Gépi Látás

Kézzel írt karakterfelismerés

Látrányi Péter Krisztián | VEC2Ao |

Tartalomjegyzék

Bevezetés

Az írásfelismerés fontossága többek között abban rejlik, hogy az írásunk szinte olyan pontossággal azonosít minket, mint az ujjlenyomatunk vagy a DNS-ünk, sőt, esetenként jobban is, hiszen még az egypetéjű ikrekre sem jellemző, hogy tökéletesen azonos lenne az írásképük, míg ez a DNS-szerkezetükre és az ujjlenyomatukra teljesül.

A kézírást már régóta használják azonosítási célokra az informatikában is. Az első aláírás-felismerő rendszert 1965-ben fejlesztették ki, ezt pedig további fejlesztések követték. Az aláírások felismerése azonban még nem okoz akkora nehézséget, mint a karakterfelismerés, mivel az ember aláírásában az azonosságokat vizsgáljuk és nem törődünk azzal, hogy ténylegesen milyen betűkből áll össze a név. Amikor azonban az egyes betűket kell felismernünk, figyelembe kell vennünk, hogy emberenként, sőt, még az egyes embereknél időben is szembetűnő eltérések tapasztalhatók az írásképben. Ezt figyelembe véve pedig nem állíthatjuk, hogy egy betű kinézetét egyetlen vagy akár egy tucat mintapélda alapján meg tudjuk határozni, és ezek segítségével egy leírt karaktert valamilyen osztályba be tudunk sorolni.

Karakterfelismerés nehézségei:

A karakterfelismerést nehezíti:

- **„Ugyanazon" karakterek emberenként meglehetősen különbözőek lehetnek méretben, alakban és stílusban,** emellett még ugyanazon ember esetén is megfigyelhetők bizonyos eltérések a különböző időpillanatokban leírt karakterek között.
- Minden más képpel egyetemben a karaktereket, illetve szövegeket tartalmazó képek esetén is számolnunk kell a képalkotás – digitalizálás – folyamán, illetve az egyéb okokból a képre került **zajokkal és egyéb képhibákkal** (pl. elmosódás).
- A karakterek kinézetének nincsenek megszeghetetlen, tudományosan lefektetett alapszabályai. Ennek megfelelően csak minták alapján tudunk valamiféle szabályosságot megállapítani az egyes karaktertípusok kinézetével kapcsolatban.

A fejlesztendő programot Pythonban fogom fejleszteni a
NumPY és a Scikit-Learn csomagok felhasználásával.
A program működési elvét a Lineáris SVM algoritmus határozza meg.

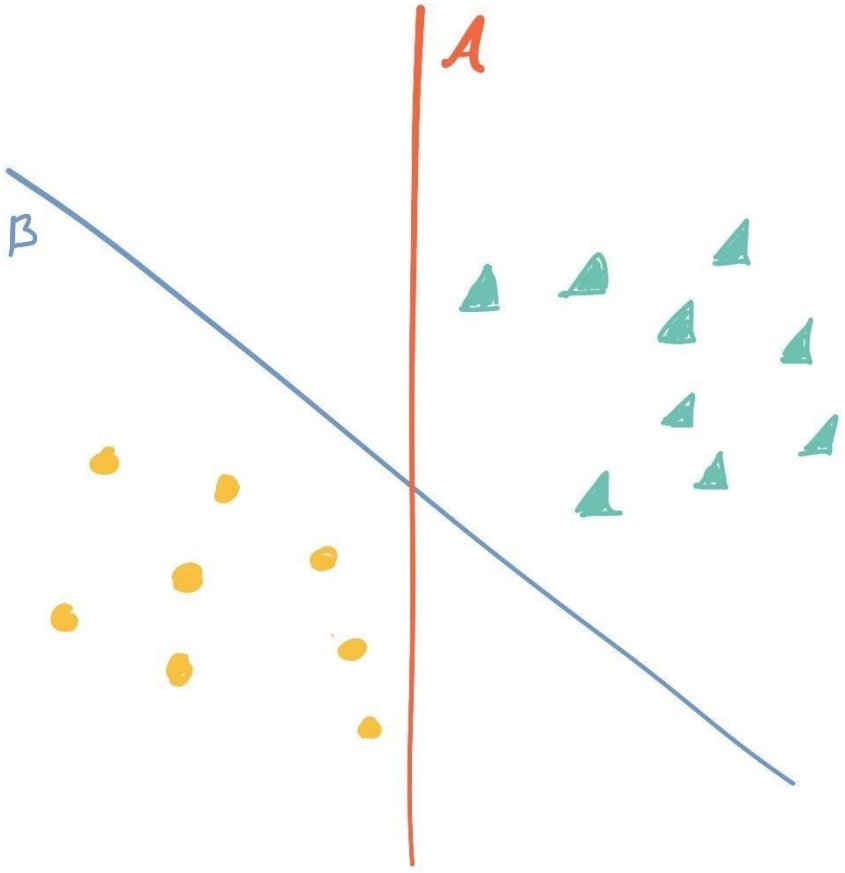
Támaszvektor-gépek (SVM)

