In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from tqdm import tqdm
np.random.seed(0)
```

Fitting a 2AFC psychometric function

Consider a two-alternative forced-choice (2AFC) psychophysical experiment (fancy name: heterochromatic brightness matching). Subjects are shown a blue spot and a red spot and must decide which appears brighter. The intensity of the blue spot is fixed, and that of the red spot is randomly varied over trials. The purpose of the experiment is to estimate the intensity of red that matches the blue as well as the difference in intensity required for the two to be noticeably different. For a red spot of brightness I, the probability of the observer saying "The red spot is brighter" is:

$$p(I) = \lambda * 12 + (1 - \lambda) * \Phi(I; \mu, \sigma),$$

where $\Phi(I; \mu, \sigma)$ is the cumulative distribution function of the Gaussian (an erf function, normcdf in matlab) with mean μ and standard deviation σ, evaluated at I. The parameter λ is called the "lapse rate" and is the proportion of trials the observer didn't pay attention and just guessed. The function $p(I)$ is known as the psychometric function. You will start by simulating performance in this task. Then, you'll simulate the inverse (scientific) side of the problem, and use this probabilistic model as a means of fitting/analyzing the simulated data set, estimating its parameters and comparing models.

In [2]:
```
from scipy.stats import norm

def erf(x, mu, sigma):
    return norm.cdf(x, loc=mu, scale=sigma)

def psychometric(I, lambd, mu, sigma):
    return lambd * 0.5 + (1 - lambd) * erf(I, mu, sigma)
```

a)

Write a function `B=simpsych(lambda,mu,sigma,I,T)` to simulate an experiment. The arguments (I, T) are vectors of equal length, the first containing a list of intensities and the second containing the number of trials to be run for each corresponding intensity. The function should generate draws from $p(I)$, and return a vector, B, (of the

same length as I and T), containing the number of trials for which the simulated observer responded that the red spot was brighter, for each intensity I.

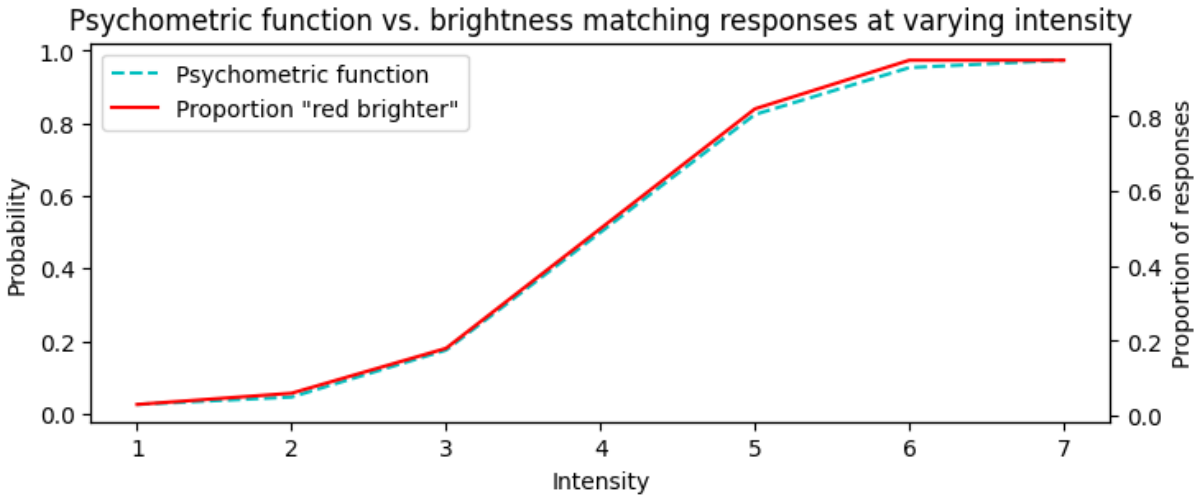
In [3]:
```
from scipy.stats import binom

def simpsych(lambd, mu, sigma, I, T):
    psych = psychometric(I, lambd, mu, sigma)
    return binom.rvs(n=T, p=psych)
```

b)

Illustrate the use of simpsych with `T=ones(1,7)*100` and `I=1:7` for $\lambda = 0.05$, $\mu = 4$ and $\sigma = 1$. Plot `B ./ T` vs `I` (as points) and plot the psychometric function $p(I)$ (as a curve) on the same graph.

In [4]:
```
intensities = np.arange(1, 8)
ntrials = 100 * np.ones(7, dtype='int')
nsuccess = simpsych(lambd=0.05, mu=4, sigma=1, I=intensities, T=ntrials)
```

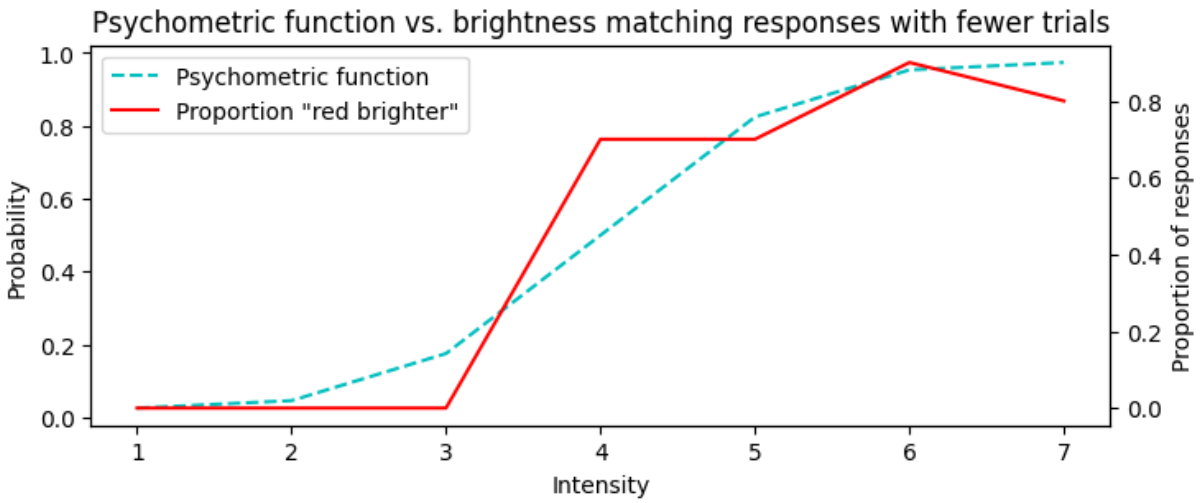
In [5]:
```
fig, ax1 = plt.subplots(figsize=(8, 3))
ax2 = plt.twinx()
psych_fn = psychometric(intensities, lambd=0.05, mu=4, sigma=1)
l1, = ax1.plot(intensities, psych_fn, 'c--')
l2, = ax2.plot(intensities, nsuccess / ntrials, 'r')
ax1.legend([l1, l2], ['Psychometric function', 'Proportion "red brighter"'])
ax1.set_xlabel('Intensity')
ax1.set_ylabel('Probability')
ax2.set_ylabel('Proportion of responses')
plt.title('Psychometric function vs. brightness matching responses at varyir
plt.show()
```



c)

Do the same with `T=ones(1,7)*10` and plot the results (including the psychometric function). What is the difference between this and the plot of the previous question?

In [6]:
```
ntrials2 = 10 * np.ones(7, dtype='int')
nsuccess2 = simpsych(lambd=0.05, mu=4, sigma=1, I=intensities, T=ntrials2)
fig, ax1 = plt.subplots(figsize=(8, 3))
ax2 = plt.twinx()
l1, = ax1.plot(intensities, psych_fn, 'c--')
l2, = ax2.plot(intensities, nsuccess2 / ntrials2, 'r')
ax1.legend([l1, l2], ['Psychometric function', 'Proportion "red brighter"'])
ax1.set_xlabel('Intensity')
ax1.set_ylabel('Probability')
ax2.set_ylabel('Proportion of responses')
plt.title('Psychometric function vs. brightness matching responses with fewe
plt.show()
```



In this plot, the proportion of "red brighter" doesn't match the psychometric function as well. Variation in the response proportion is greater because the number of trials is lower. This results in estimates with greater dispersion around the underlying psychometric value.

d)

Write a function `nll = nloglik(mu,sigma,lambda,I,T,B)` that returns the negative log likelihood of parameters `mu`, `sigma`, and `lambda` for data set `I,T,B` (we're negating it because we will be minimizing this function to solve for the optimal parameters).

In [7]:
```
def nloglik(mu, sigma, lambd, I, T, B):
    psych = psychometric(I, lambd, mu, sigma)
    likelihoods = binom.pmf(k=B, n=T, p=psych)
    return -np.sum(np.log(likelihoods))
```

e)

Use the matlab function `fminsearch` to estimate the values of `mu`, `sigma`, and `lambda` that minimize the function `nloglik(mu,sigma,lambda,...)` for the dataset you generated in (b). You'll need to specify a start point for the search – for this problem, `[2,2,.05]` is a reasonable choice. Were the estimates close to the true values used to generate the data?

In [8]:
```
from scipy.optimize import minimize

x0 = [2, 2, 0.5]
optimizer = minimize(lambda x: nloglik(x[0], x[1], x[2], intensities, ntrial
x_opt = optimizer.x
print("Estimated values:")
print(f"mu: {x_opt[0]}")
print(f"sigma: {x_opt[1]}")
print(f"lambda: {x_opt[2]}")
```

