- To find optimal model parameters for the model $H(x) = w_0 + w_1 x$ and squared loss, we minimized empirical risk:

$$R_{\text{sq}}(w_0, w_1) = R_{\text{sq}}(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i))^2$$

- This is a function of multiple variables, and is differentiable, so it has a gradient!

$$\nabla R(\vec{w}) = \begin{bmatrix} -\frac{2}{n} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i)) \\ -\frac{2}{n} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i)) x_i \end{bmatrix} \begin{array}{l} \leftarrow \frac{\partial R}{\partial w_0} \\ \leftarrow \frac{\partial R}{\partial w_1} \end{array}$$

- **Key idea**: To find $\vec{w}^* = \begin{bmatrix} w_0^* \\ w_1^* \end{bmatrix}$, we *could* use gradient descent!

- Why would we, when closed-form solutions exist? ← could be quicker!

J
Gradient Descent Visualization
Watch later
Share
$R(w_0, w_1)$
Gradient
Negative Gradient
0:38 / 1:57
YouTube

Parameters: $w_0 = -0.1, w_1 = -5.1$

Risk: $R(w_0, w_1) = 18.9$

"learning rate"

multiplier on the gradient

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \boxed{\alpha} \nabla R(\vec{w}^{(t)})$$

Step Size = 0.1

localhost

Implementing partial derivatives

$$R_{\text{sq}}(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2$$

mean

$$\nabla R(\vec{w}) = \begin{bmatrix} -\frac{2}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i)) \\ -\frac{2}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))x_i \end{bmatrix}$$

```
In [7]: def dR_w0(w0, w1):
            return -2 * np.mean(y - (w0 + w1 * x))
        def dR_w1(w0, w1):
            return -2 * np.mean((y - (w0 + w1 * x)) * x)
```

9.1

localhost
?
10 . 1

Implementing gradient descent

- The update rule we'll follow is:

$$\vec{w}^{(t+1)} = \overset{\begin{bmatrix} w_0^{(t)} \\ w_1^{(t)} \end{bmatrix}}{\vec{w}^{(t)}} - \alpha \underbrace{\nabla R(\vec{w}^{(t)})}_{\begin{bmatrix} \frac{\partial R}{\partial w_0} \\ \frac{\partial R}{\partial w_1} \end{bmatrix}}$$

- We can treat this as two separate update equations:

$$w_0^{(t+1)} = w_0^{(t)} - \alpha \frac{\partial R}{\partial w_0}(\vec{w}^{(t)})$$

$$w_1^{(t+1)} = w_1^{(t)} - \alpha \frac{\partial R}{\partial w_1}(\vec{w}^{(t)})$$

```
        w1 = w1 - alpha * dR_w1(w0, w1)
        w0_history.append(w0)
        w1_history.append(w1)
        if np.abs(w0_history[-1] - w0_history[-2]) <= threshold:
            break
    return w0_history, w1_history
```

```
In [10]: w0_history, w1_history = gradient_descent_for_regression(0, 0, 0.01)
```

```
In [11]: w0_history[-1]

Out[11]: 142.1051891023626
```

```
In [12]: w1_history[-1]

Out[12]: -8.146983792459055
```

- increase the learning rate

- try a different $\vec{w}^{(0)}$

- compute $\nabla R(\vec{w}^{(t)})$ using just a sample of the data

$\Rightarrow$ stochastic gd

- It seems that we converge at the right value! But how many iterations did it take? What could we do to speed it up?

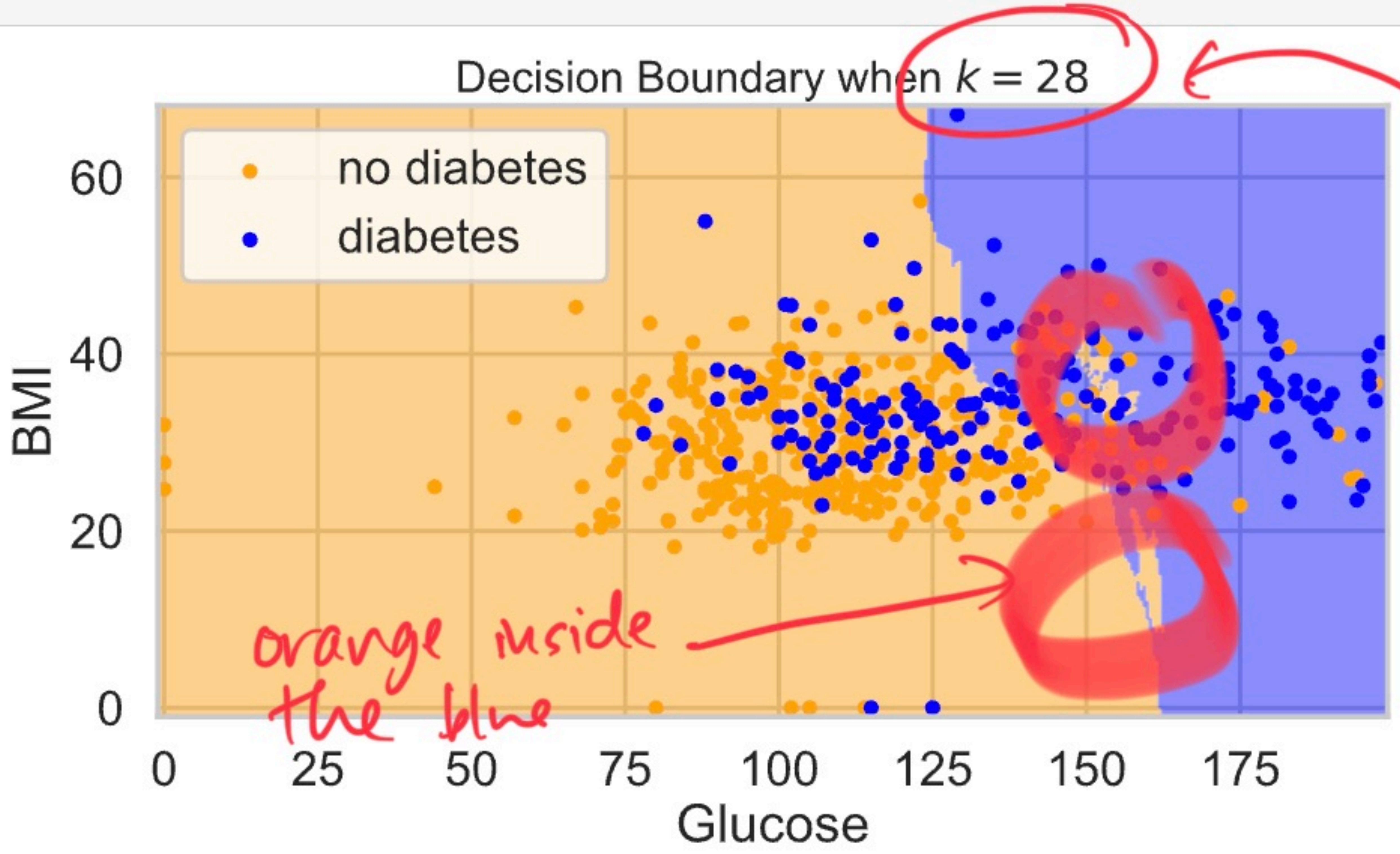
```
In [13]: len(w0_history)

Out[13]: 20664
```

- The **decision boundaries** of a classifier visualize the regions in the feature space that separate different predicted classes.

- The decision boundaries for `model_knn` are visualized below.

If a new person's feature vector lies in the **blue region**, we'd predict they **do have diabetes**, **otherwise**, we'd predict **they don't**.

```
In [28]: util.show_decision_boundary(model_knn, X_train, y_train, title='Decision Boundary when $k = 28$')
```



chosen through cross-validation to generalize well and prevent overfitting

still a little overfit?

- What would the decision boundaries look like if k increased or decreased?

Play with the slider below to find out!

```
In [29]: from ipywidgets import interact
         interact(lambda k: util.visualize_k(k, X_train, y_train), k=(1, 51));
```

k 1

Decision Boundary when $k = 1$

no diabetes
diabetes
BMI
60
40
20
0
0
25
50
75
100
125
150
175
Glucose
lots of overfitting!

Activity

It seems that a k-NN classifier that uses $k = 1$ should achieve 100% training accuracy. Why **doesn't** the model defined below have 100% training accuracy?

```
In [38]: model_k1 = KNeighborsClassifier(n_neighbors=1)
         model_k1.fit(X_train, y_train)

Out[38]: ▾ KNeighborsClassifier
           KNeighborsClassifier(n_neighbors=1)

In [39]: # Training accuracy – high, but not 100%.
         model_k1.score(X_train, y_train)

Out[39]: 0.9913194444444444

In [40]: # Accuracy on test set is lower than when k = 28!
         model_k1.score(X_test, y_test)

Out[40]: 0.6822916666666666

In [41]: test_scores['knn with k = 1'] = model_k1.score(X_test, y_test)
         test_scores

Out[41]: knn with k = 28     0.75
         knn with k = 1      0.68
         dtype: float64
```

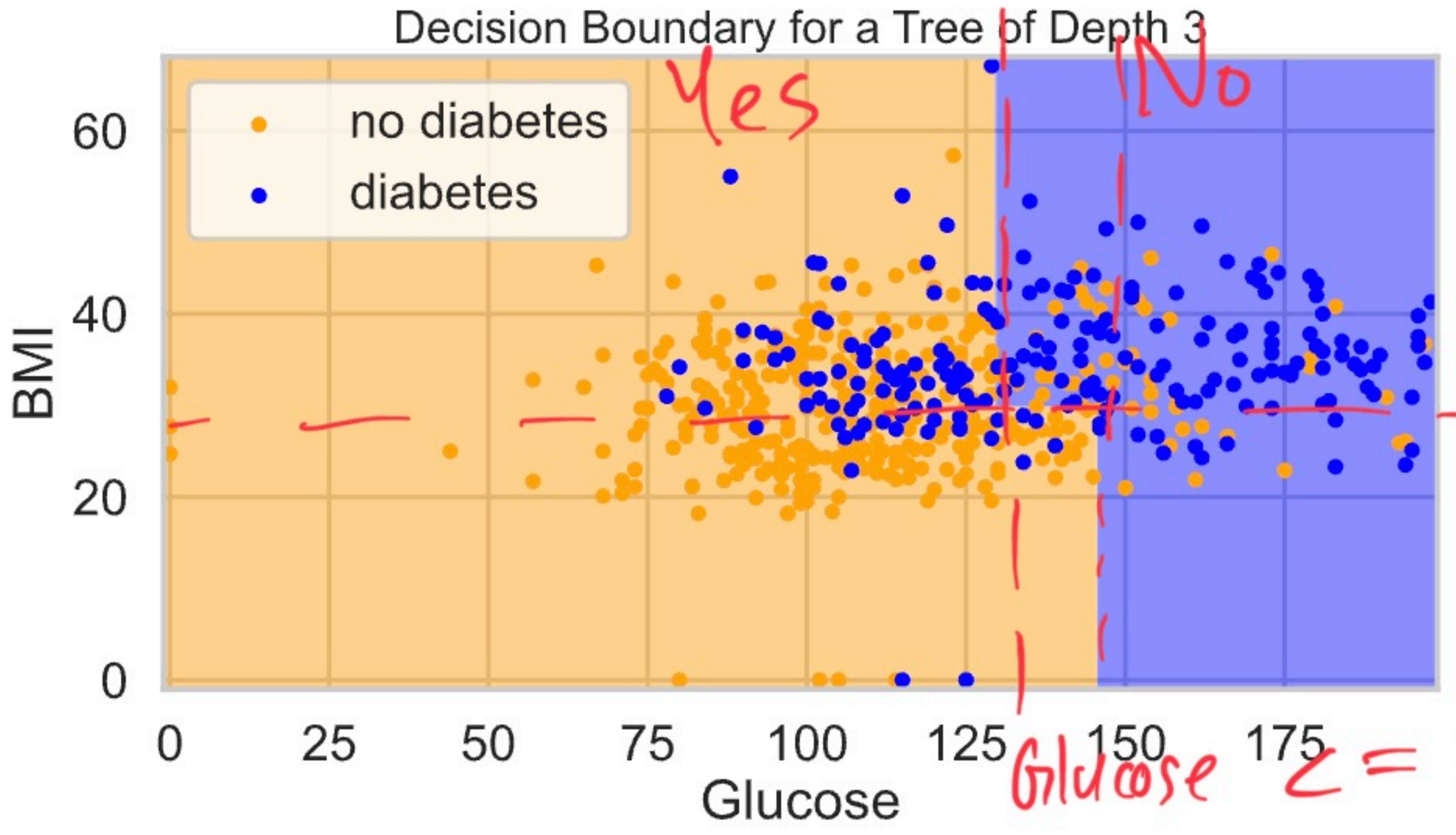
overlap

same glucose, same BMI, different classes!

if this happens, impossible to have 100% training acc.

Decision boundaries for a decision tree classifier

```
In [53]: util.show_decision_boundary(model_tree, X_train, y_train, title='Decision Boundary for a Tree of Depth 3')
```



Glucose <= 129.5

partition the feature space using Yes/No questions

- Observe that the decision boundaries – at least when we set `max_depth` to 3 – look less "jagged" than with the k-NN classifier.

localhost

Activity

ChickenClassifiers have many hyperparameters, one of which is height. As we increase the value of height, the model variance of the resulting ChickenClassifier also increases.

First, we consider the training and testing accuracy of a ChickenClassifier trained using various values of height. Consider the plot below.

Accuracy
0
0.2
0.4
0.6
0.8
1
10
20
30
height
Option 1
Option 2
Option 3

overfitting to train, accuracy ↑

height

height ↑, overfit to training set, complexity ↑

Which of the following depicts **training accuracy vs. height**?

○ Option 1

[✓ ○ Option 2]

○ Option 3

Which of the following depicts **testing accuracy vs. height**?

○ Option 1

○ Option 2

[✓ ○ Option 3]

after a "sweet spot", test accuracy ↓ since we overfit to training set