# Machine Learning - 20/05/2021

## Conceito de Machine Learning

Muitas pessoas imaginam que data science é, em maior parte, aprendizado de máquina e que os cientistas de dados constroem, praticam e ajustam modelos de aprendizado de máquina o dia inteiro. E, novamente, muitas dessas pessoas não sabem o que é aprendizado de máquina.

### Na verdade, a ciência de dados está transformando problemas de negócios em problemas de dados, coletando, entendendo, limpando e formatando dados, assim, o aprendizado de máquina está mais para uma reflexão sobre tais dados.

De qualquer modo, aprendizado de máquina é uma referência interessante e essencial que você deve saber a fim de praticar data science.

# Modelagem

## Antes de podemos falar sobre o aprendizado de máquina, precisamos falar sobre modelos.

O que é um modelo? É simplesmente a especificação de uma relação matemática (ou probabilistica) existente entre variáveis diferentes.

Por exemplo, se você esta tentado a levantar dinheiro para o seu site de rede social, talvez você precise de um modelo de negócios (possivelmente em uma planilha) que tenha entradas como "número de usuários", "rendimento de propaganda por usuário" e "número de funcionários" e exiba como saída seu lucro anual pelos próximos anos.

Um livro de receitas implica um modelo que relaciona entradas como "número de pessoas" e "apetite" para as quantidades dos ingredientes necessários. E, se você já assistiu pôquer na televisão, você sabe que eles estimam a "probabilidade de ganhar" de cada jogador em tempo real baseado em um modelo que leva em consideração as cartas que foram reveladas até então e a distribuição de cartas no baralho.

O modelo de negócios é, provavelmente, baseado em relações simples de matemática: lucro é o rendimento menos as despesas, o rendimento é soma das unidades vendidas vezes o preço médio e assim por diante. O modelo do livro de receitas é baseado em tentativas e erros, alguém foi na cozinha e experimentou combinações diferentes de ingredientes até encontrar uma que gostasse. O modelo do poquer é baseado na teoria da probabilidade, nas regras poquer e em um processo aleatório pelo qual as cartas são distribuídas.

# O Que É Aprendizado de Máquina?

Todo mundo possui sua própria definição, mas usaremos aprendizado de máquina para nos referir à **criação e ao uso de modelos que aprendem a partir dos dados.** Em outros contextos isso pode ser chamado de modelo preditivo ou mineração de dados, mas vamos manter o aprendizado de máquina. Normalmente, nosso objetivo será usar os dados existentes desenvolver modelos que possamos usar para prever possíveis saídas para os dados novos, como:

- Prever se uma mensagem de e-mail é spam ou não
- Prever se uma transação do cartão de crédito é fraudulenta
- Prever qual a probabilidade de um comprador clicar em uma propaganda
- Prever qual time de futebol ganhará o Brasileirão

Consideraremos os modelos **supervisionados** (nos quais existe um conjunto de dados rotulados com a resposta correta para aprendizagem) e modelos **não supervisionados** (nos quais não existem tais rótulos). Existem vários outros tipos como semisupervisionados (nos quais apenas alguns dados são rotulados) mas não serão abordados.

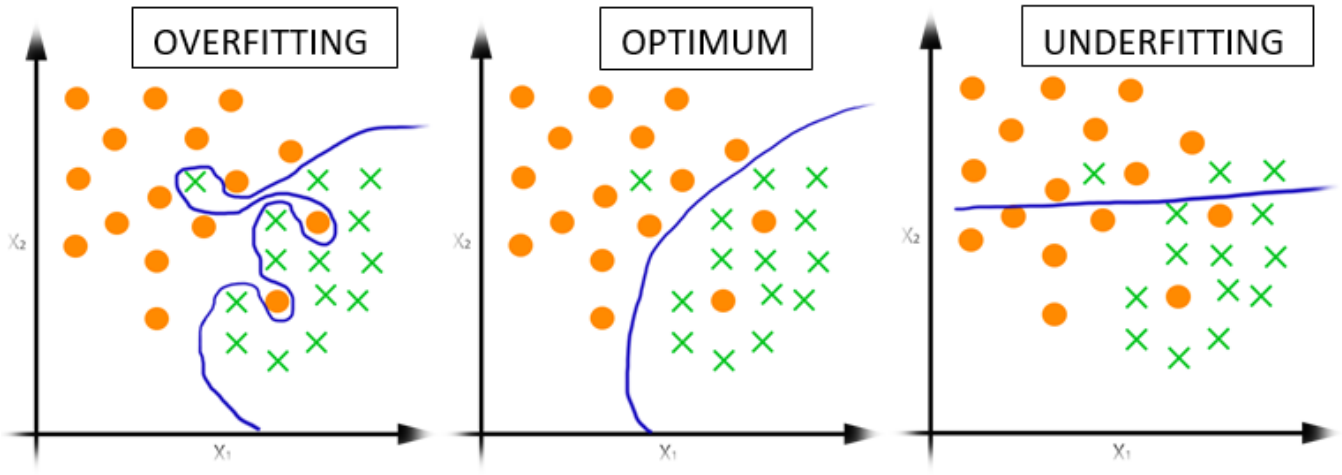
Agora, até mesmo na situação mais simples existe um universo inteiro de modelos que podem descrever a relação na qual estamos interessados. Na maioria dos casos, nós mesmos escolheremos uma família parametrizada de modelos e então usaremos os dados para aprender parâmetros que são, de certa forma, ótimos.

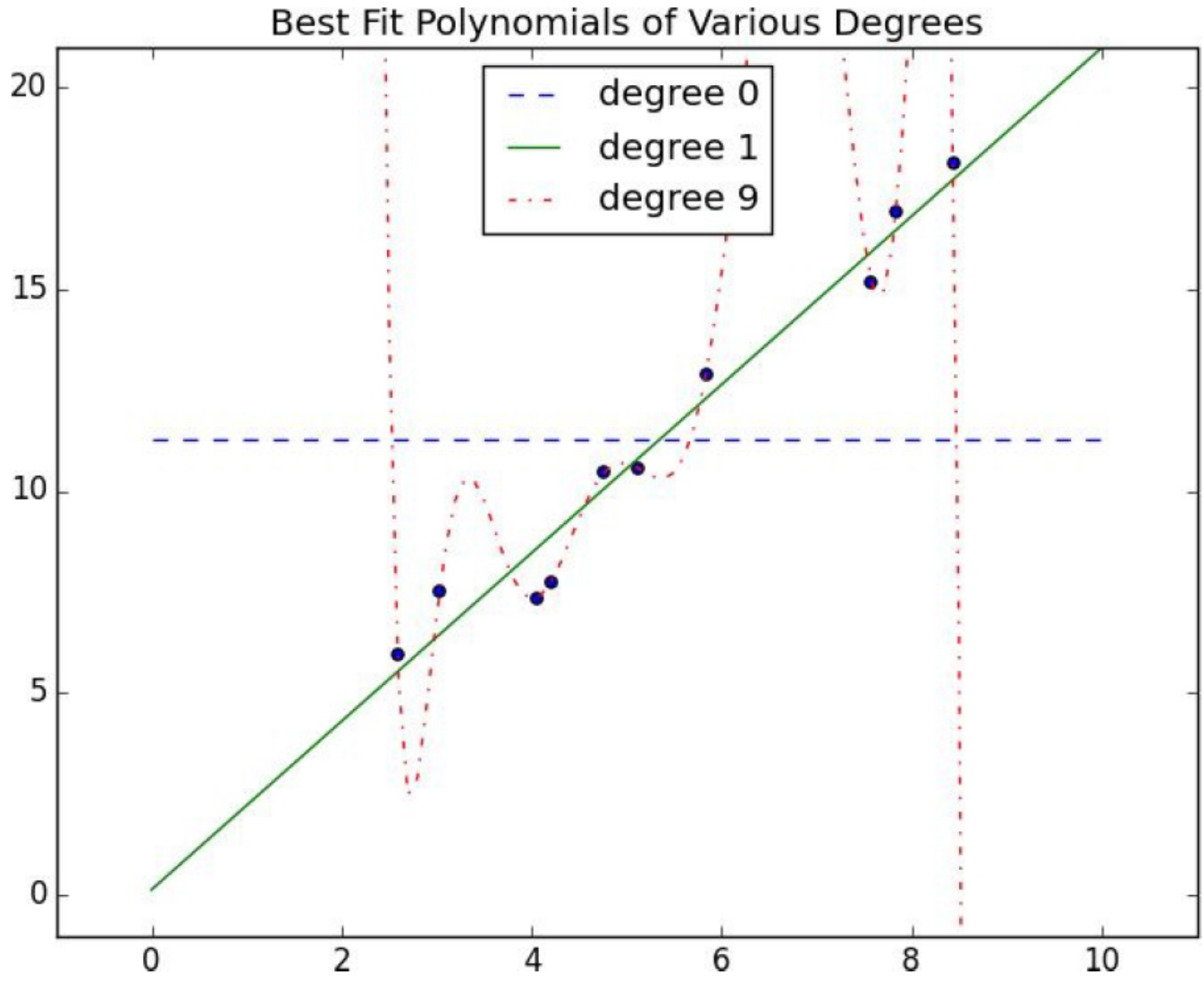
Por exemplo, podemos presumir que a altura de uma pessoa é (mais ou menos) uma função linear do seu peso e então usar os dados para descobrir qual funçao linear é essa. Ou, podemos presumir que uma árvore de decisão é uma boa maneira de identificar quais doenças nossos pacientes possuem e então usar os dados para descobrir a otima árvore de decisão.

