Regression in Use

How to Apply Linear Regression, Logistic regression and Cross Validation in Python.

Notation…

- Up to now we sticked to the notation of Prof. Andrew NG from Stanford University.
- From now on we are changing our notation, i.e., we will use the notation of Prof. Christopher Brooks from Michigan University.
- In particular:
  - $w_i = \theta_i \; for \; i \geq 1$
  - $b = \theta_0$
  - $\alpha = \lambda$
- In fact Prof Andrew NG used this new notation in his new lectures as well.

Ridge Regression

- Ridge regression learns w, b using the same least-squares criterion but adds a penalty for large variations in w parameters

$$RSS_{RIDGE}(\mathbf{w}, b) = \sum_{\{i=1\}}^{N} (\mathbf{y}_i - (\mathbf{w} \cdot \mathbf{x}_i + b))^2 \boxed{+ \alpha \sum_{\{j=1\}}^{p} w_j^2}$$

- Once the parameters are learned, the ridge regression prediction formula is the same as ordinary least-squares.
- The addition of a parameter penalty is called regularization.
- Regularization prevents overfitting by restricting the model, typically to reduce its complexity.
- Ridge regression uses L2 regularization: minimize sum of squares of w entries

Ridge Regression

- The influence of the regularization term is controlled by the α(previously λ) parameter.
- Higher α(previously λ) means
  - more regularization
  - and simpler models.

$$RSS_{RIDGE}(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (\boldsymbol{y}_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2 \boxed{+ \alpha \sum_{\{j=1\}}^{p} w_j^2}$$

- Please note, again, α here is the λ that we used previously. It’s not the step site.

Ridge Regression and Normalization.

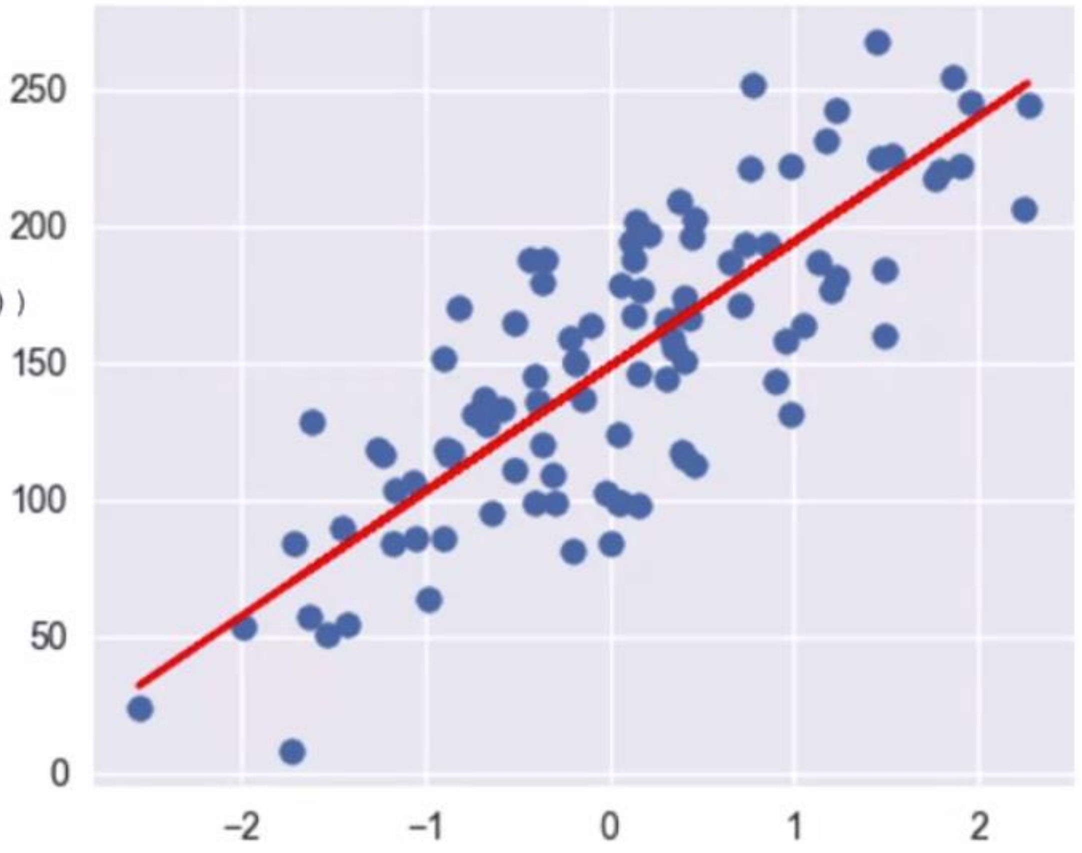
- Ridge regression is the linear regression with L2 regularization.
- Normalization works fine with ridge regression since it enables to apply fair penalty to each of the coefficients

Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test =
      train_test_split(X_R1, y_R1, random_state = 0)

linreg = LinearRegression().fit(X_train, y_train)

print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```

`linreg.coef_` `linreg.intercept_`

$$\hat{y} = \underset{\uparrow\ \texttt{linreg.coef\_}}{\mathbf{w_0}} x_0 + \underset{\uparrow\ \texttt{linreg.intercept\_}}{b}$$



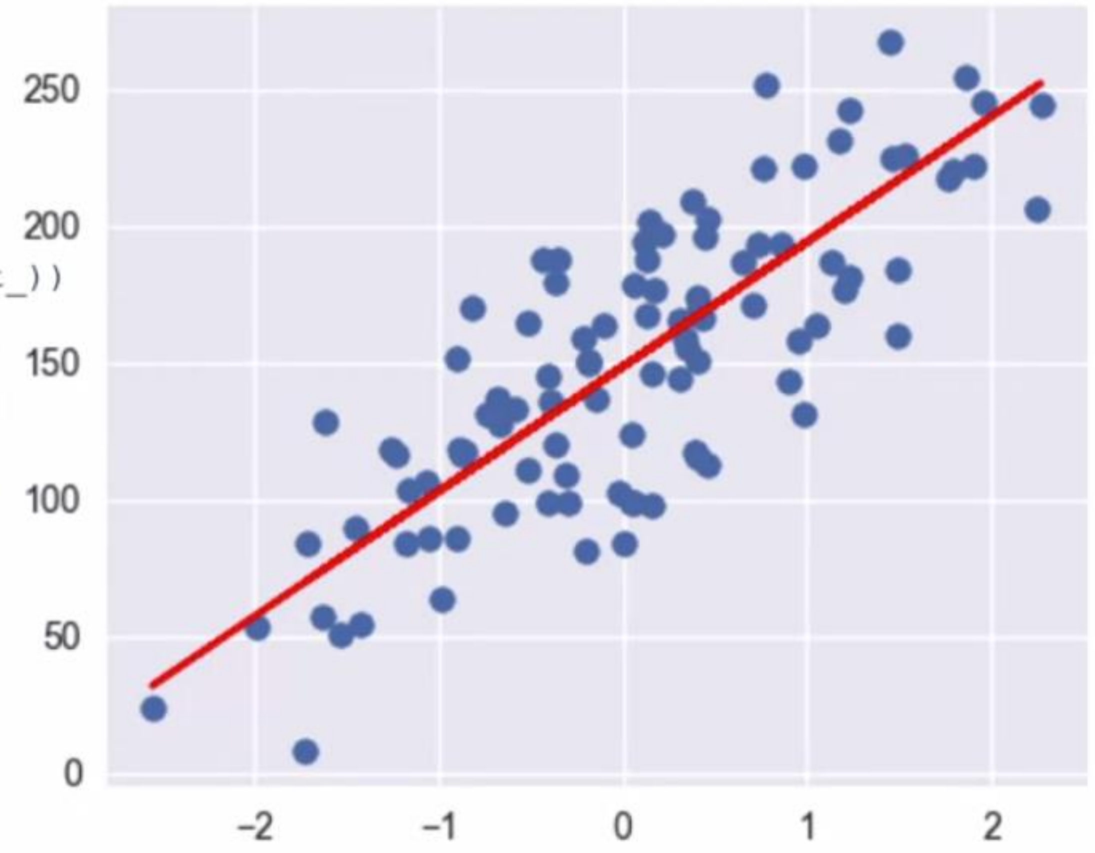
Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test =
          train_test_split(X_R1, y_R1, random_state = 0)

