zadani

November 7, 2020

Vítejte u domácí úlohy do SUI. V rámci úlohy Vás čeká několik cvičení, v nichž budete doplňovat poměrně malé fragmenty kódu, místo na ně je vyznačené jako `pass` nebo `None`. Pokud se v buňce s kódem již něco nachází, využijte/neničte to. V dvou případech se očekává textová odpověď, tu uvedete přímo do zadávající buňky. Buňky nerušte ani nepřidávejte.

Maximálně využívejte `numpy` a `torch` pro hromadné operace na celých polích. S výjimkou generátoru minibatchí by se nikde neměl objevit cyklus jdoucí přes jednotlivé příklady.

U všech cvičení je uveden počet bodů za funkční implementaci a orientační počet potřebných řádků. Berte ho prosím opravdu jako orientační, pozornost mu věnujte pouze, pokud ho významně překračujete. Mnoho zdaru!

1 Informace o vzniku řešení

Vyplňte následující údaje **(3 údaje, 0 bodů)**

- Jméno autora: Peter Grofčík
- Login autora: xgrofc00
- Datum vzniku: 21.10.2020

[1]:
```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy.stats
```

2 Přípravné práce

Prvním úkolem v této domácí úloze je načíst data, s nimiž budete pracovat. Vybudujte jednoduchou třídu, která se umí zkonstruovat z cesty k negativním a pozitivním příkladům, a bude poskytovat: - pozitivní a negativní příklady (`dataset.pos`, `dataset.neg` o rozměrech [N, 7]) - všechny příklady a odpovídající třídy (`dataset.xs` o rozměru [N, 7], `dataset.targets` o rozměru [N])

K načítání dat doporučujeme využít `np.loadtxt()`. Netrapte se se zapouzdřování a gettery, berte třídu jako Plain Old Data.

Načtěte trénovací (`{positives,negatives}.trn`), validační (`{positives,negatives}.val`) a testovací (`{positives,negatives}.tst`) dataset, pojmenujte je po řadě (`train_dataset`, `val_dataset`, `test_dataset`).

(6+3 řádků, 1 bod)

[2]:
```
class dataset:
    def __init__(self, pos, neg):
        self.pos = np.loadtxt(pos)
        self.neg = np.loadtxt(neg)
        self.xs = np.vstack((self.pos,self.neg))
        self.targets = np.concatenate(([1]*self.pos.shape[0],[0]*self.neg.
 ↪shape[0]))

train_dataset = dataset("positives.trn", "negatives.trn")
val_dataset = dataset("positives.val", "negatives.val")
test_dataset = dataset("positives.tst", "negatives.tst")
print('positives', train_dataset.pos.shape)
print('negatives', train_dataset.neg.shape)
print('xs', train_dataset.xs.shape)
print('targets', train_dataset.targets.shape)
```

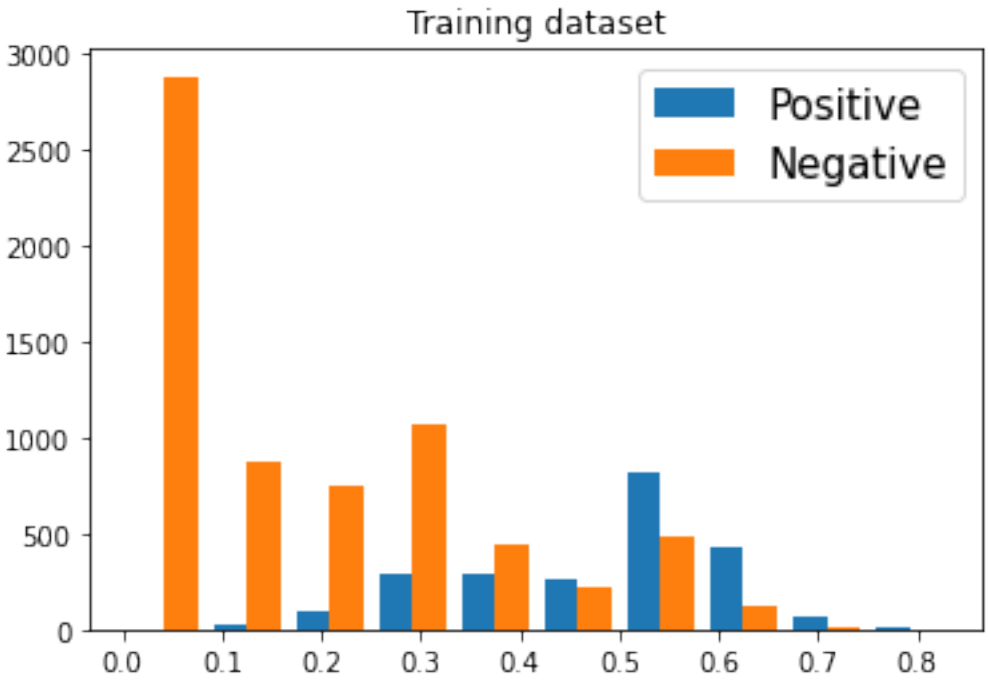
```
positives (2280, 7)
negatives (6841, 7)
xs (9121, 7)
targets (9121,)
```

