Isnard Gurgel Follow
Aug 20, 2020 · 22 min read

Guia de bolso para Ensemble Methods

Conceitos, intuições e dicas para você forjar um modelo melhor para os seus problemas e os seus dados.

Conteúdo produzido para a Tera

Bem vindx de volta!

Espero que vocês tenham entendido a importância, os conceitos e aprendido algumas técnicas de Feature Engineering. Isso porque depois de tanto tempo importando, explorando, limpando e preparando os dados, finalmente chegou a hora de treinar o modelo e testar a sua performance, certo?

SIM! Yeaaaah!

Só que aí vem uma outra dúvida: qual modelo escolher?

Bem, já falamos em textos anteriores sobre os tipos de problemas. Se é um problema de regressão ou de classificação. Mas isso não é o suficiente. Afinal diferentes algorítimos podem resolver o mesmo tipo de problema como uma regressão logística e um algorítimo de árvore de decisão, para um problema de classificação binário, por exemplo. Ou ainda uma regressão linear que pode ser um problema modelado tanto com uma LinearRegression quanto por uma Regressão Linear do Gradiente Descendente Estocástico ou SGDRegressor.

Saber responder a estas perguntas exige experiência e conhecimento sobre o funcionamento e as peculiaridades de cada modelo ou testar e avaliar o desempenho de cada algoritmo para o seu dataset em específico. O sklearn facilita muito o processo de treinar, testar e avaliar as métricas desejadas em diferentes modelos. E isso será suficiente na maioria das vezes.

> *Mas e se existisse uma abordagem mais poderosa que usa técnicas de combinar, agrupar, somar e empilhar algoritmos de forma que seja possível criar soluções mais robustas, precisas e com um menor viés?*
> *Isso existe! Estamos falando de métodos de ensamble. É sobre eles que vamos falar neste*

> *Modelos de Ensemble em machine learning combinam decisões de diversos modelos para melhorar a performance geral.*

Mas para entender o porquê de se criar um ensemble de modelos é preciso entender a motivação por trás desse conceito. A verdade é que todo algoritmo tem suas limitações. Elas podem ser desde a estratégia que usam para aprender, a o grau de complexidade que são capazes de atingir, a quantidade de observações necessárias para aprender sem que haja um overfitting e por aí vai. Criar um algoritmo único que seja generalizável para praticamente qualquer tarefa é algo custoso e complexo. Pensando nisso, as técnicas de ensemble combinam diferentes modelos e suas formas de aprendizados e combina seus outputs para respostas muito mais robustas e estáveis, uma vez que, idealmente, as fraquezas de um algoritmo são compensadas pelo(s) outro(s).

Pensando de uma forma mais técnica, a principal causa de erro no aprendizado dos modelos são causadas por **ruído**, **viés** e **variância**. A partir disso, as técnicas de ensembles podem ser SIMPLES (quando não desejamos atacar uma causa específica) ou AVANÇADAS, reduzindo de forma mais específica a **variância** (bagging), o **viés** (boosting) ou **melhorando as predições** por meio de combinações de algorítimos e n camadas de complexidade (stacking).

Implementação de métodos de ensembles

Vamos abordar algumas das técnicas simples e avançadas de ensembles passando por seus conceitos, exemplos e implementação de código.

Técnicas Simples de Ensamble

- **Método de votação ou hard voting:**
  Nessa técnica, múltiplos modelos são usados **apenas em problemas de classificação**, para fazer a previsão de cada data point. As previsões para cada modelo são consideradas como um voto separado. A predição que receber a maior quantidade de votos, ou a **moda**, se pensarmos em estatística, é a escolhida. Transformando esse raciocínio em código teríamos:

```
model3.fit(x_train,y_train)

pred1=model1.predict(x_test)
pred2=model2.predict(x_test)
pred3=model3.predict(x_test)

final_pred = np.array([])
for i in range(0,len(x_test)):
    final_pred = np.append(final_pred, mode([pred1[i], pred2[i],
pred3[i]]))
```

Ao mesmo tempo, temos o módulo VotingClassifier do sklearn que torna este processo muito mais simples.

```
from sklearn.ensemble import VotingClassifier
model1 = LogisticRegression(random_state=1)
model2 = DecisionTreeClassifier(random_state=1)
model3 = KNeighborsClassifier(random_state=1)

model = VotingClassifier(estimators=[
                                     ('lr', model1),
                                     ('dt', model2),
                                     ('knn', model3)],
                                     voting='hard')
model.fit(x_train,y_train)
model.accuracy_score(y_test, model.predict(X_test))
```

- **Método de média ou soft voting**
  Já nesta técnica, vamos calcular as médias das da previsões de cada modelo para cada data point. Esse método **pode ser usado tanto para problemas de regressão quanto de classificação**. Porém enquanto ele pode fazer previsões em problemas de regressão, ele prevê apenas probabilidades para problemas de classificação.
  Representando este raciocínio com código, temos:

```
model1 = tree.DecisionTreeClassifier()
model2 = KNeighborsClassifier()
model3= LogisticRegression()
```

```
pred2=model2.predict_proba(x_test)
pred3=model3.predict_proba(x_test)

finalpred=(pred1+pred2+pred3)/3
```

Ou mais uma vez, podemos usar o VotingClassifier ou VotingRegressor do sklearn para agilizar o processo:

```
# Instantiate the individual models
clf_knn = KNeighborsClassifier(5)
clf_dt = DecisionTreeClassifier()
clf_lr = LogisticRegression()

# Create an averaging classifier
clf_avg = VotingClassifier(
     estimators=[
        ('knn', clf_knn),
        ('dt', clf_dt),
        ('lr', clf_lr)],
     voting='soft',
     weights=[1, 2, 1])

clf_avg.fit(x_train,y_train)
clf_avg.accuracy_score(y_test, clf_avg.predict(X_test))
```

