HOG-1

a) Names Email Address

Qi Wan jhun0324@berkeley.edu

Description of Team: Best Group Ever

How did I work?

Comments:

b) I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

Qingyang Zhao

HW09 - 02

a) $R(f(x)|x) = \sum_{i=1}^{C} L(f(x), i) P(Y=i|x)$

Recall $L(f(x), i) = \begin{cases} 0, & f(x)=y \quad f(x) \in \{1 \cdots C\} \\ \lambda_s, & f(x) \neq y \quad f(x) \in \{1 \cdots C\} \\ \lambda_r, & f(x) = C+1 \end{cases}$

objective: minimize $R(f(x)|x)$, $f(x) \in \{1, \cdots, C+1\}$

Say, ① for $f(x)=y \in \{1, \cdots, C\}$ $R(f(x)|x) = \sum_{i \neq y} L(f(x), i) P(Y=i|x) + L(f(x), y) P(Y=y|x)$

$R(f(x)|x) = \lambda_s \sum_{i \neq y} P(Y=i|x) + 0 \cdot P(Y=y|x)$

$= \lambda_s (1 - P(Y=y|x))$

② for $f(x)=y=C+1$ $R(f(x)|x) = \sum_{i \neq C+1} L(f(x), i) P(Y=i|x)$ $L(C+1, i) = \lambda_r$

$= \lambda_r \sum_{i \neq C+1} P(Y=i|x)$ $i \in \{1, \cdots, C\}$

$= \lambda_r$ doubt,

Our decision policy is to minimize Risk: ① when we choose $f(x)=C+1$, $P(Y=i|x) < 1 - \frac{\lambda_r}{\lambda_s} \Rightarrow \lambda_r < \lambda_s(1-P(Y=y|x))$ $\lambda_r = R(f(x)=C+1|x) < R(f(x)=i|x) = \lambda_s(1-P(Y=y|x))$

② when we choose $f(x)=i$, $P(Y=i|x) \geq P(Y=j|x)$ for all j, and $P(Y=i|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$

which means $\begin{cases} \lambda_s(1-P(Y=i|x)) < \lambda_s(1-P(Y=j|x)) \text{ for all } j \text{ choose } f(x)=i \\ \lambda_r \geq \lambda_s(1-P(Y=y|x)). \end{cases}$

To sum up, when $P(Y=y|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$ (not choose $C+1$ class) and $P(Y=i|x) \geq P(Y=j|x)$ for all j, choose $f(x)=i$

Decision Policy: $P(Y=y|x) < 1 - \frac{\lambda_r}{\lambda_s}$ choose doubt

gives us the minimum risk.

b) If $\lambda_r = 0$, when $P(Y=y|x) < 1$, it will always classify as a "doubt"

Intuitive explanation: loss give no penalty on doubt, the classifier will say "choose doubt, because other choice will give me penalty because of the wrong guess and non-zero probability"

when $P(Y=y|x)=1$, trivial, only one class, of course choose that class!

If $\lambda_r > \lambda_s$, result in never choose "doubt"

penalty is so big, choose doubt will not minimize the risk. (for probability is less/equal than 1)

HW09-03

2.00p.m.

a) MLE: Choose $L=1$ when $P(X|Y=1) > P(X|Y=2)$

$L=2$ when $P(X|Y=1) < P(X|Y=2)$

Decision Boundary $P(X;Y=1) = P(X;Y=2)$

$L=1$: $\frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)} > \frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2)}$

$\Sigma = UDU^T$

$\Sigma^{-1} = UD^{-1}U^T$

$= UD^{-\frac{1}{2}}U^TUD^{-\frac{1}{2}}U^T$

$= \Sigma^{-\frac{1}{2}} \cdot \Sigma^{-\frac{1}{2}}$

$a^Ta - b^Tb = (a-b)^T(a+b)$

$\frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2) - \frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) > 0$

$[\Sigma^{-\frac{1}{2}}(x-\mu_2)]^T[\Sigma^{-\frac{1}{2}}(x-\mu_2)] - [\Sigma^{-\frac{1}{2}}(x-\mu_1)]^T[\Sigma^{-\frac{1}{2}}(x-\mu_1)] > 0$

$[\Sigma^{-\frac{1}{2}}(x-\mu_2-x+\mu_1)]^T[\Sigma^{-\frac{1}{2}}(x-\mu_2+x-\mu_1)] > 0$

$(\mu_1-\mu_2)^T\Sigma^{-1}(2x-\mu_1-\mu_2) > 0$

$(\mu_1-\mu_2)^T\Sigma^{-1}x > \frac{(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1+\mu_2)}{2}$

For MLE: Decision Rule: $\begin{cases} L=1, & (\mu_1-\mu_2)^T\Sigma^{-1}X > \frac{(\mu_1-\mu_2)^T\Sigma^{-1}(\mu+\mu_2)}{2} \\ L=2, & \text{otherwise} \end{cases}$

MAP: Choose $L=1$ $P(Y=1|X) > P(Y=2|X)$

$P(X|Y=1)P(Y=1) > P(X|Y=2)P(Y=2)$

$\frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)}\pi_1 > \frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2)}\pi_2$

$-\frac{1}{2}[(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) - (x-\mu_2)^T\Sigma^{-1}(x-\mu_2)] > \ln\frac{\pi_2}{\pi_1}$

$(\mu_1-\mu_2)^T\Sigma^{-1}(2x-\mu_1-\mu_2) > 2\ln\frac{\pi_2}{\pi_1}$

$(\mu_1-\mu_2)^T\Sigma^{-1}X > \frac{2\ln\frac{\pi_2}{\pi_1} + (\mu_1-\mu_2)^T\Sigma^{-1}(\mu+\mu_2)}{2}$

For MAP: Decision Rule: $\begin{cases} L=1, & (\mu_1-\mu_2)^T\Sigma^{-1}X > \ln\frac{\pi_2}{\pi_1} + \frac{(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1+\mu_2)}{2} \\ L=2, & \text{otherwise} \end{cases}$

When $\pi_1 = \pi_2 = \frac{1}{2}$, the decision rules for MAP & MLE are the same.

Hw09-3

b)

$$\Sigma_{YY} = E[(Y - E[Y])(Y - E[Y])^T]$$

$$= E\left[\begin{bmatrix} Y_1 - \frac{1}{2} \\ Y_2 - \frac{1}{2} \end{bmatrix}\begin{bmatrix} Y_1 - \frac{1}{2} & Y_2 - \frac{1}{2} \end{bmatrix}\right] = E\begin{bmatrix} Y_1^2 + \frac{1}{4} - Y_1 & Y_1 Y_2 - \frac{1}{2}(Y_1 + Y_2) + \frac{1}{4} \\ Y_1 Y_2 - \frac{1}{2}(Y_1 + Y_2) + \frac{1}{4} & Y_2^2 + \frac{1}{4} - Y_2 \end{bmatrix}$$

$E[Y_1^2] = 1 \cdot \frac{1}{2}$ $\quad E[Y_1 Y_2] = 1 \times \frac{1}{4}$ $\quad E[Y_1] = \frac{1}{2}$ $\quad E[Y_2] = \frac{1}{2}$

$$\Sigma_{YY} = \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

$$\Sigma_{XX} = E[(X - E[X])(X - E[X])^T]$$

$$= E[XX^T] - E[X]E[X]^T$$

$$= E[E[XX^T | X]] - (\frac{\mu_1 + \mu_2}{2})(\frac{\mu_1 + \mu_2}{2})^T$$

$E[XX^T \mid X \text{ comes from } X_1] = \Sigma + \mu_1 \mu_1^T$

$E[XX^T \mid X \text{ comes from } X_2] = \Sigma + \mu_2 \mu_2^T$

$$= P(X \text{ comes from } X_1) \cdot E[XX^T \mid X \text{ comes from } X_1] + P(X \text{ comes from } X_2) E[XX^T \mid X \text{ comes from } X_2] - (\frac{\mu_1 + \mu_2}{2})(\frac{\mu_1 + \mu_2}{2})^T$$

$$= \frac{1}{2}(\Sigma + \mu_1 \mu_1^T) + \frac{1}{2}(\Sigma + \mu_2 \mu_2^T) - \frac{1}{4}(\mu_1 + \mu_2)(\mu_1 + \mu_2)^T$$

$$= \Sigma + \frac{1}{4}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

$$\Sigma_{XY} = E[E[(X - E[X])(Y - E[Y])^T]]$$

$$= E[XY^T] - E[X]E[Y]^T$$

$$= \frac{1}{2}[\mu_1 \ \mu_2] - \frac{1}{2}[\frac{\mu_1 + \mu_2}{2} \ \frac{\mu_1 + \mu_2}{2}]$$

$$= \frac{1}{4}[\mu_1 - \mu_2 \ \ \mu_2 - \mu_1]$$

$E[E[X Y_1 \mid Y_1]]$ → X from X_1

$= E[XY_1 \mid Y_1 = 1] \cdot P(Y_1 = 1)$

$= \frac{\mu_1}{2}$

$E[E[X Y_2 \mid Y_2]]$

$= E[XY_2 \mid Y_2 = 1] P(Y_2 = 1)$

$= \frac{\mu_2}{2}$

HW09-3

c) $$\max_{u,v} \rho(u^TX, v^TY) = \max_{u,v} \frac{E[u^T(X-E[X])(Y-E[Y])v]}{\left(E[u^T(X-E[X])(X-E[X])^Tu]\,E[v^T(Y-E[Y])(Y-E[Y])^Tv]\right)^{\frac{1}{2}}}$$

$$= \max_{u,v} \frac{u^T\Sigma_{XY}v}{\sqrt{u^T\Sigma_{XX}u \cdot v^T\Sigma_{YY}v}}$$

$$\Sigma_{XY} = \frac{1}{4}[\mu_1-\mu_2 \;\; \mu_2-\mu_1]$$

$$\Sigma_{XX} = \Sigma + \frac{1}{4}(\mu_1-\mu_2)(\mu_1-\mu_2)^T$$

$$\Sigma_{YY} = \frac{1}{4}\begin{bmatrix}1 & -1\\ -1 & 1\end{bmatrix}$$

$$\Sigma_{YY}^{-\frac{1}{2}} = \frac{1}{4}$$

$$\begin{bmatrix}1 & -1\\ -1 & 1\end{bmatrix} = \begin{bmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix}\begin{bmatrix}2 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix}$$