# Sobreajuste (Overfitting) e Sub-Ajuste (Underfitting)

Um perigo comum em aprendizado de máquina é o **overfitting**, ou seja, produzir um modelo de bom desempenho com os dados de treinamento mas que não lide muito bem com novos dados. Isso pode implicar o aprender com base no ruído dos dados. Ou pode implicar em aprender a identiflcar entradas específicas em vez de qualquer fator que sejam de fato preditivos da saída desejada.

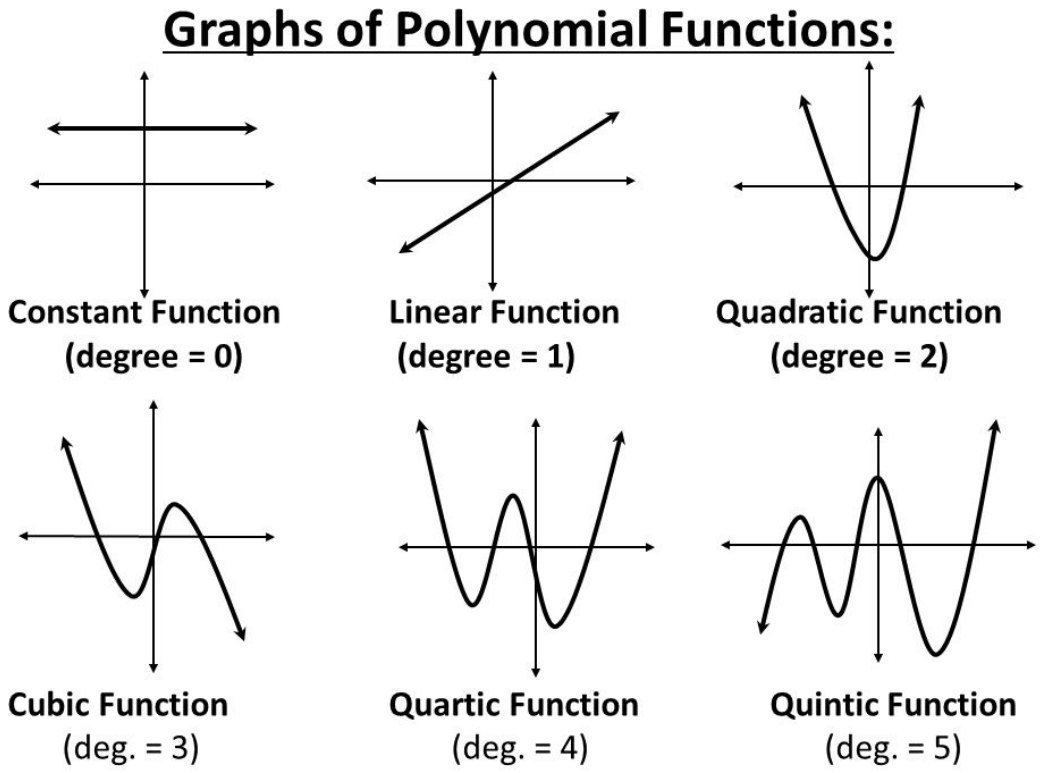
O outro lado é o **underfitting**, ou seja, produzir um modelo que não desempenha bem nem com os dados usados no treino, apesar de que, quando acontece isso, você decide que seu modelo não é bom o suficiente e continua a procurar por melhores.