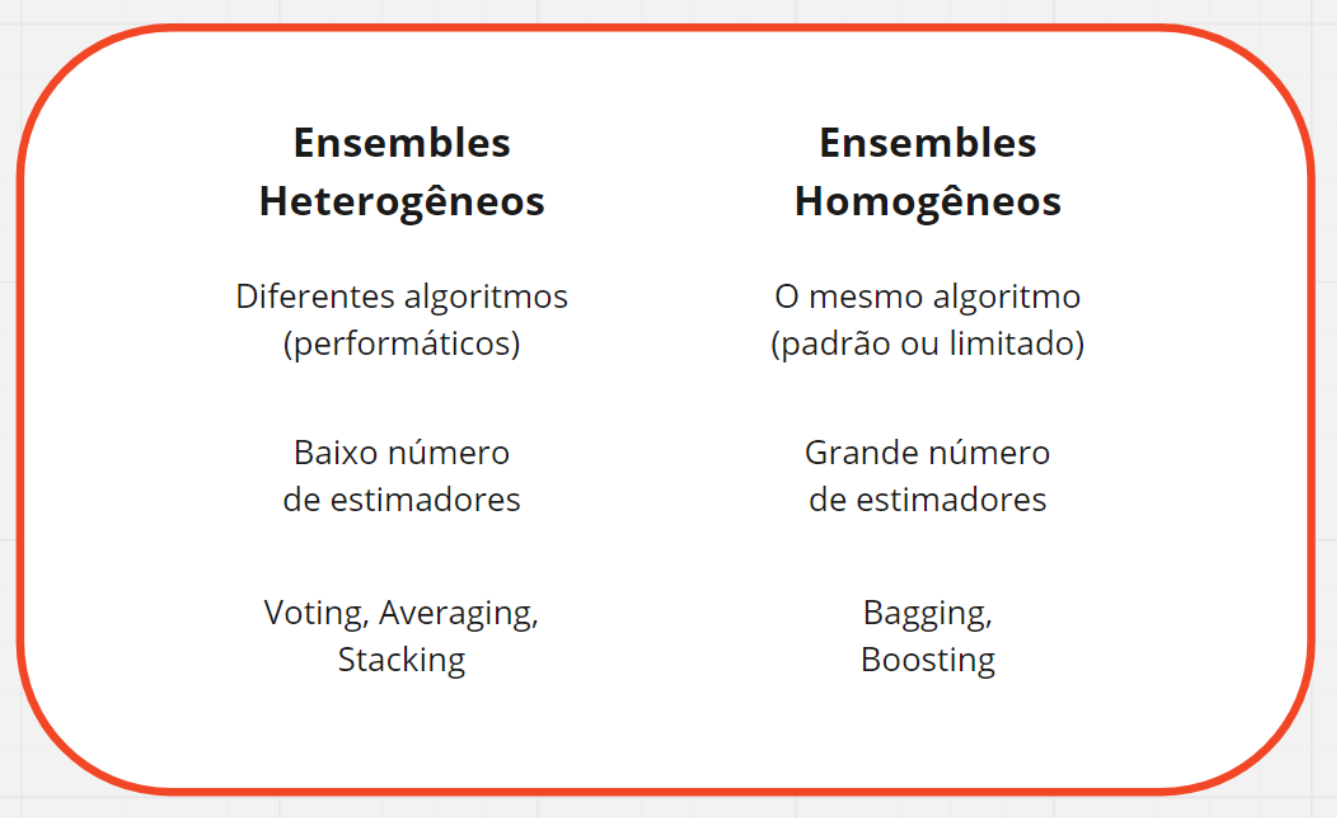
Perceba que para ter um *Averaging Regressor* ou um *Averaging Classifier*, é necessário passar o argumento voting, seguido do parâmetro ‘soft’, e, note ainda, que este modo permite adicionar pesos a cada um dos modelos. Isto vai criar não apenas uma média, mas uma média ponderada que pode penalizar ou priorizar os modelos instanciados. Para isso, devemos passar uma lista no argumento weights. Ok! Agora vamos para um outro nível.

Técnicas Avançadas de Ensamble

Ensembles, como vocês já puderam perceber, é o conceito de agrupar ou combinar algoritmos ou modelos para resolver uma tarefa. Dito isto, basicamente existem duas abordagens em técnicas de ensembles:
ensemble de modelos heterogêneos e ensemble de modelos homogeneos.
Um fator interessante que diferencia essas abordagens, além do óbvio de que uma usa

A abordagem heterogênea é a mais intuitiva. Nela, cada modelo, de forma individual, já é ajustado para ter a melhor performance possível. Cada modelo analisa os dados e dá o seu veredicto. **Podemos pensar no conjunto de modelos desta abordagem como um time de especialistas.** Estamos falando de alguns métodos que já vimos como *Voting* e *Averaging,* e, de *Stacking,* que ainda vamos ver.

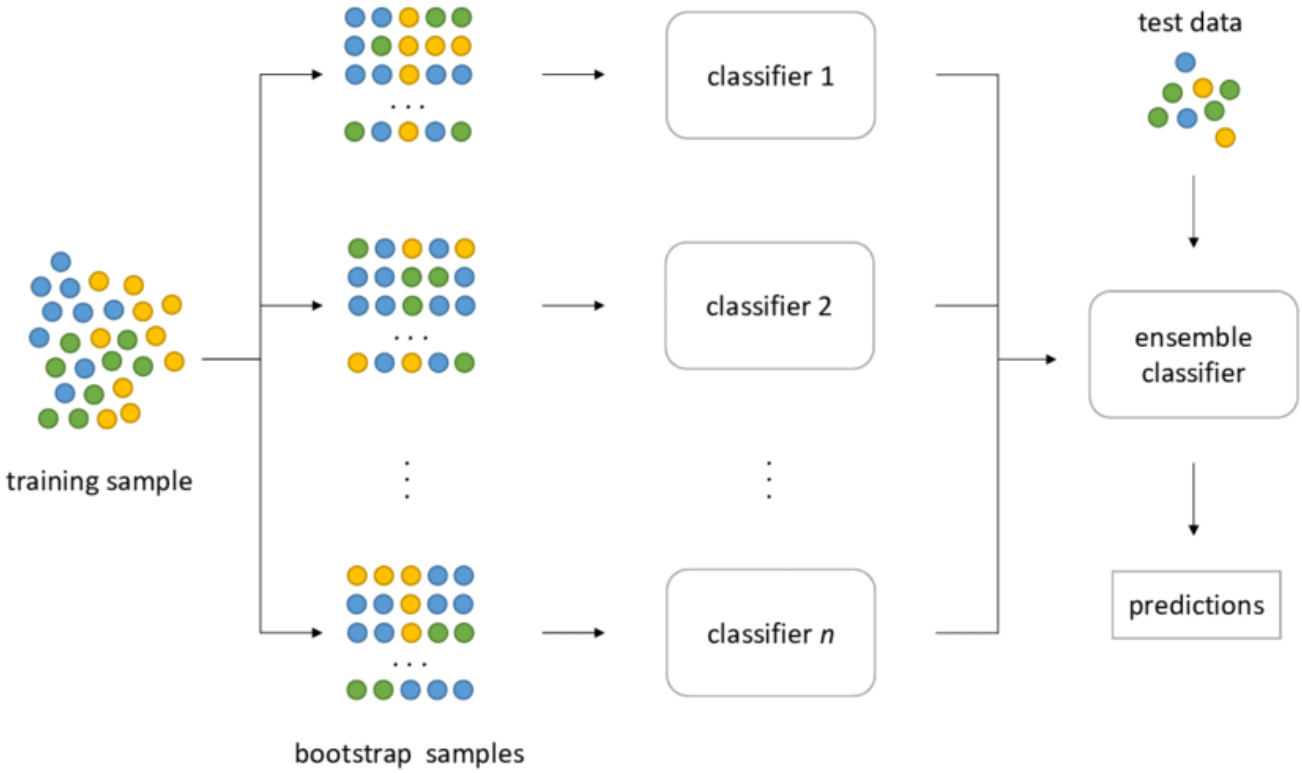
Na abordagem homogênea, temos princípios quase contra-intuitivos, pensando que no final das contas, queremos sempre uma melhor performance. Isto porque nas técnicas que usam algoritmos homogêneos, ao invés de ajustar ou fazer um *tunning* do modelo base ou inicial, criamos limitadores para que eles tenham uma performance baixa, ou seja, apenas um pouco acima de uma previsão aleatória (acima de 50% para uma previsão binária, por exemplo). Para que se chegue a um resultado performático precisamos de muitos modelos trabalhando na mesma tarefa. **Podemos definir esta abordagem como o poder do trabalho em equipe.** Estamos falando dos métodos de bagging e de boosting.



