Lecture 5

Generalized Additive Models

Additional Examples for Linear Models

Review – Generalized Linear Models

- Remember that a GLM has three properties
  - A value *y* generated from a distribution
  - The mean of that distribution depends on a linear combination of variables *X*
  - A link function relates the variables X to the mean μ

- A **generalized additive model** extends the GLM formula to look at combinations of smooth functions of the independent variables

$$E(Y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \quad \text{GLM}$$

$$E(Y) = \beta_0 + f_1(x_1) + f_2(x_2) + \ \ldots \quad \text{GAM}$$

$$E(Y) = \beta_0 + S_{1,1}(x_1) + S_{2,1}(x_1) + \ldots$$

Review - Smoothing

- Splines offer a versatile way to create smooth functions from data

- A spline is a summation of multiple component functions
  - P-Splines use truncated power functions (polynomials fit over a specified range)
  - B-Spline use basis functions, smooth functions defined recursively

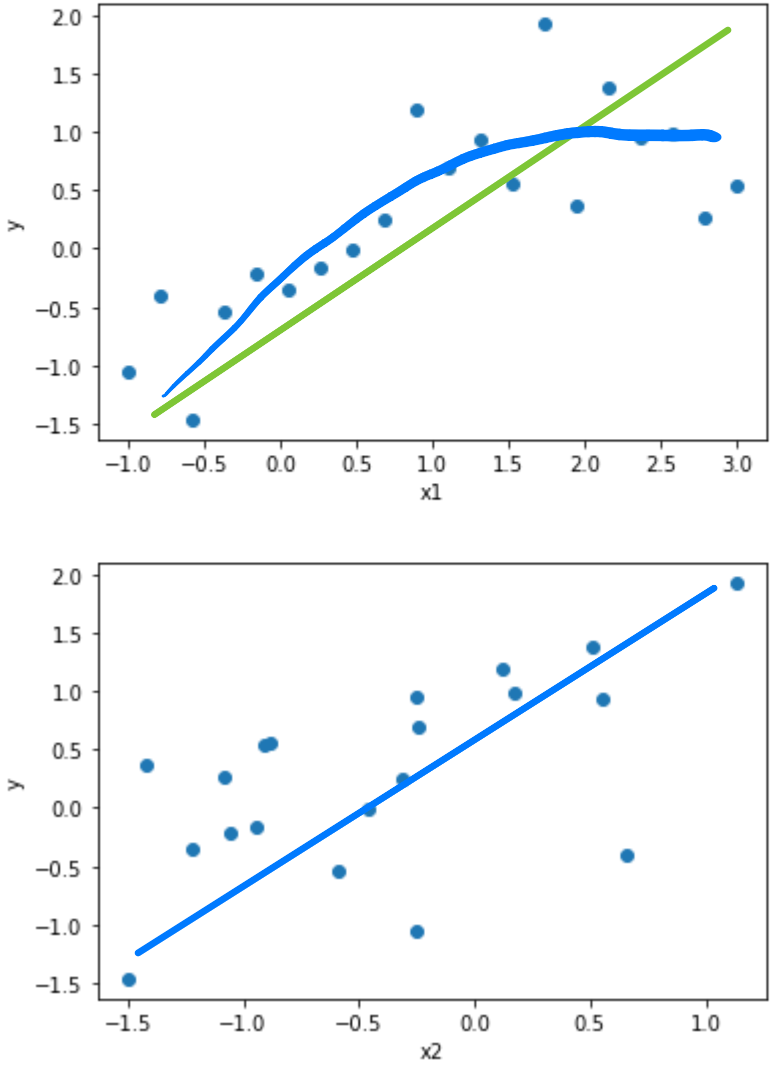
- Generalized additive models use splines to create a smooth function of the independent variable. The response variable is fit to the component functions of the spline

Example

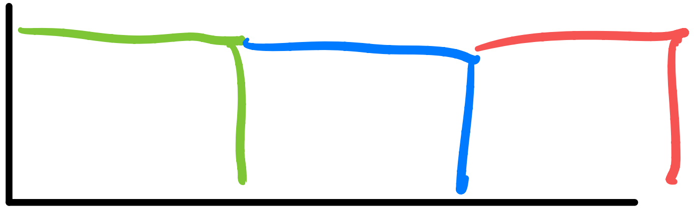
I have a response variable *y* and two predictor variables *x1* and *x2.*

I plotted *x2* against *y* and I think there is a linear relationship

Plotting *x1* against *y*, it looks like the relationship is not linear.



