Homework 7 (tutorial on sklearn regression)
Follow the tutorial below and created any required result to show that you went through the tutorial

Part 1: 7.1: Tutorial on GridSearchCV

GridSearCV is the method that allow the complete search on the hyper parameter in the estimator

Step 1: Create estimator called PolyRegress. By typing the following code into the jupyter notebook.

```
from sklearn.base import BaseEstimator
from sklearn.linear_model import LinearRegression
class PolyRegress(BaseEstimator):
    #inheriting from sklearn base class
    def __init__(self, deg=None):
        self.deg = deg
    def fit(self, X, y, deg=None):
        self.model = LinearRegression(fit_intercept=False)
        self.model.fit(np.vander(X, N=self.deg + 1
                                 , increasing = True), y)
    def predict(self, x):
        return self.model.predict(np.vander(x, N=self.deg + 1
                                 , increasing = True))
    @property #treat method as property
    def coef_(self):
        return self.model.coef_
```

Notice that the newly created class take “BaseEstimator” as the template (inherit from BaseEstimator class). This means that anything that is inside the BaseEstimator, the newly created class have it. In addition to what inherit from “BaseEstimator” class, the newly created class also override some of the method that is already in BaseEstimator.

To know what is already in PolyRegress class besides what to what we put in, type

```
PolyRegress.
```

Then press “Tab” key at the end

Required answer 1): You will get several options after pressure “tab” after type PolyRegress. What are those options (the drop down menu that show up after pressing “tab”)

What inherited from BaseEstimator, such as “get_params” method, may or may not be used in our newly created class.

Another thing to notice is that, this time we have `@property` before declaring function inside class. This means that the function `coef` become property instead of method (access by using `.coef` instead of `.coef()`, more explanation is below).

Inside the class PolyRegress, we have `__init__` function. This function is for initializing the object with the class PolyRegress.

PolyRegress also has .fit() function. This function create another object class LinearRegression. The name or identifier of this object is model. Model is the object inside the class object that is why we initialize by self.model. This is equivalent to type

```
model = LinearRegression(fit_intercept=False)
```

```
Outside the class. However, inside class, we use self.model = instead
of just model =. This means we refers to .model that is the member of
class PolyRegress (not object outside PolyRegress).
```

Step 2: create polynomial with known roots. We start this step by

```
import numpy as np
coef = np.poly([1,2,3,4,5])
p_obj = np.poly1d(coef)
```

Now, coef is the coefficient of polynomial that will have roots = [1,2,3,4,5]
p_obj is the polynomial object or function that can take the value of x and calculate f(x) value at any x according to the coefficient from coef variable.

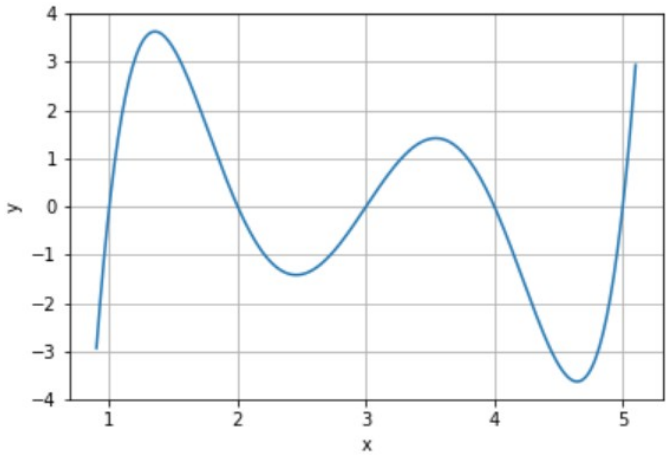
Required answer 2): Once you type

```
p_obj
```

as an input, what is the output?

Now we are going to check the graph of our polynomial. numpy.linspace can be used to create array of x from point a to b with a linear interval. p_obj is the object that can take the input as a numpy array too.

Required answer 3): Create the graph of polynomial p_obj for the input range of 0.9 to 5.1 with 300 points. You should get



You must have the axis name ‘x’ on x-axis and axis name ‘y’ on y-axis. You must also create grid lines. To do all of the above requirements, you may use

import matplotlib.pyplot as plt

```
plt.grid()
plt.xlabel('x')
```

`plt.grid()` will make grid lines. `plt.xlabel('x')` will make label of the x-axis to be `'x'`.

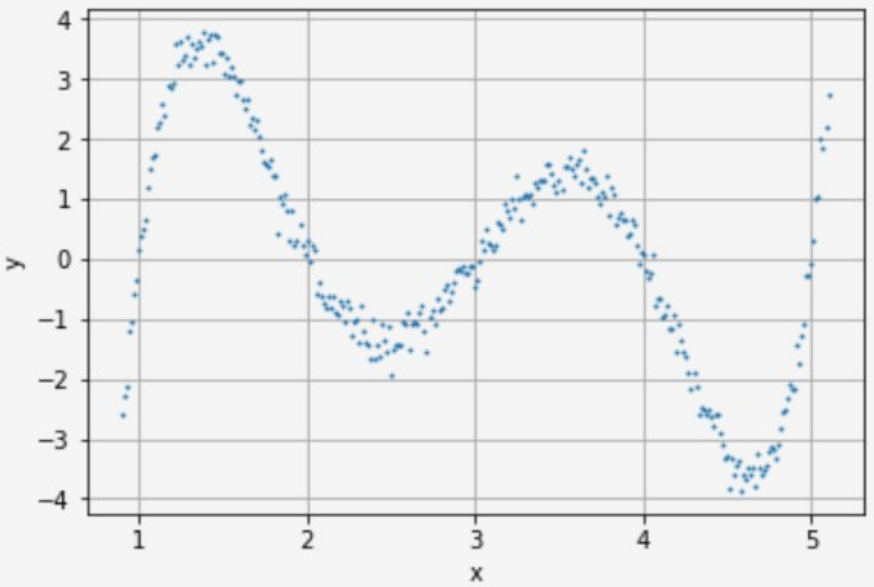
Step 3: add noise into the data.

