Lecture 3 - Jan 17

Data Preprocessing
Linear Basis Function Expansions

Recommended Reading

- Week 2 notes on GitHub

- *Elements of Statistical Learning*
  - Ch 2: Overview of Supervised Learning **(just a summary)**
    - 2.6.3: Function Approximation
    - 2.7: Structured Regression Models
    - 2.8.3: Classes of Restricted Estimates - Basis Functions and Dictionary Methods

- *Data Mining and Machine Learning*
  - 23.5: Kernel Regression

Note: The phrase "kernel regression" is used differently in this book -- the thing they refer to as kernel regression is "linear basis functions" in *ESL* and my notes.

Upcoming Deadlines

Python Exam (Jan 20)
Homework 1 (Jan 27)

Large Dataset Example

Example: Multivariate Regression for Air Quality

For this example, we will use the Beijing Multi-Site Air-Quality Data Data Set dataset[1] available from the UC Irvine Machine Learning Repository. It is a hourly data set considers 6 main air pollutants and 6 relevant meteorological variables at multiple sites in Beijing.

[1] Zhang, S., Guo, B., Dong, A., He, J., Xu, Z. and Chen, S.X. (2017) Cautionary Tales on Air-Quality Improvement in Beijing. *Proceedings of the Royal Society A*, Volume 473, No. 2205. https://doi.org/10.1098/rspa.2017.0457

The dataset includes a number of variables associated with time, weather, and location: the year, month, day, hour, temperature, pressure, dew point, precipitation, wind direction, wind speed, and station name.

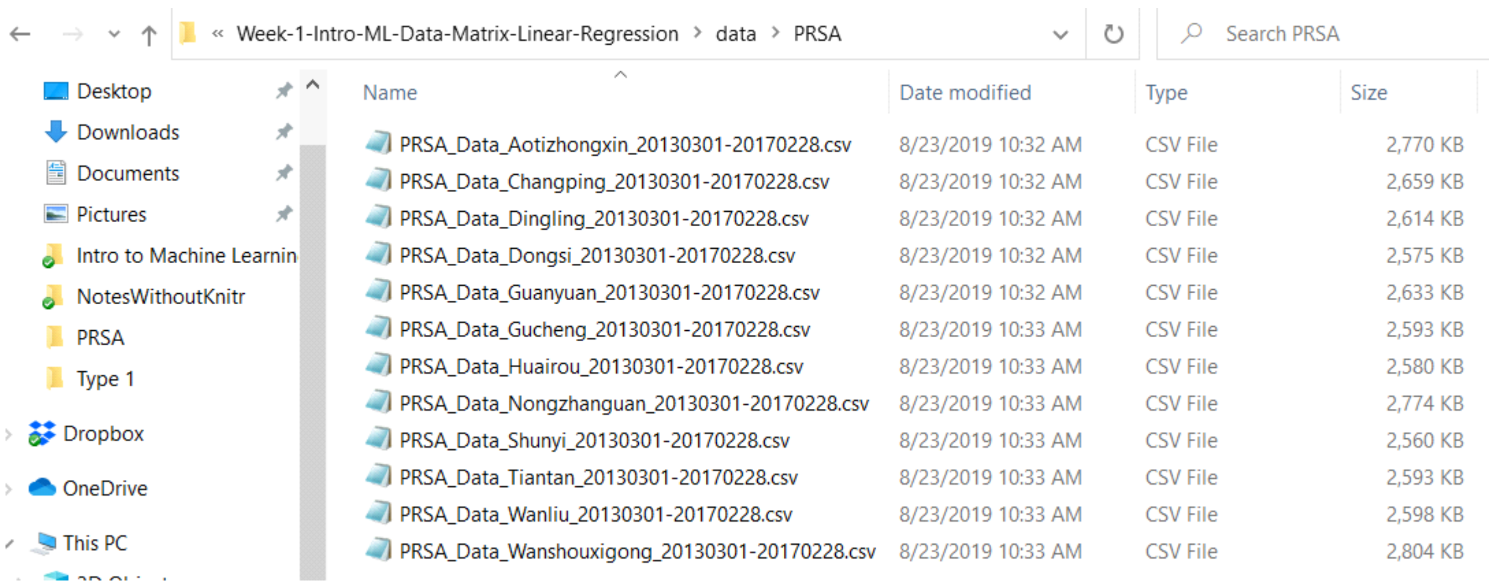
The dataset also includes air pollutant levels for: fine inhalable particulate matter with diameter $\leq 2.5\ \mu$m ($PM_{2.5}$), inhalable particles with diameter $\leq 10\ \mu$m (PM_{10}), sulfur dioxide (SO_2), nitrogen dioxide (NO_2), carbon monoxide (CO), and ozone (O_3).

We will try to predict these pollution levels based the time, weather, and location variables.

