1 Introduction

Brain-computer interfaces (BCIs) transform neural activity into a meaningful action on the environment, whether via a robotic arm or other user interface. Here, we will go over some simple ways to turn recorded neural activity into movement.

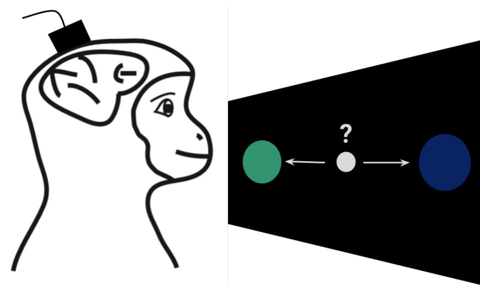


Figure 1: BCI target reaching task

Suppose we implanted an electrode array into the primary motor cortex (M1) of a monkey and isolated two neurons. We then recorded the activity of these two neurons as the monkey made arm reaches to targets at different positions around a screen. Now, we want to build a BCI controller using these two neurons to allow the monkey to move a cursor to either a left target or a right target. How do we determine which target the monkey is moving towards based on the activity of these two neurons?

One way we can "decode" the intended target from the monkey's neural activity is to look at the firing rate of individual neurons. As the monkey makes reaches in different directions, we record the response of each single neuron. Using these responses, we can plot a **tuning curve** for each neuron. Then, given an instantaneous firing rate for each neuron, we can reference the tuning curves of each neuron to make a guess as to which target the monkey is trying to reach.

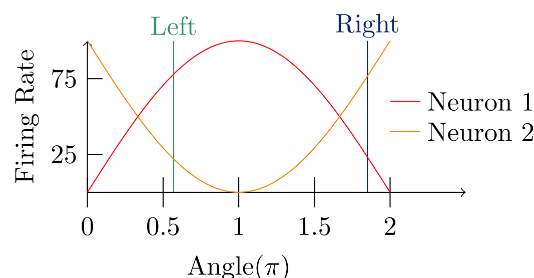


Figure 2: Tuning curves of two neurons

For example, suppose the tuning curves for the two neurons in our monkey are shown in Figure 2. If we only know that the firing rate of neuron 1 is 75 spikes/s, the monkey could be moving to the left or to the right. However, if we also know that the firing rate of neuron 2 is 25 spikes/s, then we know that the monkey is moving towards the left target. Similarly, when neuron 2 is firing at 75 spikes/s and neuron 1 is firing at 25 spikes/s, the monkey is moving towards the right target. Thus, for our BCI controller, we can say that every time we observe (Neuron 1, Neuron 2) activity of (75, 25), the cursor should move to the left target, and the cursor should move to the right target when we observe (Neuron 1, Neuron 2) activity of (25, 75). While this is straightforward for two targets, how do we generalize this to three or more targets?

As we increase the number of targets the monkey is reaching towards and the number of neurons we record from, we need scalable and generalizable ways to decode the monkey's reach direction from neural activity. In particular, the goal of using machine learning algorithms for BCI control is to make predictions of intended outcome when we see neural activity that *we have not previously observed*. Our goal is to make these predictions as accurately as possible using data we have seen before. We will first examine how to predict the intended target from neural activity through **K-means clustering**, and then look at how we can make interpreting the activity of large populations of neurons tractable via **principal components analysis**.

2 K-Means Clustering

The purpose of K-means clustering is to classify each observed data point into one of K classes. In the BCI decoding example described above, we want to use the observed neural activity of neuron 1 and neuron 2 to determine which target, left, or right, the monkey wants to reach towards. Rather than looking at the tuning curves of neuron 1 and neuron 2 separately, we can jointly view the activity of neuron 1 and neuron 2 as two axes in neural space.

Figure 3 shows the activity of neuron 1 and neuron 2 as the monkey makes reaches to the left and right. We want to build a BCI decoder to predict reach direction from neural activity, and we observe the firing rate combination represented by "New Point". Which target do we assign it to? The simplest and most intuitive decision would be to assign our new point to the closest cluster, which is exactly what we do in K-means clustering.

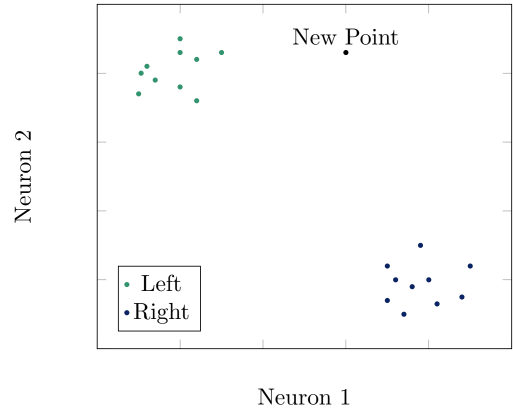
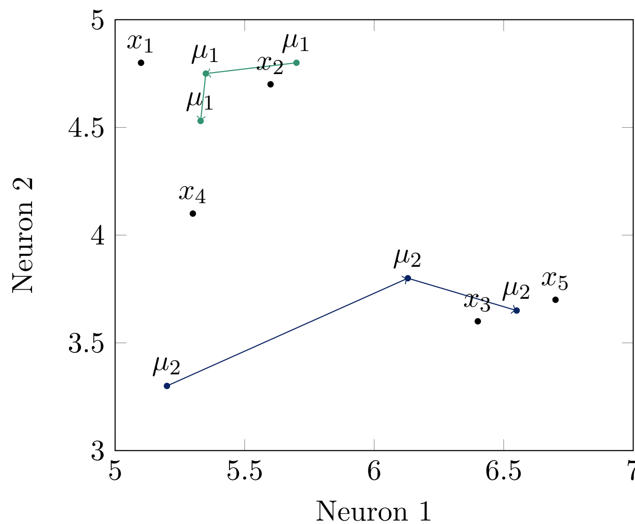


Figure 3: Firing rate on reach trials

K-means clustering is a two-step iterative algorithm. To initialize K-means, we create a cluster center for each class k (left and right reaches). Then, we assign each data point to a cluster (we label each joint firing rate as a leftward or rightward reach). Then, we update our cluster centers to be the mean of all points assigned to that class (mean of all joint firing rates assigned to left or right reacahes). These latter two steps (cluster assignment and update) are repeated until each point is assigned to the closest cluster and the cluster means are not changing (Figure 4). The iterative update process becomes the basis for the **expectation-maximization algorithm** (EM-algorithm). K-means is a special case of EM which is guaranteed to converge to a local (not global!) optimum.

Here is an example where we have 5 data points and 2 classes ($k = 1$ and $k = 2$). We can view $k = 1$ as a left reach and $k = 2$ as a right reach:



Data Point	$k = 1$	$k = 2$
x_1	~~1~~, 1	~~0~~, 0
x_2	~~1~~, 1	~~0~~, 0
x_3	~~0~~, 0	~~1~~, 1
x_4	~~0~~, 1	~~1~~, 0
x_5	~~0~~, 0	~~1~~, 1

Figure 4: K-Means in action

Expectation-Maximization (EM) Algorithm:
N datapoints, K classes (or targets)

0. Initialize the means for each cluster, $\mu_1 ... \mu_k$.
1. Assign each data point x_n to the class with the closest mean: (E-step)

$$r_{nk} = \begin{cases} 1 \text{ if } k = \arg\min_j \|x_n - \mu_j\|^2 \\ 0 \text{ otherwise} \end{cases}$$

2. Set μ_k equal to the mean of all the data points assigned to clluster k: (M-step)

$$\mu_k = \frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}}$$

3. Iterate over steps 1 and 2 until convergence (means and cluster assignments stop changing).

r_{nk} is a vector of length K for data point x_n. It has a 1 in the kth position if x_n is assigned to class k, and 0 everywhere else. Each value in the table corresponds to the r_{nk} value for that data point on each iteration.

2.1 An Alternate Perspective on K-Means

Another way to think about k-means clustering is to think of it as trying to find clusters such that the points within a cluster are closer together than points outside of the cluster. We can take this approach to optimize a cost function, where our cost is the sum of inter-point distances.

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|x_n - \mu_k\|^2$$

This cost function is the sum of the distance between each data point x_n and the cluster it was assigned to. J is lower when points assigned to a cluster k are closer to the cluster mean μ_k, and large when points assigned to a cluster k are far from μ_k. Using this equation in the EM-algorithm:

1. **E-step**: $\{r_{n1} \dots r_{nk}\}$ is a set of $(k-1)$ zeros and a single 1. This is essentially a vector for each data point x_n with length k. It contains zeros for each class k that point x_n is not assigned to, and a 1 in the class k that x_n is assigned to. This is a **one-hot encoding** vector. We want to minimize j with respect to r_{nk} for each data point x_n separately. Thus, for each x_n, we assign:

$$r_{nk} = \begin{cases} 1 \text{ if } k = \arg\min_j \|x_n - \mu_j\|^2 \\ 0 \text{ otherwise} \end{cases}$$

   This is the same equation as the E-step from earlier.

2. **M-step**: We want to find the μ_k that minimizes J. Since J is quadratic (and concave), we can take the derivative of J and set it equal to 0 to optimize for μ_k.

$$\frac{dJ}{d\mu_k} = \frac{d}{d\mu_k}(\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|^2) = -2\sum_{n=1}^{N} r_{nk}(x_n - \mu_k) = 0$$

$$\sum_{n=1}^{N} r_{nk}x_n = \sum_{n=1}^{N} r_{nk}\mu_k$$

$$\mu_k = \frac{\sum_{n=1}^{N} r_{nk}x_n}{\sum_{n=1}^{N} r_{nk}}$$

This is the same update equation for the mean that we came up with earlier, but optimizing over the cost function provides a more rigorous mathematical guarantee of optimality.

2.2 Choosing K

Sometimes, as in the case of decoding reaches towards two targets, we have a good idea what K should be. In some applications of k-means (such as spike sorting), however, what K should be is not always clear. So how do we choose K?

Let's start by examining the extremes. On one end, we can say $K = 1$, and assign all of our data points to one cluster. This instance of K-means is very simple, but probably not very accurate. At the other end of the spectrum, we can say $K = N$, where N is the number of data points. This means that every data point will be assigned to its own cluster. The accuracy of this instance of K-means on *data that we have seen before* is going to be 100%. However, this instance of K-means *does not generalize well* to previously unseen data. The "right K" is somewhere between 0 and N.

In most machine learning methods, we want to fit the parameters of our model to some data that we call the **training data**, but to estimate the *generalizability* of our model, we see how well it performs on data that it hasn't seen before. This dataset is the **test data**, and we use it to evaluate model performance. So if we are using K-means as our machine learning model, we might train N different models of K-means where $K = 1...N$. Then, when we plot K vs. model accuracy for each model, we would expect to see a curve like Figure 4. Since the accuracy on the test data peaks at $K = 3$, we would choose to classify our data into 3 clusters in this instance. If there isn't enough data to split into training and test sets and still get accurate estimates of model accuracy, we can use **cross-validation**. In cross-validation, data are divided into folds of equal size. One fold is left out as the test dataset, and the model is trained on the remaining folds. This process is then repeated with another fold as the "held-out" test dataset and all other folds as training data, until all trials have been a part of the test dataset. This is called **leave-one-out cross validation** and allows us to both fit and test model accuracy on a smaller dataset.

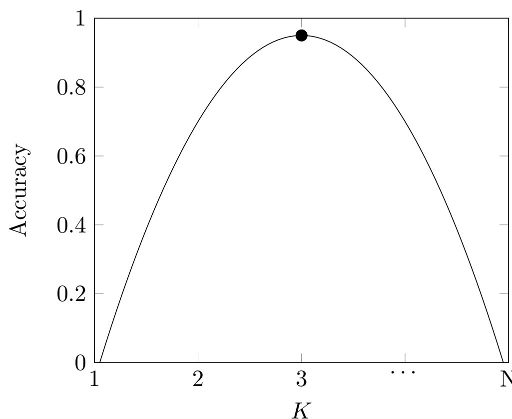


Figure 5: Model accuracy as a function of K

3 Principal Components Analysis (PCA)

So far, we have operated under the assumption that we are recording from two neurons. What if we have a lot more? What if we have 100? Can we still use k-means? We can, but it might be slow. If we still want to cluster in 2 dimensions, how do we know which 2 neurons to pick? Rather than clustering in a very high-dimensional neural space, can use principal components analysis (PCA) to choose the dimensions of neural activity to cluster in. PCA is a **dimensionality reduction technique**, where the goal is to take M-dimensional data and project it into a D-dimensional space such that $D < M$ and the variance of the data is maximally preserved.

Let's start with an example in two dimensions ($M = 2$):

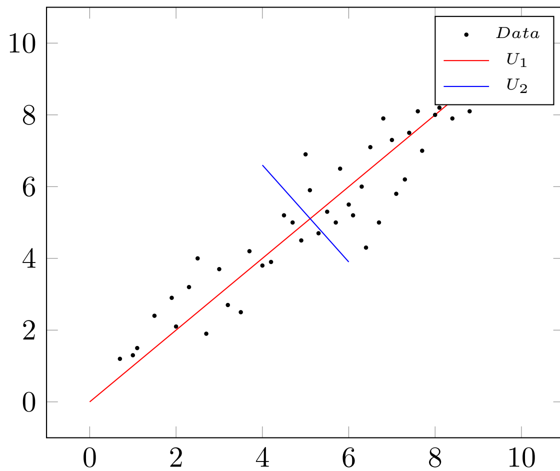


Figure 6: Principal axes of 2D data

We can project this data down to one dimension (remember, $D < M$) using PCA. PCA will find orthogonal axes corresponding to the dimension of greatest variance. In our example, U_1 is the axis of greatest variance. U_2 is an orthogonal axis to U_1 that captures the second-greatest amount of variance. To reduce our 2D data to 1D, we would find the value each data point takes on U_1, and throw out U_2. This is because U_1 captures more variation in the data (i.e. the data are more spread out along U_1) than U_2. How does PCA find the axes of greatest variance of our data?

We use the covariance matrix to estimate the spread of the data. This is the higher-dimensional version of the variance that you may have seen in a statistics class:

$$S = \frac{1}{N}\sum_{n=1} N(x_n - \mu)(x_n - \mu)^T, \text{ where } \mu = \frac{1}{N}\sum_{n=1}^{N} x_n$$

To find the axes of greatest variance, PCA takes the **eigendecomposition** of the covariance matrix S, giving its eigenvalues and eigenvectors. What are eigenvectors and eigenvalues? An eigenvector of a linear transformation is one in which the transformation is not rotated or translated, only scaled. The eigenvalue is the amount of scaling along that eigenvector:

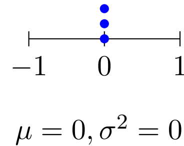
$$A\mathbf{v} = \lambda\mathbf{v}$$

where $\mathbf{v}$ is the eigenvector of the linear transformation A and λ is the eigenvalue.

What does this mean in the context of PCA on neural data?
Eigenvectors are the directions of a **linear transformation** where the transformation is only scaled. In PCA, we find the eigenvectors and eigenvalues of the **covariance matrix**.

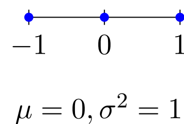
- What's the linear transformation in PCA?
  The covariance matrix

- How is the covariance matrix a linear transformation?
  It scales or rotates the data (it smears it). For example, let's say we have 3 points. If I tell you the mean of the three points is 0, and the variance is 0, then our points look something like this:



$\mu = 0, \sigma^2 = 0$

  Alternatively, if the mean of the points is 0 and the variance is 1, then our points probably look something like this:



$\mu = 0, \sigma^2 = 1$

  As we can see, a nonzero variance spreads the data out along our axis. In multiple dimensions, we need more numbers to explain how the data are spread out along multiple axes, which we represent with the covariance matrix. **So we can think of the covariance matrix as a "smearing transformation" that smears the data away from its mean.**

By finding the eigenvectors and eigenvalues of the covariance matrix, we are finding the direcions of greatest smearing of our data–the eigenvalues are how much our data are smeared along that axis. Thus, the axis of greatest variance corresponds to the axis of "greatest smearing"—or the eigenvector of the covariance matrix with the largest eigenvalue.

Mathematically, we write this as:

$$S = U\Lambda U^T, \text{ where } U = \begin{bmatrix} | & | & \dots & | \\ U_1 & U_2 & \dots & U_D \\ | & | & \dots & | \end{bmatrix} \text{ and } \Lambda = \begin{bmatrix} \lambda_1 & & 0 & \\ & \lambda_2 & & \\ & 0 & \ddots & \\ & & & \lambda_D \end{bmatrix}$$

such that $\lambda_1 > \lambda_2 > \dots > \lambda_D$

We can then project the mean-centered high-dimensional point $(x_n - \mu)$ onto an eigenvector U_i to find the low-dimensional point along that eigenvector that we'll call z_n:

$$z_n = (x_n - \mu)^T \cdot U_i$$

The projection of x_n onto U_i is defined as:

$$\|x_n\| \cos\theta = \frac{\|x_n\|\|U_i\| \cos\theta}{\|U_i\|} = \frac{x_n^T U_i}{\|U_i\|}$$

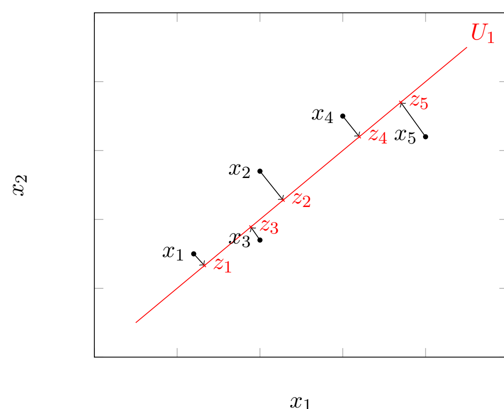
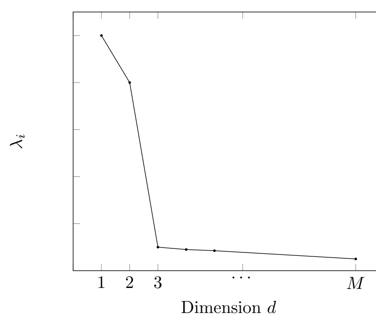


Figure 7: Projection onto PC1

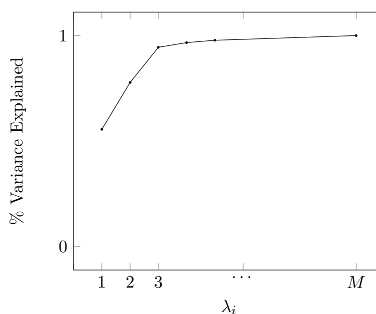
Figure 7 shows the reduction of our original $\langle x_1, x_2 \rangle$ data to coordinates on the U_1 axis. The value of each data point on U_1 is indicated by the projection (dot product) of the point onto U_1. Thus, PCA found this axis along which are data are most spread out, and we transformed our data to be along this axis rather than in the original space. Before we use PCA, we subtract the mean from our data so that the lower-dimensional points are in U_i coordinates and not in the original $\langle x_1, x_2 \rangle$ coordinates. In other words, mean subtracting the data point will center the new points around 0 on U_1.

In summary, we've used PCA to take two-dimensional data down to one dimension. What if we start off with 100-dimensional data? How do we choose D, the dimensionality of the lower dimensional space? We look at the eigenvalues to tell us how much "smearing" or variance is captured along a given axis. To capture all the variance in the data, we need all the eigenvalues and eigenvectors. However, we can look at the eigenvalues to make our decision about D:

1. Plot the eigenvalues of the covariance matrix and look for an "elbow"



2. Take the k eigenvalues that explain at least 90% of the variance



Here we might pick $D = 2$ because most of the variance is explained by 2 eigenvalues. In BCI decoding, we can think of the firing rate of each neuron as a single point in a space with dimensionality equal to the number of neurons. Using PCA, we reduce the dimensionality of the data to some manageable size while maintaining most of the information captured in the neural activity.

4 BCI Decoders in the Real World

So far, we have gone over how to predict *target* from *firing rate*: reduce the dimensionality of recorded neural activity using PCA, then predict the target using K-means clustering in this lower-dimensional space. In practice, predicting targets isn't the most useful application of a decoder—in real-world environments where BCIs are used to control prosthetic and robotic limbs, there are dozens of possible objects or targets a person could be reaching for.

Most BCIs that are used for robot control typically record from populations of neurons and decode the velocity of the movement. The key difference between these decoders and the methods described here are that these real-world decoders require *continuous* estimation of the velocity, rather than classification into a discrete target. Many of the principles that underlie K-means and PCA are applicable to building BCI decoders in general. Broadly, we can split BCI decoding into two steps: **feature extraction** (e.g. PCA) and **decoding** (e.g. K-means clustering).

4.1 Feature Extraction

While PCA is a simple and easy way to extract structure in neural activity for decoding, it makes strong assumptions about noise and variability in the data—more specifically, it assumes that *any* variation in the data is signal and not noise. There are a lot of bells and whistles we can attach to PCA to incorporate different forms of noise modeling.

Two key extensions of PCA are **probabilistic principal components analysis (PPCA)** and **factor analysis (FA)**. PPCA adds an *isotropic* noise model to PCA. In other words, each low-dimensional factor z_n could have given rise to a cloud of high-dimensional points. This means that the projections from x_n to z_n are no longer orthogonal. Like PPCA, FA also adds a noise model to PCA, but the key difference is that the noise model is *anisotropic*.Mathematically, adding these noise models means that we can't find closed-form solutions to the principal axes—we need to use probability distributions and **maximum likelihood estimation** to optimize for the principal axes.

4.2 Decoding

How do we go beyond target prediction to continuous velocity estimation? We need some function that maps firing rates to velocity. The simplest instantiation of this is a linear mapping, or an **Optimal Linear Estimate (OLE)** decoder. Essentially, we say that the velocity of the robotic arm is a linear combination of the firing rates of our population of neurons:

$$\mathbf{v} = \mathbf{B}\mathbf{x}$$

Here, $\mathbf{v}$ is the velocity and $\mathbf{x}$ is the firing rate. $\mathbf{B}$ maps firing rates to velocity, and is estimated from training data. $\mathbf{B}$ is estimated such that the predicted velocity $\mathbf{v}$ best matches the training data velocity, using **least squares regression**.

As with PCA, we can build up a lot of extensions of this linear relationship between velocity and firing rate. A commonly used extension is the **Kalman filter**. The Kalman filter adds a noise estimate and time-dependence to the linear decoder by assuming that the current velocity is a function of both the current firing rate and the previous velocity. Once again, incorporating noise estimates and prior states of the robotic arm mean that we need to generalize our mathematical equations to probability distributions and find the most probable outcome of our observations.

5 Summary and Key Takeaways

BCI Decoders almost always involve two steps: 1) Feature extraction and 2) Decoding. PCA is a method of feature extraction, and K-means clustering is a simple decoding method. Both these methods can be expanded upon to build more sophisticated decoders.

In nearly all of these extensions, we take into account the *probability* of an output in response to a given input. Thus, the underlying tenet of almost all machine learning algorithms is **maximum likelihood estimation**. When we incorporate these probabilistic estimates into our models, we get much more powerful and nuanced insights into the potential outcomes of our observed data. While neither PCA nor K-means clustering incorporates these probabilistic approaches, take Byron's class (Neural Signal Processing) to get an in-depth look at these approaches.