HINT: $$\left(\Sigma + \frac{1}{4}(\mu_1-\mu_2)(\mu_1-\mu_2)^T\right)^{-1} = \Sigma^{-1} - \frac{\frac{1}{4}\Sigma^{-1}(\mu_1-\mu_2)(\mu_1-\mu_2)^T\Sigma^{-1}}{1+\frac{1}{4}(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}$$

$$\max_{u,v} \rho(u^TX, v^TY) = \max_{u,v} \frac{\frac{1}{4}u^T(\mu_1-\mu_2)\begin{bmatrix}1 & -1\end{bmatrix}v}{\sqrt{u^T\Sigma_{XX}u\left(v^T\begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}1 & -1\end{bmatrix}v\right)\cdot\frac{1}{4}}}$$

$$\Leftrightarrow \max_{u} \frac{u^T(\mu_1-\mu_2)}{\sqrt{u^T\Sigma_{XX}u}}$$

set $u' = \Sigma_{XX}^{\frac{1}{2}}u$

$$\max_{u'} \frac{u'^T\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)}{\sqrt{u'^Tu'}} \Leftrightarrow \max_{\|u'\|_2=1} u'^T\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)$$

when $u' \parallel \Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)$ has maximum, $u' = \frac{\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)}{\|\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)\|_2}$

$$u' \propto \Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)$$

$$u \propto \Sigma_{XX}^{-1}(\mu_1-\mu_2)$$

$$u \propto \left(\Sigma + \frac{1}{4}(\mu_1-\mu_2)(\mu_1-\mu_2)^T\right)^{-1}(\mu_1-\mu_2) = \Sigma^{-1}(\mu_1-\mu_2) - \frac{\frac{1}{4}(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}{1+\frac{1}{4}(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}\Sigma^{-1}(\mu_1-\mu_2)$$

$$= \left(1 - \frac{\frac{1}{4}(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}{1+\frac{1}{4}(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}\right)\Sigma^{-1}(\mu_1-\mu_2)$$

$$u \propto \Sigma^{-1}(\mu_1-\mu_2)$$

$\Sigma_{XY} = E[E[(X-E[X])(Y-E[Y])^T]]$

$= E[XY^T] - E[X]E[Y]^T$

$= [\pi_1\mu_1 \ \pi_2\mu_2] - (\pi_1\mu_1+\pi_2\mu_2)[\pi_1 \ \pi_2]$

$= [\pi_1\mu_1 - (\pi_1\mu_1+\pi_2\mu_2)\pi_1 \ \ \pi_2\mu_2 - (\pi_1\mu_1+\pi_2\mu_2)\pi_2]$

$= [\pi_1(1-\pi_1)(\mu_1-\mu_2) \ \ \pi_1(1-\pi_2)(\mu_2-\mu_1)]$

$= \boxed{\pi_1(1-\pi_2)(\mu_1-\mu_2)[1 \ -1]}$

$E[X]\cdot E[Y]^T = (\pi_1\mu_1+\pi_2\mu_2)[\pi_1 \ \pi_2]$

$E[XY^T] = \begin{bmatrix} E[E[XY_1|Y_2]] \\ E[E[XY_2|Y_2]] \end{bmatrix}$

$E[E[XY_1|Y_1]] = E[X_1Y_1|Y_1=1]\cdot P(Y_1=1) + E[X_2Y_1|Y_1=0]P(Y_1=0)$

$= \pi_1\mu_1$

$E[E[XY_2|Y_2]] = E[X_2Y_2|Y_2=1]\cdot P(Y_2=1) + E[X_2Y_2|Y_2=0]P(Y_2=0)$

$= \pi_2\mu_2$

$E[XY^T] = [\pi_1\mu_1 \ \ \pi_2\mu_2]$

$\max_{u,v} \rho(u^TX, v^TY) = \max_{u,v} \frac{u^T\Sigma_{XY}v}{\sqrt{u^T\Sigma_{XX}u \ v^T\Sigma_{YY}v}}$

$\Sigma_{XX}^{-1} = \left(\Sigma + \pi_1(1-\pi_1)(\mu_1-\mu_2)(\mu_1-\mu_2)^T\right)^{-1} = \Sigma^{-1} - \frac{\pi_1\pi_2(\Sigma^{-1}(\mu_1-\mu_2))(\mu_1-\mu_2)^T\Sigma^{-1}}{1+\pi_1\pi_2(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}$

$\max_{u,v} \rho(u^TX, v^TY) = \max_{u,v} \frac{u^T\pi_1(1-\pi_1)(\mu_1-\mu_2)[1 \ -1]v}{\sqrt{u^T\Sigma_{XX}u\left(v^T\begin{bmatrix}1\\-1\end{bmatrix}[1 \ -1]v\right)\cdot\pi_1(1-\pi_1)}}$

$= \max_{u,v} \text{const} \ \frac{u^T(\mu_1-\mu_2)}{\sqrt{u^T\Sigma_{XX}u}}$

set $u' = \Sigma_{XX}^{\frac{1}{2}}u$, $\rightarrow = \max_{u'} \frac{u'\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)}{\sqrt{u'^Tu'}} = \max_{\|u'\|_2=1} u'\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)$

$u' = \frac{\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)}{\|\Sigma_{XX}^{-\frac{1}{2}}(\mu_1-\mu_2)\|_2}$ has maximum (parallel), $u \propto \Sigma_{XX}^{-1}(\mu_1-\mu_2)$

$u \propto \Sigma^{-1}(\mu_1-\mu_2) - \frac{\pi_1\pi_2(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}{1+\pi_1\pi_2(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}\Sigma^{-1}(\mu_1-\mu_2)$

$u \propto \left(1 - \frac{\pi_1\pi_2(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}{1+\pi_1\pi_2(\mu_1-\mu_2)^T\Sigma^{-1}(\mu_1-\mu_2)}\right)\Sigma^{-1}(\mu_1-\mu_2)$, $\rightarrow$ const $\rightarrow$ $\boxed{u \propto \Sigma^{-1}(\mu_1-\mu_2)}$

HW09-3

d) when $P(X \text{ comes from } X_1) = P(Y_1=1) = P(Y_2=0) = \pi_1$ $\quad E[Y_1] = \pi_1$

$P(X \text{ comes from } X_2) = P(Y_1=0) = P(Y_2=1) = \pi_2$ $\quad E[Y_2] = \pi_2$

$$\Sigma_{YY} = E[(Y-E[Y])(Y-E[Y])^T] = E\left[\begin{bmatrix} Y_1-\pi_1 \\ Y_2-\pi_2 \end{bmatrix}\begin{bmatrix} Y_1-\pi_1 & Y_2-\pi_2 \end{bmatrix}\right] = E\begin{bmatrix} Y_1^2+\pi_1^2-2Y_1\pi_1 & Y_1Y_2-\pi_1Y_2-\pi_2Y_1+\pi_1\pi_2 \\ Y_1Y_2-\pi_1Y_2-\pi_2Y_1+\pi_1\pi_2 & Y_2^2+\pi_2^2-2Y_2\pi_2 \end{bmatrix}$$

$E[Y_1^2]=\pi_1 \quad E[Y_1Y_2]=0 \quad E[Y_1]=\pi_1 \quad E[Y_2^2]=\pi_2 \quad E[Y_2]=\pi_2$

$$\Sigma_{YY} = \begin{bmatrix} \pi_1-\pi_1^2 & -\pi_1\pi_2 \\ -\pi_1\pi_2 & \pi_2-\pi_2^2 \end{bmatrix} = \begin{bmatrix} \pi_1(1-\pi_1) & -\pi_1\pi_2 \\ -\pi_1\pi_2 & \pi_2(1-\pi_2) \end{bmatrix} = \begin{bmatrix} \pi_1(1-\pi_2) & -\pi_2(1-\pi_2) \\ -\pi_1(1-\pi_1) & \pi_2(1-\pi_1) \end{bmatrix} = \boxed{\pi_1(1-\pi_1)\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}}$$

$\Sigma_{XX} = E[(X-E[X])(X-E[X])^T]$ $\quad E[X] = E[E[X|X]]$

$= E[XX^T] - E[X]E[X]^T$ $\quad = E[X_1]P(X \text{ comes from } X_1) + E[X_2]P(X \text{ comes from } X_2)$

$= \pi_1\mu_1 + \pi_2\mu_2$

$E[XX^T] = E[E[XX^T|X]]$

$= E[X_1X_1^T]P(X \text{ comes from } X_1) + E[X_2X_2^T]P(X \text{ comes from } X_2)$

$= [\Sigma + \mu_1\mu_1^T]\pi_1 + [\Sigma + \mu_2\mu_2^T]\pi_2$

$= \Sigma + \mu_1\mu_1^T\pi_1 + \mu_2\mu_2^T\pi_2$

$\Sigma_{XX} = E[XX^T] - E[X]E[X]^T$

$= \Sigma + \mu_1\mu_1^T\pi_1 + \mu_2\mu_2^T\pi_2 - (\pi_1\mu_1+\pi_2\mu_2)(\pi_1\mu_1+\pi_2\mu_2)^T$

$= \Sigma + \mu_1\mu_1^T\pi_1 + \mu_2\mu_2^T\pi_2 - \pi_1^2\mu_1\mu_1^T - \pi_1\pi_2\mu_2\mu_1^T - \pi_1\pi_2\mu_1\mu_2^T - \pi_2^2\mu_2\mu_2^T$

$= \Sigma + \mu_1\mu_1^T\pi_1(1-\pi_1) + \mu_2\mu_2^T\pi_2(1-\pi_2) - \mu_1\mu_2^T\pi_1\pi_2 - \mu_2\mu_1^T\pi_1\pi_2$

$$\boxed{= \Sigma + \pi_2(1-\pi_1)(\mu_1-\mu_2)(\mu_1-\mu_2)^T}$$

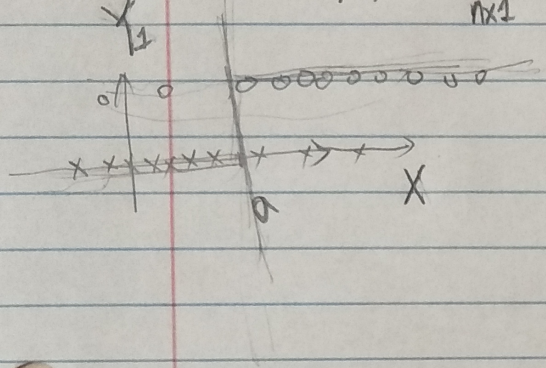
Σ_{XY} is on the next Page

HW09-3

e) First $\mu_1 = \sum_{i \in \{i \mid y_1^{(i)}=1\}} x_i$ $\mu_0 = \sum_{i \in \{i \mid y_2^{(i)}=1\}} x_i$ $\Sigma = (x-(\pi_1\mu_1+\pi_0\mu_2))(x-(\pi_1+\pi_0\mu_2))^T$

Second project X onto $\Sigma^{-1}(\mu_1-\mu_2)$

Third $X \cdot \Sigma^{-1}(\mu_1-\mu_2) = X'$ $n \times 1$



Data looks like below.

pick a which gives the smallest loss.

using step loss function.