OVERFITTING
OPTIMUM
UNDERFITTING
X2
X1


Best Fit Polynomials of Various Degrees
degree 0
degree 1
degree 9
20
15
10
5
0
0
2
4
6
8
10

A figura acima apresenta três polinômios em uma amostra de dados. A linha horizontal (azul) mostra o melhor ajuste do polinômio de grau 0 (isto é, constante). Ele sub-ajusta o dado em treinamento. O melhor ajuste de um polinómio do 9 grau (vermelho) passa por todos os pontos de dados em treinamento, mas sobreajusta gravemente e se fossemos adquirir um pouco mais de pontos de dados, provavelmente sairiam bem errados. E a linha verde, referente ao polinômio de grau 1 tem um bom equilibrio, é bem próximo a cada ponto, e (se esses dados são representativos) a linha estará próxima dos novos pontos de dados também.

Um polinômio é definido por sua ordem, que é a maior potência de X na equação. Uma linha reta é um polinômio de grau 1, enquanto uma parábola tem 2 graus.

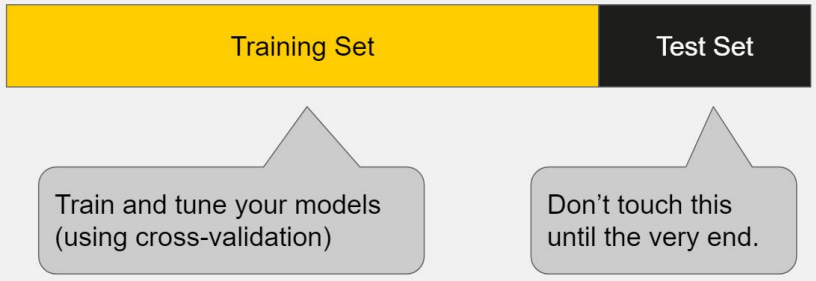



## Evitando o overfitting

Evidentemente, os modelos que são muito complexos tendem ao sobreajuste e não lidam bem com dados além daqueles com os quais foram treinados. **Então, como temos certeza que nossos modelos não são muito complexos?**

O método mais fundamental envolve o uso de dados diferentes para treinar e testar o modelo.

A maneira mais fácil de fazer isso é dividir seu conjunto de dados, a fim de que, por exemplo, dois terços dele sejam usados para treinar o modelo e depois medir o desempenho do modelo com a parte restante:




In [1]:
```
import random
def split_data(data, prob):
    """divide os dados en frações [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

In [3]:
```
random.random()
```

Out[3]:
```
0.29128942289671667
```

In [4]:
```
amostra = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
res = split_data(amostra, 0.33)
res
```

Out[4]:
```
([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]])
```

Com frequência, teremos uma **matriz X** de variáveis de entrada e um **vetor Y** de variáveis de saída. Nesse caso, precisamos nos certificar de colocar os valores correspondentes tanto nos dados em treinamento como nos dados de teste:

In [5]:
```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                                # par de valores correspondentes
    train, test = split_data(data, 1 - test_pct)   # divide o conjunto de pares de dodos
    x_train, y_train = zip(*train)                  # Truque mágico de unzip
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

