P1

February 17, 2021

0.1 P1. Robust Regression

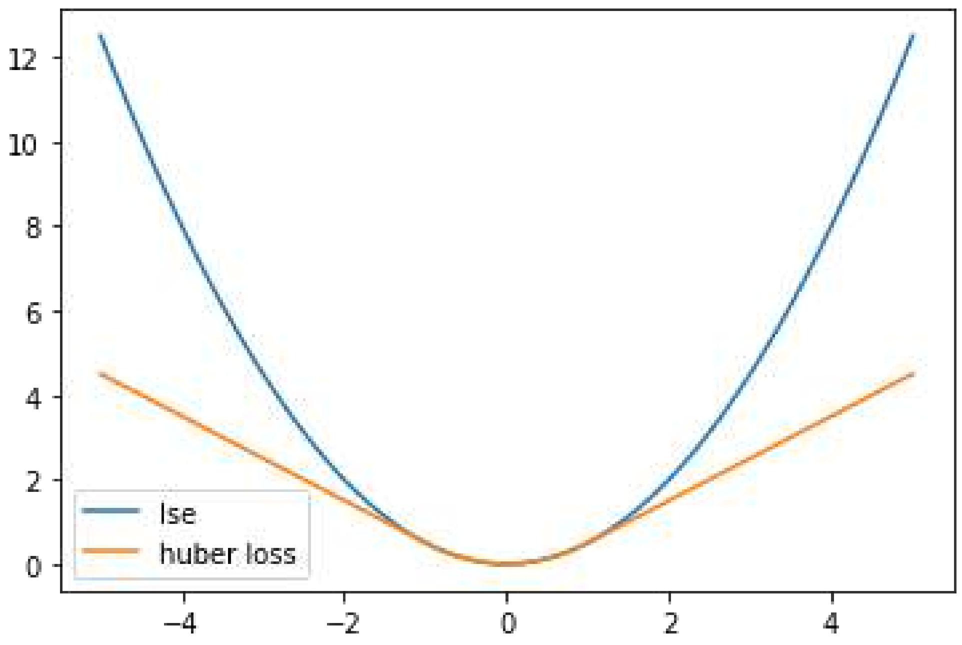
0.2 a)

[2]:
```
from matplotlib import pyplot as plt
import numpy as np
```

[36]:
```
def huber_loss(delta, y, t):
    a = y - t

    mask_le = (abs(a) <= delta).astype(int)
    mask_g = 1 - mask_le

    return 0.5 * a ** 2 * mask_le + delta * (abs(a) - 0.5 * delta) * mask_g
```

[37]:
```
lse = lambda y, t: 0.5 * (y - t) ** 2
```

[39]:
```
y = np.linspace(-5, 5, 1001)
t = 0

plt.plot(y, lse(y, 0), label='lse')
plt.plot(y, huber_loss(1, y, t), label='huber loss')
plt.legend()
plt.show()
```

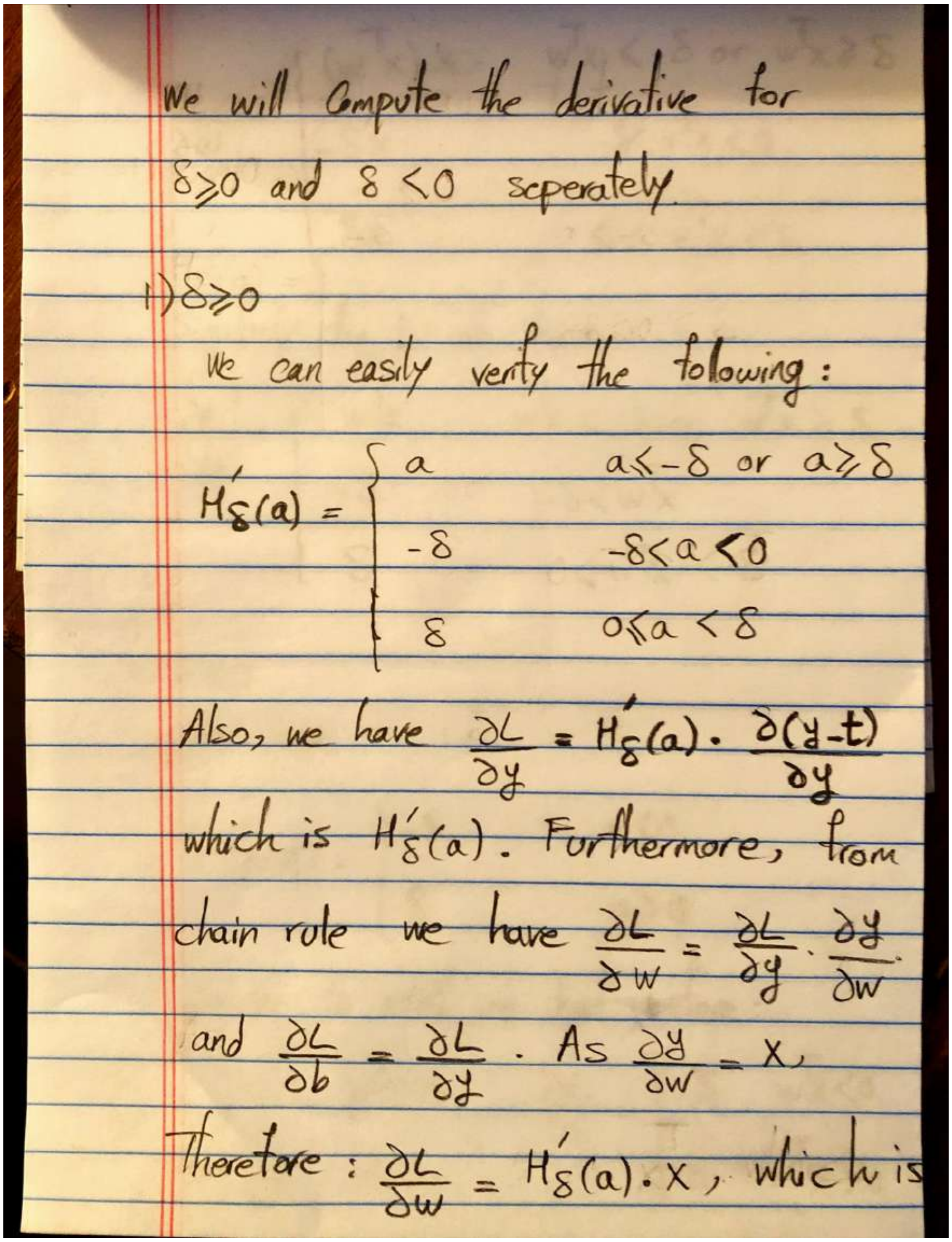


As we can see, huber loss does not penalize outliers as severely as least squarred error (It acts linearly not quadratically in those regimes). Therefore it is less sensitive to outliers.

b)

```
[3]: from IPython.display import Image
     Image(filename='image/1-1.jpg', height=600, width=600)
```

```
[3]:
```



We will Compute the derivative for $\delta \geq 0$ and $\delta < 0$ seperately.

1) $\delta \geq 0$

We can easily verify the following:

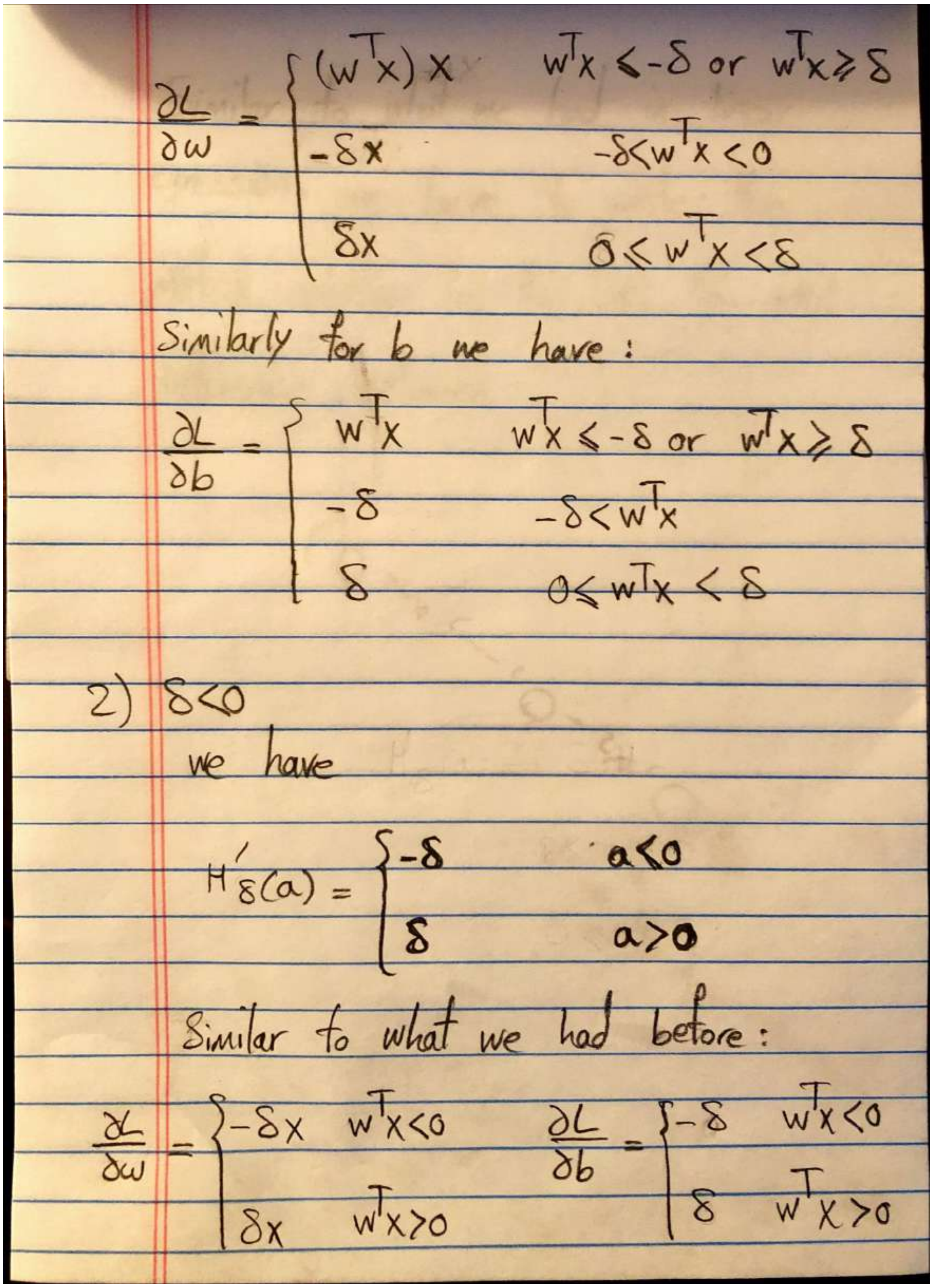
$$H'_\delta(a) = \begin{cases} a & a \leq -\delta \text{ or } a \geq \delta \\ -\delta & -\delta < a < 0 \\ \delta & 0 \leq a < \delta \end{cases}$$

Also, we have $\frac{\partial L}{\partial y} = H'_\delta(a) \cdot \frac{\partial(y-t)}{\partial y}$ which is $H'_\delta(a)$. Furthermore, from chain rule we have $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}$ and $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$. As $\frac{\partial y}{\partial w} = x$,

Therefore: $\frac{\partial L}{\partial w} = H'_\delta(a) \cdot x$, which is

```
[4]: from IPython.display import Image
     Image(filename='image/1-2.jpg', height=600, width=600)
```

[4]:



$$\frac{\partial L}{\partial w} = \begin{cases} (w^T x)\,x & w^T x \leqslant -\delta \text{ or } w^T x \geqslant \delta \\ -\delta x & -\delta < w^T x < 0 \\ \delta x & 0 \leqslant w^T x < \delta \end{cases}$$

Similarly for b we have:

$$\frac{\partial L}{\partial b} = \begin{cases} w^T x & w^T x \leqslant -\delta \text{ or } w^T x \geqslant \delta \\ -\delta & -\delta < w^T x \\ \delta & 0 \leqslant w^T x < \delta \end{cases}$$

2) $\delta < 0$

we have

$$H'_\delta(a) = \begin{cases} -\delta & a < 0 \\ \delta & a > 0 \end{cases}$$

Similar to what we had before:

$$\frac{\partial L}{\partial w} = \begin{cases} -\delta x & w^T x < 0 \\ \delta x & w^T x > 0 \end{cases} \qquad \frac{\partial L}{\partial b} = \begin{cases} -\delta & w^T x < 0 \\ \delta & w^T x > 0 \end{cases}$$

P2

February 17, 2021

0.1 P2. Locally Weighted Regression

Similar to what we had in ordinary linear regression, we can write the matrix notation of loss and take the derivative of it w.r.t. the weights w. Note that we have:

$$\frac{1}{2}\sum_{i=1}^{n} a^{(i)}(y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2 + \frac{\lambda}{2}||\mathbf{w}||^2$$
$$= (y - \mathbf{X}w)A(y - \mathbf{X}w)^T + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

We can simplify the first term as the following:

$$= (y - \mathbf{X}\mathbf{w})^T A(y - \mathbf{X}\mathbf{w})$$
$$= (y^T - \mathbf{w}^T\mathbf{X}^T)A(y - \mathbf{X}\mathbf{w})$$
$$= (y^T Ay - \mathbf{w}^T\mathbf{X}^T Ay - y^T A\mathbf{X}\mathbf{w} + \mathbf{w}^T\mathbf{X}^T A\mathbf{X}\mathbf{w})$$

Note that $\mathbf{w}^T\mathbf{X}^T Ay$ and $y^T A\mathbf{X}\mathbf{w}$ are 1×1 matrices and transpose of each other. Therefore they are equal. Thus we can write $\mathbf{w}^*$ as following:

$$\mathbf{w}^* = argmin\frac{1}{2}(y^T Ay - 2\mathbf{w}^T\mathbf{X}^T Ay + \mathbf{w}^T\mathbf{X}^T A\mathbf{X}\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

.

To solve it, we can take the derivative w.r.t. $\mathbf{w}$ and set it equal to zero:

$$\mathbf{X}^T A\mathbf{X}\mathbf{w} + \lambda\mathbf{w} = \mathbf{X}^T Ay$$

Thus:

$$(\mathbf{X}^T A\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = \mathbf{X}^T Ay$$

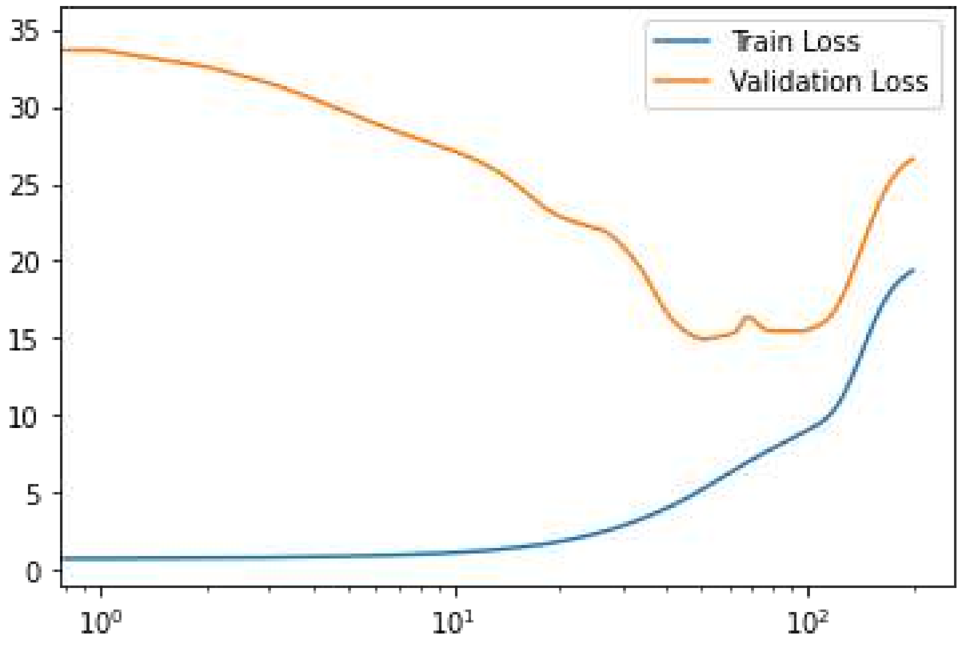
We can easily multiply both sides by $(\mathbf{X}^T A\mathbf{X} + \lambda\mathbf{I})^{-1}$ and get the formula for $\mathbf{w}^*$.

Note: I have copied the code of q2.py here as well.

[1]:
```
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 12 20:39:09 2017
```

```
"""
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_boston
from scipy.special import logsumexp
from tqdm import tqdm

np.random.seed(0)

# load boston housing prices dataset
boston = load_boston()
x = boston['data']
N = x.shape[0]
x = np.concatenate((np.ones((506,1)),x),axis=1) #add constant one feature - no␣
 ↪bias needed
d = x.shape[1]
y = boston['target']

idx = np.random.permutation(range(N))

#helper function
def l2(A,B):
    '''
    Input: A is a Nxd matrix
           B is a Mxd matirx
    Output: dist is a NxM matrix where dist[i,j] is the square norm between A[i,:
 ↪] and B[j,:]
    i.e. dist[i,j] = ||A[i,:]-B[j,:]||^2
    '''
    A_norm = (A**2).sum(axis=1).reshape(A.shape[0],1)
    B_norm = (B**2).sum(axis=1).reshape(1,B.shape[0])
    dist = A_norm+B_norm-2*A.dot(B.transpose())
    return dist


def LRLS(test_datum,x_train,y_train, tau,lam=1e-5):
    '''
    Input: test_datum is a dx1 test vector
           x_train is the N_train x d design matrix
           y_train is the N_train x 1 targets vector
           tau is the local reweighting parameter
           lam is the regularization parameter
    output is y_hat the prediction on test_datum
    '''
    ## TODO
    n_train = x_train.shape[0]

    a = -l2(test_datum.reshape(1, -1), x_train) / (2 * tau ** 2)  # shape: (1, N)
    a -= np.max(a)  # shape: (1, N)
    a = np.exp(a - logsumexp(a))  # shape: (1, N)

    A = np.eye(n_train) * a  # shape: (N, N)

    w_star = np.linalg.inv(x_train.T @ A @ x_train + lam * np.eye(d)) @ x_train.
 ↪T @ A @ y_train

    return test_datum.T @ w_star
    ## TODO


def run_validation(x,y,taus,val_frac):
    '''
    Input: x is the N x d design matrix
           y is the N x 1 targets vector
           taus is a vector of tau values to evaluate
           val_frac is the fraction of examples to use as validation data
    output is
           a vector of training losses, one for each tau value
           a vector of validation losses, one for each tau value
    '''
    ## TODO
    val_idx, train_idx = idx[:int(val_frac * N)], idx[int(val_frac * N):]
    x_val, y_val = x[val_idx], y[val_idx]
    x_train, y_train = x[train_idx], y[train_idx]

    losses_train = []
    losses_val = []

    with tqdm(total=len(taus)) as pbar:
        for tau in taus:
            loss_val = 0.
            for test_datum, t in zip(x_val, y_val):
                y_hat = LRLS(test_datum, x_train, y_train, tau)
                loss_val += (y_hat - t) ** 2
            losses_val.append(loss_val/len(x_val))

            loss_train = 0.
            for test_datum, t in zip(x_train, y_train):
                y_hat = LRLS(test_datum, x_train, y_train, tau)
                loss_train += (y_hat - t) ** 2
            losses_train.append(loss_train/len(x_train))
            if tau == 10.:
                print('Validation loss:', loss_val)
                print('Train loss:', loss_train)

            pbar.update(1)

    return losses_train, losses_val
    ## TODO



# In this excersice we fixed lambda (hard coded to 1e-5) and only set tau value.␣
 ↪Feel free to play with lambda as well if you wish
taus = np.logspace(1.0,3,200)
train_losses, test_losses = run_validation(x,y,taus,val_frac=0.3)
plt.semilogx(train_losses, label='Train Loss')
plt.semilogx(test_losses, label='Validation Loss')
plt.legend()
plt.show()
```

```
  0%|          | 1/200 [00:00<02:52,  1.16it/s]

Validation loss: 5269.391638980311
Train loss: 233.33631815494945

100%|     | 200/200 [03:08<00:00,  1.06it/s]
```



For $\tau = 10$, validation loss is 5269.39 and training loss is 233.33.

As the plot shows, the training loss curve first comes down and then goes up. This stems from the fact that for very small values of τ, we have overfitting since we just consider only close neighbors to a given data point (weights of closer points are much bigger than the weights of farther points). When τ is very big, we have underfitting since the weights of different data points (far and close) become approximately equal. This setting is analogus to the ordinary linear regression where we have underfitting and cannot capture local linearities. However, somewhere in between we will have the best results where the model neither overfits nor underfits.

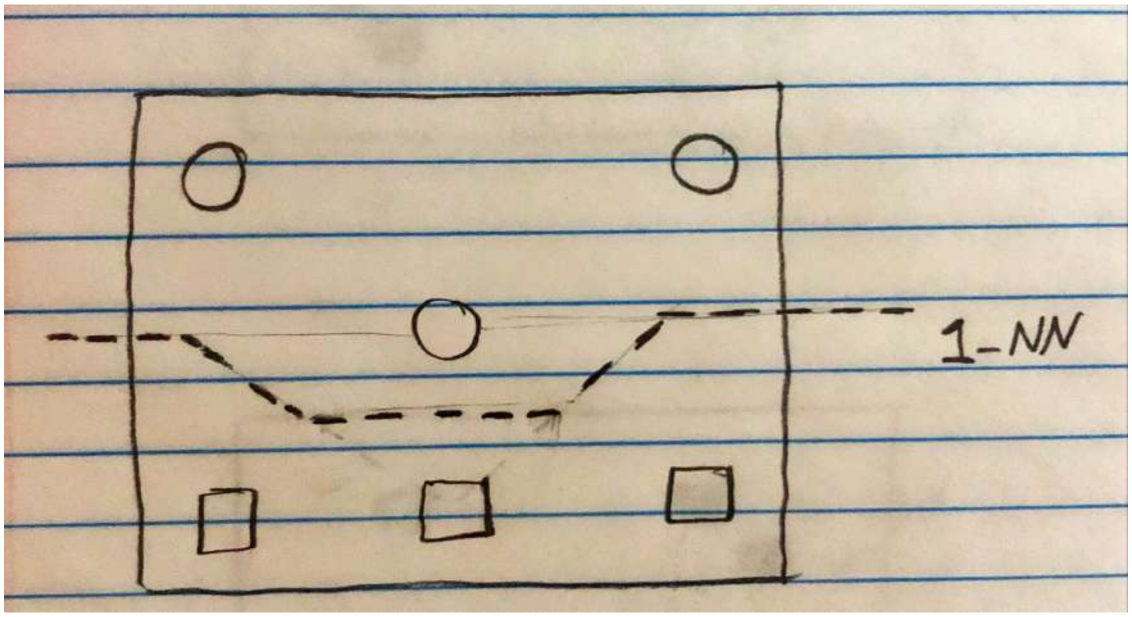
P3

February 17, 2021

0.1 P3. Decision Boundaries

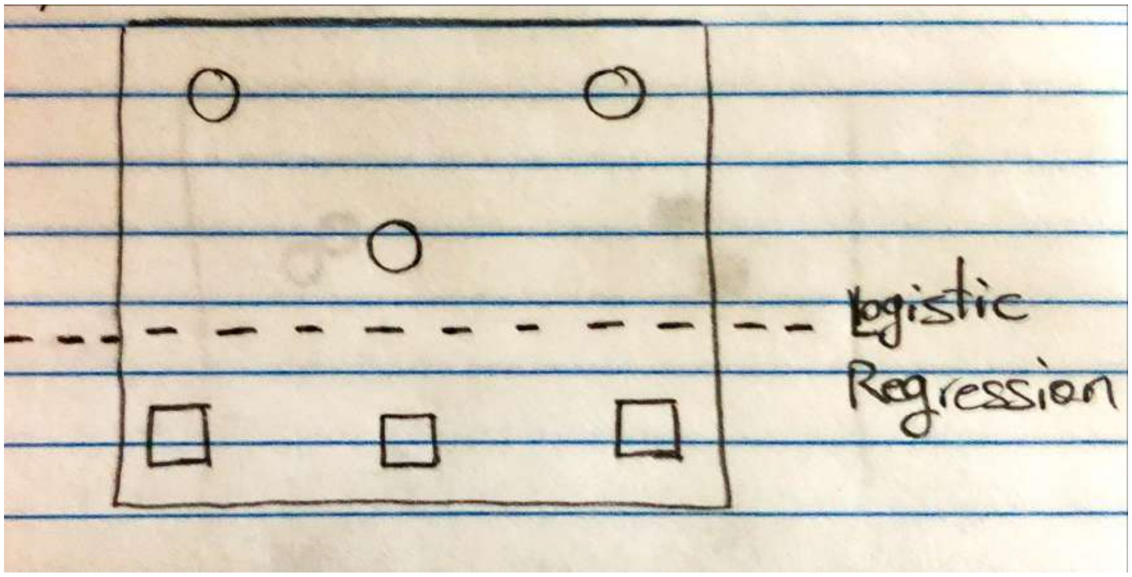
[4]:
```
from IPython.display import Image
Image(filename='image/3a-knn.jpg')
```

[4]:



[3]:
```
from IPython.display import Image
Image(filename='image/3a-log.jpg')
```

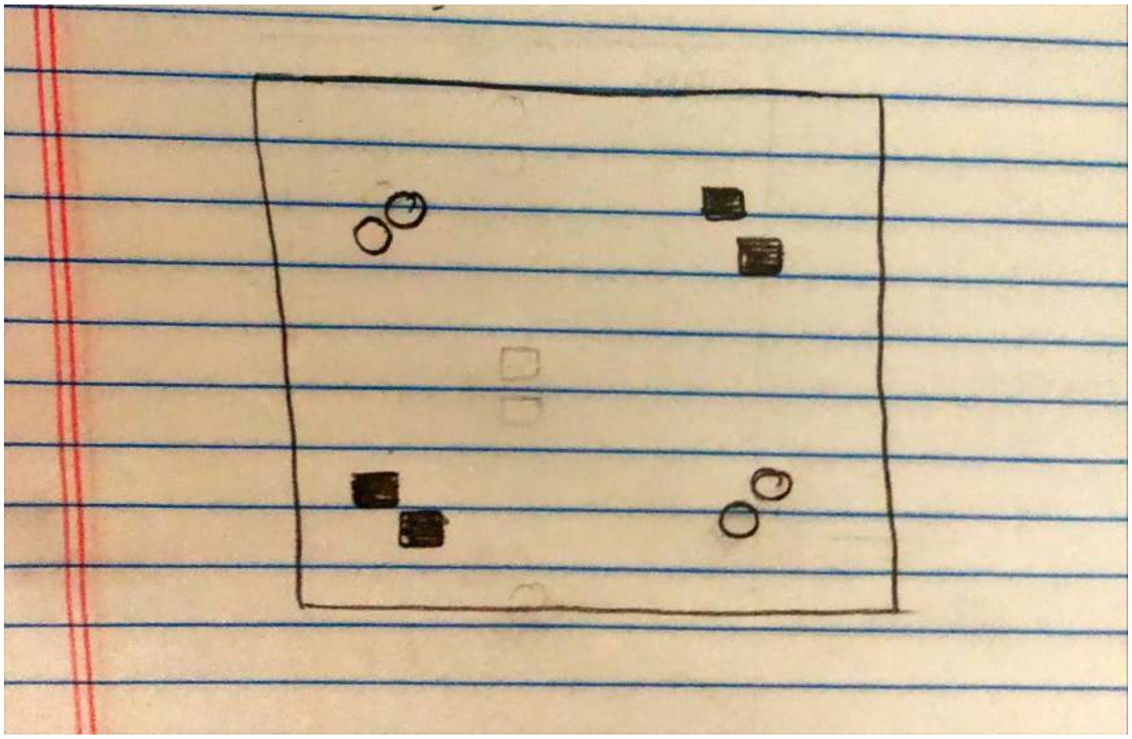
[3]:



