Welcome!
#pod-031

(Reviewed by: Deepak)

neuromatch academy

facebook Reality Labs

Agenda

- **Day-#4**
    - Tutorial part 1 (Generalized Linear Models)
    - Tutorial part 2 (Classifiers and Regularizers)

W1D4_pod 031

Tutorial #1 Explanations

Machine Learning

Machine learning (**ML**) is the study of computer algorithms that improve automatically through experience. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data in order to make predictions or decisions without being explicitly programmed to do so.

Machine learning approaches are traditionally divided into three broad categories, depending on the nature of the "signal" or "feedback" available to the learning system:

- Supervised learning: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.
- Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).
- Reinforcement learning: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). As it navigates its problem space, the program is provided feedback that's analogous to rewards, which it tries to maximize.

Other approaches have been developed which don't fit neatly into this three-fold categorisation, and sometimes more than one is used by the same machine learning system. For example topic modeling, or meta learning.

Problem Statement

Objective: Model a retinal ganglion cell spike train by fitting a temporal receptive field.

Methods:

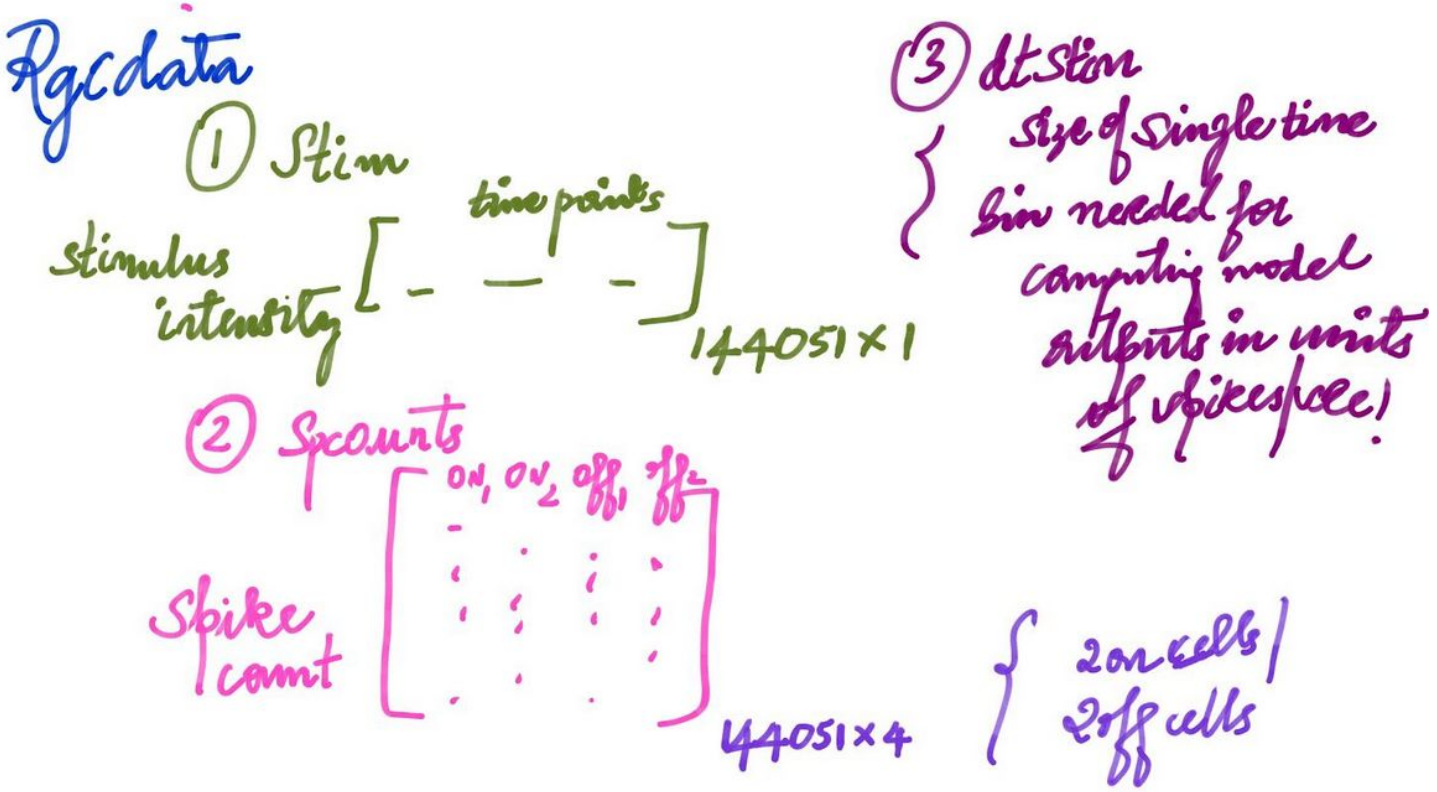
- Linear-Gaussian GLM (also known as ordinary least-squares regression model) and
- Poisson GLM (aka "Linear-Nonlinear-Poisson" model).

Data: Uzzell & Chichilnisky 2004.

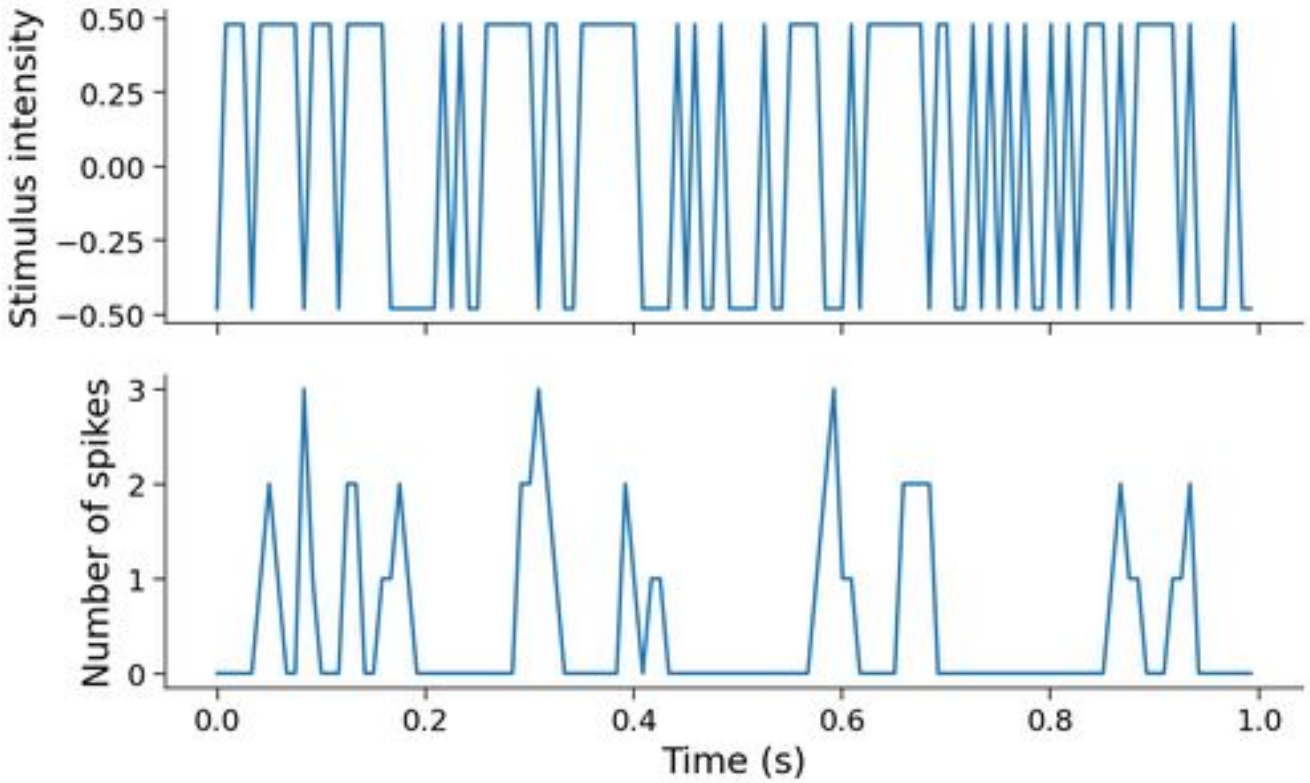
Please note that it is provided for tutorial purposes only, and should not be distributed or used for publication without express permission from the author (ej@stanford.edu).

Experiment: Presented a screen which randomly alternated between two luminance values and recorded responses from retinal ganglion cell (RGC), a type of neuron in the retina in the back of the eye. This kind of visual stimulus is called a "full-field flicker", and it was presented at ~120Hz (ie. the stimulus presented on the screen was refreshed about every 8ms). These same time bins were used to count the number of spikes emitted by each neuron.

