توضیح بخشهای مهم مقاله:

روشهای MDT1 و MDT2 را اینجا توضیح داده که دوتاشون هم از یک خط استفاده میکنن w . x – p = 0

کل داستان مقاله این هست که شما بجای اینکه بیاین داخل هر کدام از گره های DT از یک ویژگی استفاده کنین برای اینکه داده ها رو به دو بخش تقسیم کنید و زیر درخت سمت چپ و راست را بسازید بیایید از کل ویژگیها استفاده کنید و اول خط بسازید که w ضرایب اون خط است و p ضریب ثابتش است

بتوانیم با این خط داده ها رو به دو بخش تقسیم کنیم، سمت چپی و راستی که وقتی داده وارد شد بتونیم بررسی کنیم ببینیم داده جدید با کل ویژگیها در سمت چپ قرار میگره یا راست و بعد بتونیم روی داده ها کلاسیفیکیشن انجام بدیم برخلاف کلاسیفایر DT عادی که در هرکدام از گره ها ما یک تقسیم موازی محورهای مختصات انجام میدیم اینجا دیگه همچین چیزی وجود نداره و بجای اینکه از یک ویژگی استفاده کنیم بیاد از همه ویژگیها در درخت تصمیم استفاده کنه و با استفاده از یک خط داده ها را داخل گره ها را به دو بخش تقسیم میکنیم.

ایده اصلی مقاله هم به همین صورت هست ولی یه تفاوتی داره با 2 روشی که اینجا صحبت شده.

برای اینکه ما w و p را بدست بیاریم دقت کنیم که اینجا کلا با لیبل داده ها کاری نداره و میخواد داده ها را به دو بخش تقسیم کنه. برای اینکه w را بدست بیاره کاری که انجام میده 2 تا روش متفاوت برا MDT1 و MDT2 وجود داره.

روش اول MDT1 این هست که ما تو رنج 1- و 1 یک بردا تولید کنیم بطول مقادیر ویژگیهامون که مقادیرش بصورت تصادفی بین 1- و 1 قرار داره و بعد بردار را نرمال کرده به این شکل که اینجا گفته بعد که بردار نرمال شد تمام داده هایی که درون گره قرار دارند را با استفاده از این بردار مپ کنیم به فضای تک بعدی و بعد median فضای تک بعدی خواهد شد p ما .

یعنی میاد کل داده ها رو map میکنه تو فضای w و بعد w را تو اونها ضرب میکنه در فضای یک بعدی map میشوند median اونها میشه مقدار p.

روش دوم که استفاده کرده که MDT2 هست به این صورت است که میاد pca داده ها رو حساب میکنه و آیگن وکتوری که آیگن ولیو بزرگتری داره، اون رو به عنوانw در نظر میگیره و دوباره به همان صورت median که توضیح داده شد مقدار p را حساب میکند.

دوتا روش استفاده میکنن برای اینکه تشخیص داده بشه که یک گره دیگه نیازی نداره تقسیم بشه و یک گره برگ هست این هست که میاد برای تمام داده ها حساب کنه که از اون داده ها چند درصد وجود داره داخل اون leaf یا برگ و اگر اون درصدها یا مقادیر بجز کلاس غالب کوچکتر از یک دلتایی بودن اون موقع بیاد و بگه که این یک گره برگ هست و دیگه لازم نیست که تقسیم بشه.

و اینجا هم با استفاده از شکل همه چیز رو توضیح میده که شما هر بار با استفاده از یک hyperplain تقسیم میکنین داده ها رو و وقتی به اندازه کافی pure بودن دیگه تقسیم کردن را متوقف میکنیم.

روشی که داخل مقاله کار را انجام داده به این صورت است که در هرکدام از گره ها کاری که انجام میده بجای اینکه حالا یک plain ی پیدا کنه با استفاده از خوشه بندی k-means داده ها را به دوتا خوشه تقسیم میکنه و یک خوشه به سمت چپ و دیگری به زیردرخت راست میرود و وقتی به اندازه کافی pure بود گره ما split انجام نمیشه و گره برگ میشه. سوالی که وجود داره این هست که این k-means رو چطوری بکار ببره تا این عمل انجام بشه.

