Northeastern University

Logistic Regression

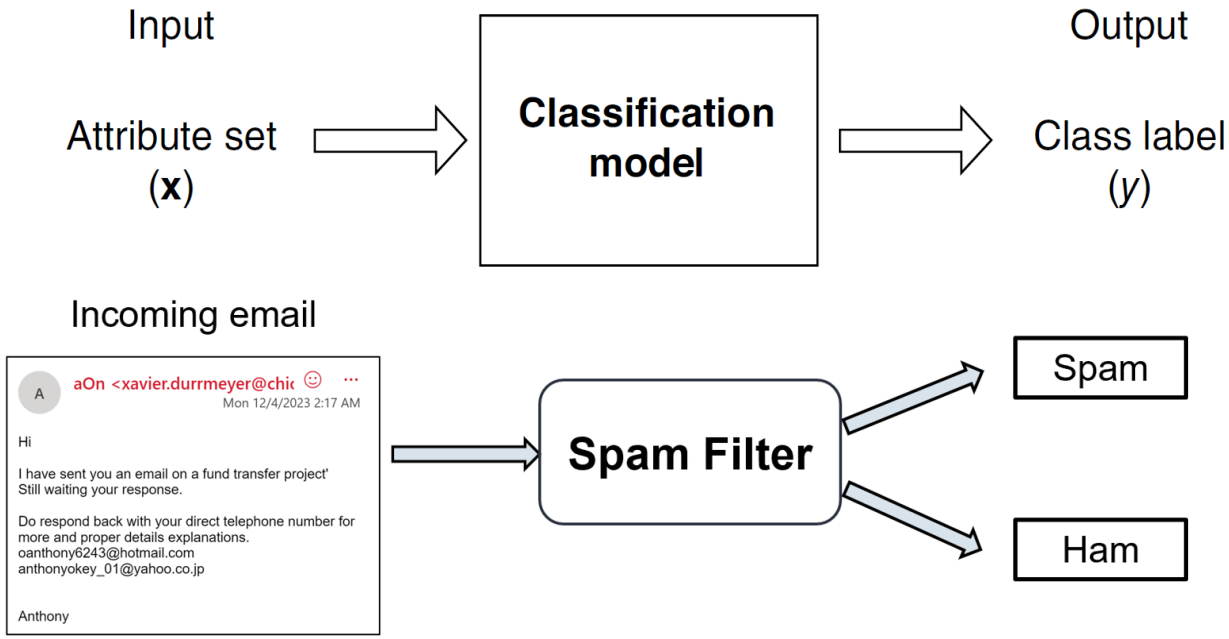
Roi Yehoshua

Agenda

- Classification problems
- Logistic regression
- Decision boundaries
- Cross-entropy loss
- Classification evaluation metrics
- Imbalanced classes
- Multiclass classification problems
- Multinomial logistic regression

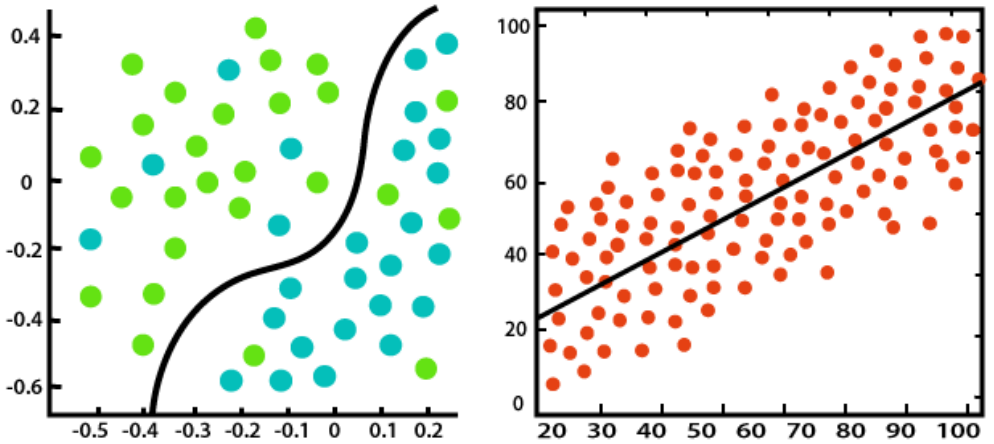
Classification

- Classification is the task of predicting the **class** of a given input sample
- **Given**: A training set of *n* labeled examples: $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
  - Each label y_i belongs to one of *K* predefined classes $C_1, C_2, \ldots, C_K$
- **Goal**: Learn a function $h(\mathbf{x})$ that maps a feature vector $\mathbf{x}$ to a class label C_i



Classification vs. Regression

- In regression, our goal was to find a function (e.g., a line) that best fits the data
- In classification, the goal is to find a function that separates data points from different classes
- The separating function defines a **decision boundary** that partitions the input space into regions corresponding to different predicted classes



Classification Regression

Types of Classification Problems

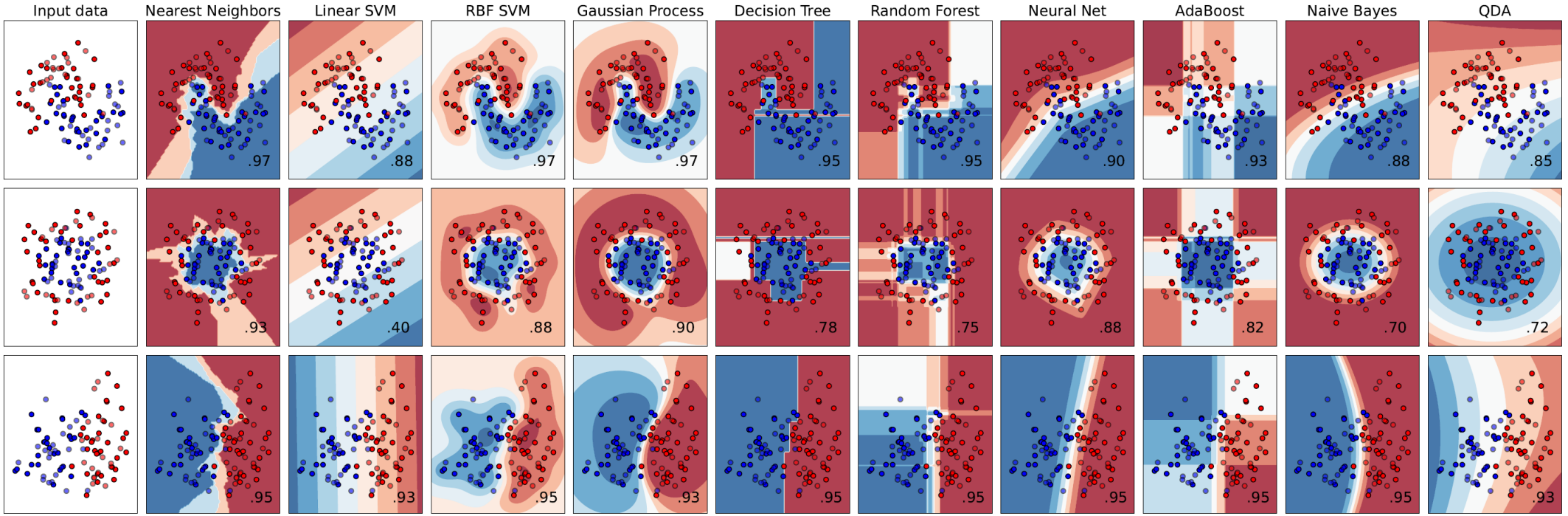
- **Binary classification**: Only two possible classes, typically labeled as **1** or **0**
  - 1 is the **positive class**, and 0 is the **negative class**
  - The positive class is usually the less frequent or more important class
    - e.g., the “spam” in email filtering, or “disease” in medical diagnosis
- **Multi-class classification**: More than two possible classes
  - e.g., in digit recognition, each image is classified into one of 10 classes {0, 1, …, 9}
- **Multi-label classification**: Each input can belong to multiple classes simultaneously
  - e.g., a news article may be tagged as both *Science* and *Technology*
- **Multi-output classification**: Each input is associated with multiple outputs
  - Each output corresponds to a different classification task (but outputs may be correlated)
  - e.g., an object an image can be classified both by its shape and color

Types of Classifiers

- Deterministic vs. probabilistic classifiers
  - A **deterministic classifier** outputs only the predicted class label
    - Examples: decision tree, perceptron
  - A **probabilistic classifier** outputs the probability distribution over classes
    - Examples: logistic regression, naïve Bayes
- Linear vs. nonlinear classifiers
  - A **linear classifier** learns a linear decision boundary to separate the classes
    - Examples: logistic regression, support vector machine, perceptron
  - A **nonlinear classifier** can learn complex, curved decision boundaries
    - Examples: decision trees, k-nearest neighbors, neural networks

Classification Algorithms

- Each classification algorithm has its own strengths and weaknesses
- Performance depends on the characteristics of the dataset
  - Size, number of features, linearity, noise, class imbalance



http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

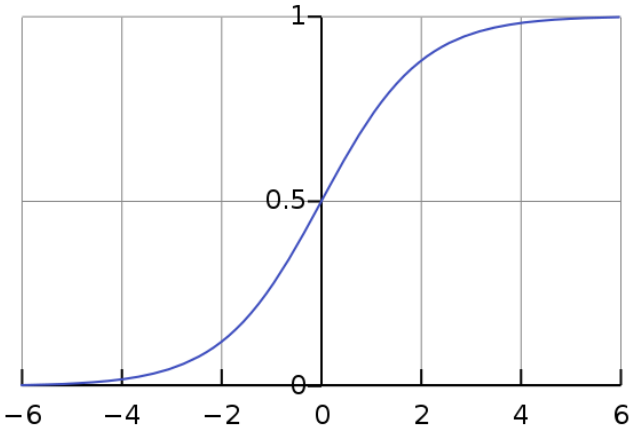
Logistic Regression

- One of the simplest and most foundational classification algorithms
- A binary, linear classifier that outputs class probabilities
- It first computes a linear combination of the input features:

$$z = w_0 + w_1 x_1 + \cdots + w_d x_d = \mathbf{w}^T \mathbf{x}$$

  - This value *z* is often called the **score** or **logit**
- The score is then passed through the **sigmoid function** that maps it into [0, 1]:

$$p = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}$$



  - *p* is the estimated **probability** that the input belongs to the positive class

$$p = P(y = 1|\mathbf{x})$$

Why the Sigmoid Function?

- The **odds** of an event is the ratio between the probability of success and failure:

$$\text{odds} = \frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = \frac{p}{1-p}$$

- The **log-odds** (or logit) is the logarithm of the odds:

$$\text{log-odds} = \log\left(\frac{p}{1-p}\right)$$

- In logistic regression, we assume the **log-odds are a linear function** of the input:

$$\log\left(\frac{p}{1-p}\right) = \mathbf{w}^T\mathbf{x}$$

- Solving for *p* gives:

$$p = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

$$\frac{p}{1-p} = e^{\mathbf{w}^T\mathbf{x}}$$

$$p = (1-p)e^{\mathbf{w}^T\mathbf{x}}$$

$$p(1+e^{\mathbf{w}^T\mathbf{x}}) = e^{\mathbf{w}^T\mathbf{x}}$$

$$p = \frac{e^{\mathbf{w}^T\mathbf{x}}}{(1+e^{\mathbf{w}^T\mathbf{x}})} = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}} = \sigma(\mathbf{w}^T\mathbf{x})$$

Example

- Predict the likelihood of choosing to bike to work based on the temperature

Temperature (°C)	Bikes to work?
10	0
15	0
20	1
25	1
30	1

- The logistic regression model is:

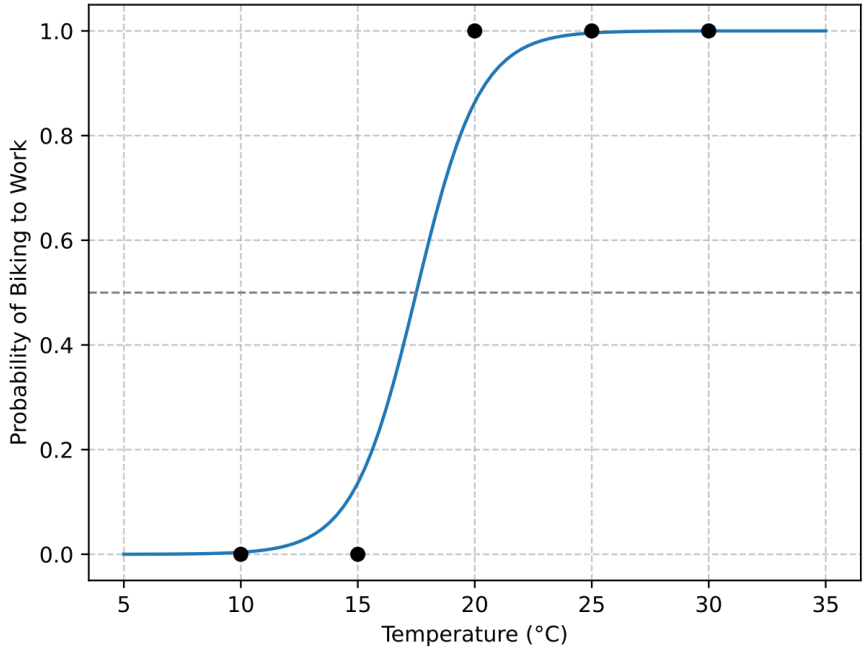
$$p = P(y = 1|x) = \frac{1}{1 + e^{-(w_0 + w_1 x)}}$$

- Assume that after optimization we find the coefficients w_0 = -12.94, w_1 = 0.74

$$p = \frac{1}{1 + e^{-(-12.94 + 0.74x)}}$$

Example

- The curve of the logistic regression model



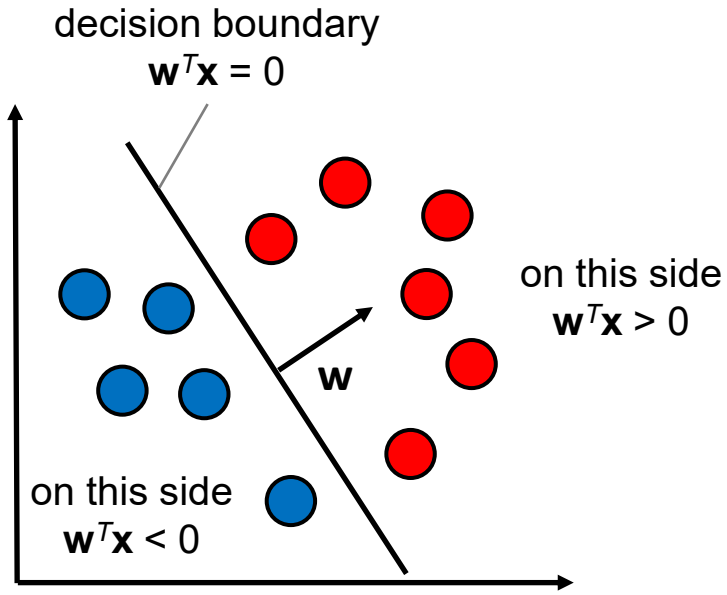
$$p = \frac{1}{1 + e^{-(-12.94+0.74x)}}$$

- For example, the probability of biking to work at 20°C is:

$$P(y = 1|x = 20) = \frac{1}{1 + e^{-(-12.94+0.74 \cdot 20)}} = 0.864$$

Decision Boundaries

- The decision boundary is defined by the points for which $p = 0.5$ (or $\mathbf{w}^T\mathbf{x} = 0$)
  - This equation defines a hyperplane of dimension (d - 1) that is orthogonal to $\mathbf{w}$



- The linearity of this boundary makes logistic regression a **linear classifier**

Training a Logistic Regression Model

- Define a suitable loss function
  - Quantifies the difference between the predicted probability *p* and the true label *y*
- Use an optimization algorithm to minimize this function on the training set
- Return the optimal coefficients **w***

Zero-One Loss

- A natural loss function for binary classification is the **0-1 loss**:

$$L_{0\text{–}1}(y, \hat{y}) = \begin{cases} 1 & \text{if } \hat{y} \neq y, \\ 0 & \text{otherwise} \end{cases}$$

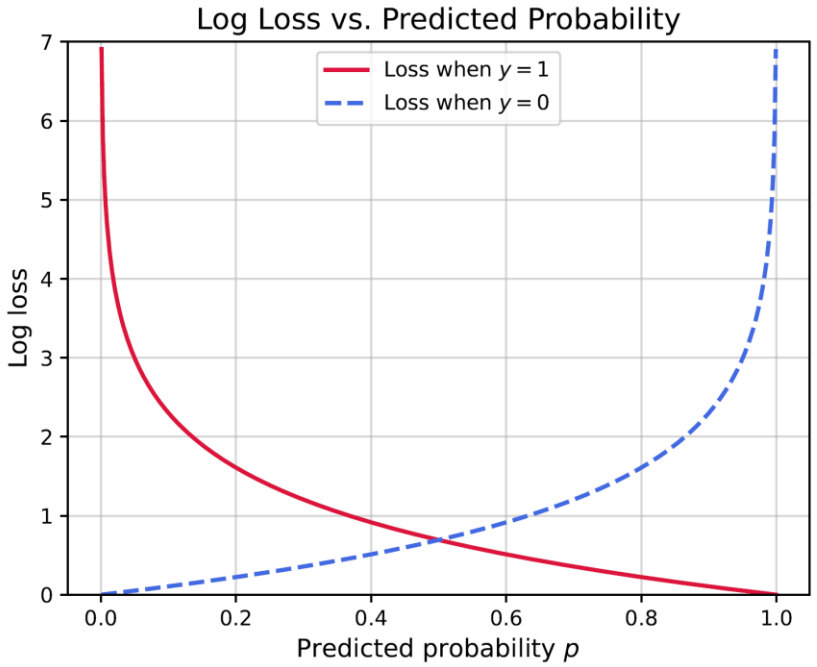
  - $y \in \{0, 1\}$ is the true label and $\hat{y} = h(\mathbf{x}) \in \{0, 1\}$ is the predicted label
- Problems with this loss function:
  - Non-differentiable
  - Ignores model's confidence: all incorrect predictions are treated equally

Log Loss

- Instead, we use a probabilistic loss function called **log loss** (or **binary cross-entropy**):

$$L_{\log}(y, p) = -y \log p - (1 - y) \log(1 - p)$$

  - $y \in \{0, 1\}$ is the true label and p is the predicted probability of the positive class
- Penalizes overconfident wrong predictions more heavily



Log Loss from MLE

- Assume the label *y* follows a **Bernoulli distribution** with parameter *p*

$$p = P(y = 1|\mathbf{x})$$

- The likelihood for a single observation is:

$$\mathcal{L}(p|y) = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases} = p^y(1-p)^{1-y}$$

- The log-likelihood is:

$$\ell(p|y) = \log \mathcal{L}(p|y) = y \log p + (1-y) \log(1-p)$$

- The negative log-likelihood (NLL) is:

$$-\ell(p|y) = -y \log p - (1-y) \log(1-p)$$

  - This is exactly the log loss

- MLE naturally leads to minimizing log loss when fitting logistic regression

Cost Function

- The cost function calculates the average log loss over the training set:

$$J(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

  - where $p_i = \sigma(\mathbf{w}^T \mathbf{x}_i)$
- In vectorized form:

$$J(\mathbf{w}) = -\frac{1}{n}\left(\mathbf{y}^T \log \mathbf{p} + (\mathbf{1}-\mathbf{y}^T)\log(\mathbf{1}-\mathbf{p})\right)$$

  - **y** = (y_1, …, y_n) is the vector of true labels
  - **p** = (p_1, …, p_n) is the vector of predicted probabilities, $\mathbf{p} = \sigma(X\mathbf{w})$
  - **1** = (1, …, 1) is a vector of ones of size *n*
- The function is convex, so every local minimum is a global minimum
- However, there is no closed-form solution for the optimal **w***
- Therefore, we need to rely on iterative optimization techniques

Gradient Descent

- We compute the partial derivative of $J(\mathbf{w})$ with respect to each weight w_j:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \frac{\partial}{\partial w_j} \left[-\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] \right]$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \frac{\partial}{\partial w_j} \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) + (1 - y_i) \frac{\partial}{\partial w_j} \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right]$$

Moving the derivative into the summation

$$= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(\frac{y_i}{\sigma(\mathbf{w}^T \mathbf{x}_i)} - \frac{1 - y_i}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \right) \frac{\partial}{\partial w_j} \sigma(\mathbf{w}^T \mathbf{x}_i) \right]$$

Using the chain rule

$$= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(\frac{y_i}{\sigma(\mathbf{w}^T \mathbf{x}_i)} - \frac{1 - y_i}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \right) \sigma(\mathbf{w}^T \mathbf{x}_i) \left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right]$$

The derivative of sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

$$= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(y_i \left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) - (1 - y_i) \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right]$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(y_i - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right] = \frac{1}{n} \sum_{i=1}^{n} \left[(p_i - y_i) x_{ij} \right]$$

- Thus , the gradient vector is: $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{n} X^T (\mathbf{p} - \mathbf{y})$

X is the feature matrix

Gradient Descent

- The update rule for (batch) gradient descent is:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbf{w} - \frac{\alpha}{n} X^T (\mathbf{p} - \mathbf{y})$$

  - α is the learning rate
- In stochastic gradient descent, we update after every training example (**x**, y):

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha (p - y)\mathbf{x}$$

Newton's Method

- Newton's method can be more efficient than gradient descent
  - Adjusts both the search direction and step size using second-order curvature (Hessian)
  - Achieves quadratic convergence near the minimum
- For convex objectives (as in logistic regression), the Hessian is positive semidefinite
  - Thus, Newton's method behaves well
- Newton's update rule: $\mathbf{w} \leftarrow \mathbf{w} - H^{-1}(\mathbf{w})\nabla_{\mathbf{w}} J(\mathbf{w})$
  - H^{-1} is the inverse of the Hessian matrix
- Issue: Computing and inverting the Hessian is expensive, especially in large datasets
- Solution: Use **quasi-Newton methods**
  - Avoid computing the full Hessian, approximates it efficiently from gradients
  - A popular method: **BFGS** (Broyden-Fletcher-Goldfarb-Shanno algorithm)

Regularized Logistic Regression

- Similar to linear regression, we can add a regularization term to the cost function
  - Especially useful in high-dimensional settings where many features may be irrelevant
- **L2 regularization (ridge)** adds a penalty on the squared norm of the weights:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \|\mathbf{w}\|_2^2$$

- **L1 regularized (lasso)** adds a penalty on the absolute values of the weights:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \|\mathbf{w}\|_1$$

- **Elastic net** combines both L1 and L2 penalties:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + r\lambda \|\mathbf{w}\|_1 + (1 - r)\lambda \|\mathbf{w}\|_2^2$$

Logistic Regression in Scikit-Learn

- The LogisticRegression class provides a highly optimized implementation
- Supports several efficient solvers for optimization:
  - **L-BFGS** (Limited-memory BFGS):
    - Default solver for small-to-medium datasets
    - Efficient for L2-regularized problems
  - **Newton-CG** (Newton's method with Conjugate Gradient):
    - More accurate on some problems, but more expensive per iteration
  - **SAG** (Stochastic Average Gradient):
    - Suitable for large-scale datasets
    - Incrementally updates the weights using mini-batches
- Solver choice affects speed, scalability, and regularization support

Logistic Regression in Scikit-Learn

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False,
tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100, multi_class='deprecated', verbose=0,
warm_start=False, n_jobs=None, l1_ratio=None)                    [source]
```

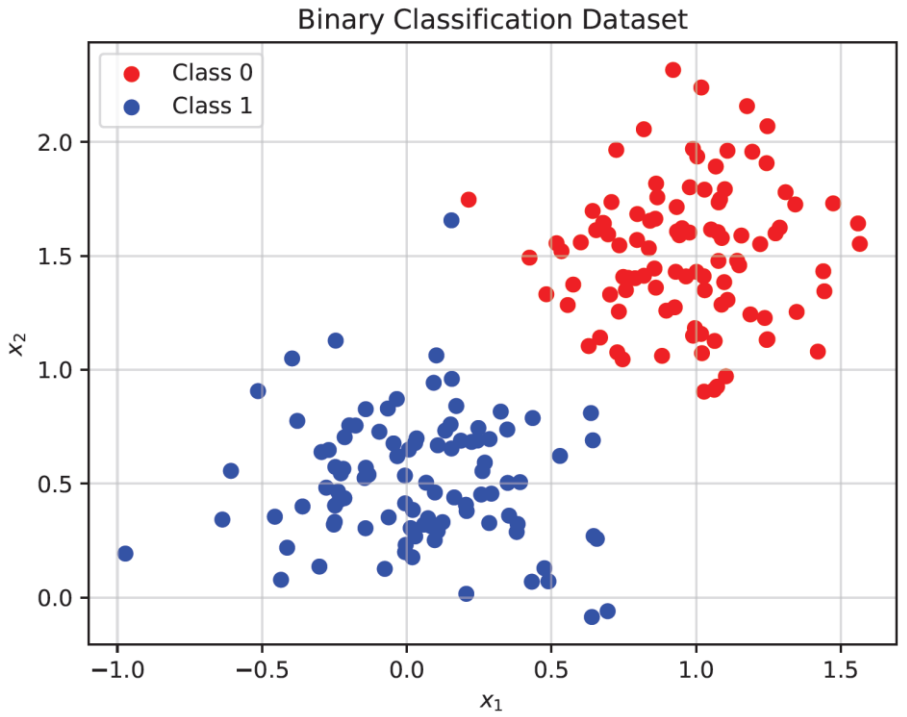
Parameter	Description
penalty	Type of regularization to apply. Can be 'l1', 'l2', 'elasticnet', or None.
tol	Tolerance for the stopping criteria
C	Inverse of the regularization strength ($1/\lambda$)
solver	Optimization algorithm to use. Options include: {'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'.
max_iter	Maximum number of iterations taken for the solver to converge
class_weight	Weights associated with classes, given as a dictionary {class:weight}, useful for handling imbalanced datasets

Logistic Regression in Scikit-Learn

- Let's demonstrate logistic regression on a synthetically generated data set
- We use **make_blobs()** to create two normally-distributed blobs with 100 points each

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=200, centers=((1, 1.5), (0, 0.5)),
                  cluster_std=(0.3, 0.3), random_state=42)
```

```
def plot_data(X, y):
    plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='red', label='Class 0')
    plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='blue', label='Class 1')
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    plt.title('Binary Classification Dataset')
    plt.legend()
    plt.grid(alpha=0.5)

plot_data(X, y)
```



Model Training

- Split the dataset into training and test:

```
from sklearn.model_selection import train_test_split

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

- Train a LogisticRegression model on the training set:

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(random_state=42)
clf.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression(random_state=42)
```

- Remember to scale the features before training if they have different ranges!

Model Training

- As in linear regression, we can inspect the learned coefficients:

```
print('Intercept:', clf.intercept_)
print('Coefficients:', clf.coef_)
```

```
Intercept: [5.32908555]
Coefficients: [[-3.55797637 -3.48339265]]
```

- We can also check how many iterations were needed for the solver to converge:

```
print('Number of iterations:', clf.n_iter_)
```

```
Number of iterations: [10]
```

Model Evaluation

- The **score()** method returns the **accuracy** of the classifier on the given data

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

- Evaluation on the training and test sets:

```
train_accuracy = clf.score(X_train, y_train)
print(f'Train accuracy: {train_accuracy:.4f}')

test_accuracy = clf.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

```
Train accuracy: 1.0000
Test accuracy: 0.9800
```

Decision Boundary

- For 2D problems, it is often useful to visualize the boundary line between the classes
- The boundary is defined as the set of points where the predicted probability is 0.5

$$\sigma(\mathbf{w}^T\mathbf{x}) = 0.5 \;\Rightarrow\; \mathbf{w}^T\mathbf{x} = 0$$

- In the case of a 2D dataset, this becomes an equation of a line:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

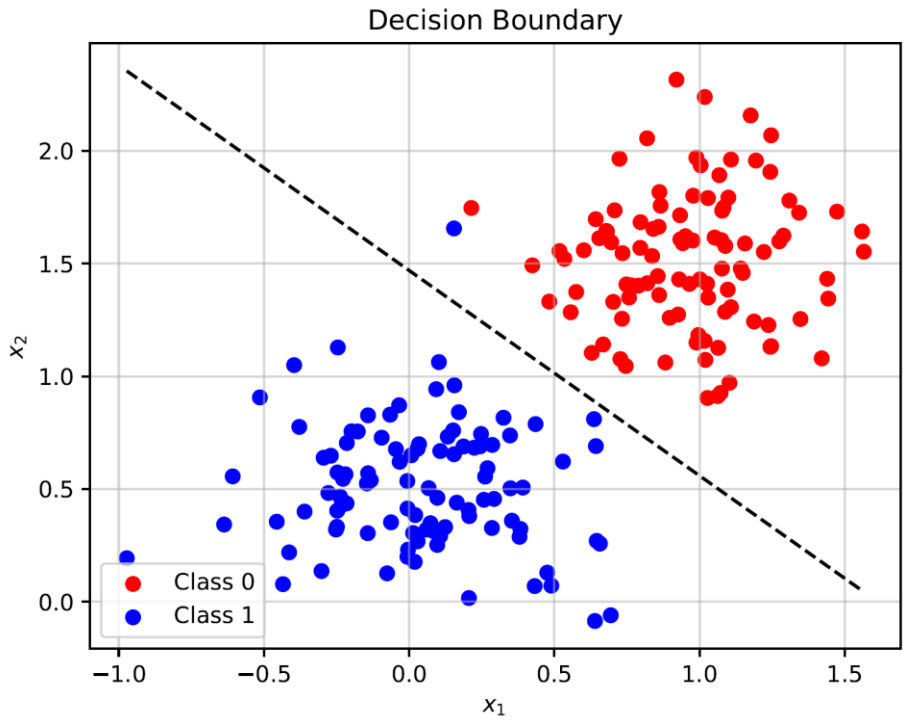
  - The slope of the line is $-w_1/w_2$ and the intercept is $-w_0/w_2$

Decision Boundary

- Plotting the decision boundary:

```python
def plot_decision_boundary(X, y, clf):
    """Plot the decision boundary between the classes."""
    # Calculate the intercept and slope of the separating line
    w0 = clf.intercept_
    w1, w2 = clf.coef_[0]
    b = -w0 / w2
    m = -w1 / w2

    # Create a range of values for x
    x_min, x_max = X[:, 0].min(), X[:, 0].max()
    x_line = np.array([x_min, x_max])

    # Compute the corresponding y values
    y_line = m * x_line + b

    # Plot the decision boundary
    plt.plot(x_line, y_line, c='k', ls='--')
    plt.title('Decision Boundary')
```

```python
plot_data(X, y)
plot_decision_boundary(X, y, clf)
```



Making Predictions

- We can now use the trained model to make predictions on new data
- LogisticRegression provides two methods for predictions:
  - **predict_proba**(X): Provides predicted probabilities for the samples in X
  - **predict**(X): Predicts class labels for the samples in X

```
X_new = np.array([[0, 0], [0.5, 1], [1, 1.5]])
clf.predict_proba(X_new)
```

```
array([[0.00645816, 0.99354184],
       [0.48795293, 0.51204707],
       [0.96179467, 0.03820533]])
```

```
clf.predict(X_new)
```

```
array([1, 1, 0])
```

The Class Imbalance Problem

- **Imbalanced datasets** are datasets where the classes are not equally represented
- In binary classification, this typically means:
  - A large number of negative (“normal”) examples
  - A small number of positive (“interesting”) examples
- Common in tasks such as medical diagnosis and fraud detection
  - where < 1% of the examples belong to the minority class (e.g., fraudulent transactions)
- Main challenges with imbalanced datasets:
  - **Unreliable evaluation metrics**
    - e.g., a model that always predicts ''legitimate'' achieves > 99% accuracy
  - **Algorithms tend to favor the majority class**
    - Can lead to very poor results on the minority class (e.g., 0-10% accuracy)
    - While correct classification of the minority class is often more important than the majority

Example: Bank Marketing

- A dataset from the UCI Machine Learning Repository
- Deals with marketing campaigns (phone calls) of a Portuguese banking institution
- The goal is to predict whether the client will subscribe (1/0) to a term deposit
- The data set includes 41,188 records and 21 features

Attribute Information:

Input variables:
bank client data:
1 - age (numeric)
2 - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown')
3 - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
4 - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
5 - default: has credit in default? (categorical: 'no','yes','unknown')
6 - housing: has housing loan? (categorical: 'no','yes','unknown')
7 - loan: has personal loan? (categorical: 'no','yes','unknown')
related with the last contact of the current campaign:
8 - contact: contact communication type (categorical: 'cellular','telephone')
9 - month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
10 - day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
11 - duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
other attributes:
12 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
13 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
14 - previous: number of contacts performed before this campaign and for this client (numeric)
15 - poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')
social and economic context attributes
16 - emp.var.rate: employment variation rate - quarterly indicator (numeric)
17 - cons.price.idx: consumer price index - monthly indicator (numeric)
18 - cons.conf.idx: consumer confidence index - monthly indicator (numeric)
19 - euribor3m: euribor 3 month rate - daily indicator (numeric)
20 - nr.employed: number of employees - quarterly indicator (numeric)

Output variable (desired target):
21 - y - has the client subscribed a term deposit? (binary: 'yes','no')

Data Exploration

- Let’s first load the data set and explore its basic properties

```
df = pd.read_csv('banking.csv')
df.head()
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp_va
0	44	blue-collar	married	basic.4y	unknown	yes	no	cellular	aug	thu	...	1	999	0	nonexistent	
1	53	technician	married	unknown	no	no	no	cellular	nov	fri	...	1	999	0	nonexistent	
2	28	management	single	university.degree	no	yes	no	cellular	jun	thu	...	3	6	2	success	
3	39	services	married	high.school	no	no	no	cellular	apr	fri	...	2	999	0	nonexistent	
4	55	retired	married	basic.4y	no	yes	no	cellular	aug	fri	...	1	3	1	success	

5 rows × 21 columns

Data Exploration

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
age               41188 non-null int64
job               41188 non-null object
marital           41188 non-null object
education         41188 non-null object
default           41188 non-null object
housing           41188 non-null object
loan              41188 non-null object
contact           41188 non-null object
month             41188 non-null object
day_of_week       41188 non-null object
duration          41188 non-null int64
campaign          41188 non-null int64
pdays             41188 non-null int64
previous          41188 non-null int64
poutcome          41188 non-null object
emp_var_rate      41188 non-null float64
cons_price_idx    41188 non-null float64
cons_conf_idx     41188 non-null float64
euribor3m         41188 non-null float64
nr_employed       41188 non-null float64
y                 41188 non-null int64
dtypes: float64(5), int64(6), object(10)
memory usage: 6.6+ MB
```

Data Exploration

- The number of samples that belong to each class:

```
df['y'].value_counts()
```

```
0    36548
1     4640
Name: y, dtype: int64
```

  - The ratio of negative to positive examples is about 89:11

- It is also useful to explore the feature statistics in each class:

```
df.groupby('y').mean()
```

y	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed
0	39.911185	220.844807	2.633085	984.113878	0.132374	0.248875	93.603757	-40.593097	3.811491	5176.166600
1	40.913147	553.191164	2.051724	792.035560	0.492672	-1.233448	93.354386	-39.789784	2.123135	5095.115991

Training and Test Split

- Let’s split the data set into 80% training and 20% test:

```
X = df.drop('y', axis=1)
y = df['y']
```

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

```
X_train.shape
```

```
(32950, 20)
```

```
X_test.shape
```

```
(8238, 20)
```

Data Preparation

- We normalize the numerical features, and one-hot encode the categorical features

```
from sklearn.compose import ColumnTransformer

categorical_features = list(X.dtypes[X.dtypes == 'object'].index)
numerical_features = list(X.dtypes[X.dtypes != 'object'].index)

transformer = ColumnTransformer([
    ('scaler', StandardScaler(), numerical_features),
    ('onehot', OneHotEncoder(), categorical_features)
])
```

- Then we define a pipeline that combines the transformer with logistic regression:

```
clf = Pipeline([
    ('trans', transformer),
    ('log_reg', LogisticRegression())
])
```

Performance Measures

- Training the classifier on the data set provides the following results:

```
clf.fit(X_train, y_train)

print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9132
Test set accuracy: 0.9088
```

- Above 91% accuracy (ratio of correct predictions)
- Looks good, no?

Performance Measures

- Let's create a dumb classifier that just classifies every sample as 0 (no subscription)

```
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score

class DumbClassifier(BaseEstimator):
    def fit(self, X, y):
        pass
    def predict(self, X):
        return np.zeros((X.shape[0], 1))
    def score(self, X, y):
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)
```

```
d_clf = DumbClassifier()
d_clf.fit(X_train, y_train)
print('Training set accuracy:', np.round(d_clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(d_clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.8878
Test set accuracy: 0.8855
```

- This classifier has about 88.5% accuracy!

We need better evaluation metrics

Confusion Matrix

- A table that summarizes the number of correct/incorrect predictions per class

		Predicted Class	
		+	−
Actual Class	+	True Positives (TP)	False Negatives (FN)
	−	False Positives (FP)	True Negatives (TN)

  - **True positives** (TP): Positive samples correctly classified as positive
  - **True negatives** (TN): Negative samples correctly classified as negative
  - **False positives** (FP): Negative samples incorrectly classified as positive
  - **False negatives** (FN): Positive samples incorrectly classified as negative

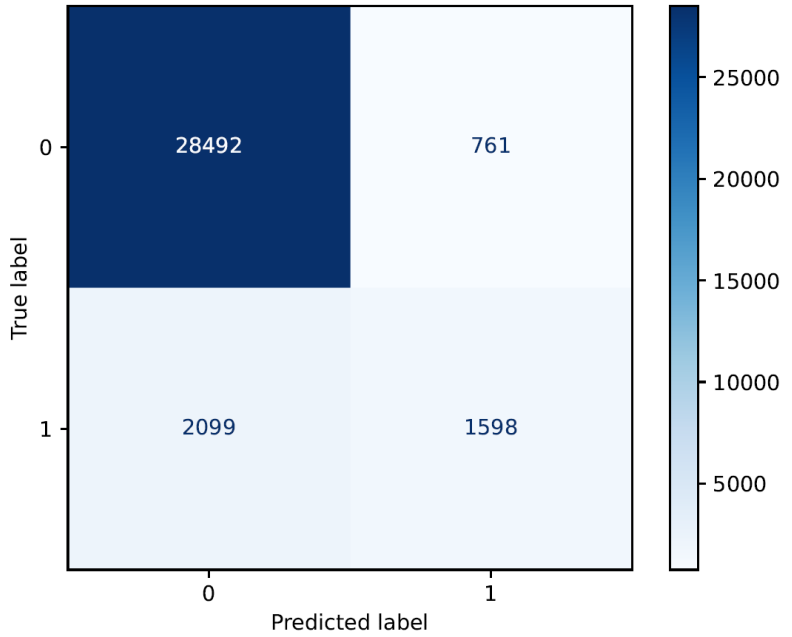
Confusion Matrix

- To compute the confusion matrix, we can use the function **confusion_matrix**:

```
from sklearn.metrics import confusion_matrix

y_pred = clf.predict(X_train)
conf_mat = confusion_matrix(y_train, y_pred)
print(conf_mat)
```

```
[[28492   761]
 [ 2099  1598]]
```

```
from sklearn.metrics import ConfusionMatrixDisplay

disp = ConfusionMatrixDisplay(confusion_matrix=conf_mat)
disp.plot(cmap='Blues')
```



Precision and Recall

- **Precision** is the ratio of the **true positives** to all **predicted positives**

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall** is the ratio of the **true positives** to all **actual positives**

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- In many problems, there is an inherent **tradeoff** between precision and recall
  - Models with higher precision tend to be cautious and only predict positive when they are very confident, leading to more actual positives missed
  - On the other hand, a model that classifies every sample as positive will have a perfect recall, but very low precision

Precision and Recall

- Computing the precision and recall using Scikit-Learn:

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_train, y_pred)
```

```
0.6691983122362869
```

```
recall_score(y_train, y_pred)
```

```
0.4255433324389589
```

- Now our model doesn't look as shiny as it did when we looked at its accuracy
  - When it claims a customer subscribed, it is correct only 67.2% of the time
  - Moreover, it only detects 43.6% of the subscriptions

F1 Score

- Precision and recall can be combined into a single metric known as **F1 score**
- F1 score is the **harmonic mean** of precision and recall:

$$F_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

  - The harmonic mean of two numbers tends to be closer to the smaller of the two
  - Both precision and recall must be high for the F1 score to be high
- To compute the F1 score, simply call the **f1_score()** function:

```
from sklearn.metrics import f1_score
f1_score(y_train, y_pred)
```

```
0.5202558635394456
```

Decision Threshold

- Decision threshold is the score above which a sample is predicted as positive
- In logistic regression
  - Decision score represents the signed distance of the input from the separating hyperplane
  - A threshold of 0 is used to predict if the sample belongs to the positive or negative class

```
y_scores = clf.decision_function(X_train[10:20])
y_scores
```

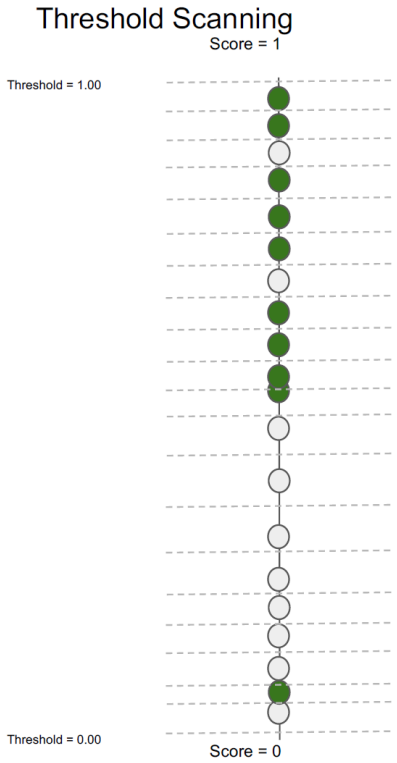
```
array([-4.02670565, -4.30432168,  0.31334842, -0.28544069, -1.08385411,
       -3.89626759,  1.40809097, -4.65313728, -1.00499063,  0.61793247])
```

```
y_pred[10:20]
```

```
array([0, 0, 1, 0, 0, 0, 1, 0, 0, 1], dtype=int64)
```

Thresholding

- By adjusting the decision threshold, we can control the precision-recall tradeoff
  - Lowering the threshold will increase recall and usually reduce precision (but not always)



Threshold	TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	F1
1.00	0	10	0	10	0.50	1	0	1	0
0.95	1	10	0	9	0.55	1	0.1	1	0.182
0.90	2	10	0	8	0.60	1	0.2	1	0.333
0.85	2	9	1	8	0.55	0.667	0.2	0.9	0.308
0.80	3	9	1	7	0.60	0.750	0.3	0.9	0.429
0.75	4	9	1	6	0.65	0.800	0.4	0.9	0.533
0.70	5	9	1	5	0.70	0.833	0.5	0.9	0.625
0.65	5	8	2	5	0.65	0.714	0.5	0.8	0.588
0.60	6	8	2	4	0.70	0.750	0.6	0.8	0.667
0.55	7	8	2	3	0.75	0.778	0.7	0.8	0.737
0.50	8	8	2	2	0.80	0.800	0.8	0.8	0.800
0.45	9	8	2	1	0.85	0.818	0.9	0.8	0.857
0.40	9	7	3	1	0.80	0.750	0.9	0.7	0.818
0.35	9	6	4	1	0.75	0.692	0.9	0.6	0.783
0.30	9	5	5	1	0.70	0.643	0.9	0.5	0.750
0.25	9	4	6	1	0.65	0.600	0.9	0.4	0.720
0.20	9	3	7	1	0.60	0.562	0.9	0.3	0.692
0.15	9	2	8	1	0.55	0.529	0.9	0.2	0.667
0.10	9	1	9	1	0.50	0.500	0.9	0.1	0.643
0.05	10	1	9	0	0.55	0.526	1	0.1	0.690
0.00	10	0	10	0	0.50	0.500	1	0	0.667

Thresholding

- For example, we can lower the threshold from 0 to -1.0:

```
y_scores = clf.decision_function(X_train)
threshold = -1.0
new_y_pred = (y_scores > threshold)
```

```
precision_score(y_train, new_y_pred)
```

```
0.5700045310376076
```

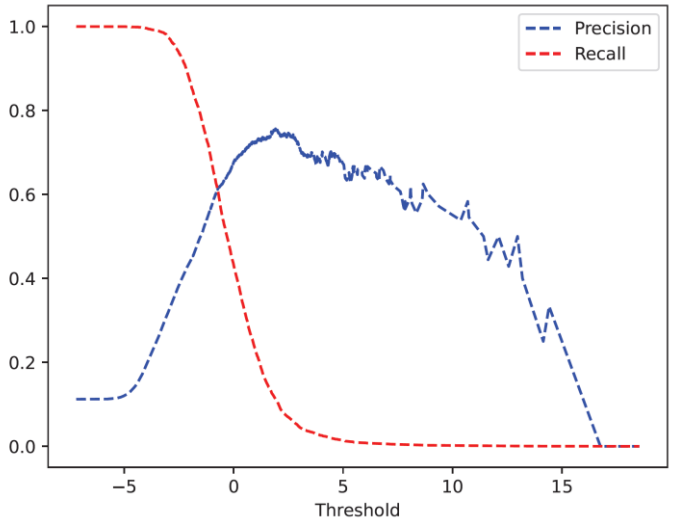
```
recall_score(y_train, new_y_pred)
```

```
0.6750737858867722
```

Precision-Recall Curve

- Visualizes the tradeoff between precision and recall for every possible threshold

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train, y_scores)
plt.plot(thresholds, precisions[:-1], 'b--', label='Precision')
plt.plot(thresholds, recalls[:-1], 'r--', label='Recall')
plt.xlabel('Threshold')
plt.legend()
```



Precision-Recall Curve

- Let’s suppose we decide to aim for 80% recall
- We can use the recalls array to find which threshold we need to use:

```
threshold = thresholds[np.where(recalls > 0.8)][-1]
threshold
```

```
-1.5731546420376081
```

```
y_recall_80 = (y_scores > threshold)
precision_score(y_train, y_recall_80)
```

```
0.4906995884773663
```

```
recall_score(y_train, y_recall_80)
```

```
0.7998390126106788
```

  - Precision decreased from 57% to 49% while recall increased from 67% to 80%

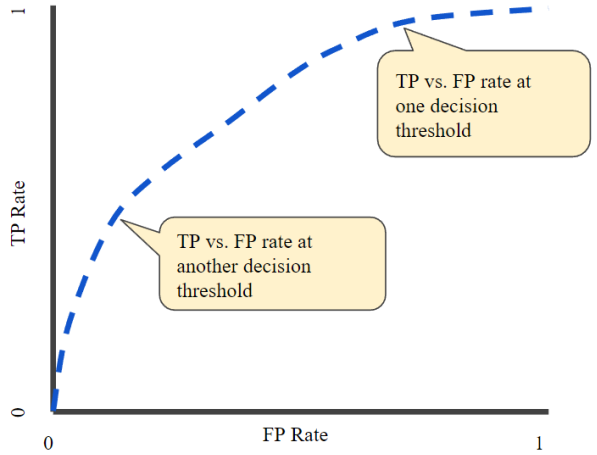
ROC (Receiver Operating Characteristic) Curve

- Shows the true positive rate and false positive rates across different thresholds
  - **False positive rate (FPR)**: fraction of the negative samples misclassified as positive

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

  - **True positive rate (TPR)**: fraction of the positive correctly classified as positive
  - Same as recall

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



- The ROC curve is monotonically increasing
- Reducing the decision threshold $\Rightarrow$ more samples are classified as positive $\Rightarrow$ both TPR and FPR increase
- The higher the curve is, the better the model is
- A model that makes random guesses resides along the main diagonal (TPR = FPR)

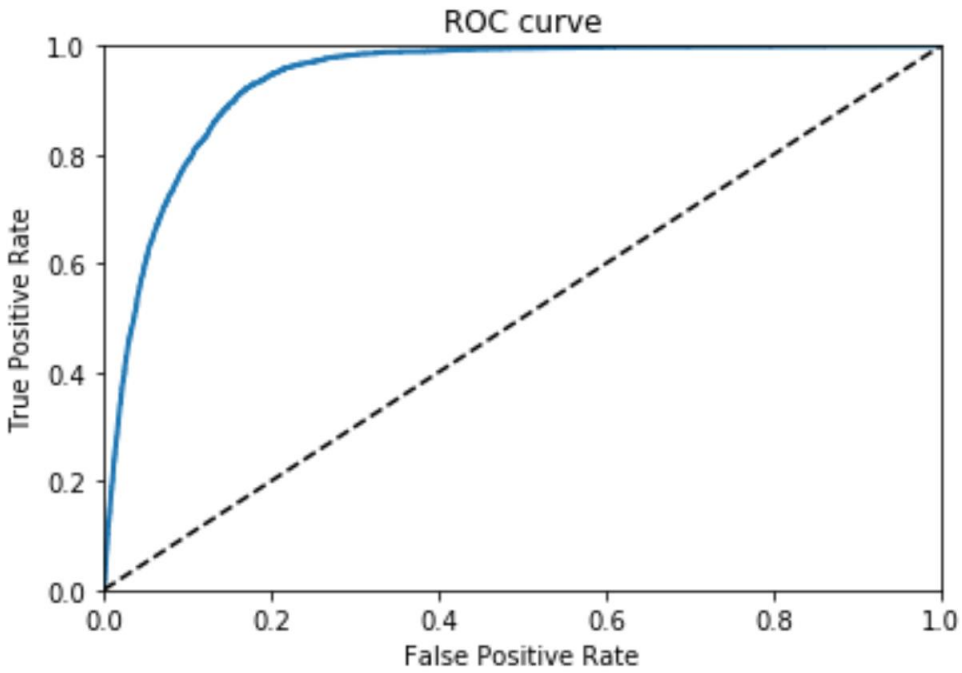
ROC Curve

- You can generate the plot using the **roc_curve()** function:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train, y_scores)
```

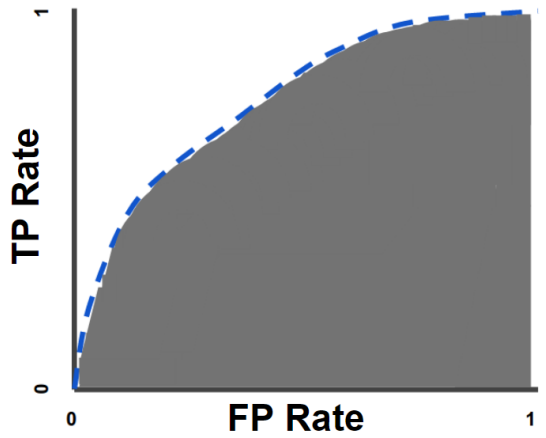
```
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, lw=2)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve')
```

```
plot_roc_curve(fpr, tpr)
```



AUC

- **AUC = area under the ROC curve**
- Gives a holistic evaluation of the model across all thresholds
- Can be used to compare different classifiers
- A perfect classifier will have AUC = 1
- A classifier that makes random guesses will have AUC = 0.5
- Computing AUC in Scikit-learn:




```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_train, y_scores)
```

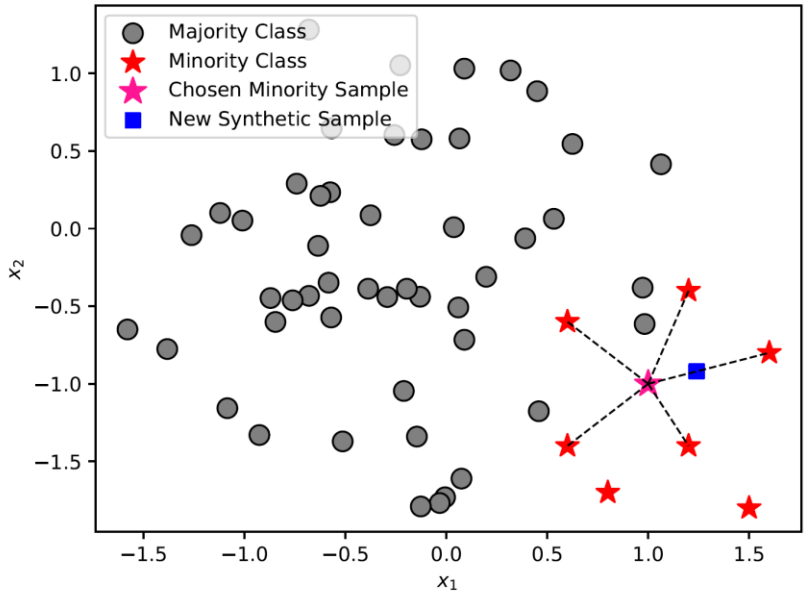
```
0.9363915446739913
```

Learning from Imbalanced Datasets

- Models trained on imbalanced datasets tend to perform poorly on the minority class
- Common strategies to tackle this issue:
  - **Sampling**: Modify the training data to make the dataset more balanced
    - **Oversampling the minority class:** replicate the positive examples or generate synthetic ones
    - **Undersampling the majority class:** reduce the number of negative examples
    - **Hybrid approaches**: combine oversampling and undersampling
  - **Cost-sensitive learning**: Penalize errors on the minority class more severely

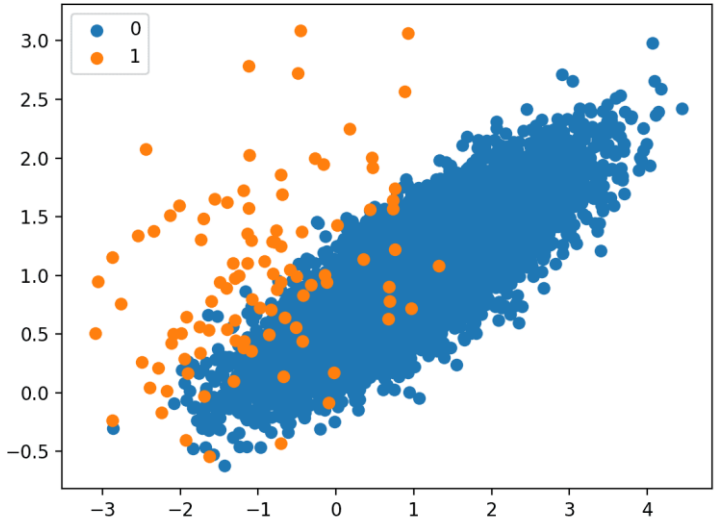
SMOTE (Synthetic Minority Oversampling)

- Generate new examples of the minority class through **interpolation**
  - Choose a random example from the minority class
  - Find the *k* nearest neighbors of that example (typically *k* = 5)
  - Randomly select one of the neighbors
  - Select a random point along the line between the example and its neighbor

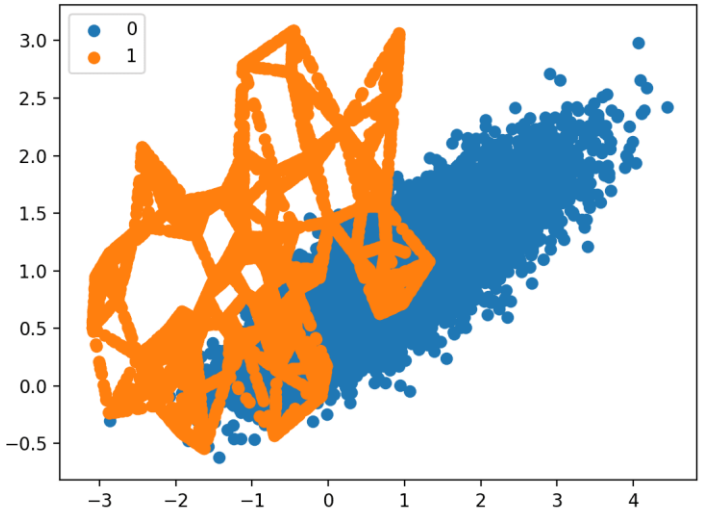


SMOTE (Synthetic Minority Oversampling)

- The training set before and after SMOTE:



Before SMOTE



After SMOTE

SMOTE

- The library imbalanced-learn provides various classes for imbalanced learning
- The SMOTE class implements this algorithm

SMOTE

class imblearn.over_sampling.SMOTE(*, *sampling_strategy='auto'*, *random_state=None*, *k_neighbors=5*) [source]

Class to perform over-sampling using SMOTE.

This object is an implementation of SMOTE - Synthetic Minority Over-sampling Technique as presented in [1].

Read more in the User Guide.

Parameters:

sampling_strategy : *float, str, dict or callable, default='auto'*

Sampling information to resample the data set.

- When `float`, it corresponds to the desired ratio of the number of samples in the minority class over the number of samples in the majority class after resampling. Therefore, the ratio is expressed as $\alpha_{os} = N_{rm}/N_M$ where N_{rm} is the number of samples in the minority class after resampling and N_M is the number of samples in the majority class.

Cost-Sensitive Learning

- A **cost matrix** assigns a weight for each block in the confusion matrix
- *C*(*i*|*j*): cost of misclassifying a class *j* example as class *i*

		Predicted Class	
	C(i|j)	+	-
Actual Class	+	*C*(+|+)	*C*(-|+)
	-	*C*(+|-)	*C*(-|-)

- The overall cost of a model *M* is:

$$C(M) = TP \cdot C(+|+) + FP \cdot C(+|-) + FN \cdot C(-|+) + TN \cdot C(-|-)$$

Cost-Sensitive Learning Example

Cost Matrix	Predicted Class		
	C(i|j)	+	-
Actual Class	+	-1	100
	-	1	0

Model M_1	Predicted Class		
		+	-
Actual Class	+	150	40
	-	60	250

Accuracy = 80%

Cost = 3910

Model M_2	Predicted Class		
		+	-
Actual Class	+	250	45
	-	5	200

Accuracy = 90%

Cost = 4255

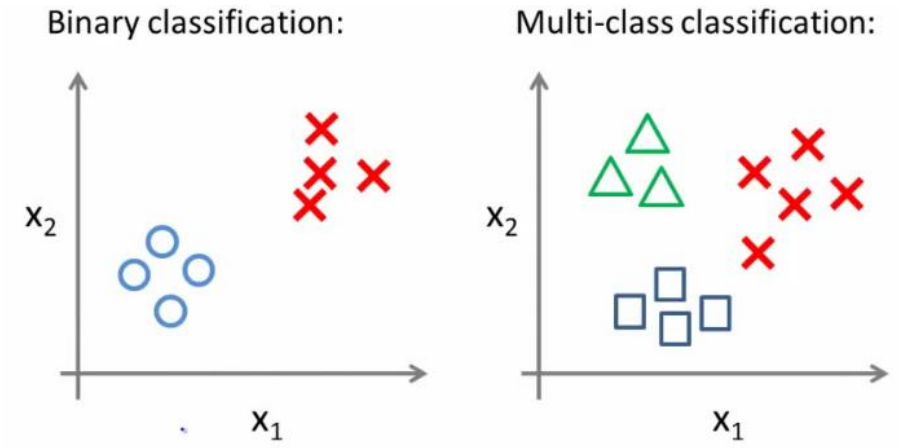
Cost-Sensitive Learning

- In Scikit-Learn, many algorithms support provide a **class_weight** parameter
- This parameter can affect the loss function or other parts of the training process
- Accepts a dictionary mapping class labels to weights
  - Example: class_weight={0: 0.2, 1: 0.8}
  - or the keyword 'balanced', which sets the weights inversely proportional to the proportion of each class in the dataset
  - The default None means that no class weighting is applied

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False,
tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100, multi_class='deprecated', verbose=0,
warm_start=False, n_jobs=None, l1_ratio=None)                          [source]
```

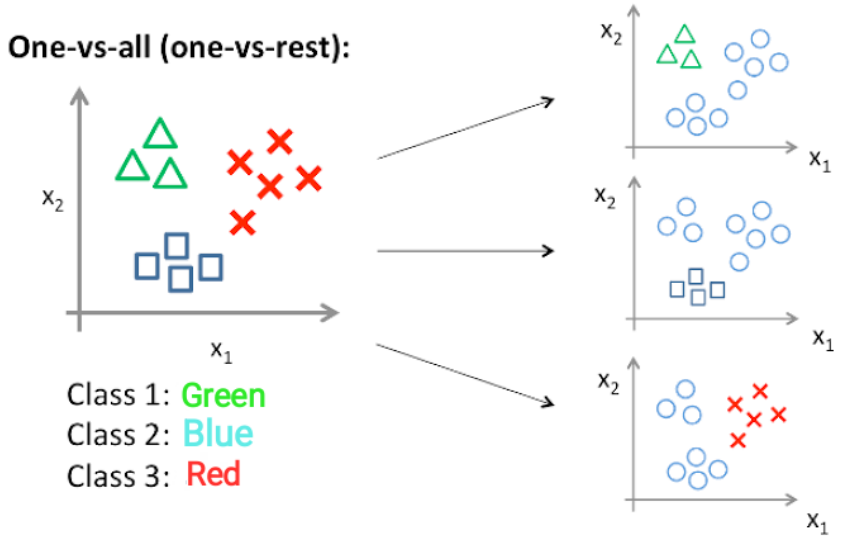
Multiclass Problems

- In many real-world problems the data is divided to more than two categories
  - e.g., face recognition, text classification
- Some ML algorithms are originally designed to handle only binary classification
  - e.g., logistic regression, SVM
- There are several approaches to extend binary classifiers into multiclass classifiers



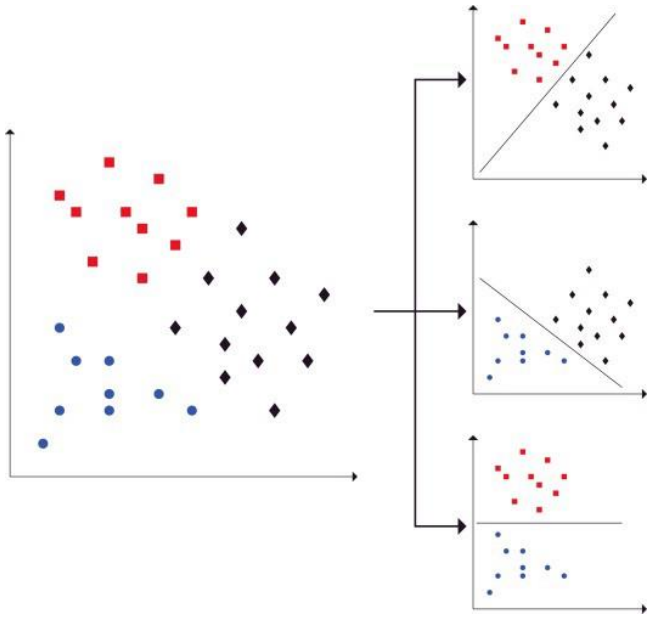
One vs. Rest (OvR)

- Train a binary classifier per class, with the samples of that class as positive samples and all other samples as negatives
- The base classifiers should provide a real-valued confidence score for their decision
- Predict the label *k* for which the class had the highest confidence score
- Issues
  - The scale of the confidence scores may differ
  - Each classification problem is imbalanced



One vs. One (OvO)

- Each binary classifier is used to distinguish between a pair of classes (c_i, c_j)
- Samples that don't belong to either c_i or c_j are ignored
- Use majority voting to make the final prediction
- The number of classifiers required is $\binom{K}{2} = \frac{K(K-1)}{2}$



- Advantages of OvO:
  - Usually provides better results than OvR
    - Each individual learning problem involves a small subset of the data
- Disadvantages of OvO:
  - Slower than OvR due to its $O(K^2)$ complexity
  - Doesn't provide probability estimates for the different classes

Example: MNIST

- To demonstrate multi-class classification, we'll use the MNIST dataset
- This is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau
- Each image is labeled with the digit it represents
- This data set has been studied so much that it is often called the "Hello World" of Machine Learning
- Description of the data set can be found here

Example: MNIST

- The following code loads the MNIST dataset using sklearn.datasets.fetch_openml

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('mnist_784', return_X_y=True)
```

```
X.shape
```

```
(70000, 784)
```

```
y.shape
```

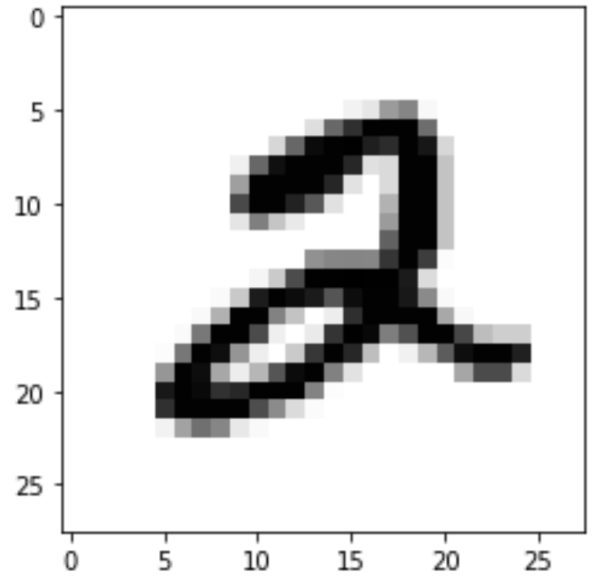
```
(70000,)
```

- There are 70,000 images of size 28×28, i.e., each image has 784 features
- Each feature represents one pixel's intensity from 0 (white) to 255 (black)

Example: MNIST

- Let’s take a peek at one digit from the data set:

```
digit_index = 5
digit_image = X[digit_index].reshape(28, 28)
plt.imshow(digit_image, cmap='binary');
```




```
y[digit_index]
```

```
'2'
```

Example: MNIST

- Let’s examine how many images we have from each digit type:

```
np.unique(y, return_counts=True)
```

```
(array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object),
 array([6903, 7877, 6990, 7141, 6824, 6313, 6876, 7293, 6825, 6958],
       dtype=int64))
```

  - The data set is well balanced
- The first 60,000 images belong to the training set and the last 10,000 to the test set
- So we can split the data set into training and test sets simply by:

```
X_train, y_train = X[:60000], y[:60000]
X_test, y_test = X[60000:], y[60000:]
```

OvR Classification

- For OvR classification, we can wrap our classifier with an OneVsRestClassifier object

class `sklearn.multiclass.`**`OneVsRestClassifier`**(*estimator*, *, *n_jobs=None*) [source]

- Let's train a logistic regression model on MNIST using OvR strategy:

```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

clf = OneVsRestClassifier(LogisticRegression(), n_jobs=-1)
clf.fit(X_train, y_train)
```

```
OneVsRestClassifier(estimator=LogisticRegression(), n_jobs=-1)
```

- The number of estimators created is:

```
len(clf.estimators_)
```

```
10
```

OvR Classification

- The accuracy of the classifier is:

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9261
Test set accuracy: 0.9181
```

- We can use it to predict the label for a given image:

```
clf.predict([X[digit_index]])
```

```
array(['2'], dtype='<U1')
```

```
np.round(clf.predict_proba([X[digit_index]]), 3)
```

```
array([[0.005, 0.   , 0.971, 0.   , 0.   , 0.002, 0.   , 0.   , 0.017,
        0.004]])
```

OvO Classification

- For OvO classification, we can wrap our classifier with an OneVsOneClassifier object:

```
from sklearn.multiclass import OneVsOneClassifier

clf = OneVsOneClassifier(LogisticRegression(), n_jobs=-1)
clf.fit(X_train, y_train)
```

```
OneVsOneClassifier(estimator=LogisticRegression(), n_jobs=-1)
```

- The number of estimators created is:

```
len(clf.estimators_)
```

```
45
```

- The accuracy scores are:

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9734
Test set accuracy: 0.9278
```

Softmax Regression

- **Softmax regression** (or **multinomial logistic regression**) is a generalization of logistic regression to multiclass problems
- Given a sample input **x**, we estimate the probabilities for each class *k* (*k* = 1, …, *K*)

$$p_i = P(y = i|\mathbf{x}) \qquad \sum_{i=1}^{k} p_i = 1$$

- Similar to logistic regression, we assume that the log odds between each probability to some reference probability (e.g., p_k) is a linear combination of the features:

$$\frac{p_i}{p_k} = e^{\mathbf{w}_i^T \mathbf{x}}$$

$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}}$$

- Note that there are *K* vectors of weights that need to be learned from the data

Softmax Regression

- Since all the probabilities sum to 1, we have:

$$p_k = 1 - \sum_{i=1}^{k-1} p_i = 1 - \sum_{i=1}^{k-1} \left(p_k e^{\mathbf{w}_i^T \mathbf{x}} \right)$$

$$p_k + p_k \left(\sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$

$$p_k \left(1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$

$$p_k = \frac{1}{1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}}}$$

- Substituting back to p_i we get:

$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}} = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{1 + \sum_{j=1}^{k-1} e^{\mathbf{w}_j^T \mathbf{x}}}, \qquad 1 \leq i < k$$

- By choosing arbitrarily $\mathbf{w}_k = 0$:

$$p_i = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{\sum_{j=1}^{k} e^{\mathbf{w}_j^T \mathbf{x}}}, \qquad 1 \leq i \leq k$$

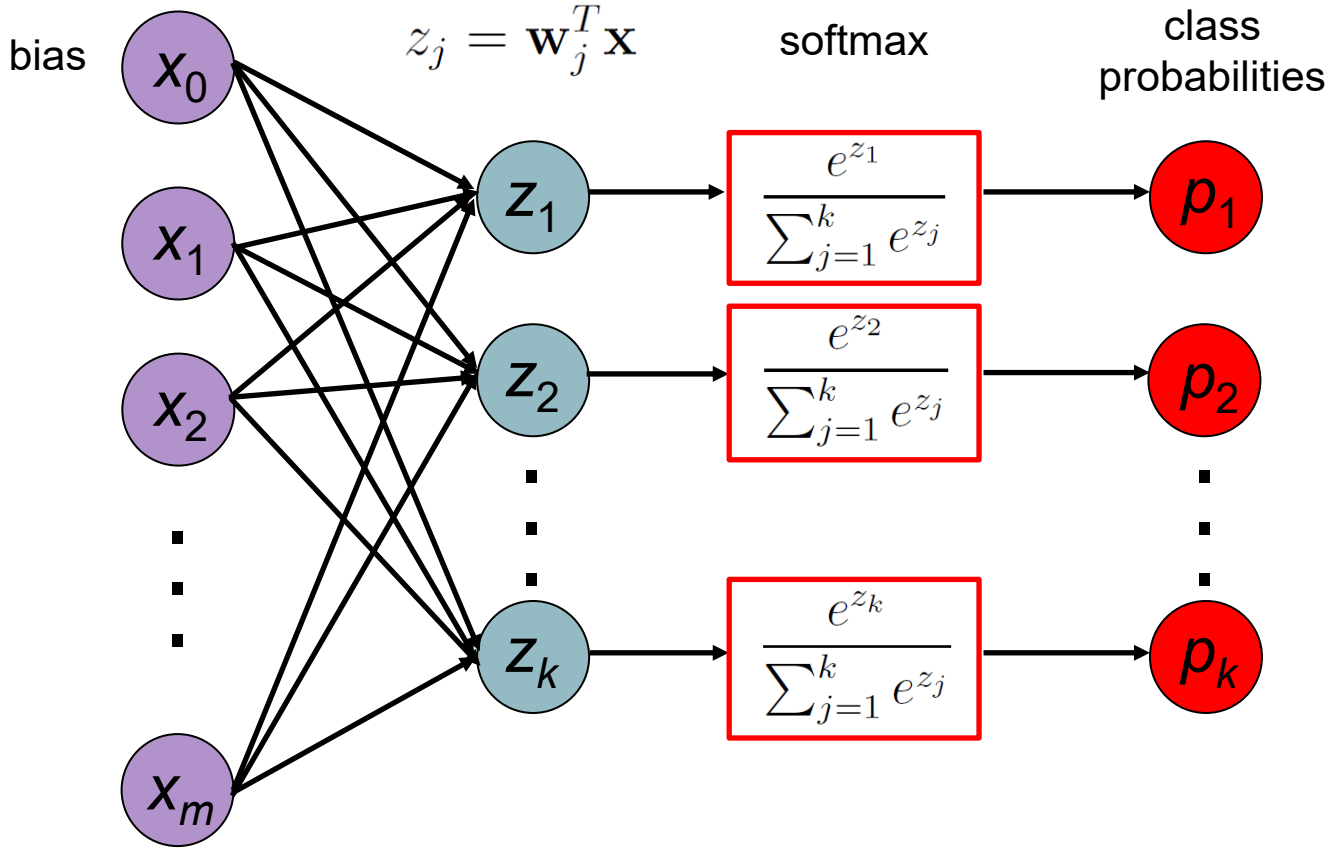
The Softmax Function

- The **softmax function**

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Smoothed version of the max function
- Example: the softmax of (1, 2, 6) is (0.007, 0.018, 0.976)
- The sigmoid function is a special case of the softmax function for a classifier with only two input classes

Softmax Regression



Maximum Likelihood Estimation

- Next, we write the likelihood function
- From our assumption we can write

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{k=1}^{K} p(C_k|\mathbf{x})^{\mathbf{y}_k} = \prod_{k=1}^{K} (h_k(\mathbf{x}))^{\mathbf{y}_k}$$

- The likelihood function is then given by

$$\mathcal{L}(\mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{i=1}^{n} p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{i=1}^{n} \prod_{k=1}^{K} (h_k(\mathbf{x}_i))^{y_{ik}}$$

- Taking the negative logarithm gives

$$J(\mathbf{w}_1, ..., \mathbf{w}_K) = -\log \mathcal{L}(\mathbf{w}_1, ..., \mathbf{w}_K) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(h_k(\mathbf{x}_i))$$

- This function is known as the **cross-entropy loss function**

Cross-Entropy Loss

- Extension of the log loss to the multi-class case
- Given a vector of estimated probabilities **p** and a vector **y** that represents a one-hot encoding of the label, the cross-entropy loss is defined as

$$L_{\mathrm{CE}}(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{k} y_i \log p_i$$

- Example: assume we have a three-class problem (k = 3) and a sample whose true label is class 2 ($\mathbf{y} = (0, 1, 0)^T$), and the model's prediction is $\mathbf{p} = (0.3, 0.6, 0.1)^T$
- Then the cross-entropy loss for that sample is:

$$L_{\mathrm{CE}} = -(0 \cdot \log 0.3 + 1 \cdot \log 0.6 + 0 \cdot \log 0.1) = 0.5108$$

Softmax Regression in Scikit-Learn

- The LogisticRegression class supports the cross-entropy loss

class `sklearn.linear_model.` **`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) [source]

- The multi_class argument can have one of the following options:
  - 'ovr' – uses the one-vs-rest (OvR) scheme
  - 'multinomial' – uses the cross-entropy loss
  - 'auto' (the default) selects 'ovr' if the data is binary, or if solver='liblinear' (which doesn't support multinomial), and otherwise selects 'multinomial'

Example: MNIST

- Let’s use softmax regression to classify the MNIST digits

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9339
Test set accuracy: 0.9255
```

- We got similar results to OvO, but this time using a single classifier

Example: MNIST

- Comparing between the different solvers:

```
for solver in ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']:
    clf = LogisticRegression(solver=solver)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()

    print('Solver:', solver)
    print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
    print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
    print('Elpased time:', np.round(end - start, 3), 'seconds')
```

```
Solver: lbfgs
Training set accuracy: 0.9339
Test set accuracy: 0.9255
Elpased time: 13.784 seconds
```

```
Solver: liblinear
Training set accuracy: 0.9312
Test set accuracy: 0.9171
Elpased time: 2490.652 seconds
```

```
Solver: saga
Training set accuracy: 0.9386
Test set accuracy: 0.9257
Elpased time: 203.123 seconds
```

```
Solver: newton-cg
Training set accuracy: 0.9448
Test set accuracy: 0.9208
Elpased time: 1524.278 seconds
```

```
Solver: sag
Training set accuracy: 0.9396
Test set accuracy: 0.9255
Elpased time: 153.617 seconds
```

Classification Metrics in Multiclass Problems

- Confusion matrix will be $K \times K$
- Most metrics (except accuracy) are analyzed as multiple 1-vs-many
- A **macro-average** computes the metric independently for each class and then takes the average (hence treating all classes equally)
- A **micro-average** aggregates the contributions of all classes to compute the average
- For example, for the precision measure

$$\text{Macro-Precision} = \frac{\sum_{k=1}^{K} \text{Precision}_k}{K}$$

$$\text{Micro-Precision} = \frac{\sum_{k=1}^{K} TP_k}{\sum_{k=1}^{K} TP_k + \sum_{k=1}^{K} FP_k}$$

- Micro-average is preferable if you suspect there might be a class imbalance

Classification Metrics in Multiclass Problems

- For multi-class problems, you can use sklearn.metrics.classification_report() to show a summary of the precision, recall, and F1 score for each class:

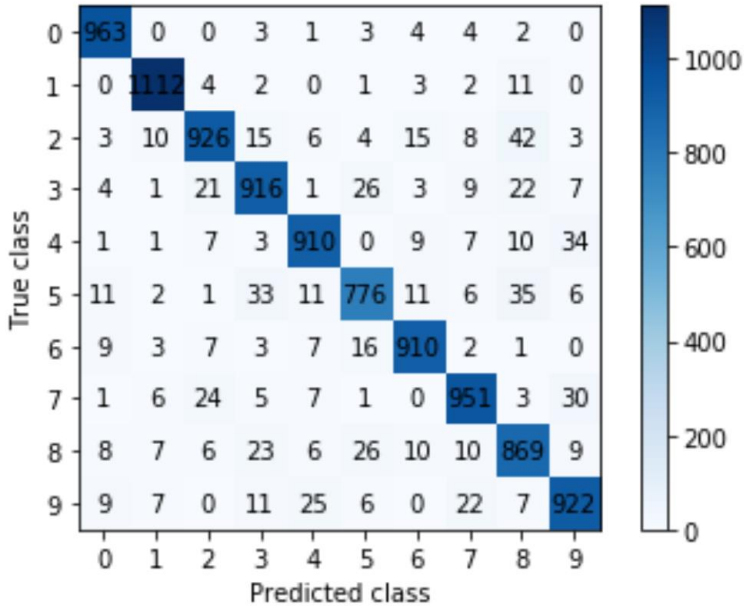
```
from sklearn.metrics import classification_report

y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.98      0.97       980
           1       0.97      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.90      0.91      0.91      1010
           4       0.93      0.93      0.93       982
           5       0.90      0.87      0.89       892
           6       0.94      0.95      0.95       958
           7       0.93      0.93      0.93      1028
           8       0.87      0.89      0.88       974
           9       0.91      0.91      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.93      0.93      0.93     10000
```

Error Analysis

- Let’s examine the confusion matrix

```
from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(y_test, y_pred)
plot_confusion_matrix(conf_mat)
```

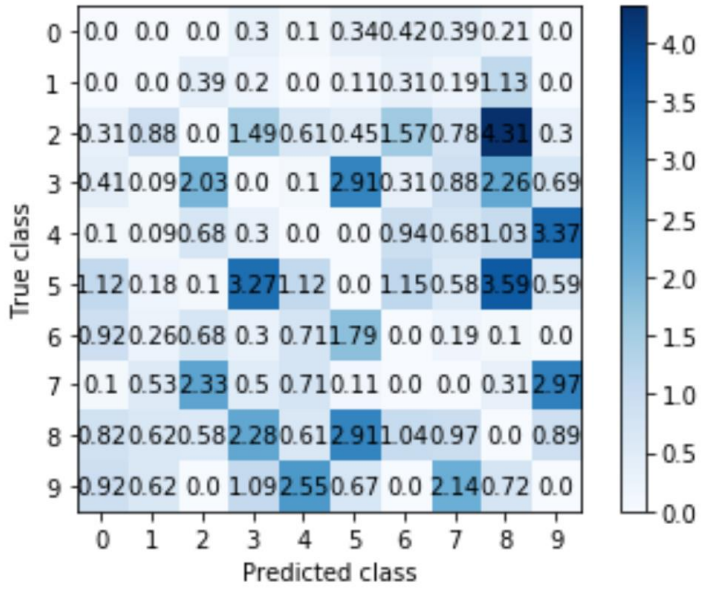


- Since the number of digits in each class are not equal, it makes more sense to examine the relative errors instead of the absolute

Error Analysis

- To make the errors in the confusion matrix be relative instead of absolute:
  - Divide each value in the matrix by the number of images in the corresponding class
  - Fill the diagonal with zeros to keep only the errors

```
norm_conf_mat = conf_mat / conf_mat.sum(axis=1) * 100
np.fill_diagonal(norm_conf_mat, 0)
norm_conf_mat = np.round(norm_conf_mat, 2)

plot_confusion_matrix(norm_conf_mat)
```

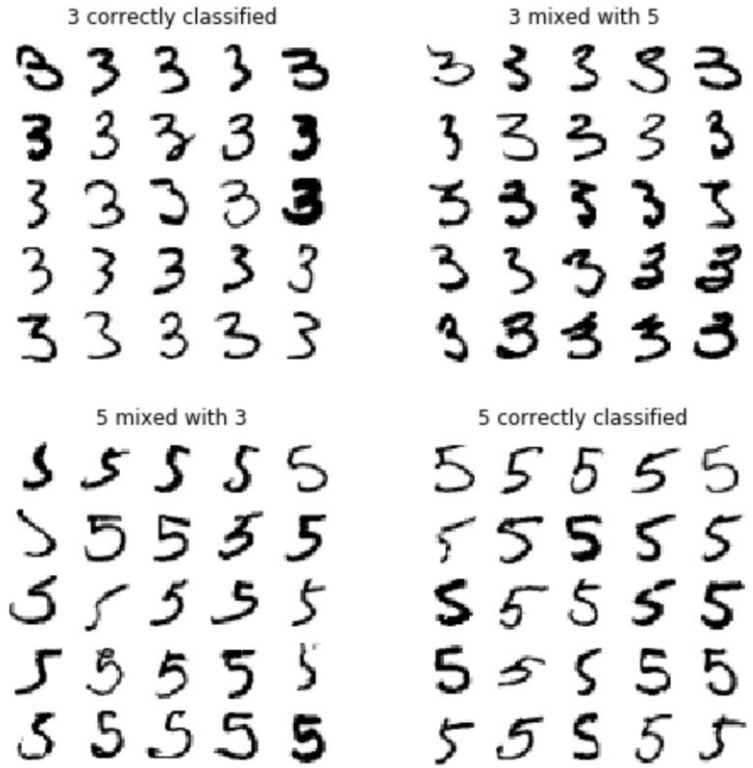


- It seems that our main confusions are between 3/5, 4/9, 5/8, and 7/9

Error Analysis

- Analyzing individual errors can help you gain insights on what your classifier is doing
- For example, we can plot examples of 3s and 5s from the test set:

```
cls_a, cls_b = '3', '5'
X_aa = X_test[(y_test == cls_a) & (y_pred == cls_a)]
X_ab = X_test[(y_test == cls_a) & (y_pred == cls_b)]
X_ba = X_test[(y_test == cls_b) & (y_pred == cls_a)]
X_bb = X_test[(y_test == cls_b) & (y_pred == cls_b)]

fig, ax = plt.subplots(2, 2, figsize=(8, 8))
plot_digits(ax[0, 0], X_aa[:25], images_per_row=5,
            title='3 correctly classified')
plot_digits(ax[0, 1], X_ab[:25], images_per_row=5,
            title='3 mixed with 5')
plot_digits(ax[1, 0], X_ba[:25], images_per_row=5,
            title='5 mixed with 3')
plot_digits(ax[1, 1], X_bb[:25], images_per_row=5,
            title='5 correctly classified')
```



Logistic Regression Summary

Pros

- Efficient to train
- Easy to implement
- Provides class probabilities
- Won't overfit easily, as it's a linear model
- Highly interpretable
  - there is a different weight for each feature
- Can handle irrelevant/redundant features
  - e.g., by assigning weights close to 0 for them
- A small number of hyperparameters

Cons

- Can learn only linear decision boundaries
- Typically outperformed by more complex algorithms
- Cannot handle missing values
- Requires scaling of the data