Data description



```
1 plot_stim_and_spikes(stim, spikes, dt_stim)
```



Generalised Linear Models

In statistics, the generalized linear model is a flexible generalization of ordinary linear regression that allows for response variables that have error distribution models other than a normal distribution.

A GLM consists of three components: A **random** component, A systematic component, and. A link **function**.

Ref:
https://statisticaloddsandends.wordpress.com/2019/10/31/understanding-the-components-of-a-generalized-linear-model-glm/

Goals

Goal: To predict the cell's activity from the stimulus intensities preceding it to help understand how RGCs process information over time.

Algorithm: Create the *design matrix* for this model, which organizes the stimulus intensities in matrix form such that the ith row has the stimulus frames preceding time point i.
D = 25 (about 200 ms) is a choice we're making based on our prior knowledge of the temporal window that influences RGC responses
The last entry in row t should correspond to the stimulus that was shown at time t, the entry to the left of it should contain the value that was show one time bin earlier, etc. Specifically, X_{ij} will be the stimulus intensity at time i + d - 1 - j.

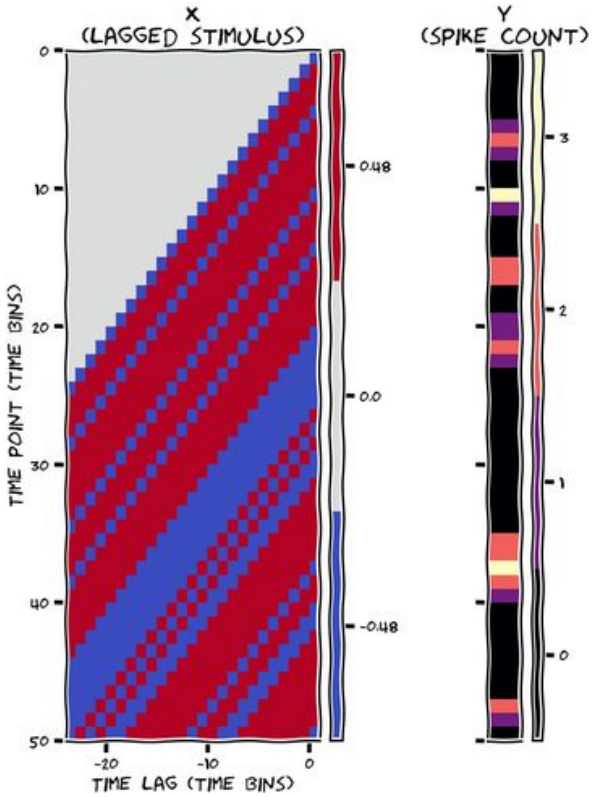
Note that for the first few time bins, we have access to the recorded spike counts but not to the stimulus shown in the recent past. For simplicity we are going to assume that values of stim are 0 for the time lags prior to the first timepoint in the dataset. This is known as "zero-padding", so that the design matrix has the same number of rows as the response vectors in spikes.

ALGORITHM:

- zero-padded version of the stimulus
- initialize empty design matrix with the correct shape
- **fill each row of the design matrix, using the zero-padded version of the stimulus**

"Heatmap": encodes the numerical value in each position of the matrix as a color.

Design matrix



Fit linear gaussian models

We will use the design matrix to compute the maximum likelihood estimate for a linear-Gaussian GLM (aka "general linear model"). The maximum likelihood estimate of Θ in this model can be solved analytically using

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

We need to augment the design matrix to account for the mean of y, because the spike counts are all >=0. We do this by adding a constant column of 1's to the design matrix, which will allow the model to learn an additive offset weigh, additional weight b (for bias), alternatively known as a "DC term" or "intercept".

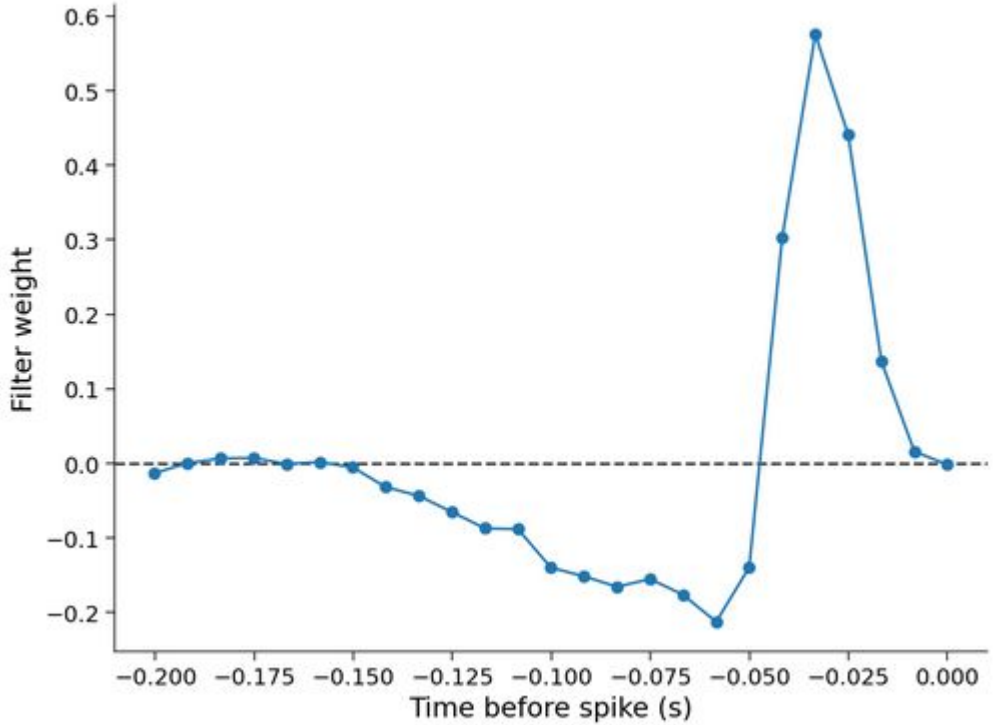
Predictions

Predict spike count for each timepoint using the stimulus information.

Algorithm:

- Create complete design matrix
- Obtain the MLE weights ($\hat{\Theta}$)
- Compute $\hat{y} = X\hat{\Theta}$

Plot maximum likelihood filter estimate



Observations

The prediction line more-or-less follows the bumps in the spikes, but it never predicts as many spikes as are actually observed. And, more troublingly, it's predicting *negative* spikes for some time points.

Bonus!

The "spike-triggered average" falls out as a subcase of the linear Gaussian GLM

$$\mathrm{STA} = X^T y \,/\, \mathrm{sum}(y),$$

Where y is the vector of spike counts of the neuron.

In the LG GLM, the term $(X^T X)^{-1}$

corrects for potential correlation between the regressors. Because the experiment that produced these data used a white noise stimulus, there are no such correlations. Therefore the two methods are equivalent.

Why no correlations in White noise?

1. **By definition/realisation:** In discrete time, white noise is a discrete signal whose samples are regarded as a sequence of serially uncorrelated random variables with zero mean and finite variance; a single realization of white noise is a **random shock**. Depending on the context, one may also require that the samples be independent and have identical probability distribution (in other words independent and identically distributed random variables are the simplest representation of white noise)

2. **By statistical proof:** necessary (but, in general, not sufficient) condition for statistical independence of two variables is that they be statistically uncorrelated; that is, their covariance is zero. Therefore, the covariance matrix R of the components of a white noise vector w with n elements must be an n by n diagonal matrix, where each diagonal element R_{ii} is the variance of component w_i; and the correlation matrix must be the n by n identity matrix.

3. **Probabilistic proof:** If w is a white random vector, but not a Gaussian one, its Fourier coefficients W_i will not be completely independent of each other; although for large n and common probability distributions the dependencies are very subtle, and their pairwise correlations can be assumed to be zero.

Linear-non linear poisson GLM (LNP)

Unfortunately in general there is no analytical solution to our statistical estimation problems of interest. Instead, we need to apply a nonlinear optimization algorithm to find the parameter values that minimize some *objective function*. This can be extremely tedious because there is no general way to check whether we have found *the optimal solution* or if we are just stuck in some local minimum.

Notes:

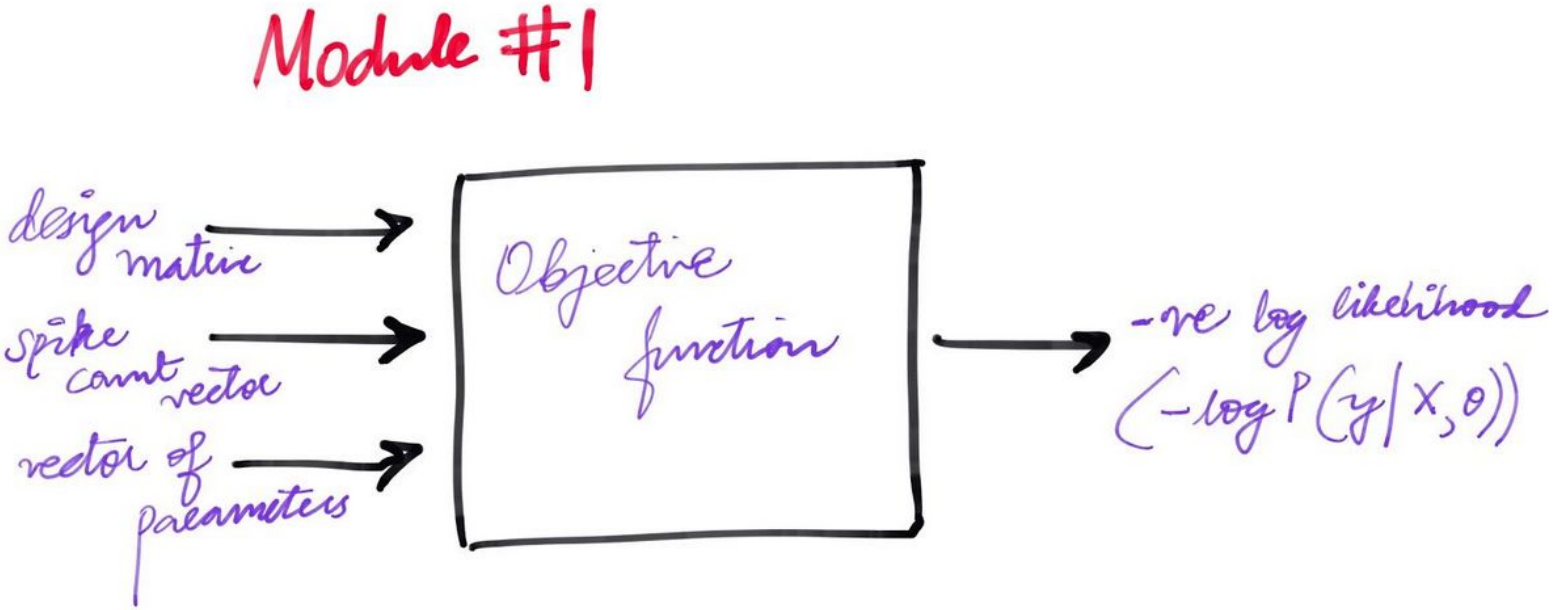
- a function is convex if and only if its curve lies below any chord joining two of its points

Scipy.optimise

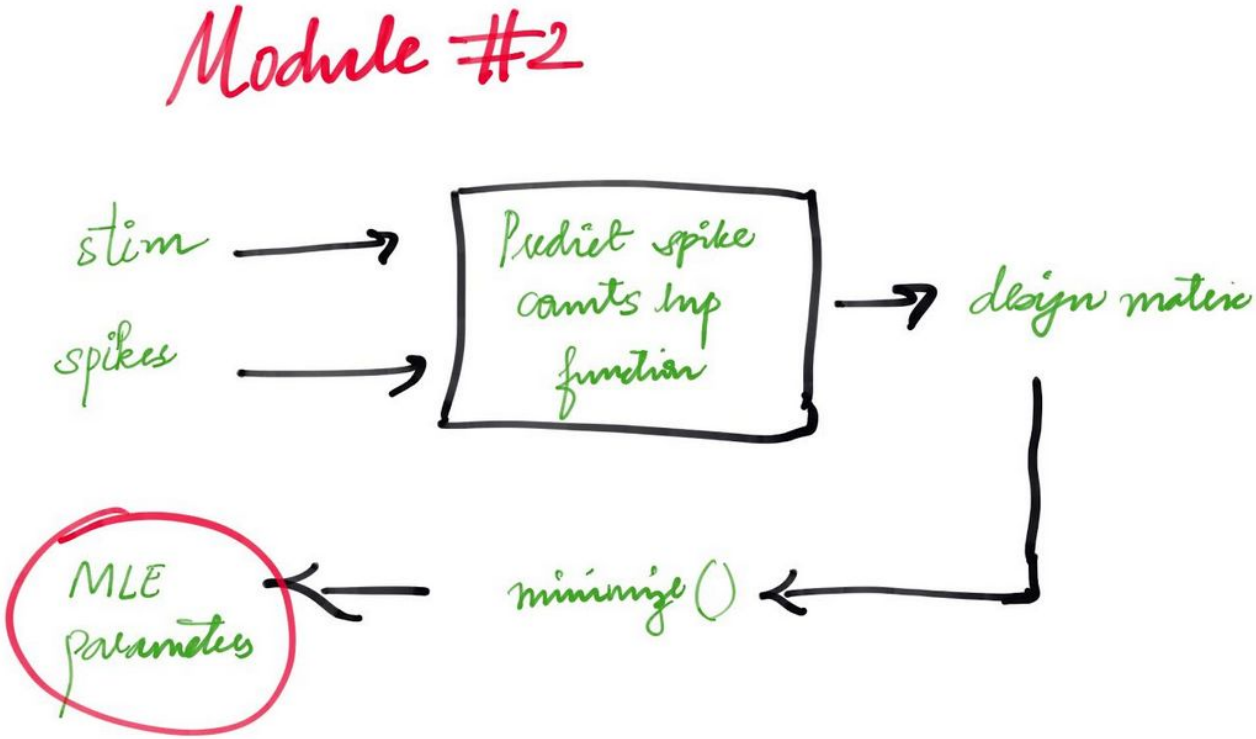
The final position of the minimization algorithm depends on the starting point, which adds a layer of complexity to such problems.

Algorithm

We will use scipy.optimize.minimize to compute maximum likelihood estimates for the filter weights in the Poisson GLM model with an exponential nonlinearity (LNP: Linear-Nonlinear-Poisson).



Algorithm


Module #2
stim
spikes
Predict spike counts log function
design matrix
minimize()
MLE parameters

In the Poisson GLM,

$$\log P(\mathbf{y} \mid X, \theta) = \sum_t \log P(y_t \mid \mathbf{x_t}, \theta),$$

where

$$P(y_t \mid \mathbf{x_t}, \theta) = \frac{\lambda_t^{y_t} \exp(-\lambda_t)}{y_t!}, \text{ with rate } \lambda_t = \exp(\mathbf{x_t}^\top \theta).$$

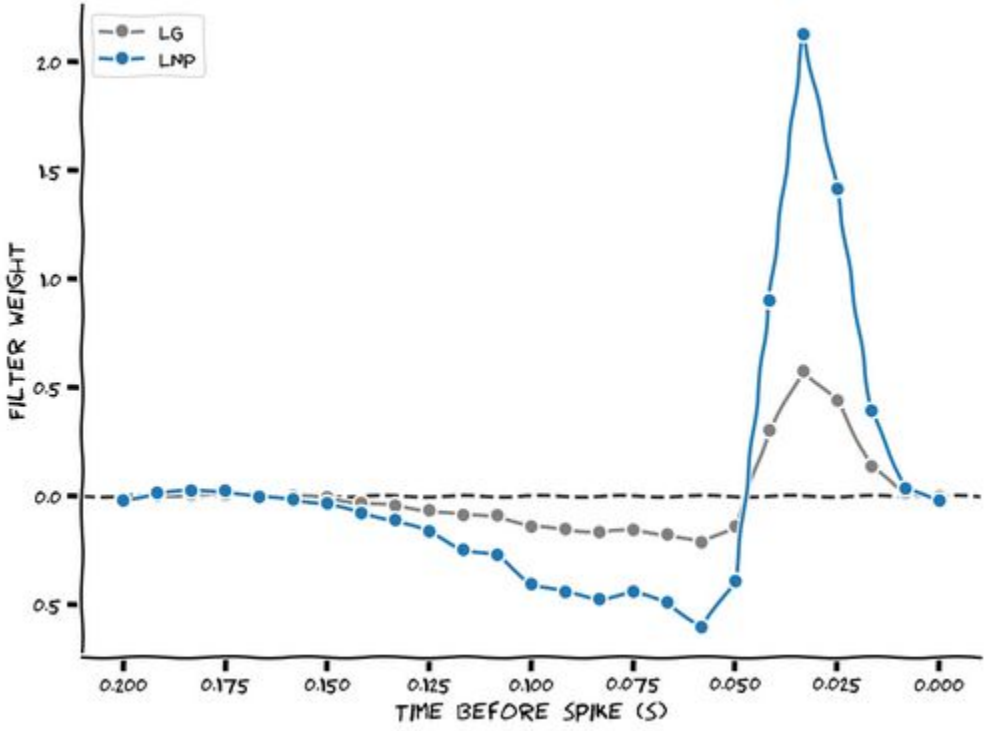
Now, taking the log likelihood for all the data we obtain: $\log P(\mathbf{y} \mid X, \theta) = \sum_t (y_t \log(\lambda_t) - \lambda_t - \log(y_t!))$.