خب کاری که کردن این بوده که دوتا ورژن از الگوریتم خودش را run کرده. ورژن 1 این هست که از K-meaine++ استفاده میکنه، حالا K-means++ روشی است که به صورت uniform یکی از داده ها را به عنوان مرکز خوشه انتخاب میکنه برای داده K-means و اون یکی مراکز داده یعنی خوشه دوم رو اگه بخوایم انتخاب کنیم احتمال انتخاب شدن هر کدام از داده ها به نسبت فاصله شون از مرکز خوشه اول افزایش پیدا میکنه احتمالش یه همچین توزیعی بکار میبره تا داده بعدی رو انتخاب کنه .

ورژن 2 این هست که برای این فرمولی که اینجا صحبت شد ازش استفاده کرده و از روش MDT2 بعنوان همون توضیح قبل که آیگن وکتور را انتخاب کنه w را حساب کرده p را حساب کرده و وقتی w و p را حساب کرد داده ها را به دو بخش تقسیم کرده بعد میانگین بخش اول را گرفته مرکز خوشه اول و میانگین بخش دوم مرکز خوشه دوم بعد اینا رو به عنوان مراکز خوشه اولیه (initial)داده یه الگوریتم K-Means و بعد اومده الگوریتم K-Means را run کرده و مراکز خوشه را بدست آورده.

بعد کاری که این انجام میده این هست که داده هایی که اول بش داده میشن را با الگوریتم K-means حالا یا با روش اول یا با روش دوم الگوریتم را اجرا میکنه داده ها به دو بخش تقسیم میشوند و این بخش کردن را تا زمانی انجام میدهد که هرکدام از برگها به اندازه کافی pure باشند و وقتی pure شد دیگه بخش کردن را ادامه نمیده. Pure بودن یا معیار عدم تقسیم یک گره را خودش معرفی کرده یعنی برا خودش یک معیار جداگانه هم معرفی کرده.

این معیار جداگانه اینجا کاری که انجام میده به اینصورت هست که یک pi(r) داره که میشه تعداد داده های از یک کلاس i در مجموعه داده اولیه مون چه درصدی ازشون متعلق به کلاس i هستند و pi(t) داره همون معیار رو نشون میده برای گره برگ یعنی داخل یک گره که بش رسیدیم میخوایم چک کنیم ببینیم این گره برگ میتونه باشه یا نه کاری که میکنه این هست که اومده بررسی کرده اون معیار رو دوباره حساب کرده داخل گره برگ یعنی چه تعداد از داده هایی که رسیده به گره برگ متعلق به کلاس i هستند.

معیار هم به اینصورت هست که بررسی میکند درصد داده هایی که متعلق به کلاس major هستند کلاسی که از همه بیشتر نمونه داره داخل داده های ما کلاس میجر تعداد داده هاش یعنی تعداد داده هایی که درون کلاس میجر قرار داره آیا درصد داده های کلاس میجر تو اون برگ بیشتر از درصد داده های کلاس میجر در کل داده ها است؟ طبیعتا اگر این خالصتر شده داده ها باید این درصد داده های کلاس که فراوانی بیشتری داره داخل برگ باید از درصد همون کلاس برای کل داده ها بیشتر باشد پس اینجا این حساب میشه در بخش اول

تو بخش دوم چیزی که حساب میکنه برای بقیه کلاس ها هم این درصد باید از درصد اولیه کمتر باشه فقط یک سری ضرایب هم براش در نظر گرفته یکی لاندا و یکی هم ضریب2 N/n(t) . N تعدا کل داده ها در کل dataset آموزشیمون. N(t) هم تعداد داده ها تو اون گره برگ است. پس اینم معیاری هست که ببینه داخل یک گره آیا باید split انجام بشه یا نشه اگر معیار برآورده نشه split انجام میشه در غیر اینصورت اگر برآورده بشه split انجام نمیشه.

کار بعدی که انجام داده در واقع مقدار لاندا در داخل دیتاست میخواسته ست کنه مقدار لاندا را تغییر داده به در این بازه از 10 به توان 6- تا 10 به توان 3 مقدار لاندا را تغییر داده.