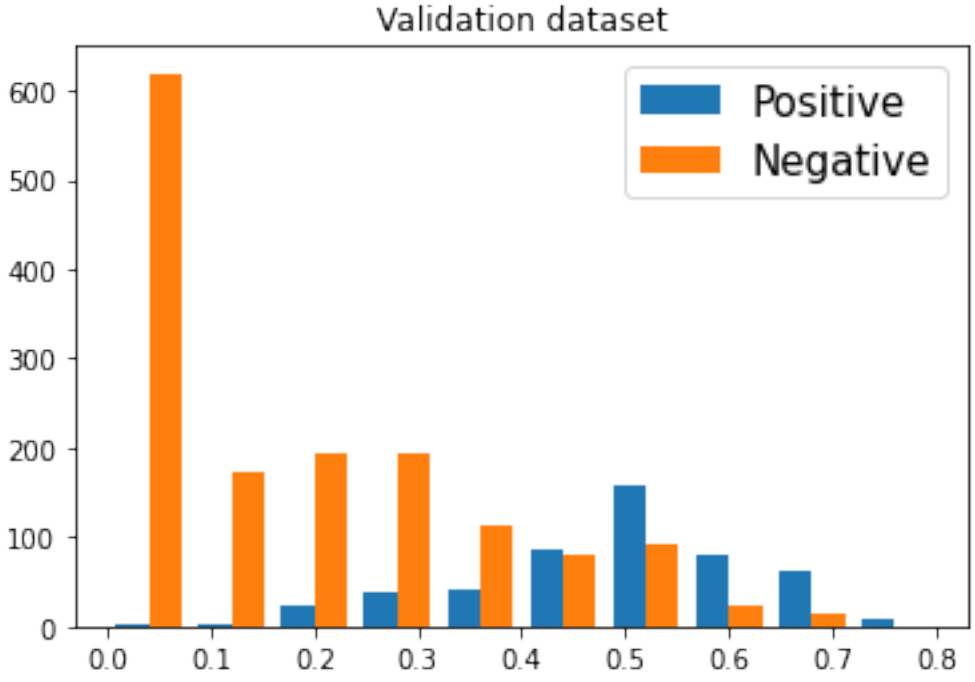
V řadě následujících cvičení budete pracovat s jedním konkrétním příznakem. Naimplementujte pro začátek funkci, která vykreslí histogram rozložení pozitivních a negativních příkladů (`plt.hist()`). Nezapomeňte na legendu, ať je v grafu jasné, které jsou které. Funkci zavolejte dvakrát, vykreslete histogram příznaku 5 – tzn. šestého ze sedmi – pro trénovací a validační data **(5 řádků, 1 bod)**.

[3]:
```
FOI = 5  # Feature Of Interest

def plot_data(poss, negs, titles):
    plt.hist([poss, negs], label=['Positive','Negative'])
    plt.title(titles)
    plt.legend(prop={'size': 15})
    plt.show()

plot_data(train_dataset.pos[:, FOI], train_dataset.neg[:, FOI], 'Training␣
 ↪dataset')
plot_data(val_dataset.pos[:, FOI], val_dataset.neg[:, FOI], 'Validation␣
 ↪dataset')
```


Training dataset
Positive
Negative
3000
2500
2000
1500
1000
500
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


Validation dataset
Positive
Negative
600
500
400
300
200
100
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

2.0.1 Evaluace klasifikátorů

Než přistoupíte k tvorbě jednotlivých klasifikátorů, vytvořte funkci pro jejich vyhodnocování. Nechť se jmenuje `evaluate` a přijímá po řadě klasifikátor, pole dat (o rozměrech [N] nebo [N, F]) a pole tříd ([N]). Jejím výstupem bude *přesnost*, tzn. podíl správně klasifikovaných příkladů.

Předpokládejte, že klasifikátor poskytuje metodu `.prob_class_1(data)`, která vrací pole posteriorních pravděpodobností třídy 1 (tj. p(y=1|x)) pro daná data. Evaluační funkce bude muset provést tvrdé prahování (na hodnotě 0.5) těchto pravděpodobností a srovnání získaných rozhodnutí s referenčními třídami. Využijte fakt, že `numpy`ovská pole lze mj. porovnávat mezi sebou i se skalárem.

(3 řádky, 1 bod)

[4]:
```
def evaluate(classifier, inputs, targets):
    compute = np.asarray(classifier.prob_class_1(inputs) > 0.5).astype(int)
    compute = (compute == targets).astype(int)
    return compute.sum()/compute.size

class Dummy:
    def prob_class_1(self, xs):
        return np.asarray([0.2, 0.7, 0.7])

print(evaluate(Dummy(), None, np.asarray([0, 0, 1])))  # should be 0.66...
```

```
0.6666666666666666
```

2.0.2 Baseline

Vytvořte klasifikátor, který ignoruje vstupní hodnotu dat. Jenom v konstruktoru dostane třídu, kterou má dávat jako tip pro libovolný vstup. Nezapomeňte, že jeho metoda `.prob_class_1(data)` musí vracet pole správné velikosti, využijte `np.ones` nebo `np.full`.

