        }
        if (array.length == 2) {
            return Math.min(array[0], array[1]);
        }
        double min = array[0];
        for (int i = 1; i < array.length; i++) {
            min = Math.min(min, array[i]);
        }
        return min;
    }

    /**
     * Calculates the class membership probabilities for the given test instance
     *
     * @param instance the instance to be classified
     * @return predicted class probability distribution
     * @exception Exception if there is a problem generating the prediction
     */
    public double[] distributionForInstance(Instance instance) throws Exception {
        double[] probs = m_Classifier.distributionForInstance(instance);
        Utils.normalize(probs);
        return probs;
    }

    /**
     * Classifies the given test instance. The instance has to belong to a dataset
     * when it's being classified. Note that a classifier MUST implement either this
     * or distributionForInstance().
     *
     * @param instance the instance to be classified
     * @return the predicted most likely class for the instance or
     *         Utils.missingValue() if no prediction is made
     * @exception Exception if an error occurred during the prediction
     */
    public double classifyInstance(Instance instance) throws Exception {
        return m_Classifier.classifyInstance(instance);
    }

    /**
     * returns a string representation of the classifier
     *
     * @return a representation of the classifier
     */
    public String toString() {
        return "CDFW Naive Bayes (Cluster Based Dependent Feature Weighted Naive Bayes)";
    }

    /**
     * Returns the revision string.
     *
     * @return the revision
     */
    public String getRevision() {
        return RevisionUtils.extract("$Revision: 1 $");
    }

    /**
     * Main method for testing this class.

```

```
*  
* @param args the options  
*/  
public static void main(String[] args) {  
    runClassifier(new DFwNB(), args);  
}  
}
```



B

CDFW NB SINIFLAYICININ JAVA UYGULAMASI

CDFW Naive Bayes Sınıflandırıcısı için geliştirilmiş Java kodu bu bölümde verilmiştir. Ayrıca, Bölüm B.2’de, WEKA UCI 32 Veri Kümesi için DFW Naive Bayes Test Kodu yer almaktadır.

B.1 CDFW Naive Bayes Sınıflandırıcısının Java Gerçeklemesi

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 * CDWNB.java
 * Copyright (C) 2022 Gieliz Chatip and Ferkan Yilmaz
 */

package weka.classifiers.bayes;

import java.security.InvalidParameterException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;

import weka.classifiers.AbstractClassifier;
import weka.clusterers.SimpleKMeans;
import weka.core.Attribute;
import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;

/**
 * <!-- globalinfo-start --> Constructs Cluster-based Dependence Weighted Naive
 * Bayes (CDWFNB).<br/>
 * <br/>
 */
```

```

* For more information refer to:<br/>
* <br/>
* G. Chatip and F. Yilmaz, "A Dependent Feature Weighting Filter for Naive Bayes
* Classifier," in The 30th IEEE Conference on Signal Processing and
* Communications Applications Conference (SIU 2022), Safranbolu Kastamonu,
* Turkiye.
* <p/>
* <!-- globalinfo-end -->
*
* <!-- technical-bibtex-start --> BibTeX:
*
* <pre>
* @inproceedings{Chatip2022,
*   author = "Chatip, G. and Yilmaz, F.",
*   title = "A Dependent Feature Weighting Filter for Naive Bayes Classifier",
*   booktitle = "The 30th IEEE Conference on Signal Processing and Communications
*   Applications Conference (SIU 2022), Safranbolu, Kastamonu, Turkiye",
*   year = "2022"
* }
* </pre>
* <p/>
* <!-- technical-bibtex-end -->
*
* <!-- options-start --> Valid options are:
* <p/>
*
* <pre>
* -D
* If set, classifier is run in debug mode and
* may output additional info to the console
* </pre>
*
* <!-- options-end -->
*
* @author Gieliz Chatip (gieliz.chatip@ieee.org)
* @author Ferkan Yilmaz (ferkan.yilmaz@ieee.org)
* @version $Revision: 001 $
*/
public class CDFWNB extends AbstractClassifier implements TechnicalInformationHandler {

    /** for serialization */
    private static final long serialVersionUID = -4374476079223154131L;

    /** The core implementation of dependent feature weighting (DFW) classifier */
    private class Classifier {
        /** The number of each class value occurs in the dataset */
        private double[] m_ClassCounts;

        /** The number of each attribute value occurs in the dataset */
        private double[] m_AttCounts;

        /** The number of two attributes values occurs in the dataset */
        private double[][] m_AttAttCounts;

        /** The number of class and two attributes values occurs in the dataset */
        private double[][][] m_ClassAttAttCounts;

        /** The number of instances in the dataset */
        private int m_NumInstances;

        /** The 2D array of the mutual dependence of each pair attributes */
        private double[][] m_mutualDependence;

        /**
         * The array of pure (raw) mutual dependence between each attribute and class
         */
        private double[] m_PureWeight;

        /** The array of mutual dependence between each attribute and class */
        private double[] m_Weight;

        /**
         * The number of correctly-classified instances of the cluster in training mode.

