Advanced Data Science

Generalised Linear Models

Carl Henrik Ek*

November 15, 2021

Abstract

In this worksheet we will look at linear models. We will show a general framework for formulating models where the we have a set of response variables that we want to explain through a linear relationship from a set of explanatory variables. While many relationships are non-linear in nature it is often challenging to interpret and explain the results from such a model. Therefore, linear models remains very important as they provide a nice trade-off between explanation of the data while at the same time providing interpretable semantics.

We have a set of explanatory variables $\mathbf{x} \in \mathcal{X}$ and a set of response variables $\mathbf{y} \in \mathcal{Y}$. The data is provided to us in pairs $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ where $\mathbf{x}_i$ and y_i are in correspondence. Our aim is to build a model that allows us to predict the response variable y_i from its corresponding explanatory variable $\mathbf{x}_i$.

Linear regression makes the assumption that the relationship between the response variable and the explanatory variable can be written as a linear combination. Furthermore, it assumes that the observations of the response variable have been corrupted by an additive Gaussian noise process,

$$\begin{aligned} y_i &= \sum_{i=1}^{d} \beta_i x_{id} + \epsilon, \\ \epsilon &\sim \mathcal{N}(0, \sigma^2). \end{aligned} \tag{1}$$

While both the explanatory and response variable are deterministic, due to the noise corrupting the observations, our predictions under the model are random. In order to extract a point estimate we therefore take the expected value,

$$\begin{aligned} \mathbb{E}[y_i \mid \mathbf{x}_i] &= \mathbb{E}\left[\sum_{i=1}^{d} \beta_i x_{id} + \epsilon\right] \\ &= \mathbb{E}\left[\sum_{i=1}^{d} \beta_i x_{id}\right] + \mathbb{E}[\epsilon] \\ &= \sum_{i=1}^{d} \beta_i x_{id} + 0. \end{aligned} \tag{2}$$

Traditionally the linear regression model above is motivated by an additive noise assumption that corrupts the observed response. An equivalent explanation is to absorb the noise directly into the response variable

*che29@cam.ac.uk

and consider it a linear model of a normal distributed response,

$$\begin{aligned} y_i &= \sum_{i=1}^{d} \beta_i x_{id} + \epsilon, \\ y_i + \epsilon &= \sum_{i=1}^{d} \beta_i x_{id}, \\ \hat{y}_i &= \sum_{i=1}^{d} \beta_i x_{id}, \\ \hat{y}_i &\sim \mathcal{N}(y_i, \sigma^2) = \mathcal{N}\left(\sum_{i=1}^{d} \beta_i x_{id}, \sigma^2\right), \end{aligned} \tag{3}$$

where, as we previously derived, the linear predictor directly parametrises the mean or the first moment of the random response variable.

If we want to generalise the setting we can think of scenarios where the response variable follows a different distribution this could be a Bernoulli distributed response as in binary classification or Poisson distributed if we are describing discrete events. In specific we will look at models where the first moment of the response variable can be parametrised as a function of a linear combination of the explanatory variables,

$$g(\mathbb{E}[y_i \mid \mathbf{x}_i]) = \sum_{i=1}^{d} \beta_i x_{id}, \tag{4}$$

where the function $g(\cdot)$ is known as a *link-function* connecting the expected value and the linear predictor[1]. While the above formulation is the one that is most commonly used in the statistics literature, in a machine learning setting we can think of a tranformation of a linear mapping as,

$$\mathbb{E}[y_i \mid \mathbf{x}_i] = g^{-1}(\sum_{i=1}^{d} \beta_i x_{id}), \tag{5}$$

if you are familiar with simple compositional function models such as neural networks you can see how these are recursive formulations of similar structures[2].

The class of models of models that can be described using Eq. 4 are commonly referred to as Generalised linear models (GLM) McCullagh et al., 1989 where the generalisation comes from the fact that we consider the response variable to follow a Exponential Dispersion Family distribution [3]. The family is rich class of probability distributions that contains most of the distributions that you should be familiar working with. Each distribution is defined through two parameters, the location parameter and a scale parameter. The link function plays the role of "linking" the value of the linear predictor to the location parameter of the distribution. Many classical statistical models can be written in the language of generalised linear models in Table 1 a list of models that falls into the category is shown. Each model is characterised by three different components, an exponential dispersion family distribution of the response, a link function and a linear predictor from the explanatory variables.

The exponential dispersion family can be written on the following canonical form,

$$f(y; \theta, \phi) = e^{\frac{\theta y - b(\theta)}{a(\phi)} + c(y, \phi)}, \tag{6}$$

[1] for the linear regression case above the link function is the identity

[2] An article that describes neural networks in the light of the models we will describe can be found here `https://towardsdatascience.com/glms-part-iii-deep-neural-networks-as-recursive-generalized-linear-URL`.