Use random number that has normal distribution with mean = 0 and standard deviation of 0.2. Let’s create 300 points of the normal distribution noise. This can be obtained by

```
np.random.seed(20)
yn = y + np.random.normal(scale = 0.2, size = x.shape)
```

where y is the data without noise. yn is the data with noise. The reason that we set np.random.seed(20) is to make the result repeatable.

Required answer 4: Adding the noise with the y value in the graph from Required answer 3 and plot a graph, you should get



The size of the dot was set from

```
markersize = 1
```

in pyplot.plot option.

Step 4: we create the object named `model` with class `PolyRegress` defined earlier. We set the degree to degree 5 when create the model. This is done by

```
model = PolyRegress(deg=5)
```

Now we have 5th order polynomial regression model that is ready to be fitted with the data. To fit the model with the data use

```
model.fit(x,y)
```

x is the x-value, and y is y-value that you want the model to fit to.

Required answer 5: Fit the polynomial degree 5 with the data with noise created previously and plot the graph to compare the prediction from the model and the noisy data. Then, show the coefficient from the model and the coefficient used to create the data without noise.

Note that you can use

```
model.fit(x,yn)
```

to fit the data with noisy data (`yn`) and do

```
model.predict(x)
```

to get the prediction of the model based on the fitting with noisy data.
You can get the polynomial coefficient of the model by typing

`model.coef_`

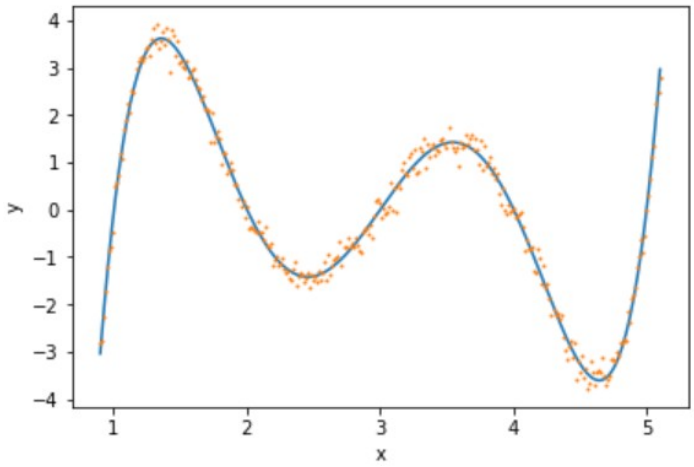
and can get the coefficient of the polynomial used to generate the data by just typing

`p_obj`

You should get something similar to

```
In [10]: model.coef_
Out[10]: array([-120.50523928,  275.14704606, -225.98809431,   85.39865118,
                 -15.07495529,    1.00527647])

In [11]: p_obj
Out[11]: poly1d([   1.,  -15.,   85., -225.,  274., -120.])
```

and



It is possible that your coefficient values are slightly different than the above given value. This can be due to some differences in the way the noise data are generated.

Notice that the coefficient of model.coef_ is in the inverse order of the one from numpy. This is because we use

```
self.model.fit(np.vander(X, N=self.deg + 1
                                , increasing = True), y)
```

in the class `PolyRegress` definition. np.vander give Vandermode matrix.

Required answer 6: What are the answers for

np.vander(np.linspace(1,10,10, dtype = int),N=5, increasing = True)

and

np.vander(np.linspace(1,10,10, dtype = int),N=5, increasing = False)

Show both answer and answer about the difference.

The coeffcients from PolyRegress class are to match for the input from the Vandermode matrix input. The data were arranged in the ascending order. Therefore, the coefficient is for ascending degree of polynomial too.

Required answer 7: Create PolyRegress2 class. Make everything the same, except using

`np.vander(..., increasing = False)` instead of `= True`

Then, redo the fitting and show your output coefficient. You should get

```
In [15]: model2.coef_
Out[15]: array([   1.00527647,  -15.07495529,   85.39865118, -225.98809431,
                275.14704606, -120.50523928])

In [16]: coef
Out[16]: array([   1.,  -15.,   85., -225.,  274., -120.])
```

Note that the input/output number does not need to match, but the answer should match.

Instead of using basis to be polynomial, we can use sine/cosine,exponential function to be the basis.

Step 5: Create class that use sine/cosine/x as bases

We need the create the array of

[[sin(x0), cos(x0), x0],
 [sin(x1), cos(x1), x1],
 [sin(x...), cos(x...), x...],
 [sin(xN), cos(xN), xN]]

This will give us all combination of all basis. Then, we can use PolynomialFeatures from sklearn to create the combination of each basis like x * sin(x) or x * sin^2(x), etc.

This can be done by typing

```
class OtherRegress(BaseEstimator):
    #inheriting from sklearn base class
    def __init__(self, deg=None, option = 'other'):
        """option can be either 'other' or 'polynomial'"""
        self.deg = deg
        self.option = option
    def fit(self, X, y, deg=None):
        self.model = LinearRegression(fit_intercept=False)
        if self.option == 'other':
            mod_poly = PolynomialFeatures(degree=self.deg)
            XX = mod_poly.fit_transform(
                np.c_[np.sin(X),np.cos(X),X])
            self.model.fit(XX,y)
            self.XX = XX
        else:
            self.model.fit(np.vander(X, N=self.deg + 1
                                     , increasing = True), y)
    def predict(self, x):
        if self.option == 'other':
            mod_poly = PolynomialFeatures(degree=self.deg)
            XX = mod_poly.fit_transform(
                np.c_[np.sin(x),np.cos(x),x])
            return self.model.predict(XX)
        else:
            return self.model.predict(np.vander(x, N=self.deg + 1
                                      , increasing = True))
    @property #treat method as property
    def coef_(self):
        return self.model.coef_
```

Please note that the code above is not going to work, unless we `import PolynomialFeatures` from `sklearn.preprocessing` first

The difference between class `OtherRegress` and `PolyRegress` is that we can have both polynomial and some sin/cosing/x as the basis in the case of `OtherRegress` object. This was done by modifying function fit and function predict inside the `OtherRegress class`. Notice the `if`-statement in `def predict` and `def fit`. The `if` statement is used to switch between polynomial bases and sine/cosine/x bases.

To create the second degree combination of various bases we can use

model_other = OtherRegress(deg = 2)

Then, we can fit this model with the data with noise that we have as

model_other.fit(x,yn)

The coefficient matrix will be the coefficient of each basis. Once do the plot between

`x` and `mode_other.predict(x)`

or

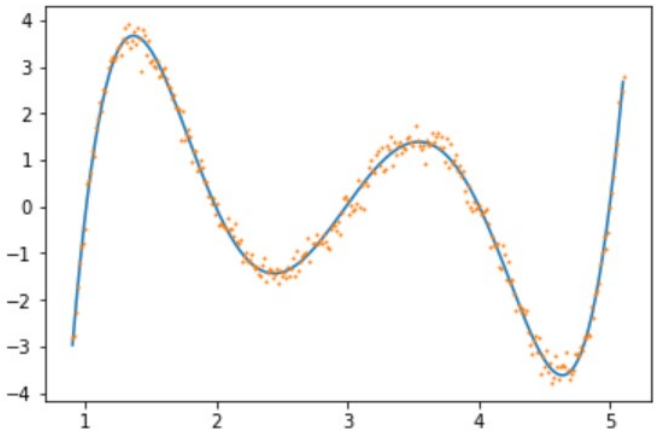
```
XX = model_other.XX
x and XX.dot(coef_mat)
```

where

```
coef_mat = model_other.coef_
```

and the noisy data,

We should get



Required answer 8: Create the prediction from OtherRegress object where degree = 2. Then, fit the model with the noisy data and make the prediction in the range of 0.9 to 5.1. You should get the graph show above.

Step 6: Do GridSearchCV
We can see that both method, polynomial and other basis (sin/cos/x) can match the data. Now assuming that we do not know what degree to be used for this matching purposes. We are going to use GridSearchCV to search for the model and the coefficient.

This GridSearchCV should tell us which model is the best model (‘other’ or ‘polynomial’) and what should be the degree to be used.

This can be done by

```
In [21]: from sklearn.model_selection import GridSearchCV
         cv_model = GridSearchCV(OtherRegress(), cv=5, param_grid={'deg':range(1,7),
                     'option':['other','polynomial']}, scoring = 'r2')
         cv_model.fit(x,yn)
```

where yn is the data with noise created earlier (you may used it in other name)

```
In : cv_model.best_estimator_
Out : OtherRegress(deg=5, option='polynomial')
```

Required answer 9: Redo the GridSearchCV, but this time do the grid search just for degree = [1,2,3,4] (not include the 5th degree) and option =[‘other’, ‘polynomial’]. Find out which hyper parameters for OtherRegress give the best fit when the degree 5 is not included. Use scoring = ‘r2’ (use r-squared as the scoring method).

Your answer should be

```
Out : OtherRegress(deg=2, option='other')
```

Step 7: put cv_results_ into pandas DataFrame
We can get the details of the GridSearchCV to find out the time required for doing each step and the average train/test score in each step of CV grid search. This can be done by calling the attribute `.cv_results_` and put it into DataFrame (pandas) format.
To use Pandas DataFrame, we need to run

```
import pandas as pd
```

To put the data from attribute cv_results_ into the data frame format, we can do it by

```
df = pd.DataFrame(cv_model.cv_results_)
df
```

Required answer 10: Get the GridSearchCV result in the pandas dataframe format for the case that the search was done for degree = [1,2,3,4] (not include the 5th degree) and option =[‘other’, ‘polynomial’] (that you did in Required answer 9). Show the data frame that you got. This should be

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_deg	param_option	params	rank_test_sc
0	0.000792	0.000495	-3.042558e+00	0.322871	1	other	{'deg': 1, 'option': 'other'}	
1	0.000547	0.000298	-1.385141e+00	0.273333	1	polynomial	{'deg': 1, 'option': 'polynomial'}	
2	0.000839	0.000501	-2.588264e-02	0.988445	2	other	{'deg': 2, 'option': 'other'}	
3	0.000447	0.000251	-2.619499e+00	0.302690	2	polynomial	{'deg': 2, 'option': 'polynomial'}	
4	0.000849	0.000417	-3.605783e+06	0.988597	3	other	{'deg': 3, 'option': 'other'}	
5	0.000332	0.000177	-3.492673e+00	0.317580	3	polynomial	{'deg': 3, 'option': 'polynomial'}	
6	0.000891	0.000503	-1.986957e+09	0.988757	4	other	{'deg': 4, 'option': 'other'}	
7	0.000338	0.000214	-9.964488e+01	0.598302	4	polynomial	{'deg': 4, 'option': 'polynomial'}	

Then, access the numerical value of the mean_test_score of

deg = 2 and option = ‘other’

by typing

```
df[['mean_test_score','params']].iloc[2]
```

`df[['mean_test_score','params']]` allows us to get the slice of the data frame that has only the column ‘`mean_test_score`’ and ‘`params`’.
`.iloc[2]` is get get the row number 2 of the dataframe slice put before the dot.

you should get

```
In [48]: df[['mean_test_score','params']].iloc[2]
Out[48]: mean_test_score                          -0.0258826
         params               {'deg': 2, 'option': 'other'}
         Name: 2, dtype: object
```

Required answer 10: Store the data in DataFrame into .csv file that can be later opened by Excel. Use .to_csv (method of dataframe created earlier to store the value into .csv file). For example

```
In [49]: df.to_csv('cv_details.csv')
```

create cv_details.csv the looks like the picture below when it is opened in Excel/Libre Calc

	A	B	C	D	E	F	G	H	
1		mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_deg	param_option	params	ran
2	0	0.00079197883606	0.000494575500488	-3.04255835784131	0.322870830793232	1	other	{'deg': 1, 'option': 'other'}	
3	1	0.000546932220459	0.000297546386719	-1.38514064801889	0.273332975625823	1	polynomial	{'deg': 1, 'option': 'polynomial'}	
4	2	0.000839471817017	0.000501012802124	-0.025882644305908	0.988445060127165	2	other	{'deg': 2, 'option': 'other'}	
5	3	0.000447416305542	0.000250816345215	-2.61949883970156	0.30269043012134	2	polynomial	{'deg': 2, 'option': 'polynomial'}	
6	4	0.000849103927612	0.000416994094849	-3605782.79501972	0.9885968630555	3	other	{'deg': 3, 'option': 'other'}	
7	5	0.000332450866699	0.000176811218262	-3.49267342789035	0.317580075311824	3	polynomial	{'deg': 3, 'option': 'polynomial'}	
8	6	0.000890588760376	0.00050311088562	-1986956943.57689	0.988756829001252	4	other	{'deg': 4, 'option': 'other'}	
9	7	0.000337934494019	0.000214052200317	-99.6448814230777	0.598302426874783	4	polynomial	{'deg': 4, 'option': 'polynomial'}	

Required answer 11: plot the graph between the best model from GridSearchCV of Required answer 9. The prediction can be access from .predict of the GridSearchCV object after doing `.fit`. Another way to get the prediction is to do `.predict` of the `.best_estimator_ attribute`.

Part 2: Use Stochastic Gradient Descent on the noisy data previously created.

Step 1:

Required answer 12: Create the polynomial that has 30000 data points with noise. The root of this polynomial should bet at

-2,-1, 0, 1, 2,

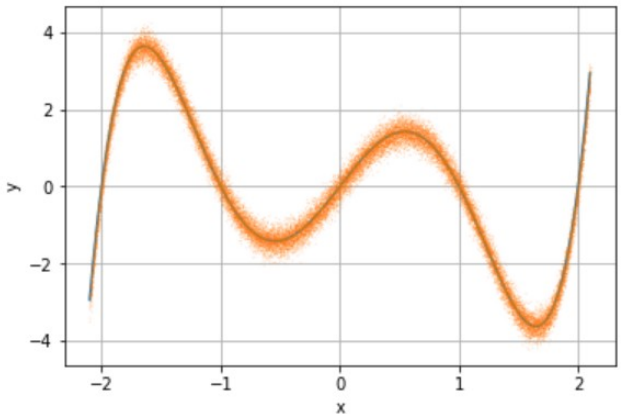
The x-data are the linear space from -2.1 to 2.1 with 30000 data points.

Then, plot the graph of your true answer without noise versus the data with noise. Use numpy.random.seed(20) so that your result is the same as mine. Put this seed command again before adding the noisy data with the no-noise data. The noise should have standard deviation of 0.2 and mean value at zero. Use np.random.normal for this noise, so that we get the normal distribution noise.

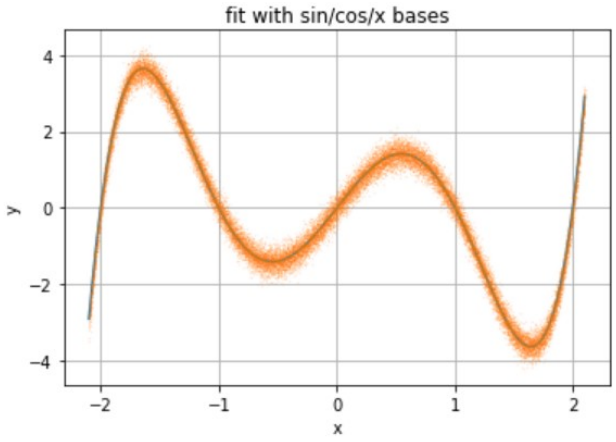
Use the knowledge that you learned in the previous tutorial to do this.

You should get

```
plt.plot(x_p2,y_p2)
plt.plot(x_p2,yn_p2, 'o', markersize = 0.1, alpha = 0.5)
plt.xlabel('x'); plt.ylabel('y')
plt.grid(); plt.show()
```



Required answer 13: Use `OtherRegress(deg = 2)` to create the estimator with sin/cos/x as the bases. Then, fit the estimator with the noisy data. After plot the prediction of estimator (of `OtherRegress`, `deg = 2`) against the noisy data, and show the coefficients from the degree 2 fitting. You should get




```
In [48]: p2_mod.coef_

Out[48]: array([  -2.21638295, -235.69885391,    3.26660342,  147.79586029,
                  -1.16267768,  -10.59722252,    1.04715888,   -1.05370527,
                 102.46108913,    0.70328027])
```

Step 2: Calculate r-squared of the prediction

Required answer 14: Calculate the r-square of the prediction by using r2_score. r2_score has to be imported from sklearn.metrics first. You should get.

```
In [52]: from sklearn.metrics import r2_score

         r2_score(yn_p2, y_pred_p2)

Out[52]: 0.98899039991317295
```

Step 3: Use Stochastic Gradient Descent to do the prediction, instead of the LinearRegression used inside the OtherRegress.

We already know that the sine, cosine, and x bases can be used to match the data. Now, we are going to use Stochastic Gradient Descent (SGD) to find the coefficient (shown earlier in `p2_mod.coef_`)

To use SGD, with sine/cosine/x bases, we need to create the basis and do the some scaling first. This is because SGD is sensitive to scaling (it works faster with scaling). To make our process reproducible easily, we create Pipeline of the preprocessing steps.

Pipeline is the easy way to combine the preprocessing steps together. Each objects that are combined in the pipeline needs to have method fit/transform/fit_transform. When we run the pipeline object, the data will go through fit_transform process of each object, except the last object will just use fit method (check sklearn website for more information).

The step in our pipeline will be

1) create basis for stochastic gradient descent regression. Specifically, we want to create the array where the first, second and third columns are sin(x), cos(x), and x.

2) create polynomial combination of the bases created in 1)
3) scale each column based on its mean and standard deviation (same as calculating z-score). After scaling, each column should have the average of 0 and standard deviation of 1.
4) check the values that are to be sent to SGD

The main reason that we need to redo the preprocessing steps easily is that whatever preprocessing we did to the training data set, we need to do the same preprocessing to the test data set in order to do the last checking step. In this tutorial, we neglect the train-test splitting step (you should do this train-test-splitting when you work with the real data)

Step 4: creation object for regression basis.

We are about to create class that has the method fit and fit_transform. When the object of this class fit_transform the data, it will change (in place) the data to be 3 column array as we needed. To make this object of this class callable in pipeline, we write

In [56]:
```
class scx_bases(BaseEstimator):
    '''sine/cosine/x bases'''
    def __init__(self):
        pass
    def fit(self,X, y = None):
        return self
    def transform(self,X, y = None): #add sin / cos column to x
        return np.c_[np.sin(X), np.cos(X), X]
    def fit_transform(self,X, y = None):
        #x need to be a column vector
        self.fit(X)
        self.ans = self.transform(X, y = None)
        return self.ans
```

Then, we create object for class scx_base, and other object for PolynomialFeatures and StandardScaler from

In [66]:
```
from sklearn.preprocessing import StandardScaler
scx = scx_bases()
poly = PolynomialFeatures(degree=2)
std_scale = StandardScaler()
```

Then, the pipeline can be created from

```
from sklearn.pipeline import Pipeline
preprocess_pipe =  Pipeline([
                ('scx', scx),
                ('poly', poly),
                ('std_scl', std_scale)])
```

To use pipeline, we just call the fit_transform of the preprocess_pipe object. The array passing to this object must be a column vector. We can do this by

```
x2_scl = preprocess_pipe.fit_transform(x_p2.reshape(-1,1))
```

Required answer 14: Show the first 10 rows of x2_scl (variable after the above preprocessing pipeline).

You should get

```
Out[85]: array([[ 0.        , -1.11091961, -1.92103527, -1.73199307,  0.3967661 ,
                  1.30050977,  1.37660828, -0.3967661 ,  2.33150879,  2.23584438],
                [ 0.        , -1.11101056, -1.92078177, -1.7318776 ,  0.39710855,
                  1.30030489,  1.37665056, -0.39710855,  2.33079524,  2.2353972 ],
                [ 0.        , -1.11110149, -1.92052825, -1.73176213,  0.39745095,
                  1.3000999 ,  1.37669276, -0.39745095,  2.33008171,  2.23495004],
                [ 0.        , -1.1111924 , -1.92027471, -1.73164666,  0.3977933 ,
                  1.2998948 ,  1.37673488, -0.3977933 ,  2.32936821,  2.23450292],
                [ 0.        , -1.11128329, -1.92002115, -1.73153119,  0.39813559,
                  1.29968961,  1.37677692, -0.39813559,  2.32865474,  2.23405583]])
```

Required answer 15: Create pandas DataFrame based on x2_scl data and show first five row. This can be done by

```
In [87]: pd.DataFrame(x2_scl).head()
```

You should get

Out[87]:

	0	1	2	3	4	5	6	7	8	9
0	0.0	-1.110920	-1.921035	-1.731993	0.396766	1.300510	1.376608	-0.396766	2.331509	2.235844
1	0.0	-1.111011	-1.920782	-1.731878	0.397109	1.300305	1.376651	-0.397109	2.330795	2.235397
2	0.0	-1.111101	-1.920528	-1.731762	0.397451	1.300100	1.376693	-0.397451	2.330082	2.234950
3	0.0	-1.111192	-1.920275	-1.731647	0.397793	1.299895	1.376735	-0.397793	2.329368	2.234503
4	0.0	-1.111283	-1.920021	-1.731531	0.398136	1.299690	1.376777	-0.398136	2.328655	2.234056

Required answer 16: Output the DataFrame of x2_scl into .csv format with the file name of "ans_16.csv" by using the knowledge learned from the previous tutorial.

Step 5: Find the parameters for SGDRegressor

SGDRegressor needs several parameters during the initialization step. It is also difficult to find good hyper-parameters that allow SGD to fit well with the data. Therefore, we use GridSearchCV to find these hyper-parameters.
The important SGD parameters are

1) eta0, used for adjusting the learning rate
2) learning rate, used for specifying the equation of the learning rate
3) max_iter, used for specifying the number of data to be used in each searching step.

Realize that we did not adjust n_iter (this attribute will be removed in the next sklearn version). n_iter is the number of time that we did the training. This is also called epoch. If we train the model more, we will get more accurate result (train too many times, can make the model to over-fit with the noise in the model).

The GridSearchCV to adjust the above hyper parameters can be done by

In [81]:
```
from sklearn.linear_model import SGDRegressor
reg = SGDRegressor(warm_start=True)
para_eta0 = [10**(-i) for i in range(1,10)]
para_max_iter = [3,5,7,9,11]
para_learning_rate = ['constant','optimal','invscaling']
param_grid = {'eta0':para_eta0, 'learning_rate':para_learning_rate,
              'max_iter':para_max_iter}
sgd_model = GridSearchCV(reg, param_grid=param_grid, scoring='r2', cv = 5)
sgd_model.fit(x2_scl,yn_p2)
```

Required answer 16: Do the GridSearchCV to find the best hyper parameters to be used for SGDRegressor, by following the above tutorial.

Your result must look like

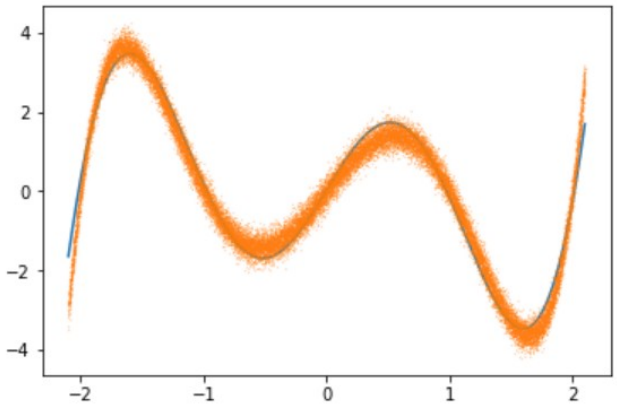
In [82]:
```
best_mod = sgd_model.best_estimator_
best_mod
```

Out[82]:
```
SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.1,
       fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
       loss='squared_loss', max_iter=7, n_iter=None, penalty='l2',
       power_t=0.25, random_state=None, shuffle=True, tol=None, verbose=0,
       warm_start=True)
```

At this point, you can get the prediction from SGD (or best_mod), by calling .predict(x2_scl). We need to pass the data that are already scaled (already gone through the pipeline).

Required answer 17: From best_mod, plot the prediction (solid line) against the nosy data. Then, calculate the r-squared value by using r2_score. Your result should look like

In [83]:
```
y2_sgd_pred = best_mod.predict(x2_scl)
plt.plot(x_p2,y2_sgd_pred)
plt.plot(x_p2,yn_p2, 'o', markersize = 0.1)
plt.show()
```



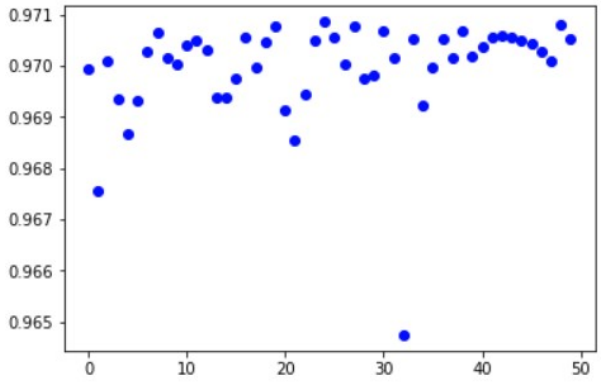
In [84]:
```
r2_score(yn_p2,y2_sgd_pred)
```

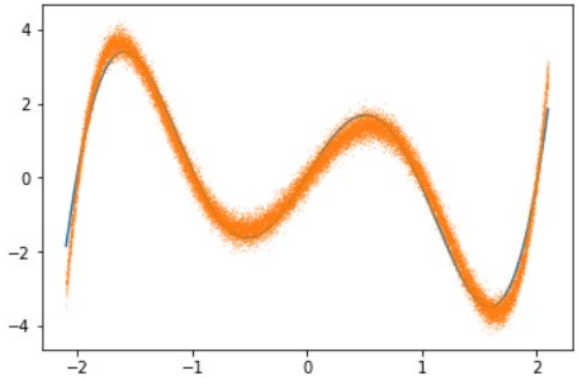
Out[84]:
```
0.96922762870277812
```

We can manually train the SGD N times (in code N = epoch = 50), with

In [136]:
```
sgd = SGDRegressor(warm_start=True,eta0=0.1
                   ,learning_rate='invscaling'
                   ,max_iter=7)
r2 = []
epoch = 50
for i in range(epoch):
    sgd.fit(x2_scl,yn_p2)
    y_pred = sgd.predict(x2_scl)
    r2.append(r2_score(yn_p2,y_pred))

plt.figure()
plt.plot(list(range(epoch)),r2,'bo')
plt.show()

plt.plot(x_p2,sgd.predict(
    preprocess_pipe.fit_transform(x_p2.reshape(-1,1))))
plt.plot(x_p2,yn_p2, 'o', markersize = 0.1)
plt.show()
```

This should gives





Required answer 18: Manually train the SGDRegressor 50 times based on the known hyper parameters from GridSearchCV as shown in the code above.

Part 3: Incremental learning

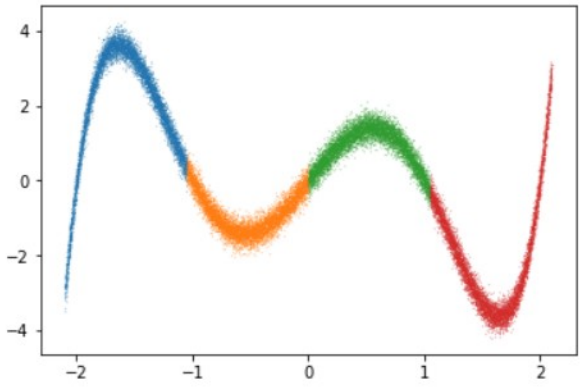
We are going to split the data into 9 files, save into .csv format. Then each part will be loaded one-by-one to train the Stochastic Gradient Descent estimator.

These .csv files contain each part of the data.

We can easily split the data into 4 files by the following code

```
In [164]: s_idx = [i * int(30000/4) for i in range(4)] + [30000]
          s_idx

Out[164]: [0, 7500, 15000, 22500, 30000]
```

```
In [165]: data = np.c_[x_p2,yn_p2]
          #data.shape = (30000,2)
          for i in range(4):
              tem = data[s_idx[i]:s_idx[i+1]]
              plt.plot(tem[:,0],tem[:,1],'o', markersize = 0.1)
              np.savetxt('data'+str(i)+'.csv',tem)
          plt.show()
```



Notice that we use for-loop in this splitting process. We created the file by using numpy.savetxt, and use the index to get unique name of each file.

Required answer 19: Split the data from part 2 (that has 30,000 row of data) into 9 file (the example above is for 4 file, but you are asked to do it for 9 file. The first column of the data must be x_p2 (x value without scaling). The second column of the data must be yn_p2 (part-2 y-value with noise).

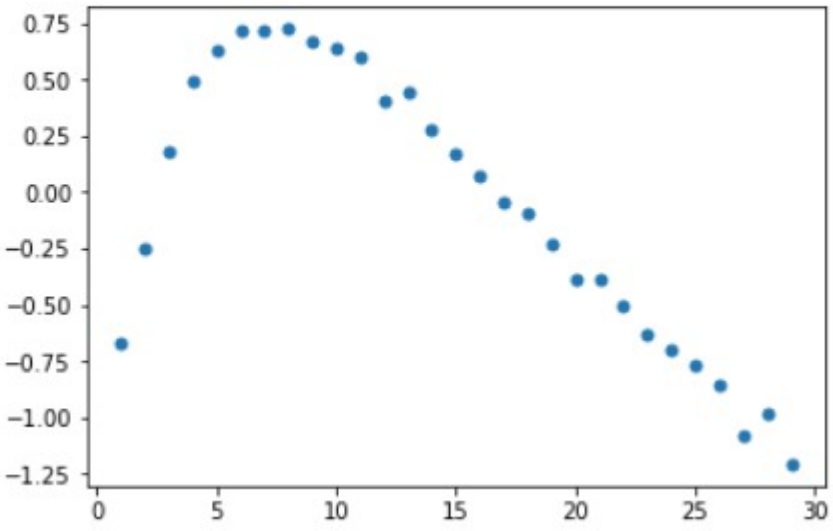
Incremental learning of SGD require a lot of data, however, we do not have much data. Yet, we are trying to do the incremental learning anyway.

Required answer 20: Do the incremental learning by typing the following code. The code below is for the case where the data were separated into 9 parts (as you did in Required answer 19).

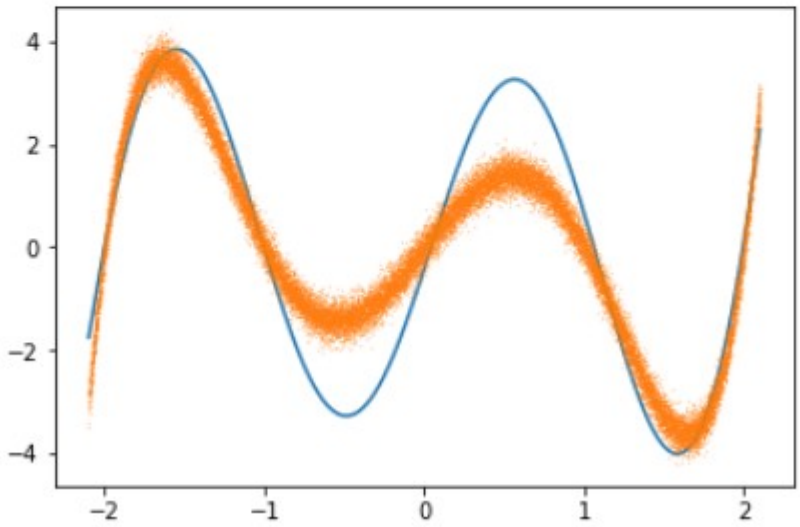
In [47]:
```
scx = scx_bases()
poly = PolynomialFeatures(degree=3)
std_scale = StandardScaler()

from sklearn.pipeline import Pipeline
preprocess_pipe =  Pipeline([
                ('scx', scx),
                ('poly', poly),
                ('std_scl', std_scale)])
```

In [48]:
```
#making the test dataset for r2 calculation
i = 0
data = np.loadtxt('data' + str(i) + '.csv')
x_ = data[:,0]
x_scl = preprocess_pipe.fit_transform(x_)
#start to open each file and run sgd.partial_fit
def r2_for_train_n_times(n, return_all = False):
    sgd = SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.001,
       fit_intercept=True, l1_ratio=0.15, learning_rate='invscaling',
       loss='squared_loss', max_iter=3, n_iter=None, penalty='l2',
       power_t=0.25, random_state=None, shuffle=True, tol=None, verbose=0,
       warm_start=True)
    #fit the preprocessing model only once, then later just use transform
    #otherwise, each minibatch has its own x_bar and SD and do different
    #preprocessing procedure
    i = 0
    data = np.loadtxt('data' + str(i) + '.csv')
    x_ = data[:,0]
    y_ = data[:,1]
    x_scl = preprocess_pipe.transform(x_)
    for it_train in range(n):
        sgd.partial_fit(x_scl,y_)

    for i in range(1, 9):
        data = np.loadtxt('data' + str(i) + '.csv')
        x_ = data[:,0]
        y_ = data[:,1]
        x_scl = preprocess_pipe.transform(x_)
        for it_train in range(n):
            sgd.partial_fit(x_scl,y_)
    x2_scl__ = preprocess_pipe.transform(x_p2)
    y_pred = sgd.predict(x2_scl__)
    r2 = r2_score(yn_p2,y_pred)
    if return_all:
        return r2, sgd
    else:
        return r2
r2_list = []
n_list = list(range(1,30))
for n in n_list:
    r2_list.append(r2_for_train_n_times(n))
```

In [49]:
```
plt.plot(n_list,r2_list,'o', markersize=5)
plt.show()
```



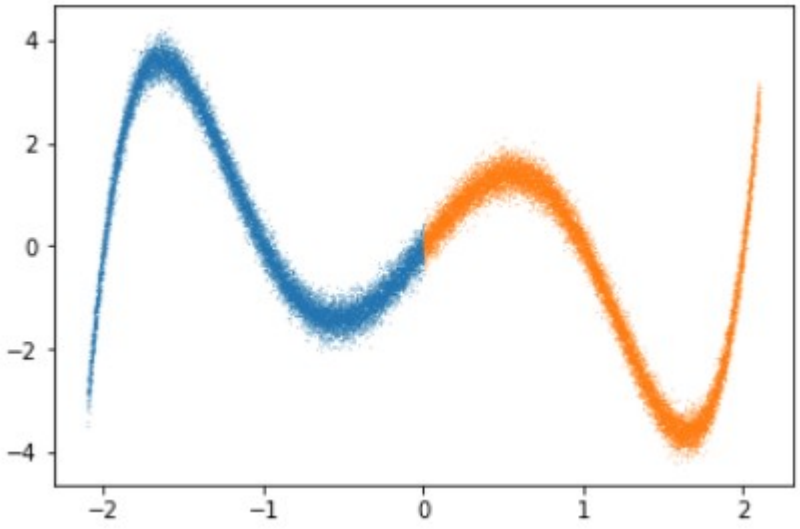
In [50]:
```
r2,sgd = r2_for_train_n_times(8, return_all=True)

x2_scl__ = preprocess_pipe.transform(x_p2)
y_pred = sgd.predict(x2_scl__)

plt.plot(x_p2, y_pred)
plt.plot(x_p2, yn_p2,'o',markersize = 0.1)
plt.show()
```



You should get the graph shown above. Notice that by seeing the data one at a time, we cannot really capture all data with SGDRegressor this way. At most we get r2 of 0.73. We stop SGD training when we get highest r2 value. SGD need some randomness in the data. However, we systematically separate the data. This means that SGD can see just one part of the data at a time. If we have some randomness in the data, then SGD can see some data at x = 1 to 2, even though the majority of the data is at -2 to -1.

Required answer 21: Instead separate the data into 9 parts, you are asked to separate the data into two parts only. Find out the number of epoch (training times) required to get the best r-squared value. Then, plot the prediction for this case. After you use 2 parts (instead of 9 parts), you may need to increase eta0 to 0.1 or do more training (increase "n" for the code above). Increase eta0 is easier to do (and faster). Your result should look like (or better than)

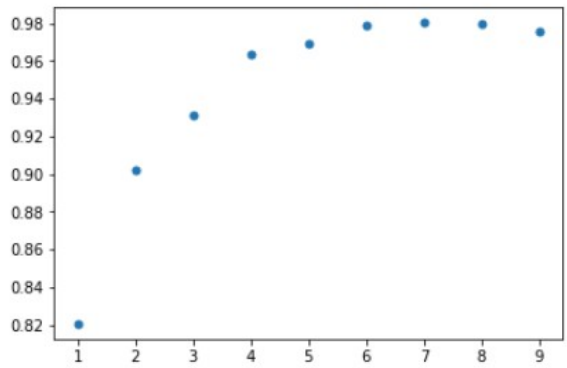
Data domain



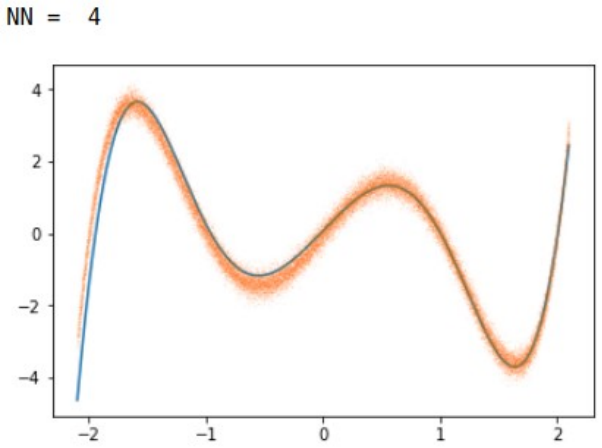
NN = 1,2,3 … is the intermediate result from the program so that we know the progress of the calculation.

```
NN =  1
NN =  2
NN =  3
NN =  4
NN =  5
NN =  6
NN =  7
NN =  8
NN =  9
```

The graph below shows the r-squared versus number of training iteration.



The picture below shows the prediction (solid blue line) versus the orange dots (noisy data) when the Stochastic gradient descent see only just half of the data at a time.



You must fit the noisy data with polynomail degree 3 (or other degree if it can give a better r2 value). The basis for the PolynomailFeatures must be sine/cosine/x as generated by scx_bases class shown earlier.

In conclusion, you have seen that even though SGDRegressor see only half of the data at a time, it can give a good match to the result.

Least-square regression cannot directly be used to do this, because it needs to see the whole data set all at once, in order to solve the system of linear equations (that you have seen from the close-form of least-square regression).