

# Metody arbitrażu statystycznego

Michał Sierakowski

IBM, University of Warsaw

*m.sierakowski@mimuw.edu.pl*

*Slack channel: <https://ibm.enterprise.slack.com/archives/C09KE8UFH33>*

4 listopada 2025r.

# Przegląd

1 Zasady

2 Wykład

3 Laby

- Lab I.50
- Lab I.51

4 Praca domowa

- Lab H.I.52

# Zasady

- **ocena końcowa** = 40% ćwiczenia + 10% kolokwium  
+ 15% prezentacja + 35% projekt

- [0 – 50%]: ocena 2
- [50% – 60%]: ocena 3
- [60% – 70%]: ocena 3,5
- [70% – 80%]: ocena 4
- [80% – 90%]: ocena 4,5
- [90% – 100%]: ocena 5

# Zasady: ćwiczenia (waga 40%)

- **indywidualne zadania w trakcie zajęć**

- punktowane: {0, 1}
- zrzut ekranu i wyjaśnienie na zajęciach

- **grupowe zadania w trakcie zajęć**

- punktowane: {0, 1} tylko dla **obecnych na zajęciach członków grupy** (co należy zaznaczyć zgłaszając rozwiązanie)
- zrzut ekranu i wyjaśnienie na zajęciach

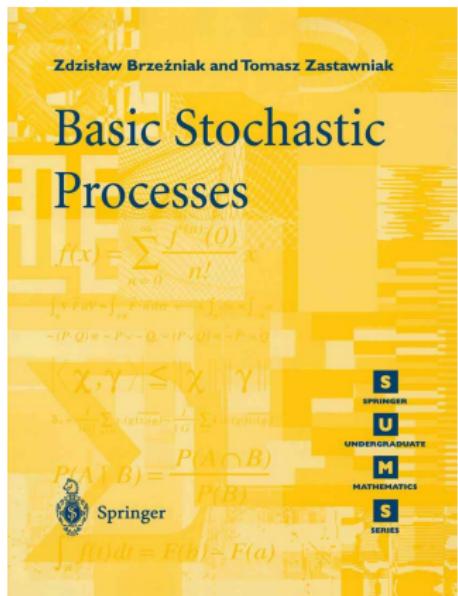
- **indywidualne zadania domowe**

- punktowane: {0, 1}
- **czas wykonania: do rozpoczęcia kolejnych zajęć**
- zrzut ekranu lub listing

- **grupowe zadania domowe**

- punktowane: {0, 1} dla **wszystkich członków grupy**
- **czas wykonania: do rozpoczęcia kolejnych zajęć**
- zrzut ekranu lub listing

# Motywacja



# Pragmatycznie o AI

## ✓ Machine Learning

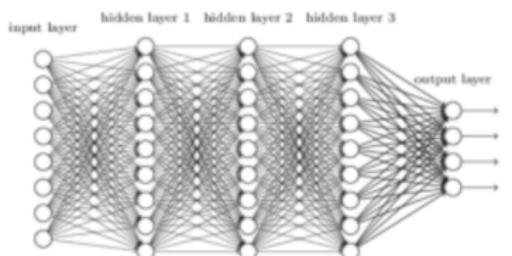
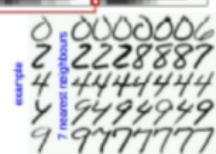
- ✓ Supervised ML
- ✓ Unsupervised ML

## Deep Learning

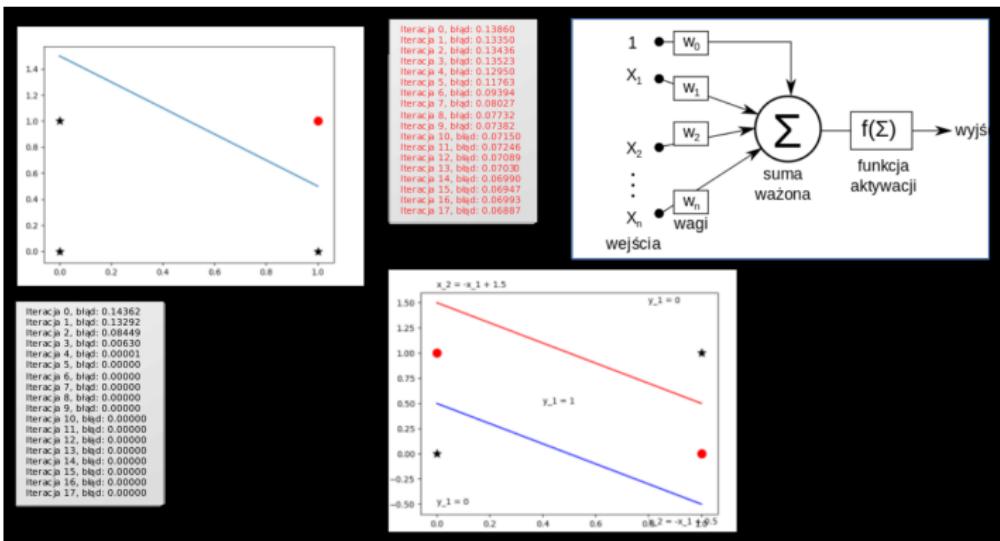
- ✓ Reinforcement Learning

### Example: handwritten digits

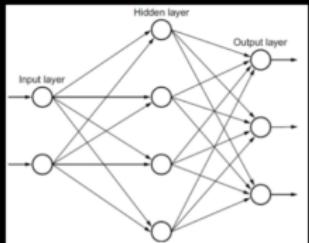
- 16x16 bitmaps
- 8-bit grayscale
- Euclidian distance
  - over raw pixels



# Pragmatycznie o AI



# Pragmatycznie o ML



```
Listing 7.3 neuron.py
from typing import List, Callable
from util import dot_product

class Neuron:
    def __init__(self, weights: List[float], learning_rate: float, activation_function: Callable[[float], float]) -> None:
        self.weights = weights
        self.activation_function = Callable[[float], float] = activation_function
        self.derivative_activation_function: Callable[[float], float] = derivative_activation_function
        self.learning_rate = float = learning_rate
        self.bias = float = 0.0
        self.delta = float = 0.0

    def output(self, inputs: List[float]) -> float:
        self.output_cache = dot_product(inputs, self.weights)
```



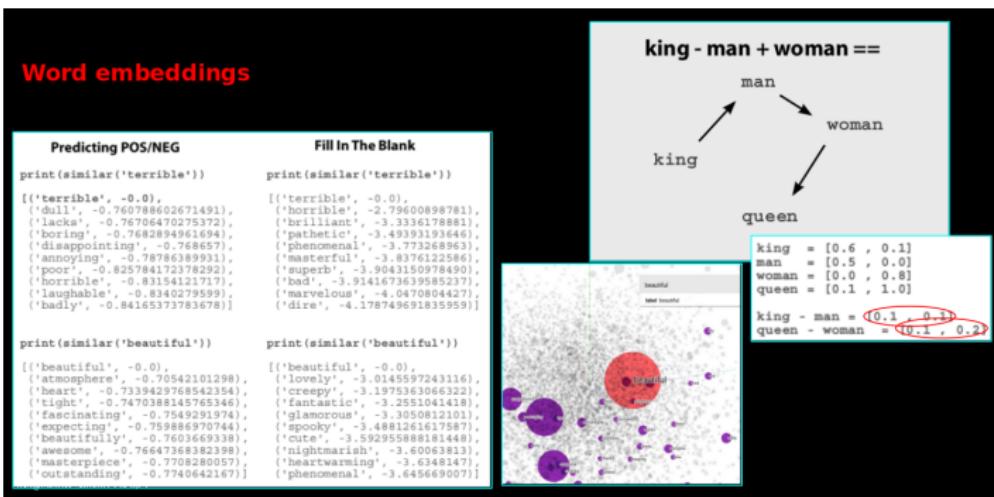
```
Listing 7.3 layer.py
from typing import List, Callable, Optional
from random import random
from neuron import Neuron
from util import dot_product

class Layer:
    def __init__(self, previous_layer: Optional[Layer], num_neurons: int, learning_rate: float,
                 activation_function: Callable[[float], float] = sigmoid, derivative_activation_function: Callable[[float], float] = derivative_sigmoid):
        self.previous_layer = previous_layer
        self.num_neurons = num_neurons
        self.neurons: List[Neuron] = [Neuron(weights=[random() for _ in range(previous_layer.num_neurons + 1)], learning_rate=learning_rate,
                                             activation_function=activation_function,
                                             derivative_activation_function=derivative_activation_function) for _ in range(num_neurons)]
```

```
Listing 7.4 network.py
from typing import List, Tuple
from functions import reduce
from layer import Layer
from util import dot_product, sigmoid, derivative_sigmoid
T = TypeVar('T') # output type of interpretation of neural network

class Network:
    def __init__(self, layer_structures: List[int], learning_rate: float, activation_function: Callable[[float], float] = sigmoid, derivative_activation_function: Callable[[float], float] = derivative_sigmoid) -> None:
        if len(layer_structures) < 3:
            raise ValueError('Error: Should be at least 3 layers (1 input, 1 hidden, 1 output)')
        self.layers: List[Layer] = []
        input_layer: Layer = Layer(layer_structures[0], learning_rate, activation_function, derivative_activation_function)
        self.layers.append(input_layer)
        for i in range(1, len(layer_structures)):
            next_layer = Layer(layer_structures[i], num_neurons=layer_structures[i+1],
                               next_layer=Layer(layer_structures[i], num_neurons, learning_rate,
                                               activation_function, derivative_activation_function))
            self.layers.append(next_layer)
```

# Osadzanie słów



Rysunek: Osadzanie słów

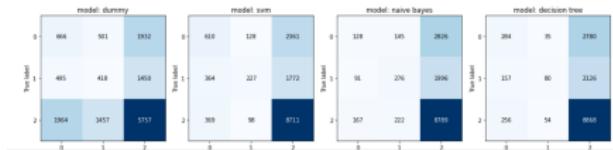
# Współczesne modele NLP

	Word order awareness	Context awareness (cross-words interactions)
Bag-of-unigrams	No	No
Bag-of-bigrams	Very limited	No
RNN	Yes	No
Self-attention	No	Yes
Transformer	Yes	Yes

Rysunek: Modele NLP

# Klasyfikacja wieloklasowa

```
model=dummy, f1_micro: mean=0.47, std=0.01
model=ssvm, f1_micro: mean=0.65, std=0.0
model=naive bayes, f1_micro: mean=0.63, std=0.0
model=decision tree, f1_micro: mean=0.63, std=0.0
model=random forest, f1_micro: mean=0.63, std=0.0
```



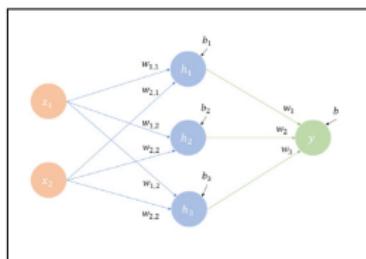
## Lab I.50 klasyfikacja wieloklasowa

Wykonaj polecenia zawarte w dołączonym notatniku

Lab.I.50.raw.ipynb.

Uwaga: Przejrzyj również plik helper.py w celu poprawienia wyników zawartych w przykładach z powyższego notatnika.

# Prosta sieć neuronowa



## Lab I.51 Prosta sieć neuronowa

Po prostu wykonaj polecenia w notatniku `Lab.I.51.raw.ipynb` i pobierz utworzony w pakiecie Keras model sieci neuronowej.  
Wykorzystaj bibliotekę `ibm_watson_studio_lib`.

## Zamiast Lab I.32

**Lab H.I.52** Proszę poprawnie wykonać wszystkie w kroki w dołączonym notatniku `Lab.H.I.52.raw.ipynb` wykorzystując inny niż w kodzie model LLM dostępny w IBM Cloud.

# The End