Quadro-resumo das abordagens de Ensembles.

Bagging

elementos usados para criar os novos datasets menores, são repostos ao final de cada rearranjo. É exatamente isso o que temos: uma junção de bootstraps.
Em outras palavras estamos falando de treinar n número de modelos individuais, de forma paralela, onde cada modelo é treinado em um sub grupo dos dados determinado randomicamente. Em uma imagem seria:



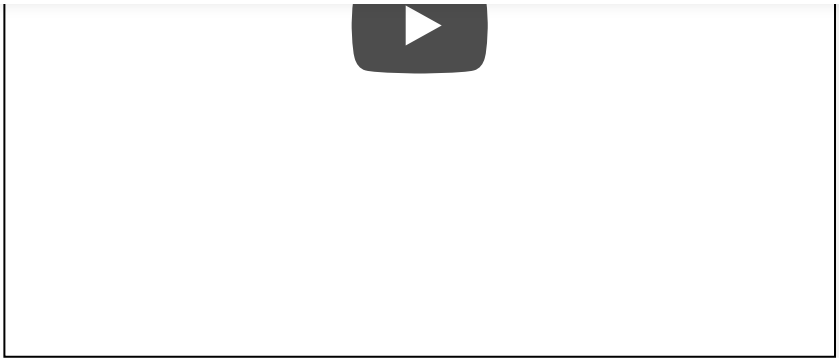
fonte: https://www.researchgate.net/figure/The-bagging-approach-Several-classifier-are-trained-on-bootstrap-samples-of-the-training_fig4_322179244

Falando assim não parece muito promissor. Mas se você entender o conceito que embasa esse método, talvez ele comece a fazer mais sentido.
Estamos falando sobre a teoria da sabedoria das multidões, ou *the wisdom of the crowd*.

> *A combinação das predições dos modelos individuais é superior as predições individuais feitas por um único modelo.*

Assista ao vídeo e entenda um pouco melhor sobre ela.

BBC - The Code - The Wisdom of the Cro...



The Wisdom of the Crowd

Em poucas palavras, este método gera uma massa crítica de estimadores que vão receber os dados e , como se fossem pessoas que chutam palpites (um pouco melhor que aleatórios) utilizam a média todos os outputs como uma resposta precisa, igual a técnica que a gente acabou de ver no vídeo.
Dentro de algoritmos de Bagging temos basicamente duas técnicas:
1. Bagging Meta-Estimator
2. Random Forest

> *Para exemplificar o funcionamento dos modelos e suas performances, vamos usar o dataset do desafio Loan Prediction Problem do site Analytics Vidhya.*
> *Baixe clicando aqui.*

```
#tratamentos aplicados aos dados:
#preenchimento de missing values
df['Gender'].fillna('Male', inplace=True)

#separar o dataset em treino e test
train, test = train_test_split(df, test_size=0.3, random_state=0)

x_train=train.drop('Loan_Status',axis=1)
y_train=train['Loan_Status']
```

```
x_test=pd.get_dummies(x_test)
```

Bagging Meta-Estimator

Bagging meta-estimator é um algoritmo que pode ser usando tanto para problemas de classificação (BaggingClassifier) quanto para problemas de regressão(BaggingRegressor). Em geral, estes são os passos seguidos pelo algoritmo meta-estimator :

1. Subsets aleatórios são criados a partir do dataset original(Bootstrapping).
2. Cada subset (recorte do dataset original) possui todas as features.
3. Um algoritmo base (escolhido pelo usuário) é “fitado” em cada um desses datasets menores.
4. As predições de cada modelo são combinadas para se chegar ao resultado final.

Os códigos base para os Meta-Estimators são:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import BaggingRegressor

BaggingClassifier(base_estimator=None, n_estimators=10,
max_samples=1.0, max_features=1.0, bootstrap=True,
bootstrap_features=False, oob_score=False, warm_start=False,
n_jobs=None, random_state=None, verbose=0)class

BaggingRegressor(base_estimator=None, n_estimators=10,
max_samples=1.0, max_features=1.0, bootstrap=True,
bootstrap_features=False, oob_score=False, warm_start=False,
n_jobs=None, random_state=None, verbose=0)
```

Principais parâmetros usados nos algoritmos:

- **base_estimator**:
  Define o estimador base para *fittar* nos subsets aleatórios do datase
  Se nenhum modelo for passado, o default é uma DecisionTree.


muito grande pode demorar muito para rodar e um número muito pequeno pode não levar a um resultado satisfatório.

- **max_samples**:
  Este parâmetro controla o tamanho máximo de linhas dos subsets.
  Estamos falando do número máximo de amostras para treinar cada estimador base.

- **max_features**:
  Este parâmetro controla o número máximo de features dos subsets
  Estamos falando do número máximo de colunas para treinar cada estimador base.

- **n_jobs**:
  O número de jobs a rodar em paralelo.
  Defina o número de cores igual ao do cores no seu processador.
  Bem melhor do que saber isso de cor é passar o argumento '-1' e modelo usará automaticamente todos os cores do seu computador.

- **random_state**:
  Especifica o método do random split (dentro do modelo, não no train_test_split).
  Quando o random_state é igual para dois ou mais modelos, a seleção randômica dos modelos será a mesma.
  Este parâmetro é útil para comparar diferentes configurações dos modelos mas mantendo a mesma separação dos dados.

Exemplo de implementação no dataset Loan Prediction Problem

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
model = BaggingClassifier(DecisionTreeClassifier(random_state=1))
model.fit(x_train, y_train)
model.score(x_test,y_test)
>>0.75135135135135134
```