```

```

    */
    public int m_NumCorrectlyClassified;

    /**
     * Initializes the classifier
     *
     * @param numClasses The number of class labels
     * @param totalAttValues The total number of attribute values
     */
    public void initialize(int numClasses, int totalAttValues) {
        m_NumInstances = 0;
        // allocate space for counts and frequencies
        m_ClassCounts = new double[numClasses];
        m_AttCounts = new double[totalAttValues];
        m_AttAttCounts = new double[totalAttValues][totalAttValues];
        m_ClassAttAttCounts = new double[numClasses][totalAttValues][totalAttValues];
        m_NumCorrectlyClassified = 0;
    }

    /**
     * Calculate the attribute counts and the mutual counts of two attributes.
     *
     * @param instances Set of instances serving as training data
     */
    public void calculateCounts(Instances instances) {
        for (int k = 0; k < instances.numInstances(); k++) {
            m_NumInstances++;
            // The histogram of distinct class values
            int classVal = (int) instances.instance(k).classValue();
            m_ClassCounts[classVal]++;
            int[] valIndex = new int[m_NumAttributes];
            int[] attIndex = new int[m_NumAttributes];
            for (int i = 0; i < m_NumAttributes; i++) {
                valIndex[i] = (int) instances.instance(k).value(i);
                attIndex[i] = m_StartAttIndex[i] + valIndex[i];
                m_AttCounts[attIndex[i]]++;
            }
            for (int Att1 = 0; Att1 < m_NumAttributes; Att1++) {
                for (int Att2 = 0; Att2 < m_NumAttributes; Att2++) {
                    m_AttAttCounts[attIndex[Att1]][attIndex[Att2]]++;
                    m_ClassAttAttCounts[classVal][attIndex[Att1]][attIndex[Att2]]++;
                }
            }
        }
    }

    /**
     * Calculate the weights
     */
    public void calculateWeights() {
        // compute mutual information between each pair attributes (including class)
        m_mutualDependence = new double[m_NumAttributes][m_NumAttributes];
        for (int att1 = 0; att1 < m_NumAttributes; att1++) {
            for (int att2 = att1 + 1; att2 < m_NumAttributes; att2++) {
                if (m_NumInstances != 0) {
                    m_mutualDependence[att1][att2] = mutualDivergence(att1, att2);
                    m_mutualDependence[att2][att1] = m_mutualDependence[att1][att2];
                } else {
                    m_mutualDependence[att1][att2] = 0;
                    m_mutualDependence[att2][att1] = 0;
                }
            }
        }
    }

    double ave = 0;
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        ave += m_mutualDependence[att1][m_ClassIndex];
    }
    ave /= (m_NumAttributes - 1);

    for (int att1 = 0; att1 < m_NumAttributes; att1++) {

```

```

        if (att1 == m_ClassIndex)
            continue;
        if (ave != 0)
            m_mutualDependence[att1][m_ClassIndex] /= ave;
    }

    double mean = 0;
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        for (int att2 = 0; att2 < m_NumAttributes; att2++) {
            if (att2 == m_ClassIndex || att2 == att1)
                continue;
            mean += m_mutualDependence[att1][att2];
        }
    }
    mean /= ((m_NumAttributes - 1) * (m_NumAttributes - 2));
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        for (int att2 = 0; att2 < m_NumAttributes; att2++) {
            if (att2 == m_ClassIndex || att2 == att1)
                continue;
            if (mean != 0)
                m_mutualDependence[att1][att2] /= mean;
        }
    }

    double[] aveMutualDivergence = new double[m_NumAttributes];
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        for (int att2 = 0; att2 < m_NumAttributes; att2++) {
            if (att2 == m_ClassIndex || att2 == att1)
                continue;
            aveMutualDivergence[att1] += m_mutualDependence[att1][att2];
        }
        aveMutualDivergence[att1] /= (m_NumAttributes - 2);
    }

    m_PureWeight = new double[m_NumAttributes];
    for (int att1 = 0; att1 < m_NumAttributes; att1++) {
        if (att1 == m_ClassIndex)
            continue;
        m_PureWeight[att1] = m_mutualDependence[att1][m_ClassIndex] -
            aveMutualDivergence[att1];
    }
    m_Weight = m_activation.apply(m_PureWeight);
}

/**
 * compute the mutual divergence between each pair attributes (including class)
 *
 * @param args att1, att2 are two attributes
 * @return the mutual information between each pair attributes
 */
private double mutualDivergence(int att1, int att2) {
    int attIndex1 = m_StartAttIndex[att1];
    int attIndex2 = m_StartAttIndex[att2];
    double[] PriorsAtt1 = new double[m_NumAttValues[att1]];
    double[] PriorsAtt2 = new double[m_NumAttValues[att2]];
    double[][] PriorsAtt1Att2 = new double[m_NumAttValues[att1]][m_NumAttValues[att2]];
    double eps = Math.ulp(1.0);
    double mutualDependence = eps;
    for (int i = 0; i < m_NumAttValues[att1]; i++) {
        PriorsAtt1[i] = m_AttCounts[attIndex1 + i] / m_NumInstances;
    }

    for (int j = 0; j < m_NumAttValues[att2]; j++) {
        PriorsAtt2[j] = m_AttCounts[attIndex2 + j] / m_NumInstances;
    }

    for (int i = 0; i < m_NumAttValues[att1]; i++) {

```

```

        for (int j = 0; j < m_NumAttValues[att2]; j++) {
            PriorsAtt1Att2[i][j] = m_AttAttCounts[attIndex1 + i][attIndex2 + j] /
                m_NumInstances;
        }
    }

    for (int i = 0; i < m_NumAttValues[att1]; i++) {
        for (int j = 0; j < m_NumAttValues[att2]; j++) {
            double dependence = 1.0 / (1.0 - Math.pow(PriorsAtt1[i] * PriorsAtt2[j], 2.0) + eps)
                + 1.0 / (1.0 - Math.pow(PriorsAtt1Att2[i][j], 2.0) + eps)
                - 2.0 / (1.0 - PriorsAtt1[i] * PriorsAtt1Att2[i][j] * PriorsAtt2[j] + eps);

            if (dependence < 0) {
                if (Math.abs(dependence) > 1.0e-6) {
                    System.err.println("Inconsistent dependence value!");
                }
                dependence = Math.abs(-dependence);
            }

            double n = 1.0;
            m_MaxDependency = Math.max(m_MaxDependency,
                (Math.pow(PriorsAtt1Att2[i][j], n) - Math.pow(PriorsAtt1[i] * PriorsAtt2[j], n)
                    + eps));
            m_MinDependency = Math.min(m_MinDependency,
                (Math.pow(PriorsAtt1Att2[i][j], n) - Math.pow(PriorsAtt1[i] * PriorsAtt2[j], n)
                    + eps));

            dependence *= PriorsAtt1Att2[i][j];

            mutualDependence += dependence + eps;
        }
    }

    return mutualDependence;
}

/**
 * Calculates the class membership probabilities for the given test instance
 *
 * @param instance the instance to be classified
 * @return predicted class probability distribution
 * @exception Exception if there is a problem generating the prediction
 */
public double[] distributionForInstance(Instance instance) throws Exception {
    // Definition of local variables
    double[] probs = new double[m_NumClasses];
    // store instance's att values in an int array
    int[] attIndex = new int[m_NumAttributes];
    for (int att = 0; att < m_NumAttributes; att++) {
        attIndex[att] = m_StartAttIndex[att] + (int) instance.value(att);
    }

    // calculate probabilities for each possible class value
    for (int classVal = 0; classVal < m_NumClasses; classVal++) {
        probs[classVal] = (m_ClassCounts[classVal] + 1.0 / m_NumClasses) / (m_NumInstances +
            1.0);
        for (int att = 0; att < m_NumAttributes; att++) {
            if (att == m_ClassIndex)
                continue;
            double probGivenClass = 0;
            probGivenClass += m_ClassAttAttCounts[classVal][attIndex[att]][attIndex[att]];
            probGivenClass += 1.0 / m_NumAttValues[att];
            probGivenClass /= m_ClassCounts[classVal] + 1.0;
            double prob = Math.pow(probGivenClass, m_Weight[att]);
            probs[classVal] *= prob;
        }
    }

    return probs;
}

/**
 * Classifies the given test instance. The instance has to belong to a dataset

```

```

    * when it's being classified. Note that a classifier MUST implement either this
    * or distributionForInstance().
    *
    * @param instance the instance to be classified
    * @return the predicted most likely class for the instance or
    *         Utils.missingValue() if no prediction is made
    * @exception Exception if an error occurred during the prediction
    */
public double classifyInstance(Instance instance) throws Exception {
    double[] dist = distributionForInstance(instance);
    Utils.normalize(dist);
    if (dist == null) {
        throw new Exception("Null distribution predicted");
    }
    switch (instance.classAttribute().type()) {
    case Attribute.NOMINAL:
        double max = 0;
        int maxIndex = 0;

        for (int i = 0; i < dist.length; i++) {
            if (dist[i] > max) {
                maxIndex = i;
                max = dist[i];
            }
        }
        if (max > 0) {
            return maxIndex;
        } else {
            return Utils.missingValue();
        }
    case Attribute.NUMERIC:
    case Attribute.DATE:
        return dist[0];
    default:
        return Utils.missingValue();
    }
}

/** Number of cluster to be used */
private int m_NumCluster = 1;

private Classifier[] m_Classifiers;

/** The number of values for each attribute in the dataset */
private int[] m_NumAttValues;

/** The number of values for all attributes in the dataset */
private int m_TotalAttValues;

/** The number of classes in the dataset */
private int m_NumClasses;

/** The number of attributes including class in the dataset */
private int m_NumAttributes;

/** The index of the class attribute in the dataset */
private int m_ClassIndex;

/** The starting index of each attribute in the dataset */
private int[] m_StartAttIndex;

/** The number of distinct values for each attribute in the dataset */
private int[] m_AttSize;

private interface Function {

    double apply(double alpha) throws InvalidParameterException;

    double[] apply(double[] alpha) throws InvalidParameterException;
}

/** Activation enumeration class */