(4 řádky, 1 bod)

[5]:
```
class PriorClassifier:
    def __init__(self, tp):
        self.type = tp
    def prob_class_1(self, xs):
        return np.full(xs.size, self.type)

baseline = PriorClassifier(0)
val_acc = evaluate(baseline, val_dataset.xs[:, FOI], val_dataset.targets)
print('Baseline val acc:', val_acc)
```

```
Baseline val acc: 0.75
```

3 Generativní klasifikátory

V této části vytvoříte dva generativní klasifikátory, oba založené na Gaussovu rozložení pravděpodobnosti.

Začněte implementací funce, která pro daná 1-D data vrátí Maximum Likelihood odhad střední hodnoty a směrodatné odchylky Gaussova rozložení, které data modeluje. Funkci využijte pro natrénovaní dvou modelů: pozitivních a negativních příkladů. Získané parametry – tzn. střední hodnoty a směrodatné odchylky – vypište.

(5 řádků, 0.5 bodu)

[6]:
```
def function(data):
    return data.sum()/data.size, np.sqrt(((data-(data.sum()/data.size))**2).
 ↪sum()/data.size)

positives = function(train_dataset.pos[:,FOI])
negatives = function(train_dataset.neg[:,FOI])
print("Positive : ", positives)
print("Negative : ", negatives )
```

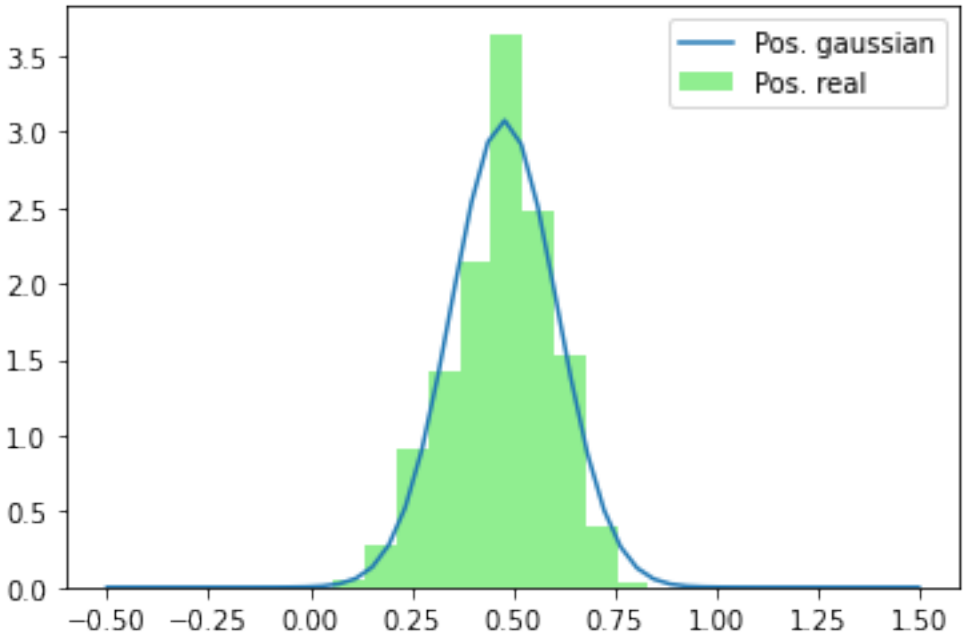
```
Positive :  (0.478428821613158, 0.12971703647258465)
Negative :  (0.17453641132613792, 0.17895975196381242)
```