یک کاره دیگه که انجام داده و من نفهمیدم چرا و کار احمقانه ای است اومده داده تست که میاد، اینجا، طبیعتا داده تست باید فاصلش با 1 c و c2 حساب بشه وقتی حساب کردیم ببینیم به کدام نزدیکتر است و داده تست را بفرستیم اونجا.به c1 نزدیکتر باشه بیاد چپ و به c2 نزدیکتر باشه میاد راست. حالا اومده گفته من این کار رو انجام نمیدم میام برای اینکه سرعت بیشتر بشه من نفهمیدم چرا سرعت باید بیشتر بشه اومده c2 – c1 کرده و تقسیم بر نرمشان کرده و اسمش گذاشته w و p را هم حساب کرده با این شکل. دیگه گفته من w را دارم p را هم دارم و میام موقع تست از همون روش MDT2 استفاده میکنیم.

این کل داستان این مقاله هست و اومده کار رو پیش برده.

بعد اومده رو یک سری دیتاست هم تست کرده این دیتاست هایی که ملاحظه میفرمایید نتیجه را ارزیابی کرده و مدعی است که معیار nonsplit که اینجا داره برای داده های imbalance بهتر جواب میده

جدولی که من گزارش کردم داخلش هم داره accuracy را گزارش میکنه و هم mid precision را داره گزارش میکنه و هم مقدار میانگین recall را گزارش میکنه fmeasure را هم گزارش میکنه. برای هر کدام از دیتاست ها.

یکی از روش ها از K-means++ داره استفاده میکنه چون K-mean++ هر بار که run بشه ممکنه مرکز خوشه جدیدی به ما بده برای اینکه تو این مطمئن بشه دقتهاش دقیقتر هستند 5 بار اجرا کرده و میانگین ازشون گرفته و برای روش دومش دیگه این کار رو انجام نداده چون مرکز خوشه اینجا همیشه یک چیز ثابتی است و تغییر نمیکنه و برای داده هاش هم اونایی که بخش تست نداشته خودشون داده خروجیشون اومده 80 به 20 تقسیم کرده برای اینکه پارامتر لاندا را بدست بیاره دوباره از اون بخش train 20% را برداشته با استفاده از validation این مقادیر بهترین مقدار را بدست آورده و مورد استفاده قرار داده.

کد:

کتابخانه ها

from sklearn.datasets import load\_svmlight\_file

برای خواندن داده های آموزشی که به فرمت خاصی است و بتونه اون فرمت خاص رو بخونه. فرمت medsvm یا یه همچین چیزی.

sys.setrecursionlimit(1500)

چون تو بعضی مواقع ظاهرن عمق درخت زیاد میشد و خطا میداد من این تعداد دفعاتی که به صورت recursive میتونن توابع همدیگر را صدا بزنن من مقدارش را افزایش دادم اینجا.

def my\_mode(y):

unq\_labels = np.unique(y)

count = [np.sum(y == label) for label in unq\_labels]

i = np.argmax(count)

return unq\_labels[i]

خب این کاری که انجام میده mod یه دونه آرایه را حساب میکنه. یک آرایه را دریافت میکنه اینجا و Mod آن را حساب میکنه برای این استفاده میشه که داخل یک گره برگ اگه خواستیم کلاس غالب را حساب کنیم از این mod استفاده میکنیم و کلاس غالب را بدست میاریم.

class tree\_node:

def \_\_init\_\_(self, w=None, p=None, is\_leaf=False, label=None, childs=None):

self.w = w

self.p = p

self.is\_leaf = is\_leaf

self.label = label

if childs is None:

self.childs = []

خب این tree node هست که میتونه داخلش w داشته باشه p داشته باشه ، برگ هست یا نه اگه برگ هست لیبلش چیه؟ و اگر برگ نیست بچه هاش چه چیزی اند.

class BDTKS:

def \_\_init\_\_(self, lmbd, ver):

بعد کلاس اصلی را داریم که از الگوریتم پیاده سازی شده. که مقدار لاندا و ver را اولش دریافت میکنه داخل تابع init ش و ver هم ورژن 1 یا 2 است که از Kmeans++ استفاده کنه یا از ازون روش خاص خودش که برای مقدار دهی اولیه استفاده کرده.

def fit(self, x, y):

بعد تابع fit رو داریم،

n\_labels = int(np.max(y) + 1)

کاری که میکنه اول تعداد لیبل ها رو بدست میاره. کاری که انجام دادیم به اینصورت هست که برای تمام دیتاست ها لیبل ها به صورت هستند که کلاس ها 1،2،3، 0 الی آخراند... بنابر این اگر شما max لیبل ها را بدست بیارید و + 1 کنید میشه تعدا لیبلها شما چون من دیتاست رو به یک شکل در نظر گرفتم.

p\_r = [np.sum(y == i) / y.shape[0] for i in range(n\_labels)]

self.p\_r = p\_r

اینجا احتمال همون p\_r که داخل مقاله براتون توضیح دادم حساب میشه برا تک تک کلاس ها چون بعدن ازش استفاده خواهد شد.

self.N = y.shape[0]

تعداد کل داده ها ی دیتاست اولیه را اینجا ست میکنیم دوباره.

self.n\_labels = n\_labels

تعداد لیبل ها یا کلاسها را ست میکنیم.

if self.ver == "v1":

و اگر کاربر ورژن 1 خواسته بود برای fit کردن داده ها

self.root = self.make\_tree\_v1(x, y)

این تابع را صدا میزنیم

else:

self.root = self.make\_tree\_v2(x, y)

و اگر ورژن 2 خواسته بود این را صدا میزند.

def make\_tree\_v1(self, x, y):

is\_leaf = self.is\_leaf\_node(x, y)

make\_tree\_version ما کاری که انجام میده اول تابع is\_leaf را صدا میزنه بعدا در ادامه توضیح خواهم داد و بررسی میکنه که این گره برگ میتواند باشد یا خیر

if is\_leaf == True:

label = my\_mode(y)

اگر گره برگ بود در اینصورت mod داده ها را حساب میکنه که میشه لیبل مان

node = tree\_node(is\_leaf=True, label=label)

return node

یک گره جدی میسازه که میگه برگ هست و این گره جدید را به عنوان خروجی بر میگرداند.

else:

kmeans = KMeans(n\_clusters=2, init="k-means++", n\_init=1).fit(x)

و اگر نه الگوریتم Kmeans را اجرا میکنیم که از روش initialization اولیه kmeans++ استفاده میکنه همون چیزی که داخل مقاله استفاده شده و داده را fit میکنیم.

c = kmeans.cluster\_centers\_

مراکز خوشه ها را بدست میاریم

cluster\_labels = kmeans.labels\_

لیبل خوشه ها را بدست میاریم. لیبل خوشه اگر 0 بود یعنی به خوشه اول تعلق دارد اگر 1 بود یعنی به خوشه دوم تعلق دارد.

w = (c[1] - c[0]) / np.linalg.norm(c[1] - c[0])

طبق اون چیزی که داخل مقاله گفته شده من w را اینجا حساب میکنم عبارت اول تقسیم بر نرمشان

p = np.dot(w, (c[0] + c[1]) / 2)

بعد P را حساب میکنم که p میشود مقدار وسط اینها طبق چیزی که داخل مقاله گفته شده یعنی داخل مقاله گفته شده که: c[0] + c[1]) / 2 ضرب در w کنیم میشه مقدار p شما

idx = cluster\_labels == 0

x\_1 = x[idx]

y\_1 = y[idx]

x\_2 = x[np.logical\_not(idx)]

y\_2 = y[np.logical\_not(idx)]

بعد اندیس خوشه اول و دوم را اینجا بدست میاریم.

node = tree\_node()

node.w = w

node.p = p

node.childs.append(self.make\_tree\_v1(x\_1, y\_1))

node.childs.append(self.make\_tree\_v1(x\_2, y\_2))

return node

یک گره جدید میسازیم،w و p را set میکنیم در نهایت با داده های سمت چپ (x\_1, y\_1) یک درخت میسازیم، در واقع یک گره جدید میسازیم اضافه میکنیم به فرزندان. اینجا هم یک گره جدید میسازیم به صورت بازگشتی به فرزندانش اضافه میشه و در نهایت گره ای که ایجاد شده را در خروجی بر میگردانیم.

def make\_tree\_v2(self, x, y):

is\_leaf = self.is\_leaf\_node(x, y)

if is\_leaf == True:

label = my\_mode(y)

node = tree\_node(is\_leaf=True, label=label)

return node

ورژن 2 این بخشش یکسان است .

kmeans = KMeans(n\_clusters=2, init=c, n\_init=1).fit(x)

او بخشی که تفاوت داره قبل از اینکه بیایم الگوریتم kmeans را اجرا کنیم.

else:

pca = PCA(n\_components=1)

pca.fit(x)

w\_ = pca.components\_

میایم رو داده هامون PCA را set میکنیم و انجام میدیم و میگیم فقط یک component میخوایم که مهمترین یا principle کامپوننت را برای ما برگردان.

new\_pos = np.dot(x, w\_.T)

داده هامون در principle component ضرب میکنیم

p\_ = np.median(new\_pos)

و median ش را به عنوان p در نظر میگیریم.

idx = (new\_pos <= p\_).squeeze()

بعد اون داده هایی که محل جدیدشون یعنی اون محلی که w درش ضرب شده را اگر کوچکتر از p باشند میشه کلاس 1 در واقع باید بریم سمت چپ و اگر بزرگتر باشند باید بریم سمت راست.

c = np.zeros((2, x.shape[1]))

c[0] = np.mean(x[idx, :], axis=0)

c[1] = np.mean(x[np.logical\_not(idx), :], axis=0)

داده ها به دو بخش تقسیم میشن. میانگین بخش اول میشه مقدار اولیه برای c[0] و میانگین بخش دوم میشه مقدار اولیه برای c[1]

kmeans = KMeans(n\_clusters=2, init=c, n\_init=1).fit(x)

بعد الگوریتم kmeans را با استفاده از اون مراکز خوشه جدید که الان حساب کردیم بر روی خوشه اولیه اعمال کرده و run میکنیم

c = kmeans.cluster\_centers\_

cluster\_labels = kmeans.labels\_

و وقتی مراکز خوشه جدید نیز بدست آمد دیگه بقیه مراحل مشابه روش اول است ک شرح داده شد.

def is\_leaf\_node(self, x, y):

N\_t = y.shape[0]

p\_t = [np.sum(y == label) / N\_t for label in range(self.n\_labels)]

l\_major = np.argmax(p\_t)

is\_leaf = p\_t[l\_major] > self.p\_r[l\_major]

is\_leaf کاری که انجام میده خیلی ساده است. دقیقا N\_t را حساب میکنیم که N\_t را داخل مقاله توضیح دادم بعد p\_t را حساب میکنیم بعد l\_major حساب میکنیم کدام یک از کلاسها بیشترین احتمال را دارد. خب 2 تا شرط داشتیم داخل مقاله یکی اینکه p\_t بزرگتر از p\_r باشد برای l\_major اینجا اون شرط را چک میکنیم.

p\_r = copy(self.p\_r)

بعد از p\_r یعنی احتمال تک تک کلاسها که داخل تابع fit حساب کردیم ازش یک کپی میگیریم

del p\_t[l\_major]

del p\_r[l\_major]

چون میخوام l\_major را هم از p\_t و p\_r حذف کنم

idx = [

not p\_t\_i < p\_r\_i \* self.lmbd \* self.N / N\_t \*\* 2

for p\_t\_i, p\_r\_i in zip(p\_t, p\_r) ]

و بعد اون شرط دوم را چک کنم که گفته بود این و در نتیجه اینجا not قرار دادم قرار دادن not به این دلیل است که اگر تمامی شرطها این بخشش true باشه در این صورت همشان 1 بر خواهند گرداند و این not باعث میشه همشون صفر بشن

is\_leaf = is\_leaf and np.sum(idx) == 0

return is\_leaf

اگر من اعضای همه این آرایه را اینجا جمع کنم و نتیجه جمع هم صفر شود یعنی تمام این شرط ها که به این شکل قرار داشتند true بودن. و بنابرایم شرط من ارضا شده و من میگم is\_leaf را برگرداند. حالا هر کدام از این شرطها درست باشند is\_leaf میشه true وگرنه میشه false .

def predict(self, x):

prd = np.zeros((x.shape[0],))

for i, s in enumerate(x):

تابع Predict دانه دانه داده های ما رو بر میداره x اون چیزی هست که برا predict به ما داده شده

node = self.root

while node.is\_leaf == False:

if np.dot(np.expand\_dims(node.w, axis=0), s.T) - node.p < 0:

node = node.childs[0]

else:

node = node.childs[1]

از گره root شروع میکنیم تا زمانی که به گره برگ نرسیدیم w را ضربدر اون داده میکنیم و منهای p مکنیم اگر کوچکتر از 0 بود زیردخت سمت چپ وگرنه زیردرخت سمت راست.

prd[i] = node.label

return prd

و در نهایت وقتی به گره برگ رسیدیم لیبل اون گره برگ لیبل داده ما میشود و برای تمام داده ها این کار رو انجام میدیم و نتیجه را بر میگردانیم.

def load\_data(file\_name):

اینجا تابع load\_data است که برای بیشتر دیتاست ها .t هم داریم که اون برا تستش است و برای بعضیها هم نداریم.

adr = "datasets/" + file\_name

data = load\_svmlight\_file(adr)

اصل داده را اینجا با این فایل میخونیم.

trn\_x = data[0].todense()

trn\_y = data[1]

بخش اول چون ماتریس اسپارس برمیگردونه من این را با استفاده از todense تبدیل کردم به ماتریس عادی و یک tuple بر میگردونه بخش اول تاپل x است و بخش دوم تاپل y ما است.

trn\_x, unq\_ind = np.unique(trn\_x, axis=0, return\_index=True)

trn\_y = trn\_y[unq\_ind]

چون بعضی از داده ها تکراری هستند بخش تکراری را اینجا حذف میکنیم چون بخش تکراری باعث میشه که اگه داده ها تکراری باشند معیار purity ما درست ست نشه و در نهایت بیوفته تو loop بینهایت. بنابراین داده های unique باید حذف بشن تو مقاله هم به این نکته اشاره شده و داده هایی که دقیقن یکسان هستند حذف میشن.

trn\_y, unq\_labels = pd.factorize(trn\_y, sort=True)

اینجا تابع factorize را به این دلیل استفاده کردم که اگر لیبل ها 1 و 1- باشند یا هرچیز دیگری من تبدیلش میکنم به 0و1و2و3و...

if os.path.isfile(adr + ".t"):

و اینجا چک میکنم که اگر فایل .t وجود داشت

tst\_y = np.zeros(tmp.shape)

اون فایل .t رو هم میخونیم و بر میگردانیم به عنوان داده های تست.

else:

trn\_x, tst\_x, trn\_y, tst\_y = train\_test\_split(

trn\_x, trn\_y, test\_size=0.2, stratify=trn\_y

)

و اگر نبود اون موقع 20 درصد از داده های train را به عنوان داده test در نظر میگیریم.

for i, label in enumerate(unq\_labels):

tst\_y[tmp == label] = i

این for هم که اینجا نوشتیم به ابن دلیل است که همون کلاسها که گفتیم خروجیشون رو 0و1 یا 2 میکنیم اینجا برای داده های test هم دقیقا همین کار رو انجام میدیم.

trn\_x, val\_x, trn\_y, val\_y = train\_test\_split(

trn\_x, trn\_y, test\_size=0.2, stratify=trn\_y

)

return trn\_x, trn\_y, tst\_x, tst\_y, val\_x, val\_y

بعد گفته بود برای داده های validation هم دقیقا 20 درصد از داده های train را برداریم که 20 درصد را به عنوان validation بر میداریم و return میکنیم.

data\_sets = [

]

اینجا هم لیست داده ها است که فقط برای اونکه پروتیین بود مشکل داشت و نمیخوند.

lmbds = [10 \*\* i for i in range(-6, 4)]

n\_itr = 5

اینجا هم بقیه کارها است که لاندا رو اون مقادیری که گفته تو اون رنج تغییر میدیم و تعداد iteration هایی که انجام داد تا میانگین گیری انجام بده بینشون رو هم اینجا انجام میدیم.

best\_lmbd = lmbds[i]

و در نهایت بهترین مقدار لاندا را بدست میاریم. Best lambda ازبین بهترین داده باید چک بشه.

trn\_x = np.concatenate((trn\_x, val\_x), axis=0)

trn\_y = np.concatenate((trn\_y, val\_y), axis=0)

وقتی بهترین مقدار لاندا را بدست آوردم اون موقع من داده های validation رو هم چسبوندم به داده های train و با استفاده از اونا training را انجام دادم.

if ver == "v2":

دقت کنیم که اگر ورژن 2 باشه دیگه لازم نیست 5 بار اجرا بشه ولی اگر ورژن 1 بود اون موقع 5 بار اجرا میکنیم و بینشون میانگین میگیریم و در نهایت خروجی ها رو مینویسیم.

dataset: acoustic, tree version: v2, accuracy: 0.6927683329104288, precision: 0.6795139575832575, recall: 0.7031722943198542, f1: 0.6871312044023824

dataset: acoustic, tree version: v1, accuracy: 0.6278710987059122±0.0029529197581418497, precision: 0.6176704967094815±0.0030127068419269305, recall: 0.6095401712381221±0.003266923346059559, f1: 0.6132980555328655±0.003120735162195264

dataset: aloi, tree version: v2, accuracy: 0.8621313896431804, precision: 0.8685424246039026, recall: 0.8610464858928482, f1: 0.8609697665098238

dataset: aloi, tree version: v1, accuracy: 0.8627030632175694±0.0013157464700836667, precision: 0.8681002361357614±0.0012063740965428723, recall: 0.8612732406843243±0.0014403751593030413, f1: 0.8608865041654654±0.0013392483004973305

dataset: combined, tree version: v2, accuracy: 0.7383405227099721, precision: 0.7212406901031256, recall: 0.7421924954532756, f1: 0.7289551371565776

dataset: combined, tree version: v1, accuracy: 0.6787211367673179±0.002117716609861629, precision: 0.6667273074105797±0.0021825051371660467, recall: 0.6826413731314693±0.0016946521434809967, f1: 0.6722117779817731±0.0019909345282712655

dataset: covtype, tree version: v2, accuracy: 0.9489427983786993, precision: 0.9151155223582464, recall: 0.9090766470071779, f1: 0.911927670583267

dataset: covtype, tree version: v1, accuracy: 0.9474540244227775±0.0006766674076935546, precision: 0.9085559428719865±0.0018013408042082577, recall: 0.9062326065681512±0.0020068107387960373, f1: 0.9073585804365436±0.0016745330369653953

dataset: gisette, tree version: v2, accuracy: 0.91, precision: 0.9105320495361989, recall: 0.91, f1: 0.9099708305490979

dataset: gisette, tree version: v1, accuracy: 0.9132000000000001±0.005192301994298872, precision: 0.9132735535691822±0.005204317277697014, recall: 0.9132000000000001±0.005192301994298872, f1: 0.9131961804056099±0.005191793992148989

dataset: ijcnn1, tree version: v2, accuracy: 0.9526722718400017, precision: 0.8813209088957132, recall: 0.8281231872410851, f1: 0.8521638423606146

dataset: ijcnn1, tree version: v1, accuracy: 0.9534857853240425±0.002764883482735632, precision: 0.8728485930248508±0.0072688430833843295, recall: 0.8485750351858838±0.010958438130453095, f1: 0.8601541917853218±0.00912528281756856

dataset: letter, tree version: v2, accuracy: 0.8778, precision: 0.8795812271895911, recall: 0.8786047814113894, f1: 0.8787686891621311

dataset: letter, tree version: v1, accuracy: 0.8777999999999999±0.003003997336882974, precision: 0.878917514219746±0.002968176944933172, recall: 0.8786276477062783±0.00308480098643103, f1: 0.8784194421189838±0.0030224209842833203

dataset: madelon, tree version: v2, accuracy: 0.71, precision: 0.7118454088952654, recall: 0.71, f1: 0.7093670660549641

dataset: madelon, tree version: v1, accuracy: 0.6563333333333333±0.0056174331821175535, precision: 0.6566577420890235±0.005724954549233328, recall: 0.6563333333333333±0.0056174331821175795, f1: 0.6561581886615176±0.005569489832511994

dataset: mushrooms, tree version: v2, accuracy: 0.9993846153846154, precision: 0.9994068801897984, recall: 0.9993614303959131, f1: 0.9993837752770641

dataset: mushrooms, tree version: v1, accuracy: 0.9997538461538461±0.00030147566065021666, precision: 0.9997538250175515±0.00030183176423802103, recall: 0.999753521233577±0.0003022050434953692, f1: 0.9997535211396192±0.00030187372066155314

dataset: pendigits, tree version: v2, accuracy: 0.9499714122355631, precision: 0.9504241863787101, recall: 0.9503539862444205, f1: 0.9499611985516067

dataset: pendigits, tree version: v1, accuracy: 0.9531160663236135±0.005224604597626354, precision: 0.9535907381774239±0.004917408444083991, recall: 0.9534000343844304±0.005225441854743281, f1: 0.9530589813646385±0.005216196330794021

dataset: phishing, tree version: v2, accuracy: 0.9118409680207433, precision: 0.9116988386973317, recall: 0.9118613902847571, f1: 0.91176919166243

dataset: phishing, tree version: v1, accuracy: 0.9178910976663787±0.005937965095010439, precision: 0.9178337597446433±0.005945368338059241, recall: 0.917856843742522±0.005890262779921182, f1: 0.9178128469759151±0.005935411383629882

dataset: satimage, tree version: v2, accuracy: 0.869, precision: 0.8554349281075943, recall: 0.8486447740717672, f1: 0.8517076079905334

dataset: satimage, tree version: v1, accuracy: 0.876±0.00783581520966388, precision: 0.8621224988887473±0.009053709659306632, recall: 0.8604219133614983±0.009586469591399967, f1: 0.8611802721641029±0.009261873183933131

dataset: segment, tree version: v2, accuracy: 0.9473684210526315, precision: 0.9497664625193533, recall: 0.9476190476190477, f1: 0.9476666018669633

dataset: segment, tree version: v1, accuracy: 0.9454545454545453±0.004879444510615115, precision: 0.9481612641029884±0.00490293533782049, recall: 0.9456650246305418±0.004869516198391351, f1: 0.9454351277703379±0.00501534931624732

dataset: seismic, tree version: v2, accuracy: 0.6353717330626745, precision: 0.6231096381346583, recall: 0.6293920512141772, f1: 0.6151563304063107

dataset: seismic, tree version: v1, accuracy: 0.5850088809946714±0.0027592649564655448, precision: 0.5580182952243075±0.002515222528986952, recall: 0.565549085182436±0.0028708143735514747, f1: 0.5576161510053842±0.0027165591150167083

dataset: shuttle, tree version: v2, accuracy: 0.998551724137931, precision: 0.9133983646437986, recall: 0.7651523650181586, f1: 0.8132000614240203

dataset: shuttle, tree version: v1, accuracy: 0.9987172413793104±0.00012791197924823437, precision: 0.9006324750371787±0.036258328899443124, recall: 0.8079995509390508±0.06377066484132382, f1: 0.834281753734154±0.05037292229335471

dataset: skin\_nonskin, tree version: v2, accuracy: 0.9974725381549528, precision: 0.9958985448253868, recall: 0.9979248796267386, f1: 0.996905032107535

dataset: skin\_nonskin, tree version: v1, accuracy: 0.9976280742684942±0.0001454907120355508, precision: 0.9962272250054198±0.0001796918192765888, recall: 0.9979720632736739±0.00021091605734511567, f1: 0.9970946560874399±0.0001783254070198046

dataset: usps, tree version: v2, accuracy: 0.898355754857997, precision: 0.8910601480062113, recall: 0.8869440183080896, f1: 0.8882665198561608

dataset: usps, tree version: v1, accuracy: 0.8926756352765322±0.004395963129070812, precision: 0.8854037745394301±0.004926375443411372, recall: 0.8809273673821997±0.005073120746546317, f1: 0.8825435715927406±0.004990484441941117

dataset: w8a, tree version: v2, accuracy: 0.967828238913785, precision: 0.7330211917829309, recall: 0.8063162999848974, f1: 0.7641315857616959

dataset: w8a, tree version: v1, accuracy: 0.9793324861213296±1.1102230246251565e-16, precision: 0.8044208854308049±1.1102230246251565e-16, recall: 0.924265889433603±0.0, f1: 0.853540829830649±0.0