```

```

public static enum Activation implements Function {
    ACTIVATION_NONE(Activation.NONE), ACTIVATION_SIGMOID(Activation.SIGMOID),
    ACTIVATION_TANH(Activation.TANH);

    public final static int NONE = 0;
    public final static int SIGMOID = 1;
    public final static int TANH = 2;

    /**
     * Comparator to be used to compare the activation functions by their indexing
     */
    public static class Comparator implements java.util.Comparator<Activation> {
        public int compare(Activation lhsObj, Activation rhsObj) {
            return Integer.compare(lhsObj.value(), rhsObj.value());
        }
    }

    private int m_value;

    /** Constructor */
    private Activation(int value) {
        m_value = value;
    }

    /** The indexing value of activation function */
    public int value() {
        return m_value;
    }

    /**
     * Apply the activation function on input value.
     *
     * @param the double value to be applied the activation
     * @return the double value resulted from that the activation is applied on the
     *         input.
     */
    @Override
    public double apply(double in) throws InvalidParameterException {
        switch (this) {
            case ACTIVATION_NONE:
                return in;
            case ACTIVATION_SIGMOID:
                return 1.0 / (1.0 + Math.exp(-in));
            case ACTIVATION_TANH:
                return 0.5 * (1.0 + Math.tanh(in));
            default:
                throw new InvalidParameterException("Unknown activation type.");
        }
    }

    /**
     * Apply the activation function on all elements of input array.
     *
     * @param the double array vector to be applied the activation
     * @return the double array resulted from that the activation is applied on all
     *         elements of the input array.
     */
    @Override
    public double[] apply(double[] in) throws InvalidParameterException {
        int len = in.length;
        double[] out = new double[len];
        for (int i = 0; i < len; i++) {
            out[i] = apply(in[i]);
        }
        return out;
    }

    @Override
    public String toString() {
        switch (this) {
            case ACTIVATION_NONE:
                return "none";
            case ACTIVATION_SIGMOID:

```

```

        return "sigmoid";
    case ACTIVATION_TANH:
        return "tanh";
    default:
        throw new java.lang.RuntimeException("Unknown activation type!");
    }
}

/** Activation function has been chosen as sigmoid function. */
private Activation m_activation = Activation.ACTIVATION_SIGMOID;

public static double m_MaxDependency = Double.NEGATIVE_INFINITY;
public static double m_MinDependency = Double.POSITIVE_INFINITY;

/**
 * Returns a string describing this classifier.
 *
 * @return a description of the data generator suitable for displaying in the
 *         explorer/experimenter gui
 */
public String globalInfo() {
    return "Constructs Correlation-based Feature Weighted Naive Bayes (CFWNB).\n\n"
        + "For more information refer to:\n\n" + getTechnicalInformation().toString();
}

/**
 * Returns an instance of a TechnicalInformation object, containing detailed
 * information about the technical background of this class, e.g., paper
 * reference or book this class is based on.
 *
 * @return the technical information about this class
 */
public TechnicalInformation getTechnicalInformation() {
    TechnicalInformation result;

    result = new TechnicalInformation(Type.ARTICLE);
    result.setValue(Field.AUTHOR, "Chatip, G. and Yilmaz, F.");
    result.setValue(Field.TITLE, "A Dependent Feature Weighting Filter for Naive Bayes "
        + "Classifier");
    result.setValue(Field.BOOKTITLE,
        "The 30th IEEE Conference on Signal Processing and Communications Applications "
        + "Conference (SIU 2022)");
    result.setValue(Field.MONTH, "May");
    result.setValue(Field.YEAR, "2022");
    result.setValue(Field.ADDRESS, "Safranbolu Kastamonu, Turkiye");
    result.setValue(Field.ABSTRACT, "Naive Bayes (NB) classification is one of the most "
        + "extensively used algorithms in data mining and machine "
        + "learning due to its high efficiency and structural "
        + "simplicity based on conditional independence of attributes. "
        + "In this paper, we present a dependence metric to quantify the "
        + "dependence among attributes and class attributes and propose "
        + "feature-feature significance (FFS) and feature-class "
        + "significance (FCS) to discover highly predictive attributes "
        + "over less predictive ones in NB classification. We show how "
        + "to get feature weights from FFS and FCS and propose a novel "
        + "dependent feature weighted (DFW) NB classification. To increase "
        + "performance further, we recommend clustering the random "
        + "sample of interest due to the non-homogeneous dependence "
        + "nature of features, and then using feature weighting to "
        + "alleviate the conditional independence. As a consequence, "
        + "we propose a cluster-based DFW (CDFW) NB as a result of "
        + "weighting the DFW filters of random sub-samples by their accuracy "
        + "and then merging them for performance augmentation. The "
        + "experimental results show that the NB with DFW filter provides "
        + "good results when compared to the conventional NB and all other "
        + "feature weighting techniques.");
    result.setValue(Field.KEYWORDS,
        "Feature weighting, naive Bayes classification, cluster-based dependence, mutual "
        + "dependence.");
    result.setValue(Field.PUBLISHER, "IEEE");
    return result;
}

```

```

/**
 * Returns default capabilities of the classifier.
 *
 * @return the capabilities of this classifier
 */
public Capabilities getCapabilities() {
    Capabilities result = super.getCapabilities();
    result.disableAll();

    // attributes
    result.enable(Capability.NOMINAL_ATTRIBUTES);

    // class
    result.enable(Capability.NOMINAL_CLASS);
    result.enable(Capability.MISSING_CLASS_VALUES);

    return result;
}

/** Table for transition from class key top class name */
HashMap<Double, String> m_value2unique = new HashMap<Double, String>();

/** Table for transition from class name top class key */
HashMap<String, Double> m_unique2Value = new HashMap<String, Double>();

private void determineUniqueNames(Instances instances) {
    for (Instance current : instances) {
        double key = current.value(instances.classIndex());
        if (!Utils.isMissingValue(key)) {
            if (!m_value2unique.containsKey(key)) {
                // String unique_name = current.toString(instances.classIndex());
                String unique_name = current.stringValue(instances.classIndex());
                m_value2unique.put(key, unique_name);
                m_unique2Value.put(unique_name, key);
            }
        }
    }
}

/** From class key to class name */
public String toClassName(double classKey) {
    return m_value2unique.get(classKey);
}

/** From class name to class key */
public double toClassKey(String uniqueName) {
    return m_unique2Value.get(uniqueName);
}

/**
 * Generates the classifier.
 *
 * @param instances set of instances serving as training data
 * @exception Exception if the classifier has not been generated successfully
 */
public void buildClassifier(Instances instances) throws Exception {

    // can classifier handle the data?
    getCapabilities().testWithFail(instances);

    // remove instances with missing class
    instances = new Instances(instances);
    instances.deleteWithMissingClass();

    // Determine unique names of classes
    determineUniqueNames(instances);

    // reset variable
    m_NumClasses = instances.numClasses();
    m_ClassIndex = instances.classIndex();
    m_NumAttributes = instances.numAttributes();
    // m_Props.m_NumInstances = instances.numInstances();

```

```

m_TotalAttValues = 0;

// allocate space for attribute reference arrays
m_StartAttIndex = new int[m_NumAttributes];
m_NumAttValues = new int[m_NumAttributes];

// allocate space for attribute reference arrays
m_AttSize = new int[m_NumAttributes];

// set the starting index of each attribute and the number of values for
// each attribute and the total number of values for all attributes (not
// including class).
for (int i = 0; i < m_NumAttributes; i++) {
    m_AttSize[i] = instances.attribute(i).numValues();
}

// set the starting index of each attribute and the number of values for
// each attribute and the total number of values for all attributes (not
// including class).
for (int i = 0; i < m_NumAttributes; i++) {
    m_StartAttIndex[i] = m_TotalAttValues;
    m_NumAttValues[i] = instances.attribute(i).numValues();
    m_TotalAttValues += m_NumAttValues[i];
}

Instances tmpInstances = new Instances(instances, 0);
for (int i = 0; i < 2 * m_NumCluster; i++) {
    tmpInstances.addAll(new Instances(instances));
}
tmpInstances.setClassIndex(-1);
tmpInstances.randomize(new Random(System.nanoTime()));

SimpleKMeans clusterer = new SimpleKMeans();
clusterer.setNumClusters(m_NumCluster);
clusterer.buildClusterer(tmpInstances);

Instances[] clusteredInstances = new Instances[m_NumCluster];
for (int clusterIdx = 0; clusterIdx < m_NumCluster; clusterIdx++) {
    clusteredInstances[clusterIdx] = new Instances(instances, 0);
}
for (int idx = 0; idx < tmpInstances.numInstances(); idx++) {
    int clusterIdx = clusterer.clusterInstance(tmpInstances.get(idx));
    clusteredInstances[clusterIdx].add(tmpInstances.get(idx));
}

m_Classifiers = new Classifier[m_NumCluster];
for (int clusterIdx = 0; clusterIdx < m_NumCluster; clusterIdx++) {
    m_Classifiers[clusterIdx] = new Classifier();
    // Allocate space for counts and frequencies
    m_Classifiers[clusterIdx].initialize(m_NumClasses, m_TotalAttValues);
    // Calculate the counts
    m_Classifiers[clusterIdx].calculateCounts(clusteredInstances[clusterIdx]);
    // Calculate the weights
    m_Classifiers[clusterIdx].calculateWeights();
    int noCorrect = 0;
    for (int idx = 0; idx < clusteredInstances[clusterIdx].numInstances(); idx++) {
        Instance instance = clusteredInstances[clusterIdx].get(idx);
        double predicted_key = m_Classifiers[clusterIdx].classifyInstance(instance);
        String predicted_name = this.toClassName(predicted_key);
        String original_name = instance.stringValue(m_ClassIndex);
        if (original_name.equalsIgnoreCase(predicted_name)) {
            noCorrect++;
        }
    }
    if (clusteredInstances[clusterIdx].numInstances() != 0)
        m_Classifiers[clusterIdx].m_NumCorrectlyClassified = noCorrect;
}
}

/**
 * Set the number of cluster, denoted by parameter "q" in the paper, before building the
 * CDFW Naive Bayes classifier (i.e., before calling "buildClassifier(Instances
 * instances)" function). This number

```

```

    */
    public void setNumClusters(int numClusters) {
        m_NumCluster = numClusters;
    }

    /** Auxiliary min function */
    public static double min(double... array) {
        if (array == null || array.length == 0) {
            return Double.NaN;
        }
        if (array.length == 1) {
            return array[0];
        }
        if (array.length == 2) {
            return Math.min(array[0], array[1]);
        }
        double min = array[0];
        for (int i = 1; i < array.length; i++) {
            min = Math.min(min, array[i]);
        }
        return min;
    }

    /**
     * Calculates the class membership probabilities for the given test instance
     *
     * @param instance the instance to be classified
     * @return predicted class probability distribution
     * @exception Exception if there is a problem generating the prediction
     */
    public double[] distributionForInstance(Instance instance) throws Exception {
        double totalNumInstances = 0;
        ArrayList<double[]> clusterProbs = new ArrayList<>();
        for (int idx = 0; idx < m_NumCluster; idx++) {
            clusterProbs.add(m_Classifiers[idx].distributionForInstance(instance));
            totalNumInstances += (double) m_Classifiers[idx].m_NumInstances;
        }
        double[] clusterWeights = new double[m_NumCluster];
        for (int idx = 0; idx < m_NumCluster; idx++) {
            clusterWeights[idx] = (double) m_Classifiers[idx].m_NumCorrectlyClassified /
                totalNumInstances;
        }

        double[] probs = new double[m_NumClasses];
        for (int classIdx = 0; classIdx < m_NumClasses; classIdx++) {
            for (int clusterIdx = 0; clusterIdx < m_NumCluster; clusterIdx++) {
                double value = clusterWeights[clusterIdx] * clusterProbs.get(clusterIdx)[classIdx];
                probs[classIdx] += value;
            }
        }
        Utils.normalize(probs);
        return probs;
    }

    /**
     * Classifies the given test instance. The instance has to belong to a dataset
     * when it's being classified. Note that a classifier MUST implement either this
     * or distributionForInstance().
     *
     * @param instance the instance to be classified
     * @return the predicted most likely class for the instance or
     *         Utils.missingValue() if no prediction is made
     * @exception Exception if an error occurred during the prediction
     */
    public double classifyInstance(Instance instance) throws Exception {
        double[] dist = distributionForInstance(instance);
        if (dist == null) {
            throw new Exception("Null distribution predicted");
        }
        switch (instance.classAttribute().type()) {
            case Attribute.NOMINAL:
                double max = 0;
                int maxIndex = 0;

```

```

        for (int i = 0; i < dist.length; i++) {
            if (dist[i] > max) {
                maxIndex = i;
                max = dist[i];
            }
        }
        if (max > 0) {
            return maxIndex;
        } else {
            return Utils.missingValue();
        }
        case Attribute.NUMERIC:
        case Attribute.DATE:
            return dist[0];
        default:
            return Utils.missingValue();
    }
}

/**
 * returns a string representation of the classifier
 *
 * @return a representation of the classifier
 */
public String toString() {
    return "CDFW Naive Bayes (Cluster Based Dependent Feature Weighted Naive Bayes)";
}

/**
 * Returns the revision string.
 *
 * @return the revision
 */
public String getRevision() {
    return RevisionUtils.extract("$Revision: 1 $");
}

/**
 * Main method for testing this class.
 *
 * @param args the options
 */
public static void main(String[] args) {
    runClassifier(new CDFWNB(), args);
}
}

