zadani

November 8, 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: Branislav Mateáš
- Login autora: XMATEA00
- Datum vzniku: 6.11.2020

[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`).

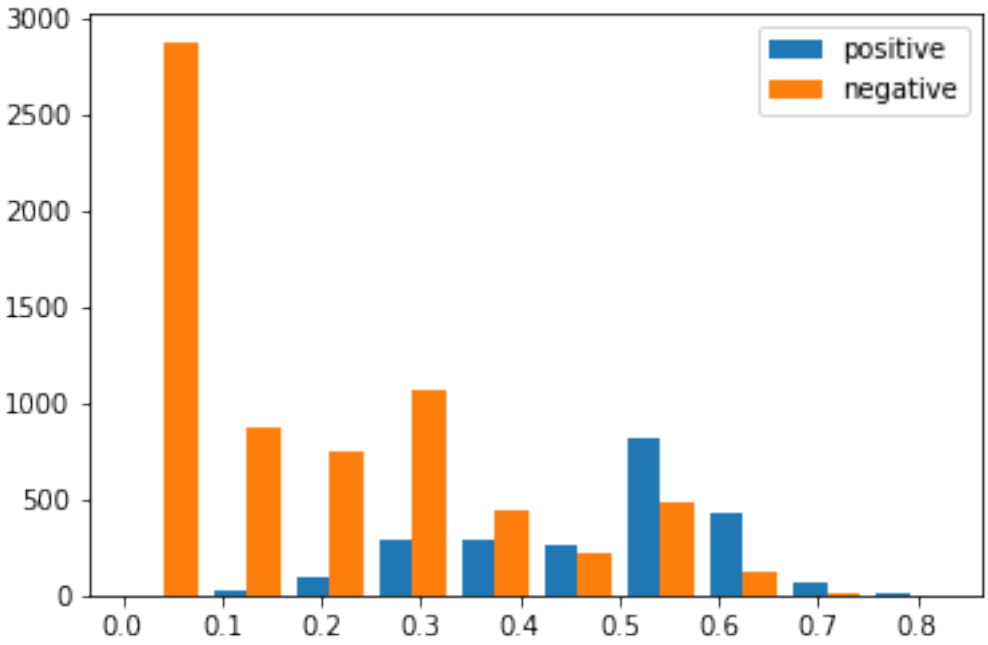
(6+3 řádků, 1 bod)

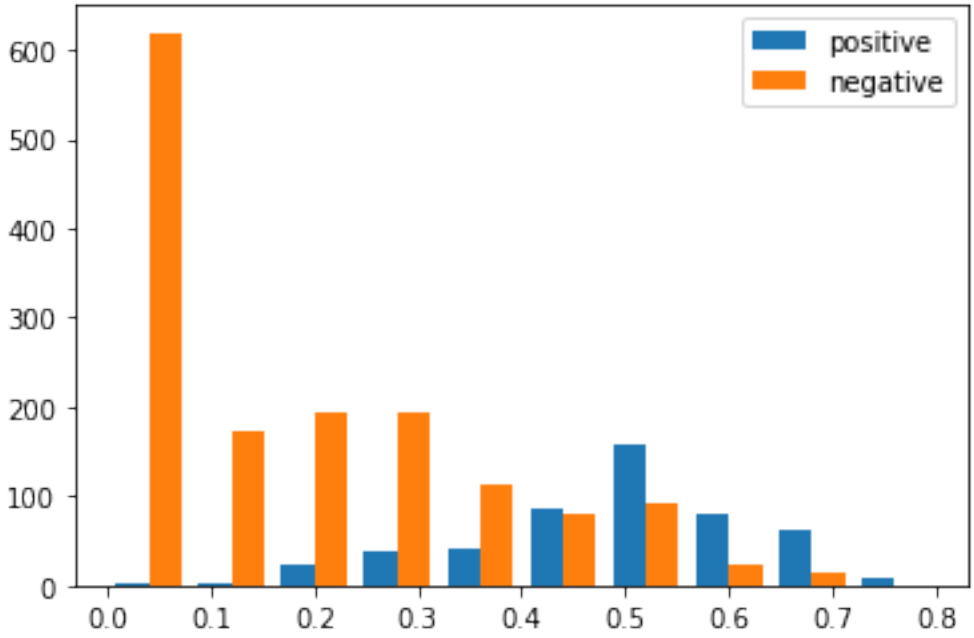
[2]:
```
class GetData:
    def __init__(self, pos, neg):
        self.pos = np.loadtxt(pos)
        self.neg = np.loadtxt(neg)
        self.xs = np.concatenate((self.pos,self.neg))
        self.targets = np.concatenate((np.ones(len(self.pos), dtype = int), np.
 ↪zeros(len(self.neg), dtype=int)))


train_dataset = GetData('./positives.trn','./negatives.trn')
val_dataset = GetData('./positives.val','./negatives.val')
test_dataset = GetData('./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):
    plt.hist([poss,negs],label=['positive','negative'])
    plt.legend()
    plt.show()

plot_data(train_dataset.pos[:, FOI], train_dataset.neg[:, FOI])
plot_data(val_dataset.pos[:, FOI], val_dataset.neg[:, FOI])
```