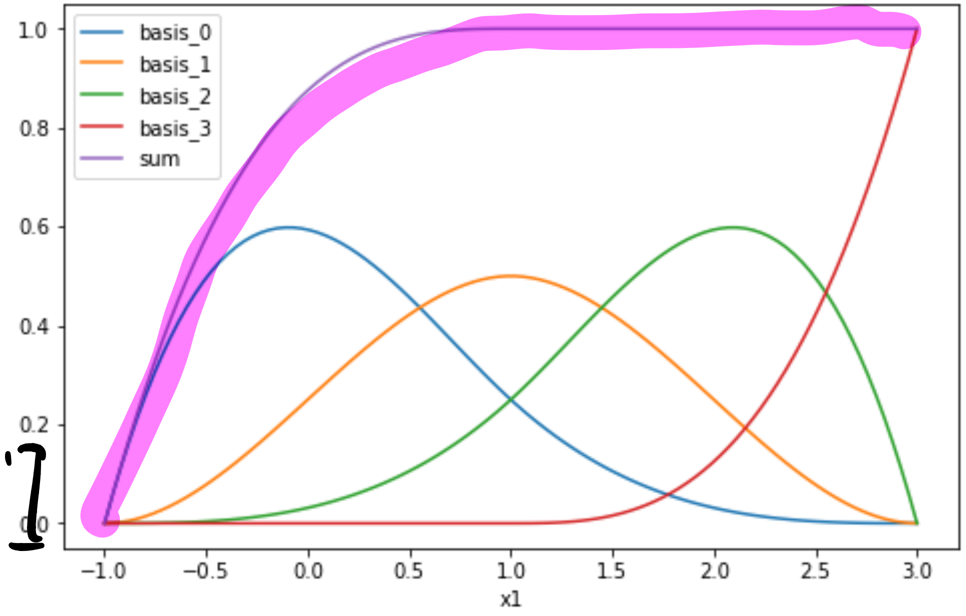
Solving using GAMs – Part 1



```
# Here we use a different API for statsmodels – statsmodels.gam.api
# We define the basis function of a B-Spline for the variable to smooth

import statsmodels.gam.api as smg
bs = smg.BSplines(df['x1'], df=[5], degree=3)
```

Spline properties

Independent variable



BSplines(df[['x1','x2']],
df=[5,5],
degree=[3,3]),

DoF = # knots + Degree

# of curves = # knots + 2

Solving using GAMs – Part 2

```
gam_bs = smg.GLMGam(y, x_df[['x2']], smoother=bs)
res_bs = gam_bs.fit()
res_bs.summary()
```

Generalized Linear Model Regression Results

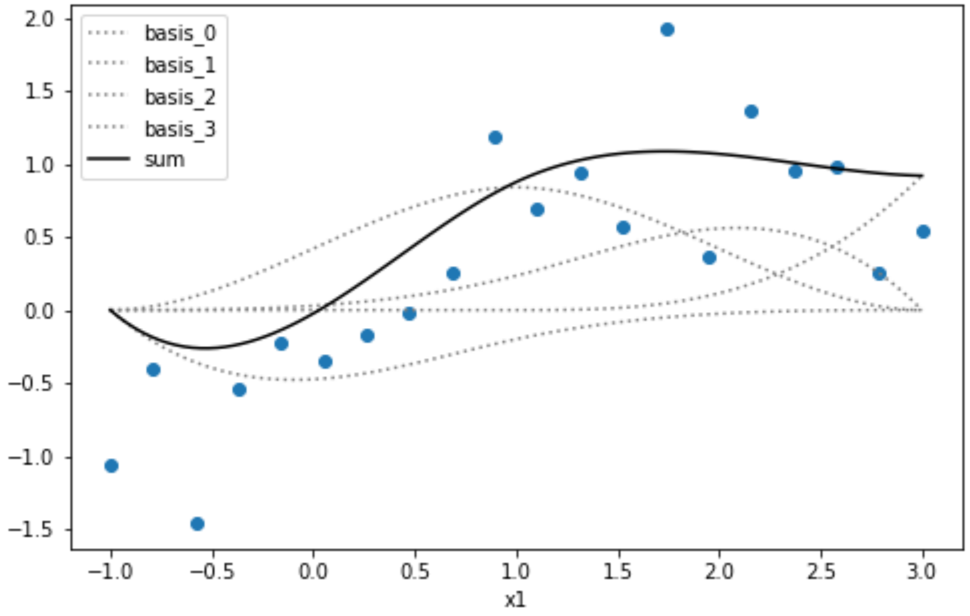
Dep. Variable:	y	No. Observations:	20
Model:	GLMGam	Df Residuals:	15.00
Model Family:	Gaussian	Df Model:	4.00
Link Function:	Identity	Scale:	0.13980
Method:	PIRLS	Log-Likelihood:	-5.8265
Date:	Wed, 28 Feb 2024	Deviance:	2.0970
Time:	14:08:00	Pearson chi2:	2.10
No. Iterations:	3	Pseudo R-squ. (CS):	0.9825
Covariance Type:	nonrobust		

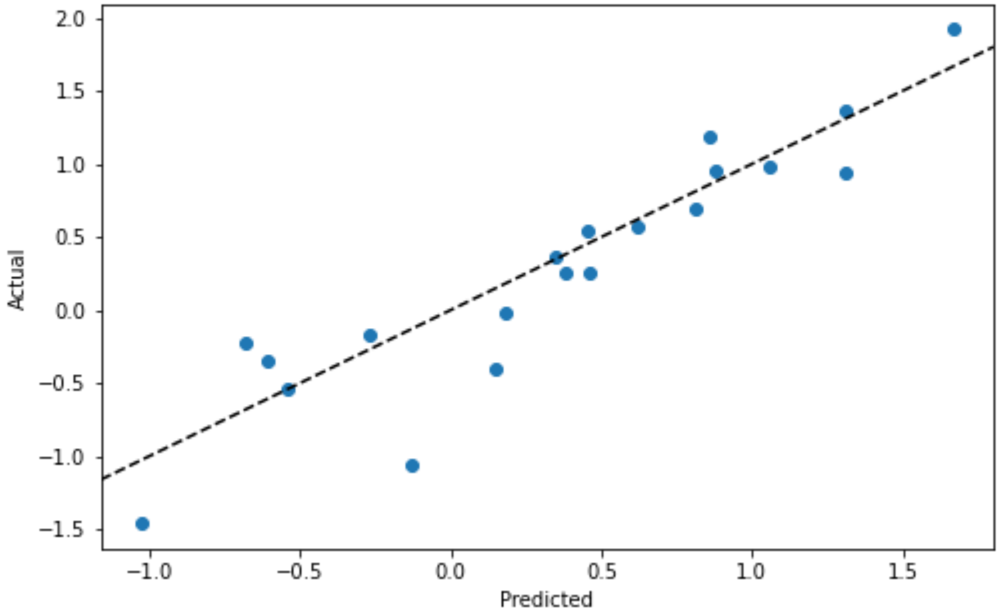
	coef	std err	z	P>|z|	[0.025	0.975]
x2	0.5116	0.136	3.767	0.000	0.245	0.778
x1_s0	-0.7979	0.477	-1.674	0.094	-1.732	0.136
x1_s1	1.6813	0.624	2.695	0.007	0.459	2.904
x1_s2	0.9405	0.460	2.045	0.041	0.039	1.842
x1_s3	0.9197	0.337	2.728	0.006	0.259	1.581

Linear ->

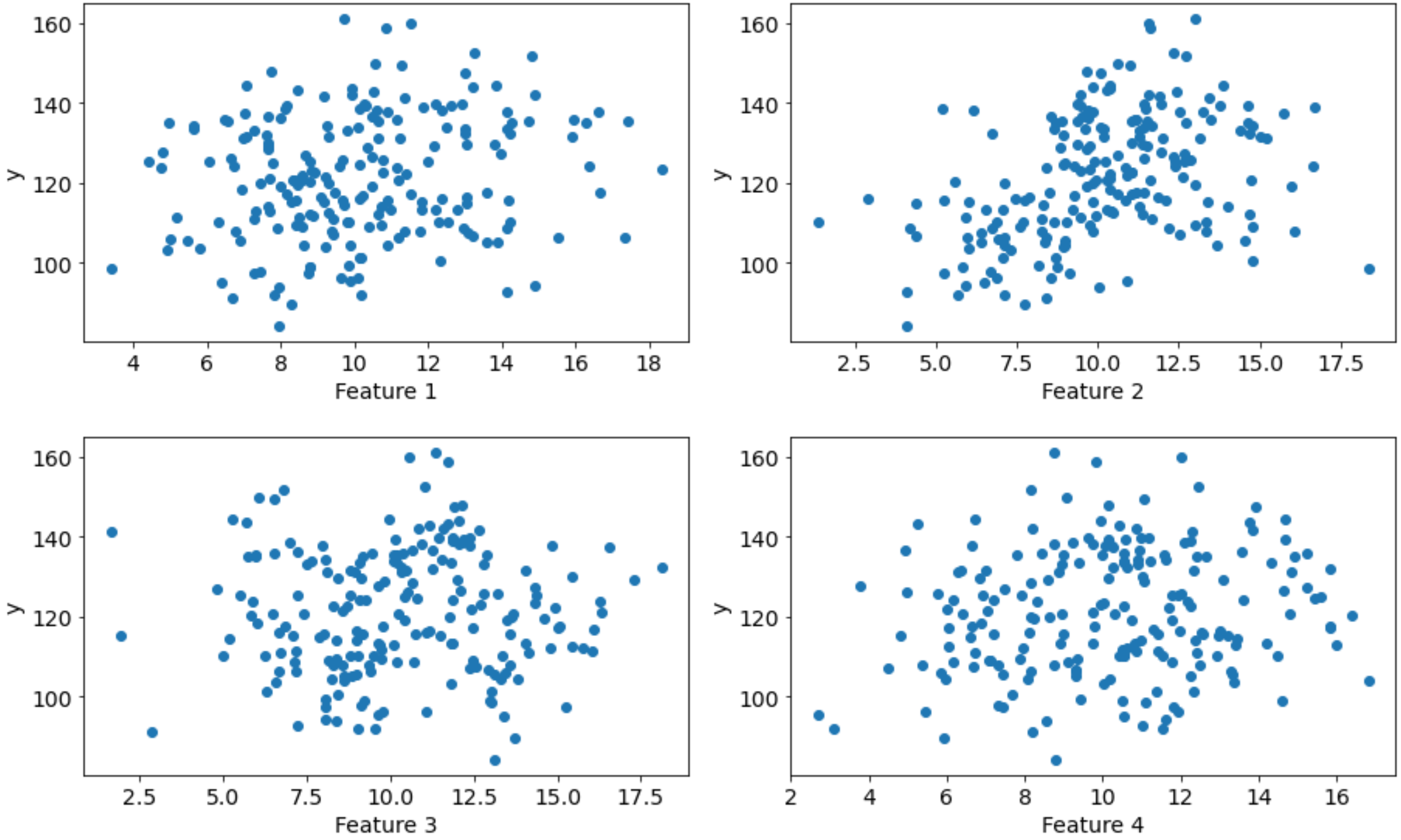
Spline

Solving using GAMs – Results





Larger GAM Example



First Attempt

```
# Create B-Splines components
bs = smg.BSplines(X_df[["X1","X2","X3","X4"]].values,
                  df=[6,6,6,6],
                  degree=[3,3,3,3],
                  variable_names=["X1","X2","X3","X4"])

# Fit both linear components and spline components
gam_bs = smg.GLMGam(y, sm.add_constant(X_df), smoother=bs)
res_bs = gam_bs.fit()
res_bs.summary()
```

First Attempt - Outputs

Generalized Linear Model Regression Results			
Dep. Variable:	y	No. Observations:	200
Model:	GLMGam	Df Residuals:	179.00
Model Family:	Gaussian	Df Model:	20.00
Link Function:	Identity	Scale:	129.12
Method:	PIRLS	Log-Likelihood:	-758.77
Date:	Thu, 29 Feb 2024	Deviance:	23112.
Time:	14:49:03	Pearson chi2:	2.31e+04
No. Iterations:	3	Pseudo R-squ. (CS):	0.6170

	coef	std err	z	P>|z|	[0.025	0.975]
const	46.9737	26.135	1.797	0.072	-4.250	98.197
X1	2.4399	1.133	2.154	0.031	0.220	4.660
X2	1.2000	0.883	1.359	0.174	-0.530	2.930
X3	0.6954	0.652	1.066	0.286	-0.583	1.974
X4	2.1265	0.769	2.765	0.006	0.619	3.634
X1_s0	28.9273	19.103	1.514	0.130	-8.513	66.368
X1_s1	-3.3080	8.287	-0.399	0.690	-19.551	12.935
X1_s2	6.9120	7.823	0.884	0.377	-8.421	22.245
X1_s3	-7.9881	9.121	-0.876	0.381	-25.865	9.889
X1_s4	-10.6272	7.084	-1.500	0.134	-24.511	3.257
X2_s0	-11.5132	15.070	-0.764	0.445	-41.049	18.023
X2_s1	-8.8954	7.006	-1.270	0.204	-22.627	4.836
X2_s2	25.0088	5.616	4.453	0.000	14.002	36.016
X2_s3	-9.5146	11.342	-0.839	0.402	-31.745	12.715
X2_s4	-7.6418	10.979	-0.696	0.486	-29.161	13.877
X3_s0	35.2909	12.479	2.828	0.005	10.833	59.748
X3_s1	-30.3752	6.379	-4.762	0.000	-42.878	-17.872
X3_s2	27.9369	5.379	5.194	0.000	17.395	38.479
X3_s3	-54.9017	8.406	-6.531	0.000	-71.377	-38.427
X3_s4	29.6096	7.202	4.111	0.000	15.493	43.726
X4_s0	11.7727	12.401	0.949	0.342	-12.533	36.078
X4_s1	5.2336	6.277	0.834	0.404	-7.068	17.535
X4_s2	4.1847	5.747	0.728	0.466	-7.078	15.448
X4_s3	-11.4696	7.743	-1.481	0.139	-26.646	3.707
X4_s4	-5.2375	6.554	-0.799	0.424	-18.084	7.609

Some spline components are okay, some aren’t. Consider reducing degree

None of these spline components fit well

After Tuning

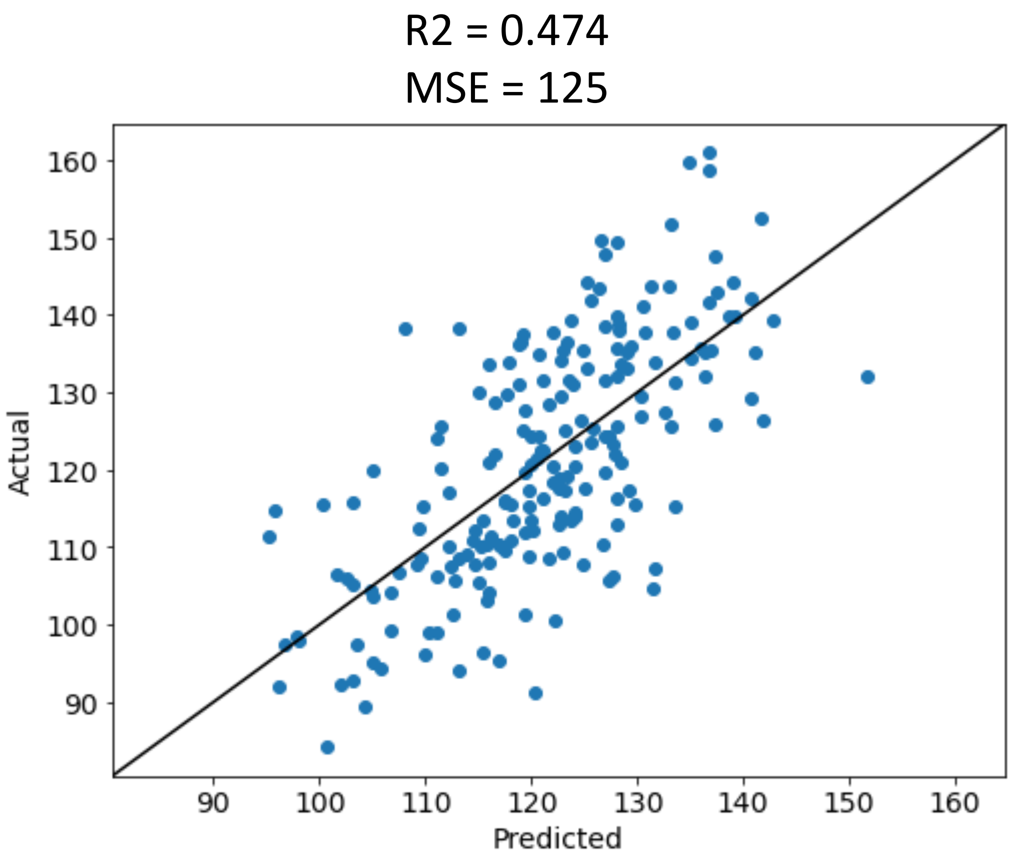
```
bs = smg.BSplines(X_df[["X1","X2","X3"]].values,
                  df=[3,4,6],
                  degree=[2,3,3],
                  variable_names=["X1","X2","X3"])

gam_bs = smg.GLMGam(y, sm.add_constant(X_df), smoother=bs)
res_bs = gam_bs.fit()
res_bs.summary()
```

End Results

Generalized Linear Model Regression Results			
Dep. Variable:	y	No. Observations:	200
Model:	GLMGam	Df Residuals:	188.00
Model Family:	Gaussian	Df Model:	11.00
Link Function:	Identity	Scale:	133.49
Method:	PIRLS	Log-Likelihood:	-767.00
Date:	Thu, 29 Feb 2024	Deviance:	25095.
Time:	14:55:34	Pearson chi2:	2.51e+04
No. Iterations:	3	Pseudo R-squ. (CS):	0.5727

	coef	std err	z	P>|z|	[0.025	0.975]
const	86.3226	14.025	6.155	0.000	58.835	113.810
X1	1.7742	0.333	5.329	0.000	1.122	2.427
X2	1.2035	0.689	1.747	0.081	-0.147	2.554
X3	0.8158	0.653	1.250	0.211	-0.463	2.095
X4	0.8471	0.291	2.911	0.004	0.277	1.418
X1_s0	-16.0966	7.507	-2.144	0.032	-30.811	-1.382
X1_s1	-11.4453	3.559	-3.216	0.001	-18.420	-4.471
X2_s0	-44.3761	15.061	-2.946	0.003	-73.894	-14.858
X2_s1	62.3833	11.006	5.668	0.000	40.811	83.955
X2_s2	-33.5788	5.023	-6.686	0.000	-43.423	-23.735
X3_s0	35.2208	12.302	2.863	0.004	11.109	59.333
X3_s1	-33.1091	6.281	-5.271	0.000	-45.420	-20.799
X3_s2	27.5168	5.249	5.242	0.000	17.229	37.805
X3_s3	-59.0373	8.318	-7.098	0.000	-75.339	-42.735
X3_s4	30.5917	7.107	4.304	0.000	16.662	44.521

End Results



Benefits of Parametric Modelling

-> Explainable

-> Fast

-> Less overfitting

Example – Feature Selection

I work in a cookie factory and I want to predict if my cookies will pass a quality control inspection.

I have a lot of data features coming from the cookie making process, but relatively few test cookies.

I want to make a model that predicts cookie failure well, without overfitting.

Reject It

	Feature 1	Feature 2	Feature 3	Feature 4	Failure
Cookie 1	99.13	70.23	117.75	88.07	False
Cookie 2	214.25	200.35	84.64	82.56	True
Cookie 3	101.75	113.47	122.20	105.54	True
…			…		

Comparing Marginal Distributions

- Split the data into two groups – one for True results and one for False results

- Calculate the marginal distribution of each feature in each group

- Use a **z-test** to check the significance of the difference between the distributions

$$Z = \frac{\mu_0 - \mu_1}{\sqrt{\sigma_0^2 + \sigma_1^2}}$$

Comparing Marginal Distributions

N = number of features

```
from scipy.stats import norm
for i in range(N):
    mu_0, sigma_0 = norm.fit(sensor_data[result, i])
    mu_1, sigma_1 = norm.fit(sensor_data[~result, i])
    z = (mu_0-mu_1) / np.sqrt(sigma_0**2 + sigma_1**2)
```

Feature	Z-Score
30	-0.509
67	-0.501
74	-0.445
44	-0.435
..	…
50	0.304
9	0.307
16	0.347
5	0.355

↑ = more negative
= less likely to fail

↓ = more positive
= more likely

Example – Ordered Responses

- Ordered values are values that take on two or more values in a way that has a defined order but undefined intervals
  - High / medium / low
  - Very unlikely / unlikely / neutral / likely / very likely

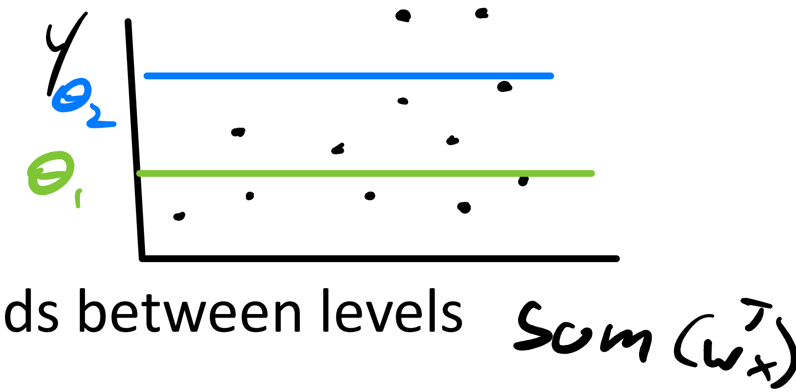
- Typical multi-class classification can work for ordered variables, but usual loss functions don’t capture the ordering information

Ordinal Regression

- Ordinal regression predicts the value of an ordered response variable

- Consider a dataset with independent variables **X** and an ordered dependent variable **y** that can take on values $i \in \{1..k\}$

- We will fit a model using two variables
  - A weight vector, **w**, as with traditional regression
  - A set of thresholds, $\{\theta_1 \ldots \theta_k\}$, that specify the thresholds between levels



latent variable

Ordinal Regression

- We define an ordinal regression model as

$$\Pr(y \leq i \mid x) = f(\theta_i - w^T x)$$

- The logit function is commonly used, giving us

$$f(\theta_i - w^T x) = \frac{1}{1 + e^{-(\theta_i - w^T x)}}$$

- We can also use the probit function (this is the default in statsmodels)

$$f(\theta_i - w^T x) = 1 - \Phi(\theta_i - w^T x)$$

Ordinal Regression in Statsmodels

```
classes = ['low', 'high', 'very high', 'medium']
class_names = ['very low','low','medium','high','very high’]

classes_cat = pd.Series(pd.Categorical(classes,
                                       ordered=True,
                                       categories=class_names))

from statsmodels.miscmodels.ordinal_model import OrderedModel
om = OrderedModel(classes_cat, X, distr='logit')
res_om = om.fit()
res_om.summary()
```

data

names

If classes is pd. Series

class_names = classes.unique()

If classes is list

class_names = set(classes)

Ordinal Regression Results

OrderedModel Results

Dep. Variable:	y	Log-Likelihood:	-69.730
Model:	OrderedModel	AIC:	157.5
Method:	Maximum Likelihood	BIC:	180.9
Date:	Thu, 29 Feb 2024		
Time:	13:54:55		
No. Observations:	100		
Df Residuals:	91		
Df Model:	5		

	coef	std err	z	P>|z|	[0.025	0.975]
x1	1.1274	0.150	7.516	0.000	0.833	1.421
x2	-0.5250	0.099	-5.314	0.000	-0.719	-0.331
x3	0.2279	0.083	2.750	0.006	0.065	0.390
x4	-0.3227	0.081	-3.980	0.000	-0.482	-0.164
x5	0.2440	0.085	2.858	0.004	0.077	0.411
very low/low	0.9199	1.760	0.523	0.601	-2.530	4.369
low/medium	1.2027	0.195	6.162	0.000	0.820	1.585
medium/high	1.4267	0.169	8.449	0.000	1.096	1.758
high/very high	1.6808	0.166	10.137	0.000	1.356	2.006

```
res_om.pred_table()
```

row_0	0	1	2	3	4	All
col_0						
0	6	3	1	0	0	10
1	2	13	5	0	0	20
2	0	4	21	5	0	30
3	0	0	6	21	3	30
4	0	0	0	4	6	10
All	8	20	33	30	9	100

Example – Negative Binomial Distribution

My bakery empire has expanded from cookies and bread to now produce fancy cakes.

I want to predict how many cakes I will need to bake so that I can meet my demand, considering that I will damage some during decorating.

I have some data on the parameters of the cake orders I have filled and the number of successful cakes.

I want to use this data to size my batches so that I can get 100 successful ~~cookies~~ cakes each time.

Flour	Oil	Sugar	Egg	Flavour	Decoration	failure
1	1.392	0.747	1	vanilla	complex	FALSE
1	0.896	0.609	2	chocolate	complex	FALSE
1	1.034	1.260	1	vanilla	plain	FALSE
1	1.293	1.097	1	sponge	very complex	FALSE
1	1.858	1.029	2	carrot	plain	TRUE
1	1.430	1.071	1	red velvet	very complex	FALSE
1	1.785	1.706	2	vanilla	complex	TRUE
1	1.809	1.256	2	sponge	plain	TRUE
1	1.095	1.098	1	carrot	complex	FALSE
1	1.166	0.763	2	sponge	simple	FALSE
1	1.263	1.138	1	chocolate	plain	FALSE
1	1.309	0.778	1	chocolate	complex	FALSE

20 failures out of 100 attempts

Step 1 – Predict success of an individual cake

- We can predict the likelihood of success of a single cake using a classification model

```
from sklearn.linear_model import LogisticRegression

X = pd.concat([pd.get_dummies(cake_df['Decoration']),
        pd.get_dummies(cake_df['Flavour']),
        cake_df[['Oil','Sugar','Flour','Egg']]], axis=1)

lr = LogisticRegression()
lr.fit(X, cake_df[‘failure'])
```

Step 2 – Check Quality of Prediction

```
from sklearn.metrics import confusion_matrix, f1_score

print(confusion_matrix(lr.predict(X), cake_df[‘failure’]))
>>> [[67 1]
     [13 19]]

print(f1_score(lr.predict(X), cake_df[‘failure’]))
>>> 0.7307692307692308
```

Step 3 – Predict on new sample

I have a new cake order to fill. The customer would like to order 50 cakes.

Flour	1
Oil	1.357
Sugar	1.022
Egg	1
Flavour	vanilla
Decoration	plain

new data

```
lr.predict_proba(X.iloc[99,:].values.reshape(1,-1))
>>> array([[0.59409458, 0.40590542]])
```

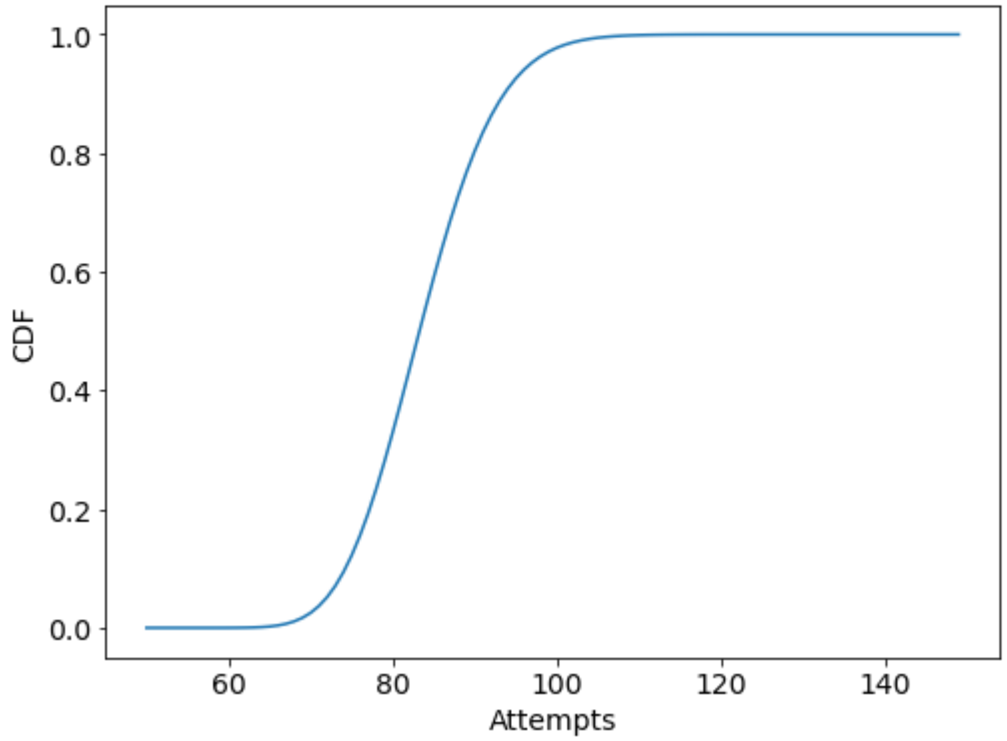
P(success)

P(failure)

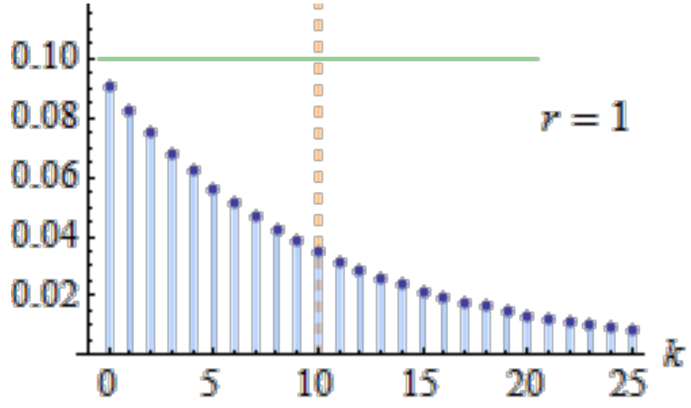
Step 4 – Build negative binomial distribution

nbinom(r, p)

```
from scipy.stats import nbinom
distribution = nbinom(50, 0.59409458)
print(distribution.ppf(0.95)+50)
>>> 97.0
```



Negative Binomial



- Parameters are number of successes (r) and success probability (p)
- Measures the number of failures in a Bernoulli process before *r* successes