1. (a)

I did this homework with Luning Zhao.

We work on our homework seperately, but we discuss when meeting problems.

Homework is fine.

(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.

Siyao Jia

2. (a) Assume $P(Y=j|X)$ gives the highest probability

$$R(f(x)|x) = L(f(x), j) \cdot P(Y=j|x) + \sum_{i \neq j} L(f(x), i)\, P(Y=i|x)$$

if $f(x)=j$, $R_1(f(x)=j|x) = \sum_{i \neq j} L(f(x), i)\, P(Y=i|x)$

$$= \lambda_c \sum_{i \neq j} P(Y=i|x)$$

$$= \lambda_c (1 - P(Y=j|x))$$

if $f(x)=c+1$, $R_2(f(x)=c+1|x) = \lambda_d \sum P(Y=i|x) = \lambda_d$

if $f(x)=k \neq j$ and $f(x) \neq c+1$, $R_3(f(x)=k|x) = \lambda_c (1 - P(Y=k|x))$

Because $P(Y=j|x) \geq P(X=k|x)$, so $R_1(f(x)=j|x) \leq R_3(f(x)=k|x)$.

If $P(Y=j|x) \geq 1 - \frac{\lambda_d}{\lambda_c}$, $R_1(f(x)=j|x) \leq \lambda_c (1 - (1 - \frac{\lambda_d}{\lambda_c})) = \lambda_d$,

so $R_1(f(x)=j|x) \leq R_2(f(x)=c+1|x)$

On the contrary, if $P(Y=j|x) \leq 1 - \frac{\lambda_d}{\lambda_c}$, $R_1(f(x)=j|x) \geq R_2(f(x)=c+1|x)$.

In summary, $f_{opt}(x)$ obtains the minimum risk.

(b) If $\lambda_d = 0$, $P(Y=i|x) \leq 1$, so we should always choose class $c+1$.

If $\lambda_d > \lambda_c$, $P(Y=i|x) \geq 0$, so we should always choose class $i$.

This is exactly what we expect

Because if $\lambda_d = 0$, there is no penalty if we choose class $c+1$, just as good as choosing the correct class, so we should always choose class $c+1$

If $\lambda_d > \lambda_c$, then we would have more penalty if we choose class $c+1$, worse than we make a wrong choice, so we should always choose class $i$

3. (a) · MLE: $P(X|L) = N(\mu_L, \Sigma) = \frac{1}{\sqrt{2\pi|\Sigma|}} e^{-(X-\mu_L)^T\Sigma^{-1}(X-\mu_L)}$

if $P(X|L=1) \geq P(X|L=2) \Rightarrow -(X-\mu_1)^T\Sigma^{-1}(X-\mu_1) \geq -(X-\mu_2)^T\Sigma^{-1}(X-\mu_2)$

$2\mu_1^T\Sigma^{-1}X - \mu_1^T\Sigma^{-1}\mu_1 - 2\mu_2^T\Sigma^{-1}X + \mu_2^T\Sigma^{-1}\mu_2 \geq 0$

So if $2(\mu_1-\mu_2)^T\Sigma^{-1}X \geq -\mu_2^T\Sigma^{-1}\mu_2 + \mu_1^T\Sigma^{-1}\mu_1$, choose label 1

otherwise, choose label 2

· MAP: $P(L|X) \propto P(X|L)\,P(L) = e^{-(X-\mu_L)^T\Sigma^{-1}(X-\mu_L)}\,\pi_L$

$P(X|L=1) \geq P(X|L=2) \Rightarrow -(X-\mu_1)^T\Sigma^{-1}(X-\mu_1) + \log\pi_1 \geq -(X-\mu_2)^T\Sigma^{-1}(X-\mu_2) + \log\pi_2$

$2\mu_1^T\Sigma^{-1}X - \mu_1^T\Sigma^{-1}\mu_1 - 2\mu_2^T\Sigma^{-1}X + \mu_2^T\Sigma^{-1}\mu_2 + \log\frac{\pi_1}{\pi_2} \geq 0$

So if $2(\mu_1-\mu_2)^T\Sigma^{-1}X \geq -\mu_2^T\Sigma^{-1}\mu_2 + \mu_1^T\Sigma^{-1}\mu_1 + \log\frac{\pi_2}{\pi_1}$, choose label 1

Otherwise, choose label 2

When $\pi_1 = \pi_2$, the two decision rules are the same.

