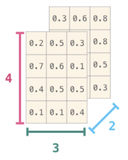
```
np.random.random((4,3,2))
```



A supervised machine learning "recipe"
• Step 1: Get labeled data: $(\mathbf{x}i, yi), i = 1,2,\cdots,N$.
• Step 2: Choose a candidate model f: $\hat{y} = f(x)$.
• Step 3: Select a loss function.
• Step 4: Find the model parameter values that minimize the loss function (training).
• Step 5: Use trained model to predict y^* for new samples not used in training (inference).
• Step 6: Evaluate how well your model generalizes.

```
1 g = np.linspace(0., 1, num=6)
2 print(g)

[0.  0.2 0.4 0.6 0.8 1. ]
```

Sometimes, we may want to construct an array from existing arrays by "stacking" the existing arrays, either vertically or horizontally. We can use `vstack()` (or `row_stack`) and `hstack()` (or `column_stack`), respectively.

```
1 a = np.array([1, 2, 3])
2 b = np.array([4, 5, 6])
3 np.vstack((a,b))

array([[1, 2, 3],
       [4, 5, 6]])
```

+ Code + Text

```
1 a = np.array([[7], [8], [9]])
2 b = np.array([[4], [5], [6]])
3 np.hstack((a,b))

array([[7, 4],
       [8, 5],
       [9, 6]])
```

Week 1 - data analysis

What problems are "good" for ML, overall? Problems that may not be well suited to ML

• There is an accurate and simple algorithm that will produce the desired output. • There is no "good" data available to train the model.

• The model can be "tricked", with potentially severe consequences.
• Need to audit or explain the output.

Problems that are often good candidates for ML

• There is "good" data available to train the model
• The thing we want to predict is measurable and observable
• Human expertise does not exist or is insufficient
• Humans cannot easily explain their expertise
• We will get more data during operation + can improve with experience

Data leakage

In machine learning, we train models on a training set of data, then evaluate their performance on a set of data that was not used in training.

Sometimes, information from the training set can "leak" into the evaluation - this is called data leakage.

Or, information from the target variable (which should not be available during inference) leaks into the feature data.

Some types of data leakage

• Learning from adjacent temporal data
• Learning from duplicate data
• Learning from features that are not available at prediction time (e.g. data from the future)

• Learning from a feature that is a proxy for target variable, but that doesn't generalize

Regression metrics Many numbers:

• Correlation coefficient R (r_{xy})

• Slope coefficient w_1 and intercept w_0 • RSS, MSE, R2

Week 2 Labs

The optimal least squares values for the vector w are

$$w^* = (A^T A)^{-1} A^T \mathbf{y}$$

Note that the least-squares solutions are the solutions of the matrix equation

$$A^T A\mathbf{w} = A^T \mathbf{y}$$

Compare this to the known solution:

```
1 np.linalg.inv( A.T.dot(A) ).dot(A.T.dot(y))

array([ 5.58391608,  0.77972028, -1.6993007 ])
```

```
                Note: other ways to do the same thing...
# first, add a ones column to design matrix
x_tilde = np.hstack((np.ones((n_samples, 1)), x_train))
# using matrix operations to find w = (X^T X)^(-1) X^T y
print( (np.linalg.inv((x_tilde.T @ x_tilde)) @ x_tilde.T @ y_train) )

# using solve on normal equations: X^T X w = X^T y
# solve only works on matrix that is square and of full-rank
# see https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html
print( np.linalg.solve(x_tilde.T @ x_tilde, x_tilde.T @ y_train) )

# using the lstsq solver
# problem may be under-, well-, or over-determined
# see https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html
            print( np.linalg.lstsq(x_tilde,y_train,rcond=0)[0] ) xt
```

Week 2 Linear Regression

Example: Intro ML grades (2)

Figure 5: Predicting students' grades in Intro ML using regression on previous coursework.

To some extent, a student's average grades on previous coursework "explains" their grade in Intro ML.

- The predicted value for each student, $\hat{y}_i$, is along the diagonal line. Draw a vertical line from each student's point (y_i) to the corresponding point on the line ($\hat{y}_i$). This is the residual $e = y - \hat{y}$.
- Some students fall right on the line - these are examples that are explained "well" by the model.
- Some students are far from the line. The magnitude of the *residual* is greater for these examples.
- The difference between the "true" value y and the predicted value $\hat{y}$ may be due to all kinds of differences between the "well-explained example" and the "not-well-explained-example" - not everything about Intro ML course grade can be explained by performance in previous coursework! This is what the residual captures.

Interpreting the linear regression: If slope w_1 is 0.8 points in Intro ML per point average in previous coursework, we can say that

- a 1-point increase in score on previous coursework is, on average, associated with a 0.8 point increase in Intro ML course grade.

What can we say about possible explanations? We can't say much using this method - anything is possible:

- statistical fluke (we haven't done any test for significance)
- causal - students who did well in previous coursework are better prepared
- confounding variable - students who did well in previous coursework might have more time to study because they don't have any other jobs or obligations, and they are likely to do well in Intro ML for the same reason.

This method doesn't tell us *why* this association is observed, only that it is. (There are other methods in statistics for determining whether it is a statistical fluke, or for determining whether it is a causal relationship.)

(Also note that the 0.8 point increase according to the regression model is only an *estimate* of the "true" relationship.)

Linear basis function model for regression

Standard linear model:

$$\hat{y}_i = w_0 + w_1 x_{i,1} + \cdots + w_d x_{i,d}$$

Linear basis function model:

$$\hat{y}_i = w_0\phi_0(\mathbf{x}_1) + \cdots + w_p\phi_p(\mathbf{x}_1)$$

Some notes:

- The 1s column we added to the design matrix is easily represented as a basis function ($\phi_0(\mathbf{x}) = 1$).
- There is not necessarily a one-to-one correspondence between the columns of X and the basis functions ($p \neq d$ is OK!). You can have more/fewer basis functions than columns of X.
- Each basis function can accept as input the entire vector $\mathbf{x}_i$.
- The model has $p + 1$ parameters.

Vector form of linear basis function model

The prediction of this model expressed in vector form is:

$$\hat{y}_i = \langle \boldsymbol{\phi}(\mathbf{x}_i), \mathbf{w} \rangle = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)$$

where

$$\boldsymbol{\phi}(\mathbf{x}_i) = [\phi_0(\mathbf{x}_i), \cdots, \phi_p(\mathbf{x}_i)], \mathbf{w} = [w_0, \cdots, w_p]$$

(The angle brackets denote a dot product.)

Important note: although the model can be non-linear in $\mathbf{x}$, it is still linear in the parameters $\mathbf{w}$ (note that $\mathbf{w}$ appears *outside* $\boldsymbol{\phi}(\cdot)$! That's what makes it a *linear model.*

Some basis functions have their own parameters that appear inside the basis function, i.e. we might have a model

$$\hat{y}_i = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i, \boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ are the parameters of the basis function. The model is *non-linear* in those parameters, and they need to be fixed before training.

Matrix form of linear basis function model

Given data $(\mathbf{x}_i, y_i), i = 1, \cdots, n$:

$$\Phi = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_p(\mathbf{x}_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_n) & \phi_1(\mathbf{x}_n) & \cdots & \phi_p(\mathbf{x}_n) \end{bmatrix}$$

and $\hat{\mathbf{y}} = \Phi\mathbf{w}$.

Interpreting R2: coefficient of determination (1)

$$R2 = 1 - \frac{MSE}{\sigma_y^2} = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \overline{y_i})^2}$$

For linear regression: What proportion of the variance in y is "explained" by our model?

- $R^2 \approx 1$ - model "explains" all the variance in y
- $R^2 \approx 0$ - model doesn't "explain" any of the variance in y

Interpreting R2: coefficient of determination (2)

Alternatively: what is the ratio of error of our model, to error of prediction by mean?

$$R2 = 1 - \frac{MSE}{\sigma_y^2} = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \overline{y_i})^2}$$

Gradient of MSE

$$L(\mathbf{w}) = \frac{1}{2}\|\mathbf{y} - \Phi\mathbf{w}\|^2$$

gives us the gradient

$$\nabla L(\mathbf{w}) = -\Phi^T(\mathbf{y} - \Phi\mathbf{w})$$

Solving for $\mathbf{w}$

$$\begin{aligned} \nabla L(\mathbf{w}) &= 0, \\ -\Phi^T(\mathbf{y} - \Phi\mathbf{w}) &= 0, \\ \Phi^T\Phi\mathbf{w} &= \Phi^T\mathbf{y}, \text{ or} \\ \mathbf{w} &= (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}. \end{aligned}$$

Solving a set of linear equations

If $\Phi^T\Phi$ is full rank (usually: if $n \geq d$), then a unique solution is given by

$$\mathbf{w}^* = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}$$

This expression:

$$\Phi^T\Phi\mathbf{w} = \Phi^T\mathbf{y}$$

represents a set of d equations in d unknowns, called the *normal equations*.