HW09-4

a) In the pdf a) Five models

b) Second and third order Polynomial gives the best results on both training and test error

Generative model performs the worst on test data, and runs the longest time

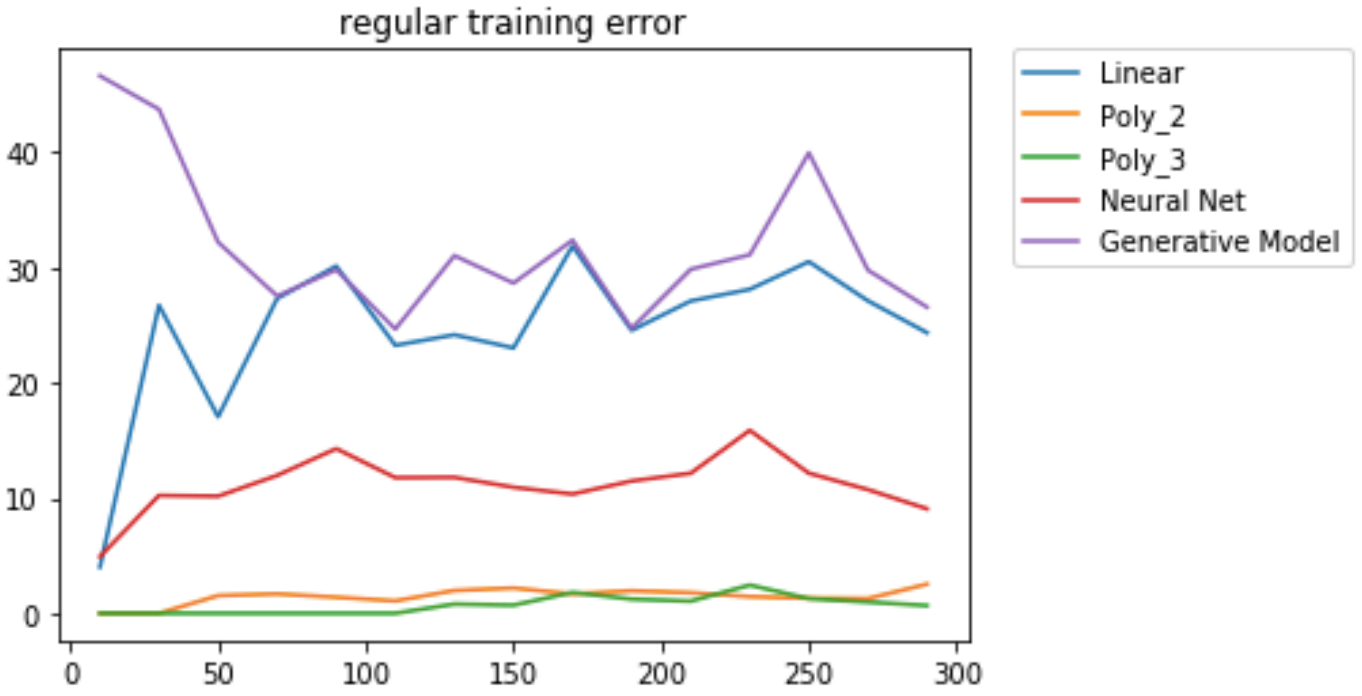
Third order Polynomial fails on shifted test data

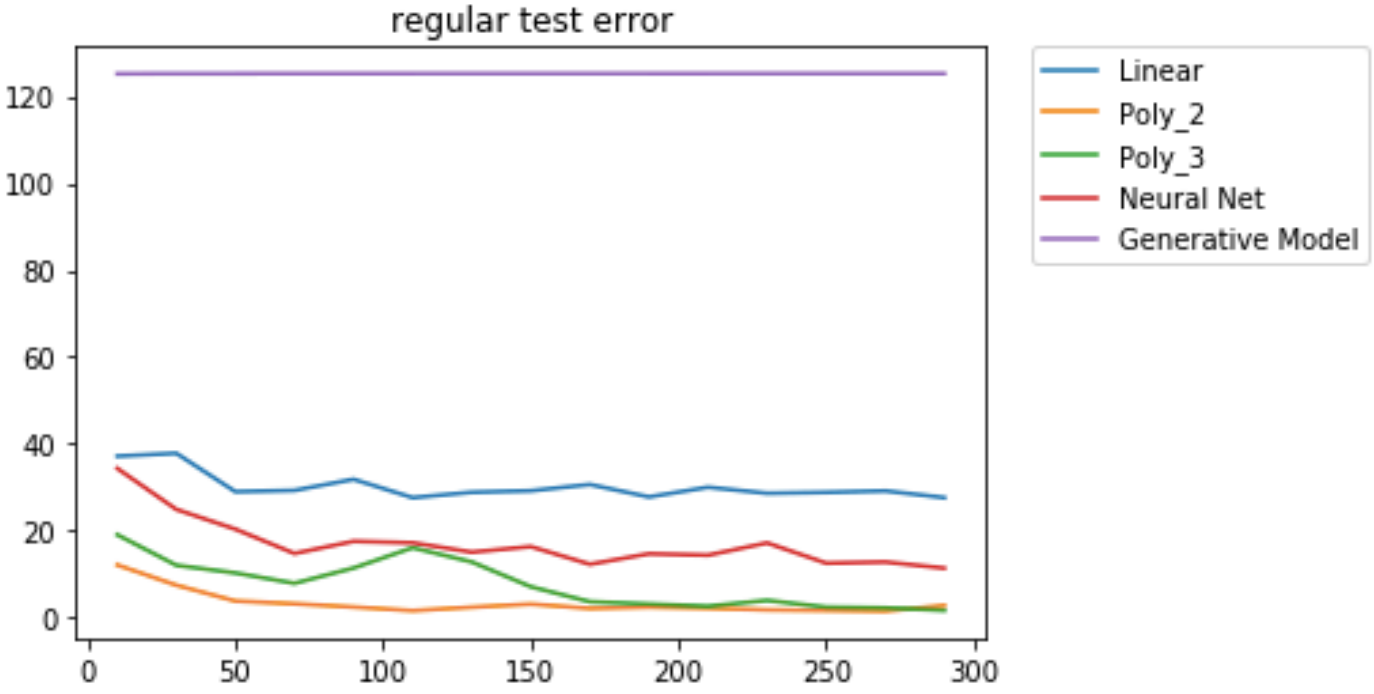
c) $\ell = 250$ gives me the smallest test error

d) $k = 3$ gives me the smallest test error

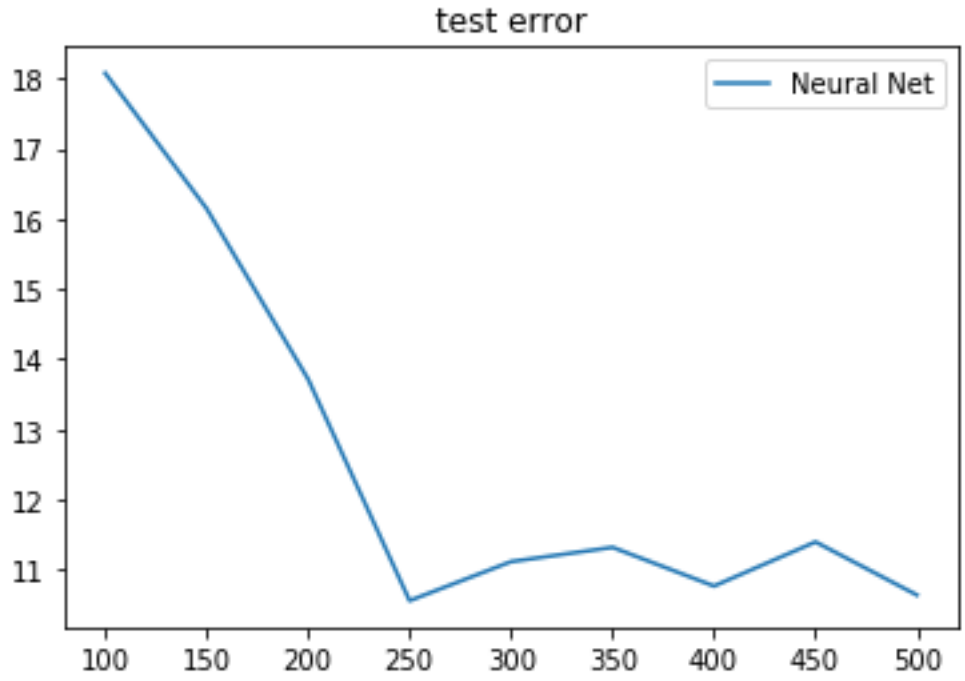
e) see the pdf and graph

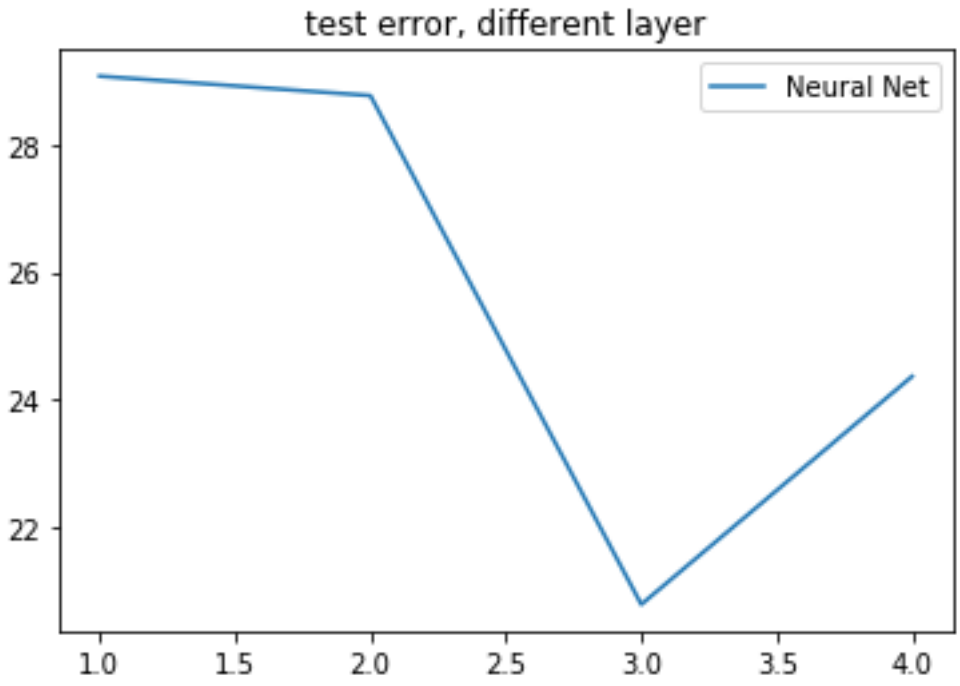
f) SGD is much better than gradient descent.


regular training error
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0
10
20
30
40
0
50
100
150
200
250
300


regular test error
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0
20
40
60
80
100
120
0
50
100
150
200
250
300

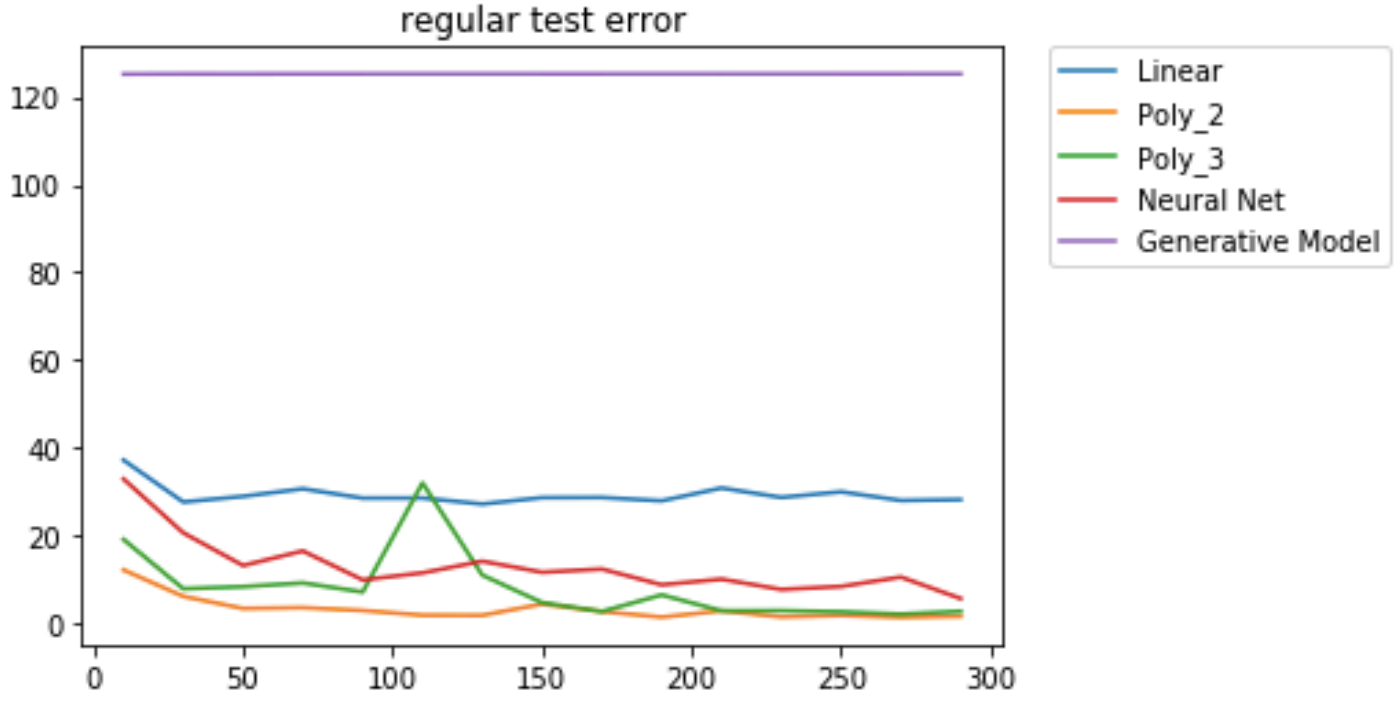

shift test error
1e7
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0.0
0.2
0.4
0.6
0.8
0
50
100
150
200
250
300


test error
Neural Net
18
17
16
15
14
13
12
11
100
150
200
250
300
350
400
450
500



k = 3 is better since it gives me a smaller test error


regular training error
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0
10
20
30
40
50
0
50
100
150
200
250
300


regular test error
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0
20
40
60
80
100
120
0
50
100
150
200
250
300


shift test error
1e7
Linear
Poly_2
Poly_3
Neural Net
Generative Model
0.0
0.2
0.4
0.6
0.8
0
50
100
150
200
250
300


training error
SGD
Gradient Descent
0
1
2
3
4
5
6
7
8
0
50
100
150
200
250
300


test error
SGD
Gradient Descent
5
10
15
20
25
30
35
0
50
100
150
200
250
300

hw09

October 30, 2017

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

In [13]:
```
# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear activation
    # a is a list of of the activations of each node in the previous layer going
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            update = np.dot(curGrad,curA.T)
            updateB = np.sum(curGrad,1).reshape(layer.b.shape)
            layer.updateWeights(-self.eta/m * np.dot(curGrad,curA.T))
            layer.updateBias(-self.eta/m * np.sum(curGrad,1).reshape(layer.b.shape))

# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
```

```
        return 0.5 * np.square(yp-y)

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        return y - yp

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def dx(z):
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))

# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        return np.tanh(z)

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        return 1 - np.square(np.tanh(z))

# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
        return np.maximum(0,z)

    @staticmethod
    def dx(z):
        return (z>0).astype('float')

# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        return z

    @staticmethod
    def dx(z):
```

```
        return np.ones(z.shape)

# This class represents a single hidden or output layer in the neural network
class DenseLayer(object):

    # numNodes: number of hidden units in the layer
    # activation: the activation function to use in this layer
    def __init__(self, numNodes, activation):
        self.numNodes = numNodes
        self.activation = activation

    def getNumNodes(self):
        return self.numNodes

    # Initialize the weight matrix of this layer based on the size of the matrix W
    def initialize(self, fanIn, scale=1.0):
        s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
        self.W = np.random.normal(0, s,
                                  (self.numNodes,fanIn))
        #self.b = np.zeros((self.numNodes,1))
        self.b = np.random.uniform(-1,1,(self.numNodes,1))

    # Apply the activation function of the layer on the input z
    def a(self, z):
        return self.activation.fx(z)

    # Compute the linear part of the layer
    # The input a is an n x k matrix where n is the number of samples
    # and k is the dimension of the previous layer (or the input to the network)
    def z(self, a):
        #print('a:\n'+str(a))
        #print('Wa:\n'+str(self.W.dot(a)))
        return self.W.dot(a) + self.b # Note, this is implemented where we assume a i

    # Compute the derivative of the layer's activation function with respect to z
    # where z is the output of the above function.
    # This derivative does not contain the derivative of the matrix multiplication
    # in the layer.  That part is computed below in the model class.
    def dx(self, z):
        return self.activation.dx(z)

    # Update the weights of the layer by adding dW to the weights
    def updateWeights(self, dW):
        self.W = self.W + dW

    # Update the bias of the layer by adding db to the bias
    def updateBias(self, db):
        self.b = self.b + db
```

```
# This class handles stacking layers together to form the completed neural network
class Model(object):

    # inputSize: the dimension of the inputs that go into the network
    def __init__(self, inputSize):
        self.layers = []
        self.inputSize = inputSize

    # Add a layer to the end of the network
    def addLayer(self, layer):
        self.layers.append(layer)

    # Get the output size of the layer at the given index
    def getLayerSize(self, index):
        if index >= len(self.layers):
            return self.layers[-1].getNumNodes()
        elif index < 0:
            return self.inputSize
        else:
            return self.layers[index].getNumNodes()

    # Initialize the weights of all of the layers in the network and set the cost
    # function to use for optimization
    def initialize(self, cost, initializeLayers=True):
        self.cost = cost
        if initializeLayers:
            for i in range(0,len(self.layers)):
                if i == len(self.layers) - 1:
                    self.layers[i].initialize(self.getLayerSize(i-1))
                else:
                    self.layers[i].initialize(self.getLayerSize(i-1))

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
    # This function returns
    # yp - the output of the network
    # a - a list of inputs for each layer of the newtork where
    #     a[i] is the input to layer i
    # z - a list of values for each layer after evaluating layer.z(a) but
    #     before evaluating the nonlinear function for the layer
    def evaluate(self, x):
        curA = x.T
        a = [curA]
        z = []
        for layer in self.layers:
            z.append(layer.z(curA))
```

```
            curA = layer.a(z[-1])
            a.append(curA)
        yp = a.pop()
        return yp, a, z

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
    def predict(self, a):
        a,_,_ = self.evaluate(a)
        return a.T

    # Train the network given the inputs x and the corresponding observations y
    # The network should be trained for numEpochs iterations using the supplied
    # optimizer
    def train(self, x, y, numEpochs, optimizer):

        # Initialize some stuff
        n = x.shape[0]
        hist = []
        optimizer.initialize(self.layers)

        # Run for the specified number of epochs
        for epoch in range(0,numEpochs):

            # Feed forward
            # Save the output of each layer in the list a
            # After the network has been evaluated, a should contain the
            # input x and the output of each layer except for the last layer
            yp, a, z = self.evaluate(x)

            # Compute the error
            C = self.cost.fx(yp,y.T)
            d = self.cost.dx(yp,y.T)
            grad = []

            # Backpropogate the error
            idx = len(self.layers)
            for layer, curZ in zip(reversed(self.layers),reversed(z)):
                idx = idx - 1
                # Here, we compute dMSE/dz_i because in the update
                # function for the optimizer, we do not give it
                # the z values we compute from evaluating the network
                grad.insert(0,np.multiply(d,layer.dx(curZ)))
                d = np.dot(layer.W.T,grad[0])

            # Update the errors
            optimizer.update(self.layers, grad, a)
```

```
            # Compute the error at the end of the epoch
            yh = self.predict(x)
            C = self.cost.fx(yh,y)
            C = np.mean(C)
            hist.append(C)
        return hist

    def trainBatch(self, x, y, batchSize, numEpochs, optimizer):

        # Copy the data so that we don't affect the original one when shuffling
        x = x.copy()
        y = y.copy()
        hist = []
        n = x.shape[0]

        for epoch in np.arange(0,numEpochs):

            # Shuffle the data
            r = np.arange(0,x.shape[0])
            x = x[r,:]
            y = y[r,:]
            e = []

            # Split the data in chunks and run SGD
            for i in range(0,n,batchSize):
                end = min(i+batchSize,n)
                batchX = x[i:end,:]
                batchY = y[i:end,:]
                e += self.train(batchX, batchY, 1, optimizer)
            hist.append(np.mean(e))

        return hist


##########################################################################
######### Part b ###########################################################
##########################################################################

##########################################################################
#########  Gradient Computing and MLE ######################################
##########################################################################
def compute_gradient_of_likelihood(single_obj_loc, sensor_loc,
                                   single_distance, noise = 1):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
```

```
    single_obj_loc: 1 * d numpy array.
    Location of the single object.

    sensor_loc: k * d numpy array.
    Location of sensor.

    single_distance: k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: d-dimensional numpy array.

    """
    loc_difference = single_obj_loc - sensor_loc # k * d.
    phi = np.linalg.norm(loc_difference, axis = 1) # k.
    weight = (phi - single_distance) / phi # k.

    grad = -np.sum(np.expand_dims(weight,1)*loc_difference,
                   axis = 0)/noise ** 2 # d
    return grad

##############################################################################
######### Part c #############################################################
##############################################################################
def log_likelihood(obj_loc, sensor_loc, distance, noise = 1):
    """
    This function computes the log likelihood (as expressed in Part a).
    Input:
    obj_loc: shape [1,2]
    sensor_loc: shape [7,2]
    distance: shape [7]
    Output:
    The log likelihood function value.
    """
    diff_distance = np.sqrt(np.sum((sensor_loc - obj_loc)**2, axis = 1))- distance
    func_value = -sum((diff_distance)**2)/(2 * noise ** 2)
    return func_value



##############################################################################
######### Part e, f, g #######################################################
##############################################################################

##############################################################################
#########  Gradient Computing and MLE ########################################
##############################################################################
def compute_grad_likelihood_part_e(sensor_loc, obj_loc, distance, noise = 1):
```

```
    """
    Compute the gradient of the loglikelihood function for part d.

    Input:
    sensor_loc: k * d numpy array.
    Location of sensors.

    obj_loc: n * d numpy array.
    Location of the objects.

    distance: n * k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: k * d numpy array.
    """
    grad = np.zeros(sensor_loc.shape)
    for i, single_sensor_loc in enumerate(sensor_loc):
        single_distance = distance[:,i]
        grad[i] = compute_gradient_of_likelihood(single_sensor_loc,
                    obj_loc, single_distance, noise)

    return grad

def find_mle_by_grad_descent_part_e(initial_sensor_loc,
           obj_loc, distance, noise = 1, lr=0.001, num_iters = 1000):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
    initial_sensor_loc: k * d numpy array.
    Initialized Location of the sensors.

    obj_loc: n * d numpy array. Location of the n objects.

    distance: n * k dimensional numpy array.
    Observed distance of the n object.

    Output:
    sensor_loc: k * d numpy array. The mle for the location of the object.

    """
    sensor_loc = initial_sensor_loc
    for t in range(num_iters):
        sensor_loc += lr * compute_grad_likelihood_part_e(\
            sensor_loc, obj_loc, distance, noise)

    return sensor_loc
```

```
 ##########################################################################
#########  Estimate distance given estimated sensor locations. #########
##########################################################################

def compute_distance_with_sensor_and_obj_loc(sensor_loc, obj_loc):
    """
    stimate distance given estimated sensor locations.

    Input:
    sensor_loc: k * d numpy array.
    Location of the sensors.

    obj_loc: n * d numpy array. Location of the n objects.

    Output:
    distance: n * k dimensional numpy array.
    """
    estimated_distance = scipy.spatial.distance.cdist(obj_loc,
                                            sensor_loc,
                                            metric='euclidean')
    return estimated_distance

##########################################################################
#########  Data Generating Functions #####################################
##########################################################################

def generate_sensors(num_sensors = 7, spatial_dim = 2):
    """
    Generate sensor locations.
    Input:
    num_sensors: The number of sensors.
    spatial_dim: The spatial dimension.
    Output:
    sensor_loc: num_sensors * spatial_dim numpy array.
    """
    sensor_loc = 100*np.random.randn(num_sensors,spatial_dim)
    return sensor_loc

def generate_dataset(sensor_loc, num_sensors = 7, spatial_dim = 2,
                num_data = 1, original_dist = True, noise = 1):
    """
    Generate the locations of n points.

    Input:
    sensor_loc: num_sensors * spatial_dim numpy array. Location of sensor.
    num_sensors: The number of sensors.
    spatial_dim: The spatial dimension.
```

```
    num_data: The number of points.
    original_dist: Whether the data are generated from the original
    distribution.

    Output:
    obj_loc: num_data * spatial_dim numpy array. The location of the num_data objects
    distance: num_data * num_sensors numpy array. The distance between object and
    the num_sensors sensors.
    """
    assert num_sensors, spatial_dim == sensor_loc.shape

    obj_loc = 100*np.random.randn(num_data, spatial_dim)
    if not original_dist:
       obj_loc += 1000

    distance = scipy.spatial.distance.cdist(obj_loc,
                                            sensor_loc,
                                            metric='euclidean')
    distance += np.random.randn(num_data, num_sensors) * noise
    return distance, obj_loc
```

0.1 a) five models

In [14]:
```
# Generative Model
def obj_loc_predict_gd(distance, sensor_loc, lr = 0.01):
    obj_loc = np.zeros((np.size(distance,axis = 0),2), dtype = np.float)
    for i in range( np.size(obj_loc,axis = 0)):
        for k in range(100):
            grad = compute_gradient_of_likelihood(obj_loc[i,:], sensor_loc, distance[
            obj_loc[i,:] = obj_loc[i,:] + grad * lr
    return obj_loc

def generative_model(distance_train, obj_loc_train, distance_test, obj_loc_test):
    initial_sensor_loc = np.zeros((np.size(distance_train,axis = 1),2), dtype = np.flo
    sensor_loc = find_mle_by_grad_descent_part_e(initial_sensor_loc, obj_loc_train,
                                                 distance_train, lr=0.0001, num_iters
    obj_loc_predict = obj_loc_predict_gd(distance_test, sensor_loc, 0.0001)
    obj_loc_predict_train = obj_loc_predict_gd(distance_train, sensor_loc)
    test_error = mean_square_root_error(obj_loc_test, obj_loc_predict)
    training_error = mean_square_root_error(obj_loc_train, obj_loc_predict_train)
    return test_error, training_error

# Linear Model
def linear_reg_model(distance_train, obj_loc_train):
    '''
    Input: distance_train, obj_loc_train
    output: linear model
    '''
```

```
    reg = LinearRegression()
    reg.fit(distance_train, obj_loc_train)
    return reg

def linear_error(distance_train, obj_loc_train, distance_test, obj_loc_test):
    linear_reg = linear_reg_model(distance_train, obj_loc_train)
    linear_reg
    obj_loc_predict = linear_reg.predict(distance_test)
    obj_loc_predict_train = linear_reg.predict(distance_train)
    test_error = mean_square_root_error(obj_loc_test, obj_loc_predict)
    training_error = mean_square_root_error(obj_loc_train, obj_loc_predict_train)
    return test_error, training_error

# Second/Third Order Polynomial Feature
def poly_reg_model(distance_train, obj_loc_train, distance_test, degree = 2):
    '''
    input: distance_train, obj_loc_train
    '''
    poly = PolynomialFeatures(degree)
    distance_train_poly = poly.fit_transform(distance_train)
    distance_test_poly = poly.fit_transform(distance_test)
    reg = LinearRegression()
    reg.fit(distance_train_poly, obj_loc_train)
    train_predict = reg.predict(distance_train_poly)
    error = mean_square_root_error(obj_loc_train, train_predict)
    return reg, distance_test_poly, error

def poly_error(distance_train, obj_loc_train, distance_test, obj_loc_test, degree = 2
    poly_reg, distance_test_poly, training_error = poly_reg_model(distance_train,
                                                                  obj_loc_train,
                                                                  distance_test, degre
    obj_loc_predict = poly_reg.predict(distance_test_poly)
    test_error = mean_square_root_error(obj_loc_test, obj_loc_predict)
    return test_error, training_error

# Neural Network
def NN_error(distance_train, obj_loc_train, distance_test, obj_loc_test,
             layer = 2, node_num = 100, eta = 1):
    model = Model(distance_train.shape[1])
    for i in range(layer):
        model.addLayer(DenseLayer(node_num,ReLUActivation()))
    model.addLayer(DenseLayer(2,LinearActivation()))
    model.initialize(QuadraticCost())
    model.train(distance_train, obj_loc_train, 500, GDOptimizer(eta))
    obj_loc_predict =  model.predict(distance_test)
    obj_loc_predict_train = model.predict(distance_train)
    test_error = mean_square_root_error(obj_loc_test, obj_loc_predict)
    training_error = mean_square_root_error(obj_loc_train, obj_loc_predict_train)
```

```
    return test_error, training_error

# Error for evaluation
def mean_square_root_error(test, predict):
    return np.sum(np.power(np.sum(np.power(test - predict,2),
                                  axis = 1),1/2))/np.size(test,axis = 0)
```

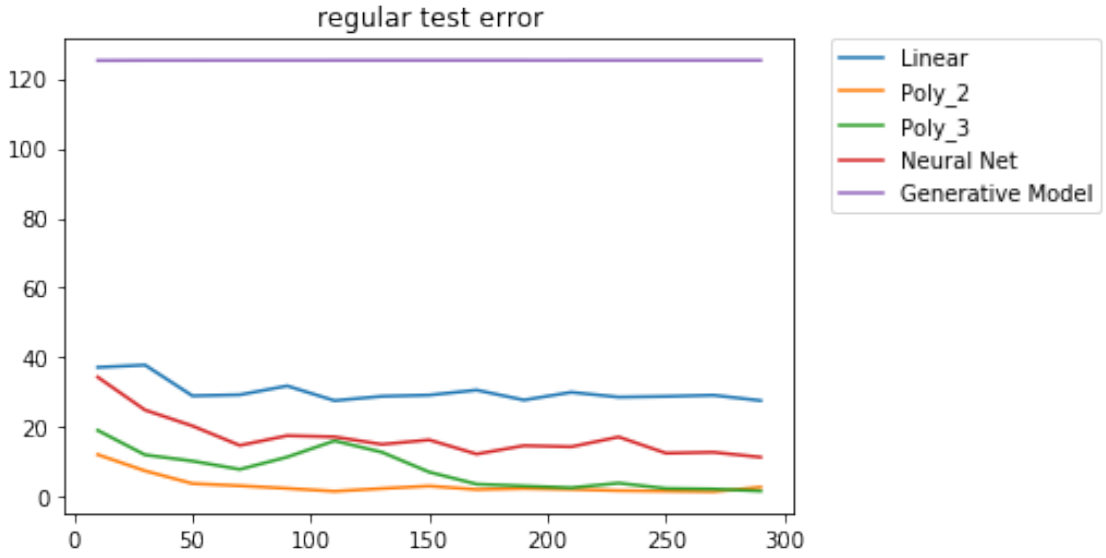
0.2 b) plot test / shift test and training errors

In [16]:
```
np.random.seed(100)
sensor_loc = generate_sensors()

num_data_list = range(10,310,20)
num_data_test = 1000
# generate test data num_data_test = 1000
distance_test, obj_loc_test = generate_dataset(sensor_loc, 7, 2,
                                               num_data = num_data_test)
error_test = []
for i in range(5):
    error_test.append([])

error_training = []
for i in range(5):
    error_training.append([])

error_test_shift = []
for i in range(5):
    error_test_shift.append([])


for num_data in num_data_list:
    # generate training data
    distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, num_data)
    # linear model
    test_error, training_error = linear_error(distance_train, obj_loc_train,
                                              distance_test, obj_loc_test)
    error_test[0].append(test_error)
    error_training[0].append(training_error)
    # polynomial featurepoly_reg 2
    test_error, training_error = poly_error(distance_train, obj_loc_train,
                                            distance_test, obj_loc_test, degree = 2)
    error_test[1].append(test_error)
    error_training[1].append(training_error)
    # polynomial featurepoly_reg 3
    test_error, training_error = poly_error(distance_train, obj_loc_train,
                                            distance_test, obj_loc_test, degree = 3)
    error_test[2].append(test_error)
```

```
        error_training[2].append(training_error)
        # Neural Network
        test_error, training_error = NN_error(distance_train, obj_loc_train,
                                              distance_test, obj_loc_test, eta = 0.000001)
        error_test[3].append(test_error)
        error_training[3].append(training_error)
        # generative model
        test_error, training_error = generative_model(distance_train, obj_loc_train,
                                                      distance_test, obj_loc_test)
        error_test[4].append(test_error)
        error_training[4].append(training_error)

        distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2,
                                                         num_data, original_dist= False)
        test_error,_ = linear_error(distance_train, obj_loc_train,
                                    distance_test, obj_loc_test)
        error_test_shift[0].append(test_error)
        # polynomial featurepoly_reg 2
        test_error,_ = poly_error(distance_train, obj_loc_train,
                                  distance_test, obj_loc_test, degree = 2)
        error_test_shift[1].append(test_error)
        # polynomial featurepoly_reg 3
        test_error,_ = poly_error(distance_train, obj_loc_train,
                                  distance_test, obj_loc_test, degree = 3)
        error_test_shift[2].append(test_error)
        # Neural Network
        test_error,_ = NN_error(distance_train, obj_loc_train,
                                distance_test, obj_loc_test, eta = 0.00000000005)
        error_test_shift[3].append(test_error)
        # generative model
        test_error, _ = generative_model(distance_train,
                                         obj_loc_train, distance_test, obj_loc_test)
        error_test_shift[4].append(test_error)
```

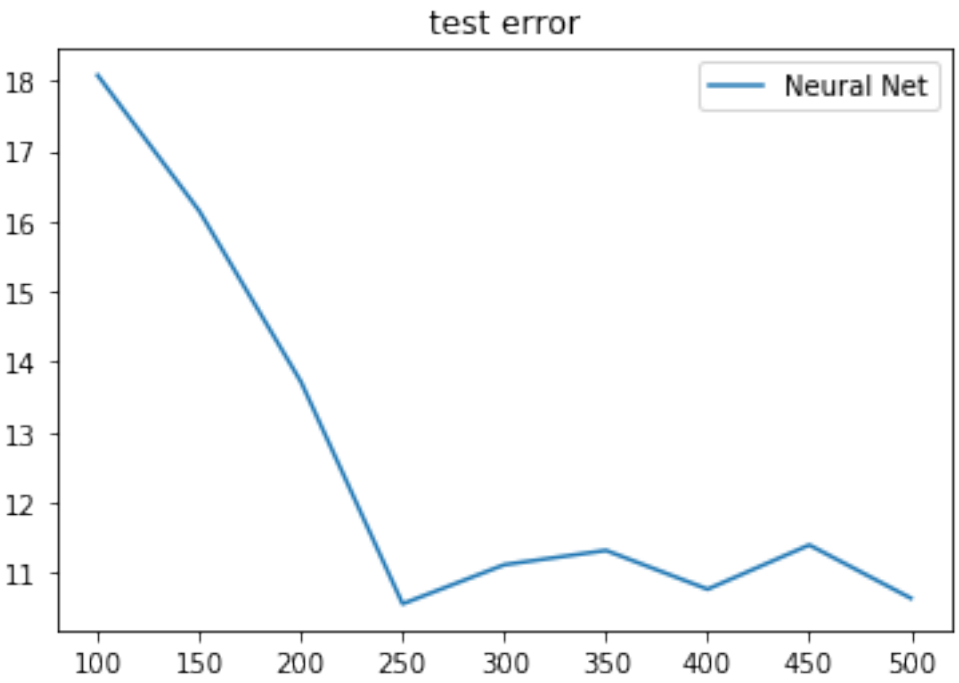
In [17]:
```
line_name = ['Linear', 'Poly_2', 'Poly_3', 'Neural Net', 'Generative Model']
for i in range(5):
    plt.plot(num_data_list,error_training[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('regular training error')
plt.show()

for i in range(5):
    plt.plot(num_data_list,error_test[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('regular test error')
plt.show()

for i in range(5):
```

```
    plt.plot(num_data_list,error_test_shift[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('shift test error')
plt.show()
```



0.3 c)

```
In [18]: node_num_list = range(100,550,50)
         distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, 200)
         error_test = []
         for i in range(5):
             error_test.append([])

         error_training = []
         for i in range(5):
             error_training.append([])

         for node_num in node_num_list:
             test_error, training_error = NN_error(distance_train, obj_loc_train,
                                                   distance_test, obj_loc_test, layer = 2,
                                                   node_num = node_num, eta = 0.000001)
             error_test[3].append(test_error)
             error_training[3].append(training_error)

In [19]: plt.plot(node_num_list,error_test[3], label = line_name[3])
         plt.legend()
         plt.title('test error')
         plt.show()
```

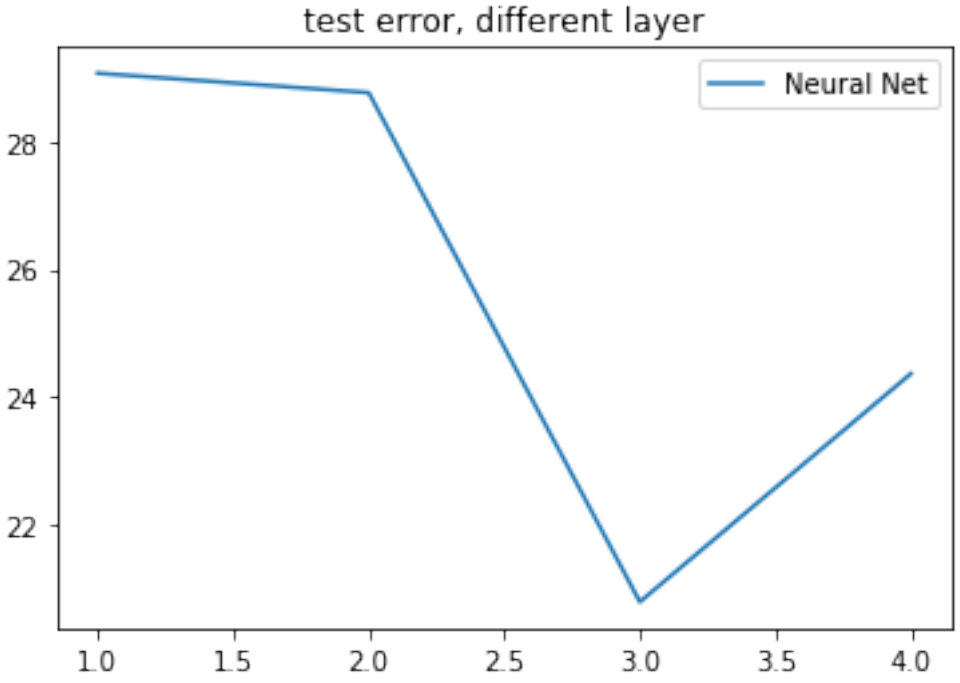


0.4 d)

```
In [153]: def solve_node_num(num_para, layer_num, input_num = 7, output_num = 2):
              b = input_num + output_num
              a = layer_num - 1
              c = - num_para
              if a > 0:
                  node_num = (int)((-b + np.sqrt(b**2 - 4*a*c))/(2*a))
              else:
                  node_num = (int)(-c/b)
              return node_num

          layer_list = range(1,5,1)
          distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, 200)
          error_test = []
          for i in range(5):
              error_test.append([])

          error_training = []
          for i in range(5):
              error_training.append([])

          for layer in layer_list:
              node_num = solve_node_num(10000, layer)
```

```
        test_error, training_error = NN_error(distance_train, obj_loc_train, distance_tes
        error_test[3].append(test_error)
        error_training[3].append(training_error)
```

In [154]:
```
plt.plot(layer_list,error_test[3], label = line_name[3])
plt.legend()
plt.title('test error, different layer')
plt.show()
```



0.5 k = 3 is better since it gives me a smaller test error

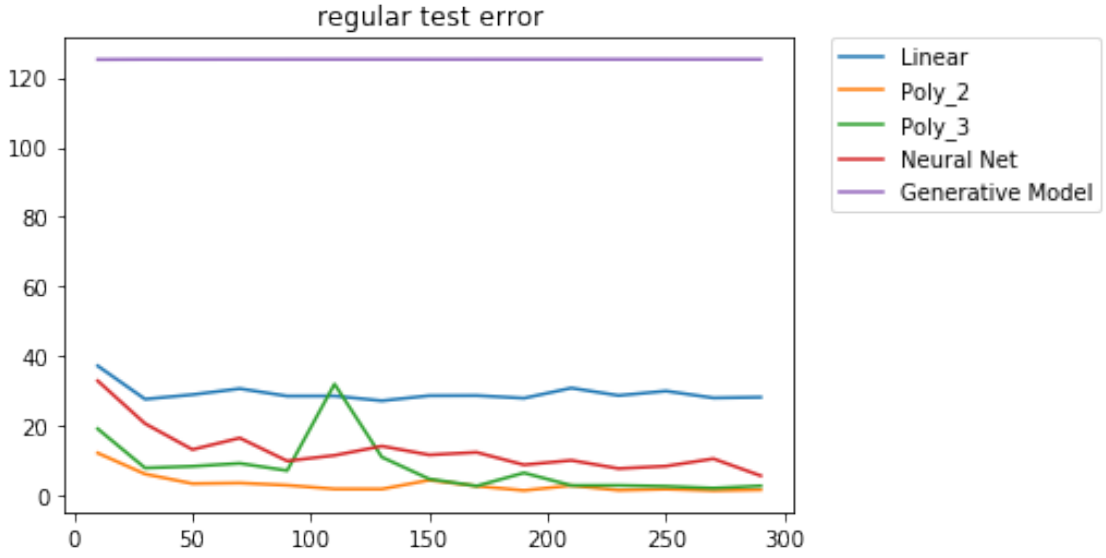
0.6 e)

In [204]:
```
np.random.seed(100)
sensor_loc = generate_sensors()

num_data_list = range(10,310,20)

# generate test data num_data_test = 1000
distance_test, obj_loc_test = generate_dataset(sensor_loc, 7, 2,
                                               num_data = num_data_test)
error_test = []
for i in range(5):
    error_test.append([])
```

```
error_training = []
for i in range(5):
    error_training.append([])

error_test_shift = []
for i in range(5):
    error_test_shift.append([])


for num_data in num_data_list:
    # generate training data
    distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, num_data)
    # linear model
    test_error, training_error = linear_error(distance_train, obj_loc_train,
                                              distance_test, obj_loc_test)
    error_test[0].append(test_error)
    error_training[0].append(training_error)
    # polynomial featurepoly_reg 2
    test_error, training_error = poly_error(distance_train, obj_loc_train,
                                            distance_test, obj_loc_test, degree = 2)
    error_test[1].append(test_error)
    error_training[1].append(training_error)
    # polynomial featurepoly_reg 3
    test_error, training_error = poly_error(distance_train, obj_loc_train,
                                            distance_test, obj_loc_test, degree = 3)
    error_test[2].append(test_error)
    error_training[2].append(training_error)
    # Neural Network
    test_error, training_error = NN_error(distance_train, obj_loc_train,
                                          distance_test, obj_loc_test, layer = 3,
                                          node_num = 250, eta = 0.000001)
    error_test[3].append(test_error)
    error_training[3].append(training_error)
     # generative model
    test_error, training_error = generative_model(distance_train, obj_loc_train,
                                                  distance_test, obj_loc_test)
    error_test[4].append(test_error)
    error_training[4].append(training_error)

    distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, num_data,
                                                     original_dist= False)
    test_error,_ = linear_error(distance_train, obj_loc_train,
                                distance_test, obj_loc_test)
    error_test_shift[0].append(test_error)
    # polynomial featurepoly_reg 2
    test_error,_ = poly_error(distance_train, obj_loc_train, distance_test,
                              obj_loc_test, degree = 2)
    error_test_shift[1].append(test_error)
```

```
        # polynomial featurepoly_reg 3
        test_error,_ = poly_error(distance_train, obj_loc_train, distance_test,
                                  obj_loc_test, degree = 3)
        error_test_shift[2].append(test_error)
        # Neural Network
        test_error,_ = NN_error(distance_train, obj_loc_train, distance_test,
                                obj_loc_test, layer = 3, node_num = 250, eta = 0.0000000
        error_test_shift[3].append(test_error)
        # generative model
        test_error, _ = generative_model(distance_train, obj_loc_train,
                                         distance_test, obj_loc_test)
        error_test_shift[4].append(test_error)
```

In [205]:
```
line_name = ['Linear', 'Poly_2', 'Poly_3', 'Neural Net', 'Generative Model']
for i in range(5):
    plt.plot(num_data_list,error_training[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('regular training error')
plt.show()

for i in range(5):
    plt.plot(num_data_list,error_test[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('regular test error')
plt.show()

for i in range(5):
    plt.plot(num_data_list,error_test_shift[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('shift test error')
plt.show()
```



0.7 f)

```
In [159]: def NN_SGD_error(distance_train, obj_loc_train, distance_test,
                           obj_loc_test, batch_size = 5, layer = 2, node_num = 100, eta = 1):
              model = Model(distance_train.shape[1])
              for i in range(layer):
```

```
            model.addLayer(DenseLayer(node_num,ReLUActivation()))
        model.addLayer(DenseLayer(2,LinearActivation()))
        model.initialize(QuadraticCost())
        model.trainBatch(distance_train, obj_loc_train, batch_size, 500, GDOptimizer(eta)
        obj_loc_predict =  model.predict(distance_test)
        obj_loc_predict_train = model.predict(distance_train)
        test_error = mean_square_root_error(obj_loc_test, obj_loc_predict)
        training_error = mean_square_root_error(obj_loc_train, obj_loc_predict_train)
        return test_error, training_error
```

In [160]:
```
np.random.seed(100)
sensor_loc = generate_sensors()

num_data_list = range(10,310,20)

# generate test data num_data_test = 1000
distance_test, obj_loc_test = generate_dataset(sensor_loc, 7, 2, num_data = num_data_
error_test = []
for i in range(5):
    error_test.append([])

error_training = []
for i in range(5):
    error_training.append([])

error_test_shift = []
for i in range(5):
    error_test_shift.append([])

batch_size = 5
for num_data in num_data_list:
    # generate training data
    distance_train, obj_loc_train = generate_dataset(sensor_loc, 7, 2, num_data)
    # Neural Network
    test_error, training_error = NN_error(distance_train, obj_loc_train,
                                          distance_test, obj_loc_test,
                                          layer = 3, node_num = 250, eta = 0.000001)
    error_test[3].append(test_error)
    error_training[3].append(training_error)
    test_error, training_error = NN_SGD_error(distance_train, obj_loc_train,
                                              distance_test, obj_loc_test,
                                              batch_size, layer = 3,
                                              node_num = 250, eta = 0.000001)
    error_test[2].append(test_error)
    error_training[2].append(training_error)
```

In [162]:
```
line_name = ['', '', 'SGD', 'Gradient Descent']
for i in range(2,4,1):
```

```
    plt.plot(num_data_list,error_training[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('training error')
plt.show()

for i in range(2,4,1):
    plt.plot(num_data_list,error_test[i], label = line_name[i])
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('test error')
plt.show()
```

encryption

October 30, 2017

In [3]:
```
import numpy as np
import tensorflow as tf
```

In [6]:
```
class BitData(object) :
        def __init__(self, modes, batchSize, propOfTrain, propOfValidation) :
                self.readData(modes)
                self.propOfTrain = propOfTrain
                self.propOfValidation = propOfValidation
                self.crossValidation()
                self.batchSize = batchSize
                self.startIndexTrain = 0

        def readData(self, modes) :
                self.feature = []
                self.numData = []

                for i, mode in enumerate(modes) :
                        fileName = mode + '.csv'
                        numLines = 0
                        with open(fileName, 'r') as f :
                                allLines = f.readlines()
                                for j, line in enumerate(allLines) :
                                        if (len(line) >= 64):
                                                self.feature.append(self.toBits(line))
                                                numLines += 1

                        self.numData.append(numLines)
                        if i == 0 :
                                label = np.array([1,0])
                                self.label = np.tile(label, (self.numData[0], 1))
                        elif i == 1 :
                                label = np.array([0,1])
                                labels = np.tile(label, (numLines, 1))
                                self.label = np.vstack((self.label, labels))
                        else :
                                print("Wrong name of a mode!")
                self.feature = np.array(self.feature)
```

```
        self.numData = np.array(self.numData)


    def crossValidation(self) :
        self.numTrain = np.int_(self.numData * self.propOfTrain)
        endIndexValidation = np.int_(self.numData * (self.propOfTrain
                                                     + self.propOfValidation))

        self.trainFeature = np.vstack((self.feature[0 : self.numTrain[0], ],
                self.feature[self.numData[0] : self.numData[0] + self.numTrain
        self.trainLabel = np.vstack((self.label[0 : self.numTrain[0], ],
                self.label[self.numData[0] : self.numData[0] + self.numTrain[1]
        s = np.arange(self.trainFeature.shape[0])
        np.random.shuffle(s)

        self.trainFeature = self.trainFeature[s,]
        self.trainLabel = self.trainLabel[s,]

        self.validationFeature = np.vstack((self.feature
                                    [self.numTrain[0] : endIndexValidation[0],
                self.feature[self.numData[0] + self.numTrain[1] : self.numData
                 + endIndexValidation[1], ]))
        self.validationLabel = np.vstack((self.label
                                   [self.numTrain[0] : endIndexValidation[0], ]
                self.label[self.numData[0] +
               self.numTrain[1] : self.numData[0] + endIndexValidation[1], ]))
        s = np.arange(self.validationFeature.shape[0])
        np.random.shuffle(s)
        self.validationFeature = self.validationFeature[s,]
        self.validationLabel = self.validationLabel[s,]

        self.testFeature = np.vstack((self.feature
                               [endIndexValidation[0] : self.numData[0], ],
                self.feature[self.numData[0]
                 + endIndexValidation[1] : self.numData[0] + self.numData[1], ]
        self.testLabel = np.vstack((self.label[endIndexValidation[0] : self.nu
                self.label[self.numData[0]
               + endIndexValidation[1] : self.numData[0] + self.numData[1], ]))
        s = np.arange(self.testFeature.shape[0])
        np.random.shuffle(s)

        self.testFeature = self.testFeature[s,]
        self.testLabel = self.testLabel[s,]

        self.numTrain = np.sum(self.numTrain)

    def hasNext(self) :
        if self.startIndexTrain >= self.numTrain :
```

```
                        return False
                else :
                        return True

        def next_batch(self) :
                if self.hasNext() :
                        batch_xs = self.trainFeature
            [self.startIndexTrain : self.startIndexTrain + self.batchSize, ]
                        batch_ys = self.trainLabel
            [self.startIndexTrain : self.startIndexTrain + self.batchSize, ]
                        self.startIndexTrain += self.batchSize
                        return batch_xs, batch_ys
                else :
                        return None

        def toBits(self, line) :
                #numbers = [int(line[i:i+1],16) for i in range(0,64,1)]
                temp = "{:0256b}".format(int(line[0:64],16))
                numbers = [int(temp[i:i + 1], 2) for i in range(0, 256, 1)]
                return numbers
```

In [5]:
```
num_classes = 2
epochs = 20
batch_num = 100
modeNames = ['cbc','ctr']

prob = tf.placeholder(tf.float32)
X = tf.placeholder(shape = [None, 256], dtype = tf.float32)
Y = tf.placeholder(shape = [None, num_classes], dtype = tf.int32)

W1 = tf.get_variable(name = 'W1',
                     shape = [256, 256],
                     initializer=tf.contrib.layers.xavier_initializer())
b1 = tf.Variable(tf.random_normal([256]), name = 'bias1')

W2 = tf.get_variable(name = 'W2',
                     shape = [256, 256],
                     initializer=tf.contrib.layers.xavier_initializer())
b2 = tf.Variable(tf.random_normal([256]), name = 'bias2')

W3 = tf.get_variable(name = 'W3',
                     shape = [256, 128],
                     initializer=tf.contrib.layers.xavier_initializer())
b3 = tf.Variable(tf.random_normal([128]), name = 'bias3')

W4 = tf.get_variable(name = 'W4',
                     shape = [128, num_classes],
```

```
                    initializer=tf.contrib.layers.xavier_initializer())
b4 = tf.Variable(tf.random_normal([num_classes]), name = 'bias4')

h1 = tf.nn.relu(tf.matmul(X, W1) + b1)
h1 = tf.nn.dropout(h1, keep_prob=prob)

h2 = tf.nn.relu(tf.matmul(h1, W2) + b2)
h2 = tf.nn.dropout(h2, keep_prob=prob)

h3 = tf.nn.relu(tf.matmul(h2, W3) + b3)
h3 = tf.nn.dropout(h3, keep_prob=prob)

logits = tf.matmul(h3, W4) + b4

cost = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
train = tf.train.AdamOptimizer(learning_rate = 0.01).minimize(cost)

prediction = tf.argmax(tf.nn.softmax(logits), 1)
accuracy = tf.reduce_mean(tf.cast(
    tf.equal(prediction, tf.argmax(Y, 1)), dtype = tf.float32))

with tf.Session() as sess :
    sess.run(tf.global_variables_initializer())
    bitData = BitData(modeNames, batch_num, 0.70, 0.15)

    num_training = int(bitData.numTrain / batch_num)

    for epoch in range(epochs) :
        avg_cost = 0
        bitData.startIndexTrain = 0

        for i in range(num_training) :
            batch_xs, batch_ys = bitData.next_batch()
            c, _, pred= sess.run([cost, train, prediction],
                                 feed_dict = {X:batch_xs, Y:batch_ys, prob : 0.5})
            avg_cost += c / num_training

        validation_acc = sess.run([accuracy],
                                  feed_dict = {X:bitData.validationFeature,
                                               Y:bitData.validationLabel, prob : 1.0})
        print("Epoch {} - average cost : {}, validation accuracy : {}"
              .format(epoch, avg_cost, validation_acc))

    test_acc = sess.run([accuracy], feed_dict =
                        {X:bitData.testFeature, Y:bitData.testLabel, prob : 1.0})
    print("Test accuracy : {}".format(test_acc))
```

```
Epoch 0 - average cost : 0.7015367916323123, validation accuracy : [0.53996468]
Epoch 1 - average cost : 0.6902032159671897, validation accuracy : [0.53996468]
Epoch 2 - average cost : 0.6902498995318116, validation accuracy : [0.53996468]
Epoch 3 - average cost : 0.6901924463199136, validation accuracy : [0.53996468]
Epoch 4 - average cost : 0.6901804801577587, validation accuracy : [0.53996468]
Epoch 5 - average cost : 0.6901941061756891, validation accuracy : [0.53996468]
Epoch 6 - average cost : 0.6901804893702222, validation accuracy : [0.53996468]
Epoch 7 - average cost : 0.6902065902165317, validation accuracy : [0.53996468]
Epoch 8 - average cost : 0.6901804721889762, validation accuracy : [0.53996468]
Epoch 9 - average cost : 0.6902755381887812, validation accuracy : [0.53996468]
Epoch 10 - average cost : 0.6902400742542613, validation accuracy : [0.53996468]
Epoch 11 - average cost : 0.6901943164501902, validation accuracy : [0.53996468]
Epoch 12 - average cost : 0.6901830328395228, validation accuracy : [0.53996468]
Epoch 13 - average cost : 0.6902651019405919, validation accuracy : [0.53996468]
Epoch 14 - average cost : 0.6902088809879342, validation accuracy : [0.53996468]
Epoch 15 - average cost : 0.6902241014992069, validation accuracy : [0.53996468]
Epoch 16 - average cost : 0.6901830766447913, validation accuracy : [0.53996468]
Epoch 17 - average cost : 0.69038688717338, validation accuracy : [0.53996468]
Epoch 18 - average cost : 0.6903186776722797, validation accuracy : [0.53996468]
Epoch 19 - average cost : 0.6901907454226084, validation accuracy : [0.53996468]
Test accuracy : [0.54000074]
```

0.1 My Own Question

First attempts on Project: Task: to distinguish two modes of encryption Encryptyion Algorithm: AES Mode[0]: CBC Mode[1]: CTR

Data Description: Input: 10,000 data samples generated by Mode[0] (encrypt 256 bits sequence - all zeros) 10,000 data samples generated by Mode[1] (encrypt 256 bits sequence - all zeros)

Output: class: 0 1

Results: Using neural network, test accuracy

MNIST_tutorial

October 30, 2017

In [2]:
```
"""A very simple MNIST classifier.
See extensive documentation at
https://www.tensorflow.org/get_started/mnist/beginners
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

FLAGS = None


def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])

  # The raw formulation of cross-entropy,
  #
  #   tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
  #                                 reduction_indices=[1]))
  #
  # can be numerically unstable.
  #
```

```
        # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
        # outputs of 'y', and then average across the batch.
        cross_entropy = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
        train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

        sess = tf.InteractiveSession()
        tf.global_variables_initializer().run()
        # Train
        for _ in range(1000):
          batch_xs, batch_ys = mnist.train.next_batch(100)
          sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

        # Test trained model
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                            y_: mnist.test.labels}))

      if __name__ == '__main__':
        parser = argparse.ArgumentParser()
        parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data
                            help='Directory for storing input data')
        FLAGS, unparsed = parser.parse_known_args()
        tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

```
Extracting /tmp/tensorflow/mnist/input_data/train-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/train-labels-idx1-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-labels-idx1-ubyte.gz
0.9179
```

```
      An exception has occurred, use %tb to see the full traceback.


      SystemExit
```

```
/Users/zhaoqingyang/anaconda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2889
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```