Ebben a fejezetben a support vector machine (SVM) algoritmust mutatom be, amely klasszifikációra, regresszióra, de még akár klaszterezésre is alkalmazható. SVM-et már a ’60-as években használtak, népszerűsége azonban a modern SVM kialakulása után kezdett növekedni. Azóta ezzel a módszerrel jelentős eredményeket értek el. Többek között használható arc- és kézírásfelismerésre, képek klasszifikációjára

Fontos megkötés, hogy csak numerikus prediktorokat tud kezelni, valamint a kimeneti változó értéke csak -1 vagy 1 lehet, de előnye, hogy nem használ semmilyen véletlent. Az ebben a fejezetben bemutatásra kerülő SVM-ek alapkérdése megegyezik: hogyan lehet a prediktorok terét egy hipersíkkal elválasztani.

Lineáris SVM

A kiinduló feladatban a változók $\mathbb{R}p$ terében egy p-1 dimenziós hipersíkot keres, mely tökéletesen elválasztja a kimeneti két kategóriáját egymástól. [4] Valahogy így:



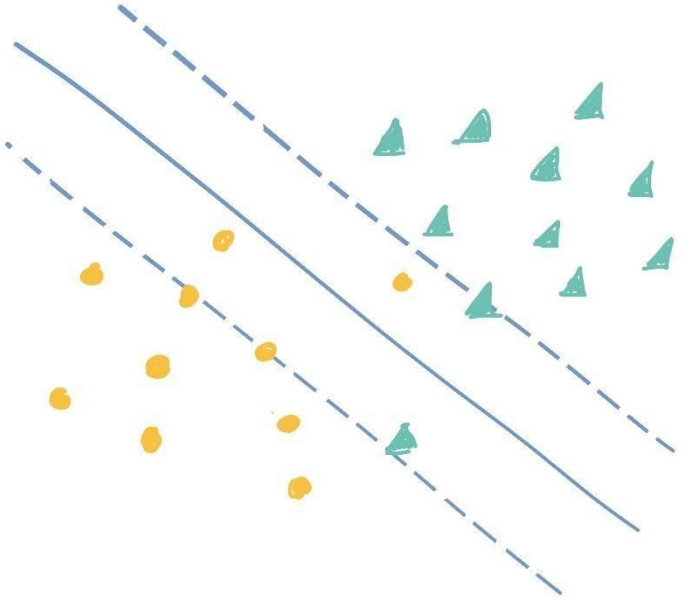
$\mathbf{w} \cdot \mathbf{x} + b = 0$ Hipersík egyenlete

A fenti ábrán a két csoportot ugye könnyű elválasztani, de vegyük észre, hogy végtelen számú lehetséges elválasztási lehetőség van. Fel is rajzoltam kettőt (A, B)

