Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Aprendizagem Automática

Bank Notes Classification

Autores:
Lucas Fischer
Joana Martins

Professor:
Ludwig Krippahl

Outubro de 2018

1 Introdução

O presente relatório tem o objetivo de descrever a implementação desenvolvida pelo grupo para solucionar os problemas propostos pelo enunciado do projeto. Dispômos de um conjunto de dados que representa as medições de quatro *features* que permitem classificar notas em "falsas"e "verdadeiras". Pretende-se implementar classificadores que o façam automaticamente, depois de treinados com uma fracção dos dados.

O relatório está organizado da seguinte forma. Começamos por descrever a estrutura geral do código desenvolvido, depois deste passo descrevemos os três classificadores e os parâmetros que foram optimizados nos mesmos de modo a reduzir a sua estimativa de erro, bem como o processo de optimização dos parametros. Finalmente, no ponto 4, apresentamos uma tabela comparativa de resultados obtidos para os três classificadores e tiramos conclusões sobre qual o melhor classificador para esta aplicação.

2 Estrutura do código

O código desenvolvido para este projeto foi implementado em **Python 3.6** e, de modo a facilitar a sua compreensão, o mesmo foi dividido em três ficheiros `.py`, que seguem a estrutura presente na figura 1:

- **tp1.py** - Este ficheiro é responsável pela inicialização da aplicação. Este ficheiro simplesmente importa a classe `Assignment.py` e executa um dos seus métodos.

- **assignment.py** - Esta classe implementa toda a lógica de execução dos três classificadores (*Logistic Regression*, *K-Nearest-Neighbours* e *Naïve* Bayes). É nesta classe que se encontram os passos comuns entre os classificadores como:

  - **Ler os dados** - através da inicialização de um objeto desta classe (método `__init__`)
  - **Processá-los** - utilizando a técnica de *shuffling* e *normalização/standardização*, através do método `process_data` da classe `Assignment`.
  - **Treinar os classificadores** - de maneira a obter os melhores parâmetros para cada classificador, através da técnica *cross validation*.
  - **Estimar o erro real** - calculado sobre o *test set* para obter uma estimativa não-enviesada do erro real.

- **helper_funcs.py** - Neste ficheiro estão presentes várias funções auxiliares à execução dos classificadores. É neste ficheiro que se encontra o código que implementa:

  - **Normalização/Standardização**
  - **Cálculo do erro** (quer de *cross validation* quer do *test set*, para todos os classificadores)
  - ***Plot* dos gráficos**
  - **Teste de McNemar**

- **naive_bayes.py** - Esta classe contém todo o código responsável pela implementação do classificador *Naïve* Bayes, i.e., esta classe implementa o processo de treino e de classificação deste classificador.

```
ML_Banknotes/
    ├── images          ; Diretoria que contem as imagens dos plots
    ├── tp1.py          ; Ficheiro onde é iniciada a aplicação
    ├── assignment.py   ; Classe que executa os classificadores
    ├── helper_funcs.py ; Ficheiro que contem funções auxiliares
    ├── naive_bayes.py  ; Classe que implementa o classificador Naive Bayes
    └── TP1-data.csv    ; Ficheiro CSV com os dados do problema
```

Figura 1: Esquema da estrutura do código.

3 Descrição dos classificadores

3.1 Optimização de parâmetros

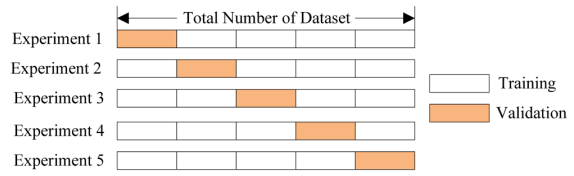

Figura 2: Esquema da divisão dados em conjuntos de treino e validação. (retirado de [2])

A optimização dos parâmetros foi obtida através do processo de *cross-validation*. *Cross-validation* é uma técnica que permite avaliar modelos com base num conjunto de instâncias de um modelo e não apenas numa só hipótese do mesmo. Isto faz com que a certeza associada a este modelo seja mais elevada uma vez que não estamos a limitar-nos a uma instância deste logo à partida. Para conseguir avaliar o modelo é então necessário obter uma média dos erros das diferentes instâncias. Torna-se então necessário usar múltiplas amostras distintas e representativas para treino e validação de cada uma delas. Uma forma de conseguirmos obter estas amostras é utilizando a técnica de **K-Folds** que consiste em dividir o nosso *training set* em **K** sub-conjuntos, reservando um desses sub-conjuntos para *validation set*, e usando os restantes **K-1** como *training set*. A reserva de uma fracção dos dados para validação da instância permite obter uma estimativa não-enviesada do erro desta. De uma forma análoga, os nossos dados iniciais não são inteiramente utilizados como *training set*. Uma fracção dos dados é reservada como *test set* para possibilitar obter uma estimativa não-enviesada do erro "real", i.e., o erro que esperamos encontrar com os dados futuros.

3.2 *Logistic Regression*

O primeiro classificador que implementámos foi o *Logistic Regression*. Este classificador toma partido da técnica de *regression* que consiste em encontrar por interpolação, uma função que recebe dados e devolve um resultado composto por um ou mais valores contínuos, com o intuito de prever correctamente o resultado para dados futuros (extrapolação) [1]. *Logistic Regression* utiliza esta técnica para obter um discriminante onde a probabilidade de encontrar um ponto pertencente à classe C_1 é igual à probabilidade de encontrar um ponto da classe C_0. A obtenção deste discriminante é realizada através da minimização do erro em prever probabilidades (conhecido como *logistic loss*) pelo método

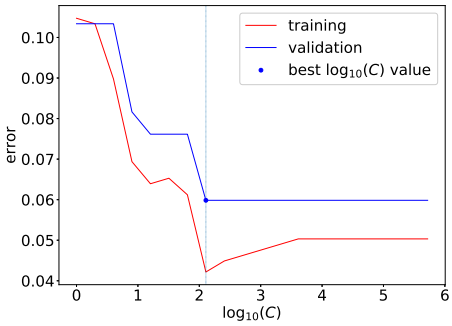

Figura 3: Erro de *cross-validation* para a classificador *Logistic Regression* em função do parâmetro de regularização C para um exemplo de execução do código. A linha tracejada a cinzento claro é um guia visual para o ponto associado ao menor error de validação (círculo azul).

de máxima verosimelhança. A probabilidade de classificar um ponto com *features* $\vec{x}$ como pertencente à classe C_1, $p(C_1|\vec{x})$ é descrita pela função logística $g(\vec{x}, \tilde{\omega})$, que representa o descriminante e em que $\tilde{\omega}$ é o conjunto dos seus parâmetros. Esta função tem características muito distintas da função quadrática usada no cálculo do erro da regressão linear. Enquanto que na regressão linear se pretende penalizar os pontos mais afastados da curva que estamos a tentar encontrar, na regressão logística este comportamento não é desejado visto que levaria a que pontos afastados do discriminante o afectassem em demasia. De facto, no caso da classificação, os pontos mais afastados do discriminante são os menos importantes visto que a maior dificuldade reside em correctamente separar os pontos de classes distintas que se encontram próximos deste.

Na nossa solução este classificador é implementado no ficheiro `helper_funcs.py` tirando partido da classe `LogisticRegression` proveniente da biblioteca *SciKit Learn*. Esta classe aceita diversos parâmetros opcionais sendo um deles o parâmetro **C**, o parâmetro a optimizar para este classificador. Este parâmetro **C** descreve o inverso da força de regularização (técnica utilizada para evitar *overfitting* por meio de adicionar uma penalização à função de custo) sobre este classificador, naturalmente, quanto menor for o valor de **C** maior será o efeito de regularização.

Na figura 3 podemos observar o processo de *cross-validation* para este classificador onde são desenhadas as linhas que descrevem o erro de treino e o erro de validação em função do valor de **C**. Visto que queremos encontrar o valor de **C** que minimize o erro de validação, o C escolhido encontra-se então no mínimo da função geradora da linha do erro de validação, assinalado na figura com o ponto azul.

3.3 *K-Nearest-Neighbours*

O seguinte classificador por nós implementado foi o *K-Nearest-Neighbours*, que em contraste com o *Logistic Regression* é considerado um classificador de *lazy learning*. *Lazy learning* é o termo dado a classificadores que não têm modelos para descrever as suas hipóteses e que não precisam de ser treinados uma vez que a decisão de atribuição da classe de futuros pontos é feita com base nos pontos existentes até ao momento.

Este classificador atribui a um ponto, a classe a que pertence a maioria dos seus k pontos vizinhos mais próximos. Esta proximidade pode ser medida através de diferentes funções de distância, sendo que a que foi usada no código deste trabalho é a empregue pelo *SciKit Learn* por defeito, a distância de Minkowski com $p = 2$, também designada por distância Euclideana.

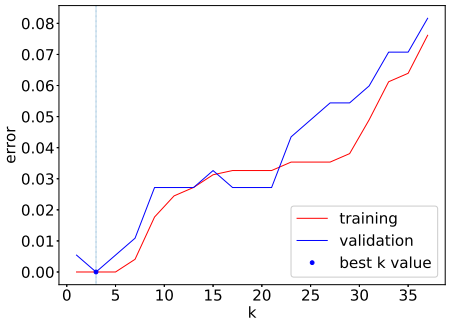

Figura 4: Erro de *cross-validation* para o classificador *K-Nearest-Neighbours* em função do parâmetro k (número de vizinhos) para um exemplo de execução do código. A linha tracejada a cinzento claro é um guia visual para o ponto associado ao menor error de validação (círculo azul).

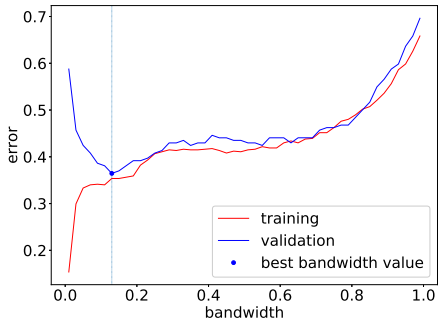

Figura 5: Erro de *cross-validation* para o classificador *Naïve Bayes* em função do parâmetro bandwidth (largura da *Kernel density estimator*) para um exemplo de execução do código. A linha tracejada a cinzento claro é um guia visual para o ponto associado ao menor error de validação (círculo azul).

A implementação deste classificador na nossa solução é similar à implementação do *Logistic Regression* uma vez que também foi utilizada uma classe proveniente da biblioteca *SciKit Learn*

Na figura 5 podemos observar, tal como para o *Logistic Regression*, o processo de *cross-validation* com a representação gráfica das linhas que descrevem o erro de treino e validação em função dos diferentes valores de k utilizados. É possível também observar o ponto correspondente ao melhor valor de k para este exemplo, aquele para o qual se verificou o menor valor dos erros de validação.

3.4 *Naïve* Bayes

O classificador de Bayes funciona sobre o conceito básico de escolher a classe que máximize a probabilidade de uma classe dado um conjunto de *features*. O problema com este método é que não é um classificador prático uma vez que é práticamente impossível obter todas as probabilidades conjuntas de todas as classes com todos os diferentes conjuntos de *features*. Para solucionar este problema utilizamos uma versão alterada do classificador de Bayes, chamado de *Naïve* Bayes, que parte do princípio que os diferentes conjuntos de *features* tornam-se independentes uma vez fixada uma classe (na prática este princípio raramente se verifica mas ainda assim este classificador consegue obter resultados bastante positivos).

No presente trabalho, as *features* são descritas por varíaveis contínuas. Como não é conhecida a função de distribuição de probabilidades, o cálculo das probabilidades condicionais de cada *feature*, dada uma determinada classe, foi efectuado pelo método não-paramétrico de *Kernel density estimators* (KDEs). A nossa imple-

Classificador	Test error	Precision	Recall	Accuracy	Best value (C,k,bw)
Logistic Regression	0.0066	0.9851	0.9900	0.9934	128.0000
K-nearest-neighbours	0.0000	1.0000	1.0000	1.0000	3.0000
Naïve Bayes	0.0618	0.8692	0.9254	0.9382	0.1300

Tabela 1: Valores das métricas *test error*, *precision* e *accuracy* e parâmetro de optimização do modelo escolhido com base no menor erro de validação.

mentação deste classificador encontra-se essencialmente no ficheiro `naive_bayes.py` que contém a classe `NaiveBayes` onde se pode encontrar o código que implementa o processo de treino, a estimativa de erros, bem como o processo de classificação em si. No método `get_prior_and_kde` é implementado o código que obtém as probabilidades *a priori* de todas as classes bem como a criação de um KDE para cada conjunto de *features* utilizando um valor **bw**(*bandwidth*) que define a largura da função de *kernel*. Os métodos `calculate_training_error` e `calculate_test_error` são métodos similares que são executados sobre os conjutos de treino/validação e teste respectivamente. Nestes métodos encontra-se a implementação dos processos de treino e classificação, onde primeiro iteramos sobre todas as *features* dos nossos dados e transformamos, através de um processo chamado de *broadcasting* da biblioteca *Numpy*, cada conjunto de *features* nos seus logaritmos de probabilidade dados pelo KDE criado para a *feature* sobre a qual estamos a iterar. Uma vez obtidos os logaritmos das probabilidades e fazendo uso novamente do processo de *broadcasting* transformamos todas as linhas na soma das colunas correspondentes a essa linha com a probabilidade *a priori* de pertencer a uma determinada classe. Finalmente encontramos a classe que maximize esta soma de maneira a determinar qual a classe de cada um dos ponto que queremos classificar, de modo a conseguirmos calcular o erro de treino e de teste.

4 Resultados

Tendo implementado diferentes classificadores com base no mesmo conjunto inicial de dados, torna-se interessante analisar se algum deles é significativamente mais eficaz que os outros. Esta análise pode ser feita para pares de classificadores com o teste de McNemar. Este baseia-se no facto de, na comparação de dois métodos de classificação, o quociente:

$$q = \frac{(|e_{01} - e_{1,0}| - 1)^2}{e_{01} + e_{1,0}} \approx \chi_1^2 \qquad (1)$$

seguir uma distribuição χ_1^2 com um grau de liberdade, onde e_{01} é o número de pontos a que o primeiro classificador atribuiu a classe incorreta e o segundo a classe correta, e e_{10} o número de pontos correctamente classificados pelo primeiro e incorrectamente pelo segundo.

A nossa hipótese nula é a de ambos os classificadores serem igualmente eficazes e escolhendo um intervalo de confiança de 95% temos que, se o valor do quociente 1 for superior a 3.84, a hipótese nula não se verifica. Neste caso é então possível distinguir qual o classificador mais eficaz. Este é dado pelo que tem menor erro de teste de entre os dois. Os resultados das

comparações para todas os pares distintos de dois dos três classificadores são impressos na linha de comandos quando é chamado o método `mcNemar_test()` da classe `Assignment`.

Importa referir que, para se poder verificar para cada ponto os valores de e_{01} e e_{10} para diferentes classificadores, é necessário que o processo de *shuffling* dos dados seja feito uma única vez, previamente à execução de cada um dos classificadores.

Num exemplo de execução deste teste verificou-se que os classificadores *K-Nearest-Neighbours* e *Logistic Regression* são significativamente diferentes e que o *K-Nearest-Neighbours* é provavelmente melhor que o *Logistic Regression* visto que tem menor erro de teste. A comparação do *K-Nearest-Neighbours* com o *Naïve* Bayes indica que também neste caso os dois classificadores não são igualmente eficazes, sendo o *K-Nearest-Neighbours*, que tem menor erro de teste, provavelmente o melhor. Finalmente, os classificadores *Logistic Regression* e o *Naïve* Bayes são diferenciáveis na sua eficácia de acordo com o teste de McNemar. Entre estes, o erro de teste calculado revelou-se menor para o *Logistic Regression* pelo que este é provavelmente o melhor de entre os dois.

Complementarmente, quando um classificador é executado fora do teste de McNemar, por exemplo com a chamada (para a *Logistic Regression*):

```
assignment.logistic_reg(),
```

é retornada para a linha de comandos uma tabela com os resultados de diferentes métricas (*test error*, *precision*, *recall*, *accuracy*) para a hipótese optimizada do modelo, e o valor do parâmetro de optimização para esta (C, k ou *bandwidth* conforme o modelo).

Referências

[1] Ethem Alpaydin. *Instroduction to Machine Learning*, chapter Supervised learning. The MIT Press, 2014.

[2] Dan B. What is cross validation, 2018.