[3] `https://en.wikipedia.org/wiki/Exponential_dispersion_model`

Model	Response Variable	Link	Explanatory Variable
Linear Regression	Normal	Identity	Continuous
Logistic Regression	Binomial	Logit	Mixed
Poisson Regression	Poisson	Log	Mixed
ANOVA	Normal	Identity	Categorical
ANCOVA	Normal	Identity	Mixed
Loglinear	Poisson	Log	Categorical
Multinomial response	Multinomial	Generalized Logit	Mixed

Table 1: *Specific instances of Generalized linear Model URL.*

where θ is the location and ϕ the scale parameter respectively. A special instantiation of is the Gaussian distribution,

$$f(y; \mu, \sigma^2) = e^{\frac{\mu y - \frac{1}{2}\mu^2}{\sigma^2} - \frac{y^2}{2\sigma^2} - \frac{1}{2}\ln(2\pi\sigma^2)} \tag{7}$$

where we can identify,

$$\begin{aligned} \theta &= \mu \\ \phi &= \sigma^2 \\ b(\theta) &= \frac{1}{2}\theta^2 \\ a(\phi) &= \phi \\ c(y, \phi) &= \frac{y^2}{2\phi} - \frac{1}{2}\ln(2\pi\phi) \end{aligned} \tag{8}$$

The benefit of writing the distribution on the general form is that we can easily derive a general expression for both the first and the second moment of the predictive distribution,

$$\begin{aligned} \mathbb{E}[y \mid \mathbf{x}] &= \frac{\partial}{\partial\theta} b(\theta) \\ \mathbb{V}[y \mid \mathbf{x}] &= a(\phi)\frac{\partial^2}{\partial\theta^2} b(\theta). \end{aligned} \tag{9}$$

This means that given a GLM we can compute predictions and our uncertainty about those predictions in closed form. We will now proceed to look at how we can fit these models to data and learn the parameters $\boldsymbol{\beta}$ that connects the explanatory variables with the response.

1 Learning Generalised Linear Models

Fitting the GLM models described above to data implies learning the parameters $\boldsymbol{\beta}$ of the linear predictor. We will do so by finding the maximum likelihood estimate. By specifying the distribution of the response variable we have in effect specified a likelihood of each individual data-point. To reach the joint likelihood of each response variable we will make the assumption that each observed response is conditionally independent given the parameters and the explanatory variables. This leads to the following objective function,

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmax}} \prod_{i=1}^{N} p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}). \tag{10}$$

GLM models are very well studied in the literature and many different highly efficient approaches exists for fitting the parameters to data. While this is an interesting topic in itself it is beyond the scope of this course. Instead we will put our trust in the very well documented library `https://www.statsmodels.org` rather than implementing this ourselves.

Below the code for implementing two separate GLM models is shown. The data is count data and the most suitable model would be a Poisson regression model for this type. In addition we also include a standard Gaussian linear model with a single parameter parametrising the slope of the line.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

# generate data
x = np.array([[i] for i in np.random.choice(range(1,10), 20)])
y = np.array([np.exp(i) + np.random.normal(0,0.25*np.exp(i),1)[0] for i in x])

# make sure that the values are positive
y[y<0] = 0

# create models
m_poisson = sm.GLM(y,x, family=sm.families.Poisson())
m_poisson_results = m_poisson.fit()

m_gaussian = sm.GLM(y,x, family=sm.families.Gaussian())
m_gaussian_results = m_gaussian.fit()

# prediction
x_pred = np.arange(1,11).reshape(-1,1)
y_pred = m_poisson_results.get_prediction(x_pred).summary_frame(alpha=0.05)

y_pred_gaussian = m_gaussian_results.get_prediction(x_pred).summary_frame(alpha=0.05)


# plot results
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

plt.scatter(x,y,marker='X',color='blue',edgecolor='black',s=100,alpha=0.5,zorder=1)

plt.plot(x_pred,y_pred['mean'],color='red',linewidth=3.0,zorder=2)
plt.plot(x_pred,y_pred['mean_ci_lower'], color='red',linestyle='--',zorder=2)
plt.plot(x_pred,y_pred['mean_ci_upper'], color='red',linestyle='--',zorder=2)

plt.plot(x_pred,y_pred_gaussian['mean'],color='cyan',linewidth=3.0,zorder=2)
plt.plot(x_pred,y_pred_gaussian['mean_ci_lower'], color='cyan',linestyle='--',zorder=2)
plt.plot(x_pred,y_pred_gaussian['mean_ci_upper'], color='cyan',linestyle='--',zorder=2)


plt.tight_layout()
```

The important part of the code above is the `m_poisson = sm.GLM(y,x, family=sm.families.Poisson())` call where we specify the distribution of the of the response variable. `Statsmodels` currently implements the following distributions, `Binomial`, `Gamma`, `Gaussian`, `InverseGaussian`, `NegativeBinomial`, `Poisson`, `Tweedie` which should give you a rich playground of models to work with.

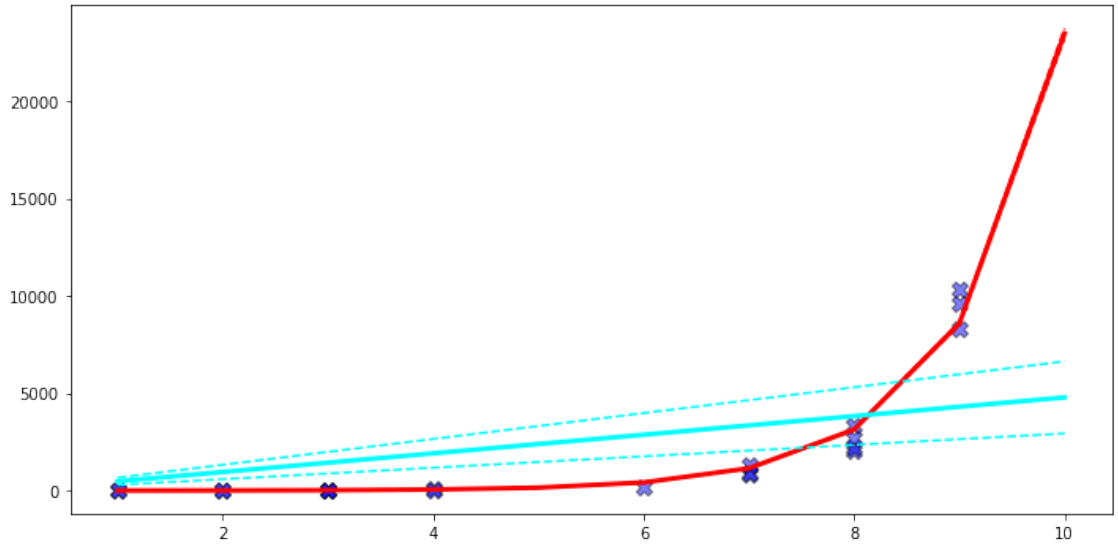

Figure 1: *The figure above shows the output of fitting a Poisson regression model in red and the Gaussian model in cyan. The dotted lines shows the* 95% *confidence interval for the* ***parameters*** *of the fit. As we can see the fit is (not surprisingly) much worse for the Gaussian model compared to the Poisson.*

2 Basis-function Models

In the framework we have described above the linear prediction is directly on the explanatory variables. However we can generalise this slightly if we instead want to perform the regression over a set of basis functions. In statistics we often refer to the matrix of the explanatory variables as the *design matrix* of the data. We can easily "design" a matrix that maps the explanatory variables to a "feature space" and then perform the regression over this domain instead. In Figure 1 the cyan plot uses only a single parameter in its prediction and is therefore only able to parametrise lines that intersect at $(0, 0)$, in order to also parametrise the intersection of the line in the model we require to expand the input domain with a constant one as,

$$\mathbf{X} = \begin{bmatrix} x_0 & 1 \\ x_1 & 1 \\ \vdots & \vdots \\ x_N & 1 \end{bmatrix}.$$

Using the above as the explanatory variables we will be able to fit both the intersection and the slope. However, we can take this concept further, and generalise this to do regression over a different expanded explanatory space. Say that we know that the relationship between the response and the explanatory variables is,

$$y = \beta_0 \sin(x) + \beta_1 \sin(\frac{x^2}{40}) + \beta_2 x.$$

We can now create a new design matrix that includes all the functions that we need and apply the `GLM` framework as before,

$$\mathbf{X} = \begin{bmatrix} \sin(x_0) & \sin(\frac{x_0^2}{40}) & x_0 \\ \sin(x_1) & \sin(\frac{x_1^2}{40}) & x_1 \\ \vdots & \vdots & \vdots \\ \sin(x_N) & \sin(\frac{x_N^2}{40}) & x_N \end{bmatrix}.$$

Below we will implement this using the `OLS` model which is just a Gaussian response and an identity link function. The motivation for using `OLS` rather than `GLM` is that the former provides easier means of analysing predictions.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

np.random.seed(42)

x = np.linspace(0,15,50)
y = 0.2*np.sin(x) + 0.5*np.sin(x**2/40) + 0.05*x
y += 0.05*np.random.randn(x.shape[0])

m_linear = sm.OLS(y,x)
results = m_linear.fit()

design = np.concatenate((np.sin(x).reshape(-1,1),
                         np.sin(x**2/40).reshape(-1,1), x.reshape(-1,1)),axis=1)
m_linear_basis = sm.OLS(y,design)
results_basis = m_linear_basis.fit()

x_pred = np.linspace(-5,20,200).reshape(-1,1)
design_pred = np.concatenate((np.sin(x_pred), np.sin(x_pred**2/40), x_pred),axis=1)
y_pred_linear = results.get_prediction(x_pred).summary_frame(alpha=0.05)
y_pred_linear_basis = results_basis.get_prediction(design_pred).summary_frame(alpha=0.05)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)
ax.scatter(x,y,zorder=2)
ax.plot(x_pred, y_pred_linear['mean'], color='red',linestyle='--',zorder=1)
ax.plot(x_pred, y_pred_linear['obs_ci_lower'],color='red',linestyle='-',zorder=1)
ax.plot(x_pred, y_pred_linear['obs_ci_upper'],color='red',linestyle='-',zorder=1)
ax.fill_between(x_pred.flatten(), y_pred_linear['obs_ci_lower'],
                y_pred_linear['obs_ci_upper'],color='red',alpha=0.3,zorder=1)

ax.plot(x_pred, y_pred_linear_basis['mean'], color='cyan',linestyle='--',zorder=1)
ax.plot(x_pred, y_pred_linear_basis['obs_ci_lower'],color='cyan',
        linestyle='-',zorder=1)
ax.plot(x_pred, y_pred_linear_basis['obs_ci_upper'],color='cyan',
        linestyle='-',zorder=1)
ax.fill_between(x_pred.flatten(), y_pred_linear_basis['obs_ci_lower'],
                y_pred_linear_basis['obs_ci_upper'],color='cyan',alpha=0.3,zorder=1)

plt.tight_layout()
```

If we look at the actual values of the $\boldsymbol{\beta}$ that the method returns Tab **??** we can see that the approach recovers the values of the coefficients used to generate the data.

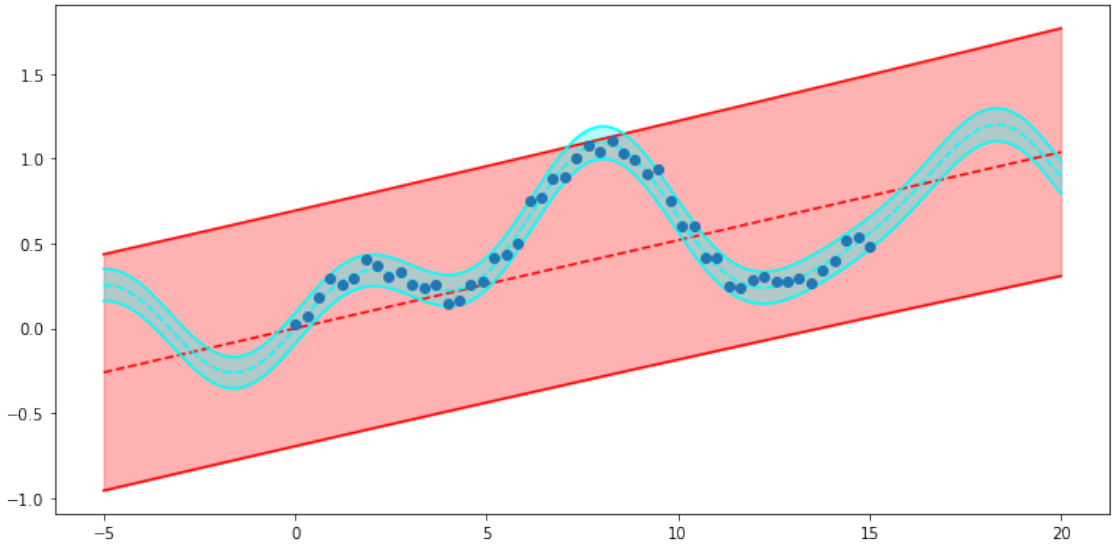

Figure 2: *The figure above shows fitting a linear Gaussian model to a non-linear function. The prediction in red shows the model where the design matrix is directly on the explanatory variables while the cyan uses a design matrix of non-linear functions.*

Code

```
print(results_basis.summary())
```

coef	std	err
x1	0.2155	0.009
x2	0.4956	0.010
x3	0.0482	0.001

Furthermore, if we look at the `log-likelihood` of the two models we can see that it is substantially higher for the second models compared to the first. This indicates, as we can see visually, that the second design matrix results in a much better fit to the data.

While the example above worked well maybe it felt a bit contrived as we basically needed to know the function a-priori. So how about

Question 1

What happens if you expand the design matrix with a `sin(x**2/20)` and a `sin(x**3)` term? Why do you get the results that you get?

Question 2

What happens if you create a design matrix as follows,

$$y = \beta_0 \sin(x) + \beta_1 \sin(\frac{x^2}{40}) + \beta_2 x + \beta_0(-\sin(x)).$$

Explain why you get the coefficients you do.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

np.random.seed(42)

x = np.linspace(0,15,50)
y = 0.2*np.sin(x) + 0.5*np.sin(x**2/40) + 0.05*x
y += 0.05*np.random.randn(x.shape[0])

m_linear = sm.OLS(y,x)
results = m_linear.fit()

design = np.concatenate((np.sin(x).reshape(-1,1),
                         np.sin(x**2/40).reshape(-1,1), x.reshape(-1,1),
                         -np.sin(x).reshape(-1,1)),axis=1)
m_linear_basis = sm.OLS(y,design)
results_basis = m_linear_basis.fit()

x_pred = np.linspace(-5,20,200).reshape(-1,1)
design_pred = np.concatenate((np.sin(x_pred), np.sin(x_pred**2/40), x_pred,
                              -np.sin(x_pred)),axis=1)
y_pred_linear = results.get_prediction(x_pred).summary_frame(alpha=0.05)
y_pred_linear_basis = results_basis.get_prediction(design_pred).summary_frame(alpha=0.05)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)
ax.scatter(x,y,zorder=2)
ax.plot(x_pred, y_pred_linear['mean'], color='red',linestyle='--',zorder=1)
ax.plot(x_pred, y_pred_linear['obs_ci_lower'],color='red',linestyle='-',zorder=1)
ax.plot(x_pred, y_pred_linear['obs_ci_upper'],color='red',linestyle='-',zorder=1)
ax.fill_between(x_pred.flatten(), y_pred_linear['obs_ci_lower'],
                y_pred_linear['obs_ci_upper'],color='red',alpha=0.3,zorder=1)

ax.plot(x_pred, y_pred_linear_basis['mean'], color='cyan',linestyle='--',zorder=1)
ax.plot(x_pred, y_pred_linear_basis['obs_ci_lower'],color='cyan',
        linestyle='-',zorder=1)
ax.plot(x_pred, y_pred_linear_basis['obs_ci_upper'],color='cyan',
        linestyle='-',zorder=1)
ax.fill_between(x_pred.flatten(), y_pred_linear_basis['obs_ci_lower'],
                y_pred_linear_basis['obs_ci_upper'],color='cyan',alpha=0.3,zorder=1)

plt.tight_layout()
```

As you can see from the the last example if we have multiple possible explanations we end up with a model where there are several symmetric explanations that cannot be differentiated under the objective function. To overcome we have to encode a *preference* to the solution we want.

2.1 Regularisation

In order to encode a preference towards specific solutions we will include an additional term in the objective that does only depend on the parameters $\boldsymbol{\beta}$. This is commonly done using a L_p-norm,

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmax}} \prod_{i=1}^{N} p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}) + \alpha \left(\sum_{j=1}^{d} \beta_j^p \right)^{\frac{1}{p}} . \tag{14}$$

The most commonly used two norms is the L_2 which is referred to as *ridge-regression* as it will encode a preference towards solutions where the parameters take equal large values and L_1 which is referred to as *lasso* as it prefers solutions that "homes" in on a few active parameters to explain the data. In Figure 3 you can see the regularisation surface for the two norms.

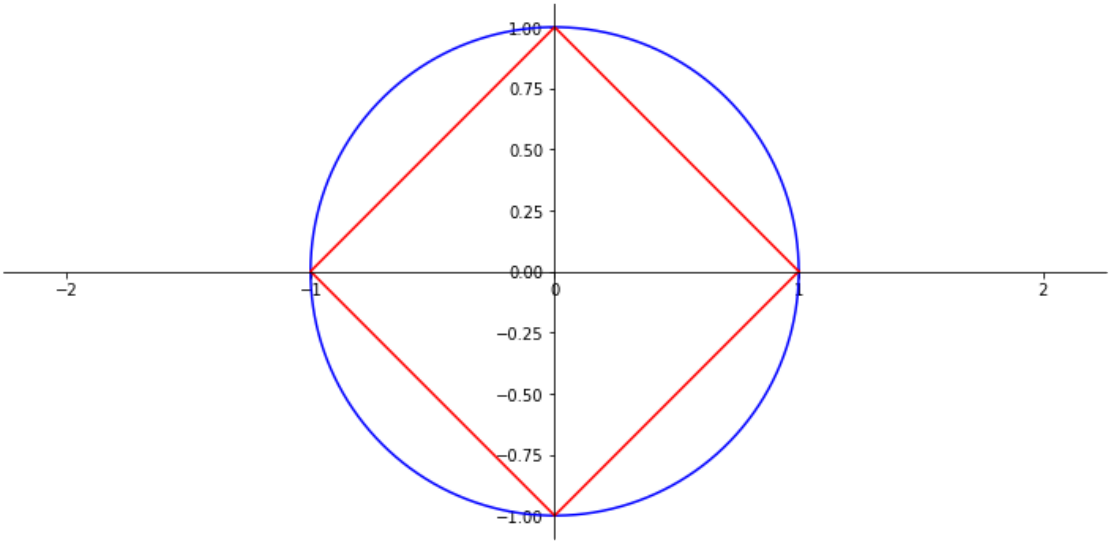

Figure 3: *The above plot shows the regularisation surfaces of L_2 in blue and L_1 regularisation in red.*

Question 3

Why does the two regularisations presented above "prefer" different solutions?

In `statsmodels` we can use the regularisation above by changing the call to fit the model from `fit` to `fit_regularised`. Taking two additional scalar parameters `alpha` and `L1_wt` where the former sets the strength of regularizer and the latter balances the L_1 and L_2 regularization.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

np.random.seed(42)

x = np.linspace(0,15,50)
y = 0.2*np.sin(x) + 0.5*np.sin(x**2/40) + 0.05*x
y += 0.05*np.random.randn(x.shape[0])

design = np.concatenate((np.sin(x).reshape(-1,1),
                         np.sin(x**2/40).reshape(-1,1), x.reshape(-1,1),
                         np.sin(x**2/20).reshape(-1,1),
                         np.sin(x**3).reshape(-1,1)),axis=1)
m_linear_basis = sm.OLS(y,design)
results_basis_0 = m_linear_basis.fit_regularized(alpha=0.10,L1_wt=0.0)
results_basis_1 = m_linear_basis.fit_regularized(alpha=0.10,L1_wt=0.3)
results_basis_2 = m_linear_basis.fit_regularized(alpha=0.10,L1_wt=0.6)
results_basis_3 = m_linear_basis.fit_regularized(alpha=0.10,L1_wt=1.0)

x_pred = np.linspace(-5,20,200).reshape(-1,1)
design_pred = np.concatenate((np.sin(x_pred), np.sin(x_pred**2/40),
                              x_pred,np.sin(x_pred**2/20).reshape(-1,1),
                              np.sin(x_pred**3).reshape(-1,1)),axis=1)

y_pred_0 = results_basis_0.predict(design_pred)
y_pred_1 = results_basis_1.predict(design_pred)
y_pred_2 = results_basis_2.predict(design_pred)
y_pred_3 = results_basis_3.predict(design_pred)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)
ax.scatter(x,y,zorder=2)

ax.plot(x_pred, y_pred_0, color='red', linestyle='--', zorder=1)
ax.plot(x_pred, y_pred_1, color='orange', linestyle='--', zorder=1)
ax.plot(x_pred, y_pred_2, color='magenta', linestyle='--', zorder=1)
ax.plot(x_pred, y_pred_3, color='cyan', linestyle='--', zorder=1)

plt.tight_layout()
```

2.2 Localised Basis Functions

Another way to think about designing a set of basis functions is to make them connected to the data, we could think about placing a non-linear transformation of each point

$$\begin{aligned} y_i &= \sum_{j=1}^{N} \beta_j \phi(\mathbf{x}_j, \mathbf{x}_i) \\ \phi(\mathbf{x}_j, \mathbf{x}_i) &= e^{-\frac{(\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}}(\mathbf{x}_i - \mathbf{x}_j)}{\ell^2}} \end{aligned} \tag{15}$$

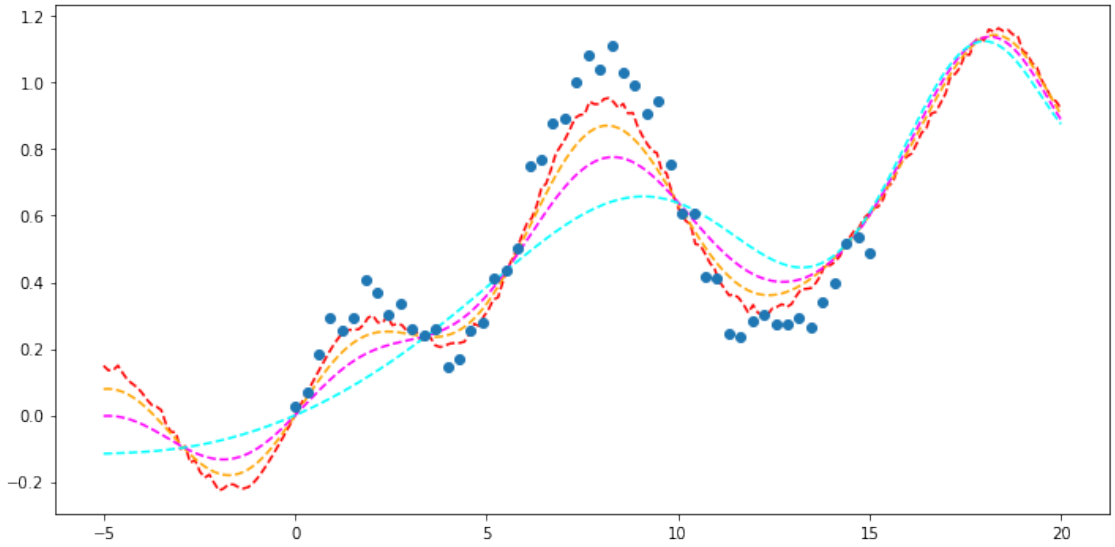

Figure 4: *The above plot shows the results of applying different levels of regularisation, where the red solution is L_2 and the cyan is L_1 and the orange and magenta are different combinations of the two.*

As we can see from Figure 5 the interpretation of the solution for the localised basis function model changes slightly. What we can now say is something similar to, "how important is each of the training data points for prediction". Especially interesting is the very strongly regularised solution shown in cyan. This can be interpreted as "if we want to predict the data, which is the single most representative point". The notion of a localised basis function can through this interpretation provide a rich explanation that is often very interpretable.

The type of data-centric basis functions that we explained above are very commonly used and if continue your study of machine learning during the year you will see a lot more of these basis functions as they are the foundation of a large range of different models from Support Vector Machines Vapnik, 1999 to Gaussian processes Rasmussen et al., 2006. Importantly as they solve an $N \times N$ regression problem the crucial ingredient is how do we regularise to solution space so that we can recover a solution.

3 Tick

You should now try out the model framework that we described above on a classic machine learning data-set called the Motorcycle Impact Data Silverman, 1985. You can download the data-set from here. We can load the data-set using `pandas` csv import,

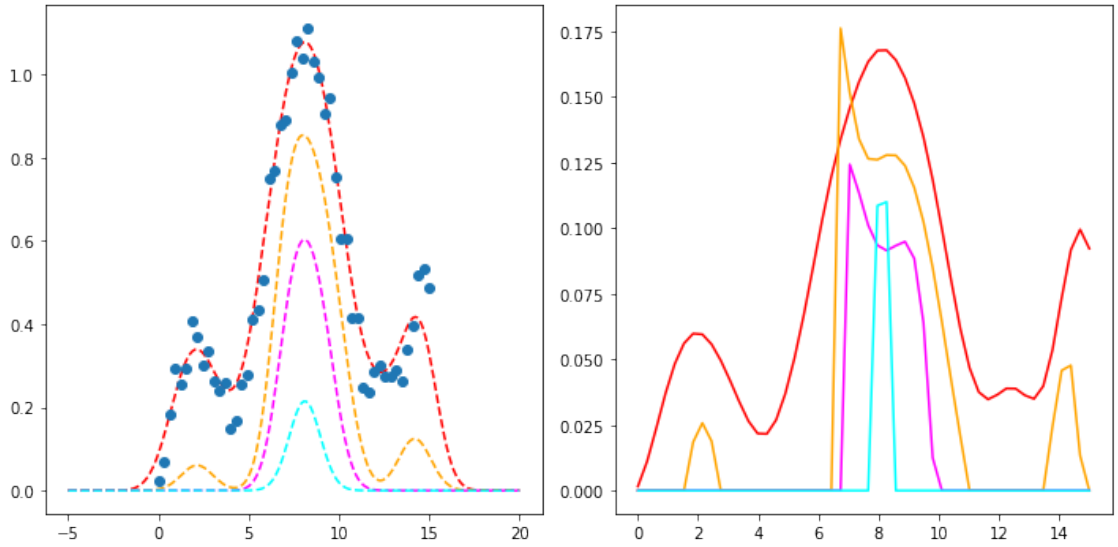

Figure 5: *The left pane of the figure above shows the fit of the model using a basis function located at each data-point. The right pane shows the actual parameter values associated with the different regularisation strategies as above.*

Code

```
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv

df = read_csv('/tmp/dataset-72001.csv')
x = np.array(df.values)[:,0]
y = np.array(df.values)[:,1]

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

ax.scatter(x,y)
```

Now try to use the framework that we have derived to explain the data-set above. There is no right or wrong answer here, what we are looking for is a motivation of why you have made the choices that you have made and how this effects the conclusions that you can draw.

Things to think about,

- would it make sense to split up the data-set in different regions and fit separate models?
- what is the criteria that you split the data using?
- what would be a sensible design matrix?
- what `GLM` models would make sense to fit?

During the tick-session we will ask you a few questions on how you have reasoned when fitting the data. Again, what we are looking for is motivation not "the best fit".

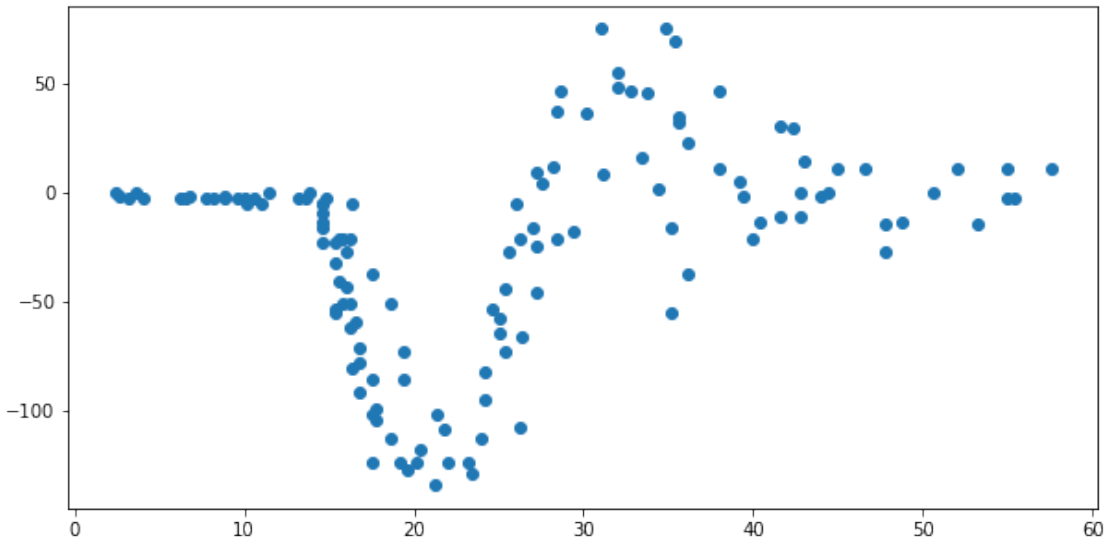

Figure 6: *The figure above shows the motorcycle impact data-set. The data-set is challenging as the characteristics of the response changes over the input-domain.*

4 Conclusions

This has been quite a short introduction to a set of very useful models. We have cut quite a few corners theoretically but importantly this is something that we have to do when working as data-scientist. We need to be able to apply models while there is still some uncertainty in how they work and be able to translate this uncertainty to how we interpret the results that we get. You are by no means expected to be experts on GLMs at all but you should be able to use them.

The next part of your challenge is now to include the models that we have built up and use them in the coursework. Try to first visualise the data, make a clear narrative of why you are selecting a specific model and try to use the tools that we derived her in order to provide context to the predictions that you make.

The `statsmodel` package gives you a rich set of tools that you should be able to include directly into your project. Importantly, remember that the most important thing is to be able to say *why* a model does what it does not necessarily choosing the *right* model.

During the lecture Neil mentioned the idea that what separates statisticians from machine learners is that the former cares about $\boldsymbol{\beta}$ while the machine learners cares about $\hat{y}$[4]. If you look at the `statmodels` package this concept becomes very clear. Using the GLM code as we did in the first exercise doesn't actually provide us with the uncertainty in the predictions only in the parameters. It was because of this we changed from the `GLM` model class to `OLS` when using the basis functions.

References

McCullagh, P. and J. A. Nelder (1989). *Generalized Linear Models*. London, UK: Chapman Hall / CRC: Chapman Hall / CRC.

Rasmussen, Carl Edward and Christopher K I Williams (2006). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Silverman, B. W. (1985). "Some Aspects of the Spline Smoothing Approach To Non-Parametric Regression Curve Fitting". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 47.1, pp. 1–21. DOI: 10.1111/j.2517-6161.1985.tb01327.x.

[4] the predictions

Vapnik, Vladimir N (1999). *The Nature of Statistical Learning Theory.*