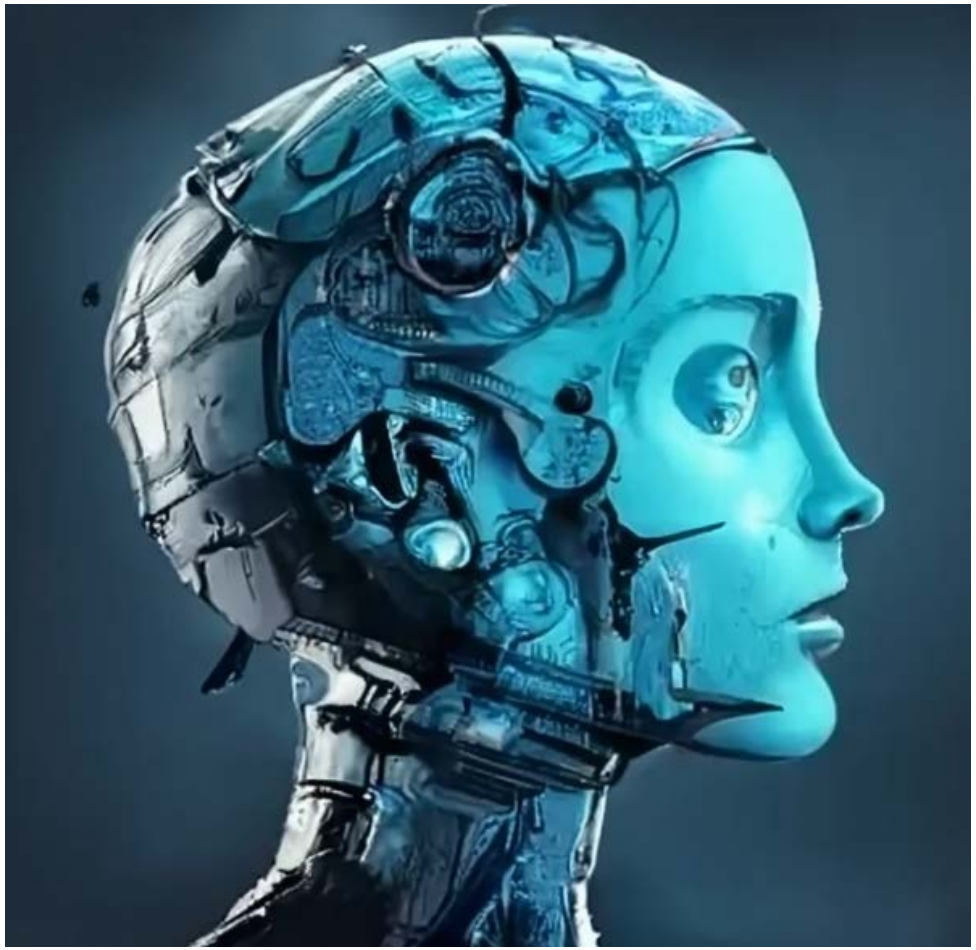
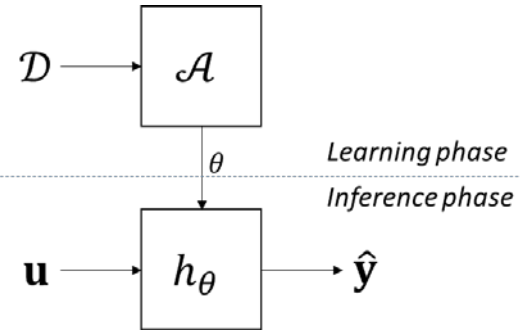
Machine Intelligence

Classification

Dr. Cory Merkel



Classification Overview



- So far, we have looked at problems where $\hat{y} = h_\theta(\mathbf{x}) \in \mathbb{R}$
  - We have called this regression

- Often, we will be interested another type of problem, where $\hat{y} = h_\theta(\mathbf{x}) \in \{c_1, c_2, \ldots c_M\}$ (output is one of M possibilities, or *classes*
  - This is called classification
  - Not much has changed: We went from a continuous to a discrete output
  - In this case, we call h a discriminant function

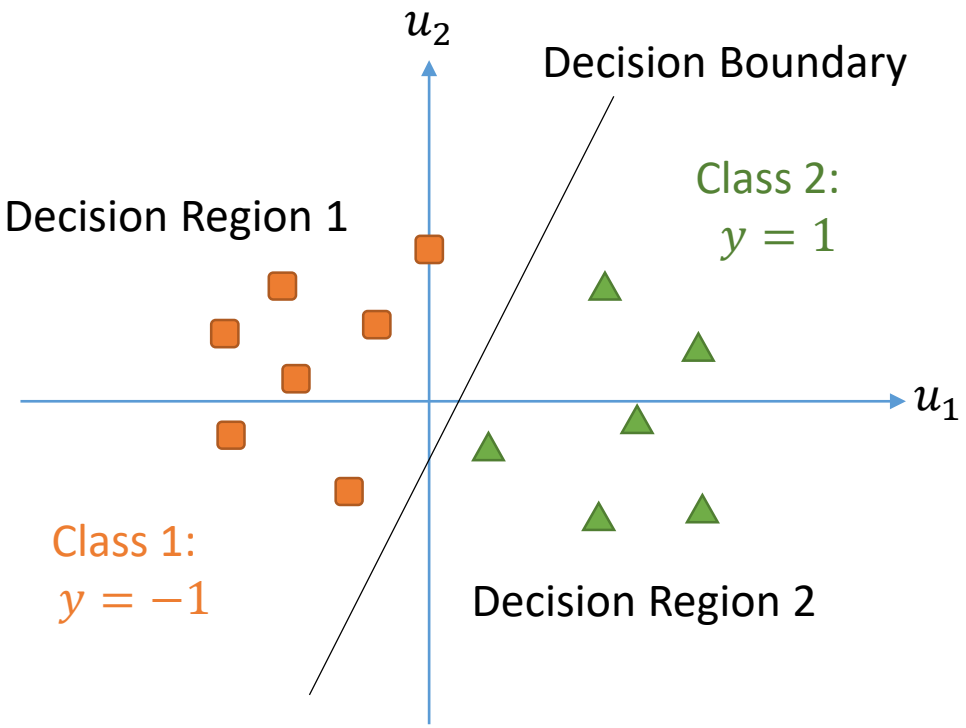
{0,1,2,…,8,9}?

{seizure, no seizure}?

Tumor size, Tumor shape

{benign, malignant}

Classification Overview

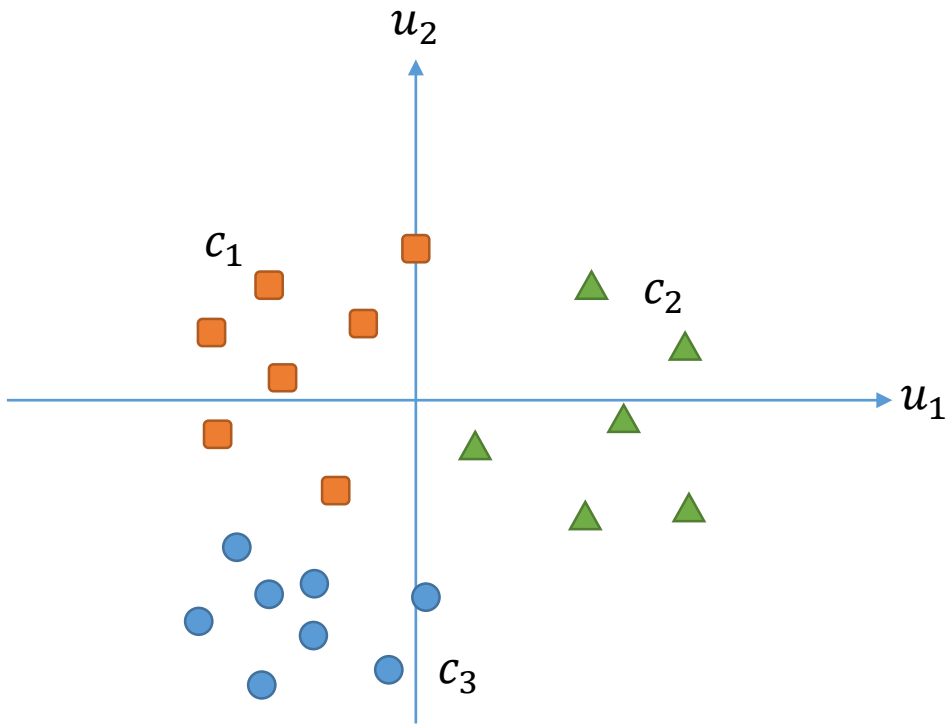


- For a 2-class classification problem, we may encode the expected outputs as $\{c_1, c_2\} = \{-1, 1\}$
- A linear discriminant function can be used to separate the classes:
  - $\hat{y} = \begin{cases} -1, w_0 + w_1 u_1 + \cdots + w_N u_N \leq 0 \\ 1, \text{otherwise} \end{cases}$

- We call the hyperplane $\mathcal{H}: w_0 + w_1 u_1 + \cdots + w_N u_N = 0$ the *decision boundary*
  - The two half-spaces on either side of the decision boundary are decision regions (everything in a decision region gets classified the same)
  - When a single hyperplane perfectly divides the data into the two classes, we say the data are *linearly separable*

Classification Overview

- What if we have more than 2 classes? $\{c_1, c_2, c_3\} = ?$



Classification Overview

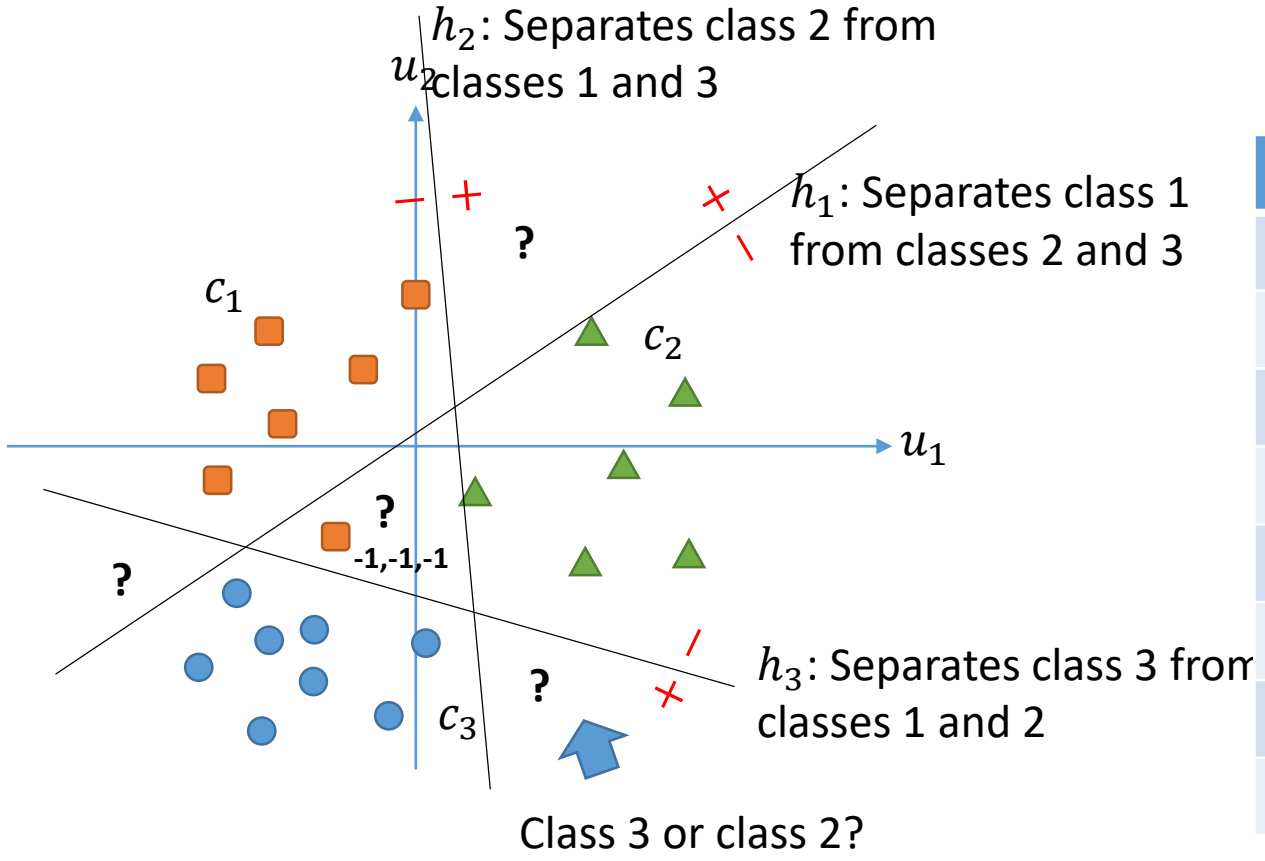
- For M classes, we could try to create M classifiers.
  - Each classifier separates class c_i from all other classes

Called one vs. all classifier

Only works, in general, when each class is linearly separable from the union of all other classes.

- Note that class 1 is not linearly separable from classes 2 and 3

Some regions become ambiguous



h_1	h_2	h_3	h
-1	-1	-1	?
-1	-1	1	3
-1	1	-1	2
-1	1	1	?
1	-1	-1	1
1	-1	1	?
1	1	-1	?
1	1	1	?

Classification Overview

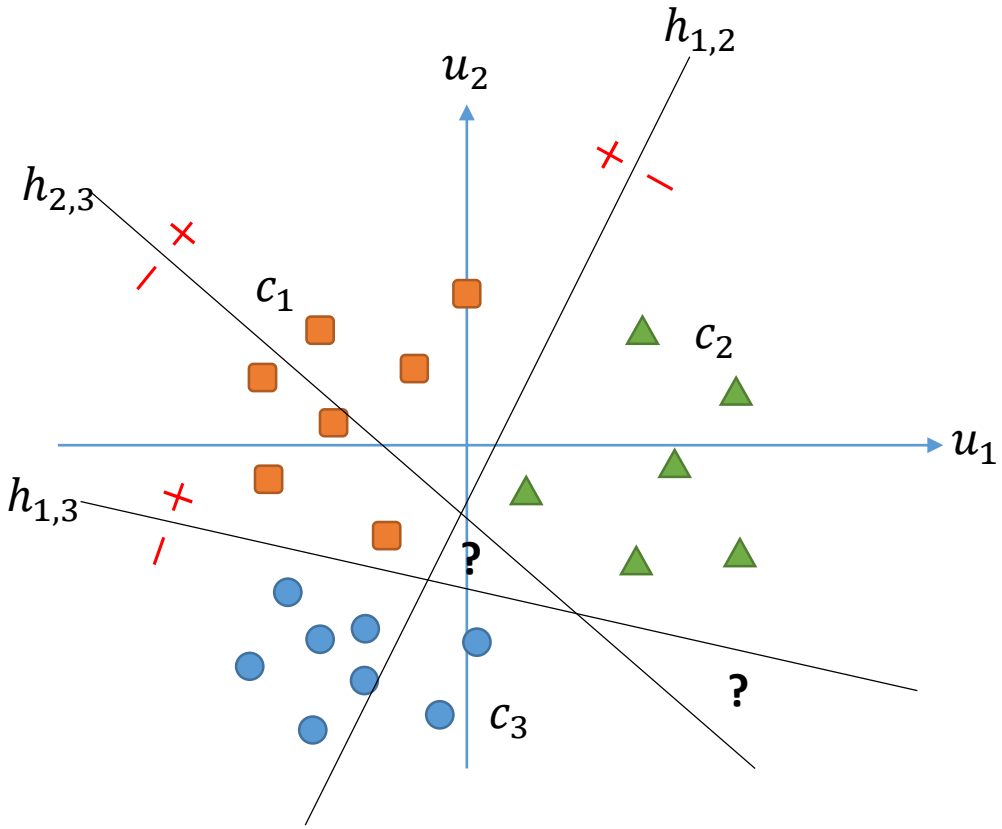
- For M classes, we could create $M(M-1)/2$ classifiers
  - Each classifier separates class c_i from c_j for all i and j

Called one vs. one classifier

Need a majority vote between all classifiers

Some regions become ambiguous

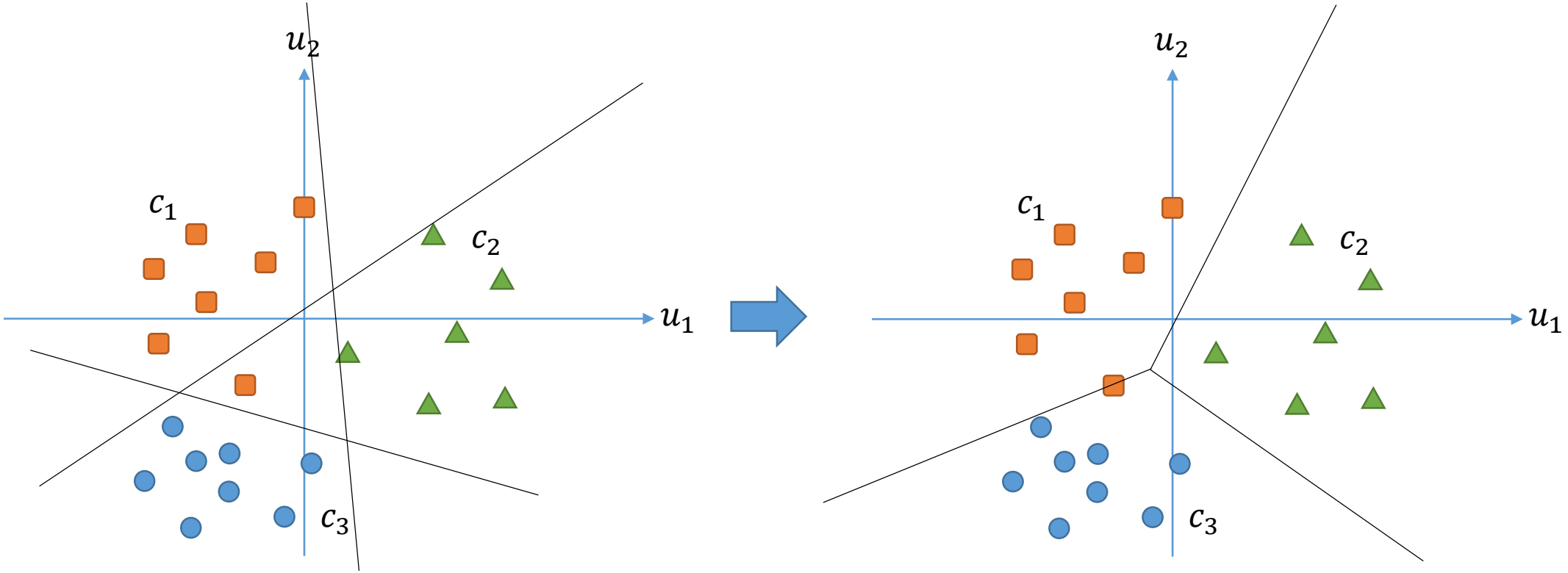
Large number of classifiers as M becomes large



$h_{1,3}$	$h_{2,3}$	$h_{1,2}$	h
-1	-1	-1	3
-1	-1	1	3
-1	1	-1	2
-1	1	1	?
1	-1	-1	?
1	-1	1	1
1	1	-1	2
1	1	1	1

Classification Overview

- For M classes, we could create a single classifier made of M classifiers
  - Separates class c_i from all other classes
  - Encode the output $\hat{\mathbf{y}} = [\hat{y}_1, \ldots, \hat{y}_M]$ and choose the largest component as the class

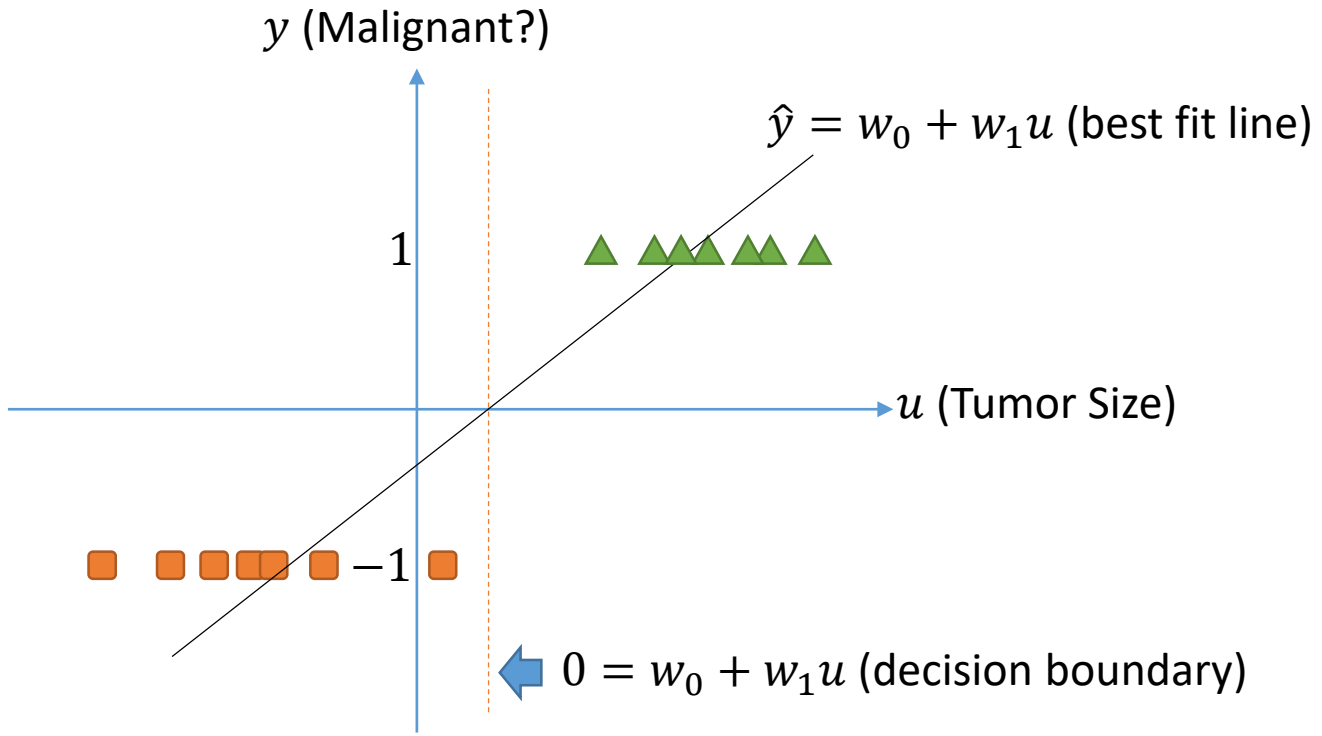


Classification with Least Squares

- Consider the following 1-feature, 2-class classification problem:

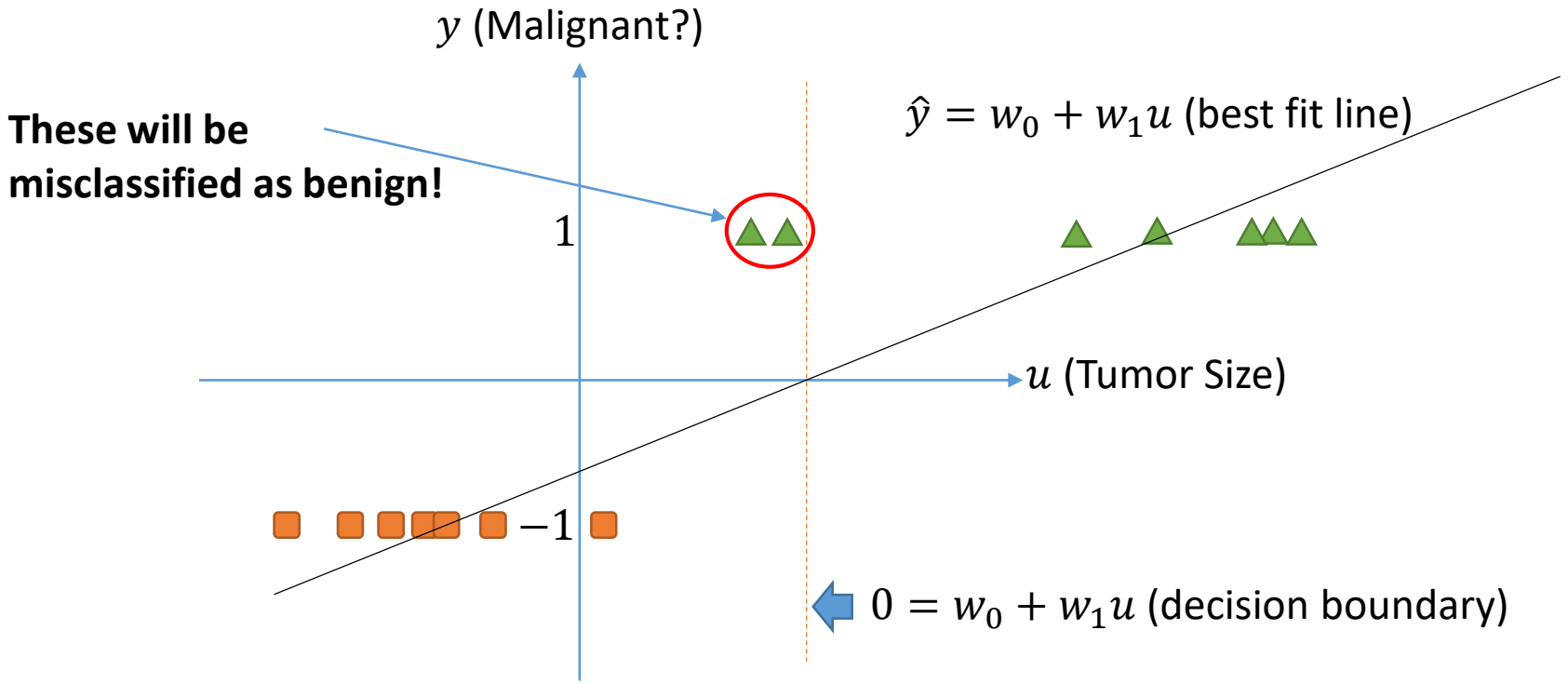
- Tumor is malignant if $w_0 + w_1 \times TumorSize > 0$
  - Could also use $y = 0$ for benign class and check if $\hat{y} > 0.5$

- This is a tempting approach
  - What will cause it to perform poorly?



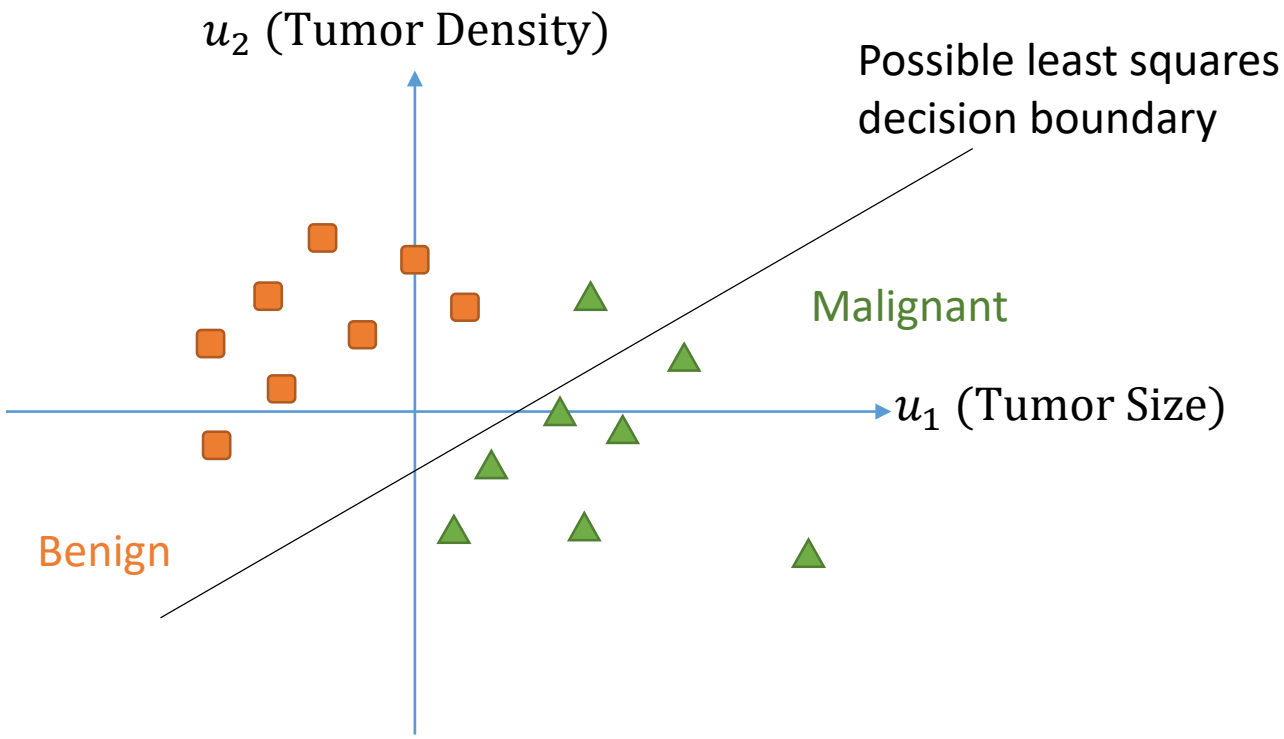
Classification with Least Squares

- What happens if our data are distributed differently?
  - This method is very sensitive to the distribution, not robust to outliers



Classification with Least Squares

- We may also use least squares for classification problems with more than one feature
  - Again, may not be robust depending on data distribution



Classification with Least Squares

- For a general M-class classification problem, we can use the pseudoinverse:

$$\mathbf{W} = (\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T\mathbf{Y}$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{u}$$

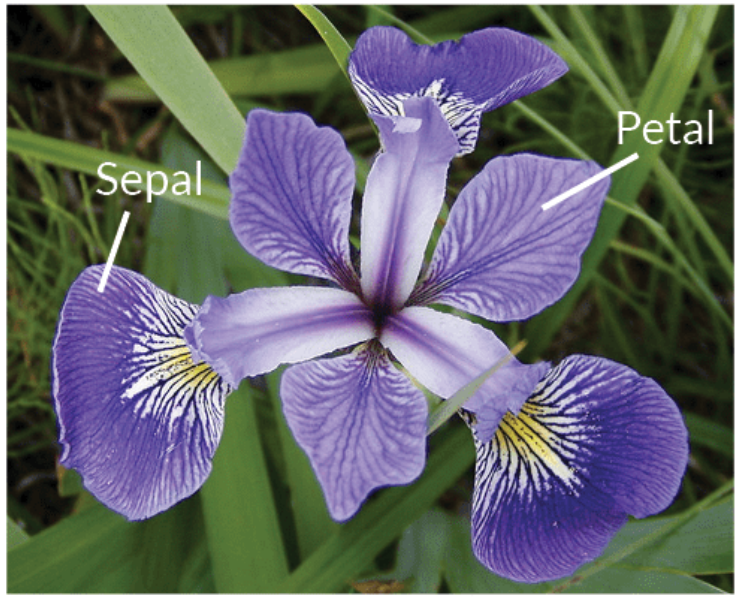
$$class = \arg\max_i \hat{y}_i$$

Where

- $\mathbf{W}$ is an $M \times N + 1$ matrix with each row representing the linear discriminant function for a particular class. (Note that the "+1" term is for the column of ones added to $\mathbf{U}$ for the bias
- $\mathbf{U}$ is an $n \times N + 1$ matrix of input observations with a column of ones added
- $\mathbf{Y}$ is an $n \times M$ matrix of output observations with a one-hot encoding
  - If row $\mathbf{u}^p$ belongs to class i, then $y_i^p = 1$ and $y_j^p = -1$ or 0 for all $j \neq i$

Example: Iris Dataset

- The iris dataset is a well-known machine learning dataset that has 3 classes of flowers ($M = 3$) and 4 features ($N = 4$)



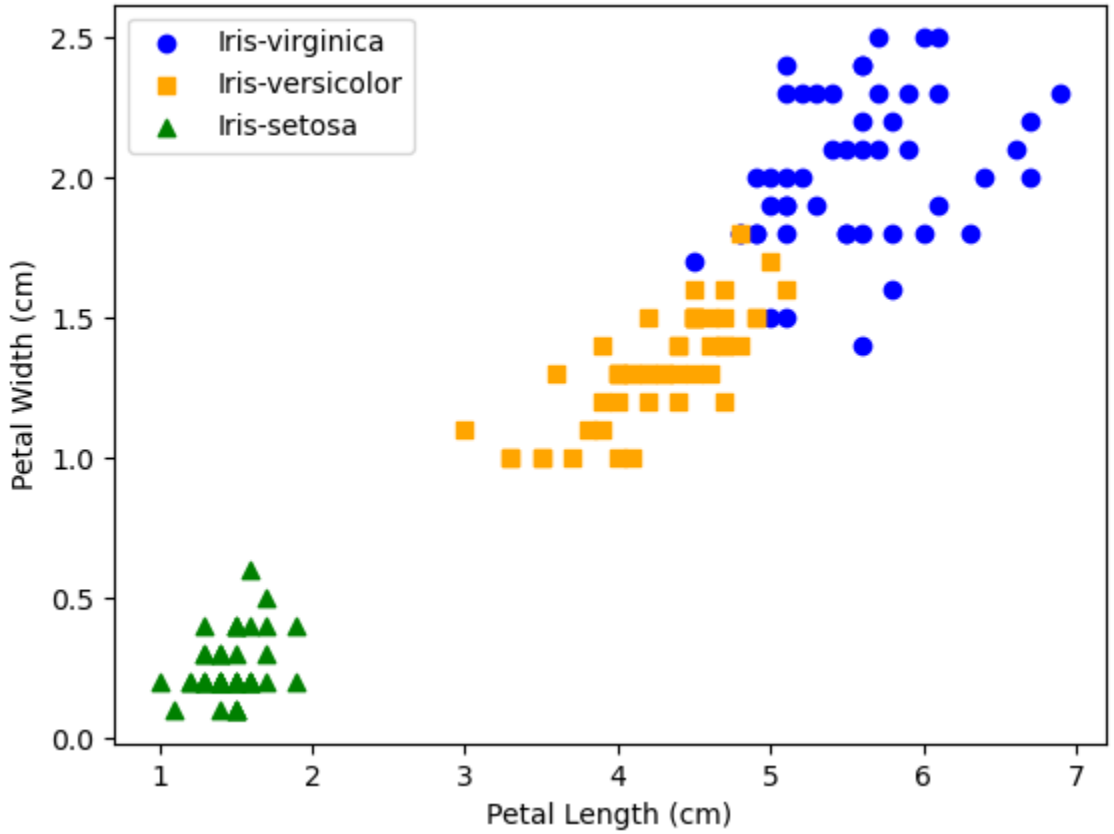
Iris Versicolor

Iris Setosa

Iris Virginica

Example: Iris Dataset

- To have a nice visualization, we will use a subset of the features:



$$\mathbf{U} = \begin{pmatrix} \text{Ones Column for Bias} & \text{Petal Length} & \text{Petal Width} \\ 1 & 1.5\text{ cm} & 0.25\text{ cm} \\ \dots & & \\ 1 & 4\text{ cm} & 1.5\text{ cm} \\ \dots & & \\ 1 & 6.2\text{ cm} & 2\text{ cm} \end{pmatrix}$$

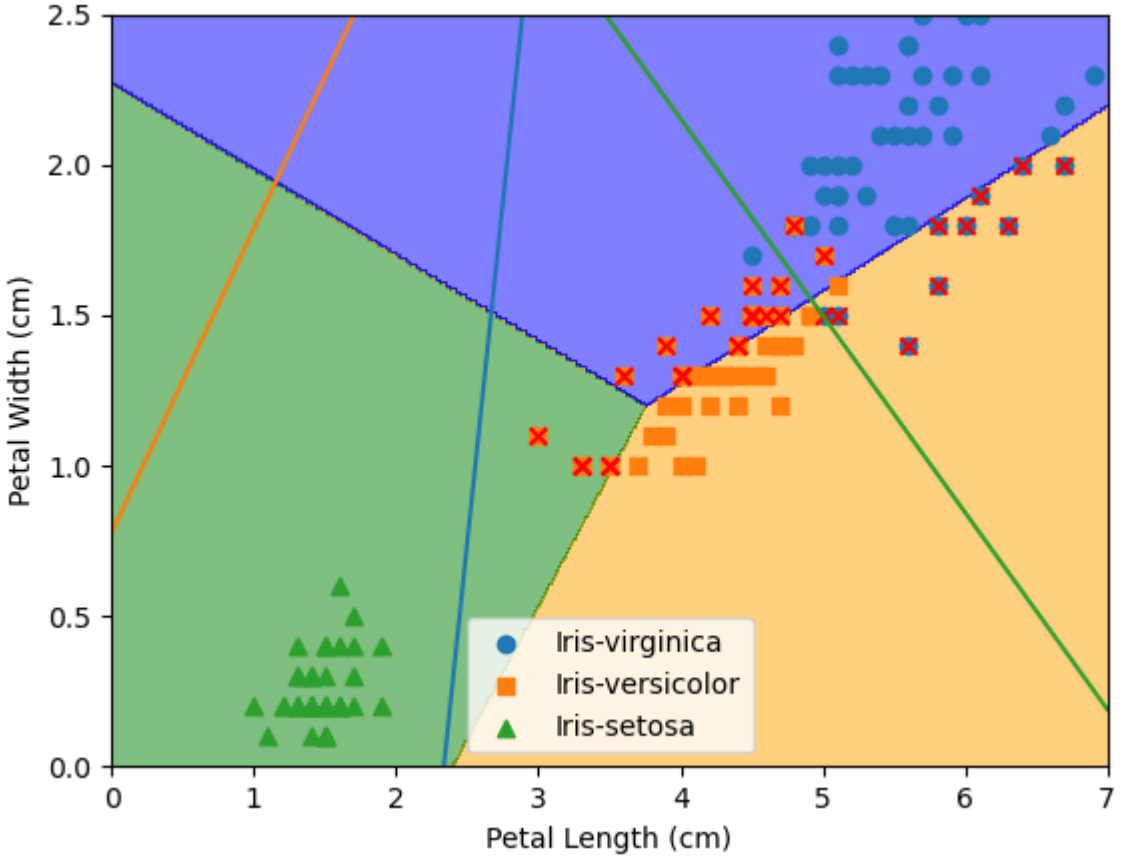
$$\mathbf{Y} = \begin{pmatrix} \text{Iris-setosa} & \text{Iris-versicolor} & \text{Iris-virginica} \\ 1 & 0 & 0 \\ \dots & & \\ 0 & 1 & 0 \\ \dots & & \\ 0 & 0 & 1 \end{pmatrix}$$

Example: Iris Dataset

- With linear regression, we get a training accuracy of ~77%
  - Accuracy = 100%×(# correctly classified points/ total # of points)

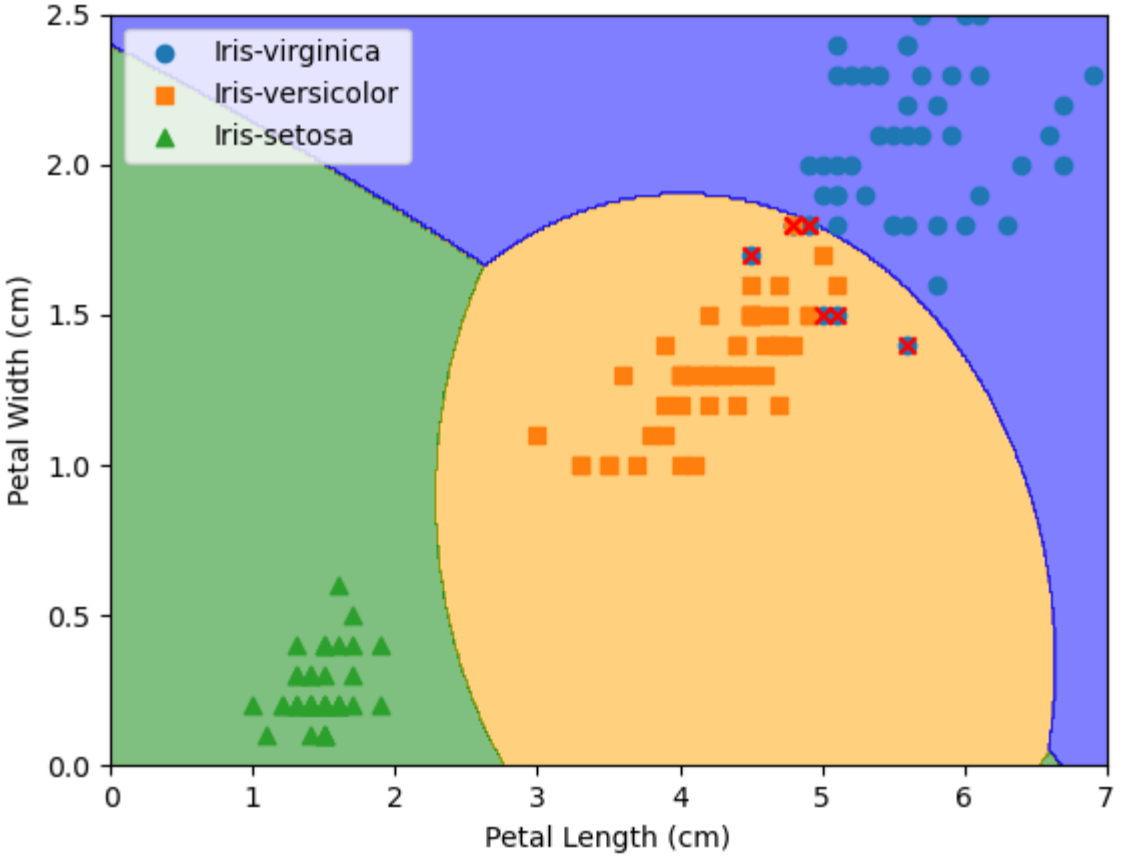
- The plot shows decision boundaries along with the hyperplanes represented by the 3 rows in $\mathbf{W}$
  - $0.5 = w_0 + u_1 w_1 + u_2 w_2$

- Since $class = \arg\max_i \hat{y}_i$ is a highly non-linear function, it's not always intuitively obvious how the individual class hyperplanes lead to the decision regions



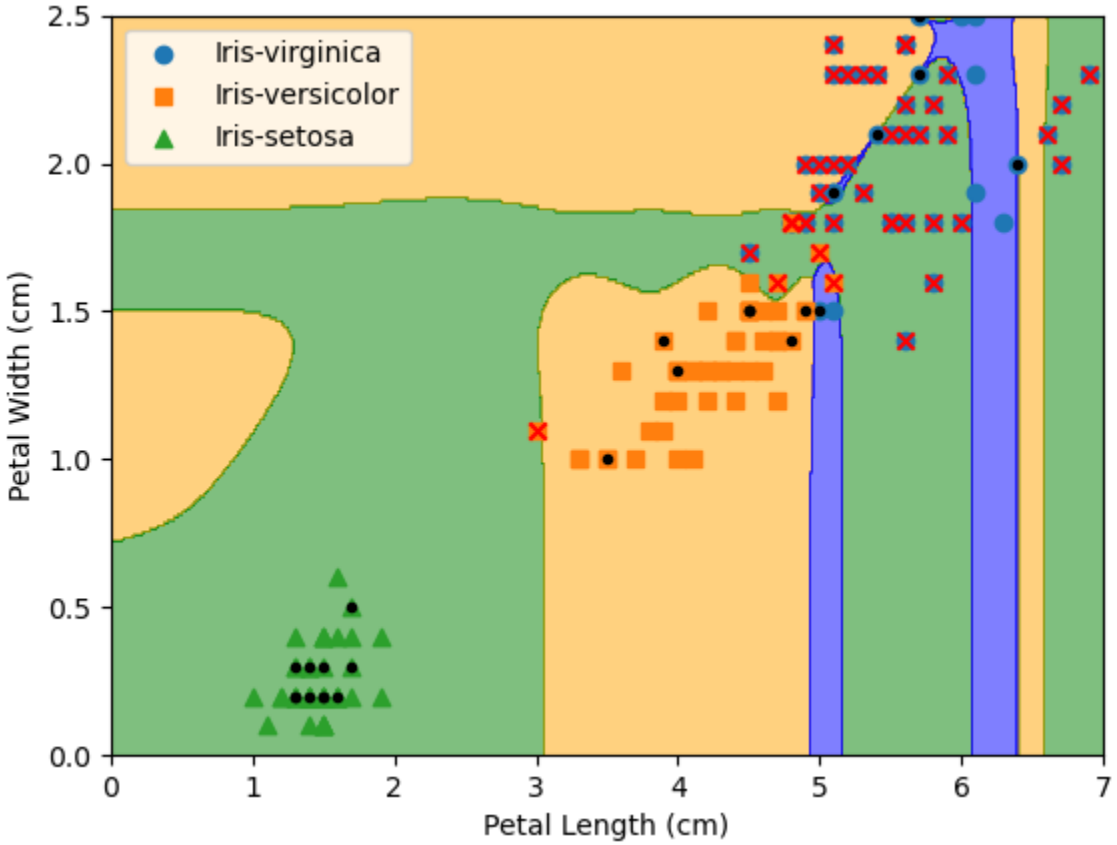
Example: Iris Dataset

- Just as in linear regression, we can use non-linear features in classification based on least squares
  - Here, we add a quadratic term, so we have $\mathbf{u} = [1, u_1, u_2, u_1^2, u_2^2]$
  - Training accuracy increases to ~94%



Example: Iris Dataset

- Just as we saw with regression, we can overfit and underfit in classification
  - Here, we show the case where we've added polynomial features up to order 15
  - We've used 50 of the 150 points in the dataset for training (marked by black dots)
  - 100% Training accuracy, 66% test accuracy
- We can reduce overfitting as before, with more training data and reduced model capacity or regularization

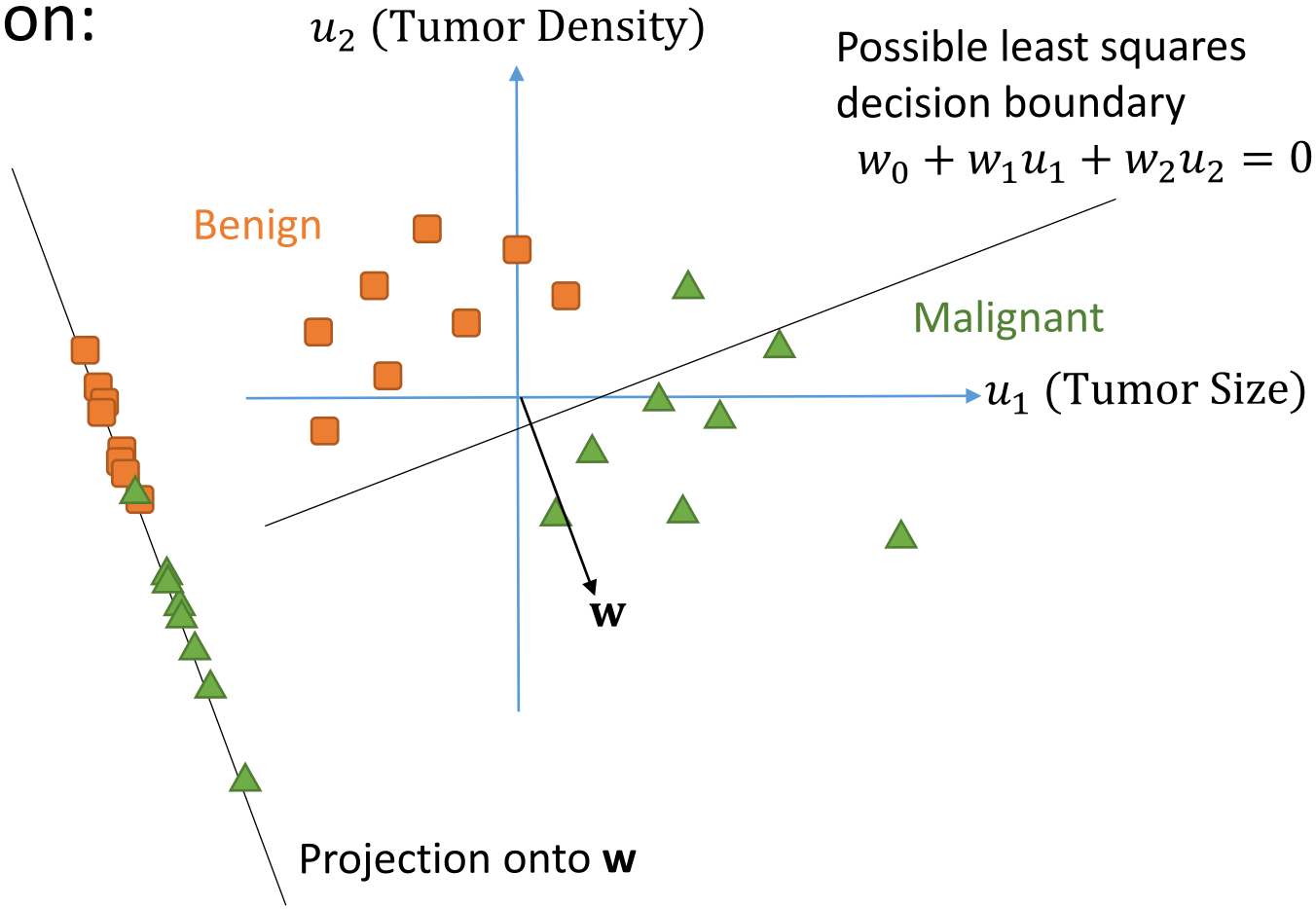


Fisher's Linear Discriminant

- Note that our least squares approach takes our inputs and projects them onto 1 dimension:

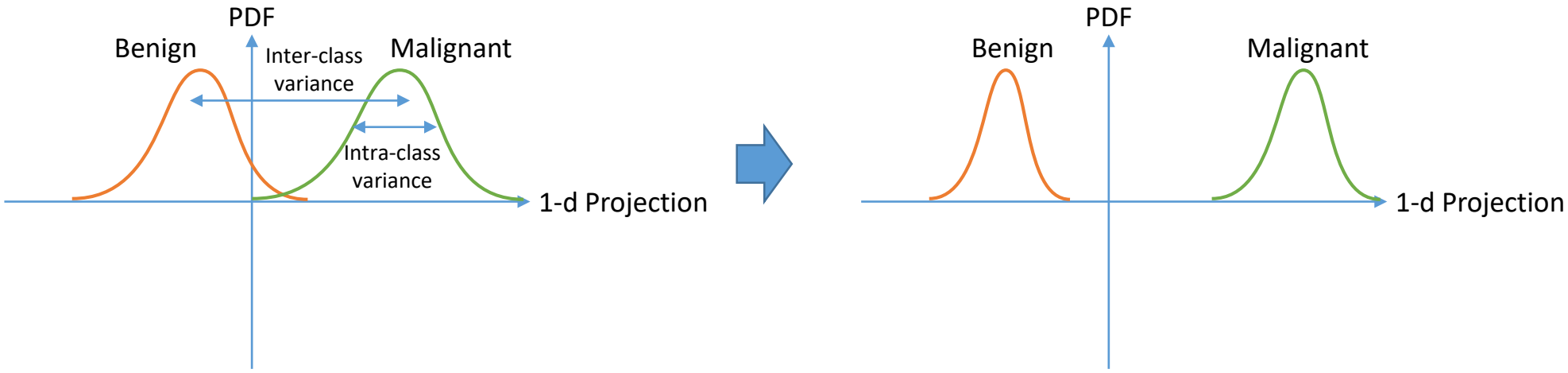
$$\hat{y} = \mathbf{uw} + w_0$$

- After the projection, there is no longer a clear separation between the data



Fisher's Linear Discriminant

- In general, we want our projection to do two things to the data
  - Maximize the inter-class variance
  - Minimize the intra-class variance
  - Now, it will be easy to find a threshold between the two classes



Fisher's Linear Discriminant: 2 Classes

- Let's describe the data of the two classes by their means:

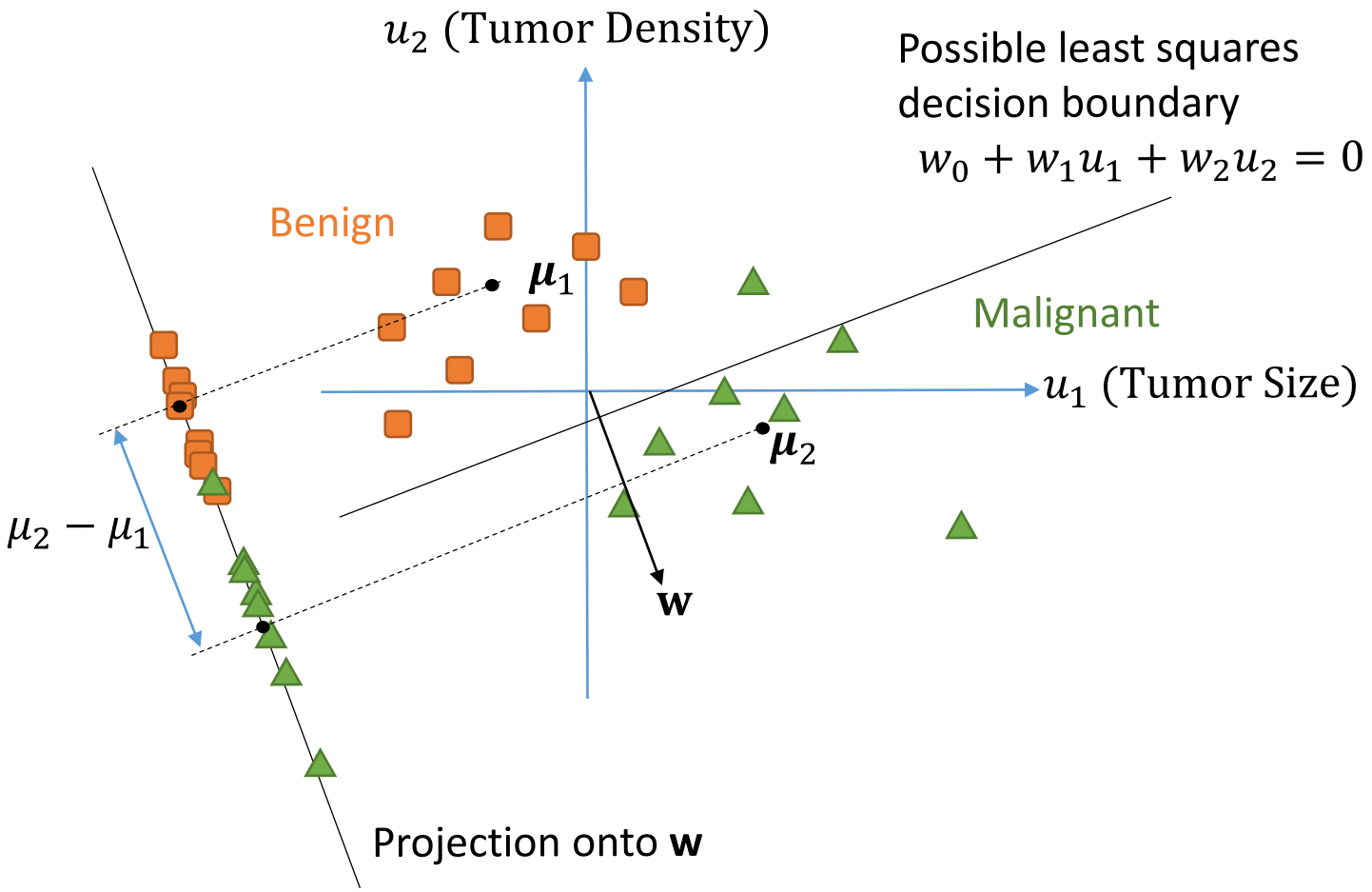
$$\boldsymbol{\mu}_1 = \frac{1}{n_1} \sum_{p \in c_1} \mathbf{u}^{(p)} \qquad \boldsymbol{\mu}_2 = \frac{1}{n_2} \sum_{p \in c_2} \mathbf{u}^{(p)}$$

- The inter-class variance is related to the projection of the means onto $\mathbf{w}$:

$$\mu_2 - \mu_1 = \mathbf{w}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T$$

Inter-class variance: $$(\mu_2 - \mu_1)^2 = \mathbf{w}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\mathbf{w}^T$$

Fisher's Linear Discriminant: 2 Classes



Fisher's Linear Discriminant: 2 Classes

- The intra-class variance is the variance of the projected points from their mean value:

$$s_1^2 = \sum_{p \in c_1} \left(\hat{y}^{(p)} - \mu_1\right)^2 \qquad s_2^2 = \sum_{p \in c_2} \left(\hat{y}^{(p)} - \mu_2\right)^2$$

- Since we want to maximize the inter-class variance and minimize the intra-class variance, we can formulate the following loss:

$$\mathcal{L} = \frac{(\mu_2 - \mu_1)^2}{s_1^2 + s_2^2} = \frac{\mathbf{w}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\mathbf{w}^T}{\sum_{p \in c_1}(\mathbf{w}\mathbf{u}^{(p)} - \mu_1)^2 + \sum_{p \in c_2}(\mathbf{w}\mathbf{u}^{(p)} - \mu_2)^2} =$$

$$= \frac{\mathbf{w}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\mathbf{w}^T}{\sum_{p \in c_1}(\mathbf{w}\mathbf{u}^{(p)} - \mu_1)(\mathbf{w}\mathbf{u}^{(p)} - \mu_1)^T + \sum_{p \in c_2}(\mathbf{w}\mathbf{u}^{(p)} - \mu_2)(\mathbf{w}\mathbf{u}^{(p)} - \mu_2)^T}$$

$$= \frac{\mathbf{w}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^{\boldsymbol{T}}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\mathbf{w}^T}{\mathbf{w}[\sum_{p \in c_1}(\mathbf{u}^{(p)} - \boldsymbol{\mu}_1)(\mathbf{u}^{(p)} - \boldsymbol{\mu}_1)^T + \sum_{p \in c_2}(\mathbf{u}^{(p)} - \boldsymbol{\mu}_2)(\mathbf{u}^{(p)} - \boldsymbol{\mu}_2)^T]\mathbf{w}^T} = \frac{\mathbf{w}\boldsymbol{S}_B\mathbf{w}^T}{\mathbf{w}\mathbf{S}_W\mathbf{w}^{\boldsymbol{T}}}$$

$\mathbf{S}_B$: Inter-class covariance matrix

$\mathbf{S}_W$: Intra-class covariance matrix

Fisher's Linear Discriminant: 2 Classes

- If we differentiate the loss w.r.t. $\mathbf{w}$ and equate to 0 we get:

$$(\mathbf{w}\mathbf{S}_B\mathbf{w}^T)\mathbf{S}_W\mathbf{w}^T = (\mathbf{w}\mathbf{S}_W\mathbf{w}^T)\mathbf{S}_B\mathbf{w}^T$$

or

$$a\mathbf{S}_W\mathbf{w}^T = b\mathbf{S}_B\mathbf{w}^T$$

Where a and b are scalars.

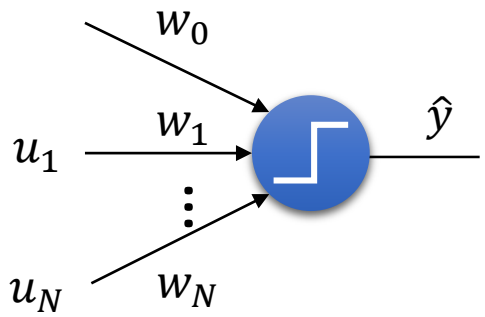
$$\boldsymbol{S}_B\mathbf{w}^T = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\mathbf{w}^T = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T \times scalar$$

$$\mathbf{w} = \frac{b}{a}\mathbf{S}_W^{-1}\mathbf{S}_B\mathbf{w}^T \propto \boldsymbol{S}_W^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T$$

- So, we can project data onto an $\mathbf{w}$ in this direction and the projected data will have optimal Fischer criteria
  - We can find the threshold theoretically or by looking at the histograms of the projection

Perceptron

- One of the earliest ANNs, proposed by Rosenblatt
  - Inspired by biological neuron

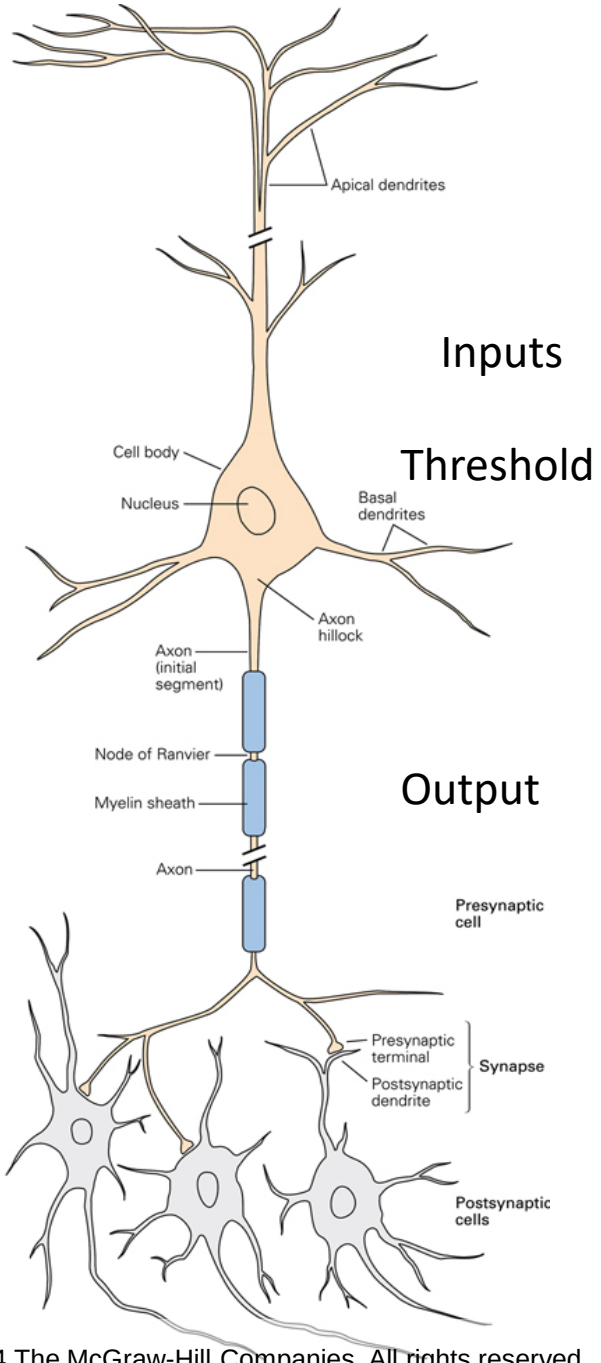


Output indicates if a neuron is firing (1) or not firing (-1 or 0)

$$\hat{y} = f(s) = \begin{cases} -1 & s < \phi \\ 1 & s \geq \phi \end{cases}$$

$$s \equiv w_0 + u_1 w_1 + \cdots + u_N w_N$$

Where ϕ is a threshold, often set to 0



Perceptron

- The perceptron model is learned using a rule similar to gradient descent
  - Note that we can't use gradient descent directly because $f(s)$ is not differentiable at $s = 0$
- Perceptron learning rule:

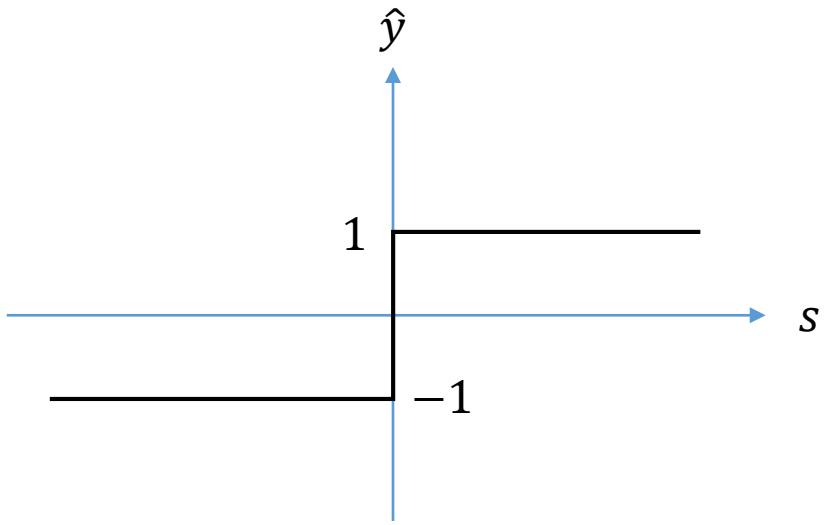
```
while not done:
        for each point in training set:
                if point is misclassified:
                        w_i := w_i + αu_i y
```

- It can be proven that this algorithm will converge to an exact solution (if one exists) in a finite number of iterations
- How can we extend this for multi-class classification?

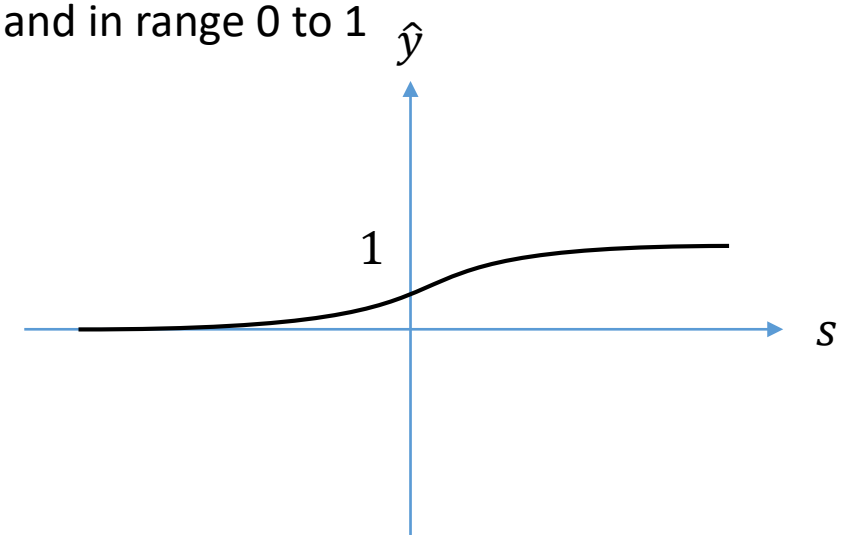
Logistic Regression

- The Perceptron is a nice model for classification because it is bounded between 2 values, which can represent 2 different classes in a binary classification problem

Perceptron Model



Perceptron-like model, but differentiable and in range 0 to 1



- It would nice to have something like the Perceptron, which is bounded, but smooth so we can differentiate it.
  - It would also be nice if the range was between 0 and 1 so we can interpret the output as a probability!

Logistic Regression

- What type of function should be choose as an offset and smoothed version of the Perceptron?
  - Let's think about classification probabilistically
- For a binary classification problem, what is the probability that an input belongs to class 1?

$$p(c_1|\mathbf{u}) = ?$$

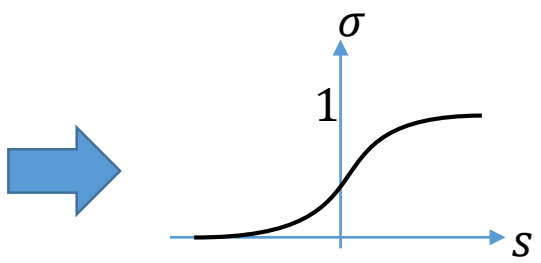
Recall Bayes' theorem: $p(a|b) = \frac{p(b|a)p(a)}{p(b)}$

So,

$$p(c_1|\mathbf{u}) = \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u})} = \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u}|c_1)p(c_1) + p(\mathbf{u}|c_2)p(c_2)} = \frac{1}{1 + \frac{p(\mathbf{u}|c_2)p(c_2)}{p(\mathbf{u}|c_1)p(c_1)}}$$

$$= \frac{1}{1 + \exp\left(-\ln \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u}|c_2)p(c_2)}\right)} = \frac{1}{1 + \exp(-s)} = \sigma(s)$$

Logistic sigmoid function



Logistic Regression

$$p(c_1|\mathbf{u}) = \frac{1}{1 + \frac{p(\mathbf{u}|c_2)p(c_2)}{p(\mathbf{u}|c_1)p(c_1)}} = \frac{1}{1 + \exp\left(-\ln \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u}|c_2)p(c_2)}\right)}$$

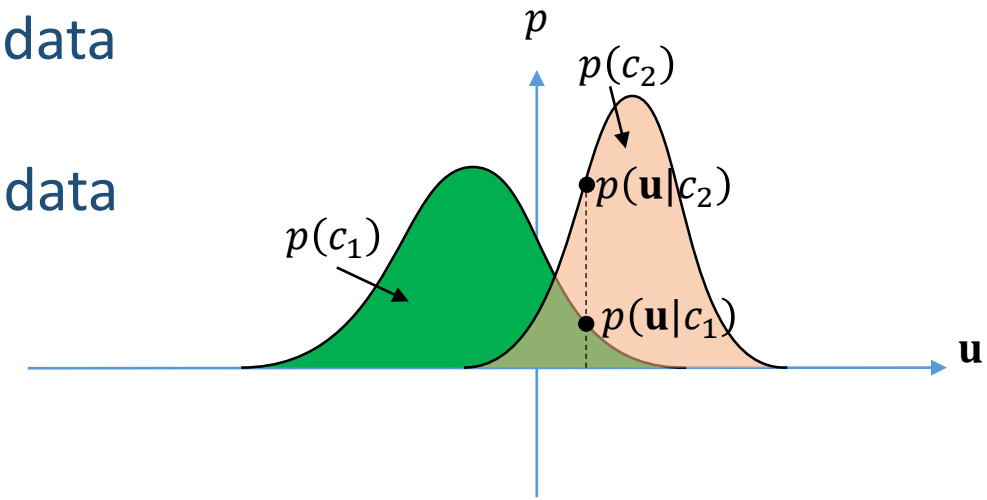
Note that the above two expressions are the same, so why did we go to the trouble of taking the exponent of the natural log?

- In the first case, we would be learning $\frac{p(\mathbf{u}|c_2)p(c_2)}{p(\mathbf{u}|c_1)p(c_1)}$, which we would have to force to always be positive, leading to optimization difficulties
- In the second case, we would be learning $\ln \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u}|c_2)p(c_2)}$, which can be positive or negative while keeping $p(c_1|\mathbf{u})$ between 0 and 1

Logistic Regression

- Now we have $\hat{y} = h_{\mathbf{w}}(\mathbf{u}) = p(c_1|\mathbf{u}) = \sigma(s)$
  with $s = w_0 + w_1 u_1 + w_2 u_2 + \cdots + w_N u_n$, as before.

- This means, we want learn $\mathbf{w}$ such that s represents $\ln \frac{p(\mathbf{u}|c_1)p(c_1)}{p(\mathbf{u}|c_2)p(c_2)}$
  - Large positive s means that we see more $\mathbf{u}$ values like this one in c_1 based on the training data
  - Large negative s means that we see more $\mathbf{u}$ values like this one in c_2 based on the training data



Logistic Regression

- $\mathbf{w}_{ML} = \arg\max_{\mathbf{w}} \sum \log p_h\left(\mathcal{D}^{(i)}|\mathbf{w}\right) = \arg\max_{\mathbf{w}} \sum_i \log p_h\left(y^{(i)}|\mathbf{u}, \mathbf{w}\right)$

- We can also minimize the negative value of this and divide by # of samples

$$\mathbf{w}_{ML} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \arg\min_{\mathbf{w}} -\frac{1}{n}\sum_i \log p_h\left(y^{(i)}|\mathbf{u}, \mathbf{w}\right)$$

- If $y^{(i)} \in \{0,1\}$, then we can write this as

$$\mathcal{L}(\mathbf{w}) = -\sum_i \log\left[\left(\hat{y}^{(i)}\right)^{y^{(i)}}\left(1-\hat{y}^{(i)}\right)^{1-y^{(i)}}\right] = -\frac{1}{n}\sum_i\left[y^{(i)}\log\hat{y}^{(i)} + \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right]$$

  - This is called the *cross-entropy loss*

Logistic Regression

- We can perform gradient descent using our logistic regression model as we have before:

Initialize **w** (e.g. to random values)
while **stop_condition** = False:
$$w_i \coloneqq w_i - \alpha \nabla_{w_i} \mathcal{L}(\mathcal{D}; w) \; \forall \; j$$

- We need the gradient of the cross-entropy loss function

$$\frac{\partial \mathcal{L}}{\partial w_i} = -\frac{1}{n} \sum_p \frac{\partial}{\partial w_i} \left[y^{(p)} \log \hat{y}^{(p)} + \left(1 - y^{(p)}\right) \log\left(1 - \hat{y}^{(p)}\right) \right]$$

Now that we're done with probabilities, let's switch back to using p for the index into the dataset…

Logistic Regression

$$\frac{\partial \mathcal{L}}{\partial w_i} = -\frac{1}{n}\sum_p \left[y^{(p)} \frac{\partial}{\partial \hat{y}^{(p)}} \log \hat{y}^{(p)} \frac{\partial \hat{y}^{(p)}}{\partial w_i} + \frac{\partial}{\partial \hat{y}^{(p)}} \left(1 - y^{(p)}\right) \log\left(1 - \hat{y}^{(p)}\right) \frac{\partial \hat{y}^{(p)}}{\partial w_i} \right]$$

- Recall that $\hat{y}^{(p)}$ is our logistic sigmoid $\sigma(s)$
  - The derivative of the logistic sigmoid has the nice property $\frac{\mathrm{d}\sigma(s)}{\mathrm{d}s} = \sigma(s)\left(1 - \sigma(s)\right)$

$$\frac{\partial \mathcal{L}}{\partial w_i} = -\frac{1}{n}\sum_p \left[y^{(p)} \frac{1}{\sigma(s^{(p)})} \frac{\partial \sigma\left(s^{(p)}\right)}{\partial s^{(p)}} \frac{\partial s^{(p)}}{\partial w_i} + \left(1 - y^{(p)}\right) \frac{1}{1 - \sigma(s^{(p)})} \left(-\frac{\partial \sigma\left(s^{(p)}\right)}{\partial s^{(p)}} \frac{\partial s^{(p)}}{\partial w_i} \right) \right]$$

$$= -\sum_p \left[y^{(p)} \frac{1}{\sigma(s^{(p)})} \sigma\left(s^{(p)}\right) \left(1 - \sigma\left(s^{(p)}\right)\right) \frac{\partial s^{(p)}}{\partial w_i} + \left(1 - y^{(p)}\right) \frac{1}{1 - \sigma(s^{(p)})} \left(-\sigma\left(s^{(p)}\right) \left(1 - \sigma\left(s^{(p)}\right)\right) \frac{\partial s^{(p)}}{\partial w_i} \right) \right]$$

- $s \equiv w_0 + w_1 u_1 + \cdots w_N u_N$, so this simplifies to

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{1}{n}\sum_p \left(\hat{y}^{(p)} - y^{(p)}\right) u_i^{(p)}$$

Same loss gradient as we had for least squares!!!

Logistic Regression

- The logistic regression training algorithm:

Initialize $\mathbf{w}$ (e.g. to random values)
while **stop_condition** = False:
    $\Delta\mathbf{w}$ = zeros
    for p from 1 to n:
        $s^{(p)} = \mathbf{u}^{(p)}\mathbf{w}^T$
        $\hat{y}^{(p)} = \sigma\left(s^{(p)}\right)$
        $\Delta\mathbf{w} \coloneqq \Delta\mathbf{w} - \alpha\frac{1}{n}\left(\hat{y}^{(p)} - y^{(p)}\right)\mathbf{u}^{(p)}$
    $\mathbf{w} \coloneqq \mathbf{w} + \boldsymbol{\Delta}\mathbf{w}$

Note, we can remove the loop and write this more compactly as $\mathbf{w} \coloneqq \mathbf{w} - \alpha(\hat{\mathbf{y}} - \mathbf{y})\mathbf{U}$

Logistic Regression for $M > 2$

- For more than two classes, we can use M sigmoid models and combine their results into a single decision by choosing the maximum value (just as we did with multiclass least squares for classification)
  - Sometimes we call this "winner-takes-all"
- The drawback of this is that we lose the probabilistic interpretation of the output:
  - Imagine we have a 3-class classifier, with 3 different sigmoid functions and we get the following output: $\hat{\mathbf{y}} = [0.1, 0.9, 0.73]$
  - Since 0.9 is the largest, we would assign class 2 as the output
  - Notice, however, that the sum of the outputs is > 1!
    - 0.9 no longer means that the classifier has a 90% confidence that this is class 2

Logistic Regression for $M > 2$

- A better approach is to go back to the beginning:

$$p(c_k|\mathbf{u}) = ?$$

- If we have M classes, this becomes

$$p(c_k|\mathbf{u}) = \frac{p(\mathbf{u}|c_k)p(c_k)}{p(\mathbf{u})} = \frac{p(\mathbf{u}|c_k)p(c_k)}{\sum_i p(\mathbf{u}|c_i)p(c_i)}$$

$$= \frac{\exp(\ln p(\mathbf{u}|c_k)p(c_k))}{\sum_i \exp(\ln p(\mathbf{u}|c_i)p(c_i))} = \boxed{\frac{\exp(s_k)}{\sum_i \exp(s_i)}}$$

- This is the *softmax function*
  - Smooth approximation to the max function
    - The largest value will get amplified and the smaller ones will be attenuated

Logistic Regression for $M > 2$

- Now, we can again write down the maximum likelihood function and derive a new loss function.
  - We will use a binary encoding for our output. e.g., for $M = 3$, a target output corresponding to class c_2 is $\mathbf{y} = [0,1,0]$. A target output corresponding to class k is $\mathbf{y} = \mathbf{e}_k$, where $\mathbf{e}_k$ has a '1' in position k and '0' everywhere else.

$$p(\mathbf{Y}|\mathbf{U},\mathbf{W}) = \prod_{i=1}^{n}\prod_{k=1}^{M} p\left(c_k|\mathbf{u}^{(i)},\mathbf{W}\right)^{y_k^{(i)}} = \prod_{p=1}^{n}\prod_{k=1}^{M} \hat{y}_k^{(p)\,y_k^{(p)}}$$

- Get the loss by taking the negative log and dividing by the number of datapoints:

$$\mathcal{L} = -\frac{1}{n}\sum_{p}\sum_{k} y_k^{(p)} \log \hat{y}_k^{(p)}$$

- Amazingly, the gradient of the new loss function with respect to the parameters w_{ij} (defined as the parameter connecting input j to model output i) is the same as we had before:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{1}{n}\sum_{p}\left(\hat{y}_i^{(p)} - y_i^{(p)}\right) u_j^{(p)}$$

- We can use the same gradient descent algorithm for multi-class classification with logistic regression

Regularized Logistic Regression

- Just like linear regression, we can add a regularization term to our loss function:

$$\mathcal{L} = -\frac{1}{n}\sum_{p}\sum_{k} y_k^{(p)} \log \hat{y}_k^{(p)} + \lambda \mathcal{L}_{\mathbf{W}}$$

- For example, when $\mathcal{L}_{\mathbf{W}}$ is the l_2 regularization term, we have

$$\mathcal{L} = -\frac{1}{n}\sum_{p}\sum_{k} y_k^{(p)} \log \hat{y}_k^{(p)} + \frac{\lambda}{2n}\sum_{i}\sum_{k} w_{k,i}^2$$

Logistic Regression

- Our general gradient descent scheme for logistic regression with M classes becomes:

Initialize $\mathbf{W} \in \mathbb{R}^{M \times N}$ (e.g. to random values)
while **stop_condition** = False:
    $\Delta\mathbf{W}$ = zeros
    for p from 1 to n:
        $\mathbf{s}^{(p)} = \mathbf{W}\mathbf{u}^{(p)^T}$
        $\hat{\mathbf{y}}^{(p)} = \boldsymbol{\sigma}\left(\mathbf{s}^{(p)}\right)$ # $\boldsymbol{\sigma}$ is the softmax function applied to each element of $\mathbf{s}^{(p)}$
        for k from 1 to M:
            $\Delta\mathbf{W}_k := \Delta\mathbf{W}_k - \alpha\left[\frac{1}{n}\left(\hat{y}_k^{(p)} - y_k^{(p)}\right)\mathbf{u}^{(p)} + \lambda\frac{\partial\mathcal{L}}{\mathbf{W}_k}\right]$ # $\mathbf{W}_k$ is row k of W
    $\mathbf{W}_k := \mathbf{W}_k + \Delta\mathbf{W}_k$

e.g., $\frac{\lambda}{n}\mathbf{W}_k$ for ridge regression

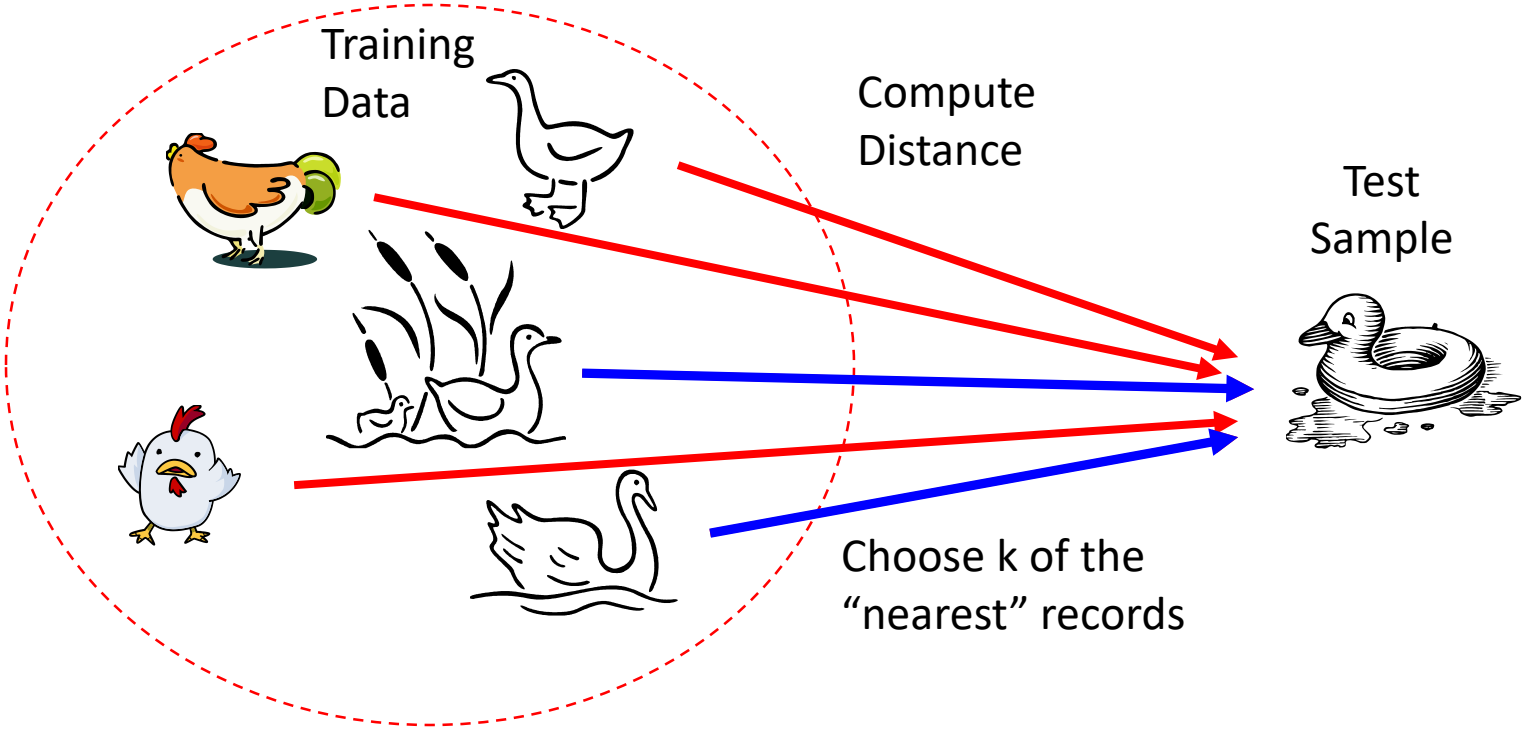
Logistic Regression with Smaller Batches

- Just like linear regression, we can perform logistic regression in mini batches
  - n' is the batch size (number of examples used to estimate the gradient)
  - Recall that a batch size of 1 is called stochastic gradient descent

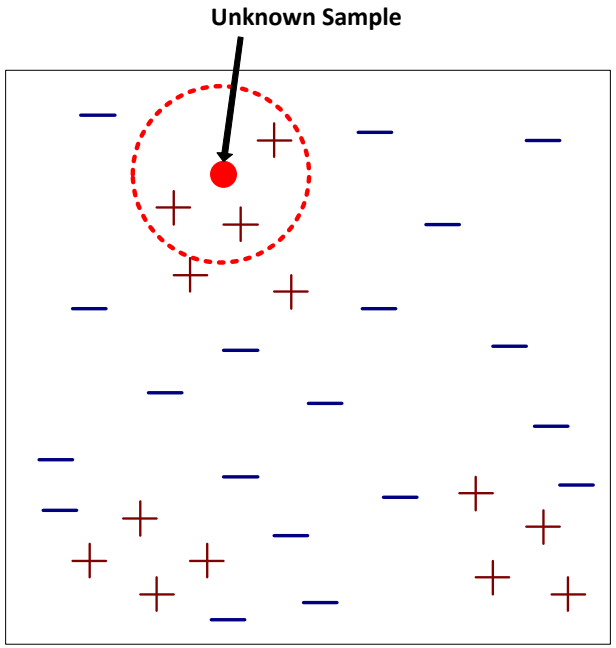
Initialize $\mathbf{W} \in \mathbb{R}^{M \times N}$ (e.g. to random values)
while **stop_condition** = False:
    for batch from 1 to num_batches:
        $\Delta\mathbf{W}$ = zeros
        for p in batch:
            $\mathbf{s}^{(p)} = \mathbf{W}\mathbf{u}^{(p)^T}$
            $\hat{\mathbf{y}}^{(p)} = \boldsymbol{\sigma}(\mathbf{s}^{(p)})$ # $\boldsymbol{\sigma}$ is the softmax function applied to each element of $\mathbf{s}^p$
            for k from 1 to M:
                $\Delta\mathbf{W}_k := \Delta\mathbf{W}_k - \alpha\left[\frac{1}{n'}\left(\hat{y}_k^{(p)} - y_k^{(p)}\right)\mathbf{u}^{(p)} + \lambda\frac{\partial\mathcal{L}}{\mathbf{W}_k}\right]$ # $\mathbf{W}_k$ is row k of W
        $\mathbf{W}_k := \mathbf{W}_k + \Delta\mathbf{W}_k$

Nearest Neighbor Classifier

- Basic idea: If it looks like a duck and quacks like a duck, then it's probably a duck



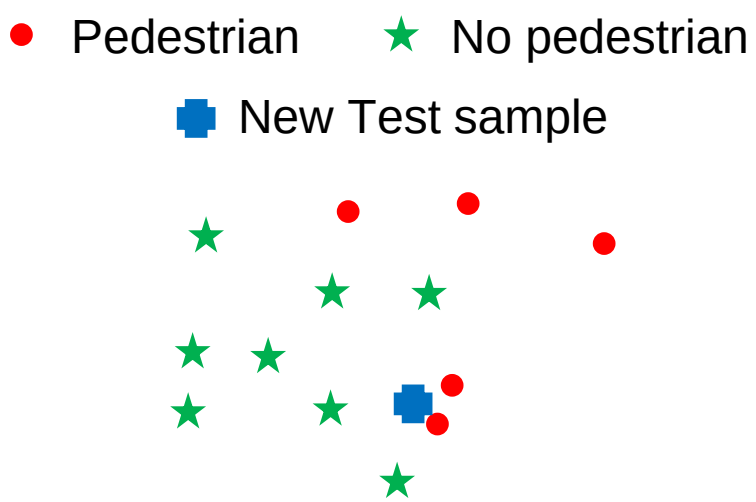
Nearest-Neighbor Classification



- Setup Requires:
  - The set of training samples
  - Distance Metric to compute distance between samples
  - The value of *k*, the number of nearest neighbors to retrieve
- To classify an unknown sample:
  - Compute its distance to all training samples
  - Identify *k* nearest neighbors (k=1,3,5…)
  - Use class labels of nearest neighbors to determine the class label of unknown sample (e.g., by majority vote)

K-Nearest Neighbor- Classification

- Given the feature plot, the nearest neighbor would indicate pedestrian detection.

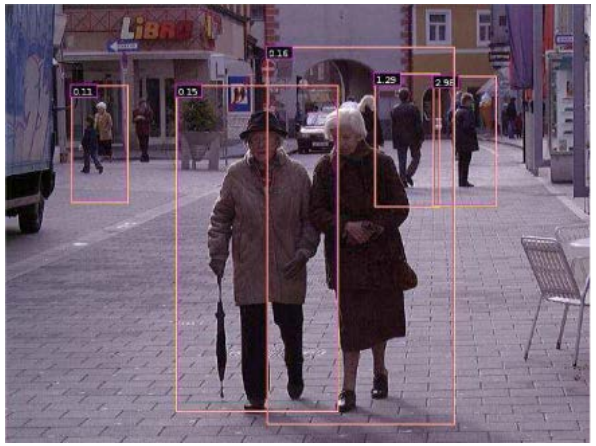


Increase *k* to get more robust.

At *k*=4, we could have a tie.

Use odd number of neighbors.

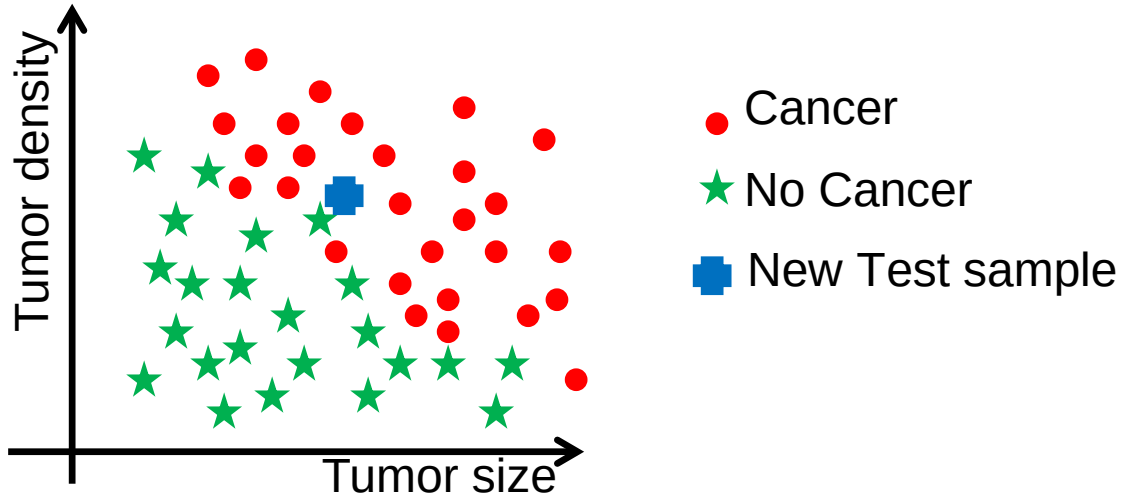
k=1,3,5



Classification Example

Two features:
- Tumor size
- Tumor density

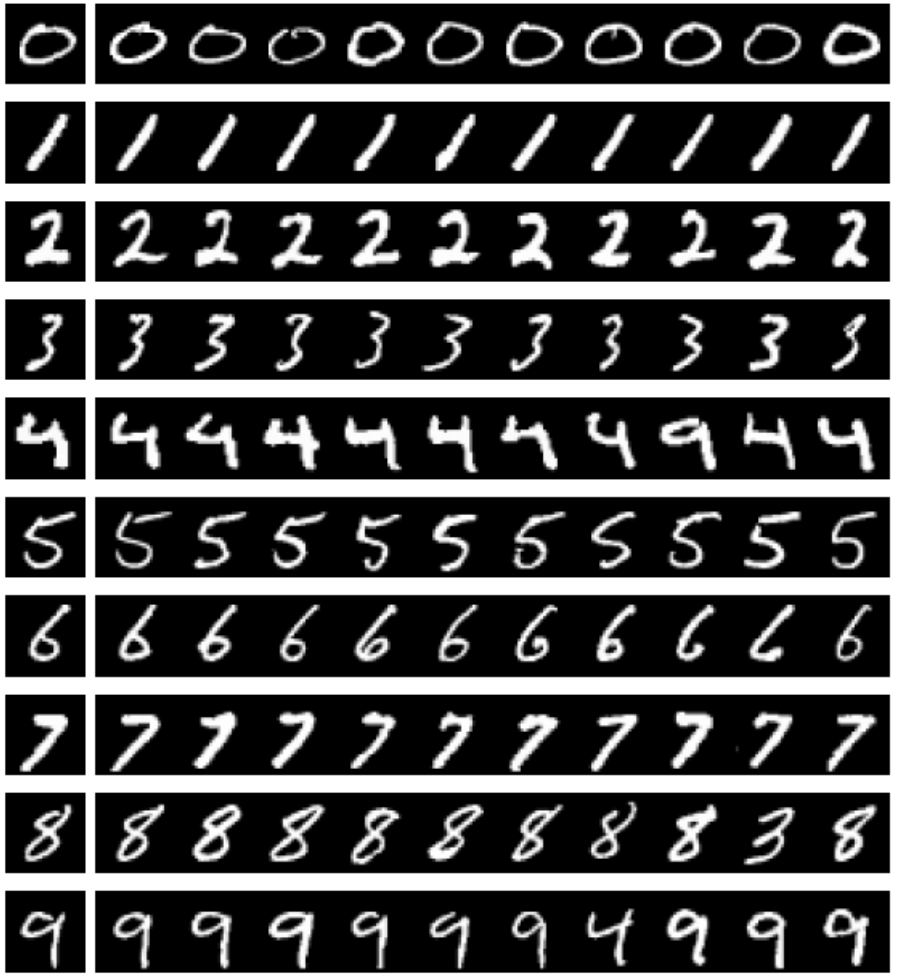
Two classes:
- Cancer
- No Cancer



- Labeled training data (● , ★)
- A new test sample (■)
- Is the new test sample cancerous?

K-Nearest Neighbor- Classification

- Using MNIST dataset of 60K handwritten character, each 28 × 28 pixels, a 784 dim vector.
- Here we have ten random test samples of each digit, coupled with its ten nearest training samples.
- A 10-NN classifier produces an accuracy of ~95%.



Statistical Machine Learning- Template Methods, J. Domke

K-Nearest Neighbor- Summary

- Based on the same basic idea, can be adapted to do regression or classification.
- Simplest of all algorithms to understand, but requires the sorting of all data which can be very expensive.
- In general, for very high number of samples, higher *k* is preferred.
- The higher the dimensional space, the more samples you will need.
  - The curse of dimensionality states that the number of samples you need are exponential with *d* (binary is 2^d)!
- Separate test / train datasets need to be used to solve for the optimal *k*.
- Can weigh *k* nearest neighbors by distance for improved results.