W1 - Intro

Rules-based, ML System

W2 - Linear regression

$f(x) = \log(x_1^{w_1}) + w_2 x_2^2 + w_3$

parameters linear, inputs non-linear

MSE (Mean squared error)

$L(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

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

RSS (residual sum of squares)

$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(e_i)^2$ convenient

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

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

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

$\Rightarrow \frac{MSE}{\sigma_y^2} = 1 - R2$

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

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

$y_i = \sum_{j=0}^{d} w_j e^{-jx_i}$, $\phi(x) = [\phi_0(x) \dots \phi_d(x)]$

$\hat{y} = \Phi w$, given $(x_1, y_1), \dots, (x_4, y_4)$

$$\Phi = \begin{bmatrix} 1 & e^{-x_1} & e^{-2x_1} \\ 1 & e^{-x_2} & e^{-2x_2} \\ 1 & e^{-x_3} & e^{-2x_3} \\ 1 & e^{-x_4} & e^{-2x_4} \end{bmatrix}$$

$y = C + Dx + \epsilon$, $\epsilon \sim N(0, \sigma^2)$

training data from $y = A + Bx + \epsilon$, $R2 = 1$ ($\sigma = 0$)

test data on $y = C + Dx + \epsilon$, $R2 \le 1$ ($\sigma = 0$)

W3 - Gradient descent, bias variance

$L(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \langle w, x_i \rangle)^2$

$w^{t+1} = w^t - \frac{\alpha}{n}\sum_{i=1}^{n}(-x_i)(y_i - \langle w^t, x_i \rangle)$

i	x_0	x_1	y	$-x_i(y_i - \hat{y}_i)$
0	1	-1.49	-0.999	[7.949, -15.81851]
1	1	1.348	-1.098	[-10.692, -18.60216]
2	1	3.212	19.117	[-38.107, -132.63548]

$w_0 = -3$, $w_1 = -5$, $\alpha = 0.1$ → [-40.87, -157.09515]

$\hat{y}_i = w_0 x_0 + x_i y_i = [6.95, -11.74, -19.06]$

$w_i^{t+1} = [-3, -5] - \frac{0.1}{3}[40.87, -157.09515]$

$= [-4.64, 0.23]$

Momentum: update includes a vector v, that accumulates gradient of past steps. Each update is a linear combination of the gradient and the previous updates. (Go faster if gradient keeps pointing in the same direction)

AdaGrad: "per-parameter learning rate". Track per-parameter of gradient to normalize parameter update step. Weights with large gradient have smaller learning rate, weights with small gradient have larger learning rate. Take smaller steps in steep directions, take bigger steps where the gradient is flat.

RMSProp: Leaky AdaGrad

Uses EWMA to emphasize recent gradient magnitudes

Adam: Adaptive Moment Estimation.

Uses ideas from momentum (first moment) and RMSProp (second moment). plus bias correction.

Total 1600 samples, in one epoch:

	full batch GD	mini-batch GD	Stochastic GD on one sample
how many samples to compute the gradient of loss for that sample	1600	1600	1600
how many times to update weights (iterations)	1	200	1600

$f(x_t, \hat{w}) \to f$; $t(x_t) \to t$

$E_\epsilon[(y_i - \hat{y}_i)^2] = (t - f)^2 + \sigma_\epsilon^2$

$E_D[(t - f)^2 + \sigma_\epsilon^2] = (t - E_D[f])^2 + (E_D[f^2] - E_D[f]^2) + \sigma_\epsilon^2$

(Bias)^2 ← variance over D → irreducible error

$Var = \frac{p}{n}\sigma_\epsilon^2$ (p: number of parameters, n: number of samples)

$y = A + Bx + \epsilon$, $\epsilon \sim N(0, \sigma^2)$

training data: $0 \le R2 \le 1$, $R2 = 1$ ($\sigma = 0$)

test data: $R2 \le 1$, $R2 = 1$ ($\sigma = 0$)

W4 - Model selection and regularization.

Hold-out validation

- Divide data into training, validation, test sets.
- Learn on training set for each candidate model.
- Measure error for all models on validation set.
- Select model that minimizes error on validation set.
- Evaluate that model on test set.

Problems: Fitted model (and test error) varies a lot depending on sample selected for training and validation

Fewer samples available for estimating parameters.

Especially bad for problems with small number of samples.

K-fold cross validation

- Divide data into K equal-sized parts (typically 5, 10)
- For each, evaluate model. K-1 parts training, last validation.
- Average K validation scores and choose based on avg.

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

K-fold CV - how to split?

- No structure — Shuffle split (avoid accidental patterns)
- Group structure — keeps members of each group in either training set, or validation set, but not both.
- Time series data - keeps validation data in the future, relative to training data

One standard error rule: use simplest model whose mean error is within on SE of the minimum one.

- Given data X, y, compute score $S_{p,i}$ for model p on fold i
- Compute avg ($\bar{S}_p$), standard deviation σ_p, and standard error of scores: $SE_p = \frac{\sigma_p}{\sqrt{k-1}}$

For MSE, $p^* = \text{argmin}_p \bar{S}_p$, $t = \bar{S}_{p^*} + SE_{p^*}$, $p^* = \min\{p | \bar{S}_p \le t\}$

For R2, $p^* = \text{argmax}_p \bar{S}_p$, $t = \bar{S}_{p^*} - SE_{p^*}$, $p^* = \min\{p | \bar{S}_p \ge t\}$

Ridge regression (L2): $\phi(w) = \alpha\sum_{j=1}^{d}|w_j|^2$

LASSO regression (L1): $\phi(w) = \alpha\sum_{j=1}^{d}|w_j|$

Both penalize large w_j; Both have parameter α that controls levels of regularization; Intercept w_0 not in regularization sum, it depends on mean of y.

Ridge (L2)

- minimizes $|w|^2$, minimal penalty for non-zero coefs.
- heavily penalizes large coefs
- tends to make many small coefs; Not for feature selection

LASSO (L1) (Sparse)

- minimizes $|w_j|$, tends to make coefs either 0 or larger
- does feature selection ($w_j \to 0 \Rightarrow$ un-selecting feature)

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

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

Greater α, less complex model. → bias↑, variance↓

Ridge → coefficients smaller; LASSO → more weights zero

W5 - Logistic regression and classification

	Prediction: positive	Prediction: negative
Actual value positive	True positive $y=1, \hat{y}=1$	False negative $y=1, \hat{y}=0$
Actual value negative	False positive $y=0, \hat{y}=1$	True negative $y=0, \hat{y}=0$

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

$P = TP + FN$, $N = FP + TN$ (actual)

	Actual - Positive	Actual - Negative
Predicted - Positive	$PPV = \frac{TP}{TP+FP}$ (precision)	$FDR = \frac{FP}{TP+FP}$
	$TPR = \frac{TP}{TP+FN}$ (recall, sensitivity)	$FPR = \frac{FP}{FP+TN}$
Predicted - Negative	$FOR = \frac{FN}{TN+FN}$	$NPV = \frac{TN}{TN+FN}$
	$FNR = \frac{FN}{TP+FN}$	$TNR = \frac{TN}{TN+FP}$ (specificity)

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

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

email: spam, not spam

threshold ↑ → FP ↓ → precision ↑

↳ TP ↓ or stay → recall ↓ or stay

$P(y=1|x_i) = f(x_i)$, $P(y_i=0|x_i) = 1 - f(x_i)$

logistic/sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$

$z = w_0 + \sum_{j=1}^{d} w_j x_j$, $P(y=1|x) = \frac{1}{1+e^{-z}}$, $P(y=0|x) = \frac{e^{-z}}{1+e^{-z}}$

Softmax function: $g_k(z) = \frac{e^{z_k}}{\sum_{k=0}^{K-1} e^{z_k}} = P(y=k|x)$

Maximum likelihood estimate

$\hat{w} = \underset{w}{\text{argmax}}\, P(y|X,w)$, $P(y|X,w) = \prod_{i=1}^{n} P(y_i|x_i,w)$

$P(y_i|x_i,w) = y_i P(y_i=1|x_i,w) + (1-y_i) P(y_i=0|x_i,w)$

Binary cross-entropy loss function:

$L(w) = -\sum_{i=1}^{n} [y_i \ln P(y_i=1|x_i,w) + (1-y_i) \ln P(y_i=0|x_i,w)]$

$\frac{\partial L(w)}{\partial z} = \frac{1}{1+e^{-z_i}} - y_i$, $w^{t+1} = w^t + \alpha \sum_{i=1}^{n} [y_i - (w^t \cdot x_i)] x_i$

Bayes rule: $P(y_k|x_i) = \frac{P(x_i|y_k)P(y_k)}{P(x_i)}$

fitted logistic regression for binary classification

$w = [-1.8, 7, -4, 4]$, first is bias

Input $= [0, 0, 0, 0.2]$, $z = w_0 + \sum_{i=1}^{n} w_i x_i = -1.8$

$P(y=1|x) = \frac{1}{1+e^{-z}} = \frac{1}{1+6.0496} \approx 0.142$

fitted softmax classifier for 3-class

weights: $\begin{bmatrix} 10, & -0.6, & 1, & -2.6, & -1, \\ 2.3, & 0.3, & -0.5, & -0.1, & -1.1 \\ -11.9, & -0.1, & -0.7, & 2.8, & 2 \end{bmatrix}$

Input $= [5.5, 4.2, 1.4, 0.2]$

$S_0 = 10.0 \times 1 + (-0.6) \times 5.5 + 1.0 \times 4.2 + (-2.6) \times 1.4 + (-1.0) \times 0.2 = 8.16$

$S_1 = 1.59$, $S_2 = -9.97$, $P(y=0|x) = \frac{e^{S_0}}{e^{S_0}+e^{S_1}+e^{S_2}} \approx 0.999$

$P(y=1|x) \approx 0.001$, $P(y=2|x) \approx 0.000$

x_1: hours studied, x_2: undergrad GPA, y: receive an A

$\hat{w}_0 = -8$, $\hat{w}_1 = 0.05$, $\hat{w}_2 = 0.86$. $P(y=1|x_1=25, x_2=3.85) = ?$

$z = w_0 + w_1 x_1 + w_2 x_2 = -3.369$, $P(y=1|x) = \frac{1}{1+e^{-z}} = 0.0333$

how many hours need for 50% chance of an A?

$P(y=1|x) = 0.5 = \frac{1}{1+e^{-z}} \Rightarrow z = -8 + 0.05 x_1 + 3.35 \times 0.86 = 0$

$P(y=0|x) = g_0(z_0, z_1) = \frac{e^{z_0}}{e^{z_0}+e^{z_1}}$, $P(y=1|x) = g_1(z_0, z_1) = \frac{e^{z_1}}{e^{z_0}+e^{z_1}}$

$= \frac{1}{1+e^{z_0-z_1}} = \frac{1}{1-e^{-(z_1-z_0)}} = \sigma(z_1 - z_0)$

threshold $= 0.5$, initial weights $w_0 = 3$, $w_1 = 1$, $w_2 = 1$

$x_{i,0}$	$x_{i,1}$	$x_{i,2}$	y_i	z_i	$P(y_i=1\|x_i)$	$\hat{y}_i(0.01)$	Correct
1	-1	1	0	0.00	0.50	1	No
1	-3	1	1	-3.00	0.00	0	No
1	-1	-2	1	-3.00	0.50	0	No
1	2	-2	0	9.00	1.00	1	No

$z_i = \sum_{k=0}^{n} W_k X_{i,k}$, $P(y_i=1|x_i) = \frac{1}{1+e^{-z_i}}$

$L(w) = \sum_{i=1}^{n} -(y_i \log \frac{1}{1+e^{-(w \cdot x_i)}} + (1-y_i) \log \frac{e^{-(w \cdot x_i)}}{1+e^{-(w \cdot x_i)}})$

$w^{t+1} = w^t + \alpha \sum_{i=1}^{n} (y_i - \frac{1}{1+e^{-(w \cdot x_i)}}) x_i$

$L(w) = -(\log \frac{e^{-(0)}}{1+e^{-(0)}} + \log \frac{1}{1+e^{-(-8)}} + \log \frac{1}{1+e^{-(-2)}})$

(note the log is a natural log) $+ \log \frac{e^{-(-9)}}{1+e^{-(-9)}}) = 20.x$

if $\alpha = 0.4$, $(y_i - \frac{1}{1+e^{-(w \cdot x_i)}}) x_i =$

$(0-0.5) \cdot [1, -1, 1]$

$(1-0.000335) \cdot [1, -3, 1]$

$(1-0.0476426) \cdot [1, -1, -2]$

$(0-0.999877) \cdot [1, 2, -2]$

$= \begin{bmatrix} 0.45376, & -5.45132, \\ 0.8542 \end{bmatrix}$ times α $+ [w_0, w_1, w_2]$

w_0 w_1 w_2

3.18 1.82 1.24

$z_i' = \sum_{k=0}^{n} W_k' X_{i,k}$; $L(w) = -(\log \frac{e^{-z_0'}}{1+e^{-z_0'}} + \log \frac{1}{1+e^{-z_1'}} + \log \frac{1}{1+e^{-z_2'}} + \log \frac{e^{-z_3'}}{1+e^{-z_3'}})$

COVID

```
df_tr_poly = df_tr.daysElapsed.values.reshape(-1,1)**np.arange(1, 5)
df_ts_poly = df_ts.daysElapsed.values.reshape(-1,1)**np.arange(1, 5)
reg_covid = LinearRegression().fit(df_tr_poly, df_tr.deathIncrease)
deathIncrease_fitted = reg_covid.predict(df_tr_poly)
metrics.r2_score(df_tr.deathIncrease, deathIncrease_fitted)
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))
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):
      x_train_dtest =  x_train_kfold[:, :dtest]
      x_val_dtest   =  x_val_kfold[:, :dtest]
      reg_dtest = LinearRegression().fit(x_train_dtest, y_train_kfold)
      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)
      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)
```

Linear Regression K-fold

The outer loop can be used to divide the data into training and validation, but then we'll also need an inner loop to train and validate each model for this particular fold.

In this case, suppose we want to evaluate polynomial models with different orders from

$$d = 1, \quad \hat{y} = w_0 + w_1 x$$

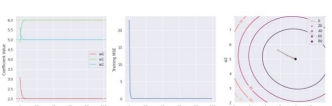
to

$$d = 10, \quad \hat{y} = w_0 + w_1 x + w_2 x^2 + \ldots + w_{10} x^{10}$$

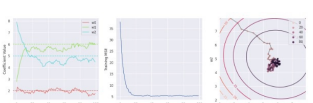
We could do something like this:

```
nfold = 5
kf = KFold(n_splits=nfold,shuffle=True)
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))
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):
    x_train_dtest =  x_train_kfold[:, :dtest]
    x_val_dtest   =  x_val_kfold[:, :dtest]
    reg_dtest = LinearRegression().fit(x_train_dtest, y_train_kfold)
    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)
```

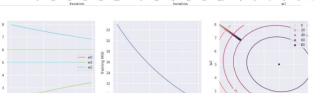
```
# mse
idx_min = np.argmin(mse_val.mean(axis=1))
target = mse_val[idx_min,:].mean() + mse_val[idx_min,:].std()/np.sqrt(nfold-1)
idx_one_se = np.where(mse_val.mean(axis=1) <= target)
d_one_se = np.min(dtest_list[idx_one_se])
d_one_se
# r2
idx_max = np.argmax(r2_val.mean(axis=1))
target_r2 = r2_val[idx_max,:].mean() - r2_val[idx_max,:].std()/np.sqrt(nfold-1)
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
```



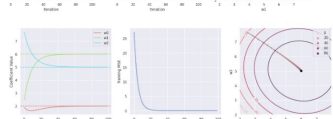
a. Batch gradient descent with large learning rate, no noise in training data.



b. Mini-batch (small batch size) gradient descent with moderate learning rate, high noise in training data.



c. Batch gradient descent with small learning rate, no noise in training data.



d. Batch gradient descent with moderate learning rate, no noise in training data.

Scalar

```
# Scalar
scaler = StandardScaler().fit(Xtr)
Xtr_std = scaler.transform(Xtr)
Xts_std = scaler.transform(Xts)

vec = CountVectorizer(stop_words='english')
Xtr_vec = vec.fit_transform(Xtr_str)
Xts_vec = vec.transform(Xts_str)
```

Linear Regression

```
# Linear Regression
features = ['Parks_Nearby', 'Grocery_Stores_Nearby', 'Schools_Nearby', 'Public_Transit_Nearby']
target = ['Walkability_Score']
X = df[features]
y = df[target]
random_state = 23
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.2, random_state=random_state)
model = LinearRegression().fit(Xtr, ytr)
yts_hat = model.predict(Xts)
rsq = r2_score(yts, yts_hat)
mse = mean_squared_error(yts, yts_hat)
```

```
# Single split - random shuffle
random_state = 8
Xtr_one_shuf, Xts_one_shuf, ytr_one_shuf, yts_one_shuf = train_test_split(X, y, test_size=1/5, random_state=random_state)
model = LinearRegression()
model.fit(Xtr_one_shuf, ytr_one_shuf)
yts_one_shuf_pred = model.predict(Xts_one_shuf)
r2_one_shuf = r2_score(yts_one_shuf, yts_one_shuf_pred)
# Single split - sorted data, no shuffle
Xtr_one_order, Xts_one_order, ytr_one_order, yts_one_order = train_test_split(X, y, test_size=1/5, shuffle=False)
model = LinearRegression()
model.fit(Xtr_one_order, ytr_one_order)
yts_one_order_pred = model.predict(Xts_one_order)
r2_one_order = r2_score(yts_one_order, yts_one_order_pred)
# Multiple splits - random shuffle
n_fold = 5
r2_kf_shuffle = np.zeros(shape=(n_fold,))
kf = KFold(n_splits=n_fold, shuffle=True, random_state=random_state)
for i, (idx_tr, idx_ts) in enumerate(kf.split(X)):
    model = LinearRegression()
    model.fit(X[idx_tr], y[idx_tr])
    y_pred_kfold = model.predict(X[idx_ts])
    r2_kf_shuffle[i] = r2_score(y[idx_ts], y_pred_kfold)
r2_kf_shuffle_mean = np.mean(r2_kf_shuffle)
# Multiple splits - time series
n_fold = 5
r2_ts = np.zeros(shape=(n_fold,))
ts = TimeSeriesSplit(n_splits=n_fold)
for i, (idx_tr, idx_ts) in enumerate(ts.split(X)):
    model = LinearRegression()
    model.fit(X[idx_tr], y[idx_tr])
    y_pred = model.predict(X[idx_ts])
    r2_ts[i] = r2_score(y[idx_ts], y_pred)
r2_ts_mean = np.mean(r2_ts)
```

Ridge Regression

```
x_names = ['carat', 'cut', 'color', 'clarity', 'depth', 'table', 'x', 'y', 'z']
y_names = ['price']
random_state = 13
Xtr_df, Xts_df = train_test_split(df[x_names], test_size=0.3, random_state=random_state, shuffle=True)
ytr_df, yts_df = train_test_split(df[y_names], test_size=0.3, random_state=random_state, shuffle=True)
Xtr, Xts, ytr, yts = np.array(Xtr_df), np.array(Xts_df), np.array(ytr_df), np.array(yts_df)
alpha_list = np.array([0, 10, 20, 50, 100, 200, 500])
nfold = 5
mse_val = np.zeros((len(alpha_list), nfold))
# k-ford
kf = KFold(n_splits=nfold, shuffle=False)
# For each fold, standardize the data
for ifold, (idx_tr, idx_val) in enumerate(kf.split(Xtr)):
    X_train_fold, X_val_fold = Xtr[idx_tr], Xtr[idx_val]
    y_train_fold, y_val_fold = ytr[idx_tr], ytr[idx_val]
    scaler = StandardScaler().fit(X_train_fold)
    X_train_fold_std = scaler.transform(X_train_fold)
    X_val_fold_std = scaler.transform(X_val_fold)
    # For each alpha in the list, fit a Ridge regression model on the standardized data
    for i, alpha in enumerate(alpha_list):
        model = Ridge(alpha=alpha)
        model.fit(X_train_fold_std, y_train_fold)
        y_pred = model.predict(X_val_fold_std)
        # update the appropriate entry in mse_val
        mse_val[i, ifold] = mean_squared_error(y_val_fold, y_pred)
mse_mean = np.mean(mse_val, axis=1)
alpha_min_mse = alpha_list[np.argmin(mse_mean)]
# entire training set
scaler = StandardScaler().fit(Xtr)
Xtr_std = scaler.transform(Xtr)
Xts_std = scaler.transform(Xts)
model = Ridge(alpha=alpha_min_mse)
model.fit(Xtr_std, ytr)
y_pred = model.predict(Xts_std)
mse_ridge = mean_squared_error(yts, y_pred)
```

K-fold CV with Fourier basis expansion

You decide to use a linear regression model with Fourier basis transformation of the `hourofweek` feature:

$$\hat{y} = w_0 + w_1 x + \sum_{t \in \texttt{tlist}} w_{t,c}\cos(2\pi x/t) + w_{t,s}\sin(2\pi x/t)$$

where each sine and cosine pair represents the periodic behavior over a particular time interval.

For example, if `tlist = [0.5, 1]`, then your model would be:

$$\hat{y} = w_0 + w_1 x + w_{0.5,c}\cos(2\pi x/0.5) + w_{0.5,s}\sin(2\pi x/0.5) + w_{1,c}\cos(2\pi x/1) + w_{1,s}\sin(2\pi x/1)$$

Now, you are ready to fit a K-fold CV! In your CV, you will fit and evaluate a `LinearRegression` model (using the `sklearn` implementation) on an increasing number of columns of the data, as described above -

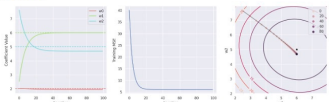
- in the first iteration of your K-fold CV, you will evaluate the regression for `tlist_eval = []`.
- In the second iteration of your K-fold CV, you will evaluate the regression for `tlist_eval = [1]`.
- In the third iteration, you will evaluate the regression for `tlist_eval = [1, 2]`

and so on, until, in the final iteration, you will evaluate the regression for *all* the values in `tlist`.

(Of course, you won't re-compute the Fourier basis transformation inside the loop - you'll just select the appropriate rows and columns from `Xtr_trans` in each iteration.)

Since you have prepared a "ones column" in the data, you will pass `fit_intercept=False` as an argument to the `LinearRegression`, so that it won't also fit another "intercept" term (in addition to the coefficient for the "ones column".)

e. Batch gradient descent with moderate learning rate, high noise in training data.



f. Mini-batch (small batch size) gradient descent with moderate learning rate, no noise in training data.

```
# Linear Regression
df_tr, df_ts = train_test_split(df, train_size=10000, shuffle=False, random_state=42)
features = ['hour', 'month', 'dayofweek', 'hourofweek']
target = 'ridership'
model = LinearRegression()
model.fit(df_tr[features], df_tr[target])
y_pred = model.predict(df_ts[features])
r2_lr = r2_score(df_ts[target], y_pred)
# K-fold CV with fourier basis expansion
tlist = np.arange(1, 51)
Xtr = df_tr['hourofweek'].values
Xts = df_ts['hourofweek'].values
ytr = df_tr['ridership'].values
yts = df_ts['ridership'].values
Xtr_trans = np.column_stack((np.ones_like(Xtr), Xtr))
Xts_trans = np.column_stack((np.ones_like(Xts), Xts))
for t in tlist:
    cos_tr = np.cos(2 * np.pi * Xtr / t)
    sin_tr = np.sin(2 * np.pi * Xtr / t)
    cos_ts = np.cos(2 * np.pi * Xts / t)
    sin_ts = np.sin(2 * np.pi * Xts / t)
    Xtr_trans = np.column_stack((Xtr_trans, cos_tr, sin_tr))
    Xts_trans = np.column_stack((Xts_trans, cos_ts, sin_ts))
Xtr_trans = np.round(Xtr_trans, 10)
Xts_trans = np.round(Xts_trans, 10)

nfold = 5
r2_val = np.zeros((len(tlist) + 1, nfold))
kf = KFold(n_splits=nfold, shuffle=False)
for i, t in enumerate(range(len(tlist) + 1)):
    # The first model uses only the intercept and original X (2 columns)
    # Subsequent models add Fourier features in pairs (cos and sin)
    num_columns = 2 + 2 * i
    X_subset = Xtr_trans[:, :num_columns]
    for fold, (train_idx, val_idx) in enumerate(kf.split(X_subset)):
            X_train, X_val = X_subset[train_idx], X_subset[val_idx]
            y_train, y_val = ytr[train_idx], ytr[val_idx]
            model = LinearRegression(fit_intercept=False)
            model.fit(X_train, y_train)
            y_pred = model.predict(X_val)
            r2_val[i, fold] = r2_score(y_val, y_pred)

r2_mean = r2_val.mean(axis=1)
r2_se = np.std(r2_val, axis=1, ddof=1) / np.sqrt(nfold)

idx_max = np.argmax(r2_mean)
# Compute the threshold for the one-SE rule
one_se = r2_mean[idx_max] - r2_se[idx_max]
# Find the simplest model (smallest number of Fourier features) whose R² is within one SE of the best model
tlist_opt = []
for i in range(len(r2_mean)):
    if r2_mean[i] >= one_se:
        tlist_opt = tlist[:i]  # Select the first i values of tlist
        break
# Train a model on the entire training set for this `tlist_opt`, then evaluate its performance on the test set.
# Save the test R2 score in r2_one_se.
num_columns = 2 + 2 * len(tlist_opt)
Xtr_opt = Xtr_trans[:, :num_columns]
Xts_opt = Xts_trans[:, :num_columns]
model_opt = LinearRegression(fit_intercept=False)
model_opt.fit(Xtr_opt, ytr)
y_pred_opt = model_opt.predict(Xts_opt)
r2_one_se = r2_score(yts, y_pred_opt)
```

Logistic Regression

```
# Standardize data in each fold.
random_state = 14
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.3, random_state=random_state)
med_values = np.nanmedian(Xtr, axis=0)
Xtr_filled = np.nan_to_num(Xtr, nan=med_values)
Xts_filled = np.nan_to_num(Xts, nan=med_values)
C_test = np.logspace(-1,3,10)
nfold = 3
acc_val = np.zeros((len(C_test), nfold))
for iC, C in enumerate(C_test):
    kf = KFold(n_splits=nfold, shuffle=False)
    for ifold, (Itr, Ival) in enumerate(kf.split(Xtr_filled)):
        Xtr_fold, Xval_fold = Xtr_filled[Itr], Xtr_filled[Ival]
        ytr_fold, yval_fold = ytr[Itr], ytr[Ival]
        scaler = StandardScaler().fit(Xtr_fold)
        Xtr_std = scaler.transform(Xtr_fold)
        Xvl_std = scaler.transform(Xval_fold)
        clf = LogisticRegression(random_state = random_state, solver = 'liblinear', penalty='l1', C = C)
        clf.fit(Xtr_std, ytr_fold)
        yhat = clf.predict(Xvl_std)
        acc_val[iC, ifold] = accuracy_score(yval_fold, yhat)
acc_mean = np.mean(acc_val, axis=1)
C_best = C_test[np.argmax(acc_mean)]
# entire training set
scaler = StandardScaler().fit(Xtr_filled)
Xtr_std = scaler.transform(Xtr_filled)
Xts_std = scaler.transform(Xts_filled)
clf_best = LogisticRegression(random_state = random_state, solver = 'liblinear', penalty='l1', C = C_best)
clf_best.fit(Xtr_std, ytr)
y_hat = clf_best.predict(Xts_std)
```

If the doctor spends t minutes reviewing a file, the probability of *missing* the correct treatment is

$$P(\text{fail}) = \exp(-t^2/100).$$

Policy 0: No extra review

Policy 1: Review all files

Now suppose the specialist team will spend

$$\frac{N}{25} \times 60$$

minutes total on extra review, where N is the number of files in the validation set, and they will spend an equal amount of time on each of the N files in the validation set.

Policy 2: Review high-risk files with 0.5 threshold

Now, suppose that the doctor only reviews the files of patients for whom the model outputs a probability of mortality greater than 50%. They will spend the same total amount of time as before,

$$\frac{N}{25} \times 60$$

Compute:

- `t_2` - the time (in minutes) spent on each file
- `p_2` - the probability of failing to identify the correct intervention, given this `t` and then use them to find the expected number of deaths in the validation set, with this policy. Save this value in `mort_2`.

Policy 3: Review high-risk files with optimal threshold

For all thresholds in `np.arange(0, 1, 0.01)`, compute the expected mortality, given the doctor only reviews the files of patients for whom the model outputs a probability of mortality greater than that threshold. They will spend the same total amount of time as before,

$$\frac{N}{25} \times 60$$

minutes total on extra review, where N is the number of files in the validation set. Save the results in `mort_thr`.

```
Xtr_str, Xts_str, ytr, yts = train_test_split(df['statement'].values, df['label_binary'].values, shuffle=True, random_state=0, test_size=0.25)
vec = CountVectorizer(stop_words='english')
Xtr_vec = vec.fit_transform(Xtr_str)
Xts_vec = vec.transform(Xts_str)
C_test = np.logspace(-3, 3, num=20)
nfold = 5
acc_val = np.zeros((len(C_test), nfold))
kf = KFold(n_splits=nfold)
for ifold, (Itr, Ival) in enumerate(kf.split(Xtr_vec)):
    for iC, C in enumerate(C_test):
        clf = LogisticRegression(random_state = 0, penalty='l1', solver='liblinear', C = C)
        clf.fit(Xtr_vec[Itr], ytr[Itr])
        yhat = clf.predict(Xtr_vec[Ival])
        acc_val[iC, ifold] = accuracy_score(ytr[Ival], yhat)
acc_mean = np.mean(acc_val, axis=1)
C_best = C_test[np.argmax(acc_mean)]
model_best = LogisticRegression(penalty='l1', C=C_best, random_state=0, solver='liblinear')
model_best.fit(Xtr_vec, ytr)
y_pred_best = model_best.predict(Xts_vec)
acc_best = accuracy_score(yts, y_pred_best)
count_best = np.count_nonzero(model_best.coef_)
acc_std = acc_val.std(axis=1)
acc_one_se = acc_mean - acc_std
C_one_se = C_test[np.argmax(acc_mean >= np.max(acc_one_se))]
model_one_se = LogisticRegression(penalty='l1', C=C_one_se, random_state=0, solver='liblinear')
model_one_se.fit(Xtr_vec, ytr)
y_pred_one_se = model_one_se.predict(Xts_vec)
acc_one_se = accuracy_score(yts, y_pred_one_se)
count_one_se = np.count_nonzero(model_one_se.coef_)
```

```
# Policy 0: No extra review
mort_0 = np.sum(yvl)

# Policy 1: Review all files
N = len(yvl)
# Step 1: Compute time spent per file
t_1 = (N / 25) * 60  / N  # Minutes per file
# Step 2: Compute probability of missing correct intervention
p_1 = np.exp(-t_1**2 / 100)
# Step 3: Compute expected mortality under Policy 1
mort_1 = np.sum(yvl * p_1)
# Step 1: Predict probabilities of mortality for the validation set

# Policy 2: Review high-risk files with 0.5 threshold
y_proba_v1 = clf.predict_proba(Xvl)[:, 1]
# Identify high-risk and low-risk patients
high_risk_indices = [i for i in range(len(yvl)) if y_proba_v1[i] > 0.5]
low_risk_indices = [i for i in range(len(yvl)) if y_proba_v1[i] <= 0.5]
# Number of high-risk files
N_high_risk = len(high_risk_indices)
# Step 2: Calculate t_2
N = len(yvl)
T = (N / 25) * 60
t_2 = T / N_high_risk
# Step 3: Calculate p_2
p_2 = np.exp(-t_2**2 / 100)
# Step 4: Estimate expected deaths
mort_2 = 0
for i in high_risk_indices:
    if yvl[i] == 1:  # Patient would have died without review
        mort_2 += p_2  # Adjusted probability of death
    else:
        mort_2 += 0  # No death for survivors
for i in low_risk_indices:
    mort_2 += yvl[i]  # No review, so deaths remain unchanged

# Policy 3: Review high-risk files with optimal threshold
# Step 1: Predict probabilities of mortality for the validation set
y_proba_v1 = clf.predict_proba(Xvl)[:, 1]  # Predicted probabilities for validation set
# Total number of files in the validation set
N = len(yvl)
# Total time spent on extra review (same as in Policy 1 and Policy 2)
T = (N / 25) * 60
# Step 2: Initialize arrays to store results
thr_list = np.arange(0, 1, 0.01)  # List of thresholds
mort_thr = np.zeros(len(thr_list))  # Array to store expected mortality for each threshold
# Step 3: Iterate over thresholds
for i, t in enumerate(thr_list):
    # Identify high-risk patients (probability > threshold)
    high_risk_indices = np.where(y_proba_v1 > t)[0]
    low_risk_indices = np.where(y_proba_v1 <= t)[0]
    # Number of high-risk files
    N_high_risk = len(high_risk_indices)
    # Skip if there are no high-risk patients (to avoid division by zero)
    if N_high_risk == 0:
        mort_thr[i] = np.sum(yvl[low_risk_indices])  # No review, so deaths remain unchanged
        continue
    # Calculate time spent on each high-risk file
    t_2_3 = T / N_high_risk
    # Calculate probability of failing to identify the correct intervention
    p_2 = np.exp(-t_2_3**2 / 100)
    # Estimate expected deaths
    mort_high_risk = np.sum(yvl[high_risk_indices] * p_2)  # Adjusted probability for high-risk patients
    mort_low_risk = np.sum(yvl[low_risk_indices])  # No review for low-risk patients
    # Total expected deaths
    mort_thr[i] = mort_high_risk + mort_low_risk
```

Predicting and Preventing Deaths in the ICU

```
# Prepare
X = df[col_names[:-1]]
y = df['In-hospital_death'].values
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=800, shuffle=True, random_state=0)
Xtr, Xvl, ytr, yvl = train_test_split(Xtr, ytr, test_size=800, shuffle=True, random_state=0)
col_meds = Xtr.median()
Xtr = Xtr.fillna(col_meds)
Xvl = Xvl.fillna(col_meds)
Xts = Xts.fillna(col_meds)
# Logistic Regression
clf = LogisticRegression(random_state = 0, penalty=None)
clf.fit(Xtr, ytr)
y_pred_lr = clf.predict(Xvl)
acc_lr = accuracy_score(yvl, y_pred_lr)
most_frequent_label = np.bincount(ytr).argmax()
y_pred_base = np.full_like(yvl, most_frequent_label)
acc_base = accuracy_score(yvl, y_pred_base)
# ROC curve
y_proba = clf.predict_proba(Xvl)[:, 1]
fpr, tpr, thresholds = roc_curve(yvl, y_proba)
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

Evaluate on the test set

```
mort_ts = np.sum(yts)
# Policy 0: no review, so no lives are saved
saved_0 = 0
# Policy 1: review all
N = len(yts)
total_time = (N / 25) * 60
t_ts_1 = total_time / N
p_ts_1 = np.exp(-(t_ts_1 ** 2) / 100)
at_risk_patients = np.sum(yts)
mort_ts_1 = at_risk_patients * p_ts_1
saved_1 = mort_ts - mort_ts_1
# Policy 2: review using 50% threshold
y_proba_ts = clf.predict_proba(Xts)[:, 1]
high_risk_patients = y_proba_ts > 0.5
num_high_risk = np.sum(high_risk_patients)
t_ts_2 = total_time / num_high_risk if num_high_risk > 0 else 0
p_ts_2 = np.exp(-(t_ts_2 ** 2) / 100)
at_risk_high_risk = np.sum(yts[high_risk_patients])
at_risk_low_risk = np.sum(yts[~high_risk_patients])
mort_ts_2 = at_risk_high_risk * p_ts_2 + at_risk_low_risk
saved_2 = mort_ts - mort_ts_2
# Policy 3: review using thr_opt
high_risk_patients = y_proba_ts > thr_opt
num_high_risk = np.sum(high_risk_patients)
t_ts_3 = total_time / num_high_risk if num_high_risk > 0 else 0
p_ts_3 = np.exp(-(t_ts_3 ** 2) / 100)
at_risk_high_risk = np.sum(yts[high_risk_patients])
at_risk_low_risk = np.sum(yts[~high_risk_patients])
mort_ts_3 = at_risk_high_risk * p_ts_3 + at_risk_low_risk
saved_3 = mort_ts - mort_ts_3
```

Find

- `thr_opt`, the threshold that minimizes mortality on the validation set, and
- `mort_3`, the expected mortality on the validation set with this threshold.

```
thr_opt = thr_list[np.argmin(mort_thr)]
# mort_3 = np.min(mort_thr)
mort_3 = np.min(mort_thr[mort_thr > 0]) if np.any(mort_thr > 0) else np.min(mort_thr)
```