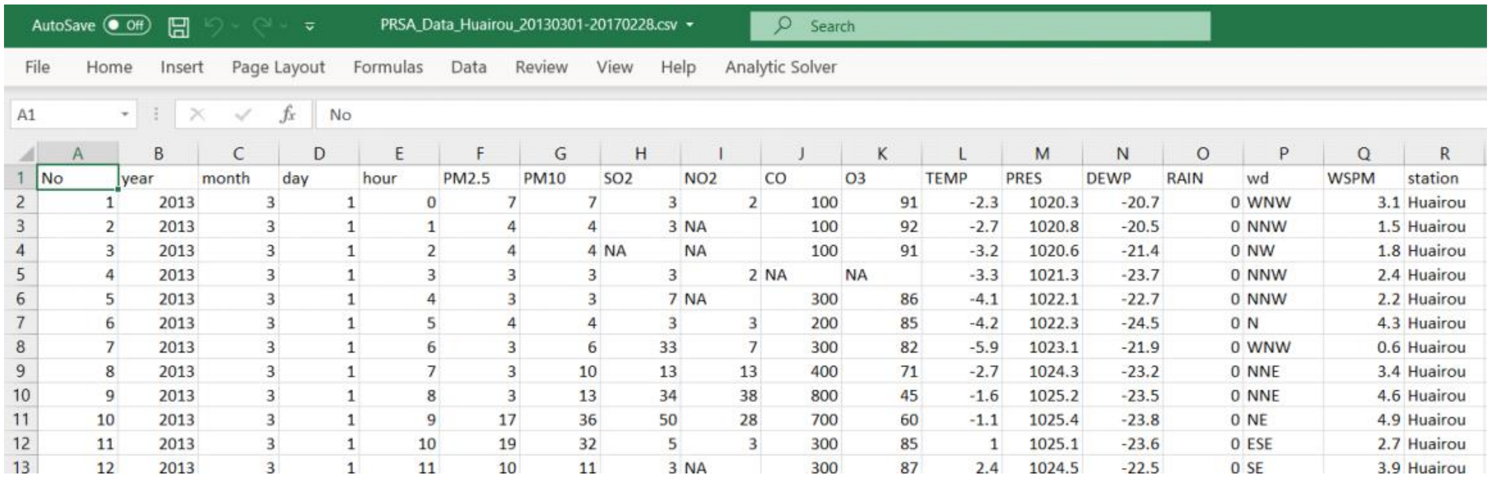
This dataset is not as neat as the US high school graduation data: there are numerical variables, but there are a few new wrinkles:

- Some variables are text
- There is missing numbers in the dataset
- There is a variable that simply indexes the data

All of these issues would prevent us from using ordinary least squares, so we need to solve these issues. In most applied problems, there are similar issues that must be managed before you can do much machine learning.



Name	Date modified	Type	Size
PRSA_Data_Aotizhongxin_20130301-20170228.csv	8/23/2019 10:32 AM	CSV File	2,770 KB
PRSA_Data_Changping_20130301-20170228.csv	8/23/2019 10:32 AM	CSV File	2,659 KB
PRSA_Data_Dingling_20130301-20170228.csv	8/23/2019 10:32 AM	CSV File	2,614 KB
PRSA_Data_Dongsi_20130301-20170228.csv	8/23/2019 10:32 AM	CSV File	2,575 KB
PRSA_Data_Guanyuan_20130301-20170228.csv	8/23/2019 10:32 AM	CSV File	2,633 KB
PRSA_Data_Gucheng_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,593 KB
PRSA_Data_Huairou_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,580 KB
PRSA_Data_Nongzhanguan_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,774 KB
PRSA_Data_Shunyi_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,560 KB
PRSA_Data_Tiantan_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,593 KB
PRSA_Data_Wanliu_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,598 KB
PRSA_Data_Wanshouxigong_20130301-20170228.csv	8/23/2019 10:33 AM	CSV File	2,804 KB



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	No	year	month	day	hour	PM2.5	PM10	SO2	NO2	CO	O3	TEMP	PRES	DEWP	RAIN	wd	WSPM	station
2	1	2013	3	1	0	7	7	3	2	100	91	-2.3	1020.3	-20.7	0	WNW	3.1	Huairou
3	2	2013	3	1	1	4	4	3	NA	100	92	-2.7	1020.8	-20.5	0	NNW	1.5	Huairou
4	3	2013	3	1	2	4	4	NA	NA	100	91	-3.2	1020.6	-21.4	0	NW	1.8	Huairou
5	4	2013	3	1	3	3	3	3	2	NA	NA	-3.3	1021.3	-23.7	0	NNW	2.4	Huairou
6	5	2013	3	1	4	3	3	7	NA	300	86	-4.1	1022.1	-22.7	0	NNW	2.2	Huairou
7	6	2013	3	1	5	4	4	3	3	200	85	-4.2	1022.3	-24.5	0	N	4.3	Huairou
8	7	2013	3	1	6	3	6	33	7	300	82	-5.9	1023.1	-21.9	0	WNW	0.6	Huairou
9	8	2013	3	1	7	3	10	13	13	400	71	-2.7	1024.3	-23.2	0	NNE	3.4	Huairou
10	9	2013	3	1	8	3	13	34	38	800	45	-1.6	1025.2	-23.5	0	NNE	4.6	Huairou
11	10	2013	3	1	9	17	36	50	28	700	60	-1.1	1025.4	-23.8	0	NE	4.9	Huairou
12	11	2013	3	1	10	19	32	5	3	300	85	1	1025.1	-23.6	0	ESE	2.7	Huairou
13	12	2013	3	1	11	10	11	3	NA	300	87	2.4	1024.5	-22.5	0	SE	3.9	Huairou

Cleaning/Preprocessing the Data

First, we must **clean** the data (manipulate it into a data matrix we can use for machine learning). The data is stored in 12 separate comma-separated value (CSV) files, so we will use the `glob` library to iterate through the files and use the `pandas` library to store each as a dataframe and then concatenate them into one big dataframe.

```
import glob

# import all the files in data/PRSA
dataFiles = glob.glob('data/PRSA/*')

# empty list to store the data
data = []

# iterate through files, read files
for file in dataFiles:
    # convert file to dataframe and add it to the list
    data.append(pd.read_csv(file, sep = ','))

# concatenate all the dataframes into one
data = pd.concat(data)

# display the data
data
```

	No	year	month	day	hour	PM2.5	PM10	SO2	NO2	CO	O3	TEMP	PRES	DEWP	RAIN	wd	WSPM	station
0	1	2013	3	1	0	4.0	4.0	4.0	7.0	300.0	77.0	-0.7	1023.0	-18.8	0.0	NNW	4.4	Aotizhongxin
1	2	2013	3	1	1	8.0	8.0	4.0	7.0	300.0	77.0	-1.1	1023.2	-18.2	0.0	N	4.7	Aotizhongxin
2	3	2013	3	1	2	7.0	7.0	5.0	10.0	300.0	73.0	-1.1	1023.5	-18.2	0.0	NNW	5.6	Aotizhongxin
3	4	2013	3	1	3	6.0	6.0	11.0	11.0	300.0	72.0	-1.4	1024.5	-19.4	0.0	NW	3.1	Aotizhongxin
4	5	2013	3	1	4	3.0	3.0	12.0	12.0	300.0	72.0	-2.0	1025.2	-19.5	0.0	N	2.0	Aotizhongxin
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
35059	35060	2017	2	28	19	11.0	32.0	3.0	24.0	400.0	72.0	12.5	1013.5	-16.2	0.0	NW	2.4	Wanshouxigong
35060	35061	2017	2	28	20	13.0	32.0	3.0	41.0	500.0	50.0	11.6	1013.6	-15.1	0.0	WNW	0.9	Wanshouxigong
35061	35062	2017	2	28	21	14.0	28.0	4.0	38.0	500.0	54.0	10.8	1014.2	-13.3	0.0	NW	1.1	Wanshouxigong
35062	35063	2017	2	28	22	12.0	23.0	4.0	30.0	400.0	59.0	10.5	1014.4	-12.9	0.0	NNW	1.2	Wanshouxigong
35063	35064	2017	2	28	23	13.0	19.0	4.0	38.0	600.0	49.0	8.6	1014.1	-15.9	0.0	NNE	1.3	Wanshouxigong

420768 rows × 18 columns

Next, we can use pandas to drop unimportant variables:

- Drop the '**No**' column with `drop()` since it just stores an index that has no physical significance.
- Drop rows with empty values with `dropna()` .

```
# drop the 'No' column
data = data.drop(columns = ['No'])

# drop rows with missing data
data = data.dropna()

# display the data
data
```

	year	month	day	hour	PM2.5	PM10	SO2	NO2	CO	O3	TEMP	PRES	DEWP	RAIN	wd	WSPM	station
0	2013	3	1	0	4.0	4.0	4.0	7.0	300.0	77.0	-0.7	1023.0	-18.8	0.0	NNW	4.4	Aotizhongxin
1	2013	3	1	1	8.0	8.0	4.0	7.0	300.0	77.0	-1.1	1023.2	-18.2	0.0	N	4.7	Aotizhongxin
2	2013	3	1	2	7.0	7.0	5.0	10.0	300.0	73.0	-1.1	1023.5	-18.2	0.0	NNW	5.6	Aotizhongxin
3	2013	3	1	3	6.0	6.0	11.0	11.0	300.0	72.0	-1.4	1024.5	-19.4	0.0	NW	3.1	Aotizhongxin
4	2013	3	1	4	3.0	3.0	12.0	12.0	300.0	72.0	-2.0	1025.2	-19.5	0.0	N	2.0	Aotizhongxin
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
35059	2017	2	28	19	11.0	32.0	3.0	24.0	400.0	72.0	12.5	1013.5	-16.2	0.0	NW	2.4	Wanshouxigong
35060	2017	2	28	20	13.0	32.0	3.0	41.0	500.0	50.0	11.6	1013.6	-15.1	0.0	WNW	0.9	Wanshouxigong
35061	2017	2	28	21	14.0	28.0	4.0	38.0	500.0	54.0	10.8	1014.2	-13.3	0.0	NW	1.1	Wanshouxigong
35062	2017	2	28	22	12.0	23.0	4.0	30.0	400.0	59.0	10.5	1014.4	-12.9	0.0	NNW	1.2	Wanshouxigong
35063	2017	2	28	23	13.0	19.0	4.0	38.0	600.0	49.0	8.6	1014.1	-15.9	0.0	NNE	1.3	Wanshouxigong

382168 rows × 17 columns

Converting a Categorical Variable to One-Hot

Note the number of datapoints went from 420768 to 382168, a loss of about 9% of the data, due to some data being incomplete. Simply dropping datapoints can sometimes bias the model, but this is a relatively small amount of data, so it should not be a big problem.

The next problem we have is that the **station** variable is not numerical but is rather categorical, representing the site where the datapoint was measured. This does not work with ordinary least squares. We could delete them from the dataset as well, but these sites are in different parts of the city which may experience different pollution patterns, so this information may help the model make predictions.

One way to deal with categorical variables in machine learning problems is to convert them to **one-hot vectors** which are standard basis vectors of $\mathbb{R}^m$ where m is the number of categories. In other words, they are made up of all 0s except for one 1, representing the one category in the datapoint.

Luckily, `Pandas` has a `get_dummies()` function, which does this conversion for us!

```
# convert the 'station' column to binary variables
data = pd.get_dummies(data, columns = ['station'])

# display the data
data
```

	year	month	day	hour	PM2.5	PM10	SO2	NO2	CO	O3	...	station_Dingling	station_Dongsi	station_Guanyuan	station_Gucheng
0	2013	3	1	0	4.0	4.0	4.0	7.0	300.0	77.0	...	0	0	0	0
1	2013	3	1	1	8.0	8.0	4.0	7.0	300.0	77.0	...	0	0	0	0
2	2013	3	1	2	7.0	7.0	5.0	10.0	300.0	73.0	...	0	0	0	0
3	2013	3	1	3	6.0	6.0	11.0	11.0	300.0	72.0	...	0	0	0	0
4	2013	3	1	4	3.0	3.0	12.0	12.0	300.0	72.0	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
35059	2017	2	28	19	11.0	32.0	3.0	24.0	400.0	72.0	...	0	0	0	0
35060	2017	2	28	20	13.0	32.0	3.0	41.0	500.0	50.0	...	0	0	0	0
35061	2017	2	28	21	14.0	28.0	4.0	38.0	500.0	54.0	...	0	0	0	0
35062	2017	2	28	22	12.0	23.0	4.0	30.0	400.0	59.0	...	0	0	0	0
35063	2017	2	28	23	13.0	19.0	4.0	38.0	600.0	49.0	...	0	0	0	0

382168 rows × 28 columns

Note the station variable has been replaced with a binary variable for each station. In all, we lost 1 column (**station**) and gained 12 more, **station_(station name)**, resulting in 28 total columns.

Wind Direction

The last problem we see is the wind direction column:

```
data['wd']
```

```
0        NNW
1          N
2        NNW
3         NW
4          N
        ...
35059     NW
35060    WNW
35061     NW
35062    NNW
35063    NNE
Name: wd, Length: 382168, dtype: object
```

These values in the **wd** column represent the wind speed. So, why not simply replace them with one-hot vectors like the **station** column?

We could, but this obscures some information in the data. These are stored as categorical variables, but they correspond to angles. If we used one-hot vectors, we may lose that physical structure.

Another option is to simply convert them to angles and store them in radians as $\theta = 0, \frac{\pi}{8}, \frac{2\pi}{8}, \ldots, \frac{15\pi}{8}$. (In reality, different angles are possible, but the original data was rouded to these values.) This leads to another problem: here, it will appear to the algorithm that, for example, the difference between an angles of 0 and $\frac{15\pi}{8}$ will be much greater than the difference between angles of 0 and π, which is not quite right in this context.

A better solution is to store the wind direction as a vector on the unit circle as $(\cos(\theta), \sin(\theta))$. So, let's write a simple function that converts radian measures to this vector.

```
# convert an angle to a list of unit circle coordinates
def unitCircle(angle):
    return [np.cos(angle), np.sin(angle)]
```

Then, let's make a dictionary that maps each direction to the appropriate angle

```
# list all wind directions, along the unit circle
directions = ['E', 'ENE', 'NE', 'NNE', 'N', 'NNW', 'NW', 'WNW', 'W', 'WSW', 'SW', 'SSW', 'S', 'SSE', 'SE', 'ESE']

# make a dictionary associating each direction with coordinates on the unit circle
directionDict = {direction : unitCircle(i*np.pi/8) for (i, direction) in enumerate(directions)}

# create a dataframe from the dictionary
directionDf = pd.DataFrame.from_dict(directionDict, orient = 'index', columns = ['unitX', 'unitY'])

# display the dataframe
directionDf
```

	unitX	unitY
E	1.000000e+00	0.000000e+00
ENE	9.238795e-01	3.826834e-01
NE	7.071068e-01	7.071068e-01
NNE	3.826834e-01	9.238795e-01
N	6.123234e-17	1.000000e+00
NNW	-3.826834e-01	9.238795e-01
NW	-7.071068e-01	7.071068e-01
WNW	-9.238795e-01	3.826834e-01
W	-1.000000e+00	1.224647e-16
WSW	-9.238795e-01	-3.826834e-01
SW	-7.071068e-01	-7.071068e-01
SSW	-3.826834e-01	-9.238795e-01
S	-1.836970e-16	-1.000000e+00
SSE	3.826834e-01	-9.238795e-01
SE	7.071068e-01	-7.071068e-01
ESE	9.238795e-01	-3.826834e-01

Our goal will be to add **unitX** and **unitY** columns to our `data` dataframe, map the existing **wd** value to the appropriate x and y coordinates, and delete the **wd** column.

`pandas` has the `join` function to take the dataframe we just created to do just this mapping.

```python
# join the direction dataframe with the data, mapping directions to unit circle coordinates
data = data.join(directionDf, on = 'wd')

# drop the wind direction column
data = data.drop(columns = ['wd'])

# display the data
data
```

	year	month	day	hour	PM2.5	PM10	SO2	NO2	CO	O3	...	station_Guanyuan	station_Gucheng	station_Huairou	station_Nongzha
0	2013	3	1	0	4.0	4.0	4.0	7.0	300.0	77.0	...	0	0	0	
1	2013	3	1	1	8.0	8.0	4.0	7.0	300.0	77.0	...	0	0	0	
2	2013	3	1	2	7.0	7.0	5.0	10.0	300.0	73.0	...	0	0	0	
3	2013	3	1	3	6.0	6.0	11.0	11.0	300.0	72.0	...	0	0	0	
4	2013	3	1	4	3.0	3.0	12.0	12.0	300.0	72.0	...	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
35059	2017	2	28	19	11.0	32.0	3.0	24.0	400.0	72.0	...	0	0	0	
35060	2017	2	28	20	13.0	32.0	3.0	41.0	500.0	50.0	...	0	0	0	
35061	2017	2	28	21	14.0	28.0	4.0	38.0	500.0	54.0	...	0	0	0	
35062	2017	2	28	22	12.0	23.0	4.0	30.0	400.0	59.0	...	0	0	0	
35063	2017	2	28	23	13.0	19.0	4.0	38.0	600.0	49.0	...	0	0	0	

382168 rows × 29 columns

Finally, our data is totally numerical, so we can now fit a regression model!

Train and Test Sets

We convert the pollutant columns to a `NumPy` array by applying the `to_numpy()` function to `data`, selecting just the pollutant columns. These are our responses, or y variables in the regression problem, `dataY`. We take the opposite columns to be the data matrix, `dataX`.

Then, we use the `train_test_split` function to randomly assign 75% of the data to the training set and 25% to the test set.

```
pollutants = ['PM2.5', 'PM10', 'SO2', 'NO2', 'CO', 'O3']

# pollutants are the responses
dataY = data[pollutants].to_numpy()

# all data except pollutants are predictors
dataX = data.drop(columns = pollutants).to_numpy()

# split the dataset and labels to 75% training set and 25% test set
trainX, testX, trainY, testY = train_test_split(dataX, dataY, test_size = 0.25, random_state = 1)
```

Let's print the dimensions of our newly created training and test data to see if it makes sense.

```
print('Training set dimensions')
print(trainX.shape)
print(trainY.shape)

print('\nTest set dimensions')
print(testX.shape)
print(testY.shape)
```

```
Training set dimensions
(286626, 23)
(286626, 6)

Test set dimensions
(95542, 23)
(95542, 6)
```

Fitting Models for Each Response Variable

Note that the training set will be quite large (286626 x 286626), which will be difficult with our simple matrix multiplication in the `OrdinaryLeastSquares` class we write, as it is likely to lead to overflows, so let's use the optimized `LinearRegression` class built into the popular `scikit-learn` library and predict each pollutant separately. If you are interested in linear algebra, it is good to know it uses singular value decomposition (SVD).

```python
# import the linear regression model
from sklearn.linear_model import LinearRegression

# instantiate an OLS model
model = LinearRegression()

for i in range(trainY.shape[1]):
    # choose the pollutant
    print('\n==================== Modeling', pollutants[i], 'pollution =====================')

    # fit the model to the training data (find the beta parameters)
    model.fit(trainX, trainY[:, i])

    # return the predicted outputs for the datapoints in the training set
    trainPredictions = model.predict(trainX)

    # print the coefficient of determination r^2
    print('The r^2 score is', model.score(trainX, trainY[:, i]))

    # print quality metrics
    print('The mean absolute error on the training set is', mean_absolute_error(trainY[:, i], trainPredictions))

    # print the beta values
    #print('The beta values are', model.beta)
    print('The beta values are', np.round(model.coef_, 2))

    # return the predicted outputs for the datapoints in the test set
    predictions = model.predict(testX)

    # print quality metrics
    print('The mean absolute error on the test set is', mean_absolute_error(testY[:, i], predictions))
```

```
==================== Modeling PM2.5 pollution =====================
The r^2 score is 0.23510056982521432
The mean absolute error on the training set is 50.4452441776565
The beta values are [ -1.63  -1.33   0.     1.4   -6.03  -1.38   3.78  -4.81  -2.93   1.    -5.7  -10.36   8.58   1.85   3.
08 -18.2    6.82   2.71   3.83   0.57   5.83   8.89 -17.08]
The mean absolute error on the test set is 50.261496050071955

==================== Modeling PM10 pollution =====================
The r^2 score is 0.16088318842493265
The mean absolute error on the training set is 60.58794671634826
The beta values are [ -2.95  -1.23   0.3    1.71  -5.89  -2.52   2.82  -5.94  -1.11   5.08 -11.56 -21.97  11.65   4.52  11.
63 -24.54   7.82  -0.21   4.36   4.07   9.15   8.82 -19.96]
The mean absolute error on the test set is 60.53901994023608

==================== Modeling SO2 pollution =====================
The r^2 score is 0.26542293236601733
The mean absolute error on the training set is 11.999224649937082
The beta values are [-4.67 -1.28 -0.    0.23 -0.78 -0.2  -0.09 -0.51 -1.76  1.68 -0.34 -3.47  2.1   1.71 -1.13 -4.76  2.93
-1.64 -1.3   2.66  1.55  2.28 -5.13]
The mean absolute error on the test set is 12.009820053231506

==================== Modeling NO2 pollution =====================
The r^2 score is 0.38335604870521767
The mean absolute error on the training set is 21.00338363899922
The beta values are [ -1.64   0.15   0.06   0.63  -2.28  -0.69   0.77  -2.17  -7.41   8.64  -5.72 -23.06   5.69   7.18   1.
95 -22.63   9.45  -4.22   4.5   12.55   5.67   3.33  -4.44]
The mean absolute error on the test set is 20.970213803565937

==================== Modeling CO pollution =====================
The r^2 score is 0.3025692914710755
The mean absolute error on the training set is 657.5228802970772
The beta values are [   1.5    10.29   -2.29   17.04  -95.96  -22.61   42.01  -46.73  -86.39   24.28  -53.2  -293.71  139.6
4   22.19   64.78 -350.93  134.72    3.43  102.02   45.62  161.16  155.14 -194.05]
The mean absolute error on the test set is 658.771412804499

==================== Modeling O3 pollution =====================
The r^2 score is 0.5271262972430173
The mean absolute error on the training set is 29.095340341815362
The beta values are [  1.51  -1.8   -0.02   0.94   3.92   0.14  -1.05   1.02   7.37  -1.61  -0.03  11.74  -2.79  -1.62   2.
47   7.51  -0.86  -0.94  -4.02  -6.2   -3.65  -1.72 -12.27]
The mean absolute error on the test set is 29.162701684609818
```

Fitting a Multivariate Regression Model

The `OrdinaryLeastSquares` class we wrote was for multiple regression (multiple inputs) but not multivariate regression (multiple outputs). We will again use the `LinearRegression` class from `scikit-learn`.

```
# instantiate an OLS model
model = LinearRegression()

# fit the model to the training data (find the beta parameters)
model.fit(trainX, trainY)

# return the predicted outputs for the datapoints in the training set
trainPredictions = model.predict(trainX)

# print the coefficient of determination r^2
print('The r^2 score is', model.score(trainX, trainY))

# print quality metrics
print('The mean absolute error on the training set is', mean_absolute_error(trainY, trainPredictions))

# return the predicted outputs for the datapoints in the test set
predictions = model.predict(testX)

# print the beta values
betas = [np.round(row, 2) for row in model.coef_]
for i in range(6):
    print('\nThe beta values for predicting', pollutants[i], 'are\n', betas[i])

# print quality metrics
print('\nThe mean absolute error on the test set is', mean_absolute_error(testY, predictions))
```

```
The r^2 score is 0.3124097213393054
The mean absolute error on the training set is 138.44233663696912

The beta values for predicting PM2.5 are
 [ -1.63  -1.33   0.     1.4   -6.03  -1.38   3.78  -4.81  -2.93   1.    -5.7  -10.36   8.58   1.85   3.08 -18.2    6.82
2.71   3.83   0.57   5.83   8.89 -17.08]

The beta values for predicting PM10 are
 [ -2.95  -1.23   0.3    1.71  -5.89  -2.52   2.82  -5.94  -1.11   5.08 -11.56 -21.97  11.65   4.52  11.63 -24.54   7.82  -
0.21   4.36   4.07   9.15   8.82 -19.96]

The beta values for predicting SO2 are
 [-4.67 -1.28 -0.    0.23 -0.78 -0.2  -0.09 -0.51 -1.76  1.68 -0.34 -3.47  2.1   1.71 -1.13 -4.76  2.93 -1.64 -1.3   2.66
1.55  2.28 -5.13]

The beta values for predicting NO2 are
 [ -1.64   0.15   0.06   0.63  -2.28  -0.69   0.77  -2.17  -7.41   8.64  -5.72 -23.06   5.69   7.18   1.95 -22.63   9.45  -
4.22   4.5   12.55   5.67   3.33  -4.44]

The beta values for predicting CO are
 [   1.5    10.29   -2.29   17.04  -95.96  -22.61   42.01  -46.73  -86.39   24.28  -53.2  -293.71  139.64   22.19   64.78 -
350.93  134.72    3.43  102.02   45.62  161.16  155.14 -194.05]

The beta values for predicting O3 are
 [  1.51  -1.8   -0.02   0.94   3.92   0.14  -1.05   1.02   7.37  -1.61  -0.03  11.74  -2.79  -1.62   2.47   7.51  -0.86  -
0.94  -4.02  -6.2   -3.65  -1.72 -12.27]

The mean absolute error on the test set is 138.61911072270158
```

Limitations of Lines, Planes, and Hyperplanes

Parameter Notation

There is an unfortunate situation common to multidisciplinary fields like machine learning: the norms of notation vary in differen subfields and related disciplines.

In particular, the notation for model parameters in machine learning unfortunately depends on the subfield where you are reading--$\boldsymbol{\beta}$ is common in statistical literature on linear regression, $\boldsymbol{w}$ or $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ are common in neural networks and deep learning literature, and $\boldsymbol{\theta}$ is common in numerical mathematics and the wider machine learning literature. As a general rule, be careful to read when the notation is introduced in books or papers. The context may not always make it clear.

Going forward, we will follow the norms of *The Elements of Statistical Learning* by Hastie, et. al., and others and refer to model parameters as $\boldsymbol{\theta_i}$ rather than $\boldsymbol{\beta_i}$ in these notes.

Limitations of Fitting Lines, Planes, and Hyperplanes

So far, we have used linear regression to find the best-fit line ($d = 1$), plane ($d = 2$), or hyperplane ($d \geq 3$) by ordinary least squares. Mathematically, this means we assumed our regression function was in the form

$$f(x_i) = \theta^T X = \theta_0 + \sum_{j=1}^{d} \theta_j x_{ij}$$

We fit the "best" line/plane/hyperplane by constructing the least squares loss function

$$L(\theta) = \sum_{i=1}^{n} (f(x_i) - y_i)^2 = \|\theta^T X - y\|_2^2$$

and solving the minimization problem

$$\min_{\theta} L(\theta)$$

which gave us a simple formula for the coefficients,

$$\theta = (X^T X)^{-1} X y$$

that we implemented and used on a few examples.

In some cases, fitting hyperplanes gave some very good results. However, sometimes these shapes just do not fit well. Examples:

- Predicting hourly temperatures in Melbourne, FL over a span of several days with a line would be a bad idea. It should oscillate as temperatures go down at night and back up in the daytime.
- Predicting the number of COVID cases in the US per day or the value of an investment over time with an interest rate with a line would not work well because we know these things grow exponentially.
- The best-fit pollution model from the example above did not explain most of the variation in the pollution data for most pollutants.

Here, fitting lines or hyperplanes does not work, so there are a number of other least squares linear models for regression that attempt to solve these types of problems that we will study over the next couple of weeks. In general, we need to solve the optimization problem

$$\min_{f,\theta} L(f, \theta)$$

for some loss function L where θ can once again be any real vector of parameters, but there is a new part: f can have a different structure. Now, without further restrictions, this problem has infinitely many solutions.

Indeed, if you consider $d = 1$, then the dataset (x_1, y_1), ..., (x_n, y_n) with distinct x_i's is just some points on the 2D plane, so there are infinitely many functions f that will give $L = 0$ when using the sum of squares loss function.

Math Note: Not only are there infinitely many solutions, but the space of functions with 0 loss is actually infinite-dimensional!

When we have the first problem, clearly we need to use a different type of function f if we hope to build a good model. With the second problem, there are broadly three classes of remedies in linear regression:

- **Basis function methods** restrict f to be within a pre-defined set of functions linear in the θ_i's but apply some nonlinear function(s) to the inputs.
- **Kernel regression methods** adjust the loss function L by weighting the sum of errors by localized properties like proximity to nearby training points.
- **Regularization methods** adjust the loss function L by adding terms to the loss function that *penalize* negative characteristics of fitted functions

to nearby training points.

- **Regularization methods** adjust the loss function L by adding terms to the loss function that *penalize* negative characteristics of fitted functions

Many approaches combine more than 1 of these ideas.

$$\cancel{\hat{f}}(x_i) = x_i^T \theta$$

$$L(\theta) = \|X\theta - y\|^2 + 3\|\theta\|^2$$

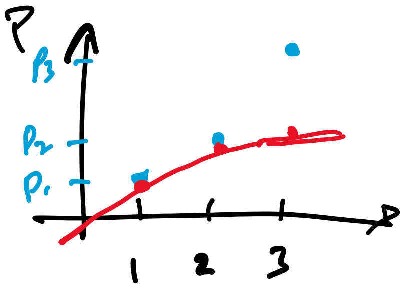
$X_1, X_2, \dots, X_n$

$$\xrightarrow{\hspace{2cm}}$$

time

$y_1, y_2, \dots, y_n$

$$L(\theta) = \sum_{i=1}^{n} \left(\hat{f}(x_i) - y_i\right)^2 \cdot \boxed{p_i} \leftarrow \text{penalty}$$



Linear Basis Function Model

A linear basis function (LBF) model is

$$\hat{f}(x_i) = \theta_0 h_0(x_i) + \cdots + \theta_d h_m(x_i)$$
$$= \sum_{m=0}^{M} \theta_m h_m(x_i) = \theta^T \cdot h(x_i)$$

← linear in the parameters $\theta_0, \theta_1, \ldots, \theta_n$ but it may be nonlinear in the data x_i

where $\underbrace{h(x_i) = \begin{bmatrix} h_0(x_i) \\ \vdots \\ h_M(x_i) \end{bmatrix}}_{h : \mathbb{R}^d \to \mathbb{R}^{M+1},\ \theta \in \mathbb{R}^{M+1}}$ and $h_0, \ldots, h_m$ are some functions we choose

The data matrix will be transformed to

$$X_h = \begin{bmatrix} h_0(x_1) & h_1(x_1) & \cdots & h_m(x_1) \\ h_0(x_2) & h_1(x_2) & \cdots & h_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(x_n) & h_1(x_n) & \cdots & h_m(x_n) \end{bmatrix}$$

$\hat{f}(X) = \theta^T X_h$, so the sum of squared errors loss function is

$$L(\theta) = \|\theta^T X_h - y\|^2$$

This is equivalent to the prior loss function $\|\theta^T X - y\|^2$, just with a different data matrix $\Rightarrow$ the optimal parameters can be computed as

$$(X^T X)^{-1} X_h y$$

computed as

$$\Theta = (X_h^T X_h)^{-1} X_h y$$

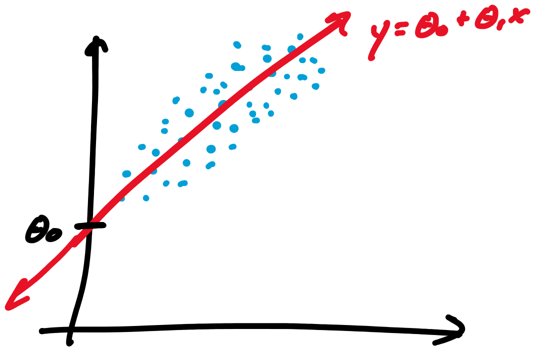
Examples

Suppose $d=1$...

① best-fit line

$h_0(x) = 1, \ h_1(x) = x$

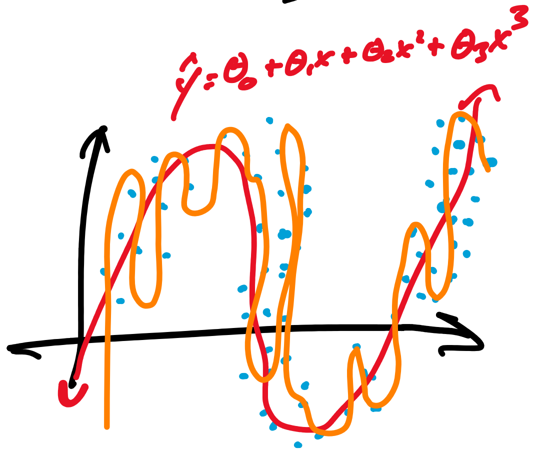
$\hat{f}(x_i) = \theta_0 + \theta_1 x_i$



② best-fit polynomial:

$h_m(x) = x^m, \ m = 1, \dots, M$

$\hat{f}(x_i) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \dots + \theta_M x^M$



exponential

$\hat{f}(x_i) = \theta_0 + \theta_1 e^{\underbrace{a}_{\text{constant}} x_i}$

logarithmic

$\hat{f}(x_i) = \theta_0 + \theta_1 \ln(x_i)$

partial Fourier series

$\hat{f}(x_i) = \theta_0 + \theta_1 \cos(x) + \theta_2 \sin(x) + \theta_3 \cos(2x) + \theta_4 \sin(2x) + \dots + \theta_{2M} \sin(Mx)$

$$\hat{f}(x_i) = \theta_0 + \theta_1\cos(x) + \theta_2\sin(x) + \theta_3\cos(2x) + \theta_4\sin(2x) + \dots + \theta_{2M}\sin(Mx)$$

$$= \theta_0 + \sum_{m=1}^{M} \theta_{2m-1}\cos(mx) + \theta_{2m}\sin(mx)$$

Suppose $d=2\dots$ $x_i = (x_{i1}, x_{i2})$

① <u>best fit plane</u>

$h_0(x_i) = 1$

$h_1(x_i) = x_{i1}$

$h_2(x_i) = x_{i2}$

$\hat{f}(x_i) = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2}$

② <u>best-fit quadratic surface</u>

$h_0(x_i) = 1$

$h_1(x_i) = x_{i1}$, $h_2(x_i) = x_{i2}$

$h_3(x_i) = x_{i1}^2$, $h_4(x_i) = x_{i1}x_{i2}$, $h_5(x_i) = x_{i2}^2$

$$\hat{f}(x_i) = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \theta_3 x_{i1}^2 + \theta_4 x_{i1}x_{i2} + \theta_5 x_{i2}^2$$

LBFs

These methods replace the function f with

$$f(x_i) = \theta^T h(x_i) = \sum_{m=0}^{M} \theta_m h_m(x_i)$$

for some user-selected functions h_0, ..., h_M and we denote

$$h(x_i) = \begin{pmatrix} h_0(x_i) \\ \vdots \\ h_M(x_i) \end{pmatrix}$$

Notice that h is a function mapping $\mathbb{R}^d$ to $\mathbb{R}^{M+1}$ for some positive integer M.

Note that, regardless of what the basis functions h_m are, f is still linear with respect to the model parameters θ_i. Thus, if we create a modified data matrix,

$$X_h = \begin{pmatrix} h_0(x_1) & h_1(x_1) & \cdots & h_M(x_1) \\ h_0(x_2) & h_1(x_2) & \cdots & h_M(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(x_n) & h_1(x_n) & \cdots & h_M(x_n) \end{pmatrix}$$

Then, we can express f as

$$f(X) = \theta^T X_h$$

and we can express the loss function as

$$L(\theta) = \left\|\theta^T X_h - y\right\|_2^2$$

This is the same loss function we minimized before except the known numerical matrix X has changed to this *preprocessed* version X_h. Still, it is contant with respect to the model parameters θ_i, so the optimal parametrs will simply change to

$$\theta = \left(X_h^T X_h\right)^{-1} X_h y$$

Implementing LBF

```
class OrdinaryLeastSquaresExact:

    # fit the model to the data
    def fit(self, X, y, ones = True):
        # add a column of ones if needed
        if ones:
            X = np.hstack((np.ones([X.shape[0],1]), X))

        # save the ones variable
        self.ones = ones

        # save the data for the class
        self.data = X

        # save the training labels
        self.outputs = y

        # find the beta values that minimize the sum of squared errors
        X = self.data
        self.theta = np.linalg.inv(X.T @ X) @ X.T @ y

    # predict the output from input (testing) data
    def predict(self, X):
        # initialize an empty matrix to store the predicted outputs
        yPredicted = np.empty([X.shape[0],1])

        # append a column of ones at the beginning of X
        if self.ones:
            X = np.hstack((np.ones([X.shape[0],1]), X))

        # apply the function f with the values of beta from the fit function to each testing datapoint
        for row in range(X.shape[0]):
            yPredicted[row] = self.theta @ X[row,]

        return yPredicted
```

```
# return polynomial basis functions for d=1
def univariatePolynomialBasis(M):
    def polynomialM(x):
        # create an empty array
        out = np.array([])

        # create the output
        for i in range(M+1):
            # append x^i
            out = np.append(out, x ** i)

        # return the polynomial values
        return out

    # return the polynomial function
    return polynomialM
```

```
poly = univariatePolynomialBasis(3)
print(np.apply_along_axis(poly, 1, [[3], [4]]))
print(poly(3))
```

```
[[ 1.   3.   9.  27.]
 [ 1.   4.  16.  64.]]
[ 1.   3.   9.  27.]
```

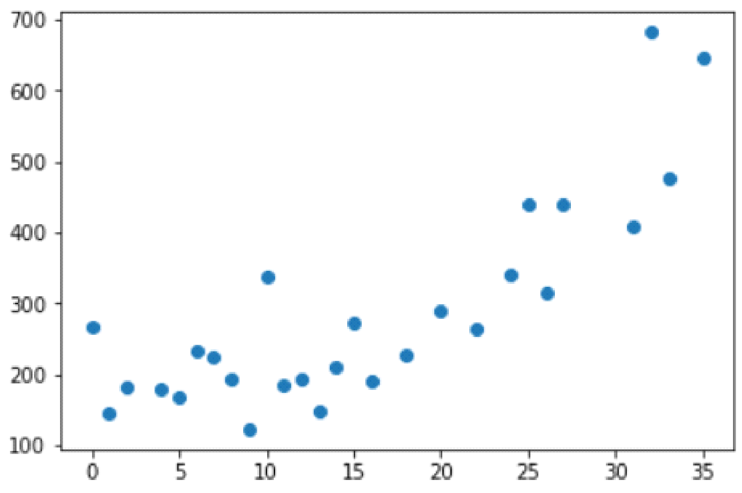
```python
from sklearn.linear_model import LinearRegression

M = 9
fig, axes = plt.subplots(nrows = M, figsize = (10, 4 * M))

for i in range(1, M + 1):
    print(i)
    poly = univariatePolynomialBasis(i)
    Xh = np.apply_along_axis(poly, 1, X)

    # fit the model
    model = LinearRegression(fit_intercept = False)
    model.fit(Xh, y)

    # predict the outputs
    predictions = model.predict(Xh)

    # print the predictions
    print('The predicted y values are', predictions.T[0])

    # print the real y values
    print('The real y values are', y)

    # print the coefficients
    parameters = model.coef_
    print('The theta values are', parameters)

    # plot the training points
    axes[i - 1].scatter(X, y, label = 'Training Data')

    # plot the fitted model with the training data
    xModel = np.atleast_2d(np.linspace(6,10,100)).T

    # compute the predicted outputs
    yModel = np.sum(parameters * np.apply_along_axis(poly, 1, xModel), axis = 1)

    axes[i - 1].plot(xModel, yModel, 'r')
```

Example

```
data = pd.read_csv('data/shampoo.csv')

y = data['Sales'].to_numpy()

X = np.zeros([y.shape[0], 1])
for i in range(y.shape[0]):
    X[i] = i

(trainX, testX, trainY, testY) = train_test_split(X, y, test_size = 0.25, random_state = 1)
```

```
plt.scatter(trainX, trainY)
```

```
<matplotlib.collections.PathCollection at 0x16852591048>
```



```python
# return polynomial basis functions for d=1
def expBasis(M):
    def exponential(x):
        # create an empty array
        out = np.array([])

        # create the output
        for i in range(-M - 1, M + 1):
            # append x^i
            out = np.append(out, np.exp(x*i/M))

        # return the polynomial values
        return out

    # return the polynomial function
    return exponential
```

```python
a = expBasis(5)
a(1)
```

```
array([0.30119421, 0.36787944, 0.44932896, 0.54881164, 0.67032005,
       0.81873075, 1.        , 1.22140276, 1.4918247 , 1.8221188 ,
       2.22554093, 2.71828183])
```

```python
from sklearn.linear_model import LinearRegression

M = 40
fig, axes = plt.subplots(nrows = M, figsize = (10, 4 * M))

for i in range(1, M + 1):
    print(i)
    exp = expBasis(i)
    Xh = np.apply_along_axis(exp, 1, X)

    # fit the model
    model = LinearRegression(fit_intercept = False)
    model.fit(Xh, y)

    # predict the outputs
    predictions = model.predict(Xh)

    # print the coefficients
    parameters = model.coef_
    #print('The theta values are', parameters)

    # plot the training points
    axes[i - 1].scatter(X, y, label = 'Training Data')

    # plot the fitted model with the training data
    xModel = np.atleast_2d(np.linspace(0,35,500)).T

    # compute the predicted outputs
    yModel = np.sum(parameters * np.apply_along_axis(exp, 1, xModel), axis = 1)

    axes[i - 1].plot(xModel, yModel, 'r')

    # return quality metrics
    print('The r^2 score is', r2_score(y, predictions))
    print('The mean absolute error is', mean_absolute_error(y, predictions))
```