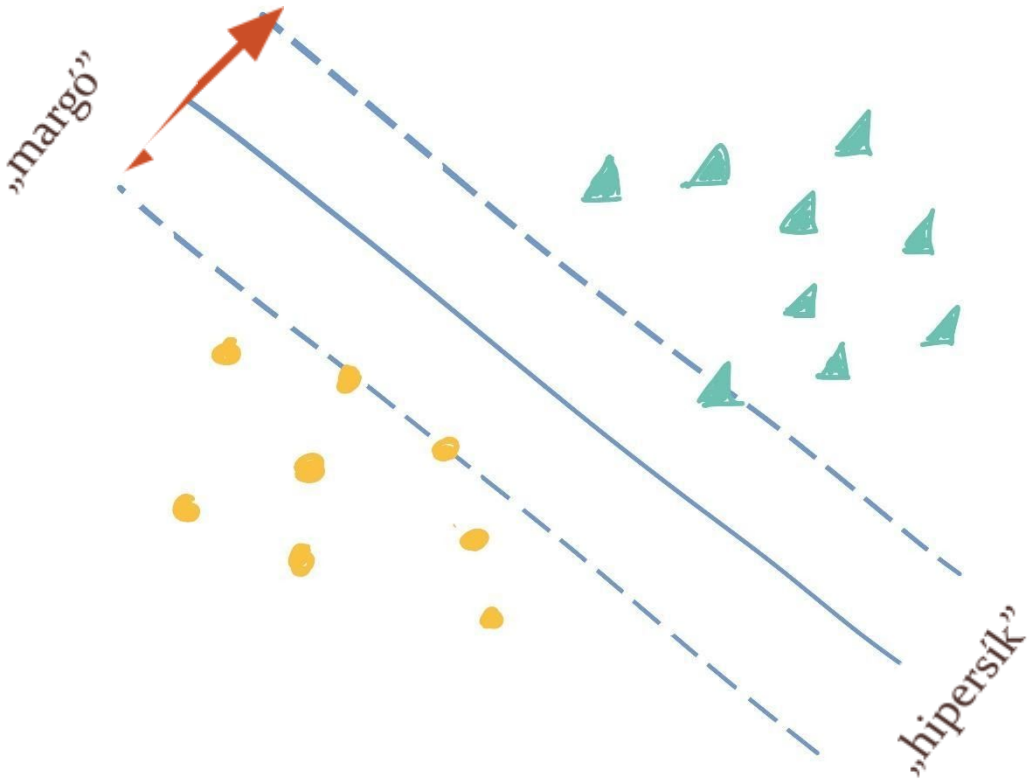
A tesztadatok alapján mindkettő jó eredményt ad, de melyiket választanánk a gyakorlati életben? Én a B-t tenném. Azon egyszerű oknál, hogy az nagyjából egyforma távolságra helyezkedik el mindkét csoporttól. A lineáris SVM lényegében ezt a fajta gondolkodást próbálja megvalósítani. Vagyis azt a határvonalat keresi, ami egyforma távolságra van mindkét osztálytól. Ehhez pedig segéd "margókat" használ. Lényegében kijelöl két párhuzamos egyenest, nevezzük margóknak őket, a hipersík körül, és arra törekszik, hogy ezek a margók minél távolabb kerüljenek a hipersíktól.

Amennyiben a két osztályunk lineárisan jól elválasztható egymástól, akkor az úgynevezett **szigorú margókat (hard margin)** tudjuk alkalmazni, mint a lenti példában is.

(Ha pedig nem tudjuk az osztályokat egyértelműen elválasztani egymástól, mert van néhány pontunk, amik megakadályozzák, hogy egyértelműen elválaszthassuk a két osztályt lineárisan, ekkor a **Hinge loss**-t hívjuk segítségül. Ebben az esteben **lágy margókról (soft margin)** beszélünk. Ez lényegében egy olyan függvény, ami o értéket vesz fel, ha a pont a margó jó oldalán helyezkedik el, és egyenletesen növekszik ahogyan egyre távolodunk a rossz irányba. A Hinge loss segítségével most már tudjuk mérni a pontok rosszaságát)
Ezek az esetek így néznek ki:



Szigorú margós eset:



Vegyük észre, hogy mindkét margón van legalább egy megfigyelt pont. Ezek a “**support vektorok**”. Lényegében ezek azok a pontok, amik számítanak a hipersík meghatározásában.

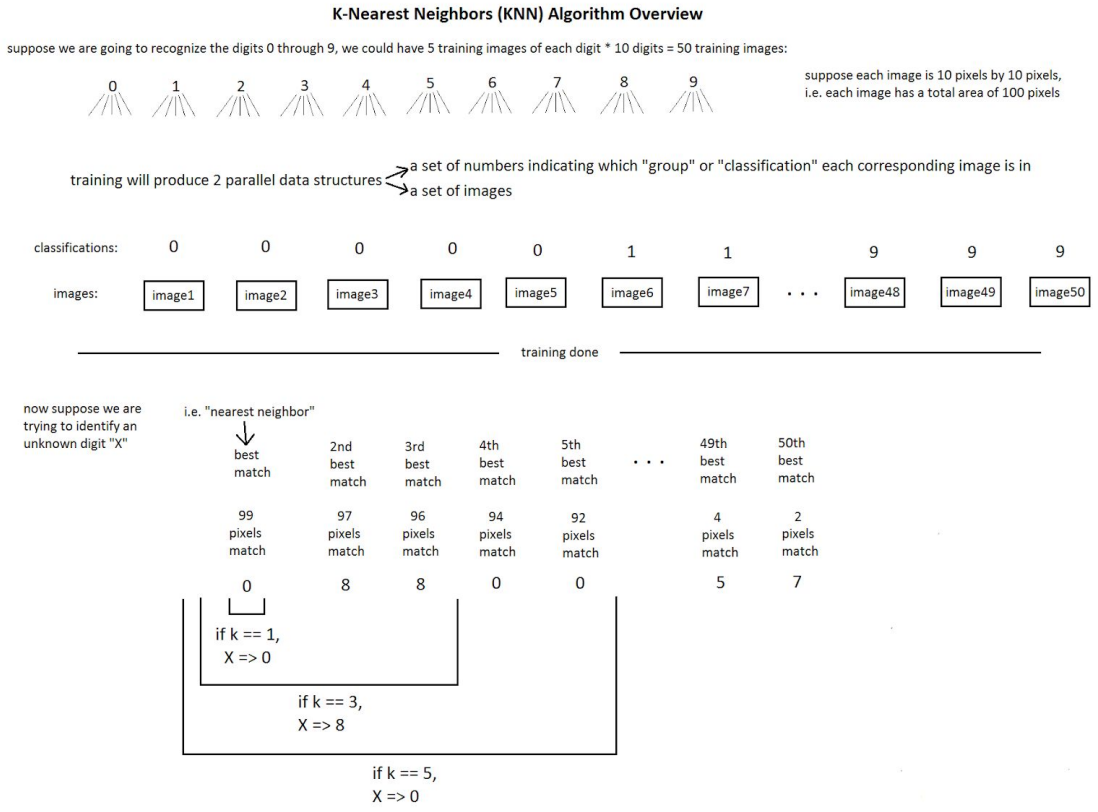
Support vektor egyenletei:

$$w * x + b = +1$$
$$w * x + b = -1$$

A feladat a hipersík és a támaszvektor közötti távolság („margó”) maximalizálása.

Egyéb algoritmusok karakterfelismerésre

KNN K legközelebbi szomszéd algoritmus[2]:



Lépései:

1. Euklideszi távolság számítása a test adatpont és a betanított adatok között.

2. A kiszámított távolság rendezése növekvő sorrendben

3. A legközelebbi k szomszédos elemek megtalálása a rendezett tömbben

4. Leggyakoribb osztály keresés a k szomszédos elemek között

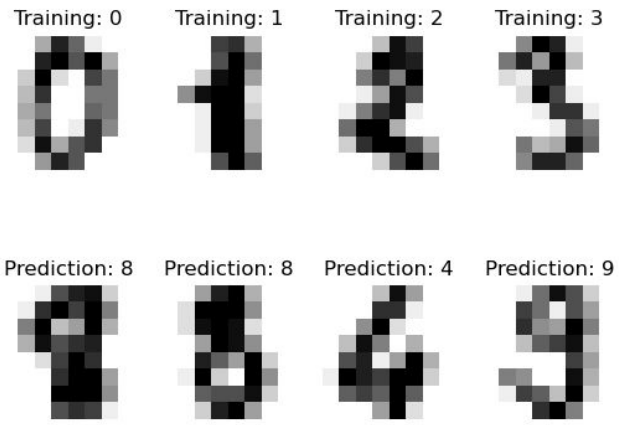
5. A prediktált osztály visszaadása

6. Megegyezőség vizsgálata, pontossági szám alapján, hogy meggyőződjünk arról, hogy a predikció jó-e, avagy sem

Scikit-Learn

Az SK Learn digits adatacsomagja 180 mintát 8x8 pixel felbontásban tartalmaz minden egyes számjegyre (0-9). A minták előállításához 43 ember kézírását használták fel.
A példákban **fehér háttérrel, feketével írt számok** találhatóak, ezért ugyanilyen tulajdonságú négyzetes felbontású képekre működik a legjobban az algoritmus.

Az adatcsomag a Numpy, SciPy és a matplotlib csomagokra épül.



Programfejlesztés Lépései

- Importálandó könyvtárak importálása
- A tanuló algoritmus betanítása.
- A teszt kép beolvasása, átméretezése.
- Teszt kép normalizálása (intenzitás csökkentése)
- Predikció
- Eredmény kiíratása
- Tesztelés
- Hibakezelés

A program futtatásához szükséges eszközök:

- ➢ Python program futtatására alkalmas fordító
- ➢ A program futtatásához szükséges Scikit-learn, Scikit-image csomagok telepítése
- ➢ Kép/ek amelyek karaktereket tartalmaznak
- ➢ A DigitRecognition.py forrásfájl

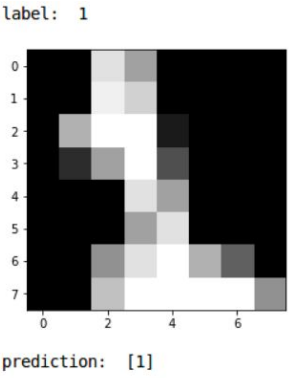
A program a jpg, png, gif, bmp fájl formátumot támogatja.

A program fejlesztése során a Thonny 3.2.7-es verzióját használtam.
Az Scikit-learn csomag esetében a 0.21.3, míg a Scikit-image-nél a 0.17.1 -es verziót.

A programot teszteltem a fent megnevezett program, és a csomagok legújabb verziójával is, és szerencsére a program továbbra is hiba nélkül lefutott:

Thonny	3.3.1
Scikit-learn	0.23.2
Scikit-image	0.18.0

Tesztelés eredményei



A programot minden egyes számra 10 képpel teszteltem. Így összességében 100 tesztesetet végeztem.

A tesztképeket a Paint nevezetű programban állítottam elő, továbbá internetről letöltött képeket is használtam. A karaktereket egérrel, illetve érintéses bevitellel rajzoltam. A képeknek különböző a vonalvastagságuk (jellemzően 1 és 8 pixel között), eltérő a felbontásuk és a színeik, továbbá a karakterek különböző szögben és méretben helyezkednek el a képeken.

Pontossági Mátrix

Helyes Kategória	1	2	3	4	5	6	7	8	9	
1	10	0	0	0	0	0	0	0	0	100% 0%
2	5	4	0	0	0	0	1	0	0	40% 60%
3	0	0	6	0	2	0	0	1	1	60% 40%
4	0	0	0	10	0	0	0	0	0	100% 0%
5	0	0	0	0	10	0	0	0	0	100% 0%
6	0	0	0	0	0	9	0	2	0	90% 20%
7	0	0	0	0	0	0	10	0	0	100% 0%
8	0	0	2	0	0	1	0	7	0	70% 30%
9	0	0	0	0	0	0	0	1	9	90% 10%
	66.67%	100.00%	75.00%	100.00%	83.33%	90.00%	90.91%	63.64%	90.00%	
	33.33%	0.00%	25.00%	0.00%	16.67%	10.00%	9.09%	36.36%	10.00%	
	1	2	3	4	5	6	7	8	9	

Prediktált Kategória

A mátrixból, kiolvasható, az, hogy az adott számból, **hányszor** sikerült **helyesen,** illetve **helytelenül felismerni** az adott karaktert, továbbá, az is látszik az ábrán, hogy az adott számjegyet milyen **gyakorisággal, milyen kategóriába** azonosította be a program a tesztesetekből.

A főátlóban szereplő gyakoriságok a helyesen felismert képek darabszámát tartalmazza.

Az algoritmusnak leginkább a **2**-es, **3**-as karakterek felismerése jelentette a problémát.

A **2-es** karakterek esetében 10/4 képet sikerült csak a programnak sikeresen felismerni, így a hibaarány ezen számjegy esetén 60%-os.
A 2-es karaktert 10/5-ször 1-es számjegyként kategorizálta, míg egyszer 7-es számjegyként.
A hiba leginkább az alacsonyra vett gamma érték miatt következik be, mert így nagyon hasonló, kétértelmű esetekkel is összeveti a program a képet.
Amennyiben a gamma értéket megnöveljük pl. 0.004-re már 10/7 -szer ad az algoritmus megfelelő eredeményt a 2-es karakter esetén.

A **3-as** karakterek esetében a program 4x hibázott így összességében 60%-ban sikeresen felismerte a 3-as számjegyet.

Az 1-es, 4-es, 5-ös, 7-es karaktereket minden esetben hiba nélkül felismerte.
A 6-os, 8-as, 9-es esetekben 1x,2x hibázott az algoritmus.

Összességében az elvégzett tesztesetekre a program **84%**-ban helyes eredményt adott. (A mintául szolgáló adatcsomag tesztelésekor a Scikit Learn 97%-os pontosságot állapított meg.)

Az algoritmus hatékonyabbá tehető a **mintaképek számának növelésével.**
Jelenleg az Sklearn digit adatcsomag csak 180 mintát tartalmaz minden egyes számjegyre, amennyiben ezt legalább ezres nagyságrendre növelnék nagyságrendekkel hatékonyabb lenne az algoritmus, bár a futásidő is minimálisan megnőne.
Továbbá lehetne **negatív példákat** is megtanítani az algoritmusnak.

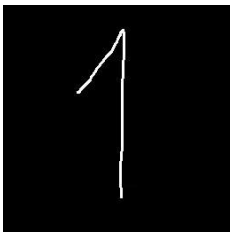
A kereséshez használt **C és a gamma** értékek állítgatásával is hatékonyabbá tehető az algoritmus.
A programban az alpértelmezett C érték 100, a gamma érték pedig 0,001.
A tesztelés során a 2-es képek esetén a gamma értékek átállításával jelentős hatékonyság növekedést sikerült elérnem.

Felhasználói Leírás

Ahhoz, hogy a karakterfelismerő programot használni tudjuk szükségünk van egy **Python** nyelvű fejlesztői környezetre, ami képes a **Scikit-learn**, illetve a **Scikit-image**-ben található beágyazott függvények futtatására

Ilyen szoftver például a Thonny a Scikit-image és a Scikit-learn csomagokkal telepítve.

A program fejlesztése során a Thonny 3.2.7-es verzióját használtam.
Az Scikit-learn csomag esetében a 0.21.3, míg a Scikit-image-nél a 0.17.1 -es verziót.



A program futtatásához továbbá rendelkeznünk kell tesztelendő kép/ek-el, amiknek az **elérési útját** a program futtatása során a parancssorban kell magadnunk (elérési út, fájlnév, kiterjesztés formában).
Pl: C:\Users\latranyi\Documents\images\1_e.jpg

A program a jpg, png, gif, bmp fájl formátumot támogatja.

A program a futtatás során a megadott képet float64-es formátuban olvassa be, majd átméretezi 8x8-as felbontásba. A következő lépésben a kép intenzitástartományt 0 – 16 érték közé szorítja. Végül osztályba sorolja (a lineáris SVM módszer alapján) a képet és kijelzi az eredményt.

A program az eredményt a parancssorban közli az alábbi formában:

```
Shell
>>>
>>>
>>>
>>> %Run Digit_Recognition.py
 Kézzel írt karakterfelismerő program


 Kérem adja meg az input fájl elérési útját: images/1_e.jpg
 The predicted digit is [1]
>>>
```

Amennyiben a program futtatása során hiba lépne fel kérem ellenőrizze, hogy:

- A program által használt adatcsomagok verziója megfelelő-e.
- A megadott fájl elérési útja helyes-e, illetve, hogy az adott fájl nincs-e zárolás alatt.
- Továbbá nézze meg, hogy a megadott fájl kiterjesztését támogatja-e a program.

A program szabadon használható, fejleszthető, nem áll szerzői jogi védelem alatt.

Felhasznált irodalom

[1] Brown, Erik W. Applying Neural Networks to CharacterRecognition,1992

[2] https://hu.wikipedia.org/wiki/Python_(programoz%C3%A1si_nyelv)

[3]https://scikitlearn.org/stable/auto_examples/classification/plot_digits_classification.html

[4] Virtás Dávid: Prediktív módszerek és gépi tanulás alkalmazásai a biztosításban Szakdolgozat, 2019