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):
    data = np.where(classifier.prob_class_1(inputs) > 0.5, 1, 0)
    result = np.equal(data, targets)
    return np.sum(result)/len(data)

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, val):
        self.val = val
    def prob_class_1(self, data):
        return np.full(len(data), self.val, dtype=int)

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)

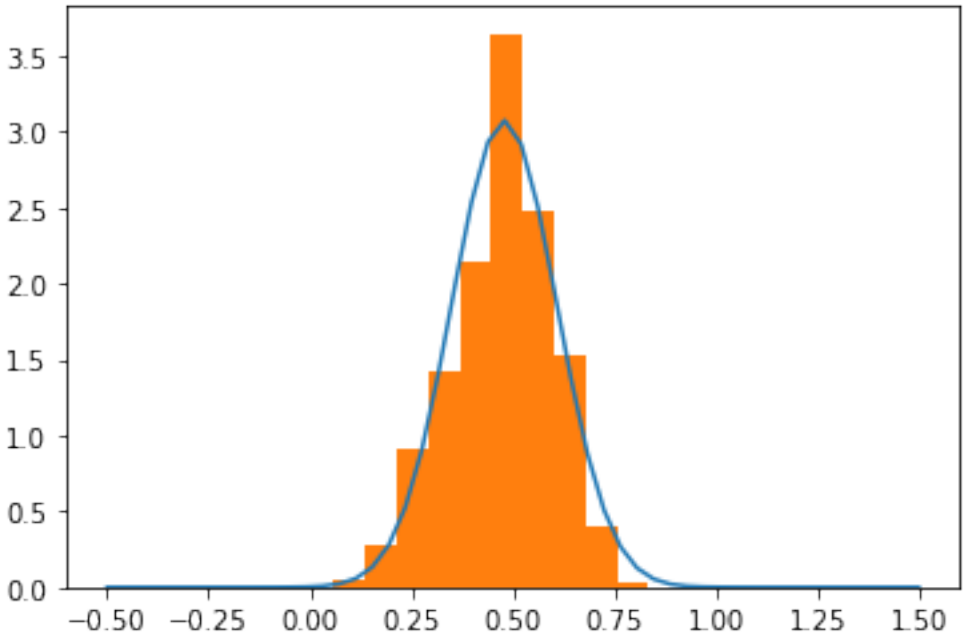
[6]:
```
def MLE(data):
    return (np.mean(data),np.std(data))

mean, std = MLE(train_dataset.pos[:, FOI])
print('POS examples: mean({})\t deviation({})'.format(mean, std))
mean, std = MLE(train_dataset.neg[:, FOI])
print('NEG examples: mean({})\t deviation({})'.format(mean, std))
```

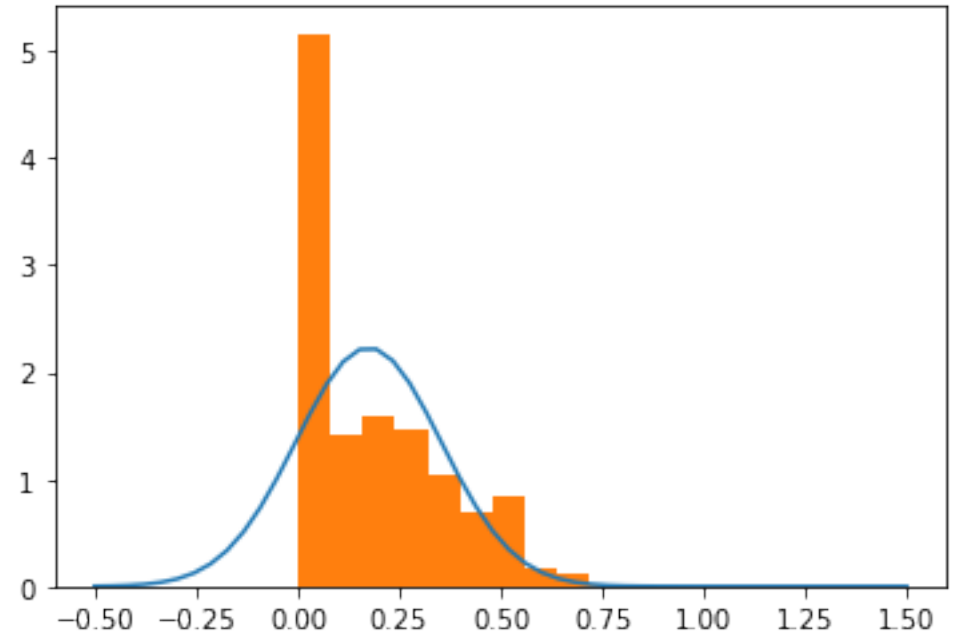
```
POS examples: mean(0.478428821613158)    deviation(0.12971703647258465)
NEG examples: mean(0.17453641132613792)  deviation(0.17895975196381242)
```

Ze získaných parametrů vytvořte scipyovská 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]:
```
pos_mean, pos_std = MLE(train_dataset.pos[:, FOI])
neg_mean, neg_std = MLE(train_dataset.neg[:, FOI])

x = np.linspace(-0.5, 1.5)
plt.plot(x, scipy.stats.norm.pdf(x,pos_mean, pos_std))
plt.hist(train_dataset.pos[:, FOI], density=True)
plt.show()
plt.plot(x, scipy.stats.norm.pdf(x,neg_mean, neg_std))
plt.hist(train_dataset.neg[:, FOI], density=True)
plt.show()
```





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 BinaryClassifier:

    def __init__(self, norm, prior):
        self.norm = norm
        self.prior = prior
    def prob_class_1(self, data):
        P1 = self.norm[1].pdf(data) * self.prior[1]
        P2 = self.norm[0].pdf(data) * self.prior[0] + self.norm[1].pdf(data) *␣
 ↪self.prior[1]
        return P1/P2
```

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)

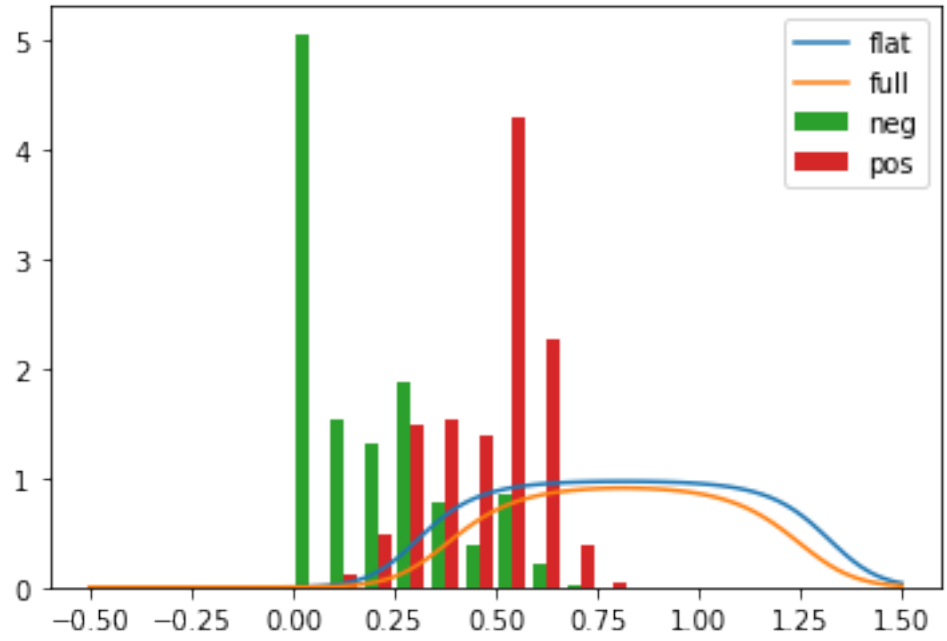
[9]:
```
classifier_flat_prior = BinaryClassifier([scipy.stats.norm(neg_mean, neg_std),␣
 ↪scipy.stats.norm(pos_mean, pos_std)], [0.5, 0.5])
classifier_full_prior = BinaryClassifier([scipy.stats.norm(neg_mean, neg_std),␣
 ↪scipy.stats.norm(pos_mean, pos_std)], [0.75, 0.25])

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]:
```
x = np.linspace(-0.5, 1.5, 100)
plt.plot(x, classifier_flat_prior.prob_class_1(x))
plt.plot(x, classifier_full_prior.prob_class_1(x))
plt.hist((train_dataset.neg[:, FOI], train_dataset.pos[:, FOI]), density=True)
plt.legend(["flat", "full", "neg", "pos"])
plt.show()
```



Interpretujte, přímo v této textové buňce, každou rozhodovací hranici, která je v grafu patrná **(3 věty, 2 body)**: Rozhodovacie hranice sú pri pravdepodobnosti 0.5

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):
    data = np.arange(xs.shape[0])
    data = np.random.permutation(data)
    i = 0
    for i in range(0, xs.shape[0] - batch_size +1, batch_size):
        idx = data[i:i+batch_size]
        yield torch.Tensor(xs[idx]).float(), torch.Tensor(targets[idx]).float()
    if xs.shape[0] % batch_size != 0:
        idx = data[i:i+1]
        yield torch.Tensor(xs[idx]).float(), torch.Tensor(targets[idx]).float()

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

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
```

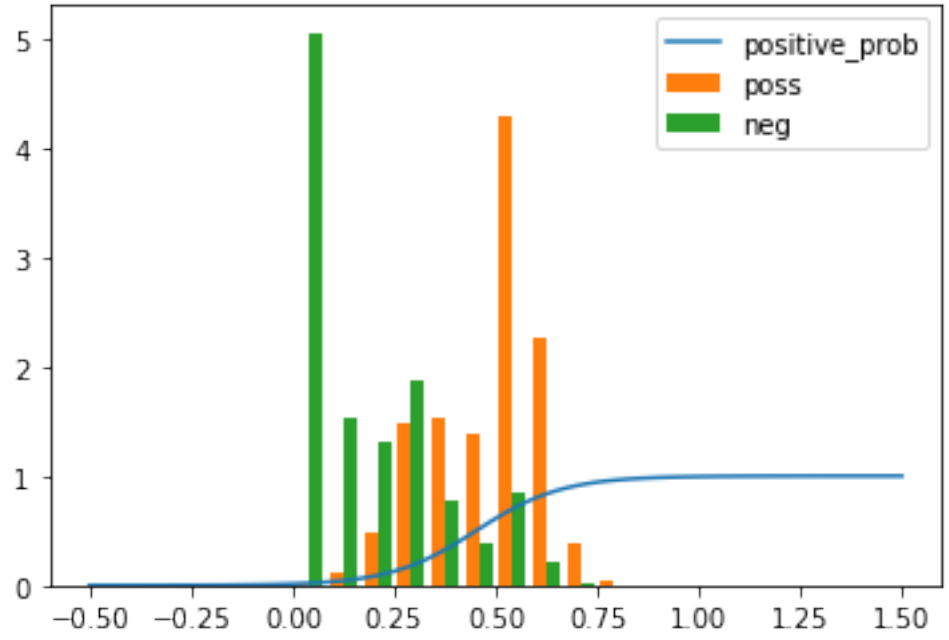
```
        batch_size -- size of minibatches
    '''
    model = LogisticRegression()
    best_model = copy.deepcopy(model)
    losses = []
    accuracies = []
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    ref_acc = 0
    for epoch in range(int(nb_epochs)):
        s_loss = []
        s_acc = []
        for x, t in batch_provider(train_dataset.xs[:,fea_no], train_dataset.
↪targets, batch_size):
            optimizer.zero_grad()
            output = model(x)
            loss = F.binary_cross_entropy(output, t)
            loss.backward()
            optimizer.step()
            s_loss.append(loss.item())
            val = evaluate(model, val_dataset.xs[:,fea_no], val_dataset.targets)
            s_acc.append(val)
        avg_loss = np.asarray(s_loss).sum()/len(s_loss)
        avg_acc = np.asarray(s_acc).sum()/len(s_acc)
        losses.append(avg_loss)
        accuracies.append(avg_acc)
        model.forward(avg_acc)
        if avg_acc > ref_acc:
            ref_acc = avg_acc
            best_model = copy.deepcopy(model)


    return best_model, losses, accuracies
```

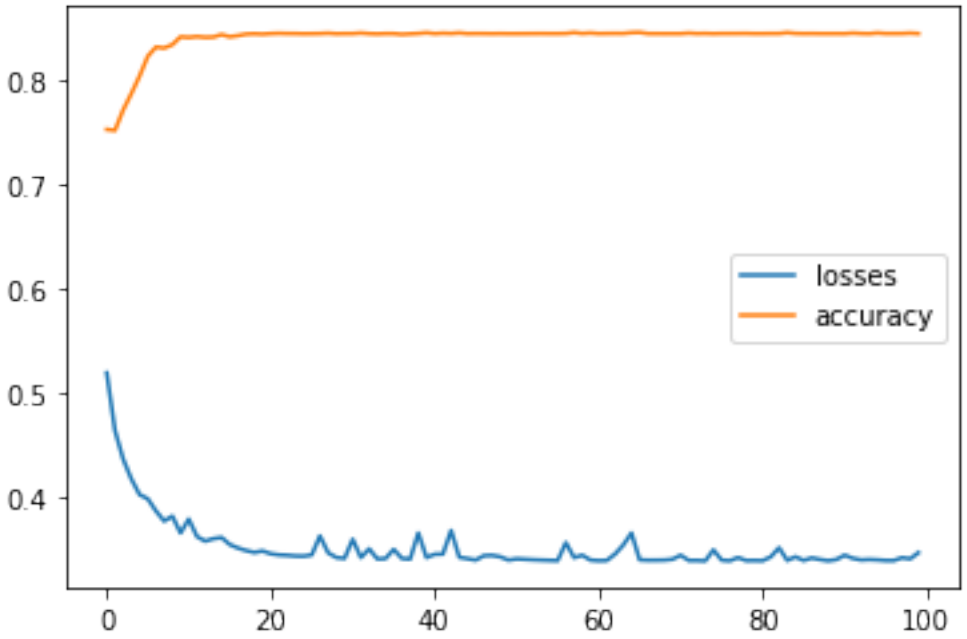
Funkci zavolejte a natrénujte model. Uveďte zde parametry, které vám dají slušný výsledek. Měli byste dostat přesnost srovnatelnou s generativním klasifikátorem s nastavenými priory. Neměli byste potřebovat víc než 100 epoch. Vykreslete průběh trénovací loss a validační přesnosti, osu x značte v epochách.

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)**

[18]:
```
b_size = 100
bt, l, a = train_single_fea_llr(FOI, 100, 0.23, b_size)

x = np.linspace(-0.5,1.5, 100)
with torch.no_grad():
```

```
    data = bt.prob_class_1(x)
plt.plot(x, data)
plt.hist((train_dataset.pos[:,FOI], train_dataset.neg[:,FOI]), density=True)
plt.legend(["positive_prob", "poss", "neg"])
plt.show()
plt.plot(l)
plt.plot(a)
plt.legend(["losses", "accuracy"])
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]:
```
pass
```

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 `Adama` 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)

[16]:
```
pass
```

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)

[17]:
```
xs_full = test_dataset.xs
xs_foi = test_dataset.xs[:, FOI]
targets = test_dataset.targets

print('Baseline:', evaluate(PriorClassifier(0), 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(bt, xs_foi, targets))
#print('logistic regression all features:', evaluate(None, None, targets))
```

```
Baseline: 0.75
Generative classifier (w/o prior): 0.8
Generative classifier (correct): 0.847
Logistic regression: 0.855
```

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.

[ ]:
```

```