Lecture 9 & 10 - Module 4.2 Logistic Regression

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

Feb 4 & 6, 2025

Outline

Outline

Learning objectives

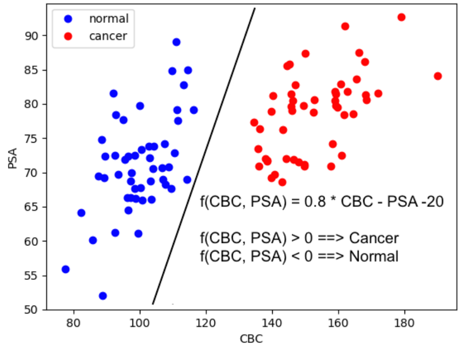
Understanding the following concepts

- Logistic function
- Cross-entropy cost function
- Fitting logistic regression by gradient descent
- Probabilistic view of logistic regression

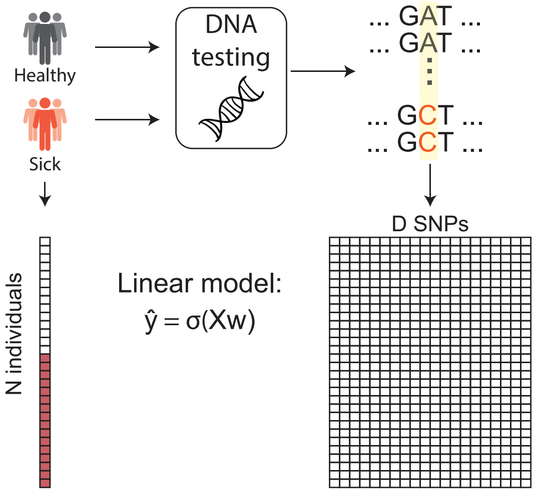
Outline

Linear function for binary classification

With one or two-dimensional input, it is not hard to think of a linear function $w_1 x_2 + w_2 x_2$ that separates positive and negative examples.



With high-dimensional input ($D >> 2$), however, it becomes impossible to do.



Logistic function

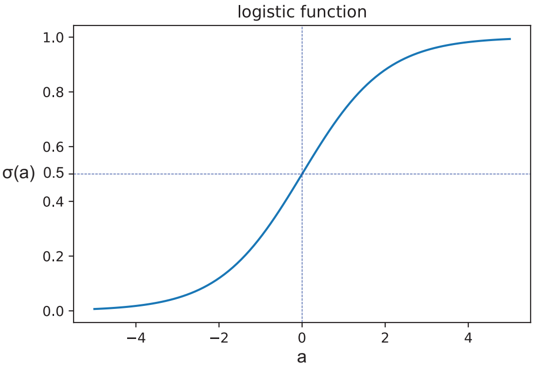
Logistic function transforms the real-value $a = \mathbf{xw} \in \mathbb{R}$ into $\hat{y} \in [0, 1]$, which can be interpreted as the *probability* being class 1.

$$\hat{y} = \sigma(a) = \frac{1}{1 + \exp(-a)} \tag{1}$$

The inverse of the logistic function is called **logit function** (try work out the math):

$$\log \frac{\hat{y}}{1 - \hat{y}} = a \tag{2}$$

which is the log-odd ratio of the probability being positive case over the probability being negative class.



$\sigma(a) = 0.5$ if $a = 0$, which indicates "neutral" (i.e., either positive or negative). Therefore, $a = 0$ is the decision boundary.

Cross entropy as the preferred loss function to others (Colab)

Given that $\hat{y} = \sigma(\mathbf{xw}) = 1/(1 + \exp(-\mathbf{xw}))$, we consider four candidate loss functions. Assuming $y = 1, x = 1$ and therefore the more positive w is the lower the error.

Direct loss is not differentiable:

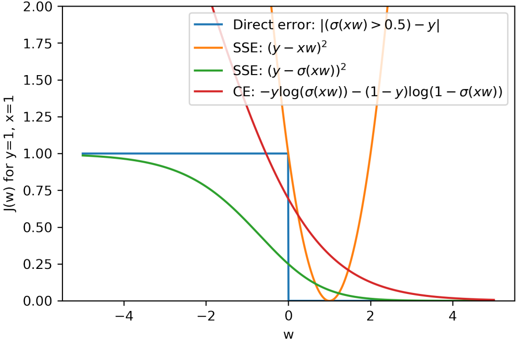
$$\mathcal{L}(\hat{y}, y) = |\mathbb{I}[\hat{y} > 0.5] - y|$$

The SSE loss using **xw** as prediction increases for highly positive **xw**:

$$\mathcal{L}(\mathbf{xw}, y) = (y - \mathbf{xw})^2$$

SSE loss using $\hat{y} = \sigma(\mathbf{xw})$ is not convex:

$$\mathcal{L}(\mathbf{xw}, y) = (y - \hat{y})^2$$



Cross-Entropy (CE) is convex and has a nice probabilistic interpretation (Section 4)

$$CE(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y})$$

Numerically precise implementation of CE using `np.log1p`

`np.log1p(x)` computes $\log(1+x)$ for accurate floating point.

```
a = np.dot(x, w)
J = np.sum(y * np.log1p(np.exp(-a)) + (1-y) * np.log1p(np.exp(a)))
```

$$
\begin{aligned}
J(w) &= \sum_n -y^{(n)} \log\left(\frac{1}{1+\exp(-a^{(n)})}\right) - (1-y^{(n)}) \log\left(1 - \frac{1}{1+\exp(-a^{(n)})}\right) \\
&= \sum_n y^{(n)} \log(1+\exp(-a^{(n)})) - (1-y^{(n)}) \log(\frac{\exp(-a^{(n)})}{1+\exp(-a^{(n)})}) \\
&= \sum_n y^{(n)} \log(1+\exp(-a^{(n)})) - (1-y^{(n)}) \log(\frac{1}{\exp(a^{(n)})+1}) \\
&= \sum_n y^{(n)} \log(1+\exp(-a^{(n)})) + (1-y^{(n)}) \log(1+\exp(a^{(n)}))
\end{aligned}
$$

Try this:

```
np.log(1+1e-100) # 0
np.log1p(1e-100) # 1e-100
```

Outline

Gradient calculation

Let's start with one training example $\{\mathbf{x}, y\}$ to not clutter the notation. Let $\hat{y} = 1/(1 + \exp(-a))$, where $a = \mathbf{xw}$. Our goal is to minimize CE w.r.t. $\mathbf{w}$:

$$J(\mathbf{w}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

We break down the partial derivative of $J(w)$ w.r.t. w_d for feature d by chain rule:

$$\frac{\partial J(\mathbf{w})}{\partial w_d} = \frac{\partial J(\mathbf{w})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w_d}$$

Let's solve these three gradients one by one:

$$\begin{aligned}
\frac{\partial J(\mathbf{w})}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} - y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) \\
&= -y \frac{\partial}{\partial \hat{y}} \log \hat{y} - (1 - y) \frac{\partial \log(1 - \hat{y})}{\partial (1 - \hat{y})} \frac{\partial (1 - \hat{y})}{\partial \hat{y}} \\
&= -\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}(-1) = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}
\end{aligned} \tag{3}$$

$$
\begin{aligned}
\frac{\partial \hat{y}}{\partial a} &= \frac{\partial}{\partial a}(1+\exp(-a))^{-1} \\
&= \frac{\partial(1+\exp(-a))^{-1}}{\partial 1+\exp(-a)} \frac{\partial 1+\exp(-a)}{\partial - a} \frac{\partial - a}{\partial a} \\
&= -(1+\exp(-a))^{-2}\exp(-a)(-1) \\
&= (1+\exp(-a))^{-2}\exp(-a) \\
&= \frac{1}{1+\exp(-a)} \frac{\exp(-a)}{1+\exp(-a)} \\
&= \frac{1}{1+\exp(-a)} \left(1 - \frac{1}{1+\exp(-a)}\right) \\
&= \hat{y}(1-\hat{y}) \\
\frac{\partial a}{\partial w_d} &= \frac{\partial}{\partial w_d} \sum_d x_d w_d = x_d
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J(\mathbf{w})}{\partial w_d} &= \frac{\partial J(\mathbf{w})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w_d} \\
&= (-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}})(\hat{y}(1-\hat{y}))x_d \\
&= -y(1-\hat{y})x_d + (1-y)\hat{y}x_d \\
&= -yx_d + y\hat{y}x_d + \hat{y}x_d - y\hat{y}x_d \\
&= (\hat{y}-y)x_d
\end{aligned}
$$

The gradient suggests that to update weight w_d, we use the prediction error weighted by the corresponding feature x_d. We can represent the gradients over all features $d \in \{1\ldots,D\}$ in matrix form:

$$
\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{x}^\top(\hat{y}-y)
$$

Logistic regression training algorithm by gradient descent

For N individuals, we can add the gradients together for each feature:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^{N} \frac{\partial J^{(n)}(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^{N} (\hat{y}^{(n)} - y^{(n)})\mathbf{x}^{(n)} = \mathbf{X}^\intercal(\hat{\mathbf{y}} - \mathbf{y})$$

Unlike in the linear regression case, where we have closed-form solution for $\mathbf{w}$ (i.e., $\mathbf{w} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$), to train a logistic regression, we cannot solve $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = 0$ for $\mathbf{w}$.

Compare with the gradients for the linear regression weights in Module 4.1.

To update the logistic regression model, we perform **gradient descent** by *subtracting* the gradients from the existing weight *iteratively*: at the t-th iteration, we do:

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \alpha \frac{\partial J(\mathbf{w}^{(t-1)})}{\partial \mathbf{w}^{(t-1)}}$$

- We do subtraction because we want to minimize the error function by making the weights go in the opposite direction of error derivative.
- We multiply the gradients by a learning rate $\alpha \in [0, 1]$ to avoid overshooting the optimal values of $\mathbf{w}$.

Logistic regression training algorithm by gradient descent

Algorithm 1 LogisticRegression.fit($\mathbf{X}$, $\mathbf{y}$, $\alpha = 0.005$, $\epsilon = 10^{-5}$, max_iter=10^5)

Randomly initialize regression coefficients $w_d \sim \mathcal{N}(0, 1)\,\forall d$

for niter $= 1 \ldots$ max_iter **do**

  $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \alpha \frac{\partial J(\mathbf{w}^{(t-1)})}{\partial \mathbf{w}^{(t-1)}}$

  $\hat{\mathbf{y}} = 1/(1 + \exp(-\mathbf{X}\mathbf{w}^{(t)}))$

  $J(\mathbf{w}^{(t)}) = \sum_n -y^{(n)} \log(\hat{y}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}^{(n)})$

  if $|J(\mathbf{w}^{(t)}) - J(\mathbf{w}^{(t-1)})|) < \epsilon$ **then**

    break // *Converged so we quite before completing all iterations*

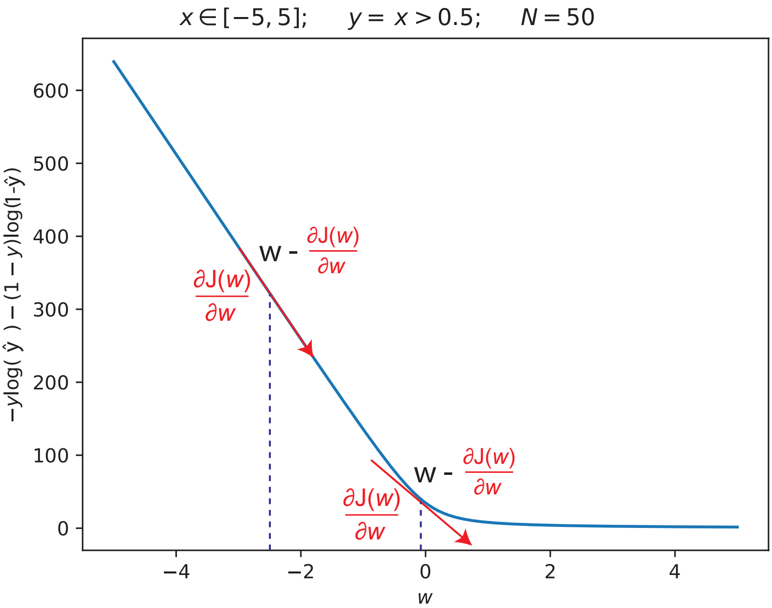
  end if

end for

Toy data (Colab)

```
N=50
x = np.linspace(-5,5, N)
y = (x > 0.5).astype(int)

lr = 0.001
niter = 10000
w = np.random.randn(1)
w0 = w
ce_all = np.zeros(niter)
for i in range(niter):
    y_hat = 1 / (1 + np.exp(-w * x))
    ce_all[i] = np.sum(-y * np.log(y_hat) - (1-y) * np.log(1-y_hat))
    dw = np.sum((y_hat - y) * x)
    w = w - lr * dw
```

Cross-entropy as a function of w



Verifying gradient calculation 1: small perturbation

Note that gradient is defined as:

$$\frac{\partial}{\partial w_d} J(w_1, w_2, \ldots, w_D) = \lim_{\epsilon \to 0} \frac{J(w_d + \epsilon, \mathbf{w}_{\setminus d}) - J(w_d - \epsilon, \mathbf{w}_{\setminus d})}{2\epsilon}$$

Analytically derived gradient can be error-prone. We can verify the gradient as follow:

1. $\epsilon \sim \mathit{Uniform}([0, 10^{-5}])$
2. $w_d^{(+)} = w_d + \epsilon$
3. $w_d^{(-)} = w_d - \epsilon$
4. $\nabla w_d = \frac{J(w_d^{(+)}, \mathbf{w}_{\setminus d}) - J(w_d^{(-)}, \mathbf{w}_{\setminus d})}{2\epsilon}$ (numerically estimated gradient)
5. $\frac{(\frac{\partial J(\mathbf{w})}{\partial w_d} - \nabla w_d)^2}{(\frac{\partial J(\mathbf{w})}{\partial w_d} + \nabla w_d)^2}$ must be small (e.g., 10^{-8}) otherwise your gradient calculation and/or your loss function are/is incorrect

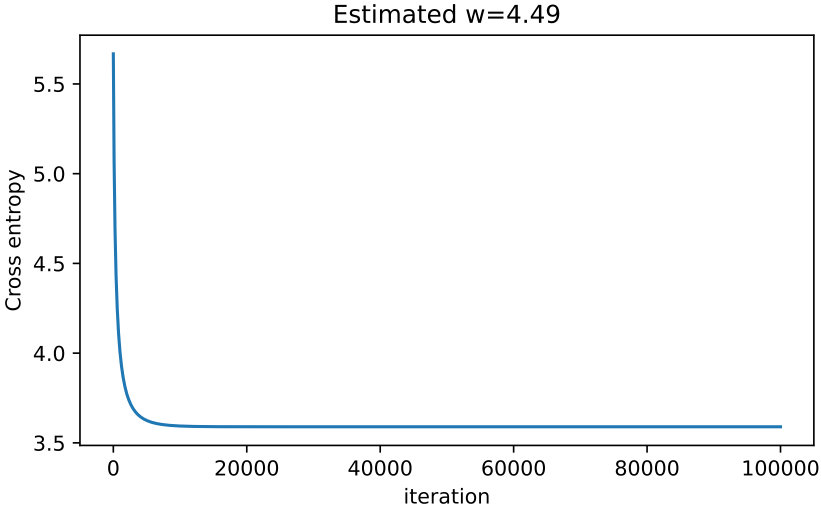
Python code for small perturbation test on a toy data (colab)

```
N=50
x = np.linspace(-5, 5, N)
y = (x > 0.5).astype(int)

# small perturbation
w = np.random.randn(1)
w0 = w
epsilon = np.random.randn(1)[0] * 1e-5
w1 = w0 + epsilon
w2 = w0 - epsilon
a1 = w1*x
a2 = w2*x
ce1 = np.sum(y * np.log1p(np.exp(-a1)) + (1-y) * np.log1p(np.exp(a1)))
ce2 = np.sum(y * np.log1p(np.exp(-a2)) + (1-y) * np.log1p(np.exp(a2)))
dw_num = (ce1 - ce2)/(2*epsilon) # approximated gradient

yh = 1/(1+np.exp(-x * w))
dw_cal = np.sum((yh - y) * x) # analytical gradient

print(dw_cal) # -22.812099331382
print(dw_num) # -22.812099334497717
print((dw_cal - dw_num)**2/(dw_cal + dw_num)**2) # 4.66e-21
```

Verifying gradient calculation 2: Monitor error decrease at each iteration

```
N=50
x = np.linspace(-5,5, N)
y = (x > 0.5).astype(int)

lr = 0.001
niter = 10000
w = np.random.randn(1)
w0 = w
ce_all = np.zeros(niter)
for i in range(niter):
    a = w * x
    ce_all[i] = np.sum(y * np.log1p(np.exp(-a)) \
        + (1-y) * np.log1p(np.exp(a))) # store CE at each iteration
    dw = np.sum((y_hat - y) * x)
    w = w - lr * dw
```

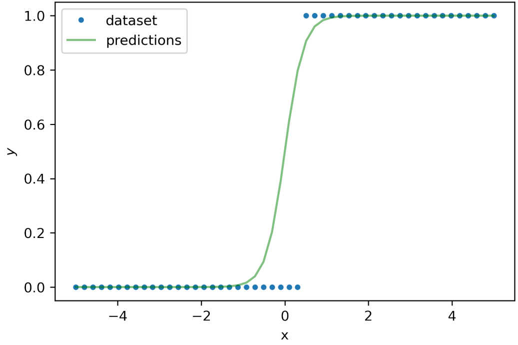
Verifying gradient calculation 2: Monitor decrease of training CE



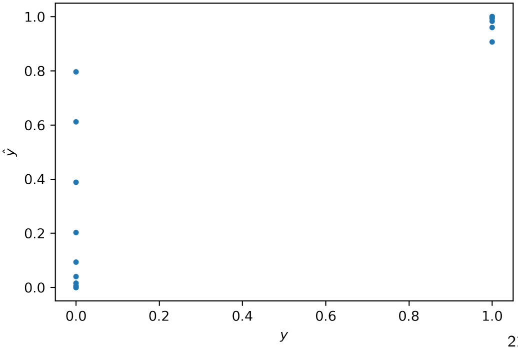
Visualizing predictions on 1D data

For the above toy data, we can also visualize the model prediction on the training set.

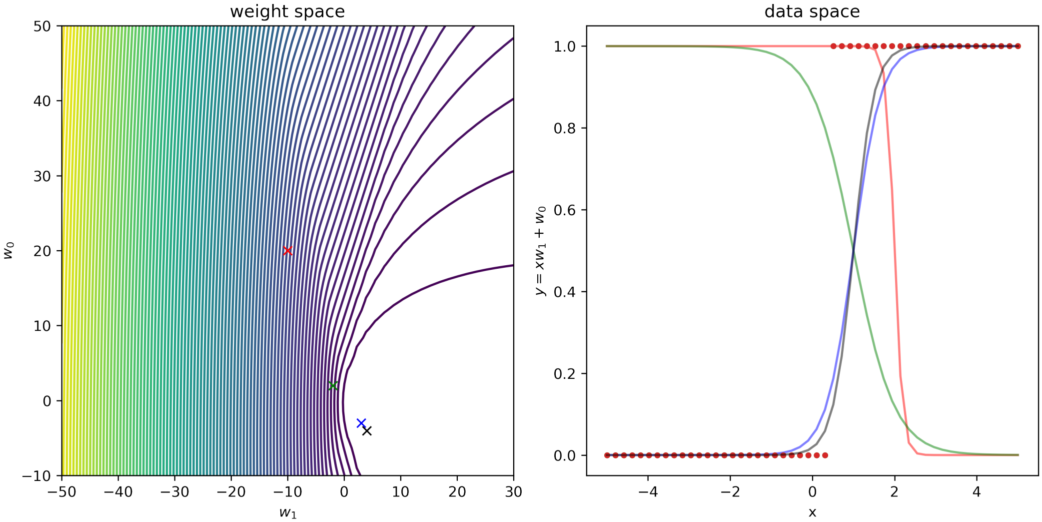
For 1D data, we can just visualize y as a function of x. We see the points where $\hat{y}$ is 0.90 when the input is 0.5 (the ground truth is $x > 0.5$)



We can also visualize $\hat{y}$ as a function of y. The plot indicates that when $\hat{y} > 0.8$, we have 100% precision (TP/(TP+FP)). At lower threshold, we start to have more false positives (i.e., lower precision).



Visualizing weights contour and their predictions



Outline

Bernoulli distribution and Binomial distribution

Bernoulli distribution has the following probability mass function (PMF):

$$p(y|\pi) = \pi^y (1-\pi)^{1-y} \tag{4}$$

where π is the rate of $y = 1$. A common example used is coin toss. If the coin lands on heads $y = 1$ or tails $y = 0$. A fair coin will have $\pi = 0.5$.

Binomial distribution models N independent Bernoulli trials. We can make N coin tosses to get a dataset of $\mathcal{D} = \{y^{(n)}\}^N$, where $y^{(n)} \in \{0, 1\}$. Assuming N_1 tosses are heads and $N - N_1$ are tails, the Binomial distribution for heads is:

$$\begin{aligned} p(\mathbf{y}|\pi) &= \binom{N}{N_1} \prod_{n=1}^{N} \pi^{y^{(n)}} (1-\pi)^{1-y^{(n)}} \\ &= \binom{N}{N_1} \pi^{\sum_n y^{(n)}} (1-\pi)^{\sum_n 1-y^{(n)}} = \binom{N}{N_1} \pi^{N_1} (1-\pi)^{N-N_1} \end{aligned} \tag{5}$$

It is more convenient to work with log likelihood:

$$\mathcal{L}(\pi) \propto N_1 \log \pi + (N - N_1) \log(1-\pi)$$

Maximum likelihood estimation w.r.t. the Bernoulli rate π

Suppose we are interested in knowing the Bernoulli rate π. We can directly maximize the log likelihood w.r.t. π. We do this by solving $\frac{\partial \mathcal{L}}{\partial \pi} = 0$ for π:

$$\frac{\partial \mathcal{L}}{\partial \pi} = \frac{\partial}{\partial \pi} N_1 \log \pi + \frac{\partial}{\partial \pi}(N - N_1) \log(1 - \pi) = \frac{N_1}{\pi} - \frac{N - N_1}{1 - \pi}$$

where $N_1 = \sum_n y^{(n)}$. Solving $\frac{N_1}{\pi} - \frac{N-N_1}{1-\pi} = 0$ for π:

$$\frac{N_1}{\pi} - \frac{N - N_1}{1 - \pi} = 0 \implies N_1 - N_1 \pi = \pi N - \pi N_1 \implies \pi = \frac{N_1}{N}$$

Therefore, the maximum likelihood estimate of π is simply the proportion of the positive values.

Maximum likelihood estimation w.r.t. the logistic regression coefficients

Replacing the Bernoulli rate π with predicted probability $\hat{y}^{(n)} = \sigma(\mathbf{x}^{(n)}\mathbf{w} + w_0)$ by the logistic regression for each example:

$$\mathcal{L}(\mathbf{w}) = \log p(\mathbf{y}|\hat{\mathbf{y}}) = \sum_{n=1}^{N} y^{(n)} \log \hat{y}^{(n)} + (1 - y^{(n)}) \log(1 - \hat{y}^{(n)}) \tag{6}$$

We can solve $\frac{\partial \mathcal{L}}{\partial \hat{y}^{(n)}} = 0$ in Eq (3) for $\hat{y}^{(n)}$ and obtain trivial solution: $\hat{y}^{(n)} = y^{(n)}$.

Recall the inverse of the logistic function is the logit function:

$$\log \frac{\hat{y}^{(n)}}{1 - \hat{y}^{(n)}} = \mathbf{x}^{(n)}\mathbf{w} + w_0 \tag{7}$$

If $\mathbf{x}^{(n)}\mathbf{w} = 0 \,\forall n$, we have $\hat{y}^{(n)} = \sigma(w_0) \equiv \pi \,\forall n$ and

$$\log \frac{\hat{y}^{(n)}}{1 - \hat{y}^{(n)}} = \log \frac{\pi}{1 - \pi} = w_0 \quad \forall n \quad (\text{pop quiz: For a fair coin, what's } w_0\text{?}) \tag{8}$$

So the intercept (or more precisely the "bias" w.r.t. the coin) w_0 here captures the log odds of the prior probability. Note: $w_0 = \log \frac{\pi}{1-\pi}$ is not a MLE of w_0.

Maximum likelihood estimation w.r.t. the logistic regression coefficients

Our main interest is in $\mathbf{w}$. It is easy to see that maximizing this likelihood w.r.t. $\mathbf{w}$ is equivalent to minimizing the cross entropy (CE) since $CE = -\mathcal{L}(\mathbf{w})$:

$$\begin{aligned}
\mathcal{L}(\mathbf{w}) &= \log p(\mathbf{y}|\hat{\mathbf{y}}) \\
&= \sum_{n=1}^{N} y^{(n)} \log \hat{y}^{(n)} + (1 - y^{(n)}) \log(1 - \hat{y}^{(n)}) \\
&= -\left(\sum_{n=1}^{N} -y^{(n)} \log \hat{y}^{(n)} - (1 - y^{(n)}) \log(1 - \hat{y}^{(n)}) \right) \\
&= -J(\mathbf{w})
\end{aligned}$$

That is,

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \arg\min_{\mathbf{w}} J(\mathbf{w}) \tag{9}$$

Pop quiz: What do we need to change in the logistic regression algorithm in Slide 14 if we are maximizing the likelihood?

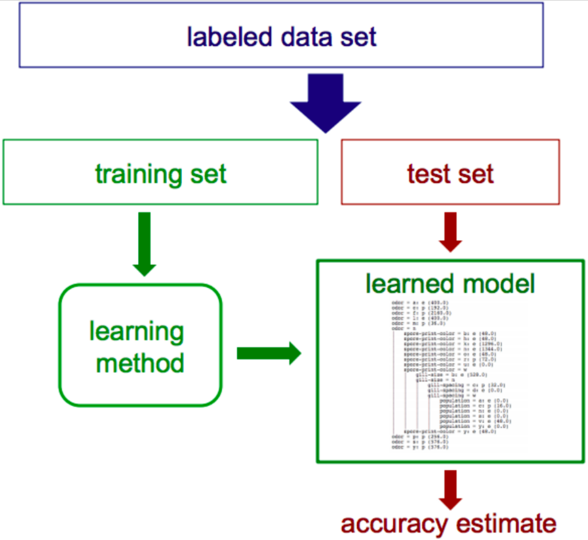
Outline

Titanic dataset

1. 'pclass' - passenger class (1 = first; 2 = second; 3 = third)
2. 'survived' - yes (1) or no (0)
3. 'sex' - sex of passenger (binary) ('male'=0 and 'female' = 1)
4. 'age' - age of passenger in years (float)
5. 'sibsp' - number of siblings/spouses aboard (integer)
6. 'parch' - number of parents/children aboard (integer)
7. 'fare' - fare paid for ticket (float)

	pclass	survived	sex	age	sibsp	parch	fare
0	1.0	1.0	1	29.0000	0.0	0.0	211.3375
1	1.0	1.0	0	0.9167	1.0	2.0	151.5500
2	1.0	0.0	1	2.0000	1.0	2.0	151.5500
3	1.0	0.0	0	30.0000	1.0	2.0	151.5500
4	1.0	0.0	1	25.0000	1.0	2.0	151.5500
...	...	...	...	...	...	...	...
1040	3.0	0.0	0	45.5000	0.0	0.0	7.2250
1041	3.0	0.0	1	14.5000	1.0	0.0	14.4542
1042	3.0	0.0	0	26.5000	0.0	0.0	7.2250
1043	3.0	0.0	0	27.0000	0.0	0.0	7.2250
1044	3.0	0.0	0	29.0000	0.0	0.0	7.8750

Recall the typical workflow to evaluate a classification model



Classifying survivor and non-survivor from Titanic (Colab)

Goal: For a given passenger, we want to predict whether he or she survived using the rest of the variables.

We split the data into 80% training and 20% testing

```
from sklearn import model_selection
import pandas as pd
from sklearn.preprocessing import StandardScaler

data = pd.read_csv('titanic.csv')

X = data.drop(["survived"], axis=1).values
y = data["survived"].values
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size = 0.2, random_state=1, shuffle=True)

# standardize training and test data separately (why?)
X_train = StandardScaler().fit(X_train).transform(X_train)
X_test = StandardScaler().fit(X_test).transform(X_test)
```

Logistic regression classification

```
# using our version (not sklearn)
logitreg = LogisticRegression(max_iters=1e3)
fit = logitreg.fit(X_train, y_train)
effect_size = pd.DataFrame(fit.w[:(len(fit.w)-1)]).transpose() # linear coefficents
effect_size.columns = data.drop(["survived"], axis=1).columns
```

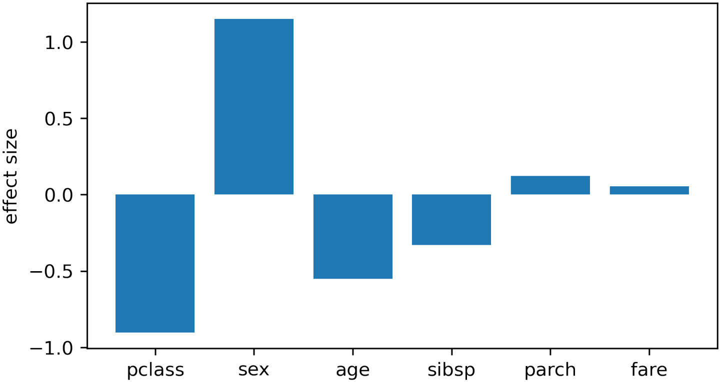
$$a = w_0 + w_{pclass}\mathbf{x}_{pclass} + w_{sex}\mathbf{x}_{sex} + w_{age}\mathbf{x}_{age} + w_{sidsp}\mathbf{x}_{sidsp} + w_{parch}\mathbf{x}_{parch} + w_{fare}\mathbf{x}_{fare}$$

$$\hat{y} = \frac{1}{1 + \exp(-a)}$$

- We train logistic regression on the training data: `logitreg.fit(X_train, y_train)`
- We can examine which variables are important in predicting survivor based on the linear coefficients b_j: `print(effect_size.to_string(index=False))` or visualize them as barplot (next slide).

Which variables are important in predicting survior?

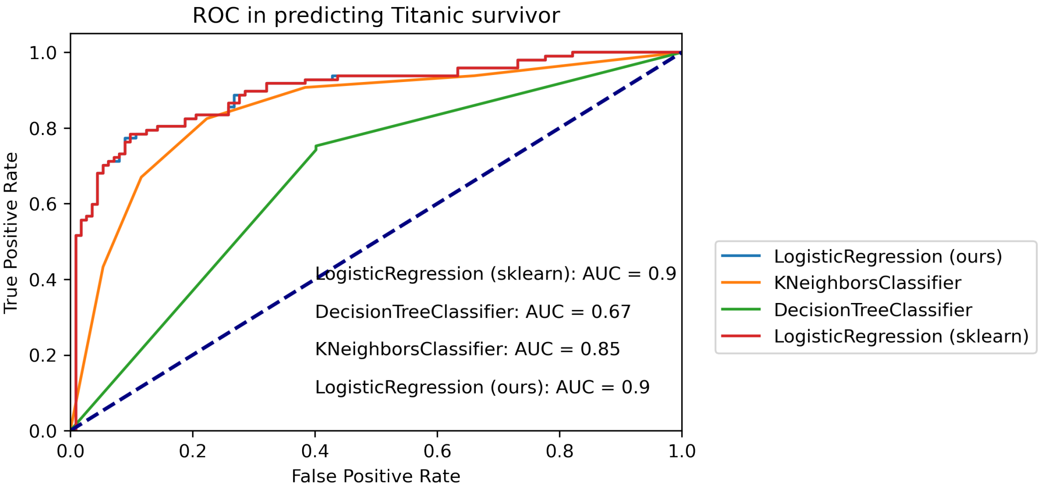
```
import matplotlib.pyplot as plt
plt.bar(list(effect_size.columns.values),
    effect_size.stack().tolist())
plt.ylabel("effect size")
plt.show()
```



Compare our logistic regression code with sklearn implementation

```python
from sklearn.linear_model import LogisticRegression as sk_LogisticRegression

# omitted some code due to space see colab for the full version

logitreg = LogisticRegression(max_iters=1e3)
fit = logitreg.fit(X_train, y_train)
y_test_prob = fit.predict(X_test)
fpr, tpr, _ = roc_curve(y_test, y_test_prob)
auroc = roc_auc_score(y_test, y_test_prob)
perf["LogisticRegression (ours)"] = {'fpr':fpr, 'tpr':tpr, 'auroc':auroc}


models = [KNeighborsClassifier(), DecisionTreeClassifier(), sk_LogisticRegression()]

for model in models:
    fit = model.fit(X_train, y_train)
    y_test_prob = fit.predict_proba(X_test)[:,1]
    fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
    auroc = roc_auc_score(y_test, y_test_prob)
    if type(model).__name__ == "LogisticRegression":
        perf["LogisticRegression (sklearn)"] = {'fpr':fpr,'tpr':tpr,'auroc':auroc}
    else:
        perf[type(model).__name__] = {'fpr':fpr,'tpr':tpr,'auroc':auroc}
```

ROC curve on Titanic survivor prediction



Summary

- Logistic regression
  - Logistic activation function: sigmoid
  - Cross-entropy (CE) loss
  - Gradient descent
- Probabilistic interpretation
  - Bernoulli distribution
  - Maximum likelihood estimation of Bernoulli is equivalent to minimizing CE loss
  - Recall in linear regression: MLE of Gaussian is equivalent to minimizing SSE loss
- Application and interpretation of logistic regression linear coefficients.
- **Model interpretability** is one of the major benefits of the linear models.