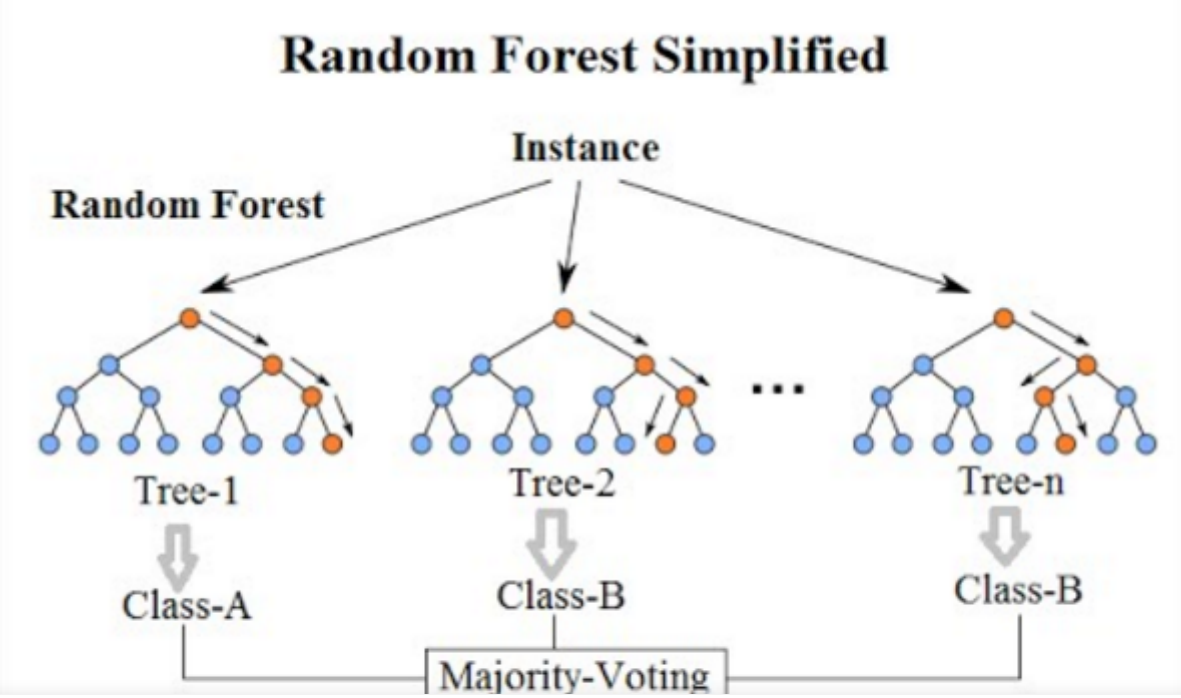
Baseado em tudo o que você aprendeu até aqui na sua jornada, como você melhoraria este score? Te desafio a postar no grupo, o seu score e compartilhar com os colegas

Random Forest é outro algoritmo de machine learning que usa a técnica de bagging. Como o próprio nome sugere, os estimadores base da Random Forest são sempre DecisionTrees. Enquanto o Bagging Meta-Estimator pode usar todas as features para treinar cada estimador, na RandomForest, um subset randômico de features é selcionado e usado para decidir qual o melhor *split* a cada *node* de cada árvore de decisão.

Como resultado dessa estratégia, o viés da RandomForest aumenta um pouco mas extraindo a média de ávores não-correlacionadas a variância diminui, resultando em um modelo com uma boa performance geral.

Vamos ao passo-a-passo do que faz um modelo RandomForest:

1. Subsets aleatórios são criados a partir do dataset original(Bootstrapping).
2. A cada *node* dentro de cada DecisionTree, apenas uma quantidade randômica de features são consideradas para decidir o melhor *split*.
3. Um modelo de DecisionTree é " fittado" em cada um dos subsets.
4. A predição final é calculada a partir da média das predições de todas as árvores de decisão.



Fonte: https://medium.com/@vigneshmadanan/ensemble-learning-explained-part-2-498ab87788b5

Confira a base para codar RandomForests tanto Regressor quanto Classifier:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestRegressor

RandomForestClassifier(n_estimators='warn', criterion='gini',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False,
n_jobs=None, random_state=None, verbose=0, warm_start=False,
class_weight=None)

RandomForestRegressor(n_estimators=100, *, criterion='mse',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False,
n_jobs=None, random_state=None, verbose=0, warm_start=False,
ccp_alpha=0.0, max_samples=None)
```

Principais parâmetros usados em RandomForests:

- **n_estimators:**
  É o número de DecisionTrees que devem ser criadas.
  O número de estimadores deve ser pensado com cuidado, uma vez que um número muito grande pode demorar muito para rodar e um número muito pequeno pode não levar a um resultado satisfatório.

- **criterion**:
  Define a função a ser usada para fazer o *splitting*
  A função escolhida vai mensurar a qualidade do split para cada feature Regressão: default = mse
  Classificação: default = gini

- **max_features** :
  Define o número máximo de features permitido para calcular os *splits* em cada

- **max_depth**:
  Um RandomForest tem várias árvores de decisão. Este parâmetro define a profundidade máxima de cada árvore.

- **min_samples_split:**
  Usado para definir o número mínimo de amostras necessárias em uma *node* antes que ela possa realizar uma tentativa de *splitting*.
  Se o número de amostras for menor que o estabelecido pelo min_samples_split, o split não ocorrerá, e o node se torna uma folha.

- **min_samples_leaf:**
  Este parâmetro define o número mínimo de amostras necessárias nas folhas ou *leaf nodes* de cada árvore de decisão.
  Um número muito baixo neste parâmetro faz com que o modelo tenda a capturar mais ruído dos dados de treino.

- **max_leaf_nodes:**
  Este parâmetro especifica o número máximo de folhas ou *leaf nodes* em cada árvore de decisão. A árvore para de fazer *splits* quando o número de leaf nodes se torna igual ao estabelecido pelo max_leaf_nodes.

- **n_jobs**:
  Já foi abordado em Meta-Estimator.
  Escolha o valor "-1" se você quiser usar todos os cores da sua máquina.

- **random_state**:
  Também já falamos sobre isso em Meta-Estimator.
  Escolha o mesmo número em diferentes configurações para comparar a performance dos modelos sem se preocupar com a forma de separação dos dados dentro do modelo.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)
```

```
clf.score(x_test,y_test)
>>???

#Aproveitando, vamos avaliar por uma nova métrica
# Acurácia do modelo é a frequência com que o modelo acertou na
classificação

y_pred=clf.predict(X_test)
print(metrics.accuracy_score(y_test, y_pred))
>>???
```

Desafio:
Seja o primeiro a comentar no grupo sobre o score e a acurácia da RandomForest. No que ela foi diferente ou não da Meta-Estimator? Por que isso aconteceu?

