Lecture 2 - Module 2. K-Nearest Neighbours

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

January 9, 2025

Outline

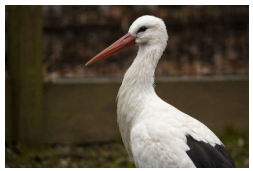
Objectives

- You may ask "Why start the course with KNN among many other methods"?
  Simple to understand. It allows us to quickly focus on learning the fundamental ML concepts including hyperparameter selection and evaluation.
- Variations of k-nearest neighbours for classification and regression
  - How to train a KNN for classification
  - How to train a KNN for regression
- The choice of K as hyper-parameter
- Computational complexity
- Pros and cons of KNN

Outline

Classifying new objects based on similar objects observed before

We guess type of unseen instances based on their similarity to our past experience. For instance,



Type of bird:
(a) stork
(b) pigeon
(c) penguin

Language:
(a) South Asian
(b) African
(c) Middle eastern

"Accretropin" is a
(a) protein
(b) drug
(c) rock band

Predicting housing price based on the price of neighbour or similar houses (example of regression)

Terminologies of nearest neighbour (KNN) classification

- Training: KNN does nothing but "remembering" the training data set

$$\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$$

where
  - $\mathcal{D}$: training set
  - $\mathbf{x}^{(n)}$: a D-dimensional input feature of the n^{th} training example
  - $y^{(n)}$: a categorical output of the n^{th} training example
  - N: number of training examples (i.e., the total number of $(\mathbf{x}^{(n)}, y^{(n)})$ pairs)
  - n: index of training example

  Because of that, KNN is known as examplar-based (or instance-based) learner or non-parametric method.
- Prediction: KNN predicts the label of a new data point based on labels from the K most similar examples in the training set (more details follow).

Pop quiz

1. Input dimension D? 4
2. Number of training examples N? 150
3. What's the value stored in $x_1^{(50)}$? 7.0
4. What's the value stored in $x[1, 3]$? 0.2

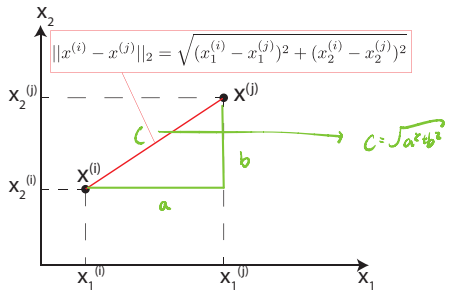
index	sepal length (x_0)	sepal width (x_1)	petal length (x_2)	petal width (x_3)	label (y)
0	5.1	3.5	1.4	0.3	Setosa
1	4.9	3.0	1.7	0.2	Setosa
⋮	⋮	⋮	⋮	⋮	⋮
50	7.0	3.2	4.7	1.4	Versicolor
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	Virginica

Pairwise distance

The key step in KNN in predicting new data point $\mathbf{x}^{(*)}$ is to find the K most similar examples in the training set. Let's first consider $K = 1$, i.e., we want to find the most similar example from the training set for the new data point $\mathbf{x}^{(*)}$. There are many ways to measure similarity. One common way is by *Euclidean distance*:

$$distance(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2 = \sqrt{\sum_{d=1}^{D} (x_d^{(i)} - x_d^{(j)})^2} \quad (1)$$

Euclidean distance between any pair of data points is the length of the line segment between them. The plot illustrates the Euclidean distance between point $\mathbf{x}^{(i)}$ and point $\mathbf{x}^{(j)}$ in two dimensions.



Making prediction using KNN algorithm ($K = 1$)

Predict the label of a new data point $\mathbf{x}^{(*)}$ based on labels from the most similar example (i.e., $K = 1$) in the training set in terms of *Euclidean distance*:

$$i = \underset{i \in \text{training set}}{\arg\min} \; \textit{distance}(\mathbf{x}^{(i)}, \mathbf{x}^{(*)}) = \underset{i \in \text{training set}}{\arg\min} \; ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2 \quad (2)$$

$$y^{(*)} = y^{(i)} \quad (3)$$

We can also use other distance or similarity functions.

For continuous input features, we can use:

- Manhattan distance: $\sum_{d=1}^{D} |x_d^{(i)} - x_d^{(j)}|$
- Cosine similarity: $\frac{\sum_{d=1}^{D} x_d^{(i)} x_d^{(j)}}{(\sqrt{\sum_d (x_d^{(i)})^2} \sqrt{\sum_d (x_d^{(j)})^2})}$

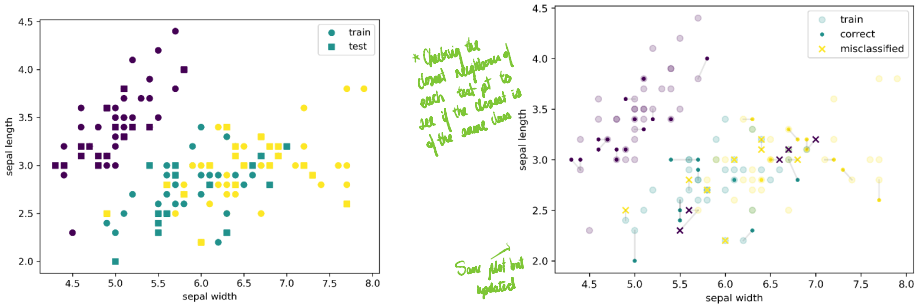
For discrete or categorical input features, we can use:

- Hamming distance: $\sum_{d=1}^{D} \mathbb{I}(x_d^{(i)} \neq x_d^{(j)})$, where $\mathbb{I}(b) = \begin{cases} 1 & \text{if b is True} \\ 0 & \text{if b is False} \end{cases}$

Note: For similarity, we need to choose the max: $i = \underset{i \in \text{training set}}{\arg\max} \; \textit{similarity}(\mathbf{x}^{(i)}, \mathbf{x}^{(*)})$

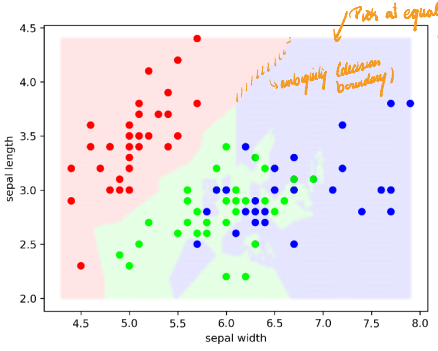
KNN algorithm ($K = 1$) applied to Iris dataset (See Colab for the code)

- $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^{N=150}$ 2D input: sepal length (x_1) and sepal width (x_2) (i.e., $x^{(n)} \in \mathbb{R}^2$)
- 3-class output: flower classes $y \in \{\textit{setosa}, \textit{versicolor}, \textit{virginica}\}$ (50 flowers each)
- Randomly split data into **training** (100 flowers) and **testing** (50 flowers)
- **Prediction accuracy** on the test set: $\frac{1}{50} \sum_{t=1}^{50} \mathbb{I}(\hat{y}^{(t)} = y^{(t)}) = 74\%$
- The right panel displays the testing data points connected with its nearest neighbour (big circles) from the training set. The correctly classified test data points are small circles and the incorrect predictions are crosses.



Decision boundaries created by the KNN (K=1) (see Colab)

- Voronoi diagram illustrates the decision boundary of KNN (K=1).
- It generates 200 values evenly spaced from min value to max value for sepal length $\in [2.0, 4.4]$ and sepal width $\in [4.3, 7.9]$.
- Therefore, there are 40,000 data points being predicted by the KNN based on the 100 training data points.
- The color shades were generated by predicting all input features



- The 40K points were colored based on the class label of their nearest neighbours from the training set.
- Each color corresponds to a flower class
- Try understand why some regions were assigned the specific color
- Is K=1 too flexible or too rigid?
- What does it look like if $K = N$?

Outline

KNN ($K \geq 1$) for discrete class predictions

We set the predicted label to be the most frequently occurring class among the K neighbours $c \in \{1, \ldots, C\}$ for a new data point $\mathbf{x}^{(*)}$:

$$\hat{y}^{(*)} = \arg\max_{c} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} \mathbb{I}(y^{(n)} = c) \quad (4)$$

c: class

$\mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})$: compute K-nearest neighbours of $\mathbf{x}^{(*)}$

sum all the neighbours that has the class c

→ then chose the highest sum

Here

- $\mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})$ denotes the K closest examples to $\mathbf{x}^{(*)}$ based on a distance function (e.g., Euclidean), and $\mathcal{D} = \{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^{N}$ is the training data
- When $K = 1$, the KNN reduces to finding *the most similar* example as above

*this is a general case

KNN ($K \geq 1$) for class probabilities

We calculate the class probabilities of each class $c \in \{1, 2, 3\}$ for a new data point $\mathbf{x}^{(*)}$:

$$p(y^{(*)} = c | \mathbf{x}^{(*)}) = \frac{1}{K} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} \mathbb{I}(y^{(n)} = c) \quad (5)$$

*Similar to (4), but we take the fraction of it

Here

- $\mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})$ denotes the K closest examples to $\mathbf{x}^{(*)}$ based on a distance function (e.g., Euclidean), and $\mathcal{D} = \{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^{N}$ is the training data
- When $K = 1$, the KNN reduces to finding *the most similar* example as above

For example, for $K = 3$,

- $\mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D}) = \{n_1, n_2, n_3\}$, which are the indices of the three most similar training examples of data point $\mathbf{x}^{(*)}$.
- Pop quiz: suppose $y^{(n_1)} = 1$, $y^{(n_2)} = 1$, $y^{(n_3)} = 3$, what are the probabilities for each class $p(y^{(*)} = c | \mathbf{x}^{(*)})$? Answer: 2/3, 0, 1/3

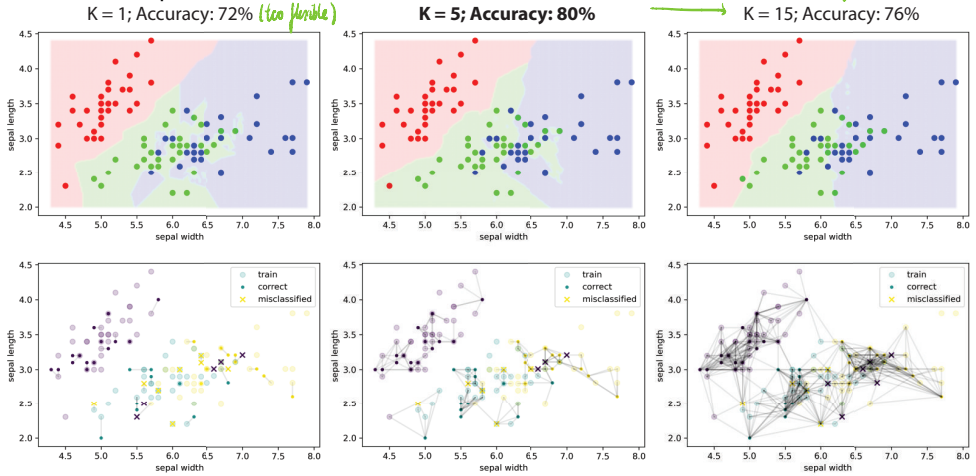
* $\frac{2}{3} + \frac{0}{3} + \frac{1}{3} = 1$

Outline

The choice of K

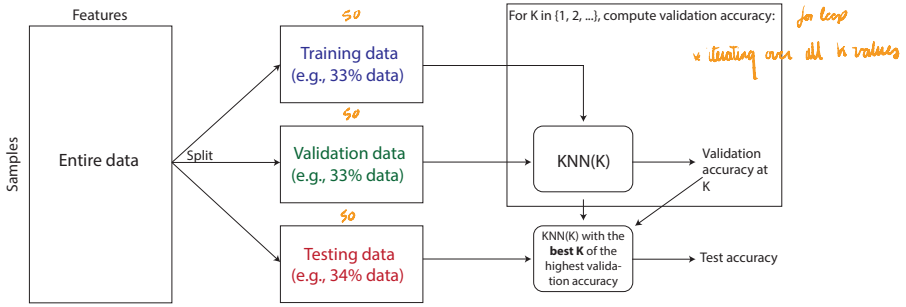
Choice of K is a **hyper-parameter** of the model, which are chosen empirically based on the model performance on a **held-out validation dataset**.

→ diagram can be too rigid



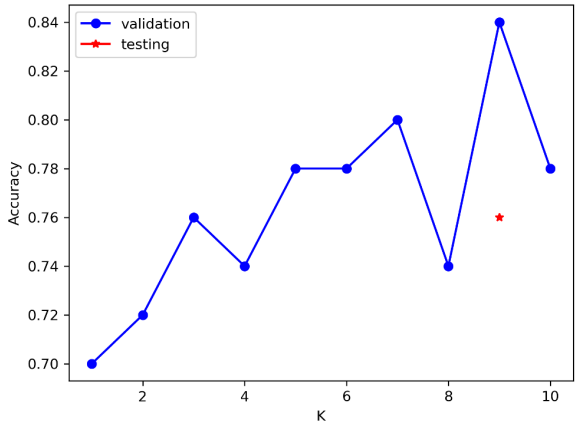
How to choose K

- The goal is to accurately predict unseen data that are not in the training set
- The accuracy can be approximated by the accuracy on the **test set**.
- To this end, we split the entire data into training, validation, and testing data.
- We use training data to "train" KNN, validation data to choose the hyperparameter from a finite set (i.e., $K \in \{1, 5, \ldots, \}$) that confers the highest validation accuracy, and finally we evaluate the chosen model on the testing data.



How to choose K using validation set (See Colab)

- To choose the best K, we further split the training data (N=100) into 50% training and 50% validation
- The red asterisk is the test accuracy based on the best $K = 9$.



Outline

Time complexity

- Assume we are using Euclidean distance $\sqrt{\sum_{d=1}^{D}(x_d^{(n)} - x_d^{(*)})^2}$
- For each data point in the training set, calculate the distance in $O(D)$ for a total of $O(ND)$
- Find the K points with smallest of distances in $O(NK)$ or more precisely $\sum_{k=0}^{K-1} N - k$ (i.e., first find the smallest, then the second smallest, and so on)
- The computational complexity for a single test query: $O(ND + NK)$

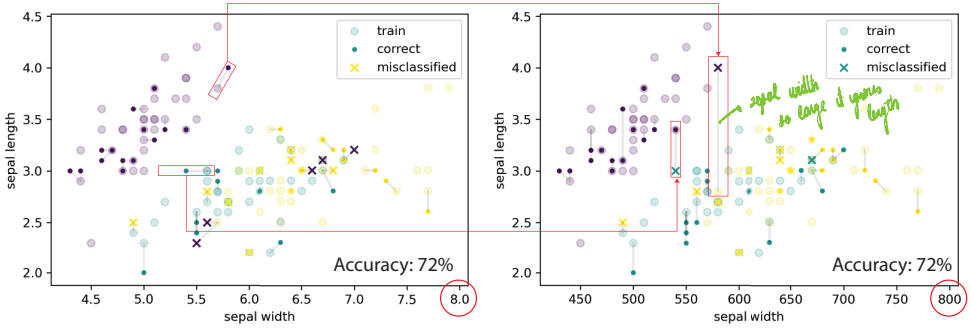
Improving the nearest neighbour search by k-d tree (optional)

Rough idea:

- We improve search of K nearest points by first building a tree by recursively clustering the training sets.
- To search the K nearest neighbours of a query point, we then go through the tree without looking at all of the N training points. $O(2^D + \log N)$

KNN is sensitive to feature scaling

- Euclidean distance: $distance(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2 = \sqrt{\sum_{d=1}^{D}(x_d^{(i)} - x_d^{(j)})^2}$
- Here we multiplied the sepal width by 100.
- As a result, the highlighted point was misclassified since the sepal width is almost the same and the sepal length difference is ignored due to much lower scale – it becomes negligible in the overall distance, and width becomes the main factor contributing to the distance between any two points



Standardizing features before running KNN prediction

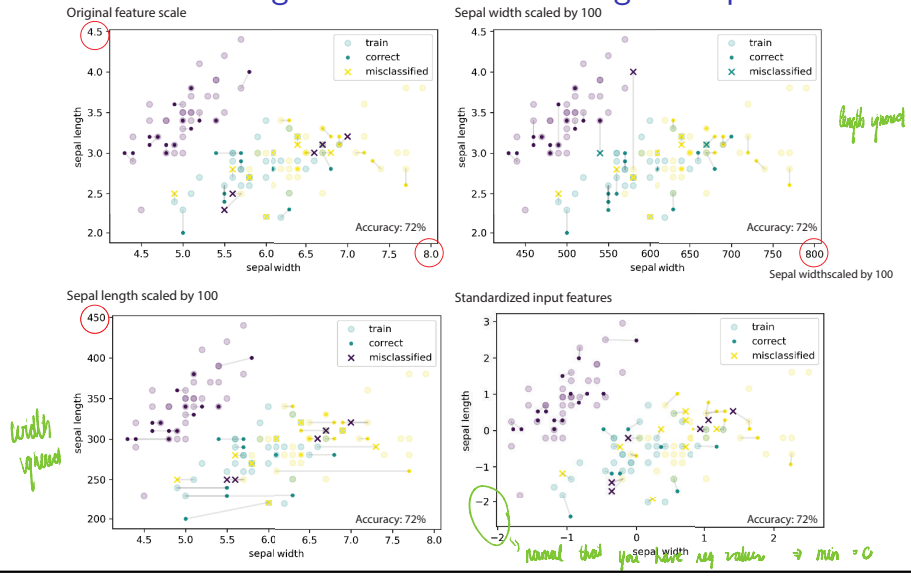
We can standardize each feature d as follows:

$$x_d^{(n)} = \frac{x_d^{(n)} - \bar{x}_d}{sd(x_d)}$$

where

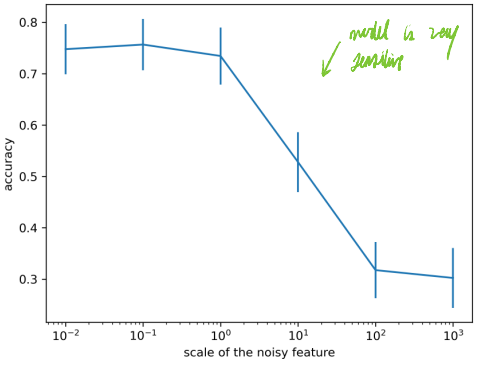
- $\bar{x}_d$ is the mean of feature d: $\bar{x}_d = \frac{1}{N}\sum_n x_d^{(n)}$
- $sd(x_d)$ is the standard deviation of feature d: $sd(x_d) = \sqrt{\frac{1}{N}\sum_n (x_d^{(n)} - \bar{x}_d)^2}$

Standardizing features before running KNN prediction

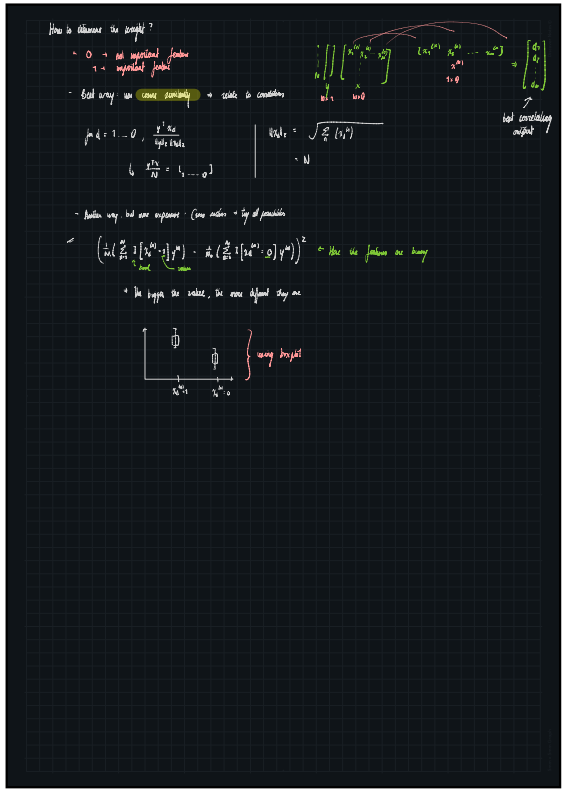


KNN is sensitive to random class-irrelevant features

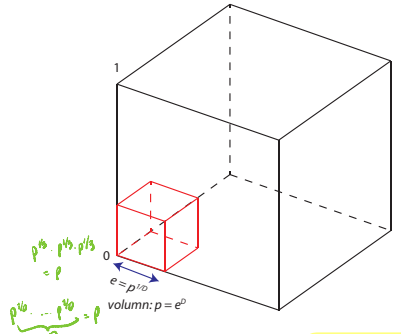
- Here we create a random noise feature sampled from a standard normal distribution $x_\epsilon^{(n)} \sim \mathcal{N}(0, 1)$
- Scale the noise feature by a factor of $s \in [.01, .1, 1, 10, 100, 1000]$
- Add the scaled noise feature to the two input features: $\mathbf{x}^{(n)} = [x_{\text{sepal length}}^{(n)}, x_{\text{sepal width}}^{(n)}, s \times x_\epsilon^{(n)}]$

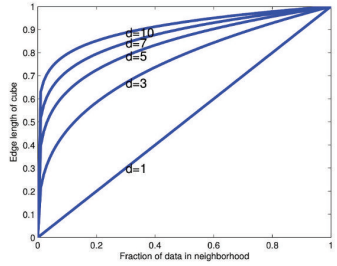


- As shown on the left, the larger the scaling factor s, the lower the prediction accuracy of the KNN (K=3).
- It is because our current KNN implementation is unable to distinguish feature importance
- Pop quiz: Can you think of a way to fix that? Answer: remove features with low correlation with the label.



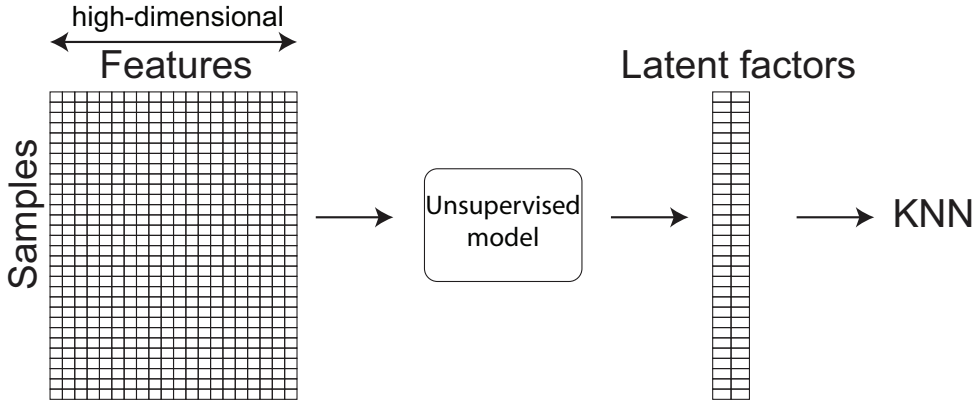
Curse of dimensionality (Murphy22 16.1.2)





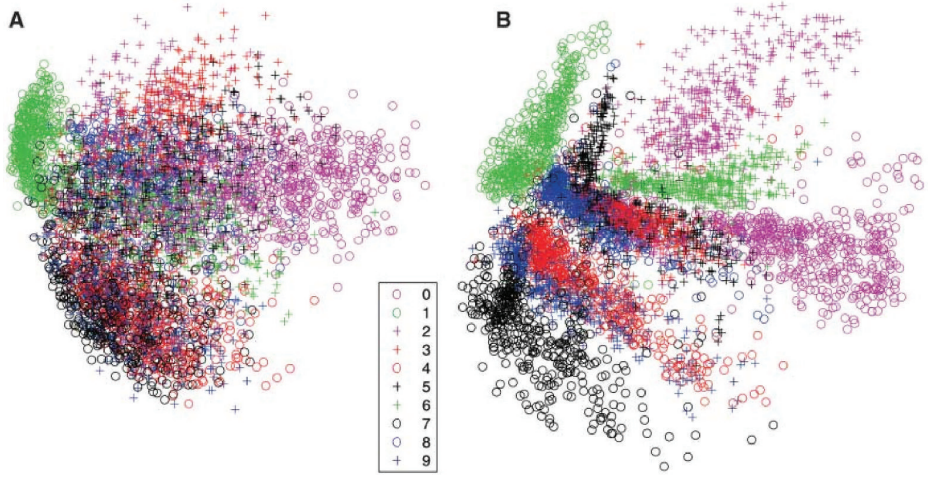
- Basic idea: the volume of space grows exponentially fast with dimension, so you might have to look quite far away in space to find your nearest neighbour.
- If $D = 10$ and we use $p = 10\%$ of the data points, the edge length $0.1^{1/10} = 0.8$. Even if we use 1% of the data, we still need an edge length of $0.01^{1/10} = 0.63$.
- Data points are sparsely distributed with high D and most pair of data points seem far from each other, which makes it difficult to find meaningful nearest neighbors.