We can solve this as we would any set of linear equations (see supplementary notebook on computing regression coefficients by hand.)

Optimizing w - simple linear regression (1)

Given

$$MSE(w_0, w_1) = \frac{1}{n}\sum_{i=1}^n [y_i - (w_0 + w_1 x_i)]^2$$

we take

$$\frac{\partial MSE}{\partial w_0} = 0, \frac{\partial MSE}{\partial w_1} = 0$$

Optimizing w - simple linear regression (2)

First, the intercept:

$$MSE(w_0, w_1) = \frac{1}{n}\sum_{i=1}^n [y_i - (w_0 + w_1 x_i)]^2$$

$$\frac{\partial MSE}{\partial w_0} = -\frac{2}{n}\sum_{i=1}^n [y_i - (w_0 + w_1 x_i)]$$

using chain rule, power rule.
(We can then drop the -2 constant factor when we set this expression equal to 0.)

Optimizing w - simple linear regression (3)

Set this equal to 0, "distribute" the sum, and we can see

$$\frac{1}{n}\sum_{i=1}^n [y_i - (w_0 + w_1 x_i)] = 0$$

$$\implies w_0^* = \bar{y} - w_1^* \bar{x}$$

where $\bar{x}, \bar{y}$ are the means of x, y.

Optimizing w - simple linear regression (4)

Now, the slope coefficient:

$$MSE(w_0, w_1) = \frac{1}{n}\sum_{i=1}^n [y_i - (w_0 + w_1 x_i)]^2$$

$$\frac{\partial MSE}{\partial w_1} = \frac{1}{n}\sum_{i=1}^n 2(y_i - w_0 - w_1 x_i)(-x_i)$$

11

Optimizing w - simple linear regression (5)

$$\implies -\frac{2}{n}\sum_{i=1}^n x_i(y_i - w_0 - w_1 x_i) = 0$$

Solve for w_1^*:

$$w_1^* = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

Note: some algebra is omitted here, but refer to the secondary notes for details.

Optimizing w - simple linear regression (6)

The slope coefficient is the ratio of *sample covariance* σ_{xy} to *sample variance* σ_x^2:

$$\frac{\sigma_{xy}}{\sigma_x^2}$$

where $\sigma_{xy} = \frac{1}{n}\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$ and $\sigma_x^2 = \frac{1}{n}\sum_{i=1}^n (x_i - \bar{x})^2$

Optimizing w - simple linear regression (7)

We can also express it as

$$\frac{r_{xy}\sigma_y}{\sigma_x}$$

where sample correlation coefficient $r_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$.

(Note: from Cauchy-Schwartz law, $|\sigma_{xy}| < \sigma_x\sigma_y$, we know $r_{xy} \in [-1, 1]$)

MSE for optimal simple linear regression

$$MSE(w_0^*, w_1^*) = \sigma_y^2 - \frac{\sigma_{xy}^2}{\sigma_x^2}$$

$$\implies \frac{MSE(w_0^*, w_1^*)}{\sigma_y^2} = 1 - \frac{\sigma_{xy}^2}{\sigma_x^2\sigma_y^2}$$

- the ratio on the left is the *fraction of unexplained variance*: of all the variance in y, how much is still "left" unexplained after our model explains some of it? (best case: 0)
- the ratio on the right is the *coefficient of determination*, R2 (best case: 1).

Ordinary least squares solution for multiple/linear basis function regression

Setup: L2 norm

Definition: L2 norm of a vector $\mathbf{x} = (x_1, \cdots, x_n)$:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$$

We will want to minimize the L2 norm of the residual.

Setup: Gradient vector

To minimize a multivariate function $f(\mathbf{x}) = f(x_1, \cdots, x_n)$, we find places where the **gradient** is zero, i.e. each entry must be zero:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

The gradient is the vector of partial derivatives.

MSE for multiple/LBF regression

Given a vector $\mathbf{y}$ and matrix Φ (with d columns, n rows),

$$L(\mathbf{w}) = \frac{1}{2}\|\mathbf{y} - \Phi\mathbf{w}\|^2$$

where the norm above is the L2 norm.
(we defined it with a $\frac{1}{2}$ constant factor for convenience.)

```
def linear_basis(x):
  return np.hstack([np.ones(x.shape),x])

def polynomial_basis(x, d):

  return np.hstack([x**i for i in range(d)])

s = 0.1

def radial_basis(x, mu_list):

  return np.hstack([ np.exp(-1*(x-mu)**2/s**2) for mu in mu_list])

s = 0.05
```

```
def sigmoid_basis(x, mu_list):

  return np.hstack([((1+np.exp((-x-mu)/s))**-1) for mu in mu_list])

def fourier_basis(x, d):

  sins = np.hstack([np.sin(np.pi*i*x) for i in range(1,d+1)])

  coss = np.hstack([np.cos(np.pi*i*x) for i in range(1,d+1)])

  return np.hstack([sins, coss])
```

Basis functions

Week 3 GD Bias Variance

We had

$$\mathbf{w}^* = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}$$

where Φ is an $n \times d$ matrix. If $n \geq d$ then it is (usually) full rank and a unique solution exists.

How long does it take to compute?

Runtime of a "naive" solution using "standard" matrix multiplication:

- $O(d^2 n)$ to multiply $\Phi^T\Phi$
- $O(dn)$ to multiply $\Phi^T y$
- $O(d^3)$ to compute the inverse of $\Phi^T\Phi$

Since n is generally much larger than d, the first term dominates and the runtime is $O(d^2 n)$.

Example: gradient descent for linear regression (1)

With a mean squared error loss function

$$\begin{aligned} L(w, X, y) &= \frac{1}{n}\sum_{i=1}^n (y_i - (w, x_i))^2 \\ &= \frac{1}{n}\|y - Xw\|^2 \end{aligned}$$

Example: gradient descent for linear regression (2)

we will compute the weights at each step as

$$\begin{aligned} w^{t+1} &= w^t + \frac{\alpha^t}{n}\sum_{i=1}^n (y_i - (w^t, x_i))x_i \\ &= w^t + \frac{\alpha^t}{n}X^T(y - Xw^t) \end{aligned}$$

(dropping the constant 2 factor)

To update w, must compute n loss functions and gradients - each iteration is $O(nd)$. We need multiple iterations, but in many cases it's more efficient than the previous approach.

However, if n is large, it may still be expensive!

GD notebook

```
def gd_step(w, X, y, lr):
  # use current parameters to get y_hat
  y_hat = np.dot(X,w)
  error = y_hat-y
  # compute gradient for this y_hat
  grad = np.matmul(X.T, error)
  # update weights
  w_new = w - (lr/X.shape[0])*grad
  # we don't have to actually compute MSE
  # but I want to, for visualization
  mse = np.mean(error**2, axis=0)
  return (w_new, mse, grad)
```

```
def sgd_step(w, X, y, lr, n):
  idx_sample = np.random.choice(X.shape[0],
       n, replace=True)
  X_sample = X[idx_sample, :]
  y_sample = y[idx_sample]
  # use current parameters to get y_hat
  y_hat = np.dot(X_sample,w)
  error = y_hat-y_sample
  # compute gradient for this y_hat
  grad = np.matmul(X_sample.T, error)
  # update weights
  w_new = w - (lr/n)*grad
  # we don't have to actually compute MSE
  # but I want to, for visualization
  # note: MSE is computed on entire data, not sample
  mse = np.mean((y-np.dot(X, w))**2, axis=0)
  return (w_new, mse, grad)
```

```
def gd_step_momentum(w, X, y, lr, eta, v):
  # use current parameters to get y_hat, error
  y_hat = np.dot(X,w)
  error = y_hat-y

  # compute gradient and velocity
  grad = np.matmul(X.T, error)
  v_new = eta*v - (lr/X.shape[0])*grad
  # update weights
  w_new = w - (lr/X.shape[0])*grad + eta*v_new
  # we don't have to actually compute MSE
  # but I want to, for visualization
  mse = np.mean(error**2, axis=0)
  return (w_new, mse, grad, v_new)

itr = 100
lr = 0.1
eta = 0.9
w_init = [w_true[0], 3, 7]
w_steps = np.zeros((itr, len(w_init)))
mse_steps = np.zeros(itr)
grad_steps = np.zeros((itr, len(w_init)))
v_steps = np.zeros((itr, len(w_init)))
w_star = w_init
velocity = np.zeros(len(w_init))

for i in range(itr):
  w_star, mse, gradient, velocity =
       gd_step_momentum(w_star, X, y, lr, eta, velocity)
  w_steps[i] = w_star
  mse_steps[i] = mse
  grad_steps[i] = gradient
  v_steps[i] = velocity
```

Standard ("batch") gradient descent

For each step t along the error curve:

$$\begin{aligned} \mathbf{w}^{t+1} &= \mathbf{w}^t - \alpha\nabla L(\mathbf{w}^t) \\ &= \mathbf{w}^t - \frac{\alpha}{n}\sum_{i=1}^n \nabla L_i(\mathbf{w}^t, \mathbf{x}_i, y_i) \end{aligned}$$

Repeat until stopping criterion is met.

GD

Each iteration is now only $O(d)$, but we may need more iterations than for gradient descent.

However, in many cases we still come out ahead (especially if n is large!).

Variables:
Batch size
Learning rate

$$0 + \frac{p}{n}\sigma_\epsilon^2 + \sigma_\epsilon^2$$

squared bias + variance + irreducible error

$$Var = (p\sigma_\epsilon^2)/n$$

These two plots together show how well the variance of y is "explained" by the linear regression model:

- The total variance of y is shown in the first plot, where each vertical line is
$$y_i - \bar{y}$$
- The unexplained variance of y is shown in the second plot, where each vertical line is the error of the model.
$$y_i - \hat{y}_i$$

In this example, all of the variance of y is "explained" by the linear regression.

Irreducible error

Even if we learn exactly the right function, $f(x, w) = E(x)$, and we also learn the correct parameters, but still have error in our estimate of y because of the stochastic "noise" in the data. We cannot "learn" ϵ, it is unpredictable.

Decomposition of test MSE

We know that the expected MSE on a new, unseen test point, can be decomposed into those three factors. Let us denote $E[f(x, w)]$ as $\bar{f}(x, w)$, then

$$\begin{aligned} \text{Test MSE} &= E[(y - \hat{y})^2] \\ &= E[(t(x) + \epsilon - f(x, w))^2] \\ &= E(\epsilon)^2 + E[(t(x) - f(x, w))^2] \\ &= E(\epsilon)^2 + E[(t(x) - \bar{f}(x, w))^2 + E[f(x, w) - \hat{f}(x, w)]^2] \\ &= \sigma^2 + \text{Bias}^2 + \text{Var}(f(x, w)) \end{aligned}$$

where the expectation is over the draw of the training set and the noise.

- The σ^2 term captures the random noise in the data, which cannot be "learned"
- The $E[(t(x) - \bar{f}(x, w))^2]$ term captures any systematic difference between the function learned by our model and the true function, as will occur when the model cannot express the true function.
- The $E[(f(x, w) - \hat{f}(x, w))^2]$ term captures the difference between the coefficient estimate due to a particular random training set, and the average coefficient estimate - in other words, the variance of the prediction over many fitted instances.

Summary of results for linear regression

Suppose the model class is linear with n samples and p parameters.

Result 1: Uniqueness of coefficient estimate

When $n < p$, the least squares estimate of the coefficients is not unique.

Result 2: Bias of estimate of target variable

When $n \geq p$ and the least squares estimate of the coefficients is unique, *and* there is no under-modeling, then the estimate of the target variable is unbiased.

Result 3: Variance of estimate of target variable

When $n \geq p$, the least squares estimate of the coefficients is unique, there is no under-modeling, *and* the test point is drawn from the same distribution as the training data, then the variance of the estimate of the target variable increases linearly with the number of parameters and inversely with the number of samples used for training:

$$Var = \frac{p}{n}\sigma_\epsilon^2$$

Result 4: Overall prediction error

The overall expected in-sample prediction error for the ordinary least squares linear regression is

$$0 + \frac{p}{n}\sigma_\epsilon^2 + \sigma_\epsilon^2$$

where the three terms represent the squared bias, the variance, and the irreducible error.

Function to generate linear reg data

```
def generate_linear_regression_data(n=100, d=1, coef=[5], intercept=1, sigma=0,
uninformative=0, collinear=0, collinear_sigma = 0):
  x = np.random.uniform(-1,1,size=(n,d-uninformative-collinear))
  y = (np.dot(x, coef) + intercept).squeeze() + sigma * np.random.randn(n)
  if collinear:
    # select random columns from x
    collinear_cols = np.random.choice(d-uninformative-collinear,
size=collinear, replace=True)
    # value of new collinear columns
    # (adds noise so new columns are not exactly collinear)
    collinear_features =  x[:, collinear_cols] +
collinear_sigma*np.random.normal(size=(n,collinear))
    x = np.column_stack((x, collinear_features))
  if uninformative:
    x = np.column_stack((x, sigma*np.random.randn(n,uninformative)))

  return x, y
```

Week 4 Model Selection

Why use a subset of features?
• High risk of overfitting if you use all features!
• For linear regression, there's a unique OLS solution only if $n \geq d$
• For linear regression, when $N \geq p$, variance increases linearly with number of parameters, inversely with number of samples.

Problems with hold-out validation
• Fitted model (and test error!) varies a lot depending on samples selected for training and validation.
• Fewer samples available for estimating parameters.
• Especially bad for problems with small number of samples.

Leave-p-out CV

- In each iteration, p validation points
- Remaining $n - p$ points are for training
- Repeat for *all* possible sets of p validation points

This is *not* like K-fold CV which uses non-overlapping validation sets (they are only the same for $p = 1$)!

Computation (leave-p-out CV)

$\binom{n}{p}$ iterations, in each:

- train on $n - p$ samples
- score on p samples

Usually, this is too expensive - but sometimes LOO CV can be a good match to the model (KNN).

Computation (K-fold CV)

K iterations, in each:

- train on $n - n/k$ samples
- score on n/k samples

K-fold CV - how to split?

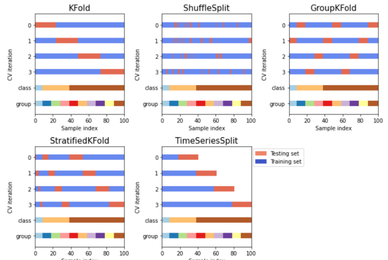


Figure 6: K-fold CV variations.

Selecting the right K-fold CV is very important for avoiding data leakage! (Also for training/test split.)

Refer to the function documentation for more examples.

One standard error rule

- Model selection that minimizes mean error often results in too-complex model
- One standard error rule: use simplest model where mean error is within one SE of the minimum mean error

One standard error rule - algorithm (1)

- Given data X, y
- Compute score $S_{p,i}$ for model p on fold i (of K)
- Compute average ($\bar{S}_p$), standard deviation σ_p, and standard error of scores:

$$SE_p = \frac{\sigma_p}{\sqrt{K-1}}$$

One standard error rule - algorithm (2)

"Best score" model selection: $p^* = \text{argmin}_p \bar{S}_p$

One SE rule for "lower is better" scoring metric: Compute target score: $S_t = \bar{S}_{p^*} + SE_{p^*}$

then select simplest model with score lower than target:

$$p^{*\text{-1SE}} = \min\{p | \bar{S}_p \leq S_t\}$$

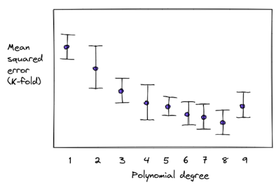


Figure 12: Model selection using one SE rule on MSE. The best scoring model is $d = 8$, but $d = 6$ is simplest model within one SE of the best scoring model, and so $d = 6$ would be selected according to the one-SE rule.

Note: this assumes you are using a "smaller is better" metric such as MSE. If you are using a "larger is better" metric, like R2, how would we change the algorithm?

One standard error rule - algorithm (3)

"Best score" model selection: $p^* = \text{argmax}_p \bar{S}_p$

One SE rule for "higher is better" scoring metric: Compute target score: $S_t = \bar{S}_{p^*} - SE_{p^*}$

then select simplest model with score higher than target:

$$p^{*\text{-1SE}} = \min\{p | \bar{S}_p \geq S_t\}$$

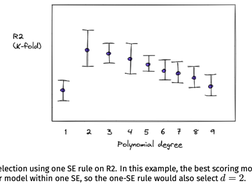


Figure 13: Model selection using one SE rule on R2. In this example, the best scoring model is $d = 2$, and there is no simpler model within one SE, so the one-SE rule would also select $d = 2$.

Regularization

Penalty for model complexity

With no bounds on complexity of model, we can always get a model with zero training error on finite training set - overfitting.

Basic idea: apply penalty in loss function to discourage more complex models

Regularization vs. standard LS

Least squares estimate:

$$\hat{w} = \underset{w}{\text{argmin}}\, MSE(w), \quad MSE(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Regularized estimate w/ **regularizing function** $\phi(w)$:

$$\hat{w} = \underset{w}{\text{argmin}}\, J(w), \quad J(w) = MSE(w) + \phi(w)$$

Common regularizers

Ridge regression (L2):

$$\phi(w) = \alpha \sum_{j=1}^{d} |w_j|^2$$

LASSO regression (L1):

$$\phi(w) = \alpha \sum_{j=1}^{d} |w_j|$$

Code to select highest mean R2

```
idx_max = np.argmax(r2_val.mean(axis=1))
d_max_r2 = dtest_list[idx_max]
d_max_r2
```

Common features: Ridge and LASSO
• Both penalize large wj
• Both have parameter α that controls level of regularization
• Intercept $w0$ not included in regularization sum (starts at 1!), this depends on mean of y and should not be constrained.

Differences: Ridge and LASSO (1)
Ridge (L2):
• minimizes $|w_j|2$,
• minimal penalty for small non-zero coefficients
• heavily penalizes large coefficients
• tends to make many "small" coefficients • Not for feature selection
Differences: LASSO (2)
LASSO (L1)
• minimizes $|wj|$
• tends to make coefficients either 0 or large (sparse!)
• does feature selection (setting wj to zero is equivalent to un-selecting feature)

Standardization (1)

Before learning a model with regularization, we typically *standardize* each feature and target to have zero mean, unit variance:

- $x_{i,j} \rightarrow \frac{x_{i,j} - \bar{x}_j}{s_{x_j}}$
- $y_i \rightarrow \frac{y_i - \bar{y}}{s_y}$

Standardization (2)

Why?

- Without scaling, regularization depends on data range
- With mean removal, no longer need w_0, so regularization term is just L1 or L2 norm of coefficient vector

L1 and L2 norm with standardization (1)

Assuming data standardized to zero mean, unit variance, the Ridge cost function is:

$$J(\mathbf{w}) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{d}|w_j|^2$$
$$= ||\mathbf{A}\mathbf{w} - \mathbf{y}||^2 + \alpha||\mathbf{w}||^2$$

L1 and L2 norm with standardization (2)

LASSO cost function ($||\mathbf{w}||_1$ is L1 norm):

$$J(\mathbf{w}) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{d}|w_j|$$
$$= ||\mathbf{A}\mathbf{w} - \mathbf{y}||^2 + \alpha||\mathbf{w}||_1$$

Ridge regularization

Why minimize $||\mathbf{w}||^2$?

Without regularization:

- large coefficients lead to high variance
- large positive and negative coefficients cancel each other for correlated features (remember attractiveness ratings in linear regression case study...)

Ridge term and derivative

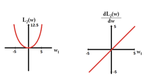


Figure 2: L2 term and its derivative for one parameter.

Ridge closed-form solution

$$J(\mathbf{w}) = ||\mathbf{A}\mathbf{w} - \mathbf{y}||^2 + \alpha||\mathbf{w}||^2$$

Taking derivative:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = 2\mathbf{A}^T(\mathbf{y} - \mathbf{A}\mathbf{w}) + 2\alpha\mathbf{w}$$

Setting it to zero, we find

$$\mathbf{w}_{ridge} = (\mathbf{A}^T\mathbf{A} + \alpha\mathbf{I})^{-1}\mathbf{A}^T\mathbf{y}$$

LASSO term and derivative

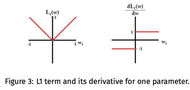


Figure 3: L1 term and its derivative for one parameter.

- No closed-form solution: derivative of $|w_j|$ is not continuous
- But there is a unique minimum, because cost function is convex, can solve iteratively

Effect of regularization level

Greater α, more complex model.

- Ridge: Greater α makes coefficients smaller.
- LASSO: Greater α makes more weights zero.

Selecting regularization level

How to select α? by CV!

- Outer loop: loop over CV folds
- Inner loop: loop over α

Delay Function

```
def create_dly_data(X,y,dly): """
Create delayed data
"""
n,p = X.shape
Xdly = np.zeros((n-dly,(dly+1)*p)) for i in range(dly+1):
        Xdly[:,i*p:(i+1)*p] = X[dly-i:n-i,:]
    ydly = y[dly:]
return Xdly, ydly
```

Week 4 Model SelectionNB

```
def generate_polynomial_regression_data(n=100,
      xrange=[-1,1], coefs=[1,0.5,0,2], sigma=0.5):
x = np.random.uniform(xrange[0], xrange[1], n)
y = np.polynomial.polynomial.polyval(x,coefs) +
      sigma * np.random.randn(n)
return x.reshape(-1,1), y

KFold using scikit vs for loop

kf = KFold(n_splits=nfold,shuffle=True)

for isplit, idx in enumerate(kf.split(x_train)):
  idx_tr, idx_val = idx

# this is same as

nfold = 5
# number of folds (you choose!)
nval = x_train.shape[0]
//nfold # number of validation samples per fold
idx_split = [i*nval for i in range(nfold)]
idx_list = np.arange(x_train.shape[0])
# list of training data indices
np.random.shuffle(idx_list)
# shuffle list of indices

for i, idx in enumerate(idx_split):
  idx_val = idx_list[idx:idx+nval]
  idx_tr = np.delete(idx_list, idx_val)
```

```
#Kfold to find optimal model order d
nfold = 5
# create a k-fold object
nfold = 5
kf = KFold(n_splits=nfold,shuffle=True)

# model orders to be tested
dmax = 10
dtest_list = np.arange(1,dmax+1)
nd = len(dtest_list)

mse_val = np.zeros((nd,nfold))
r2_val  = np.zeros((nd,nfold))

# create transformed features up to d_max
x_train_trans = x_train**np.arange(1,dmax+1)

# loop over the folds
# the first loop variable tells us how many out of nfold
# folds we have gone through
# the second loop variable tells us how to split the data
for isplit, idx in enumerate(kf.split(x_train)):

  # these are the indices for the training and validation indices
  # for this iteration of the k folds
  idx_tr, idx_val = idx

  x_train_kfold = x_train_trans[idx_tr]
  y_train_kfold = y_train[idx_tr]
  x_val_kfold = x_train_trans[idx_val]
  y_val_kfold = y_train[idx_val]

  for didx, dtest in enumerate(dtest_list):

    # get transformed features
    x_train_dtest =  x_train_kfold[:, :dtest]
    x_val_dtest   =  x_val_kfold[:, :dtest]

    # fit data
    reg_dtest = LinearRegression().fit(x_train_dtest, y_train_kfold)

    # measure MSE on validation data
    y_hat = reg_dtest.predict(x_val_dtest)
    mse_val[didx, isplit] = metrics.mean_squared_error(y_val_kfold, y_hat)
    r2_val[didx, isplit] = metrics.r2_score(y_val_kfold, y_hat)

    # measure MSE on training data
    y_hat_tr = reg_dtest.predict(x_train_dtest)
    mse_tr[didx, isplit] =
      metrics.mean_squared_error(y_train_kfold, y_hat_tr)
    r2_tr[didx, isplit] = metrics.r2_score(y_train_kfold, y_hat_tr)
r2_tr.mean(axis=1)
r2_val.mean(axis=1)
```

One SE Rule

We apply this rule as follows:
Find the MSE for each fold for each model candidate
For each model candidate, compute the mean and standard error of the MSE over the K folds.
We will compute the standard error as σMSE/$\sqrt{K-1}$
where σMSE is the standard deviation of the MSE over the K folds.
Find the model with the smallest mean MSE (across the K folds).
Compute the target as mean MSE + SE for this model.
Select the least complex model whose mean MSE is below the target.

This works for any metric that is a "lower is better" metric.
If you are using a "higher is better" metric, such as R2, for example, you would modify the last two steps:
Find the model with the largest mean R2 (across the K folds).
Compute the mean R2 - SE of R2 for this model. Call this quantity the target.
Select the least complex model whose mean R2 is above the target.

```
# for mse
idx_min = np.argmin(mse_val.mean(axis=1))
target = mse_val[idx_min,:].mean() + mse_val[idx_min,:].std()/np.sqrt(nfold-1)
# np.where returns indices of values where condition is satisfied
idx_one_se = np.where(mse_val.mean(axis=1) <= target)
d_one_se = np.min(dtest_list[idx_one_se])
d_one_se

# for r2
idx_max = np.argmax(r2_val.mean(axis=1))
target_r2 = r2_val[idx_max,:].mean() - r2_val[idx_max,:].std()/np.sqrt(nfold-1)
# np.where returns indices of values where condition is satisfied
idx_one_se_r2 = np.where(r2_val.mean(axis=1) >= target_r2)
d_one_se_r2 = np.min(dtest_list[idx_one_se_r2])
d_one_se_r2
```

L2 increases bias, decrease variance
L1 increases bias, decrease variance

Kfold on covid dataset

```
df_tr = df[(df.date <= '2020-05-01') & (df.date > '2020-03-01')]
df_ts = df[(df.date < '2020-06-30') & (df.date >= '2020-05-01')]
# model orders to be tested
dmax = 4
dtest_list = np.arange(1,dmax+1)
nd = len(dtest_list)

idxval = [10, 20, 30, 40, 50]
nfold = len(idxval)

mse_val = np.zeros((nd,nfold))
r2_val  = np.zeros((nd,nfold))
mse_tr  = np.zeros((nd,nfold))
r2_tr   = np.zeros((nd,nfold))

# loop over the folds
# the first loop variable tells us how many out of nfold
# folds we have gone through
# the second loop variable tells us how to split the data
for isplit, idx in enumerate(idxval):

  x_train_kfold = df_tr_poly[:idx]
  y_train_kfold = df_tr.deathIncrease.values[:idx]
  x_val_kfold = df_tr_poly[idx:idx+10]
  y_val_kfold = df_tr.deathIncrease.values[idx:idx+10]

  for didx, dtest in enumerate(dtest_list):

    # get transformed features
    x_train_dtest =  x_train_kfold[:, :dtest]
    x_val_dtest   =  x_val_kfold[:, :dtest]

    # fit data
    reg_dtest = LinearRegression().fit(x_train_dtest, y_train_kfold)

    # measure MSE on validation data
    y_hat = reg_dtest.predict(x_val_dtest)
    mse_val[didx, isplit] =
      metrics.mean_squared_error(y_val_kfold, y_hat)
    r2_val[didx, isplit] = metrics.r2_score(y_val_kfold, y_hat)

    # measure MSE on training data
    y_hat_tr = reg_dtest.predict(x_train_dtest)
    mse_tr[didx, isplit] =
      metrics.mean_squared_error(y_train_kfold, y_hat_tr)
    r2_tr[didx, isplit] = metrics.r2_score(y_train_kfold, y_hat_tr)
r2_tr.mean(axis=1)
r2_val.mean(axis=1)
```

Collinearity and Multicollinearity

Then, we'll compute the matrix of correlation coefficients for the five features.
An element of this matrix that is large (close to 1) indicates a pair of highly correlated variables.

```
np.corrcoef(x, rowvar=False)
```

An aside - our function creates new columns that are each a linear function of one other column.
In this case, we can see the collinearity by looking at the correlation coefficient, as we have done here.
More generally, a new column may be a linear combination of multiple columns.
This is known as multicollinearity and it wouldn't necessarily be obvious from the correlation coefficient or from the pairplot.
To identify multicollinearity, we should check the Variance Inflation Factor (VIF).
A VIF of 1 means no collinearity; a large VIF indicates multicollinearity.

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = [variance_inflation_factor(x, i) for i in range(x.shape[1])]
vif
```

Bias vs Variance L1 L2

As α increases the weight of the L2 penalty term, the variance of the model decreases, but the bias increases.
The optimal α for a given problem will depend on whether bias outweighs variance in the MSE breakdown.
In a model with high bias, regularization will make the overall MSE worse. In a model with high variance, regularization will increase bias but will still improve MSE.

How should we choose α , if it increases bias but decreases variance?
The choice of α is essentially a model selection problem.
We can use cross validation to select the α for which the validation MSE is the lowest.

LASSO reduces variance, but introduces some extra bias.
Because the coefficient values are pulled away from their "true" values, towards zero, the mean prediction (red line) is different from the true function (black line).

H4.4 - Fourier basis

```
#grade (write your code in this cell and DO NOT DELETE THIS LINE)
fft_tr_raw = np.array([[(np.cos((2*np.pi*Xtr)/t)),(np.sin((2*np.pi*Xtr)/t))] for t in tlist])
fft_tr_final = np.array([a for tup in fft_tr_raw for a in tup]).T

fft_ts_raw = np.array([[(np.cos((2*np.pi*Xts)/t)),(np.sin((2*np.pi*Xts)/t))] for t in tlist])
fft_ts_final = np.array([a for tup in fft_ts_raw for a in tup]).T


Xtr_trans = np.hstack((Xtr.reshape((-1,1))**np.arange(2), fft_tr_final))
Xts_trans = np.hstack((Xts.reshape((-1,1))**np.arange(2), fft_ts_final))
```

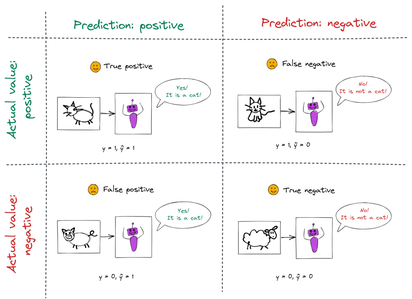
H4.4 Kfold

```
#grade
tlen = len(tlist)
nfold = 5
#  Create a k-fold object
kf = KFold(n_splits=nfold,shuffle=False)
msesq = np.zeros((tlen+1,nfold))

# Loop over the folds
for i, idx_split in enumerate(kf.split(Xtr_trans)):

    # Get the training and validation data in the split
    idx_tr, idx_val = idx_split
    Xtr_t0 = Xtr_trans[:,:2]
    X_tr0 = Xtr_t0[idx_tr]
    y_tr0 = ytr[idx_tr]
    X_val0 = Xtr_t0[idx_val]
    y_val0 = ytr[idx_val]
    lr_kfold = LinearRegression().fit(X_tr0, y_tr0)
    yhat_kfold0 = lr_kfold.predict(X_val0)
    msesq[0,i] = mean_squared_error(y_val0, yhat_kfold0)

    for it, t in enumerate(tlist):
        Xtr_t = Xtr_trans[:,:2+2*(it+1)]
        X_tr = Xtr_t[idx_tr]
        y_tr = ytr[idx_tr]
        X_val = Xtr_t[idx_val]
        y_val = ytr[idx_val]

        # Fit linear regression on training data
        lr_kfold = LinearRegression().fit(X_tr, y_tr)
        yhat_kfold = lr_kfold.predict(X_val)
        msesq[1+it,i] = mean_squared_error(y_val, yhat_kfold)
```

L4 Kfold code

```
  # TODO: Use K-fold CV to select dly
  # Number of folds
  nfold = 10
  kf = KFold(n_splits=nfold,shuffle=False)
  # Initialize a matrix Rsq to hold values of the R^2
  #across the model orders and folds.
  Rsq = np.zeros((nd,nfold))
  nt, nneuron = Xred.shape
  # Loop over the folds
  for i, idx_split in enumerate(kf.split(Xdly)):
   # Get the training and validation data in the split
      idx_tr, idx_val = idx_split
       for it, dtest in enumerate(dtest_list):
            # DO NOT call create_dly_data again
            # just select the appropriate subset of columns of Xdly
            # X_dtest = Xdly with the columns
            #corresponding to only the `dtest+1` most recent times.
            X_dtest = Xdly[:,:(dtest+1)*nneuron]
            # Split the data (X_dtest,ydly) into training and
            # validation # using idx_tr and idx_val
            Xtr = X_dtest[idx_tr]
            ytr = ydly[idx_tr]
             Xval = X_dtest[idx_val]
            yval = ydly[idx_val]
            # Fit linear regression on training data
            lr_kfold = LinearRegression().fit(Xtr, ytr)
            yhat_kfold = lr_kfold.predict(Xval)
            # Measure the R2 on validation data and store
            # in the matrix Rsq r2_kfold = r2_score(yval, yhat_kfold)
            Rsq[it,i] = r2_kfold
```

Week 5 Logistic Regression



Metrics related to error

The most basic classifier performance metric is **accuracy**, defined as

$$\frac{TP + TN}{TP + FP + TN + FN} = \frac{TP + TN}{P + N}$$

i.e., the portion of samples classified correctly.

However, accuracy is not always a useful metric. For example, imagine you are training a model to classify credit card transactions as fraudulent (1) or not fraudulent (0), but only 1% of transactions are fraudulent. A very basic classifier that always outputs 0 will have 99% accuracy! It is clear that accuracy is not a very useful metric here.

For a data set with highly imbalanced classes ($P >> N$ or $P << N$), **balanced accuracy** is often a more appropriate metric:

$$\frac{1}{2}\left(\frac{TP}{P} + \frac{TN}{N}\right)$$

Balanced accuracy gives the proportion of correct predictions in each class, averaged across classes.

In addition to the overall accuracy, a number of other metrics are used in various contexts. These are defined in terms of the four basic numbers described above: TP, FN, FP, TN.

The number of *true positive* (TP) outcomes, *true negative* (TN) outcomes, false positive (FP) outcomes, and false negative (FN) outcomes, are often presented together in a **confusion matrix**:

Actual ↓ Pred. →	**1**	**0**
1	TP	FN
0	FP	TN

We may also define two more quantities:

- The number of actual positive values (when $y = 1$) **P** = TP+FN, is the sum of the "actual positive" cells.
- The number of actual negative values (when $y = 0$) **N** = FP+TN is the sum of the "actual negative" cells.

The **total population**, P + N, is the total number of samples.

- **True Positive Rate** (TPR) also called *recall* or *sensitivity*:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} = P(\hat{y} = 1 | y = 1)$$

- **True Negative Rate** (TNR) also called *specificity*:

$$TNR = \frac{TN}{N} = \frac{TN}{FP + TN} = P(\hat{y} = 0 | y = 0)$$

- **Positive Predictive Value** (PPV) also called *precision*:

$$PPV = \frac{TP}{TP + FP} = P(y = 1 | \hat{y} = 1)$$

- **Negative Predictive Value** (NPV):

$$NPV = \frac{TN}{TN + FN} = P(y = 0 | \hat{y} = 0)$$

- **False Positive Rate** (FPR):

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} = 1 - TNR = P(\hat{y} = 1 | y = 0)$$

- **False Discovery Rate** (FDR):

$$FDR = \frac{FP}{FP + TP} = 1 - PPV = P(y = 0 | \hat{y} = 1)$$

- **False Negative Rate** (FNR):

$$FNR = \frac{FN}{FN + TP} = 1 - TPR = P(\hat{y} = 0 | y = 1)$$

- **False Omission Rate** (FOR):

$$FOR = \frac{FN}{FN + TN} = 1 - NPV = P(y = 1 | \hat{y} = 0)$$

These metrics are illustrated in the following table:

	Actual – Positive	Actual – Negative
Predicted – Positive	**True Positive (TP)** PPV = $\frac{TP}{TP+FP}$ TPR = $\frac{TP}{TP+FN}$	**False Positive (FP)** FDR = $\frac{FP}{TP+FP}$ FPR = $\frac{FP}{TP+TN}$
Predicted – Negative	**False Negative (FN)** FOR = $\frac{FN}{FN+TN}$ FNR = $\frac{FN}{TP+FN}$	**True Negative (TN)** NPV = $\frac{TN}{FN+TN}$ TNR = $\frac{TN}{TN+FP}$

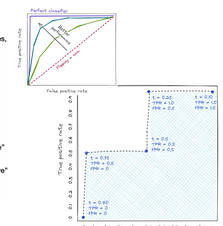
Selected classifier metrics.

Another metric, known as F1 score, combines precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) in one metric:

$$F_1 = 2\left(\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}\right)$$

Fairness metrics

- **Fairness through unawareness** is not a measure of fairness, but describes a situation in which features related to group membership are not used in classification. (This doesn't necessarily mean that the classifier produces fair outcomes! For example, if the classifier is trained on data that reflects an underlying bias in society, the classifier will be biased even if it is not trained on features related to group membership.)



TPR vs FPR aka ROC curve

Logistic function for binary classification

Let $z = w_0 + \sum_{j=1}^{d} w_d x_d$, then

$$P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-z}}, \quad P(y = 0|\mathbf{x}) = \frac{e^{-z}}{1 + e^{-z}}$$

(note: $P(y = 1) + P(y = 0) = 1$)

Logistic function with threshold

Choose a threshold t, then

$$\hat{y} = \begin{cases} 1, & P(y = 1|\mathbf{x}) \geq t \\ 0, & P(y = 1|\mathbf{x}) < t \end{cases}$$

Logistic classifier properties

• Class probabilities depend on distance from separating hyperplane
• Points far from separating hyperplane have probability ≈ 0 or ≈ 1
• When ||**w**|| is larger, class probabilities go towards extremes (0,1) more quickly
• Unlike linear regression, weights do not correspond to change in output associated with one-unit change in input.
• Sign of weight does tell us about relationship between a given feature and target variable.

Softmax function as a PMF

Acts like a probability mass function:

- $g_k(\mathbf{x}) \in [0, 1]$ for each k
- $\sum_{k=1}^{K} g_k(\mathbf{x}) = 1$
- larger input corresponds to larger "probability"

Softmax function for multi-class logistic regression (1)

Class probabilities are given by

$$P(y = k|\mathbf{x}) = \frac{e^{z_k}}{\sum_{\ell=1}^{K} e^{z_\ell}}$$

Softmax function for multi-class logistic regression (2)

When $z_k >> z_\ell$ for all $\ell \neq k$:

- $g_k(\mathbf{x}) \approx 1$
- $g_\ell(\mathbf{x}) \approx 0$ for all $\ell \neq k$

Assign highest probability to class k when z_k is largest.

Learning logistic model parameters

Weights $\mathbf{W}$ are the unknown **model parameters**:

$$\mathbf{z} = \mathbf{W}\mathbf{x}, \mathbf{W} \in R^{K \times d}$$

$$P(y = k|\mathbf{x}) = g_k(\mathbf{z}) = g_k(\mathbf{W}\mathbf{x})$$

Given training data $(\mathbf{x}_i, y_i), i = 1, \ldots, n$, we must learn $\mathbf{W}$.

Maximum likelihood estimation (1)

Let $P(\mathbf{y}|\mathbf{X}, \mathbf{W})$ be the probability of observing class labels $\mathbf{y} = (y_1, \ldots, y_n)^T$ given inputs $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ and weights $\mathbf{W}$.

The maximum likelihood estimate is

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\text{argmax}}\, P(\mathbf{y}|\mathbf{X}, \mathbf{W})$$

It is the estimate of parameters for which these observations are most likely.

Maximum likelihood estimation (2)

Assume outputs y_i are independent of one another,

$$P(\mathbf{y}|\mathbf{X}, \mathbf{W}) = \prod_{i=1}^{n} P(y_i|\mathbf{x}_i, \mathbf{W})$$

We take the log of both sides, because then the product turns into a sum...

Maximum likelihood estimation (3)

Define the **negative log likelihood**:

$$L(\mathbf{W}) = -\ln P(\mathbf{y}|\mathbf{X}, \mathbf{W})$$
$$= -\sum_{i=1}^{n} \ln P(y_i|\mathbf{x}_i, \mathbf{W})$$

(the term in the sum is also called cross-entropy)

Note that maximizing the likelihood is the same as minimizing the negative log likelihood.

Maximum likelihood estimation (4)

Now we can re-write max likelihood estimator with a loss function to minimize:

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\text{argmax}}\, P(\mathbf{y}|\mathbf{X}, \mathbf{W}) = \underset{\mathbf{W}}{\text{argmin}}\, L(\mathbf{W})$$

The next step will be to plug in our sigmoid function.

Binary cross-entropy loss (1)

For binary classification with class labels 0, 1:

$$\begin{aligned} \ln P(y_i|\mathbf{x}_i, \mathbf{w}) &= y_i \ln P(y_i = 1|\mathbf{x}_i, \mathbf{w}) + (1 - y_i) \ln P(y_i = 0|\mathbf{x}_i, \mathbf{w}) \\ &= y_i \ln \sigma(z_i) + (1 - y_i) \ln(1 - \sigma(z_i)) \\ &= y_i(\ln \sigma(z_i) - \ln \sigma(-z_i)) + \ln \sigma(-z_i) \\ &= y_i \ln \frac{\sigma(z_i)}{\sigma(-z_i)} + \ln \sigma(-z_i) \\ &= y_i \ln \frac{1 + e^{z_i}}{1 + e^{-z_i}} + \ln \sigma(-z_i) \\ &= y_i \ln \frac{e^{z_i}(e^{-z_i} + 1)}{1 + e^{-z_i}} + \ln \sigma(-z_i) \\ &= y_i z_i - \ln(1 + e^{z_i}) \end{aligned}$$

(Note: $\sigma(-z) = 1 - \sigma(z)$)

Binary cross-entropy loss (2)

Binary cross-entropy loss function (negative log likelihood):

$$\sum_{i=1}^{m} \ln(1 + e^{z_i}) - y_i z_i$$

Cross-entropy loss for multi-class classification (2)

Then,

$$\ln P(y_i|\mathbf{x}_i, \mathbf{W}) = \sum_{k=1}^{K} r_{ik} \ln P(y_i = k|\mathbf{x}_i, \mathbf{W})$$

Cross-entropy loss function is

$$\sum_{i=1}^{n} \left[\ln \left(\sum_{k} e^{z_{ik}} \right) - \sum_{k} z_{ik} r_{ik} \right]$$

Minimizing cross-entropy loss

To minimize, we would take the partial derivative:

$$\frac{\partial L(W)}{\partial W_{kj}} = 0$$

for all W_{kj}

But, there is no closed-form expression

- can only estimate weights via numerical optimization

(e.g. gradient descent)

- **Causal discrimination** says that two samples that are identical w.r.t all features except group membership, should have same classification.
- **Group fairness** (also called *statistical parity*) says that for groups a and b, the classifier should have equal probability of positive classification:

$$P(\hat{y} = 1|G = a) = P(\hat{y} = 1|G = b)$$

- **Conditional statistical parity** is a related metric, but now we are also controlling for factor F:

$$P(\hat{y} = 1|G = a, F = f) = P(\hat{y} = 1|G = b, F = f)$$

- **Balance for positive/negative class** is similar to *group fairness*, but it is for classifiers that produce soft output. It applies to every probability S produced by the classifier. This says that the expected value of probability assigned by the classifier should be the same for both groups -
- For **positive class balance**

$$E(S|y = 1, G = a) = E(S|y = 1, G = b)$$

- For **negative class balance**

$$E(S|y = 0, G = a) = E(S|y = 0, G = b)$$

- **Predictive parity** (also called *outcome test*) says that the groups should have equal PPV, i.e. the prediction should carry similar meaning (w.r.t. probability of positive outcome) for both groups:

$$P(y = 1|\hat{y} = 1, G = a) = P(y = 1|\hat{y} = 1, G = b)$$

- Predictive parity also implies equal FDR:

$$P(y = 0|\hat{y} = 1, G = a) = P(y = 0|\hat{y} = 1, G = b)$$

- **Calibration** (also called *test fairness*, *matching conditional frequencies*) is similar to *predictive parity*, but it is for classifiers that produce soft output. It applies to every probability S produced by the classifier:

$$P(y = 1|S = s, G = a) = P(y = 1|S = s, G = b)$$

- **Well-calibration** extends this definition to add that the probability of positive outcome should actually be s:

$$P(y = 1|S = s, G = a) = P(y = 1|S = s, G = b) = s$$

- **False positive error rate balance** (also called *predictive equality*) says that groups should have equal FPR:

$$P(\hat{y} = 1|y = 0, G = a) = P(\hat{y} = 1|y = 0, G = b)$$

- False positive error rate balance also implies equal TNR:

$$P(\hat{y} = 0|y = 0, G = a) = P(\hat{y} = 0|y = 0, G = b)$$

- **False negative error rate balance** (also called *equal opportunity*) says that groups should have equal FNR. This is equivalent to group fairness **only** if the prevalence of positive result is the same among both groups:

$$P(\hat{y} = 0|y = 1, G = a) = P(\hat{y} = 0|y = 1, G = b)$$

- False negative error rate balance also implies equal TPR:

$$P(\hat{y} = 1|y = 1, G = a) = P(\hat{y} = 1|y = 1, G = b)$$

- **Equalized odds** (also called *disparate mistreatment*) says that both groups should have equal TPR *and* FPR:

$$P(\hat{y} = 0|y = i, G = a) = P(\hat{y} = 0|y = i, G = b), i \in 0, 1$$

- Note that if the prevalence of the (actual) positive result is *different* between groups, then it is not possible to satisfy FP and FN error rate balance *and* predictive parity at the same time!
- **Conditional use accuracy equality** says that the groups have equal PPV *and* NPV:

$$P(y = 1|\hat{y} = 1, G = a) = P(y = 1|\hat{y} = 1, G = b)$$
$$P(y = 0|\hat{y} = 0, G = a) = P(y = 0|\hat{y} = 0, G = b)$$

- **Overall accuracy equality** says that the groups have equal overall accuracy

$$P(\hat{y} = y|G = a) = P((\hat{y} = y|G = b)$$

- **Treatment equality** says that the groups have equal ratio of FN to FP, $\frac{FN}{FP}$

Transformations, bias, variance

- Can use basis functions to map problem to transformed feature space (if "natural" decision boundary is non-linear)
- Variance increases with d and decreases with n
- Can add a regularization penalty to loss function

"Recipe" for logistic regression (binary classifier)

- Choose a **model**:

$$P(y=1|x,w) = \sigma\left(w_0 + \sum_{i=1}^{d} w_d x_d\right)$$

$$\hat{y} = \begin{cases} 1, & P(y=1|\mathbf{x}) \geq t \\ 0, & P(y=1|\mathbf{x}) < t \end{cases}$$

- Get **data** - for supervised learning, we need **labeled** examples: $(x_i, y_i), i = 1, 2, \cdots, n$
- Choose a **loss function** that will measure how well model fits data: binary cross-entropy

$$\sum_{i=1}^{n} \ln(1 + e^{z_i}) - y_i z_i$$

- Find model **parameters** that minimize loss: use numerical optimization to find weight vector w
- Use model to **predict** $\hat{y}$ for new, unlabeled samples.

"Recipe" for logistic regression (multi-class classifier)

- Choose a **model**: find probability of belonging to each class, then choose the class for which the probability is highest.

$$P(y=k|\mathbf{x}) = \frac{e^{z_k}}{\sum_{\ell=1}^{K} e^{z_\ell}} \text{ where } \mathbf{z} = \mathbf{W}\mathbf{x}$$

- Get **data** - for supervised learning, we need **labeled** examples: $(x_i, y_i), i = 1, 2, \cdots, n$
- Choose a **loss function** that will measure how well model fits data: cross-entropy

$$\sum_{i=1}^{n} \left[\ln\left(\sum_k e^{z_{ik}}\right) - \sum_k z_{ik} r_{ik} \right] \text{ where}$$

$$r_{ik} = \begin{cases} 1 & y_i = k \\ 0 & y_i \neq k \end{cases}$$

- Find model **parameters** that minimize loss: use numerical optimization to find weight vector w
- Use model to **predict** $\hat{y}$ for new, unlabeled samples.

Naive Bayes classifier

A quick look at a different type of model!

Probabilistic models (1)

For logistic regression, minimizing the cross-entropy loss finds the parameters for which

$P(\mathbf{y}|\mathbf{X}, \mathbf{W})$

is maximized.

Probabilistic models (2)

For linear regression, assuming normally distributed stochastic error, minimizing the **squared error** loss finds the parameters for which

$P(\mathbf{y}|\mathbf{X}, \mathbf{w})$

is maximized.

Surprise! We've been doing maximum likelihood estimation all along.

Probabilistic models (3)

ML models that try to

- get a good fit for $P(y|X)$: **discriminative** models.
- fit $P(X, y)$ or $P(X|y)P(y)$: **generative** models.

Linear regression and logistic regression are both considered discriminative models; they say "given that we have this data, what's the most likely label?" (e.g. learning a mapping from an input to a target variable).

Generative models try to learn "what does data for each class look like" and then apply Bayes rule.

Bayes rule

For a sample $\mathbf{x}_i$, y_k is label of class k:

$$P(y_k|\mathbf{x}_i) = \frac{P(\mathbf{x}_i|y_k)P(y_k)}{P(\mathbf{x}_i)}$$

- $P(y_k|\mathbf{x}_i)$: posterior probability. "What is the probability that this sample belongs to class k, given its observed feature values are $\mathbf{x}_i$?"
- $P(\mathbf{x}_i|y_k)$: conditional probability: "What is the probability of observing the feature values $\mathbf{x}_i$ in a sample, given that the sample belongs to class k?"
- $P(y_k)$: prior probability
- $P(\mathbf{x}_i)$: evidence

Generate data for logistic regression

```
# Generate training data
X = np.random.uniform(0, 1, size=(n_samples,2))
y = np.array(w_true[0]+w_true[1]*X[:,0]+w_true[2]*X[:,1] >= 0).astype(int)

# Add some noise?
X = X + sigma*np.random.randn(X.shape[0]*X.shape[1]).reshape(X.shape[0], X.shape[1])
```

Logistic regression training

```
def sigmoid(z):
  return 1/(1+np.exp(-z))
def cross_entropy_loss(w, X, y):
  p_pred = sigmoid(np.dot(X,w))
  loss_pos = y*np.log(p_pred)
  loss_neg = (1-y)*np.log(1-p_pred)
  return np.sum(-1*(loss_pos + loss_neg))
def gd_step(w, X, y, lr):
  p_pred = sigmoid(np.dot(X,w))
  gradient = np.dot(X.T, y - p_pred)
  w = w + lr * gradient
  l = cross_entropy_loss(w,X,y)
  return w, l, gradient
itr = 20000
lr = 0.005
tol = 0.01
w_init = np.random.randn(3)
print(w_init)
w_steps = np.zeros((itr, len(w_init)))
l_steps = np.zeros(itr)
grad_steps =  np.zeros((itr, len(w_init)))
stop = 0

w_star = w_init
for i in range(itr):
  w_star, loss, grad = gd_step(w_star, X_aug, y, lr)
  w_steps[i] = w_star
  l_steps[i] = loss
  grad_steps[i] = grad
  if np.linalg.norm(grad, ord=1) <= tol:
    print("Stopping gradient descent at iteration %d" % i)
    stop = i
    break
  stop = i
```

Logistic regression using sklearn

```
n_samples = 1000
sigma = 0
coefs=np.array([0.3, 1, -1.5, -2])
xrange=[-1,1]
def generate_polynomial_classifier_data(n=100, xrange=[-1,1], coefs=[1,0.5,0,2], sigma=0.5):
  x = np.random.uniform(xrange[0], xrange[1], size=(n, 2))
  ysep = np.polynomial.polynomial.polyval(x[:,0],coefs)
  y = (x[:,1]>ysep).astype(int)
  x[:,0] = x[:,0] + sigma * np.random.randn(n)
  x[:,1] = x[:,1] + sigma * np.random.randn(n)
  return x, y
clf = LogisticRegression(penalty='none',
                         tol=0.01, solver='saga')
clf.fit(X, y)
clf.score(X, y)
```

Fairness checks on COMPAS

First, let's compare the accuracy for the two groups:

```
[19]  1 (df['two_year_recid']==df['is_med_or_high_risk']).astype(int).groupby(df['race']).mean()

race
African-American    0.638258
Caucasian           0.669927
dtype: float64
```

It isn't exactly the same, but it's similar - within a few points. This is a type of fairness known as **overall accuracy equality**.

Next, let's see whether a defendant who is classified as medium/high risk has the same probability of recidivism for the two groups.

In other words, we will compute the PPV for each group:

$$PPV = \frac{TP}{TP + FP} = P(y = 1|\hat{y} = 1)$$

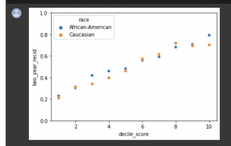
+ Code + Text

```
[20]  1 df[df['is_med_or_high_risk']==1]['two_year_recid'].groupby(df['race']).mean()

race
African-American    0.629715
Caucasian           0.591335
Name: two_year_recid, dtype: float64
```

Again, similar (within a few points). This is a type of fairness known as **predictive parity**.

We can extend this idea, to check whether a defendant with a given score has the same probability of recidivism for the two groups:

```
1 d = pd.DataFrame(df.groupby(['decile_score','race']).agg({'two_year_recid': 'mean'}))
2 d = d.reset_index()
3 im = sns.scatterplot(data=d, x='decile_score', y='two_year_recid', hue='race');
4 im.set(ylim=(0,1));
```



We can see that for both African-American and Caucasian defendants, for any given COMPAS score, recidivism rates are similar. This is a type of fairness known as **calibration**.

What can we do, as machine learning engineers, that might help?

Exploratory data analysis - look for possible underlying bias in the data.
Avoid using sensitive group as a feature (or a proxy for a sensitive group), but this doesn't necessarily help. (It didn't help in this example.)
Sometimes, using the sensitive group to explicitly add fairness might be better.
Make sure different groups are well represented in the data.
End users must be made aware of what the model output means -
 for example, judges should understand that a "high risk" label only means a 65% chance of reoffending.
Evaluate final model for bias with respect to sensitive groups.
May not be possible to satisfy multiple fairness metrics simultaneously, work with end users to decide which fairness metrics to prioritize,
 and to create a model that is fair w.r.t. that metric.

Why use saga solver?

works well when there is a large number of samples,
supports logistic regression with no regularization penalty, L1 penalty, L2 penalty,
 or ElasticNet (which uses both penalties),
and also supports multinomial regression with multiple classes, using the softmax function.

Notes on logistic regression

- When there is under-modeling, adding transformed features can reduce bias.
- However, adding features also increases variance.
- We can use cross validation to choose a model that fits the data well but also generalizes.
- We can add a regularization penalty to our loss function to reduce variance.

Confusion matrix from df

```
cm = pd.crosstab(df['is_med_or_high_risk'], df['two_year_recid'],
                 rownames=['Predicted'], colnames=['Actual'])
p = plt.figure(figsize=(5,5));
p = sns.heatmap(cm, annot=True, fmt="d", cbar=False)
```

Code for H5.10

```
from numpy import log as ln
from math import e
import numpy as np
def fun(x,neg=False):
  if neg:
    return ((e**-x)/(1+e**-x))
  return (1/(1+e**-x))
x = np.array([[1,2,2],
  [1,-3,2],
  [1,-1,0],
  [1,-2,2]])
y = np.array([0,1,0,1])
w = np.array([1.,-4.,-3.])
z = np.array(np.matmul(x,w))
print("z = ",z)
px = np.array([fun(z_i, False) for z_i in z])
print(px)
loss = -1*np.sum((y*ln(1/(1+e**-z)) + (1-y)*ln(e**-z/(1+e**-z))))
print("loss = ",loss)
alpha = 0.6
print("this ", np.sum(x*((y-(1/(1+e**-z))))[:,None], axis=0))
nw = w+ alpha*np.sum(x*((y-(1/(1+e**-z))))[:,None], axis=0)
print("new weights = ", nw)
nz = np.array(np.matmul(x,nw))
print("new z", nz)
nloss = -1*np.sum((y*ln(1/(1+e**-nz)) + (1-y)*ln(e**-nz/(1+e**-nz))))
print("new loss",nloss)
```

Code for H5.9

```
from numpy import log as ln
from math import e
import numpy as np

def get_stats(confusion_matrix):
  '''
  Assumption
        predicted
        1      0
  actual
  1     1      0   TP     FN
  0     1      0   FP     TN
  '''
  TP = confusion_matrix[0,0]
  TN = confusion_matrix[1,1]
  FP = confusion_matrix[1,0]
  FN = confusion_matrix[0,1]
  # TP = np.diag(confusion_matrix)
  # TN = np.sum(confusion_matrix) - (FP + FN + TP)


  # Sensitivity, hit rate, recall, or true positive rate
  TPR = TP/(TP+FN)
  # Specificity or true negative rate
  TNR = TN/(TN+FP)
  # Precision or positive predictive value
  PPV = TP/(TP+FP)
  # Negative predictive value
  NPV = TN/(TN+FN)
  # Fall out or false positive rate 1 - TNR
  FPR = FP/(FP+TN)
  # False negative rate
  FNR = FN/(TP+FN)
  # False discovery rate
  FDR = FP/(TP+FP)
  # False Ommision rate 1- NPV
  FOR = FN/(FN + TN)

  # Overall accuracy
  ACC = (TP+TN)/(TP+FP+FN+TN)

  # actual positivity rate
  # slightly off in calculations, unsure why
  APR = (TP+FN)/(TP+FP+FN+TN)

  # predicted positivity rate
  PPR = (TP+FP)/(TP+FP+FN+TN)
  return {'TPR': TPR, 'TNR': TNR, 'PPV': PPV, 'NPV': NPV,
'FPR': FPR, 'FNR': FNR, 'FDR': FDR, 'FOR': FOR, 'ACC': ACC,
'APR': APR, 'PPR': PPR}
a = np.array([[13470, 3270], [69, 329]])
b = np.array([[10049, 3578], [1643, 8692]])
c = a+b
# T to deal with opposite convention in hw !!!!!!
stats_list = [get_stats(a.T), get_stats(b.T), get_stats(c.T)]
for smap in stats_list:
  for key, value in smap.items():
    print(f"{key} = {value}")
  print()
```

https://www.wolframalpha.com/

https://scikit-learn.org/stable/modules/classes.html

https://numpy.org/doc/stable/reference/routines.html

https://pandas.pydata.org/docs/reference/index.html

$(x+y)^n = \sum^n{}_{k=0} {}^nC_k\, x^{n-k} y^k = \sum^n{}_{k=0} {}^nC_k\, x^k y^{n-k}$

Example: Let us expand $(x+3)^5$ using the binomial theorem. Here y = 3 and n = 5. Substituting and expanding, we get:

$(x+3)^5 = {}^5C_0\, x^5 3^0 + {}^5C_1\, x^{5-1} 3^1 + {}^5C_2\, x^{5-2} 3^2 + {}^5C_3\, x^{5-3} 3^3 + {}^5C_4\, x^{5-4} 3^4 + {}^5C_5\, x^{5-5} 3^5$

$= x^5 + 5\,x^4 . 3 + 10\,x^3 . 9 + 10\,x^2 . 27 + 5x . 81 + 3^5$

$= x^5 + 15\,x^4 + 90x^3 + 270\,x^2 + 405\,x + 243$