```

B.2 WEKA UCI 32 Veri Kümesi için CDFW Naive Bayes Test Kodu

```

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 * CDFWNBTest.java
 * Copyright (C) 2022 Gieliz Chatip and Ferkan Yilmaz
 */

```

```

package weka.classifiers.bayes;

import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.util.ArrayList;
import java.util.Enumuration;
import java.util.List;
import java.util.Locale;
import java.util.Random;

import weka.core.Attribute;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.Filter;
import weka.filters.supervised.attribute.Discretize;
import weka.filters.unsupervised.attribute.ReplaceMissingValues;

/**
 * <!-- globalinfo-start --> Test code for Cluster-based Dependence Weighted Naive
 * Bayes (CDWFNB) Classifier.<br/>
 * <br/>
 * For more information refer to:<br/>
 * <br/>
 * G. Chatip and F. Yilmaz, "A Dependent Feature Weighting Filter for Naive Bayes
 * Classifier," in The 30th IEEE Conference on Signal Processing and
 * Communications Applications Conference (SIU 2022), Safranbolu Kastamonu,
 * Turkiye.
 * <p/>
 * <!-- globalinfo-end -->
 *
 * <!-- technical-bibtex-start --> BibTeX:
 *
 * <pre>
 * @inproceedings{Chatip2022,
 *   author = "Chatip, G. and Yilmaz, F.",
 *   title = "A Dependent Feature Weighting Filter for Naive Bayes Classifier",
 *   booktitle = "The 30th IEEE Conference on Signal Processing and Communications
 * Applications Conference (SIU 2022), Safranbolu, Kastamonu, Turkiye",
 *   year = "2022"
 * }
 * </pre>
 * <p/>
 * <!-- technical-bibtex-end -->
 *
 * <!-- options-start --> Valid options are:
 * <p/>
 *
 * <!-- options-end -->
 *
 * @author Gieliz Chatip (gieliz.chatip@ieee.org)
 * @author Ferkan Yilmaz (ferkan.yilmaz@ieee.org)
 * @version $Revision: 001 $
 */
public class CDFWNBTest {

    public static void main(String[] args) {
        // String filename = "./data/anneal.arff";
        // String filename = "./data/anneal.orig.arff";
        // String filename = "./data/audiology.arff";
        // String filename = "./data/autos.arff";
        // String filename = "./data/balance.scale.arff";
        // String filename = "./data/breast.cancer.arff";
        // String filename = "./data/breast.w.arff";
        // String filename = "./data/colic.arff";
        // String filename = "./data/colic.orig.arff";
        // String filename = "./data/credit.a.arff";
        // String filename = "./data/credit.g.arff";
        // String filename = "./data/diabetes.arff";
        // String filename = "./data/glass.arff";
        // String filename = "./data/heart.c.arff";
        // String filename = "./data/heart.h.arff";
    }
}

```

```

// String filename = "./data/heart.statlog.arff";
// String filename = "./data/hepatitis.arff";
// String filename = "./data/hypothyroid.arff";
// String filename = "./data/ionosphere.arff";
// String filename = "./data/iris.arff";
// String filename = "./data/letter.arff";
// String filename = "./data/lymph.arff";
// String filename = "./data/mushroom.arff"; /**/
// String filename = "./data/segment.arff";
// String filename = "./data/sick.arff";
// String filename = "./data/soybean.arff";
// String filename = "./data/splice.arff";
// String filename = "./data/vehicle.arff";
String filename = "./data/vote.arff";
// String filename = "./data/vowel.arff";
// String filename = "./data/waveform5000.arff";
// String filename = "./data/zoo.arff";

int numClusters = 1; /*The number of cluster */

DecimalFormatSymbols formatSymbols = new DecimalFormatSymbols(Locale.ENGLISH);
formatSymbols.setDecimalSeparator(',');
formatSymbols.setGroupingSeparator(' ');
DecimalFormat numberFormat = new DecimalFormat("##0.00", formatSymbols);
try {
    DataSource source = new DataSource(filename);
    Instances instances = source.getDataSet();
    instances.setClassIndex(instances.numAttributes() - 1);

    boolean hasNonNominal = false;
    boolean hasMissingValues = false;
    Enumeration<Attribute> enu = instances.enumerateAttributes();
    while (enu.hasMoreElements() && (!hasNonNominal || !hasMissingValues)) {
        Attribute attribute = (Attribute) enu.nextElement();
        if (attribute.type() != Attribute.NOMINAL) {
            hasNonNominal = true;
            if (hasMissingValues)
                break;
        }
        Enumeration<Instance> enum2 = instances.enumerateInstances();
        while (enum2.hasMoreElements()) {
            if (((Instance) enum2.nextElement()).isMissing(attribute)) {
                hasMissingValues = true;
                if (hasNonNominal)
                    break;
            }
        }
    }
}
if (hasNonNominal) {
    System.err.println("Warning: discretizing data set");
    Discretize discretizeFilter = new Discretize();
    discretizeFilter.setUseBetterEncoding(true);
    // discretizeFilter.setBinRangePrecision(10);
    // discretizeFilter.setUseBinNumbers(true);
    discretizeFilter.setUseKononenko(true);
    discretizeFilter.setAttributeIndices("first-last");
    discretizeFilter.setInputFormat(instances);
    instances = Filter.useFilter(instances, discretizeFilter);
}

if (hasMissingValues) {
    System.err.println("Warning: filling in missing values in data set");
    ReplaceMissingValues missingValuesFilter = new ReplaceMissingValues();
    missingValuesFilter.setInputFormat(instances);
    instances = Filter.useFilter(instances, missingValuesFilter);
}

int size = instances.size();
Random random = new Random(System.nanoTime());
int testSize = size / 2;
int trainSize = size - testSize;

int no_epoch = 100000;