Dimensionality reduction + KNN



We will discuss unsupervised learning in Module 7.

Real-world data are **not** uniformly distributed (Hinton *Science* 2006)



A. 500 MNIST digits of each class clustered by the first two PCs
B. Same digits clustered by the output from an autoencoder of architecture 784-1000-500-250-2

Weighted KNN

Weighted KNN weighs the contributions of each example by their distance from the test data:

$$p(y^{(*)} = c|\mathbf{x}^{(*)}) = \frac{1}{W_K(\mathbf{x}^{(*)}, \mathcal{D}))} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} similarity(\mathbf{x}^{(n)}, \mathbf{x}^{(*)}) \mathbb{I}(y^{(n)} = c) \quad (6)$$

where

- similarity is defined as either the inverse of the Euclidean distance or cosine similarity
- the normalization factor is defined as:

$$W_K(\mathbf{x}^{(*)}, \mathcal{D})) = \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} similarity(\mathbf{x}^{(n)}, \mathbf{x}^{(*)})$$

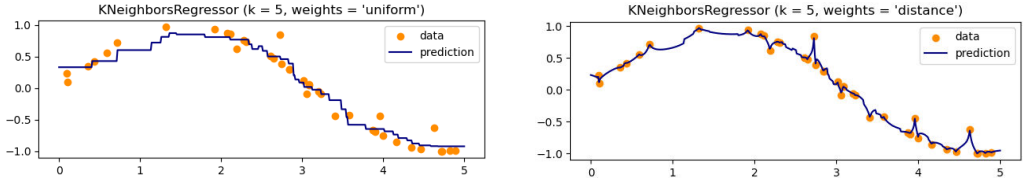
If $similarity(\mathbf{x}^{(n)}, \mathbf{x}^{(*)}) = 1 \forall n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})$, weighted KNN reduces to the unweighted KNN.

KNN regression

Adapting a classification KNN for regression is quite straightforward.

$$y^{(*)} = \frac{1}{K} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} y^{(n)} \quad \text{(Unweighted)}$$

$$y^{(*)} = \frac{1}{W_K(\mathbf{x}^{(*)}, \mathcal{D}))} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} similarity(\mathbf{x}^{(n)}, \mathbf{x}^{(*)}) y^{(n)} \quad \text{(Weighted)}$$



Predicting $y = sine(x)$ from a 1D feature x using Scikit-learn KNN

Summary

KNN performs classification/regression by finding similar instances in the training set:

- need a notion of distance, performance improves a lot with a better similarity measure e.g. see here
- Optimal number of neighbours (i.e., K) is chosen by the validation performance
- Weighted KNN weighs the neighbours' contributions by their distance from the test data point

KNN is a non-parametric method and a lazy learner:

- non-parametric: our model has no parameters (in fact the training data points are model parameters)
- Lazy, because we don't do anything during the training
  - test-time complexity grows with the size of the training data, as well as space complexity (store all data)
  - good performance when we have lots of data, see here
  - KNN is sensitive to curse of dimensionality, feature scaling and noise

Lecture 3 - Module 1.2: Model evaluation (part 1)

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

January 14 & 16, 2025

Outline

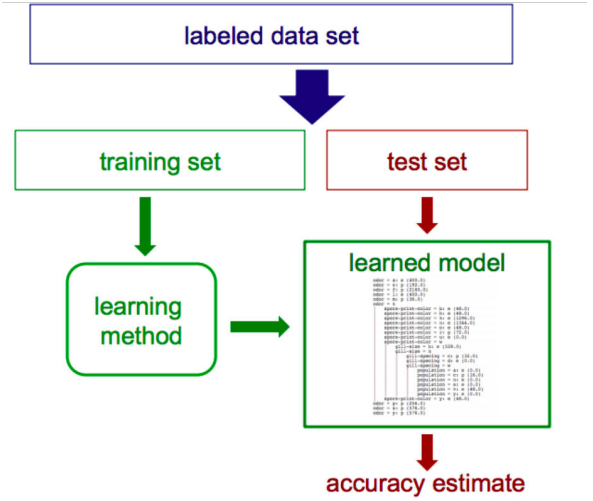
Learning objectives

Understanding the following concepts

- Evaluating generalization performance
- Confusion table
- Receiver operating characteristics (ROC)

Outline

A typical workflow to evaluate a classification model



Generalization error (or generalization accuracy)

- What we really care about is the performance of the model on *new data*.
- In other words, we want to see how our model *generalizes* to unseen data.
- An assumption that justifies deployment of our model on unseen data is the fact that our training and unseen data come from the **same** distribution.
- In fact very often we assume that there exists some distribution $p(x, y) = p(y|x)p(x)$ over our features and labels, such that our training data is composed of independent samples from the same distribution – that is $x^{(n)} \sim p_x$ and $y^{(n)} \sim p_{y|x}$ for all $x^{(n)}, y^{(n)} \in \mathcal{D}$ (i.e., the data are *i.i.d.*).
- We assume that unseen data are also samples from *the same* distribution.

Generalization error is the **expected error** of our model $f : x \mapsto y$ under this distribution:

$$Err(f) = \mathbb{E}_{x,y\sim p}[\ell(f(x), y)].$$

Here ℓ is some *loss function* such as classification error $\ell(y, \hat{y}) = \mathbb{I}(y \neq \hat{y})$ or squared loss $\ell(y, \hat{y}) = (y - \hat{y})^2$ that we often use in regression.

Test set

- Unfortunately, we don't have access to the true data distribution, we only have *samples* from the distribution.
- We can estimate the generalization error by setting aside a portion of our dataset that **we do not use in any way in learning or selecting the model**.
- This part of the dataset is called the **test set**. Let's use $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ to this partitioning of our original dataset $\mathcal{D}$.

The **test error** is

$$\widehat{Err}(f) = \mathbb{E}_{x,y\sim\mathcal{D}_{\text{test}}}[\ell(f(x), y)] = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{x,y\in\mathcal{D}_{\text{test}}} \ell(f(x), y).$$

where $|\mathcal{D}_{\text{test}}|$ is the cardinality of the test set $\mathcal{D}_{\text{test}}$ (i.e., the number of test samples).

Prostate cancer prediction problem

Suppose you want to learn to predict if a person has a prostate cancer based on two easily-measured variables obtained from blood sample: Complete Blood Count (CBC) and Prostate-specific antigen (PSA). We have collected data from patients known to have or not have prostate cancer:

CBC	PSA	Status
142	67	Normal
132	58	Normal
178	69	Cancer
188	46	Normal
183	68	Cancer
...		

Goal: Train classifier to predict the class of new patients, from their CBC and PSA.

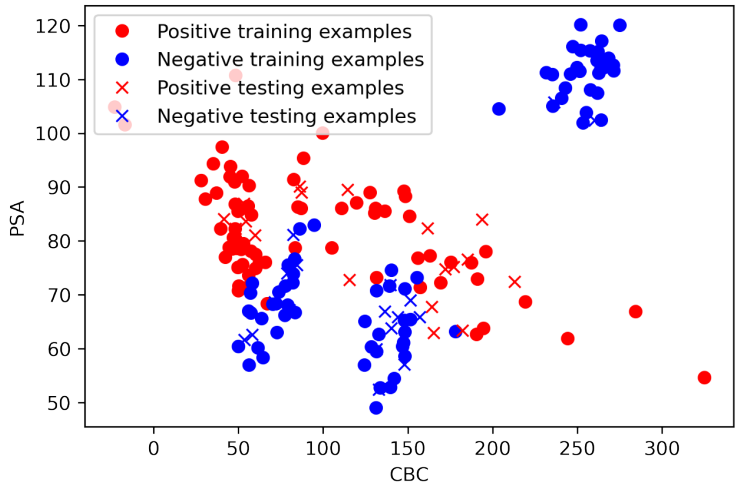
Split the dataset into training and testing datasets

We split the data into 80% training and 20% testing. In Scikit-learn, this is easy to do:

```
1 from sklearn import model_selection
2
3 X_train, X_test, y_train, y_test =
  ↪  model_selection.train_test_split(X, y, test_size = 0.2,
  ↪  random_state=1, shuffle=True)
```

- `random_state` set to a fixed number for reproducibility
- `shuffle` by default is `True` to randomly permute the orders of the rows to avoid splitting examples of the same class into training or testing set. For example, if rows are ordered by classes.

Prostate cancer data



Model prediction

Model training:

```
1 from sklearn.neighbors import KNeighborsClassifier
2 knn = KNeighborsClassifier() # n_neighbors=5 (default)
3 fit = knn.fit(X_train, y_train)
```

} trains

Model prediction on the test data:

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

- Our prediction is binary 0 (healthy) or 1 (cancer) based on whether the predicted probabilities are greater than 0.5.

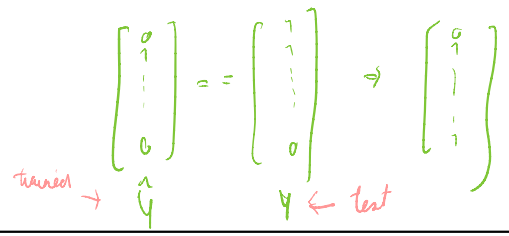
Classification Accuracy

We then evaluate the prediction accuracy:

$$\textit{Accuracy} = \frac{\text{Correct predictions}}{\text{Total number of predictions}}$$

```
1 acc_test = np.sum(y_test_pred==y_test)/len(y_test) % 0.975
```

test prediction accuracy



Outline

True/false positives and negatives

True positive (TP)
Positive example that is predicted to be positive

- A person who is predicted to have cancer and in fact has cancer

False positive (FP)
Negative example that is predicted to be positive

- A person who is predicted to have cancer but in fact is healthy

True negative (TN)
Negative example that is predicted to be negative

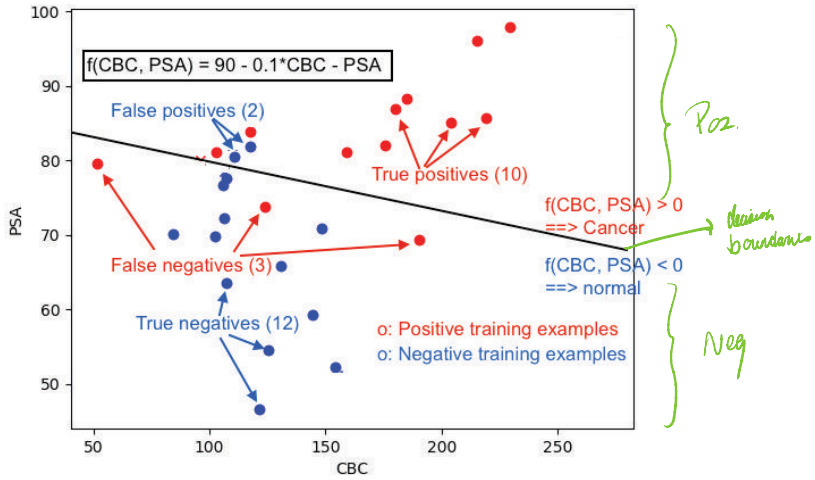
- A person who is predicted to be healthy and in fact is healthy

False negative (FN)
Positive example that is predicted to be negative

- A person who is predicted to be healthy but in fact has cancer

Classification errors

Here: TP = 10, TN = 12, FP = 2, FN = 3.



Confusion matrix

Confusion matrix: A table of counts for TPs, FPs, TNs, and FNs

	Predicted negative	Predicted positive
Actual negative	TN = 12	FP = 2
Actual positive	FN = 3	TP = 10

In scikit-learn, we can get the confusion matrix for the KNN by:

```
from sklearn.metrics import confusion_matrix

X_train, X_test, y_train, y_test = \
train_test_split(X, y, test_size=0.5, random_state=100)

knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
y_test_pred = knn.predict(X_test)

cm = confusion_matrix(y_test, y_test_pred)
```


True and false positive rates

At a specific threshold, we can calculate TPR and FPR:

True positive rate (TPR) (aka sensitivity)

The proportion of positive examples that are predicted to be positive

- e.g., fraction of cancer patients who are predicted to have cancer.

$$TPR = \frac{TP}{TP + FN}$$

False positive rate (FPR) (aka 1 - specificity)

The proportion of negative examples that are predicted to be positive

- e.g., fraction of non-cancer patients who are predicted to have cancer.

$$FPR = \frac{FP}{FP + TN}$$

Computing TPR and FPR from the confusion table

	Predicted negative	Predicted positive
Actual negative	TN = 12	FP = 2
Actual positive	FN = 3	TP = 10

$$TPR = \frac{TP}{TP + FN} = \frac{10}{10 + 3} = 77\%$$

$$FPR = \frac{FP}{FP + TN} = \frac{2}{2 + 12} = 14\%$$

True/false positive rates (pop quiz)

	predicted negative	predicted positive
actual negative	1	2
actual positive	3	7

$$TPR = \frac{TP\ 7}{TP_7 + FN_3} = \frac{?}{? + ?} = ?\ \ 0.7$$

$$FPR = \frac{FP\ 2}{FP_2 + TN_1} = \frac{?}{? + ?} = ?\ \ 0.66$$

Outline

A classification model often produces probabilities instead of hard decision

We can express our *uncertainty* via *the class probabilities*:

$$p(y^{(*)} = c|\mathbf{x}^{(*)}) = \frac{1}{K} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} \mathbb{I}(y^{(n)} = c)$$

In binary classification (i.e., cancer or normal), we can reduce the formula to predicting only the positive class:

$$p(y^{(*)} = 1|\mathbf{x}^{(*)}) = \frac{1}{K} \sum_{n \in \mathcal{N}_K(\mathbf{x}^{(*)}, \mathcal{D})} \mathbb{I}(y^{(n)} = 1)$$

Classification threshold: how to decide who are cancer patients?

- By default, `fit.predict(X_test)` uses 0.5 as threshold.

```
if pred_prob > 0.5:
    cancer = 1
else:
    cancer = 0
```

- This results in the following predictions:

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.1	0	0
1	0.4	0	0
2	0.5	0	1
3	0.8	1	1

- What accuracy do we get with a different threshold say 0.6?
- Can we evaluate the model using *all* thresholds?

True/false positive rates (pop quiz)

What will be the TPR and FPR if **the threshold is 1**?

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.1	0	0 ✓
1	0.4	0	0 ✓
2	0.5	0	1 ✗
3	0.8	0	1 ✗

	PN	PP
AN	2	0
AP	2	0

True positive rate (TPR):

$$TPR = \frac{TP}{TP+FN} = 0$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP+TN} = 0$$

True/false positive rates (pop quiz)

What will be the TPR and FPR if **the threshold is 0.6**?

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.1	0	0 ✓
1	0.4	0	0 ✓
2	0.5	0	1 ✗
3	0.8	1	1 ✓

	PN	PP
AN	2	0
AP	1	1

True positive rate (TPR):

$$TPR = \frac{TP}{TP+FN} = \frac{1}{1+1} = 0.5$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP+TN} = 0$$

True/false positive rates (pop quiz)

What will be the TPR and FPR if **the threshold is 0.3**?

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.1	0	0 ✓
1	0.4	1	0 ✗
2	0.5	1	1 ✓
3	0.8	1	1 ✓

	PN	PP
AN	1	1
AP	0	2

True positive rate (TPR):

$$TPR = \frac{TP}{TP+FN} = \frac{2}{2+0} = 1$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP+TN} = \frac{1}{1+1} = 0.5$$

True/false positive rates (pop quiz)

What will be the TPR and FPR if **the threshold is -1**?

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.1	1	0
1	0.4	1	0
2	0.5	1	1
3	0.8	1	1

	PN	PP
AN	0	2
AP	0	2

Nothing predicted negative

True positive rate (TPR):

$$TPR = \frac{TP}{TP + FN} = 1$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP + TN} = 1$$

*Highest sensitivity, but also highest FPR (predicted everything to be positive)

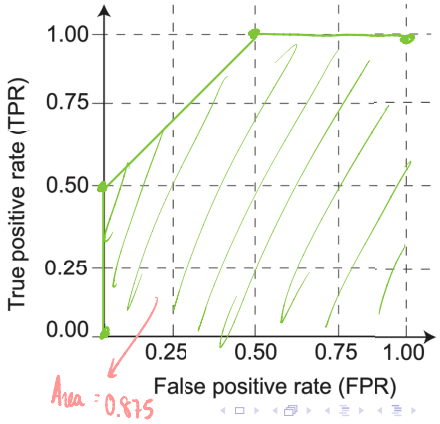
Receiver Operating Characteristic (ROC) curve

- We can create a table for TPR and FPR at each Threshold.
- ROC curve plots TPR (y-axis) versus FPR (x-axis)
- *Area under the curve (AUC)* is a metric common used to evaluate the model.

Threshold	TPR	FPR
1	0	0
0.6	0.5	0
0.3	1	0.5
-1	1	1

Consider three extreme cases:

1. What does ROC look like for a dummy model that predicts everything 0.5?
2. What does ROC look like for a perfect model?
3. What does ROC look like for a model opposite to perfect?



Dummy model for thresholds -1, 0.3, 0.6, 1

patient index	$p(y=1\|x)$	pred_label	true_label
0	0.5	1 1 0 0	0
1	0.5	1 1 0 0	0
2	0.5	1 1 0 0	1
3	0.5	1 1 0 0	1

	PN	PP
AN		
AP		

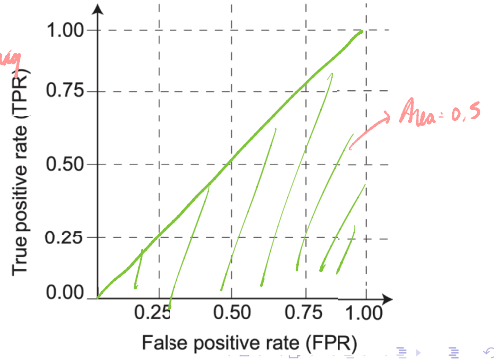
True positive rate (TPR):

Predict everything pos or neg

$$TPR = \frac{TP}{TP + FN} = 1.0$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP + TN} = 1.0$$



Perfect model for thresholds -1, 0.3, 0.6, 1

patient index	$p(y=1\|x)$	pred_label	true_label
0	0	1 0 0 0	0
1	0	1 0 0 0	0
2	1	1 1 1 0	1
3	1	1 1 1 0	1

	PN	PP
AN		
AP		

True positive rate (TPR):

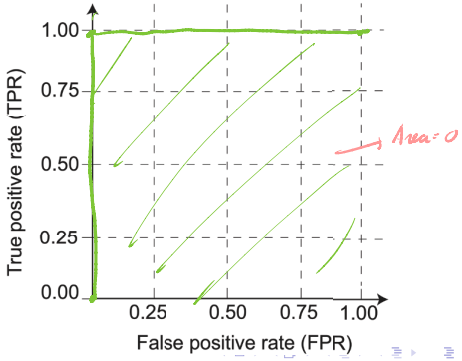
TPR=1 FPR=0

$$TPR = \frac{TP}{TP + FN} = 1.1.0$$

False positive rate (FPR):

$$FPR = \frac{FP}{FP + TN} = 1.0.1$$

⇒ Example of a Perfect model



Opposite to perfect model for thresholds -1, 0.3, 0.6, 1

patient index	$p(y=1\|x)$	pred_label				true_label
0	1	1	1	1	0	0
1	1	1	1	1	0	0
2	0	1	0	0	0	1
3	0	1	0	0	0	1

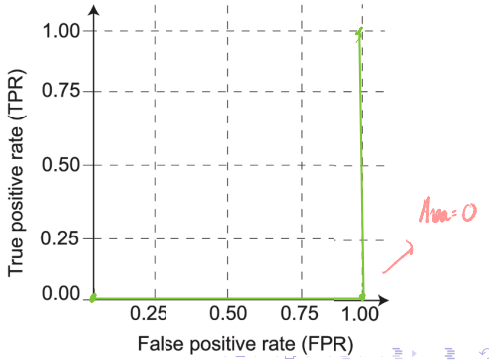
	PN	PP
AN		
AP		

True positive rate (TPR):

$$TPR = \frac{TP}{TP + FN} = 1,\ 0,\ 0$$

False positive rate (FPR):

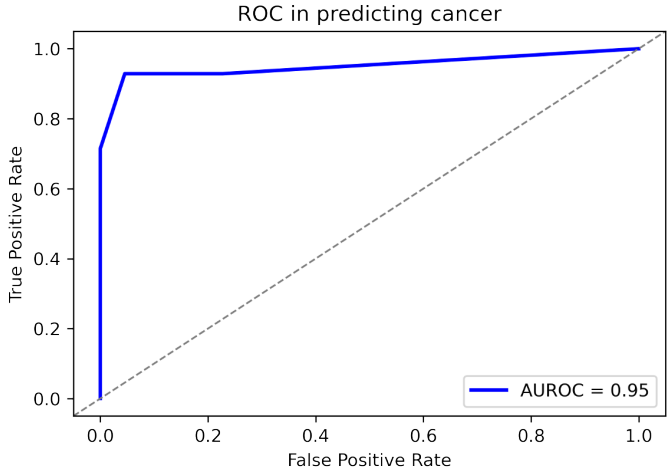
$$FPR = \frac{FP}{FP + TN} = 1,\ 1,\ 0$$



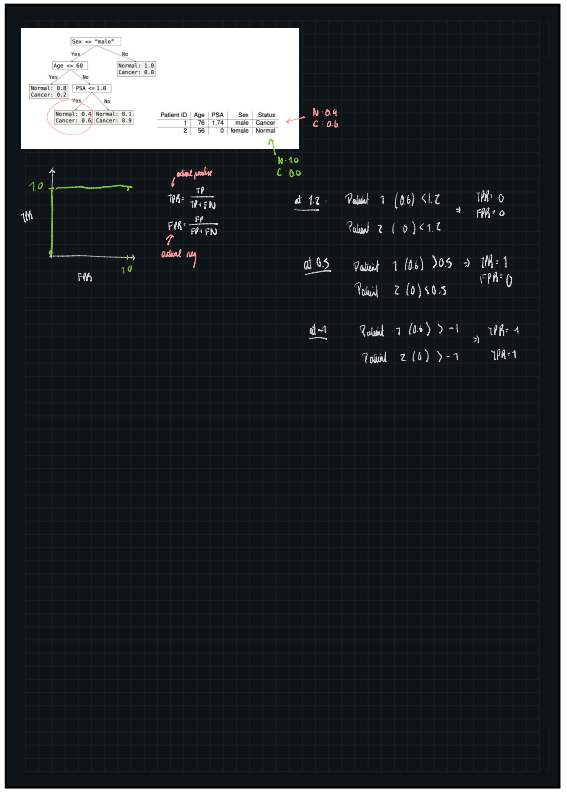
Computing ROC and AUROC using Scikit-learn function (Colab)

```
1  from sklearn.metrics import roc_curve, roc_auc_score
2
3  knn = KNeighborsClassifier()
4  knn.fit(X_train, y_train)
5  y_test_prob = knn.predict_proba(X_test)[:,1] # column 0 is healthy,
   ↪  column 1 is cancer
6  fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
7  roc_auc = roc_auc_score(y_test, y_test_prob)
8  plt.clf()
9  plt.plot(fpr, tpr, "b-", lw=2, label="AUROC = %0.2f"%roc_auc)
10 plt.axline((0, 0), (1, 1), linestyle="--", lw=1, color='gray')
11 plt.xlabel('False Positive Rate')
12 plt.ylabel('True Positive Rate')
13 plt.title('ROC in predicting cancer')
14 plt.legend(loc="best")
```


The resulting ROC



Need to have one of these plots for A1



Lecture 4 - Module 1.3 Model evaluation (part 2)

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

January 16, 2025

Outline

Learning objectives

Understanding the following concepts

- Cross-validation
- Method comparison
- Precision-recall curve

Outline

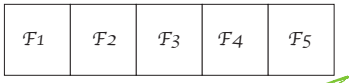
K-fold Cross Validation

- In the example from the last lecture, we split the data into training and testing
- Suppose the split is half-half, we train the model using only half of the data and evaluate the model using the other half:

- This is quite wasteful. How can we evaluate our model on *every data point* while training on the rest of the data points?
- Answer: K-fold cross-validation

Five-fold cross validation

Step 1. Randomly split the data $\mathcal{D}$ into 5 folds

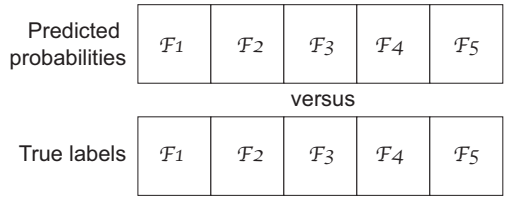


Step 2. Training and prediction

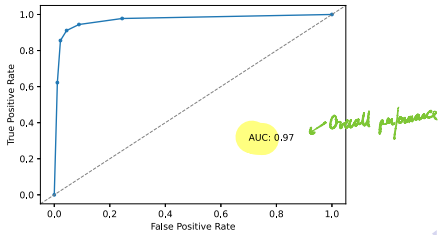
- How many times each data point is trained?
- Answer: 4 times
- How many times each data point is predicted?
- Answer: 1 * only 1 data pt used for testing

Evaluate on all K folds

Step 3. Evaluate predictions on all 5 folds by ROC



ROC curve of KNN predicted on ALL data points



Cross validation in Python scikit-learn (Colab)

```
def cross_validate(model, X_input, Y_output):
    kf = KFold(n_splits=5, random_state=1, shuffle=True)
    true_labels = np.array([0] * X_input.shape[0])
    pred_scores = np.array([0.0] * X_input.shape[0])
    for train_index, test_index in kf.split(X_input):
        model.fit(X_input[train_index], Y_output[train_index])
        true_labels[test_index] = Y_output[test_index]
        pred_scores[test_index] =
        ↪ model.predict_proba(X_input[test_index])[:,1]
    return true_labels, pred_scores
true_labels,pred_scores = cross_validate(model, X, y)
```

↳ take second column and store in pred_scores

Outline

Method comparisons

- There are many machine learning methods implemented in scikit-learn
- How do we know which one performs the best on *our data set*?
- To get the answer, we will need to compare these methods using cross validation
- Let's compare three machine learning methods namely
  - K-nearest neighbours (KNN)
  - Decision tree classifier (DT) (Module 3)
  - Logistic regression (LR) (Module 4.2)
- Note: for each method (or class), we create an *object* of the method using their initializer method defined under that class
- Training and prediction follows the *generic* syntax

Method comparisons using scikit-learn (Colab)

```
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:
    model_name = type(model).__name__
    print(model_name)
    label,pred = cross_validate(model, X, y)
    fpr, tpr, thresholds = roc_curve(label, pred)
    auc = roc_auc_score(label, pred)
    perf[model_name] = {'fpr':fpr,'tpr':tpr,'auc':auc}
```

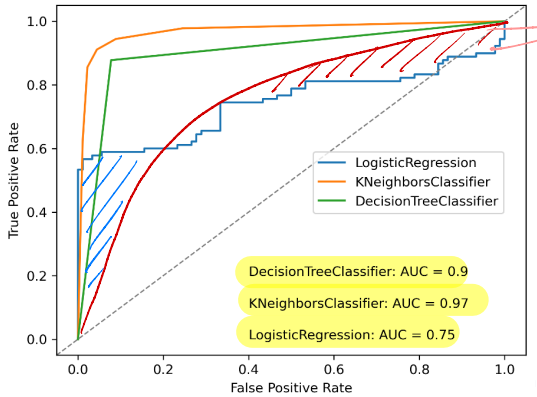

Plot the ROC curves for all method in one plot

```
import matplotlib.pyplot as plt

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, model_name + ': AUC = '+
             str(round(model_perf['auc'],2)))
    i += 0.1

plt.legend(loc='upper center',
           bbox_to_anchor=(0.75, 0.5))
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")

plt.savefig('roc_multimethods.eps')
```

make the model_name use on diff lines

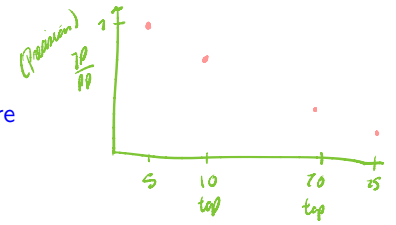
ROC curves and AUC for all of the four methods

- KNN (K=5) performs the best with 0.97 AUC
- DT achieves 0.85 AUC
- LR did worse (AUC = 0.73) because our data are not linearly separable
- In contrast, DT and KNN are non-linear methods



When you have the same Area under ROC?
→ You care more about better TPR among top predictions
→ 4 > 3
4 Precision ↑
① KN
② Decis
③ Log.
} Non-linear, so perf higher
The current data is non-linearly separable

Outline



⇒ Top K precision curve decreases

Sensitivity/Recall, Specificity, and Precision

Sensitivity or Recall: Proportion of true positive example among ALL positive (P)

$$TPR = Sensitivity = Recall = \frac{TP}{TP + FN} = \frac{TP}{P} \quad (1)$$

Precision: Proportion of true positive example among **the predicted positive** (PP)

$$Precision = \frac{TP}{PP} \quad (2)$$

→ Important when you have limited tests & and you want the top

F1-score: F1 = 2 × (precision × recall) / (precision + recall)

Precision is very important in many circumstances, e.g.,

- We can only afford testing 5 drugs among 100 predicted drugs
- We can admit a small number of high-risk patients among all patients

```
from sklearn.metrics import precision_recall_curve
precision, recall, thres = precision_recall_curve(label, pred)
auprc = auc(recall, precision)
```

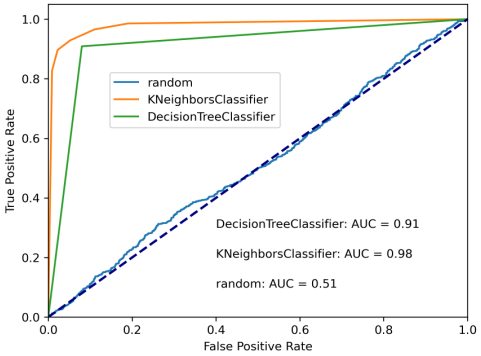

Constructing ROC and PR curve by varying the thresholds

TPR	FPR	Threshold
0.0	0.000000	1.999820
0.0	0.001111	0.999820
0.0	0.020000	0.984463
...	...	...
0.3	0.067778	0.936420
0.3	0.090000	0.918972
0.4	0.090000	0.918953
0.4	0.291111	0.719385
0.5	0.291111	0.717308
...	...	...
0.9	0.946667	0.058096
1.0	0.946667	0.056398
1.0	1.000000	0.000270

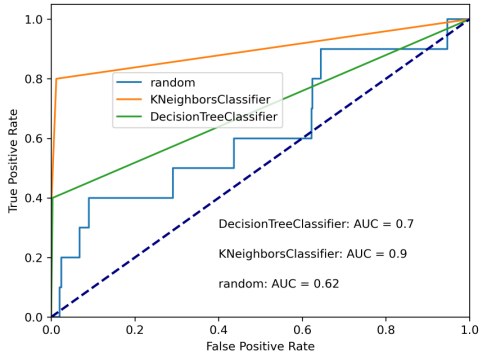
Precision	Recall	Threshold
0.010216	0.9	0.037235
0.010227	0.9	0.037284
0.010239	0.9	0.038246
...	...	...
0.010870	0.8	0.206335
0.010884	0.8	0.206341
...	...	...
0.010870	0.7	0.288363
...	...	...
1.000000	0.0	0.999557

ROC is designed for class-balanced data (Colab)

When we have 50% positive and 50% negative labels, a line that goes along the diagonal indicates random guess (P=900,N=900).

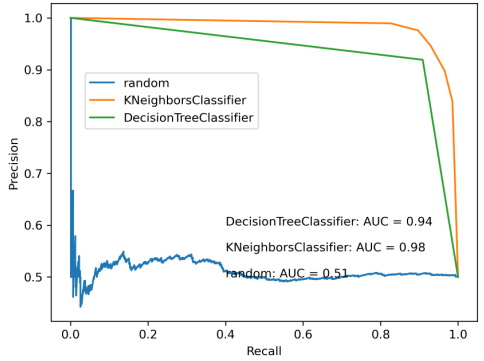


However, suppose we have 1% positive and 99% negative labels, random prediction will no longer follow the diagonal line (P=10,N=900).

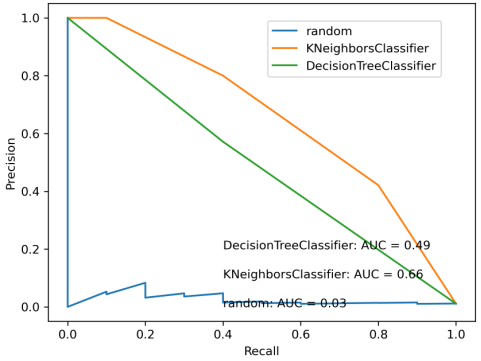


Precision-recall curve is a better choice for imbalanced data (Colab)

When we have 50% positive and 50% negative labels (P=900,N=900).



When we have 1% positive and 99% negative labels (P=10,N=900).



Summary

- Approximate generalization performance using test data
- ROC is an effective way to test overall model performance using all thresholds
- Cross-validation makes use of the full data for both training and evaluation
- Generic model implementation in Scikit-learn enables efficient method comparison
- Precision-recall is an alternative metric to ROC and it is better suited to measure performance on imbalanced data and circumstance where precision is important.

Lectures 5 & 6 Module 3. Decision Tree

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

January 21 & 23, 2025

Outline

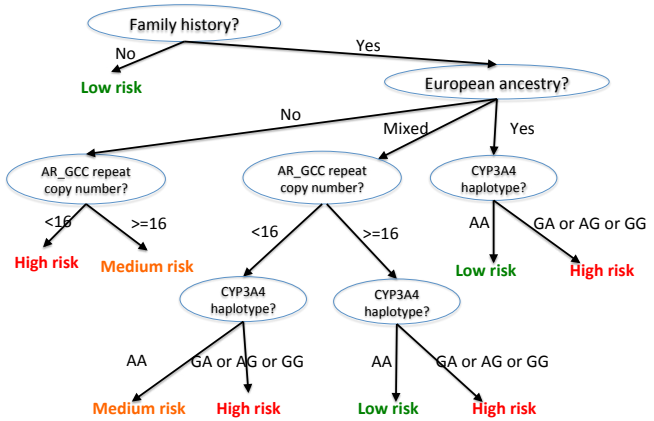
Learning objectives

Understanding the following concepts

- DT prediction
- Cost functions
- DT training algorithm
- Overfitting in DT

Outline

Decision Tree (DT) aka Classification And Regression Tree (CART)



A toy DT for diagnosing non-small cell lung cancer

Pros:

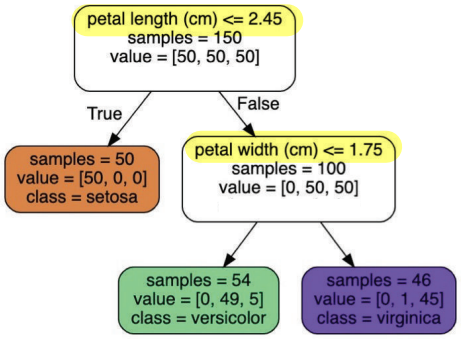
- Easy to interpret
- Handle mixed discrete and continuous inputs
- Insensitive to scaling
- Built-in variable selection
- Non-linear approach

Cons:

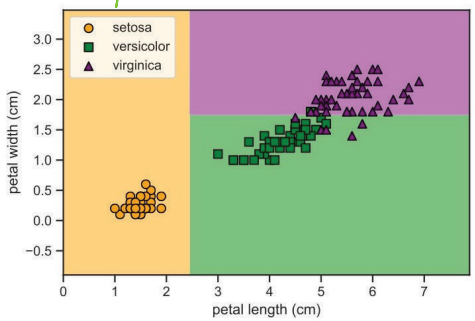
- Produce coarse grained probabilities especially at low tree depth
- Prone to overfitting at high tree depth

DT trained on Iris flower data for multi-class prediction

An example of classifying Iris flower using a classification tree (Murphy22 Figure 18.3):



* You can use the same feature multiple times



Notation review for general supervised learning problem

Our dataset $\mathcal{D}$ consists of N pairs of input vector and target variable:

$$\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^{N}$$

Target variable

- Classification: $y^{(n)} \in \{1, \ldots, C\}$
- Regression: $y^{(n)} \in \mathbb{R}$

Input features

- Continuous feature i: $x_i^{(n)} \in \mathbb{R}$
- Categorical feature j: $x_j^{(n)} \in \{1, \ldots, C_j\}$

One-hot-encoding

For computational convenience, we often represent the categorical variable using **one-hot encoding** by creating C_j binary input features from variable j (although this is not necessary for DT to work):

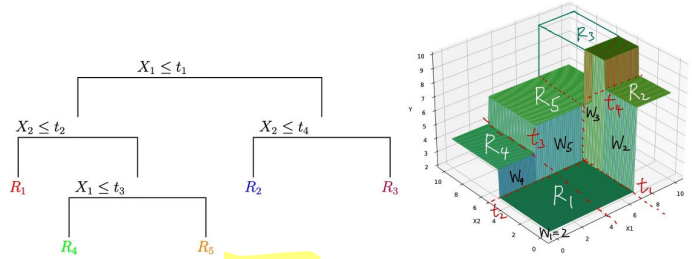
$$x_{j,1}^{(n)}, \ldots, x_{j,C_j}^{(n)} \in \{0, 1\}$$

Data	Color
$x^{(1)}$	Red
$x^{(2)}$	Green
$x^{(3)}$	Blue

$\Rightarrow$

Variable	Red	Green	Blue
$x^{(1)}$	1	0	0
$x^{(2)}$	0	1	0
$x^{(3)}$	0	0	1

A simple illustration of a regression tree



- Divide the training data into 5 regions $R_k \in \{R_1, \ldots, R_5\}$ by the thresholds $t_1, \ldots, t_4$. For instance, $R_1 = \{\mathbf{x}_1 \le t_1, \mathbf{x}_2 \le t_2\}$.
- The height of a region (i.e., regression weight) is the *average* target values in region k:

$$w_k = \frac{\sum_{n=1}^{N} y^{(n)} \mathbb{I}[\mathbf{x}^{(n)} \in R_k]}{\sum_{n=1}^{N} \mathbb{I}[\mathbf{x}^{(n)} \in R_k]} = \frac{1}{N_k} \sum_{n \in \mathcal{S}_k} y^{(n)} \quad (\mathcal{S}_k \text{ is the set of data indices in Region } k)$$

- The predicted value of a new data point $\mathbf{x}^{(*)}$ is set to the weight of the region it belongs:

$$\hat{y}^{(*)} \equiv f(\mathbf{x}^{(*)}; \theta) = \sum_{k=1}^{K} w_k \mathbb{I}[\mathbf{x}^{(*)} \in R_k] \quad (\theta \text{ denotes the trained tree and its regression weights})$$

belongs to region

Prediction per region in classification

Most frequent label in region k (i.e., the mode):

$$w_k = \arg\max_c \sum_{n \in \mathcal{S}_k} \mathbb{I}[y^{(n)} = c]$$

take class from highest returning example

$$\hat{y}^{(*)} = \sum_{k=1}^{K} w_k \mathbb{I}[\mathbf{x}^{(*)} \in R_k]$$

Class probability in region k (i.e., proportion of class c in region k):

$$p(y = c | R_k) = \frac{1}{N_k} \sum_{n \in \mathcal{S}_k} \mathbb{I}[y^{(n)} = c] \equiv \pi_k(c)$$

frequency of survival

$$p(\hat{y}^{(*)} = c) = \sum_{k=1}^{K} \pi_k(c) \mathbb{I}(\mathbf{x}^{(*)} \in R_k)$$

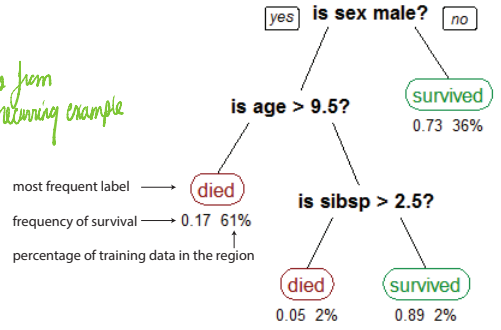


image source from here

Survived is the positive label and death is the negative label in the above DT trained on the Titanic dataset.

Two main questions to be answered in this learning module

1. How to split data at each tree node?
   - How to evaluate quality of a split?
   - How to choose feature & threshold?
2. How to decide when to stop the tree from splitting?

1. Evaluate each split by a **cost function**
2. Choose feature and threshold that **minimize** the cost
3. Use a validation set to choose tree depth

Outline

A common cost function for regression is *Mean Squared Error (MSE)*

Our prediction on the training data points in region k is their average target values:

$$w_k = \frac{1}{N_k} \sum_{n \in \mathcal{S}_k} y^{(n)}$$

Note: each training data point falls in exactly one of the regions.
Sum of squared error (SSE) and Mean squared error per region k are:

$$\text{SSE}(R_k, \mathcal{D}; w_k) = \sum_{n \in \mathcal{S}_k} (y^{(n)} - w_k)^2, \quad \text{MSE}(R_k, \mathcal{D}; w_k) = \frac{1}{N_k} \text{SSE}(R_k, \mathcal{D}; w_k)$$

SSE and MSE over all K regions (i.e., all training data points) are:

$$SSE(\mathcal{D}; \theta) = \sum_{k=1}^{K} \text{SSE}(R_k, \mathcal{D}; w_k) = \sum_{k=1}^{K} \sum_{n \in \mathcal{S}_k} (y^{(n)} - w_k)^2, \quad MSE(\mathcal{D}; \theta) = \frac{1}{N} SSE(\mathcal{D}; \theta)$$

↳ sum of all regions

Measuring classification cost using *Misclassification Rate*

Most frequent label in region k (i.e., the mode):

$$w_k = \arg\max_c \sum_{n \in \mathcal{S}_k} \mathbb{I}[y^{(n)} = c]$$

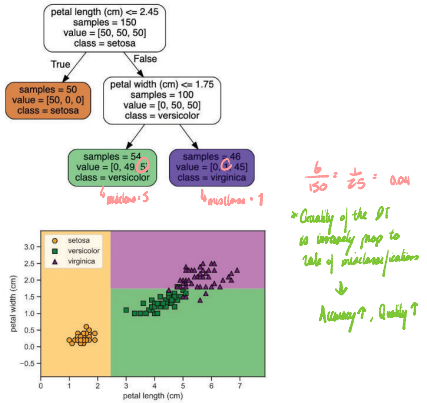
Cost per region k:

$$\text{cost}(R_k, \mathcal{D}; w_k) = \sum_{n \in \mathcal{S}_k} \mathbb{I}[y^{(n)} \neq w_k]$$

Total cost is the overall misclassification rate:

$$\text{total cost}(\mathcal{D}; \theta) = \sum_{k=1}^{K} \text{cost}(R_k, \mathcal{D}; w_k)$$

$$\text{misclassification rate} = \frac{1}{N} cost(\mathcal{D}; \theta)$$

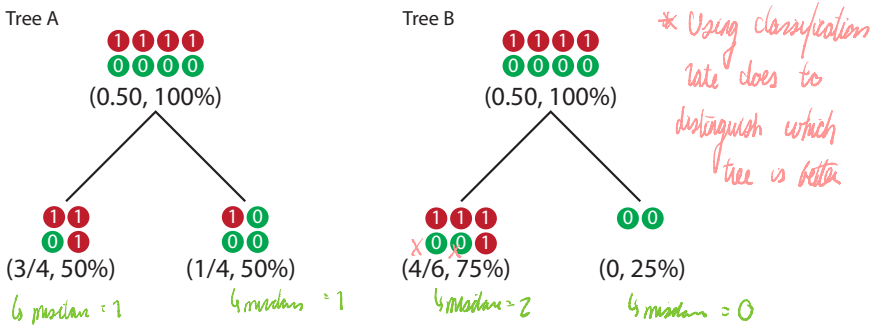


Pop quiz: what's training misclassifcation rate of the above DT?

Answer: $6/150 = 0.04$

Misclassification rate is not sensitive enough

The total misclassification rates are the same for these two trees but the classification cost at each leaf node is different:



$$cost(\mathcal{D};\theta) = \frac{1}{N}\sum_{k=1}^{K}\sum_{n\in\mathcal{S}_k}\mathbb{I}[y^{(n)} \neq w_k] = 2/8 = 0.25$$

Idea: we need a more sensitive measure of the **model uncertainties** in each region.

Entropy: a measure of uncertainty (the lower the better)

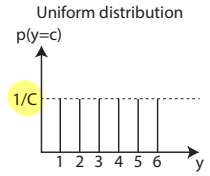
Entropy is the expected amount of information in observing a random variable y:

$$H(y) = -\sum_{c=1}^{C} p(y=c)\log_2 p(y=c)$$

The *less* certain the *higher* the entropy. For example,

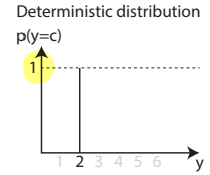
- Uniform distribution where $p(y=c) = \frac{1}{C}\forall c$ has the highest entropy:

$$H(y) = -\sum_{c=1}^{C}\frac{1}{C}\log_2(\frac{1}{C}) = \log_2(C)$$

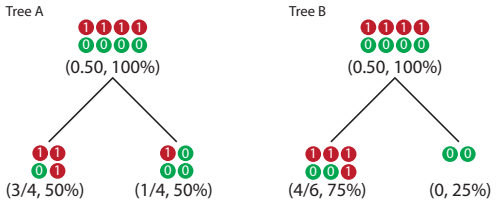


- When $p(y=c)=1$ and $p(y=c')=0\,\forall c' \neq c$ the entropy is the lowest:

$$H(y) = -\log_2(1) = 0$$



Entropy for classification cost



$$H(y;R) = -\sum_{c} p(y=c;R)\log_2 p(y=c;R)$$

$$= -p(y=0;R)\log_2 p(y=0;R) - p(y=1;R)\log_2 p(y=1;R)$$

Tree A

$$H(y;R_{left}) = -\frac{1}{4}\log_2(\frac{1}{4}) - \frac{3}{4}\log_2(\frac{3}{4}) \approx 0.81$$

$$H(y;R_{right}) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} \approx 0.81$$

$$\text{Mean Cost} = \frac{4}{8}H(y;R_{left}) + \frac{4}{8}H(y;R_{right}) = 0.81$$

Tree B

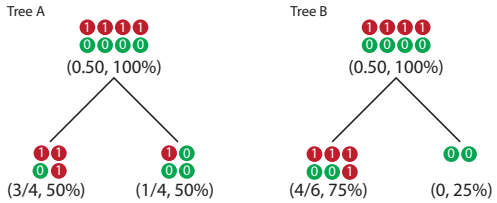
$$H(y;R_{left}) = -\frac{2}{6}\log_2(\frac{2}{6}) - \frac{4}{6}\log_2(\frac{4}{6}) \approx 0.92$$

$$H(y;R_{right}) = -1\log_2 1 - 0 \approx 0$$

$$\text{Mean Cost} = \frac{6}{8}H(y;R_{left}) + \frac{2}{8}H(y;R_{right}) = 0.69$$

Tree B has lower cost than Tree A.

Entropy for classification cost



$$H(y;R) = -\sum_{c} p(y=c;R)\log_2 p(y=c;R)$$

$$= -p(y=0;R)\log_2 p(y=0;R) - p(y=1;R)\log_2 p(y=1;R)$$

Tree A

$$H(y;R_{left}) = -\frac{1}{4}\log_2(\frac{1}{4}) - \frac{3}{4}\log_2(\frac{3}{4}) \approx 0.81$$

$$H(y;R_{right}) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} \approx 0.81$$

$$\text{Mean Cost} = \frac{4}{8}H(y;R_{left}) + \frac{4}{8}H(y;R_{right}) = 0.81$$

Tree B

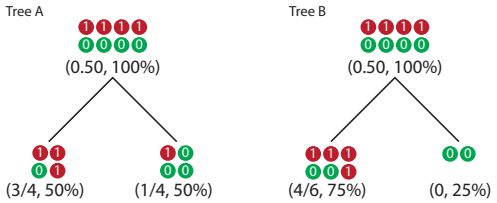
$$H(y;R_{left}) = -\frac{2}{6}\log_2(\frac{2}{6}) - \frac{4}{6}\log_2(\frac{4}{6}) \approx 0.92$$

$$H(y;R_{right}) = -1\log_2 1 - 0 \approx 0$$

$$\text{Mean Cost} = \frac{6}{8}H(y;R_{left}) + \frac{2}{8}H(y;R_{right}) = 0.69$$

Tree B has lower cost than Tree A.

Entropy for classification cost



$$H(y;R) = -\sum_c p(y=c;R)\log_2 p(y=c;R)$$
$$= -p(y=0;R)\log_2 p(y=0;R) - p(y=1;R)\log_2 p(y=1;R)$$

Tree A

$$H(y;R_{left}) = -\frac{1}{4}\log_2(\frac{1}{4}) - \frac{3}{4}\log_2(\frac{3}{4}) \approx 0.81$$
$$H(y;R_{right}) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} \approx 0.81$$
$$\text{Mean Cost} = \frac{4}{8}H(y;R_{left}) + \frac{4}{8}H(y;R_{right}) = 0.81$$

* The lower the mean cost, the lower the entropy → better efficiency

Tree B

$$H(y;R_{left}) = -\frac{2}{6}\log_2(\frac{2}{6}) - \frac{4}{6}\log_2(\frac{4}{6}) \approx 0.92$$
$$H(y;R_{right}) = -1\log_2 1 - 0 \approx 0$$
$$\text{Mean Cost} = \frac{6}{8}H(y;R_{left}) + \frac{2}{8}H(y;R_{right}) = \mathbf{0.69}$$

Tree B has lower cost than Tree A.

Resume here in Lec 6 (Jan 24)

Gini Index (GI): the default cost in DT (the lower the better)

$$p(y=c;R_k) = \frac{1}{N_k}\sum_{n\in\mathcal{S}_k}\mathbb{I}(y^{(n)}=c) \equiv \pi_k(c)$$

$$GI(R_k) = \sum_{c=1}^{C}\pi_k(c)(1-\pi_k(c)) = \sum_{c=1}^{C}\pi_k(c) - \sum_{c=1}^{C}\pi_k(c)^2 = 1 - \sum_{c=1}^{C}\pi_k(c)^2$$

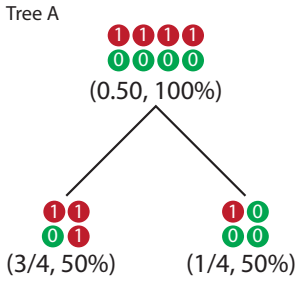
$$GI = \frac{1}{N}\sum_{k=1}^{K} N_k GI(R_k)$$

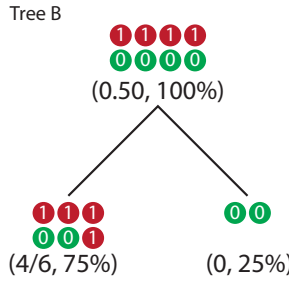
* The lower the variance, the more certain the model is

- In theory, $\pi_k(c)(1-\pi_k(c))$ is the *variance* of the Bernoulli distribution for the Boolean indicator $\mathbb{I}[y=c]$ in region k with rate $\pi_k(c)$ for each class c:
$$y \sim (\pi_k(c))^{\mathbb{I}[y=c]}(1-\pi_k(c))^{\mathbb{I}[y\neq c]}$$
- Therefore, $\sum_c \pi_k(c)(1-\pi_k(c))$ measures *the sum of variances* over all classes.
- Therefore, minimizing GI is equivalent to minimizing the expected variance.
- For a "pure" region, where all data points in the region belong to a single class c, $p(y=c;R_k)=1$ and $p(y=c';R_k)=0$ for $c'\neq c$, $GI(R_k)=0$. * Cannot further split to improve prediction

Compute GI for the two example trees

* GI & Entropy will usually give you the same results

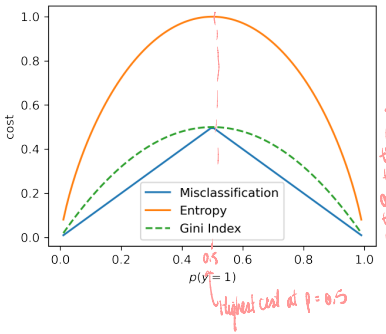




$$GI(y;R_{A,left}) = 1 - (0.25^2 + 0.75^2) = 0.375$$
$$GI(y;R_{A,right}) = 1 - (0.75^2 + 0.25^2) = 0.375$$
$$GI(y;TreeA) = \frac{4}{8}GI(y;R_{left}) + \frac{4}{8}GI(y;R_{right}) = 0.375$$

$$GI(y;R_{B,left}) = 1 - ((\frac{2}{6})^2 + (\frac{4}{6})^2) = 0.44$$
$$GI(y;R_{B,right}) = 1 - (1+0) = 0$$
$$GI(y;TreeB) = \frac{6}{8}GI(y;R_{left}) + 0 = \mathbf{0.33}$$

Analysis of the 3 *training* cost functions in binary classification (Colab)



Two-class cost as the function $p(y=1) \equiv p$:

Not a smooth fct

$$\text{misclas. rate} = 1 - \max(p, 1-p) = \min(p, 1-p)$$

Smooth fct because they actually take into account the continuous p

$$\textit{entropy} = -p\log_2(p) - (1-p)\log_2(1-p)$$

$$\textit{gini index} = p(1-p) + (1-p)p = 2p(1-p)$$

POS/NEG NEG/POS

Comments:

- In binary classification, the lower class proportion is the misclassification rate
- We compute gini index each class. In binary classification, the two gini indices are the same.

- All 3 cost functions quantify the "impurity" of a tree node.
- Training a DT involves minimizing one of the cost functions.
- Entropy and Gini Index behave similarly and are usually better choice than misclassification rate as a training cost function because they take into account *the prediction uncertainties* by using prediction probabilities in each leaf node.

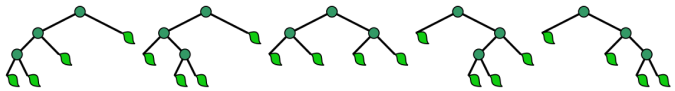
Outline

Combinatorial search space of all possible trees

- **Objective**: find the optimal DT that minimizes the training cost.
- The number of all possible topologies of *full* binary trees with T tests (i.e., T non-leaf nodes) follows the Catalan sequence, where the T^{th} Catalan number is

$$C_T = \frac{1}{T+1}\binom{2T}{T} = \frac{(2T)!}{(T+1)(2T-T)!T!} = \frac{(2T)!}{(T+1)!T!}$$

For 3 tests, for instance, there are 5 full binary trees to choose from:



- The number of trees grows quite fast as we increase the number of tests. Starting with $T = 0$, the first 12 numbers in the Catalan sequence are: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786.
- It turns out finding the optimal decision tree is an **NP-hard** combinatorial optimization problem (NP stands for Nondeterministic Polynomial).

Peudocode for greedy heuristic search at each tree node (See Colab)

Algorithm 1 greedy_test(node)

```
min_cost = cost(node.y) // (better than min_cost = ∞
best_feature_index = best_feature_thres = NULL
for each feature d ∈ {1, ..., D} do
    for each threshold t ∈ T_d for feature d do // Tests for all unique values of feature d
        cost_left = cost(node.y[node.X[:, d] ≤ t])
        cost_right = cost(node.y[node.X[:, d] > t])
        cost_d = (N_left / N) cost_left + (N_right / N) cost_right
        if cost_d < min_cost then
            min_cost = cost_d;
            best_feature_index = d;
            best_feature_thres = threshold
        end if
    end for
end for
return min_cost, best_feature_index, best_feature_thres
```

D T

N

Compared

* for continuous data set → take all unique values

N: # training examples
D: # features
T: # threshold per feature

Time complexity? O(DTN)

↳ this is only for 1 node

Recursively find the best feature and threshold at every node (Colab)

Algorithm 2 `_fit_tree`(node, max_depth, min_leaf_instance)

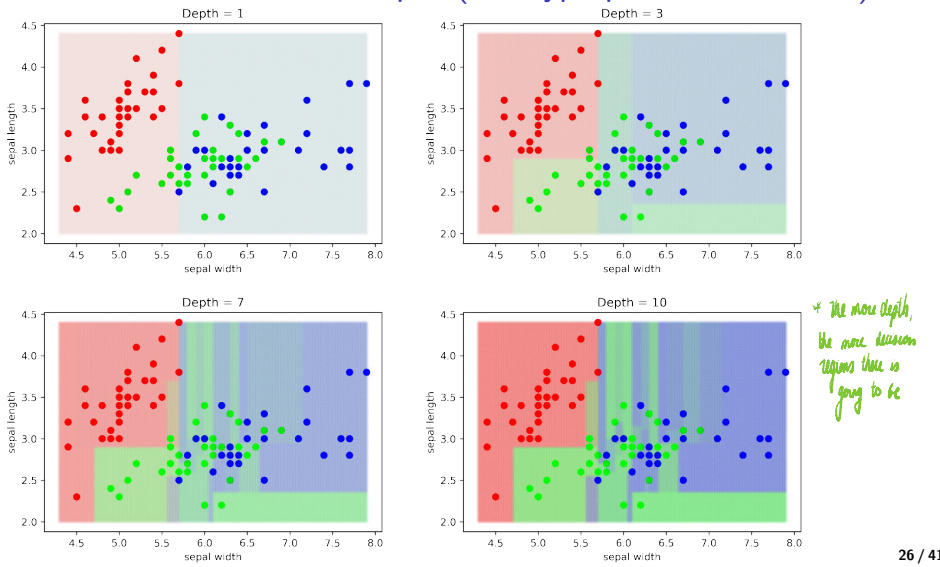
1: **if** node.depth $\leq$ max_depth and node.X.shape[0] $>$ min_leaf_instance and cost(node.y[node.data_indices]) > 0 **then**
2: split_cost, split_feat, split_value = greedy_test(node)
3: node.split_feature = split_feat
4: node.split_value = split_value
5: test_logicals = node.X[node.data_indices, split_feat] <= split_value
6: left_node = Node(data_indices=node.data_indices[test_logical], parent=node)
7: right_node = Node(data_indices=node.data_indices[!test_logical], parent=node)
8: `_fit_tree`(left_node, max_depth, min_leaf_instance)
9: `_fit_tree`(right_node, max_depth, min_leaf_instance)
10: node.left = left_node
11: node.right = right_node
12: **end if**

Algorithm 3 fit(tree, max_depth, min_leaf_instance=1)

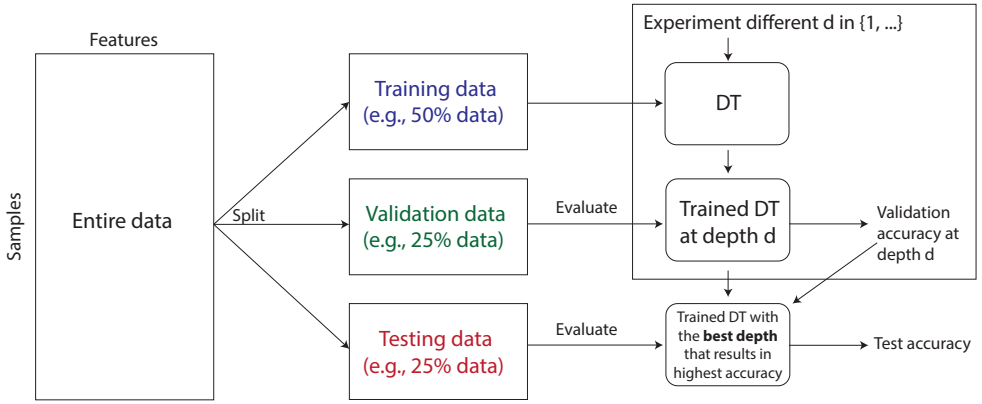
1: `_fit_tree`(tree.root, max_depth, min_leaf_instance)

Outline

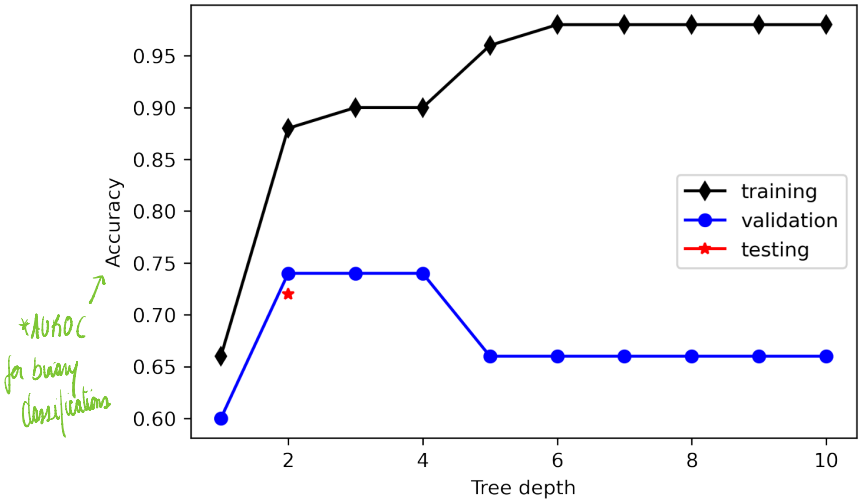
How to choose maximum tree depth (the hyperparameter for DT)?



Choose tree depth using a validation set (same as choosing K in KNN)



Choose optimal tree depth using a validation set (Colab)



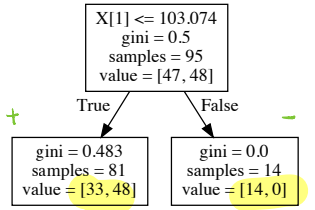
Outline

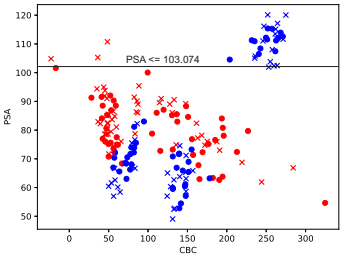
Decision tree using Scikit-Learn

```
1  from sklearn import model_selection,tree
2  import graphviz
3  for depth in range(1,6):
4      dt = tree.DecisionTreeClassifier(max_depth=depth)
5      dt.fit(X_train, y_train)
6      p_train = dt.predict(X_train)
7      p_test = dt.predict(X_test)
8      #plot tree
9      dot_data = tree.export_graphviz(dt, out_file=None)
10     graph = graphviz.Source(dot_data)
11     graph.render("prostate_tree_depth_"+str(depth))
```

Note: Line 9-11 requires installing graphviz: `pip install graphviz`

Decision tree (max_depth = 1)





Training data:

	PN	PP
AN	14	33
AP	0	48

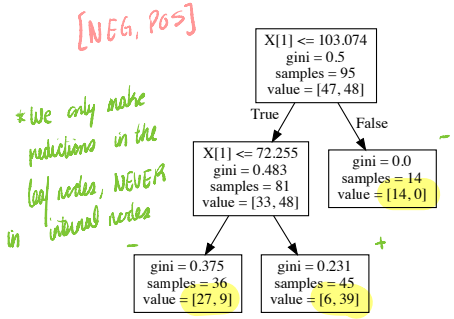
TPR = 48/(48+0) = 1.0
FPR = 33/(33+14) = 0.7

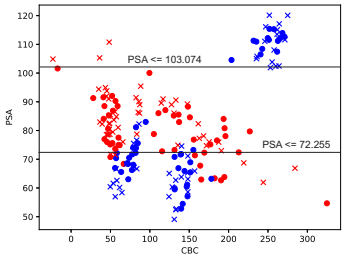
Test data:

	PN	PP
AN	13	30
AP	3	49

TPR = 49/(49+3) = 0.94
FPR = 30/(30+13) = 0.7

Decision tree (max_depth = 2)





Training data:

	PN	PP
AN	41	6
AP	9	39

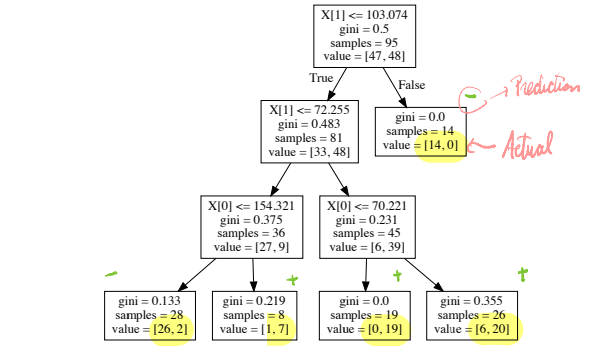
TPR = 39/(39+9) = 0.81
FPR = 6/(6+41) = 0.13

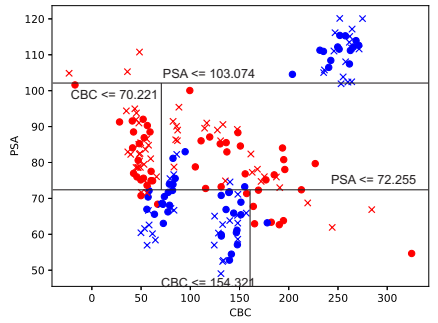
Test data:

	PN	PP
AN	36	7
AP	8	44

TPR = 44/(44+8) = 0.85
FPR = 7/(7+36) = 0.16

Decision tree (max_depth = 3)





* Depending on the thresholds, we will have different TPR and FPR

Training data:

	PN	PP
AN	40	7
AP	2	46

TPR = 46/(46+2) = 0.96
FPR = 7/(7+40) = 0.15

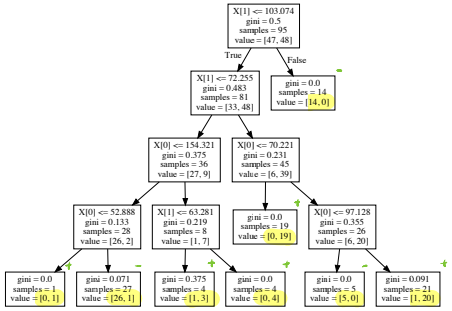
Compute this in midterm (case will be given)

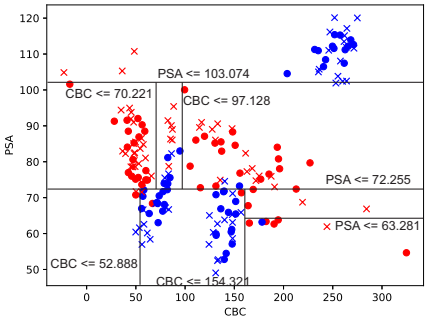
Test data:

	PN	PP
AN	35	8
AP	5	47

TPR = 47/(47+5) = 0.9
FPR = 8/(8+35) = 0.19

Decision tree (max_depth = 4) - overfitting occurs





Training data:

	PN	PP
AN	45	2
AP	1	47

TPR = 47/(47+1) = 0.98
FPR = 2/(2+45) = 0.04

// this is pretty good

Test data:

	PN	PP
AN	37	6
AP	11	41

TPR = 41/(41+11) = 0.79
FPR = 6/(6+37) = 0.14

Decision tree (max_depth = 5 & 6) - more overfitting

Tree depth = 5

Training data:

	PN	PP
AN	46	1
AP	1	47

TPR = 47/(47+1) = 0.98
FPR = 1/(1+46) = 0.02

Test data:

	PN	PP
AN	37	6
AP	11	41

TPR = 41/(41+11) = 0.79
FPR = 6/(6+37) = 0.14

Tree depth = 6

Training data:

	PN	PP
AN	47	0
AP	0	48

TPR = 48/(48+0) = 1.0
FPR = 0/(0+47) = 0.0

Test data:

	PN	PP
AN	37	6
AP	11	41

TPR = 41/(41+11) = 0.79
FPR = 6/(6+37) = 0.14

Outline

Feature importance

- Reduction of the cost (e.g., Gini index) due to the split at node $j \in \{1, \ldots, J\}$:

$$\Delta g_j = \underbrace{\frac{N_j}{N}}_{\text{Gini index}} \times g_j - \frac{N_j^{(left)}}{N} g_j^{(\text{left})} - \frac{N_j^{(right)}}{N} g_j^{(\text{right})}$$

this needs to be reduced to select feature

where $\frac{N_j}{N}$ is the fraction of samples in node j and g_j is the cost at node j; $\frac{N_j^{(left)}}{N}$ and $\frac{N_j^{(right)}}{N}$ are fractions of samples in the left and right child of node j; $g_j^{(\text{left})}$ and $g_j^{(\text{right})}$ are cost in the left and right child of node j.

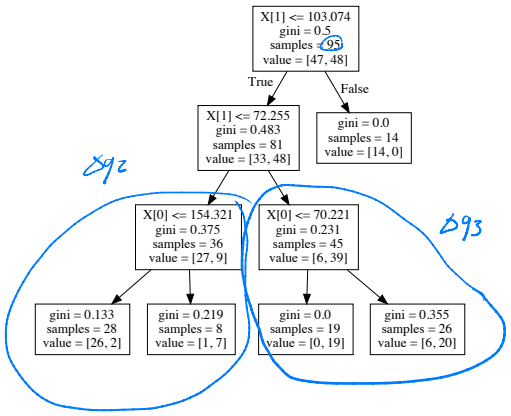
- The feature importance of feature $d \in \{1, \ldots, D\}$ is the total reduction over all the J non-leaf nodes, where feature d is used:

$$r_d = \sum_{j=1}^{J} \Delta g_j \mathbb{I}[v_j = d]$$

where $\mathbb{I}[v_j = d]$ indicates feature d is used in node j. The formula for computing feature importance as one of the optional tasks in Assignment 1 has been updated.

Example: feature importance of x_0 in DT at depth 3

* We're now looking at the internal nodes (it's where you perform the training tests)



- Gini reduction at node 2 (level 3 left node):

$$\Delta g_2 = \frac{36}{95}0.375 - \frac{28}{95}0.133 - \frac{8}{95}0.219 = 0.0844$$

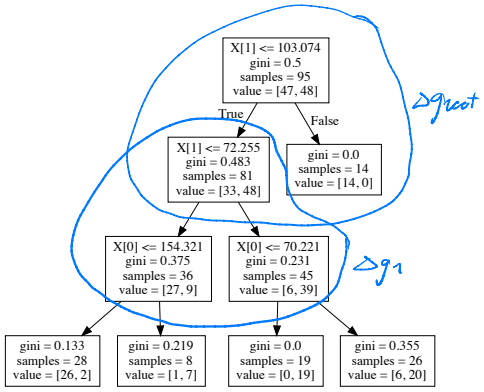
- Gini reduction at node 3 (level 3 right node):

$$\Delta g_3 = \frac{45}{95}0.231 - \frac{19}{95}0 - \frac{26}{95}0.355 = 0.0123$$

- Feature importance of x_0:

$$r_0 = \Delta g_2 + \Delta g_3 = 0.0844 + 0.0123 = \mathbf{0.0967}$$

Example: feature importance of x_1 in DT at depth 3



- Gini reduction at root due to x_1:

$$\Delta g_{root} = \frac{95}{95}0.5 - \frac{81}{95}0.483 - \frac{14}{95}0 = 0.0881$$

- Gini reduction at node 1 (i.e., left child of root) also due to x_1:

$$\Delta g_1 = \frac{81}{95}0.483 - \frac{36}{95}0.375 - \frac{45}{95}0.231 = 0.160$$

- Feature importance of x_1:

$$r_1 = \Delta g_{root} + \Delta g_1 = 0.0881 + 0.160 = \mathbf{0.2481}$$

Therefore, x_1 has more than 2.5 times higher feature importance score than x_0 ($0.2481/0.0967 = 2.566$).

Outline

Summary of DT

- Costs: Misclassification rate, Entropy, Gini index
- DT is a greedy and recursive algorithm that finds the best feature and the best threshold at each tree node to minimize the loss (e.g., Gini index) within the node.
- Compared to KNN, robust to scaling and noise, fast predictions, more interpretable
- Overfitting occurs when the model gets too complicated (i.e., high tree depth in the case of DT) and fits extremely well on the training data but generalizes poorly to the test data.
- Feature importance score derived from a trained DT can be used to rank features.
- DT is the building block of common ensemble learning methods such as Random Forest and XGBoost (Module 6)

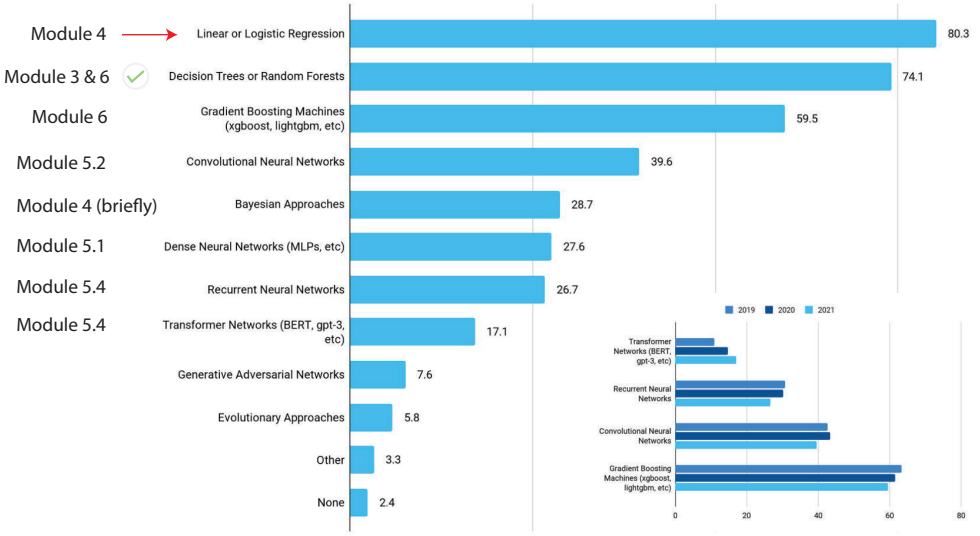
Lecture 7 & 8 - Module 4.1. Linear regression

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

January 28 & 30, 2025

Most commonly used ML algorithms in Kaggle 2021 survey



Outline

Outline

Learning objectives

Understanding the following concepts

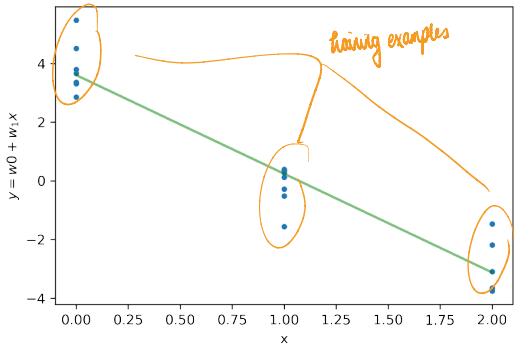
- Simple linear model
- Finding the best linear fit by minimizing SSE
- Matrix algebra in solving multiple regression
- Probabilistic interpretation
- Feature transformation by non-linear basis functions

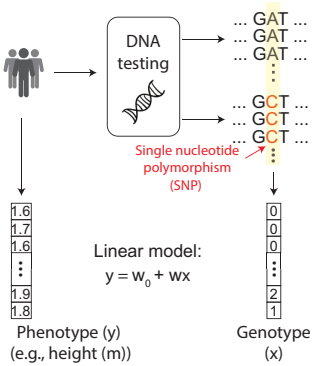
Outline

Linear regression using a one-dimensional input

We want to predict real-valued quantity or often known as **response or target variable** $y \in \mathbb{R}$ by finding a mapping function that maps from a **one-dimensional input x** to the real-valued y. We can fit a linear function on training examples $\{x^{(n)}, y^{(n)}\}_{n=1}^{N}$ (e.g., predicting phenotype from one genetic mutation):

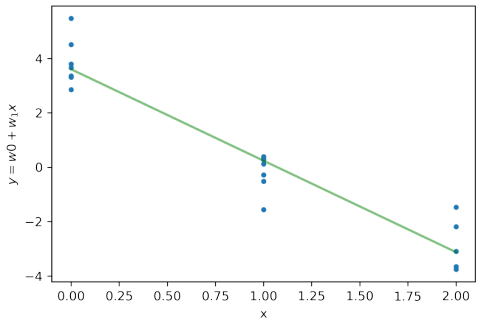
$$f(x^{(n)}; w_0, w_1) = w_0 + w_1 x^{(n)} \quad (1)$$





Linear regression using a one-dimensional input

$$f(x^{(n)}; w_0, w_1) = w_0 + w_1 x^{(n)}$$



- $w_0 = 3.4$ is the **intercept** or **bias**, which is not be confused with the "model bias"
- $w_1 = -3.25$ is the **slope** of the linear function or the **regression coefficient**.

Simple linear regression using one input feature

In statistics, we often write down the regression formula as:

$$y^{(n)} = w_0 + w_1 x^{(n)} + \epsilon^{(n)}$$

where $\epsilon^{(n)}$ is the prediction error for example n.

Using matrix notation, we can write the regression formula as:

$$\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} = \begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \vdots & \vdots \\ 1 & x^{(N)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} + \begin{bmatrix} \epsilon^{(1)} \\ \epsilon^{(2)} \\ \vdots \\ \epsilon^{(N)} \end{bmatrix} \quad (2)$$

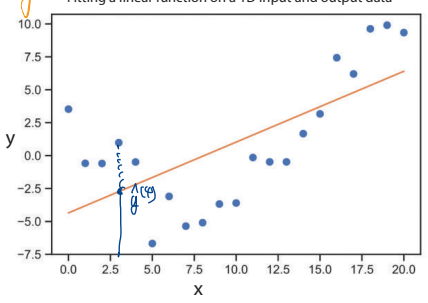
$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon} \quad (3)$$

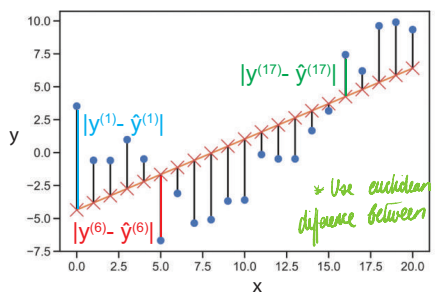
where $\mathbf{y}$ and $\boldsymbol{\epsilon}$ are both $N \times 1$ vectors, $\mathbf{X}$ is a $N \times 2$ matrix, and $\mathbf{w}$ is a 2×1 vector.

Residual error as a measure of prediction loss

Every straight line we fit incurs a prediction error on the training data point unless the fitted line goes through that data point. The **residual error** is Euclidean distance between the observed response $y^{(n)}$ value and the predicted response $\hat{y}^{(n)} = \mathbf{x}^{(n)}\mathbf{w}$:

$$l_n = ||y^{(n)} - \hat{y}^{(n)}||_2 = \sqrt{(y^{(n)} - \hat{y}^{(n)})^2} = |y^{(n)} - \hat{y}^{(n)}| \quad (4)$$





Notation: $||\mathbf{a}||_2 = \sqrt{\sum_i a_i^2}$ is L_2-norm and $||\mathbf{a}||_2^2 = \sum_i a_i^2$ is the squared of the L2-norm.

Fitting a linear regression function by minimizing the sum of squared error

Sum of squared error (SSE) as a function of the linear coefficients $\mathbf{w}$ is defined as:

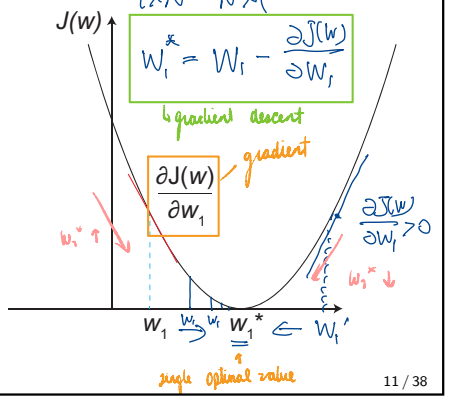
$$J(\mathbf{w}) = ||\mathbf{y}-\hat{\mathbf{y}}||_2^2 = \sum_{n=1}^{N}(y^{(n)}-\hat{y}^{(n)})^2 = \sum_{n=1}^{N}(y^{(n)}-w_0-w_1x^{(n)})^2 = (\mathbf{y}-\hat{\mathbf{y}})^\top(\mathbf{y}-\hat{\mathbf{y}}) = \boldsymbol{\epsilon}^\top\boldsymbol{\epsilon} \quad (5)$$

Goal: find the best $\mathbf{w}$ that minimizes the SSE:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w}) \quad (6)$$

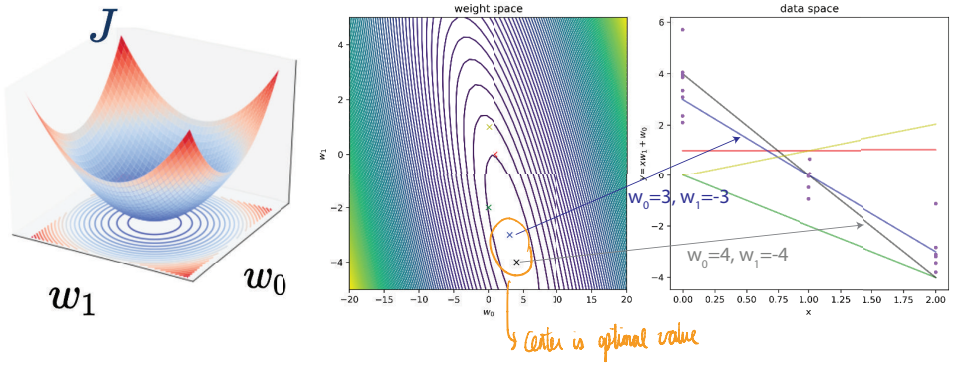
Given the error function is differentiable everywhere, the slope at the current value of w_1 projecting onto the error parabola is shown on the right.

The SSE error function is **convex** (more details in Module 4.4). The optimal w_1^* is found where **the slope or the partial derivative is zero** $\frac{\partial J(\mathbf{w})}{\partial w_1^*} = 0$.



Contour plot visualizes the error surface as a function of w_1 and w_0

$$J(\mathbf{w}) = \sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2 = \sum_{n=1}^{N}(y^{(n)} - w_0 - w_1x^{(n)})^2$$



Derivation of ordinary least squared (OLS) solution for w_0

$$\frac{\partial J(\mathbf{w})}{\partial w_0} = \frac{\partial}{\partial w_0}\sum_{n=1}^{N}(y^{(n)} - w_0 - w_1x^{(n)})^2$$

$$= \sum_{n=1}^{N}\frac{\partial(y^{(n)} - w_0 - w_1x^{(n)})^2}{\partial(y^{(n)} - w_0 - w_1x^{(n)})}\frac{\partial(y^{(n)} - w_0 - w_1x^{(n)})}{\partial w_0}$$

$$= \sum_{n=1}^{N}2(y^{(n)} - w_0 - w_1x^{(n)})(-1)$$

Set $\frac{\partial J(w_0)}{\partial w_0}$ to zero and multiplying $\frac{1}{2}$ and -1 on both sides gives:

$$\sum_{n=1}^{N}(y^{(n)} - w_0 - w_1x^{(n)}) = 0$$

Solving for w_0:

$$\sum_{n=1}^{N}y^{(n)} - \sum_{n=1}^{N}w_0 - \sum_{n=1}^{N}w_1x^{(n)} = 0$$

$$\sum_{n=1}^{N}w_0 = \sum_{n=1}^{N}y^{(n)} - w_1\sum_{n=1}^{N}x^{(n)}$$

$$Nw_0 = \sum_{n=1}^{N}y^{(n)} - w_1\sum_{n=1}^{N}x^{(n)}$$

$$w_0 = \frac{1}{N}\sum_{n=1}^{N}y^{(n)} - w_1\frac{1}{N}\sum_{n=1}^{N}x^{(n)}$$

$$w_0 = \bar{y} - w_1\bar{x}$$

Therefore, $\hat{w}_0 = \bar{y} - w_1\bar{x}$

Derivation of OLS solution for w_1

$$\frac{\partial J(\mathbf{w})}{\partial w_1} = \frac{\partial}{\partial w_1}\sum_{n=1}^{N}(y^{(n)} - w_0 - w_1x^{(n)})^2$$

$$= \sum_{n=1}^{N}2(y^{(n)} - w_0 - w_1x^{(n)})(-x^{(n)})$$

Set $\frac{\partial J(\mathbf{w})}{\partial w_1}$ to zero and multiplying $\frac{1}{2}$ and -1 on both sides gives:

$$\sum_{n=1}^{N}(y^{(n)} - w_0 - w_1x^{(n)})x^{(n)} = 0 \quad (7)$$

Plug the bias estimate $\hat{w}_0$ in (7) and solve for w_1:

$$\sum_{n=1}^{N}(y^{(n)} - (\bar{y} - w_1\bar{x}) - w_1x^{(n)})x^{(n)} = 0$$

$$\sum_{n=1}^{N}(y^{(n)} - \bar{y} + w_1\bar{x} - w_1x^{(n)})x^{(n)} = 0$$

$$\sum_{n=1}^{N}(y^{(n)} - \bar{y})x^{(n)} - w_1\sum_{n=1}^{N}(x^{(n)} - \bar{x})x^{(n)} = 0$$

$$\hat{w}_1 = \frac{\sum_{n=1}^{N}(y^{(n)} - \bar{y})x^{(n)}}{\sum_{n=1}^{N}(x^{(n)} - \bar{x})x^{(n)}}$$

Derivation of OLS solution for w_1 (cont'd)

Note that

$$\sum_{n=1}^{N}(y^{(n)}-\bar{y})x^{(n)} = \sum_{n=1}^{N}(y^{(n)}-\bar{y})(x^{(n)}-\bar{x})$$

because:

$$\sum_{n=1}^{N}(y^{(n)}-\bar{y})(x^{(n)}-\bar{x}) = \sum_{n=1}^{N}y^{(n)}x^{(n)} - \sum_{n=1}^{N}y^{(n)}\bar{x} - \sum_{n=1}^{N}\bar{y}x^{(n)} + \sum_{n=1}^{N}\bar{y}\bar{x}$$

$$= \sum_{n=1}^{N}y^{(n)}x^{(n)} - N\bar{y}\bar{x} - \sum_{n=1}^{N}\bar{y}x^{(n)} + N\bar{y}\bar{x} = \sum_{n=1}^{N}(y^{(n)}-\bar{y})x^{(n)}$$

Similarly,

$$\sum_{n=1}^{N}(x^{(n)}-\bar{x})x^{(n)} = \sum_{n=1}^{N}(x^{(n)}-\bar{x})(x^{(n)}-\bar{x}) = \sum_{n=1}^{N}(x^{(n)}-\bar{x})^2$$

Update equations for simple linear regression

Therefore, the simple linear regression OLS solutions are:

$$\hat{w}_1 = \frac{\sum_{n=1}^{N}(y^{(n)}-\bar{y})x^{(n)}}{\sum_{n=1}^{N}(x^{(n)}-\bar{x})x^{(n)}} = \frac{\sum_{n=1}^{N}(y^{(n)}-\bar{y})(x^{(n)}-\bar{x})}{\sum_{n=1}^{N}(x^{(n)}-\bar{x})^2}; \quad \hat{w}_0 = \bar{y} - \hat{w}_1\bar{x} \qquad (8)$$

Compare this solution with Pearson correlation coefficient (PCC) yields useful insights:

$$\hat{r}_{xy} = \frac{\sum_{n=1}^{N}(y^{(n)}-\bar{y})(x^{(n)}-\bar{x})}{\sqrt{\sum_{n=1}^{N}(x^{(n)}-\bar{x})^2}\sqrt{\sum_{n=1}^{N}(y^{(n)}-\bar{y})^2}} \in [-1,1]$$

Therefore, we can see that

$$\hat{w}_1 = \hat{r}_{xy}\frac{\sqrt{\sum_{n=1}^{N}(y^{(n)}-\bar{y})^2}}{\sqrt{\sum_{n=1}^{N}(x^{(n)}-\bar{x})^2}}$$

If the standard deviation for y and x are the same (e.g., through standardization described next), $\hat{w}_1 = \hat{r}_{xy}$. However, in general, the regression slope and PCC are *not* the same: while $\hat{r}_{xy}$ gives a bounded measure independent of the scale of the two variables, $\hat{w}_1$ measures the change in the expected value of y corresponding to 1-unit increase/decrease of x.

Update equations for simple linear regression

A special case arises when $\mathbf{y}$ and $\mathbf{x}$ are *centered*: $\dot{y}^{(n)} = y^{(n)} - \bar{y}$ and $\dot{x}^{(n)} = x^{(n)} - \bar{x}$, such that $\bar{\dot{y}} = 0$ and $\bar{\dot{x}} = 0$ because

$$\bar{\dot{y}} = \frac{1}{N}\sum_n \dot{y}^{(n)} = \frac{1}{N}\sum_n (y^{(n)}-\bar{y}) = \frac{1}{N}\sum_n y^{(n)} - \frac{1}{N}\sum_n \bar{y} = \bar{y} - \bar{y} = 0$$

As a result,

$$\hat{w}_1 = \frac{\sum_{n=1}^{N}(\dot{y}^{(n)}-\bar{\dot{y}})(\dot{x}^{(n)}-\bar{\dot{x}})}{\sum_{n=1}^{N}(\dot{x}^{(n)}-\bar{\dot{x}})^2} = \frac{\sum_{n=1}^{N}\dot{y}^{(n)}\dot{x}^{(n)}}{\sum_{n=1}^{N}(\dot{x}^{(n)})^2}; \quad \hat{w}_0 = \bar{y} - \hat{w}_1\bar{x} = 0$$

Without the intercept, the linear function becomes $\hat{y}^{(n)} = w_1\dot{x}^{(n)}$

If we only center the input variable such that $\bar{x} = 0$ but not y, then $w_0 = \bar{y}$ and the linear function $\hat{y}^{(n)} = w_1\dot{x}^{(n)} + \bar{y}$ measures the *change* from the average $\bar{y}$.

Also, PCC of centered x and y is the same as cosine similarity between x and y.

Standardization involves centering and scaling the input feature

Another special case arises if we further scale $\dot{x}$ by its standard deviation $\sigma_{\dot{x}} = \sqrt{\frac{1}{N}\sum_n(\dot{x}^{(n)})^2}$:

$$\tilde{x}^{(n)} = \dot{x}^{(n)}/\sigma_{\dot{x}}$$

then we have

$$\sigma^2_{\tilde{x}} = \frac{1}{N}\sum_n(\tilde{x}^{(n)})^2 = \frac{1}{N}\sum_n(\dot{x}^{(n)}/\sigma_{\dot{x}})^2 = \frac{1}{N}\sum_n(\dot{x}^{(n)})^2/\sigma^2_{\dot{x}} = \sigma^2_{\dot{x}}/\sigma^2_{\dot{x}} = 1$$

The regression coefficient becomes more simplified (while $w_0 = 0$ after centering x, y):

$$\hat{w}_1 = \frac{\sum_{n=1}^{N}\tilde{y}^{(n)}\tilde{x}^{(n)}}{\sum_{n=1}^{N}(\tilde{x}^{(n)})^2} = \frac{1}{N}\sum_{n=1}^{N}\tilde{y}^{(n)}\tilde{x}^{(n)} = \frac{1}{N}\tilde{\mathbf{x}}^\top\tilde{\mathbf{y}}$$

Together, the procedure of centering and scaling of the response and/or the input features is common practice known as **standardization** mainly to simplify computation and to make the model robust to different numerical scales of the input and response.

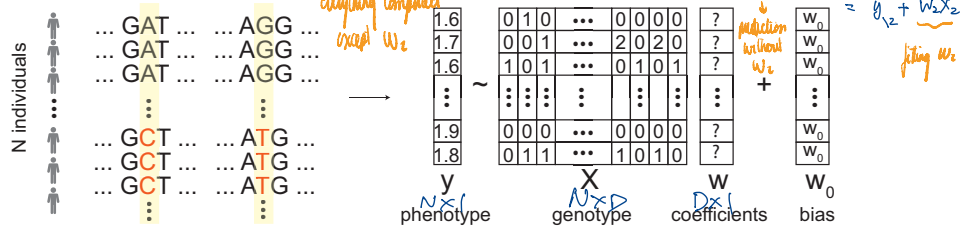
Outline

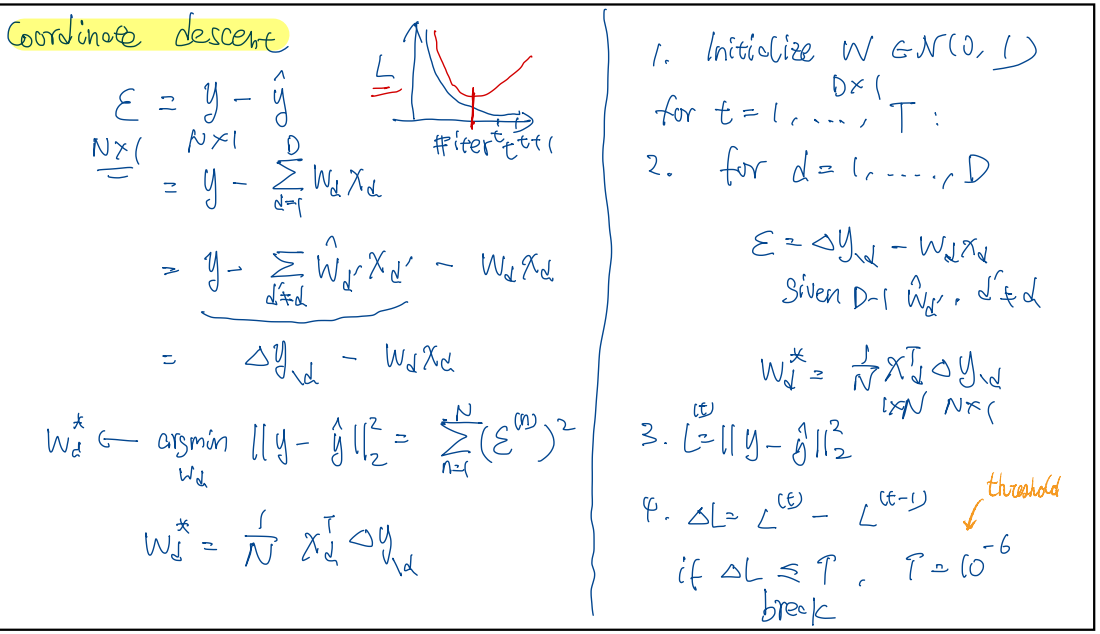
Multiple linear regression

- Goal: learn how to fit a *multiple regression* to predict the outcome variable y using *multiple* input features
- We can write the response as a weighted linear sum of the input features:

$$\hat{y}^{(n)} = f(\mathbf{x}^{(n)}; \mathbf{w}) = w_0 + w_1 x_1^{(n)} + w_2 x_2^{(n)} + \ldots + w_D x_D^{(n)} = w_0 + \sum_{d=1}^{D} w_d x_d^{(n)}$$

$$= \hat{w}_0 + \hat{w}_1 x_1^{(n)} + w_2 x_2^{(n)} + \ldots + \hat{w}_D x_D^{(n)} = \hat{w}_0 + \sum_{d \neq 2} \hat{w}_d x_d + w_2 x_2 = \hat{y}_{\setminus 2} + w_2 x_2$$





Multiple regression in matrix form

Suppose N training examples and D features. The data matrices are:

- Response: $\mathbf{y} \in \mathbb{R}^{N \times 1}$
- Input feature matrix: $[\mathbf{1}, \mathbf{X}] \in \mathbb{R}^{N \times (D+1)}$
- Regression coefficients: $[w_0; \mathbf{w}] \in \mathbb{R}^{(D+1) \times 1}$

We can rewrite the multiple regression function as:

$$\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_D^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_D^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & x_2^{(N)} & \ldots & x_D^{(N)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{bmatrix} + \begin{bmatrix} \epsilon^{(1)} \\ \epsilon^{(2)} \\ \vdots \\ \epsilon^{(N)} \end{bmatrix}$$

$$\mathbf{y} = \mathbf{1} w_0 + \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

Our goal is to find the ordinary least square (OLS) solution for the coefficients $\mathbf{w}$:

$$w_0^*, \mathbf{w}^* \leftarrow \arg\min_{w_0, \mathbf{w}} ||\mathbf{y} - \mathbf{X}\mathbf{w} - w_0||_2^2 = (\mathbf{y} - \mathbf{X}\mathbf{w} - w_0)^\intercal (\mathbf{y} - \mathbf{X}\mathbf{w} - w_0) = \sum_n (\epsilon^{(n)})^2$$

Notation: $||\mathbf{a}||_2 = \sqrt{\sum_i a_i^2}$ is L_2-norm and $||\mathbf{a}||_2^2 = \sum_i a_i^2$ is the squared of the L2-norm.

Multiple regression OLS derivation for the bias term w_0

Let the loss function be $J(\mathbf{w}, w_0) = (\mathbf{y} - \mathbf{X}\mathbf{w} - w_0)^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w} - w_0)$.

$$\begin{aligned}
\frac{\partial J(\mathbf{w})}{\partial w_0} &= \frac{\partial}{\partial w_0}\left((\mathbf{y} - \mathbf{X}\mathbf{w}) - \mathbf{1}w_0\right)^\intercal\left((\mathbf{y} - \mathbf{X}\mathbf{w}) - \mathbf{1}w_0\right) \\
&= \frac{\partial}{\partial w_0}\left\{(\mathbf{y} - \mathbf{X}\mathbf{w})^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) - (\mathbf{y} - \mathbf{X}\mathbf{w})^\intercal\mathbf{1}w_0 - w_0\mathbf{1}^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) + \mathbf{1}^\intercal\mathbf{1}w_0^2\right\} \\
&= \frac{\partial}{\partial w_0}(\mathbf{y} - \mathbf{X}\mathbf{w})^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) - \frac{\partial}{\partial w_0}2w_0\mathbf{1}^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) + \frac{\partial}{\partial w_0}Nw_0^2 \\
&= 0 - 2\mathbf{1}^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) + 2Nw_0 \\
&= -2\mathbf{1}^\intercal\mathbf{y} + 2\mathbf{1}^\intercal\mathbf{X}\mathbf{w} + 2Nw_0 \\
&= -2\sum_n y^{(n)} + 2(\sum_n \mathbf{x}^{(n)})\mathbf{w} + 2Nw_0 \overset{\text{set}}{=} 0
\end{aligned}$$

Solving $\frac{\partial J(\mathbf{w})}{\partial w_0} = 0$ for w_0:

$$\hat{w}_0 = \frac{1}{N}\sum_n y^{(n)} - \frac{1}{N}(\sum_n \mathbf{x}^{(n)})\mathbf{w} \equiv \bar{y} - \bar{\mathbf{x}}\mathbf{w}$$

Plugging in the OLS estimate for the bias term w_0 into the loss

$$\begin{aligned}
J(\mathbf{w}, \hat{w}_0) &= ||\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}\hat{w}_0||_2^2 \\
&= ||\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}(\bar{y} - \bar{\mathbf{x}}\mathbf{w})||_2^2 \\
&= ||(\mathbf{y} - \mathbf{1}\bar{y}) - (\mathbf{X}\mathbf{w} - \mathbf{1}\bar{\mathbf{x}}\mathbf{w})||_2^2 \\
&= ||(\mathbf{y} - \mathbf{1}\bar{y}) - (\mathbf{X} - \mathbf{1}\bar{\mathbf{x}})\mathbf{w}||_2^2 \\
&\equiv ||\dot{\mathbf{y}} - \dot{\mathbf{X}}\mathbf{w}||_2^2
\end{aligned}$$

where both the response and input features are mean-centered such that for each example n:

$$\dot{y}^{(n)} = y^{(n)} - \bar{y}$$
$$\dot{\mathbf{x}}^{(n)} = \mathbf{x}^{(n)} - \bar{\mathbf{x}}$$

Note: the second equation is element-wise subtraction: for each feature d,

$$\dot{x}_d^{(n)} = x_d^{(n)} - \bar{x}_d$$

Solving all multiple regression coefficients simultaneously

Here we can assume that either the data are centered (i.e., $\mathbf{y} = \dot{\mathbf{y}}, \mathbf{X} = \dot{\mathbf{X}}$) or we have $\mathbf{x}_0 = \mathbf{1}$. In the latter case, the estimate for w_0 is the first element in $\hat{\mathbf{w}}$.

$$\begin{aligned}
\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}}(\mathbf{y} - \mathbf{X}\mathbf{w})^\intercal(\mathbf{y} - \mathbf{X}\mathbf{w}) = \frac{\partial}{\partial \mathbf{w}}(\mathbf{y}^\intercal\mathbf{y} \underbrace{-\mathbf{y}^\intercal\mathbf{X}\mathbf{w} - \mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{y}}_{2\mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{y}} + \mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{X}\mathbf{w}) \\
&= \frac{\partial}{\partial \mathbf{w}}(\mathbf{y}^\intercal\mathbf{y} - 2\mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{y} + \mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{X}\mathbf{w}) = \underbrace{\frac{\partial}{\partial \mathbf{w}}\mathbf{y}^\intercal\mathbf{y}}_{0} - 2\frac{\partial}{\partial \mathbf{w}}\mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{y} + \frac{\partial}{\partial \mathbf{w}}\mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{X}\mathbf{w} \\
&\overset{\dagger}{=} -2\mathbf{X}^\intercal\mathbf{y} + 2\mathbf{X}^\intercal\mathbf{X}\mathbf{w} = 2\mathbf{X}^\intercal(\mathbf{X}\mathbf{w} - \mathbf{y}) = 2\mathbf{X}^\intercal(\hat{\mathbf{y}} - \mathbf{y})
\end{aligned}$$

†To get this equality, we make use of two general properties in matrix differentiation[1]:

$$\frac{\partial \mathbf{b}^\intercal\mathbf{a}}{\partial \mathbf{b}} = \mathbf{a}, \quad \frac{\partial \mathbf{b}^\intercal\mathbf{A}\mathbf{b}}{\partial \mathbf{b}} = 2\mathbf{A}\mathbf{b}$$

Setting the derivative to zero and solve for $\mathbf{w}$ gives the closed-form solution:

$$0 = -2\mathbf{X}^\intercal\mathbf{y} + 2\mathbf{X}^\intercal\mathbf{X}\mathbf{w} \quad \rightarrow \quad \mathbf{X}^\intercal\mathbf{X}\mathbf{w} = \mathbf{X}^\intercal\mathbf{y} \quad \rightarrow \quad \mathbf{w} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$$

[1] Matrix cookbook Section 2.4 Eq 69 & Eq 85

Summary Statistics (Sumstat)
Simple regression weights:

$$\text{PCC: } \hat{\beta}_{D\times 1} = \frac{1}{N}\tilde{X}^\intercal_{D\times N}\tilde{y}_{N\times 1}$$
$$R_{D\times D} = \frac{1}{N}\tilde{X}^\intercal\tilde{X}$$

Uniqueness of the OLS solution

$$\mathbf{w}^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y} \tag{9}$$

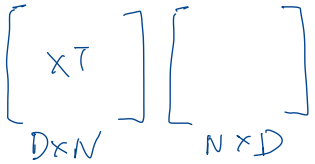
- The OLS solution is available when the $D \times D$ square matrix $\mathbf{A} = \mathbf{X}^\intercal\mathbf{X}$ is invertible.
- Matrix inverse $\mathbf{A}^{-1}$ can be computed by eigendecomposition:

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1} \implies \mathbf{A}^{-1} = \mathbf{Q}^{-1}\mathbf{\Lambda}^{-1}\mathbf{Q}$$

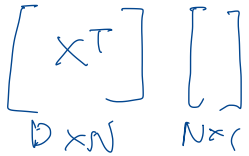
  where
  - $\mathbf{Q}$ is the square $D \times D$ matrix, whose i^{th} column is the i^{th} eigenvector
  - $\mathbf{Q}$ is also an orthonormal matrix, which means $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I} \implies \mathbf{Q}^\top = \mathbf{Q}^{-1}$
  - Each eigenvector $\mathbf{u}$ in $\mathbf{Q}$ is orthonormal to itself: $\mathbf{u}^\top\mathbf{u} = 1$ and two different eigenvectors are orthogonal to each other: $\mathbf{u}^\top\mathbf{v} = 0$
  - $\mathbf{\Lambda}$ is the diagonal matrix whose diagonal elements are the corresponding eignevalues, i.e., $\Lambda_{i,i} = \lambda_i$
- Therefore, $\mathbf{A}$ is not invertible if some of its eigenvalues are zeros, which can happen when two features are perfectly correlated, e.g., $\mathbf{x}_2 = 1 - \mathbf{x}_1$, meaning that the number of linearly independent columns (i.e., rank) is smaller than D.

* To make matrix invertible, + add noise ($+\epsilon$)
→ invertible makes it more stable for us to compute $\mathbf{w}^*$

Time complexity



$$\mathbf{w}^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$$



- The inner product $\mathbf{A} = \mathbf{X}^\intercal\mathbf{X}$ takes $O(ND^2)$
- The inversion of the $D \times D$ matrix $\mathbf{A}^{-1}$ takes $O(D^3)$
- Computing $\mathbf{X}^\intercal\mathbf{y}$ takes $O(ND)$

Therefore, the total time complexity is $O(ND^2 + D^3)$.

This can be very expensive or infeasible for large D (e.g., 1 million SNPs or 20,000 genes) and/or large N (e.g., half million individuals). For this reason, although we can derive closed-form solution, *Stochastic Gradient Descent* (SGD) is used for large dataset (Module 4.4).

or coordinate descent (CD)

Multivariate regression

We can also adapt the equation for multiple response variables. Instead of a response vector $\mathbf{y} \in \mathbb{R}^N$, we have a response matrix $\mathbf{Y} \in \mathbb{R}^{N \times C}$ for C response variables.
The multivariate regression function is:

$$\mathbf{Y} = \mathbf{X}\mathbf{W} \tag{10}$$

N×D D×C

where $\mathbf{W} \in \mathbb{R}^{D \times C}$.
The OLS solution for $\mathbf{W}$ is then:

$$\mathbf{W} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{Y} \tag{11}$$

Note here the OLS coefficient $\mathbf{w}_k$ for each response variable k is computed *independently* in the above solution. Therefore, the resulting OLS $\mathbf{W}$ is identical to fitting each $D \times 1$ regression coefficient $\mathbf{w}_k$ separately and then concatenate the C vectors together to form the matrix $\mathbf{W}$.

Outline

Probabilistic interpretation: Gaussian response variable

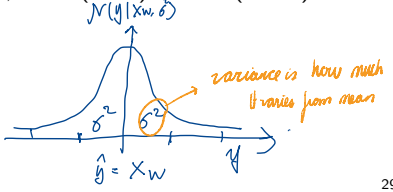
The general multivariate normal (MVN) distribution is:

$$p(\mathbf{y}|\mathbf{X}) = \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, \Sigma) = \det(2\pi\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{y}-\mathbf{X}\mathbf{w})^\intercal \Sigma^{-1}(\mathbf{y}-\mathbf{X}\mathbf{w})\right) \quad (12)$$

where Σ is a $N \times N$ covariance matrix between the N samples. If we assume the samples are *i.i.d.*, then Σ is a diagonal matrix $\sigma^2\mathbf{I}$, where $\mathbf{I}$ is an identity matrix:

$$\Sigma \overset{i.i.d.}{=} \begin{bmatrix} \sigma^2 & 0 & 0 & \dots & 0 \\ 0 & \sigma^2 & 0 & \dots & 0 \\ 0 & 0 & \sigma^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma^2 \end{bmatrix} = \sigma^2\mathbf{I}, \quad \Sigma^{-1} = \sigma^{-2}\mathbf{I}, \quad \det(2\pi\Sigma)^{-1/2} = (2\pi\sigma^2)^{-N/2}$$

where $\det(\mathbf{A})$ is the determinant of a square matrix.



variance is how much it varies from mean

The MVN can then be simplified as the product of individual Gaussians:

$$p(\mathbf{y}|\mathbf{X}) = (2\pi\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\mathbf{w})^\intercal(\mathbf{y}-\mathbf{X}\mathbf{w})\right)$$

$$= (2\pi\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2}\sum_n (y^{(n)} - \mathbf{x}^{(n)}\mathbf{w})^2\right)$$

Taking the logarithm of the likelihood, we have

$$\ln p(\mathbf{y}|\mathbf{X}) = \underbrace{-\frac{N}{2}\ln(2\pi\sigma^2)}_{\text{constant w.r.t. } \mathbf{w}} - \sum_n \left(\frac{1}{2\sigma^2}(y^{(n)} - \mathbf{x}^{(n)}\mathbf{w})^2\right)$$

$$\propto -\frac{1}{2\sigma^2}\sum_n (y^{(n)} - \mathbf{x}^{(n)}\mathbf{w})^2$$

$$= -\frac{1}{2\sigma^2}\boxed{||\mathbf{y} - \mathbf{X}\mathbf{w}||_2^2}$$

the smaller this is, the higher the prob.

The last equation indicates that the log likelihood is proportional to the negative SSE.

Maximum likelihood estimation

Given that,

$$\ln p(\mathbf{y}|\mathbf{X}) \propto -\frac{1}{2\sigma^2}\sum_n (y^{(n)} - \mathbf{x}^{(n)}\mathbf{w})^2 = -\frac{1}{2\sigma^2}J(\mathbf{w})$$

Since σ^2 is a constant, minimizing SSE w.r.t. $\mathbf{w}$ is equivalent to maximizing the Gaussian likelihood w.r.t. $\mathbf{w}$:

$$\arg\min_{\mathbf{w}} J(\mathbf{w}) = \arg\max_{\mathbf{w}} \ln p(\mathbf{y}|\mathbf{X})$$

The maximum likelihood estimator for $\mathbf{w}$ is then identical to the OLS $\mathbf{w}$:

$$\mathbf{w}^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$$

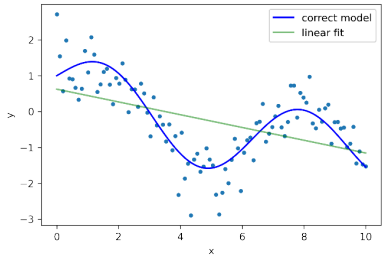
Optimal:

Bayesian inference of $P(W|X,y) = \frac{P(y|X,w)P(w)}{P(y|X)} = \frac{}{\int p(y,w|x)dw}$

Outline

Fitting non-linear data by transforming the input features

Consider the toy dataset below. It is obvious that our attempt to model y as a linear function of $\hat{y} = w_0 + xw_1$ would produce a bad fit.



Idea: we can create more features using the given features. For example, we can use a M-degree polynomial function and treat each power degree as a standalone feature:

$$\hat{y} = x^0 w_0 + x^1 w_1 + x^2 w_2 + \ldots + x^M w_M = \sum_{m=0}^{M} x^m w_m$$

intercept

Fitting non-linear data by transforming the input features

In general, we can transform input feature x with a (non-linear) **basis function** $\phi(x)$. The multiple linear regression operates on the basis-transformed features:

$$\hat{y} = \sum_{m=1}^{M} \phi_m(x) w_m = \Phi(x)\mathbf{w} \qquad (13)$$

$(1 \times (M+1))\ ((M+1) \times 1)$

We then simply replace all of the occurrences of x with $\Phi(x)$ in the OLS solution:

$$\hat{\mathbf{w}} = (\Phi(x)^\intercal \Phi(x))^{-1} \Phi(x)^\intercal \mathbf{y}, \quad \text{where}$$

$$\Phi(\mathbf{x}) = \begin{bmatrix} \phi_0(x^{(1)}) & \phi_1(x^{(1)}) & \ldots & \phi_M(x^{(1)}) \\ \phi_0(x^{(2)}) & \phi_1(x^{(2)}) & \ldots & \phi_M(x^{(2)}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x^{(N)}) & \phi_1(x^{(N)}) & \ldots & \phi_M(x^{(N)}) \end{bmatrix}$$

$N \times (M+1)$

Transforming high-dimensional features

- This can also be done on high-dimensional input feature $\mathbf{x}^{(n)}$ by transforming each feature with $x_d^{(n)}$ with say M-degree polynomial:

$$\Phi(x_d^{(n)}) = [(x_d^{(n)})^0, \ldots, (x_d^{(n)})^M]$$

(d: actual feature index) → Create M features out of d features

- We can then concatenate all transformed $D \times (M+1)$ features

$$\mathbf{x}^{(n)} = [(x_1^{(n)})^0, \ldots, (x_1^{(n)})^M, \ldots, (x_D^{(n)})^0, \ldots, (x_D^{(n)})^M] = [\Phi(x_1^{(n)}), \ldots, \Phi(x_D^{(n)})]$$

$(1 \times (D \times M))$

- This create a input matrix of dimension $N \times (D \times (M+1))$:

$$\Phi(\mathbf{X}) = \begin{bmatrix} 1 & \Phi(x_1^{(1)}) & \Phi(x_2^{(1)}) & \ldots & \Phi(x_D^{(1)}) \\ 1 & \Phi(x_1^{(2)}) & \Phi(x_2^{(2)}) & \ldots & \Phi(x_D^{(2)}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \Phi(x_1^{(N)}) & \Phi(x_2^{(N)}) & \ldots & \Phi(x_D^{(N)}) \end{bmatrix}$$

$D \times M + 1$

$W \rightarrow \hat{y}$

$(D \times M + 1) \times 1$

$N \times 1$

$\hat{y} = 1 w_0 + \Phi(x) W$

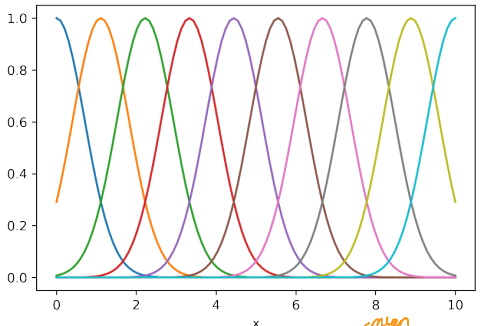
dimension?

↑ redundant so $(x_d^{(n)})^0$ is not needed

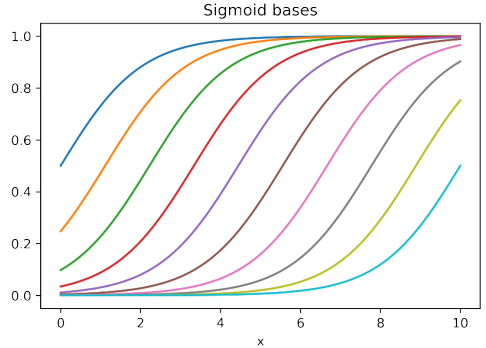
Nonlinear basis functions

There are many nonlinear basis functions. Using scalar input $x \in \mathbb{R}$ as an example,

Gaussian bases



Sigmoid bases



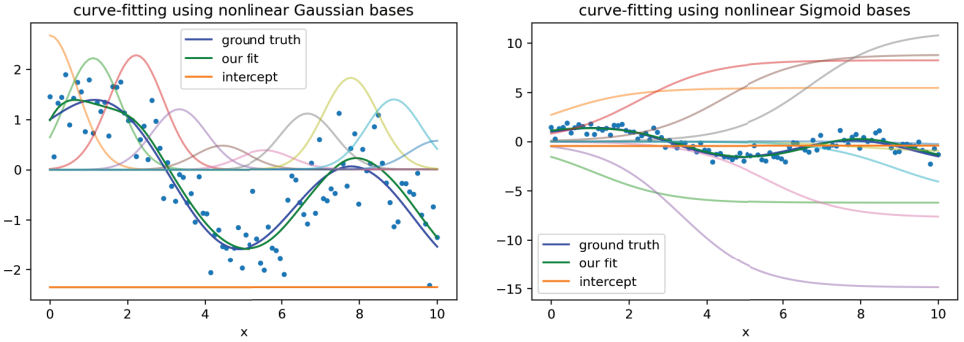
$$\phi_d(x) = \exp\left(-\frac{(x-\mu_d)^2}{2s^2}\right)$$

mean; variance

$$\phi_d(x) = \frac{1}{1+\exp\left(-\frac{(x-\mu_d)^2}{s}\right)}$$

where each type of basis function has a different mean $\mu_d \in [0, 10]$ and $s = 1$.

Linear regression with nonlinear basis (See Colab for the implementations)



In both plots, the green curve (our fit) is the sum of weighted Gaussian bases (i.e., the colorful curves) plus the intercept:

* Creating more features, so that it fits better

$$\hat{y} = w_0 + \sum_d w_d \phi_d(\mathbf{X})$$

Summary

- Response variable y is modelled as a weighted linear sum of the features $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ ← more than 1 feature
- We fit the model by minimizing the sum of squared errors (SSE) $\sum_n (y^{(n)} - \mathbf{X}^{(n)}\mathbf{w})^2$ → achieve global optimal
- OLS solution: $\mathbf{w}^* = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$ with $O(ND^2 + D^3)$ time complexity
- Minimizing SSE is equivalent to maximizing the log Gaussian likelihood given that the data points are $i.i.d$.
- Using non-linear basis functions to construct features can help model non-linear data. However, these non-linear basis functions are rigid and non-learnable.
- In Module 5.1, we will discuss neural networks which have *learnable* non-linear basis functions.

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

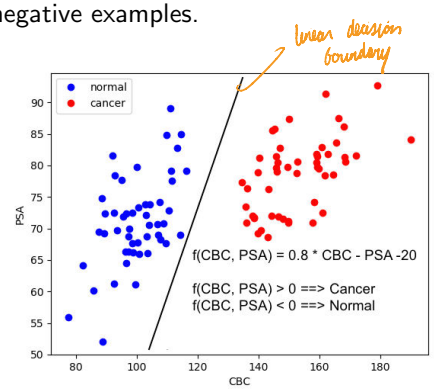
Understanding the following concepts

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

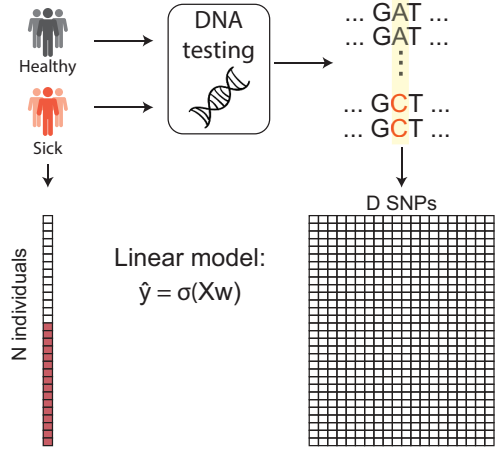
Outline

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

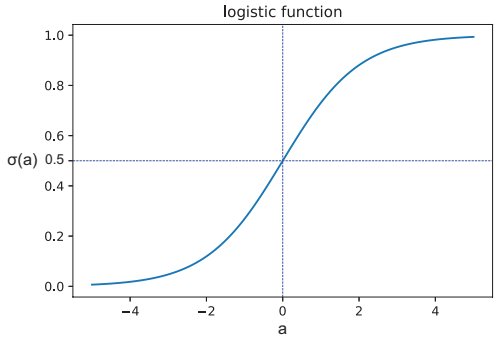
Logistic function transforms the real-value $a = \mathbf{xw} \in \mathbb{R}$ into $\hat{y} \in [0, 1]$, which can be interpreted as the *probability* being class 1.

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

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

$$\log \frac{\hat{y}}{1 - \hat{y}} = a \quad (2)$$

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



$\exp(0) = 1$

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

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

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

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

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

$$\frac{\hat{y}}{1 - \hat{y}} = \exp(a) \Rightarrow \quad a = \log \frac{\hat{y}}{1 - \hat{y}}$$

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:

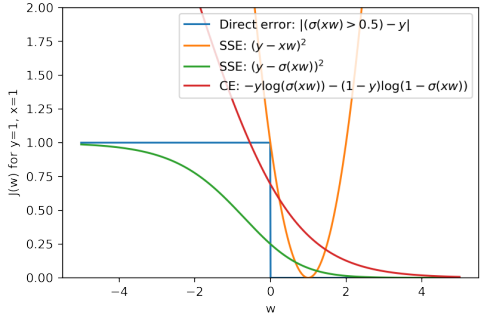
$$\mathcal{L}(\hat{y}, y) = |\mathbb{I}[\hat{y} > 0.5] - y|$$

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)

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

	$\hat{y} = 0$	$\hat{y} = 1$
$y = 0$	0	$+\infty$
$y = 1$	$+\infty$	0

$\log(0) = -\infty$

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)))
```

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

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:

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

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

$$\frac{\partial a}{\partial w_d} = \frac{\partial}{\partial w_d}\sum_d x_d w_d = x_d$$

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

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:

$$\nabla J(w) = \begin{bmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_D} \end{bmatrix} = \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{x}^\top(\hat{y}-y)$$

Logistic regression training algorithm by gradient descent

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

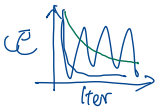
$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^{N} \frac{\partial J^{(n)}(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^{N} (\hat{y}^{(n)} - y^{(n)}) \mathbf{x}^{(n)} = \mathbf{X}^\intercal (\hat{\mathbf{y}} - \mathbf{y})$$

D×N N×1

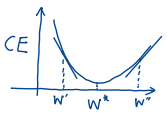
Unlike in the linear regression case, where we have closed-form solution for $\mathbf{w}$ (i.e., $\mathbf{w} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}$), to train a logistic regression, we cannot solve $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = 0$ for $\mathbf{w}$.

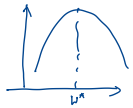
Compare with the gradients for the linear regression weights in Module 4.1.

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:



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





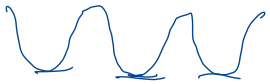
- We do subtraction because we want to minimize the error function by making the weights go in the opposite direction of error derivative.
- We multiply the gradients by a learning rate $\alpha \in [0, 1]$ to avoid overshooting the optimal values of $\mathbf{w}$.

Logistic regression training algorithm by gradient descent

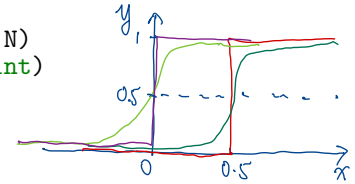
Algorithm 1 LogisticRegression.fit($\mathbf{X}$, $\mathbf{y}$, $\alpha = 0.005$, $\epsilon = 10^{-5}$, max_iter=10^5)

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
```



If $w = +\infty$

$$\hat{y} = \begin{cases} 0.5 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

$\hat{y} = \sigma(wx) = \frac{1}{1 + \exp(-wx)} = \frac{1}{1 + \exp(0)}$

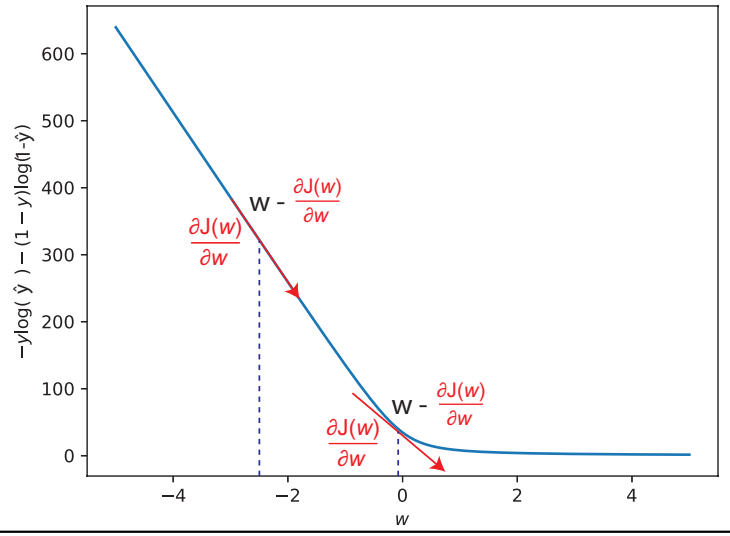
$a = wx + b$, $w = 1$, $T = 0.5$, $b = -0.5$

$a = wT + b = 0 \Rightarrow b = -wT$

$w = \frac{-b}{T}$

Cross-entropy as a function of w

$x \in [-5, 5]$; $y = x > 0.5$; $N = 50$



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. $\epsilon \sim Uniform([0, 10^{-5}])$
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

$\frac{\partial J(w)}{\partial w} \approx \frac{\nabla J(w)}{\nabla w}$

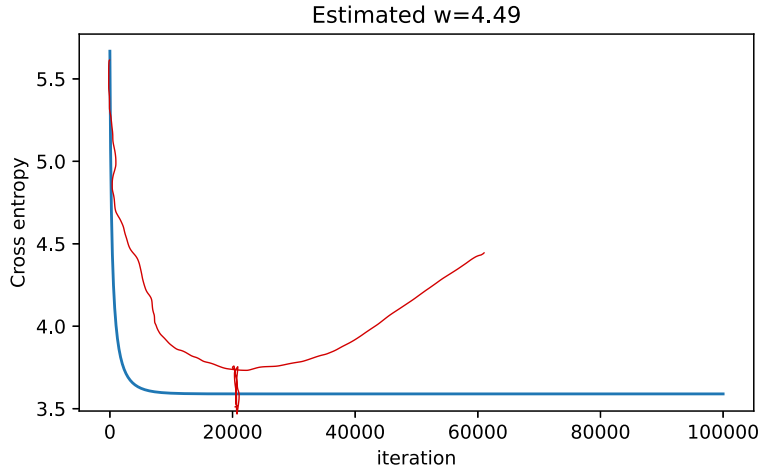
Python code for small perturbation test on a toy data (colab)

```
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

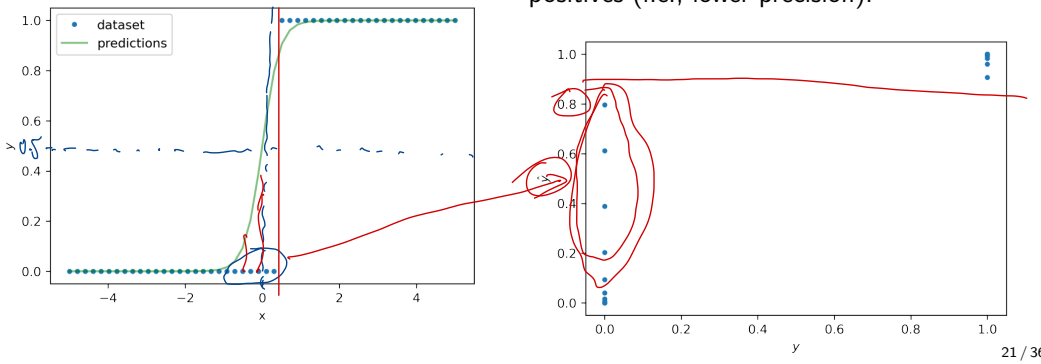


Visualizing predictions on 1D data

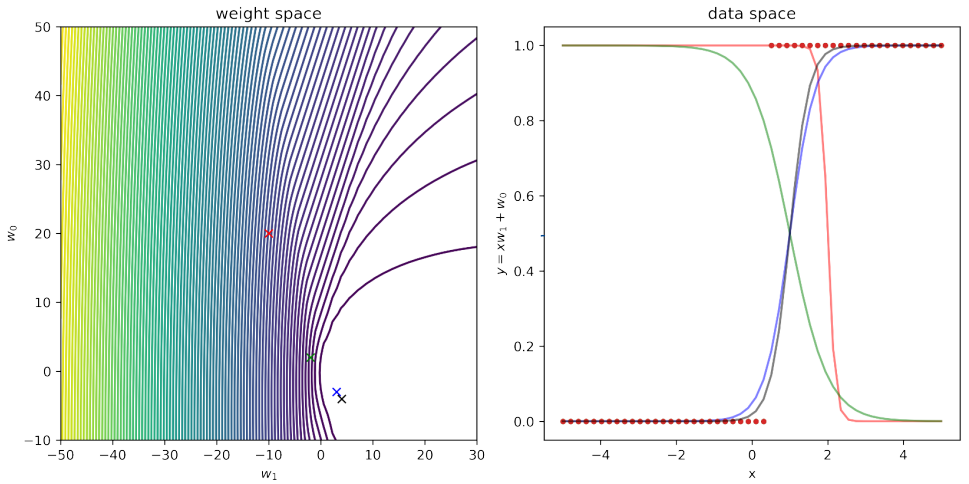
For the above toy data, we can also visualize the model prediction on the training set.

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:

$$p(\mathbf{y}|\pi) = \binom{N}{N_1} \prod_{n=1}^{N} \pi^{y^{(n)}} (1-\pi)^{1-y^{(n)}}$$

$$= \binom{N}{N_1} \pi^{\sum_n y^{(n)}} (1-\pi)^{\sum_n 1-y^{(n)}} = \binom{N}{N_1} \pi^{N_1} (1-\pi)^{N-N_1} \tag{5}$$

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

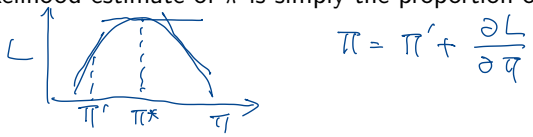
Suppose we are interested in knowing the Bernoulli rate π. We can directly maximize the log likelihood w.r.t. π. We do this by solving $\frac{\partial \mathcal{L}}{\partial \pi} = 0$ for π:

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

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?

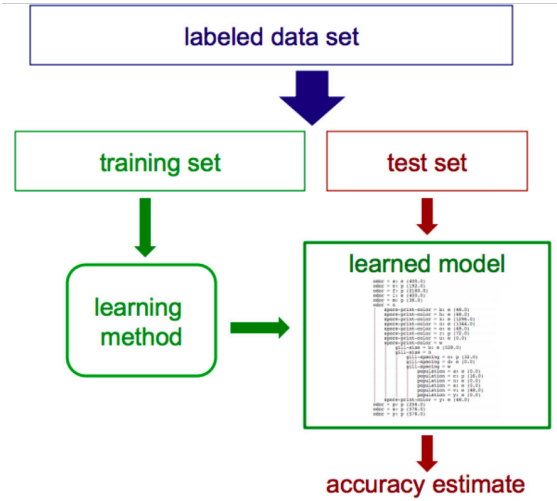
Outline

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
```

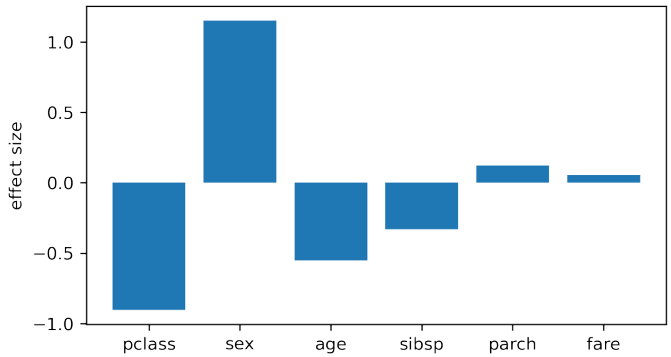
$$a = w_0 + w_{pclass}\mathbf{x}_{pclass} + w_{sex}\mathbf{x}_{sex} + w_{age}\mathbf{x}_{age} + w_{sidsp}\mathbf{x}_{sidsp} + w_{parch}\mathbf{x}_{parch} + w_{fare}\mathbf{x}_{fare}$$

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

- 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?

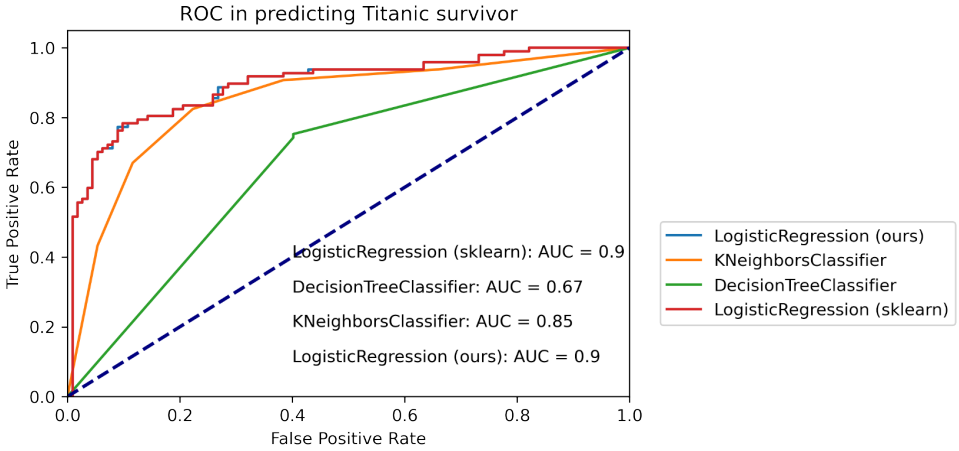
```
import matplotlib.pyplot as plt
plt.bar(list(effect_size.columns.values),
    effect_size.stack().tolist())
plt.ylabel("effect size")
plt.show()
```



Compare our logistic regression code with sklearn implementation

```
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.

Module 4.3 Lecture 10 & 11. Multi-class regression

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

Feb 11 & 13, 2025

Learning objectives

- Multi-class linear classifiers
  - Softmax function
  - Loss function
- Probabilistic view of multi-class classification
- Applications in toy data and Iris flower type classification

Reinforcing the concepts learned in the logistic regression lecture

The concepts of this lecture are very similar to the logistic regression lecture and serves as a way to enforce we have learned previously.

Outline

Multi-class classification

Classes are mutually exclusive (i.e., each data point belongs to one and exactly one of the K classes). An example is classifying Iris flowers using hand-crafted features.

(a)

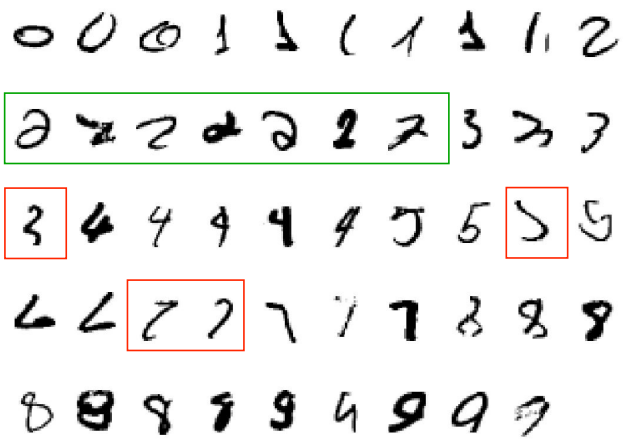
(b)

(c)

index	sepal length	sepal width	petal length	petal width	label
0	5.1	3.5	1.4	0.3	Setosa
1	4.9	3.0	1.7	0.2	Setosa
⋮	⋮	⋮	⋮	⋮	⋮
50	7.0	3.2	4.7	1.4	Versicolor
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	Virginica

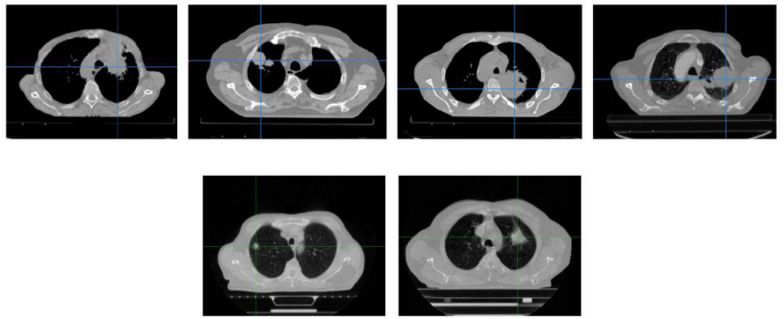
Multi-class classification example: MNIST digit prediction

Predicting digit $y \in \{0, \ldots, 9\}$ from 28×28 images of hand-written digits.



Multi-class example: predicting lung cancer stages

Predicting cancer stages of lung cancer patients using the Lung-RT dataset. Sample images are shown below. Top row: From left to right the stage I, II, IIIa and IIIb of squamous cell carcinomas. Bottom row: From left to right the stage I and II. The lung tumor position is indicated by the crossing lines.



Ubaldi, L., et al. "Strategies to develop radiomics and machine learning models for lung cancer stage and histology prediction using small data samples." Physica Medica 90 (2021): 13-22.

One-hot-encoding for categorical label and softmax function for prediction

Suppose we have C classes: $t^{(n)} \in \{1, \ldots, C\}$. It is computationally convenient to convert the categorical label to a **one-hot-encoding** vector where one of the K values is 1 and the rest of the values are zeros:

$$\mathbf{y}^{(n)} = \left[\mathbb{I}[t^{(n)} = 1], \quad \mathbb{I}[t^{(n)} = 2], \quad \ldots \mathbb{I}[t^{(n)} = C]\right] \quad \text{or} \quad y_c^{(n)} = \begin{cases} 1 & \text{if } t^{(n)} = c \\ 0 & \text{otherwise} \end{cases}$$

The linear function that predicts the score of the c^{th} class from a D-dimensional input feature vector $\mathbf{x}^{(n)}$ is:

$$a_c^{(n)} = \sum_{d=1}^{D} x_d^{(n)} w_{d,c} = \mathbf{x}^{(n)} \mathbf{w}_c + b_c$$

To convert $a_c^{(n)} \in \mathbb{R}$ into probability so that $\sum_c \hat{y}_c^{(n)} = 1$, we use **softmax** function:

$$\hat{y}_c^{(n)} = \frac{\exp(a_c^{(n)})}{\sum_{c'=1}^{C} \exp(a_{c'}^{(n)})}$$

For C classes, we will have C sets of linear coefficients $\mathbf{w}_c$ or $D \times C$ matrix $\mathbf{W}$.

Sigmoid function is a special case of softmax funtion when $C = 2$

The sigmoid function we learned in binary classification is a just special case of softmax when $C = 2$:

$$\hat{y}_1 = \frac{\exp(a_1)}{\exp(a_1) + \exp(a_2)} = \frac{1}{1 + \exp(a_2 - a_1)} = \frac{1}{1 + \exp(-(a_1 - a_2))}$$

where

$$a_1 - a_2 = \mathbf{x}\mathbf{w}_1 - \mathbf{x}\mathbf{w}_2 = \mathbf{x}(\mathbf{w}_1 - \mathbf{w}_2) \equiv \mathbf{x}\mathbf{w} = a$$

Therefore, for binary classification, we only need to learn *one* set of coefficients $\mathbf{w}$ that corresponds to the class $y = 1$.

The probability of $y = 0$ is simply $1 - \hat{y}$.

For $C > 2$ classes, we only need to learn $C - 1$ sets of coefficients

More generally, the C class probabilities need to sum to 1: $\sum_{c=1}^{C} \hat{y}_c = 1$. We only need to learn $C - 1$ sets of coefficients and derive the probability for the last class as follows:

$$\hat{y}_C = \frac{\exp(\mathbf{x}\mathbf{w}_C)}{\sum_{c=1}^{C} \exp(\mathbf{x}\mathbf{w}_c)} = \frac{1}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c - \mathbf{x}\mathbf{w}_C)} = \frac{1}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\underbrace{(\mathbf{w}_c - \mathbf{w}_C)}_{\mathbf{w}_c^*})}$$

$$= 1 - \frac{\sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c^*)}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c^*)} = 1 - \sum_{c'=1}^{C-1} \hat{y}_{c'}$$

where $\mathbf{w}_c^*$ is the redefined coefficients for $c \neq C$. Therefore, for prediction, we only need to fit $D \times (C - 1)$ coefficients $\mathbf{W}$. This is equivalent to setting $\mathbf{w}_C^* = \mathbf{0}$ (because $\exp(\mathbf{x}\mathbf{w}_C^*) = 1$).

However, learning the full $D \times C$ matrix allows us to associate D features with every class including the C^{th} class via $\mathbf{w}_C$. In the remaining lecture, we will assume that we are fitting the full $D \times C$ matrix $\mathbf{W}$.

Loss function is still cross entropy but for multiple classes

For multi-class loss, we can write the cross entropy as follows:

$$J(\mathbf{W}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c$$

entropy: $-\sum_c \hat{y}_c \log \hat{y}_c$

Similar to the binary cross-entropy, the multi-class cross entropy also has probabilistic interpretation (Section 6).

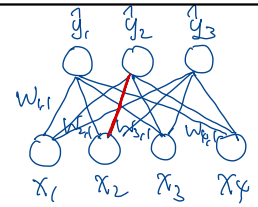
When $C = 2$, we see this loss becomes the same as the binary cross entropy loss:

$$\begin{aligned} J(\mathbf{W}) &= -y_1 \log \hat{y}_1 - y_2 \log \hat{y}_2 \\ &= -y_1 \log \hat{y}_1 - (1 - y_1) \log(1 - \hat{y}_1) \\ &= -y^* \log \hat{y}^* - (1 - y^*) \log(1 - \hat{y}^*) \end{aligned}$$

The last equality assumes $y^* \in \{0, 1\}$.

Outline

$W: D \times C$



$a_j = \sum_{d=1}^{D} x_d W_{dj}$

$\hat{y}_c = \frac{\exp(a_c)}{\sum_{c=1}^{C} \exp(a_c)}$

$J = \sum_{c=1}^{C} -y_c \log \hat{y}_c$

$\frac{\partial J}{\partial W_{dj}} = \frac{\partial}{\partial W_{dj}} \sum_{c=1}^{C} -y_c \log \hat{y}_c = \sum_c \underbrace{\frac{\partial -y_c \log \hat{y}_c}{\partial \hat{y}_c}}_{①} \underbrace{\frac{\partial \hat{y}_c}{\partial a_j}}_{②} \underbrace{\frac{\partial a_j}{\partial W_{dj}}}_{③}$

① $= -\frac{y_c}{\hat{y}_c}$

③ $= x_d$

② $\frac{\partial \hat{y}_c}{\partial a_j} = \frac{\partial}{\partial a_j} \exp(a_c) \left(\sum_c \exp(a_c)\right)^{-1}$

$\frac{\partial f(x)g(x)}{\partial x} = \frac{\partial f(x)}{\partial x} g(x) + f(x) \frac{\partial g(x)}{\partial x}$

$= \frac{\partial \exp(a_c)}{\partial a_j} \left(\sum_c \exp(a_c)\right)^{-1} + \exp(a_c) \frac{\partial}{\partial a_j} \left(\sum_c \exp(a_c)\right)^{-1}$

$= \frac{\partial \exp(a_c)}{\partial a_j} \left(\sum_c \exp(a_c)\right)^{-1} - \exp(a_c) \left(\sum_c \exp(a_c)\right)^{-2} \frac{\partial \exp(a_j)}{\partial a_j}$

$= \begin{cases} j = c: & \hat{y}_c - \exp(a_c)^2 \left(\sum_c \exp(a_c)\right)^{-2} = \hat{y}_c - \hat{y}_c^2 = \hat{y}_c(1 - \hat{y}_c) \\ j \neq c: & 0 - \frac{\exp(a_c)}{\sum_c \exp(a_c)} \frac{\exp(a_j)}{\sum_c \exp(a_c)} = -\hat{y}_c \hat{y}_j \end{cases}$

$= \hat{y}_c (\mathbb{I}[j=c] - \hat{y}_j)$

$\frac{\partial J}{\partial w_{d,j}} = \sum_c -\frac{y_c}{\hat{y}_c} \hat{y}_c (\mathbb{I}[j=c] - \hat{y}_j) x_d$

$= -\sum_c y_c (\mathbb{I}[j=c] - \hat{y}_j) x_d$

$= -\left(\sum_c y_c \mathbb{I}[j=c] - \sum_c y_c \hat{y}_j \right) x_d$

$= -(y_j - \hat{y}_j) x_d$

$= (\hat{y}_j - y_j) x_d$

$f: \mathbb{R}^D \to \mathbb{R}^C$

$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial w_{1,1}} & \frac{\partial J}{\partial w_{1,2}} & \cdots & \frac{\partial J}{\partial w_{1,C}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial w_{D,1}} & \frac{\partial J}{\partial w_{D,2}} & \cdots & \frac{\partial J}{\partial w_{D,C}} \end{bmatrix}$ $D \times C$

$= x^T (\hat{y} - y)$ $D \times 1 \quad 1 \times C$

N samples

$= X^T (\hat{Y} - Y)$ $D \times N \quad N \times C$

Gradient derivation of multi-class linear regression

For simplicity, we will work with one example and omit the superscript n for now. Our goal is to minimize CE w.r.t. $w_{d,j} \in \mathbf{W} \in \mathbb{R}^{D \times C}$, where d indexes the d^{th} feature and c indexes the c^{th} class:

$$J(\mathbf{W}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c, \quad \text{where} \quad \hat{y}_c = \frac{\exp(a_c)}{\sum_c \exp(a_c)} \quad \text{and} \quad a_c = \sum_{d=1}^{D} x_d w_{d,c}$$

Using chain rule, we can express the partial derivative w.r.t. $w_{d,j}$ for the j^{th} class and d^{th} feature:

$$\frac{\partial J}{\partial w_{d,j}} = \frac{\partial}{\partial w_{d,j}} -\sum_{c=1}^{C} y_c \log \hat{y}_c = -\sum_{c=1}^{C} \frac{\partial}{\partial w_{d,j}} y_c \log \hat{y}_c = \sum_{c=1}^{C} \frac{\partial -y_c \log \hat{y}_c}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial a_j} \frac{\partial a_j}{\partial w_{d,j}} \tag{1}$$

Note that here we need to sum over all classes to compute the gradient for $w_{d,j}$ because the gradients when $j = c$ and $j \neq c$ are different (more details follow). Same as in the binary logistic regression, we solve each partial derivative one by one.

$$\frac{\partial}{\partial \hat{y}_c} -y_c \log \hat{y}_c = -\frac{y_c}{\hat{y}_c} \tag{2}$$

$$\frac{\partial a_j}{\partial w_{d,j}} = \frac{\partial}{\partial w_{d,j}} \sum_d x_d w_{d,j} = x_d \tag{3}$$

$$\frac{\partial \hat{y}_c}{\partial a_j} = \frac{\partial}{\partial a_j} \frac{\exp(a_c)}{\sum_c \exp(a_c)} = \frac{\partial}{\partial a_j} \exp(a_c) \left[\sum_c \exp(a_c) \right]^{-1} + \exp(a_c) \frac{\partial}{\partial a_j} \left[\sum_c \exp(a_c) \right]^{-1}$$

If $j = c$:

$$\frac{\partial \hat{y}_c}{\partial a_j} = \frac{\exp(a_c)}{\sum_c \exp(a_c)} - [\exp(a_c)]^2 \left[\sum_c \exp(a_c) \right]^{-2}$$

$$= \hat{y}_c - \left[\frac{\exp(a_c)}{\sum_c \exp(a_c)} \right]^2 = \hat{y}_c - \hat{y}_c^2$$

If $j \neq c$:

$$\frac{\partial \hat{y}_c}{\partial a_j} = -\exp(a_c) \exp(a_j) \left[\sum_c \exp(a_c) \right]^{-2}$$

$$= -\frac{\exp(a_c)}{\sum_c \exp(a_c)} \frac{\exp(a_j)}{\sum_c \exp(a_c)} = -\hat{y}_c \hat{y}_j$$

Therefore,

$$\frac{\partial \hat{y}_c}{\partial a_j} = \begin{cases} \hat{y}_c(1 - \hat{y}_c) & \text{if } j = c \\ -\hat{y}_c \hat{y}_j & \text{if } j \neq c \end{cases} = \hat{y}_c(\mathbb{I}[j = c] - \hat{y}_j) \tag{4}$$

Gradient of multi-class linear regression

Plugging (2), (4), (3) into (1), we have

$$\frac{\partial J}{\partial w_{d,j}} = \sum_c \frac{\partial -y_c \log \hat{y}_c}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial a_j} \frac{\partial a_j}{w_{d,j}}$$

$$= \sum_c -\frac{y_c}{\hat{y}_c} \hat{y}_c (\mathbb{I}[j = c] - \hat{y}_j) x_d$$

$$= \sum_c y_c (\hat{y}_j - \mathbb{I}[j = c]) x_d$$

$$= \left[\underbrace{\left(\sum_c y_c \right)}_{1} \hat{y}_j - \underbrace{\sum_c y_c \mathbb{I}[j = c]}_{y_j} \right] x_d$$

$$= (\hat{y}_j - y_j) x_d$$

Recall the gradient for logistic: $\frac{\partial J}{\partial w_d} = (\hat{y} - y) x_d$

Now we recover the subscript for N data points $\{\mathbf{x}^{(n)}, \mathbf{y}^{(n)}\}^N$. We can represent the gradients over *all D features* for *all C classes* as a *Jacobian matrix*:

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

$$= \mathbf{X}^\top (\hat{\mathbf{Y}} - \mathbf{Y}) \tag{5}$$

where $(\mathbf{x}^{(n)})^\top$ is a $D \times 1$ vector, $\hat{\mathbf{y}}^{(n)}$ and $\mathbf{y}^{(n)}$ are $1 \times C$ vectors, $\mathbf{X} \in \mathbb{R}^{N \times D}$, and $\mathbf{Y}$ is a $N \times C$ binary matrix, where each row is a one-hot-encoded vector.

Pop quiz: is Eq (5) equivalent to updating each $\mathbf{w}_c$ separately as C separate logistic regression models?

No, one is concatenated, while the other takes into account the other classes

Multi-class regression training algorithm by gradient descent

Similar to the logistic regression (and unlike linear regression), there is no closed form solution to the coefficients $\mathbf{W}$.
To update the multi-class regression model, we perform **gradient descent** by *subtracting* the gradients from the existing weight *iteratively*:

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

- We subtract the gradients from the weights because we want to minimize the error function.
- We multiply the gradients by a learning rate $\alpha \in [0, 1]$ to avoid overshooting the optimal values.

Multi-class regression training algorithm by gradient descent

Algorithm 1 MulticlassRegression.fit($\mathbf{X}$, $\mathbf{Y}$, $\alpha = 0.005$, $\epsilon = 10^{-5}$, max_iters=10^5)

Randomly initialize regression coefficients $w_{d,c} \sim \mathcal{N}(0, 1) \forall d \forall c$
for niter $= 1 \ldots$ max_iters **do**
  $\mathbf{W}^{\{t\}} \leftarrow \mathbf{W}^{\{t-1\}} - \alpha \mathbf{X}^\top (\hat{\mathbf{Y}} - \mathbf{Y})$
  for n $= 1, \ldots,$ N **do**
    for c $= 1, \ldots,$ C **do**
      $\hat{y}_c^{(n)} = \exp(\mathbf{x}^{(n)} \mathbf{w}_c^{\{t\}}) / \sum_c \exp(\mathbf{x}^{(n)} \mathbf{w}_c^{\{t\}})$
    end for
  end for
  $J(\mathbf{W}^{\{t\}}) = -\sum_n \sum_c y_c^{(n)} \log(\hat{y}_c^{(n)})$
  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

Key implementation of the Multi-class prediction class

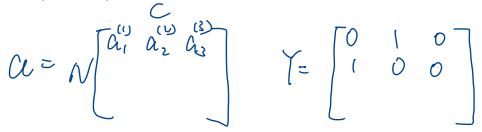
```
1  class Multinomial_logistic:
2      def __init__(self, nFeatures, nClasses):
3          self.W = np.random.rand(nFeatures, nClasses)
4
5      def predict(self, X):
6          y_pred = np.exp(np.matmul(X, self.W))
7          return y_pred / y_pred.sum(axis=1).reshape(X.shape[0], 1)
8
9      def grad(self, X, y):
10         return np.matmul(X.transpose(), self.predict(X) - y)
11
12     def ce(self, X, y):
13         return -np.sum(y * np.log(self.predict(X))) // element-wise
14
15     def fit(self, X, y, lr=0.005, niter=100):
16         for i in range(niter):
17             self.W = self.W - lr * self.grad(X, y)
```


Outline

Toy data simulation

- Simulate a 3-class response $\mathbf{y}$ using a 4-dimensional binary input $\mathbf{X} \in \{0,1\}^{N \times D}$, which are sampled from Bernoulli with 0.5 rate: $x_d^{(n)} \sim \pi^{x_d^{(n)}}(1-\pi)^{1-x_d^{(n)}}$
- Features 2, 0, and (1,3) are the causal features of class 0, 1, and 2, respectively.
- The 4×3 true causal effects $\mathbf{W}$ are also binary with $w_{d,c} = 1$ if feature d is causal on class c, otherwise $w_{d,c} = 0$.
- For each example, its class label is set to be the class with the highest linear combination $y^{(n)} \leftarrow \arg\max_c \mathbf{x}^{(n)}\mathbf{w}_c$. Note that $\mathbf{y}^{(n)}$ is one-hot encoded.

```
N = 150
X = np.column_stack((np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N)))
W_true = np.array([[0,1,0],
                   [0,0,1],
                   [1,0,0],
                   [0,0,1]])
a = np.matmul(X, W_true)
y = np.zeros_like(a)
y[np.arange(len(a)), a.argmax(1)] = 1
```



Splitting the data into 1/3 training, 1/3 validation, 1/3 testing

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size = 0.33, random_state=1, shuffle=True)

X_train, X_valid, y_train, y_valid =
↪   model_selection.train_test_split(
    X_train, y_train, test_size = 0.5, random_state=1, shuffle=True)
```

Here the validation is being used for monitoring whether the model starts to overfit.

Recall: we can verify gradient calculation via 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. $w_{d,c}^{(+)} = w_{d,c} + \epsilon$
3. $w_{d,c}^{(-)} = w_{d,c} - \epsilon$
4. $\nabla w_{d,c} = \frac{J(w_{d,c}^{(+)}) - J(w_{d,c}^{(-)})}{2\epsilon}$ (numerically estimated gradient)
5. $\frac{(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} - \nabla w_{d,c})^2}{(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} + \nabla w_{d,c})^2}$ must be small (e.g., 10^{-8}) otherwise your gradient calculation and/or your loss function is incorrect

```
mlr = Multinomial_logistic(D, C)
print(mlr.check_grad(X_train, y_train)) # expected very small value
```


In Python

```
def check_grad(self, X, y):
    N, C = y.shape
    D = X.shape[1]
    diff = np.zeros((D, C))
    W = self.W.copy()

    for i in range(D):
        for j in range(C):
            epsilon = np.zeros((D, C))
            epsilon[i, j] = np.random.rand() * 1e-4

            self.W = self.W + epsilon
            J1 = self.ce(X, y)
            self.W = W # restore original W

            self.W = self.W - epsilon
            J2 = self.ce(X, y)
            self.W = W # restore original W

            numeric_grad = (J1 - J2) / (2 * epsilon[i, j])
            derived_grad = self.grad(X, y)[i, j]

            diff[i, j] = np.square(derived_grad - numeric_grad).sum() /
            np.square(derived_grad + numeric_grad).sum()
    return diff.sum()
```

Modifying the `fit` to monitor training and validation errors

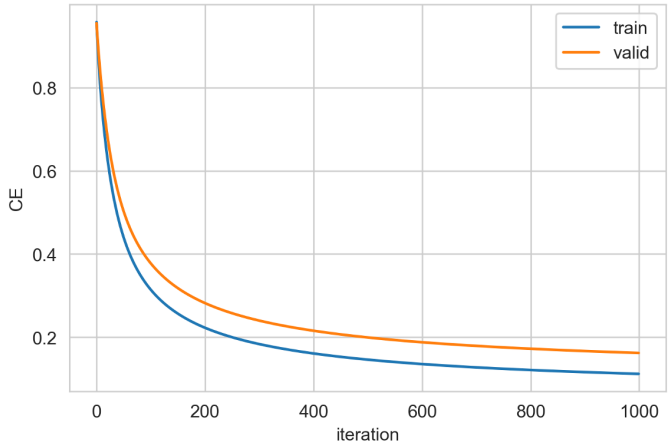
```
class Multinomial_logistic:
    # modify it to add stopping criteria (what can you think of?)
    def fit(self, X, y, X_valid=None, y_valid=None, lr=0.005,
    ↪  niter=100):
        losses_train = np.zeros(niter)
        losses_valid = np.zeros(niter)
        for i in range(niter):
            self.W = self.W - lr * self.grad(X, y)
            loss_train = self.ce(X, y)
            losses_train[i] = loss_train
            if X_valid is not None and y_valid is not None:
                loss_valid = self.ce(X_valid, y_valid)
                losses_valid[i] = loss_valid
                print(f"iter {i}: {loss_train:.3f};
                ↪  {loss_valid:.3f}")
            else:
                print(f"iter {i}: {loss_train:.3f}")
        return losses_train, losses_valid
```


Fitting multiclass logistic regression for 1000 iterations with lr set to 0.005

```
ce_train, ce_valid = mlr.fit(X_train, y_train, X_valid, y_valid,
↪  niter=1000)

plt.clf()
plt.plot(ce_train/X_train.shape[0], label='train') // avg train CE
plt.plot(ce_valid/X_valid.shape[0], label='valid') // avg valid CE
plt.xlabel("iteration")
plt.ylabel("CE")
plt.legend()
# plt.show()
plt.savefig("training_ce.png", bbox_inches="tight", dpi=300)
```


Fitting multiclass logistic regression for 1000 iterations with lr set to 0.005

From the plot, we can see that the training and validation error curves both continue to decrease. Therefore, there is no sign of overfitting.



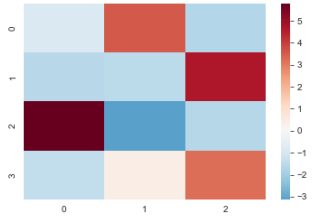
Evaluating prediction accuracy

Prediction accuracies are 100% for training, validation, and test datasets, which is not surprising since our data are generated by a linear function.

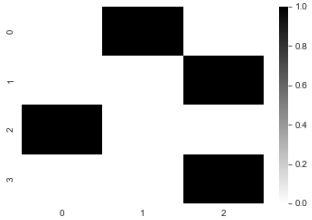
```
def evaluate(y, y_pred):
    accuracy = sum(y_pred.argmax(axis=1) == y.argmax(axis=1))
    accuracy = accuracy / y.shape[0]
    return accuracy

train_accuracy = evaluate(mlr.predict(X_train), y_train)
valid_accuracy = evaluate(mlr.predict(X_valid), y_valid)
test_accuracy = evaluate(mlr.predict(X_test), y_test)

print(train_accuracy) # 1
print(valid_accuracy) # 1
print(test_accuracy) # 1
```

Visualizing the linear coefficients using `seaborn.heatmap`

Fitted coefficient $\hat{\mathbf{W}}$:



True coefficient $\mathbf{W}$:

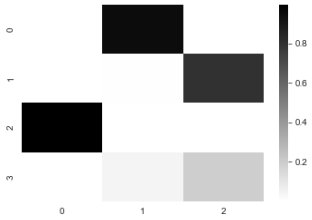


Transforming the fitted coefficient by softmax:

$$\hat{w}^*_{d,c} = \frac{\exp(\hat{w}_{d,c})}{\sum_d \exp(\hat{w}_{d,c})} \implies$$

```
W_hat = np.exp(W_hat)
W_hat = W_hat /
    ↪   W_hat.sum(axis=1)[:,None]
```

Softmaxed fitted coefficient $\hat{\mathbf{W}}^*$:



Outline

Iris flower classification

C = 3, D = 4, N = 150

(a) (b) (c)

index	sepal length	sepal width	petal length	petal width	label
0	5.1	3.5	1.4	0.3	Setosa
1	4.9	3.0	1.7	0.2	Setosa
⋮	⋮	⋮	⋮	⋮	⋮
50	7.0	3.2	4.7	1.4	Versicolor
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	Virginica

Iris flower classification 1/3 training, 1/3 validation, 1/3 testing

- After 1000 iterations, we observe a slight overfitting as the training error continues to decrease but the validation error starts to increase at around 550 iterations.
- The best model with the lowest validation error is at iteration `optimal_niter = ce_valid.argmin()` (i.e., at iteration 538).
- Retraining the model for `optimal_niter`, we obtained 98% accuracy on training; 90% on validation, and 96% on the testing sets.

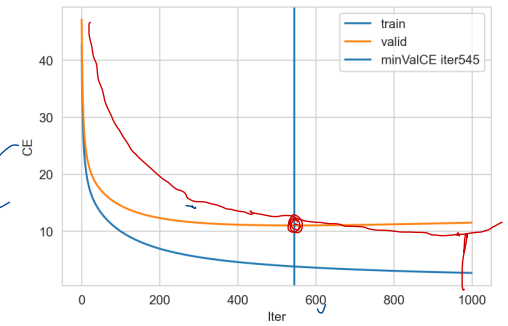
$\hat{W} = (X^T X)^{-1} X^T y$

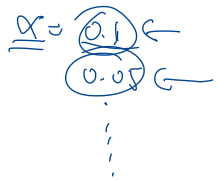
$\nabla W = X^T (\hat{y} - y)$

$\hat{y} = XW$

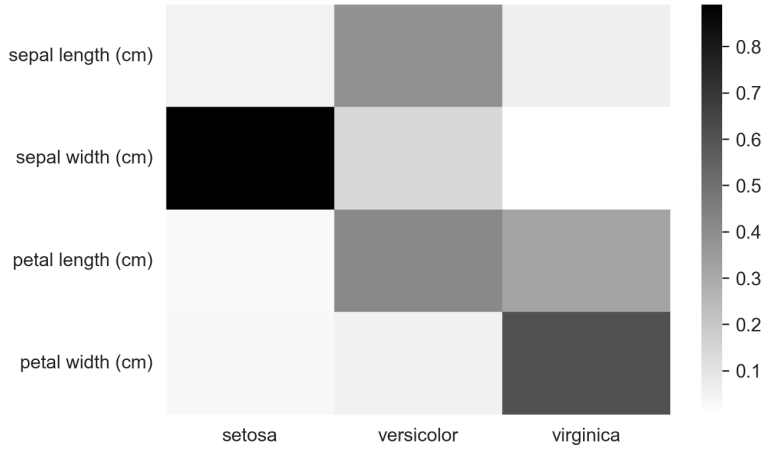
SSE

$SSE = \|\hat{y} - y\|_2^2$





Feature-flower types association via $softmax(\hat{\mathbf{W}})$



Outline

Maximizing log multinomial likelihood is equivalent to minimizing CE

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

$$p(y|\boldsymbol{\pi}) = \prod_c \pi_c^{[y=c]} \tag{6}$$

Representing y as the one-hot encoded binary vector, replacing π_c by $\hat{y}_c$, and taking the logarithm of the Categorical likelihood give:

$$\log p(\mathbf{y}|\hat{\mathbf{y}}) = \sum_c y_c \log \hat{y}_c \tag{7}$$

Multinomial distribution models the outcome of multiple categorical trials (i.e., N training examples):

$$\log p(\mathbf{Y}|\hat{\mathbf{Y}}) = \sum_{n=1}^{N}\sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)} = -J(\mathbf{W}) \tag{8}$$

For this reason, the multi-class logistic is often known as *multinomial logistic regression* or *multinomial regression*. Pop quiz: Which line(s) do we need to change in Slide 16 if we are to maximize the Categorical likelihood?

Summary

- Multiclass prediction model:
  - Multiclass function: softmax
  - Cross-entropy (CE) loss
  - Gradient descent
  - The $D \times C$ $\mathbf{W}$ linear coefficient matrix offers interpretability for associations between features and classes
- Probabilistic interpretation
  - Categorical or multinomial distribution
  - Maximum likelihood estimate of Multinomial is equivalent to minimizing CE loss
  - Recall:
    - Logistic regression: MLE of log Bernoulli likelihood is equivalent to minimizing CE loss
    - Linear regression: MLE of log Gaussian likelihood is equivalent to minimizing SSE loss
- Applications and interpretations of logistic regression linear coefficients on the toy and Iris flower datasets

Outline

Midterm exam on Feb 18 in class

Format:

- 1 hour in-class;
- Open book, calculator allowed but not digital one, no tablet, cell phone or laptop
- Format is the same as the practice midterm;
- Hand-written; $\max(\text{Correct} - \text{Incorrect}, 0)$
- 20% MC (6 questions); 30% MS (4 questions); 50% SA (6 questions).

Understand the basics. For example,

- Basic ML experiments: training, validation, testing
- K-fold cross validation
- Model evaluation: accuracy, confusion table, TRP, FPR, Precision, Recall, ROC
- Overfitting
- Algorithms and costs functions for KNN, Decision tree, linear regression, basis functions, logistic regression, and multiclass regression
- Partial derivatives of linear models