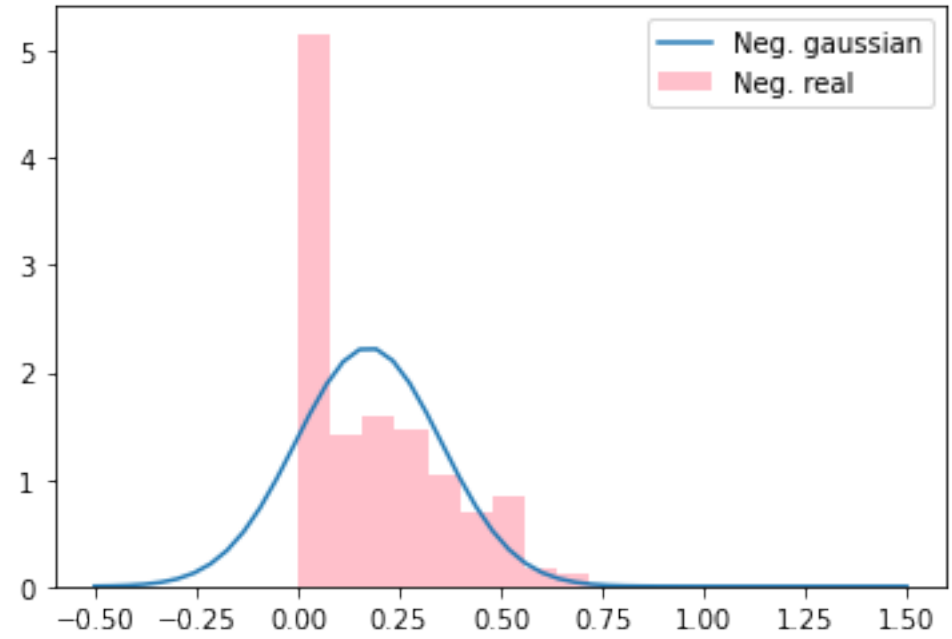
Ze získaných parametrů vytvořte `scipy`ovská gaussovská rozložení `scipy.stats.norm`. S využitím jejich metody `.pdf()` vytvořte graf, v němž srovnáte skutečné a modelové rozložení pozitivních a negativních příkladů. Rozsah x-ové osy volte od -0.5 do 1.5 (využijte `np.linspace`) a u volání `plt.hist()` nezapomeňte nastavit `density=True`, aby byl histogram normalizovaný a dal se srovnávat s modelem.

(2+8 řádků, 1 bod)

[7]:
```
def plot_gaus(dataset, data, labelname, col):
    rozloz = scipy.stats.norm(data[0], data[1])
    plt.plot(np.linspace(-0.5, 1.5), rozloz.pdf(np.linspace(-0.5, 1.5)),␣
 ↪label=labelname+"gaussian")
    plt.hist(dataset, density=True, label=labelname+"real", color=col)
    plt.legend(prop={'size': 10})
    plt.show()

plot_gaus(train_dataset.pos[:,FOI],positives, "Pos. ", "lightgreen")
plot_gaus(train_dataset.neg[:,FOI],negatives, "Neg. ", "pink")
```





Naimplementujte binární generativní klasifikátor. Při konstrukci přijímá dvě rozložení poskytující metodu `.pdf()` a odpovídající apriorní pravděpodobnost tříd. Jako všechny klasifikátory v této domácí úloze poskytuje metodu `prob_class_1()`.

(9 řádků, 2 body)

[8]:
```
class bin_class:
    def __init__(self, roz, apri):
        self.pdf=roz
        self.apri=apri

    def prob_class_1(self, xs):
        pos = self.pdf[0].pdf(xs)*self.apri[0]
        neg = self.pdf[1].pdf(xs)*self.apri[1]
        return pos/(pos+neg)


classic=bin_class([scipy.stats.norm(positives[0],positives[1]),
                   scipy.stats.norm(negatives[0],negatives[1])],[0.75,0.25])
```

Nainstancujte dva generativní klasifikátory: jeden s rovnoměrnými priory a jeden s apriorní pravděpodobností 0.75 pro třídu 0 (negativní příklady). Pomocí funkce `evaluate()` vyhodnoťte jejich úspěšnost na validačních datech.

(2 řádky, 1 bod)

[9]:
```
classifier_flat_prior = bin_class([scipy.stats.norm(positives[0],positives[1]),
                                   scipy.stats.
 ↪norm(negatives[0],negatives[1])], [0.5,0.5])
classifier_full_prior = bin_class([scipy.stats.norm(positives[0],positives[1]),
                                   scipy.stats.
 ↪norm(negatives[0],negatives[1])], [0.25,0.75])

print('flat:', evaluate(classifier_flat_prior, val_dataset.xs[:, FOI],␣
 ↪val_dataset.targets))
print('full:', evaluate(classifier_full_prior, val_dataset.xs[:, FOI],␣
 ↪val_dataset.targets))
```