Because we are going to minimize the negative log likelihood with respct to the parameters θ, we can ignore the last term that does not depend on θ. For faster implementation, let us rewrite this in matrix notation:

$$\mathbf{y}^T \log(\lambda) - \mathbf{1}^T \lambda, \text{ with rate } \lambda = \exp(X^\top \theta)$$

Finally, don't forget to add the minus sign for your function to return the negative log likelihood.

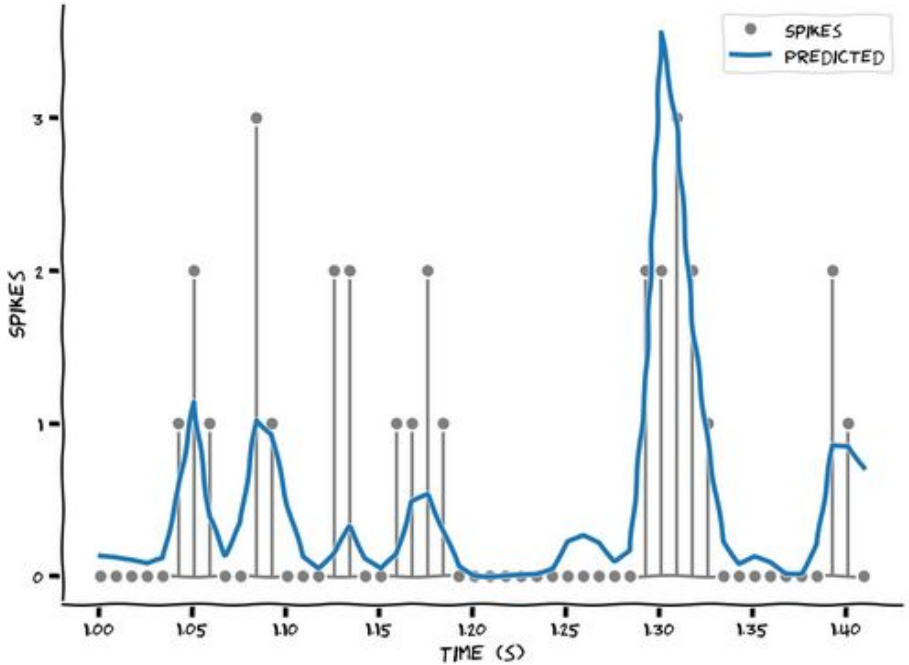
Plotting the LG and LNP weights together



Observations

We see that they are broadly similar, but the LNP weights are generally larger

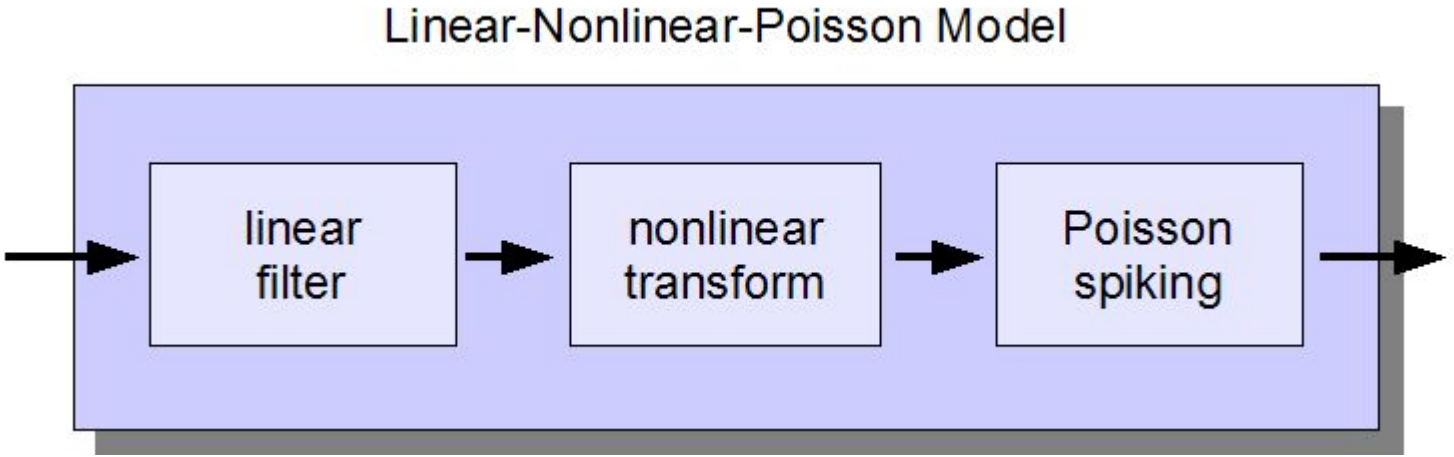
We see that the LNP model does a better job of fitting the actual spiking data. Importantly, it never predicts negative spikes!



Exploring the LNP model

Why LNP?

The linear filtering stage performs dimensionality reduction, reducing the high-dimensional spatio-temporal stimulus space to a low-dimensional feature space, within which the neuron computes its response. The nonlinearity converts the filter output to a (non-negative) spike rate, and accounts for nonlinear phenomena such as spike threshold (or rectification) and response saturation. The Poisson spike generator converts the continuous spike rate to a series of spike times, under the assumption that the probability of a spike depends only on the instantaneous spike rate.



Ref: https://en.wikipedia.org/wiki/Linear-nonlinear-Poisson_cascade_model

Why LNP?

In the linear-nonlinear-Poisson (LNP) model, one assumes that spike trains are produced by an inhomogeneous Poisson process with rate

$$\rho(t) = f(\boldsymbol{k} \cdot \boldsymbol{x}_t) \tag{11.5}$$

given by a cascade of two simple steps (Fig. 11.5B). The linear stage, $\boldsymbol{k} \cdot \boldsymbol{x}_t$, is a linear projection of $\boldsymbol{x}_t$, the (vector) stimulus at time t, onto the receptive field $\boldsymbol{k}$; this linear stage is then followed by a simple scalar nonlinearity $f(.)$ which shapes the output (and in particular enforces the non-negativity of the output firing rate $\rho(t)$). A great deal of the systems neuroscience literature concerns the quantification of the receptive field parameters $\boldsymbol{k}$.

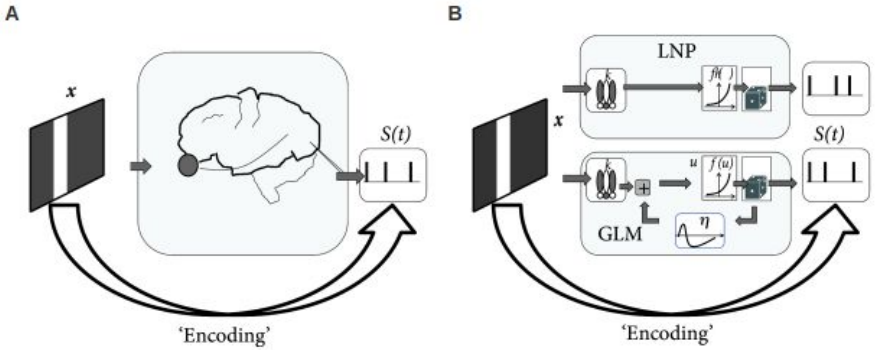


Fig. 11.5: The encoding problem in visual neuroscience. **A**. A stimulus is presented on a screen while a spike train is recorded from an area in visual cortex. **B**. Models designed to predict the spike train first filter the stimulus $\mathbf{x}$ with a spatial filter $\mathbf{k}$ (linear processing step), pass the result $u = \mathbf{k} \cdot \mathbf{x}$ through a nonlinearity f and then generate spikes stochastically with Poisson statistics. The main difference between an Linear-Nonlinear-Poisson (LNP, top) and a soft-threshold generalized integrate-and-fire model (GLM, bottom) is the presence of spike-triggered currents $\tilde{\eta}(s)$ in the latter.

Ref: https://neuronaldynamics.epfl.ch/online/Ch11.S2.html

Recommended Reading

Book of Stephen Boyd and Lieven Vandenberghe Convex Optimization.

Tutorial #2
Explanations

Objectives

In this tutorial, we'll implement logistic regression, a special case of GLMs used to model binary outcomes.

Oftentimes the variable you would like to predict takes only one of two possible values. Left or right? Awake or asleep? Car or bus? In this tutorial, we will decode a mouse's left/right decisions from spike train data. Our objectives are to:

1. Learn about logistic regression, how it is derived within the GLM theory, and how it is implemented in scikit-learn
2. Apply logistic regression to decode choies from neural responses
3. Learn about regularization, including the different approaches and the influence of hyperparameters

Logistic Regression

Logistic Regression is a binary classification model and GLM with a *logistic* link function and a *Bernoulli* noise model.

Logistic regression invokes a standard procedure:

1. Define a *model* of how inputs relate to outputs.
2. Adjust the parameters to maximize (log) probability of your data given your mode

Concept of logistic regression

The fundamental input/output equation of logistic regression is:

$$\hat{y} \equiv p(y = 1|x, \theta) = \sigma(\theta^T x)$$

Note that we interpret the output of logistic regression, $\hat{y}$, as the **probability that y = 1** given inputs x and parameters θ.

Here $\sigma()$ is a "squashing" function called the **sigmoid function** or **logistic function**. Its output is in the range $0 \leq y \leq 1$. It looks like this:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Recall that $z = \theta^T x$. The parameters decide whether $\theta^T x$ will be very negative, in which case $\sigma(\theta^T x) \approx 0$, or very positive, meaning $\sigma(\theta^T x) \approx 1$.

Task

Mice had the task of turning a wheel to indicate whether they perceived a Gabor stimulus to the left, to the right, or not at all. Neuropixel probes measured spikes across the cortex.

We **decode the decision from neural data** using Logistic Regression considering trials where the mouse chose "Left" or "Right" and ignore NoGo trials.

Decoding model: predict behavior (y) from the neural responses (X)

ModelFit

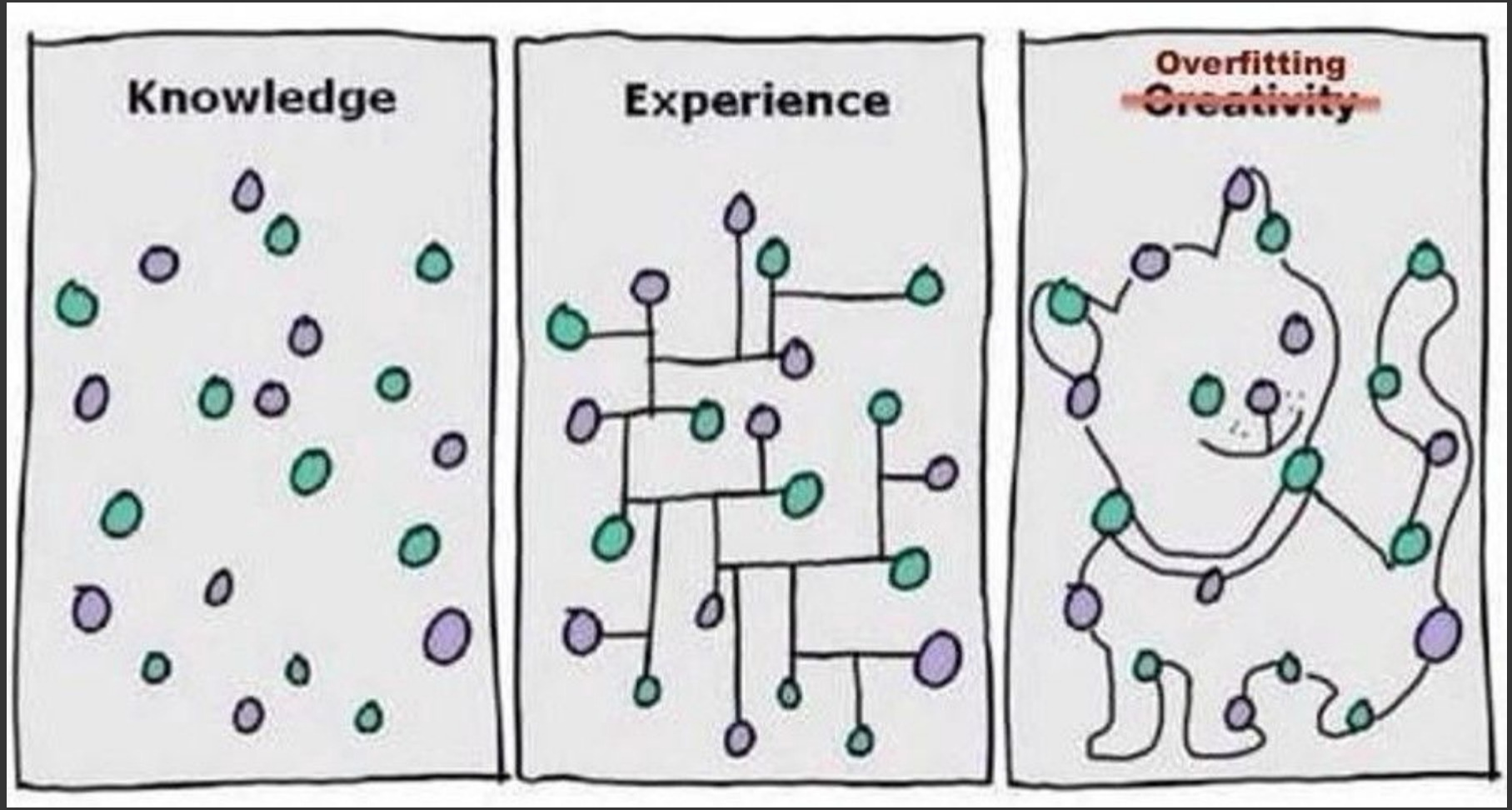
Fitting the model performs maximum likelihood optimization, learning a set of *feature weights* which can be used to *classify* new data, or predict the labels for each sample.

Evaluation

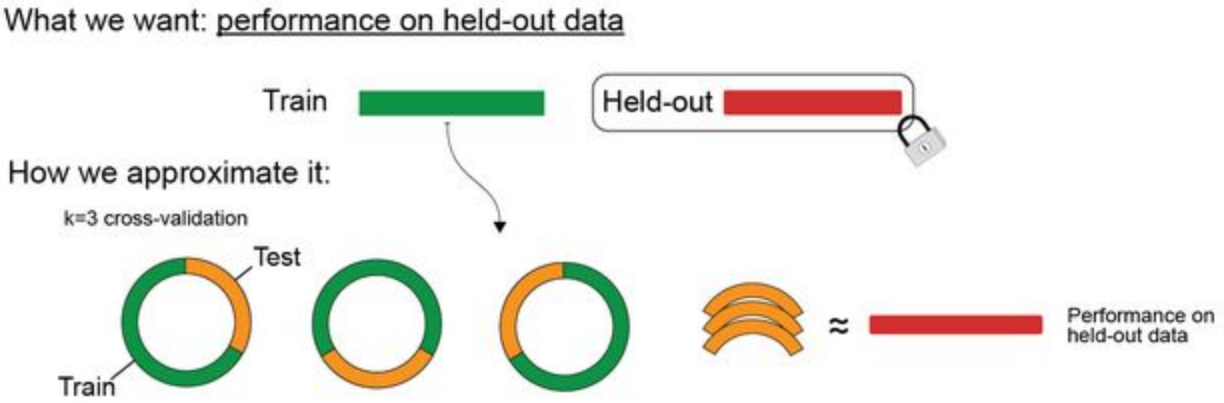
Accuracy score: The accuracy of the classifier is the proportion of trials where the predicted label matches the true label.

Observation: 100% classification accuracy on the training data relates to the concept of *overfitting*: the classifier may have learned something idiosyncratic about the training data. If that's the case, it won't have really learned the underlying data->decision function, and thus won't generalize well to new data.

Why? The model has almost three times as many features as samples. This is a situation where overfitting is very likely (almost guaranteed). Neuro data commonly has more features than samples. Eg: Having more neurons than independent trials, there are commonly more measured voxels than independent trials in fMRI data.

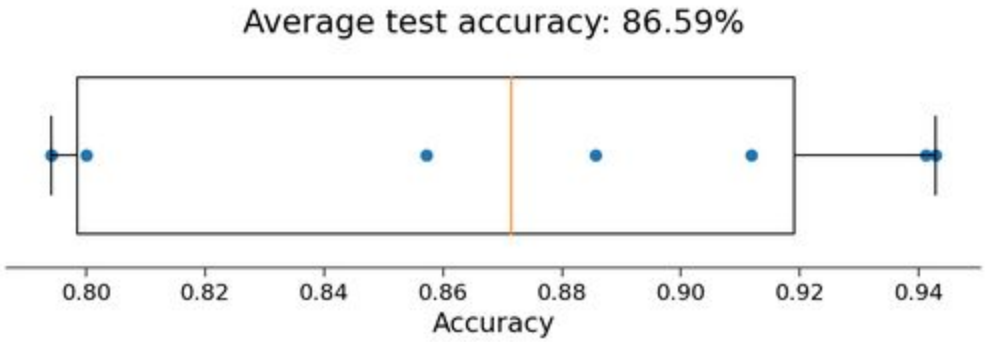

Knowledge
Experience
Overfitting
Creativity

Cross Validation



Results

For k=8



Regularisation

Why more features than samples leads to overfitting

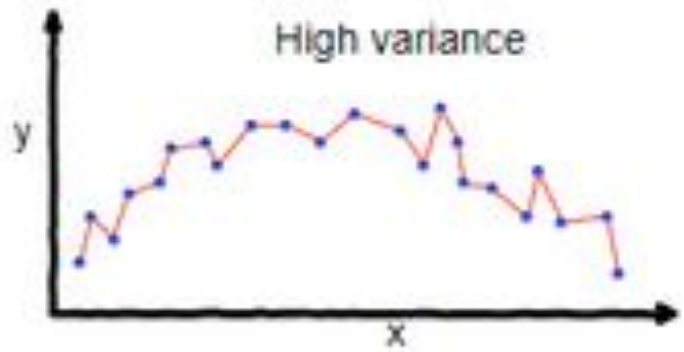
Bias-Variance Tradeoff: In brief, the variance of model estimation increases when there are more features than samples. That is, you would get a very different model every time you get new data and run .fit().

Why does this happen? Here's a tiny example to get your intuition going. Imagine trying to find a best-fit line in 2D when you only have 1 datapoint. There are simply a infinite number of lines that pass through that point. This is the situation we find ourselves in with more features than samples.

What we can do about it

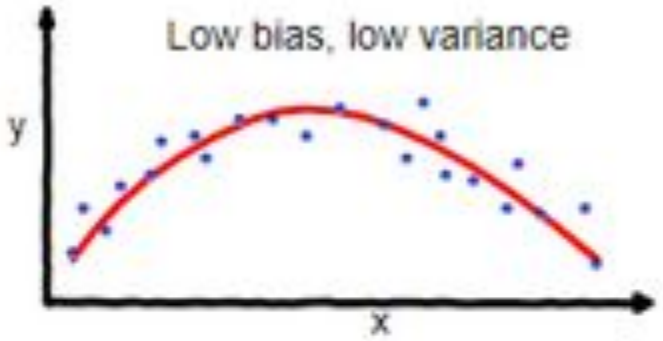
you can decrease model variance if you don't mind increasing its bias. Here, we will increase bias by assuming that the correct parameters are all small. In our 2D example, this is like preferring the horizontal line to all others. This is one example of *regularization*.

Without Regularisation



overfitting

With regularisation



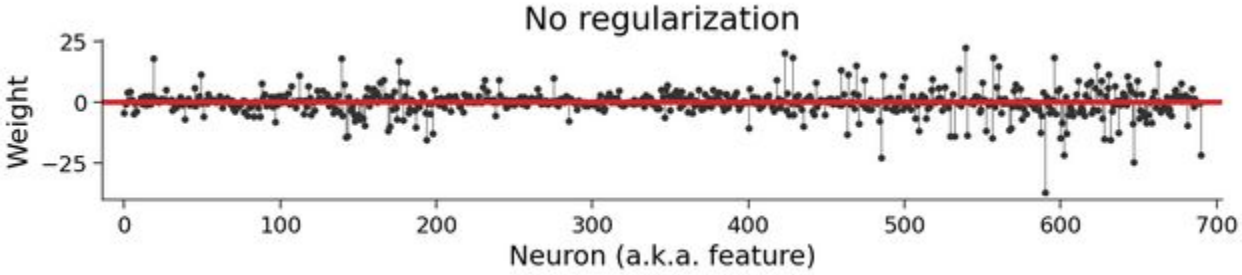
Good balance

Regularisation

Regularization forces a model to learn a set solutions you *a priori* believe to be more correct, which reduces overfitting because it doesn't have as much flexibility to fit idiosyncrasies in the training data. This adds model bias, but it's a good bias because you know (maybe) that parameters should be small or mostly 0.

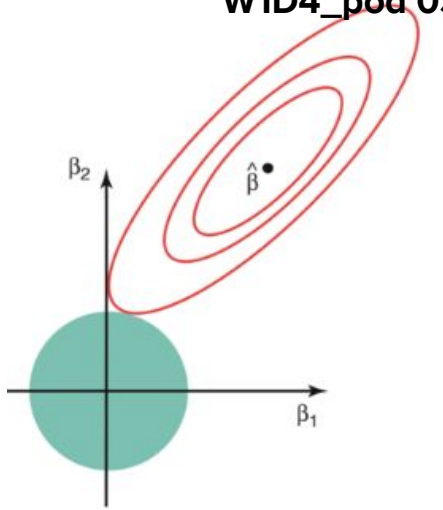
In a GLM, a common form of regularization is to *shrink* the classifier weights. In a linear model, you can see its effect by plotting the weights.

Each dot visualizes a value in our parameter vector θ. Since each feature is the time-averaged response of a neuron, each dot shows how the model uses each neuron to estimate a decision.



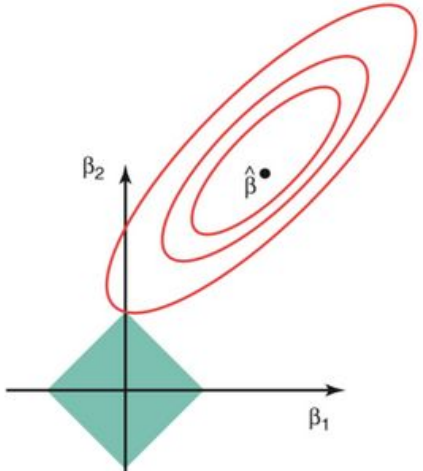
L2 Regularisation

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{1}^{n} w_i^2$$



L1 Regularisation

$$Loss = Error(Y - \hat{Y}) + \lambda \sum_{1}^{n} |w_i|$$



Error landscapes of L1 and L2

For same amount of Bias generated, the area occupied by L1 Norm is small. But L1 Norm doesn't concede any space close to the axes. This is what causes the point of intersection between the L1 Norm and Gradient Descent Contour to *converge near the axes leading to feature selection*.

In case of L2 Norm, the various combinations of β value generate a *particular same* Bias, form a circle.

Ref: https://towardsdatascience.com/regularization-in-machine-learning-connecting-the-dots-c6e030bfaddd

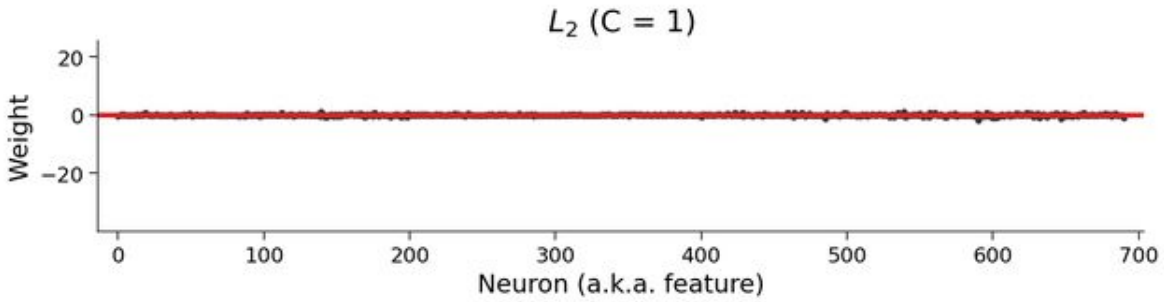
L2 regularisation

Regularization comes in different flavors. A very common one uses an *L2* or "ridge" penalty. This changes the objective function to

$$-\log \mathcal{L}'(\theta|X,y) = -\log \mathcal{L}(\theta|X,y) + \frac{\beta}{2}\sum_i \theta_i^2,$$

Where β is a *hyperparameter* that sets the *strength* of the regularization.

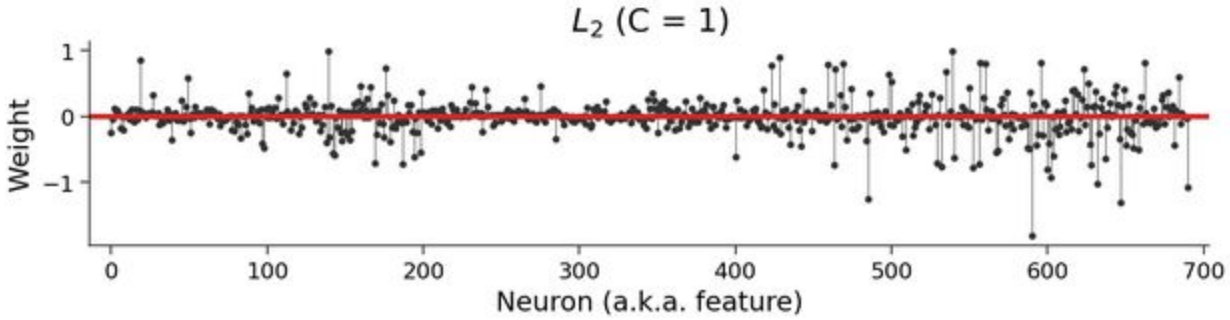
You can use regularization in scikit-learn by changing the penalty, and you can set the strength of the regularization with the C hyperparameter ($C=1/\beta$, so this sets the *inverse* regularization). Default: C=1



Now you can see that the weights have the same basic pattern, but the regularized weights are an order-of-magnitude smaller.

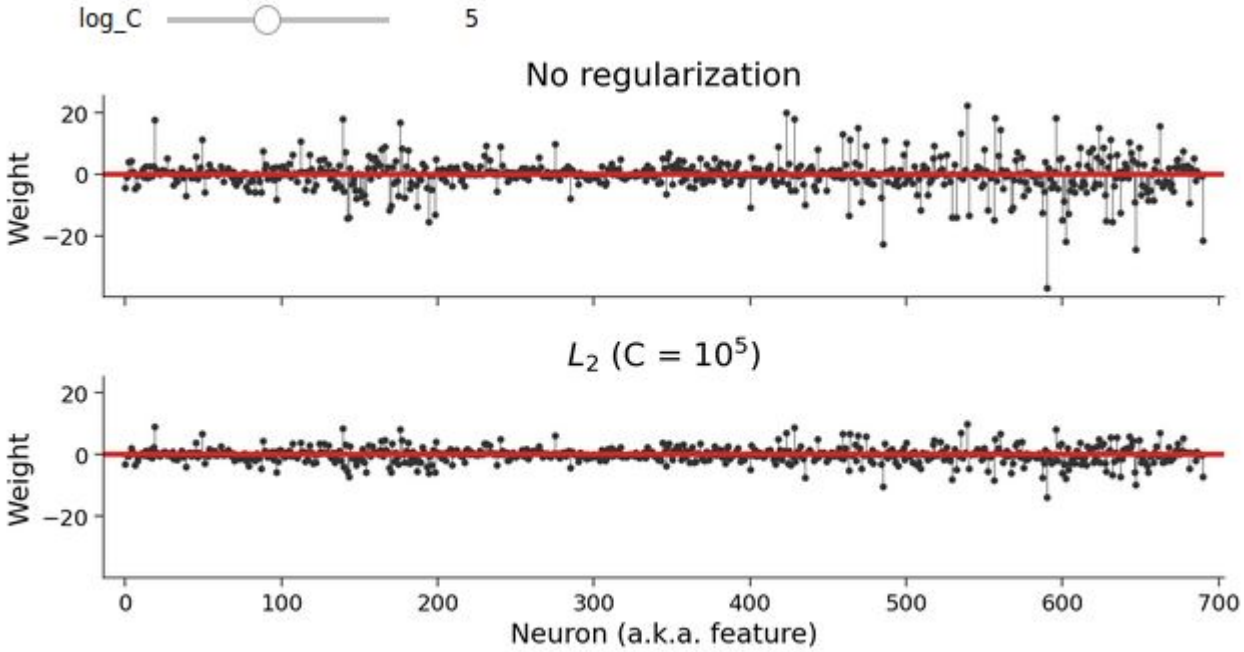
L2 regularisation with scaling

Allowing the y axis scales to adjust to each set of weights:



The effect of varying C on parameter size

how weights depend on the *strength* of the regularization:

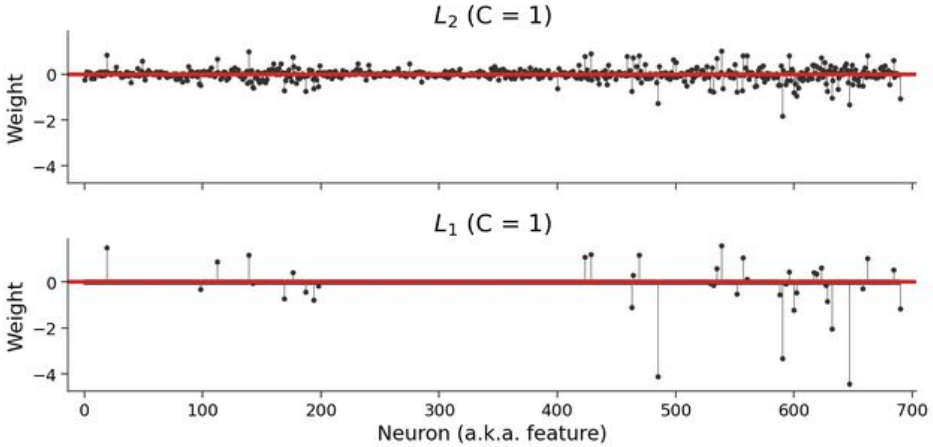


L1 regularisation

There is also the *L1*, or "Lasso" penalty. This changes the objective function to

$$-\log \mathcal{L}'(\theta|X,y) = -\log \mathcal{L}(\theta|X,y) + \frac{\beta}{2}\sum_i |\theta_i|$$

In practice, using the summed absolute values of the weights causes *sparsity*: instead of just getting smaller, some of the weights will get forced to 0:



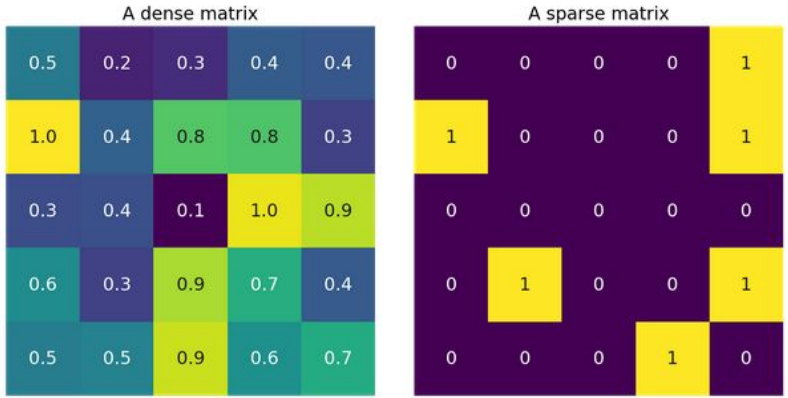
L1 vs L2 regularisation

When should you use *L1* vs. *L2* regularization?

Both penalties shrink parameters, and both will help reduce overfitting. However, the models they lead to are different.

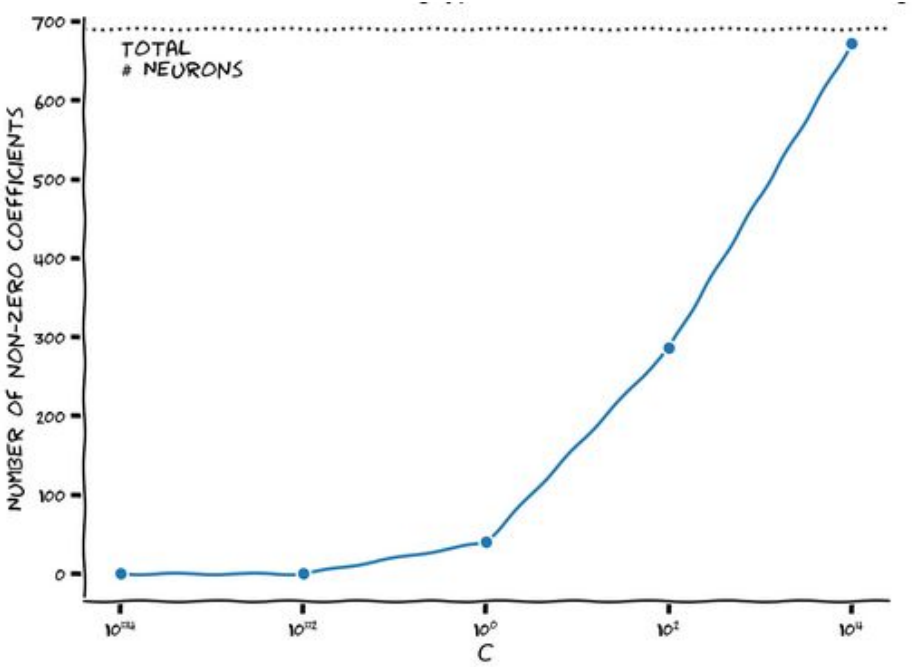
In particular, the *L1* penalty encourages *sparse* solutions in which most parameters are 0.

Dense vs sparse:



Effect of L1 on parameter sparsity

Return **the number of coefficients in the parameter vector that are equal to 0**.



Observations

Smaller C (bigger β) leads to sparser solutions.

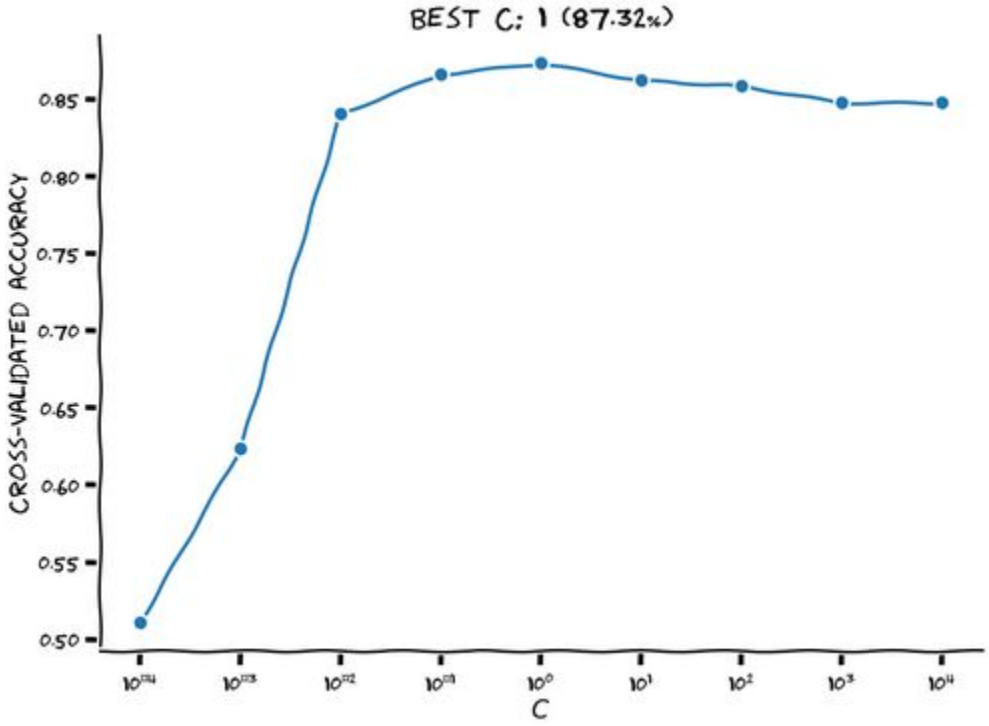
Link to neuroscience: When is it OK to assume that the parameter vector is sparse? Whenever it is true that most features don't affect the outcome. One use-case might be decoding low-level visual features from whole-brain fMRI: we may expect only voxels in V1 and thalamus should be used in the prediction.

WARNING: be careful when interpreting ϑ. Never interpret the nonzero coefficients as *evidence* that only those voxels/neurons/features carry information about the outcome. This is a product of our regularization scheme, and thus *our prior assumption that the solution is sparse*. Other regularization types or models may find very distributed relationships across the brain. Never use a model as evidence for a phenomena when that phenomena is encoded in the assumptions of the model.

Choosing regularisation penalty

The answer is the same as when you want to know whether you have learned good parameter values: use cross-validation. The best hyperparameter will be the one that allows the model to generalize best to unseen data.

Effect of regularisation coefficient



Observations

This plot suggests that the right value of C does matter — up to a point. Remember that C is the *inverse* regularization. The plot shows that models where the regularization was too strong (small C values) performed very poorly. For $C > 10^{-2}$, the differences are marginal, but the best performance was obtained with an intermediate value ($C \approx 10$).

Appendix: The Logistic Regression model in full

The fundamental input/output equation of logistic regression is:

$$p(y_i = 1|x_i, \theta) = \sigma(\theta^T x_i)$$

The logistic link function

You've seen $\theta^T x_i$ before, but the σ is new. It's the *sigmoidal* or *logistic* link function that "squashes" $\theta^T x_i$ to keep it between 0 and 1:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

The Bernoulli likelihood

You might have noticed that the output of the sigmoid, $\hat{y}$ is not a binary value (0 or 1), even though the true data y is! Instead, we interpret the value of $\hat{y}$ as the *probability that y = 1*:

$$\hat{y_i} \equiv p(y_i = 1|x_i, \theta) = \frac{1}{1 + \exp(-\theta^T x_i)}$$

To get the likelihood of the parameters, we need to define *the probability of seeing y given $\hat{y}$*. In logistic regression, we do this using the Bernoulli distribution:

$$P(y_i \mid \hat{y}_i) = \hat{y}_i^{y_i}(1 - \hat{y}_i)^{(1-y_i)}$$

So plugging in the regression model:

$$P(y_i \mid \theta, x_i) = \sigma(\theta^T x_i)^{y_i}(1 - \sigma(\theta^T x_i))^{(1-y_i)}.$$

This expression effectively measures how good our parameters θ are. We can also write it as the likelihood of the parameters given the data:

$$\mathcal{L}(\theta \mid y_i, x_i) = P(y_i \mid \theta, x_i),$$

and then use this as a target of optimization, considering all of the trials independently:

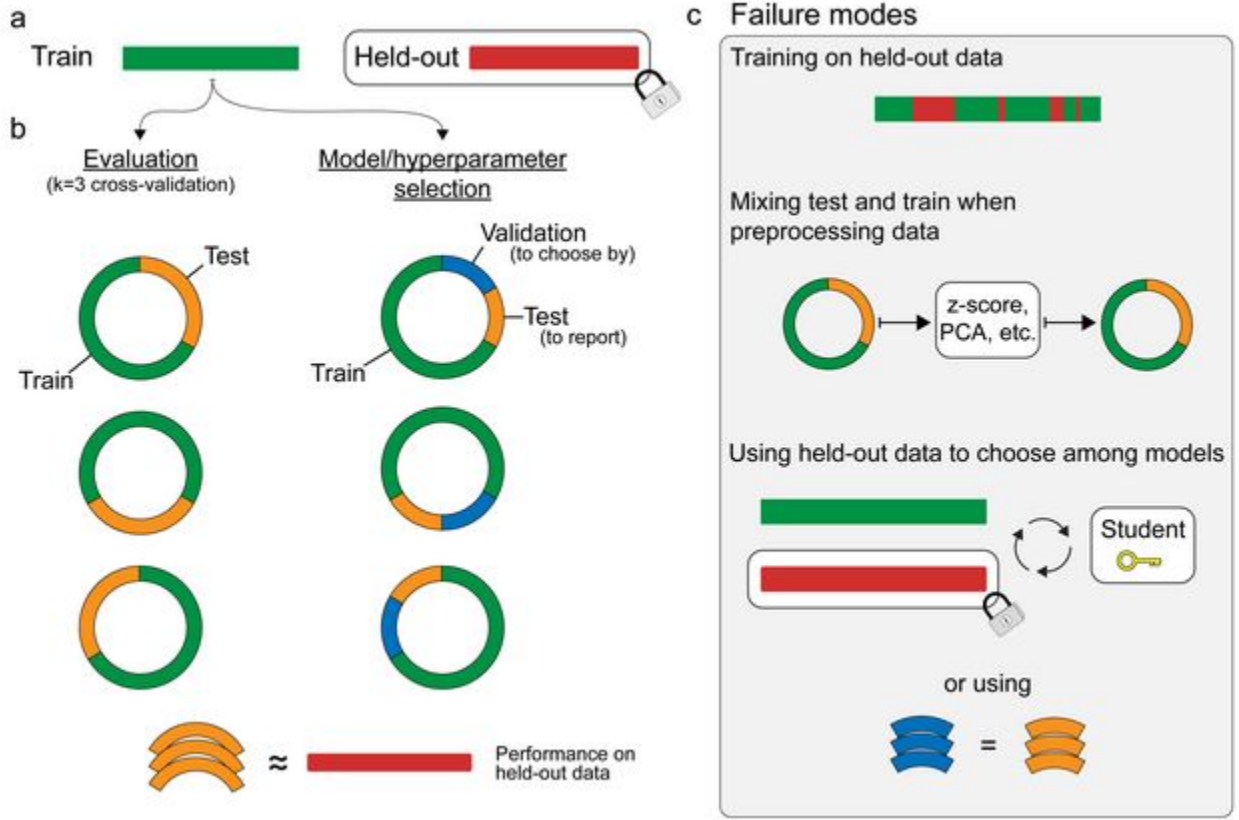
$$\log\mathcal{L}(\theta|X, y) = \sum_{i=1}^{N} y_i \log(\sigma(\theta^T x_i)) \; + \; (1 - y_i)\log(1 - \sigma(\theta^T x_i)).$$

Model Selection

We used all of the data to choose the hyperparameters. That means we don't have any fresh data left over to evaluate the performance of the selected model. In practice, you would want to have two *nested* layers of cross-validation, where the final evaluation is performed on data that played no role in selecting or training the model.

Indeed, the proper method for splitting your data to choose hyperparameters can get confusing. Here's a guide that the authors of this notebook developed while writing a tutorial on using machine learning for neural decoding (https://arxiv.org/abs/1708.00909).

Machine learning for neural decoding



Summary

Summary Lesson #1

two different models

- Linear-Gaussian (LG) model.
- Linear-Nonlinear-Poisson (LNP) model

to model how retinal ganglion cells respond to a flickering white noise stimulus.

Learning: Construct a design matrix to pass to different GLMs

Summary Lesson #2

Yet another GLM — logistic regression with cross validation — to ensure good model performance even when the number of parameters d is large compared to the number of data points N on the neural decoding problem: predict animal behavioural choice from neural activity.