```
Estimated values:
mu: 3.994884304565403
sigma: 0.9717290920129736
lambda: 0.07656418667196602
```

The estimates are close to the true values $\mu = 4$, $\sigma = 1$ and $\lambda = 0.05$ used to generate the data.

f)

A variant of `fminsearch`, `fminunc`, also returns the Hessian (the matrix of second derivatives) of the negative log likelihood at the optimal values `mu`, `sigma` and `lambda`. (Note: `fminunc` is less robust than `fminsearch`, and if the optimizer strays too far from the true values, there may be numerical problems due to overflow of the likelihood; in this case, try a different starting point.) The inverse of the Hessian provides an estimate of the covariance matrix of the parameter estimates. Use this to determine 95% confidence intervals on each parameter (Hint: a 95% confidence interval is the mean ±1.96 standard deviations of the parameter estimate. Compute the standard deviation of a marginal of the 3-D Gaussian that has covariance equal to the inverse Hessian.) Do the true parameter values (4, 1 and .05) fall within these confidence intervals?

In [9]:
```
def marginal_std(C, axis):
    u = np.identity(C.shape[0])[axis]
    var = u @ C @ u
    return np.sqrt(var)
```

```
def confidence_interval(mu, std):
    return (mu - std * 1.96, mu + std * 1.96)
```

In [10]:
```
x0 = [4, 1, 0.05]
f = lambda x: nloglik(x[0], x[1], x[2], intensities, ntrials, nsuccess)
optimizer = minimize(f, x0, method="BFGS")
x_opt = optimizer.x
cov = optimizer.hess_inv
print("Estimated values:")
print(f"mu: {x_opt[0]}")
print(f"sigma: {x_opt[1]}")
print(f"lambda: {x_opt[2]}")
```

```
Estimated values:
mu: 3.9948886239779053
sigma: 0.9717287033252878
lambda: 0.07657618185539616
```

In [11]:
```
x_std = np.array([
    marginal_std(cov, axis=0),
    marginal_std(cov, axis=1),
    marginal_std(cov, axis=2)
])
print("Standard deviation:")
print(f"mu: {x_std[0]}")
print(f"sigma: {x_std[1]}")
print(f"lambda: {x_std[2]}")
```

```
Standard deviation:
mu: 0.0865179015471084
sigma: 0.12067645082165726
lambda: 0.025603951372538684
```

In [12]:
```
intervals = confidence_interval(x_opt, x_std)
print("Confidence intervals:")
for i, var in enumerate(['mu', 'sigma', 'lambda']):
    print(f'{var}: [{intervals[0][i]}, {intervals[1][i]}]')
```

```
Confidence intervals:
mu: [3.825313536945573, 4.164463711010238]
sigma: [0.7352028597148396, 1.208254546935736]
lambda: [0.026392437165220342, 0.126759926545572]
```

Yes, the true parameters (4, 1 and .05) fall within the 95% confidence intervals for `mu`, `sigma` and `lambda`. But the marginal has relatively large standard deviations for `sigma` and `lambda` compared to their parameter values, indicating a low-certainty estimate.

g)

Produce a second set of confidence intervals for the parameters using a bootstrap method. For each of the 7 intensities, resample 100 trials (i.e., responses that the red spot is brighter or darker) from the 100 trials of that intensity in the original data, with

replacement. Refit the model to the resampled data using `fminsearch`. Plot the histograms (function `hist`) of `mu`, `sigma` and `lambda` estimates obtained over 500 such resampled datasets, and define your confidence intervals as the region between the 2.5th and 97.5th percentiles of these distributions. How well do these values agree with those from part (f)?

In [13]:
```
def resample_dataset(nsuccess):
    B = np.zeros_like(nsuccess, dtype='int')
    for i, k in enumerate(nsuccess):
        outcomes = np.zeros(100)
        outcomes[:k] = 1
        inds = np.random.choice(100, 100, replace=True)
        B[i] = np.sum(outcomes[inds])
    return B
```

In [14]:
```
def fit(x0, I, T, B):
    f = lambda x: nloglik(x[0], x[1], x[2], I, T, B)
    optimizer = minimize(f, x0, method="Nelder-Mead")
    x_opt = optimizer.x
    return x_opt
```

In [15]:
```
X_opt = np.zeros((500, 3))
for i in tqdm(range(500)):
    B = resample_dataset(nsuccess)
    x0 = [4, 1, 0.05]
    X_opt[i, :] = fit(x0, intensities, 100, B)
```

```
100%|██████████| 500/500 [00:03<00:00, 134.95it/s]
```

In [16]:
```
intervals = np.quantile(X_opt, [0.025, 0.975], axis=0)
for i, var in enumerate(['mu', 'sigma', 'lambda']):
    print(f'{var}: [{intervals[0][i]}, {intervals[1][i]}]')
```

```
mu: [3.843695226144007, 4.153050745306348]
sigma: [0.7253456926876624, 1.2502586932746107]
lambda: [0.013244764430847029, 0.12510628064270918]
```

These values **agree well** with those from part (f).

i)

Simulate the experiment using `simpsych` twice, once using the original parameters and again changing the `mu` value to 6. We now consider a couple of approaches to test whether those two datasets differ significantly in `mu`. First, pool the datasets (treat it as one big dataset for a single psychometric function) and fit using your code from part (d). Now, write a new function, `nloglik2`, that computes the likelihood of a 4-parameter model: shared values of `sigma` and `lambda` for the two datasets, but separate parameters `mu1` and `mu2` for each dataset. Fit this model to the data from the two

simulations. Report the fit parameters for these. Now, compare the two models using AIC and BIC (reusing the results of the fit from part (g)). What are the AIC and BIC statistics and do they "seem" large enough to support the more complex model (i.e., that the two `mu` values differ)?

In [17]:
```
nsuccess1 = simpsych(lambd=0.05, mu=4, sigma=1, I=intensities, T=ntrials)
nsuccess2 = simpsych(lambd=0.05, mu=6, sigma=1, I=intensities, T=ntrials)
```

Fit using `nloglik`:

In [18]:
```
x0 = [5, 1, 0.05]
fit(x0, intensities, 2 * ntrials, nsuccess1 + nsuccess2)
print(f"mu: {x_opt[0]}")
print(f"sigma: {x_opt[1]}")
print(f"lambda: {x_opt[2]}")
```

```
mu: 3.9948886239779053
sigma: 0.9717287033252878
lambda: 0.07657618185539616
```

In [19]:
```
def nloglik2(mu1, mu2, sigma, lambd, I, T, B1, B2):
    psych1 = psychometric(I, lambd, mu1, sigma)
    psych2 = psychometric(I, lambd, mu2, sigma)
    likelihood1 = binom.pmf(k=B1, n=T, p=psych1)
    likelihood2 = binom.pmf(k=B2, n=T, p=psych2)
    return -np.sum(np.log(likelihood1) + np.log(likelihood2))
```

Fit using `nloglik2`:

In [20]:
```
x0 = [4, 6, 1, 0.05]
f = lambda x: nloglik2(x[0], x[1], x[2], x[3], intensities, ntrials, nsucces
optimizer = minimize(f, x0, method="Nelder-Mead")
x_opt = optimizer.x
print(f"mu1: {x_opt[0]}")
print(f"mu2: {x_opt[1]}")
print(f"sigma: {x_opt[2]}")
print(f"lambda: {x_opt[3]}")
```

```
mu1: 4.096775966348485
mu2: 6.034264364166409
sigma: 1.1300996814082842
lambda: 0.03039780700037924
```

Compute AIC and BIC for pooled dataset using `nloglik` and `X_opt` from part (g):

In [21]:
```
theta = X_opt.mean(axis=0)
data = nsuccess1 + nsuccess2
nll = nloglik(*theta, intensities, 2 * ntrials, data)
aic = 2 * len(theta) + 2 * nll
bic = len(theta) * np.log(len(data)) + 2 * nll
print(f"AIC: {aic}")
print(f"BIC: {bic}")
```

```
AIC: 352.1862375882357
BIC: 352.02396803540165
```

Compute AIC and BIC for datasets using `nloglik2` and `x_opt` above:

In [22]:
```
theta = x_opt
data1 = nsuccess1
data2 = nsuccess2
nll = nloglik2(*theta, intensities, ntrials, data1, data2)
aic = 2 * len(theta) + 2 * nll
bic = len(theta) * np.log(len(data1) + len(data2)) + 2 * nll
print(f"AIC: {aic}")
print(f"BIC: {bic}")
```

```
AIC: 68.43124588935922
BIC: 70.98747520782025
```

The AIC and BIC values for the complex model are around 65, and those for the simple value are around 265. The simple model AIC/BIC values are of O(100); these are values for a well-fit model. So, I think the AIC/BIC values for the complex model, which are also of O(100), indicate a relatively good fit. I think the complex model **is supported** by these AIC/BIC values.

i)

Finally, construct a permutation test of the null hypothesis (i.e., the hypothesis that there has been no change in mu between the two datasets). For each intensity, combine the 100 trials from each condition into a total of 200, then randomly partitioning this into two groups of 100. Fit both resampled datasets with your original one-sample model, noting the difference between the two mu estimates. Repeat this process 500 times to produce a null distribution of the differences. Now fit the two datasets separately and compute the difference between the two `mu` estimates. How likely (at what quantile; one- tailed p-value) is that difference in `mu` according to the null distribution? Do these results make sense given the true parameter values from which you simulated the datasets?

In [23]:
```
def resample_datasets(nsuccess1, nsuccess2):
    resampled1 = np.zeros_like(nsuccess1)
    resampled2 = np.zeros_like(nsuccess2)
    for i, n in enumerate(nsuccess1 + nsuccess2):
        t = np.zeros(200)
        t[:n] = 1
        np.random.shuffle(t)
        n1 = np.sum(t[:100])
        n2 = np.sum(t[100:])
        resampled1[i] = n1
        resampled2[i] = n2
    return resampled1, resampled2
```

In [24]:
```
resampled1, resampled2 = resample_datasets(nsuccess1, nsuccess2)
```

In [25]:
```
def fit_separate(data1, data2):
    x0 = [5, 1, 0.05]
    x_opt1 = fit(x0, intensities, 100, data1)
    x_opt2 = fit(x0, intensities, 100, data2)
    mu1 = x_opt1[0]
    mu2 = x_opt2[0]
    return mu1, mu2
```

In [26]:
```
mu1, mu2 = fit_separate(resampled1, resampled2)
print("Shuffled:")
print(f"mu1: {mu1}")
print(f"mu2: {mu2}")
```

```
Shuffled:
mu1: 5.275837239315857
mu2: 4.843051939124104
```

When the datasets are shuffled across each other, both give estimates of `mu` around 5.

Let's repeat 500 times and get a null distribution over `mu2 - mu1`:

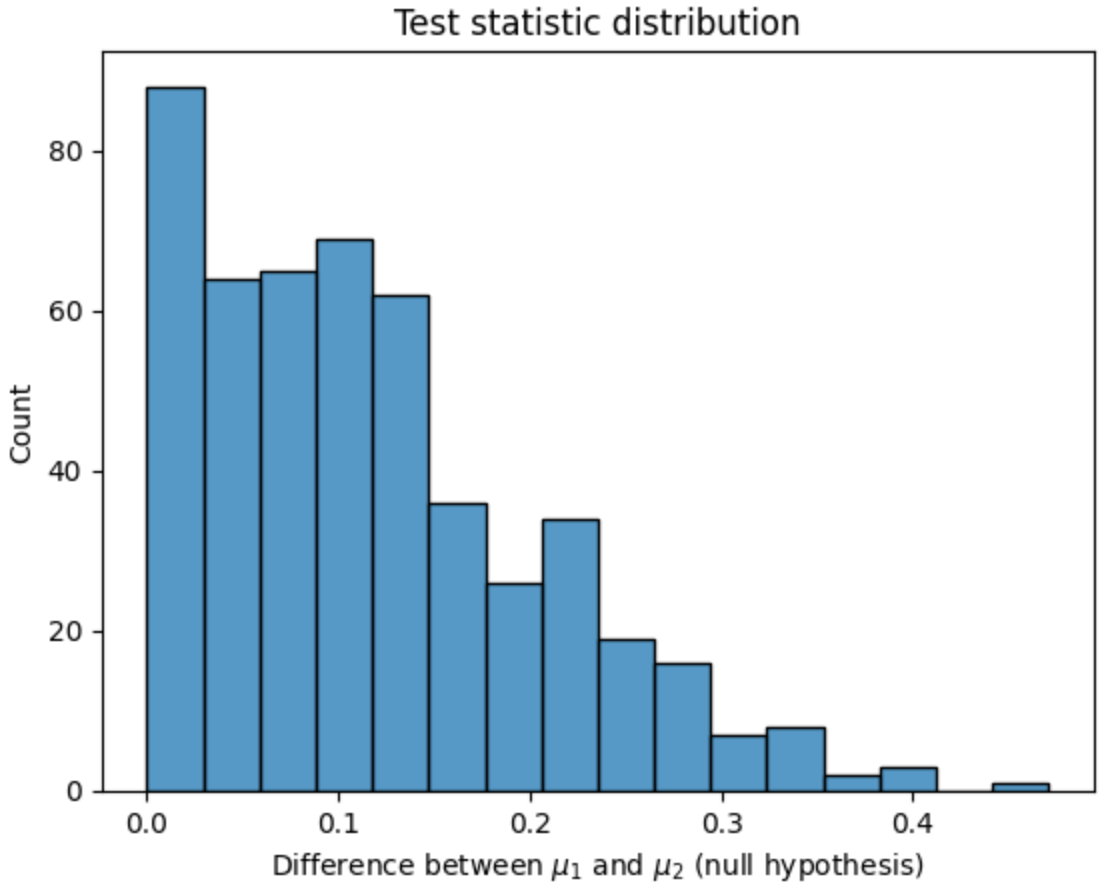
In [27]:
```
diffs = np.zeros(500)
for i in tqdm(range(500)):
    resampled1, resampled2 = resample_datasets(nsuccess1, nsuccess2)
    mu1, mu2 = fit_separate(resampled1, resampled2)
    diffs[i] = np.abs(mu2 - mu1)
```

```
100%|██████████| 500/500 [00:08<00:00, 55.94it/s]
```

How is the test statistic distributed?

In [28]:
```
sns.histplot(diffs)
plt.title('Test statistic distribution')
plt.ylabel('Count')
plt.xlabel('Difference between $\mu_1$ and $\mu_2$ (null hypothesis)')
print(f"Test statistic max: {max(diffs)}")
```

```
Test statistic max: 0.4709543547232844
```



Now fit the (unshuffled) datasets separately:

In [29]:
```
mu1, mu2 = fit_separate(nsuccess1, nsuccess2)
print("Unshuffled:")
print(f"mu1: {mu1}")
print(f"mu2: {mu2}")
```

```
Unshuffled:
mu1: 4.096290764492823
mu2: 6.040772323841598
```

These have a difference close to 2, compared to the shuffled differences which are concentrated tightly around 0.

The test statistic has a max around 0.5. Not a single value in the test statistic distribution exceeds `mu2 - mu1` for our two original datasets. Since we drew 500 samples, this indicates a p-value <0.002.

This means it is highly, highly unlikely that the two datasets were generated by two distributions with the same μ. It is sensible to conclude that we are not mistakenly assuming a difference. This does make sense given the ground truth: we generated the datasets with $\mu = 4$ and $\mu = 6$ respectively, and given enough samples, random draws from a unimodal Gaussian cannot replicate them.

In [1]:
```
import pandas as pd
from matplotlib import pyplot as plt
from scipy.io import loadmat
import seaborn as sns
import numpy as np
np.random.seed(0)
```

Psychopathy

You are interested in causes and treatment options for psychopathy. You obtained a dataset, contained in the file psychopathy.mat obtained from a prison for violent offenders in upstate New York (not everyone in the prison is a psychopath, but they are more prevalent than in the general population). All study participants underwent a structural scan with a mobile, truck-mounted MRI. Each row of the matrix represents data from one prisoner. The first column contains the estimated cortical volume of paralimbic areas, relative to the population median, in cm3. The second column contains the Hare Psychopathy Checklist (PCL-R) scores, which range from 0 to 40 (the higher the score, the more psychopathic traits someone exhibits). These scores are not distributed normally in either the general population (median = 4) or this prison subpopulation (median = 20). The third column indicates whether they already participated in an experimental treatment program known as “decompression therapy” (0 = did not yet participate, 1 = did already participate). To avoid self-selection effects, everyone in this dataset agreed to the therapy, but prisoners were randomly assigned to an earlier and a later treatment group, so that the untreated prisoners could serve as a control group.

In [2]:
```
X = loadmat('psychopathy.mat')['DATA']
df = pd.DataFrame(X)
df.index.name = 'subject'
df.columns.name = 'measurement'
df.columns = ['paralimbic_vol', 'pclr_score', 'treated']
df = df.astype({'pclr_score': int, 'treated': bool})
display(df.sample(5))
```

	paralimbic_vol	pclr_score	treated
subject			
26	-1.924975	21	False
27	-1.140256	27	False
48	1.202118	21	False
22	1.254971	10	True
30	-1.017219	15	True

a)

Use polynomial regression to model PCL-R scores as a function of relative volume of paralimbic areas. (Note, you might make use of your code from HW2.) Use cross-validation to determine the best polynomial degree.

In [3]:
```
def fit(X, y):
    U, S, Vt = np.linalg.svd(X)
    inds = np.arange(X.shape[1])
    S_inv = np.zeros((X.shape[1], X.shape[0]))
    S_inv[inds, inds] = 1 / S
    X_inv = np.dot(Vt.T, np.dot(S_inv, U.T))
    beta = np.squeeze(np.dot(X_inv, y))
    return beta
```

In [4]:
```
def design_matrix(x, order=1):
    n = len(x)
    X = []
    for i in range(order + 1):
        X.append(x ** i)
    return np.concatenate([X]).T
```

Run a leave-one-out cross-validation round on every sample in the dataset. We will average MSE across these runs to estimate generalization error for solutions of each order.

In [5]:
```
def hold_one_split(X, y, i):
    return x.drop(index=i), y.drop(index=i), x.loc[i], y.loc[i]
```

In [6]:
```
x = df.paralimbic_vol
y = df.pclr_score
orders = pd.Series([1, 2, 3, 4, 5], name='order')
mse = np.zeros((len(x), 5))
for i in range(len(x)):
    x_train, x_test, y_train, y_test = hold_one_split(x, y, i)
    for order in orders:
        X_train = design_matrix(x_train, order=order)
        X_test = design_matrix(x_test, order=order)
        beta = fit(X_train, y_train)
        y_pred = X_test @ beta
        mse[i, order - 1] = np.mean((y_pred - y_test) ** 2)
```

MSE measured over k-fold cross-validation, where k is the number of samples in the dataset:

In [7]:
```
display(pd.DataFrame(mse, columns=orders).describe().iloc[1:3])
```

order	1	2	3	4	5
mean	521.373184	606.884811	51080.329123	3.871023e+07	2.490201e+10
std	338.264853	380.468207	43551.055231	3.366031e+07	2.165602e+10

The first-order polynomial fits the target best, as it has the lowest averaged leave-one-out MSE (521). The standard deviation of these MSEs (for MSE is itself a random variable) is also the lowest for order 1.

However, the MSE estimates for orders 1 and 2 are somewhat overlapping (note their standard deviation). So I'm not sure how confident we should be in using it for model selection.

b)

Use bootstrapping methods to estimate the 95% confidence interval of the average paralimbic volume of the decompression treatment group vs. the control group. If the random assignment worked, the confidence intervals should overlap. Do they? Also, do these data suggest that there is a statistically reliable difference from the general population, in terms of paralimbic volume?

In [8]:
```
treated = df[df.treated]
untreated = df[~df.treated]

treated_avgs = np.zeros(500)
untreated_avgs = np.zeros(500)
for i in range(500):
    treated_resampled = treated.sample(len(treated), replace=True)
    untreated_resampled = untreated.sample(len(untreated), replace=True)
    treated_avgs[i] = treated_resampled.paralimbic_vol.mean()
    untreated_avgs[i] = untreated_resampled.paralimbic_vol.mean()

confidence = np.quantile(treated_avgs, [0.05, 0.95])
print(f"Treated 95% confidence interval: [{confidence[0]}, {confidence[1]}]"
confidence = np.quantile(untreated_avgs, [0.05, 0.95])
print(f"Untreated 95% confidence interval: [{confidence[0]}, {confidence[1]}
```

```
Treated 95% confidence interval: [-1.3100843375340006, -0.7767115288206876]
Untreated 95% confidence interval: [-1.0208844831583854, -0.2263564929210839
5]
```

The confidence intervals overlap. We may interpret these data as saying that with 95% certainty, the true values for average treated and untreated relative paralimbic volume are between [-1.3 and -0.78] and [-1.0 and -0.23], respectively. This means that for both treated and untreated, there is <5% probability that the true value is greater than or equal to 0. In other words, if in fact there is no statistically reliable difference, we have <5% probability of incorrectly concluding there is a difference (rejecting the null

hypothesis). This is equivalent to a p-value <0.05, meeting a conventional criterion for "statistical significance".

However, I note that it is difficult to reason properly about statistical differences without access to the full distribution of paralimbic measurements from the general population. We only have access to a point estimate (the median) of the general population's paralimbic volume. Imagine that variability in the general population is already much greater than the widths of our confidence intervals. We do not know whether this is the case. If so, I doubt these confidence intervals should be interpreted as "statistically reliable".

c)

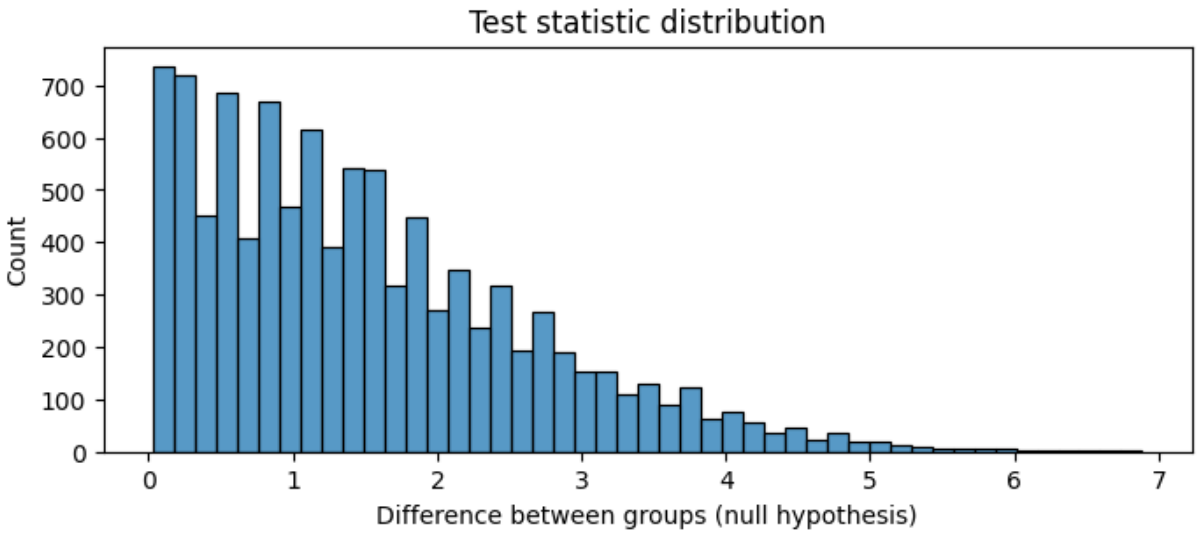
Do a permutation test to assess whether decompression therapy has an effect. Designate an appropriate test statistic and calculate its p-value.

We have already split the dataset into `treated` and `untreated` groups. We will test whether the difference in mean PCL-R score between the treated and untreated group is significantly greater than the expected difference in mean PCL-R scores for two equal-sized groups randomly sampled from the whole population.

In [9]:
```
diffs = np.zeros(10000)
for i in range(10000):
    pooled = pd.concat([treated, untreated])
    shuffled = pooled.sample(len(pooled), replace=False)
    group1 = shuffled.iloc[:len(pooled) // 2]
    group2 = shuffled.iloc[len(pooled) // 2:]
    mean1 = group1.pclr_score.mean()
    mean2 = group2.pclr_score.mean()
    diffs[i] = np.abs(mean2 - mean1)
```

Here is the distribution of differences for two groups randomly sampled from the population:

In [10]:
```
plt.subplots(figsize=(8, 3))
plt.title('Test statistic distribution')
plt.ylabel('Count')
plt.xlabel('Difference between groups (null hypothesis)')
sns.histplot(diffs)
plt.show()
```



Now we compute mean PCL-R scores for the treated and untreated groups separately, and compare this difference to our null statistic.

In [11]:
```
mean_treated = treated.pclr_score.mean()
mean_untreated = untreated.pclr_score.mean()
treatment_diff = mean_untreated - mean_treated
print(f"Treatment difference: {treatment_diff}")
```

```
Treatment difference: 3.2285714285714278
```

How likely is it that a random sample drawn from the test statistic will exceed this difference value? We can simply ask what proportion of samples exceeded this value in our permutation test. That is our p-value.

In [12]:
```
p = np.sum(diffs > treatment_diff) / len(diffs)
p
```

Out[12]:
```
0.0882
```

Our p-value exceeds 0.08. So, by conventional standards the decompression therapy **does not** have a statistically significant effect.

In [ ]:
```

```

In [1]:
```
import pandas as pd
from matplotlib import pyplot as plt
from scipy.io import loadmat
import seaborn as sns
import numpy as np
np.random.seed(0)
```

Classification in a 2-dimensional space

The file `fisherData.mat` contains two data matrices, `data1` and `data2`, whose rows contain hypothetical normalized responses of 2 mouse auditory neurons to different stimuli – The first matrix contains responses to dogs barking, and the second are responses to cat vocalizations. You would like to know whether the responses of these two neurons could be used by the mouse to differentate the two types of sound. We'll implement three classifiers.

In [2]:
```
data = loadmat('fisherData.mat')
index = pd.Series(np.arange(len(data['data1'])), name='trial')
columns = pd.Series([0, 1], name='neuron')
dog = pd.DataFrame(data['data1'], columns=columns, index=index)
cat = pd.DataFrame(data['data2'], columns=columns, index=index)
```

a)

The prototype classifier assigns to observations the label of the class of training samples whose mean is closest to the observation. To be precise, given an example x, the classifier assigns it a class $\hat{y}$ according to the rule

$$\hat{y} = \begin{cases} A & \text{if } \hat{w}^T x > 0 \\ B & \text{otherwise} \end{cases}$$ where the discriminant vector $\hat{w}$ is $\frac{\mu_A - \mu_B}{\|\mu_A - \mu_B\|}$.

The origin of our coordinate system is taken to be exactly between the means of each class (we achieve this by subtracting $\frac{1}{2}(\mu_A + \mu_B)$ from x as a preprocessing step).

Assume that samples of class A are drawn from $\mathcal{N}(\mu_A, I)$ and samples of class B are drawn from $\mathcal{N}(\mu_B, I)$. Then classes A and B have likelihood functions

$$p(x|y = A) = \frac{1}{\sqrt{(2\pi)^N |I|}} e^{-\frac{1}{2}(x-\mu_A)^T I (x-\mu_A)} = \frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{1}{2}(x-\mu_A)^T (x-\mu_A)} \text{ and}$$

$$p(x|y = B) = \frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{1}{2}(x-\mu_B)^T (x-\mu_B)}.$$

A maximum likelihood classifier assigns to observations the label of the class whose likelihood function is maximized by the observation. It assigns an example x to class $\hat{y}$

according to the rule

$$\hat{y} = \begin{cases} A & \text{if } p(x|y=A) > p(x|y=B) \\ B & \text{otherwise} \end{cases} = \begin{cases} A & \text{if } \frac{p(x|y=A)}{p(x|y=B)} > 1 \\ B & \text{otherwise} \end{cases} = \begin{cases} A & \text{if } \ln[\frac{p(x|y=A}{p(x|y=E} \\ B & \text{otherwise} \end{cases}$$

We must show that the two different decision rules stated above are equivalent.

Consider the quantity $\ln \frac{p(x|y=A)}{p(x|y=B)}$. We may rewrite it as

$$\ln[\frac{\frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{1}{2}(x-\mu_A)^T(x-\mu_A)}}{\frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{1}{2}(x-\mu_B)^T(x-\mu_B)}}] = \ln[\frac{e^{-\frac{1}{2}(x-\mu_A)^T(x-\mu_A)}}{e^{-\frac{1}{2}(x-\mu_B)^T(x-\mu_B)}}] = -\frac{1}{2}(x-\mu_A)^T(x-\mu_A) + \frac{1}{2}(x-\mu_B)^T(x$$

$$= -\frac{1}{2}x^Tx + \frac{1}{2}x^T\mu_A + \frac{1}{2}\mu_A^Tx - \frac{1}{2}\mu_A^T\mu_A + \frac{1}{2}x^Tx - \frac{1}{2}x^T\mu_B - \frac{1}{2}\mu_A^Tx + \frac{1}{2}\mu_B^T\mu_B$$

$$= x^T\mu_A - x^T\mu_B - \frac{1}{2}(\mu_A^T\mu_A - \mu_B^T\mu_B)$$

$$= x^T(\mu_A - \mu_B) - \frac{1}{2}(\mu_A^T\mu_A - \mu_B^T\mu_B).$$

We may then write the inequality in our decision rule as
$x^T(\mu_A - \mu_B) > \frac{1}{2}(\mu_A^T\mu_A - \mu_B^T\mu_B)$ or

$w^Tx > b$, where $w = \mu_A - \mu_B$ and $b = \frac{1}{2}\mu_A^T\mu_A - \frac{1}{2}\mu_B^T\mu_B$.

Since the origin of our coordinate system is equidistant from μ_A and μ_B, $\mu_A^T\mu_A = \mu_B^T\mu_B$ and $b = 0$. Let $\hat{w}$ be the unit vector $\frac{w}{\|w\|}$. Our inequality becomes

$\hat{w}^Tx > 0.$

We may then write the ML decision rule as

$$\hat{y} = \begin{cases} A & \text{if } \hat{w}^Tx > 0 \\ B & \text{otherwise} \end{cases} \text{ where } \hat{w} = \frac{\mu_A - \mu_B}{\|\mu_A - \mu_B\|}.$$

This coincides with the decision rule for the prototype classifier. Therefore, the prototype classifier is the maximum likelihood classifier under the assumptions made above.

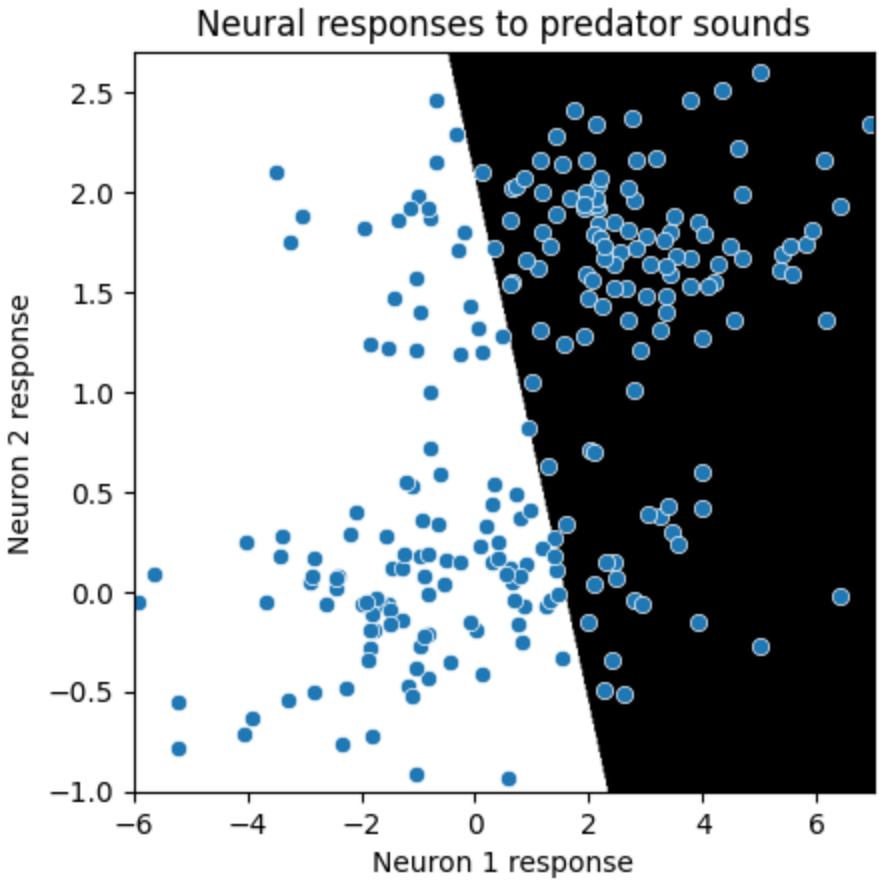
In [3]:
```
xx = np.linspace(-6, 7, 500)
yy = np.linspace(-1, 2.7, 500)
XX, YY = np.meshgrid(xx, yy)
D = np.stack([XX.flatten(), YY.flatten()]).T
```

In [4]:
```
mu1 = dog.mean(axis=0).values
mu2 = cat.mean(axis=0).values
pdf1 = np.exp(-0.5 * np.sum((D - mu1) ** 2, axis=1))
pdf2 = np.exp(-0.5 * np.sum((D - mu2) ** 2, axis=1))
```

```
mask = pdf1 > pdf2
im = mask.reshape(len(xx), len(yy))
```

We may visualize the solution by generating a binary image showing the classification output, with the data points scatterplotted on top.

In [5]:
```
plt.subplots()
plt.title('Neural responses to predator sounds')
plt.imshow(np.flipud(im), extent=(xx.min(), xx.max(), yy.min(), yy.max()), c
sns.scatterplot(pd.concat([dog, cat]), x=0, y=1)
plt.xlabel('Neuron 1 response')
plt.ylabel('Neuron 2 response')
plt.gca().set_aspect((xx.max() - xx.min())/(yy.max() - yy.min()))
```

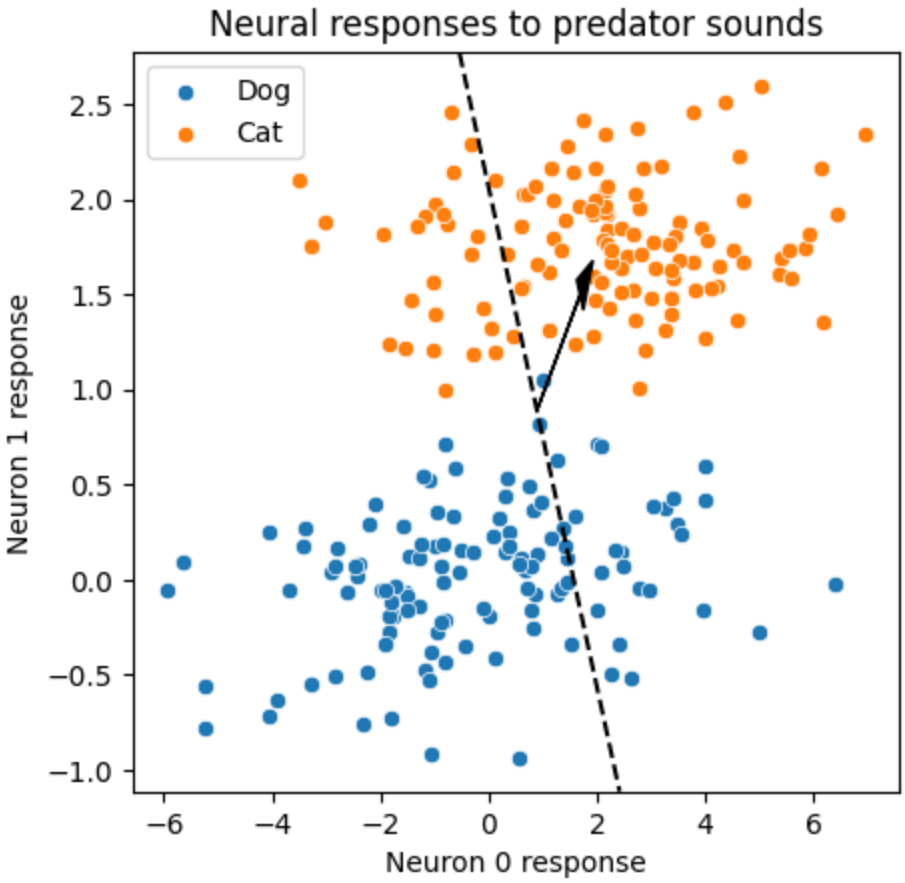


Compute the weight vector, and plot it with the decision boundary.

In [6]:
```
center = (mu1 + mu2) / 2
w = mu2 - mu1
w /= np.sqrt(w @ w)
```

In [7]:
```
plt.subplots()
plt.title('Neural responses to predator sounds')
sns.scatterplot(dog, x=0, y=1, label='Dog')
sns.scatterplot(cat, x=0, y=1, label='Cat')
plt.gca().arrow(*center, *w, head_width=0.2, color='k')
plt.axline(center, slope=-w[0]/w[1], color="black", linestyle='--')
plt.xlabel('Neuron 0 response')
```

```
plt.ylabel('Neuron 1 response')
plt.gca().set_aspect(13/3.7)
```



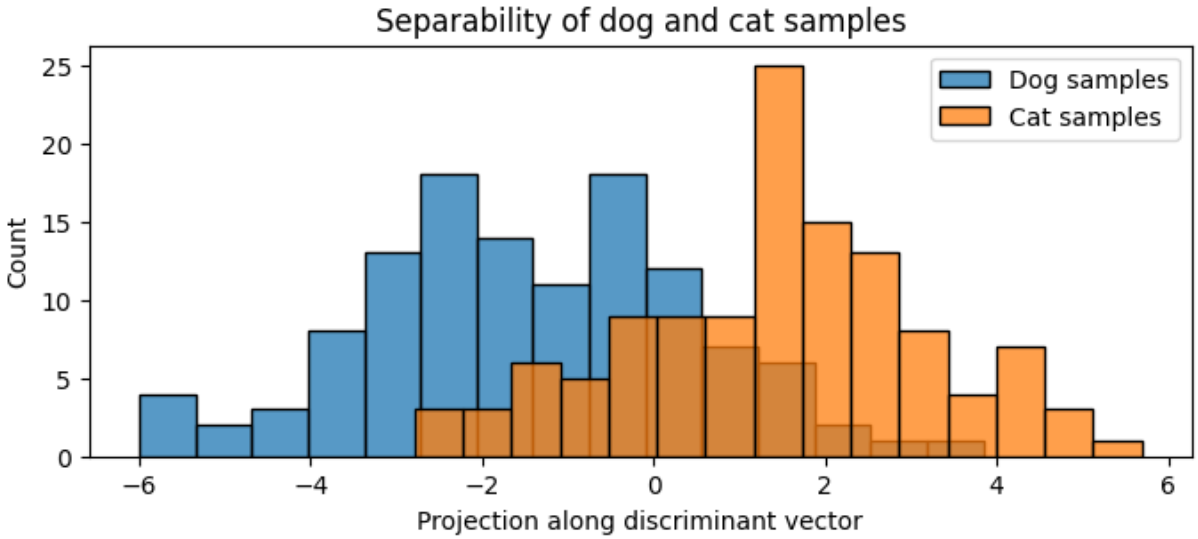
What fraction of points are correctly classified?

In [8]:
```
X = pd.concat([dog, cat]) - center
y = np.concatenate([-np.ones(len(dog)), np.ones(len(cat))])
frac = np.sum(np.sign(X @ w) == y) / X.shape[0]
print(f"Fraction correct: {frac:0.3f}")
```

```
Fraction correct: 0.783
```

We can plot the distributions of projection along the discriminant vector for each class.

In [9]:
```
dog_proj = (dog - center) @ w
cat_proj = (cat - center) @ w
plt.subplots(figsize=(8, 3))
sns.histplot(dog_proj, label='Dog samples', bins=15)
sns.histplot(cat_proj, label='Cat samples', bins=15)
plt.title('Separability of dog and cat samples')
plt.xlabel('Projection along discriminant vector')
plt.ylabel('Count')
plt.legend()
plt.show()
```



The two distributions are not very well separated. Consider a decision line bisecting their overlap (near x=0 in the histogram above). "Dog" samples to the right of this line and "cat" samples to the left of this line will be misclassified.

b)

Here we follow the same proof format as in part (a), but with the inverse covariance matrix C^{-1} factoring into most calculations.

Fisher's linear discriminant assigns to observations the class label which maximizes the average squared between-class mean distance, while minimizing the sum of within-class squared distances. It assigns an example x to a class $\hat{y}$ according to the rule

$\hat{y} = \begin{cases} A & \text{if } \hat{w}^T x > 0 \\ B & \text{otherwise} \end{cases}$ where the discriminant vector $\hat{w}$ is $C^{-1}(\frac{\mu_A}{\|\mu_A\|} - \frac{\mu_B}{\|\mu_B\|})$. C is the average of class covariance matrices $\frac{1}{2}(C_A + C_B)$.

Again, we set the origin of our coordinate system to be exactly between the means of each class by subtracting $\frac{1}{2}(\mu_A + \mu_B)$ from x in preprocessing.

Assume that samples of class A are drawn from $\mathcal{N}(\mu_A, C)$ and samples of class B are drawn from $\mathcal{N}(\mu_B, C)$, for some shared covariance matrix C. Then classes A and B have likelihood functions

$$p(x|y = A) = \frac{1}{\sqrt{(2\pi)^N |C|}} e^{-\frac{1}{2}(x-\mu_A)^T C^{-1}(x-\mu_A)}$$

$$p(x|y = B) = \frac{1}{\sqrt{(2\pi)^N |C|}} e^{-\frac{1}{2}(x-\mu_B)^T C^{-1}(x-\mu_B)}.$$

A maximum likelihood classifier assigns an example x to class $\hat{y}$ according to the rule

$$\hat{y} = \begin{cases} A & \text{if } \ln[\frac{p(x|y=A)}{p(x|y=B)}] > 0 \\ B & \text{otherwise} \end{cases}$$

Again we must show that the two different decision rules above are equivalent. We may rewrite the quantity $\ln \frac{p(x|y=A)}{p(x|y=B)}$ as

$$\ln[\frac{\frac{1}{\sqrt{(2\pi)^N|C|}}e^{-\frac{1}{2}(x-\mu_A)^TC^{-1}(x-\mu_A)}}{\frac{1}{\sqrt{(2\pi)^N|C|}}e^{-\frac{1}{2}(x-\mu_B)^TC^{-1}(x-\mu_B)}}] = \ln[\frac{e^{-\frac{1}{2}(x-\mu_A)^TC^{-1}(x-\mu_A)}}{e^{-\frac{1}{2}(x-\mu_B)^TC^{-1}(x-\mu_B)}}] = -\frac{1}{2}(x-\mu_A)^TC^{-1}(x-\mu_A) + \frac{1}{2}(x$$

$$= -\frac{1}{2}x^TC^{-1}x + \frac{1}{2}x^TC^{-1}\mu_A + \frac{1}{2}\mu_A^TC^{-1}x - \frac{1}{2}\mu_A^TC^{-1}\mu_A + \frac{1}{2}x^TC^{-1}x - \frac{1}{2}x^TC^{-1}\mu_B -$$

$$= x^TC^{-1}\mu_A - x^TC^{-1}\mu_B - \frac{1}{2}(\mu_A^TC^{-1}\mu_A - \mu_B^TC^{-1}\mu_B)$$

$$= x^TC^{-1}(\mu_A - \mu_B) - \frac{1}{2}(\mu_A^TC^{-1}\mu_A - \mu_B^TC^{-1}\mu_B).$$

We may then write the inequality in our decision rule as
$x^TC^{-1}(\mu_A - \mu_B) > \frac{1}{2}(\mu_A^TC^{-1}\mu_A - \mu_B^TC^{-1}\mu_B)$ or

$w^Tx > b$, where $w = C^{-1}(\mu_A - \mu_B)$ and $b = \frac{1}{2}\mu_A^TC^{-1}\mu_A - \frac{1}{2}\mu_B^TC^{-1}\mu_B$.

Since the origin of our coordinate system is equidistant from μ_A and μ_B, $\mu_A^TC^{-1}\mu_A = \mu_B^TC^{-1}\mu_B$ and $b = 0$. We may rescale w to the unit vector $\hat{w}$ by replacing μ_A and μ_B in its equation with unit vectors $\frac{\mu_A}{\|\mu_A\|}$ and $\frac{\mu_B}{\|\mu_B\|}$.

Then our inequality becomes

$\hat{w}^Tx > 0$ where $\hat{w} = C^{-1}(\frac{\mu_A}{\|\mu_A\|} - \frac{\mu_B}{\|\mu_B\|})$.

We may then write the ML decision rule as

$$\hat{y} = \begin{cases} A & \text{if } \hat{w}^Tx > 0 \\ B & \text{otherwise} \end{cases}.$$

This coincides with the decision rule for Fisher's linear discriminant. Therefore, it is a maximum likelihood classifier under the assumptions made above.

In [10]:
```
def cov(data):
    var = data - data.mean(axis=0)
    n = len(var) - 1
    return var.T @ var / n
```

Below we estimate the common covariance for the cat and dog datasets:

In [11]:
```
C = (cov(cat) + cov(dog)) / 2
display(C)
```

neuron	0	1
neuron		
0	4.972170	0.159544
1	0.159544	0.125787

Now we repeat the plotting exercises in part (a).

In [12]:
```
def inv(X):
    U, s, Vt = np.linalg.svd(X)
    return Vt.T @ np.diag(1 / s) @ U
```

In [13]:
```
C_inv = inv(C)
for i, d in enumerate(D):
    pdf1[i] = np.exp(-0.5 * ((d - mu1).T @ C_inv @ (d - mu1)))
    pdf2[i] = np.exp(-0.5 * ((d - mu2).T @ C_inv @ (d - mu2)))
pdf1 /= np.sum(pdf1)
pdf2 /= np.sum(pdf2)

mask = pdf1 > pdf2
im = mask.reshape(len(xx), len(yy))
```

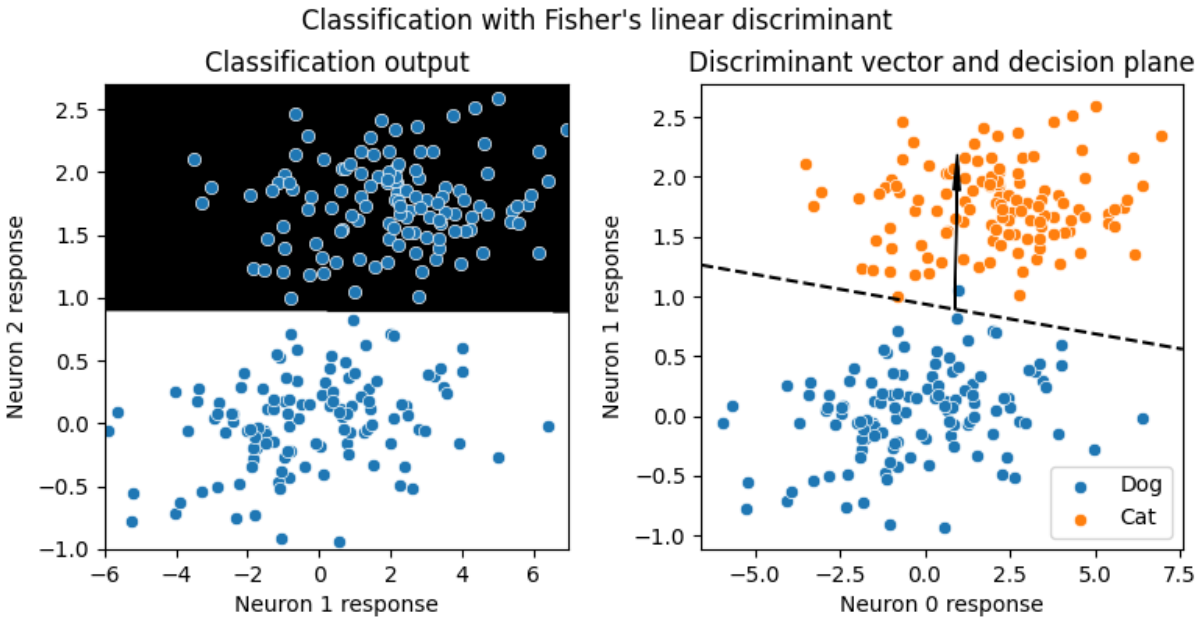
We show classification output as a binary image with data points scattered on top, and we plot the data with weight vector and decision boundary.

In [14]:
```
def unit(vec):
    return vec / np.sqrt(vec @ vec)
```

In [15]:
```
w = C_inv @ (unit(mu2) - unit(mu1))
w /= np.sqrt(w @ w)
```

In [16]:
```
fig, axs = plt.subplots(1, 2, figsize=(8, 4))
plt.suptitle("Classification with Fisher's linear discriminant")

plt.sca(axs[0])
plt.imshow(np.flipud(im), extent=(xx.min(), xx.max(), yy.min(), yy.max()), c
sns.scatterplot(pd.concat([dog, cat]), x=0, y=1)
plt.title('Classification output')
plt.xlabel('Neuron 1 response')
plt.ylabel('Neuron 2 response')
plt.gca().set_aspect((xx.max() - xx.min())/(yy.max() - yy.min()))

plt.sca(axs[1])
sns.scatterplot(dog, x=0, y=1, label='Dog')
sns.scatterplot(cat, x=0, y=1, label='Cat')
plt.gca().arrow(*center, *w, head_width=0.2, color='k')
plt.axline(center, slope=-w[0]/w[1], color="black", linestyle='--')
plt.title('Discriminant vector and decision plane')
plt.xlabel('Neuron 0 response')
plt.ylabel('Neuron 1 response')
plt.gca().set_aspect((xx.max() - xx.min())/(yy.max() - yy.min()))
```

```
plt.tight_layout()
plt.show()
```



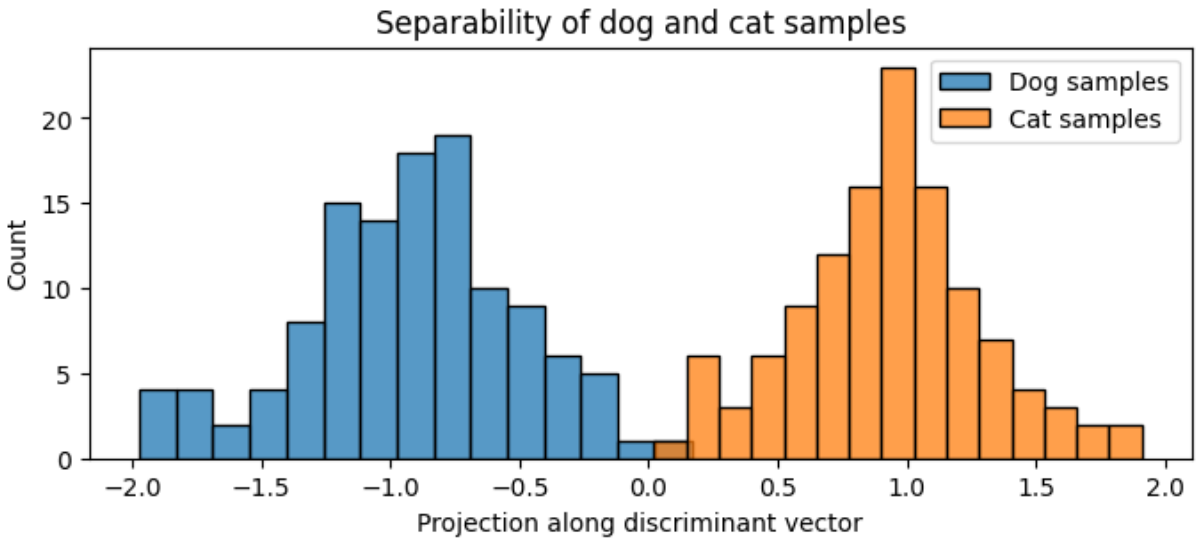
What fraction of points are correctly classified?

In [17]:
```
frac = np.sum(np.sign(X @ w) == y) / X.shape[0]
print(f"Fraction correct: {frac:0.3f}")
```

```
Fraction correct: 0.996
```

We now plot the distributions of projection along the discriminant vector for each class.

In [18]:
```
dog_proj = (dog - center) @ w
cat_proj = (cat - center) @ w
plt.subplots(figsize=(8, 3))
sns.histplot(dog_proj, label='Dog samples', bins=15)
sns.histplot(cat_proj, label='Cat samples', bins=15)
plt.title('Separability of dog and cat samples')
plt.xlabel('Projection along discriminant vector')
plt.ylabel('Count')
plt.legend()
plt.show()
```



The two distributions are quite well separated.

c)

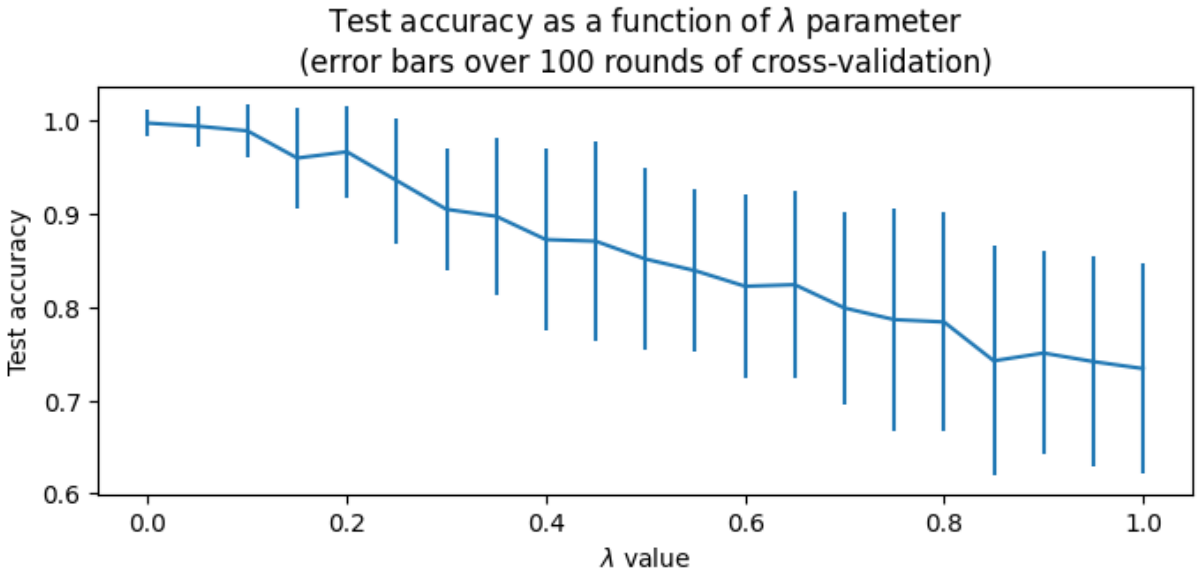
Now we compute the ridge-regularized Fisher's discriminant by estimating the covariance matrix as $\Sigma_{Estimated} = (1-\lambda)\Sigma_{Data} + \lambda I$, where Σ_{Data} is the sample covariance matrix from part (b). We run 100 repeats of cross-validation with a 95:5 train-test split for λ from 0 to 1 in increments of 0.05.

In [19]:
```
from sklearn.model_selection import train_test_split
```

In [20]:
```
lambdas = np.arange(0, 1 + 1e-10, 0.05)
score_avg = np.zeros_like(lambdas)
score_std = np.zeros_like(lambdas)
for i, lambd in enumerate(lambdas):
    fracs = np.zeros(100)
    for j in range(100):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=

        X_dog = X_train[y_train == -1]
        X_cat = X_train[y_train == 1]
        C = (cov(X_dog) + cov(X_cat)) / 2

        C_est = (1 - lambd) * C + lambd * np.identity(C.shape[0])
        w = inv(C_est) @ (unit(mu2) - unit(mu1))
        w /= np.sqrt(w @ w)

        fracs[j] = np.sum(np.sign(X_test @ w) == y_test) / X_test.shape[0]
    score_avg[i] = np.mean(fracs)
    score_std[i] = np.std(fracs)
```

In [21]:
```
plt.subplots(figsize=(8, 3))
plt.errorbar(x=lambdas, y=score_avg, yerr=score_std)
plt.title("Test accuracy as a function of $\lambda$ parameter\n(error bars c
```

```
plt.xlabel('$\lambda$ value')
plt.ylabel('Test accuracy')
plt.show()
```



I think $\lambda = 0.0$ (Fisher's discriminant) is the best value for λ, since it has the highest test accuracy and the lowest standard deviation across repeat cross-validation runs.

d)

Finally we consider the quadratic classifier. First we estimate the mean and covariance of data measured for each condition.

In [22]:
```
C_dog = cov(dog)
C_cat = cov(cat)
print("Covariance matrix for dog class:")
display(C_dog)
print("Covariance matrix for cat class:")
display(C_cat)
```

```
Covariance matrix for dog class:
```

neuron	0	1
neuron		
0	5.218407	0.251279
1	0.251279	0.138283

```
Covariance matrix for cat class:
```

neuron	0	1
neuron		
0	4.725933	0.067809
1	0.067809	0.113292

The quadratic classifier for this problem assigns an example x to class $\hat{y}$ according to the rule

$$\hat{y} = \begin{cases} dog & \text{if } \ln[\frac{p(x|y=dog)}{p(x|y=cat)}] > 0 \\ cat & \text{otherwise} \end{cases}$$

$p(x|y = dog)$ and $p(x|y = cat)$ are Gaussian with PDFs

$$p(x|y = dog) = (2\pi)^{-\frac{N}{2}}|C_{dog}|^{-\frac{1}{2}}e^{-\frac{1}{2}(x-\mu_{dog})^T C_{dog}^{-1}(x-\mu_{dog})}$$

$$p(x|y = cat) = (2\pi)^{-\frac{N}{2}}|C_{cat}|^{-\frac{1}{2}}e^{-\frac{1}{2}(x-\mu_{cat})^T C_{cat}^{-1}(x-\mu_{cat})}.$$

We may rewrite the quantity $\ln \frac{p(x|y=dog)}{p(x|y=cat)}$ as

$$\ln[\frac{|C_{dog}|^{-\frac{1}{2}}e^{-\frac{1}{2}(x-\mu_{dog})^T C_{dog}^{-1}(x-\mu_{dog})}}{|C_{cat}|^{-\frac{1}{2}}e^{-\frac{1}{2}(x-\mu_{cat})^T C_{cat}^{-1}(x-\mu_{cat})}}]$$

$$= -\frac{1}{2}|C_{dog}| - \frac{1}{2}(x-\mu_{dog})^T C_{dog}^{-1}(x-\mu_{dog}) + \frac{1}{2}|C_{cat}| + \frac{1}{2}(x-\mu_{cat})^T C_{cat}^{-1}(x-\mu_{cat})$$

.

This means we can write our decision rule as

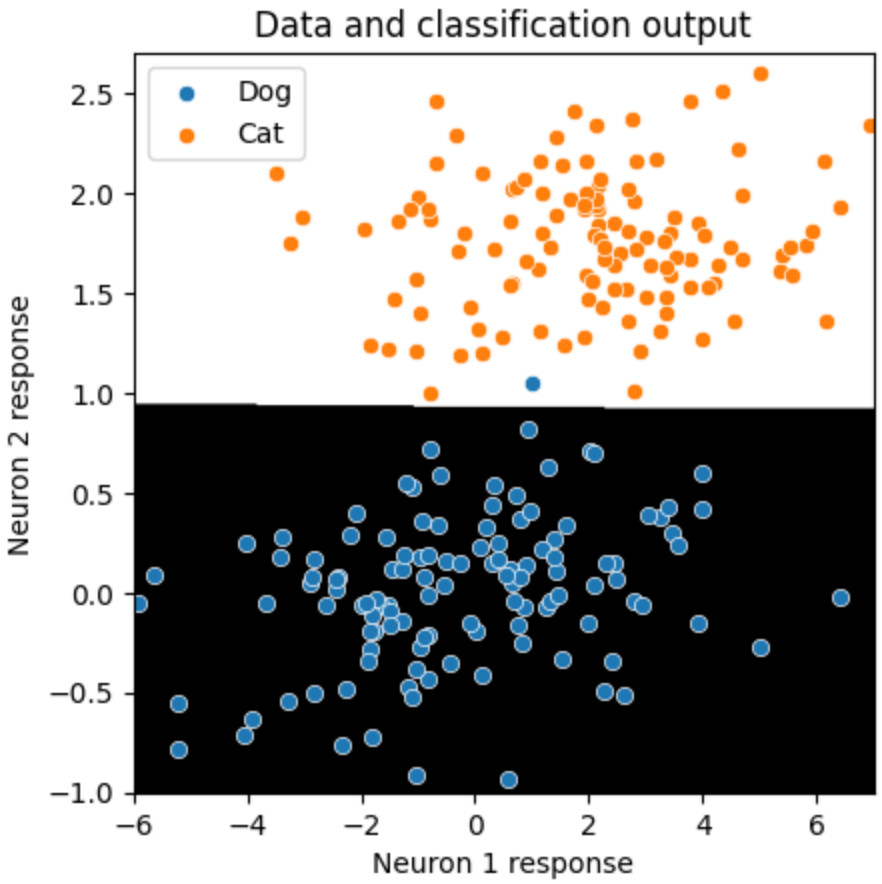
$$\hat{y} = \begin{cases} dog & \text{if } -\frac{1}{2}|C_{dog}| - \frac{1}{2}(x-\mu_{dog})^T C_{dog}^{-1}(x-\mu_{dog}) + \frac{1}{2}|C_{cat}| + \frac{1}{2}(x-\mu_{cat})^T C_{cat}^{-1} \\ cat & \text{otherwise.} \end{cases}$$

In [23]:
```
def det(C):
    return C[0, 0] * C[1, 1] - C[0, 1] * C[1, 0]
```

In [24]:
```
C_dog_inv = inv(C_dog.values)
C_cat_inv = inv(C_cat.values)
mu_dog = mu1
mu_cat = mu2
const = -0.5 * det(C_dog.values) + 0.5 * det(C_cat.values)

def classify(x):
    dog_term = -0.5 * (x - mu_dog).T @ C_dog_inv @ (x - mu_dog)
    cat_term = 0.5 * (x - mu_cat).T @ C_cat_inv @ (x - mu_cat)
    lpr = const + dog_term + cat_term
    return -1 if lpr > 0 else 1
```

We show classification output as a binary image with data points scattered on top and colored by class.

In [25]:
```
for i, d in enumerate(D):
    mask[i] = 0 if classify(d) == -1 else 1
im = mask.reshape(len(xx), len(yy))

plt.subplots()
plt.title("Quadratic classification of cats and dogs")
plt.imshow(np.flipud(im), extent=(xx.min(), xx.max(), yy.min(), yy.max()), c
sns.scatterplot(dog, x=0, y=1, label='Dog')
sns.scatterplot(cat, x=0, y=1, label='Cat')
plt.title('Data and classification output')
plt.xlabel('Neuron 1 response')
plt.ylabel('Neuron 2 response')
plt.gca().set_aspect((xx.max() - xx.min())/(yy.max() - yy.min()))
plt.show()
```



We compute the fraction correct.

In [26]:
```
X = pd.concat([dog, cat]).values
y_pred = [classify(x) for x in X]
frac = np.sum(y_pred == y) / X.shape[0]
print(f"Fraction correct: {frac:0.3f}")
```

```
Fraction correct: 0.996
```

QDA, unregularized LDA, and regularized LDA have the same fraction correct in my experiments. They outrank the prototype classifier.

Between unregularized LDA and regularized LDA, I **prefer regularized LDA**. This is because both unregularized LDA and the prototype classifier are special cases of regularized LDA, and will be selected by cross-validation if either of them are in fact better.

Between regularized LDA and QDA, I would **prefer QDA in most cases** as it is parameter-free model and so doesn't require a cross-validation step.

Still, there are situations I might prefer an inferior classifier. Like QDA, unregularized LDA and prototype classifiers don't require cross-validation. With too few data, the cross-validation in regularized LDA could result in overfitting, or selecting a model that doesn't generalize well to new examples.

I might also use a prototype or LDA classifier if I have certain prior beliefs about how the dimensions in my dataset relate to one another semantically. If all dimensions are isotropic in their units (as in the case of pixel data or readings from a bank of identical sensors), I may choose a prototype classifier. If I expect (or measure) that the classes in my data have identical covariance, this may bias me toward unregularized LDA.

In [ ]: