Lecture 6. Linear classification (part 1): Logistic Regression

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

September 20, 2022

Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

Probabilistic view of logistic regression

Application: Titanic surviver prediction

Summary

Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

Probabilistic view of logistic regression

Application: Titanic surviver prediction

Summary

Learning objectives

Understanding the following concepts

- linear classifiers
- logistic regression
    - model
    - loss function
- maximum likelihood view multi-class classification

Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

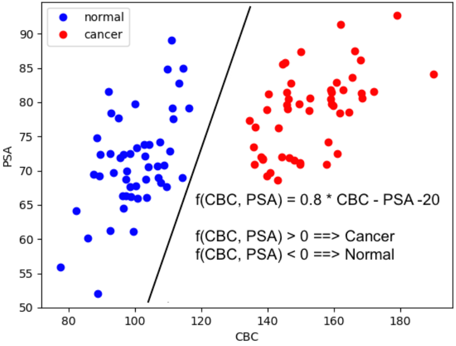
Probabilistic view of logistic regression

Application: Titanic surviver prediction

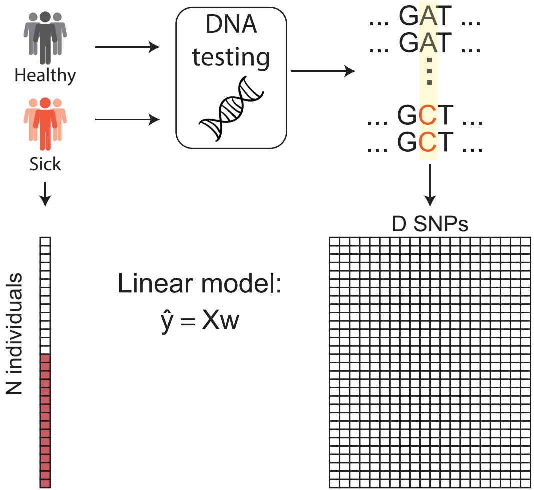
Summary

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.

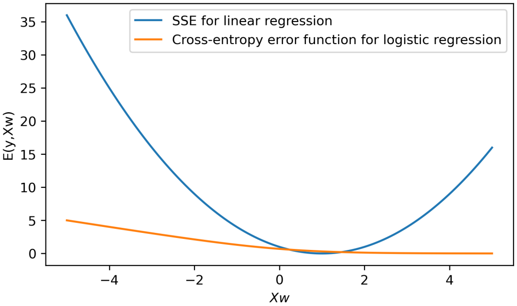


Linear regression for continuous response is not suitable for classification

For a binary classification problem, the target variable $y \in \{0, 1\}$.
Recall from Lecture 5, the cost function for linear regression is sum of squared error (SSE):

$$J(\mathbf{w}) = \sum_{n=1}^{N} (y^{(n)} - \hat{y}^{(n)})^2$$

where $\hat{y}^{(n)} = \sum_d w_d x_d^{(n)}$. However, $\hat{y}^{(n)} \in \mathbb{R}$, which means our prediction is unbounded.



Given that the true label $y = 1$, the SSE is 0 if and only if $\hat{y}^{(n)} = 1$. SEE increases even if $\hat{y}^{(n)}$ is highly positive (indicating that the model is confident about the true positive label). In contrast, another error function called **cross-entropy** decreases $\hat{y}^{(n)}$ becomes more positive and converges to 0 as $\hat{y}^{(n)} \to \infty$.

Logistic function

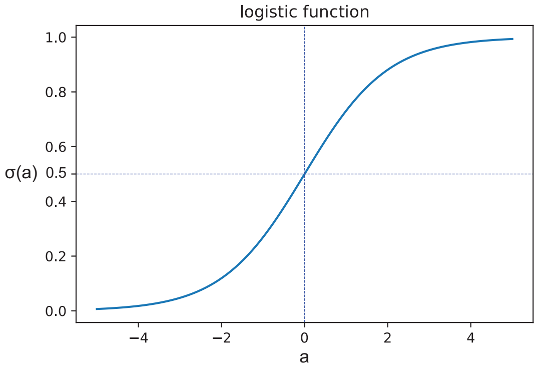
Logistic function transforms the real-value $\mathbf{a} = \mathbf{Xw} \in \mathbb{R}$ into $\hat{y} \in [0, 1]$, which can be interpreted as the *probability* being class 1.

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

The inverse of the logistic function is called **logit function**:

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

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 "neural" (i.e., either positive or negative). Therefore, $a = 0$ is the decision boundary.

Cross entropy loss functions

Given that $\hat{y} = 1/(1 + \exp(-\mathbf{xw}))$, we consider three candidate loss functions.

Direct loss is not differentiable so we cannot use gradient descent (Section 3):

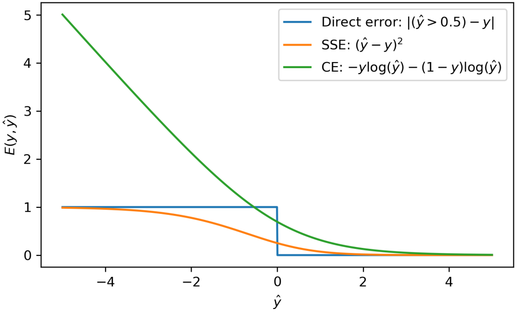
$$\mathcal{L}(\hat{y}, y) = |\mathbb{I}(\hat{y} > 0.5) - y|$$

The SSE loss is non-convex to optimize:

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

cross-entropy (CE) is convex in **w** and has probabilistic interpretation (Section 4)

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



Numerically accurate implementation of the CE loss using `np.log1p`

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

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

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

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

Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

Probabilistic view of logistic regression

Application: Titanic surviver prediction

Summary

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 $a = \mathbf{x}\mathbf{w}$. 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:

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

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

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

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:

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

Logistic regression training algorithm by gradient descent

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

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

Unlike in the linear regression case, where we have closed-form solution for **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 **w**. To update the logistic regression model, we perform **gradient descent** by *subtracting* the gradients from the existing weight *iteratively*:

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

- We do subtracting because we want to minimize the error function.
- We multiple the gradient by a learning rate $\alpha \in [0, 1]$ to avoid overshoot the optimal value

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 $||J(\mathbf{w}^{(t)}) - J(\mathbf{w}^{(t-1)})||_2) < \epsilon$ **then**

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

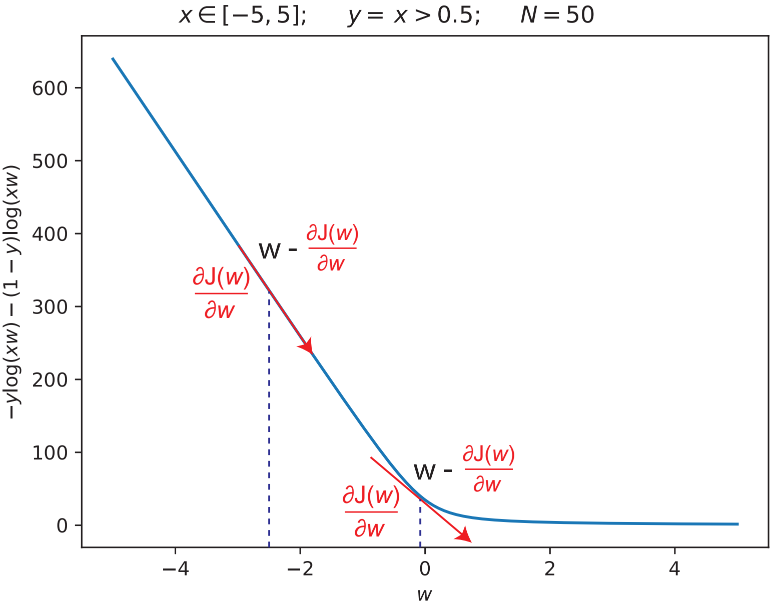
  end if

end for

Toy data

```
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

Gradient calculation by hand as shown above can be error-prone. We can verify the gradient as follow:

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

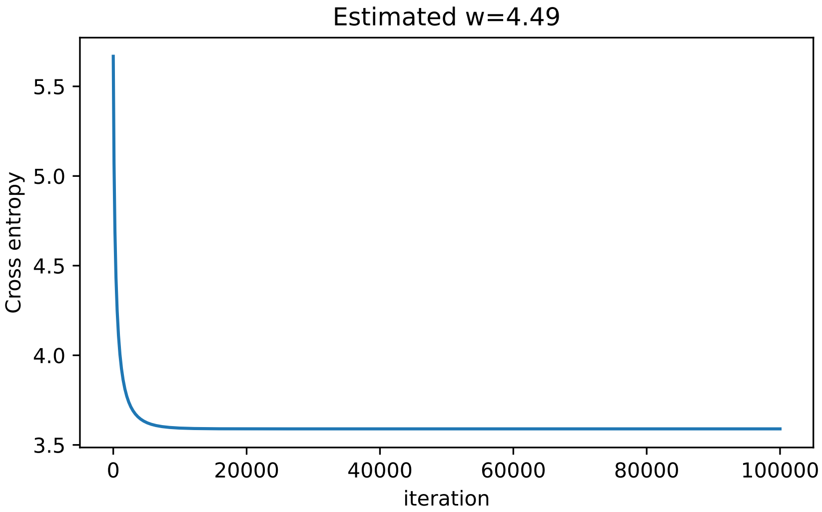
Python code for small perturbation test on a toy data

```
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) # hand calculated gradient

print(dw_cal)
print(dw_num)
print((dw_cal - dw_num)**2/(dw_cal + dw_num)**2)
```

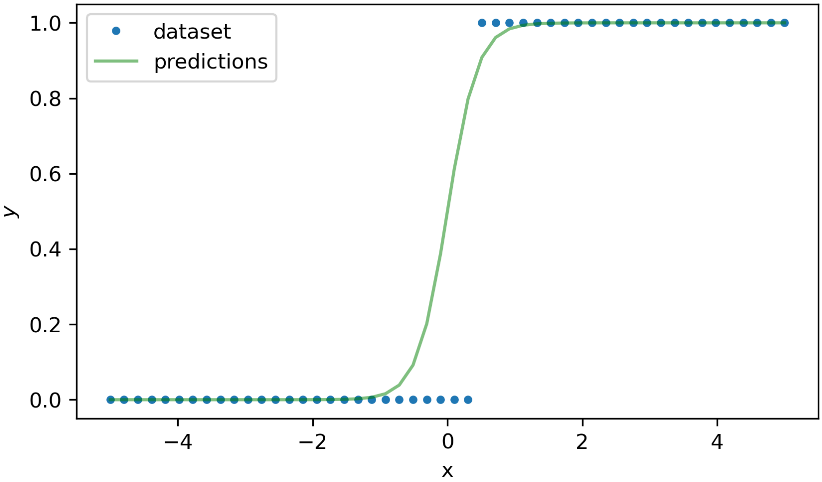
Verifying gradient calculation 2: Monitor error decreases at each iteration

```
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)))
    dw = np.sum((y_hat - y) * x)
    w = w - lr * dw
```

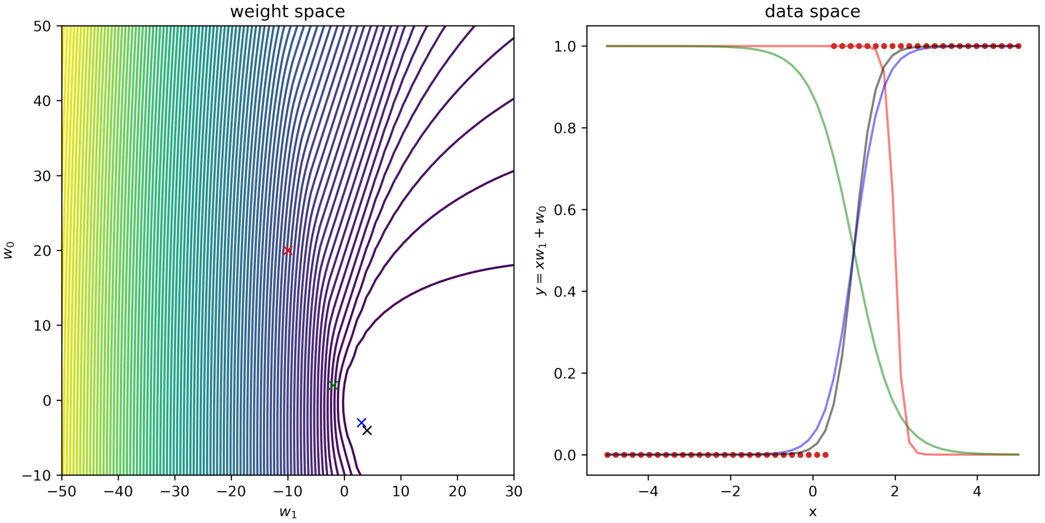
Verifying gradient calculation 2: Monitor error decreases at each iteration



Visualizing predictions on 1D data



Visualizing weights contour and their predictions



Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

Probabilistic view of logistic regression

Application: Titanic surviver prediction

Summary

Bernoulli distribution

A Bernoulli distribution has the following probability distribution function (PDF):

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

where π is the rate of $y = 1$. A common example used is coin toss. If the coin lands on its head $y = 1$; otherwise $y = 0$. A fair coin will have $\pi = 0.5$.

We can toss N times to get a dataset of $\mathcal{D} = \{y^{(n)}\}^N$, where $y^{(n)} \in \{0, 1\}$. Assuming the coin tosses are i.i.d., we can express the joint distribution of N tosses as the product of PDF or the *likelihood*:

$$p(\mathbf{y}|\pi) = \prod_{n=1}^{N} \pi^{y^{(n)}} (1-\pi)^{1-y^{(n)}} \tag{5}$$

It is more convent to work with logarithmic form of the likelihood or log likelihood because **the log of the products equal to the sum of the log**:

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

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 trying to solve for π by setting $\frac{\partial \mathcal{L}}{\partial \pi} = 0$:

$$\frac{\partial \mathcal{L}}{\partial \pi} = \frac{\partial}{\partial \pi} \sum_{n=1}^{N} y^{(n)} \log \pi + (1 - y^{(n)}) \log(1 - \pi) = \sum_{n=1}^{N} y^{(n)} \frac{\partial \log \pi}{\partial \pi} + (1 - y^{(n)}) \frac{\partial \log(1 - \pi)}{\partial \pi}$$

$$= \sum_{n=1}^{N} \frac{y^{(n)}}{\pi} - \frac{1 - y^{(n)}}{1 - \pi} = \frac{\sum_{n=1}^{N} y^{(n)}}{\pi} - \frac{\sum_{n=1}^{N} 1 - y^{(n)}}{1 - \pi} = \frac{N_1}{\pi} - \frac{N - N_1}{1 - \pi}$$

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:

$$\mathcal{L}(\mathbf{w}) = \log p(\mathbf{y}|\hat{\mathbf{y}}) = \sum_{n=1}^{N} y^{(n)} \log \hat{y}^{(n)} + (1 - y^{(n)}) \log(1 - \hat{y}^{(n)}) \tag{7}$$

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

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

$$\log \frac{\hat{y}^{(n)}}{1 - \hat{y}^{(n)}} = \mathbf{x}^{(n)}\mathbf{w} + w_0 \tag{8}$$

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

$$\log \frac{\hat{y}^{(n)}}{1 - \hat{y}^{(n)}} = \log \frac{\pi}{1 - \pi} = w_0 \quad \forall n \tag{9}$$

So the intercept b here captures the log odds of the prior probability.

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

However, 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 $CE = -\mathcal{L}(\mathbf{w})$:

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

That is,

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \arg\min_{\mathbf{w}} J(\mathbf{w}) \tag{10}$$

Outline

Objectives

Linear classifier

Learning logistic regression by gradient descent

Probabilistic view of logistic regression

Application: Titanic surviver prediction

Summary

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)

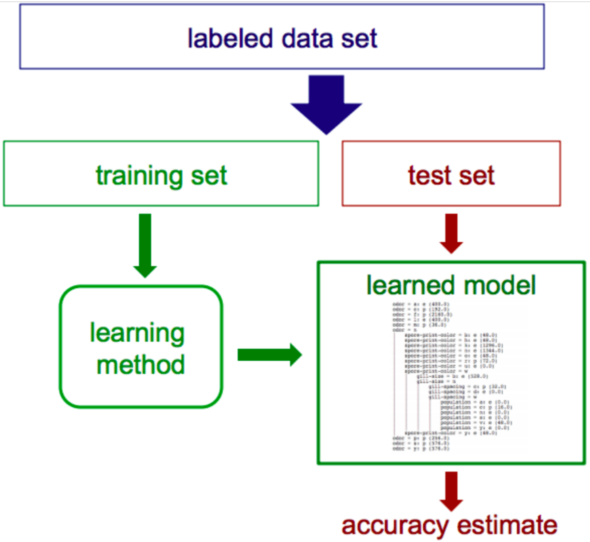
	pclass	survived	sex	age	sibsp	parch	fare
0	1.0	1.0	1	29.0000	0.0	0.0	211.3375
1	1.0	1.0	0	0.9167	1.0	2.0	151.5500
2	1.0	0.0	1	2.0000	1.0	2.0	151.5500
3	1.0	0.0	0	30.0000	1.0	2.0	151.5500
4	1.0	0.0	1	25.0000	1.0	2.0	151.5500
...	...	...	...	...	...	...	...
1040	3.0	0.0	0	45.5000	0.0	0.0	7.2250
1041	3.0	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

Classifying survivor and non-survivor from Titanic

Goal: For a given passenger, we want to predict whether he or she survive 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 normalize

data = pd.read_csv('data/LogisticRegression/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)

X_train = normalize(X_train)
X_test = normalize(X_test)
```

Recall the typical workflow to evaluate a classification model



Logistic regression classification

```
logitreg = LogisticRegression() # create an object (OOP)
fit = logitreg.fit(X_train, y_train)
effect_size = pd.DataFrame(fit.w[:(len(fit.w)-1)]).transpose() #
↪  linear coeffficents
effect_size.columns = data.drop(["survived"], axis=1).columns
print(effect_size.to_string(index=False))
#    pclass       sex       age     sibsp    parch     fare
# -1.051683  2.326165 -0.037907 -0.348339  0.151557  0.001107
```

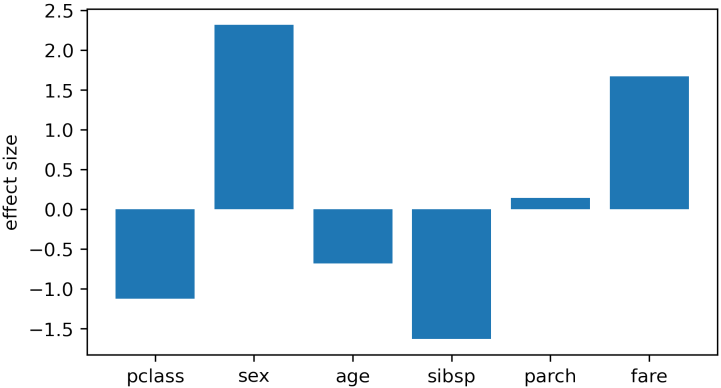
$$a = w_0 + w_{pclass}\mathbf{x}_{pclass} + w_{sex}\mathbf{x}_{sex} + w_{age}\mathbf{x}_{age} + w_{sidsp}\mathbf{x}_{sidsp} + w_{parch}\mathbf{x}_{parch}$$

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

- 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))`

Which variables are important in predicting survior?

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



Logistic regression prediction

Model training:

```
logitreg = LogisticRegression() # create an object (OOP)
fit = logitreg.fit(X_train, y_train)
```

Model prediction:

```
y_test_pred = fit.predict(X_test)
```

- We then apply the trained model `fit` to predict survivor: `y_train_pred=fit.predict(X_train)`, `y_test_pred=fit.predict(X_test)`
- Our prediction is binary 0 (not survived) or 1 (survived) based on whether the predicted probabilities are greater than 0.5.

Classification Accuracy

```python
# accuracy = correctly classified / total classified
acc_train = sum(y_train_pred==y_train)/len(y_train)
acc_test = sum(y_test_pred==y_test)/len(y_test)
print(f"train accuracy: {acc_train:.3f}; test accuracy:
 ↪  {acc_test:.3f}")
# train accuracy: 0.778; test accuracy: 0.823
```

- We then evaluate the prediction accuracy:

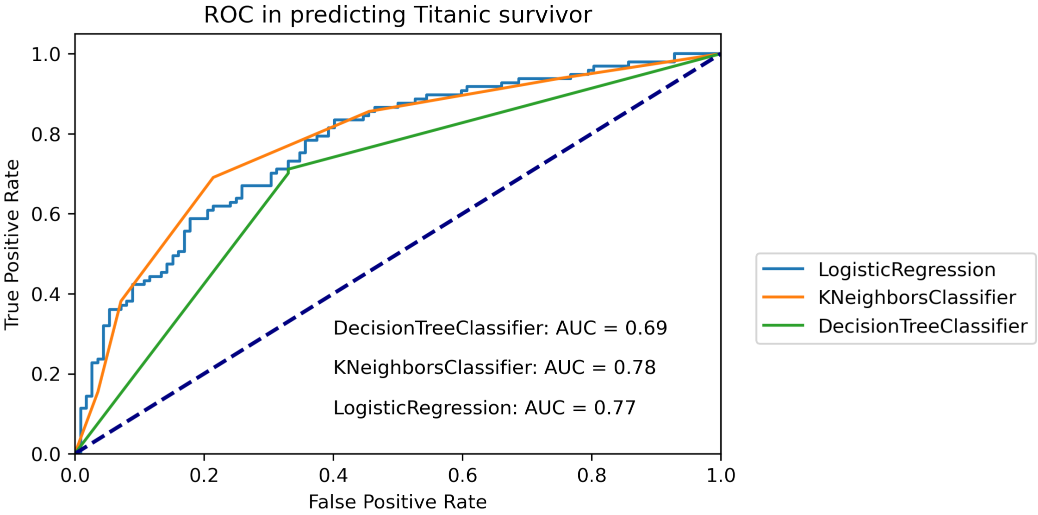
$$Accuracy = \frac{\text{Correctly classified passengers}}{\text{Total number of classified passengers}}$$

- The accuracy for predicting survivors in the training dataset (77.8%) is a bit lower than the accuracy in predicting survivors in the testing dataset (82.3%).
- In practice, the accuracy for the training tends to be higher than the accuracy on the testing
- When the training accuracy is much higher than the testing accuracy, the model is *overfitting* the data

Code to generate ROC curve and calculate AUROC

```
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

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

perf = {}
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)
    perf[type(model).__name__] = {'fpr':fpr,'tpr':tpr,'auroc':auroc}

i = 0
for model_name, model_perf in perf.items():
    plt.plot(model_perf['fpr'], model_perf['tpr'],label=model_name)
    plt.text(0.4, i+0.1, model_name + ': AUC = '+ str(round(model_perf['auroc'],2)))
    i += 0.1
```

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 estimate 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