b) Take the following example:

[2]:
```python
from IPython.display import Image
Image(filename='image/3b1.jpg')
```

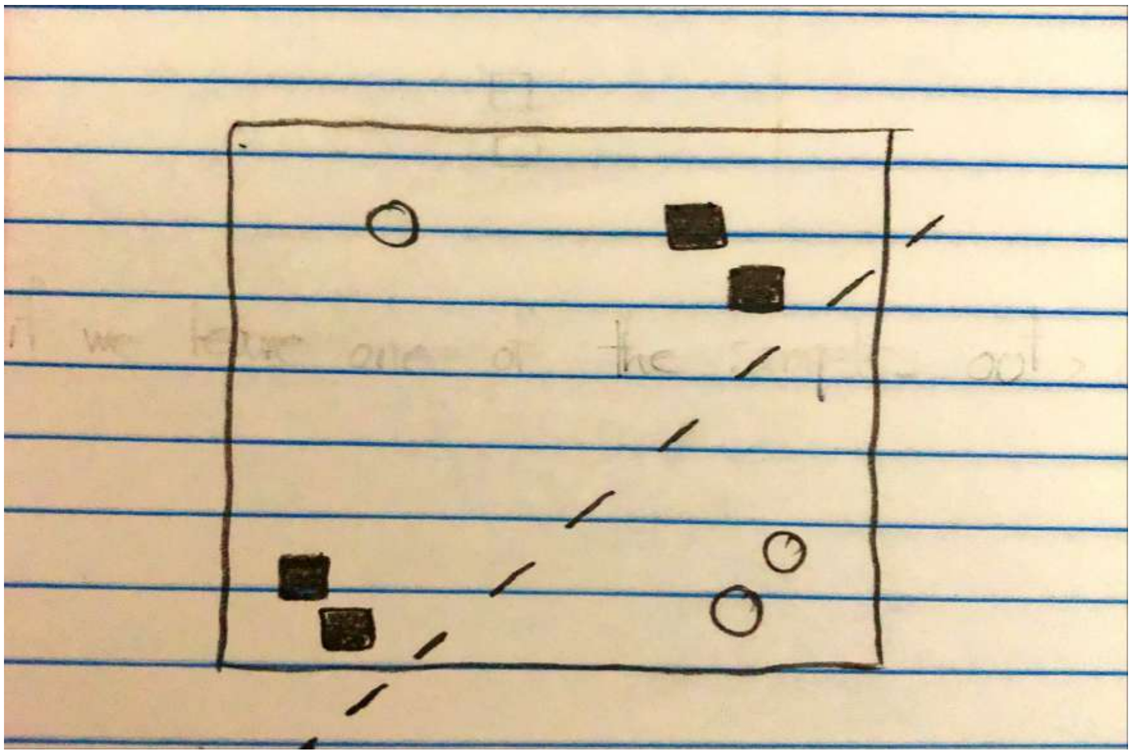
[2]:



These data points are not seperable with a line. Even if we left out one of the points they are again not seperable. For logistic regression, the best model that can be trained on the 7 points would be the following where only one of the points is misclassified:

[1]:
```python
from IPython.display import Image
Image(filename='image/3b2.jpg')
```

[1]:



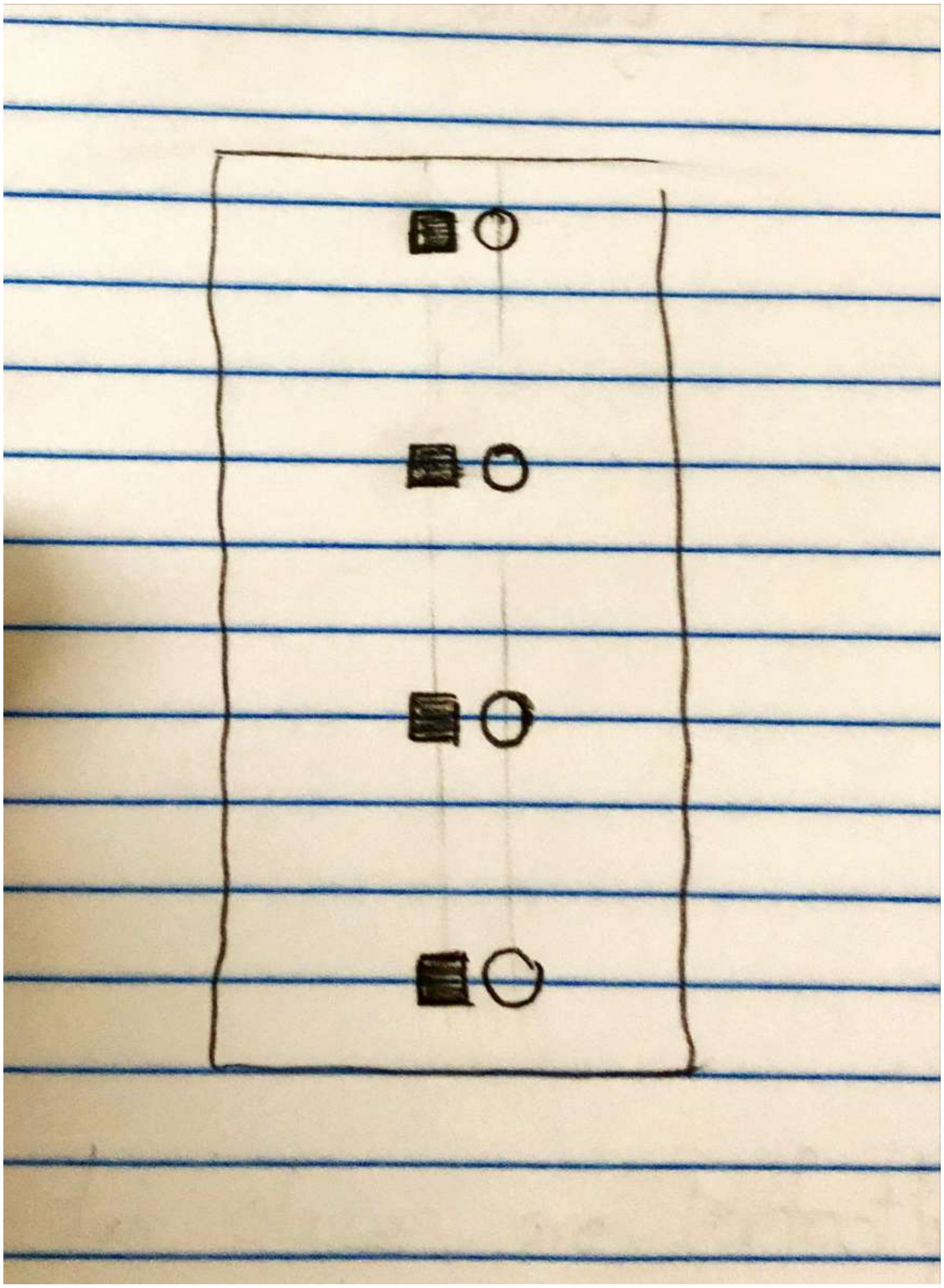
In fact, with this classifier, every point on the left side of the line will be classified a square, and every point on the right side would be classified a circle. Note that the model's prediction on the validation set, which is the removed circle, would be wrong as it will be classified a square. Therefore the loss is 1 for every model we train (Note that I am taking 0-1 loss here in order to be able to compare logistic regression with KNN). One can easily check that the same argument holds for other points as well if we take them as the left-out validation point.

However, for KNN, as the removed point is always closer to a point with the same class, the prediction would alway be correct and the loss is zero. Therefore all of the KNN models would perform than the logistic regression models.

c) Take the following example:

[5]:
```python
from IPython.display import Image
Image(filename='image/3c.jpg')
```

[5]:



In this caes the data points are perfectly seperable with a line, even if we remove one of the points. Furthermore, the left-out data piont would be classified correctly. Thus, the loss for all of the models would be zero.

However, in the case of KNN, as the closest point to the removed data point comes from the its opposite class the loss would be one.