In [10]:
```
data = list(zip(amostra, label))
data
```

Out[10]:
```
[([1, 2, 3], 10), ([4, 5, 6], 20), ([7, 8, 9], 30)]
```

In [6]:
```
label = [10, 20, 30]
```

In [15]:
```
res2 = train_test_split(amostra, label, 0.66)
res2
```

Out[15]:
```
(([7, 8, 9],), ([1, 2, 3], [4, 5, 6]), (30,), (10, 20))
```

A fim de fazer algo como:

model = SomekindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)

**Se o modelo foi sobreajustado para os dados em treinamento, então ele deve desempenhar mal sobre os dados de teste (completamente separados). De outra maneira, se ele desempenha bem sobre os dados de teste, então você pode ficar mais confiante que ele está ajustado em vez de sobreajustado.**

**Porém, existem duas formas de dar tudo errado.**

A primeira é **se existirem padrões comuns aos dados de teste e de treinamento que não seriam generalizados em um conjunto maior de dados**. Por exemplo, imagine que seu conjunto de dados consiste da atividade do usuário, uma linha por usuário por semana. Em tal caso, a maioria dos usuários aparecerá em ambos os dados de teste e de treinamento e alguns modelos talvez aprendessem a identificar os usuários em vez de descobrir relações envolvendo atributos. Não é de grande preocupação, apesar de ter acontecido comigo uma vez.

Um problema maior é se voce usar a divisão de **testes/treinamento não apenas para avaliar um modelo mas, também, para escolher entre os vários modelos**. Nesse caso, embora cada modelo individual possa não ser sobreajustado, o ato "escolha um modelo que desempenhe melhor nos dados de teste" é um quase treinamento que faz com que o conjunto de testes funcione como um segundo conjunto de treinamento. (É claro que o modelo que tiver melhor desempenho no teste terá um melhor desempenho no conjunto de teste.)

Em uma situação como essa, você deveria dividir os dados em três partes: um conjunto de treinamento para construir modelos, um conjunto de validação para escolher entre os modelos treinados e um conjunto de teste para avaliar o modelo final.

# Precisão

Considere o seguinte cenário: imagine que foi inventado um teste simples, não invasivo que pode ser feito com um recém-nascido que prediz, com uma precisão maior que 98%, se o recém-nascido desenvolverá leucemia. Os detalhes são: prever a leucemia se e somente se o nome do bebê for Luke (que se parece um pouco com o som de leukemia, leucemia em inglês)

Como veremos a seguir, esse teste possui mesmo mais de 98% de acurácia. Apesar disso, é um teste incrivelmente estúpido e uma boa ilustração do motivo pelo qual nós não usamos "acurácia" para medir a eficiência de um modelo.

**Matriz de confusão**

Imagine construir um modelo para fazer uma avaliação binária (sim ou não, é ou não é, etc.). Esse e-mail é spam? Deveríamos contratar este candidato? Este viajante é um terrorista em segredo?

Dado um conjunto de dados etiquetados e um modelo preditivo, cada ponto de dados se estabelece em quatro categorias

- Verdadeiro Positivo: "Esta mensagem é spam e previmos spam corretamente."
- Falso Positivo (Erro Tipo 1): "Esta mensagem não é spam, mas previmos que era."
- Falso Negativo (Erro Tipo 2): "Esta mensagem é spam, mas previmos que não era."
- Verdadeiro Negativo: "Esta mensagem não é spam e previmos que não era"

Representamos essa contagem em uma **matriz de confusao**:

| | Spam | not Spam |
|---|---|---|
| predict “Spam” | True Positive | False Positive |
| predict “Not Spam” | False Negative | True Negative |

Vamos ver como meu teste de leucemia se encaixa nessa estrutura. Por agora, aproximadamente 5 bebês de 1000 se chamam Luke (https://www.babycenter.com/baby-names-luke-2918.htm (https://www.babycenter.com/baby-names-luke-2918.htm)). A incidência de leucemia é de aproximadamente 1.4%, ou 14 de cada 1000 pessoas (https://seer.cancer.gov/statfacts/html/leuks.html (https://seer.cancer.gov/statfacts/html/leuks.html)).

Se acreditarmos que esses dois fatores são independentes e aplicar meu teste "Luke para leucemia" em um milhão de pessoas, esperaríamos ver uma matriz de confusão com esta:

| | leukemia | no leukemia | total |
|---|---|---|---|
| “Luke” | 70 | 4,930 | 5,000 |
| not “Luke” | 13,930 | 981,070 | 995,000 |
| total | 14,000 | 986,000 | 1,000,000 |

Podemos usar isso então para computar diversas estatísticas sobre o desempenho do modelo. Por exemplo, a acurácia é definida como a fração de premissas corretas:

In [57]:
```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print(accuracy(70, 4930, 13930, 981079)) #0.98114
```

```
0.9811401697384724
```

Parece um número bem interessante. Mas, evidentemente, não é um bom teste, o que significa que não deveríamos colocar muita crença na acurácia bruta.

## Precision e Recall

É comum considerar a combinação de precision (precisão) e recall (revocação).
**Precision** significa o quão acertivas nossas previsões positivas foram, ou seja, dos verdadeiros positivos quantos realmente são positivos:

In [51]:
```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print(precision(70, 4939, 13930, 981070)) #0.015
```

```
0.015743324033479472
```

**Recall** mede qual fração dos positivos nossos modelos conseguem identificar, ou seja, do universo de positivos quantos ele conseguiu identificar:

In [53]:
```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print (recall(70, 4930, 13930, 981070)) #.005
```

```
0.005639231922335642
```

## F1-Score

As vezes, precisão e recall são combinados ao **F1 Score**, definido assim:

In [56]:
```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)
    return 2 * p * r / (p + r)

print (f1_score(79, 4930, 13930, 981070))
```

```
0.00830791881375539
```

Essa é a **média harmonica** (http://en.wikipedia.org/wiki/Harmonic_mean (http://en.wikipedia.org/wiki/Harmonic_mean)) do precision e recall cujo valor se localiza necessariamente entre eles.

Normalmente, a escolha de um modelo envolve uma decisão de perda-e-ganho (trade-off) entre precision e recall. Um modelo que prevê "sim" quando está um pouco confiante provavelmente terá um recall alto, mas um precision baixo; um modelo que prevê "sim" apenas quando é extremamente confiante é provável que tenha um recall baixo e um precision alto.

Imagine que houvesse 10 fatores de risco para a leucemia, e quanto mais deles você tivesse maior probabilidade de desenvolver leucemia. Nesse caso, você pode imaginar uma série de testes: "prediz leucemia se pelo menos um fator de risco", "prediz leucemia se pelo menos dois fatores de risco" e assim por diante. À medida que você aumenta o limiar, aumenta a precisão do teste (já que as pessoas com mais fatores de risco têm maior probabilidade de desenvolver a doença) e diminui o recall do teste (uma vez que cada vez menos portadores de doenças atingirão o limiar). Em casos como este, escolher o limite correto é uma questão de encontrar o trade-off correto.

# O trade-off de viés-variância

Outra maneira de pensar sobre o problema do overfitting (sobreajuste) é como um trade-off entre viés e variância.

Ambas são medidas do que aconteceria se você fosse treinar seu modelo novamente muita vezes em diferentes conjuntos de dados de treinamento (da mesma população).

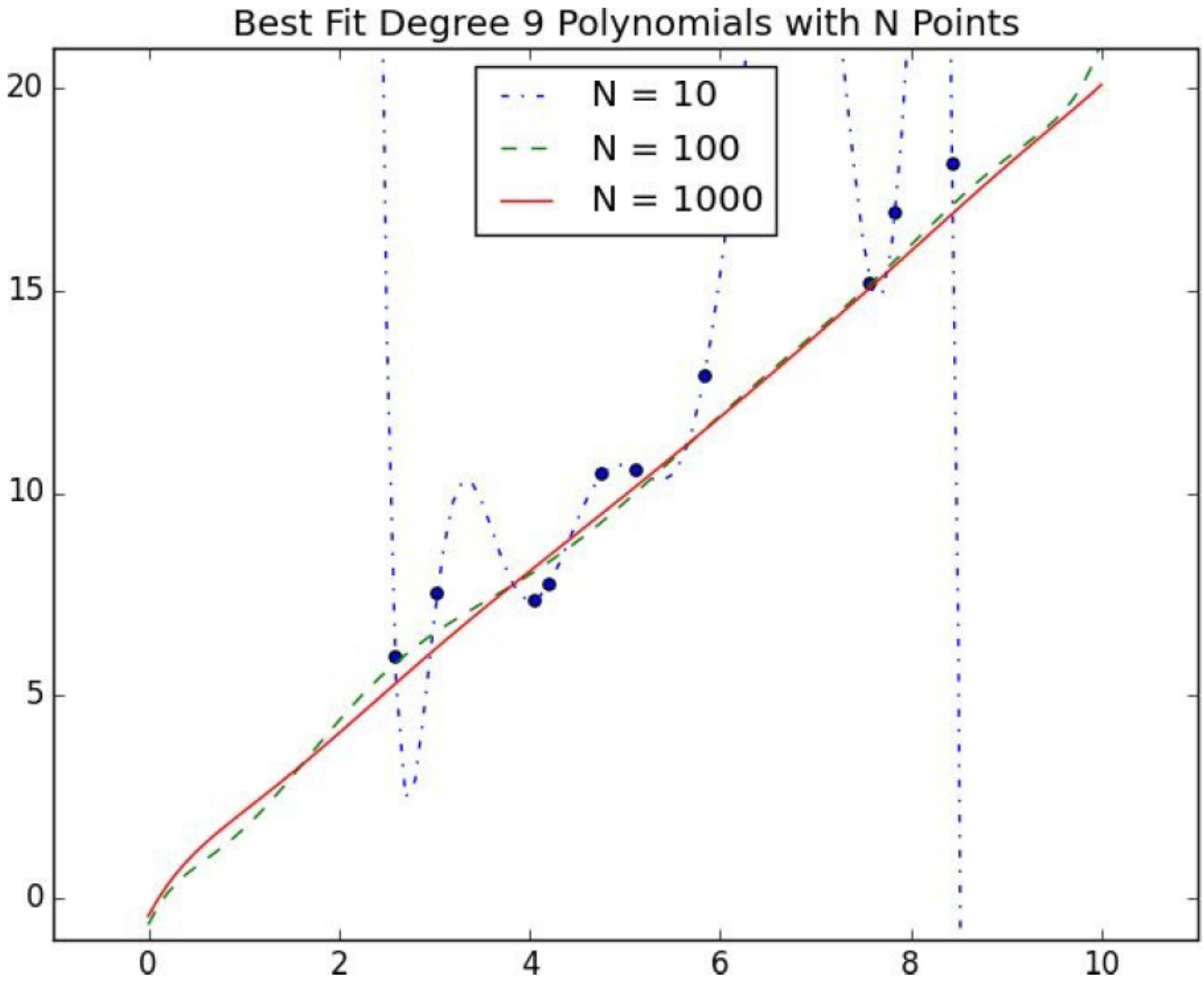
Por exemplo, o modelo polinomial de grau 0 exibido quando falamos de sobreajuste e sub-ajuste cometerá muitos erros para qualquer conjunto de dados em treinamento (tirados da mesma população), o que significa que ele possui um **viés (polarizaçao) alto**. O que significa dizer que ele possui uma baixa variância. **Viés alto e variância baixa geralmente pertencem ao underfitting**.

Por outro lado, o modelo polinomial de grau 9 se encaixa no conjunto de treinamento com perfeição. Possui baixo viés, mas variância muito alta (quaisquer dois conjuntos em treinamento dariam origem a modelos bem diferentes). Eles correspondem ao sobreajuste (overfitting).

Pensar sobre problemas de modelos dessa forma pode ajudar a descobrir o que fazer quando seu modelo não funciona muito bem.

Se o seu modelo possui um alto viés (o que significa que ele não possui um bom desempenho no seu conjunto em treinamento), algo mais a tentar é adicionar mais caracteristicas.

Se o seu modelo tem alta variância, então você pode de modo similar remover caracteristicas, mas outra solução seria obter mais dados (se puder).




Na figura acima, ajustamos um polinômio de grau 9 para diferentes amostras. O modelo ajustado com base nos 10 pontos de dados está em todo lugar, como vimos anteriormente. Se treinássemos com 100 pontos de dados, haveria muito menos sobreajuste. E o modelo treinado a partir dos 1000 pontos de dados é muito parecido com o modelo de grau 1.

Mantendo uma constante na complexidade do modelo, quanto mais dados há, mais dificil é para sobreajustar.

Por outro lado, mais dados não ajuda com o problema do viés. Se seu modelo não usa recursos suficientes para capturar regularidades nos dados, colocar mais dados não ajudará.

In [ ]:

## Exemplo

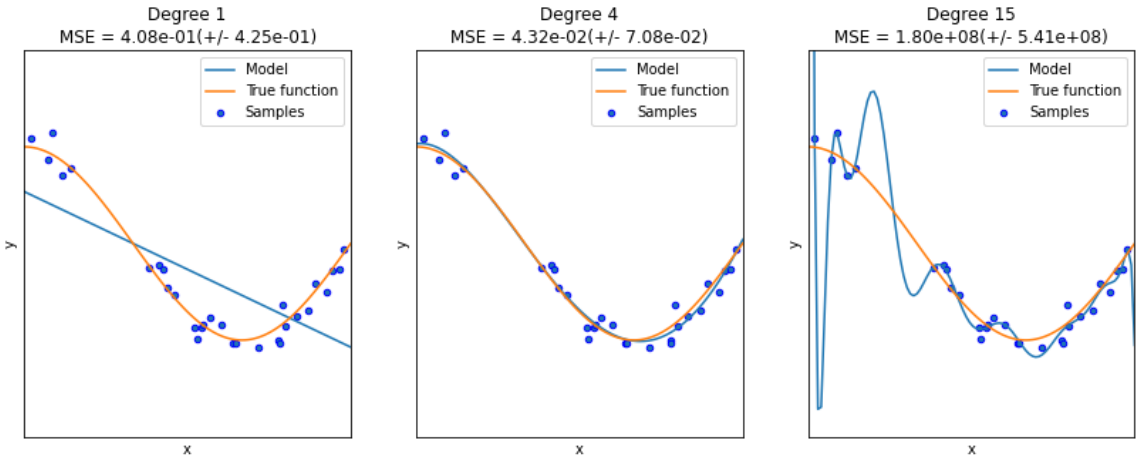
Este exemplo demonstra os problemas de underfitting e overfitting usando a regressão linear com recursos polinomiais para aproximar funções não-lineares.

O gráfico mostra a função que queremos aproximar, que é uma parte da função cosseno. Além disso, as amostras da função real e as aproximações de diferentes modelos são exibidas.

Os modelos possuem características polinomiais de diferentes graus. Podemos ver que uma função linear (polinômio com grau 1) não é suficiente para ajustar as amostras de treinamento. Isso é chamado de underfitting.
Um polinômio de grau 4 aproxima-se quase perfeitamente da função verdadeira.
No entanto, para graus mais altos, o modelo irá sobreajustar, ou seja, ele aprende o ruído dos dados de treinamento.

Nós avaliamos quantitativamente overfitting / underfitting usando validação cruzada.
Calculamos o erro quadrático médio (MSE) no conjunto de validação, quanto maior, menor a probabilidade de o modelo generalizar corretamente a partir dos dados de treinamento.

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score


def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                             include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_feature
s),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                             scoring="neg_mean_squared_error", cv=1
0)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label
="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))
plt.show()
```


Degree 1
MSE = 4.08e-01(+/- 4.25e-01)
Model
True function
Samples
y
x
Degree 4
MSE = 4.32e-02(+/- 7.08e-02)
Model
True function
Samples
y
x
Degree 15
MSE = 1.80e+08(+/- 5.41e+08)
Model
True function
Samples
y
x