```

```

double predAccuracy = 0;
double[] predEpochAccuracy = new double[no_epoch];
for (int epoch = 0; epoch < no_epoch; epoch++) {
    instances.randomize(random);

    Instances trainInst = new Instances(instances, 0, trainSize);
    Instances testInst = new Instances(instances, trainSize, testSize);

    CDFWNB dependentFeatureNB = new CDFWNB();
    dependentFeatureNB.setNumClusters(numClusters);
    dependentFeatureNB.buildClassifier(trainInst);

    List<Double> predictionList = new ArrayList<Double>();
    List<Double> originalList = new ArrayList<Double>();

    int classIndex = instances.classIndex();
    for (int idx = 0; idx < testSize; idx++) {
        Instance instance = testInst.get(idx);

        double originalKey = instance.value(classIndex);
        originalList.add(originalKey);

        double predictedKey = dependentFeatureNB.classifyInstance(instance);
        predictionList.add(predictedKey);
    }

    double number = 0;
    for (int idx = 0; idx < testSize; idx++) {
        double o = originalList.get(idx);
        double p = predictionList.get(idx);
        if (o == p) {
            number += 1;
        }
    }

    double epochAccuracy1 = (double) 100 * (number / testSize);
    double completed = (double) 100 * (epoch + 1) / (no_epoch);

    predAccuracy += epochAccuracy1;
    predEpochAccuracy[epoch] = epochAccuracy1;

    StringBuilder sb = new StringBuilder();
    sb.append(">> completed: ");
    sb.append(numberFormat.format(completed));
    sb.append("% ");
    sb.append(" epoch: ");
    sb.append(epoch);
    sb.append(" ");
    sb.append(" epoch accuracy: ");
    sb.append(numberFormat.format(epochAccuracy1));
    sb.append("% ");
    sb.append(" averaged accuracy: ");
    sb.append(numberFormat.format(predAccuracy / (epoch + 1)));
    sb.append("\u00B1");
    sb.append(numberFormat.format(std(predEpochAccuracy, 0, epoch + 1)));
    sb.append("% ");
    sb.append(CDFWNB.m_MinDependency);
    sb.append(" ");
    sb.append(CDFWNB.m_MaxDependency);
    System.out.println(sb.toString());
}
predAccuracy /= no_epoch;

StringBuilder sb = new StringBuilder();
sb.append("\n");
sb.append("Averaged Accuracy: ");
sb.append(numberFormat.format(predAccuracy));
sb.append("\u00B1");
sb.append(numberFormat.format(std(predEpochAccuracy, 0, no_epoch)));
sb.append("%");
sb.append("\n");
System.out.println(sb.toString());

```

```

    } catch (Exception e) {
        e.printStackTrace();
    }
}

static double mean(double[] array, int pos, int length) {
    int last_pos = pos + length;
    double avg = 0;
    for (int i = pos; i < last_pos; i++) {
        avg += array[i];
    }
    avg = avg / length;
    return avg;
}

static double variance(double[] array, int pos, int length) {
    int last_pos = pos + length;
    double avg = mean(array, pos, length);
    double var = 0;
    for (int i = pos; i < last_pos; i++) {
        var += Math.pow(array[i] - avg, 2.0);
    }
    var = var / (length - 1);
    return var;
}

static double std(double[] array, int pos, int length) {
    double var = variance(array, pos, length);
    double std = Math.sqrt(var);
    return std;
}
}

```

Konferans Bildirisi

1. Gieliz Chatip, and Ferkan Yilmaz, "A Dependent Feature Weighting Filter for Naive Bayes Classifier", *in proceedings of The 30th IEEE Conference on Signal Processing and Communications Applications* (SIU 2022), May 15-18, 2022, Safranbolu, Turkey.