Boosting

Enquanto os modelos de bagging aprendem com a sabedoria das multidões, os modelos de Ensemble que usam técnicas de boosting possuem um processo de aprendizado que é o mais natural possível para nós. Estamos falando de um aprendizado gradual ou sequencial. Intuitivamente é muito similar à forma como nós, seres humanos aprendemos. Imagine que a cada tentativa de aprender algo novo, nós recebemos um feedback sobre os erros e acertos em relação à tentativa. Então corrigimos as nossas novas tentativas a partir do feedback de cada ação, ou seja, nós aprendemos de forma iterativa.

Só que um pouco diferente de nós, nos algoritmos de boosting, ao invés de ter o mesmo modelo sendo corrigido a cada iteração, temos um novo modelo a cada feedback, para tentar corrigir os erros do modelo anterior.

Comparativo sobre formas de aprendizado entre bagging e boosting

Um dos maiores perigos dos algoritmos de boosting é que eles acabem aprendendo com o ruído nos dados, o que pode levar o modelo a um overfitting.

Exemplos de Algoritmos de Boosting:

- **AdaBoost**
- **GBM**
- **XGBM**
- Light GBM
- CatBoost

AdaBoost

Adaptive boosting or AdaBoost é um dos mais simples algoritmos de boosting. Geralmente são usadas DecisionTrees como estimadores base. Neste algoritmo

incorretamente preditas e força o modelo seguinte a prever esses valores de forma correta.

Vamos aos steps que o AdaBoost performa:

1. Inicialmente são dadas a todas as observações do dataset pesos iguais.
2. Um modelo é criado sob um subset dos dados.
3. Seguindo este princípio, predições são feitas com o dataset inteiro.
4. Os erros são calculados comparando as predições com os valores reais.
5. Enquanto cria o próximo modelo, é dado um peso mairo aos data points que foram previstos de forma incorreta.
6. Os pesos podem ser determinados usando o valor do erro. Por exemplo, quanto maior o erro, maior é o peso atribuído a observação.
7. Este processo é repetido até que a função do erro não muda mais, ou o limite máximo de estimadores é alcançado.

Código:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import AdaBoostRegressor

AdaBoostClassifier(base_estimator=None, *, n_estimators=50,
learning_rate=1.0, algorithm='SAMME.R', random_state=None)

AdaBoostRegressor(base_estimator=None, *, n_estimators=50,
learning_rate=1.0, loss='linear', random_state=None)
```

Principais Parâmetros:

- **base_estimators**:
  Para classificação:Default= DecisionTreeClassifier(max_depth = 1)
  Regressão: Default = DecisionTreeRegression(max_depth = 3)

É a função usada para atualizar os pesos dos novos estimadores.
Por default temos o argumento *linear*, mas também pode ser *square* ou *exponential*.

- **n_estimators:**
  Define o número base de estimadores.
  O valor default é 50, mas pode ser aumentado para obter uma melhor performance.
  Obs.: Se houver um *fit* perfeito (um estimador atingir um erro maior que 50%), nenhum novo estimador é criado.

- **learning_rate:**
  Este parâmetro controla a contribuição dos estimadores na combinação final. Por default é de 1.0
  É importante ter em mente que existe um trade-off entre learning_rate e n_estimators.

- **random_state** :
  Manter o mesmo número para garantir consistência em teste entre modelos

Exemplo de implementação no dataset Loan Prediction Problem

```
model = AdaBoostClassifier(random_state=1)
model.fit(x_train, y_train)
model.score(x_test,y_test)
0.81081081081081086
```

GradientBoosting

Gradient Boosting or GBM é outro algoritmo de machine learning que funciona para problemas de regressão e de classificação. GBM usa a técnica de boosting, combinando diferentes estimadores entre weak lerners e strong learners. No GradientBoosting são usadas RegressionTrees como estimadores base, cada árvore da sequência é construída a partir dos erros calculados pelas árvores anteriores.

Uma peculiaridade deste modelo é que apenas o estimador inicial é “fitado” nos dados. Todos os modelos subsequentes são “fitados” nos erros dos anteriores até que o erro

```
GradientBoostingClassifier(loss='deviance', learning_rate=0.1,
n_estimators=100, subsample=1.0, criterion='friedman_mse',
min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_depth=3,
min_impurity_decrease=0.0, min_impurity_split=None, init=None,
random_state=None, max_features=None, verbose=0,
max_leaf_nodes=None, warm_start=False, presort='auto',
validation_fraction=0.1, n_iter_no_change=None, tol=0.0001)

GradientBoostingRegressor(*, loss='ls', learning_rate=0.1,
n_estimators=100, subsample=1.0, criterion='friedman_mse',
min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_depth=3,
min_impurity_decrease=0.0, min_impurity_split=None, init=None,
random_state=None, max_features=None, alpha=0.9, verbose=0,
max_leaf_nodes=None, warm_start=False, presort='deprecated',
validation_fraction=0.1, n_iter_no_change=None, tol=0.0001,
ccp_alpha=0.0)[source]¶
```

Principais Parâmetros:

- **min_samples_split**
  Define o número mínimo de amostras ou observações que são necessáras para que haja uma tentativa de *split*.
  Esse argumento é usado para controlar o overfitting. Valores altos impedem o modelo de aprender relações muito específicas à amostras em particular selecionadas pela árvore.

- **min_samples_leaf**
  Determina o número mínimo necessário de amostras em uma folha Geralmente, se tivermos classes desbalanceadas, valores baixos devem escolhidos porque os valores serão baixos nas regiões em que as classes minoritárias serão a maioria.

- **min_weight_fraction_leaf**
  Similar ao min_sample_leaf, mas ao invés do número de observações, deve ser passado como uma fração do número total de observações.

- **max_depth**
  A profundidade máxima de cada árvore. Usado para controlar o over-fitting. Se

- **max_leaf_nodes**
  Número máximo de folhas em uma árvore. Pode ser usado no lugar de max_depth. Uma vez que árvores binárias são criadas, a profundidade de “n” irá produzir no máximo 2^n folhas.
  Se este parâmetro for utilizado, o GBM vai ignorar o max_depth.

- **max_features**
  O número de features a serem consideradas enquanto se busca o melhor split. Isto será selecionado de forma randômica.
  Como regra geral, a raiz quadrada do total de features funciona bem, mas pode testar entre 30–40% do número total de features.
  Valores altos nesse parâmetro podem levar a overfitting, mas geralmente depende muito do caso.

Exemplo de implementação no dataset Loan Prediction Problem

```
model= GradientBoostingClassifier(learning_rate=0.01,random_state=1)
model.fit(x_train, y_train)
model.score(x_test,y_test)
0.81621621621621621
```

XGBoost

XGBoost (extreme Gradient Boosting) é uma implementação avançada de algoritmos de gradient boosting. XGBoost se provou ser um algoritmo de ML altamente eficiente sendo amplamente usado em competições de machine learning e hackathons. XGBoost tem um alto poder preditivo e é quase 10x mais rápido que qualquer outra técnica de gradient boosting. Nele também é encontrado uma variedade de regularização que reduz o overfitting e melhora a performance geral. Por causa disso, também é conhecido como uma técnica de **boosting regularizada**.

Vejamos como o XGBoost é melhor em comparação a outras técnicas:

1. **Regularização:**
   - A implementação do GBM não tem regularização como o XGBoost.

- O XGBoost também é compatível com implementação em Hadoop.

3. High Flexibility:

- XGBoost permite que o usuário customize objetivos e critério de avaliação de forma otimizada, adicionando toda uma nova dimensão ao modelo.

4. Handling Missing Values:

- O XGBoost tem uma rotina embutida para lidar com dados faltantes.

5. Tree Pruning:

- O XGBoost faz *splits* até o número especificado no the max_depth e depois começa a podar a árvore de trás para frente, removendo splits que não trazem um ganho positivo para o modelo.

6. Built-in Cross-Validation:

- O XGBoost permite que o usuário rode uma cross-validation a cada iteração do processo de boosting e portanto é fácil obter o número ideal exato de iterações de boosting em uma única execução.

Code:

```
XGBClassifier(max_depth=None, learning_rate=None, n_estimators=100,
verbosity=None, objective=None, booster=None, tree_method=None,
n_jobs=None, gamma=None, min_child_weight=None, max_delta_step=None,
subsample=None, colsample_bytree=None, colsample_bylevel=None,
colsample_bynode=None, reg_alpha=None, reg_lambda=None,
scale_pos_weight=None, base_score=None, random_state=None,
missing=nan, num_parallel_tree=None, monotone_constraints=None,
interaction_constraints=None, importance_type='gain', gpu_id=None,
validate_parameters=None, **kwargs)

XGBRegressor(max_depth=None, learning_rate=None, n_estimators=100,
verbosity=None, objective=None, booster=None, tree_method=None,
n_jobs=None, gamma=None, min_child_weight=None, max_delta_step=None,
subsample=None, colsample_bytree=None, colsample_bylevel=None,
colsample_bynode=None, reg_alpha=None, reg_lambda=None,
scale_pos_weight=None, base_score=None, random_state=None,
missing=nan, num_parallel_tree=None, monotone_constraints=None,
interaction_constraints=None, importance_type='gain', gpu_id=None,
validate_parameters=None, **kwargs)
```

- **nthread**
  Isso é usado para processamento paralelo. O número de núcleos no sistema deve ser inserido. Se você deseja executar em todos os núcleos, não insira esse valor. O algoritmo irá detectá-lo automaticamente.

- **eta**
  Análogo à taxa de aprendizado em GBM. Torna o modelo mais robusto diminuindo os pesos em cada etapa.

- **min_child_weight**
  Define a soma mínima de pesos de todas as observações necessárias em uma *child*. Usado para controlar o overfitting. Valores mais altos impedem um modelo de aprender relações que podem ser altamente específicas para a amostra específica selecionada para uma árvore.

- **max_depth**
  É usado para definir a profundidade máxima. Uma profundidade maior permitirá ao modelo aprender relações muito específicas para uma amostra específica.

- **max_leaf_nodes**
  Número máximo de folhas em uma árvore. Pode ser usado no lugar de max_depth. Uma vez que árvores binárias são criadas, a profundidade de “n” irá produzir no máximo 2^n folhas.
  Se este parâmetro for utilizado, o GBM vai ignorar o max_depth.

- **gamma**
  Um *node* é dividido apenas quando a divisão resultante fornece uma redução positiva na função de perda. Gamma especifica a redução de perda mínima necessária para fazer uma divisão. Torna o algoritmo conservador. Os valores podem variar dependendo da função de perda e devem ser ajustados

- **subsample**
  O mesmo que a subamostra de GBM. Indica a fração de observações a serem amostradas aleatoriamente para cada árvore. Valores mais baixos tornam o algoritmo mais conservador e evitam o ajuste excessivo, mas valores muito pequenos podem levar a um ajuste insuficiente.


Exemplo de implementação no dataset Loan Prediction Problem

```
import xgboost as xgb
model=xgb.XGBClassifier(random_state=1,learning_rate=0.01)
model.fit(x_train, y_train)
model.score(x_test,y_test)
0.82702702702702702
```

Desafio:
Qual modelo performou melhor até agora?
Será que conseguimos aprimorar esse score ainda mais?

CatBoost & LightBoost

Não vamos entrar no detalhe de código e implementação desses algoritmos porque eles são uma espécie de variação do XGBoost, mas com algumas propriedades interessantes. Nós explicamos essas características e deixamos alguns links para saber mais tanto na descrição, quanto ao final deste texto.

CatBoost

Manipular variáveis categóricas é um processo tedioso, especialmente quando você tem um grande número dessas variáveis. **Quando suas variáveis categóricas têm muitos rótulos (ou seja, possuem uma alta cardinalidade), fazer um one-hot-encoding neles aumenta exponencialmente a dimensionalidade** e torna-se realmente difícil trabalhar com o conjunto de dados.

O CatBoost pode lidar automaticamente com variáveis categóricas e não requer amplo pré-processamento de dados, como outros algoritmos de machine learning. **Aqui** você encontra um artigo que explica o CatBoost em detalhes.

Light GBM

Antes de discutir como o Light GBM funciona, vamos primeiro entender por que precisamos desse algoritmo quando temos tantos outros (como os que vimos acima). **O Light GBM supera todos os outros algoritmos quando o conjunto de dados é extremamente grande.** Comparado aos outros algoritmos, o Light GBM leva menos

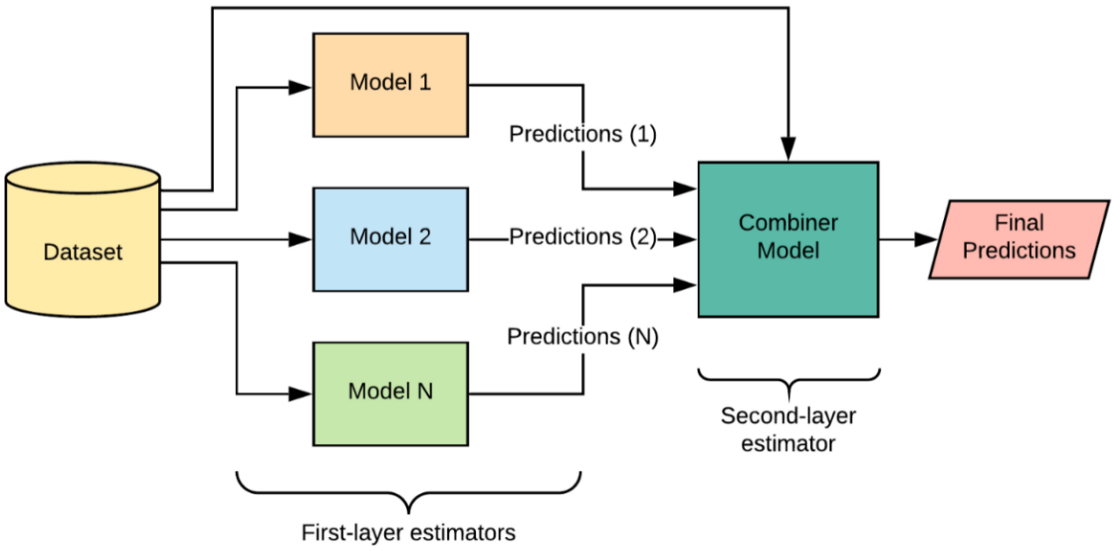
com uma abordagem de padrão *level-wise*. Você pode ler mais sobre Light GBM e seus comparativos **neste** artigo.

Stacking

A intuição por trás do método de stacking é reunir os melhores algoritmos e suas habilidades especiais, todos super ajustados em suas melhores performances e fazê-los realizar as tarefas em que eles performam melhor. No final um modelo "combinador" que conhece as forças e as fraquezas de todos os modelos recebe tanto os imputs de todos os modelos quanto processa os dados do dataset original para combinar tudo isso e gerar o output final.
Ao pensar em tudo isso, uma imagem nos vem à mente:

O Stacking é o Avengers dos métodos de Ensembles

Wow!!! Incrível… então vamos para mais um sklearn.AlgumaCoisa?
Não! Infelizmente o scikit-learn não possui uma implementação de stacking.



Exemplo de uma arquitetura de um Stacking

Para termos um entendimento geral no funcionamento,
Então vamos tomar dois caminhos:

1. Criar o nosso próprio stacking manualmente.
2. Usar uma outra ferramenta para auxiliar nessa construção.

Criando o nosso próprio stacking

Vamos montar o nosso próprio stacking em 5 passos:

1- Preparar o dataset

Primeiro vamos usar técnicas conhecidas pandas para selecionar as features que queremos usar como input para os nossos modelos da primeira camada:

```
#selecionar as features
feat_selecionadas = ['feat1', 'feat2', 'feat3', ..., 'featN']
features = df.iloc[:, feat_selecionadas]

target = df['target_col']

#Limpar e transformar os dados (caso necessário)
#Data Cleaning
#Por exemplo: preencher com 0 valores NA
```

```
#Separar os dados em treino(60%) e teste(40%)
X_train, X_test, y_train, y_test = train_test_split(
                                                    features,
                                                    target,
                                                    test_size=0.4,
                                                    random_state=42)
```

2- Construir a primeira camada de estimadores

Agora vamos instanciar e "fitar" a primeira camada de estimadores ao dados de treino. É importante ressaltar que todos os estimadores são treinados com as mesmas combinações de input de features e de targets.

```
#Esses modelos podem ser outros algoritmos, são exemplos
# 1. Gaussian Naive Bayes classifier
clf_nb = GaussianNB() clf_nb.fit(X_train, y_train)

# 2. 5-nearest neighbors classifier usando algoritmo 'Ball-Tree'
clf_knn = KNeighborsClassifier(n_neighbors=5,
                               algorithm= 'ball_tree')
clf_knn.fit(X_train, y_train)
```

3- Fundir as predições da primeira camada de estimadores com o dataset original

Vamos calcular as predições dos datasets de treino baseadas nos algoritmos da primeira camada de estimadores e na sequência vamos adicionar essas predições ao dataset original

```
# Predições feitas pela primeira camada de estimadores no X_train
pred_nb = clf_nb.predict(X_train)
pred_knn = clf_knn.predict(X_train)
```

```
# criando um DataFrame com as predições
```


```
# concatenar X_train com o pred_df
X_train_2nd = pd.concat([X_train, pred_df], axis=1)
```

4- Construir a segunda camada com o meta-estimador

Agora vamos criar a segunda camada. Qualquer modelo pode ser escolhido para essa camada. Modelos mais básicos como LogisticRegression ou LinearRegression costumam funcionar bem. Siga a implementação instanciando o modelo e fitando nos dados de treino (com as predições da primeira camada):

```
# Vamos instanciar o estimador da segunda camada
# Example: a Logistic Regression classifier
clf_stack = LogisticRegression()

# treinando a segunda camada com o segundo training set
clf_stack.fit(X_train_2nd, y_train)
```

5- Usar o stacked ensemble para fazer predições

Neste último step vamos usar o modelo que construímos para fazer predições usando dados inéditos ao modelo como os datasets de teste.
Precisamos repetir o passo 3 antes de finalmente obter as nossas predições finais:

Obter as predições da primeira camada de estimadores:

```
# Predições da pela primeira camada no X_test
pred_nb = clf_nb.predict(X_test)
pred_knn = clf_knn.predict(X_test)

# adicionar a um DataFrame
pred_df = pd.DataFrame({
    'pred_nb': pred_nb,
    'pred_knn': pred_knn
})
```


```
# concatenar X_test com o pred_df
X_train_2nd = pd.concat([X_test, pred_df], axis=1)
```

Obter a predição final:

```
#Aleluia!!!
pred_stack = clf_stack.predict(X_test_2nd)
```

Desafio:
Percebam que faz muito sentido criar uma função para o passo 3.
Crie uma função que automatiza o passo 3 para passar tanto os dados de treino quanto os dados de teste. Se estiver ousado, crie uma função que automatiza todo o processo.
Não vou dar dicas! =]]
Obs.: Vocês podem usar a turma do slack como comunidade.

Criando stacking ensembles com MLXTEND

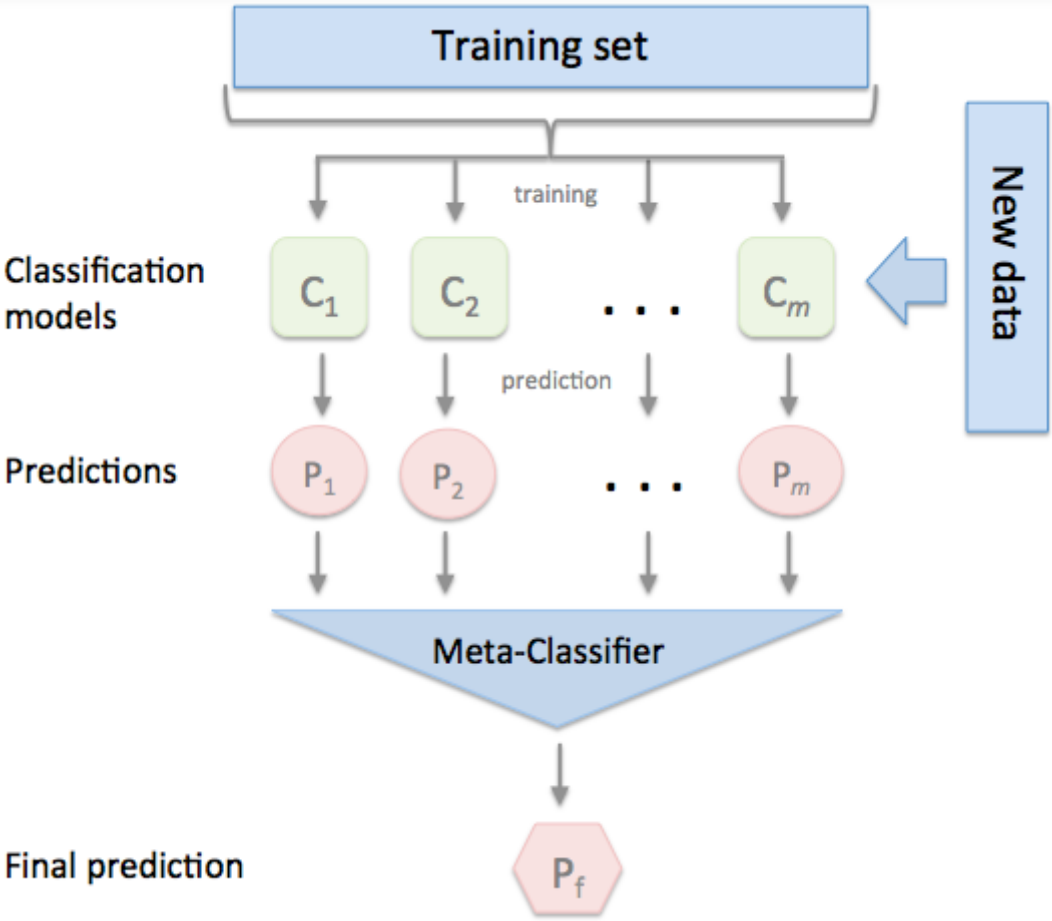
MLXTEND é uma síntese de **M**achine **L**earning E**xten**sions, uma biblioteca de Python que contém muitas utilidades e ferramentas para machine learning e data science. Acessem http://rasbt.github.io/mlxtend/ para documentação e informações sobre instalação.

Algumas utilidades e ferramentas:
- Feature selection
- Ensemble Methods
- Visualização
- Avaliação de modelos

Além disso, essa biblioteca é compatível com os estimadores do Scikit-learn!

Arquitetura e características

A arquitetura do modelo implementado pela mlxtend é similar, porém um pouco diferente à que criamos anteriormente.



Fonte: Documentação MLXTEND

Isso impacta em características diferentes:

- Os estimadores são treinados nas features completas
- O meta-estimador é treinado usando as predições da primeira camada como únicos meta-features
- O meta-estimador pode ser treinado tanto com labels quanto com probabilidades no target da primeira camada. A possibilidade de usar probabilidades além de labels pode te permitir resolver problemas mais complexos.

Implementação

Para stackings de classificação:

```
from mlxtend.classifier import StackingClassifier

# Instanciar a primeira camada de classifiers
clf1 = Classifier1(params1)
```

```
# Instanciar a segunda camada, do meta-classificador
clf_meta = ClassifierMeta(paramsMeta)

# Criando o Stacking classifier
clf_stack = StackingClassifier(
    classifiers=[clf1, clf2, ... clfN],
    meta_classifier=clf_meta,
    use_probas=False,
    use_features_in_secondary=False)

# use os métodos fit e predict igual ao scikit-learn
clf_stack.fit(X_train, y_train)
pred = clf_stack.predict(X_test)
```

Para stackings de regressão:

```
from mlxtend.classifier import StackingClassifier

# Instanciar a primeira camada de classifiers
reg1 = Regressor(params1)
reg2 = Regressor(params2)
...
regN = Regressor(paramsN)

# Instanciar a segunda camada, do meta-classificador
reg_meta = RegressorMeta(paramsMeta)

# Criando o Stacking Regressor
reg_stack = StackingRegressor(
    regressor=[reg1, reg2, ... regN],
    meta_regressor=reg_meta,
    use_features_in_secondary=False)

# use os métodos fit e predict igual ao scikit-learn
reg_stack.fit(X_train, y_train)
pred = reg_stack.predict(X_test)
```

Conclusão

Que bom que você chegou até aqui. Essa jornada foi extensa e rica. Agora você ampliou o entendimento sobre alguns dos algoritmos mais conhecidos e usados por data scientists do mundo como o RandomForest, AdaBoost e XGBoost. Além disso, também

Não deixe de dar uma conferida nos materiais anexos, principalmente nos notebooks nas descrições de stackings vencedor em uma competição do kaggle. É isso mesmo, sua jornada está só começando, mas vale muito a pena!

Por último, lembre que tem sempre alguém da Tera pronto para te ajudar a qualquer dúvida ou dificuldade.

Para saber mais:

Simple guide for ensemble learning methods

What, why, how and Bagging — Boosting demystified, explained rather unconventionally, read on:)

towardsdatascience.com

Ensemble Learning to Improve Machine Learning Results

How ensemble methods work: bagging, boosting and stacking

blog.statsbot.co

A Comprehensive Guide to Ensemble Learning (with Python codes)

When you want to purchase a new car, will you walk up to the first car shop and purchase one based on the advice of the...

www.analyticsvidhya.com

vsmolyakov/experiments_with_python

Permalink Dismiss GitHub is home to over 50 million developers working together to host and review code, manage...

github.com

Binning, bagging, and stacking, are basic parts of a data scientist's toolkit and a part of a series of statistical...

medium.com

How to Develop Voting Ensembles With Python - Machine Learning Mastery

Voting is an ensemble machine learning algorithm. For regression, a voting ensemble involves making a prediction that...

machinelearningmastery.com

Basic Ensemble Learning (Random Forest, AdaBoost, Gradient Boosting)- Step by Step Explained

We all do that. Before making big decisions, we ask people's opinions, like friends, family, even our dogs/cats. So...

towardsdatascience.com

Ensemble Learning Explained! Part 2

This is a two-part article on Ensemble Learning. The first part is linked here. This article is going to focus on the...

medium.com

Otto Group Product Classification Challenge

Classify products into the correct category

www.kaggle.com


Thanks toTera