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

$= P(L=1)\,E[(X-E(X))(X-E(X))^T|L=1) + P(L=2)\,E[(X-E(X))(X-E(X))^T|L=2)$

$= \frac{1}{2}E[(X-\frac{\mu_1+\mu_2}{2})(X-\frac{\mu_1+\mu_2}{2})^T|L=1) + \frac{1}{2}E[(X-\frac{\mu_1+\mu_2}{2})(X-\frac{\mu_1+\mu_2}{2})^T|L=2)$

$\frac{1}{2}E[(X-\mu_1+\frac{\mu_1-\mu_2}{2})(X-\mu_1+\frac{\mu_1-\mu_2}{2})^T|L=1) + \frac{1}{2}E[\sim]$

$= \frac{1}{2}E[(X-\mu_1)(X-\mu_1)^T + \frac{\mu_1-\mu_2}{2}(X-\mu_1)^T + (\frac{\mu_1-\mu_2}{2})^T(X-\mu_1) + (\frac{\mu_1-\mu_2}{2})(\frac{\mu_1-\mu_2}{2})^T]+\sim$

$= \frac{1}{2}\Sigma + \frac{1}{8}(\mu_1-\mu_2)(\mu_1-\mu_2)^T + \frac{1}{2}\Sigma + \frac{1}{8}(\mu_2-\mu_1)(\mu_2-\mu_1)^T$

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

$$\Sigma_{xy} = E[(x - E(x))(y - E(y))^T]$$

$$= P(L=1)\, E[(x - E(x))(y - E(y))^T \mid L=1] + P(L=2)\, E[(x - E(x))(y - E(y))^T \mid L=2]$$

$$= \frac{1}{2} E\left[\left(x - \frac{\mu_1+\mu_2}{2}\right)\left(\begin{bmatrix}1\\0\end{bmatrix} - \begin{bmatrix}0.5\\0.5\end{bmatrix}\right)^T\right] + \frac{1}{2} E\left[\left(x - \frac{\mu_1+\mu_2}{2}\right)\left(\begin{bmatrix}0\\1\end{bmatrix} - \begin{bmatrix}0.5\\0.5\end{bmatrix}\right)^T\right]$$

$$= \frac{1}{2} \times \frac{1}{2} (\mu_1 - \mu_2) \begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\end{bmatrix} + \frac{1}{2} \times \frac{1}{2} (\mu_2 - \mu_1) \begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}$$

$$= \frac{1}{8} \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix} + \frac{1}{8} \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix}$$

$$= \frac{1}{4} \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix}$$

$$\Sigma_{yy} = E[(y - E(y))(y - E(y))^T]$$

$$= \frac{1}{2}\left(\begin{bmatrix}1\\0\end{bmatrix} - \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix}\right)\left(\begin{bmatrix}1 & 0\end{bmatrix} - \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}\right) + \frac{1}{2}\left(\begin{bmatrix}0\\1\end{bmatrix} - \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix}\right)\left(\begin{bmatrix}0 & 1\end{bmatrix} - \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}\right)$$

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

$$= \frac{1}{8} \begin{bmatrix}1 & -1\\-1 & 1\end{bmatrix} + \frac{1}{8} \begin{bmatrix}1 & -1\\-1 & 1\end{bmatrix}$$

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

(c) CCA tells us: $u^*$ is proportional to $\Sigma_{xx}^{-1} \Sigma_{xy} v^*$

$$u^* \propto \left(\Sigma + \frac{1}{4}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T\right)^{-1} \frac{1}{4} \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix} v^*$$

$$\propto \left(\Sigma^{-1} - \frac{\Sigma^{-1} \frac{1}{4}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T \Sigma^{-1}}{1 + (\mu_1 - \mu_2)^T \Sigma^{-1} \frac{1}{4}(\mu_1 - \mu_2)}\right) \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix} v^*$$

$$\propto \left(\Sigma^{-1} + \frac{1}{4}\Sigma^{-1}\left((\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2)\right) - \frac{1}{4}\Sigma^{-1}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T \Sigma^{-1}\right) \begin{bmatrix}\mu_1 - \mu_2 & \mu_2 - \mu_1\end{bmatrix} v^*$$

$$\propto \Sigma^{-1} \begin{bmatrix}\mu_1 - \mu_2\end{bmatrix} \begin{pmatrix}v_1^* - v_2^*\end{pmatrix}$$

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

from (a) we know $f(x) \propto (\mu_1 - \mu_2)^T \Sigma^{-1} x \propto u^{*T} x$

So $f(x) \propto u^{*T} x$

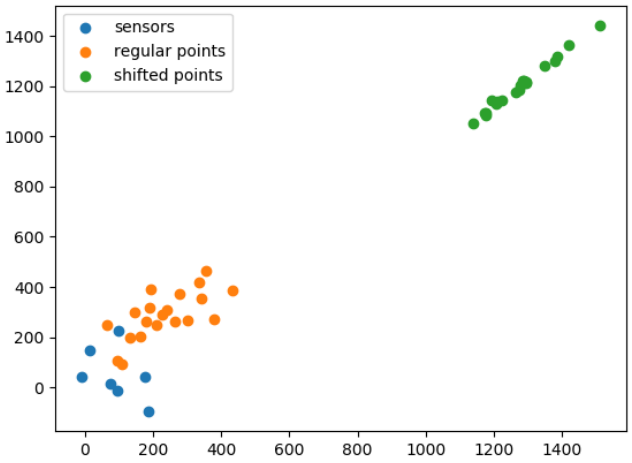
```
import numpy as np
import scipy.spatial
from starter import *


#####################################################################
## Models used for predictions.
#####################################################################
def compute_update(single_obj_loc, sensor_loc, single_distance):
    """
    Compute the gradient of the log-likelihood 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.
    grad = loc_difference / np.expand_dims(phi, 1)  # k * 2.
    update = np.linalg.solve(grad.T.dot(grad), grad.T.dot(single_distance - phi))

    return update


def get_object_location(sensor_loc, single_distance, num_iters=20, num_repeats=10):
    """
    Compute the gradient of the log-likelihood function for part a.

    Input:

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

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

    Output:
    obj_loc: 1 * d numpy array. The mle for the location of the object.

    """
    obj_locs = np.zeros((num_repeats, 1, 2))
    distances = np.zeros(num_repeats)
    for i in range(num_repeats):
        obj_loc = np.random.randn(1, 2) * 100
        for t in range(num_iters):
            obj_loc += compute_update(obj_loc, sensor_loc, single_distance)

        distances[i] = np.sum((single_distance - np.linalg.norm(obj_loc - sensor_loc
, axis=1))**2)
        obj_locs[i] = obj_loc

    obj_loc = obj_locs[np.argmin(distances)]

    return obj_loc[0]


def generative_model(X, Y, Xs_test, Ys_test):
    """
    This function implements the generative model.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
```

```
    Output:
    mse: Mean square error on test data.
    """
    initial_sensor_loc = np.random.randn(7, 2) * 100
    estimated_sensor_loc = find_mle_by_grad_descent_part_e(
        initial_sensor_loc, Y, X, lr=0.001, num_iters=1000)

    mses = []
    for i, X_test in enumerate(Xs_test):
        Y_test = Ys_test[i]
        Y_pred = np.array(
            [get_object_location(estimated_sensor_loc, X_test_single) for X_test_sin
gle in X_test])
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)
    return mses


def oracle_model(X, Y, Xs_test, Ys_test, sensor_loc):
    """
    This function implements the generative model.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
    sensor_loc: location of the sensors.
    Output:
    mse: Mean square error on test data.
    """
    mses = []
    for i, X_test in enumerate(Xs_test):
        Y_test = Ys_test[i]
        Y_pred = np.array([
            get_object_location(sensor_loc, X_test_single) for X_test_single in X_te
st])
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)
    return mses

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

def linear_regression(X, Y, Xs_test, Ys_test):
    """
    This function performs linear regression.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
    Output:
    mse: Mean square error on test data.
    """

    ## YOUR CODE HERE
    ##################
    poly = PolynomialFeatures(degree=1)
    X_p = poly.fit_transform(X)

    lg = LinearRegression()
    lg.fit(X_p, Y)

    mses = []
    for i in range(len(Xs_test)):
        X_test = poly.fit_transform(Xs_test[i])
        Y_test = Ys_test[i]
        Y_pred = lg.predict(X_test)
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)

    return mses
```

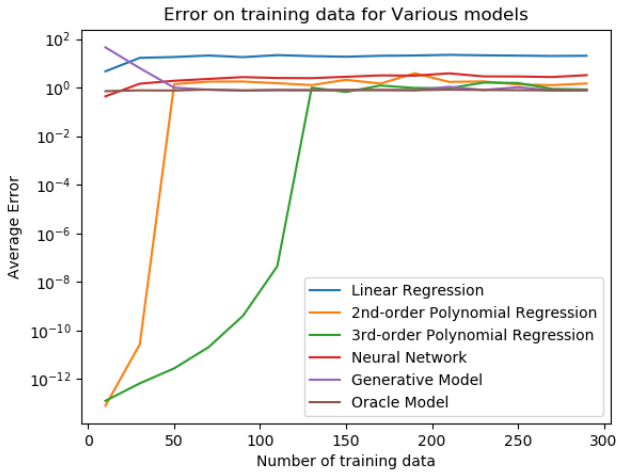
```
def poly_regression_second(X, Y, Xs_test, Ys_test):
    """
    This function performs second order polynomial regression.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
    Output:
    mse: Mean square error on test data.
    """
    ## YOUR CODE HERE
    #################
    poly = PolynomialFeatures(degree=2)
    X_p = poly.fit_transform(X)

    lg = LinearRegression()
    lg.fit(X_p, Y)

    mses = []
    for i in range(len(Xs_test)):
        X_test = poly.fit_transform(Xs_test[i])
        Y_test = Ys_test[i]
        Y_pred = lg.predict(X_test)
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)

    return mses


def poly_regression_cubic(X, Y, Xs_test, Ys_test):
    """
    This function performs third order polynomial regression.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
    Output:
    mse: Mean square error on test data.
    """
    ## YOUR CODE HERE
    #################
    poly = PolynomialFeatures(degree=3)
    X_p = poly.fit_transform(X)

    lg = LinearRegression()
    lg.fit(X_p, Y)

    mses = []
    for i in range(len(Xs_test)):
        X_test = poly.fit_transform(Xs_test[i])
        Y_test = Ys_test[i]
        Y_pred = lg.predict(X_test)
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)

    return mses

from sklearn.neural_network import MLPRegressor

def neural_network(X, Y, Xs_test, Ys_test):
    """
    This function performs neural network prediction.
    Input:
    X: independent variables in training data.
    Y: dependent variables in training data.
    Xs_test: independent variables in test data.
    Ys_test: dependent variables in test data.
    Output:
    mse: Mean square error on test data.
    """
```

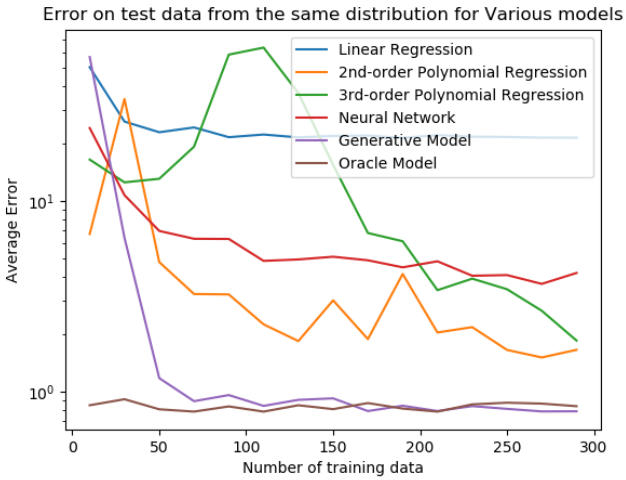
```
    ## YOUR CODE HERE
    ################
    clf = MLPRegressor(hidden_layer_sizes = (100, 100), activation = 'relu', solver=
'lbfgs')
    clf.fit(X, Y)

    mses = []
    for i, X_test in enumerate(Xs_test):
        Y_test = Ys_test[i]
        Y_pred = clf.predict(X_test)
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)

    return mses
```

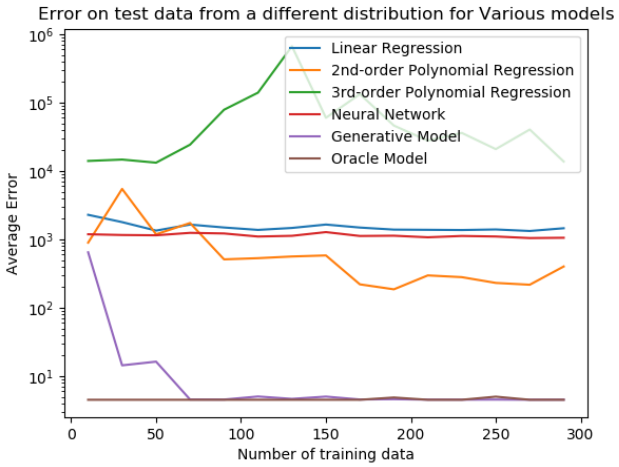
4 (b)




(c)




In general, error on large training data is similar for different model, around 1. For test data generated with same distribution, average error first decreases with number of training data, then becomes constant. For test data generated from different distribution, average almost doesn't change with number of training data.

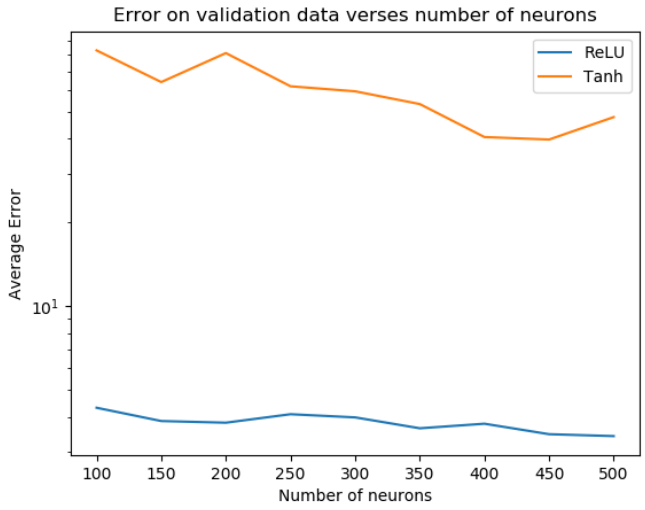
Polynomial model: computationally efficient,
but linear model error doesn't decrease with number of training data.
And 3rd-order polynomially regression has large test error because of overfitting.

Neural Network: if test data is generated from the same distribution as training data, the average error decreases with number of training data, but when it's from different distribution, the test error is large and doesn't decrease with number of training data.

Generative Model: test error decreases quickly with number of training data, but need to give a loss function.

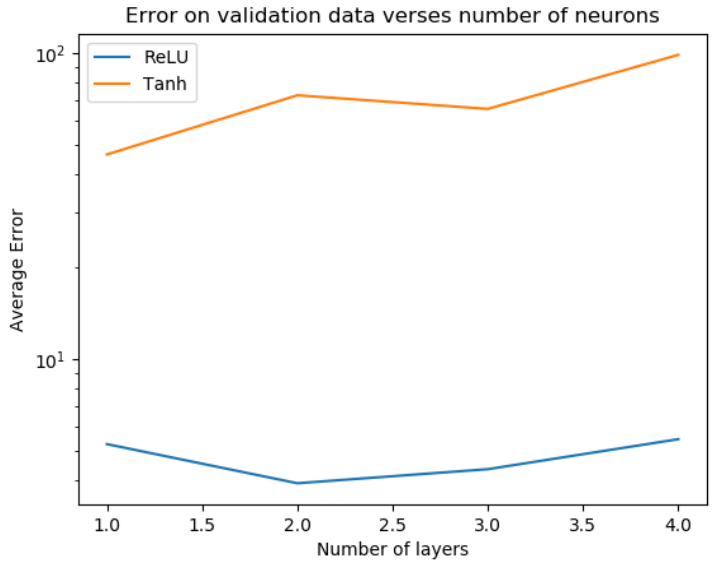
Oracle model: error is minimized, but it's not real since we don't know sensor location.

(d)




Best number of neurons is 450 for Tanh, 500 for ReLU

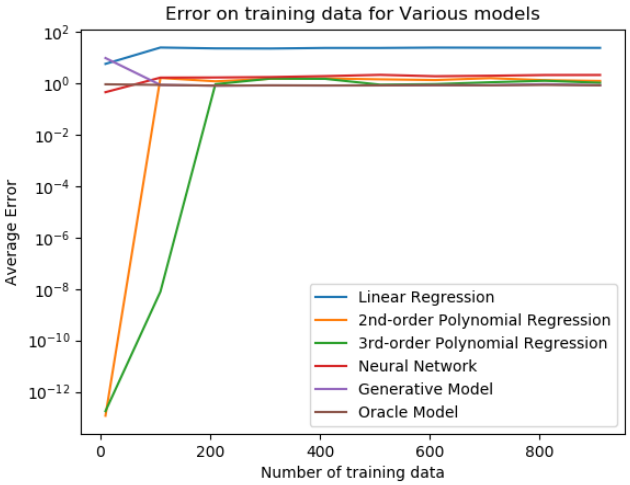
(e)

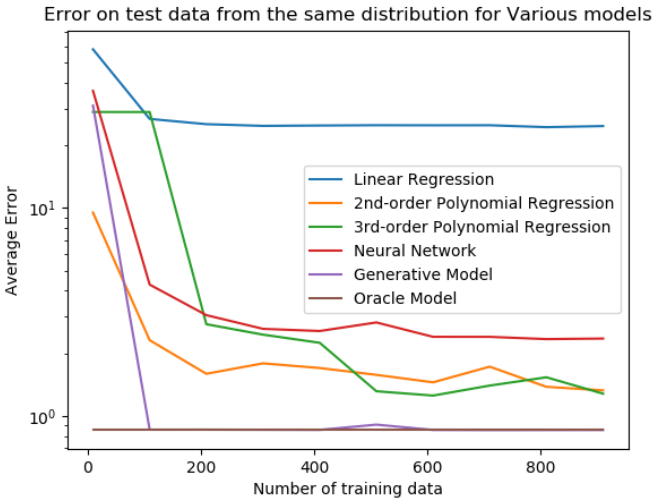


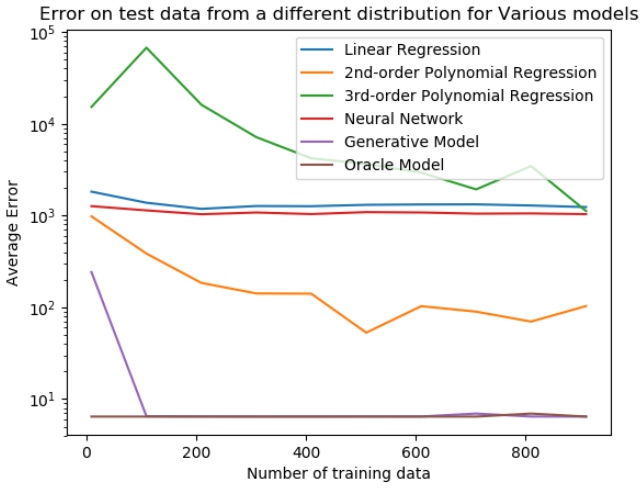

relationship between l and k:
2l+7l+(k-1)l^2=10000

Best number of layers is 1 for Tanh, 2 for ReLU

(f)




No, I can’t get it generalize to shifted test data

4d



```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.neural_network import MLPRegressor
from starter import *


def neural_network(X, Y, X_test, Y_test, num_neurons, activation):
    """
    This function performs neural network prediction.
    Input:
        X: independent variables in training data.
        Y: dependent variables in training data.
        X_test: independent variables in test data.
        Y_test: dependent variables in test data.
        num_neurons: number of neurons in each layer
        activation: type of activation, ReLU or tanh
    Output:
        mse: Mean square error on test data.
    """
    ## YOUR CODE HERE
    #################
    if activation == 'ReLU':
        activation = 'relu'
    clf = MLPRegressor(hidden_layer_sizes = (num_neurons, num_neurons), activation =
 activation, solver='lbfgs')
    clf.fit(X, Y)

    mses = []
    for i, x_test in enumerate(X_test):
        y_test = Y_test[i]
        y_pred = clf.predict([x_test])
        mse = np.mean(np.sqrt(np.sum((y_pred - y_test)**2, axis=1)))
        mses.append(mse)

    return np.mean(mses)


##############################################################################
########################PLOT PART 2###########################################
##############################################################################
def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
    return generate_dataset(
        sensor_loc,
        num_sensors=k,
        spatial_dim=d,
        num_data=n,
        original_dist=original_dist,
        noise=noise)


np.random.seed(0)
n = 200
num_neuronss = np.arange(100, 550, 50)
mses = np.zeros((len(num_neuronss), 2))

# for s in range(replicates):

sensor_loc = generate_sensors()
X, Y = generate_data(sensor_loc, n=n)  # X [n * 2] Y [n * 7]
X_test, Y_test = generate_data(sensor_loc, n=1000)
for t, num_neurons in enumerate(num_neuronss):
    ### Neural Network:
    mse = neural_network(X, Y, X_test, Y_test, num_neurons, "ReLU")
    mses[t, 0] = mse

    mse = neural_network(X, Y, X_test, Y_test, num_neurons, "tanh")
    mses[t, 1] = mse

    print('Experiment with {} neurons done...'.format(num_neurons))

### Plot MSE for each model.
plt.figure()
```

4d



```
activation_names = ['ReLU', 'Tanh']
for a in range(2):
    plt.plot(num_neuronss, mses[:, a], label=activation_names[a])

plt.title('Error on validation data verses number of neurons')
plt.xlabel('Number of neurons')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('num_neurons.png')
```

4e



```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.neural_network import MLPRegressor
from starter import *

def calc_layers(num_layers):
    k = num_layers
    l = np.roots([k-1, 9, -10000])
    idx = np.where(l>0)[0]
    l = int(l[idx])
    layers = []
    for i in range(k):
        layers.append(l)
    return layers

def neural_network(X, Y, X_test, Y_test, num_layers, activation):
    """
    This function performs neural network prediction.
    Input:
        X: independent variables in training data.
        Y: dependent variables in training data.
        X_test: independent variables in test data.
        Y_test: dependent variables in test data.
        num_layers: number of layers in neural network
        activation: type of activation, ReLU or tanh
    Output:
        mse: Mean square error on test data.
    """
    ## YOUR CODE HERE
    #################
    if activation == 'ReLU':
        activation = 'relu'
    clf = MLPRegressor(hidden_layer_sizes = calc_layers(num_layers), activation = ac
tivation, solver='lbfgs')
    clf.fit(X, Y)

    mses = []
    for i, x_test in enumerate(X_test):
        y_test = Y_test[i]
        y_pred = clf.predict([x_test])
        mse = np.mean(np.sqrt(np.sum((y_pred - y_test)**2, axis=1)))
        mses.append(mse)

    return np.mean(mses)


##############################################################################
########################PLOT PART 2###########################################
##############################################################################
def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
    return generate_dataset(
        sensor_loc,
        num_sensors=k,
        spatial_dim=d,
        num_data=n,
        original_dist=original_dist,
        noise=noise)


np.random.seed(0)
n = 200
num_layerss = [1, 2, 3, 4]
mses = np.zeros((len(num_layerss), 2))

# for s in range(replicates):
sensor_loc = generate_sensors()
X, Y = generate_data(sensor_loc, n=n)  # X [n * 2] Y [n * 7]
X_test, Y_test = generate_data(sensor_loc, n=1000)
for t, num_layers in enumerate(num_layerss):
    ### Neural Network:
    mse = neural_network(X, Y, X_test, Y_test, num_layers, "ReLU")
    mses[t, 0] = mse
```

4e



```
    mse = neural_network(X, Y, X_test, Y_test, num_layers, "tanh")
    mses[t, 1] = mse

    print('Experiment with {} layers done...'.format(num_layers))

### Plot MSE for each model.
plt.figure()
activation_names = ['ReLU', 'Tanh']
for a in range(2):
    plt.plot(num_layerss, mses[:, a], label=activation_names[a])

plt.title('Error on validation data verses number of neurons')
plt.xlabel('Number of layers')
plt.ylabel('Average Error')
plt.legend(loc='best')
plt.yscale('log')
plt.savefig('num_layers.png')
```

4f



```
import numpy as np
import matplotlib.pyplot as plt

from starter import *
from plot1 import *


def neural_network(X, Y, Xs_test, Ys_test):
    """
    This function performs neural network prediction.
    Input:
        X: independent variables in training data.
        Y: dependent variables in training data.
        X_test: independent variables in test data.
        Y_test: dependent variables in test data.
        num_layers: number of layers in neural network
        activation: type of activation, ReLU or tanh
    Output:
        mse: Mean square error on test data.
    """
    ## YOUR CODE HERE
    #################
    clf = MLPRegressor(hidden_layer_sizes = (500, 500), activation = 'relu', solver=
'lbfgs')
    clf.fit(X, Y)

    mses = []
    for i, X_test in enumerate(Xs_test):
        Y_test = Ys_test[i]
        Y_pred = clf.predict(X_test)
        mse = np.mean(np.sqrt(np.sum((Y_pred - Y_test)**2, axis=1)))
        mses.append(mse)

    return mses


def main():
    ##############################################################################
    ########################PLOT PART 1###########################################
    ##############################################################################
    np.random.seed(0)

    ns = np.arange(10, 1010, 100)
    replicates = 5
    num_methods = 6
    num_sets = 3
    mses = np.zeros((len(ns), replicates, num_methods, num_sets))

    def generate_data(sensor_loc, k=7, d=2, n=1, original_dist=True, noise=1):
        return generate_dataset(
            sensor_loc,
            num_sensors=k,
            spatial_dim=d,
            num_data=n,
            original_dist=original_dist,
            noise=noise)

    for s in range(replicates):
        sensor_loc = generate_sensors()
        X_test, Y_test = generate_data(sensor_loc, n=1000)
        X_test2, Y_test2 = generate_data(
            sensor_loc, n=1000, original_dist=False)
        for t, n in enumerate(ns):
            X, Y = generate_data(sensor_loc, n=n)  # X [n * 2] Y [n * 7]
            Xs_test, Ys_test = [X, X_test, X_test2], [Y, Y_test, Y_test2]
            ### Linear regression:
            mse = linear_regression(X, Y, Xs_test, Ys_test)
            mses[t, s, 0] = mse

            ### Second-order Polynomial regression:
            mse = poly_regression_second(X, Y, Xs_test, Ys_test)
            mses[t, s, 1] = mse
```

4f



```
            ### 3rd-order Polynomial regression:
            mse = poly_regression_cubic(X, Y, Xs_test, Ys_test)
            mses[t, s, 2] = mse

            ### Neural Network:
            mse = neural_network(X, Y, Xs_test, Ys_test)
            mses[t, s, 3] = mse

            ### Generative model:
            mse = generative_model(X, Y, Xs_test, Ys_test)
            mses[t, s, 4] = mse

            ### Oracle model:
            mse = oracle_model(X, Y, Xs_test, Ys_test, sensor_loc)
            mses[t, s, 5] = mse

            print('{}th Experiment with {} samples done...'.format(s, n))

    ### Plot MSE for each model.
    plt.figure()
    regressors = [
        'Linear Regression', '2nd-order Polynomial Regression',
        '3rd-order Polynomial Regression', 'Neural Network',
        'Generative Model', 'Oracle Model'
    ]
    for a in range(6):
        plt.plot(ns, np.mean(mses[:, :, a, 0], axis=1), label=regressors[a])

    plt.title('Error on training data for Various models')
    plt.xlabel('Number of training data')
    plt.ylabel('Average Error')
    plt.legend(loc='best')
    plt.yscale('log')
    plt.savefig('train_mse.png')
    plt.show()

    plt.figure()
    for a in range(6):
        plt.plot(ns, np.mean(mses[:, :, a, 1], axis=1), label=regressors[a])

    plt.title(
        'Error on test data from the same distribution for Various models')
    plt.xlabel('Number of training data')
    plt.ylabel('Average Error')
    plt.legend(loc='best')
    plt.yscale('log')
    plt.savefig('val_same_mse.png')
    plt.show()

    plt.figure()
    for a in range(6):
        plt.plot(ns, np.mean(mses[:, :, a, 2], axis=1), label=regressors[a])

    plt.title(
        'Error on test data from a different distribution for Various models')
    plt.xlabel('Number of training data')
    plt.ylabel('Average Error')
    plt.legend(loc='best')
    plt.yscale('log')
    plt.savefig('val_different_mse.png')
    plt.show()


if __name__ == '__main__':
    main()
```

5. (a) $\binom{n}{f_0^{(n)}\ f_1^{(n)}\ \cdots\ f_m^{(n)}} = \dfrac{n!}{f_0^{(n)}!\ f_1^{(n)}!\cdots f_m^{(n)}!}$

$$\approx \frac{\left(\frac{n}{e}\right)^n}{\prod_{i=0}^{m} \left(f_i^{(n)}/e\right)^{f_i^{(n)}}}$$

$$= \frac{n^n}{\prod_i \left(f_i^{(n)}\right)^{f_i^{(n)}}}$$

$$= \frac{e^{n \log n}}{\prod_i e^{f_i^{(n)} \log f_i^{(n)}}}$$

$$= e^{n \log n - \sum_i f_i^{(n)} \log f_i^{(n)}}$$

$$= e^{\log n \sum_i f_i^{(n)} - \sum_i f_i^{(n)} \log f_i^{(n)}}$$

$$= e^{\sum_i f_i^{(n)} \left(\log n - \log f_i^{(n)}\right)}$$

$$= e^{n \sum_i \frac{f_i^{(n)}}{n} \log \frac{n}{f_i^{(n)}}}$$

$$= e^{n H(f^{(n)}/n)} \qquad \text{where } H(p) = \sum_{i=0}^{m} p_i \ln \frac{1}{p_i}$$

(b) $$\lim_{n\to\infty} \frac{1}{n} \log P(F^{(n)} = f^{(n)}) = \lim_{n\to\infty} \frac{1}{n} \log \left[ e^{n \sum_i \frac{f_i^{(n)}}{n} \log \frac{n}{f_i^{(n)}}} \prod_i p_j^{f_i^{(n)}} \right]$$

$$= \lim_{n\to\infty} \frac{1}{n} \left( \sum_i f_i^{(n)} \log \frac{n}{f_i^{(n)}} + \sum_i f_j^{(n)} \log p_j \right)$$

$$= \lim_{n\to\infty} -\sum_i \frac{f_i^{(n)}}{n} \log \frac{f_i^{(n)}/n}{p_j}$$

$$= -KL(f, p)$$

(c) $$KL(p, q_\theta) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\vec{x}, y) \log \frac{P(\vec{x}, y)}{q_\theta(\vec{x}, y)}$$

$$= \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\vec{x}, y) \log \frac{P(\vec{x}, y)}{q_\theta(y|\vec{x})\, q(\vec{x})}$$

$$= \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\vec{x}, y) \left( \log \frac{P(\vec{x}, y)}{q(\vec{x})} - \log q_\theta(y|\vec{x}) \right)$$

$$= C - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\vec{x}, y) \log q_\theta(y|\vec{x})$$

(d) $$\min_\theta KL(p, q_\theta) = \min_\theta C - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\vec{x}, y) \log q_\theta(y|\vec{x})$$

$$= \min_\theta - \sum_{x_i} \sum_{y_i} P(\vec{x}_i, y_i) \log q_\theta(y_i|\vec{x}_i)$$

$$= \min_\theta - \frac{1}{n} \sum_i \log q_\theta(y_i|\vec{x}_i)$$

6. Q: Suppose $X$ is a random variable that can take the value of $x_1, x_2, \cdots, x_n$ with probability $\lambda_1, \lambda_2 \cdots \lambda_n$. $\varphi(x)$ is a convex function.

Prove Jensen's Inequality that: $\varphi(E(x)] \leq E[\varphi(x)]$

A: $\varphi(E(x)] = \varphi(\lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_n x_n)$

$E[\varphi(x)] = \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2) + \cdots + \lambda_n \varphi(x_n)$

First, when $n=2$, $\varphi(E(x)) = \varphi(\lambda_1 x_1 + \lambda_2 x_2)$

$E[\varphi(x)] = \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2)$

Because $\varphi(x)$ is a convex function, $\varphi[E(x)) \leq E[\varphi(x)]$

Then assume for some $n$, $\varphi(E(x)) \leq E[\varphi(x)]$

Let's show for $n+1$, $\varphi[E(x)] = \varphi(\lambda_1 x_1 + \cdots + \lambda_n x_n + \lambda_{n+1} x_{n+1})$

$\leq \varphi(\lambda_1 x_1 + \cdots + \lambda_n x_n) + \lambda_{n+1} \varphi(x_{n+1})$

$\leq \varphi(\lambda_1 x_1) + \varphi(\lambda_2 x_2) + \cdots + \lambda_{n+1} \varphi(x_{n+1})$

So we prove it.