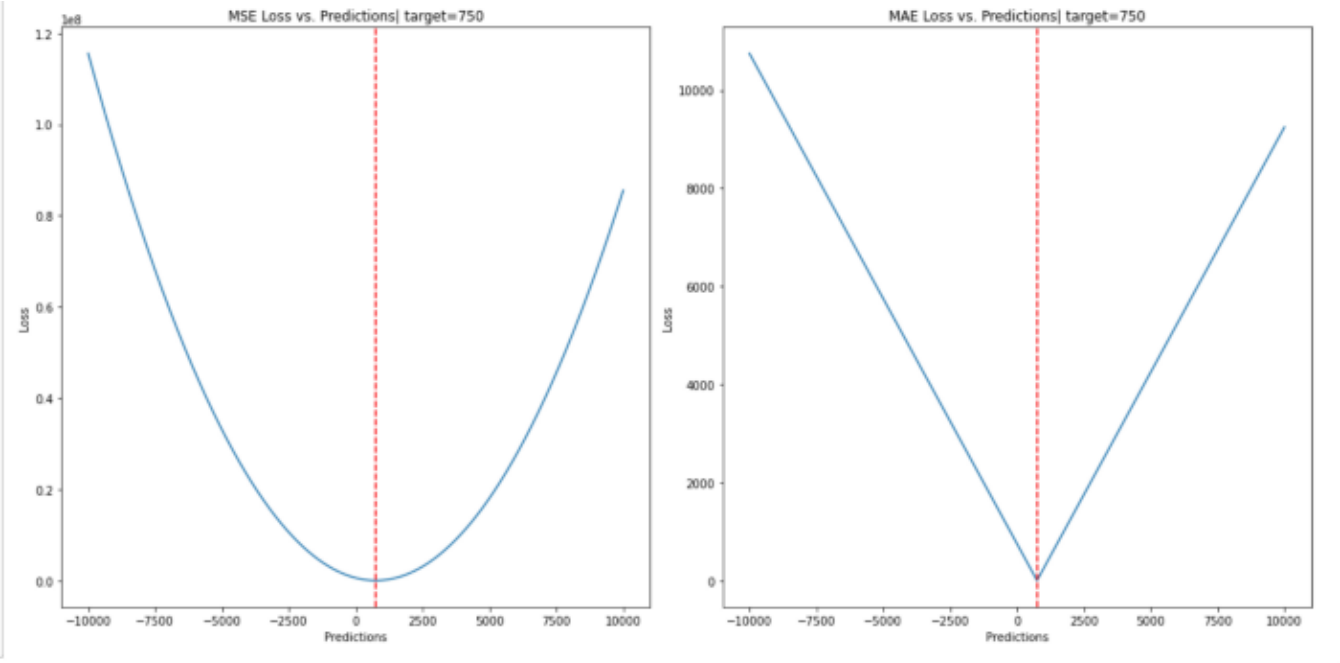
Funções Custom de Loss

Olá! Que bom ver vocês novamente por aqui :) Hoje abordaremos o tema de `Custom Loss Functions`

Mas o que é isso?

Retomando as aulas de Gradient Descent, vimos que existem alguns algoritmos que têm como principal objetivo **minimizar** uma função. Essa função, para ser minimizável, precisa que calculemos o *gradiente* de diferentes pontos (logo, ela precisa ser diferenciável) e, dois exemplos para os casos de regressão, eram o `MSE` e o `MAE`, ilustrados pelas imagens abaixo:



Fonte

Podemos ver que **diferentes funções de loss implicam em diferentes formas de aprendizado no caso do GradientDescent**

No caso, isso implica em diferentes métricas finais de algoritmos que usam isso por trás, vamos usar o caso do Gradient Boosting na aula de hoje para ilustrar isso. Mas, antes disso, vamos começar carregando os datasets.

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
```

```
dataset = fetch_california_housing()
dataset_features = dataset["data"]
target = dataset["target"]
feature_names = dataset["feature_names"]
target_name = dataset["target_names"]

dataset = pd.DataFrame(
    np.hstack((dataset_features, target.reshape(-1, 1))),
    columns=np.hstack((feature_names, target_name)),
)
dataset.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

Vamos treinar um lightgbm regressor por meio de validação cruzada e usando `RMSE` como nossa métrica final.

```
from lightgbm.sklearn import LGBMRegressor
from sklearn.model_selection import cross_val_score

model = LGBMRegressor(random_state=42, objective='mse')
neg_mse = cross_val_score(
    model,
    X=dataset[feature_names],
    y=dataset[target_name],
    cv=5,
    scoring="neg_mean_squared_error",
)
```

O scikit tem uma API própria que sempre tenta maximizar valores, inclusive no caso nos casos em que métricas com menor valore implicam em modelos melhores. Por isso, ele aplica o negativo dessas métricas. Logo, para obter o `RMSE` precisamos inverter para positivo:

```
mse = neg_mse*(-1)
rmse = np.sqrt(mse)
rmse.mean()
```

Aqui, obtemos um RMSE de 0.6150957727940838. Agora, vamos alterar a função objetivo (ou loss function) que o modelo é otimizado e ver quanto obtemos de resultado.

```
model = LGBMRegressor(random_state=42, objective='mae')
neg_mse = cross_val_score(
    model,
    X=dataset[feature_names],
    y=dataset[target_name],
    cv=5,
    scoring="neg_mean_squared_error",
)
mse = neg_mse*(-1)
rmse = np.sqrt(mse)
rmse.mean()
```

Chegamos ao resultado de 0.6284721889226077. Ou seja, **funções objetivo diferentes implicam em resultados diferentes** ! No caso do lightgbm, várias outras funções objetivas já foram implementadas, das quais podemos destacar:

```
- quantile
- poisson
- mape
```

Em mais outras, com seus possíveis detalhes, podem ser verificadas aqui

Cada função objetivo tenta “mapear” algum comportamento. Uma vez que você conhece mais ou menos o que você gostaria de ter na saída do seu modelo, o uso de uma determinada função objetivo pode ser mais adequado :)

Para exemplificar, se você quer modelar **contagens** (e.g. *quantos* itens vão existir em estoque), usar uma **Poisson** faz mais sentido que a **MSE**. Caso você queira que as suas predições se aproximem mais da *mediana* e não da *média* do target e, portanto, sendo mais **robustos** contra outliers, faz mais sentido você pensar em uma **Quantile** com *alpha=0.5*.

Nos dois cenários, contamos com funções **já implementadas** pela biblioteca. Mas e quando temos algum caso em que queremos modelar algo em que não temos a implementação nativa?

É para isso que você chegou até aqui!

O problema

Vamos retomar o exemplo que estamos usando.

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

É um dataset de preços de imóveis em que a variável target está na unidade de milhares ou milhões de dólares. Vamos dividir nossos dados em treino e teste para podermos analisar os erros de teste.

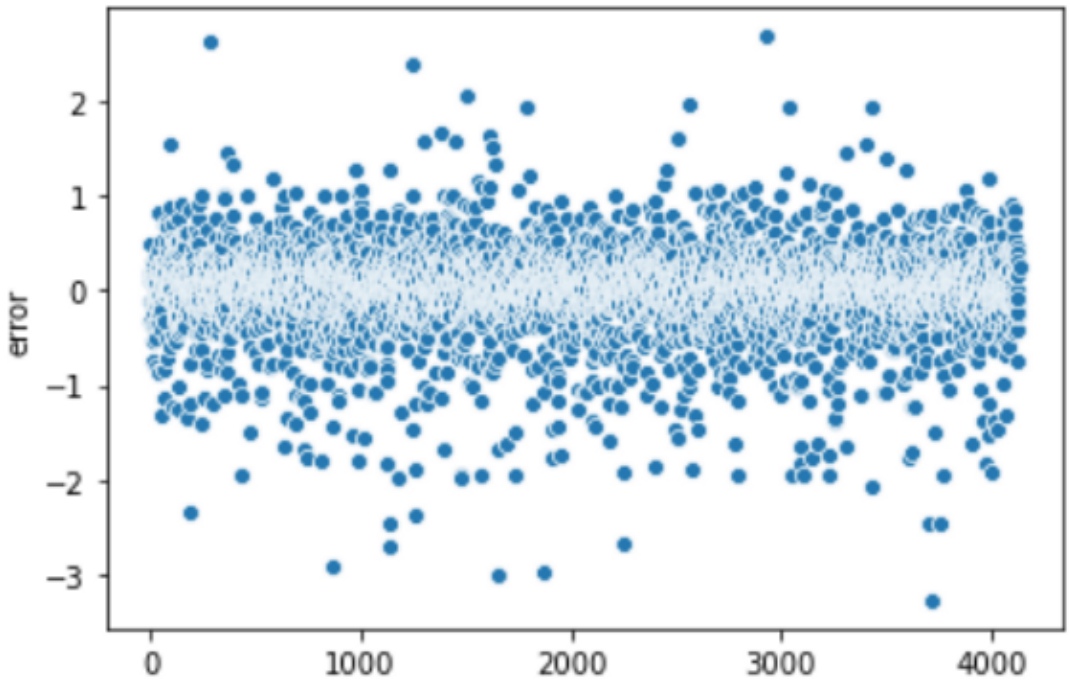
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_test, y_train, y_test = train_test_split(
    dataset[feature_names], dataset[target_name], test_size=0.2,
random_state=42
)

model = LGBMRegressor(random_state=42, objective='mse')
model.fit(X_train, y_train)
np.sqrt(mean_squared_error(y_test, model.predict(X_test)))
```

Aqui, obtemos um resultado de 0.46351720797890145.

Contudo, supomos que estamos observando um cenário em que queremos *sugerir* preços para as pessoas. O nosso modelo poderia, então, sugerir valores **acima** do preço da casa (prejudicando quem quer comprar) quanto **abaixo** do preço real (prejudicando quem quer vender). Vamos ver como isso acontece no nosso caso?

```
predict = model.predict(X_test)
metrics = pd.DataFrame(
    np.hstack([y_test.values.reshape(-1, 1),
model.predict(X_test).reshape(-1, 1)]),
    columns=["target", "prediction"])
```

```
metrics['error'] =  metrics['prediction'] - metrics['target']
_ = sns.scatterplot(metrics.index, metrics['error'])
```



Aqui, os erros positivos implicam que o valor previsto foi **maior** que o **real** (vamos chamar esse caso de *superestimar*), enquanto os erros negativos implicam que o valor previsto foi **menor** que o **real** (vamos chamar aqui de *subestimar*). Será que conseguimos reduzir o número de casos **muito** superestimados?

```
len(metrics[(metrics['error']>2.5)])
```

```
2
```

Número de casos que o erro foi muito superestimado

Vemos aqui que **a grande maioria dos erros estão perto do 0**. Contudo, vemos erros bem agressivos, perto da casa dos dois mil ou mais! Tanto para cima, quanto para baixo. Será que conseguimos montar uma loss que penaliza muito quando os erros forem muito *superestimados*?

No caso, então, queremos uma métrica de Loss que **não seja simétrica** uma vez que erros muito positivos devem ter **maior peso** que os negativos

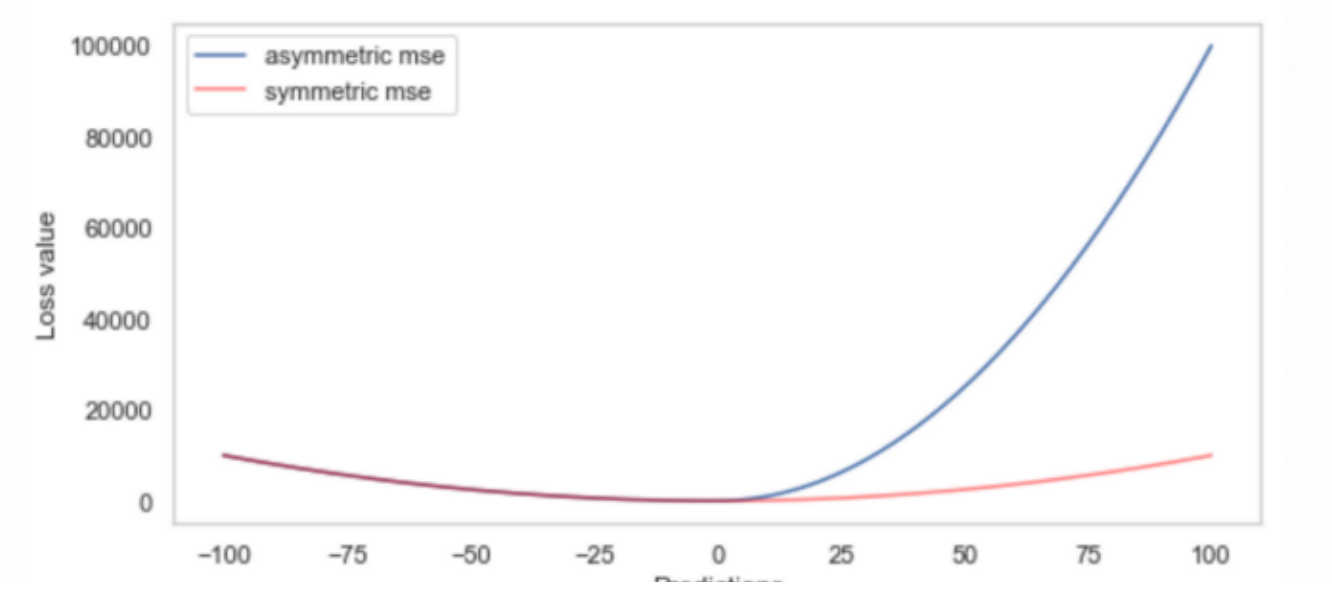
Desenhando a função de custo

Como mencionamos no começo e, entrando nos detalhes, por conta de algumas variações de implementações dos métodos de Gradient Descent, nós precisamos que a nossa função de custo obrigatoriamente tenha um *gradiente convexo* (derivada de primeira ordem) e uma *hessiana* (derivada de segunda ordem). Vou deixar o matematiquês de lado, para os curiosos, que podem ver essas definições aqui, aqui, aqui e aqui.

Não se assustem com os termos :) Vamos ver que é bem fácil de impementar no caso de gradient boosting!

Utilizando o exemplo da referência, temos o seguinte:

> Vamos examinar o que isso parece na prática e experimentar um pouco com dados simulados. Primeiro, vamos utilizar que erros superestimados são muito piores que os subestimados. Além disso, vamos usar que o MSE é uma boa métrica para o nosso erro nas duas direções. Então, vamos customizar uma função MSE que dá 10 vezes mais penalidade para erros positivos do que negativos. Isso é ilustrado pelo gráfico abaixo



Fonte

Indo para o código, teríamos algo como explícito abaixo

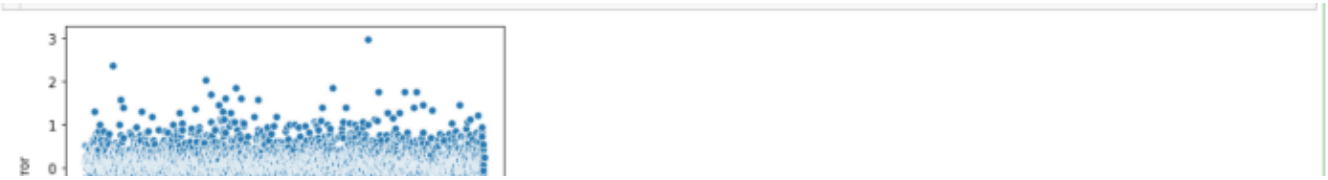
```
def custom_asymmetric_train(y_true, y_pred):

    # a ordem importa, aqui!
    residual = (y_true - y_pred).astype("float")

    #primeira derivada de uma MSE
    grad = np.where(residual>2.5, -2*residual*10, -2*residual)

    # segunda derivada de uma MSE
    hess = np.where(residual>2.5, 2*10, 2.0)

    return grad, hess
```

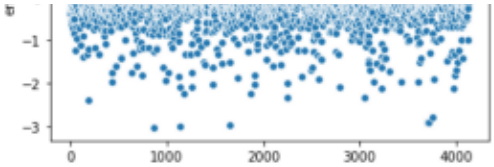
Uma vez definida, basta passar no lugar do `objective`

```
model = LGBMRegressor(random_state=42,
objective=custom_asymmetric_train)
model.fit(X_train, y_train)

np.sqrt(mean_squared_error(y_test, model.predict(X_test)))
```

Aqui, obtemos um valor de 0.46965705514098055, um pouco maior do que havíamos obtido inicialmente. Porém, se plotarmos os erros, temos:

```
predict = model.predict(X_test)
metrics = pd.DataFrame(
    np.hstack([y_test.values.reshape(-1, 1),
model.predict(X_test).reshape(-1, 1)]),
    columns=["target", "prediction"])
metrics['error'] = metrics['prediction'] - metrics['target']

_ = sns.scatterplot(metrics.index, metrics['error'])
```






```
len(metrics[(metrics['error']>2.5)])
```

```
1
```

Apesar de termos aumentado levemente o erro quadrático médio, conseguimos reduzir um pouco o número de casos que nosso modelo chuta para cima !

Conclusão

Parabéns por ter avançado mais um pouco na sua jornada para se tornar um cientista de dados :)

Para saber mais

- https://towardsdatascience.com/custom-loss-functions-for-gradient-boosting-f79c1b40466d