linreg = LinearRegression().fit(X_train, y_train)

print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```

Underscore denotes a quantity derived from training data, as opposed to a user setting.

`linreg.coef_` `linreg.intercept_`

$$\hat{y} = \mathbf{w_0} x_0 + b$$

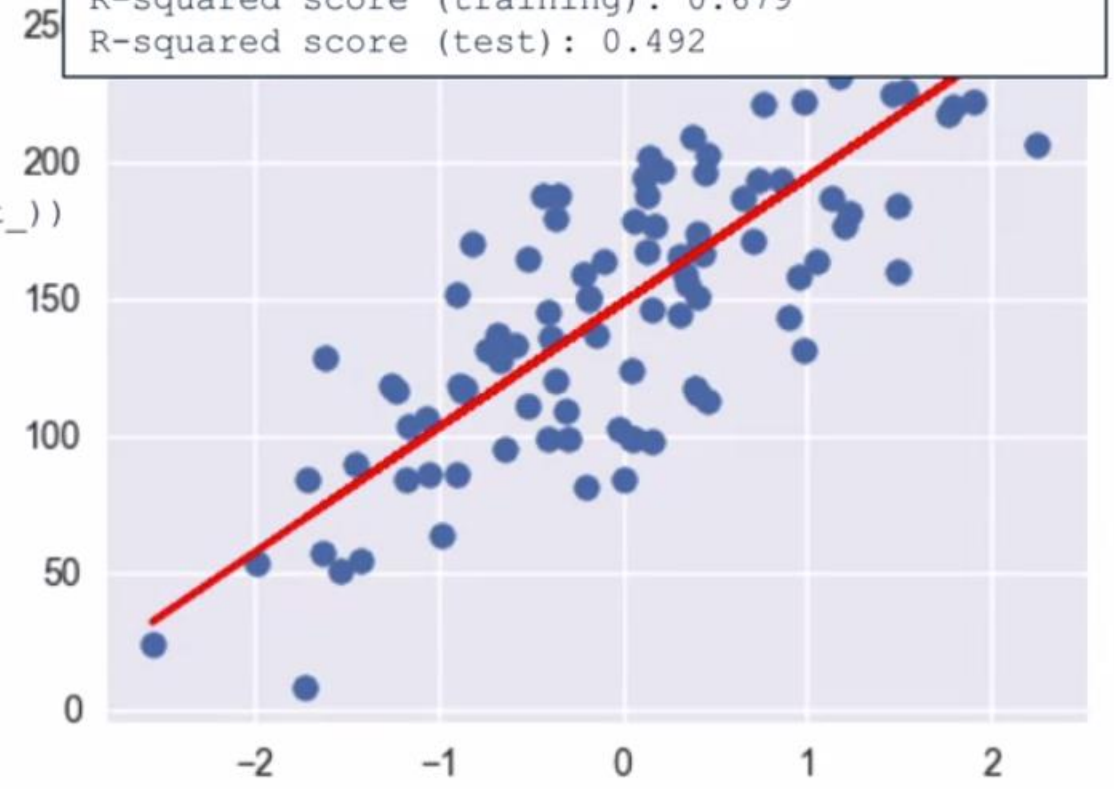


Least-Squares Linear Regression in Scikit-Learn

```
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test =
          train_test_split(X_R1, y_R1, random_state = 0)

linreg = LinearRegression().fit(X_train, y_train)

print("linear model intercept (b): {}".format(linreg.intercept_))
print("linear model coeff (w): {}".format(linreg.coef_))
```

```
linear model coeff (w): [ 45.70870465]
linear model intercept (b): 148.44575345658873
R-squared score (training): 0.679
R-squared score (test): 0.492
```

$$\hat{y} = \underbrace{\mathbf{w_0}}_{\texttt{linreg.coef\_}} x_0 + \underbrace{b}_{\texttt{linreg.intercept\_}}$$



The Need for Feature Normalization

- Important for some machine learning methods that all features are on the same scale (e.g. faster convergence in learning, more uniform or 'fair' influence for all weights)
  - *e.g. regularized regression, k-NN, support vector machines, neural networks, …*
- Can also depend on the data. More on feature engineering later in the course. For now, we do MinMax scaling of the features:
  - *For each feature x_i : compute the min value x_i^{MIN} and the max value x_i^{MAX} achieved across all instances in the training set.*
  - *For each feature: transform a given feature x_i value to a scaled version x_i' using the formula*

$$x_i' = (x_i - x_i^{MIN})/(x_i^{MAX} - x_i^{MIN})$$

Feature Normalization: The test set must use identical scaling to the training set

- Fit the scaler using the training set, then apply the same scaler to transform the test set.
- Do not scale the training and test sets using different scalers: this could lead to random skew in the data.

Lasso regression is another form of regularized linear regression that uses an L1 regularization penalty for training (instead of ridge's L2 penalty)

- L1 penalty: Minimize the sum of the absolute values of the coefficients

$$RSS_{LASSO}(\boldsymbol{w}, b) = \sum_{\{i=1\}}^{N} (\boldsymbol{y}_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b))^2 \boxed{+ \alpha \sum_{\{j=1\}}^{p} |w_j|}$$

- This has the effect of setting parameter weights in *w* to zero for the least influential variables. This is called a sparse solution: a kind of feature selection
- The parameter α controls amount of L1 regularization (default = 1.0).
- The prediction formula is the same as ordinary least-squares.
- When to use ridge vs lasso regression:
  - *Many small/medium sized effects: use ridge.*
  - *Only a few variables with medium/large effect: use lasso.*

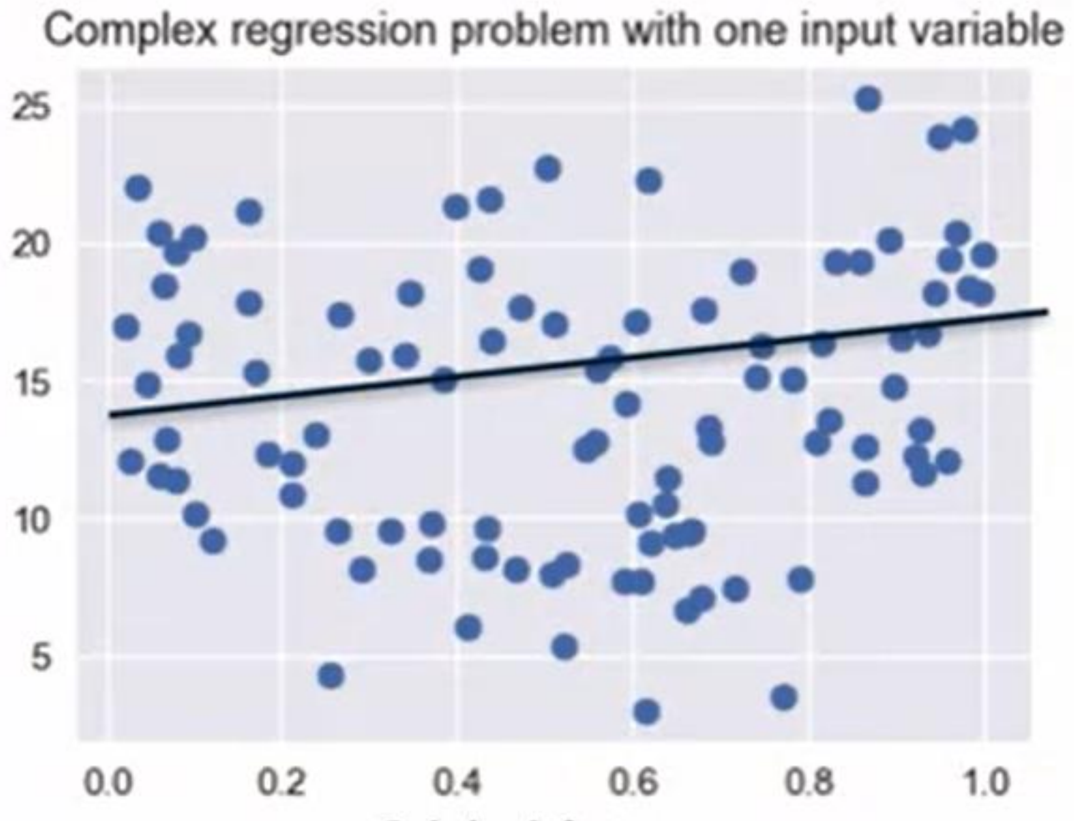
Polynomial Features with Linear Regression

$$\boldsymbol{x} = (x_0, x_1) \Rightarrow \boldsymbol{x}' = (x_0, x_1, x_0^2, x_0x_1, x_1^2)$$

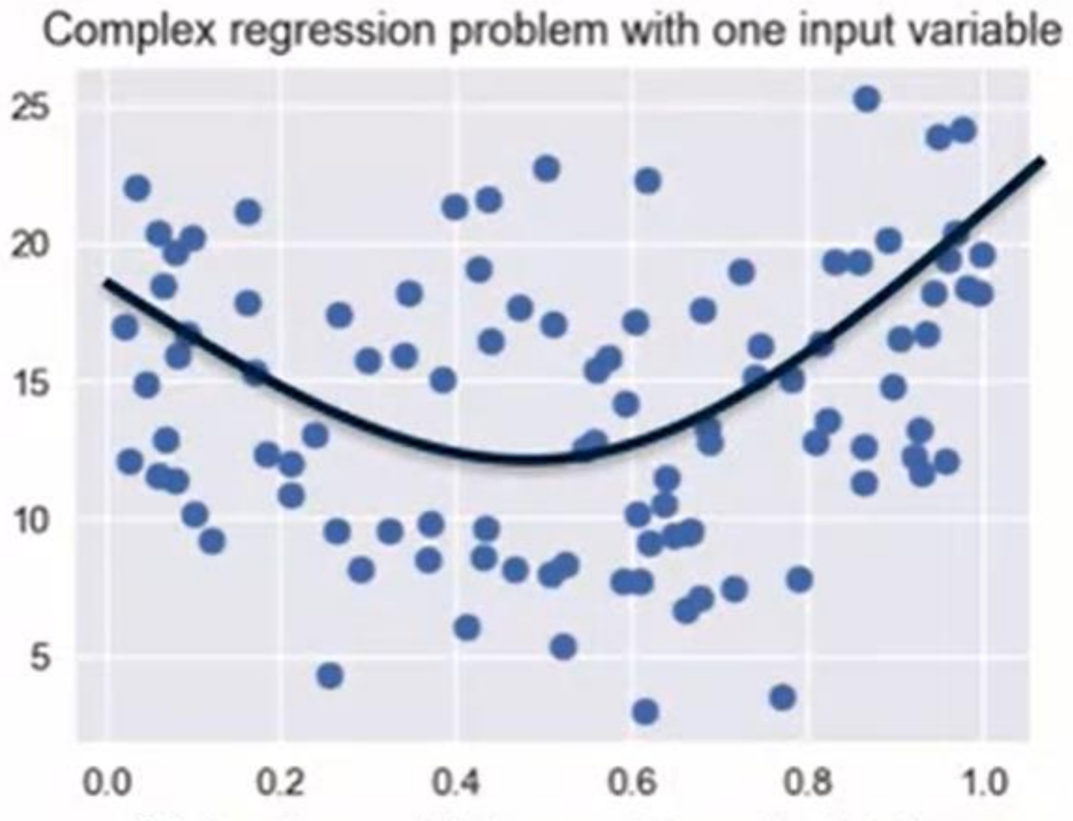
$$\hat{y} = \hat{w}_0 x_0 + \hat{w}_1 x_1 + \hat{w}_{00}\, x_0^2 + \hat{w}_{01} x_0 x_1 + \hat{w}_{11} x_1^2 + b$$

- Generate new features consisting of all polynomial combinations of the original two features (x_0, x_1).
- The *degree* of the polynomial specifies how many variables participate at a time in each new feature (above example: degree 2)
- This is still a weighted linear combination of features, so it's still a linear model, and can use same least-squares estimation method for *w* and *b*.

Least-Squares Polynomial Regression



Original feature



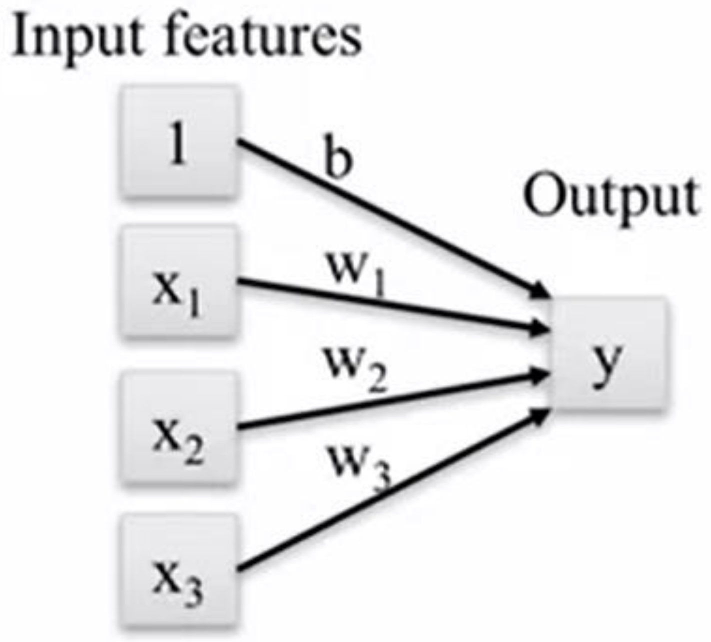
With polynomial degree 2 (quadratic) feature

Polynomial Features with Linear Regression

- **Why would we want to transform our data this way?**
  - *To capture interactions between the original features by adding them as features to the linear model.*
  - *To make a classification problem easier (we'll see this later).*
- **More generally, we can apply other non-linear transformations to create new features**
  - *(Technically, these are called non-linear basis functions)*
- **Beware of polynomial feature expansion with high degree, as this can lead to complex models that overfit**
  - *Thus, polynomial feature expansion is often combined with a regularized learning method like ridge regression.*

Logistic Regression

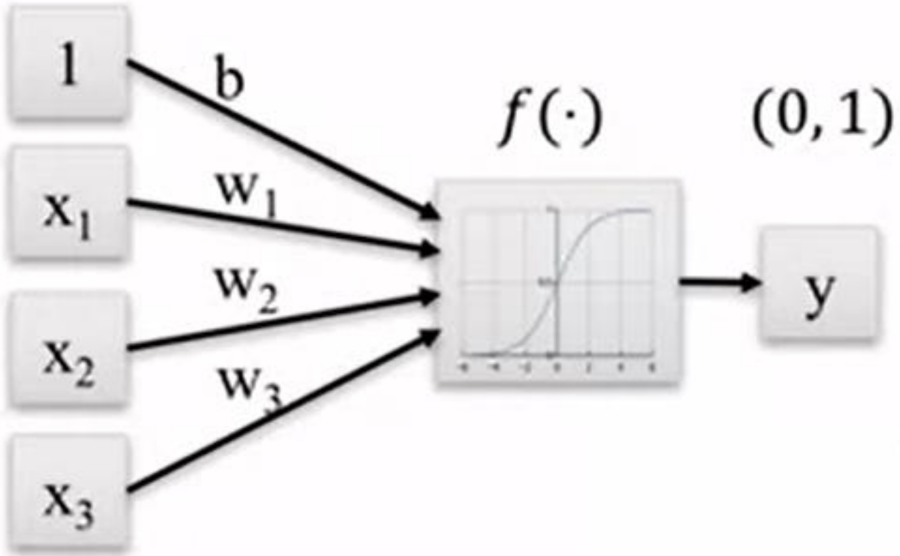
Linear Regression

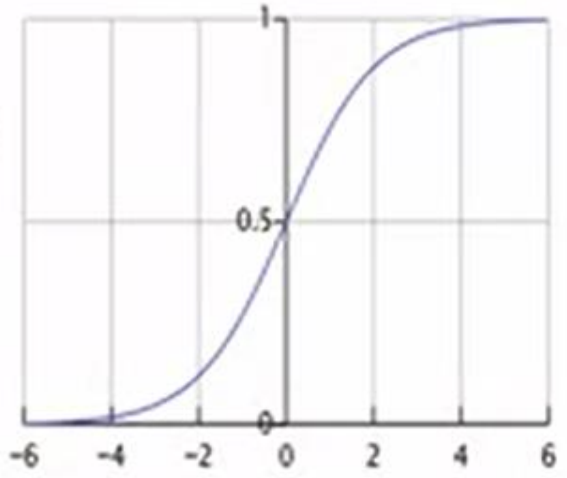


$$\hat{y} = \hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n$$

Linear models for classification: Logistic Regression

Input features



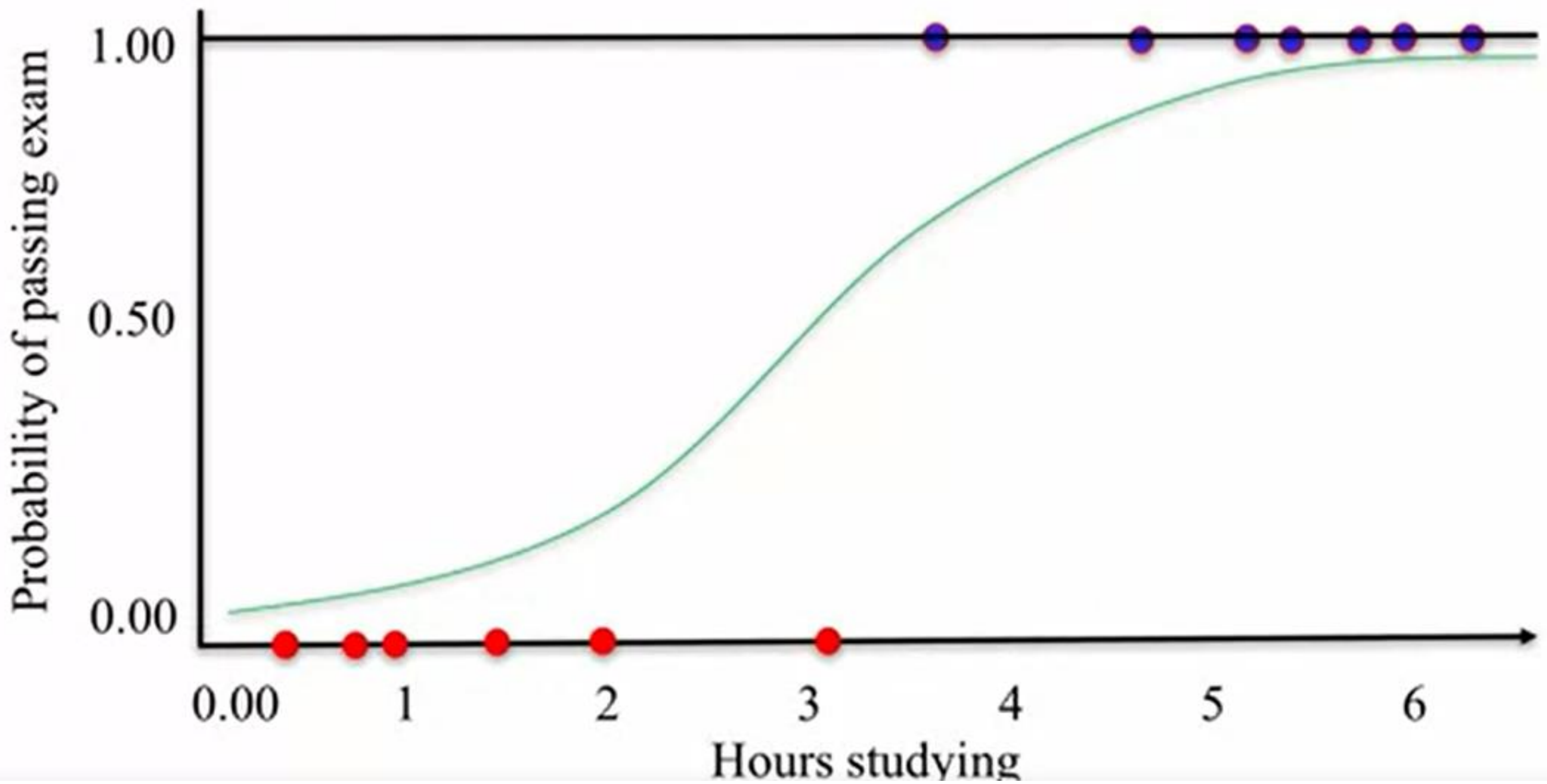


The logistic function transforms real-valued input to an output number y between 0 and 1, interpreted as the probability the input object belongs to the positive class, given its input features $(x_0, x_1, \ldots, x_n)$

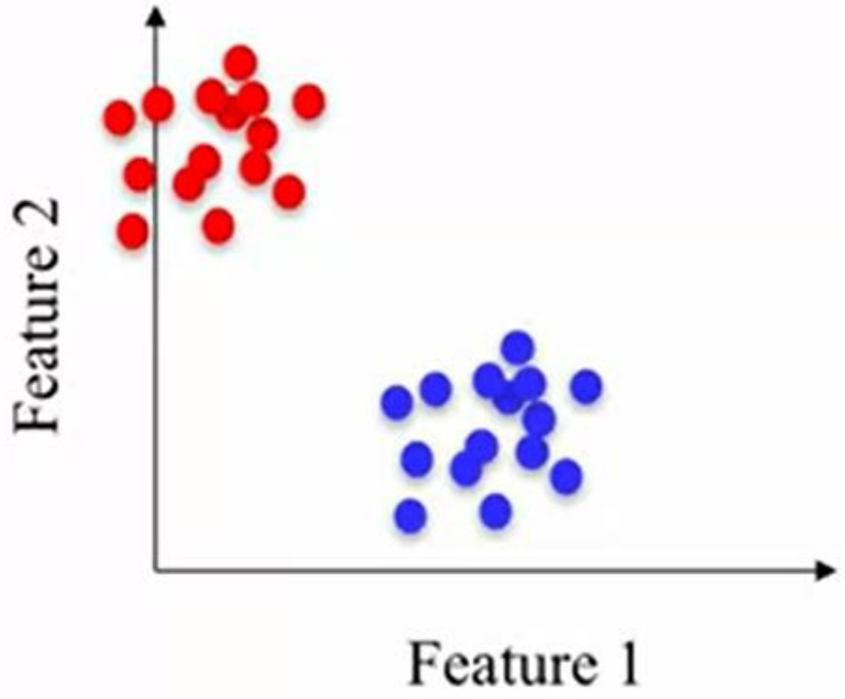
$$\hat{y} = \text{logistic}(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)$$

$$= \frac{1}{1 + \exp\left[-(\hat{b} + \hat{w}_1 \cdot x_1 + \cdots \hat{w}_n \cdot x_n)\right]}$$

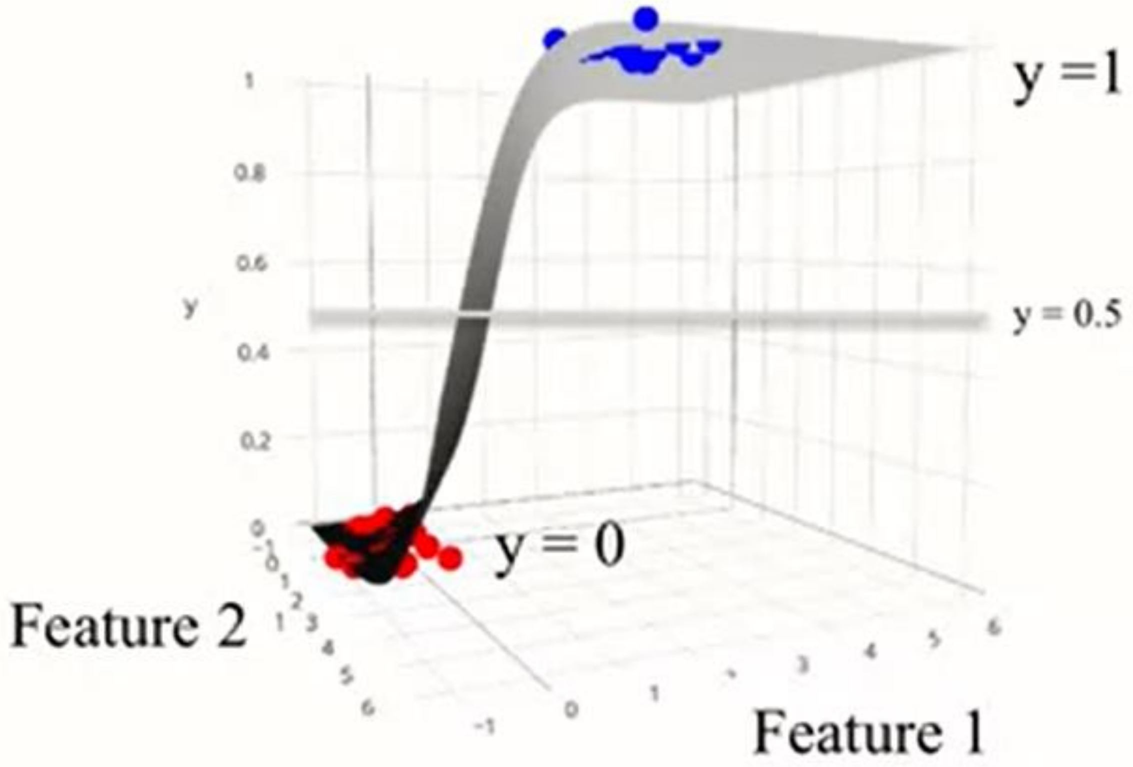
Linear models for classification: Logistic Regression



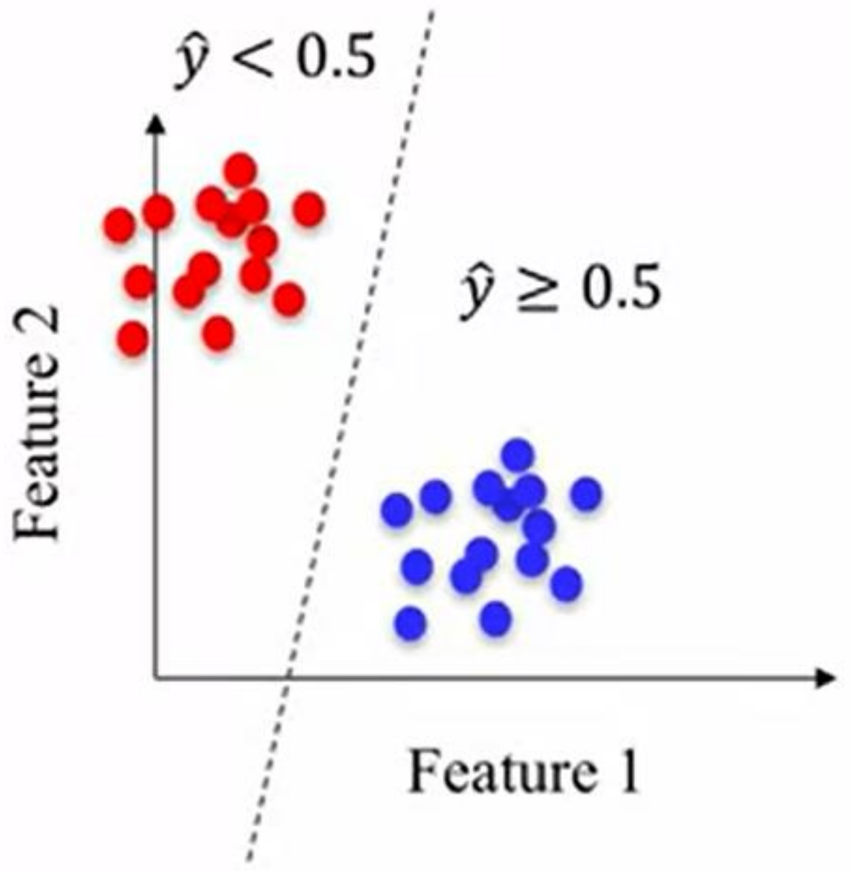
Logistic Regression for binary classification



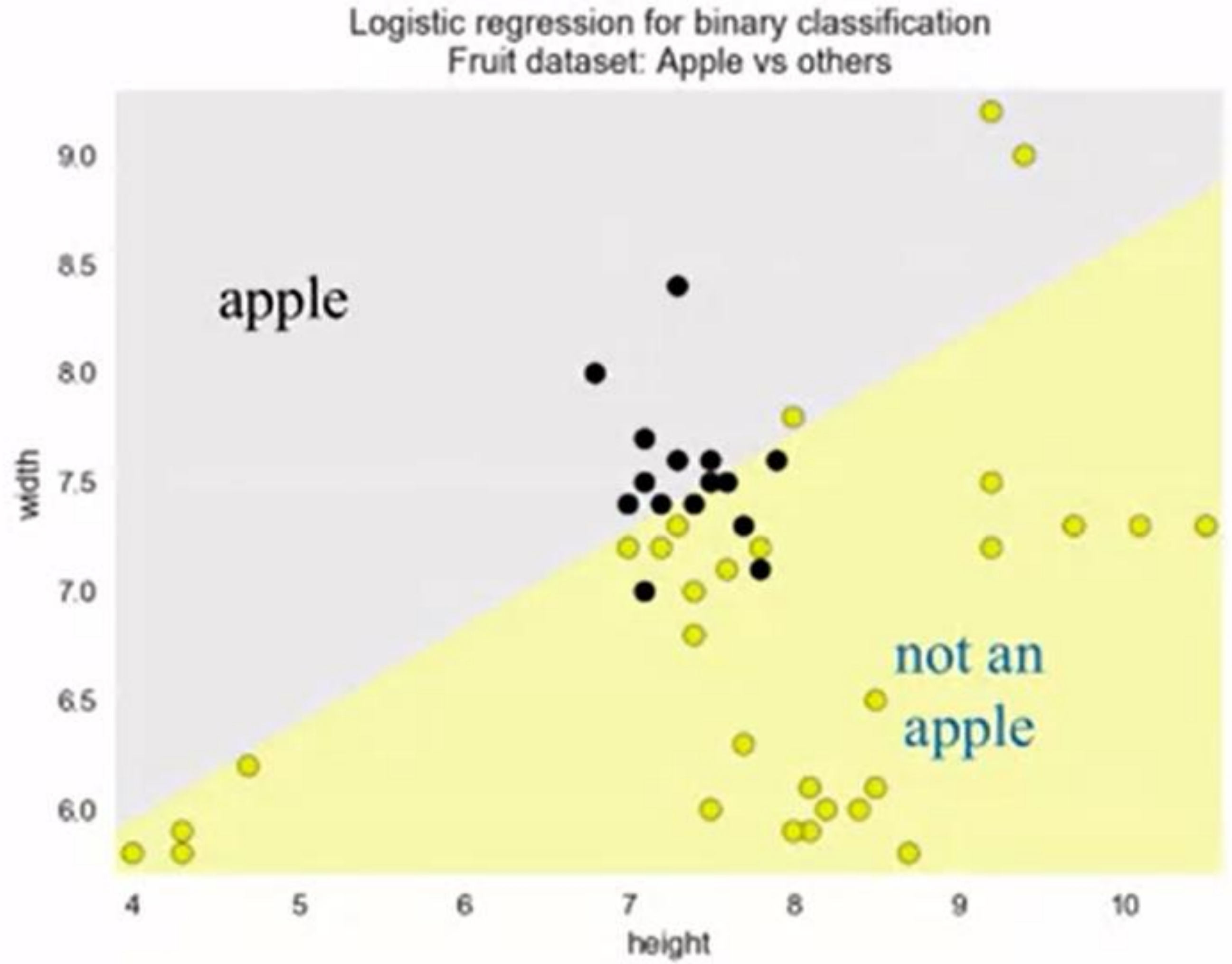
Logistic Regression for binary classification



Logistic Regression for binary classification

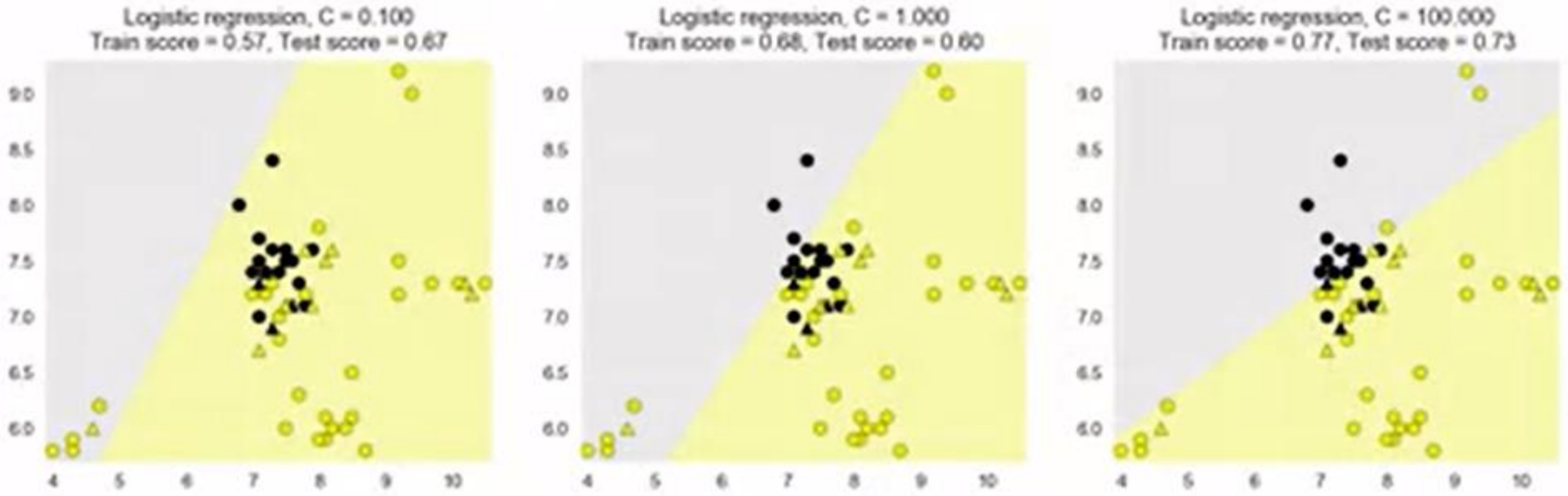


Simple logistic regression problem: two-class, two-feature version of the fruit dataset



Logistic Regression: Regularization

- L2 regularization is 'on' by default (like ridge regression)
- Parameter C controls amount of regularization (default 1.0) The inverse of alpha...
- As with regularized linear regression, it can be important to normalize all features so that they are on the same scale.

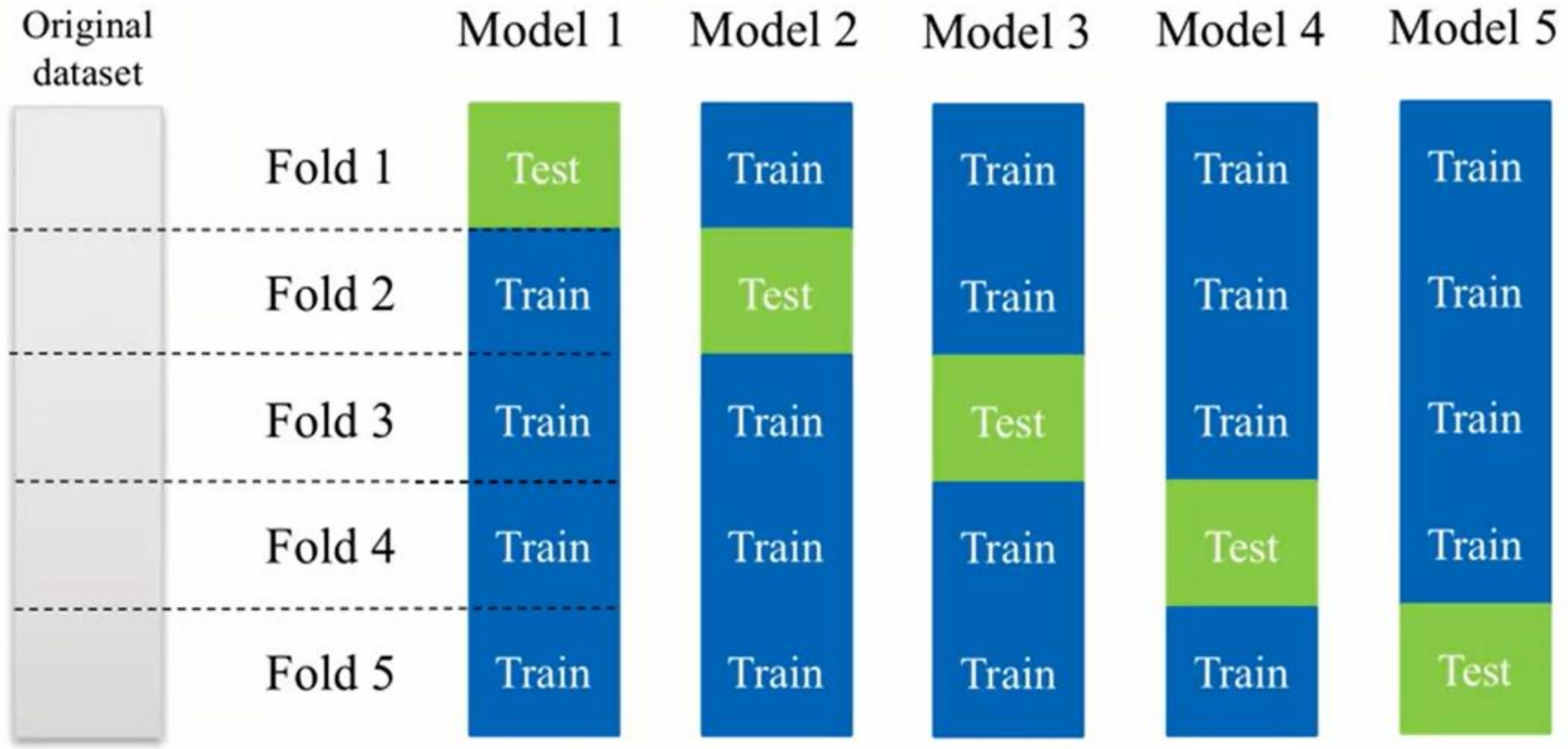


Cross Validation

Cross Validation

- Cross-validation is a method that goes beyond evaluating a single model using a single Train/Test split of the data by using **multiple** Train/Test splits, each of which is used to train and evaluate a separate model.
- You may have noted that by choosing different values for the random state seed parameter in the Train/Test split function, when you're working on some examples or assignments, that the accuracy score you get from running a classifier can vary quite a bit just by chance depending on the specific samples that happen to end up in the training set.
- Cross-validation basically gives more stable and reliable estimates of how the classifiers likely to perform on average by running multiple different training test splits and then averaging the results, instead of relying entirely on a single particular training set.
- You need roughly k times more time.

Cross-validation Example (5-fold)



	Model 1	Model 2	Model 3	Model 4	Model 5
Fold 1	Test	Train	Train	Train	Train
Fold 2	Train	Test	Train	Train	Train
Fold 3	Train	Train	Test	Train	Train
Fold 4	Train	Train	Train	Test	Train
Fold 5	Train	Train	Train	Train	Test

```
RBF-kernel SVC (with MinMax scaling) test set accuracy: 0.96
```

Cross-validation

Example based on k-NN classifier with fruit dataset (2 features)

```
In [30]: from sklearn.model_selection import cross_val_score

         clf = KNeighborsClassifier(n_neighbors = 5)
         X = X_fruits_2d.as_matrix()
         y = y_fruits_2d.as_matrix()
         cv_scores = cross_val_score(clf, X, y)

         print('Cross-validation scores (3-fold):', cv_scores)
         print('Mean cross-validation score (3-fold): {:.3f}'
              .format(np.mean(cv_scores)))
```

```
Cross-validation scores (3-fold): [ 0.77  0.74  0.83]
Mean cross-validation score (3-fold): 0.781
```

```
In [ ]:
```

Stratified Cross-validation

(Folds and dataset shortened for illustration purposes.)

fruit_label	fruit_name
1	Apple
1	Apple
1	Apple
1	Apple
1	Apple
2	Mandarin
...	...
3	Orange
...	...
4	Lemon
4	Lemon
4	Lemon
4	Lemon
4	Lemon

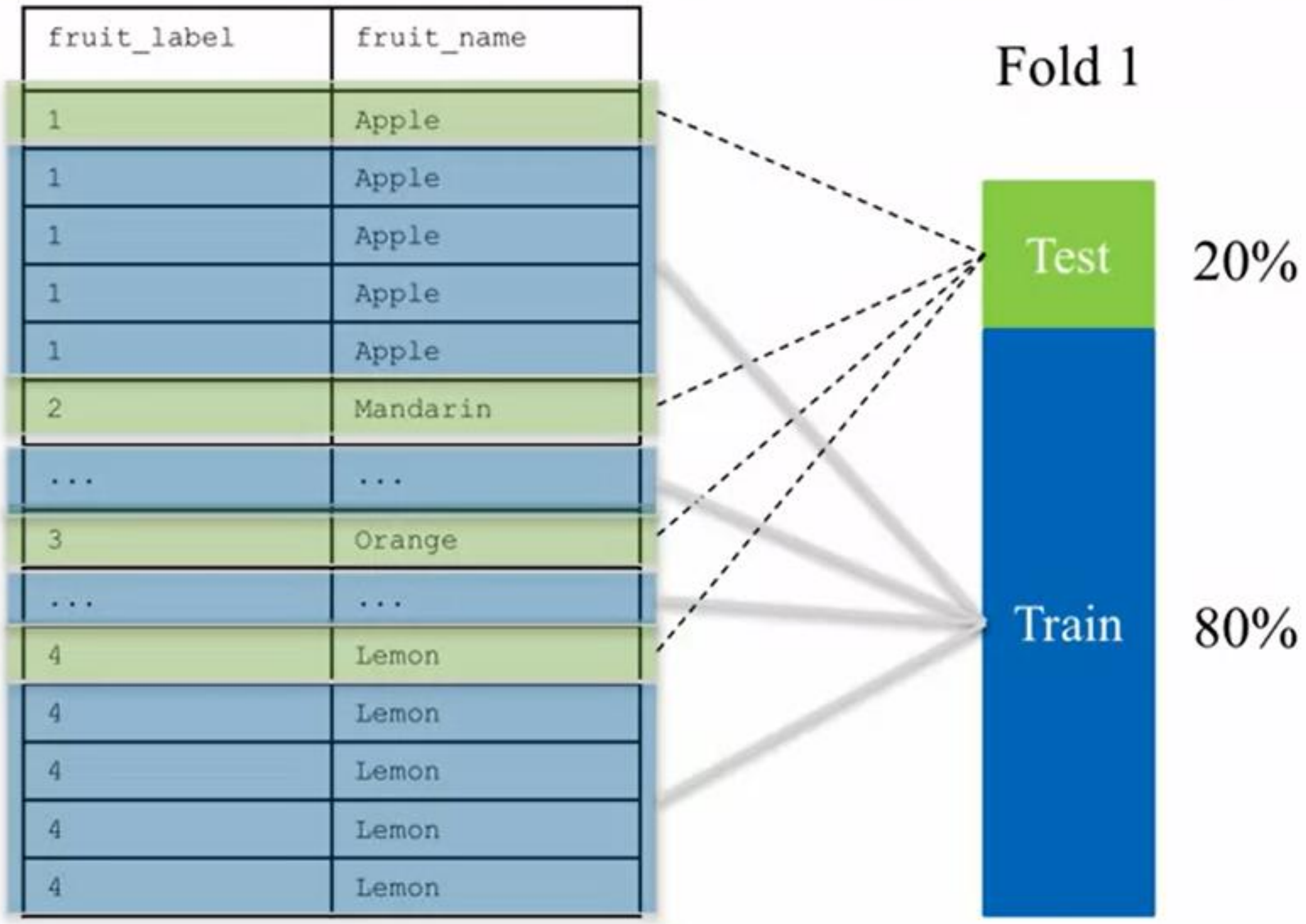
Example has 20 data samples
= 4 classes with 5 samples each.

5-fold CV: 5 folds of 4 samples each.

Fold 1 uses the first 20% of the dataset as the test set, which only contains samples from class 1.

Classes 2, 3, 4 are missing entirely from test set and so will be missing from the evaluation.

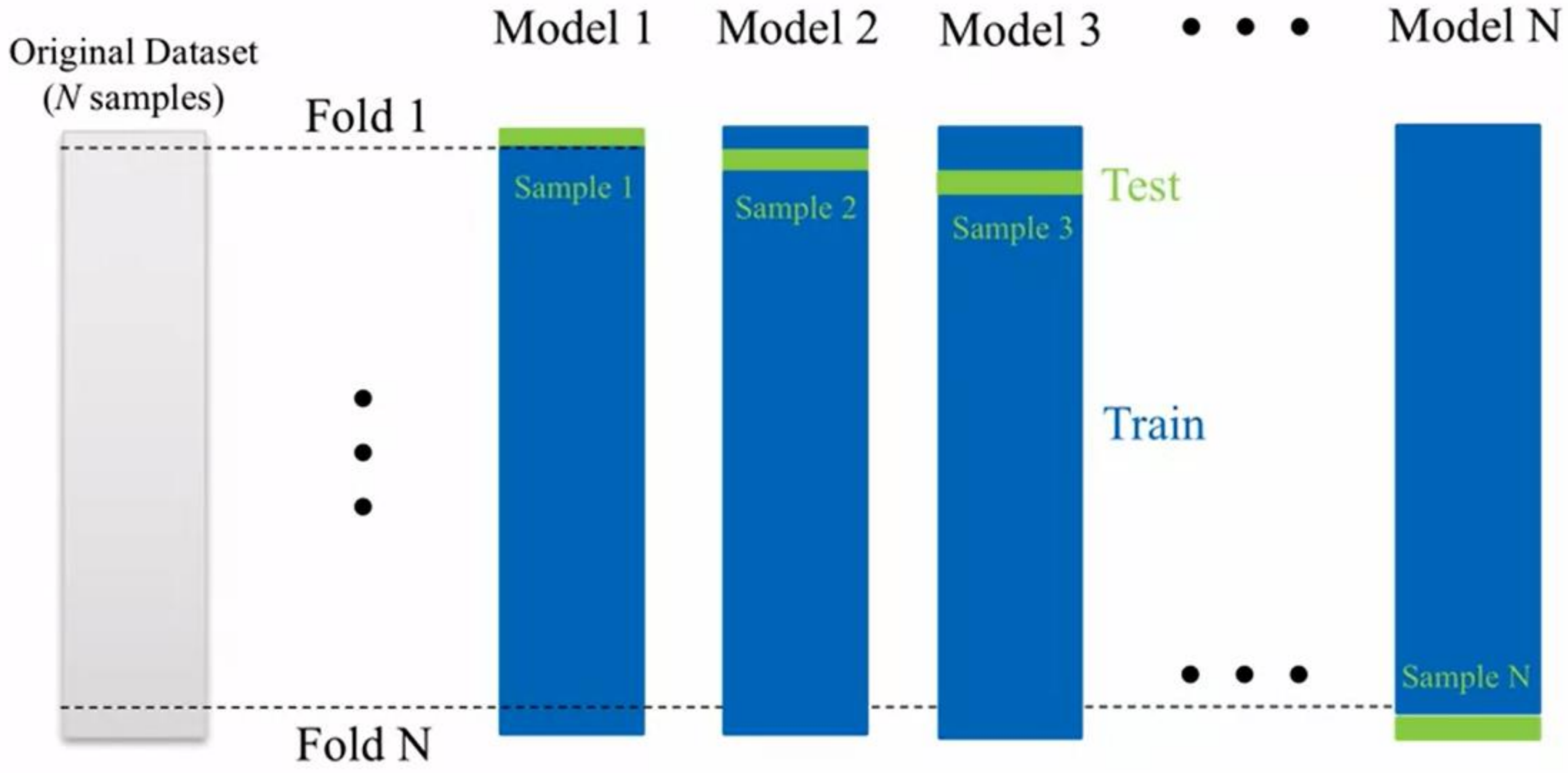
Stratified Cross-validation



fruit_label	fruit_name
1	Apple
1	Apple
1	Apple
1	Apple
1	Apple
2	Mandarin
...	...
3	Orange
...	...
4	Lemon
4	Lemon
4	Lemon
4	Lemon
4	Lemon

Stratified folds each contain a proportion of classes that matches the overall dataset. Now, all classes will be fairly represented in the test set.

Leave-one-out cross-validation (with *N* samples in dataset)



How to Finalize a Model?

- You finalize a model by applying the chosen machine learning procedure on **all of your data.**
- That's it!

- With the finalized model, you can:

  - Save the model for later or operational use.
  - Make predictions on new data.

- What about the cross-validation models or the train-test datasets?
  - They've been discarded. They are no longer needed. They have served their purpose to help **you choose a procedure to finalize**.