Aula 5 — Classificação

Inteligência Artificial

Universidade Federal do ABC

29 de novembro de 2017

Introdução

- Hoje vamos deixar o Pacman um pouco de lado
  - Vamos tentar classificar dígitos manuscritos
- Mas antes do fim da aula vamos ver os algoritmos de classificação no contexto do Pacman também :D

Problema

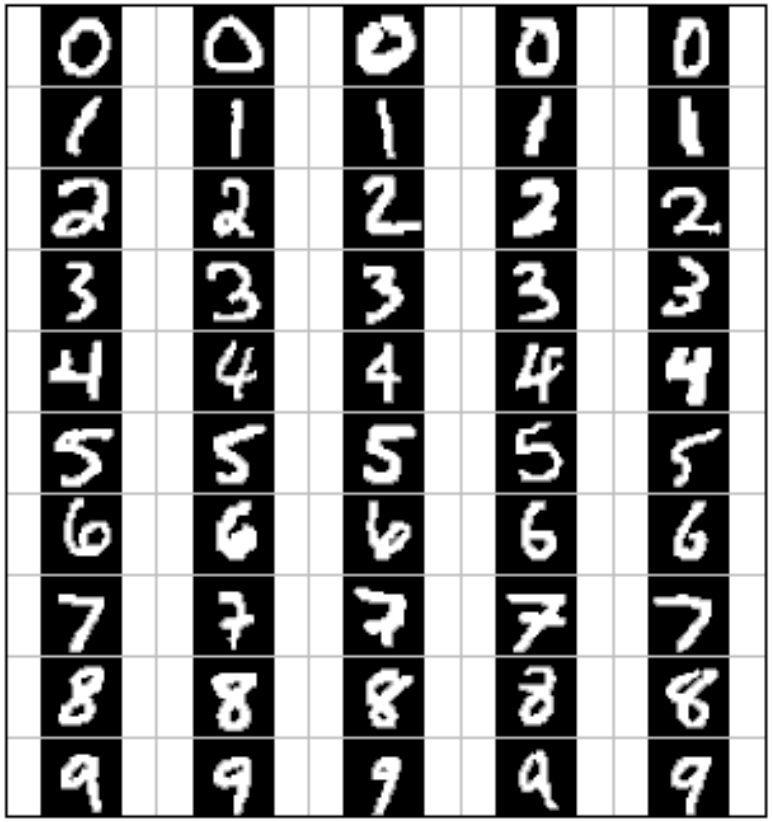


Figura 1: OCR

Arquivos

- Baixe o arquivo aula5_classificacao.zip e descompacte
- Você modificará os seguintes arquivos nos exercícios:
  - perceptron_pacman.py
  - perceptron.py
  - dataClassifier.py
  - answers.py
- Para verificar a corretude das suas soluções você executará o arquivo:
  - autograder.py
- Os seguintes arquivos contém informações relevantes para os exercícios:
  - classificationMethod.py: Classe pai dos classificadores
  - sample.py: leitura dos dados para classificação
  - mostFrequent.py: Classificador básico baseado em classe majoritária

Exercício 1: *Perceptron*

- Dada uma lista de atributos $\mathbf{f}$ representando um exemplo de entrada:
  - $\text{score}(\mathbf{f}, y) = \sum_i f_i w_i^y$
  - O exemplo de entrada é classificado como da classe de maior score
- Como aprender os pesos?
  - Ao errarmos, considere y a classe desejada e y' a obtida:

$$w^y = w^y + \mathbf{f}$$

$$w^{y'} = w^{y'} - \mathbf{f}$$

Exercício 1: *Perceptron*

- Finalizar implementação de `train` em `perceptron.py`
- `trainingData[i]`=$\mathbf{f}$ do i-ésimo exemplo
- `self.weights[y]`=$\mathbf{w^y}$
- Ambos são Counters
  - Counter implementa +/-/* elemento a elemento
- `trainingLabels[i]`= rótulo do i-ésimo exemplo
- `self.classify([trainingData[i]])` retorna a classe com maior score (em uma lista)

Exercício 1: *Perceptron*

- Para testar sua implementação:
  - `python2.7 dataClassifier.py -c perceptron`
  - A taxa de acerto na validação e no teste deve estar entre 40-70%
- `python2.7 autograder.py -q q1`

Exercício 2: *Análise de pesos*

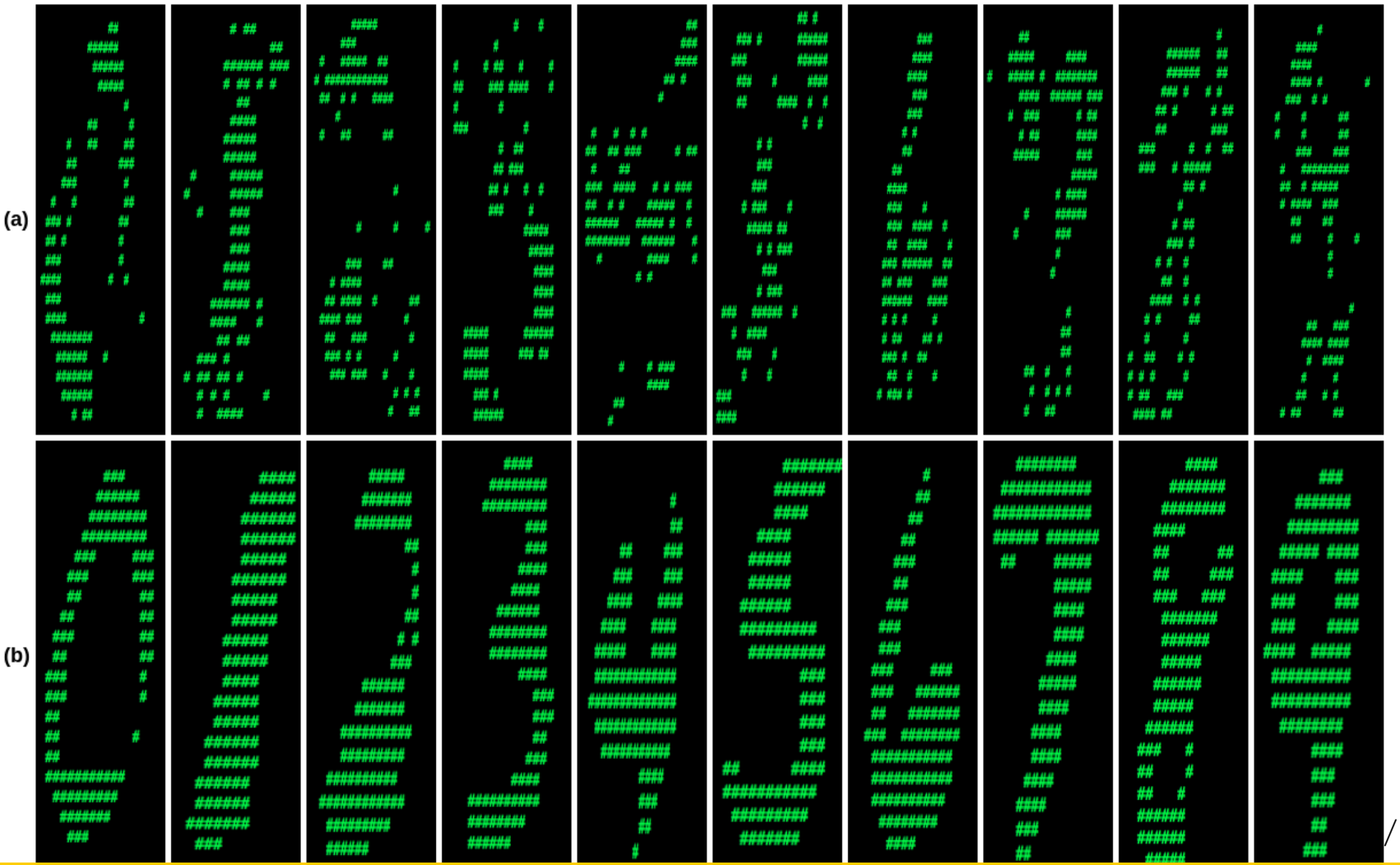
- Redes neurais, perceptron sendo uma delas, são conhecidas por serem modelos **black-box**
- Vamos dar uma olhada na razão
- A ideia é verificar as características que são proeminentes em uma classe

Exercício 2: *Análise de pesos*

- Finalize `findHighWeightFeatures(self, label)` em `perceptron.py`
  - Retornar uma lista das 100 características com maior peso para cada classe
  - O seguinte comando permite visualizar os pesos:
    - `python dataClassifier.py -c perceptron -w`
  - Dica, se `dct` é um dicionário/Counter:
    - `sorted(dct, key=dct.get, reverse=True)`
    - retorna uma lista das chaves de `dct` ordenadas pelos valores em ordem decrescente

Exercício 2: *Análise de pesos*

- Responda em `answers.py` qual das imagens abaixo é mais parecida com o que você encontrou



Exercício 3: *Engenharia de Atributos*

- Muitas vezes uma melhor representação dos dados tem um impacto maior do que um classificador mais complexo
- Vamos tentar melhorar nossos atributos
  - Atualmente são apenas indicadores se um determinado pixel está ativo

Exercício 3: *Engenharia de Atributos*

- Finalizar `EnhancedFeatureExtractorDigit` em `dataClassifier.py`
  - Verifique os erros que o classificador está obtendo com sua representação atual para ter ideias para novas características
  - O método `analysis` pode ser utilizado para extrair informações sobre o aprendizado
  - Largura e altura do grid de dígitos pode ser recuperada por: DIGIT_DATUM_WIDTH e DIGIT_DATUM_HEIGHT

Exercício 3: *Engenharia de Atributos*

- Seus atributos devem ser binários
- Ideias:
  - Percentual de pixels ativos em cada linha está acima de determinado limiar?
  - Percentual de pixels ativos no total está acima de determinado limiar?
  - Número de regiões contíguas de pixels ativos é igual a... (*one-of-k encoding*)
- Para testar:
  - `python2.7 dataClassifier.py -d digits -c naiveBayes -f -a -t 1000`
- Para avaliar:
  - `python2.7 autograder.py -q q3`
  - Precisa obter >82% na validação e >78% no teste

Exercício 4: *Clonando comportamento*

- Vamos desenvolver um perceptron modificado em `perceptron_pacman.py`
  - Nada muito distante do seu código em `perceptron.py`
- Os dados serão **estados** e os rótulos as **ações possíveis**
  - Todos os rótulos vão **compartilhar** os pesos
  - Atributos gerados em relação ao estado e uma ação possível

Exercício 4: *Clonando comportamento*

- Para cada ação, computar:

$$\text{score}(s, a) = \mathbf{w} * f(s, a)$$

- Rótulo predito igual à ação com maior score
- Atualização dos pesos ocorre de forma parecida, porém, temos um único vetor de pesos compartilhado:
  - a = ação correta:

$$\mathbf{w} = \mathbf{w} + f(s, a)$$

  - a' = ação incorreta:

$$\mathbf{w} = \mathbf{w} - f(s, a')$$

Exercício 4: *Clonando comportamento*

- Complete o método `train` em `perceptron_pacman.py`
- Para testar:
  - `python2.7 dataClassifier.py -c perceptron -d pacman`
- Para avaliar:
  - `python2.7 autograder.py -q q4`
  - Acurácia de $>70\%$

Exercício 5: *MIRA*

- Podemos melhorar o treinamento do perceptron ajustando os pesos apenas o *necessário*
  - Essa é a ideia do Margin Infused Relaxed Algorithm (MIRA)
- Vamos finalizar a implementação do MIRA na função `trainAndTune` em `mira.py`
  - Como esperado, é similar a implementação do perceptron

Exercício 5: *MIRA*

- Cada classe tem seu vetor de pesos
  - `self.weights[label]`
- Os exemplos são processados um por vez e rotulados de acordo com a classe com maior score
- Em caso de erros ($y \neq y'$), realizar os ajustes dos pesos:

$$\mathbf{w}^{y} = \mathbf{w}^{y} + \tau \times \mathbf{f}$$

$$\mathbf{w}^{y'} = \mathbf{w}^{y'} - \tau \times \mathbf{f}$$

$$\tau = min(C, \frac{(\mathbf{w}^{y'} - \mathbf{w}^{y})\mathbf{f} + 1}{2\|\mathbf{f}\|_2^2})$$

Exercício 5: *MIRA*

- Note que é necessário encontrar o valor de C (entre os possíveis em `Cgrid` que apresenta melhor acurácia nos dados de **validação**
  - Pesos são aprendidos usando `trainingData`, acurácia estimada em `validationData`
  - Ao final da execução, guardar em `self.weights` os pesos obtidos com o melhor valor de C
  - No caso de empates, preferência por menor valor de C
- Para testar:
  - `python2.7 dataClassifier.py -c mira --autotune`
  - Acurácia de validação e teste devem ser em torno de 60%
- Para avaliar:
  - `python2.7 autograder.py -q q5`

Exercício 6: *Engenharia de Atributos – Pacman*

- Você desenvolverá seus próprios atributos para clonar o comportanto de outro agente
- Os seguintes agentes estão disponíveis para servirem de alvo:
  - *StopAgent*: agente que fica parado
  - *FoodAgent*: agente que se preocupe apenas em comer comida, ignorando o ambiente
  - *SuicideAgent*: agente que sempre se move em direção ao fantasma mais próximo, independente se ele está assustado
  - *ContestAgent*: agente que leva em conta todo o ambiente

Exercício 6: *Engenharia de Atributos – Pacman*

- Estão disponíveis jogos gravados de cada agente em `data/pacmandata`
  - 15 para treino, 10 para validação e 10 para teste
- Acrescente novos atributos no método `EnhancedPacmanFeatures` em `dataClassifier.py`
- Com seus atributos, você deve conseguir pelo menos:
  - 90% de acurácia no ContestAgent
  - 80% de acurácia nos demais

Exercício 6: *Engenharia de Atributos – Pacman*

- Para testar seu código contra um agente:
  - `python2.7 dataClassifier.py -c perceptron -d pacman -f -g nomeAgente -t 1000 -s 1000`
- Caso queira utilizar o seu perceptron_pacman
  - `python2.7 pacman.py -p ClassifierAgent --agentArgs agentToClone=nomeAgente`
- `python2.7 autograder.py -q q6`