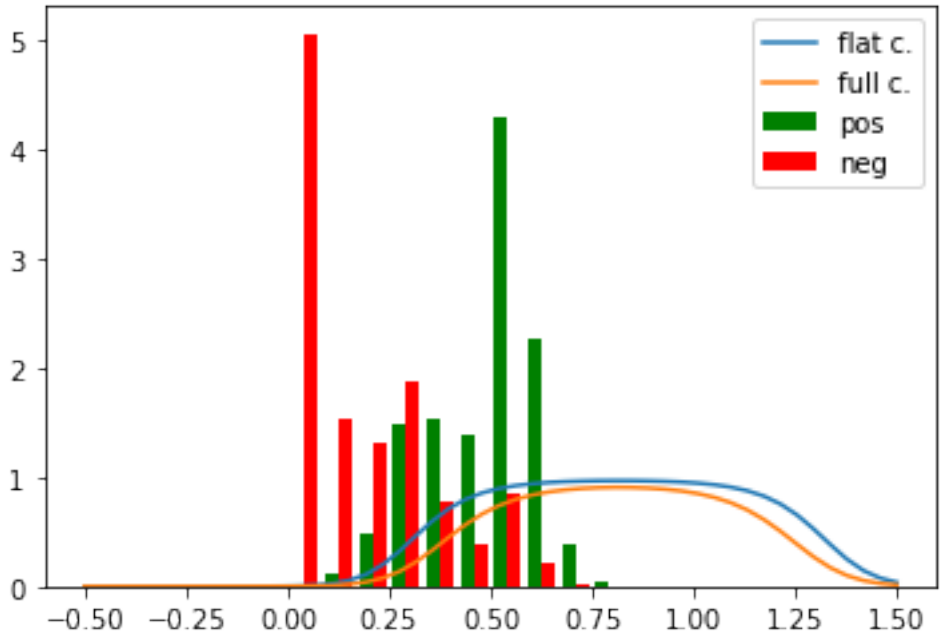
```
flat: 0.809
full: 0.8475
```

Vykreslete průběh posteriorní pravděpodobnosti třídy 1 jako funkci příznaku 5 pro oba klasifikátory, opět v rozsahu <-0.5; 1.5>. Do grafu zakreslete i histogramy rozložení trénovacích dat, opět s `density=True` pro zachování dynamického rozsahu.

(8 řádků, 1 bod)

[10]:
```
def plot_pas(dataset, classifiers, classifierlabel, datasetlabel, col):
    plt.plot(np.linspace(-0.5, 1.5), classifiers[0].prob_class_1(np.linspace(-0.
 ↪5, 1.5)), label=classifierlabel[0])
    plt.plot(np.linspace(-0.5, 1.5), classifiers[1].prob_class_1(np.linspace(-0.
 ↪5, 1.5)), label=classifierlabel[1])
    plt.hist(dataset, density=True, label=datasetlabel, color=col)
    plt.legend(prop={'size': 10})
```

```
plot_pas([train_dataset.pos[:,FOI],train_dataset.neg[:,FOI] ],
         [classifier_flat_prior,classifier_full_prior],
         ["flat c.", "full c."],
         ["pos","neg"],["green","red"])
```



Interpretujte, přímo v této textové buňce, každou rozhodovací hranici, která je v grafu patrná **(3 věty, 2 body)**: Jedna z rozhodovacích hraníc sa nachádza na intervale (0.1 - 0.2) kedy oba klasifikátory začínajú predpokladať možný výskyt triedy 1 aj keď stále v menšom množstve ako triedu 0. Do dosiahnutia tejto hranice oba klasifikátory predpokladajú výskyt triedy 0 s pravdepodobnosťou 100%. Rozhodovacia hranica +-0.35 je miesto kde sa krížia histogramy pozitývnych a negatívnych dát (ich pomer je primeraný +- 1:1). V tomto momente by mala ideálna pasteriózna funkcia mať hodnotu 50%. V tomto momente Flat classifier má približne 50% pravdepodobnosť a je teda vhodnejší ako FULL ktorý stále predpokladá mierne vyššiu pravdepodobnosť pre neg dáta. Hranica 0.65 kedy už sa oba classifiery blížia k 100% vyhodnoteniu pre triedu 1.

4 Diskriminativní klasifikátory

V následující části budete přímo modelovat posteriorní pravděpodobnost třídy 1. Modely budou založeny na PyTorchi, ten si prosím nainstalujte. GPU rozhodně nepotřebujete, veškeré výpočty budou velmi rychlé, ne-li bleskové.

Do začátku máte poskytnutou třídu klasifikátoru z jednoho příznaku.

[11]:
```
import torch
import torch.nn.functional as F

class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.w = torch.nn.parameter.Parameter(torch.tensor([1.0]))
        self.b = torch.nn.parameter.Parameter(torch.tensor([0.0]))

    def forward(self, x):
        return torch.sigmoid(self.w*x + self.b)

    def prob_class_1(self, x):
        prob = self(torch.from_numpy(x))
        return prob.detach().numpy()
```

Pro trénovaní diskriminativních modelů budete potřebovat minibatche. Implementujte funkci, která je bude z daných vstupních a cílových hodnot vytvářet. Výsledkem musí být možno iterovat, ideálně funkci napište jako generátor (využijte klíčové slovo `yield`). Jednotlivé prvky výstupu budou dvojice PyTorchových `FloatTensor`ů (musíte zkonvertovat z numpy a nastavit typ) – první prvek vstupní data, druhý očekávané výstupy. Počítejte s tím, že vstup bude `numpy`ovské pole, rozumná implementace využije `np.random.permutation()` a Advanced Indexing.

Připravený kód funkci použije na konstrukci tří minibatchí pro trénování identity, měli byste vidět celkem pět prvků náhodně uspořádaných do dvojic, ovšem s tím, že s sebou budou mít odpovídající výstupy.

(6 řádků, 2 body)

[12]:
```
def batch_provider(xs, targets, batch_size=10):
    rand = np.random.permutation(np.arange(xs.shape[0]))
    for start_idx in range(0, xs.shape[0], batch_size):
        end_idx = min(start_idx + batch_size, xs.shape[0])
        index = rand[start_idx:end_idx]
        yield torch.tensor(xs[index],dtype=torch.float32),torch.
 ↪tensor(targets[index],dtype=torch.float32)

inputs = np.asarray([1.0, 2.0, 3.0, 4.0, 5.0])
targets = np.asarray([1.0, 2.0, 3.0, 4.0, 5.0])
for x, t in batch_provider(inputs, targets, 2):
    print(f'x: {x}, t: {t}')
```

```
x: tensor([4., 2.]), t: tensor([4., 2.])
x: tensor([5., 3.]), t: tensor([5., 3.])
x: tensor([1.]), t: tensor([1.])
```

Dalším krokem je implementovat funkci, která model vytvoří a natrénuje. Jejím výstupem bude (1) natrénovaný model, (2) průběh trénovací loss a (3) průběh validační přesnosti. Jako model vracejte ten, který dosáhne nejlepší validační přesnosti. Jako loss použijte binární cross-entropii

(`F.binary_cross_entropy()`), akumulujte ji přes minibatche a logujte průměr. Pro výpočet validační přesnosti využijte funkci `evaluate()`. Oba průběhy vracejte jako obyčejné seznamy.

V implementaci budete potřebovat dvě zanořené smyčky: jednu pro epochy (průchody přes celý dataset) a uvnitř druhou, která bude iterovat přes jednotlivé minibatche. Na konci každé epochy vyhodnoťte model na validačních datech. K datasetům (trénovacímu a validačnímu) přistupujte bezostyšně jako ke globálním proměnným.

(cca 14 řádků, 3 body)

[13]:
```
def train_single_fea_llr(fea_no, nb_epochs, lr, batch_size):
    ''' fea_no -- which feature to train on
        nb_epochs -- how many times to go through the full training data
        lr -- learning rate
        batch_size -- size of minibatches
    '''
    model = LogisticRegression()
    losses = []
    accuracies = []
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    for x in range(0, nb_epochs):
        model_actual = batch_provider(train_dataset.xs[:, fea_no],␣
↪train_dataset.targets, batch_size)
        los = []
        for val, tar in model_actual:
            y_pred = model(val)
            loss = F.binary_cross_entropy(y_pred, tar, reduction='mean')
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            los.append(loss.detach().numpy())

        accuracies.append(evaluate(model, val_dataset.xs[:, fea_no],␣
↪val_dataset.targets))
        losses.append(np.average(np.array(los)))
        if(accuracies[x-1] < accuracies[x] or x == 0):
            best_model = copy.deepcopy(model)

    return best_model, losses, accuracies
```

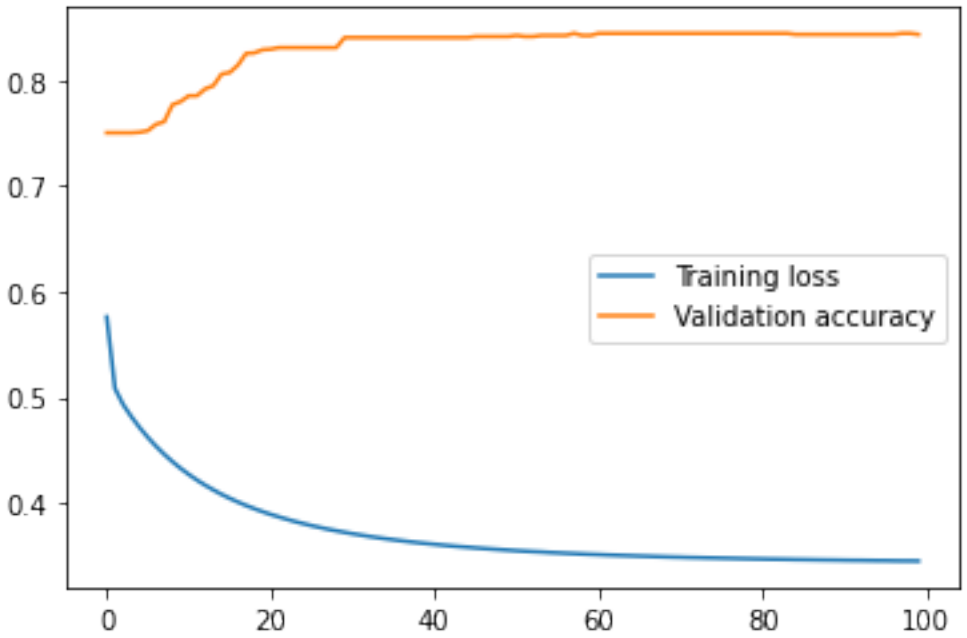
Funkci zavolejte a natrénujte model. Uveďte zde parametry, které vám dají slušný výsledek. Měli byste dostat přesnost srovnatelnou s generativním klasifikátorem s nastavenými priory. Neměli byste potřebovat víc než 100 epoch. Vykreslete průběh trénovací loss a validační přesnosti, osu x značte v epochách.

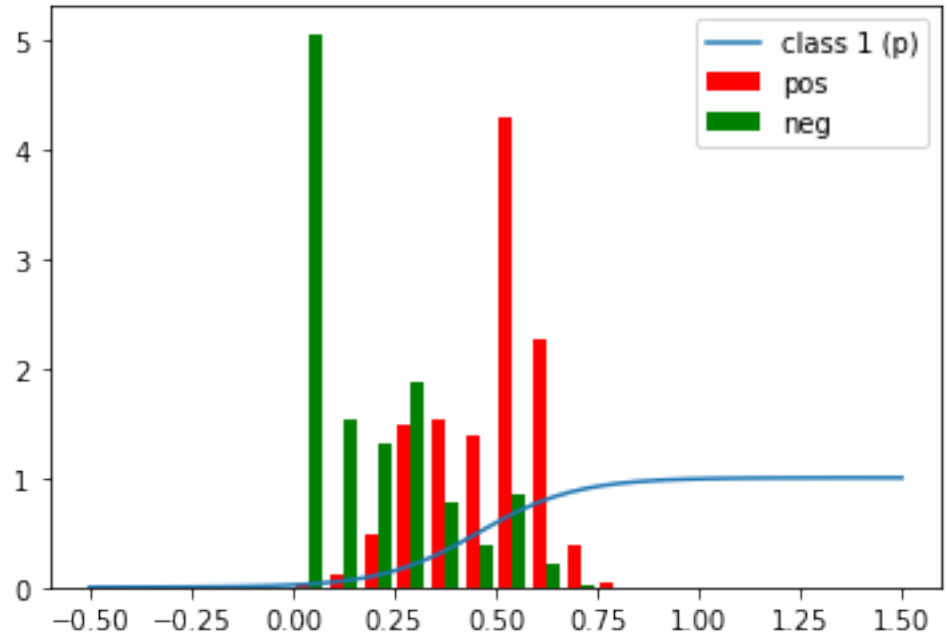
V druhém grafu vykreslete histogramy trénovacích dat a pravděpodobnost třídy 1 pro x od -

0.5 do 1.5, podobně jako výše u generativních klasifikátorů. Při výpočtu výstupů využijte `with torch.no_grad():`. **(1 + 6 + 9 řádků, 1 bod)**

[21]:
```
def plot_loss_acc(nb_of_epochs,data):
    plt.plot(nb_of_epochs, data[0], label="Training loss")
    plt.plot(nb_of_epochs, data[1], label="Validation accuracy")
    plt.legend(prop={'size': 10})
    plt.show()

model, loss, acc, = train_single_fea_llr(FOI, 100, 0.05, 73)
#73 bcs it has biggest rest after dividing 9121 in <1,100> and that means even␣
 ↪last minibatch will be almost full
plot_loss_acc(list(range(100)), [loss, acc])
with torch.no_grad():
    plt.plot(np.linspace(-0.5, 1.5), model.prob_class_1(np.linspace(-0.5, 1.
 ↪5)), label='class 1 (p)')
    plt.hist([train_dataset.pos[:,FOI], train_dataset.neg[:,FOI]],␣
 ↪density=True, label=['pos', 'neg'], color=['red','green'])
    plt.legend(prop={'size': 10})
    plt.show()
```





4.1 Všechny vstupní příznaky

V posledním cvičení natrénujete logistickou regresi, která využije všech sedm vstupních příznaků.

Prvním krokem je naimplementovat příslusný model. Bezostyšně zkopírujte tělo třídy `LogisticRegresion` a upravte ji tak, aby zvládala libovolný počet vstupů, využijte `torch.nn.Linear`. U výstupu metody `.forward()` dejte pozor, aby měl výstup tvar `[N]`; pravděpodobně budete potřebovat `squeeze`.

(9 řádků, 1 bod)

[15]:
```
class LogisticRegression_whole(torch.nn.Module):
    def __init__(self, features):
        super().__init__()
        self.w = torch.nn.parameter.Parameter(torch.tensor([1.0]))
        self.b = torch.nn.parameter.Parameter(torch.tensor([0.0]))
        self.h = torch.nn.Linear(features,1)

    def forward(self, x):
#         return torch.sigmoid(self.w*x + self.b)
        return torch.sigmoid(self.w*self.h(x.float())+self.b).squeeze(1)

    def prob_class_1(self, x):
        prob = self(torch.from_numpy(x))
        return prob.detach().numpy()
```

Podobně jako u jednodimenzionálni regrese implementujte funkci pro trénovaní plné logistické regrese. V ideálním případě vyfaktorujete společnou implementaci, které budete pouze předávat různá trénovací a validační data.

Zvídaví mohou zkusit `Adam`a jako optimalizátor namísto obyčejného `SGD`.

Funkci zavolejte, natrénujte model. Opět vykreslete průběh trénovací loss a validační přesnosti. Měli byste se s přesností dostat nad 90 %.

(ne víc než cca 30 řádků při kopírování, 1 bod)

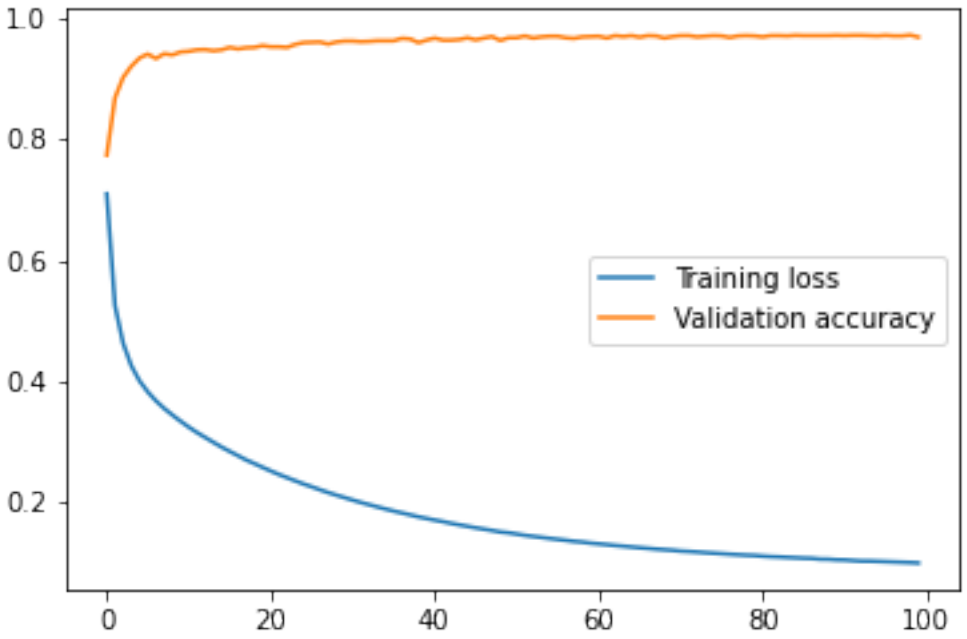
[30]:
```
def train_all(trainingData, validationData, nb_epochs, lr, batch_size):
    ''' fea_no -- which feature to train onrozhodovací
        nb_epochs -- how many times to go through the full training data
        lr -- learning rate
        batch_size -- size of minibatches
    '''
    model = LogisticRegression_whole(trainingData.xs.shape[1])
    losses = []
    accuracies = []
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)


    for x in range(0, nb_epochs):
        model_actual = batch_provider(trainingData.xs, trainingData.targets,␣
 ↪batch_size)
        los = []
        for val, tar in model_actual:
            y_pred = model(val)
            loss = F.binary_cross_entropy(y_pred, tar, reduction='mean')
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            los.append(loss.detach().numpy())

        accuracies.append(evaluate(model, validationData.xs, validationData.
 ↪targets))
        losses.append(np.average(np.array(los)))
        if(accuracies[x-1] < accuracies[x] or x == 0):
            best_model = copy.deepcopy(model)

    return best_model, losses, accuracies

model_all, loss_all, acc_all = train_all(train_dataset, val_dataset, 100, 0.05,␣
 ↪73)
#73 bcs it has biggest rest after dividing 9121 in <1,100> and that means even␣
 ↪last minibatch will be almost full
plot_loss_acc(list(range(100)), [loss_all, acc_all])
```



5 Závěrem

Konečně vyhodnoťte všech pět vytvořených klasifikátorů na testovacích datech. Stačí doplnit jejich názvy a předat jim příznaky, na které jsou zvyklé.

(0.5 bodu)

[31]:
```
xs_full = test_dataset.xs
xs_foi = test_dataset.xs[:, FOI]
targets = test_dataset.targets

print('Baseline:', evaluate(baseline, xs_foi, targets))
print('Generative classifier (w/o prior):', evaluate(classifier_flat_prior,␣
 ↪xs_foi, targets))
print('Generative classifier (correct):', evaluate(classifier_full_prior,␣
 ↪xs_foi, targets))
print('Logistic regression:', evaluate(model, xs_foi, targets))
print('Logistic regression all features:', evaluate(model_all, xs_full,␣
 ↪targets))
```

```
Baseline: 0.75
Generative classifier (w/o prior): 0.8
Generative classifier (correct): 0.847
Logistic regression: 0.8555
Logistic regression all features: 0.97
```

Blahopřejeme ke zvládnutí domácí úlohy! Notebook spusťte načisto (Kernel -> Restart & Run all), vyexportuje jako PDF a odevzdejte pojmenovaný svým loginem.

Mimochodem, vstupní data nejsou synteticky generovaná. Nasbírali jsme je z projektu; Vaše klasifikátory v této domácí úloze predikují, že daný hráč vyhraje; takže by se daly použít jako heuristika pro ohodnocování listových uzlů ve stavovém prostoru hry. Pro představu, odhadujete to z pozic pět kol před koncem partie pro daného hráče. Poskytnuté příznaky popisují globální charakteristiky stavu hry jako je například poměr délky hranic předmětného hráče k ostatním hranicím.

[ ]: