Machine Learning

Professor: Horacio Larreguy
TA: Eduardo Zago

ITAM Applied Research 2 / Economics Research Seminar,
11/01/2023

General Perspective

Introduction to Machine Learning

Probit and Logit

Regularizers: LASSO and L2

Random Forest

What is Machine Learning?

- Field of AI, involves **teaching computers** to learn from data.
- **Objective:** make automated predictions/decisions.
- **Road Map:** training a model that can learn patterns and relationships in the data.
- **Mathematically:**

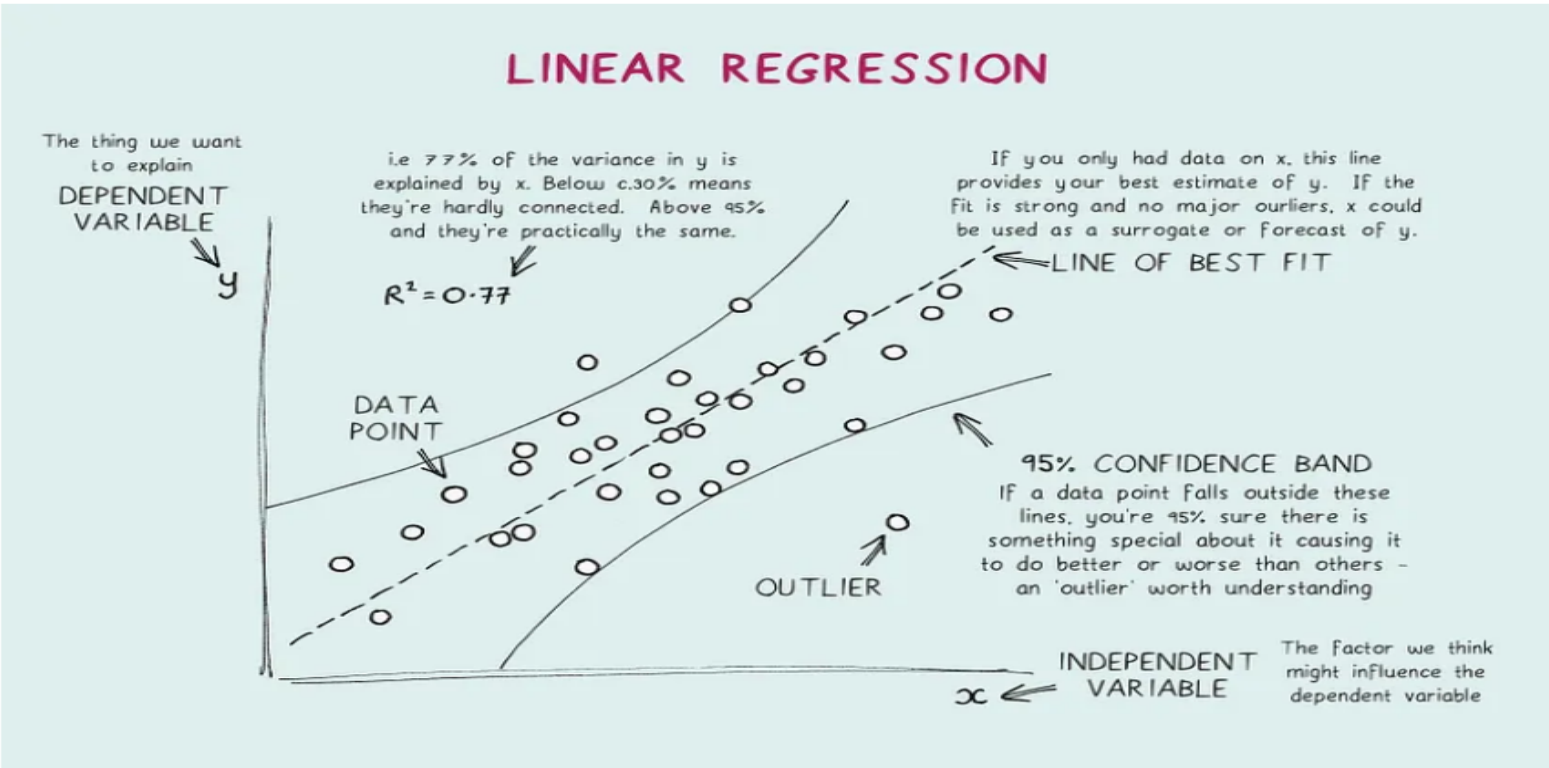
$$y = f(x; w) \tag{1}$$

Where $\mathbf{x}$ is the input data, $\mathbf{f}()$ is a mapping function, $\mathbf{w}$ is the set of parameters to be trained, $\mathbf{y}$ is the expected output.

Types of Problems

- **Regression:** A problem where the output y lives in a continuous space. $f()$ returns a real value $y \in \mathbf{R}^1$

Figure: Regression

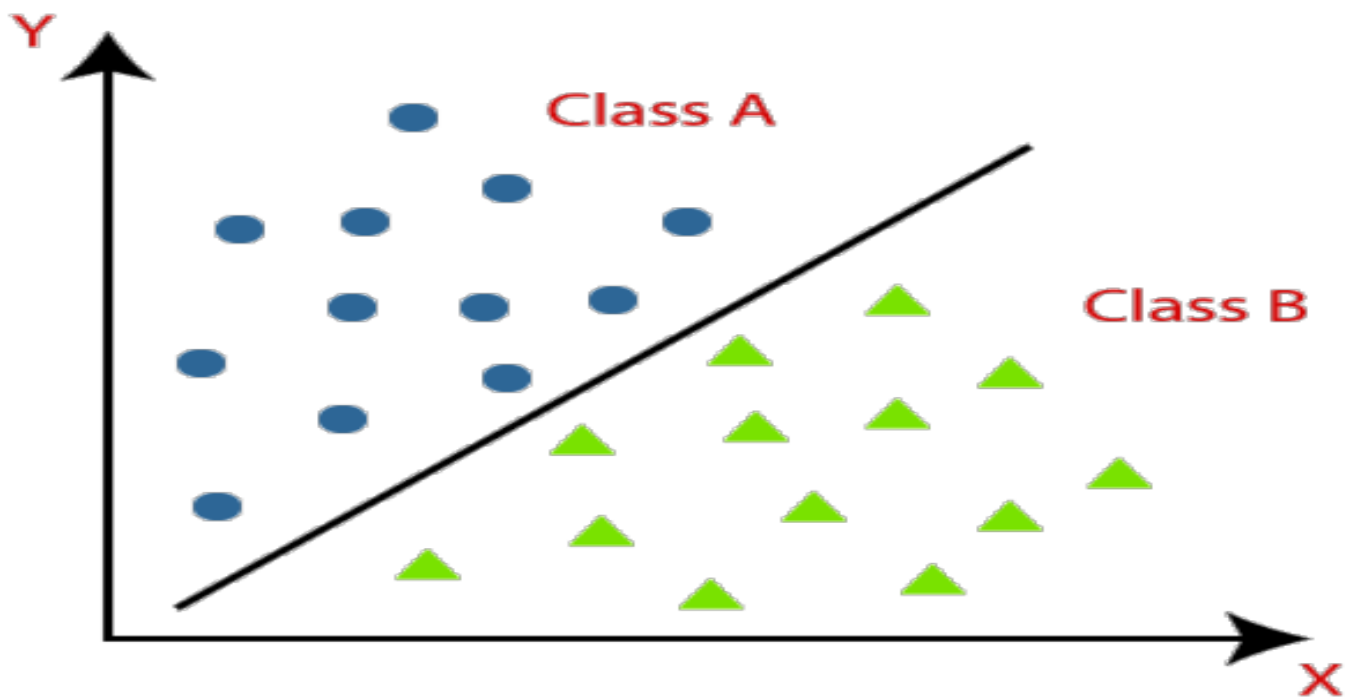


[1] Image by Dan White 1000

Types of Problems

- **Classification:** A problem where the output y lives in a discrete space. $f()$ returns a categorical value $y \in \{b_1, b_2, ..., b_n\}$[2]

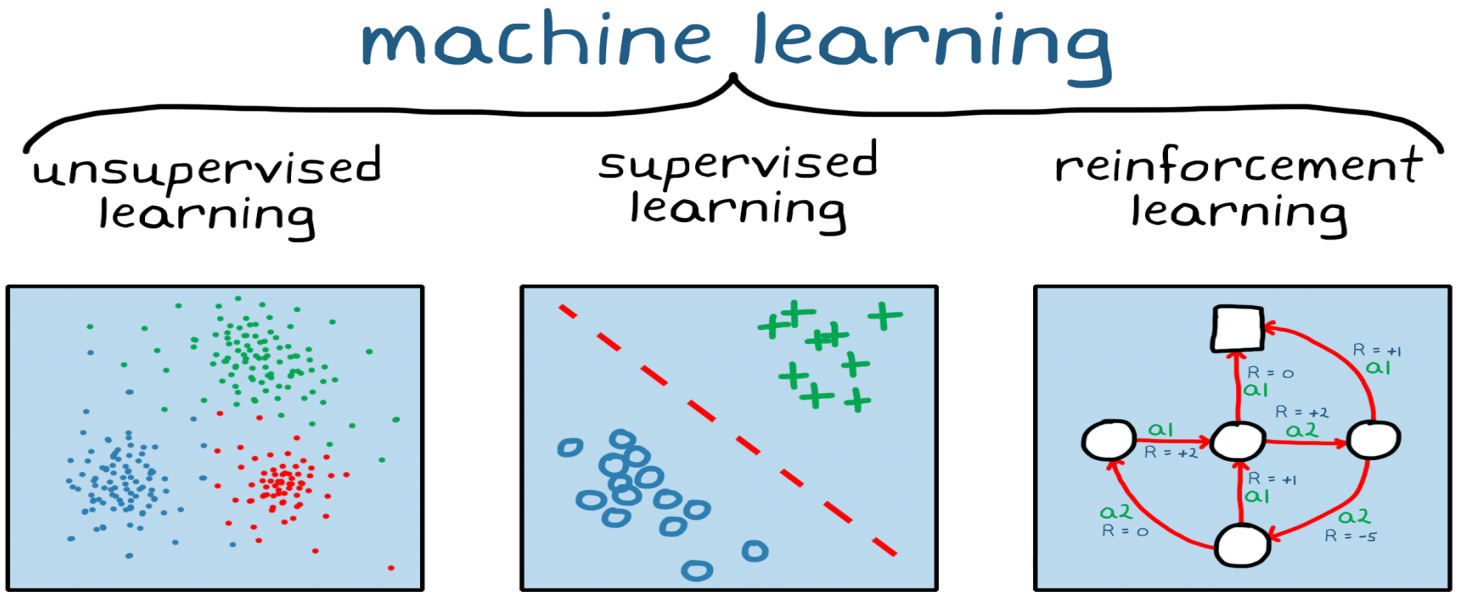
Figure: Classification



[2]Image taken from ITAM's Deep Learning Course Slides

Types of ML

1. **Supervised learning** involves learning from labeled data.
2. **Unsupervised learning** involves learning from unlabeled data.
3. **Reinforcement learning** involves learning from feedback in a dynamic environment.(MathWorks)



Data Sets

- In Supervised Learning, we will be using a **data-driven** approach, meaning that we will develop training algorithms relying on a **labeled data set**.
- These training algorithms will learn patterns and characteristics of the data by processing many examples of each class.
- This labeled data set will be divided into three: **training**, **validation** and **test** sets.

Data Sets

1. **Training set:** Initial set used to estimate the values for our parameters $\{w_i\}$, through error minimization and parameter updating.

2. **Validation set:** Used to evaluate how well the model performs on **data it has never seen**. We can improve the accuracy through the modification of the *hyper-parameters*.

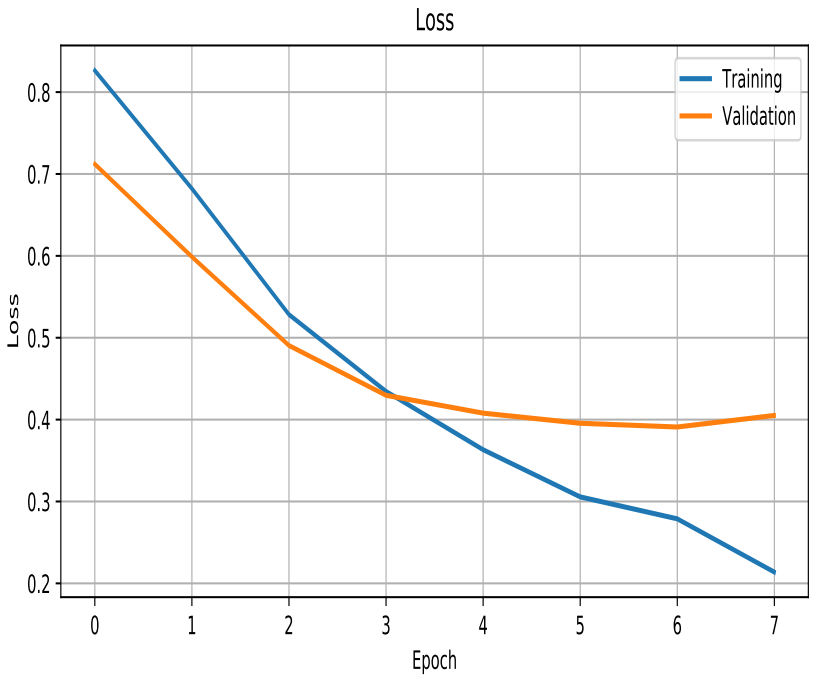
3. **Test set:** This data set is used for final evaluation of the performance of the model. Final test for **generalization**.

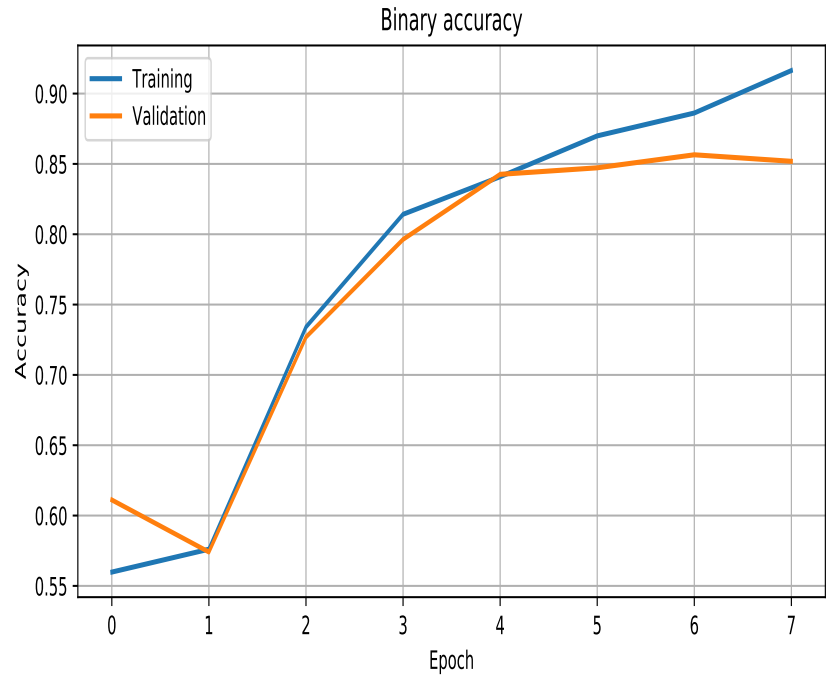
Evaluation Metrics

- To evaluate the performance of our models we will mostly use two metrics: **accuracy** and **loss**.
- **Accuracy:** is a measure of how often the model makes correct predictions, expressed as a percentage.
- **Loss:** measures the difference between the predicted output and the actual output for a given input.
- When training a model, we will look to have **low loss** and **high accuracy**, in all of our data sets.

The Learning Curve

- A **learning curve** is a graph that shows the performance of a machine learning model on a task over time, as the amount of training data increases.[3]

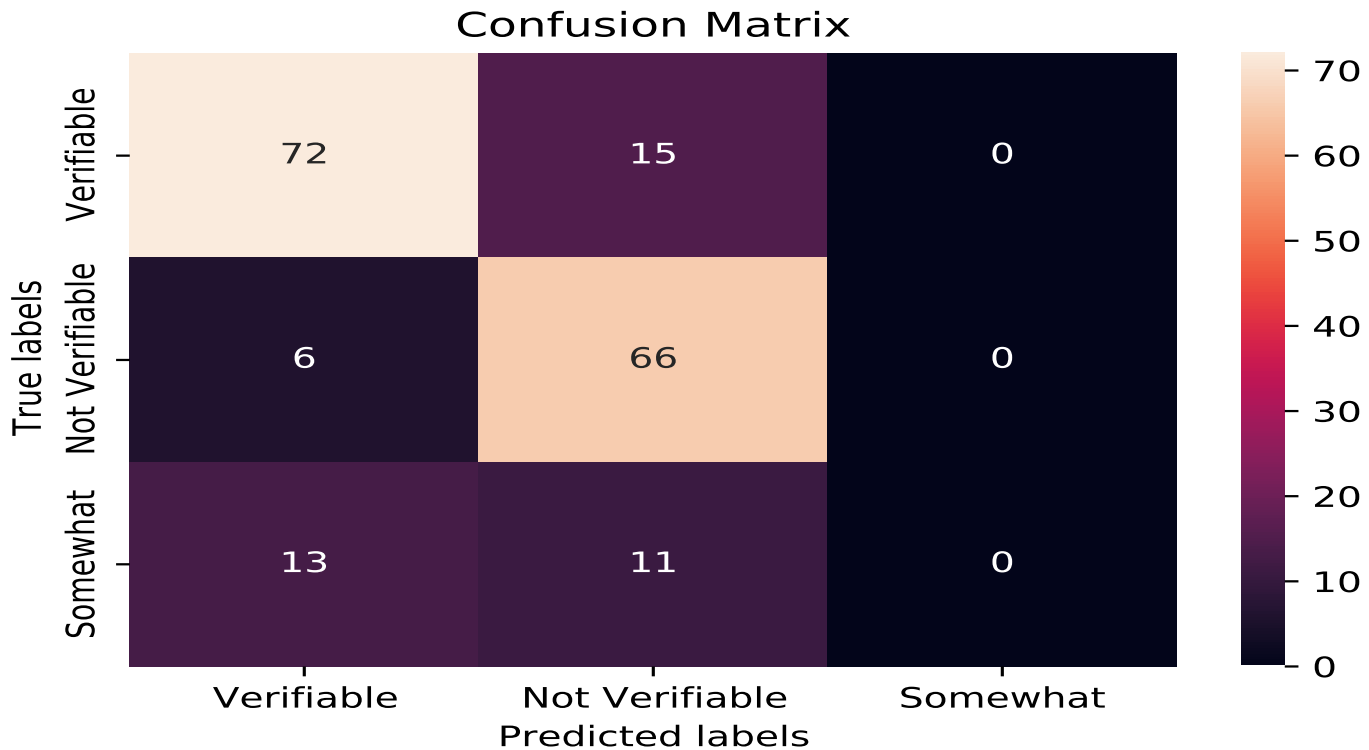




[3] An epoch is when all the training data is used at once and is defined as the total number of iterations of all the training data in one pass.

The Confusion Matrix

- A **confusion matrix** is a table used to evaluate the performance of a classification model. It shows the number of true positives, false positives, true negatives, and false negatives for each class.



Introduction to scikit-learn package

- Python contains some of the richest libraries to do Machine and Deep Learning.
- `sklearn` (scikit-learn) is the go to package for Machine Learning in Python. Provides simple and efficient tools for predictive data analysis.
- Built on NumPy, SciPy, and matplotlib. We can install it by running the following command:

```
# Install sklearn using pip:
pip install sklearn
```

Hyper-Parameters

- Before going forward, we must explain what a *hyper-parameter* is and why is it so important
- **Hyper-parameters** are are parameters chosen by the programmer that can improve or worsen the results of our model.
- External variables of the model set through trial and error.
- One great example is the **train-test split ratio**, which is how much of the whole data set do we want to be used for training and how much for test.[4]

[4] The go to split is 80% training and 20% test.

Hyper-Parameters

- For our own data sets, where X are the features and y the leabels or outcome, we could simply use `sklearn` function `train_test_split()`.

```
X_train, X_test, y_train,
        y_test = train_test_split(X, y,
        test_size = 0.2, random_state = 42)
```

- Where we are dividing our data set into a training set $[X\_train, y\_train]$ and a test set $[X\_test, y\_test]$.

Data-Generating Process

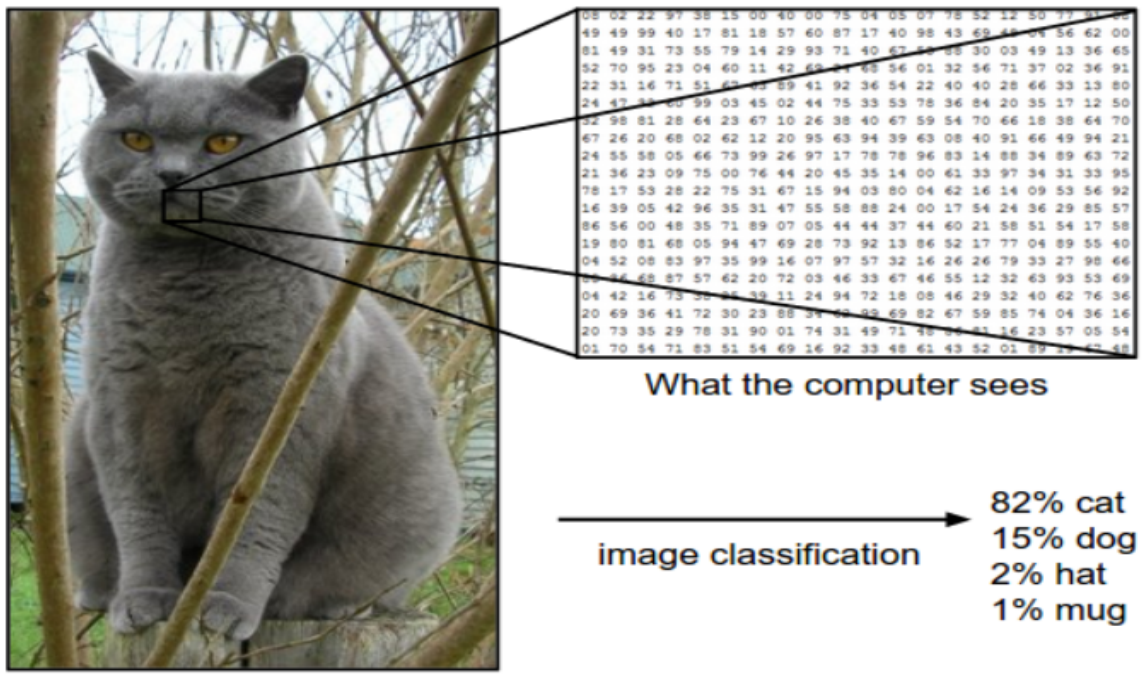
- Recall from Equation 1 that when we are **training a model**, we are learning or estimating the values of w that best map x on y.
- The model that best maps x on y is also called the **best fit**. These model will be the one that accurately matches the **True Data-Generating Process** of the context in hand.
- True Data-Generating Process = True Distribution of our Data
- In other words, we are looking for the model that best fits the distribution of our data.

Example: Image Classification

- We, as humans, can accurately classify between a dog and a cat when presented with images of both.
- We use visual characteristics to classify them, somehow we know the distribution of these characteristics, **but how could a computer do something similar?**

Example: Image Classification

- A computer, and most importantly, a Machine Learning model will try to learn the distribution of the numerical data available for each class.
- This could be numerical features such as height, size, weight, etc.
- Or pixel data, taken directly from the images.

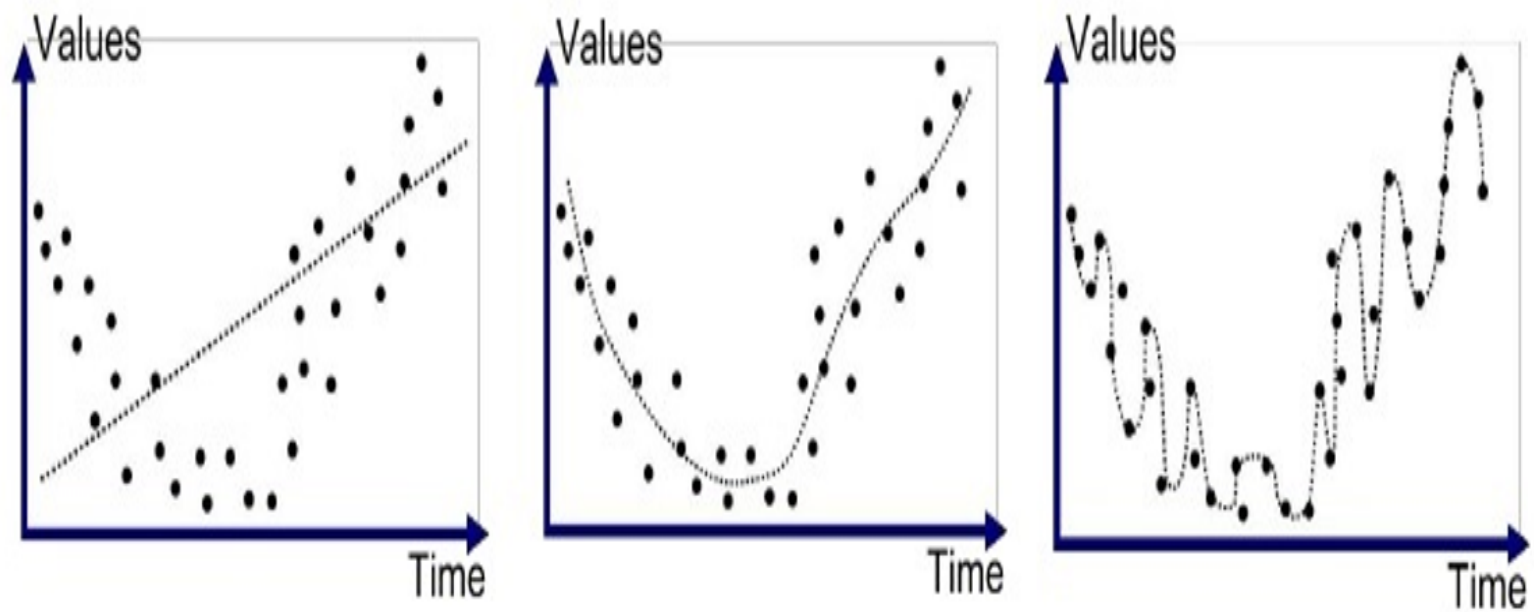


Example: Image Classification

- Thus, the ML and DL model's objective, will be to learn this data generating process, this distribution.
- When training a model, one can deal with 3 particular situations:

1. **Under-fitting**: training excluded the true data-generating process and induces bias (high training error).
2. Training matched the true data-generating process.
3. **Over-fitting:** included the generating process but also other possible generating processes. Variance rather than bias dominates the estimation error.

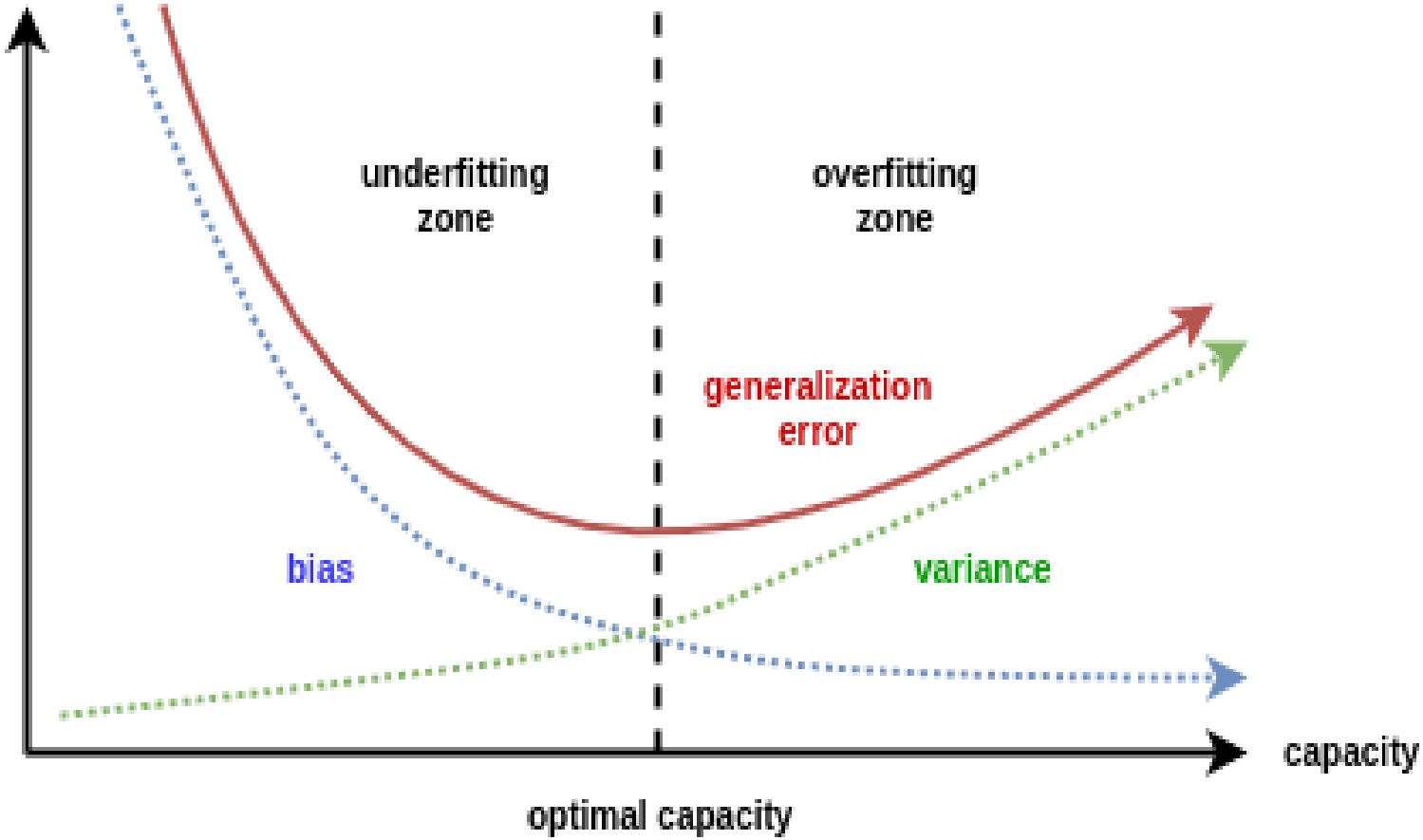
Visual Representation, Types of Fits



Underfitted | Good Fit/Robust | Overfitted

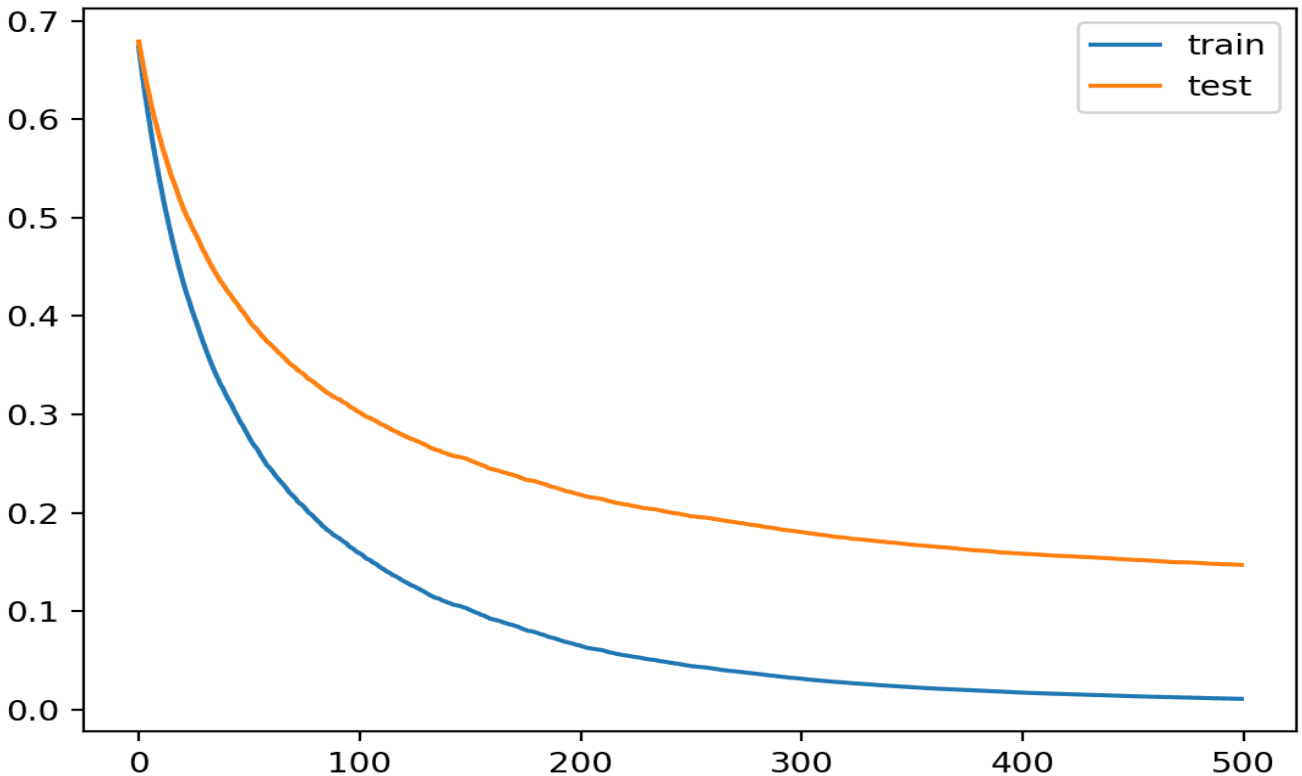
Under-fitting

- High training error and low training accuracy $\Longrightarrow$ under-fitted



Over-fitting

- Low training error and high training accuracy, but high validation error and low validation accuracy $\Longrightarrow$ over-fitted



Understanding Probit and Logit

- Probit and Logit are both types of generalized linear models (GLMs) that are used for binary classification.
- Recall that for a regression model, the prediction $\hat{y}$ is given by:

$$\hat{y} = \hat{\beta}_0 + \sum_{i=1}^{m} \hat{\beta}_i X_i \tag{2}$$

- When this outcome variable is a dummy, the coefficients prediction is taken as a probability (LPM).
- This prediction may deviate from the range between zero and one.

Logit

- Logit estimates the probability of belonging to a certain class (binary classification) by using a set of predictors (independent variables) X and the Logistic Function.
- The Logistic Function is the next:

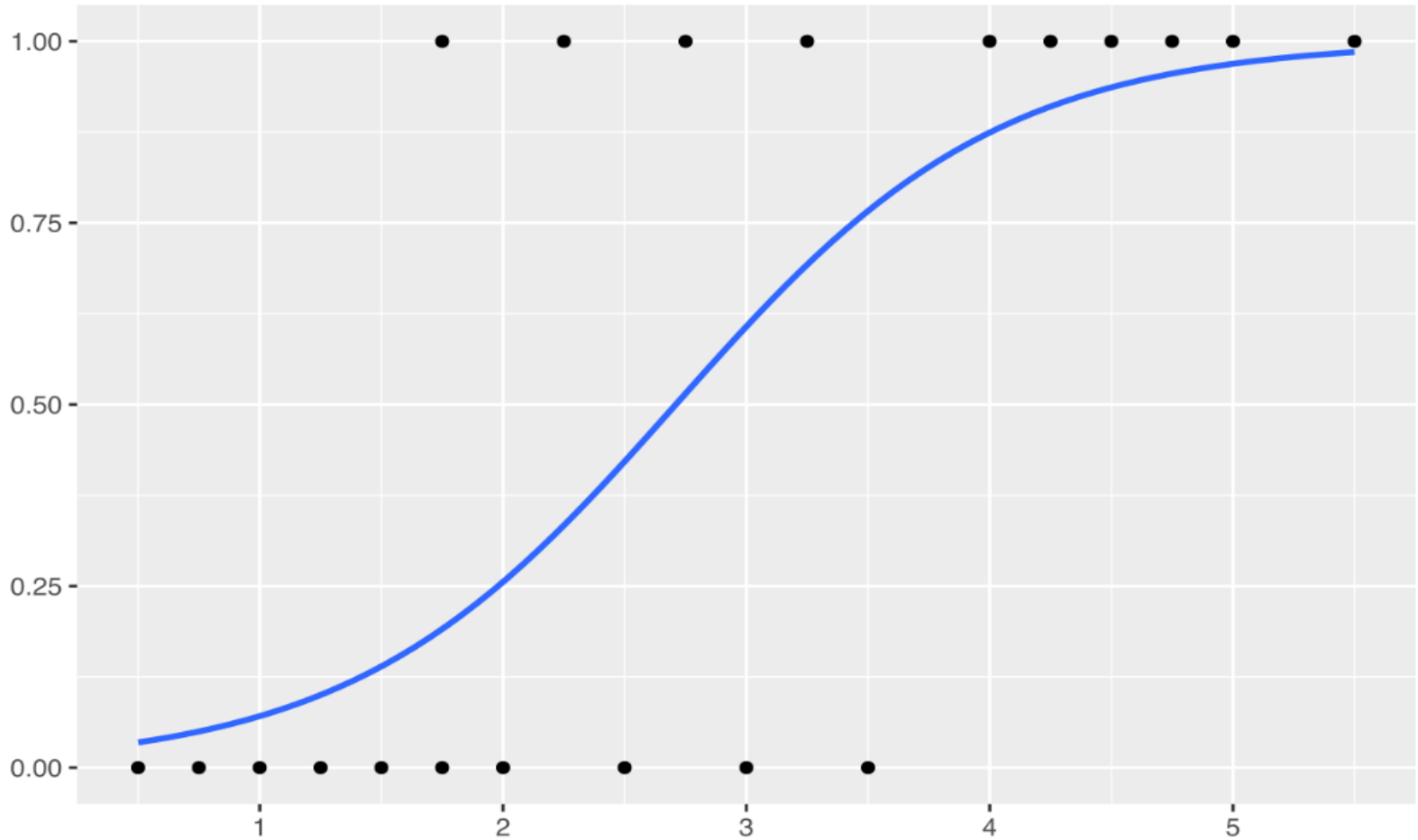
$$l(t) = \frac{1}{1 + e^{-t}} \tag{3}$$

Where,

$$t = \hat{\beta}_0 + \sum_{i=1}^{m} \hat{\beta}_i X_i$$

Logistic Function

- Graphically, the logistic function looks like this:



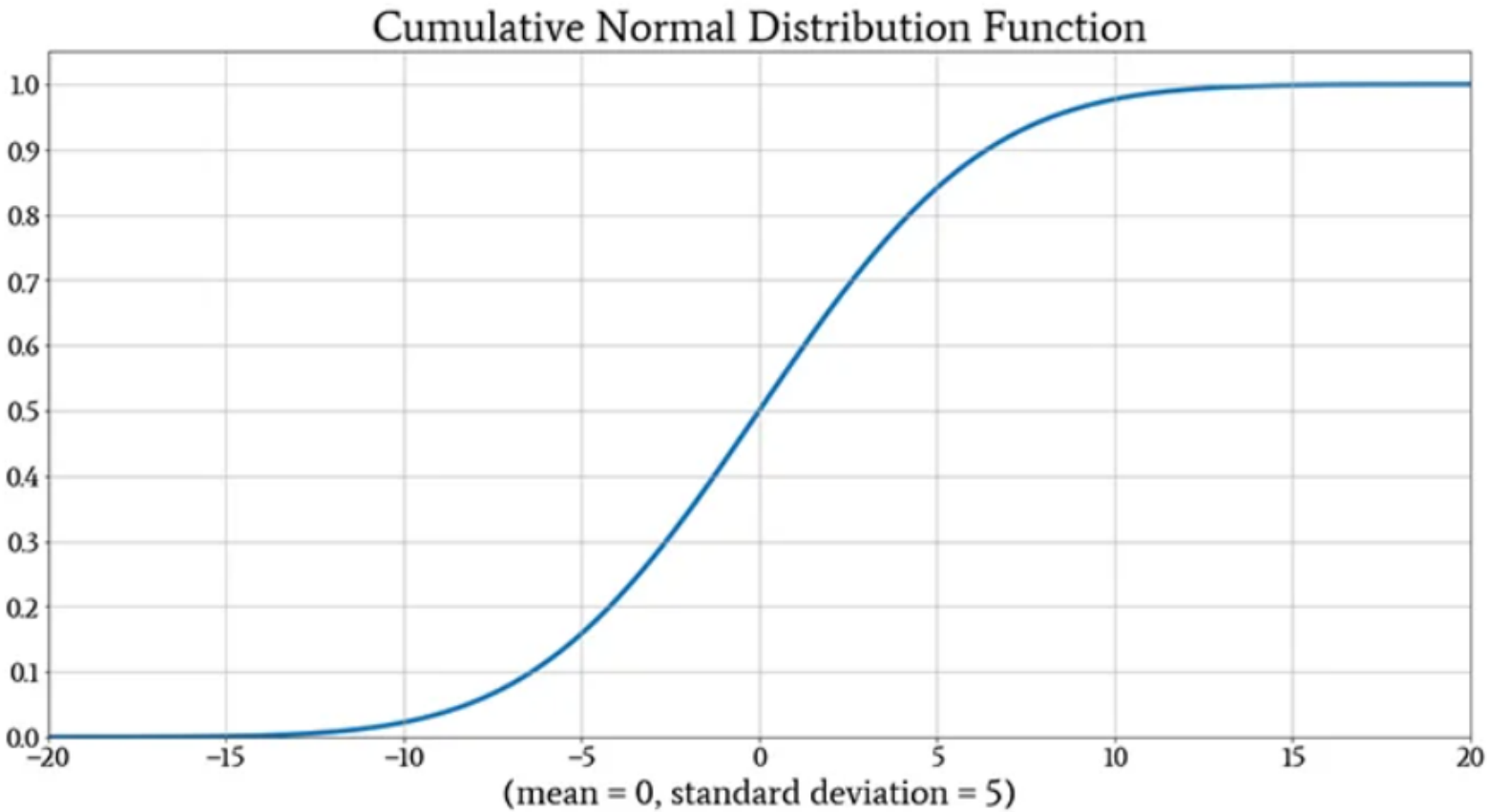
Probit

- **Probit** estimates the probability of belonging to a certain class (binary classification) by using a set of predictors (independent variables) X but uses the **cumulative normal distribution** to default probabilities between 0 and 1.
- The prediction then is given by is the next:

$$\hat{y} = \Phi(\hat{\beta}_0 + \sum_{i=1}^{m} \hat{\beta}_i X_i) \tag{4}$$

Normal CDF

- Graphically, the Normal CDF looks like this:



Logit Example Using the Iris Data Set

- We first load the necessary packages and functions into the kernel:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import
    LogisticRegression
from sklearn.model_selection import
    train_test_split
from sklearn.metrics import accuracy_score
```

- Notice we will just work with the package `sklearn`

Logit Example Using the Iris Data Set

- We can take a look at the structure of this data set:

```python
print("Examples:\n", X[:10])
print("Examples:\n", Y)
```

```
Examples:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]]
```

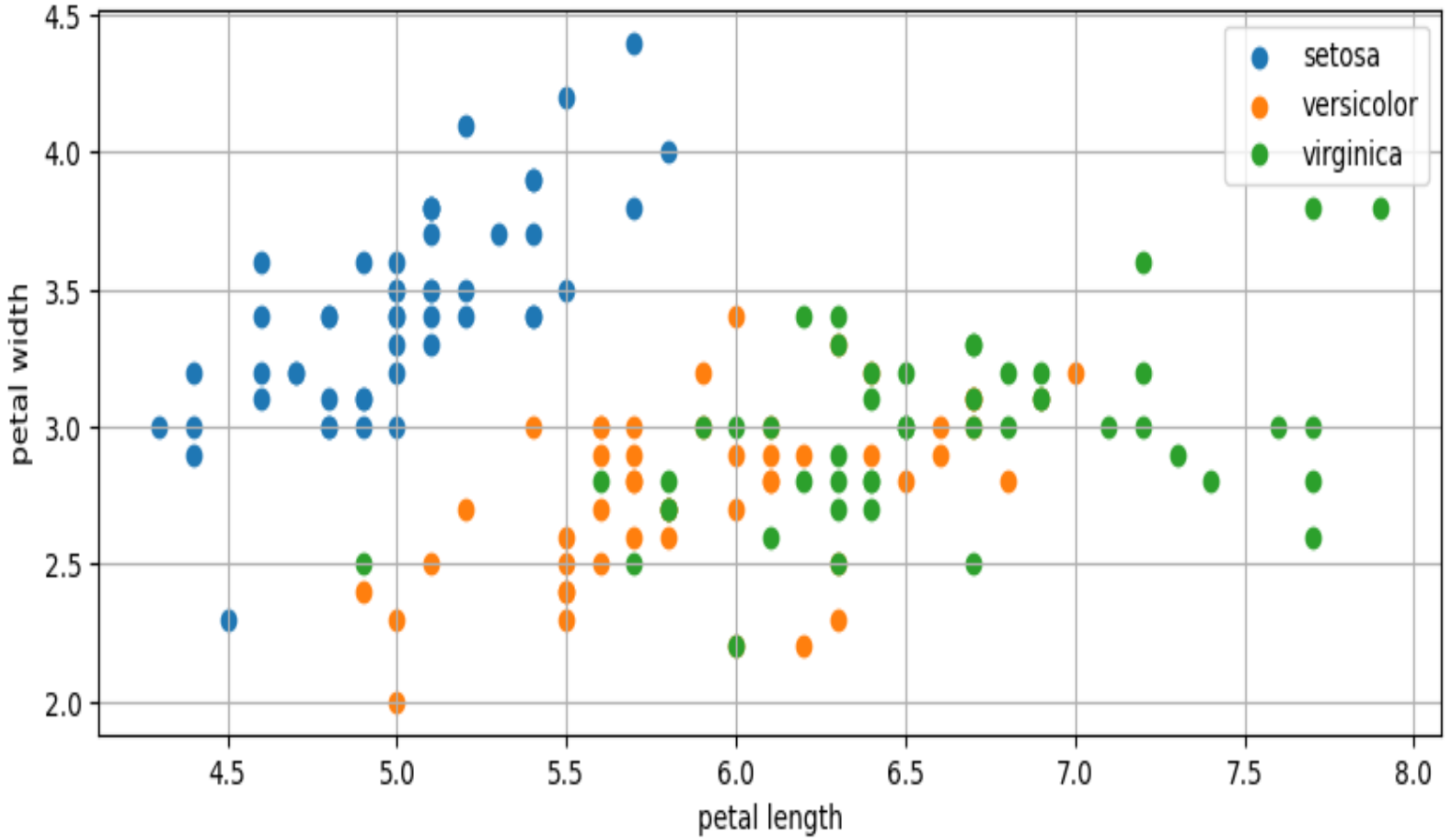
```
Examples:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Visualizing our Data

- We can better understand the problem at hand if we visualize and understand better our data:
- Let's graph two classes:

```
# Plot petal length and petal width
plt.figure(figsize=(10, 4))
plt.scatter(X[Y==0, 0], X[Y==0, 1], label=iris.target_names[0])
plt.scatter(X[Y==1, 0], X[Y==1, 1], label=iris.target_names[1])
plt.scatter(X[Y==2, 0], X[Y==2, 1], label=iris.target_names[2])
plt.legend()
plt.grid(True)
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.show()
```

Visualizing our Data



- Notice that the Setosa class can be perfectly distinguished from the other two classes using only the petal length and width.

Preparing our Data

- Logit is a binary classification model, meaning it only does prediction for two classes.
- Thus, for this example we will stay with only two classes and two features:

```
X = iris.data[iris.target != 2, :2]
y = iris.target[iris.target != 2]
```

- And we split into training and validation sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
        test_size = 0.2, random_state = 42)
```

Training the Model

- Now let's train our model. For this we can use the function `LogisticRegression()`. We first define the model and train it:

```
logit = LogisticRegression(max_iter=1000)
logit.fit(X_train, y_train)
```

- And we calculate the accuracy in both training and test sets:

```
# Predictions in training and test
y_pred_train = logit.predict(X_train)
y_pred_logit = logit.predict(X_test)
# Calculate accuracy
acc_train = accuracy_score(y_train, y_pred_train)
acc_logit = accuracy_score(y_test, y_pred_logit)
```

Prediction in Train and Test

- We get a 100% accuracy in both training and test sets. Meaning there is no over-fitting and our model can perfectly differentiate between classes.

```
print("Test Accuracy of the Logit model:", acc_logit)
#### Test Accuracy of the Logit Model 1.0

print("Train Accuracy of the Logit model:", acc_train)
#### Train Accuracy of the Logit Model 1.0
```

- Although we must admit that this result was kind of obvious given the graph that we saw at the beginning.

Regularizers

- To solve the problem of over-fitting in the Probit and Logit models, we will turn our attention to regularizers.
- **Regularization** are a set of strategies designed to reduce the test (validation) error, possibly at the expense of increased training error.
- We wish to go from over-fitting to the best fit.
- L1 and L2 regularization are the most common ones.
- **L1** adds a penalty term to the ordinary least squares (OLS) objective function.
- Coefficients become more sparsed
- **L2** also adds a penalty term to the loss function that discourages large weights in the model.

L1 (LASSO) and L2 (Weight Decay)

- If J is the objetive function, the difference is in the specification the difference is in Ω

$$\hat{J}(\Theta; X, y) = J(\Theta; X, y) + \alpha\Omega(\Theta)$$

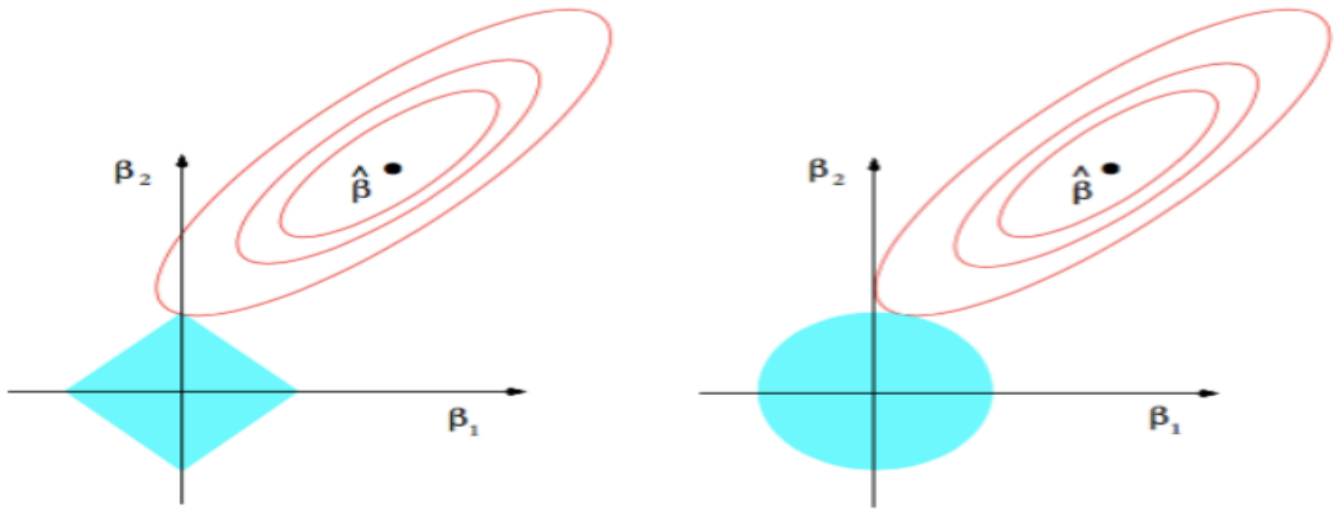


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

L1 and L2 in Python

- To add the regularizers to train a Logit model one just has to do the following:

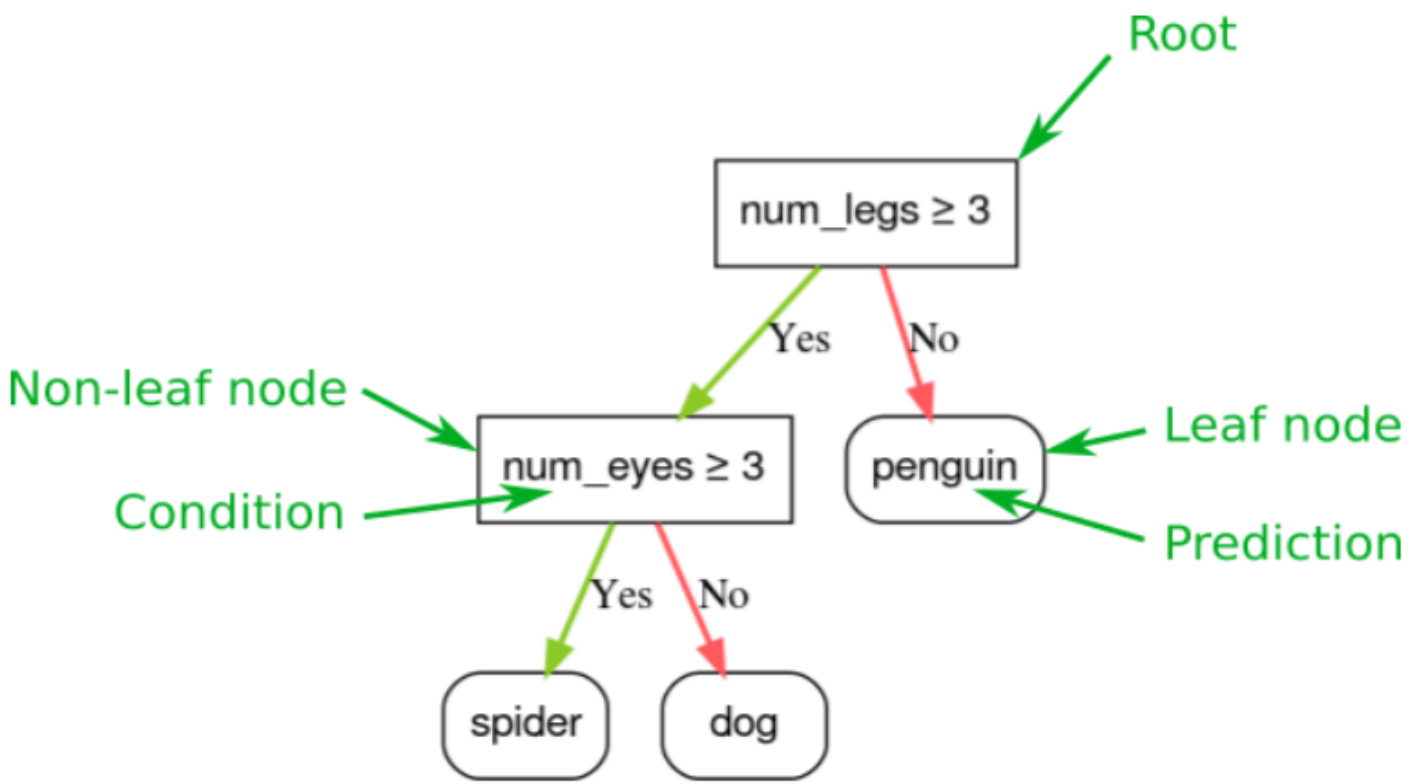
```
# For L1
logit = LogisticRegression(penalty =
        'l1', max_iter=1000)

# For L2
logit = LogisticRegression(penalty =
        'l2', max_iter=1000)
```

Random Forest

- **Random forest** is an ensemble learning algorithm that combines multiple decision trees to improve the accuracy and stability of predictions.
- Each created decision tree comes from a **random** subset of the training data and the features
- The prediction is a combination (often the mean) of the results of these decision trees.
- These decision trees and the 'answers' to the questions in them are built with a data driven approach.

Decision Trees

- **Decision trees** are models composed of a collection of questions organized hierarchically in the shape of a tree.
- The questions are usually called a condition, a split, or a test.



Comparison with Probit and Logit

- **Main advantage:** more complex model, performs better in big data sets with lots of features.
- Also, you still need to choose your features (your X), but not how you parametrize them.
- **Main disadvantage:** we must choose an initial set of hyper-parameters and tune them to improve accuracy and prevent over-fitting.

Hyper Parameters

- Random forest has 4 main *hyper-parameters* that we choose:

1. **Number of estimators** (number of trees in the forest)
2. **The depth of the trees**
3. **The minimum number of samples required to split an internal node**
4. **The minimum number of samples required to be at a leaf node.**

Example Using the Iris Data Set

- We will go over DataCamp's example together. We load the necessary packages into the kernel.
- Because of its simplicity, we will again use the Iris Data set.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import datasets
iris = datasets.load_iris()
```

Preparing the Data Set

- We will now work with all the classes and all the features.

```
data=pd.DataFrame({'sepal length':iris.data[:,0],
                   'sepal width':iris.data[:,1],
                   'petal length':iris.data[:,2],
                   'petal width':iris.data[:,3],
                   'species':iris.target})
data.head()
```

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Preparing the Data Set

- We generate our X and our y and split it in training and test sets (20% test, as per usual):

```
X = data[['sepal length', 'sepal width',
          'petal length', 'petal width']] # Features
y = data['species'] # Labels

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                  test_size=0.2, random_state = 41)
```

Preparing the Model

- Let us start defaulting all hyper-parameters except the number of estimators (we will see how changing them can improve our test accuracy)

```
clf = RandomForestClassifier(n_estimators = 100)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
```

Over-fitting

- Checking the accuracy in test (90%) and in training (100%) we notice there is a bit of over-fitting.
- Although 90% is not bad, we can improve it by hyper-parameter tuning.

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
#### Accuracy .9
print("Accuracy:",metrics.accuracy_score(y_train,
          clf.predict(X_train)))
#### Accuracy 1.0
```

- First, let us check how we can see the importance of each feature to the prediction:

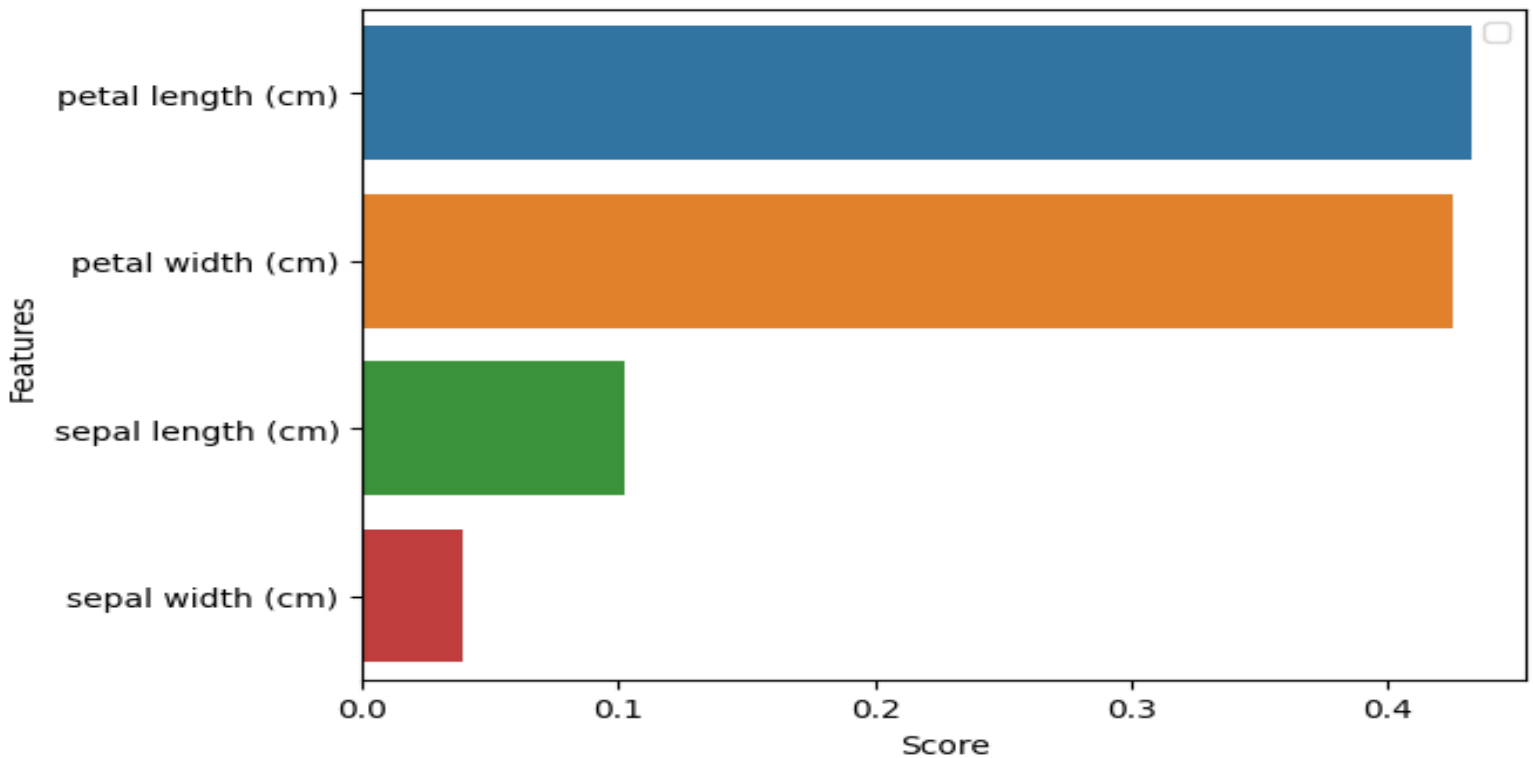
Feature Importance

- One of the most useful things about Random Forest is that, even though it could be an amazingly complex model, with tons of features, we can understand how each one of them is affecting the prediction.
- For this we estimate each feature contribution to the prediction:

```
feature_imp = pd.Series(clf.feature_importances_,
        index = iris.feature_names).sort_values(ascending=False)
feature_imp
### Output
petal length (cm)    0.432701
petal width (cm)     0.425739
sepal length (cm)    0.102160
sepal width (cm)     0.039400
dtype: float64
```

Feature Importance

```
sns.barplot(x=feature_imp, y=feature_imp.index)
plt.xlabel('Score')
plt.ylabel('Features')
plt.title("Feature Importance")
plt.legend()
plt.show()
```



Hyper-Parameter Tuning

- To solve the over-fitting, we can automatically choose between a set of hyper-parameters to see which one gives the model with the best test accuracy.
- To do this we will use `sklearn` funtion `GridSearchCV`.
- We first define the set of parameters:

```
param_grid = {
    "n_estimators": [10, 50, 100],
    "max_depth": [3, 5, 7],
    "min_samples_split": [2, 3, 4],
    "min_samples_leaf": [1, 2, 3],
}
```

Hyper-Parameter Tuning

- Then we go to each possible combination and obtain the best set of parameters:

```
grid_search = GridSearchCV(clf, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Print the best parameters
print("Best parameters:", grid_search.best_params_)

# Output
#### Best parameters: {'max_depth': 5, 'min_samples_leaf': 1,
#### 'min_samples_split': 3, 'n_estimators': 10}
```

- And we obtain a much better accuracy (93.3%)

```
y_pred = grid_search.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print("Accuracy after Grid Search:", acc)
#### Accuracy after Grid Search: .933333
```

QUESTIONS: