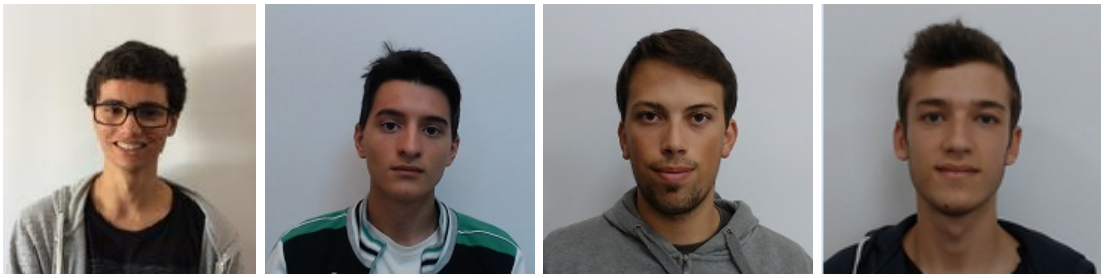
Universidade do Minho
Mestrado Integrado em Engenharia Informática
4º ano - 2º Semestre

Aprendizagem Automática II

Desenvolvimento de uma abordagem de AutoML integrada numa ferramenta de machine learning existente (DeepMol)



a80813 - Ângelo André Castro de Sousa
a83798 – David António Vieira dos Santos Moura Neto
a84092 - Diogo Paulo da Costa Pereira
a85334 – Pedro Paulo Fontes Delgado

6 de abril de 2021

Conteúdo

Lista de Figuras

1 Introdução

No âmbito da unidade curricular Aprendizagem Automática II, do perfil Ciência de Dados do Mestrado Integrado de Engenharia Informática, foi proposta a implementação de um novo módulo de Machine Learning, para adicionar à ferramenta em desenvolvimento ***DeepMol***. Este novo módulo, denomina-se ***AutoML***, cujo objetivo principal é facilitar o uso do *pipeline* de *Machine Learning* a utilizadores sem experiência de programação. Neste relatório são explicadas as várias implementações desenvolvidas para a automatização da ferramenta desde os *inputs*, a escolha de featurização, os modelos a utilizar e os seus parâmetros, decisão sobre se e que hiperparâmetros otimizar e finalmente a seleção de métricas a usar para comparar os modelos e avaliar os resultados.

2 Abordagem utilizada

Sendo este um projeto de desenvolvimento de um módulo integrado numa ferramenta pré-existente e que tira partido de outros módulos já existentes do ***DeepMol***, a abordagem do grupo consistiu em primeiramente realizar um estudo da ferramenta através da leitura de *notebooks* pré-existentes, da produção de *notebooks* e da análise do código de forma a explorar o *pipeline* e perceber a função de cada um dos módulos. De seguida, procedeu-se a uma discussão sobre o design do módulo ***AutoML*** e a implementação das suas partes constituintes.

O grupo utilizou um repositório no *github* para desenvolvimento, privado dada a natureza inacabada do módulo do ***DeepMol***. No entanto, toda a documentação e o módulo ***DeepMol*** encontram-se no repositório público. Finalmente, o código desenvolvido foi integrado num *branch* do repositório *BioSystemsUM* para posterior aprovação dos docentes.

É ainda importante referir duas notas: ao longo de todo o processo de estudo, planeamento e implementação, o grupo contou com o auxílio do orientador do projeto e responsável pela ferramenta ***DeepMol*** de forma a que este módulo seguisse a sua visão; o código desenvolvido pelo grupo não alterou de nenhuma forma o código do repositório do ***DeepMol***.

Parte I
DeepMol

3 Estudo do pipeline do DeepMol

3.1 Funções dos módulos

O estudo do *pipeline* começou por ser realizado através da análise e produção de *notebooks* tirando partido dos módulos existentes de forma a perceber a função de cada um, a ordem pela qual se usavam e que tipo de problemas resolviam. Deste modo, segue-se uma lista do que o grupo apurou sobre cada um dos módulos existentes:

- **loaders** - carregamento dos *datasets*
- **Datasets** - representação dos *datasets* em memória
- **featureSelection** - escolha de atributos
- **compoundFeaturization** - extrair atributos dos compostos químicos (moléculas)
- **models** - wrappers para modelos *SkLearn* e *Keras*
- **metrics** - cálculo de métricas de classificação e regressão (suporta *scikit-learn* e métricas implementadas pelo utilizador)
- **splitters** - divisão dos *datasets* (treino/validação/teste, *k-fold*, treino/teste, etc.)
- **parameterOptimization** - otimização de hiperparâmetros

3.2 Ordem das operações

De seguida, gerou-se um diagrama com um pipeline geral que utilizasse os módulos anteriores de forma a perceber a ordem das operações. Este diagrama foi a base para o desenvolvimento do módulo de ***AutoML***.

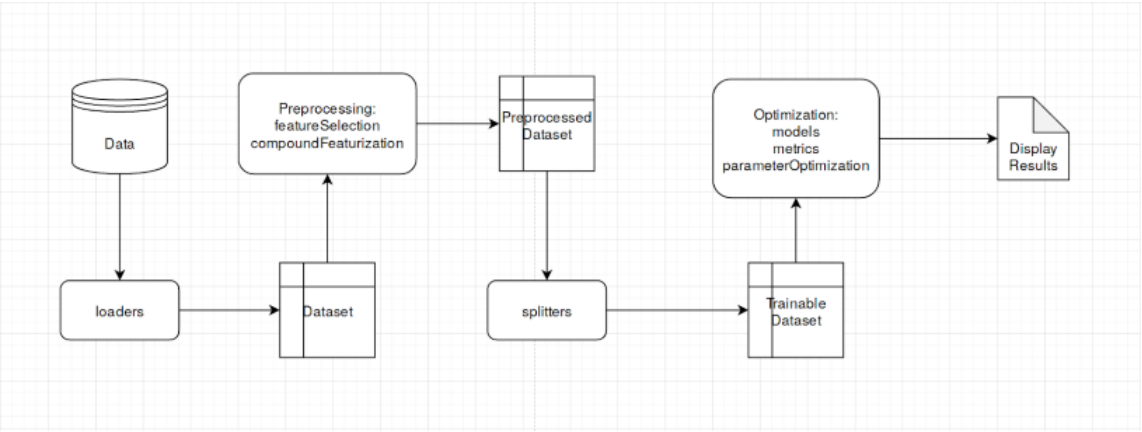


Figura 1: Diagrama que descreve o *pipeline* de *Machine Learning* com o ***DeepMol***.

Parte II
AutoML

4 Inputs

4.1 Explicação

4.1.1 Carregamento do Dataset

Parâmetros

- **dataset**: ficheiro *dataset* (csv);
- **mols**: coluna moléculas;
- **id_field**: coluna *id*;
- **labels_field**: coluna(s) *labels*;
- **features_fields**: coluna(s) *features*;
- **features2keep**: quais as *features* a manter;
- **shard_size**: tamanho dos fragmentos de dados a processar de cada vez.

4.1.2 Featurização

Parâmetros

- **featurizers**: lista de featurizers a comparar;
- **name**: qual *featurizer* a usar;
- **type**: quais os parâmetros a especificar ou usando default;
- **params**: lista de parâmetros (depende do *featurizer*).

4.1.3 Modelos

Parâmetros

- **models**: lista de modelos a comparar;
- **name**: identificador de modelo;
- **params**: depende do modelo.

4.1.4 Métricas

- **metrics**: lista de métricas a usar (as métricas encontram-se no Metrics dentro do ***DeepMol***).

Featurizers Suportados - Todos os *featurizers* suportados pelo módulo ***rdkitFingerprints*** do ***DeepMol***.

Modelos Suportados

- *RandomForestClassifier*
- *C-Support Vector Classification*
- *KNeighborClassifier*
- *DecisionTreeClassifier*
- *RidgeClassifier*
- *SGDClassifier*
- *AdaBoostClassifier*

Notas sobre os parâmetros dos modelos
- '{}' significa que não são passados parâmetros;
- um valor por cada campo significa que o model vai correr usando esses parâmetros;
- vários valores por parâmetro, é feita otimização de parâmetros e escolhida a melhor combinação.

4.2 Exemplo

```
{
    "load": {
        "dataset": "preprocessed_dataset_wfoodb.csv",
        "mols": "Smiles",
        "labels_fields": "Class",
        "id_field": "ID"
    },
    "featurizers": [
        { "name" : "morgan", "type" : "default", "params":{}},
        { "name" : "rdk", "type" : "default", "params":{}}
    ],
    "models": [
        {"name": "RandomForestClassifier", "params": {"n_estimators": [50,100,200], "
           criterion": ["entropy","gini"]}},
        {"name": "SVM", "params": {"C":[0.5,0.9]}}
    ],
    "metrics": ["roc_auc_score", "precision_score", "accuracy_score"]
}
```

5 Featurizer

O objetivo desta classe passa por construir os vários tipos de featurização. Esta classe recebe uma lista de *featurizers* e para cada um deles, verifica qual o *featurizer* a usar a partir do campo **name**, e o campo referente aos parâmetros. Com base nessa informação efetua a featurização sobre o *dataset* tirando partido das classes definidas no ficheiro *rdkFingerPrints* sendo todas elas possíveis de utilizar.

6 ModelBuilder

Esta classe é responsável por construir cada um dos modelos que não serão otimizados, ou seja, que têm os parâmetros já definidos ou que utilizam os parâmetros *deafult*. Esta classe recebe uma lista de modelos sobre a estrutura referida na secção AutoML. Para cada um deles, verifica o seu identificador e o campo referente aos parâmetros e com base nessa informação constrói o modelo através do método respetivo. À medida que vai inicializando os modelos, vai colocando-os numa lista. Por fim, retorna esta lista com todos os modelos inicializados.

Como exemplo de um deste métodos, mostramos o inicializador do modelo *AdaBoostClassifier*:

```
1 def ab_model_builder(self, params):
2     ab_model = AdaBoostClassifier(**params)
3     return SklearnModel(model=ab_model)
```

Como referido acima, o argumento parâmetros parte da estrutura dos modelos recebidos na inicialização do *ModelBuilder*. Como veremos mais tarde, essa estrutura pode ser vazia, daí se utilizar ***params* para inicializar os modelos. Por fim, é retornado *wrapper* sobre o modelo inicializado.

7 ModelBuilderOptimization

A classe *ModelBuilderOptimization* é responsável por construir os modelos para optimização, ou seja, os modelos para os quais o utilizador define mais do que um valor em cada parâmetro. Poderá numa primeira avaliação parecer estranho a separação entre a inicialização dos modelos para otimização dos restantes. No entanto, isto deve-se a uma característica do ***DeepMol*** explicada de seguida. De forma a realizar procura dos melhores hiper-parâmetros de um modelo no ***DeepMol***, utiliza-se, no nosso caso, a classe *GridHyperparamOpt*. Ora esta classe necessita, na sua inicialização, de uma função que construa o modelo com os parâmetros passados pelo utilizador. Como estes parâmetros não são conhecidos por nós quando o código é escrito, foi necessário encontrar uma solução que contornasse este problema. Uma solução possível poderia passar por propôr uma alteração no código. No entanto, visto que seguimos uma metodologia que passou por não alterar o código da ferramenta ***DeepMol***, sugerimos ao nosso orientador uma outra solução que foi aprovada pelo mesmo. Esta solução passa por criar inicializadores pré-definidos com os parâmetros *"default"* dos mesmos sendo apenas alterados aqueles que o utilizador definir na procura de hiper-parâmetros. Esta solução permite assim que os modelos possam receber qualquer parâmetro sem quaisquer restrições. Existe, no entanto, uma desvantagem desta solução. Sempre que um novo modelo for adicionado, teremos que adicionar também esta inicializador, o que é um processo trabalhoso embora não seja nada complicado. É por este motivo que listamos os modelos suportados.

De seguida apresenta-se um exemplo de um inicializador. Como se pode ver, este inicializador recebe como argumentos os parâmetros todos *"default"* como referido na documentação. Deste modo, quaisquer que sejam os parâmetros que o utilizador defina, este inicializador conseguirá sempre produzir o modelo.

```
def knc_model_builder(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='
                                                    minkowski',
        metric_params=None, n_jobs=None, model_dir=None, **kwargs):

    knc_model = KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights, algorithm=algorithm,
                                                          leaf_size=leaf_size, p=p,
        metric=metric, metric_params=metric_params, n_jobs=n_jobs, model_dir=model_dir, **kwargs)

    return SklearnModel(knc_model, model_dir)
```

Nota: Apesar de não seguir a metodologia, a primeira solução foi pesada, mas uma vez que a sua alteração no ***DeepMol*** poderia ser demorada, optamos pela segunda.

8 OptimizationSelector

É nesta classe que se realiza a comparação entre modelos para otimização. Ou seja, todos os modelos que tenham nos seus campos mais do que um valor, por exemplo, **"n_estimators": [50,100,200]** são fornecidos ao construtor desta classe. Ao receber como parâmetros os modelos a otimizar, esta classe apresenta os seguintes métodos:

- **prepare_dataset**: divide o *dataset* em treino/validação/teste de forma a que se possam avaliar os modelos;
- **select_models**: para cada modelo, invoca a função **select_best**;
- **select_best**: dado um modelo a otimizar, devolve o melhor modelo com base numa métrica.

O funcionamento desta classe pode ser explicada com base no exemplo da secção Exemplo. Neste caso, estamos a especificar dois modelos. Uma vez que ambos são para otimização, visto que contêm mais do que um valor por parâmetro, ambos serão passados para o construtor da classe *OptimizationSelector*. Outros modelos que tivessem parâmetros vazios ou apenas um valor por parâmetro não seriam do conhecimento desta classe. Tendo a classe sido inicializada, seriam chamados os métodos da lista explicada acima e pela ordem apresentada. É importante notar que o método *select_best* utiliza a classe *GridHyperparamOpt* para procurar os melhores modelos. Deste modo, como se pode observar no output gerado do exemplo que aqui se analisa, são gerados dois modelos finais (por featurizer) o que corresponde à melhor combinação do primeiro modelo e a melhor combinação do segundo modelo.

9 AutoML

Por fim chegamos à classe principal, em que se juntam todas as classes previamente explicadas e com mais algumas funções de apoio. Esta classe começa por carregar o *dataset* e analisar o ficheiro de *input* guardando a informação extraída em estruturas de dados apropriadas. A extração de informação está dividida em várias etapas, sendo a primeira efetuada pelo método *parse_load*, que extrai a informação relativa à secção *"load"* do *input*. De seguida, o *parse_featurize* analisa os campos relativos ao *featurizer* previamente explicados seguindo-se o *parse_models* que faz o mesmo para a informação dos modelos. É esta função que é responsável por atribuir o tipo de modelo baseado no tipo de *input* dado pelo utilizador. Se, por exemplo, o modelo tiver os parâmetros vazios, será etiquetado como *"default"*, caso especifique parâmetros com apenas um valor por parâmetro, será etiquetado como *"params"* e no caso de ter vários valores possíveis por parâmetro é equiquetado como *"opt"* de *"optimize"*. São estas etiquetas que permitem ao ModelBuilder e ModelBuilderOptimization inicializar os modelos de forma correta. Por fim encontra-se o *parse_metrics*, que extrai a informação da lista de métricas que servirão para comparar os modelos.

Após toda a informação estar em estruturas apropriadas, executa-se o método principal do módulo, o método *execute*. O *pipeline* implementado por este método é especificado de seguida:

Algorithm 1: Descrição do pipeline

```
Inicializar dataset;
for (featurizer in featurizers) do
    inicializar os modelos;
    separar modelos em otimizar/não otimizar;
    for (modelo_opt in modelos_a_otimizar) do
        otimizar (modelo_opt);
    end
    for (modelo in modelos_standard) do
        inicializar (modelo_opt);
    end
    todos_modelos = [modelos_a_otimizar, modelos_standard]
    for (modelo in todos_modelos) do
        treinar (modelo);
        avaliações = avaliar (modelo_opt);
        guardar avaliações;
    end
    gera_resultados (avaliações, todos_modelos);
end
```

Para a saída dos resultados foi implementada uma função que transforma os dados resultantes da avaliação dos modelos para cada *featurizer* num ficheiro *.csv* onde se apresentam os modelos usados bem como os seus parâmetros, o *featurizer* utilizado e os resultados de cada uma das métricas especificadas pelo utilizador.

10 Exemplo

10.1 Notebook

```python
from autoML.AutoML import AutoML
auto_ml = AutoML('teste.json')
auto_ml.execute()
```

10.2 Input (teste.json)

```
{
    "load": {
        "dataset": "preprocessed_dataset_wfoodb.csv",
        "mols": "Smiles",
        "labels_fields": "Class",
        "id_field": "ID"
    },
    "featurizers": [
        { "name" : "morgan", "type" : "default", "params":{}},
        { "name" : "rdk", "type" : "default", "params":{}}
    ],
    "models": [
        {"name": "RandomForestClassifier", "params": {"n_estimators": [50,100,200], "
           criterion": ["entropy","gini"]}},
        {"name": "SVM", "params": {"C":[0.5,0.9]}}
    ],
    "metrics": ["roc_auc_score", "precision_score", "accuracy_score"]
}
```

10.3 Output gerado

Model	Featurizer	roc_auc_score	precision_score	accuracy_score
RandomForestClassifier(n_estimators=50)	morgan	0.6417349726775956	0.6814159292035398	0.9521048109965635
SVC(C=0.9)	morgan	0.5864381520119225	0.8846153846153846	0.9518900343642611
RandomForestClassifier(criterion='entropy')	rdk	0.7063263453177757	0.6588235294117647	0.9551116838487973
SVC(C=0.9)	rdk	0.5796202738715845	0.9545454545454546	0.9521048109965635

Figura 2: Output gerado com os resultados

11 Conclusão

Em retrospetiva, o projeto desenvolvido cumpriu as nossas expectativas enquanto grupo uma vez que o módulo final é simples e intuitivo de utilizar e contém outputs informativos. O grupo tinha como objetivo final conseguir desenvolver um módulo flexível que pudesse encaixar bem na ferramenta pré-existente. Deste modo, o processo de estudo da ferramenta foi fundamental para o desenvolvimento do código.

Sendo este um projeto de desenvolvimento e integração, o primeiro para os elementos do grupo, vimos como uma grande mais valia o nosso envolvimento no mesmo uma vez que aprendemos a criar soluções baseadas numa base de código já existente, de notar, diferente de uma simples API. Neste caso, tornou-se muito mais importante ler os módulos e tentar reproduzir resultados.

É apenas de lamentar a falta de suporte para modelos keras. Era, dentro do grupo, unânime que se tratava de um objetivo importante a cumprir. No entanto, a falta de tempo e algumas dificuldades a nível de implementação fizeram com que uma solução final não ficasse incluída no módulo final dando prioridade à implementação de outras funcionalidades indispensáveis ao bom funcionamento do código. Quanto aos restantes objetivos, o grupo está contente por ter conseguido cumprir todos eles e o resultado final vai do encontro ao que foi idealizado no incício do projeto.